

## Lesson 13: Collisions

### 3D Game Programming With C++

#### *Digital Media Academy (Summer 2011)*

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

Entities in your game should be able to interact with each other, otherwise not much will happen. When two entities touch (overlap) in a 3D world, we say they **collided**. Your game should respond to that. For example:

- A player can collide with a power-up, which raises the player's health.
  - Your game logic should remove the power up from the scene graph to make it disappear (it's been picked up), and increase the player health.
- A player can collide with a wall.
  - Your game logic should prevent the player from moving past the wall, by preventing a change in its X, Y, or Z coordinates.
- A bullet can collide with an enemy.
  - Your game logic should remove the bullet from the scene graph to make it disappear, and lower the enemy's health.

How do we detect whether two entities in a Panda3D game are colliding? That's the point of this lesson.

### Collision solids, `CollisionNode` objects, and putting them all together

You can't detect the collision of any two arbitrary nodes in the scene graph because that problem is computationally hard (it will eat up too much CPU time and your game will slow to a crawl).

Instead, we wrap our nodes with simple **collision solids**: basic objects like spheres, tubes, polygons, and so forth. Because these objects are simple, it is much faster to detect if they are overlapping in 3D.

These collision solids are then attached to **`CollisionNode` objects**. These objects are special kinds of Panda3D nodes for which we can detect collisions. Just like all Panda3D nodes, they need to be attached to the scene graph, and we will use a `NodePath` handle to do that.

What does this look like in C++ code? Well, let's start by adding another entity to our 3D world – a banana. We'll show you how to make the panda from Lesson 10 "pick up" this banana. Here we go.

First, we need to declare `NodePath` objects that will be our handle to this banana, and to its collision node. We'll keep this object inside our `LevelOneState` object, because we'll be using it over and over. Add the declaration for `banana` to your class declaration in `LevelOneState.h`, like this:

```
class LevelOneState: public GameState {
private:
    /* ...Other variable definitions from before... */
    NodePath banana;           // the banana model itself
    NodePath bananaSphere;     // the collision sphere on the banana
public:
    /* ... */
}
```

Now, we need to:

- make a path to the file where the banana model is located
- load the banana model, save a handle to it as our `NodePath` object
- reparent the banana node to the scene graph, so it is rendered
- scale and position the banana

To do that, add the following code to your level's `init()` function:

```
// This code figures out the game's current directory
// (the path to the folder where the executable binary
// is located) and saves it into "mydir".
Filename mydir = ExecutionEnvironment::get_binary_name();
mydir = mydir.get_dirname();

// The banana model is a file in a directory that came
// with the DMA framework. We need to get a path to that
// file so we can load it. This path will be relative
// to where this game's executable binary is located.
// It's up one level (that's what ".." means) and
// a few dirs down.
string modelLocation = mydir + "../models/alice-food--banana/banana";

// Load the model.
banana = window->load_model (framework->get_models(), modelLocation);
// Attach it to scene graph.
banana.reparent_to (window->get_render());
// Scale it (the model is small, need to enlarge it).
banana.set_scale (10);
// Put the banana in front of the panda, hovering slightly.
banana.set_pos (0, -50, 10);
```

Try running the game to make sure the banana loads correctly before we move on.

Now, we need to create a collision solid (we'll use a sphere) and attach it to a collision node in the scene graph. It's done like this:

```
// Create a CollisionSphere object (which is kind of collision solid).
// The constructor CollisionSphere() takes two arguments:
//   - The XYZ coord of the center of the sphere relative to its node.
//     We'll make that coord (0, 0, 0), nice and centered.
//   - The radius of the sphere (0.5 units).
PT (CollisionSphere) bananaCS = new CollisionSphere (Point (0, 0, 0), 0.5);

// Create a collision node object.
// Name it "banana" (the only argument to CollisionNode() constructor).
// The name can be anything you want.
PT (CollisionNode) bananaCN = new CollisionNode ("banana");

// Attach the collision solid to the collision node.
bananaCN->add_solid (bananaCS);
```

