Games Programming

Philip Hanna 110CSC207

Tutorial 3: 'Racing Car'

Tutorial Overview:

This tutorial will explore how to make a basic racing car game based on a moving sprite car. This is the third tutorial and it pushes the boat out somewhat by assuming you're familiar with the material explored within the first two tutorials. 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 two tutorials.

Important: If you downloaded the CSC207 Code Repository before 08/10/07 then please note that the code in the TutorialsDemosAndExtras.tutorials.topDownRacer.RacingCar.java differs slightly from that presented within this tutorial.

Phase 0: What are we going to do?

In this tutorial we are going to develop a simple racing car game – the racing car, a sprite, can be moved around the map under the control of the player. We will setup the map such that the car best responds when it remains on the road, slows down considerably when it leaves the road, and can hit various obstacles which will stop the car from moving (e.g. walls, etc.).

We will start this tutorial by exploring the different ways in which we might go about designing this game – there is no one best solution to this problem (each solution has its advantages and disadvantages).

Phase 1: What type of driving experience do we wish?

Pause for a second and have a bit of a think about the type of driving model that you would want within a 2D top-down racing game. To give this some context, have a look at the screen shot shown below:

So what type of behaviour would we like from our car sprite? This is a key question to ask of any game in terms of developing the design. For this game we can propose the following (Note, this is not a necessary list, nor a comprehensive list – it simply represents what we will do within this design).

- The car should be able to accelerate forward until a maximum speed is reached
- The car should be able to brake quickly to a stopping position
- The car should be able to accelerate backwards until a maximum speed is reached
- The car should be able to turn left and right at a maximum turning rate

So far, so good – let's include some notion of friction and momentum

- The car should be subject to a frictional deceleration force
- The car should have forward momentum in its direction of travel

Anything else? Well, let's add in a couple of requirements based on the screenshot.

- The car should have good handling whilst driving on the road surface
- The car should be subject to a stronger frictional force whilst travelling off road
- The car should not be able to travel across the water or through buildings

Otherwise stated, we want to have a car that can accelerate, slide around corners and suffer poor handling/acceleration on non-road surfaces.

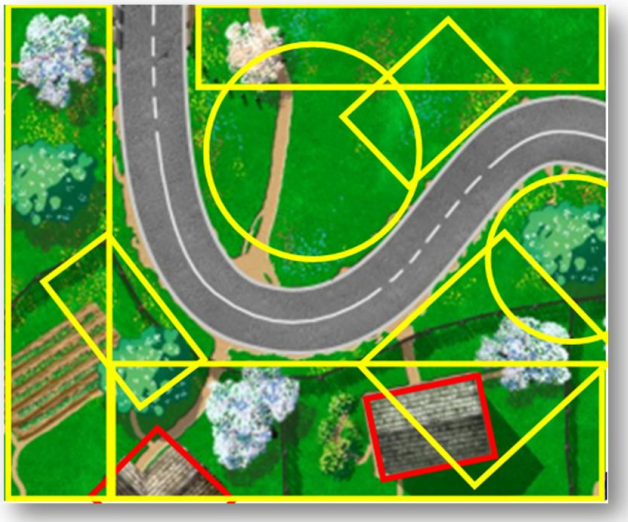
Phase 2: How do we develop this form of driving behaviour?

To a good extent we should be able to modify the various parameters of the GameObject class to get the desired form of car behaviour. However, this still leaves us with the problem of how do we tie the performance characteristics of the car to its position on the map? We could do this in a number of different ways. I've shown a couple of them below:

Approach 1: Adding invisible bounding regions

Add in a number of invisible shapes that can be tested to determine what type of surface the car is currently on, or when the car has hit an impassable region.

With specific regard to Java, the bounding regions can either be composed of simple geometric shapes and/or more complex curved shapes. The screen shots below provide two examples (red shapes are impassable; yellow shapes slow the car down).



○ **Simple bounding shapes**

○ **More complex bounding shapes**

The advantage of using simple shapes is that collision tests can be quickly executed – although the downside is that mapping simple geometric shapes onto a windy road may not be that accurate (and require that lots of simple shapes be used). The advantage of using more complex shapes is that a much improved match can be made against a windy road; however, this is at the expense of more complex overlap tests. Both approaches could be employed in terms of using simple geometric overlap tests to narrow down the number of more complex tests that need to be executed.

