Tutorial 4: ‘Platform Game’

Tutorial Overview:

This tutorial will explore how to construct a simple platform 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 additional features.

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 two tutorials and are happy with creating new classes with NetBeans, etc.

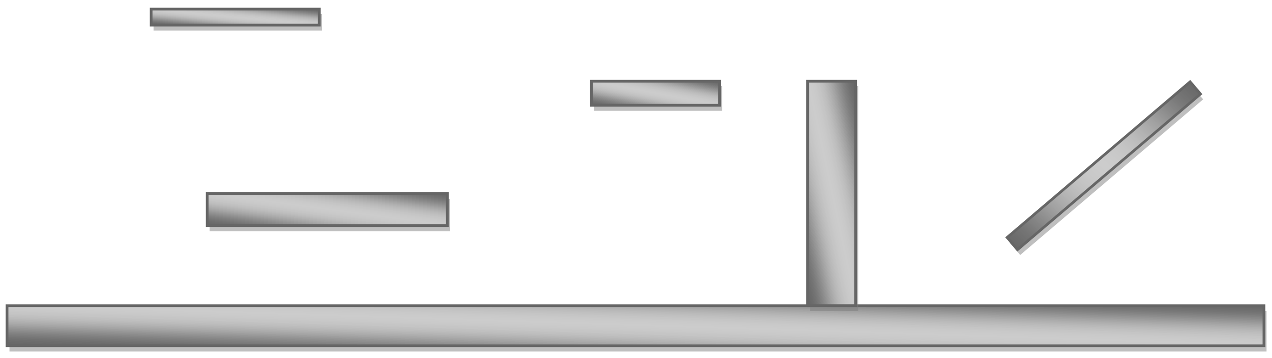
Phase 0: What are we going to do?

In this tutorial we are going to construct a simple platform game. In particular, we will look at how we can construct the platforms and, more interestingly, how we can develop an animated character that can run and jump between platforms. We will also explore how we can test for overlap between the sprite and the platforms.

Aside: As before, don’t forget you can access the completed code for this tutorial within the TutorialsDemosAndExtras.tutorials.platformer package. Well… the code contained within the platformer package slightly extends the code presented within this tutorial by adding in some help information and also enabling the player to add ball objects into the level. Run and code and have an explore!

Phase 1: Building platforms

A platform game is so called because it is composed of platforms, positioned in such a manner that they provide a means of progressing from the level start to the level exit. Hence, whenever we are creating a platform game one of the first things we need to ask ourselves is how we are going to create our platforms. Consider the following simple platform level:



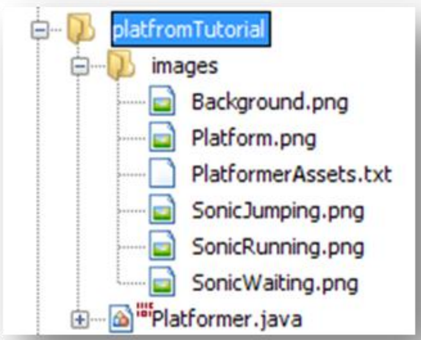
We can surmise that the level is comprised of a number of platforms and possibly a ground 'object'. Each platform might be of a different shape, in a different location and possibility with a different rotation. Just to push the boat out, platforms could also differ in other ways, e.g. some icy platforms may have low friction, others might crumble and break, etc.

Hence, whenever we wish to create our platform game we should ask what types of platform will be needed within the game and possibly consider creating a specific platform class to model the different types of platform. As this is a simple tutorial, we won't be doing that and instead will use basic platforms.

As each platform is located at a certain location and each platform game will typically consist of a number of different levels, it also makes sense that we should describe the level using a save structure that can be loaded and used to reconstruct the game level. For a platform level this is a relatively painless process, e.g. a platform level might be described in terms of a number of platforms (each with a specified type, location, rotation, etc.)., collectable items, decorative items, opponents, etc. We won't explore how we could load/build a level within this tutorial; instead have a look at the isometric tile map if you're interested in one approach towards loading/building levels.

