Games Programming

Philip Hanna 110CSC207

Tutorial 2: 'Body Example'

Tutorial Overview:

This tutorial will explore how to make use of the physics based class in terms of producing a simple moving object that responds to user control and can collide with other objects.

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 tutorial 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 develop a couple of classes that extend the Body class, which in turn extends the GameObject class. To be exact, we are going to have one spaceship which will be under the control of the player and a number of planets objects which won't do much.

The player can use the keyboard to move the spaceship around the screen. If the spaceship collides with a planet it will bounce off the planet. Finally, to make things a bit more interesting, we'll also introduce a form 'gravitational' attraction pulling the spaceship towards the centre of the screen.

Following this tutorial you will hopefully have a better understanding of how you can go about creating objects that can make respond to user input and interact with one another.

Phase 1: Creating a new package in which to build up the game

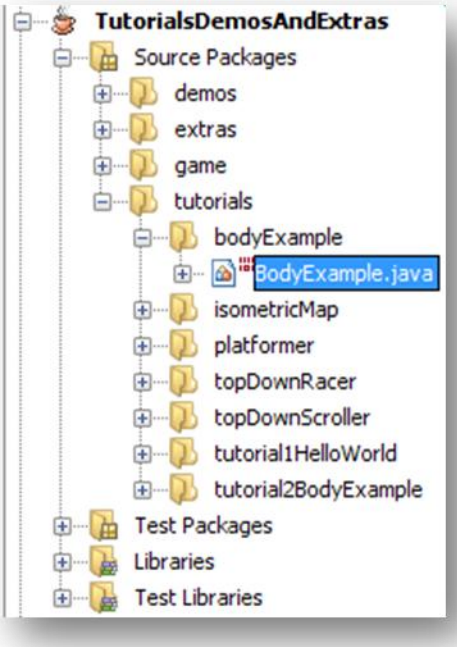
Let's create a new package in which to develop our sprite example. If needed you might want to revisit the instructions within Phase 1 of the Hello World tutorial if you are a bit uncertain about how to create a new package. Ok, so please go ahead and create a new package named 'bodyExample' within the 'tutorials' package.

Phase 2: Extending the adapting the GameEngine class.

The instructions and rational for this are also closely based on the Hello World tutorial – you might wish to revisit the tutorial if needed.

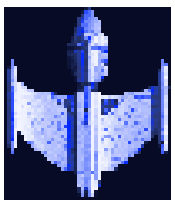
We want to create a new class, entitled BodyExample.java that extends the GameEngine class. We also want to setup the class imports and the class header. To do this, create a new class entitled BodyExample within the bodyExample package and modify the code to that shown below:

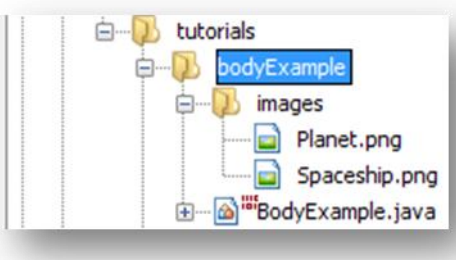
```
package tutorials.bodyExample;

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

public class BodyExample
             extends GameEngine {









}
```



We next want to add in the assets that we will need to use within this game. To do this, create a new folder entitled 'images' within the spriteExample folder. If needed, please revisit the first tutorial for instructions.

Once this is done you will then want to copy across the two graphical images that were packaged along with this tutorial, Planet.png and Spaceship.png, into the newly created images folder. All being well, NetBeans will update to show you the two image assets that have been copied across. If you want, you can double click on the images to view them within NetBeans





