Panda Soccer

A tutorial on 3D Video Game Programming with Panda3D.

Nelson Rodrigues

Centro Universitário de Mérida

04 of May, 2010

0. Recommended research

The focus of this tutorial is to teach how to program a very simple game in Panda3D, with physics, network and other basic concepts.

It is not the intention of this work to teach basic programming, the python language, or advanced Panda3D techniques.

You are encouraged to research the following subjects, in order to expand the knowledge that you will need to understand this tutorial fully, and to create more advanced games.

- Python (Tutorials at http://www.python.org/doc)
- Panda3D Manual (http://www.panda3d.org/wiki/index.php/Main_Page)
- Panda3D API Reference (http://www.panda3d.org/reference/python/annotated.php)

This tutorial covers:

- Basic game programming
- Basic networking
- Basic physics
- Setting up an installation of a Panda3D IDE on a Windows environment

This tutorial does **not** cover:

- Basic programming concepts
- 3D modelling
- Setting up an installation of Panda on a non-Windows environment

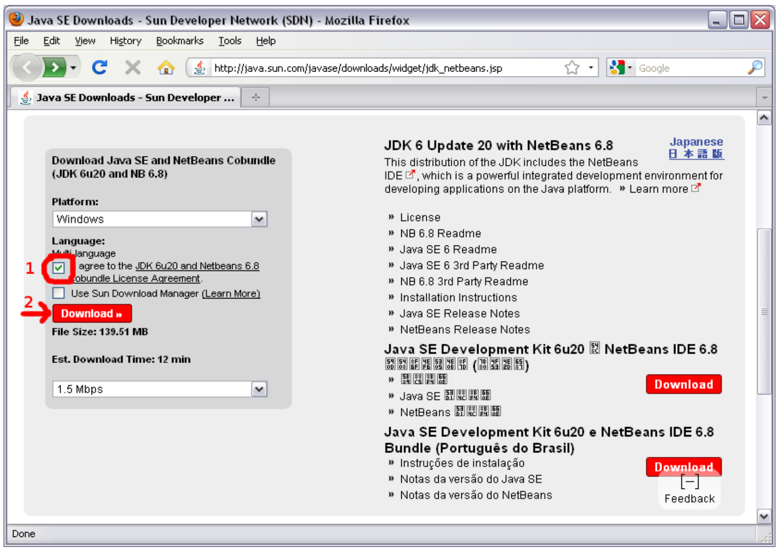
All the models and textures used in this tutorial will be previously provided, and at the time of writing are hosted at http://newita.net/?page_id=740

Most of the models, textures and even some of the text and code are from open-source and/or free sources, mainly - but not limited to - the Panda3D manual and samples.

1. Installing NetBeans

1. Navigate to http://java.sun.com/javase/downloads/widget/jdk_netbeans.jsp

2. Download and install the Java SE and NetBeans bundle (jdk-6u19-nb-6_8-windows-ml.exe)


Java SE Downloads - Sun Developer Network (SDN) - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://java.sun.com/javase/downloads/widget/jdk_netbeans.jsp
Google
Java SE Downloads - Sun Developer ...
1
2
Feedback
Done

Download Java SE and NetBeans Cobundle (JDK 6u20 and NB 6.8)

Platform: Windows

Language: Multi-language

☑ I agree to the JDK 6u20 and Netbeans 6.8 Cobundle License Agreement.

☐ Use Sun Download Manager (Learn More)

Download »

File Size: 139.51 MB

Est. Download Time: 12 min

1.5 Mbps

JDK 6 Update 20 with NetBeans 6.8

Japanese 日本語版

This distribution of the JDK includes the NetBeans IDE, which is a powerful integrated development environment for developing applications on the Java platform. » Learn more

- License
- NB 6.8 Readme
- Java SE 6 Readme
- Java SE 6 3rd Party Readme
- NB 6.8 3rd Party Readme
- Installation Instructions
- Java SE Release Notes
- NetBeans Release Notes

Java SE Development Kit 6u20 NetBeans IDE 6.8

Download

Java SE Development Kit 6u20 e NetBeans IDE 6.8 Bundle (Português do Brasil)

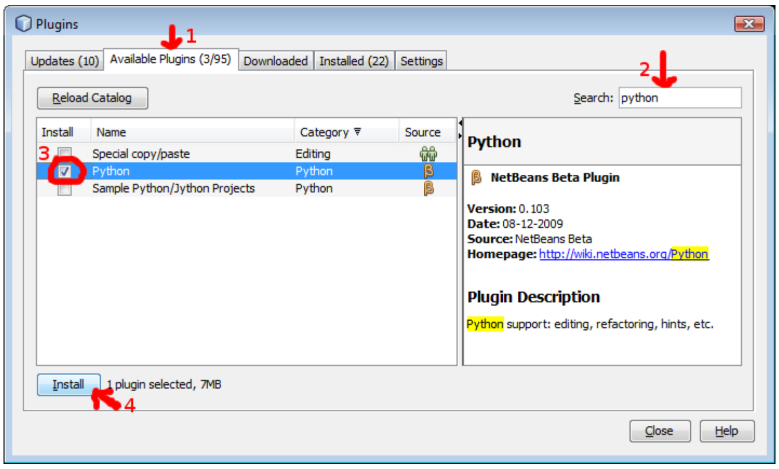
- Instruções de instalação
- Notas da versão do Java SE
- Notas da versão do NetBeans

Download

3. Execute NetBeans

4. Click Tools->Plugins

5. On the "Available Plugins" tab, search for the Python plugin, and install it.



Search: python

Install	Name	Category	Source
☐	Special copy/paste	Editing	
☑	Python	Python	
☐	Sample Python/Jython Projects	Python	

Python

NetBeans Beta Plugin

Version: 0.103
Date: 08-12-2009
Source: NetBeans Beta
Homepage: http://wiki.netbeans.org/Python

Plugin Description

Python support: editing, refactoring, hints, etc.

Install 1 plugin selected, 7MB

6. Restart NetBeans.

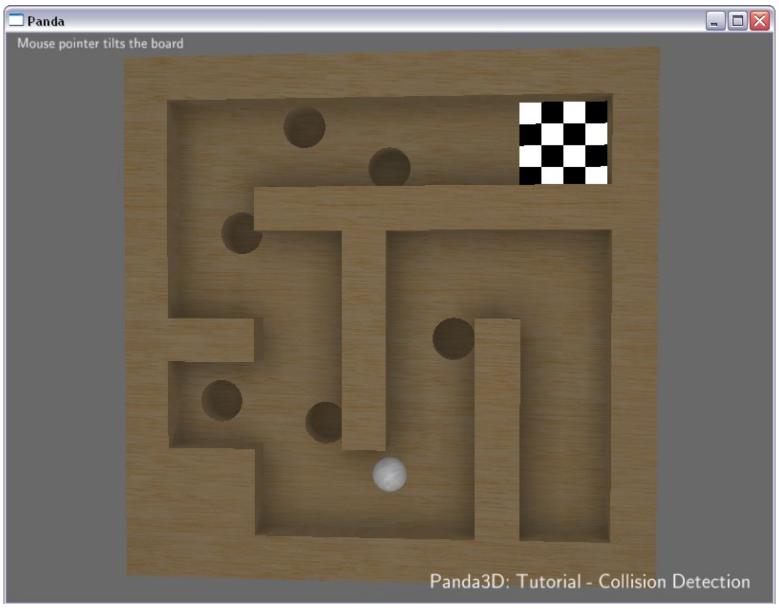
2. Installing Panda3D

1. Navigate to http://www.panda3d.org/download.php?sdk

2. Download and install Panda3D SDK 1.7.0 (Panda3D-1.7.0.exe)

3. Execute the Ball in Maze sample program, via the Start Menu.
(Click Start -> All Programs -> Panda3D 1.7.0 -> Sample Programs
-> Ball in Maze -> Run Ball in Maze)

The following window should appear:


Panda
Mouse pointer tilts the board
Panda3D: Tutorial - Collision Detection

If the game is playable, then Panda3D is correctly installed.

3. Integrating Panda3D with NetBeans

1. In NetBeans, click "Files -> New Project"

2. Select "Python -> Python Project"

3. Name your project "PandaSoccer".

4. Click "Finish".

5. On "pandasoccer.py", delete every line of the file, and write:

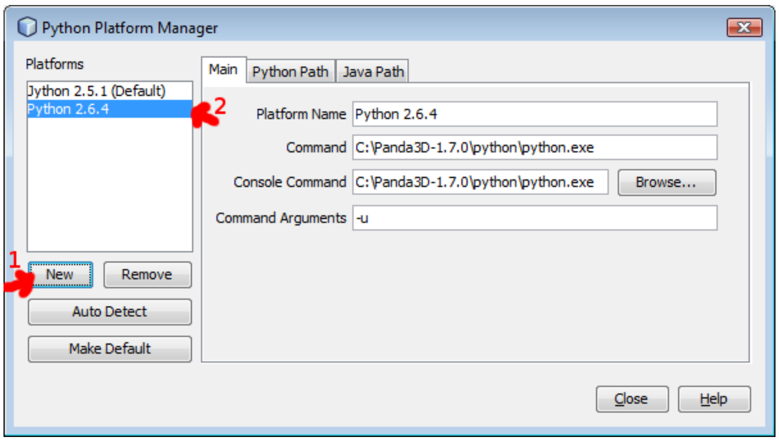
```
print "Hello World"
```

6. Run the program

If everything went well, your console will have the output "Hello World", which will mean the Python plug-in is working.

7. Click "Tools -> Python Platforms"

8. Click "New", browse to Panda's python folder (for example c:\panda3d-1.7.0\python), and select "python.exe"



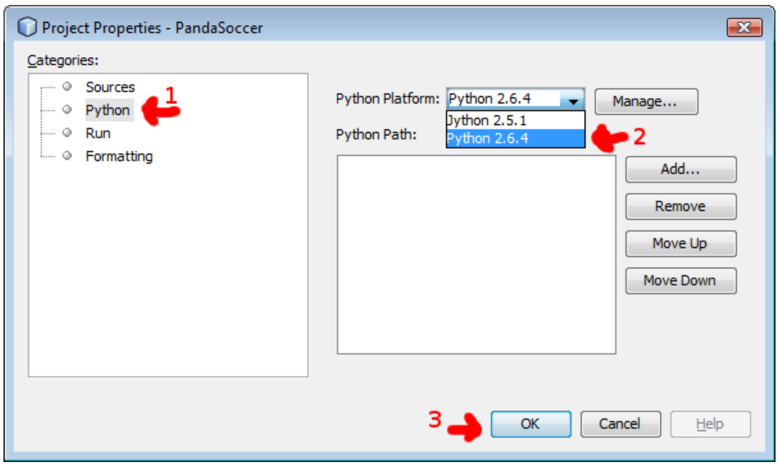
Platform Name: Python 2.6.4
Command: C:\Panda3D-1.7.0\python\python.exe
Console Command: C:\Panda3D-1.7.0\python\python.exe
Command Arguments: -u

Python 2.6.4 should now be listed under platforms.

9. Close the window.

10. On the Projects tab, right-click the name of your project, and select Properties.

11. Under Categories, select "Python", then on Python Platform, select "Python 2.6.4".



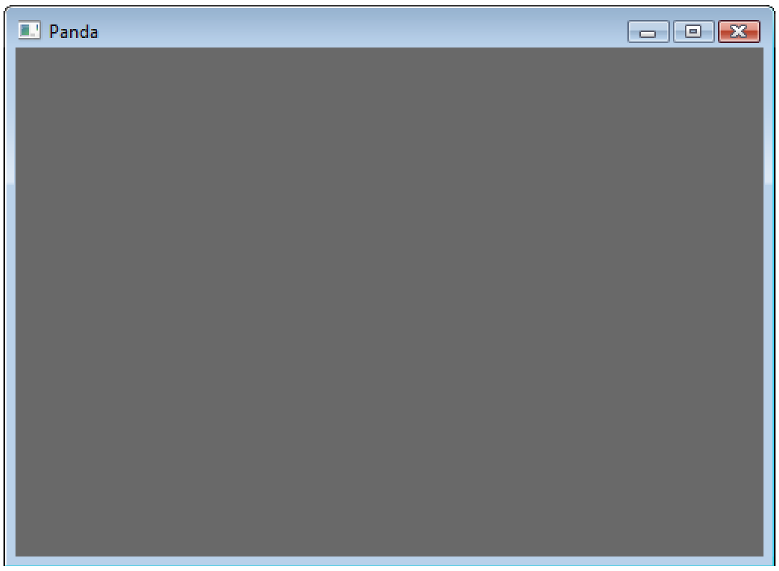
12. Click OK.

13. Delete every line on "pandasoccer.py", and write:

```
import direct.directbase.DirectStart

run()
```

14. Run the project.

You should see a blank window, in which case you now have a working Panda3D environment.



4. Starting the project

On the previous section, we've imported DirectStart in order to test if our Panda3D environment is working. DirectStart is basically a shortcut, that instantiates ShowBase automatically on import.

ShowBase is the actual class that loads most of Panda3D's core modules, and causes the 3D window to appear.

On this project, instead of using the shortcut, we'll have our main class Game inherit directly from ShowBase.

```
from direct.showbase.ShowBase import ShowBase

class Game(ShowBase):
        def __init__(self):
                ShowBase.__init__(self)

game = Game()
game.run()
```

1. Type the code above and run the project.

You should once again have a blank Panda3D window.

5. Adding the soccer field

The goal of this class is to teach how to program a game in Panda, not how to design the models necessary for the game.
To save precious time, the models have been prepared in advance, and they'll be provided for usage on the remainder of the tutorial.

1. Uncompress the models to the same directory as "pandasoccer.py".

Our Panda window is looking too empty, so first of all, we will add a soccer field to the game.

2. First we'll load the Field model, using the ShowBase method loader.loadModel()

```
    self.field = self.loader.loadModel("field")
```

Panda3D contains a data structure called the Scene Graph. The Scene Graph is a tree containing all objects that need to be rendered. At the root of the tree is an object named "render". Nothing is rendered until it is first inserted into the Scene Graph.

3. Add the model to the Scene Graph.

```
    self.field.reparentTo(render)
```

Even though the field is now being rendered, there's no guarantee it is visible, since we don't know if the camera is pointing at it.

4. Make the camera point at the field.

```
    base.cam.setPos(2300, 1050, 2300)
    base.cam.setHpr(115, -45, -20)
```

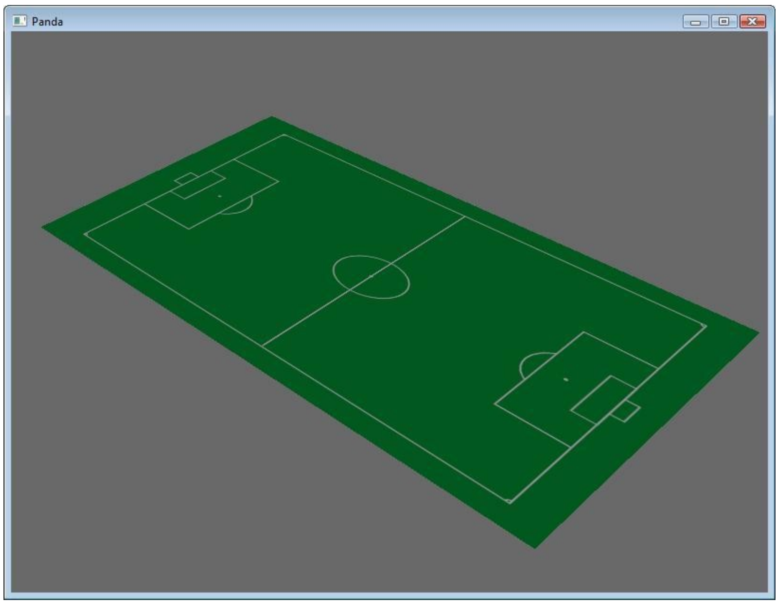
Your code should now look like this:

```
from direct.showbase.ShowBase import ShowBase

class Game(ShowBase):

  def __init__(self):
    ShowBase.__init__(self)

    # Load the field model.
    self.field = self.loader.loadModel("field")

    # Reparent the model to render, so the field becomes visible.
    self.field.reparentTo(render)

    # Position the camera so it focus on the field.
    base.cam.setPos(2300, 1050, 2300)
    base.cam.setHpr(115, -45, -20)
```

```
game = Game()
game.run()
```

5. Run the game.

If all went well, you should have a soccer field on your Panda window..



Since we're adding a lot more code to the game than just the field, let's separate contexts a bit, by giving Field it's own class.

6. Replace the following lines:

```
        # Load the field model.
        self.field = self.loader.loadModel("field")

        # Reparent the model to render, so the field becomes visible.
        self.field.reparentTo(render)
```

with

```
        self.field = Field(self)
```

7. Create the Field class

```
class Field():

        def __init__(self, game):

                # Load the field model.
                self.model = game.loader.loadModel("field")

                # Reparent the model to render, so the field becomes visible.
                self.model.reparentTo(render)
```

Your code should now look like this:

```
from direct.showbase.ShowBase import ShowBase

class Game(ShowBase):

  def __init__(self):
    ShowBase.__init__(self)

    self.field = Field(self)

    # Position the camera so it focus on the field.
    base.cam.setPos(2300, 1050, 2300)
    base.cam.setHpr(115, -45, -20)

class Field():

    def __init__(self, game):

        # Load the field model.
        self.model = game.loader.loadModel("field")

        # Reparent the model to render, so the field becomes visible.
        self.model.reparentTo(render)

game = Game()
game.run()
```

8. Run the project and confirm you still see the soccer field.

6. Adding a player

Now that we have a field, let's add a player we can control.

The player will have it's own class, inherited from the panda class Actor, which is a class designed to hold a model that can be animated, and it's set of animations.

1. So first of all, at the top of the file, import the class Actor.

```
from direct.actor.Actor import Actor
```

2. Create the Player class, inherited from Actor.

The Actor class constructor receives the path to the model, in this case "ralph" (the .egg extension is omitted), followed by the name and path of each animation.

```
class Player(Actor):

    def __init__(self):
        #Actor.__init__(self, <model path>, {<animation name>:<animation path>, ...})
        Actor.__init__(self, "ralph", {"run":"ralph-run", "walk":"ralph-walk"})

        self.reparentTo(render)
        self.setPos(0, 5, 0)
```

3. On the Game class, instantiate the player.

```
        self.player = Player()
```

The camera previously showed the soccer field from a distance, but now we want it to point at the soccer player.

4. So, on the Game class, delete the lines:

```
        base.cam.setPos(2300, 1050, 2300)
        base.cam.setHpr(115, -45, -20)
```

5. And on the Player class add the following lines:

```
        base.cam.setPos(self.getX(), self.getY() + 60, self.getZ() + 55)
        base.cam.lookAt(self)
```

The new player model is on a much smaller scale than the field model. In order to bring the models to the same scale, we have to reduce the scale of the field.

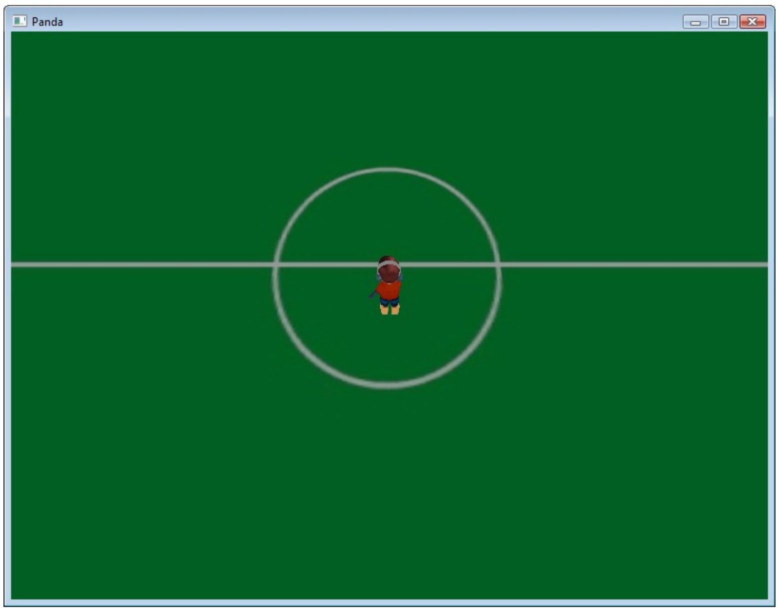
6. On the Field class, set the model scale to 0.1 on the X and Y axes, while keeping the same scale on the Z axis.

```
        self.model.setScale(.1, .1, 1)
```

Your code should now look like this:

```
from direct.actor.Actor import Actor
from direct.showbase.ShowBase import ShowBase

class Game(ShowBase):

    def __init__(self):
        ShowBase.__init__(self)

        self.field = Field(self)
        self.player = Player()

class Field():

    def __init__(self, game):

        self.model = game.loader.loadModel("field")
        self.model.reparentTo(render)
        self.model.setScale(.1, .1, 1)

class Player(Actor):

    def __init__(self):
        Actor.__init__(self, "ralph", {"run":"ralph-run", "walk":"ralph-walk"})

        self.reparentTo(render)
        self.setPos(0, 5, 0)

        base.cam.setPos(self.getX(), self.getY() + 60, self.getZ() + 55)
        base.cam.lookAt(self)

game = Game()
game.run()
```

7. Run the game, and confirm your window looks like this:



We now have a player, but we still can't control him. Let's add code to make him run around the field on command.

8. On the Player class, create methods that allow the player to move forward, turn left and right.

```
    def advance(self):
        self.setY(self, -20 * globalClock.getDt())

    def turnLeft(self):
        self.setH(self.getH() + 300 * globalClock.getDt())

    def turnRight(self):
        self.setH(self.getH() - 300 * globalClock.getDt())
```

The globalClock is an instance of the ClockObject class, which gets imported into the global namespace when the Showbase modules are loaded.
The method globalClock.getDt returns the time elapsed since the last frame was drawn. This way, by multiplying the latter by the speed of the player, we get a smooth and constant walking speed.

Now that the methods are created, we need to set up the communication between our keyboard and the player. For this purpose we will create a structure that maps the keys that are pressed at any given time.

9. On the Game class, create the following methods:

```
    def setKeys(self):
        self.keyMap = {"left":0, "right":0, "forward":0}

        self.accept("a", self.setKey, ["left",    1])
        self.accept("d", self.setKey, ["right",   1])
        self.accept("w", self.setKey, ["forward", 1])

        self.accept("a-up", self.setKey, ["left",    0])
        self.accept("d-up", self.setKey, ["right",   0])
        self.accept("w-up", self.setKey, ["forward", 0])

    def setKey(self, key, value):
        self.keyMap[key] = value
```

When a user presses a key, an event with the name of that key is launched. In order to enable a class to listen to these events, the "accept" function notifies panda that the second argument (in this case, "self.setKey"), is a handler to the event passed on the first argument ("a", "d", etc). The third argument is itself a list of arguments to be passed to the event's handler.

10. On the Game class constructor let's call the setKeys method to initiate the listeners, and disable mouse control, since we'll be controlling the player with the keyboard.

```
        base.disableMouse()

        self.setKeys()
```

Okay, so now we have a map of pressed keys on the Game class, and movement methods on the Player class. All that's missing is a link between them.

11. On the Player class, create a method that updates the player's position, depending on which keys are pressed.

```
    def update(self, keyMap):
        if keyMap["left"]:
            self.turnLeft()

        if keyMap["right"]:
            self.turnRight()

        if keyMap["forward"]:
            self.advance()
```

Since the player will be moving, we'll have to update the camera's position as well so we don't lose sight of him.

12. On the Player class, move the following lines, from the class constructor to the update method:

```
        base.cam.setPos(self.getX(), self.getY() + 60, self.getZ() + 55)
        base.cam.lookAt(self)
```

So all that's left now is to call the Player update method on the game's main cycle. Problem is, we don't have a main cycle yet. So let's create one.

