Creating a Role Playing Game with XNA Game Studio 3.0
Part 44
Sprite to Sprite Collision

To follow along with this tutorial you will have to have read the previous tutorials to understand much of what it going on. You can find a list of tutorials here: XNA 3.0 Role Playing Game Tutorials You will also find the latest version of the project on the web site on that page. If you want to follow along and type in the code from this PDF as you go, you can find the previous project at this link: http://www.jtmbooks.com/rpgtutorials/New2DRPG43.zip You can download the graphics from this link: Graphics.zip

In this tutorial I am going to fix the problem that the player can walk through NPCs at the moment. I will also be doing a few other minor things because just fixing that little problem wouldn't be a full tutorial. The solution was rather simple to implement. I will get started with making it so that the player can not walk through NPCs first. To do that I first made three small changes to the **NPC** class. I added in a new field, **collisionRadius**, a property to get that value called **CollisionRadius**, and in the constructor I set the property. Much like I did with speaking radius of the **NPC** and the attack radius of the **Monster** I will check to see if the player's sprite comes with in a certain radius of the **NPC**. If it does I will just cancel the movement. I set the value of the **collisionRadius** field to be 64 pixels. The reason is that the sprites are 64 pixels by 64 pixels. Half the of 64 is 32, which is the radius of a circle. By doubling that I account for the width of height of both sprites when I do the calculation for the collision between them. Add the following field and property to the **NPC** class and change the code of the constructor to the following.

```
float collisionRadius;

public NPC(Game game,
    Texture2D texture,
    List<Animation> animations,
    DialogComponent dialog,
    Script script)
    : base(game, texture, animations)
{
    speakingRadius = 80f;
    collisionRadius = 64f;
    this.dialog = dialog;
    this.script = script;
    dialog.Hide();
}

public float CollisionRadius
{
    get { return collisionRadius; }
}
```

The code for the **HandlePlayerInput** method is getting pretty long, and I hate long methods. What I did before implementing the collision of the player's sprite and an NPC was create a new method called **HandlePlayerMovement** and I call this method from the **HandlePlayerInput** method. I also created methods for checking if the player is in attack range called **CheckAttackRadius** and

