Creating a Role Playing Game with XNA Game Studio 3.0
Part 42
Adding the Combat Screen

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/New2DRPG41.zip You can download the graphics from this link: Graphics.zip

For this tutorial you are going to need a background image for the combat screen. I made a graphic that I made a screen that you can use for this. Instead of having you download a big file with all of the graphics I will make this available in a single zip file. You can find the graphic at http://xna.jtmbooks.com/Downloads/combatscreen.zip Download the file, decompress it and add the **grasscombatscreen.png** file to the **Backgrounds** folder in the **Content** folder of the game. I will be making other combat screens that you will be able to use that will have different backgrounds. For example, if the player is in a cave it wouldn't make sense for the combat to take place on a grass field, would it?

Before I get to the combat there is one quick change that I want to make to the game. It has to do with the way I create the **List<Animation>** for the sprites. Since the sprites all use a clone of the same **List<Animation>** I want to just create one **List<Animation>** for the sprites at the class level and add a method that will return a cloned **List<Animation>**. The first thing that you will want to do is add the following field to your **Game1** class near the top where you have all your other fields.

```
List<Animation> animations = new List<Animation>();
```

Now I will create two method: **CreateAnimations** and **CloneAnimations**. As you have probably guessed **CreateAnimatoins** will create the animations and **CloneAnimations** will clone the animations. I don't think that at this point I really need to explain the code in depth, it should be pretty familiar. The first method just creates the animations like you have seen before. The second creates a **List<Animation>** to hold the animations, clones all the animations and returns them. This is the code for those two method.

```
private void CreateAnimations()
{
    Animation tempAnimation = new Animation(2, 64, 64, 0, 0);
    animations.Add(tempAnimation);

    tempAnimation = new Animation(2, 64, 64, 128, 0);
    animations.Add(tempAnimation);

    tempAnimation = new Animation(2, 64, 64, 256, 0);
    animations.Add(tempAnimation);

    tempAnimation = new Animation(2, 64, 64, 384, 0);
    animations.Add(tempAnimation);
}
```

```
private List<Animation> CloneAnimations()
{
    List<Animation> newAnimation = new List<Animation>();

    foreach (Animation a in animations)
    {
        Animation clonedAnimation = (Animation)a.Clone();
        newAnimation.Add(clonedAnimation);
    }

    return newAnimation;
}
```

Now it is time to implement these two methods into the game. The first thing you will want to do is change the **LoadContent** method. What you will want to do is in the **LoadContent** method is call the **CreateAnimations** method. Then when you create the **AnimatedSprite** for the player use the **CloneAnimations** method to create a clone of the animations. Also, add in a call to two methods that I will be adding to the game. The first is **LoadCombatScreen** and it will load in the **CombatScreen** I placed it with the other load methods. The second method is **CreateMonsters** and this will create a couple monsters on the map for the player to interact with. This is the code for the new **LoadContent** 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");
    LoadCreatePCScreen();
    LoadStartScreen();
    LoadHelpScreen();
    LoadActionScreen();
    LoadQuitPopUpScreen();
    LoadGenderPopUpScreen();
    LoadClassPopUpScreen();
    LoadDifficultyPopUpScreen();
    LoadNameInputScreen();
    LoadCreditScreen();
    LoadIntroScreen();
    LoadViewCharacterScreen();
    LoadQuitActionScreen();
    LoadCombatScreen();

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

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

```
    CreateAnimations();

    Texture2D playerSpriteTexture = Content.Load<Texture2D>(@"Sprites\amg1large");

    playerSprite = new AnimatedSprite(this,
        playerSpriteTexture,
        CloneAnimations());

    playerSprite.CurrentAnimation = AnimationKey.Down;
    playerSprite.IsAnimating = true;

    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");
    CreateNPCS();
    CreateMonsters();
}
```