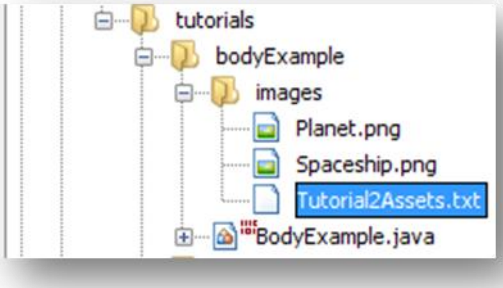
Phase 3: Loading Assets

We next want to load in the two graphical assets we will use within this game. For this tutorial, both assets are static images that are not animated. We could use a similar approach to that used within the first tutorial, i.e. explicitly writing code that loads each asset. There are two disadvantages to explicitly writing code. The first is that it quickly becomes tedious if we need to load a lot of different images. The second is that it's generally a good idea to, in the parlance of software engineering, use 'loose coupling' between components – this helps to weaken interdependence between components and improves maintainability/extensibility, etc. Expect more of this fun within your software engineering module next semester. Thankfully, there is a better way of loading assets.

If you open up the game.assets.AssetManager class and browse to the loadAssetsFromFile method (you can directly select this method from the navigator window) you will find a method for, well, loading assets from a file.

To get things going, I've included a prewritten file (Tutorial2Assets.txt) that contains instructions for loading the two images involved with this project. The first thing we need to do is to copy Tutorial2Assets.txt into the images folder you created. Once you've done that, double click on the file to open it.

```
// …

ImageAsset Planet     Planet.png
ImageAsset Spaceship  Spaceship.png
```



Have a look at the first non-commented line: the ImageAsset keyword indicates that we will load an image, which we will refer to as 'Planet' (i.e. we'll use this name whenever we want to get it from the AssetManager) which resides in a file 'Planet.png'. The header in Tutorial2Assets.txt provides the line format for loading other types of asset – expect more about this in later tutorials and within lectures (Aside: an XML based approach would have been idea, but outside the scope of this module).

In order to load the assets, add the following code to your BodyExample.java file (recall that the GameEngine will automatically call the buildAssetManager method to load in the assets).

```
public boolean buildAssetManager()
{
   assetManager.loadAssetsFromFile(
          getClass().getResource(
               "images/Tutorial2Assets.txt"));

   return true;
}
```

We have now loaded in the two images needed by the game as two graphical assets entitled "Spaceship" and "Planet". As before, we use the getClass().getResource() to get a file relative to the location of the class file.

Phase 4: game.physics.Body objects

Let us next create a spaceship sprite that we can use within our game. To do this we need to make use of the Body class which provides us with lots of nice inherited variables and methods that we can use to define how the body will interact with other bodies.

So, given a Body object what is it that we get through inheritance? Well, a number of things – if you look at the class you'll see that Body itself inherits from GameObject. That's a lot of inheritance and we end up with a lot of inherited methods and instance variables. Thankfully this is nothing to be worried about – some of the Sun classes have hundreds of inherited methods, what matters is knowing which ones are likely to be useful in a given situation.

I've highlighted below the key inherited values that may be of interest/use within games. There are a good few values so don't worry too much about being able to take them all in at once – we will see how we can use them later in the tutorial.

Important aside: In the code repository I've managed to walk all over one of the key principles of object-oriented design – data encapsulation. In the vast majority of cases, class data should be declared to be private and get/set methods provided to access the data. This helps ensure that anyone using the class can't change the data in a manner unintended by the class author. However, as you'll see below, I've made extensive use of public data. The reason for this is speed – it normally takes a much longer time to call a method to access a value as opposed to accessing the value directly. The downside is that in the text for the following values you'll see the odd warning about not changing the value directly.

Inherited values from game.engine.GameObject

- public double **x** – x location of the game object (top left corner). This location is relative to the layer in which the game object belongs. For example, if the game layer has the same width and height as the screen, then the x location will correspond to the screen x location. However, if the game layer is larger than the screen, e.g. in a side-scrolling game, then the x location of a given game object may be 'outside' of that visible on the screen.

- public double **y** – y location of the game object (top left corner). Same comments as for the x location apply here.