Phase 2: Extending the GameEngine

Create a new package to hold your platform game (I'm going to assume it will be called **platformTutorial** within the **TutorialsDemosAndExtras.tutorials** package). Also create a directory called **images** within your platformTutorial package and copy across all the images associated with this project and also the PlatformerAssets.txt asset file). Next, create a new Java class within platformTutorial called Platformer.java which holds the code shown on the next page.



```
platfromTutorial
    images
        Background.png
        Platform.png
        PlatformerAssets.txt
        SonicJumping.png
        SonicRunning.png
        SonicWaiting.png
    Platformer.java
```

```java
package tutorials.platfromTutorial;

import game.engine.*;

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

public class Platformer extends GameEngine {

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

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

    @Override
    public boolean buildAssetManager() {
        assetManager.loadAssetsFromFile(
          getClass().getResource("images/PlatformerAssets.txt"));
        return true;
    }

    @Override
    protected boolean buildInitialGameLayers() {
        PlatformLayer platformLayer = new PlatformLayer( this );
        addGameLayer( platformLayer );

        return true;
    }

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

        super.gameRender( graphics2D );
    }

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

The code differs slightly from some of the earlier tutorials in that we overload the inherited gameRender method within the GameEngine to blank the screen to black, before then calling super.gameRender to draw all visible game layers.

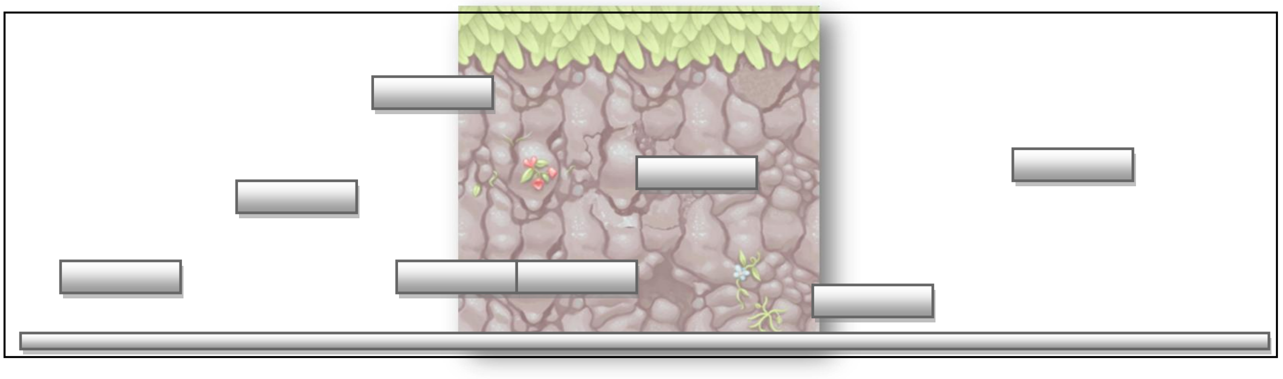
Aside: Open up the PlatformerAssets.txt file. This file contains the instructions for loading in all the assets used within this tutorial. Consider the following two lines:

```
ImageAssetSequence SonicJumping     Animated 1 500 2 Segments 3     SonicJumping.png
ImageAssetSequence SonicRunning     Animated -1 1000 0 Segments 8  SonicRunning.png
```

The first line loads in a Sonic jumping animation that will be played once (1) over half a second (500ms) and whenever the animation has completed we will display the 3rd frame (index 2, i.e. 0,1,2) of the animation (which shows Sonic in his final falling state). The animation can be found within an image file called SonicJumping which we're going to split into 3 images. The second line loads in a Sonic running animation that we will continuously loop and play (-1) over a period of 1 second (1000ms). As the animation is always running we don't really have a 'home frame', i.e. one that is displayed whenever the animation is not running, so it does not really matter what we specify (a default value of 0 was provided). We will load the running animation from an image SonicRunning which will be split into 8 images.

Phase 3: Creating the game layer(s)

In this particular tutorial we are going to create a total of two game layers: one layer to hold the background image and a second layer to hold the platforms. Consider the following:



The background cave image represents a background layer that is sized to be equal to the screen dimensions, e.g. if the screen resolution is 1024x768 then the size of the background layer is also 1024x768. The background layer will contain a single game object, whose width and height will be equal to that of the layer (i.e. the game object occupies the whole layer). The graphical realisation used by the single game object contained within the background will be an image ribbon asset, i.e. consisting of an asset that can be scrolled. So, whenever the player moves either left or right, we will also scroll the asset contained within the game object (contained within the background) by the same amount as the player has moved. This means that we're giving the impression of the background moving as the player also moves.

The second layer in the game that we will create will have the same height as the screen resolution but be much longer (e.g. if the screen resolution is 1024x768 this layer might be 10000x768). Within this layer we will place the platforms. Obviously, at any one time, only a subset of all the platforms displayed within the layer will be shown on the screen. In other words, we will have a viewport projected onto the layer and only the contents of that viewport will be displayed on the screen. Thankfully the hard work of doing this is already provided within the GameLayer class (if you fancy a bit of an adventure, open the class and have a look at the draw method).

As the player moves about the platform layer we will change the layer viewport (i.e. the centre point of the layer that will project onto the centre of the screen by default) so that it keeps the player ‘in-view’, i.e. the player can’t disappear outside of the viewport. In order to see how we can accomplish this, create a new Java class within platformTutorial called PlatformLayer.java and add the code shown below:

```
package tutorials.platfromTutorial;

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

public class PlatformLayer extends CollisionSpace {

    public static final double GRAVITY_STRENGTH = 800.0;

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

        width = 10000;
        height = gameEngine.screenHeight;

        setGravity(0, GRAVITY_STRENGTH );

        addGameObjectCollection( "Platforms" );
        addGameObjectCollection( "Balls" );

        createBackground();
        createPlatforms();
        createCharacter();
    }

    private void createBackground() {
        GameLayer backgroundLayer =
             new GameLayer( "BackgroundLayer", this.gameEngine );
        backgroundLayer.width = gameEngine.screenWidth;
        backgroundLayer.height = gameEngine.screenHeight;

        GameObject background = new GameObject( backgroundLayer );
        background.setName("Background");
        background.setPosition(
            backgroundLayer.width/2, backgroundLayer.height/2 );

        ImageAssetRibbon backgroundAsset
                = (ImageAssetRibbon)assetManager.
                        retrieveGraphicalAsset("Background");
        background.setRealisation(backgroundAsset);
        background.setGeometry( new Box( 0, 0, 1024, 768));

        backgroundLayer.addGameObject(background);
        gameEngine.addGameLayer(backgroundLayer);

        setDrawOrder(backgroundLayer.getDrawOrder()+1);
    }
```

The constructor defines the layer (which happens to be the one that will hold the platforms) to be of width 10000 with a height equal to the screen dimension. The createBackgroundLayer method creates a new game layer (we don’t have to do this within our GameEngine class). Adds to it a single game object that has an ImageAssetRibbon showing the background graphic). We give this game object a name as we will want to retrieve it later. We finally add the new game layer to the game engine and also setup the draw order of this game layer to be one above that of the background layer, i.e. the game engine will draw the platform layer after the background layer.

Continue by adding the code shown below:

```
    private void createPlatforms() {
        double groundOffset = 0;
        while( groundOffset < this.width ) {
            createPlatform( groundOffset,
                    this.height  assetManager.
                  retrieveGraphicalAssetArchetype("Platform").height );
            groundOffset += assetManager.
                  retrieveGraphicalAssetArchetype("Platform").width;
        }

        for( int idx = 0; idx < 8; idx++ )
            createPlatform( 300+150 * idx, this.height-50*(idx+1) );
        for( int idx = 0; idx < 8; idx++ )
            createPlatform( 2500-150 * idx, this.height-50*(idx+1) );
        for( int idx = 0; idx < 5; idx++ )
            createPlatform( 3500-400 * (idx%2), this.height-100*(idx+1) );
        for( int idx = 0; idx < 10; idx++ )
            createPlatform( 4500+150 * idx, this.height-50*(idx+1) );
        for( int idx = 0; idx < 12; idx++ )
            createPlatform( 6000, this.height-50*(idx+1) );
        for( int idx = 0; idx < 12; idx++ )
            createPlatform( 7200, this.height-50*(idx+1) );
    }

    private void createPlatform( double x, double y ) {
        Body platform = new Body( this );
        platform.setRealisationAndGeometry("Platform");
        platform.setPosition(x, y);
        platform.setMass( Double.MAX_VALUE );

        addGameObject( platform, "Platforms" );
    }

    private void createCharacter() {
        SonicSprite sonic = new SonicSprite( this );
        sonic.setPosition( 0,
            gameEngine.screenHeight - sonic.height - 200 );
        addGameObject( sonic );
    }
```

The code shown above populates the level with a good number of platforms, created using simple for loops. Of course, you'd never want to use this approach for an actual platform game – it is just about acceptable within this tutorial as a quick (and dirty) means of building up a simple platform level.

Each platform is created to have the same realisation ("Platform") and geometry (equal to the width and height of the platform image). We also set the mass of the platform to Double.MAX_VALUE (which is equivalent to Body.INFINITE_MASS, i.e. the platform won't move!)

The createCharacter method creates our sprite – a Sonic character in this particular case (although we have yet to define this class).

Once we have added the ‘creation’ methods to PlatformLayer we can next add the update methods (as shown below).

Within the update() method we call the all important super.update() method – as this particular layer extends CollisionSpace, whenever we call the super.update() method we will update all bodies based upon their forces, velocities, etc. and also check for collisions which can then be resolved.

```
    @Override
    public void update() {
        super.update();

        updateViewPort();
        updateGameObjects();
    }

    private void updateViewPort() {
        GameObject sonic = getGameObject( "Sonic" );

        centerViewportOnGameObject(sonic, 0.0, 0.0,
            gameEngine.screenWidth/2.0, gameEngine.screenHeight/2.0);

        GameObject background =
                gameEngine.getGameObjectFromLayer(
                  "Background", "BackgroundLayer");
        ((ImageAssetRibbon)background.getRealisation(0)).setViewPort(
                (int)viewPortLayerX, 0 );
        background.getRealisation(0).update();
    }