13. On the Game class, create a cycle that updates the game over time.

```
    def updateScene(self, task):
        self.player.update(self.keyMap)

        return task.cont
```

Tasks are special functions that are called once each frame. They're similar in concept to threads, but instead of running separately, they run cooperatively, one at a time, within the main thread. A task can return either task.cont, which means the task should be called again next frame, or task.done, to indicate it should not be called again.

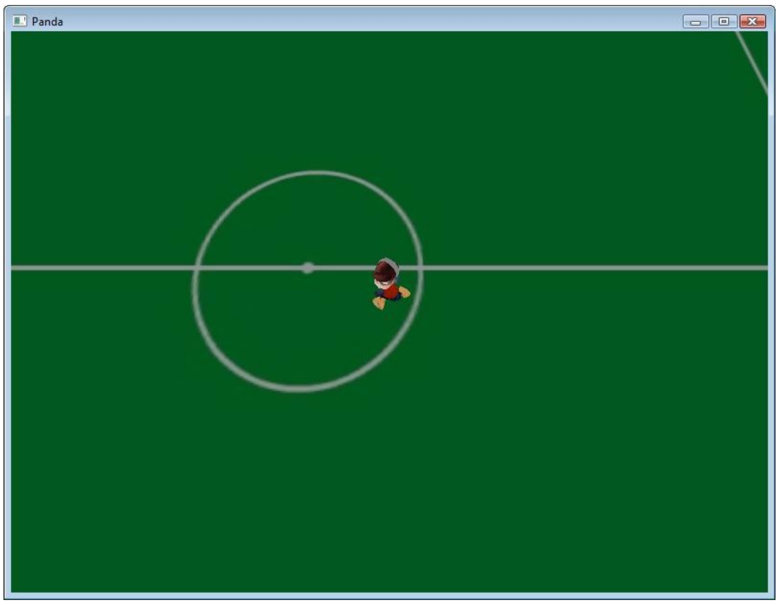
All tasks are handled through the global Task Manager object, called taskMgr. In order to add a new task function to the task list, the taskMgr.add() method is called, with the name of the task function, and an arbitrary name for the task.

14. On the Game class constructor, add the updateScene task to the Task Manager.

```
taskMgr.add(self.updateScene, "updateScene")
```

15. Run the game.

If everything went well, you should now be able to control the player.
Make him run around a bit.



You'll notice that while the player is moving around, he's not actually running, he's just floating over the floor without moving his legs or arms. While that is indeed a very impressive skill, we want a normal player running around by actually using his limbs.

When we created the player, we've already loaded his animations (ralph-run and ralph-walk), so now we just have to call them when the player is moving.

16. On the Player class constructor add a flag that indicates whether the player is moving.

```
self.isMoving = False
```

17. On the Player class, on the update method, play the correct animation depending on whether the player is moving.

```
if keyMap["forward"] or keyMap["left"] or keyMap["right"]:
   if self.isMoving is False:
      self.loop("run")
      self.isMoving = True
elif self.isMoving:
```

```
            self.stop()
            self.pose("walk", 5)
            self.isMoving = False
```

The player should now be able to run normally around the field.

At the end of this section, your code should look something like this:

```
from direct.actor.Actor import Actor
from direct.showbase.ShowBase import ShowBase

class Game(ShowBase):

    def __init__(self):
        ShowBase.__init__(self)

        self.field = Field(self)
        self.player = Player()

        base.disableMouse()

        self.setKeys()

        taskMgr.add(self.updateScene, "updateScene")

    def setKeys(self):
        self.keyMap = {"left":0, "right":0, "forward":0}

        self.accept("a", self.setKey, ["left",    1])
        self.accept("d", self.setKey, ["right",   1])
        self.accept("w", self.setKey, ["forward", 1])

        self.accept("a-up", self.setKey, ["left",    0])
        self.accept("d-up", self.setKey, ["right",   0])
        self.accept("w-up", self.setKey, ["forward", 0])

    def setKey(self, key, value):
        self.keyMap[key] = value

    def updateScene(self, task):
        self.player.update(self.keyMap)

        return task.cont

class Field():

    def __init__(self, game):

        self.model = game.loader.loadModel("field")
        self.model.reparentTo(render)
        self.model.setScale(.1, .1, 1)

class Player(Actor):

    def __init__(self):
        Actor.__init__(self, "ralph", {"run":"ralph-run", "walk":"ralph-walk"})

        self.reparentTo(render)
        self.setPos(0, 5, 0)

        self.isMoving = False

    def advance(self):
        self.setY(self, -20 * globalClock.getDt())

    def turnLeft(self):
        self.setH(self.getH() + 300 * globalClock.getDt())

    def turnRight(self):
        self.setH(self.getH() - 300 * globalClock.getDt())

    def update(self, keyMap):
        if keyMap["left"]:
            self.turnLeft()
        if keyMap["right"]:
            self.turnRight()
        if keyMap["forward"]:
            self.advance()

        base.cam.setPos(self.getX(), self.getY() + 60, self.getZ() + 55)
        base.cam.lookAt(self)

        if keyMap["forward"] or keyMap["left"] or keyMap["right"]:
            if self.isMoving is False:
                self.loop("run")
                self.isMoving = True
        elif self.isMoving:
            self.stop()
            self.pose("walk", 5)
            self.isMoving = False

game = Game()
game.run()
```

7. Adding the ball

Now we're getting somewhere. We have a field and a fully controllable player, now let's add a ball we can kick around.

So first of all, let's move the player a bit away from the center of the field, to clear some room for the ball.

1. On the Player class constructor, replace:

```
        self.setPos(0, 5, 0)
```

with:

```
        self.setPos(0, 10, 0)
```

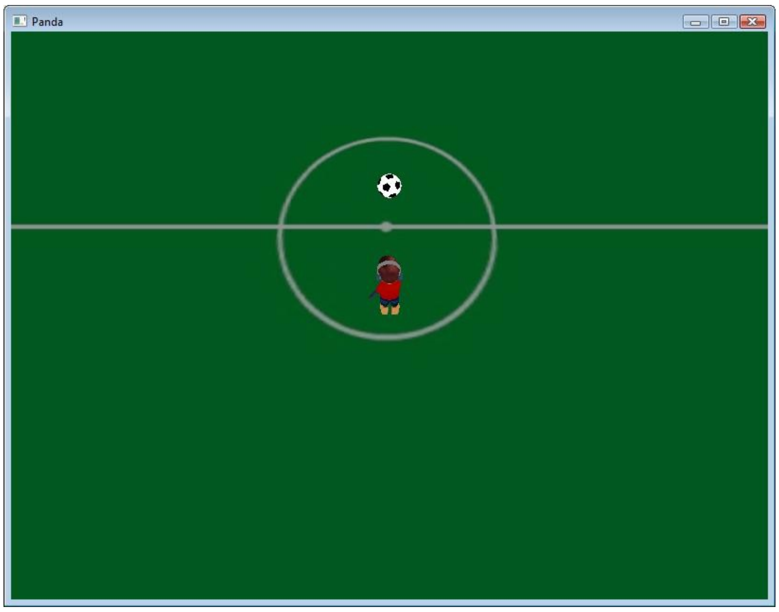
Now let's get the ball on the field.

2. First create the Ball class:

```
class Ball():

    def __init__(self, game):
        self.model = game.loader.loadModel("ball")

        self.model.reparentTo(render)
        self.model.setPos(0, 0, 5)
        self.model.setScale(.01)
```

3. On the Game class constructor, instantiate the ball:

```
        self.ball   = Ball(self)
```



If you run the game, you'll notice the ball just hovers in the air, without falling to the floor.

Without gravity or mass, the ball is just a stationary object. In order to make the ball move, we'll have to write a very basic physics implementation, so that we can apply forces to the ball.

8. Implementing a physics system

Panda3D provides integration for the Open Dynamics Engine (ODE), a platform-independent open-source physics engine, with advanced types and built-in collision detection.

To use the ODE physics system, we first need an OdeWorld. The world is an essential component in the physics structure, that holds all the bodies and controls global parameters, such as gravity, for the scene.

1. Create the class Physics, which inherits from OdeWorld.

```
class Physics(OdeWorld):

    def __init__(self):
        OdeWorld.__init__(self)
```

2. And on the Game class constructor, initialize an instance of Physics.

```
        self.physics = Physics()
```

ODE uses a lot of different classes that need to be imported. We could import each one individually, but for simplicity, let's import the whole package instead.

3. Import all the classes from PandaModules.

```
from pandac.PandaModules import *
```

Okay, so we have a physics world. Now let's configure it.

4. First of all let's add gravity to the scene. On the Physics class constructor add:

```
        self.setGravity(0, 0, -9.81)
```

To configure how materials react with each other, in terms of bouncing, friction and so on, we need to set up the surface system.

First we initialize the surface table, by using the method initSurfaceTable(<number of surfaces>). Afterwards we set up the parameters for collisions between two surfaces, using the setSurfaceEntry method, that receives such parameters as friction, bounce, dampen and others (consult the manual for more details).

5. On the Physics class constructor, configure the surface system.

```
self.initSurfaceTable(1)
self.setSurfaceEntry(0, 0, 150, 0.5, 3, 0.9, 0.00001, 0.0, 0.002)
```

The next step is configuring the space. An OdeSpace allows the OdeWorld to detect collisions between objects.
For the purpose of this tutorial we'll use an OdeSimpleSpace, which is preferred for worlds with a small amount of objects, as it does not perform collision culling.

6. On the Physics class constructor, create and configure an OdeSimpleSpace, and set it for automatic collisions.

```
self.space = OdeSimpleSpace()
self.space.setAutoCollideWorld(self)
self.contactGroup = OdeJointGroup()
self.space.setAutoCollideJointGroup(self.contactGroup)
```

So now we have a fully configured physics system, but we still need to get it running. Basically we want the physics system to position the ball correctly, on each given moment. To achieve this, we need a task that advances the physics simulation over time, and updates the position of the ball accordingly.

7. First, on the Physics class constructor, add the following lines:

```
self.deltaTimeAccumulator = 0.0
self.stepSize = 1.0 / 90.0
```

The deltaTimeAccumulator gathers the time spent each time the task is called, and once that time is higher than the stepSize, the physics simulation advances one step. By setting the stepSize to 1/90 we ensure the physics simulation runs at 90 frames per second.

8. Then, add the following methods to the Physics class:

```
def start(self, ball):
    self.ball = ball
    taskMgr.doMethodLater(.1, self.taskPhysics, "taskPhysics")

def taskPhysics(self, task):
    self.deltaTimeAccumulator += globalClock.getDt()

    while self.deltaTimeAccumulator > self.stepSize:
        self.deltaTimeAccumulator -= self.stepSize
        self.space.autoCollide()
        self.quickStep(self.stepSize)
```

```
            self.contactGroup.empty()

        self.ball.update()

        return task.cont
```

Basically, on each step of the task, we're telling the physics simulation to perform the automatic collisions, advancing the simulation one step, and then clearing the contactGroup, which is the structure that stores the collisions. Finally, we update the position of the ball.

9. Enabling physics on the ball

The physics are all set up, but the soccer ball still has no mass, and is still not a part of the physics system. So let's work on that.

In order to have a body affected by physics, we need to create an OdeBody, and give it an OdeMass. An OdeMass affects how much an object weighs and how it is divided over the body.

1. On the Ball class constructor add the following lines:

```
        self.body = OdeBody(game.physics)
        self.mass = OdeMass()
        self.mass.setSphere(260, .1)
        self.body.setMass(self.mass)
        self.body.setPosition(self.model.getPos(render))
        self.body.setQuaternion(self.model.getQuat(render))

        self.geom = OdeSphereGeom(game.physics.space, .8)
        self.geom.setBody(self.body)
```

On each step of the physics simulation, it is also necessary to update the position of the ball.

2. So let's add the following method to the ball:

```
    def update(self):
        self.model.setPosQuat(render, self.body.getPosition(), Quat(self.body.getQuaternion()))
```

Almost there, the ball is now ready to react to gravity, but the field is still not aware of the physics system, so the ball would just fall through the floor if the simulation were to start now.

3. On the Field class constructor, create a body for the field.

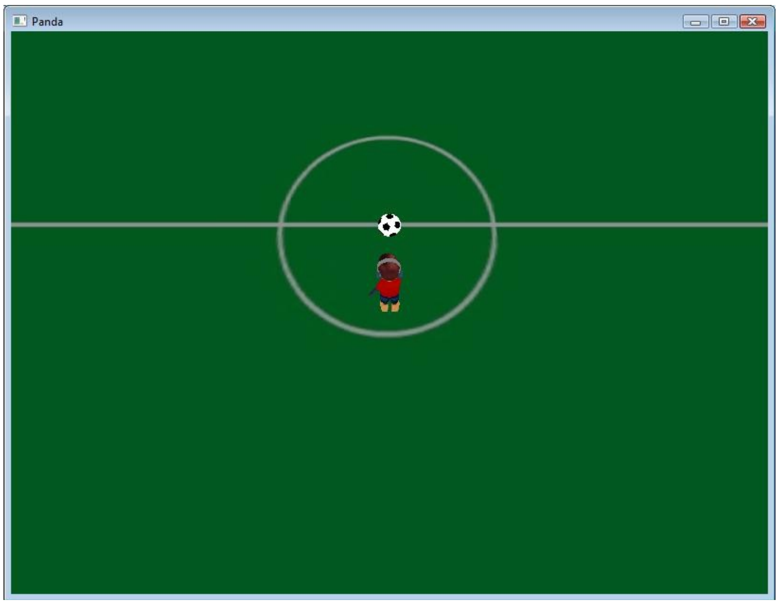
```
        self.geom = OdePlaneGeom(game.physics.space, Vec4(0, 0, 1, 0))
```

All that's left now is telling the physics simulation to start.

4. On the Game class constructor, on the last line, add:

```
        self.physics.start(self.ball)
```

5. Run the game. The ball should fall to the floor, bounce for a bit and stop.



Your code should now look like this:

```
from direct.actor.Actor             import Actor
from direct.showbase.ShowBase import ShowBase
from pandac.PandaModules          import *

class Game(ShowBase):

    def __init__(self):
        ShowBase.__init__(self)

        self.physics = Physics()

        self.field  = Field(self)
        self.ball   = Ball(self)
        self.player = Player()
        base.disableMouse()

        self.setKeys()

        taskMgr.add(self.updateScene, "updateScene")
        self.physics.start(self.ball)

    def setKeys(self):
        self.keyMap = {"left":0, "right":0, "forward":0}

        self.accept("a", self.setKey, ["left",    1])
        self.accept("d", self.setKey, ["right",   1])
        self.accept("w", self.setKey, ["forward", 1])

        self.accept("a-up", self.setKey, ["left",    0])
        self.accept("d-up", self.setKey, ["right",   0])
        self.accept("w-up", self.setKey, ["forward", 0])

    def setKey(self, key, value):
        self.keyMap[key] = value

    def updateScene(self, task):
        self.player.update(self.keyMap)

        return task.cont

class Ball():

    def __init__(self, game):
        self.model = game.loader.loadModel("ball")

        self.model.reparentTo(render)
        self.model.setPos(0, 0, 5)
        self.model.setScale(.01)

        self.body = OdeBody(game.physics)
        self.mass = OdeMass()
        self.mass.setSphere(260, .1)
        self.body.setMass(self.mass)
        self.body.setPosition(self.model.getPos(render))
        self.body.setQuaternion(self.model.getQuat(render))

        self.geom = OdeSphereGeom(game.physics.space, .8)
        self.geom.setBody(self.body)

    def update(self):
        self.model.setPosQuat(render, self.body.getPosition(), Quat(self.body.getQuaternion()))

class Field():
    def __init__(self, game):

        self.model = game.loader.loadModel("field")
        self.model.reparentTo(render)
        self.model.setScale(.1, .1, 1)

        self.geom = OdePlaneGeom(game.physics.space, Vec4(0, 0, 1, 0))

class Player(Actor):

    def __init__(self):
        Actor.__init__(self, "ralph", {"run":"ralph-run", "walk":"ralph-walk"})

        self.reparentTo(render)
        self.setPos(0, 10, 0)

        self.isMoving = False

    def advance(self):
        self.setY(self, -20 * globalClock.getDt())

    def turnLeft(self):
        self.setH(self.getH() + 300 * globalClock.getDt())

    def turnRight(self):
        self.setH(self.getH() - 300 * globalClock.getDt())

    def update(self, keyMap):
        if keyMap["left"]:
            self.turnLeft()
        if keyMap["right"]:
            self.turnRight()
        if keyMap["forward"]:
            self.advance()

        base.cam.setPos(self.getX(), self.getY() + 60, self.getZ() + 55)
        base.cam.lookAt(self)

        if keyMap["forward"] or keyMap["left"] or keyMap["right"]:
            if self.isMoving is False:
                self.loop("run")
                self.isMoving = True
        elif self.isMoving:
            self.stop()
            self.pose("walk", 5)
            self.isMoving = False

class Physics(OdeWorld):

    def __init__(self):
        OdeWorld.__init__(self)

        self.setGravity(0, 0, -9.81)

        self.initSurfaceTable(1)
        self.setSurfaceEntry(0, 0, 150, 0.5, 3, 0.9, 0.00001, 0.0, 0.002)

        self.space = OdeSimpleSpace()
        self.space.setAutoCollideWorld(self)
        self.contactGroup = OdeJointGroup()
        self.space.setAutoCollideJointGroup(self.contactGroup)

        self.deltaTimeAccumulator = 0.0
        self.stepSize = 1.0 / 90.0

    def start(self, ball):
        self.ball = ball
        taskMgr.doMethodLater(.1, self.taskPhysics, "taskPhysics")

    def taskPhysics(self, task):
        self.deltaTimeAccumulator += globalClock.getDt()

        while self.deltaTimeAccumulator > self.stepSize:
            self.deltaTimeAccumulator -= self.stepSize
            self.space.autoCollide()
            self.quickStep(self.stepSize)
            self.contactGroup.empty()

        self.ball.update()

        return task.cont

game = Game()
game.run()
```

10. Adding the kick ability

Okay, so we have a field, a fully controllable player, and a ball that is affected by physics. Unfortunately, the player still can't kick the ball, so as it is, the ball doesn't really move much.

So let's work on that.

Automatic physics collisions are just fine for handling the ball's collisions with the field, but now we want to detect a collision between the player and the ball, and instead of performing an automatic physical collision, we want to handle the collision ourselves, so that we can kick the ball whenever we want, if we're in range of the ball.

In Panda3D, collision detection requires the creation of special collision geometry, such as spheres and polygons, to determine whether there are collisions. The CollisionSolid is the fundamental object of the collision system, and among these solids, the CollisionSphere is the fastest primitive for any collision calculation.

1. So, first, on the Ball class constructor, create a CollisionSphere.

```
self.colSphere = CollisionSphere(0, 0, 0, 100)
self.colNode = self.model.attachNewNode(CollisionNode('ballNode'))
self.colNode.node().addSolid(self.colSphere)
self.colNode.show()
```

2. And on the Player class constructor, create another one for the player.

```
self.colSphere = CollisionSphere(0, 0, 2.5, 3)
self.colNode = self.attachNewNode(CollisionNode('playerNode'))
self.colNode.node().addSolid(self.colSphere)
self.colNode.show()
```

We have the collision solids, now we have to detect the collisions between them.

3. On the Game class constructor add the following lines:

```
self.queue = CollisionHandlerQueue()
self.traverser = CollisionTraverser()
self.traverser.addCollider(self.player.colNode, self.queue)
self.traverser.addCollider(self.ball.colNode, self.queue)
self.traverser.showCollisions(render)
```

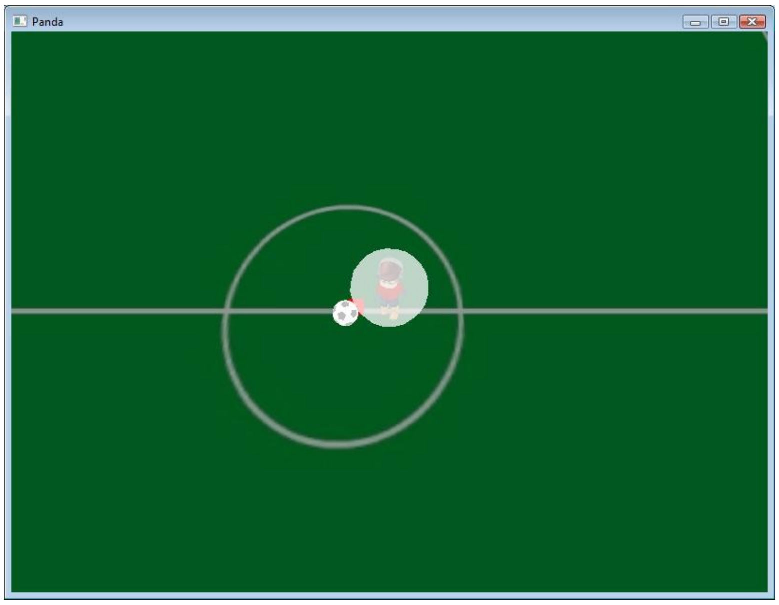
The CollisionTraverser performs the actual work of checking all solid objects for collisions, and the CollisionHandlerQueue records the collisions that were detected during the most recent traversal.

4. Finally, on the updateScene method of the Game class, add:

```
self.traverser.traverse(render)
```

5. Run the game.

You should see the collision spheres around the player and the ball. When the spheres collide, the collision point will also turn red, to show you where the collision is being detected.



So the collisions between the player and the ball are now being detected. Now the behavior we want to achieve is that when the player's sphere is colliding with the ball's sphere, the player can press a key to kick the ball in the direction the player is facing.

First of all, we need a flag that tells us when the player is actually in range of the ball.

6. On the Player class constructor, add the following line:

```
        self.inRange = False
```

Let's also add some methods to control the flag.

7. On the Player class, add the following methods:

```
    def isInRange(self):
        return self.inRange

    def setInRange(self, inRange):
        self.inRange = inRange
```

We also want the player to be able to kick the ball when it's in range, so let's create a method for that as well.

8. On the Player class, add the following method:

```
    def kick(self):
        if self.isInRange():
                        force = 2000
                        angle = self.getH(render) - 90
                        vector = Vec3(math.cos(math.radians(angle)),
math.sin(math.radians(angle)), .5)
                        self.ball.body.setForce(vector.getX() * force, vector.getY() * force,
vector.getZ() * force)
```

On this method, we're basically checking if the player is in range, and if so, we calculate what angle corresponds to the direction the player is facing, extract a vector from that angle, and finally shooting the ball according to that vector.

This method uses two things we don't have: the math package, and a ball reference. So let's fix that.

9. At the top of the file, import the math package. While we're at it, let's import the sys package as well, we will need it later on.

```
import math, sys
```

10. On the Game class constructor, replace:

```
        self.player = Player()
```

with:

```
        self.player = Player(self.ball)
```

11. And on the Player class constructor, replace:

```
    def __init__(self):
        Actor.__init__(self, "ralph", {"run":"ralph-run", "walk":"ralph-walk"})
```

with:

```
    def __init__(self, ball):
        Actor.__init__(self, "ralph", {"run":"ralph-run", "walk":"ralph-walk"})

        self.ball = ball
```

Now, let's set the key for kicking, and while we're at it, add also the escape key as a key to close the game.

12. On the setKeys method of the Game class, add the following lines:

```
        self.accept("space", self.player.kick)
        self.accept("escape", sys.exit)
```

Finally, all that's left is updating the inRange flag each time the scene is updated, so the player always knows if he's in range of the ball.