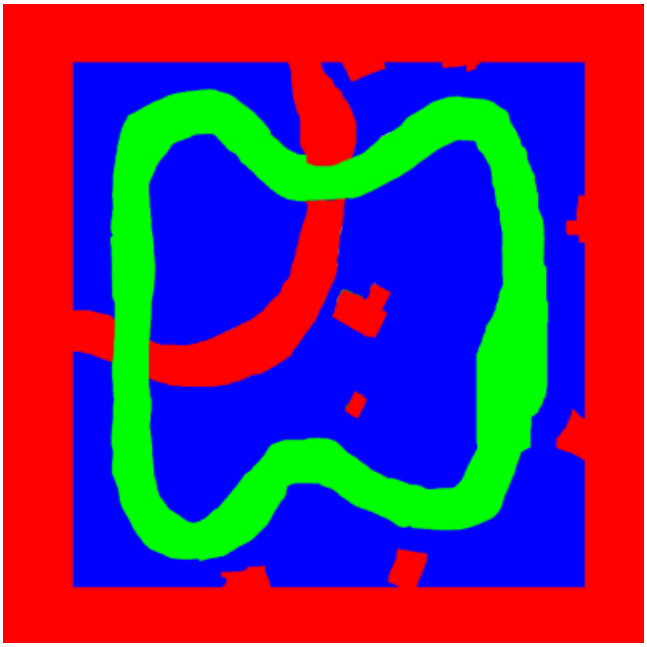
Approach 2: Developing a location map

This approach is based on using a second copy of the map that is specially drawn in such a manner that the position of the car can be tested against the location map. In this particular case, we would need to create a location map that indicates the areas on the map corresponding to road surface, off-road surfaces, and impassable regions.

We will use this approach as opposed to the use of invisible bounding regions. Why? Well, why not? We'll get to use bounding regions in other parts of the course so this should provide a useful alternative view.

Phase 3: Developing a location map

So what will this location map look like? In the best tradition of cookery programs, I've taken the liberty of previously knocking up a location map – compare the 2 images shown below:



The green areas on the location map correspond to road surfaces, the blue areas to off-road areas and the red areas to impassable regions. Admittedly, when creating this level I didn't take that much care in carefully creating the location map – things like trees, etc. are not marked as impassable, etc. – however, in as far as the map illustrates a principle it will hopefully suffice. You can also see that a generous red border surrounds the map (i.e. we don't want to permit the car to drive off the map!).

In terms of how the location map was created – I loaded the map into GIMP and then used the 'Select hand-drawn regions' tool to select regions which were then filled with the specified colour. I've included a couple of screenshots to the side that show interim development.

The use of red, green and blue is deliberate – in this game we only need to identify three different regions – on-road, off-road and impassable – i.e. three colours will suffice. As colours are stored as a mixture of red, green and blue components (i.e. the RGB model) by using the primary colours we can easily test to see what colour of pixel we are over (we'll explore this later).

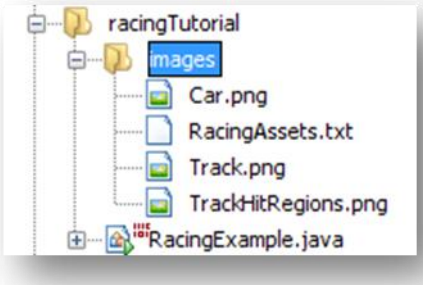
Once the game is up and running we will want to check the colour of the pixel(s) under the car using the location map. We can then use this knowledge to determine how the car will behaviour.