**But hold on a minute!** What is this PT () stuff? Let's pause and figure this out.

Recall the "Pointers" section at the end of Lesson 7 (see, we told you that you'd need to refer back to it). You'll notice that the declaration/definition like this:

```
PT (CollisionNode) bananaCN = new CollisionNode ("banana");
```

looks sort of similar to a declaration/definition of a pointer to an object, like this:

```
Player *myPlayerPtr = new Player();
```

It's not a coincidence. The variable `bananaCN` is actually a **"fancy pointer"** to an object of type/class `CollisionNode`. These "fancy pointers" (defined by the Panda3D library) have extra features like automatic memory management. But, we won't get into that in this lesson.

The key point is: **variables declared using the syntax**

**PT (ClassName) variableName**

**behave just like pointers.** So, if you want to call the method `add_solid()` on `bananaCN`, you don't use a dot – you use an arrow (`→`) like with a pointer:

```
bananaCN->add_solid (bananaCS);
```

Some Panda3D objects **require** you to use "fancy pointers" to them. `CollisionSphere` (and other collision solids) and `CollisionNode` are such objects. `NodePath` objects **are not** such objects, and you should never use pointers to them.

Now that we have a collision node with a collision sphere attached, we can do the usual `NodePath` dance to attach it to the scene graph:

```
// Create a NodePath handle to the collision node.  
bananaSphere = NodePath (bananaCN);  
  
// Use that handle to attach the collision solid to the scene graph.  
// Notice that the collision solid "bananaCN" will be a CHILD of the  
// node "banana", the 3D model showing the banana.  
// This means if the banana moves, rotates, etc., the collision sphere  
// will move with it, which is convenient.  
bananaSphere.reparent_to (banana);  
  
// This is for debugging only!  
// Normally, collision solids aren't rendered (they are just invisible  
// objects used by the collision detection algorithm).  
// But, for debugging, we will display them for now.  
bananaSphere.show();
```

Try running the game now. If everything worked correctly, you will see the collision sphere around the banana like this:



In the same way as with the banana, we can put a collision sphere around the panda. The code is very similar (the biggest difference is the sphere radius):

```
PT (CollisionSphere) playerCS = new CollisionSphere (Point (0, 0, 0), 600);  
PT (CollisionNode) playerCN = new CollisionNode ("player");  
playerCN->add_solid (playerCS);  
playerSphere = NodePath (playerCN);  
playerSphere.reparent_to (player);  
playerSphere.show();
```

Don't forget to add the declaration for `playerSphere` (a `NodePath` object) to `LevelOneState.cpp` – and then, run the game. If all went well, it should look like this:



Now that both of our game objects have collision solids on them, we can detect collisions between them and have that influence our game!

## Detecting and handling collisions

To show how to detect and react to collisions, let's write a simple "game" that does the following:

- 1) The panda starts by moving forward, towards the banana, slowly.
- 2) When the player touches the banana, it disappears and the player stops moving.
- 3) A message that says "YOU WIN" appears on the screen.

The first step is easy (you've done it before in Lesson 10). Add this code to the `logic()` method:

```
player.set_pos (player.get_pos() - Point (0, 0.3, 0));
```

The second step is a little trickier. To help us, we're going to use a `CollisionManager` object, whose class is defined by the DMA framework. First, we need to declare this object. Do it in the `private` section of `LevelOneState.h`, just like with all the other variables:

```
CollisionManager collisionMgr;
```

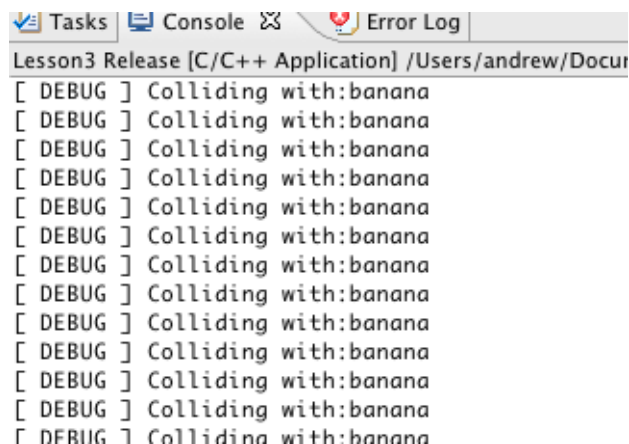
We don't need to define this object before we start using it.