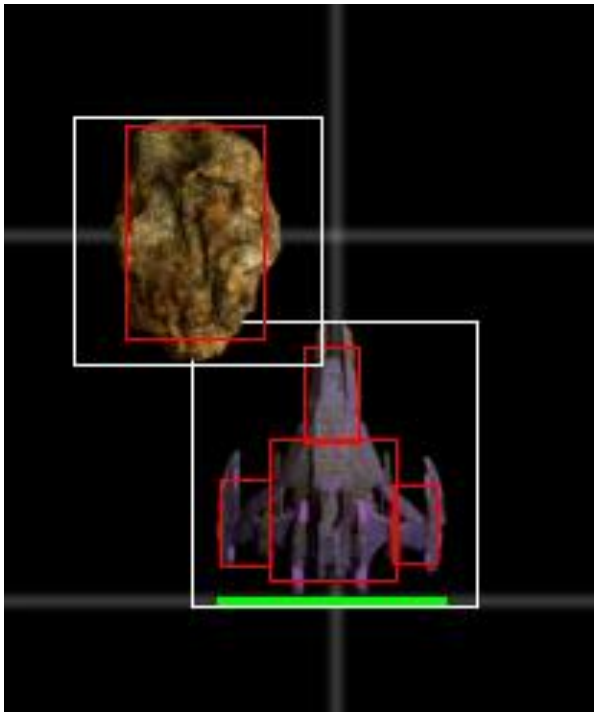
- public double **width** – width of the game object. Upon initialisation this is zero and will be automatically changed to reflect the defined geometry used by the game object, i.e. whenever we tell a game object to use a particular geometry then the width is automatically set to reflect the defined geometry. This value should not be set directly – i.e. you should define the geometry of the object to set this value.

- public double **height** – height of the game object. The same comments as for the object width also apply to its height.

- public double **rotation** –rotation of the game object, measured in radians (2Π radians = 360 degrees, i.e. 1 degree ~ 0.017 radians).

- public double **velocityX** – velocity of this game object along the x-axis. This is a velocity measured in terms of the number of pixels per second.
- public double **velocityY** – velocity of this game object along the y-axis. Same comments as for the x velocity apply here.
- public double **angularVelocity** – angular, i.e. rotational velocity of the game object, controlling how quickly it turns. The angular velocity is measured in radians/seconds.
- public int **drawOrder** – z depth of the game object. This variable controls the draw ordering of the game object. Objects with a lower draw order are drawn first, and then followed by objects with a higher draw order depth. You should never set this value directly, instead call the GameObject.setDrawOrder(…) method!
- public boolean **canIntersectOtherGraphicalObjects** – if this is set to false then the object cannot intersect with any other object (and will be ignored within any collision detection tests). It is a good idea to ensure this is set to false for as many objects as possible – as this ensures that collision checking is as fast as possible.
- public GraphicalAsset[] **graphicalRealisation** – array of graphical assets that visually defined the game object (in most cases, a game object will only be displayed as a single image, however, you can have game objects that are composed of a number of different images). You should update the realisation by using the various setRealisation(…) / setRealisationAndGeometry() methods.
- public Shape[] **geometry** – each object maintains a number of bounds (either rectangular or circular) that are used to check for collision, etc. (this will be explored in detail later in the module). For the purpose of this tutorial, and others, we just have to remember to define the geometry of a game object if we want it to interact with other objects. You should update the realisation by using the various setGeometry(…) / setRealisationAndGeometry() methods. See below for an example of a game object that has a geometry consisting of several rectangles:

The white boxes are (kinda) reflective of the width and height of the game objects. As can be seen the white boxes overlap. However, the asteroid has been defined to have one rectangular bound (shown in red), whilst the spaceship has a total of four rectangular bounds (also shown in red). As you can see the red boxes do not overlap, hence in terms of a collision the spaceship is not considered to have collided with the asteroid.