Phase 4: Developing the game

Let us now develop the classes we will need to put together to develop this game. Create a new package called **racingTutorial** and within this package create a new class **RacingExample** (this will be our game engine class). Include the following code within RacingExample (Note: the bold lines are only included here to make it easier to see where methods are defined)

```
package tutorials.racingTutorial;

import game.engine.*;

public class RacingExample extends GameEngine {

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

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

    @Override
    public boolean buildAssetManager() {
        assetManager.loadAssetsFromFile(

        this.getClass().getResource("images/RacingAssets.txt"));

        return true;
    }

    @Override
    protected boolean buildInitialGameLayers() {
        RacingExampleLayer racingLayer
          = new RacingExampleLayer( this );
        addGameLayer( racingLayer );

        return true;
    }

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

Hopefully the RacingExample.java code is familiar from the two previous tutorials – we load assets and create a new layer – now, admittedly we haven't yet added any assets or game layer to our project.



In order to add the assets, create a new directory images within the racingTutorial package. Once done, copy across the Car.png, Track.png and TrackHitRegions.png images (which should have been downloaded along with this tutorial) and the RacingAsset.txt asset file.

Next, create another class called **RacingCar** within the racingTutorial package.

Add in the code shown on the next page (page 7). There is a good amount of code within this class, so lets go through it chunk-by-chunk. As an aside, the version of the tutorial within TutorialsDemosAndExtras.tutorials.topDownRacer is well commented and might also provide a useful read.

As can be seen, I've added in a number of variables dealing with the car's forward and backward acceleration, braking deceleration and turning rate.

```
public double carForwardAcceleration = 0.10;
public double carBackwardAcceleration = 0.08;
public double carDeceleration = 0.975;
public double carTurningRate = 0.002;
```

I've also added in three values for frictional decelerations. One for turning and two for movement: the frictionalMovementDeceleration is the deceleration along the direction of car and frictionalDriftDeceleration is the deceleration perpendicular to the direction of the car. In other words, we want to model a stronger deceleration if the car is moving sideways, i.e. skidding around a corner.

```
public double frictionalDriftDeceleration = 0.05;
public double frictionalMovementDeceleration  = 0.98;
public double frictionalAngularDeceleration  = 0.95;
```

As can also be seen, I've defined maximum directional and angular velocities for the car.

Other than the update method, the only other method defined within this class is the main method, as follows:

```
public RacingCar( GameLayer gameLayer ) {
    super( gameLayer );
    setRealisationAndGeometry( "Car" );
}
```

The method defines the realisation and geometry of the car game object.

```
package tutorials.racingTutorial;

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

public class RacingCar extends GameObject {

    public double carForwardAcceleration = 0.10;
    public double carBackwardAcceleration = 0.08;
    public double carDeceleration = 0.975;
    public double carTurningRate = 0.002;

    public double frictionalDriftDeceleration = 0.05;
    public double frictionalMovementDeceleration  = 0.98;
    public double frictionalAngularDeceleration  = 0.95;

    public double maxVelocity = 5.0;
    public double maxAngularVelocity = 0.03;

    public RacingCar( GameLayer gameLayer ) {
        super( gameLayer );
        setRealisationAndGeometry( "Car" );
    }

    @Override
    public void update() {
        if( velocityx != 0.0 ||  velocityy != 0.0 ) {
            double absVelx = Math.abs(velocityx), absVely = Math.abs(velocityy);
            double drift = Math.abs( (absVelx/(absVelx+absVely))*Math.cos(rotation) +
                  (absVely/(absVelx+absVely)) * Math.sin(rotation) );

            double frictionDeceleration = frictionalMovementDeceleration -
                    frictionalDriftDeceleration * drift;

            velocityx *= frictionDeceleration;
            velocityy *= frictionDeceleration;
        }

        if( inputEvent.keyPressed[ KeyEvent.VK_SPACE ] ) {
            velocityx *= carDeceleration;
            velocityy *= carDeceleration;
        }

        angularVelocity *= frictionalAngularDeceleration;

        if( inputEvent.keyPressed[ KeyEvent.VK_UP ] ) {
            velocityx += Math.sin( rotation ) * carForwardAcceleration;
            velocityy -= Math.cos( rotation ) * carForwardAcceleration;
        } else if( inputEvent.keyPressed[ KeyEvent.VK_DOWN ] ) {
            velocityx -= Math.sin( rotation ) * carBackwardAcceleration;
            velocityy += Math.cos( rotation ) * carBackwardAcceleration;
        }