13. On the updateScene method of the Game class, replace:

```
    def updateScene(self, task):
        self.player.update(self.keyMap)

        return task.cont
```

with:

```
    def updateScene(self, task):
        self.player.update(self.keyMap)

        inRange = False
        self.traverser.traverse(render)
        for i in range(self.queue.getNumEntries()):
            entry = self.queue.getEntry(i)
            fromName = entry.getFromNodePath().getName()
            intoName = entry.getIntoNodePath().getName()

            if fromName == "playerNode":
                if intoName == "ballNode":
                    inRange = True

        if self.player.isInRange() != inRange:
            self.player.setInRange(inRange)

        return task.cont
```

14. Run the game and try to kick the ball.

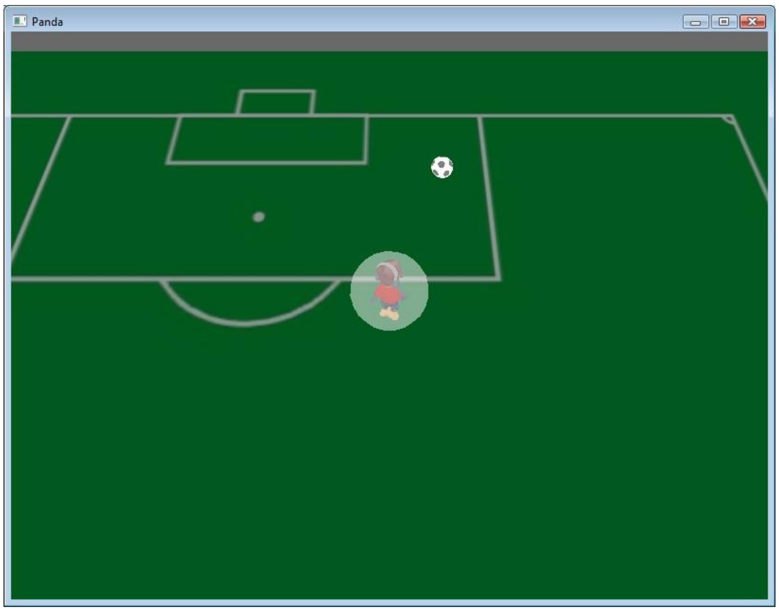
If all went well, you should be able to kick the ball, and see it flying away. Stopping the ball however, is a different matter entirely. As it is, the ball just keeps going infinitely. So, since friction isn't doing it for us, let's stop the ball manually.

15. On the update method of the Ball class, add the following lines:

```
        vel = self.body.getLinearVel()
        if vel[2] < 0.1 and self.body.getPosition().getZ() < 0.9:
            if abs(vel[0]) > 0:
                vel[0] *= .9
            if abs(vel[1]) > 0:
                vel[1] *= .9

        self.body.setLinearVel(vel)
```

16. Run the game again, and kick the ball.

This time, the ball should stop properly.



At the end of this section, your code should look like this:

```
from direct.actor.Actor             import Actor
from direct.showbase.ShowBase import ShowBase
from pandac.PandaModules         import *
import math, sys

class Game(ShowBase):

    def __init__(self):
        ShowBase.__init__(self)

        self.physics = Physics()

        self.field  = Field(self)
        self.ball   = Ball(self)
        self.player = Player(self.ball)

        base.disableMouse()

        self.queue = CollisionHandlerQueue()
        self.traverser = CollisionTraverser()
        self.traverser.addCollider(self.player.colNode, self.queue)
        self.traverser.addCollider(self.ball.colNode, self.queue)
        self.traverser.showCollisions(render)

        self.setKeys()

        taskMgr.add(self.updateScene, "updateScene")
        self.physics.start(self.ball)

    def setKeys(self):
        self.keyMap = {"left":0, "right":0, "forward":0}

        self.accept("a", self.setKey, ["left", 1])
        self.accept("d", self.setKey, ["right", 1])
        self.accept("w", self.setKey, ["forward", 1])

        self.accept("a-up", self.setKey, ["left", 0])
        self.accept("d-up", self.setKey, ["right", 0])
        self.accept("w-up", self.setKey, ["forward", 0])

        self.accept("space", self.player.kick)
        self.accept("escape", sys.exit)

    def setKey(self, key, value):
        self.keyMap[key] = value

    def updateScene(self, task):
        self.player.update(self.keyMap)

        inRange = False

        self.traverser.traverse(render)

        for i in range(self.queue.getNumEntries()):
            entry = self.queue.getEntry(i)
            fromName = entry.getFromNodePath().getName()
            intoName = entry.getIntoNodePath().getName()

            if fromName == "playerNode":
                if intoName == "ballNode":
                    inRange = True

        if self.player.isInRange() != inRange:
            self.player.setInRange(inRange)

        return task.cont

class Ball():

    def __init__(self, game):
        self.model = game.loader.loadModel("ball")

        self.model.reparentTo(render)
        self.model.setPos(0, 0, 5)
        self.model.setScale(.01)

        self.colSphere = CollisionSphere(0, 0, 0, 100)
        self.colNode = self.model.attachNewNode(CollisionNode('ballNode'))
        self.colNode.node().addSolid(self.colSphere)
        self.colNode.show()

        self.body = OdeBody(game.physics)
        self.mass = OdeMass()
        self.mass.setSphere(260, .1)
        self.body.setMass(self.mass)
        self.body.setPosition(self.model.getPos(render))
        self.body.setQuaternion(self.model.getQuat(render))

        self.geom = OdeSphereGeom(game.physics.space, .8)
        self.geom.setBody(self.body)

    def update(self):
        self.model.setPosQuat(render, self.body.getPosition(),
Quat(self.body.getQuaternion()))

        vel = self.body.getLinearVel()
        if vel[2] < 0.1 and self.body.getPosition().getZ() < 0.9:
            if abs(vel[0]) > 0:
                vel[0] *= .9

            if abs(vel[1]) > 0:
                vel[1] *= .9

        self.body.setLinearVel(vel)

class Field():

    def __init__(self, game):

        self.model = game.loader.loadModel("field")
        self.model.reparentTo(render)
        self.model.setScale(.1, .1, 1)

        self.geom = OdePlaneGeom(game.physics.space, Vec4(0, 0, 1, 0))

class Player(Actor):

    def __init__(self, ball):
        Actor.__init__(self, "ralph", {"run":"ralph-run", "walk":"ralph-walk"})
        self.ball = ball

        self.reparentTo(render)
        self.setPos(0, 10, 0)

        self.inRange = False
        self.isMoving = False

        self.colSphere = CollisionSphere(0, 0, 2.5, 3)
        self.colNode = self.attachNewNode(CollisionNode('playerNode'))
        self.colNode.node().addSolid(self.colSphere)
        self.colNode.show()

    def advance(self):
        self.setY(self, -20 * globalClock.getDt())

    def isInRange(self):
        return self.inRange

    def kick(self):
        if self.isInRange():
            force = 2000
            angle = self.getH(render) - 90
            vector = Vec3(math.cos(math.radians(angle)), math.sin(math.radians(angle)), .5)
            self.ball.body.setForce(vector.getX() * force, vector.getY() * force, vector.getZ() *
force)
            self.ball.body.setAngularVel(vector.getX() * 10, vector.getY() * 10, 0)

    def setInRange(self, inRange):
        self.inRange = inRange

    def turnLeft(self):
        self.setH(self.getH() + 300 * globalClock.getDt())

    def turnRight(self):
        self.setH(self.getH() - 300 * globalClock.getDt())

    def update(self, keyMap):
        if keyMap["left"]:
            self.turnLeft()
        if keyMap["right"]:
            self.turnRight()
        if keyMap["forward"]:
            self.advance()

        base.cam.setPos(self.getX(), self.getY() + 60, self.getZ() + 55)
        base.cam.lookAt(self)

        if keyMap["forward"] or keyMap["left"] or keyMap["right"]:
            if self.isMoving is False:
```

```
            self.loop("run")
            self.isMoving = True
        elif self.isMoving:
            self.stop()
            self.pose("walk", 5)
            self.isMoving = False

class Physics(OdeWorld):

    def __init__(self):
        OdeWorld.__init__(self)

        self.setGravity(0, 0, -9.81)

        self.initSurfaceTable(1)
        self.setSurfaceEntry(0, 0, 150, 0.5, 3, 0.9, 0.00001, 0.0, 0.002)

        self.space = OdeSimpleSpace()
        self.space.setAutoCollideWorld(self)
        self.contactGroup = OdeJointGroup()
        self.space.setAutoCollideJointGroup(self.contactGroup)

        self.deltaTimeAccumulator = 0.0
        self.stepSize = 1.0 / 90.0

    def start(self, ball):
        self.ball = ball
        taskMgr.doMethodLater(.1, self.taskPhysics, "taskPhysics")

    def taskPhysics(self, task):
        self.deltaTimeAccumulator += globalClock.getDt()

        while self.deltaTimeAccumulator > self.stepSize:
            self.deltaTimeAccumulator -= self.stepSize
            self.space.autoCollide()
            self.quickStep(self.stepSize)
            self.contactGroup.empty()

        self.ball.update()

        return task.cont

game = Game()
game.run()
```

11. Structuring the code

The code is getting pretty long by now, so before we move on, let's structure it into smaller pieces.

1. First, on the project source directory, create a "models" directory and move all models and textures to it.

Now we have to update all the references to models and just add "models/" to the path

2. On the Ball class constructor, replace:

```
        self.model = game.loader.loadModel("ball")
```

with:

```
        self.model = game.loader.loadModel("models/ball")
```

3. On the Field class constructor, replace:

```
        self.model = game.loader.loadModel("field")
```

with:

```
        self.model = game.loader.loadModel("models/field")
```

4. On the Player class constructor, replace:

```
        Actor.__init__(self, "ralph", {"run":"ralph-run", "walk":"ralph-walk"})
```

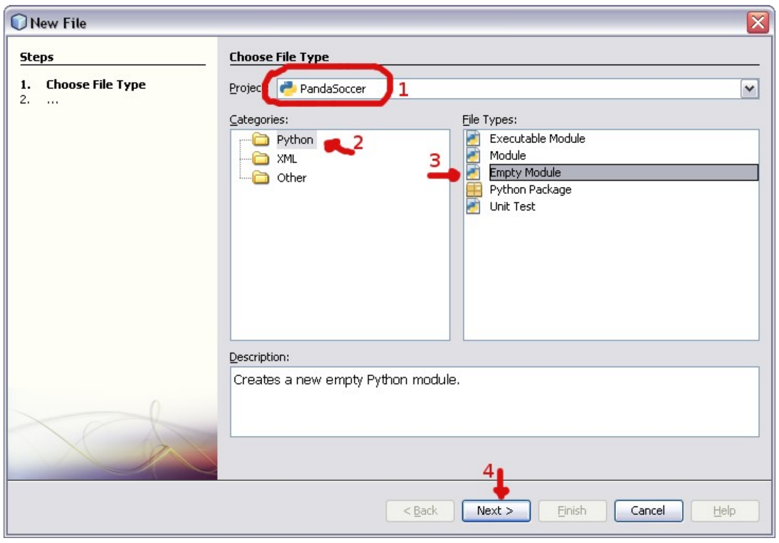
with:

```
        Actor.__init__(self, "models/ralph", {"run":"models/ralph-run", "walk":"models/ralph-
walk"})
```

Now let's separate each class into it's own file. Let's start with Physics.

5. Click File->New File

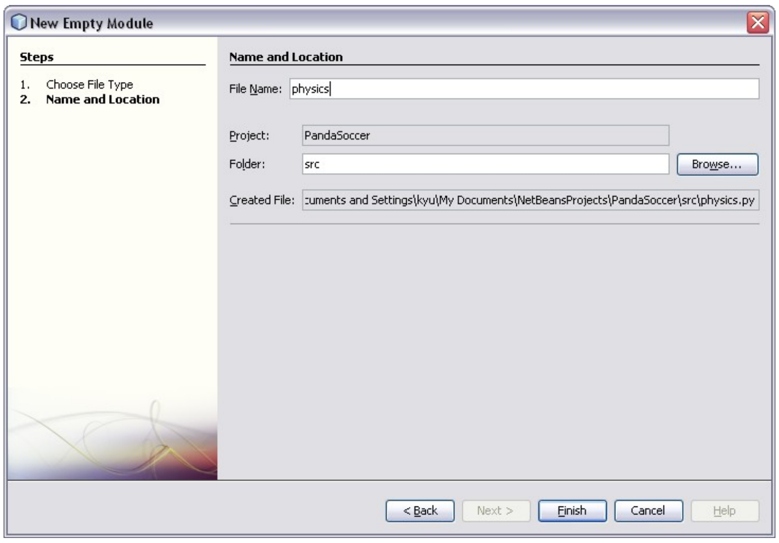
6. On the Projects drop-down list, select "PandaSoccer", on Categories select "Python" and on File Types select "Empty Module". Finally click "Next".



Description:

Creates a new empty Python module.

7. Name the file "physics" and press Finish.



File Name: physics

Project: PandaSoccer

Folder: src

Created File: :uments and Settings\kyu\My Documents\NetBeansProjects\PandaSoccer\src\physics.py

8. On pandasoccer.py, cut all of the Physics class code, and paste it on the physics.py file.

9. On pandasoccer.py, import the Physics class.

```
from physics import Physics
```

10. Finally, on physics.py, import the PandaModules.

```
from pandac.PandaModules import *
```

11. Run the application and check if everything's still running fine.

12. Repeat steps 5 through 11 for the Field and Ball classes.

13. Repeat steps 5 through 10 for the Player class.

14. Move the following line, from pandasoccer.py to player.py:

```
from direct.actor.Actor  import Actor
```

15. On pandasoccer.py, remove the math import, by replacing:

```
import math, sys
```

with:

```
import sys
```

16. And finally, on player.py, import the math module

```
import math
```

17. Run the game.

If all went well, the game should run as smoothly as before, but now the code is separated into different files and easier to tackle.

You should now have 5 different files: ball.py, field.py, pandasoccer.py, physics.py and player.py, and your code should look like this:

pandasoccer.py:

```

from direct.showbase.ShowBase import ShowBase
from ball                     import Ball
from field                    import Field
from pandac.PandaModules      import *
from physics                  import Physics
from player                   import Player
import sys

class Game(ShowBase):

   def __init__(self):
      ShowBase.__init__(self)
```

```
        self.physics = Physics()

        self.field  = Field(self)
        self.ball   = Ball(self)
        self.player = Player(self.ball)

        base.disableMouse()

        self.queue = CollisionHandlerQueue()
        self.traverser = CollisionTraverser()
        self.traverser.addCollider(self.player.colNode, self.queue)
        self.traverser.addCollider(self.ball.colNode, self.queue)
        self.traverser.showCollisions(render)

        self.setKeys()

        taskMgr.add(self.updateScene, "updateScene")
        self.physics.start(self.ball)

    def setKeys(self):
        self.keyMap = {"left":0, "right":0, "forward":0}

        self.accept("a", self.setKey, ["left", 1])
        self.accept("d", self.setKey, ["right", 1])
        self.accept("w", self.setKey, ["forward", 1])

        self.accept("a-up", self.setKey, ["left", 0])
        self.accept("d-up", self.setKey, ["right", 0])
        self.accept("w-up", self.setKey, ["forward", 0])

        self.accept("space", self.player.kick)
        self.accept("escape", sys.exit)

    def setKey(self, key, value):
        self.keyMap[key] = value

    def updateScene(self, task):
        self.player.update(self.keyMap)

        inRange = False

        self.traverser.traverse(render)

        for i in range(self.queue.getNumEntries()):
            entry = self.queue.getEntry(i)
            fromName = entry.getFromNodePath().getName()
            intoName = entry.getIntoNodePath().getName()

            if fromName == "playerNode":
```

```
            if intoName == "ballNode":
                inRange = True

        if self.player.isInRange() != inRange:
            self.player.setInRange(inRange)

        return task.cont

game = Game()
game.run()
```

ball.py:

```

from pandac.PandaModules import *

class Ball():

    def __init__(self, game):
        self.model = game.loader.loadModel("models/ball")

        self.model.reparentTo(render)
        self.model.setPos(0, 0, 5)
        self.model.setScale(.01)

        self.colSphere = CollisionSphere(0, 0, 0, 100)
        self.colNode = self.model.attachNewNode(CollisionNode('ballNode'))
        self.colNode.node().addSolid(self.colSphere)
        self.colNode.show()

        self.body = OdeBody(game.physics)
        self.mass = OdeMass()
        self.mass.setSphere(260, .1)
        self.body.setMass(self.mass)
        self.body.setPosition(self.model.getPos(render))
        self.body.setQuaternion(self.model.getQuat(render))

        self.geom = OdeSphereGeom(game.physics.space, .8)
        self.geom.setBody(self.body)

    def update(self):
        self.model.setPosQuat(render, self.body.getPosition(),
Quat(self.body.getQuaternion()))

        vel = self.body.getLinearVel()
        if vel[2] < 0.1 and self.body.getPosition().getZ() < 0.9:
            if abs(vel[0]) > 0:
                vel[0] *= .9

            if abs(vel[1]) > 0:
                vel[1] *= .9

        self.body.setLinearVel(vel)
```

field.py:

```
from pandac.PandaModules import *

class Field():

    def __init__(self, game):

        self.model = game.loader.loadModel("models/field")
        self.model.reparentTo(render)
        self.model.setScale(.1, .1, 1)

        self.geom = OdePlaneGeom(game.physics.space, Vec4(0, 0, 1, 0))
```

physics.py:

```
from pandac.PandaModules import *

class Physics(OdeWorld):

    def __init__(self):
        OdeWorld.__init__(self)

        self.setGravity(0, 0, -9.81)

        self.initSurfaceTable(1)
        self.setSurfaceEntry(0, 0, 150, 0.5, 3, 0.9, 0.00001, 0.0, 0.002)

        self.space = OdeSimpleSpace()
        self.space.setAutoCollideWorld(self)
        self.contactGroup = OdeJointGroup()
        self.space.setAutoCollideJointGroup(self.contactGroup)

        self.deltaTimeAccumulator = 0.0
        self.stepSize = 1.0 / 90.0

    def start(self, ball):
        self.ball = ball
        taskMgr.doMethodLater(.1, self.taskPhysics, "taskPhysics")
    def taskPhysics(self, task):
        self.deltaTimeAccumulator += globalClock.getDt()

        while self.deltaTimeAccumulator > self.stepSize:
            self.deltaTimeAccumulator -= self.stepSize
            self.space.autoCollide()
            self.quickStep(self.stepSize)
            self.contactGroup.empty()

        self.ball.update()

        return task.cont
```

player.py:

```
from direct.actor.Actor  import Actor
from pandac.PandaModules import *
import math

class Player(Actor):

    def __init__(self, ball):
        Actor.__init__(self, "models/ralph", {"run":"models/ralph-run", "walk":"models/ralph-
walk"})

        self.ball = ball

        self.reparentTo(render)
        self.setPos(0, 10, 0)

        self.inRange = False
        self.isMoving = False

        self.colSphere = CollisionSphere(0, 0, 2.5, 3)
        self.colNode = self.attachNewNode(CollisionNode('playerNode'))
        self.colNode.node().addSolid(self.colSphere)
        self.colNode.show()

    def advance(self):
        self.setY(self, -20 * globalClock.getDt())

    def isInRange(self):
        return self.inRange

    def kick(self):
        if self.isInRange():
            force = 2000
            angle = self.getH(render) - 90
        vector = Vec3(math.cos(math.radians(angle)), math.sin(math.radians(angle)), .5)
        self.ball.body.setForce(vector.getX() * force, vector.getY() * force, vector.getZ() *
force)
        self.ball.body.setAngularVel(vector.getX() * 10, vector.getY() * 10, 0)

  def setInRange(self, inRange):
    self.inRange = inRange

  def turnLeft(self):
    self.setH(self.getH() + 300 * globalClock.getDt())

  def turnRight(self):
    self.setH(self.getH() - 300 * globalClock.getDt())

  def update(self, keyMap):
    if keyMap["left"]:
      self.turnLeft()
    if keyMap["right"]:
      self.turnRight()
    if keyMap["forward"]:
      self.advance()

    base.cam.setPos(self.getX(), self.getY() + 60, self.getZ() + 55)
    base.cam.lookAt(self)

    if keyMap["forward"] or keyMap["left"] or keyMap["right"]:
      if self.isMoving is False:
        self.loop("run")
        self.isMoving = True
    elif self.isMoving:
      self.stop()
      self.pose("walk", 5)
      self.isMoving = False
```

12. Boundaries

Alright, let's get back to developing. As it is, the ball and the player are free to roam around outside of the field. To prevent both of them from leaving the playfield, let's add some boundaries.

For the boundary body, we will use a CollisionPlane, which is basically an infinite plane extending in all directions. The plane actually divides the universe into two spaces: the space behind the plane, which is all considered solid, and the space in front of the plane, which is all empty. Thus, if an object is anywhere behind a plane, no matter how far, it is considered to be intersecting the plane.

1. First, on the Field class, add the following method:

```
  def addBoundary(self, name, orientation, position, posOffset, velMultiplier):
    boundary = CollisionPlane(Plane(orientation, position))
    boundary.setTangible(True)
    self.boundaries[name] = [boundary, posOffset, velMultiplier]
    colNode = self.model.attachNewNode(CollisionNode(name))
    colNode.node().addSolid(boundary)
```

So basically, the addBoundary method receives a name for the boundary, an orientation and position, a position offset that indicates where the ball should be moved to if it collides with the boundary, and a velocity multiplier that indicates how the velocity of the ball should be affected after the collision.

2. Next, on the Field class constructor, let's create the four boundaries around the field:

```
    self.boundaries = {}
    self.addBoundary("boundaryRight",  Vec3(1, 0, 0),  Point3(-500, 0, 0),  (1,  0), (-1, 1))
    self.addBoundary("boundaryLeft",   Vec3(-1, 0, 0), Point3(500, 0, 0),   (-1, 0), (-1, 1))
    self.addBoundary("boundaryBottom", Vec3(0, -1, 0), Point3(0, 1000, 0),  (0, -1), (1,
-1))
    self.addBoundary("boundaryTop",    Vec3(0, 1, 0),  Point3(0, -1000, 0), (0,  1), (1,
-1))
```

So we've got four boundaries, enclosing the field. But what happens when a ball collides against them? Let's add that behavior.

3. On the Field class, add the following method:

```
  def collide(self, ball, boundaryName):
    vel = ball.body.getLinearVel()
    pos = ball.body.getPosition()

    b = self.boundaries[boundaryName]
    ball.body.setPosition(pos.getX() + b[1][0], pos.getY() + b[1][1], pos.getZ())
    ball.body.setLinearVel(VBase3(vel[0] * b[2][0], vel[1] * b[2][1], vel[2]))
```

4. And on the Game class, on the updateScene method, replace the following lines:

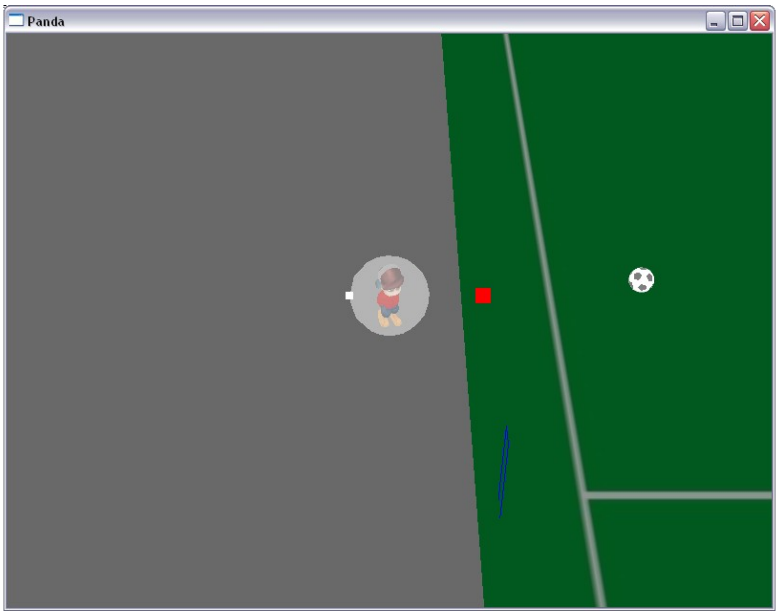
```
      if fromName == "playerNode":
        if intoName == "ballNode":
          inRange = True
```

with:

```
        if fromName == "playerNode":
            if intoName == "ballNode":
                inRange = True

        elif fromName == "ballNode":
            if intoName == "playerNode":
                pass
            else: #it's a boundary
                self.field.collide(self.ball, intoName)
```

So basically what we're doing here is, when a ball collides with anything that's not a player, it has to be a boundary, so we're calling the field's collide method, which in turn, pushes the ball back, preventing it from leaving the field.