checking for dialog called **CheckSpeakingRadius**. I will explain the code for the methods after you have read the code for them. This is the new code for the methods.

```
private void HandlePlayerInput(GameTime gameTime)
{
    player.Update(gameTime);
    if (!inDialog)
    {
        foreach (NPC npc in npcs)
            npc.Update(gameTime);

        if (CheckAttackRadius(gameTime))
            return;

        if (CheckKey(Keys.Space))
            CheckSpeakingRadius();
    }
    if (!inDialog)
        HandlePlayerMovement();
}

private void HandlePlayerMovement()
{
    Vector2 motion = new Vector2();
    playerSprite.IsAnimating = true;
    if (newState.IsKeyDown(Keys.Up) || newState.IsKeyDown(Keys.NumPad8))
    {
        motion.Y--;
        playerSprite.CurrentAnimation = AnimationKey.Up;
    }
    if (newState.IsKeyDown(Keys.Down) || newState.IsKeyDown(Keys.NumPad2))
    {
        motion.Y++;
        playerSprite.CurrentAnimation = AnimationKey.Down;
    }
    if (newState.IsKeyDown(Keys.Right) || newState.IsKeyDown(Keys.NumPad6))
    {
        motion.X++;
        playerSprite.CurrentAnimation = AnimationKey.Right;
    }
    if (newState.IsKeyDown(Keys.Left) || newState.IsKeyDown(Keys.NumPad4))
    {
        motion.X--;
        playerSprite.CurrentAnimation = AnimationKey.Left;
    }
    if (newState.IsKeyDown(Keys.NumPad9))
    {
        motion.X++;
        motion.Y--;
        playerSprite.CurrentAnimation = AnimationKey.Right;
    }
    if (newState.IsKeyDown(Keys.NumPad3))
    {
        motion.X++;
        motion.Y++;
        playerSprite.CurrentAnimation = AnimationKey.Right;
    }
    if (newState.IsKeyDown(Keys.NumPad1))
    {
```

```
            motion.X--;
            motion.Y++;
            playerSprite.CurrentAnimation = AnimationKey.Left;
        }
        if (newState.IsKeyDown(Keys.NumPad7))
        {
            motion.X--;
            motion.Y--;
            playerSprite.CurrentAnimation = AnimationKey.Left;
        }
        if (motion != Vector2.Zero)
        {
            motion.Normalize();
            motion *= playerSprite.Speed;
            foreach (NPC npc in npcs)
            {
                float distance = Vector2.Distance(
                    npc.Origin,
                    playerSprite.Origin + motion);
                if (distance <= npc.CollisionRadius)
                    return;
            }
            if (!actionScreen.CheckUnWalkableTile(playerSprite.Bounds, motion))
                playerSprite.Position += motion;
        }
        else
            playerSprite.IsAnimating = false;

        playerSprite.LockToMap();
        camera.LockToSprite(playerSprite);
        camera.LockCamera();
    }

    private void CheckSpeakingRadius()
    {
        foreach (NPC npc in npcs)
        {
            float distance = Vector2.Distance(
                npc.Origin,
                playerSprite.Origin);
            if (distance < npc.SpeakingRadius)
            {
                dialog.Show();
                npc.StartDialog(npc.DialogName);
                inDialog = true;
                break;
            }
        }
    }

    private bool CheckAttackRadius(GameTime gameTime)
    {
        foreach (Monster monster in monsters)
        {
            if (!monster.InCombat)
            {
                monster.Update(gameTime);
                float distance = Vector2.Distance(
                    monster.Origin,
```

```
                playerSprite.Origin);
            if (distance < monster.AttackRadius)
            {
                activeScreen.Hide();
                activeScreen = combatScreen;
                combatScreen.Begin(
                    Content.Load<Texture2D>(@"Backgrounds\combatbackground"),
                    player,
                    monster);
                activeScreen.Show();
                return true;
            }
        }
    }
    return false;
}
```

What the **HandlePlayerInput** method does is first call the **Update** method of **player** to update the **PlayerComponent**. If the game is not in dialog it calls the **Update** method for each of the **NPCs** in the game. There is an if statement that checks the return value of **CheckAttackRadius**. The **CheckAttackRadius** method requires a parameter of type **GameTime** because it will be calling the **Update** method of the **monsters** in the game. The method will return true if the player moves within attack range of the monster. Otherwise the method will return false. I will get to the code of the method shortly. The reason I went with the method return true if there was a combat and false otherwise is if you remember from a previous tutorial on the **CombatScreen** of the game if you don't return from the **HandlePlayerInput** method the sprite will still be animating and possibly be facing the wrong direction. If the **CheckAttackRadius** method returns true I return from the method. I then check to see if the space bar has been pressed and if it has I call the **CheckSpeakingRadius** method. There is then one last if statement that checks to make sure the game is not in dialog mode and will then call the **HandlePlayerMovement** method.

The **HandlePlayerMovement** method is where I apply the logic to keep the player from walking through **NPC**s on the map. The code is the same until you get to the part where I check to make sure that the **motion** vector is not the zero vector. It is inside that if statement where I handled checking for a collision between the player and any **NPCs** on the map. After setting the **motion** vector to be the value of the **motion** vector normalized and multiplying it by the speed of the sprite there is a foreach loop where I loop through all of the **NPCs**. I then check the distance of the origins of the player's sprite and the **NPC**'s sprite using the **Distance** method of **Vector2**. I then check to see if the distance between them is less than the **CollisionRadius** of the **NPC**. If it is less than that the movement of the sprite is canceled by return from the method.