        double speedRatioSqrd = (velocityx*velocityx + velocityy*velocityy)
                / (maxVelocity*maxVelocity);
        if( speedRatioSqrd > 1.0 ) {
            velocityx /= Math.sqrt(speedRatioSqrd);
            velocityy /= Math.sqrt(speedRatioSqrd);
        }

        if( inputEvent.keyPressed[ KeyEvent.VK_RIGHT ] ) {
            angularVelocity += carTurningRate
                    * Math.min( 1.0, 4.0 * Math.sqrt(
                        velocityx*velocityx+velocityy*velocityy)/maxVelocity );
        } else if( inputEvent.keyPressed[ KeyEvent.VK_LEFT ] ) {
            angularVelocity -= carTurningRate
                    * Math.min( 1.0, 4.0 * Math.sqrt(
                        velocityx*velocityx+velocityy*velocityy)/maxVelocity );
        }

        if( Math.abs( angularVelocity ) > maxAngularVelocity )
            angularVelocity = maxAngularVelocity * Math.signum(angularVelocity);

        x += velocityx;
        y += velocityy;
        rotation += angularVelocity;
    }
}
```

As with the space ship from the second tutorial, the update method is central to the control of the car. The very first thing we do whenever the update method is called is to apply our 'frictional' deceleration to the car's directional and rotational velocities.

```
if( velocityx != 0.0 ||  velocityy != 0.0 ) {
    double absVelx = Math.abs(velocityx),
           absVely = Math.abs(velocityy);
    double drift = Math.abs(
          (absVelx/(absVelx+absVely))*Math.cos(rotation) +
          (absVely/(absVelx+absVely)) * Math.sin(rotation) );

    double frictionDeceleration =
               frictionalMovementDeceleration -
                    frictionalDriftDeceleration * drift;

    velocityx *= frictionDeceleration;
    velocityy *= frictionDeceleration;
}

if( inputEvent.keyPressed[ KeyEvent.VK_SPACE ] ) {
    velocityx *= carDeceleration;
    velocityy *= carDeceleration;
}

angularVelocity *= frictionalAngularDeceleration;
```

The drift value is determined by working out what percentage of the current velocity is perpendicular to the current rotation of the car (i.e. what percentage of the car's velocity can be considered to be a sideways skid). This is then used to scale the amount of drift deceleration that is applied. As can be seen from the code, the car is always subject to deceleration towards a resting state (i.e. the directional and rotational velocities are reduced every turn). If the player does not accelerate, etc. then the car will quickly come to a stop. By changing the frictional values we can control how quickly the car stops.

We next check for forward and backward acceleration and update the x and y velocities using the same approach as adopted for the spaceship.

```
if( inputEvent.keyPressed[ KeyEvent.VK_UP ] )
{
     velocityx +=
          Math.sin( rotation ) * carForwardAcceleration;
     velocityy -=
          Math.cos( rotation ) * carForwardAcceleration;
}
else if( inputEvent.keyPressed[ KeyEvent.VK_DOWN ] )
{
     velocityx -=
          Math.sin( rotation ) * carBackwardAcceleration;
       velocityy +=
          Math.cos( rotation ) * carBackwardAcceleration;
}
```

```
double speedRatioSqrd
     = (velocityx*velocityx + velocityy*velocityy)
          / (maxVelocity*maxVelocity);

if( speedRatioSqrd > 1.0 ) {
     velocityx /= Math.sqrt(speedRatioSqrd);
     velocityy /= Math.sqrt(speedRatioSqrd);
}
```

Once we have updated the x and y velocities, we then check to see if the velocity has exceeded the maximum defined speed, and reduce the velocities if needed.

We next then apply a similar process to the angular velocities, i.e. updating the velocity based upon key pressed and ensuring that the velocity does not exceed the defined maximum angular velocity. When updating the angular velocity, the change in velocity is scaled relative to the speed of the car (this is not a physically valid model, however, it provides a cheap and easy way of controlling turning rates at low speeds).