5. Run the game and try to kick the ball out of the field.



The ball should now be unable to leave the field. The player however, is a different story. Let's add code to prevent the player from leaving the field as well.

6. On the Player class, add the following method:

```
    def retrocede(self):
        self.setY(self, 20 * globalClock.getDt())
```

7. On the Game class, on the updateScene method, replace the following lines:

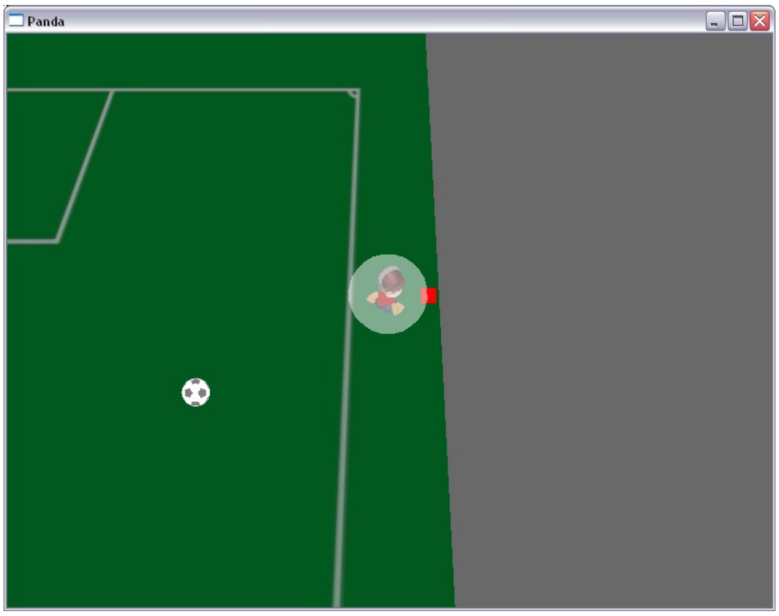
```
        if fromName == "playerNode":
            if intoName == "ballNode":
                inRange = True
```

with:

```
        if fromName == "playerNode":
            if intoName == "ballNode":
                inRange = True
            else: #it's a boundary
                self.player.retrocede()
```

So, just like with the ball, what we're doing here is pushing back the player when he collides with a boundary.

8. Run the game, and try leaving the field. Both the player and the ball should now be unable to do so.



At the end of this section your code should look like this:

pandasoccer.py:

```
from direct.showbase.ShowBase import ShowBase
from ball                     import Ball
from field                    import Field
from pandac.PandaModules      import *
from physics                  import Physics
from player                   import Player
import sys

class Game(ShowBase):

    def __init__(self):
        ShowBase.__init__(self)

        self.physics = Physics()
        self.field  = Field(self)
        self.ball   = Ball(self)
        self.player = Player(self.ball)

        base.disableMouse()

        self.queue = CollisionHandlerQueue()
        self.traverser = CollisionTraverser()
        self.traverser.addCollider(self.player.colNode, self.queue)
        self.traverser.addCollider(self.ball.colNode, self.queue)
        self.traverser.showCollisions(render)

        self.setKeys()

        taskMgr.add(self.updateScene, "updateScene")
        self.physics.start(self.ball)

    def setKeys(self):
        self.keyMap = {"left":0, "right":0, "forward":0}

        self.accept("a", self.setKey, ["left", 1])
        self.accept("d", self.setKey, ["right", 1])
        self.accept("w", self.setKey, ["forward", 1])

        self.accept("a-up", self.setKey, ["left", 0])
        self.accept("d-up", self.setKey, ["right", 0])
        self.accept("w-up", self.setKey, ["forward", 0])

        self.accept("space", self.player.kick)
        self.accept("escape", sys.exit)

    def setKey(self, key, value):
        self.keyMap[key] = value

    def updateScene(self, task):
        self.player.update(self.keyMap)

        inRange = False

        self.traverser.traverse(render)

        for i in range(self.queue.getNumEntries()):
            entry = self.queue.getEntry(i)
            fromName = entry.getFromNodePath().getName()
            intoName = entry.getIntoNodePath().getName()

            if fromName == "playerNode":
                if intoName == "ballNode":
                    inRange = True
                else: #it's a boundary
                self.player.retrocede()
            elif fromName == "ballNode":
                if intoName == "playerNode":
                    pass
                else: #it's a boundary
                    self.field.collide(self.ball, intoName)

        if self.player.isInRange() != inRange:
            self.player.setInRange(inRange)

        return task.cont

game = Game()
game.run()
```

field.py:

```

from pandac.PandaModules import *

class Field():

    def __init__(self, game):

        self.model = game.loader.loadModel("models/field")
        self.model.reparentTo(render)
        self.model.setScale(.1, .1, 1)

        self.geom = OdePlaneGeom(game.physics.space, Vec4(0, 0, 1, 0))

        self.boundaries = {}
        self.addBoundary("boundaryRight",  Vec3(1, 0, 0),  Point3(-500, 0, 0),  (1,  0), (-1, 1))
        self.addBoundary("boundaryLeft",   Vec3(-1, 0, 0), Point3(500, 0, 0),   (-1, 0), (-1, 1))
        self.addBoundary("boundaryBottom", Vec3(0, -1, 0), Point3(0, 1000, 0),  (0, -1), (1,
-1))
        self.addBoundary("boundaryTop",    Vec3(0, 1, 0),  Point3(0, -1000, 0), (0,  1), (1,
-1))

    def addBoundary(self, name, orientation, position, posOffset, velMultiplier):
        boundary = CollisionPlane(Plane(orientation, position))
        boundary.setTangible(True)
        self.boundaries[name] = [boundary, posOffset, velMultiplier]
        colNode = self.model.attachNewNode(CollisionNode(name))
        colNode.node().addSolid(boundary)

    def collide(self, ball, boundaryName):
        vel = ball.body.getLinearVel()
        pos = ball.body.getPosition()

        b = self.boundaries[boundaryName]
        ball.body.setPosition(pos.getX() + b[1][0], pos.getY() + b[1][1], pos.getZ())
        ball.body.setLinearVel(VBase3(vel[0] * b[2][0], vel[1] * b[2][1], vel[2]))
```

player.py:

```

from direct.actor.Actor  import Actor
from pandac.PandaModules import *
import math

class Player(Actor):

    def __init__(self, ball):
        Actor.__init__(self, "models/ralph", {"run":"models/ralph-run", "walk":"models/ralph-
walk"})

        self.ball = ball

        self.reparentTo(render)
        self.setPos(0, 10, 0)

        self.inRange = False
        self.isMoving = False

        self.colSphere = CollisionSphere(0, 0, 2.5, 3)
        self.colNode = self.attachNewNode(CollisionNode('playerNode'))
        self.colNode.node().addSolid(self.colSphere)
        self.colNode.show()

    def advance(self):
        self.setY(self, -20 * globalClock.getDt())

    def isInRange(self):
        return self.inRange

    def kick(self):
        if self.isInRange():
            force = 2000
            angle = self.getH(render) - 90
            vector = Vec3(math.cos(math.radians(angle)), math.sin(math.radians(angle)), .5)
            self.ball.body.setForce(vector.getX() * force, vector.getY() * force, vector.getZ() *
force)
            self.ball.body.setAngularVel(vector.getX() * 10, vector.getY() * 10, 0)

    def retrocede(self):
        self.setY(self, 20 * globalClock.getDt())

    def setInRange(self, inRange):
        self.inRange = inRange

    def turnLeft(self):
        self.setH(self.getH() + 300 * globalClock.getDt())

    def turnRight(self):
        self.setH(self.getH() - 300 * globalClock.getDt())

    def update(self, keyMap):
        if keyMap["left"]:
            self.turnLeft()
        if keyMap["right"]:
            self.turnRight()
        if keyMap["forward"]:
            self.advance()

        base.cam.setPos(self.getX(), self.getY() + 60, self.getZ() + 55)
        base.cam.lookAt(self)

        if keyMap["forward"] or keyMap["left"] or keyMap["right"]:
            if self.isMoving is False:
                self.loop("run")
                self.isMoving = True
        elif self.isMoving:
            self.stop()
            self.pose("walk", 5)
            self.isMoving = False
```

13. Adding goals

Right, so we have a field, a fully controllable player, a ball and boundaries that keep the player and ball from running away. Next up, the goals.

First of all let's create the Goal class itself.

1. Create a new python module, and call it goal.py.

2. On goal.py, create the Goal class:

```
from pandac.PandaModules import *

class Goal():

    def __init__(self, game, name, yPosition):

        self.model = game.loader.loadModel("models/goal")
        self.model.reparentTo(render)
        self.model.setScale(.025, .05, .03)
        self.model.setX(.2)
        self.model.setY(yPosition)

        self.colTubeBottom = CollisionTube(-350, 0, 50, 350, 0, 50, 50)
        self.colNode = self.model.attachNewNode(CollisionNode('goal' + name))
        self.colNode.node().addSolid(self.colTubeBottom)
        self.colTubeTop = CollisionTube(-350, 0, 200, 350, 0, 200, 50)
        self.colNode = self.model.attachNewNode(CollisionNode('goal' + name))
        self.colNode.node().addSolid(self.colTubeTop)
        self.colNode.show()
```

Nothing new so far, we just loaded the 3d model and created the collision solid, in this case two parallel CollisionTubes, one on top of each other.

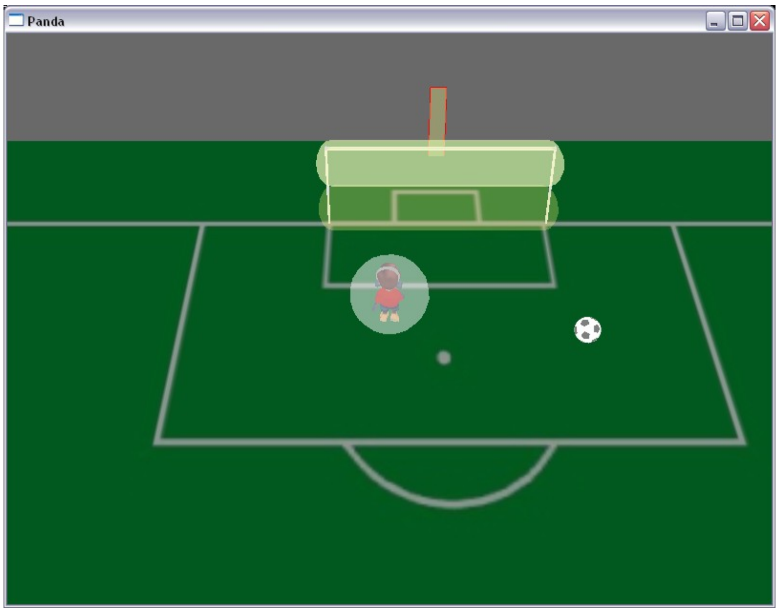
3. Next, on the Game class constructor, let's instantiate two goals, one for the red team, one for the blue team.

```
        self.goals = {
            "Blue" : Goal(self, "Blue", 87),
            "Red"  : Goal(self, "Red", -90)
        }
```

4. At the top of pandasoccer.py, import the Goal class.

```
from goal import Goal
```

5. Check if the game runs, and the goals are in place.



We have the goals, now let's write the code that handles the collisions with them. The behavior we want to add is that if the player collides with a goal, nothing happens, but if it's a ball, the positions of the player and the ball are reset to the starter positions.

6. On the Game class, on the updateScene method replace the following lines:

```
        if fromName == "playerNode":
            if intoName == "ballNode":
                inRange = True
            else: #it's a boundary
                self.player.retrocede()
        elif fromName == "ballNode":
            if intoName == "playerNode":
                pass
            else: #it's a boundary
                self.field.collide(self.ball, intoName)
```

with:

```
        if fromName == "playerNode":
            if intoName == "ballNode":
                inRange = True
            elif intoName == "goalBlue" or intoName == "goalRed":
                pass
            else: #it's a boundary
                self.player.retrocede()
        elif fromName == "ballNode":
            if intoName == "playerNode":
                pass
            elif intoName == "goalBlue" or intoName == "goalRed":
                self.resetPositions()
            else: #it's a boundary
                self.field.collide(self.ball, intoName)
```

7. On the Game class, add the resetPositions method.

```
    def resetPositions(self):
        self.ball.resetPosition()
        self.player.resetPosition()
```

8. On the Ball class, add the resetPosition method.

```
    def resetPosition(self):
        self.body.setPosition(0, 0, 5)
        self.body.setLinearVel(0, 0, 0)
```

9. On the Player class, add the resetPosition method.

```
    def resetPosition(self):
        self.setPos(self.basePos)
        self.setHpr(self.baseHpr)
```

Now the player has to receive a starting position so we can reset it when a point is scored.

10. On the Player class constructor, replace the following lines:

```
    def __init__(self, ball):
        Actor.__init__(self, "models/ralph", {"run":"models/ralph-run", "walk":"models/ralph-
walk"})

        self.ball = ball
        self.reparentTo(render)
```

with:

```
    def __init__(self, ball, basePos, baseHpr):
        Actor.__init__(self, "models/ralph", {"run":"models/ralph-run", "walk":"models/ralph-
walk"})

        self.ball = ball

        self.basePos = basePos
        self.baseHpr = baseHpr

        self.reparentTo(render)
        self.resetPosition()
```

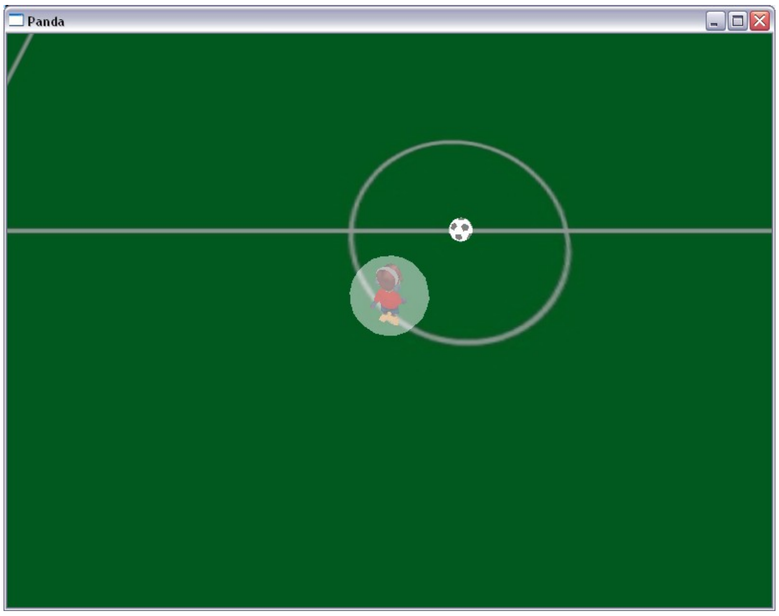
11. Finally, on the Game class constructor, replace the following lines:

```
        self.player = Player(self.ball)
```

with:

```
        self.player = Player(self.ball, (6, 10, 0), (-30, 0, 0))
```

12. Run the game and score a goal. The player and the ball should reset to the initial position.



At the end of this section, your code should look like this:

pandasoccer.py:

```
from direct.showbase.ShowBase import ShowBase
from ball                     import Ball
from field                    import Field
from goal                     import Goal
from pandac.PandaModules      import *
from physics                  import Physics
from player                   import Player
import sys

class Game(ShowBase):

    def __init__(self):
        ShowBase.__init__(self)

        self.physics = Physics()

        self.field  = Field(self)
        self.ball   = Ball(self)

        self.player = Player(self.ball, (6, 10, 0), (-30, 0, 0))
        self.goals = {
            "Blue" : Goal(self, "Blue", 87),
            "Red"  : Goal(self, "Red", -90)
        }

        base.disableMouse()

        self.queue = CollisionHandlerQueue()
        self.traverser = CollisionTraverser()
        self.traverser.addCollider(self.player.colNode, self.queue)
        self.traverser.addCollider(self.ball.colNode, self.queue)
        self.traverser.showCollisions(render)

        self.setKeys()

        taskMgr.add(self.updateScene, "updateScene")
        self.physics.start(self.ball)

    def resetPositions(self):
        self.ball.resetPosition()
        self.player.resetPosition()

    def setKeys(self):
        self.keyMap = {"left":0, "right":0, "forward":0}

        self.accept("a", self.setKey, ["left", 1])
        self.accept("d", self.setKey, ["right", 1])
        self.accept("w", self.setKey, ["forward", 1])

        self.accept("a-up", self.setKey, ["left", 0])
        self.accept("d-up", self.setKey, ["right", 0])
        self.accept("w-up", self.setKey, ["forward", 0])

        self.accept("space", self.player.kick)
        self.accept("escape", sys.exit)

    def setKey(self, key, value):
        self.keyMap[key] = value

    def updateScene(self, task):
        self.player.update(self.keyMap)

        inRange = False

        self.traverser.traverse(render)

        for i in range(self.queue.getNumEntries()):
            entry = self.queue.getEntry(i)
            fromName = entry.getFromNodePath().getName()
            intoName = entry.getIntoNodePath().getName()

            if fromName == "playerNode":
```

```
            if intoName == "ballNode":
                inRange = True
            elif intoName == "goalBlue" or intoName == "goalRed":
                pass
            else: #it's a boundary
                self.player.retrocede()
        elif fromName == "ballNode":
            if intoName == "playerNode":
                pass
            elif intoName == "goalBlue" or intoName == "goalRed":
                self.resetPositions()
            else: #it's a boundary
                self.field.collide(self.ball, intoName)

    if self.player.isInRange() != inRange:
        self.player.setInRange(inRange)

    return task.cont

game = Game()
game.run()
```

ball.py:

```
from pandac.PandaModules import *

class Ball():

    def __init__(self, game):
        self.model = game.loader.loadModel("models/ball")

        self.model.reparentTo(render)
        self.model.setPos(0, 0, 5)
        self.model.setScale(.01)

        self.colSphere = CollisionSphere(0, 0, 0, 100)
        self.colNode = self.model.attachNewNode(CollisionNode('ballNode'))
        self.colNode.node().addSolid(self.colSphere)
        self.colNode.show()

        self.body = OdeBody(game.physics)
        self.mass = OdeMass()
        self.mass.setSphere(260, .1)
        self.body.setMass(self.mass)
        self.body.setPosition(self.model.getPos(render))
        self.body.setQuaternion(self.model.getQuat(render))

        self.geom = OdeSphereGeom(game.physics.space, .8)
        self.geom.setBody(self.body)

    def resetPosition(self):
        self.body.setPosition(0, 0, 5)
        self.body.setLinearVel(0, 0, 0)

    def update(self):
        self.model.setPosQuat(render, self.body.getPosition(),
Quat(self.body.getQuaternion()))

        vel = self.body.getLinearVel()
        if vel[2] < 0.1 and self.body.getPosition().getZ() < 0.9:
            if abs(vel[0]) > 0:
                vel[0] *= .9

            if abs(vel[1]) > 0:
                vel[1] *= .9

        self.body.setLinearVel(vel)
```

goal.py:

```
from pandac.PandaModules import *

class Goal():

    def __init__(self, game, name, yPosition):

        self.model = game.loader.loadModel("models/goal")
        self.model.reparentTo(render)
        self.model.setScale(.025, .05, .03)
        self.model.setX(.2)
        self.model.setY(yPosition)

        self.colTubeBottom = CollisionTube(-350, 0, 50, 350, 0, 50, 50)
        self.colNode = self.model.attachNewNode(CollisionNode('goal' + name))
        self.colNode.node().addSolid(self.colTubeBottom)
        self.colTubeTop = CollisionTube(-350, 0, 200, 350, 0, 200, 50)
        self.colNode = self.model.attachNewNode(CollisionNode('goal' + name))
        self.colNode.node().addSolid(self.colTubeTop)
        self.colNode.show()
```

player.py:

```
from direct.actor.Actor  import Actor
from pandac.PandaModules import *
import math
```

```
class Player(Actor):

    def __init__(self, ball, basePos, baseHpr):
        Actor.__init__(self, "models/ralph", {"run":"models/ralph-run", "walk":"models/ralph-
walk"})

        self.ball = ball

        self.basePos = basePos
        self.baseHpr = baseHpr

        self.reparentTo(render)
        self.resetPosition()

        self.inRange = False
        self.isMoving = False

        self.colSphere = CollisionSphere(0, 0, 2.5, 3)
        self.colNode = self.attachNewNode(CollisionNode('playerNode'))
        self.colNode.node().addSolid(self.colSphere)
        self.colNode.show()

    def advance(self):
        self.setY(self, -20 * globalClock.getDt())

    def isInRange(self):
        return self.inRange

    def kick(self):
        if self.isInRange():
            force = 2000
            angle = self.getH(render) - 90
            vector = Vec3(math.cos(math.radians(angle)), math.sin(math.radians(angle)), .5)
            self.ball.body.setForce(vector.getX() * force, vector.getY() * force, vector.getZ() *
force)
            self.ball.body.setAngularVel(vector.getX() * 10, vector.getY() * 10, 0)

    def resetPosition(self):
        self.setPos(self.basePos)
        self.setHpr(self.baseHpr)

    def retrocede(self):
        self.setY(self, 20 * globalClock.getDt())

    def setInRange(self, inRange):
        self.inRange = inRange

    def turnLeft(self):
        self.setH(self.getH() + 300 * globalClock.getDt())
    def turnRight(self):
        self.setH(self.getH() - 300 * globalClock.getDt())

    def update(self, keyMap):
        if keyMap["left"]:
            self.turnLeft()
        if keyMap["right"]:
            self.turnRight()
        if keyMap["forward"]:
            self.advance()

        base.cam.setPos(self.getX(), self.getY() + 60, self.getZ() + 55)
        base.cam.lookAt(self)

        if keyMap["forward"] or keyMap["left"] or keyMap["right"]:
            if self.isMoving is False:
                self.loop("run")
                self.isMoving = True
        elif self.isMoving:
            self.stop()
            self.pose("walk", 5)
            self.isMoving = False
```

14. Implementing the score HUD

It is now possible to score goals, but without some feedback from the game, we don't really know which team is winning. A new comer to the game also has no idea which keys to press and has to learn them by trial-and-error. So for transmitting this information to the player, we'll implement a HUD. In video gaming, the HUD (heads-up display), is the method by which information is visually relayed to the player as part of a game's user interface.

1. Create a new python module, and call it hud.py.

2. On hud.py, create the Hud class:

```
from direct.gui.OnscreenText import OnscreenText
from pandac.PandaModules  import *

class Hud():

    def __init__(self, score):

        self.score = score
        self.textScore = OnscreenText(
            text  = "Red Team   0 - 0   Blue Team",
            style = 1,
            fg    = (1,1,1,1),
            pos   = (0, 0.90),
            align = TextNode.ACenter,
            scale = .07
        )

        self.addInstructions(-0.75, "[W]: Run forward")
        self.addInstructions(-0.80, "[A]: Turn left")
        self.addInstructions(-0.85, "[D]: Turn right")
        self.addInstructions(-0.90, "[SPACE]: Kick ball")
        self.addInstructions(-0.95, "[ESC]: Quit")

    def addInstructions(self, position, message):
        return OnscreenText(
            text  = message,
            style = 1,
            fg    = (1,1,1,1),
            pos   = (.85, position),
            align = TextNode.ALeft,
            scale = .05
        )

    def update(self):
        self.textScore.setText(
            "Red Team   " + str(self.score["goalRed"]) + " - "
            + str(self.score["goalBlue"]) + "   Blue Team"
        )
```

The OnscreenText object is a convenience wrapper around TextNode. It's a quick way to put text on screen without having to go through the trouble of creating a TextNode and setting properties on it.