On each frame (meaning in the `logic()` method), we have to tell the collision manager to scan the scene graph and detect collisions between a specific collision node (in this case, `playerSphere`) and all the other collision nodes in the game. It's done like this:

```
collisionMgr.updateCollisions (playerSphere);
```

The single argument to the `updateCollisions()` method is the `NodePath` handle to the collision object we care about. Internally, the collision manager will record all the collisions that happen during this frame.

Try to run the game now, and pay attention to the console. Every time two collision nodes are colliding (when the panda hits the banana), a debug message will be printed, like this:



The screenshot shows a console window with tabs for 'Tasks', 'Console', and 'Error Log'. The 'Console' tab is active, displaying a list of debug messages. The messages are all '[ DEBUG ] Colliding with:banana' and are repeated 12 times. The window title is 'Lesson3 Release [C/C++ Application] /Users/andrew/Docu'.

```
[ DEBUG ] Colliding with:banana  
[ DEBUG ] Colliding with:banana  
[ DEBUG ] Colliding with:banana  
[ DEBUG ] Colliding with:banana  
[ DEBUG ] Colliding with:banana  
[ DEBUG ] Colliding with:banana  
[ DEBUG ] Colliding with:banana  
[ DEBUG ] Colliding with:banana  
[ DEBUG ] Colliding with:banana  
[ DEBUG ] Colliding with:banana  
[ DEBUG ] Colliding with:banana  
[ DEBUG ] Colliding with:banana
```

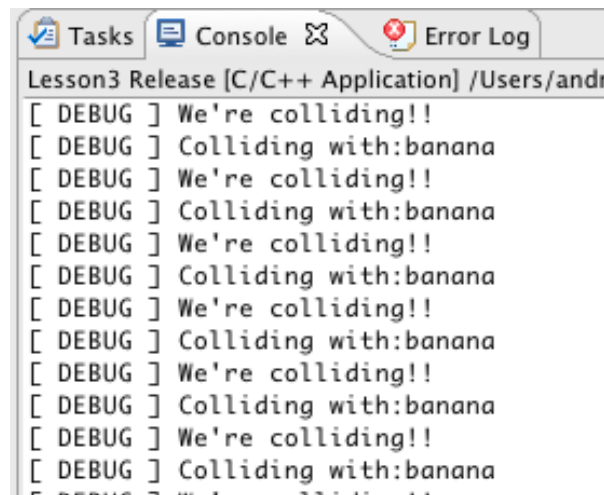


Now, we have to make our game react to the collision. First, how do we ask the collision manager if a collision has occurred? We do it like this:

```
if (collisionMgr.isCollidingWith ("banana")) {  
    dump ("We're colliding!!");  
}
```

The method `isCollidingWith()` will return true if `collisionMgr` detected and stored a collision between `playerSphere` and a node named "banana". That's why it is important to give nodes useful names – because we're going to use them as arguments to functions like this!