    private void updateGameObjects() {
        GameObject sonic = getGameObject( "Sonic" );
        sonic.update();
        GameObjectUtilities.reboundIfGameLayerExited(sonic);
    }
}
```

Once we have updated all objects, potentially including moving the Sonic sprite, we next call the updateViewPort method. It is the role of this method to ensure that the viewport positions itself to keep the SonicSprite on screen. There is a useful method defined within the GameLayer class, called **centerViewportOnGameObject**) which provides a means of focussing the layer viewport on a particular object. Do have a look at the code for this method.

Once the platform layer viewport has been updated, we turn our attention to the background layer. Recall that this layer contains a single game object that, in turn, contains a ribbon image asset. To update the image ribbon we firstly retrieve the game object from the background layer and next extract the image asset ribbon. Once done, we can change the viewport of the ribbon (exactly the same concept as for the layer viewport, i.e. the ribbon defines a viewport which displays the current contents of the ribbon).

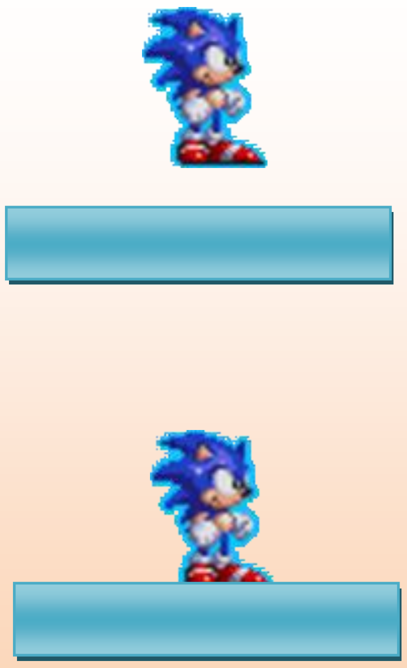
The **updateGameObjects** method updates our Sonic sprite based on user input. We also make use of the GameObjectUtilities.reboundIfGameLayerExited method to make sure the Sonic sprite cannot leave the confines of the platform layer.

Phase 4: If we didn't want to use the physics classes…

The advantage of using the CollisionSpace and Body classes within the platform game is that we don't have to worry about collision detection and collision resolution. However, as a 'what-if', let's consider how we might go about creating a platform game without using the physics classes.

An overview of the basic approach is as follows:

```
for( each moving game object ) {

  update the position of the game object, e.g.
  taking into account gravity user movement, etc.

  for( all other collideable objects, e.g.
          platforms, other moving objects, etc.) {

          check for overlap between the two objects
          if( overlap ) {

                  update the game objects as needed,
                  e.g. hurt the player, collect addition
                  health, etc.)

                  correct for overlap if needed, i,e,.
                  separate the objects so they are not
                  intersecting
          }
  }
}
```



How we can detect overlap and also separate the objects is an interesting question that we will explore later in this course whenever we look at collision detection and resolution.

Phase 5: Adding Sonic

We are now at the stage where we can add in our Sonic sprite. To do this, create a new class called **SonicSprite** within our platformTutorial package and add in the code shown on the next page.

```
package tutorials.platformer;

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

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

public class SonicSprite extends Body {

    public enum SonicState { RUNNING, WAITING, JUMPING }
    private SonicState sonicState;

    public enum SonicFacing { LEFT, RIGHT }
    private SonicFacing sonicFacing;

    private double SPRITE_BASE_RUN_VELOCITY = 10.0;
    private double SPRITE_BASE_FLY_VELOCITY = 20.0;
    private double SPRITE_BASE_JUMP_VELOCITY = 250.0;

    private double SPRITE_MAX_X_VELOCITY = 400.0;
    private static final double X_VELOCITY_DECELERATION = 20.0;

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

        setName( "Sonic" );
        setMass( 500.0 );
        restitution = 0.0;

        sonicState = SonicState.WAITING;
        sonicFacing = SonicFacing.RIGHT;

        GraphicalAsset sonicWaiting
                = assetManager.retrieveGraphicalAsset("SonicWaiting");
        setRealisationAndGeometry( sonicWaiting );
        setGeometry(
            new Box( 0, 0, sonicWaiting.width-20, sonicWaiting.height ) );
    }
```

The code shown above sets up the text variables that will be used to define our Sonic sprite. In particular, we define an enumerated type permitting Sonic to be in a running, waiting or jumping state. What about a falling state? This would be a perfectly sensible fourth state, however, in this simple tutorial I'm going to treat jumping and falling as the same thing! (i.e. Sonic is not standing/running on a platform). We also permit Sonic to be facing either to the left or the right. We also define a number of limits in terms of the movement velocities.

The constructor for the class gives our Sonic sprite a name (as we will want to retrieve it later). We also provide Sonic with an arbitrary mass – this would make a lot more sense if there are other moveable objects that Sonic can interact with, but it is still important as, by default, the Body class assumes that all objects have an infinite mass, i.e. they do not move!

Finally, in the constructor we put Sonic into a waiting state, give him an appropriate graphical image and setup Sonic's geometry to be a little bit less than the size of the waiting image. Defining the geometry is very important as we will rely upon the defined geometry to check for platform collision, etc. We make the geometry slightly less than the width of the image as the image has some white space on either side (have a look for yourself).

Let's continue updating Sonic by adding the code shown below to SonicSprite. The code defines the start of Sonic's update method. We remember the state of Sonic (running, jumping, etc.) upon entry as we will then check at the end of the method to see if the state has changed.

The first if-else checks to see if the up or down arrows keys have been pressed, and if so we give Sonic a corresponding y-velocity (i.e. in this tutorial Sonic can fly up or down). The second set of if-else conditions, checks to see if the user is moving to the left or the right. Notice that if Sonic is currently jumping (which also includes falling) we still permit the user to move Sonic either left or right, but at a reduced speed to when Sonic is standing on a platform. This is reflective of the rather physically-unsound way that many platform games permit characters to be controlled in the middle of a jump.