3. Next, instantiate the hud on the Game class constructor, and start a structure to store the team scores.

```
        self.score = {
            "goalBlue" : 0,
            "goalRed"  : 0
        }

        self.hud = Hud(self.score)
```

4. At the top of pandasoccer.py, import the Hud class.

```
from hud import Hud
```

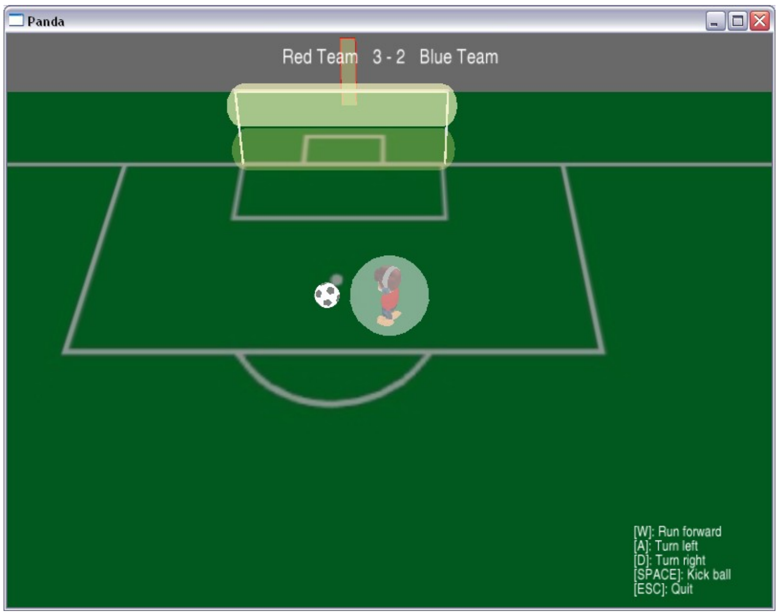
5. On the Game class, on the updateScene method, replace the following lines:

```
        elif fromName == "ballNode":
            if intoName == "playerNode":
                pass
            elif intoName == "goalBlue" or intoName == "goalRed":
                self.resetPositions()
            else: #it's a boundary
                self.field.collide(self.ball, intoName)
```

with:

```
        elif fromName == "ballNode":
            if intoName == "playerNode":
                pass
            elif intoName == "goalBlue" or intoName == "goalRed":
                self.score[intoName] += 1
                self.hud.update()
                self.resetPositions()
            else: #it's a boundary
                self.field.collide(self.ball, intoName)
```

6. Run the game. There's now instructions on the screen, and a score panel. If all went well, the score panel will update when goals are scored.



Your code should now look like this:

pandasoccer.py:

```
from direct.showbase.ShowBase import ShowBase
from ball                     import Ball
from field                    import Field
from goal                     import Goal
from hud                      import Hud
from pandac.PandaModules      import *
from physics                  import Physics
from player                   import Player
import sys

class Game(ShowBase):

   def __init__(self):
      ShowBase.__init__(self)

      self.physics = Physics()

      self.score = {
         "goalBlue" : 0,
         "goalRed"  : 0
      }

      self.hud = Hud(self.score)

      self.field  = Field(self)
      self.ball   = Ball(self)

      self.player = Player(self.ball, (6, 10, 0), (-30, 0, 0))
      self.goals = {
         "Blue" : Goal(self, "Blue", 87),
         "Red"  : Goal(self, "Red", -90)
      }

      base.disableMouse()

      self.queue = CollisionHandlerQueue()
      self.traverser = CollisionTraverser()
      self.traverser.addCollider(self.player.colNode, self.queue)
      self.traverser.addCollider(self.ball.colNode, self.queue)
      self.traverser.showCollisions(render)

      self.setKeys()

      taskMgr.add(self.updateScene, "updateScene")
      self.physics.start(self.ball)

   def resetPositions(self):
      self.ball.resetPosition()
```

```
        self.player.resetPosition()

    def setKeys(self):
        self.keyMap = {"left":0, "right":0, "forward":0}

        self.accept("a", self.setKey, ["left", 1])
        self.accept("d", self.setKey, ["right", 1])
        self.accept("w", self.setKey, ["forward", 1])

        self.accept("a-up", self.setKey, ["left", 0])
        self.accept("d-up", self.setKey, ["right", 0])
        self.accept("w-up", self.setKey, ["forward", 0])

        self.accept("space", self.player.kick)
        self.accept("escape", sys.exit)

    def setKey(self, key, value):
        self.keyMap[key] = value

    def updateScene(self, task):
        self.player.update(self.keyMap)

        inRange = False

        self.traverser.traverse(render)

        for i in range(self.queue.getNumEntries()):
            entry = self.queue.getEntry(i)
            fromName = entry.getFromNodePath().getName()
            intoName = entry.getIntoNodePath().getName()

            if fromName == "playerNode":
                if intoName == "ballNode":
                    inRange = True
                elif intoName == "goalBlue" or intoName == "goalRed":
                    pass
                else: #it's a boundary
                    self.player.retrocede()
            elif fromName == "ballNode":
                if intoName == "playerNode":
                    pass
                elif intoName == "goalBlue" or intoName == "goalRed":
                    self.score[intoName] += 1
                    self.hud.update()
                    self.resetPositions()
                else: #it's a boundary
                    self.field.collide(self.ball, intoName)

        if self.player.isInRange() != inRange:
            self.player.setInRange(inRange)
        return task.cont

game = Game()
game.run()
```

hud.py:

```
from direct.gui.OnscreenText import OnscreenText
from pandac.PandaModules  import *

class Hud():

    def __init__(self, score):

        self.score = score

        self.textScore = OnscreenText(
            text  = "Red Team   0 - 0   Blue Team",
            style = 1,
            fg    = (1,1,1,1),
            pos   = (0, 0.90),
            align = TextNode.ACenter,
            scale = .07
        )

        self.addInstructions(-0.75, "[W]: Run forward")
        self.addInstructions(-0.80, "[A]: Turn left")
        self.addInstructions(-0.85, "[D]: Turn right")
        self.addInstructions(-0.90, "[SPACE]: Kick ball")
        self.addInstructions(-0.95, "[ESC]: Quit")

    def addInstructions(self, position, message):
        return OnscreenText(
            text  = message,
            style = 1,
            fg    = (1,1,1,1),
            pos   = (.85, position),
            align = TextNode.ALeft,
            scale = .05
        )

    def update(self):
        self.textScore.setText(
            "Red Team   " + str(self.score["goalRed"]) + " - "
            + str(self.score["goalBlue"]) + "   Blue Team"
        )
```

15. Cleaning up

Before moving on, there's no reason to keep the collision solids visible, so let's clean up a bit.

1. On the Game class constructor, replace the following line:

```
        self.traverser.showCollisions(render)
```

with:

```
#       self.traverser.showCollisions(render)
```

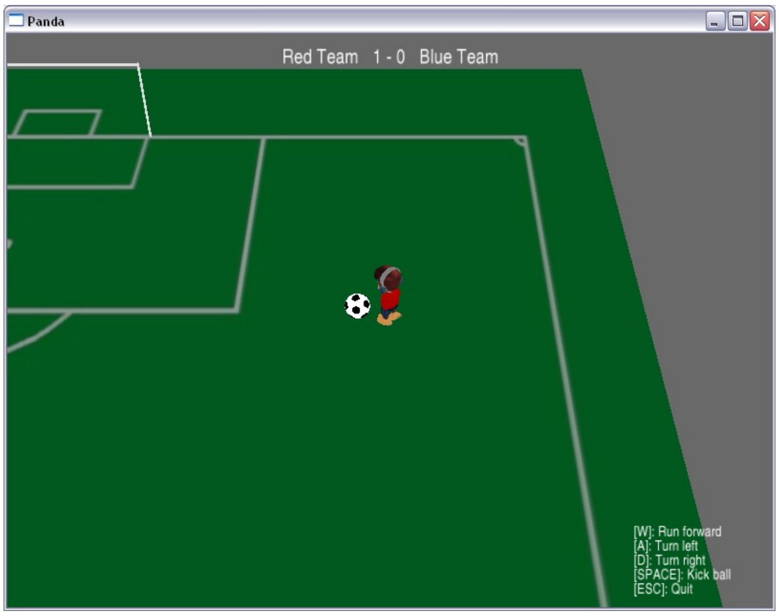
2. On the Ball, Goal and Player class constructors, replace the following line:

```
        self.colNode.show()
```

with:

```
#       self.colNode.show()
```

3. Run the game. The collision solids should be invisible now.



16. Creating a menu

Locally, everything finally works. So now, let's start building the structure to play in a network environment. For that, we'll need a way to distinguish between starting a server, and joining a server. Let's start by creating a menu with these options.

1. Create a new python module, and call it menu.py.

2. On menu.py, create the Menu class:

```
from direct.gui.DirectGui          import *
from direct.gui.OnscreenImage import OnscreenImage
from direct.gui.OnscreenText    import OnscreenText
from pandac.PandaModules      import *

class Menu():

   def __init__(self, game):
      self.background = OnscreenImage(
         image  = 'models/soccer_field.jpg',
         parent = render2d
      )

      self.title = OnscreenText(
         text   = 'Panda Soccer',
         fg     = (1, 1, 1, 1),
         parent = self.background,
         pos    = (0.02, 0.4),
         scale  = 0.2
      )

      self.ip = "127.0.0.1"

      self.buttons = []
      self.addButton("start server", game.startServer, .1)
      self.addButton("join server",  game.joinServer, -.1)

      self.entry = DirectEntry(
         command = self.setIp,
         focusInCommand = self.clearText,
         frameSize   = (-3, 3, -.5, 1),
         initialText = self.ip,
         parent      = self.buttons[1],
         pos         = (0, 0, -1.5),
         text        = "" ,
         text_align  = TextNode.ACenter,
      )
```

```
    def addButton(self, text, command, zPos):
        button = DirectButton(
            command   = command,
            frameSize = (-3, 3, -.5, 1),
            pos       = (0.0, 0.0, zPos),
            scale     = .1,
            text      = text,
        )

        self.buttons.append(button)

    def clearText(self):
        self.entry.enterText('')

    def hide(self):
        self.background.hide()
        for b in self.buttons:
            b.hide()

    def setIp(self, ip):
        print "set ip", ip
        self.ip = ip
```

Just like OnscreenText for text, OnscreenImage is a quick way to put an image on the screen. In this case, we're parenting the background image to render2d, which is a nifty trick to make it stretch to the size of the window.

The menu itself is composed of two buttons: one to start a new server and one to join an existing server. The second button is also accompanied by a DirectEntry object, which serves as a way to input the IP address of the server we wish to join.

3. At the top of pandasoccer.py, import the Menu class.

```
from menu import Menu
```

Until now, when we started the application, the soccer match would start immediately. From now on, we want the game to start only when we start or join a server on the menu.

4. So, on the Game class constructor, delete all the lines, and replace them with:

```
    def __init__(self):
        ShowBase.__init__(self)

        self.menu = Menu(self)
        self.started = False
```

This way, when the application starts, all we'll see is the menu.

Now, let's try to recover the functionality we had before. When the "Start server" button is pressed, the startServer method of the Game class is called.

5. So on the Game class, create the startServer method:

```
    def startServer(self):
        self.physics = Physics()

        self.ball   = Ball(self)

        self.player = Player(self.ball, (6, 10, 0), (-30, 0, 0))

        self.start()

        self.queue = CollisionHandlerQueue()
        self.traverser = CollisionTraverser()
        self.traverser.addCollider(self.player.colNode, self.queue)
        self.traverser.addCollider(self.ball.colNode, self.queue)
#         self.traverser.showCollisions(render)

        self.setKeys()

        taskMgr.add(self.updateScene, "updateScene")
        self.physics.start(self.ball)

        self.started = True
```

6. And also the start method, that creates the basic stuff, such as the field and the hud.

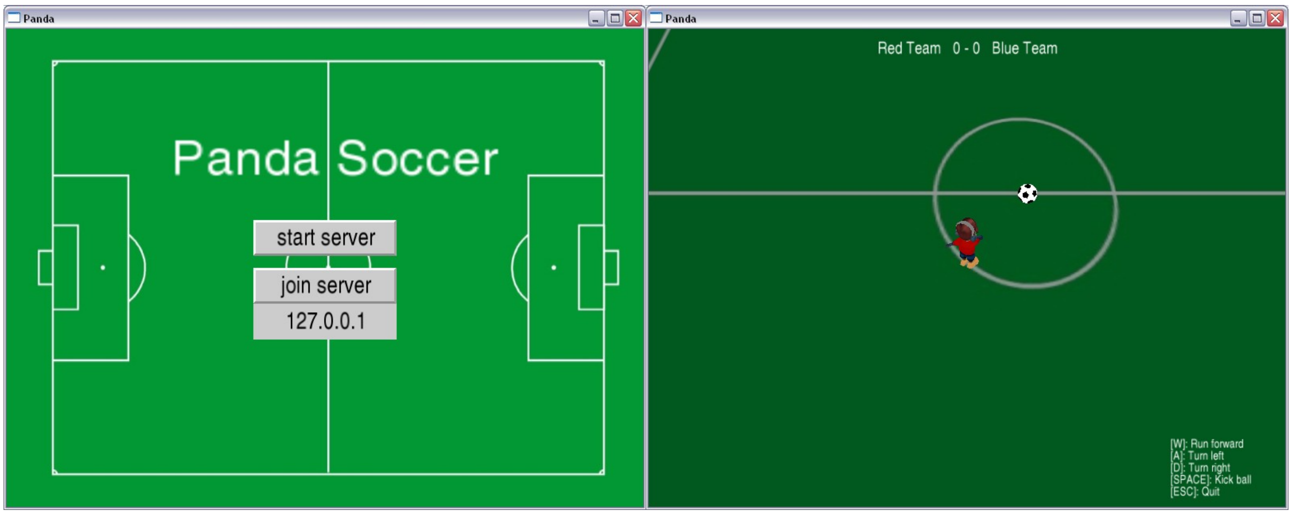
```
    def start(self):
        self.menu.hide()

        self.score = {
            "goalBlue" : 0,
            "goalRed"  : 0
        }

        self.hud = Hud(self.score)
        self.field = Field(self)

        self.goals = {
            "Blue" : Goal(self, "Blue", 87),
            "Red"  : Goal(self, "Red", -90)
        }

        base.disableMouse()
```

When the "Join server" button is pressed, the joinServer method of the Game class is called. For now let's leave that method empty.

7. Still on the Game class, create the joinServer method.

```
  def joinServer(self):
    pass
```

8. Run the game, and start a server. Everything should work just as before.



Your code should now look like this:

pandasoccer.py:

```
from direct.showbase.ShowBase import ShowBase
from ball                     import Ball
from field                    import Field
from goal                     import Goal
from hud                      import Hud
from menu                     import Menu
from pandac.PandaModules      import *
from physics                  import Physics
from player                   import Player
import sys

class Game(ShowBase):
```

```
def __init__(self):
    ShowBase.__init__(self)

    self.menu = Menu(self)

    self.started = False

def joinServer(self):
    pass

def resetPositions(self):
    self.ball.resetPosition()
    self.player.resetPosition()

def setKeys(self):
    self.keyMap = {"left":0, "right":0, "forward":0}

    self.accept("a", self.setKey, ["left", 1])
    self.accept("d", self.setKey, ["right", 1])
    self.accept("w", self.setKey, ["forward", 1])

    self.accept("a-up", self.setKey, ["left", 0])
    self.accept("d-up", self.setKey, ["right", 0])
    self.accept("w-up", self.setKey, ["forward", 0])

    self.accept("space", self.player.kick)
    self.accept("escape", sys.exit)

def setKey(self, key, value):
    self.keyMap[key] = value

def start(self):
    self.menu.hide()

    self.score = {
        "goalBlue" : 0,
        "goalRed"  : 0
    }

    self.hud = Hud(self.score)
    self.field = Field(self)

    self.goals = {
        "Blue" : Goal(self, "Blue", 87),
        "Red"  : Goal(self, "Red", -90)
    }

    base.disableMouse()

def startServer(self):
        self.physics = Physics()

        self.ball   = Ball(self)

        self.player = Player(self.ball, (6, 10, 0), (-30, 0, 0))

        self.start()

        self.queue = CollisionHandlerQueue()
        self.traverser = CollisionTraverser()
        self.traverser.addCollider(self.player.colNode, self.queue)
        self.traverser.addCollider(self.ball.colNode, self.queue)
#         self.traverser.showCollisions(render)

        self.setKeys()

        taskMgr.add(self.updateScene, "updateScene")
        self.physics.start(self.ball)

        self.started = True

    def updateScene(self, task):
        self.player.update(self.keyMap)

        inRange = False

        self.traverser.traverse(render)

        for i in range(self.queue.getNumEntries()):
            entry = self.queue.getEntry(i)
            fromName = entry.getFromNodePath().getName()
            intoName = entry.getIntoNodePath().getName()

            if fromName == "playerNode":
                if intoName == "ballNode":
                    inRange = True
                elif intoName == "goalBlue" or intoName == "goalRed":
                    pass
                else: #it's a boundary
                    self.player.retrocede()
            elif fromName == "ballNode":
                if intoName == "playerNode":
                    pass
                elif intoName == "goalBlue" or intoName == "goalRed":
                    self.score[intoName] += 1
                    self.hud.update()
                    self.resetPositions()
                else: #it's a boundary
                    self.field.collide(self.ball, intoName)
        if self.player.isInRange() != inRange:
            self.player.setInRange(inRange)

        return task.cont

game = Game()
game.run()
```

menu.py:

```
from direct.gui.DirectGui          import *
from direct.gui.OnscreenImage import OnscreenImage
from direct.gui.OnscreenText     import OnscreenText
from pandac.PandaModules        import *

class Menu():

    def __init__(self, game):
        self.background = OnscreenImage(
            image  = 'models/soccer_field.jpg',
            parent = render2d
        )

        self.title = OnscreenText(
            text   = 'Panda Soccer',
            fg     = (1, 1, 1, 1),
            parent = self.background,
            pos    = (0.02, 0.4),
            scale  = 0.2
        )

        self.ip = "127.0.0.1"

        self.buttons = []
        self.addButton("start server", game.startServer, .1)
        self.addButton("join server",  game.joinServer, -.1)

        self.entry = DirectEntry(
            command = self.setIp,
            focusInCommand = self.clearText,
            frameSize   = (-3, 3, -.5, 1),
            initialText = self.ip,
            parent      = self.buttons[1],
            pos         = (0, 0, -1.5),
            text        = "" ,
            text_align  = TextNode.ACenter,
        )
    def addButton(self, text, command, zPos):
        button = DirectButton(
            command    = command,
            frameSize = (-3, 3, -.5, 1),
            pos       = (0.0, 0.0, zPos),
            scale     = .1,
            text      = text,
        )

        self.buttons.append(button)

    def clearText(self):
        self.entry.enterText('')

    def hide(self):
        self.background.hide()
        for b in self.buttons:
            b.hide()

    def setIp(self, ip):
        print "set ip", ip
        self.ip = ip
```

17. Creating a server-client structure

Next we'll create a server and a client for our game, and attempt to send a message from the client to the server, and print it to the screen.

1. Create a new python module, and call it network.py.

2. On network.py, create the Network class:

```
from direct.distributed.PyDatagram         import PyDatagram
from direct.distributed.PyDatagramIterator import PyDatagramIterator
from pandac.PandaModules                   import *

class Network():

    MESSAGE_TEST        = 1

    def __init__(self, game):
        self.game = game

        self.manager  = QueuedConnectionManager()
        self.listener = QueuedConnectionListener(self.manager, 0)
        self.reader   = QueuedConnectionReader(self.manager, 0)
        self.writer   = ConnectionWriter(self.manager, 0)

        self.port_address = 9099

        taskMgr.add(self.taskReaderPolling, "Poll the connection reader", -40)

    def taskReaderPolling(self, task):
        if self.reader.dataAvailable():
            datagram = NetDatagram()

            if self.reader.getData(datagram):
                self.receiveMessage(datagram)

        return task.cont
```

The Network class encapsulates the code that is common to the Client and the Server.

First, it stores the different types of messages that can be traded between the client and the server (so far we only have the MESSAGE_TEST type).

It's also composed of the following elements:

- A QueuedConnectionManager, which handles the low-level connection processes, establishes connections, and handles unexpected network termination;
- A QueuedConnectionListener, which waits for clients to request a connection;
- A QueuedConnectionReader, which buffers incoming data from an active connection;
- A ConnectionWriter, that allows PyDatagrams (an object that encapsulates data) to be transmitted along an active connection.

Finally we run a reader polling task, that catches the incoming data and passes it to the handler method (receiveMessage).

Next we'll create the client class.

3. On network.py, create the NetworkClient class:

```
class NetworkClient(Network):

    def __init__(self, game, ip):
        Network.__init__(self, game)

        self.ip_address = ip
        timeout = 3000  # 3 seconds

        self.connection = self.manager.openTCPClientConnection(self.ip_address,
self.port_address, timeout)
        if self.connection:
            self.reader.addConnection(self.connection)  # receive messages from server

        self.sendMessage("Hey server!", Network.MESSAGE_TEST)

    def sendMessage(self, message, type):
        datagram = PyDatagram()
        datagram.addUint8(type)
        datagram.addString(message)

        self.writer.send(datagram, self.connection)
```

The NetworkClient class basically attempts to connect to an existing server on a given ip address, and greet the server with a test message.

Finally, we're building the server.

4. On network.py, create the NetworkServer class:

```
class NetworkServer(Network):

    def __init__(self, game):
        Network.__init__(self, game)

        self.activeConnections = []

        backlog   = 1000
        tcpSocket = self.manager.openTCPServerRendezvous(self.port_address, backlog)

        self.listener.addConnection(tcpSocket)

        taskMgr.add(self.taskListenerPolling, "Poll the connection listener", -39)

    def receiveMessage(self, datagram):
        iterator = PyDatagramIterator(datagram)
        type = iterator.getUint8()

        if type == Network.MESSAGE_TEST:
            message = iterator.getString()
            print "Server: Message received:", message

    def taskListenerPolling(self, task):
        if self.listener.newConnectionAvailable():
            rendezvous = PointerToConnection()
            netAddress = NetAddress()
            newConnection = PointerToConnection()

            if self.listener.getNewConnection(rendezvous, netAddress, newConnection):
                newConnection = newConnection.p()
                self.activeConnections.append(newConnection)
                self.reader.addConnection(newConnection) # Begin reading connection

        return task.cont
```

The NetworkServer class establishes a meeting point for incoming connections from NetworkClients - a listener - and handles the communication with those clients. In this case, it will receive the test message from the client, and print it to the console.