Inherited values from game.physics.Body

- public double **forcex** – force applied to the body along the x axis. By setting this, the physics engine will apply the specified force to the body along the x axis. Depending upon the mass, etc. of the body, this will result in movement. You could also just set the velocityx of the body, although by setting this you can apply the same force to a bunch of bodies and let the physics engine then work out the consequence.
- public doule **forcey** – force applied to the body along the y axis. The same comments as for forcex also apply here.
- public double **torque** – force driving angular velocity. The same comments as for forcex also apply here.
- public double **friction** – coefficient of friction for the body, determining how easily it will slide across other objects.
- public double **restitution** – restitution of the body, determining how much energy will be retained following a collision, e.g. a bouncy ball has a high restitution, a deflated football a low restitution, etc.
- public double **mass** – the mass of the body, i.e. how heavy it is. You should never directly set the mass, but instead use the Body.setMass(…) method. If you don't want a body is move, i.e. the body should be immovable, then you can do this by setting the mass as follows body.setMass(Body. INFINITE_MASS).
- public boolean **gravityEffected** – the physics engine will automatically apply a velocity increase on all bodies based on the set gravitational strength (Note: immovable body, i.e. those with an 'infinite' mass, are not effected). If you have a body that you'd prefer not to be subject to gravity, then set this flag to false (it's true by default).

Phew, the above has thrown an awful lot of instance variables at you. Lets see how some of them can be used.

Phase 5: Create a spaceship sprite

Now that we have looked at the various variables we inherit from Body and GameObject, let's create a sprite object for our spaceship. To do this, create a new class entitled Spaceship within the bodyExample package. Next, add in the following code to Spaceship.java:

```
package tutorials.bodyExample;

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

public class Spaceship extends Body {

    public Spaceship(GameLayer gameLayer) {
        super(gameLayer);
    }
}
```

Let's now go about customising our sprite to get the desired form of behaviour for our spaceship. First question - where should we locate the spaceship within the game layer? (I'm getting slightly ahead of myself here as we've yet to define a game layer). Let's make an arbitrary decision and locate the space ship at the centre of the layer. Let's also make the spaceship point due east (remember 90 degrees = Π/2 radiants. Add the code shown into the constructor (after the super keyword).

```
x = gameLayer.width / 2;
y = gameLayer.height / 2;

rotation = Math.PI / 2.0;
```

Ok, so far we've defined a spite that will appear at the centre of the game layer. Let's also give it a mass and also define its graphical look and geometry. Add the following code to the constructor:

```
setMass(100.0);

setRealisationAndGeometry("Spaceship");
```

The spaceship will have a mass of 100.0, a graphical appearance of the spaceship image we will load and a rectangular geometry equal to the size of the spaceship image. We've now defined our spaceship – i.e., we are using the default values of the other game object values, e.g. friction, restitution, etc. (although you might want to try playing about with these once you get the tutorial finished).

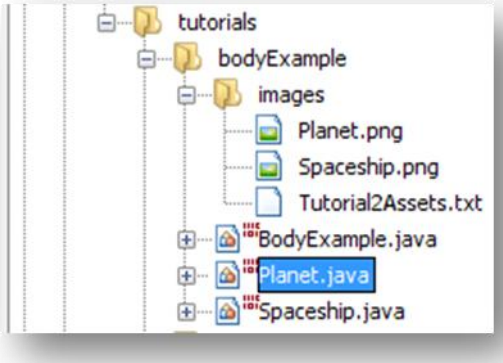
Aside: The bounding region for the spaceship is a poor choice as the spaceship is surrounded by a reasonable amount of white space – but it will suffice for this tutorial (you can find more on collision detection within Week 9).

If everything has gone to plan your Spaceship.java class should now look like that shown:

```
package tutorials.bodyExample;

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

public class Spaceship extends Body {

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

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

        rotation = Math.PI / 2.0;

        setMass(100.0);

        setRealisationAndGeometry("Spaceship");
    }
}
```

Phase 6: Create a planet sprite

Let's create another sprite – this time a planet sprite. To start, create a new class called Planet within the bodyExample package.



We'll do this class in one big step… add in the following code to your Planet class:

```
package tutorials.bodyExample;

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

public class Planet extends Body {

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

        setRealisation("Planet");
        setGeometry(new Circle(0, 0, 25));

        setMass(Double.MAX_VALUE);
        restitution = 1.0;
    }
}
```

In this example we are not bothering to define an initial position (which defaults to (0,0)) – we'll do that later when we create the objects.