If all went well, the console will look like this when the panda hits the banana:



Now, we can put it all together to make the panda stop when it hits the banana, and also to remove the banana from the scene graph. The final answer is this:

```
void LevelOneState::logic (void) {
    if (!gameDone) {
        // Move player.
        player.set_pos (player.get_pos() - Point (0, 0.3, 0));

        // Update data structure that keeps tracks of collisions for
        // this frame. We only care about collisions between player
        // and everything else.
        collisionMgr.updateCollisions (playerSphere);

        if (collisionMgr.isCollidingWith ("banana")) {
            // Remove the banana 3D model by removing it from the
            // scene graph. This also deletes the node completely.
            banana.remove_node();

            // Note that we don't need to remove the node "bananaSphere"
            // because it is a child of "banana" --- if the parent was
            // removed, the children are removed too.

            // By setting this to true, we prevent the "if" conditional
            // from ever going off ever again, effectively ending
            // anything that happens in the game.
            gameDone = true;
        }
    }
}
```

One last thing: you have to define the `gameDone` state variable (a `bool`) in the usual place, and define it to `false` in `init()` – because when the level starts, the state of the game is “not over”.

Try it now! The panda should stop and “pick up” the banana.



But there is a third step – displaying a “YOU WIN” message on the screen. How do we place text on the screen? The magical recipe is in this solution:

```
void LevelOneState::logic (void) {
    if (!gameDone) {
        // Move player.
        player.set_pos (player.get_pos() - Point (0, 0.3, 0));

        // Update data structure that keeps tracks of collisions for
        // this frame. We only care about collisions between player
        // and everything else.
        collisionMgr.updateCollisions (playerSphere);

        if (collisionMgr.isCollidingWith ("banana")) {
            // Remove the banana 3D model by removing it from the
            // scene graph. This also deletes the node completely.
            banana.remove_node();

            // Note that we don't need to remove the node "bananaSphere"
            // because it is a child of "banana" --- if the parent was
            // removed, the children are removed too.

            // By setting this true, we prevent the "if" conditional
            // from ever going off ever again, effectively ending
            // anything that happens in the game.
            gameDone = true;

            // Create a text node, a special kind of node.
            PT (TextNode) winText = new TextNode ("winText");
            // Tell the node what text to display.
            winText->set_text ("YOU WIN!");
            // Make the text center aligned.
            winText->set_align (TextNode::A_center);
            // Fetch a pointer to the window object.
            WindowFramework *window = DMAGame::instance()->getWindow();
            // Attach the node to the scene graph.
            // For 2D text, there is a special way to do it.
            // This way returns a NodePath handle to the resulting
            // text node, which we can use to scale the text,
            // position it, and so forth.
            NodePath winTextNP =
                window->get_aspect_2d().attach_new_node (winText);
            winTextNP.set_scale (0.2);
        }
    }
}
```

Try it! If everything works, you'll get a screen like this at the end:



### Exercise 13.1: Putting together a real game

Create a game where you move a panda around in a 3D world that has 3 bananas in it. When the panda touches a banana, the banana gets picked up – it disappears from the 3D world, just like in our examples in this lesson.

The controls and camera positioning aren't important, choose whatever you want: this can be a first-person game, or have a behind-the-player view, or a top-down view, or anything. It doesn't matter what keys you use to control the panda – that is completely up to you! The panda can even fly, if you'd like. The important thing in this exercise is **getting the logic of your game to work correctly**.

When all the bananas are picked up, the game should stop and a "YOU WIN" message should appear.

### Exercise 13.2:

Extend the game from Exercise 10.1 so that a score is displayed in the corner. Every time you pick up a banana, the score goes up.

**Hint:** In `LevelOneState.h`, declare a `TextNode` smart pointer using `PT()` and define it in `init()`. Call in `scoreText`. Put the text "0" in it, because initially you have a zero score. It's the same way to create text as shown in this lesson, except you are putting the declaration and definition in two different places. Because you declare the `TextNode` smart pointer in the `LevelOneState` class definition, you can use it anywhere in the `LevelOneState` methods.

Then, you can change text in the `logic()` method like this:

```
scoreText->set_text (intToString (score));
```

`intToString()` is a helper method in the DMA framework that converts an `int` variable to a `string` variable. You can only pass variables of type `string` to the method `set_text()`.

### Exercise 13.3:

Change the game from Exercise 10.2 to make the bananas spin.