Lesson 10: Moving Player and Camera

3D Game Programming With C++

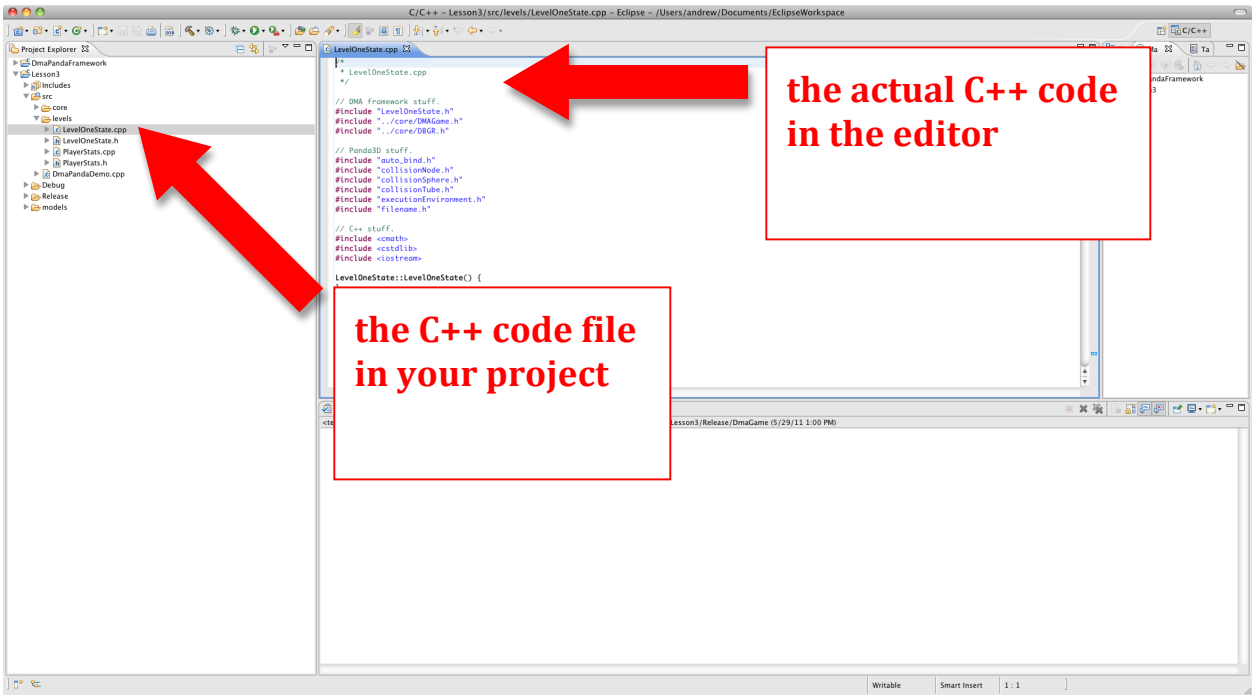
Digital Media Academy (Summer 2011)

Written by: Andrew Uzilov (andrew.uzilov@gmail.com)
Feel free to contact me with any questions.

This lesson will teach you some basics about positioning and moving things in the Panda3D world. Before you begin, make a copy of the DMA framework (you imported and compiled it in Lesson 8) and call it "Lesson10". Also, make a copy of the framework for each individual exercise. See Lesson 8 for how to make copies of a framework.

Diving into the code

Now, let's dive into the actual C++ code! In Project Explorer, navigate to the file `LevelOneState.cpp` in your project and double-click it to open the C++ code that controls how Level One in your game will work. It will appear in the center window (the code editor):



Take a quick glance through the code. Some things to note:

- Remember from Lesson 1 that every C++ program has to have a `main()` function – but there isn't a main function in this code! That's because it is somewhere else in the DMA framework. That main function elsewhere will call functions in your level to make it run.

- The important two functions for this lesson are `init()` and `logic()`. Actually, they methods – they are functions that belong to the `LevelOneState` class (that's why it says `LevelOneState::` before each of them).

- The variables `camera`, `player`, and `playerStats` are defined without being declared. This is because they are declared in the file `LevelOneState.h` – we'll explain how to do that in the next lesson.

When you run the game, **the DMA framework creates a `LevelOneState` object** (a single instance of the `LevelOneState` class). Then, the framework calls the `init()` method on this object exactly once to set up the level. Then, on each frame, the `logic()` method is called exactly once.

Everything that you want to happen during a single frame of the game should go into the `logic()` function. Right now, that function is empty – there is just a note there telling you to put stuff in it! That's why nothing is happening.