We do, however, set the graphical realisation to the planet image we load and we've given the planet a circular geometry by creating a new Circle instance (as found within the game.geometry package). The first two circle parameters specify the offset of the geometry relative to the centre of the body, i.e. a value of (0,0) means that the circle geometry is centred on the game object. The third value specifies the radius of the circle.

In respect of collisions we define a restitution of 1.0 (i.e. things will bounce of the planet) and provide each planet with an infinite mass (Double.MAX_VALUE is also the same thing as Body. INFINITE_MASS), i.e. the planet won't move.

Phase 6: Bringing things together (Part 1)

Ok, we've created two different types of body; let's now create a game layer within which we can build a number of different body instances. To do this, create a new class BodyExampleLayer and add in the following code:

```
package tutorials.bodyExample;

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

public class BodyExampleLayer extends CollisionSpace {

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

        width = gameEngine.screenWidth;
        height = gameEngine.screenHeight;

        MAXIMUM_TRAVEL_VELOCITYX = 500.0;
        MAXIMUM_TRAVEL_VELOCITYY = 500.0;
        MAXIMUM_ANGULAR_VELOCITY = 5.0;

        createGameObjects(); // We'll add this shortly
    }

    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);
    }
}
```

You will notice that we directly extend CollisionSpace as opposed to GameLayer (CollisionSpace extends GameLayer). The reason for this is that CollisionSpace is a special type of GameLayer that holds the physics simulation. In particular, whenever you call the update() method on a CollisionSpace instance it will update all bodies contained within the CollisionSpace taking into account applied forces, gravity, etc. The CollisionSpace will also check for collisions between bodies and ensure that the bodies rebound, etc. correctly. In short, CollisionSpace does all the hard work in terms of the physics.

CollisionSpace defines a number of parameters that can be used to control the physics simulation (open game.physics.CollisionSpace and have a look). In this example, we are setting three of them to define maximum velocities for the bodies (i.e. this helps ensure that things don't move too fast). We also set the width and height of the layer to be equal to the size of the screen. Finally, we also include a draw method that will blank the screen, before calling super.draw(…) to draw all game objects.

If you've entered the code shown on the last page you should find that NetBeans grumbles about not being able to find the createGameObjects() method. Let's add it now.

```
public void createGameObjects() {

   addGameObjectCollection("Planets");

   Spaceship spaceship = new Spaceship(this);
   spaceship.setName("Spaceship");
   addGameObject(spaceship);

   int numPlanets = 5;
   for (int planetIdx = 0; planetIdx < numPlanets; planetIdx++) {
     Planet planet = new Planet(this);
     planet.x = gameEngine.randomiser.nextInt((int) width);
     planet.y = gameEngine.randomiser.nextInt((int) height);

     addGameObject(planet, "Planets");
   }
}
```

This code is somewhat similar to that used as part of the Hello World tutorial. We define one game object set called "Planets" to which we add a number of planets game object (which we randomly position within the layer). We also create a total of one Spaceship object – called "Spaceship" which is duly added to the layer. We have explicitly set the name of the spaceship game object to "SpaceShip" as we'll want to be able to retrieve this object layer.

Ok, now that we've got a game layer and have added a number of game objects to it, let's add the game layer into the game engine. Add the following method into BodyExample:

```
protected boolean buildInitialGameLayers() {
     BodyExampleLayer bodyExampleLayer = new
          BodyExampleLayer(this);
     addGameLayer(bodyExampleLayer);

     return true;
}
```

Also add in the following constructor (previously we started the game at a fixed resolution, however, in this case we're going to be bold and run the game at the screen's current resolution (which will likely be higher than 1024x768)