```
if( inputEvent.keyPressed[ KeyEvent.VK_RIGHT ] ) {
     angularVelocity += carTurningRate
          * Math.min( 1.0, 4.0 * Math.sqrt(
               velocityx*velocityx+velocityy*velocityy)
                    /maxVelocity );

} else if( inputEvent.keyPressed[ KeyEvent.VK_LEFT ] ) {
     angularVelocity -= carTurningRate
          * Math.min( 1.0, 4.0 * Math.sqrt(
               velocityx*velocityx+velocityy*velocityy)
                    /maxVelocity );
}

if( Math.abs( angularVelocity ) > maxAngularVelocity )
     angularVelocity = maxAngularVelocity *
          Math.signum(angularVelocity);
```

Finally, we update the x-location, y-location and rotation of the car.

You might be wondering where the (seemingly) rather precise values for carForwardAcceleration (0.10), carBackwardAcceleration (0.08), etc. came from. The values were selected through a process of trial and error. In other words, once I got the game up and running, I played about with the various parameters until I got a behaviour that *felt* kinda right – of course, how I feel about it doesn't mean that you will agree with me, nor does it mean that behaviour will be suited to all forms of driving game, i.e. please do feel free to tweak the parameters.

Phase 5: Going for a test run

Let's create a simple game layer in which we can try out the car's handling. To do this, create a new class called **RacingExampleLayer** within the racingTutorial package. Introduce the following code:

```
package tutorials.racingTutorial;

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

public class RacingExampleLayer extends GameLayer {

    private static final int Z_DEPTH_TRACK = 0;
    private static final int Z_DEPTH_CARS = 1;

    private BufferedImage trackHitRegions = null;

    private double carLastValidX;
    private double carLastValidY;
    private double carLastValidRotation;

    public RacingExampleLayer( GameEngine gameEngine ) {
        super( "RacingLayer", gameEngine );

        createTrack();
        createCar();
    }

    public void createTrack() {
        GameObject track = new GameObject( this );
        track.setRealisationAndGeometry( "Track" );
        track.setDrawOrder( Z_DEPTH_TRACK );
        addGameObject( track );

        track.x = track.width/2;
        track.y = track.height/2;

        trackHitRegions = assetManager.retrieveGraphicalAsset(
          "TrackHitRegions" ).getImageRealisation();
    }
```

```
    public void createCar() {
        RacingCar racingCar = new RacingCar( this );
        racingCar.setName( "RacingCar" );

        racingCar.setPosition( 150, 350 );
        racingCar.setDrawOrder( Z_DEPTH_CARS );

        addGameObject( racingCar);
    }

    @Override
    public void update() {
        RacingCar racingCar
          = (RacingCar)this.getGameObject( "RacingCar" );
        racingCar.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 );
    }
}
```

We've added in another game layer that is very similar to those added into the first two tutorials. However, there are a couple of notable extensions to this game layer as discussed below.

This is the first tutorial in which we need to worry about layered graphics, i.e. we want the car to be drawn on top of the background image. To do this, we define two different z-heights:

```
private static final int Z_DEPTH_TRACK = 0;
private static final int Z_DEPTH_CARS = 1;
```

Recall that lower draw order (z-orders) are drawn before higher draw orders. Whenever we create the car and background objects we will assign their z instance variable to the appropriate z-height. We can set the draw order of a game object can calling its setDrawOrder(…) method. Having setup the z-depths we don't have to worry about the draw order, i.e. this is something that the game layer will automatically manage for us during the render phase.

We also include the following instance variable:

```
private BufferedImage trackHitRegions = null;
```

When we are playing the game we will want to check our special location map to see if the car is on the road, etc. In order to do this, we will need to store the image somewhere convenient (we have already loaded the location map into the asset manager and we could simply keep accessing it from there – but it's somewhat easier if we store a reference to the image within the layer).

The constructor for the layer sets up the layer width and height based on the size of the "Track" background image. We then call two methods to create the track and game objects (the car). As we don't have a specific track class (maybe we should have one within a more complete game – it could contain start locations and waypoint information for AI cars, etc.), we make do with a GameObject instance that displays the track.

We also retrieve the TrackHitRegions ImageAsset and make use of the inherited method 'getImageRealisation' which will return a BufferedImage realisation of any graphical asset to assign the track location map to our instance variable.

Finally, when creating the racing car we make sure to give it a specific name which will permit us to retrieve the game object later during the update phase. On the topic of the update phase, we also retrieve the car and call its update method.

Phase 6: Looking at the ground

Ok, we're nearly there. The final part is to setup the car so that its performance is based upon its location upon the map. But how do we determine what's under the car?

There is actually two parts to this question – firstly a technical question about how we can lookup the colour under the car in the location map, and secondly a design question about how much of the car are we going to lookup – this can range from considering one pixel under the car to every pixel under the car.

An enlarged image of the car is shown opposite. So just how much should we consider? We could consider every pixel and then base the performance on the mixture of colours. Alternatively, we could consider the top left, top right, bottom left and bottom right pixels as a quick means of sampling (modelling the contact location of the wheels). This is a good idea for a full game; however, in this modest tutorial we will only consider one pixel – the one under the top middle of the car. It will suffice to demonstrate the approach, although it is a bit too simplistic for a proper game.

Ok, onto the technical question - how can we survey the pixel under the car's location? This is actually reasonably easy to do in Java thanks to a very nice method within the BufferedImage class, namely **getRGB**().

A couple of different types of getRGB method are available (look them up at http://java.sun.com/javase/6/docs/api/). The getRGB() methods will give us an integer (or array of integers) representing pixels in the image. Pixels are, by default, returned in the ARGB colour model (i.e. the first 8 bits represent the alpha channel, the next eight bits the red component, etc.). By considering this pixel we will be able to work out if the car is currently on a red, green or blue colour.

Let's update your RacingExampleLayer to that shown below (i.e. update your update method and add in the new considerCarLocation method).

```
public void update()
{
     RacingCar racingCar
          = (RacingCar)this.getGameObject( "RacingCar" );
     racingCar.update();

     considerCarLocation( racingCar );
}

private void considerCarLocation( RacingCar racingCar ) {
     Point carTopMiddle = new Point(
          (int)(racingCar.x),
          (int)(racingCar.y-racingCar.width/2 ));

     AffineTransform rotationalTransform = new AffineTransform();
     rotationalTransform.rotate(
          racingCar.rotation, racingCar.x, racingCar.y );

     Point carTopMiddleRotated = new Point();
     rotationalTransform.transform(
          carTopMiddle, carTopMiddleRotated );

     int pixelTopMiddleRotated = trackHitRegions.getRGB(
               carTopMiddleRotated.x, carTopMiddleRotated.y );

     if( (pixelTopMiddleRotated & 0x00ff0000) > 0 )
          carOutsideBounds( racingCar );
     else if( (pixelTopMiddleRotated & 0x000000ff) > 0 )
          carOnRough( racingCar );

     carLastValidX = racingCar.x;
     carLastValidY = racingCar.y;
     carLastValidRotation = racingCar.rotation;
}
```

The considerCarLocation method is a tad complex; let's consider it bit-by-bit. The first thing we do is to create a Point object holding the top middle location of the car. However, this assumes that the car is in its default rotation. It's likely that the car is pointing in some other direction, so we'll need to rotate this point so that it correctly follows the current car direction.

The code for creating a new AffineTransform, and setting this up so that it will rotate things around the centre point of the car by an angle equal to the current car heading can be taken as given, i.e. just use and adapt this code whenever you want to rotate something (how it does the rotation, etc. is not really of interest to us). We use the transform method of AffineTransform to rotate the carTopMiddle point, storing the result in carTopMiddleRotated.

Ok, so we've got our current car point. The next thing we do is to lookup the pixel at this point using the getRGB method. How do we determine what colour the car is resting on? We could do this in different ways, but here the approach I'm using is to use an AND operation. We know that the integer returned by getRGB is in ARGB format, so if I AND this against a bit pattern where everything but the red bits is set to zero, e.g. 00 FF 00 00, I can then check to see if the pixel contained any red colour (and ditto for the other primary colours). If you wanted to have an arbitrary number of colours, you could simply check to see if the returned colour matches (is equal) to some defined colour, i.e. we could have used if(colour == 0x00FF0000) as well to test for pure red). Either approach is fine.

Finally, if we're off road (on blue) or hit something impassable (red), we'll call a suitable method. Let's add in these new methods:

```
    private void carOutsideBounds( RacingCar racingCar ) {
        racingCar.x = carLastValidX;
        racingCar.y = carLastValidY;
        racingCar.rotation = carLastValidRotation;

        racingCar.velocityx = 0.0;
        racingCar.velocityy = 0.0;
        racingCar.angularVelocity = 0.0;
    }

    private void carOnRough( RacingCar racingCar ) {
        racingCar.velocityx *= 0.95;
        racingCar.velocityy *= 0.95;
        racingCar.angularVelocity *= 0.98;
    }
```

The carOutsideBounds method does two things. It firstly moves the car back to the last know valid location and rotation. Why do we do this? Whenever we detect the overlap the car has already collided with an impassable object, hence we need to correct for this overlap. Returning to the last known valid location (likely in the last update) provides an easy means of correcting the overlap (you will notice that we update the last valid locations at the end of the considerCarLocation() method).

The carOnRough method simply reduced the directional and rotational velocities by a sizeable amount, i.e. the car slows down noticeable.

I've included the full RacingExampleLayer class below:

```
package tutorials.racingTutorial;

import game.engine.*;

import java.awt.*;
import java.awt.image.*;
import java.awt.geom.*;

public class RacingExampleLayer extends GameLayer {

    private static final int Z_DEPTH_TRACK = 0;
    private static final int Z_DEPTH_CARS = 1;

    private BufferedImage trackHitRegions = null;

    private double carLastValidX;
    private double carLastValidY;
    private double carLastValidRotation;

    public RacingExampleLayer( GameEngine gameEngine ) {
        super( "RacingLayer", gameEngine );

        createTrack();
        createCar();
    }

    public void createTrack() {
        GameObject track = new GameObject( this );
        track.setRealisationAndGeometry( "Track" );
        track.setDrawOrder( Z_DEPTH_TRACK );
        addGameObject( track );

        track.x = track.width/2;
        track.y = track.height/2;

        trackHitRegions = assetManager.retrieveGraphicalAsset(
            "TrackHitRegions" ).getImageRealisation();
    }

    public void createCar() {
        RacingCar racingCar = new RacingCar( this );
        racingCar.setName( "RacingCar" );

        racingCar.setPosition( 150, 350 );
        racingCar.setDrawOrder( Z_DEPTH_CARS );

        addGameObject( racingCar);
    }

    @Override
    public void update() {
        RacingCar racingCar = (RacingCar)this.getGameObject( "RacingCar" );
        racingCar.update();

        considerCarLocation( racingCar );
    }
```

```
    private void considerCarLocation( RacingCar racingCar ) {
        Point carTopMiddle = new Point(
                (int)(racingCar.x), (int)(racingCar.y-racingCar.width/2 ));

        AffineTransform rotationalTransform = new AffineTransform();
        rotationalTransform.rotate(
            racingCar.rotation, racingCar.x, racingCar.y );

        Point carTopMiddleRotated = new Point();
        rotationalTransform.transform( carTopMiddle, carTopMiddleRotated );

        int pixelTopMiddleRotated = trackHitRegions.getRGB(
                carTopMiddleRotated.x, carTopMiddleRotated.y );

        if( (pixelTopMiddleRotated & 0x00ff0000) > 0 )
            carOutsideBounds( racingCar );
        else if( (pixelTopMiddleRotated & 0x000000ff) > 0 )
            carOnRough( racingCar );

        carLastValidX = racingCar.x;
        carLastValidY = racingCar.y;
        carLastValidRotation = racingCar.rotation;
    }

    private void carOutsideBounds( RacingCar racingCar ) {
        racingCar.x = carLastValidX;
        racingCar.y = carLastValidY;
        racingCar.rotation = carLastValidRotation;

        racingCar.velocityx = 0.0;
        racingCar.velocityy = 0.0;
        racingCar.angularVelocity = 0.0;
    }

    private void carOnRough( RacingCar racingCar ) {
        racingCar.velocityx *= 0.95;
        racingCar.velocityy *= 0.95;
        racingCar.angularVelocity *= 0.98;
    }


    @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 );
    }
}
```