Positioning the camera and player in a 3D world

Let's play around with camera and player (panda) positioning. The XYZ positioning of each object (camera, player, etc.) can be changed by the `set_pos()` method. All `NodePath` objects have this method. `player` and `camera` variables are both `NodePath` objects. That means those variables have the `set_pos()` method. Actually, they have every method described in the "Public Member Functions" on this page:

http://www.panda3d.org/reference/1.7.2/cxx/classNodePath.php

That page described everything you can do to a `NodePath` object, and you might find it useful later. But for now, let's see how `set_pos()` works.

If I want to put the `camera` object at position X = 120, Y = -180, Z = 150 in our world, I would use this line of code:

```
camera.set_pos (120, -180, 150);
```

The three arguments to the `set_pos()` method are the X, Y, and Z coordinates, respectively.

Now, find the `camera.set_pos()` call in the `init()` method in `LevelOneState.cpp` and change it to this:

```
camera.set_pos (-120, -180, 150);
```

Now, use the Run button ( ) to run the game. You'll notice Eclipse automatically recompiles the code for you to update the binary (look under the "Console" tab at the bottom for activity). Now your game should look like this:

Compare this view with the view you got in Lesson 8 when you first loaded the framework. The view is the same, except we're on the other side of the panda. The axis of the panda, running "from its nose to its rear," is along the Y-axis of the world (the panda is facing towards negative Y). So, changing X from 120 to -120, we flipped to the other side of the world's Y-axis.

We can also change where the camera is pointing by giving XYZ coordinates to the `camera` object using the `look_at()` method. This method changes the HPR of an object:

```
camera.look_at (0, 0, 0);
```

Initially, the camera is looking at the center of the world (X = 0, Y = 0, Z = 0), which is where the panda is positioned. That's why the panda is centered on the screen. Let's change the camera so that it is looking at the spot right in front of the panda. Remember, the panda is facing towards the negative side of the Y axis. So, the Y coordinate in front of it will be negative. Change the code as follows and recompile/rerun the game:

```
camera.look_at (0, -50, 0);
```

Now, your 3D world should look like this; notice the center of the screen is now in front of the panda, not centered on it:

Of course, everything that we are doing to the `camera` object can also be done to the `player` object – after all, they are both `NodePath` objects! So, we can make the panda "look at" the camera as well, by adding the following line to the `init()` method. <u>Where</u> you put this line is important – it <u>must</u> go after you define and reparent the `player` variable.

```
player.look_at (-120, -180, 150);
```

Run the game and you will learn an amusing and valuable lesson: the "front" of the panda model (according to its internal coordinate space) is not really the "front" that we would expect from just looking at it!

The `look_at()` method changes the HPR of an object without changing its XYZ position or scale. Look up Lesson 9 to remind yourself what these are. But there is another

way to set the HPR: explicitly by using the `set_hpr()` method. Comment out the previous `player.look_at()` code and add the following line to make your code look like this:

```
    //player.look_at (-120, -180, 150);
    player.set_hpr (90, 0, 0);
```

The `set_hpr()` method takes the new heading, position, and rotation as arguments. By default, a new object gets an HPR of (0, 0, 0). So in this case, we rotated the panda 90 degrees clockwise by specifying H = 90, P = 0, R = 0, and the game now looks like this:

Exercise 10.1: Getting used to 3D coordinates

Save each part of this exercise as a separate Eclipse project (except Part 2 – just tell the answer to your instructor).

1) Using the code above as a staring point, position the camera so that it is very high up and directly above the panda, looking down at it, like this:

2) In the view above, where is the world's X-axis? What about the Y-axis? Which way is positive and which way is negative? (**Hint:** Remember that currently, the panda is at XYZ coordinates (0, 0, 0). Try putting the panda at various points like (0, 30, 0) or (-30, 0, 0) to explore where the axes are.

3) Set the panda HPR to (0, 0, 0) and position the camera directly in front of the panda, facing it, like this:

4) Now, tilt the panda on its side, like this (**Hint:** change the panda's HPR):

Moving objects

Now that you are comfortable with 3D coordinates, it's time to make objects in the game move!

At this point, it might be a good idea to make a copy of your project (just like we made a copy of the framework). This way, you can keep the code where you played around in Exercise 3.2 in a separate project. You might want to go back to it.