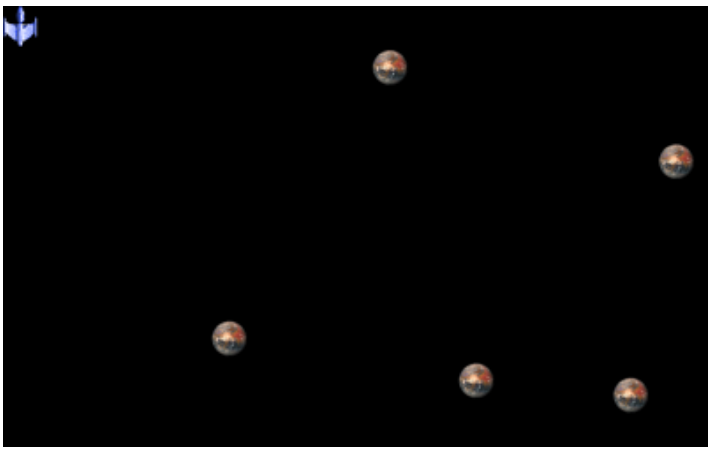
```
public BodyExample() {
   DisplayMode currentDisplayMode =
      GraphicsEnvironment.getLocalGraphicsEnvironment().
                getDefaultScreenDevice().getDisplayMode();

   gameStart(currentDisplayMode.getWidth(),
             currentDisplayMode.getHeight(),
             currentDisplayMode.getBitDepth());
}
```

For good measure, also add in a main method, which should make your completed BodyExample.java as follows:

```
package tutorials.bodyExample;

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

public class BodyExample extends GameEngine {

    public BodyExample() {
        DisplayMode currentDisplayMode =
              GraphicsEnvironment.getLocalGraphicsEnvironment().
                getDefaultScreenDevice().getDisplayMode();

        gameStart(currentDisplayMode.getWidth(),
               currentDisplayMode.getHeight(),
               currentDisplayMode.getBitDepth());
    }

    public boolean buildAssetManager() {
        assetManager.loadAssetsFromFile(
          getClass().getResource("images/Tutorial2Assets.txt"));

        return true;
    }

    protected boolean buildInitialGameLayers() {
        BodyExampleLayer bodyExampleLayer
               = new BodyExampleLayer(this);
        addGameLayer(bodyExampleLayer);

        return true;
    }

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

At this point you can now run your game – it won't do anything - other than hopefully display five planets and one spaceship (which will be subject to gravity) on the screen.

We'll next look at how we can add some interaction to the game.



Phase 7: Adding some interaction

Let's extend out basic game by adding in some interaction. A good place to start is with the spaceship in terms of enabling the user to fly it around the screen. How should we do this? A good approach would be to let the update method within the Spaceship class check the keyboard input and update the x, y and rotational velocities in accordance with key presses.

Open up the Spaceship class.

We want to setup the input so that if the user hits the left and right arrow keys the spaceship will rotate to the left and right. Additionally, let's permit the user to move the spaceship forward or backwards (along the direction of rotation) by using the up and down arrow keys. This gives us two problems to solve. The first one is how fast should the spaceship move, and the second is how do we map a velocity along the ships rotation onto the individual x and y velocities that are defined within GameObject.

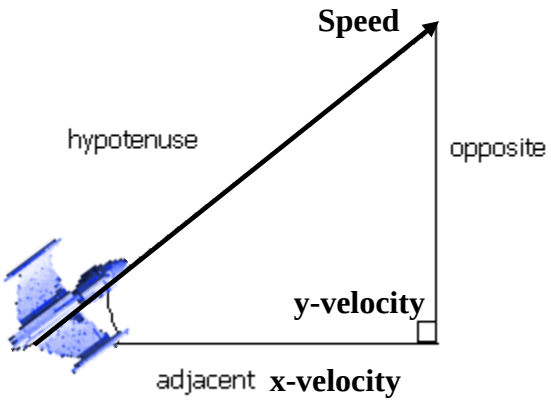
The first problem is easier to solve: add in the following class instance variables to Spaceship.java (i.e. place them before the constructor) which defines some movement factors we will later use.