The last thing to do is change the **CreateNPCS** method to use the **CloneAnimation** method. What you will do is remove all the code that had to do with creating and cloning animations and where you create the new **NPC** object use the **CloneAnimations** method to create a clone of the animations. This is the new code for the **CreateNPCS** method.

```
private void CreateNPCS()
{
    for (int i = 0; i < assetNames.Length; i++)
    {
        npcs.Add(
            new NPC(this,
                spriteTextures[i],
                CloneAnimations(),
                dialog,
                script));

        npcs[i].DialogName = dialogNames[i];
        npcs[i].IsAnimating = true;
        int direction = random.Next(0, 4);

        switch (direction)
        {
            case 0:
                npcs[i].CurrentAnimation = AnimationKey.Up;
                break;
            case 1:
                npcs[i].CurrentAnimation = AnimationKey.Down;
                break;
            case 2:
                npcs[i].CurrentAnimation = AnimationKey.Left;
                break;
            case 3:
                npcs[i].CurrentAnimation = AnimationKey.Right;
                break;
        }

        Vector2 position = new Vector2();

        position.X = TileEngine.TileWidth * random.Next(1, 10);
```

```
            position.Y = TileEngine.TileHeight * random.Next(1, 10);

            npcs[i].Position = position;
        }
    }
```

With that out of the way it is now time to get started on adding combat to the game. To do this I will be adding two new classes to the game. The first class will hold the sprites for the monsters the player will fight called **Monster** that will inherit from the **AnimatedSprite** class. The second will be a **GameScreen** called **CombatScreen**. This will be the screen where the game takes place. Before I get to these two new classes there is one thing that I need to do. I will be passing a **PlayerComponent** to the **CombatScreen** for updating and drawing the player's character and sprite.

I will need to save the position of the player's sprite and the current animation when they enter into the combat screen. Then when the combat is over I need to return them to their old values. The reason is when the player enters the combat screen I will be setting the position of the player on the screen and changing the player's current animation. If I don't save these values and then restore them when the combat ends they will have the values from the player on the combat screen and you don't want that to happen. There are other items that I will need to be able to get and set in the **PlayerComponent** and I made quite a few changes to the class. I will also need a second **SpriteBatch** objects in the **PlayerComponent**. The reason is that the **SpriteBatch** object the draws the **AnimatedSprite** is tied to the tile engine. If I try and draw the sprite with that **SpriteBatch** object it won't appear in the right place on the combat screen if it appears at all. I also need to make an addition to the **AnimatedSprite** class to be able to draw the player's sprite. For that I will add a get only property to get the current rectangle of the **AnimatedSprite** class. Add the following property to the **AnimatedSprite** class.

```
public Rectangle CurrentRectangle
{
    get
    {
        return animations[(int)currentAnimation].CurrentFrameRect;
    }
}
```

Since there were a lot of changes to the **PlayerComponent** I will give you the code for the entire class and then I will explain the changes. This is the code for the latest version of the **PlayerComponent**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;
using New2DRPG.SpriteClasses;

namespace New2DRPG.CoreComponents
```

```
{
    class PlayerComponent : Microsoft.Xna.Framework.DrawableGameComponent
    {
        AnimatedSprite sprite;
        PlayerCharacter playerCharacter;
        Game game;
        bool inCombat = false;
        SpriteBatch combatSpriteBatch;

        public PlayerComponent(Game game,
            AnimatedSprite sprite,
            PlayerCharacter playerCharacter)
            : base(game)
        {
            this.sprite = sprite;
            this.playerCharacter = playerCharacter;
            this.game = game;
            combatSpriteBatch =
                (SpriteBatch)Game.Services.GetService(typeof(SpriteBatch));
        }

        public Vector2 Position
        {
            get { return sprite.Position; }
            set { sprite.Position = value; }
        }

        public AnimationKey Animation
        {
            get { return sprite.CurrentAnimation; }
            set { sprite.CurrentAnimation = value; }
        }

        public bool IsAnimating
        {
            get { return sprite.IsAnimating; }
            set { sprite.IsAnimating = value; }
        }

        public bool InCombat
        {
            get { return inCombat; }
            set { inCombat = value; }
        }

        public int SpriteWidth
        {
            get { return sprite.Width; }
        }

        public int SpriteHeight
        {
            get { return sprite.Height; }
        }

        public override void Initialize()
        {
            base.Initialize();
        }
```

```
        public override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
            playerCharacter.Update(gameTime);
            sprite.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);
            playerCharacter.Draw(gameTime);
            if (!inCombat)
            {
                sprite.Draw(gameTime);
            }
            else
            {
                combatSpriteBatch.Draw(
                    sprite.Texture,
                    sprite.Position,
                    sprite.CurrentRectangle,
                    Color.White);
            }
        }

        public void Show()
        {
            Enabled = true;
            Visible = true;
        }

        public void Hide()
        {
            Enabled = false;
            Visible = false;
        }