In a **new project**, change your code so that the panda is positioned like it was when you first opened the framework:

```
player.set_pos (0, 0, 0);
// The player's HPR should be (0, 0, 0).
camera.set_pos (-120, -180, 150);
camera.look_at (0, 0, 0);
```

Let's make the panda move forward, one-half of a unit per frame. The code for updating stuff on every frame goes in the `logic()` method. To create motion, you need to give the panda a new set of coordinates each frame, but **the new coordinates must be relative to the panda's current coordinates**. Relative coordinates are <u>absolutely essential</u> to making motion work. To get the player's current coordinates, we can do this (in the `logic()` method):

```
float x = player.get_x();
float y = player.get_y();
```

```
float z = player.get_z();
```

Now, we can specify new coordinates relative to previous coordinates. To make the panda move forward, add this line:

```
player.set_pos (x, y-0.5, z);
```

Note that we make the Y-coordinate **smaller** (towards negative infinity) to make the panda move "forward" because the panda's head is pointing in the negative Y direction in the 3D world. Run your game and watch the panda move!

However, that's a lot of code for such a simple thing. To wrap up the XYZ coordinates in just one tidy variable, we can use the `Point` object. These objects are returned by the method `get_pos()`, and they can be passed as arguments to `set_pos()` methods. Also, we can do arithmetic on them.

Here is an example of `Point` arithmetic (don't put this into Eclipse). It creates two objects (`coords1` and `coords2`) which represent a point at XYZ coordinates (0, 10, 30) and another point at XYZ coordinates (20, -5, 15). Then, we add the two points together to get a new point at XYZ coordinates (20, 5, 45).

```
Point coords1 = Point (0, 10, 30);  // Point(...) is a constructor
Point coords2 = Point (20, -5, 15);
Point coords3 = coords1 + coords2;  // coord3 has XYZ of (20, 5, 45)
```

This code is basically saying: if I think of a point at `coords1`, and then I move to a point at `coords2` which is **relative** to `coords1` (or vice versa), where am I going to be? The answer is, I'll be a `coords3`.

Now, let's apply these shortcuts to our previous code. We can replace getting the player's current XYZ like this:

```
// Old, long way:
//float x = player.get_x();
//float y = player.get_y();
//float z = player.get_z();
// New, awesome way:
Point currentCoords = player.get_pos();
```

Then we create a new point that represents the player's **difference in position** (once again, think **relative** coordinates) between the previous frame and the current frame. We want to move 0.5 units towards negative infinity on the Y axis. So we create a "delta" point (delta means "change between two values") like this:

```
	Point delta = Point (0, -0.5, 0);
```

Finally, we take the current coords and add them to the **change** in the coords to get the new coords:

```
	player.set_pos (currentCoords + delta);
```

Your new code for the `logic()` method should now look like this (comments not shown):

```
void LevelOneState::logic (void) {
	Point currentCoords = player.get_pos();
	Point delta = Point (0, -0.5, 0);
	player.set_pos (currentCoords + delta);
}
```

Actually, we can do this in an even shorter way. Instead of explicitly putting `Point` objects into variables, we can just use them in `set_pos()` directly, like this:

```
void LevelOneState::logic (void) {
	player.set_pos (player.get_pos() + Point (0, -0.5, 0));
}
```

Try running the game using this code.

Keep in mind that we can do to `camera` the same things that we can do to `player`. Also, we can take a `Point` object and pass it to a `look_at()` method. Here is a way to make the camera follow the player; try running it:

```
void LevelOneState::logic (void) {
	player.set_pos (player.get_pos() + Point (0, -0.5, 0));
	camera.look_at (player.get_pos());
}
```

Just like we have `Point` objects for representing XYZ coordinates, we also have `HPR` objects for representing HPR. Just like with `Point` objects, you can do arithmetic on them to specify **relative** HPRs. This allows you to do continuous rotation of various kinds. For example, try this code for making the panda spin around:

```
void LevelOneState::logic (void) {
     // This "delta" variable stores an HPR object that represents a
     // heading change (delta) of 3 degrees in the counterclockwise
     // direction, and no pitch or rotation change.
     HPR delta = HPR (3, 0, 0);
     // We add the HPR change (delta) to the current HPR to get a new
     // HPR.  Because we do this one per frame, the player rotates
     // 3 degrees counterclockwise on ever frame.
     player.set_hpr (player.get_hpr() + delta);
}
```

The three argument to an `HPR` constructor are heading, pitch, and rotation, in that order.

There is yet one more way to use the `set_pos()` and `set_hpr()` methods. This one is a little tricky, but really **useful if you want to make a camera follow a player around**, or if you want to position *any* object relative to another object that is rotating.