```
    private double shipMovementAcceleration = 10.0;
    private double shipRotationalAcceleration = 0.15;
    private double shipRotationalDampening = 0.1;
```

We will use the shipMovementAcceleration to determine how quickly the ship can accelerate forward and backwards. Likewise, we'll use the shipRotationalAcceleration to determine how quickly the ship can rotate left and right. The third variable, shipRotationalDampening, brings up the notion of dampening. Do we want moving and rotating game objects to return to a non-moving/non-rotating state if the user does not press any keys? We've assumed in this game that we will dampen the rotational but not the velocity.

The other problem is a bit more complicated and requires us to use some trig, see the diagram below:

Basically, if the spaceship if going at a certain speed in a certain direction, then we can use the cos and sin relationships to map the speed onto the corresponding x and y component velocities. Within this class we will use this principle to map the speed onto changes in the x and y velocities.



To do this, add the following update method to Spaceship.java:

```
public void update() {

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


   if (inputEvent.keyPressed[KeyEvent.VK_RIGHT])
   {
     angularVelocity += shipRotationalAcceleration;
   }
   else if (inputEvent.keyPressed[KeyEvent.VK_LEFT])
   {
     angularVelocity -= shipRotationalAcceleration;
   }


   if (Math.abs(angularVelocity) < shipRotationalDampening)
   {
     angularVelocity = 0.0;
   }
   else
   {
     angularVelocity -=
        shipRotationalDampening * Math.signum(angularVelocity);
   }
}
```

Have a ponder of the code that you have just added – we have three sets of if-else statement. The first set of conditions check to see if the up or down cursor keys have been pressed. If this is the case, we update the x and y velocities based upon the current heading of the spaceship and the defined speed of the ship. The second if-else block checks to see if the left or right cursor keys have been pressed and updates the rotational velocity as appropriate. The final if-else block dampens the rotational velocity of the ship.

Phew, we've now got our spaceship defined. I've included the full code for Spaceship.java below (a bit compacted to get things onto one page):

```
package tutorials.bodyExample;

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

public class Spaceship extends Body {

    private double shipMovementAcceleration = 10.0;
    private double shipRotationalAcceleration = 0.15;
    private double shipRotationalDampening = 0.1;

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

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

        rotation = Math.PI / 2.0;

        setMass(100.0);

        setRealisationAndGeometry("Spaceship");
    }

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

        if (inputEvent.keyPressed[KeyEvent.VK_RIGHT]) {
            angularVelocity += shipRotationalAcceleration;
        } else if (inputEvent.keyPressed[KeyEvent.VK_LEFT]) {
            angularVelocity -= shipRotationalAcceleration;
        }

        if (Math.abs(angularVelocity) < shipRotationalDampening) {
            angularVelocity = 0.0;
        } else {
            angularVelocity -=
                    shipRotationalDampening * Math.signum(angularVelocity);
        }
    }
}
```

Having defined how we wish to update our Spaceship object we next need to make sure that we call the update method. We can do this within the layer update method (which will be automatically called by the game engine).

Add the following code to your BodyExampleLayer class:

```
public void update() {
    Spaceship spaceship
      = (Spaceship) this.getGameObject( "Spaceship" );

    spaceship.update();

    super.update();
}
```

This update method will ensure that whenever the layer update method is called we will retrieve the Spaceship object and call its update method. We also call the magic super.update() method that will get the physics simulation to do its thing and make sure bodies move, interact, etc. correctly.

Ok, let's run the game at this point. You will hopefully be able to pilot your spaceship around the level. However, if you're not careful you'll run off the edge of the level! That's not really what we are after. Update the code with the following line (place it after the super.update() method call, i.e. after the point where the physics engine has determined the new location for the spaceship).

```
GameObjectUtilities.reboundIfGameLayerExited(spaceship);
```

Whenever you run the game again you will now notice that you bounce off the edges of the layer. As an aside, have a look within game.engine.GameObjectUtilities, it contains a number of sometimes useful method that you can use.

Phase 8: Gravity

Finally, for a bit of fun, what would happen if we wanted to add some extra 'gravity' to our game? The CollisionSpace class already defines gravity – you can see this in effect whenever you run the game and don't press any keys. We could tweak this using the CollisionSpace.setGravity(…) method. However, lets do something a bit different. Add the following code to the BodyExampleLayer update method (before the super.update() but after the spaceship.update()).