5. On pandasoccer.py, import all the network.py classes

```
from network import Network, NetworkClient, NetworkServer
```

6. On the Game class constructor, add the following placeholders for the server and client:

```
        self.server = None
        self.client = None
```

7. On the joinServer method of the Game class, replace the following lines:

```
    def joinServer(self):
        pass
```

with:

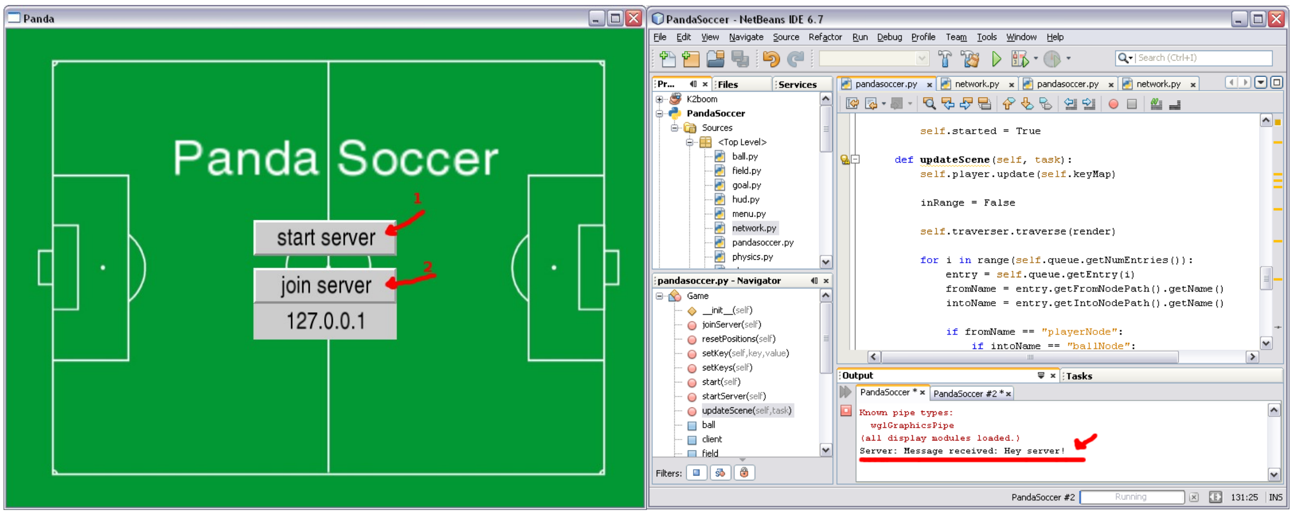
```
    def joinServer(self):
        self.client  = NetworkClient(self, self.menu.ip)
```

8. And on the startServer method of the Game class, add the following line:

```
        self.server = NetworkServer(self)
```

9. Run two instances of PandaSoccer. On one instance start a server, and on the other instance, join the server.

Check the console for the server instance. It should show the greeting message from the client.



```
            self.started = True

        def updateScene(self, task):
            self.player.update(self.keyMap)

            inRange = False

            self.traverser.traverse(render)

            for i in range(self.queue.getNumEntries()):
                entry = self.queue.getEntry(i)
                fromName = entry.getFromNodePath().getName()
                intoName = entry.getIntoNodePath().getName()

                if fromName == "playerNode":
                    if intoName == "ballNode":
```

```
Known pipe types:
  wglGraphicsPipe
(all display modules loaded.)
Server: Message received: Hey server!
```

At the end of this section, your code should look like this:

pandasoccer.py:

```
from direct.showbase.ShowBase import ShowBase
from ball                     import Ball
from field                    import Field
from goal                     import Goal
from hud                      import Hud
from menu                     import Menu
from network                  import Network, NetworkClient, NetworkServer
from pandac.PandaModules      import *
from physics                  import Physics
from player                   import Player
import sys

class Game(ShowBase):

    def __init__(self):
        ShowBase.__init__(self)

        self.menu = Menu(self)

        self.server = None
        self.client = None

        self.started = False

    def joinServer(self):
        self.client  = NetworkClient(self, self.menu.ip)

    def resetPositions(self):
        self.ball.resetPosition()
        self.player.resetPosition()

    def setKeys(self):
        self.keyMap = {"left":0, "right":0, "forward":0}

        self.accept("a", self.setKey, ["left", 1])
        self.accept("d", self.setKey, ["right", 1])
        self.accept("w", self.setKey, ["forward", 1])

        self.accept("a-up", self.setKey, ["left", 0])
        self.accept("d-up", self.setKey, ["right", 0])
        self.accept("w-up", self.setKey, ["forward", 0])

        self.accept("space", self.player.kick)
        self.accept("escape", sys.exit)

    def setKey(self, key, value):
        self.keyMap[key] = value

    def start(self):
        self.menu.hide()

        self.score = {
            "goalBlue" : 0,
            "goalRed"  : 0
        }

        self.hud = Hud(self.score)
        self.field = Field(self)

        self.goals = {
            "Blue" : Goal(self, "Blue", 87),
            "Red"  : Goal(self, "Red", -90)
        }

        base.disableMouse()

    def startServer(self):
        self.server = NetworkServer(self)

        self.physics = Physics()

        self.ball   = Ball(self)

        self.player = Player(self.ball, (6, 10, 0), (-30, 0, 0))

        self.start()

        self.queue = CollisionHandlerQueue()
        self.traverser = CollisionTraverser()
        self.traverser.addCollider(self.player.colNode, self.queue)
        self.traverser.addCollider(self.ball.colNode, self.queue)
#        self.traverser.showCollisions(render)

        self.setKeys()

        taskMgr.add(self.updateScene, "updateScene")
        self.physics.start(self.ball)

        self.started = True

    def updateScene(self, task):
        self.player.update(self.keyMap)

        inRange = False

        self.traverser.traverse(render)

        for i in range(self.queue.getNumEntries()):
            entry = self.queue.getEntry(i)
            fromName = entry.getFromNodePath().getName()
            intoName = entry.getIntoNodePath().getName()

            if fromName == "playerNode":
                if intoName == "ballNode":
                    inRange = True
                elif intoName == "goalBlue" or intoName == "goalRed":
                    pass
                else: #it's a boundary
                    self.player.retrocede()
            elif fromName == "ballNode":
                if intoName == "playerNode":
                    pass
                elif intoName == "goalBlue" or intoName == "goalRed":
                    self.score[intoName] += 1
                    self.hud.update()
                    self.resetPositions()
                else: #it's a boundary
                    self.field.collide(self.ball, intoName)

        if self.player.isInRange() != inRange:
            self.player.setInRange(inRange)

        return task.cont

game = Game()
game.run()
```

network.py:

```
from direct.distributed.PyDatagram          import PyDatagram
from direct.distributed.PyDatagramIterator import PyDatagramIterator
from pandac.PandaModules                    import *

class Network():

   MESSAGE_TEST       = 1

   def __init__(self, game):
      self.game = game

      self.manager  = QueuedConnectionManager()
      self.listener = QueuedConnectionListener(self.manager, 0)
      self.reader   = QueuedConnectionReader(self.manager, 0)
      self.writer   = ConnectionWriter(self.manager, 0)

      self.port_address = 9099

      taskMgr.add(self.taskReaderPolling, "Poll the connection reader", -40)

   def taskReaderPolling(self, task):
      if self.reader.dataAvailable():
         datagram = NetDatagram()

         if self.reader.getData(datagram):
            self.receiveMessage(datagram)

      return task.cont

class NetworkClient(Network):

   def __init__(self, game, ip):
      Network.__init__(self, game)

      self.ip_address = ip

      timeout = 3000  # 3 seconds

      self.connection = self.manager.openTCPClientConnection(self.ip_address,
self.port_address, timeout)
      if self.connection:
         self.reader.addConnection(self.connection)  # receive messages from server

      self.sendMessage("Hey server!", Network.MESSAGE_TEST)

   def sendMessage(self, message, type):
      datagram = PyDatagram()
```

```
        datagram.addUint8(type)
        datagram.addString(message)

        self.writer.send(datagram, self.connection)

class NetworkServer(Network):

    def __init__(self, game):
        Network.__init__(self, game)

        self.activeConnections = []

        backlog   = 1000
        tcpSocket = self.manager.openTCPServerRendezvous(self.port_address, backlog)

        self.listener.addConnection(tcpSocket)

        taskMgr.add(self.taskListenerPolling, "Poll the connection listener", -39)

    def receiveMessage(self, datagram):
        iterator = PyDatagramIterator(datagram)
        type = iterator.getUint8()

        if type == Network.MESSAGE_TEST:
            message = iterator.getString()
            print "Server: Message received:", message

    def taskListenerPolling(self, task):
        if self.listener.newConnectionAvailable():
            rendezvous = PointerToConnection()
            netAddress = NetAddress()
            newConnection = PointerToConnection()

            if self.listener.getNewConnection(rendezvous, netAddress, newConnection):
                newConnection = newConnection.p()
                self.activeConnections.append(newConnection)
                self.reader.addConnection(newConnection) # Begin reading connection

        return task.cont
```

18. Synchronizing the ball

We finally have communication between the server and the client, next let's use this communication channel to synchronize the position of the ball between server and clients.

1. First, on the Network class, add more types of messages by replacing the following lines:

```
    MESSAGE_TEST                = 1
```

with:

```
    MESSAGE_TEST                = 1
    MESSAGE_NEW_CLIENT      = 2
    MESSAGE_GAME_STATUS  = 3
```

2. On the Network class, add the following methods:

```
    def addFloatArray(self, datagram, array):
        for i in range(0, len(array)):
            datagram.addFloat32(array[i])

    def getFloatTuple(self, iterator, length):
        array = []
        for i in range(0, length):
            array.append(iterator.getFloat32())

        return tuple(array)
```

These are utility methods that will be used to help construct a PyDatagram. The addFloatArray method adds an array of floats to a datagram, and the getFloatTuple method extracts an array of floats from a datagram, and turns it into a tuple.

3. On the NetworkClient class constructor, replace the following line:

```
        self.sendMessage("Hey server!", Network.MESSAGE_TEST)
```

with:

```
        self.sendEmptyMessage(Network.MESSAGE_NEW_CLIENT)
```

4. Still on the NetworkClient class, add the following methods:

```
    def receiveMessage(self, datagram):
        iterator = PyDatagramIterator(datagram)
        type = iterator.getUint8()

        if type == Network.MESSAGE_GAME_STATUS:
            status = {}
            status['ball_pos']  = self.getFloatTuple(iterator, 3)
            status['ball_quat'] = self.getFloatTuple(iterator, 4)
            self.game.setStatus(status)

        elif type == Network.MESSAGE_NEW_CLIENT:
            self.game.startClient()

    def sendEmptyMessage(self, type):
        datagram = PyDatagram()
        datagram.addUint8(type)

        self.writer.send(datagram, self.connection)
```

These methods are pretty self-explanatory, the receiveMessage method handles the messages, if it's a GAME_STATUS message it updates the ball position, and if it's a NEW_CLIENT message it starts the client.

As for the sendEmptyMessage method, it's used to send messages that don't contain any text.

5. On the NetworkServer class constructor, add the following line:

```
        taskMgr.doMethodLater(.01, self.taskUpdateStatus, "Update status")
```

6. On the receiveMessage method of NetworkServer, replace the following lines:

```
        if type == Network.MESSAGE_TEST:
            message = iterator.getString()
            print "Server: Message received:", message
```

with:

```
        if type == Network.MESSAGE_TEST:
            message = iterator.getString()
            print "Server: Message received:", message

        elif type == Network.MESSAGE_NEW_CLIENT:
            self.sendEmptyMessage(Network.MESSAGE_NEW_CLIENT)
```

7. Still on the NetworkServer class, add the following methods:

```
    def sendEmptyMessage(self, type):
        datagram = PyDatagram()
        datagram.addUint8(type)

        for connection in self.activeConnections:
            self.writer.send(datagram, connection)
```

```
def sendGameStatus(self):
   datagram = PyDatagram()
   datagram.addUint8(Network.MESSAGE_GAME_STATUS)

   self.addFloatArray(datagram, self.game.ball.getPosition())
   self.addFloatArray(datagram, self.game.ball.getQuaternion())

   for connection in self.activeConnections:
      self.writer.send(datagram, connection)
```

Just like on the NetworkClient class, the sendEmptyMessage method is for sending messages that don't contain any text. As for the sendGameStatus method, it sends the position and rotation of the ball to all the clients.

8. Also on NetworkServer class, add the following task:

```
def taskUpdateStatus(self, task):
   self.sendGameStatus()

   return task.again
```

9. On pandasoccer.py, on the joinServer method, add the following lines:

```
   self.physics = Physics()
   self.ball    = Ball(self)
```

10. On pandasoccer.py, add the following methods:

```
def setStatus(self, status):
   if self.started:
      self.ball.setPosQuat(status['ball_pos'], status['ball_quat'])

def startClient(self):
   self.start()

   taskMgr.add(self.updateScene, "updateScene")
   self.physics.start(self.ball)

   self.started = True
```

At the moment we won't have a player on the client side for the camera to follow, so let's fix some code temporarily so we can check if what we've done so far works.

11. On the updateScene method of the Game class, replace the following lines:

```
def updateScene(self, task):
    <<code>>

    return task.cont
```

with:

```
def updateScene(self, task):
    if self.server:
        <<code>>
    else:
        model = self.ball.model
        base.cam.setPos(model.getX(), model.getY() + 60, 55)
        base.cam.lookAt(model)

    return task.cont
```

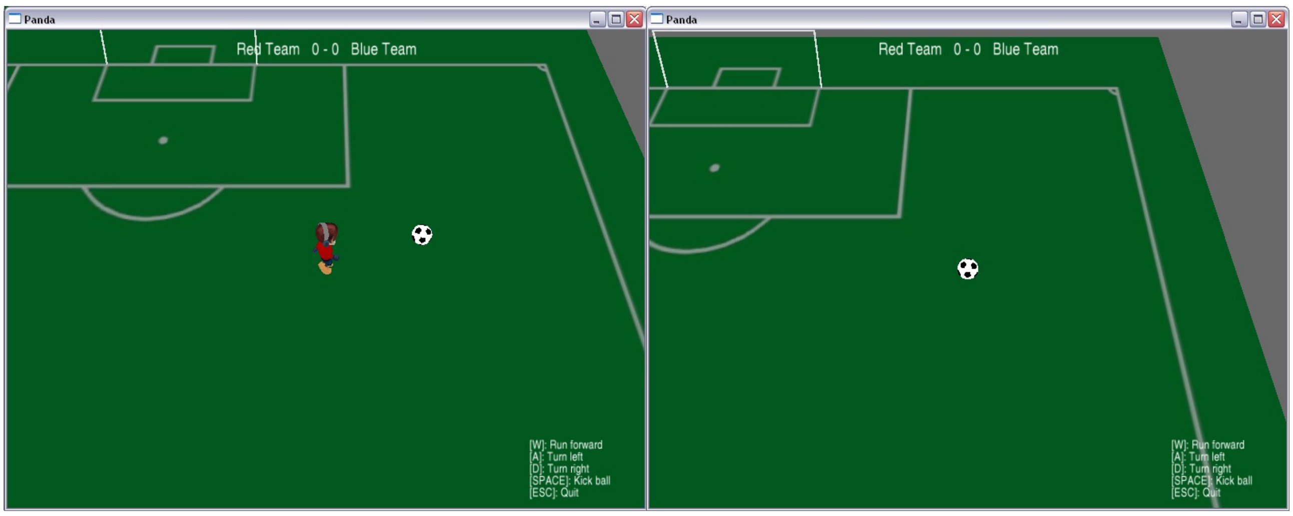
This way, the camera will follow the ball on the client side.

12. Finally, on the Ball class, add the following methods:

```
def getPosition(self):
    return self.body.getPosition()

def getQuaternion(self):
    return self.body.getQuaternion()

def setPosQuat(self, position, quaternion):
    self.body.setPosition(position)
    self.body.setQuaternion(Quat(quaternion))
```

Alright, so let's see what we got so far.

13. Run two instances of PandaSoccer. Start a server on one, and join the server on the other one. Then kick the ball on the server, and check if the ball moves accordingly on the client side.



If all went well, the ball will move on the client window when the server's player kicks the ball.

Your code should now look like this:

pandasoccer.py:

```
from direct.showbase.ShowBase import ShowBase
from ball                     import Ball
from field                    import Field
from goal                     import Goal
from hud                      import Hud
from menu                     import Menu
from network                  import Network, NetworkClient, NetworkServer
from pandac.PandaModules      import *
from physics                  import Physics
from player                   import Player
import sys

class Game(ShowBase):

   def __init__(self):
      ShowBase.__init__(self)

      self.menu = Menu(self)

      self.server = None
      self.client = None

      self.started = False

   def joinServer(self):
      self.client  = NetworkClient(self, self.menu.ip)
```

```
        self.physics = Physics()
        self.ball    = Ball(self)

    def resetPositions(self):
        self.ball.resetPosition()
        self.player.resetPosition()

    def setKeys(self):
        self.keyMap = {"left":0, "right":0, "forward":0}

        self.accept("a", self.setKey, ["left", 1])
        self.accept("d", self.setKey, ["right", 1])
        self.accept("w", self.setKey, ["forward", 1])

        self.accept("a-up", self.setKey, ["left", 0])
        self.accept("d-up", self.setKey, ["right", 0])
        self.accept("w-up", self.setKey, ["forward", 0])

        self.accept("space", self.player.kick)
        self.accept("escape", sys.exit)

    def setKey(self, key, value):
        self.keyMap[key] = value

    def setStatus(self, status):
        if self.started:
            self.ball.setPosQuat(status['ball_pos'], status['ball_quat'])

    def start(self):
        self.menu.hide()

        self.score = {
            "goalBlue" : 0,
            "goalRed"  : 0
        }

        self.hud = Hud(self.score)
        self.field = Field(self)

        self.goals = {
            "Blue" : Goal(self, "Blue", 87),
            "Red"  : Goal(self, "Red", -90)
        }

        base.disableMouse()

    def startClient(self):
        self.start()

        taskMgr.add(self.updateScene, "updateScene")
        self.physics.start(self.ball)

        self.started = True

    def startServer(self):
        self.server = NetworkServer(self)

        self.physics = Physics()

        self.ball   = Ball(self)

        self.player = Player(self.ball, (6, 10, 0), (-30, 0, 0))

        self.start()

        self.queue = CollisionHandlerQueue()
        self.traverser = CollisionTraverser()
        self.traverser.addCollider(self.player.colNode, self.queue)
        self.traverser.addCollider(self.ball.colNode, self.queue)
#       self.traverser.showCollisions(render)

        self.setKeys()

        taskMgr.add(self.updateScene, "updateScene")
        self.physics.start(self.ball)

        self.started = True

    def updateScene(self, task):
        if self.server:
            self.player.update(self.keyMap)

            inRange = False

            self.traverser.traverse(render)

            for i in range(self.queue.getNumEntries()):
                entry = self.queue.getEntry(i)
                fromName = entry.getFromNodePath().getName()
                intoName = entry.getIntoNodePath().getName()

                if fromName == "playerNode":
                    if intoName == "ballNode":
                        inRange = True
                    elif intoName == "goalBlue" or intoName == "goalRed":
                        pass
                    else: #it's a boundary
                        self.player.retrocede()
                elif fromName == "ballNode":
                    if intoName == "playerNode":
```

```
                    pass
                elif intoName == "goalBlue" or intoName == "goalRed":
                    self.score[intoName] += 1
                    self.hud.update()
                    self.resetPositions()
                else: #it's a boundary
                    self.field.collide(self.ball, intoName)

            if self.player.isInRange() != inRange:
                self.player.setInRange(inRange)
        else:
            model = self.ball.model
            base.cam.setPos(model.getX(), model.getY() + 60, 55)
            base.cam.lookAt(model)

        return task.cont

game = Game()
game.run()
```

network.py:

```
from direct.distributed.PyDatagram          import PyDatagram
from direct.distributed.PyDatagramIterator import PyDatagramIterator
from pandac.PandaModules                    import *

class Network():

    MESSAGE_TEST        = 1
    MESSAGE_NEW_CLIENT   = 2
    MESSAGE_GAME_STATUS  = 3

    def __init__(self, game):
        self.game = game

        self.manager  = QueuedConnectionManager()
        self.listener = QueuedConnectionListener(self.manager, 0)
        self.reader   = QueuedConnectionReader(self.manager, 0)
        self.writer   = ConnectionWriter(self.manager, 0)

        self.port_address = 9099

        taskMgr.add(self.taskReaderPolling, "Poll the connection reader", -40)

    def addFloatArray(self, datagram, array):
        for i in range(0, len(array)):
            datagram.addFloat32(array[i])
    def getFloatTuple(self, iterator, length):
        array = []
        for i in range(0, length):
            array.append(iterator.getFloat32())

        return tuple(array)

    def taskReaderPolling(self, task):
        if self.reader.dataAvailable():
            datagram = NetDatagram()

            if self.reader.getData(datagram):
                self.receiveMessage(datagram)

        return task.cont

class NetworkClient(Network):

    def __init__(self, game, ip):
        Network.__init__(self, game)

        self.ip_address = ip

        timeout = 3000  # 3 seconds

        self.connection = self.manager.openTCPClientConnection(self.ip_address,
self.port_address, timeout)
        if self.connection:
            self.reader.addConnection(self.connection)  # receive messages from server

        self.sendEmptyMessage(Network.MESSAGE_NEW_CLIENT)

    def receiveMessage(self, datagram):
        iterator = PyDatagramIterator(datagram)
        type = iterator.getUint8()

        if type == Network.MESSAGE_GAME_STATUS:
            status = {}
            status['ball_pos']  = self.getFloatTuple(iterator, 3)
            status['ball_quat'] = self.getFloatTuple(iterator, 4)

            self.game.setStatus(status)

        elif type == Network.MESSAGE_NEW_CLIENT:
            self.game.startClient()

    def sendEmptyMessage(self, type):
        datagram = PyDatagram()
        datagram.addUint8(type)
        self.writer.send(datagram, self.connection)

    def sendMessage(self, message, type):
        datagram = PyDatagram()
        datagram.addUint8(type)
        datagram.addString(message)

        self.writer.send(datagram, self.connection)

class NetworkServer(Network):

    def __init__(self, game):
        Network.__init__(self, game)

        self.activeConnections = []

        backlog   = 1000
        tcpSocket = self.manager.openTCPServerRendezvous(self.port_address, backlog)

        self.listener.addConnection(tcpSocket)

        taskMgr.add(self.taskListenerPolling, "Poll the connection listener", -39)
        taskMgr.doMethodLater(.01, self.taskUpdateStatus, "Update status")

    def receiveMessage(self, datagram):
        iterator = PyDatagramIterator(datagram)
        type = iterator.getUint8()

        if type == Network.MESSAGE_TEST:
            message = iterator.getString()
            print "Server: Message received:", message

        elif type == Network.MESSAGE_NEW_CLIENT:
            self.sendEmptyMessage(Network.MESSAGE_NEW_CLIENT)

    def sendEmptyMessage(self, type):
        datagram = PyDatagram()
        datagram.addUint8(type)

        for connection in self.activeConnections:
            self.writer.send(datagram, connection)

    def sendGameStatus(self):
        datagram = PyDatagram()
        datagram.addUint8(Network.MESSAGE_GAME_STATUS)

        self.addFloatArray(datagram, self.game.ball.getPosition())
        self.addFloatArray(datagram, self.game.ball.getQuaternion())

        for connection in self.activeConnections:
```

```
        self.writer.send(datagram, connection)

    def taskListenerPolling(self, task):
        if self.listener.newConnectionAvailable():
            rendezvous = PointerToConnection()
            netAddress = NetAddress()
            newConnection = PointerToConnection()

            if self.listener.getNewConnection(rendezvous, netAddress, newConnection):
                newConnection = newConnection.p()
                self.activeConnections.append(newConnection)
                self.reader.addConnection(newConnection) # Begin reading connection

        return task.cont

    def taskUpdateStatus(self, task):
        self.sendGameStatus()

        return task.again
```

ball.py:

```
from pandac.PandaModules import *

class Ball():

    def __init__(self, game):
        self.model = game.loader.loadModel("models/ball")

        self.model.reparentTo(render)
        self.model.setPos(0, 0, 5)
        self.model.setScale(.01)

        self.colSphere = CollisionSphere(0, 0, 0, 100)
        self.colNode = self.model.attachNewNode(CollisionNode('ballNode'))
        self.colNode.node().addSolid(self.colSphere)
#       self.colNode.show()

        self.body = OdeBody(game.physics)
        self.mass = OdeMass()
        self.mass.setSphere(260, .1)
        self.body.setMass(self.mass)
        self.body.setPosition(self.model.getPos(render))
        self.body.setQuaternion(self.model.getQuat(render))

        self.geom = OdeSphereGeom(game.physics.space, .8)
        self.geom.setBody(self.body)
    def getPosition(self):
        return self.body.getPosition()

    def getQuaternion(self):
        return self.body.getQuaternion()

    def resetPosition(self):
        self.body.setPosition(0, 0, 5)
        self.body.setLinearVel(0, 0, 0)

    def setPosQuat(self, position, quaternion):
        self.body.setPosition(position)
        self.body.setQuaternion(Quat(quaternion))

    def update(self):
        self.model.setPosQuat(render, self.body.getPosition(),
Quat(self.body.getQuaternion()))

        vel = self.body.getLinearVel()
        if vel[2] < 0.1 and self.body.getPosition().getZ() < 0.9:
            if abs(vel[0]) > 0:
                vel[0] *= .9

            if abs(vel[1]) > 0:
                vel[1] *= .9

        self.body.setLinearVel(vel)
```

19. Synchronizing the game

Alright, this is it. The infrastructure is built, now we just have to expand the network system to synchronize the whole game instead of just the ball.