First, try the following code snippet:

```
void LevelOneState::logic (void) {
     player.set_hpr (player.get_hpr() + HPR (-3, 0, 0));
     camera.set_pos (player.get_pos() + Point (150, 0, 100));
     camera.look_at (player.get_pos());
}
```

This makes the panda spin on its axis. The camera is positioned relative to the player's position, but **not the player's HPR** – that's why the camera isn't moving. But what if we wanted to have the camera track the player, no matter what the player did? To do this, we can invoke `set_pos()` using **two** arguments, like this (only the 2nd line of the method is different):

```
void LevelOneState::logic (void) {
     player.set_hpr (player.get_hpr() + HPR (-3, 0, 0));
     camera.set_pos (player, Point (150/0.03, 0, 100/0.03));
     camera.look_at (player.get_pos());
}
```

Try running the game now and don't get dizzy!

By passing `player` as the first argument to `camera.set_pos()`, we tell the camera to be positioned relative to the player, **but inside the player's coordinate space**. This means that we put the camera at XYZ coord (150/0.03, 0, 100/0.03) not in the world, but within the player's own coordinate system, relative to wherever it is pointing at the moment.

Why divide the camera coordinate delta by 0.03? Because of this line in `init()`:

```
player.set_scale (0.03);
```

The player has been scaled down (multiplied) by a factor 0.03, because the player model is really large. So its whole coordinate system has been rescaled. So, if we want an X position of 150 in our 3D world, but we're using `set_pos()` relative to the player, we have to scale the X coordinate back up (divide) by 0.03.

Side Note: Debugging

Usually, the first time you try something, it won't work. Don't be surprised or feel bad – that's actually normal. To figure out what went wrong, we need to do some **debugging**. There are two helpful functions in the DMA framework to help you with debugging: `dump()` and `tripletToString()`.

One of the most common problems you'll have in 3D game programming is that your object is positioned in the wrong place. Sometimes, it's hard to figure out the coordinates of this wrong position. Sometimes an object will even be off the screen, or hide behind something! So, we can use these two functions to print the object's current position to the **Console** on every frame. This way we can tell if our object is positioned the way that we expect it, with great precision.

The `tripletToString()` function takes as input a `Point`, `HPR`, or `Vector` object and converts to a string. The `dump()` function just prints whatever you tell it to the console. This way, you can track the values of each interesting variable on every frame, to see how your game state is changing internally.

For example, the following modifications to `logic()` will move the panda forward and track it with the camera, but will also tell you where the player is, and how the camera is rotated (its HPR). Try them! Pay close attention to the Console tab at the bottom of the Eclipse workspace while your game is running.

```
void LevelOneState::logic (void) {
      player.set_pos (player.get_pos() + Point (0, -0.5, 0));
      dump ("Player's new XYZ: " + tripletToString (player.get_pos()));
      camera.look_at (player.get_pos());
      dump ("Camera's new HPR: " + tripletToString (camera.get_hpr()));
}
```

Now, you know how to make **any** object (player, camera, or other) move around and rotate. Let's test your understanding!

Exercise 10.2: Motion in 3D

1) Write a program where the panda rotates **clockwise** around its Z axis (so, it's heading changes) while rising slowly up into the air. The camera should stay still (its XYZ should be (-120, -180, 150) and not change), but keep pointing at the panda.

2) Write a program where the panda moves diagonally, backwards and to its right (its XYZ coords change, but its HPR remain the same). The camera should move with the panda: it should keep looking at the panda, and the distance between the panda and the camera should **stay the same**.

   There are two ways to do this (one way involves calling `camera.set_pos()` with one argument, the other way with two arguments). Show me **both ways**.

3) Make the panda walk around in a circle while the camera tracks it (but the camera XYZ should not change). **Hint:** When it moves in a circle, the panda is walking forward relative to itself, and turning slightly on each frame.

4) Do the same as #3 above, but position the camera directly behind and slightly above the panda. The camera should follow the panda, so the camera XYZ and HPR **relative to the panda** (hint, hint) should not change.

5) Make the panda walk to a certain point, then stop. **Hint:** This involves conditionals (if/else code) to figure out if the player has reached its destination. If it has, don't call `player.set_pos()` anymore.

6) Make the panda walk back and forth. It should start at point A, move to point B, then rotate (change heading 180 degrees) towards point A, head to point A, then rotate around again to point B, and then repeat the whole process.

   Hint: This will involve several conditionals (if, else if, else) in the `logic()` method that test player state, and decide what it should be doing right now. Write this one step at a time. First, make it so that the player walks to point B, then rotates, then stops. When that's working correctly, add the next step.