```
    @Override
    public void update() {

        SonicState sonicStateOnEntry = sonicState;

        if( inputEvent.keyPressed[ KeyEvent.VK_UP ]
                && !inputEvent.keyPressed[ KeyEvent.VK_DOWN ] ) {
            velocityy -= SPRITE_BASE_FLY_VELOCITY;
        } else if( inputEvent.keyPressed[ KeyEvent.VK_DOWN ]
                && !inputEvent.keyPressed[ KeyEvent.VK_UP ] ) {
            velocityy += SPRITE_BASE_FLY_VELOCITY;
        }

        if( inputEvent.keyPressed[ KeyEvent.VK_RIGHT ]
                && !inputEvent.keyPressed[ KeyEvent.VK_LEFT ] ) {

            sonicFacing = SonicFacing.RIGHT;

            if( sonicState != SonicState.JUMPING ) {
                sonicState = SonicState.RUNNING;
                velocityx += SPRITE_BASE_RUN_VELOCITY;
            } else
                velocityx += 0.25 * SPRITE_BASE_RUN_VELOCITY;
        } else if( inputEvent.keyPressed[ KeyEvent.VK_LEFT ]
                && !inputEvent.keyPressed[ KeyEvent.VK_RIGHT ] ) {

            sonicFacing = SonicFacing.LEFT;

            if( sonicState != SonicState.JUMPING ) {
                sonicState = SonicState.RUNNING;
                velocityx -= SPRITE_BASE_RUN_VELOCITY;
            } else
                velocityx -= 0.25 * SPRITE_BASE_RUN_VELOCITY;
        } else {

            if( sonicState != SonicState.JUMPING ) {
                sonicState = SonicState.WAITING;

                GameObjectUtilities.dampenVelocities(this,
                        X_VELOCITY_DECELERATION, X_VELOCITY_DECELERATION,
                        0.0, 0.0, 0.0, 0.0);
            } else {
                GameObjectUtilities.dampenVelocities(this,
                        X_VELOCITY_DECELERATION*0.2,
                        X_VELOCITY_DECELERATION*0.2,
                        0.0, 0.0, 0.0, 0.0);
            }
        }
```

The final major else block on the previous page is concerned with dampening Sonic's velocities, i.e. if the user is not pressing left or right then we return to a non-moving state (only reducing x-velocity, we might still be falling). We dampen the movement a lot if Sonic is standing on a platform, and less so if he is in the air.

To continue with Sonic's update, add in the following code:

```
        boolean touchingPlatform = false;
        GameObjectCollection platforms =
                gameLayer.getGameObjectCollection("Platforms" );
        double sonicBasey = this.y+this.height/2;
        for( int idx = 0; !touchingPlatform
                  && idx < platforms.size; idx++ ) {
            double overlap = 10.0;
            Body platform = (Body)platforms.gameObjects[idx];
            if( sonicBasey - (platform.y-platform.height/2) < 10.0
                && sonicBasey - (platform.y-platform.height/2) > -10.0
                && this.x + this.width/2 - overlap
                        > platform.x - platform.width/2
                && this.x - this.width/2 + overlap
                        < platform.x + platform.width/2 )
                touchingPlatform = true;
        }

        if( sonicState == SonicState.JUMPING )
            if( touchingPlatform )
                if( this.velocityy < SPRITE_BASE_JUMP_VELOCITY )
                    sonicState = SonicState.WAITING;

        if( inputEvent.keyPressed[ KeyEvent.VK_SPACE] ) {
            if( touchingPlatform ) {
                sonicState = SonicState.JUMPING;
                velocityy -= SPRITE_BASE_JUMP_VELOCITY;
            }
        }
```

The code shown above contains the main standing/falling test. Now, as we've used the built in physics engine we don't have to worry about detecting and resolving collisions. This is handy from one perspective (reducing the amount of code we need to create) and more difficult from another (how do we work out if Sonic is standing on a platform – Sonic and the platform won't overlap thanks to the correction introduced by the physics engine).

To solve this problem we test Sonic against each platform and if the bottom of Sonic is on the top of the platform we know we're standing on it! The exact test is a little bit more forgiving (we permit Sonic to be 10 pixels above, or below, the surface). Why do we do this? The main reason is that whilst the physics engine will separate Sonic from the platform it is not 100% exact, i.e. Sonic might be 1.0 pixel above or below the surface. The value of 10 above or below is a tad excessive –try varying this value and see if you get different outcomes.

The bottom if statement in the code fragment shown above checks to see if Sonic should enter into a waiting state or if Sonic should jump.

Let’s finish our Sonic code (and the entire tutorial) by completing the update method and add in a draw method, as follows:

```
        if( velocityy > ((CollisionSpace)gameLayer).getGravityY() /  10.0 )
            sonicState = SonicState.JUMPING;

        if( this.sonicState != sonicStateOnEntry ) {
            switch( sonicState ) {
                case RUNNING: setRealisation( "SonicRunning" ); break;
                case WAITING: setRealisation( "SonicWaiting" ); break;
                case JUMPING: setRealisation( "SonicJumping" ); break;
            }
        }

        GameObjectUtilities.clampVelocities(this,
                SPRITE_MAX_X_VELOCITY, SPRITE_MAX_X_VELOCITY,
                Double.MAX_VALUE, Double.MAX_VALUE,  0.0, 0.0 );

        this.rotation = 0.0;
        this.angularVelocity = 0.0;
    }

    @Override
    public void draw( Graphics2D graphics2D, int drawX, int drawY ) {
        BufferedImage currentRealisation =
            getRealisation(0).getImageRealisation();
        int width = currentRealisation.getWidth();
        int height = currentRealisation.getHeight();

        if( this.sonicFacing == SonicFacing.LEFT )
            graphics2D.drawImage( currentRealisation,
                  drawX+width/2, drawY-height/2,
                  drawX-width/2, drawY+height/2,
                  0, 0, width, height, null );
        else
            graphics2D.drawImage( currentRealisation,
                  drawX-width/2, drawY-height/2,
                  drawX+width/2, drawY+height/2,
                  0, 0, width, height, null );
    }
}
```

The first if condition may force Sonic into a jumping (falling) state if the y velocity exceeds a set value. Note, this is not an ideal situation as, for example, on a lift moving up or down Sonic would automatically enter into a jumping state.

We conclude the update method by checking if we have changed state, in which case we change the current graphical realisation (we don’t change the geometry – although maybe this would be an interesting aspect to change as well). Next, we clamp the velocities to the defined maximum values and, finally, set the rotation and angularVelocity to zero (the physics engine will likely try to rotate Sonic, e.g. get him to fall over following a collision, however, in this platform game we want Sonic to always stand tall!).

Now that you have everything completed, compile and run your program! Please do consider changing/extending the code in some manner.