The **CheckSpeakingRadius** method is just the code from the old **HandlePlayerInput** method. It just loops through all of the **NPCs** on the map and checks to see it the player is within speaking radius of the **NPC**. If this condition is met it breaks out of the loop after starting the dialog with the **NPC**.

The code for the **CheckAttackRadius** method is also close to the code that was in the old **HandlePlayerInput** method. The difference here is that if there is combat that should take place instead of just returning to exit the **HandlePlayerInput** method the method will return true to let the **HandlePlayerInput** method know that there was a combat and it should exit that method. So when I check for the collision between monster and player I return true if there is one so that the method will exit. At the end of the method I just return false because there was no collision detected.

I also made a couple other changes to the game as well. What I did shorten the **LoadContent** method. I also made a few changes so when the player chooses to exit the game from the game screen instead of exiting the game entirely the player is taken to the menu, as happens with most games. What I did is move all of the method calls to load the screens of the game into a method of their own called **LoadGameScreens.** I also rearranged the order that I load in the methods. I keep saying that the order in which you render things in two dimensions is important. So the order in which you load in the screens for the game is also important. The reason is that you want to layer the screens so that some screens will be drawn on top of other screens. You for example want the pop up screens to be drawn on top of the screen that uses them. You also want the start screen to be drawn over top of all screens other than the screens that ask if the player really wants to exit. This is the code for the new **LoadContent** method and the **LoadGameScreens** method.

```
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    tileSpriteBatch = new SpriteBatch(GraphicsDevice);

    Services.AddService(typeof(SpriteBatch), spriteBatch);
    Services.AddService(typeof(ContentManager), Content);

    dialog = new DialogComponent(this);
    Components.Add(dialog);

    normalFont = Content.Load<SpriteFont>("normal");

    LoadGameScreens();

    CreateAnimations();

    spriteTextures = new Texture2D[assetNames.Length];
    for (int i = 0; i < assetNames.Length; i++)
        spriteTextures[i] = Content.Load<Texture2D>(assetNames[i]);

    script = ReadScript(@"Content\script1.script");

    LoadPlayerSprites();
    CreatePlayerAnimations();

    CreateNPCS();
    CreateMonsters();
}

private void LoadGameScreens()
{
    LoadCreatePCScreen();
    LoadGenderPopUpScreen();
    LoadClassPopUpScreen();
    LoadDifficultyPopUpScreen();
    LoadNameInputScreen();

    LoadHelpScreen();
    LoadCreditScreen();
    LoadIntroScreen();

    LoadActionScreen();
```

```
    LoadViewCharacterScreen();
    LoadCombatScreen();

    LoadStartScreen();
    LoadQuitPopUpScreen();
    LoadQuitActionScreen();

    creditScreen.Hide();
    startScreen.Hide();
    helpScreen.Hide();
    createPCScreen.Hide();

    activeScreen = introScreen;
    activeScreen.Show();
}
```

The last thing I want to do is actually implement returning to the main menu when the player chooses to exit the game. This was done in the **HandleQuitActionScreenInput** method. What I did was instead of just calling **this.Exit** if the player choose the first option, **Yes**, is call **activeScreen.Hide** to hide the current screen. I set the active screen to be the start screen and call **activeScreen.Show** to display the screen. This is the code for the method.

```
private void HandleQuitActionScreenInput()
{
    if (CheckKey(Keys.Enter) || CheckKey(Keys.Space))
    {
        switch (quitActionScreen.SelectedIndex)
        {
            case 0:
                activeScreen.Hide();
                activeScreen = startScreen;
                activeScreen.Show();
                break;
            case 1:
                activeScreen.Hide();
                activeScreen = actionScreen;
                activeScreen.Show();
                break;
        }
    }
}
```

I am planning on adding in more and more functionality that you would find in a complete role playing game. What I am thinking of adding next is being able to pick up chests on the game screen and adding in a few classes for different types of items. I encourage you to keep either visiting my site http://xna.jtmbooks.com or my blog, http://xna-rpg.blogspot.com for the latest news on my tutorials.