```
double gravityStrength = 5.0;
if (spaceship.x < this.width / 2) {
   spaceship.velocityx += gravityStrength;
} else {
   spaceship.velocityx -= gravityStrength;
}

if (spaceship.y < this.height / 2) {
   spaceship.velocityy += gravityStrength;
} else {
   spaceship.velocityy -= gravityStrength;
}
    }
```

The code shown above will check the location of the spaceship and adjust the spaceship's velocity in such a manner that it is drawn to the centre of the screen. Aside: If we really wanted to have a point form of gravitational attraction then the above is too simple and we'd need to change the x and y velocities based upon the distance and angle of the spaceship to the gravitational point.

Important and useful: You already know that, by default, ESCAPE will exit the game. You can also use CTRL-I to display performance information and CTRL-B to display extra graphical information (showing each game object's bounding area, geometry, points of contact, etc.).

Completed classes

The complete Spaceship class has already been shown - on the next couple of pages you can find completed versions of the other classes.

BodyExample.java

```
package tutorials.bodyExample;

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

public class BodyExample extends GameEngine {

    public BodyExample() {
        DisplayMode currentDisplayMode =
                GraphicsEnvironment.getLocalGraphicsEnvironment().
                getDefaultScreenDevice().getDisplayMode();

        gameStart(currentDisplayMode.getWidth(),
                currentDisplayMode.getHeight(), currentDisplayMode.getBitDepth());
    }

    public boolean buildAssetManager() {
        assetManager.loadAssetsFromFile(
                getClass().getResource("images/Tutorial2Assets.txt"));

        return true;
    }

    protected boolean buildInitialGameLayers() {
        BodyExampleLayer bodyExampleLayer = new BodyExampleLayer(this);
        addGameLayer(bodyExampleLayer);

        return true;
    }

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

Planet.java

```
package tutorials.bodyExample;

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

public class Planet extends Body {

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

        setRealisation("Planet");
        setGeometry(new Circle(0, 0, 25));

        setMass(Double.MAX_VALUE);
        restitution = 1.0;
    }
}
```

BodyExampleLayer.java

```
package tutorials.tutorial2BodyExample;

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

public class BodyExampleLayer extends CollisionSpace {

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

        width = gameEngine.screenWidth;
        height = gameEngine.screenHeight;

        MAXIMUM_TRAVEL_VELOCITYX = 500.0;
        MAXIMUM_TRAVEL_VELOCITYY = 500.0;
        MAXIMUM_ANGULAR_VELOCITY = 5.0;

        createGameObjects();
    }

    public void createGameObjects() {
        addGameObjectCollection("Planets");

        Spaceship spaceship = new Spaceship(this);
        spaceship.setName("Spaceship");
        addGameObject(spaceship);

        int numPlanets = 5;
        for (int planetIdx = 0; planetIdx < numPlanets; planetIdx++) {
            Planet planet = new Planet(this);
            planet.x = gameEngine.randomiser.nextInt((int) width);
            planet.y = gameEngine.randomiser.nextInt((int) height);

            addGameObject(planet, "Planets");
        }
    }

    public void update() {
        Spaceship spaceship = (Spaceship) this.getGameObject( "Spaceship" );
        spaceship.update();

        double gravityStrength = 5.0;
        if (spaceship.x < this.width / 2) {
            spaceship.velocityx += gravityStrength;
        } else {
            spaceship.velocityx -= gravityStrength;
        }
        if (spaceship.y < this.height / 2) {
            spaceship.velocityy += gravityStrength;
        } else {
            spaceship.velocityy -= gravityStrength;
        }

        super.update();

        GameObjectUtilities.reboundIfGameLayerExited(spaceship);
    }

    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);
    }
}
```