    }
}
```

I added two new fields to the class: a bool and a **SpriteBatch**. The bool will be used to tell if the player is in combat and the **SpriteBatch** will be used to draw the sprite if the player is in combat. I get the **SpriteBatch** that was registered with the list of services of the game in the **LoadContent** method like in other classes that I needed it.

There are six new properties in the class. The first one, **Position**, is a get and set property that will be used to get and set the position of the sprite. The second one, **Animation**, will be used to get and set the current animation of the sprite. **IsAnimating** will be used to get and set if the sprite is animating. **InCombat** will be used to get and set if the player is in combat. The last two properties, **SpriteWidth** and **SpriteHeight** are get only properties to get the height and width of the sprite.

The last change is in the **Draw** method. What I did in the **Draw** method was first call **base.Draw** to draw all parent components. I then call the **Draw** method of the **PlayerCharacter** class to draw the hit points and magic points of the character. I then check to see if the player is not in

combat. If the player is not in combat I can safely draw the sprite using the **Draw** method of the **AnimatedSprite** class. If the player is in combat I need to use the **Draw** method of the **combatSpriteBatch** object to draw the sprite. The overload that I used to draw the sprite requires as parameters: the **Texture2D** of the object, a **Vector2** for the position of the object, the source **Rectangle** of the object in the **Texture2D**, and the tint color.

Since the **CombatScreen** uses the **Monster** class I will have to add the **Monster** class first. Like I said earlier, the **Monster** class will inherit from the **AnimatedSprite** class. Right click the **SpriteClasses** folder in the solution explorer select **Add** and then **Class**. Name the new class **Monster**. This is the code for the **Monster** class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace New2DRPG.SpriteClasses
{
    class Monster : AnimatedSprite
    {
        float attackRadius;
        bool inCombat = false;
        SpriteBatch combatSpriteBatch;

        public Monster(Game game,
            Texture2D texture,
            List<Animation> animations)
            : base(game, texture, animations)
        {
            attackRadius = 80f;
            combatSpriteBatch =
                (SpriteBatch)Game.Services.GetService(typeof(SpriteBatch));
        }

        public float AttackRadius
        {
            get { return attackRadius; }
        }

        public bool InCombat
        {
            get { return inCombat; }
            set { inCombat = value; }
        }

        public override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            if (!inCombat)
            {
                base.Draw(gameTime);
```

```
            }
            else
            {
                combatSpriteBatch.Draw(
                    texture,
                    Position,
                    CurrentRectangle,
                    Color.White);
            }
        }
    }
}
```

The first thing I needed to do was add in using statements for the **XNA** framework and the **Graphics** class of the **XNA** framework. As I mentioned this class inherits from **AnimatedSprite** because it will be animating. I added in three new fields to this class. Instead of determining if the player is within speaking range of the NPC I decided to see if they are with in the attacking range of the monster. There is a bool field that tells if the monster is in combat or not. There is also a **SpriteBatch** object called **combatSpriteBatch**. As I mentioned the reason for this one is the **SpriteBatch** object that is related to the **Sprite** classes is tied to the tile engine. If I tried to draw the monster on the combat screen using that **SpriteBatch** object it would appear in relation to the map not the screen.

The constructor just sets the **attackRadius** and **combatSpriteBatch** fields. The two properties that I added to the class are **AttackRadius** which will be used to determine if the player is withing the attacking radius. The other property **InCombat** is used to get and set if the sprite is in combat.

There is an override of the **Update** method that at the moment doesn't do much. It just calls **base.Update** which calls the **Update** method of the base class. Later I will be adding more functionality to the **Update** method so I thought it would be a good time to add this in.

There is also an override of the **Draw** method. Inside the **Draw** method I check to see if the **inCombat** field is false. If it is false the monster isn't in combat mode and I can just call **base.Draw** which will call the **Draw** method of the base class. If the monster is in combat I use the **combatSpriteBatch** field to draw the sprite. The override of the draw method that I used takes the **Texture2D** of the sprite, a **Vector2** for the position of the sprite, the source **Rectangle** of the sprite in the sprite sheet and the tint color of the sprite.