We'll add a lot of code on this chapter, but the concepts being used have all been explained on previous chapters. After this final push, the game will finally be complete.

Okay, let's start with the Player class. We want the player to be controllable from a network, to be red or blue depending on the team he's on, and to start on a different position from the other players.

First, let's add some constants, for the different starter positions and the directions he can be moved.

1. On the Game class, replace the following line:

```
class Player(Actor):
```

with:

```
class Player(Actor):
    ADVANCE = 1
    RIGHT      = 2
    LEFT        = 3

    POSITIONS = [
        [(6, 10, 0),    (-30,  0, 0)],
        [(-6, -10, 0),  (-210, 0, 0)],
        [(12, 10, 0),   (-30,  0, 0)],
        [(-12, -10, 0), (-210, 0, 0)],
        [(6, 20, 0),    (-30,  0, 0)],
        [(-6, -20, 0),  (-210, 0, 0)],
        [(12, 20, 0),   (-30,  0, 0)],
        [(-12, -20, 0), (-210, 0, 0)]
    ]
```

2. Next, on the Player class constructor, replace the following lines:

```
    def __init__(self, ball, basePos, baseHpr):
        Actor.__init__(self, "models/ralph", {"run":"models/ralph-run", "walk":"models/ralph-
walk"})

        self.ball = ball

        self.basePos = basePos
        self.baseHpr = baseHpr
```

with:

```
    def __init__(self, ball, color, basePos, baseHpr):
        Actor.__init__(self, "models/" + color + "ralph", {"run":"models/ralph-run",
"walk":"models/ralph-walk"})

        self.ball   = ball
        self.color = color
        self.index = 0

        self.keys = {
            Player.ADVANCE : 0,
            Player.RIGHT   : 0,
            Player.LEFT     : 0
        }
        self.basePos = basePos
        self.baseHpr = baseHpr
```

3. Still on the Player class, add the following methods to control the animations and input externally:

```
    def setAnimationMoving(self):
        self.isMoving = 1
        self.loop("run")

    def setAnimationStop(self):
        self.isMoving = 0
        self.stop()
        self.pose("walk", 5)

    def setInput(self, key, value):
        self.keys[key] = value
```

Lastly, the old update method depended on a centralized key map. Since we now need to control multiple players via a network, let's separate that method into three different new ones, and make update dependant on an instanced key map.

4. On the Player class, replace the following method:

```
    def update(self, keyMap):
        if keyMap["left"]:
            self.turnLeft()
        if keyMap["right"]:
            self.turnRight()
        if keyMap["forward"]:
            self.advance()

        base.cam.setPos(self.getX(), self.getY() + 60, self.getZ() + 55)
        base.cam.lookAt(self)

        if keyMap["forward"] or keyMap["left"] or keyMap["right"]:
            if self.isMoving is False:
                self.loop("run")
                self.isMoving = True
        elif self.isMoving:
            self.stop()
            self.pose("walk", 5)
            self.isMoving = False
```

with:

```
    def update(self):
        if self.keys[Player.LEFT]:
            self.turnLeft()
        if self.keys[Player.RIGHT]:
            self.turnRight()
        if self.keys[Player.ADVANCE]:
            self.advance()

    def updateAnimation(self):
        if self.keys[Player.ADVANCE] or self.keys[Player.LEFT] or self.keys[Player.RIGHT]:
            if not self.isMoving:
                self.setAnimationMoving()
        elif self.isMoving:
            self.setAnimationStop()

    def updateCamera(self):
        base.cam.setPos(self.getX(), self.getY() + 60, self.getZ() + 55)
        base.cam.lookAt(self)
```

Okay, good. The Player class is now finished. Now let's update the Network classes.

First let's add the rest of the message types.

5. On the Network class, replace the following lines:

```
    MESSAGE_TEST             = 1
    MESSAGE_NEW_CLIENT       = 2
    MESSAGE_GAME_STATUS  = 3
```

with:

```
    MESSAGE_TEST                 = 1
    MESSAGE_NEW_CLIENT          = 2
    MESSAGE_GAME_STATUS     = 3
    MESSAGE_UPDATE_TEAMS = 4
    MESSAGE_PLAYER_INPUT    = 5
    MESSAGE_PLAYER_KICK       = 6
    MESSAGE_HUD_UPDATE       = 7
```

Now let's add the handlers for the new messages.

6. On the receiveMessage method of the NetworkClient class, replace the following lines:

```
        if type == Network.MESSAGE_GAME_STATUS:
            status = {}
            status['ball_pos']  = self.getFloatTuple(iterator, 3)
            status['ball_quat'] = self.getFloatTuple(iterator, 4)
        elif type == Network.MESSAGE_NEW_CLIENT:
            self.game.startClient()
```

with:

```
        if type == Network.MESSAGE_GAME_STATUS:
            status = {}
            status['ball_pos']  = self.getFloatTuple(iterator, 3)
            status['ball_quat'] = self.getFloatTuple(iterator, 4)

            self.receiveTeam(status, iterator, "Blue")
            self.receiveTeam(status, iterator, "Red")

            self.game.setStatus(status)

        elif type == Network.MESSAGE_UPDATE_TEAMS:

            status = {}
            self.receiveTeam(status, iterator, "Blue")
            self.receiveTeam(status, iterator, "Red")

            status['color'] = iterator.getString()

            self.game.updateTeams(status)

        elif type == Network.MESSAGE_HUD_UPDATE:
            self.game.score["goalBlue"] = iterator.getUint8()
            self.game.score["goalRed"]  = iterator.getUint8()

            self.game.hud.update()

        elif type == Network.MESSAGE_NEW_CLIENT:
            self.game.startClient()
```

Basically, we're now not only synchronizing the ball, but also the players and the HUD.

7. Still on the NetworkClient class, add the following methods:

```
    def receiveTeam(self, status, iterator, color):
        status[color + '_count'] = iterator.getUint8()
        status[color] = []
        for i in range(0, status[color + '_count']):
            player_pos = self.getFloatTuple(iterator, 3)
            player_hpr = self.getFloatTuple(iterator, 3)
            player_moving = iterator.getBool()
            player_index = iterator.getUint8()
            status[color].append([player_pos, player_hpr, player_moving, player_index])

    def sendPlayerInput(self, color, index, key, value):
        datagram = PyDatagram()
        datagram.addUint8(Network.MESSAGE_PLAYER_INPUT)

        datagram.addString(color)
        datagram.addUint8(index)
        datagram.addUint8(key)
        datagram.addUint8(value)

        self.writer.send(datagram, self.connection)

    def sendPlayerKick(self, color, index, type):
        datagram = PyDatagram()
        datagram.addUint8(type)

        datagram.addString(color)
        datagram.addUint8(index)

        self.writer.send(datagram, self.connection)
```

The NetworkClient is now fully equipped to handle the new types of messages, and now we'll do the same, but for NetworkServer.

8. On the receiveMessage method of NetworkServer, replace:

```
     elif type == Network.MESSAGE_NEW_CLIENT:
        self.sendEmptyMessage(Network.MESSAGE_NEW_CLIENT)
```

with:

```
     elif type == Network.MESSAGE_NEW_CLIENT:
        player = self.game.addOtherPlayer()

        datagram = PyDatagram()
        datagram.addUint8(Network.MESSAGE_UPDATE_TEAMS)
        self.sendTeamStatus(datagram, "Blue")
        self.sendTeamStatus(datagram, "Red")
        datagram.addString(player.color)

        for connection in self.activeConnections:
           self.writer.send(datagram, connection)

        self.sendEmptyMessage(Network.MESSAGE_NEW_CLIENT)

        self.sendHudUpdate(self.game.score)
     elif type == Network.MESSAGE_PLAYER_INPUT:
        color = iterator.getString()
        index = iterator.getUint8()
        key   = iterator.getUint8()
        value = iterator.getUint8()

        self.game.teams[color][index].setInput(key, value)

     elif type == Network.MESSAGE_PLAYER_KICK:
        color = iterator.getString()
        index = iterator.getUint8()

        self.game.teams[color][index].kick()
```

9. On the sendGameStatus method of NetworkServer, replace:

```
        self.addFloatArray(datagram, self.game.ball.getPosition())
        self.addFloatArray(datagram, self.game.ball.getQuaternion())
```

with:

```
        self.addFloatArray(datagram, self.game.ball.getPosition())
        self.addFloatArray(datagram, self.game.ball.getQuaternion())

        self.sendTeamStatus(datagram, "Blue")
        self.sendTeamStatus(datagram, "Red")
```

10. Still on NetworkServer, add the following methods to update the teams and the HUD:

```
    def sendHudUpdate(self, score):
        datagram = PyDatagram()
        datagram.addUint8(Network.MESSAGE_HUD_UPDATE)

        datagram.addUint8(score["goalBlue"])
        datagram.addUint8(score["goalRed"])

        for connection in self.activeConnections:
            self.writer.send(datagram, connection)

    def sendTeamStatus(self, datagram, color):
        team = self.game.teams[color]
        datagram.addUint8(len(team))

        for player in team:
            self.addFloatArray(datagram, player.getPos())
            self.addFloatArray(datagram, player.getHpr())
            datagram.addBool(player.isMoving)
            datagram.addUint8(player.index)
```

Okay, we're now done with network.py as well, only pandasoccer.py to go.

11. First, on the Game class constructor, add the following lines:

```
    self.colliders = {}
    self.lastPosition = 1

    self.playerColor = None
    self.playerIndex = None
    self.teams = {}
```

The colliders array is basically just a structure to easily identify which player collided with a boundary or the ball, based on his collision solid.

Next we'll create a method that will create a new player each time a client connects, and add him to the team that has less players.

12. On the Game class, add the following method:

```
  def addOtherPlayer(self):
    color = "Red"
    if len(self.teams["Red"]) > len(self.teams["Blue"]):
      color = "Blue"

    position = Player.POSITIONS[self.lastPosition]
    self.lastPosition += 1

    player = Player(self.ball, color, position[0], position[1])
    self.teams[color].append(player)
    player.index = len(self.teams[color]) - 1

    self.traverser.addCollider(player.colNode, self.queue)
    self.colliders[player.colNode] = player

    return player
```

When a goal is scored we want every player's position to be resetted.

13. So, on the resetPositions method of the Game class, replace the following lines:

```
  def resetPositions(self):
    self.ball.resetPosition()
    self.player.resetPosition()
```

with:

```
    def resetPositions(self):
        self.ball.resetPosition()

        for color in ("Blue", "Red"):
            for player in self.teams[color]:
                player.resetPosition()
```

While the server will be controlling the player locally, all the clients will have to control him remotely, so we can no longer have the same set of key actions for both server and client.

14. On the Game class, replace the setKeys and setKey methods:

```
    def setKeys(self):
        self.keyMap = {"left":0, "right":0, "forward":0}

        self.accept("a", self.setKey, ["left", 1])
        self.accept("d", self.setKey, ["right", 1])
        self.accept("w", self.setKey, ["forward", 1])

        self.accept("a-up", self.setKey, ["left", 0])
        self.accept("d-up", self.setKey, ["right", 0])
        self.accept("w-up", self.setKey, ["forward", 0])

        self.accept("space", self.player.kick)
        self.accept("escape", sys.exit)

    def setKey(self, key, value):
        self.keyMap[key] = value
```

with:

```
    def setClientKeys(self):
        color = self.player.color
        index = self.player.index

        self.accept("a", self.client.sendPlayerInput, [color, index, Player.LEFT,    1])
        self.accept("d", self.client.sendPlayerInput, [color, index, Player.RIGHT,   1])
        self.accept("w", self.client.sendPlayerInput, [color, index, Player.ADVANCE, 1])

        self.accept("a-up", self.client.sendPlayerInput, [color, index, Player.LEFT,    0])
        self.accept("d-up", self.client.sendPlayerInput, [color, index, Player.RIGHT,   0])
        self.accept("w-up", self.client.sendPlayerInput, [color, index, Player.ADVANCE, 0])

        self.accept("space", self.client.sendPlayerKick, [color, index,
 Network.MESSAGE_PLAYER_KICK])
```

```
        self.accept("escape", sys.exit)

    def setServerKeys(self):
        self.accept("a", self.player.setInput, [Player.LEFT,    1])
        self.accept("d", self.player.setInput, [Player.RIGHT,   1])
        self.accept("w", self.player.setInput, [Player.ADVANCE, 1])

        self.accept("a-up", self.player.setInput, [Player.LEFT,    0])
        self.accept("d-up", self.player.setInput, [Player.RIGHT,   0])
        self.accept("w-up", self.player.setInput, [Player.ADVANCE, 0])

        self.accept("space", self.player.kick)
        self.accept("escape", sys.exit)
```

15. On the setStatus method of the Game class, replace the following lines:

```
    def setStatus(self, status):
        if self.started:
            self.ball.setPosQuat(status['ball_pos'], status['ball_quat'])
```

with:

```
    def setStatus(self, status):
        if self.started:
            self.ball.setPosQuat(status['ball_pos'], status['ball_quat'])

            for color in ("Blue", "Red"):
                for i in range(0, len(self.teams[color])):
                    player = self.teams[color][i]
                    player.setPos(status[color][i][0])
                    player.setHpr(status[color][i][1])

                    if player.isMoving != status[color][i][2]:
                        if player.isMoving:
                            player.setAnimationStop()
                        else:
                            player.setAnimationMoving()
```

16. On the startClient method of the Game class, replace:

```
        self.start()
```

with:

```
        self.start()
        self.setClientKeys()
```

17. On the startServer method, replace:

```
    def startServer(self):
        self.server = NetworkServer(self)

        self.physics = Physics()

        self.ball    = Ball(self)

        self.player = Player(self.ball, (6, 10, 0), (-30, 0, 0))

        self.start()

        self.queue = CollisionHandlerQueue()
        self.traverser = CollisionTraverser()
        self.traverser.addCollider(self.player.colNode, self.queue)
        self.traverser.addCollider(self.ball.colNode, self.queue)
#         self.traverser.showCollisions(render)

        self.setKeys()

        taskMgr.add(self.updateScene, "updateScene")
        self.physics.start(self.ball)

        self.started = True
```

with:

```
    def startServer(self):
        self.server = NetworkServer(self)

        self.physics = Physics()

        self.ball = Ball(self)

        position = Player.POSITIONS[0]
        self.player = Player(self.ball, "Red", position[0], position[1])

        self.teams = {
            "Blue" : [],
            "Red"  : [self.player]
        }

        self.start()
        self.queue = CollisionHandlerQueue()
        self.traverser = CollisionTraverser()
        self.traverser.addCollider(self.player.colNode, self.queue)
        self.traverser.addCollider(self.ball.colNode, self.queue)
#         self.traverser.showCollisions(render)
        self.colliders[self.player.colNode] = self.player

        self.setServerKeys()

        taskMgr.add(self.updateScene, "updateScene")
        self.physics.start(self.ball)

        self.started = True
```

18. On the updateScene method, replace:

```
    def updateScene(self, task):
        if self.server:
            self.player.update(self.keyMap)

            inRange = False

            self.traverser.traverse(render)

            for i in range(self.queue.getNumEntries()):
                entry = self.queue.getEntry(i)
                fromName = entry.getFromNodePath().getName()
                intoName = entry.getIntoNodePath().getName()

                if fromName == "playerNode":
                    if intoName == "ballNode":
                        inRange = True
                    elif intoName == "goalBlue" or intoName == "goalRed":
                        pass
                    else: #it's a boundary
                        self.player.retrocede()
                elif fromName == "ballNode":
                    if intoName == "playerNode":
                        pass
                    elif intoName == "goalBlue" or intoName == "goalRed":
                        self.score[intoName] += 1
                        self.hud.update()
                        self.resetPositions()
                    else: #it's a boundary
                        self.field.collide(self.ball, intoName)

            if self.player.isInRange() != inRange:
                self.player.setInRange(inRange)
        else:
            model = self.ball.model
            base.cam.setPos(model.getX(), model.getY() + 60, 55)
            base.cam.lookAt(model)

        return task.cont
```

with:

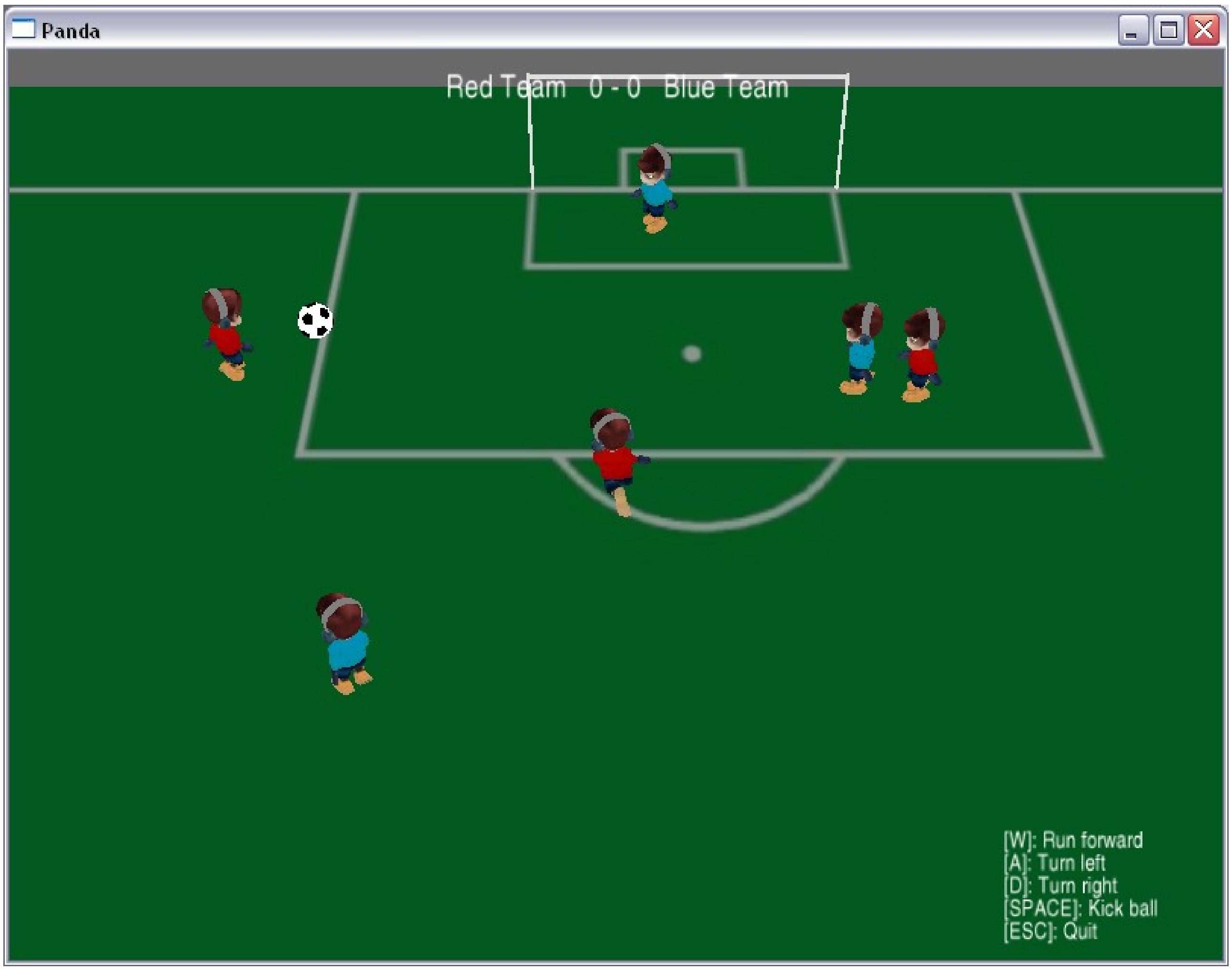
```
    def updateScene(self, task):
        for team in (self.teams['Red'], self.teams['Blue']):
            for player in team:
                player.update()

        if self.server:
            for team in (self.teams['Red'], self.teams['Blue']):
                for player in team:
                    player.updateAnimation()

        self.player.updateCamera()

        inRange = {}
        for color in ("Blue", "Red"):
            for player in self.teams[color]:
                inRange[player] = False

        if self.server:
            self.traverser.traverse(render)

            for i in range(self.queue.getNumEntries()):
                entry = self.queue.getEntry(i)
                fromName = entry.getFromNodePath().getName()
                intoName = entry.getIntoNodePath().getName()

                if fromName == "playerNode":
                    if intoName == "ballNode":
                        player = self.colliders[entry.getFromNodePath()]
                        inRange[player] = True

                    elif intoName == "goalBlue" or intoName == "goalRed" or intoName ==
"playerNode":
                        pass
                    else: #it's a boundary
                        self.colliders[entry.getFromNodePath()].retrocede()
                elif fromName == "ballNode":
                    if intoName == "playerNode":
                        pass
                    elif intoName == "goalBlue" or intoName == "goalRed":
                        self.score[intoName] += 1
                self.server.sendHudUpdate(self.score)
                self.hud.update()
                self.resetPositions()
            else: #it's a boundary
                self.field.collide(self.ball, intoName)

        for player in inRange.keys():
            if inRange[player] != player.inRange:
                player.setInRange(inRange[player])

    return task.cont
```

Last but not least, let's add a method that updates the teams every time a new client is added to the group.

19. On the Game class, add the following method:

```
def updateTeams(self, status):
    if self.teams:
        for color in ("Blue", "Red"):
            for i in range(0, len(self.teams[color])):
                player = self.teams[color][i]
                player.removeNode()
                self.teams[color][i] = None

    self.teams = {
        "Blue" : [],
        "Red"  : []
    }

    for color in ("Blue", "Red"):
        for i in range(0, status[color + "_count"]):
            player = Player(self.ball, color, status[color][i][0], status[color][i][1])
            player.index = status[color][i][3]
            self.teams[color].append(player)

    if self.playerIndex == None:
        self.player = self.teams[status['color']][len(self.teams[status['color']]) - 1]
        self.playerIndex = self.player.index
        self.playerColor = self.player.color
    else:
        self.player = self.teams[self.playerColor][self.playerIndex]
```

20. Run a couple of instances of the game. It should now be possible for multiple players to play via network with each other.



And that's it! You've made a game from start to finish. Congratulations!

While this tutorial gave you an idea of how to make a game in Panda3D, this game is meant as a very simple example. You are encouraged to learn more on the Panda3D manual, available on the Panda3D webpage, and by implementing your own games.

20. Final code listing

pandasoccer.py:

```
from direct.showbase.ShowBase import ShowBase
from ball                     import Ball
from field                    import Field
from goal                     import Goal
from hud                      import Hud
from menu                     import Menu
from network                  import Network, NetworkClient, NetworkServer
from pandac.PandaModules      import *
from physics                           import Physics
from player                            import Player
import sys

class Game(ShowBase):

    def __init__(self):
        ShowBase.__init__(self)

        self.menu = Menu(self)

        self.server = None
        self.client = None

        self.colliders = {}
        self.lastPosition = 1

        self.playerColor = None
        self.playerIndex = None
        self.teams = {}

        self.started = False

    def addOtherPlayer(self):
        color = "Red"
        if len(self.teams["Red"]) > len(self.teams["Blue"]):
            color = "Blue"

        position = Player.POSITIONS[self.lastPosition]
        self.lastPosition += 1

        player = Player(self.ball, color, position[0], position[1])
        self.teams[color].append(player)
        player.index = len(self.teams[color]) - 1

        self.traverser.addCollider(player.colNode, self.queue)
        self.colliders[player.colNode] = player

        return player

    def joinServer(self):
        self.client  = NetworkClient(self, self.menu.ip)
        self.physics = Physics()
        self.ball    = Ball(self)

    def resetPositions(self):
        self.ball.resetPosition()

        for color in ("Blue", "Red"):
            for player in self.teams[color]:
```

```
            player.resetPosition()

    def setClientKeys(self):
        color = self.player.color
        index = self.player.index

        self.accept("a", self.client.sendPlayerInput, [color, index, Player.LEFT,    1])
        self.accept("d", self.client.sendPlayerInput, [color, index, Player.RIGHT,   1])
        self.accept("w", self.client.sendPlayerInput, [color, index, Player.ADVANCE, 1])

        self.accept("a-up", self.client.sendPlayerInput, [color, index, Player.LEFT,    0])
        self.accept("d-up", self.client.sendPlayerInput, [color, index, Player.RIGHT,   0])
        self.accept("w-up", self.client.sendPlayerInput, [color, index, Player.ADVANCE, 0])

        self.accept("space", self.client.sendPlayerKick, [color, index,
Network.MESSAGE_PLAYER_KICK])

        self.accept("escape", sys.exit)

    def setServerKeys(self):
        self.accept("a", self.player.setInput, [Player.LEFT,    1])
        self.accept("d", self.player.setInput, [Player.RIGHT,   1])
        self.accept("w", self.player.setInput, [Player.ADVANCE, 1])

        self.accept("a-up", self.player.setInput, [Player.LEFT,    0])
        self.accept("d-up", self.player.setInput, [Player.RIGHT,   0])
        self.accept("w-up", self.player.setInput, [Player.ADVANCE, 0])

        self.accept("space", self.player.kick)
        self.accept("escape", sys.exit)

    def setStatus(self, status):
        if self.started:
            self.ball.setPosQuat(status['ball_pos'], status['ball_quat'])

            for color in ("Blue", "Red"):
                for i in range(0, len(self.teams[color])):
                    player = self.teams[color][i]
                    player.setPos(status[color][i][0])
                    player.setHpr(status[color][i][1])

                    if player.isMoving != status[color][i][2]:
                        if player.isMoving:
                            player.setAnimationStop()
                        else:
                            player.setAnimationMoving()

    def start(self):
        self.menu.hide()
        self.score = {
            "goalBlue" : 0,
            "goalRed"  : 0
        }

        self.hud = Hud(self.score)
        self.field = Field(self)

        self.goals = {
            "Blue" : Goal(self, "Blue", 87),
            "Red"  : Goal(self, "Red", -90)
        }

        base.disableMouse()

    def startClient(self):
        self.start()

        self.setClientKeys()

        taskMgr.add(self.updateScene, "updateScene")
        self.physics.start(self.ball)

        self.started = True

    def startServer(self):
        self.server = NetworkServer(self)

        self.physics = Physics()

        self.ball   = Ball(self)

        position = Player.POSITIONS[0]
        self.player = Player(self.ball, "Red", position[0], position[1])

        self.teams = {
            "Blue" : [],
            "Red"  : [self.player]
        }

        self.start()

        self.queue = CollisionHandlerQueue()
        self.traverser = CollisionTraverser()
        self.traverser.addCollider(self.player.colNode, self.queue)
        self.traverser.addCollider(self.ball.colNode, self.queue)
#        self.traverser.showCollisions(render)

        self.colliders[self.player.colNode] = self.player
        self.setServerKeys()

        taskMgr.add(self.updateScene, "updateScene")
        self.physics.start(self.ball)

        self.started = True

    def updateScene(self, task):

        for team in (self.teams['Red'], self.teams['Blue']):
            for player in team:
                player.update()

        if self.server:
            for team in (self.teams['Red'], self.teams['Blue']):
                for player in team:
                    player.updateAnimation()

        self.player.updateCamera()

        inRange = {}
        for color in ("Blue", "Red"):
            for player in self.teams[color]:
                inRange[player] = False

        if self.server:
            self.traverser.traverse(render)

            for i in range(self.queue.getNumEntries()):
                entry = self.queue.getEntry(i)
                fromName = entry.getFromNodePath().getName()
                intoName = entry.getIntoNodePath().getName()

                if fromName == "playerNode":
                    if intoName == "ballNode":
                        player = self.colliders[entry.getFromNodePath()]
                        inRange[player] = True

                    elif intoName == "goalBlue" or intoName == "goalRed" or intoName ==
"playerNode":
                        pass
                    else: #it's a boundary
                        self.colliders[entry.getFromNodePath()].retrocede()
                elif fromName == "ballNode":
                    if intoName == "playerNode":
                        pass
                    elif intoName == "goalBlue" or intoName == "goalRed":
                        self.score[intoName] += 1
                        self.server.sendHudUpdate(self.score)
                        self.hud.update()
                self.resetPositions()
            else: #it's a boundary
                self.field.collide(self.ball, intoName)

        for player in inRange.keys():
            if inRange[player] != player.inRange:
                player.setInRange(inRange[player])

        return task.cont

    def updateTeams(self, status):
        if self.teams:
            for color in ("Blue", "Red"):
                for i in range(0, len(self.teams[color])):
                    player = self.teams[color][i]
                    player.removeNode()
                    self.teams[color][i] = None

        self.teams = {
            "Blue" : [],
            "Red"  : []
        }

        for color in ("Blue", "Red"):
            for i in range(0, status[color + "_count"]):
                player = Player(self.ball, color, status[color][i][0], status[color][i][1])
                player.index = status[color][i][3]
                self.teams[color].append(player)

        if self.playerIndex == None:
            self.player = self.teams[status['color']][len(self.teams[status['color']]) - 1]
            self.playerIndex = self.player.index
            self.playerColor = self.player.color
        else:
            self.player = self.teams[self.playerColor][self.playerIndex]

game = Game()
game.run()
```

ball.py:

```
from pandac.PandaModules import *

class Ball():

    def __init__(self, game):
        self.model = game.loader.loadModel("models/ball")
        self.model.reparentTo(render)
        self.model.setPos(0, 0, 5)
        self.model.setScale(.01)

        self.colSphere = CollisionSphere(0, 0, 0, 100)
        self.colNode = self.model.attachNewNode(CollisionNode('ballNode'))
        self.colNode.node().addSolid(self.colSphere)
#       self.colNode.show()

        self.body = OdeBody(game.physics)
        self.mass = OdeMass()
        self.mass.setSphere(260, .1)
        self.body.setMass(self.mass)
        self.body.setPosition(self.model.getPos(render))
        self.body.setQuaternion(self.model.getQuat(render))

        self.geom = OdeSphereGeom(game.physics.space, .8)
        self.geom.setBody(self.body)

    def getPosition(self):
        return self.body.getPosition()

    def getQuaternion(self):
        return self.body.getQuaternion()

    def resetPosition(self):
        self.body.setPosition(0, 0, 5)
        self.body.setLinearVel(0, 0, 0)

    def setPosQuat(self, position, quaternion):
        self.body.setPosition(position)
        self.body.setQuaternion(Quat(quaternion))

    def update(self):
        self.model.setPosQuat(render, self.body.getPosition(),
Quat(self.body.getQuaternion()))

        vel = self.body.getLinearVel()
        if vel[2] < 0.1 and self.body.getPosition().getZ() < 0.9:
            if abs(vel[0]) > 0:
                vel[0] *= .9

            if abs(vel[1]) > 0:
                vel[1] *= .9

        self.body.setLinearVel(vel)
```

field.py:

```
from pandac.PandaModules import *

class Field():

    def __init__(self, game):

        self.model = game.loader.loadModel("models/field")
        self.model.reparentTo(render)
        self.model.setScale(.1, .1, 1)

        self.geom = OdePlaneGeom(game.physics.space, Vec4(0, 0, 1, 0))

        self.boundaries = {}
        self.addBoundary("boundaryRight",  Vec3(1, 0, 0),  Point3(-500, 0, 0),  (1,  0), (-1, 1))
        self.addBoundary("boundaryLeft",   Vec3(-1, 0, 0), Point3(500, 0, 0),   (-1, 0), (-1, 1))
        self.addBoundary("boundaryBottom", Vec3(0, -1, 0), Point3(0, 1000, 0),  (0, -1), (1,
-1))
        self.addBoundary("boundaryTop",    Vec3(0, 1, 0),  Point3(0, -1000, 0), (0,  1), (1,
-1))

    def addBoundary(self, name, orientation, position, posOffset, velMultiplier):
        boundary = CollisionPlane(Plane(orientation, position))
        boundary.setTangible(True)
        self.boundaries[name] = [boundary, posOffset, velMultiplier]
        colNode = self.model.attachNewNode(CollisionNode(name))
        colNode.node().addSolid(boundary)

    def collide(self, ball, boundaryName):
        vel = ball.body.getLinearVel()
        pos = ball.body.getPosition()

        b = self.boundaries[boundaryName]
        ball.body.setPosition(pos.getX() + b[1][0], pos.getY() + b[1][1], pos.getZ())
        ball.body.setLinearVel(VBase3(vel[0] * b[2][0], vel[1] * b[2][1], vel[2]))
```

goal.py:

```
from pandac.PandaModules import *

class Goal():

    def __init__(self, game, name, yPosition):

        self.model = game.loader.loadModel("models/goal")
        self.model.reparentTo(render)
        self.model.setScale(.025, .05, .03)
        self.model.setX(.2)
        self.model.setY(yPosition)

        self.colTubeBottom = CollisionTube(-350, 0, 50, 350, 0, 50, 50)
        self.colNode = self.model.attachNewNode(CollisionNode('goal' + name))
        self.colNode.node().addSolid(self.colTubeBottom)
        self.colTubeTop = CollisionTube(-350, 0, 200, 350, 0, 200, 50)
        self.colNode = self.model.attachNewNode(CollisionNode('goal' + name))
        self.colNode.node().addSolid(self.colTubeTop)
#         self.colNode.show()
```

hud.py:

```

from direct.gui.OnscreenText import OnscreenText
from pandac.PandaModules     import *

class Hud():

    def __init__(self, score):

        self.score = score

        self.textScore = OnscreenText(
            text  = "Red Team   0 - 0   Blue Team",
            style = 1,
            fg    = (1,1,1,1),
            pos   = (0, 0.90),
            align = TextNode.ACenter,
            scale = .07
        )

        self.addInstructions(-0.75, "[W]: Run forward")
        self.addInstructions(-0.80, "[A]: Turn left")
        self.addInstructions(-0.85, "[D]: Turn right")
        self.addInstructions(-0.90, "[SPACE]: Kick ball")
        self.addInstructions(-0.95, "[ESC]: Quit")

    def addInstructions(self, position, message):
        return OnscreenText(
            text  = message,
            style = 1,
            fg    = (1,1,1,1),
            pos   = (.85, position),
            align = TextNode.ALeft,
            scale = .05
        )

    def update(self):

        self.textScore.setText(
            "Red Team   " + str(self.score["goalRed"]) + " - "
            + str(self.score["goalBlue"]) + "   Blue Team"
        )
```

menu.py:

```
from direct.gui.DirectGui     import *
from direct.gui.OnscreenImage import OnscreenImage
from direct.gui.OnscreenText  import OnscreenText
from pandac.PandaModules      import *

class Menu():

    def __init__(self, game):
        self.background = OnscreenImage(
            image  = 'models/soccer_field.jpg',
            parent = render2d
        )

        self.title = OnscreenText(
            text   = 'Panda Soccer',
            fg     = (1, 1, 1, 1),
            parent = self.background,
            pos    = (0.02, 0.4),
            scale  = 0.2
        )

        self.ip = "127.0.0.1"

        self.buttons = []
        self.addButton("start server", game.startServer, .1)
        self.addButton("join server",  game.joinServer, -.1)

        self.entry = DirectEntry(
            command = self.setIp,
            focusInCommand = self.clearText,
            frameSize   = (-3, 3, -.5, 1),
            initialText = self.ip,
            parent      = self.buttons[1],
            pos         = (0, 0, -1.5),
            text        = "" ,
            text_align  = TextNode.ACenter,
        )

    def addButton(self, text, command, zPos):
        button = DirectButton(
            command   = command,
            frameSize = (-3, 3, -.5, 1),
            pos       = (0.0, 0.0, zPos),
            scale     = .1,
            text      = text,
        )

        self.buttons.append(button)

    def clearText(self):
        self.entry.enterText('')

    def hide(self):
        self.background.hide()
        for b in self.buttons:
            b.hide()

    def setIp(self, ip):
        print "set ip", ip
        self.ip = ip
```

network.py:

```
from direct.distributed.PyDatagram         import PyDatagram
from direct.distributed.PyDatagramIterator import PyDatagramIterator
from pandac.PandaModules                   import *

class Network():

    MESSAGE_TEST          = 1
    MESSAGE_NEW_CLIENT    = 2
    MESSAGE_GAME_STATUS   = 3
    MESSAGE_UPDATE_TEAMS  = 4
    MESSAGE_PLAYER_INPUT  = 5
    MESSAGE_PLAYER_KICK   = 6
    MESSAGE_HUD_UPDATE    = 7

    def __init__(self, game):
        self.game = game

        self.manager  = QueuedConnectionManager()
        self.listener = QueuedConnectionListener(self.manager, 0)
        self.reader   = QueuedConnectionReader(self.manager, 0)
        self.writer   = ConnectionWriter(self.manager, 0)

        self.port_address = 9099
        taskMgr.add(self.taskReaderPolling, "Poll the connection reader", -40)

    def addFloatArray(self, datagram, array):
        for i in range(0, len(array)):
            datagram.addFloat32(array[i])

    def getFloatTuple(self, iterator, length):
        array = []
        for i in range(0, length):
            array.append(iterator.getFloat32())

        return tuple(array)

    def taskReaderPolling(self, task):
        if self.reader.dataAvailable():
            datagram = NetDatagram()

            if self.reader.getData(datagram):
                self.receiveMessage(datagram)

        return task.cont

class NetworkClient(Network):

    def __init__(self, game, ip):
        Network.__init__(self, game)

        self.ip_address = ip

        timeout = 3000  # 3 seconds

        self.connection = self.manager.openTCPClientConnection(self.ip_address,
self.port_address, timeout)
        if self.connection:
            self.reader.addConnection(self.connection)  # receive messages from server

        self.sendEmptyMessage(Network.MESSAGE_NEW_CLIENT)

    def receiveMessage(self, datagram):
        iterator = PyDatagramIterator(datagram)
        type = iterator.getUint8()

        if type == Network.MESSAGE_GAME_STATUS:
            status = {}
            status['ball_pos']  = self.getFloatTuple(iterator, 3)
            status['ball_quat'] = self.getFloatTuple(iterator, 4)

            self.receiveTeam(status, iterator, "Blue")
            self.receiveTeam(status, iterator, "Red")
            self.game.setStatus(status)

        elif type == Network.MESSAGE_UPDATE_TEAMS:

            status = {}
            self.receiveTeam(status, iterator, "Blue")
            self.receiveTeam(status, iterator, "Red")

            status['color'] = iterator.getString()

            self.game.updateTeams(status)

        elif type == Network.MESSAGE_HUD_UPDATE:
            self.game.score["goalBlue"] = iterator.getUint8()
            self.game.score["goalRed"]  = iterator.getUint8()

            self.game.hud.update()

        elif type == Network.MESSAGE_NEW_CLIENT:
            self.game.startClient()

    def receiveTeam(self, status, iterator, color):
        status[color + '_count'] = iterator.getUint8()
        status[color] = []
        for i in range(0, status[color + '_count']):
            player_pos = self.getFloatTuple(iterator, 3)
            player_hpr = self.getFloatTuple(iterator, 3)
            player_moving = iterator.getBool()
            player_index = iterator.getUint8()

            status[color].append([player_pos, player_hpr, player_moving, player_index])

    def sendEmptyMessage(self, type):
        datagram = PyDatagram()
        datagram.addUint8(type)

        self.writer.send(datagram, self.connection)

    def sendMessage(self, message, type):
        datagram = PyDatagram()
        datagram.addUint8(type)
        datagram.addString(message)

        self.writer.send(datagram, self.connection)

    def sendPlayerInput(self, color, index, key, value):
        datagram = PyDatagram()
        datagram.addUint8(Network.MESSAGE_PLAYER_INPUT)

        datagram.addString(color)
        datagram.addUint8(index)
        datagram.addUint8(key)
        datagram.addUint8(value)

        self.writer.send(datagram, self.connection)

    def sendPlayerKick(self, color, index, type):
        datagram = PyDatagram()
        datagram.addUint8(type)

        datagram.addString(color)
        datagram.addUint8(index)

        self.writer.send(datagram, self.connection)

class NetworkServer(Network):

    def __init__(self, game):
        Network.__init__(self, game)

        self.activeConnections = []

        backlog   = 1000
        tcpSocket = self.manager.openTCPServerRendezvous(self.port_address, backlog)

        self.listener.addConnection(tcpSocket)

        taskMgr.add(self.taskListenerPolling, "Poll the connection listener", -39)
        taskMgr.doMethodLater(.01, self.taskUpdateStatus, "Update status")

    def receiveMessage(self, datagram):
        iterator = PyDatagramIterator(datagram)
        type = iterator.getUint8()

        if type == Network.MESSAGE_TEST:
            message = iterator.getString()
            print "Server: Message received:", message

        elif type == Network.MESSAGE_NEW_CLIENT:
            player = self.game.addOtherPlayer()

            datagram = PyDatagram()
            datagram.addUint8(Network.MESSAGE_UPDATE_TEAMS)
            self.sendTeamStatus(datagram, "Blue")
            self.sendTeamStatus(datagram, "Red")
            datagram.addString(player.color)

            for connection in self.activeConnections:
                self.writer.send(datagram, connection)
            self.sendEmptyMessage(Network.MESSAGE_NEW_CLIENT)

            self.sendHudUpdate(self.game.score)

        elif type == Network.MESSAGE_PLAYER_INPUT:
            color = iterator.getString()
            index = iterator.getUint8()
            key   = iterator.getUint8()
            value = iterator.getUint8()

            self.game.teams[color][index].setInput(key, value)

        elif type == Network.MESSAGE_PLAYER_KICK:
            color = iterator.getString()
            index = iterator.getUint8()

            self.game.teams[color][index].kick()

    def sendEmptyMessage(self, type):
        datagram = PyDatagram()
        datagram.addUint8(type)

        for connection in self.activeConnections:
            self.writer.send(datagram, connection)

    def sendGameStatus(self):
        datagram = PyDatagram()
        datagram.addUint8(Network.MESSAGE_GAME_STATUS)

        self.addFloatArray(datagram, self.game.ball.getPosition())
        self.addFloatArray(datagram, self.game.ball.getQuaternion())

        self.sendTeamStatus(datagram, "Blue")
        self.sendTeamStatus(datagram, "Red")

        for connection in self.activeConnections:
            self.writer.send(datagram, connection)

    def sendHudUpdate(self, score):
        datagram = PyDatagram()
        datagram.addUint8(Network.MESSAGE_HUD_UPDATE)

        datagram.addUint8(score["goalBlue"])
        datagram.addUint8(score["goalRed"])

        for connection in self.activeConnections:
            self.writer.send(datagram, connection)

    def sendTeamStatus(self, datagram, color):
        team = self.game.teams[color]
```

```
      datagram.addUint8(len(team))

      for player in team:
         self.addFloatArray(datagram, player.getPos())
         self.addFloatArray(datagram, player.getHpr())

         datagram.addBool(player.isMoving)
         datagram.addUint8(player.index)

   def taskListenerPolling(self, task):
      if self.listener.newConnectionAvailable():
         rendezvous = PointerToConnection()
         netAddress = NetAddress()
         newConnection = PointerToConnection()

         if self.listener.getNewConnection(rendezvous, netAddress, newConnection):
            newConnection = newConnection.p()
            self.activeConnections.append(newConnection)
            self.reader.addConnection(newConnection) # Begin reading connection

      return task.cont

   def taskUpdateStatus(self, task):
      self.sendGameStatus()

      return task.again
```

physics.py:

```
from pandac.PandaModules import *

class Physics(OdeWorld):

   def __init__(self):
      OdeWorld.__init__(self)

      self.setGravity(0, 0, -9.81)

      self.initSurfaceTable(1)
      self.setSurfaceEntry(0, 0, 150, 0.5, 3, 0.9, 0.00001, 0.0, 0.002)

      self.space = OdeSimpleSpace()
      self.space.setAutoCollideWorld(self)
      self.contactGroup = OdeJointGroup()
      self.space.setAutoCollideJointGroup(self.contactGroup)

      self.deltaTimeAccumulator = 0.0
```

```
        self.stepSize = 1.0 / 90.0

    def start(self, ball):
        self.ball = ball
        taskMgr.doMethodLater(.1, self.taskPhysics, "taskPhysics")

    def taskPhysics(self, task):
        self.deltaTimeAccumulator += globalClock.getDt()

        while self.deltaTimeAccumulator > self.stepSize:
            self.deltaTimeAccumulator -= self.stepSize
            self.space.autoCollide()
            self.quickStep(self.stepSize)
            self.contactGroup.empty()

        self.ball.update()

        return task.cont
```

player.py:

```
from direct.actor.Actor  import Actor
from pandac.PandaModules import *
import math

class Player(Actor):
    ADVANCE = 1
    RIGHT   = 2
    LEFT    = 3

    POSITIONS = [
        [(6, 10, 0),    (-30,  0, 0)],
        [(-6, -10, 0),  (-210, 0, 0)],
        [(12, 10, 0),   (-30,  0, 0)],
        [(-12, -10, 0), (-210, 0, 0)],
        [(6, 20, 0),    (-30,  0, 0)],
        [(-6, -20, 0),  (-210, 0, 0)],
        [(12, 20, 0),   (-30,  0, 0)],
        [(-12, -20, 0), (-210, 0, 0)]
    ]

    def __init__(self, ball, color, basePos, baseHpr):
        Actor.__init__(self, "models/" + color + "ralph", {"run":"models/ralph-run",
"walk":"models/ralph-walk"})

        self.ball = ball
        self.color = color
        self.index = 0

        self.keys = {
            Player.ADVANCE : 0,
            Player.RIGHT   : 0,
            Player.LEFT    : 0
        }

        self.basePos = basePos
        self.baseHpr = baseHpr

        self.reparentTo(render)
        self.resetPosition()

        self.inRange = False
        self.isMoving = False

        self.colSphere = CollisionSphere(0, 0, 2.5, 3)
        self.colNode = self.attachNewNode(CollisionNode('playerNode'))
        self.colNode.node().addSolid(self.colSphere)
#       self.colNode.show()

    def advance(self):
        self.setY(self, -20 * globalClock.getDt())

    def isInRange(self):
        return self.inRange

    def kick(self):
        if self.isInRange():
            force = 2000
            angle = self.getH(render) - 90
            vector = Vec3(math.cos(math.radians(angle)), math.sin(math.radians(angle)), .5)
            self.ball.body.setForce(vector.getX() * force, vector.getY() * force, vector.getZ() *
force)
            self.ball.body.setAngularVel(vector.getX() * 10, vector.getY() * 10, 0)

    def resetPosition(self):
        self.setPos(self.basePos)
        self.setHpr(self.baseHpr)

    def retrocede(self):
        self.setY(self, 20 * globalClock.getDt())

    def setAnimationMoving(self):
        self.isMoving = 1
        self.loop("run")

    def setAnimationStop(self):
        self.isMoving = 0
        self.stop()
        self.pose("walk", 5)

    def setInput(self, key, value):
        self.keys[key] = value

    def setInRange(self, inRange):
        self.inRange = inRange

    def turnLeft(self):
        self.setH(self.getH() + 300 * globalClock.getDt())

    def turnRight(self):
        self.setH(self.getH() - 300 * globalClock.getDt())

    def update(self):
        if self.keys[Player.LEFT]:
            self.turnLeft()
        if self.keys[Player.RIGHT]:
            self.turnRight()
        if self.keys[Player.ADVANCE]:
            self.advance()

    def updateAnimation(self):
        if self.keys[Player.ADVANCE] or self.keys[Player.LEFT] or self.keys[Player.RIGHT]:
            if not self.isMoving:
                self.setAnimationMoving()
        elif self.isMoving:
            self.setAnimationStop()

    def updateCamera(self):
        base.cam.setPos(self.getX(), self.getY() + 60, self.getZ() + 55)
        base.cam.lookAt(self)
```

And that's it! Be sure to contact me for any questions (**newita@gmail.com**), or just to tell me about games you programmed on your own!