Now that we have the **Monster** class we can actually create the monsters and place them on the map. Earlier when I gave you the code for the **LoadContent** method I included a call to **CreateMonsters**, a method that I hadn't shown you the code for because I needed the **Monster** class to write it. Now it is time to switch back the the **Game1** class and add in the **CreateMonsters** class. I placed mine near the **CreateNPCS** method because they are related.You will also need to add a field to hold the monsters. Up with the rest of the fields for the class add the following field. This is the code for the **CreateMonsters** method as well.

```
List<Monster> monsters = new List<Monster>();

private void CreateMonsters()
{
    for (int i = 0; i < 2; i++)
    {
        Monster monster = new Monster(
            this,
```

```
            Content.Load<Texture2D>(@"Sprites\spd1"),
            CloneAnimations());

        int direction = random.Next(0, 4);

        switch (direction)
        {
            case 0:
                monster.CurrentAnimation = AnimationKey.Up;
                break;
            case 1:
                monster.CurrentAnimation = AnimationKey.Down;
                break;
            case 2:
                monster.CurrentAnimation = AnimationKey.Left;
                break;
            case 3:
                monster.CurrentAnimation = AnimationKey.Right;
                break;
        }

        Vector2 position = new Vector2();

        position.X = TileEngine.TileWidth * random.Next(1, 10);
        position.Y = TileEngine.TileHeight * random.Next(1, 10);

        monster.IsAnimating = true;
        monster.Position = position;

        monsters.Add(monster);
    }
}
```

The code should look familiar to you because it was copied and pasted from the **CreateNPCS** method but modified to work with the **Monster** class instead of the **NPC** class. There is a for loop that will create a couple of monsters to add to the list of monsters. Inside the for loop I create a new instance of the **Monster** class passing in the reference to the current game object using the this keyword, I also passed in a **Texture2D** that was called **spd1** which is a red spider and I returned a clone of the list of animations using the **CloneAnimation** method. If you want to use a sprite other than the one I used in your game you will just need to change the asset name of the **Texture2D**. I then set a random direction for the sprite to be facing, set its position on the map, set that it is animating and add it to the list of monsters.

Now I have all of the components needed and ready to move on to the **CombatScreen** class. This class will inherit from the **GameScreen** class and will be loaded and the input handled like other **GameScreen** classes. Right click the **Screens** folder and add a new class called **CombatScreen**. This is the code for the **CombatScreen** class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using New2DRPG.CoreComponents;
using New2DRPG.SpriteClasses;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

```
namespace New2DRPG
{
    class CombatScreen : GameScreen
    {
        PlayerComponent player;
        Monster monster;
        BackgroundComponent background;
        Vector2 oldPosition;
        AnimationKey oldAnimation;
        int screenHeight;
        int screenWidth;

        public CombatScreen(Game game)
            : base(game)
        {
            screenHeight = Game.Window.ClientBounds.Height;
            screenWidth = Game.Window.ClientBounds.Width;
        }

        public void Begin(Texture2D image, PlayerComponent player, Monster monster)
        {
            this.player = player;
            this.monster = monster;

            background = new BackgroundComponent(game, image, true);
            background.Enabled = true;
            background.Visible = true;
            childComponents.Add(background);

            oldPosition = player.Position;
            oldAnimation = player.Animation;

            player.Position = new Vector2(
                25,
                (screenHeight - 100 - player.SpriteHeight) / 2);

            monster.Position = new Vector2(
                screenWidth - 25 - monster.Width,
                (screenHeight - 100 - monster.Height) / 2);

            monster.CurrentAnimation = AnimationKey.Left;
            monster.IsAnimating = false;
            monster.InCombat = true;

            player.Animation = AnimationKey.Right;
            player.IsAnimating = false;
            player.InCombat = true;
        }

        public void End()
        {
            childComponents.Remove(background);
            player.InCombat = false;
            player.Position = oldPosition;
            player.Animation = oldAnimation;
        }

        public override void Update(GameTime gameTime)
```

```
        {
            base.Update(gameTime);
            player.Update(gameTime);
            monster.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);
            player.Draw(gameTime);
            monster.Draw(gameTime);
        }
    }
}
```

There are the familiar using statements for the XNA framework and the XNA Graphics classes. There are also using statements for the **CoreComponents** and **SpriteClasses** namespaces. I also changed the namespace to be just **New2DRPG**. Like I already mentioned this class inherits from the **GameScreen** class. There several fields in this class. There is a **PlayerComponent** field for the player and there is also a **Monster** field for the monster the player is fighting. The next one you might find confusing. It is a **BackgroundComponent**. Usually when I use **BackgroundComponents** I just add them to the list of components. The reason I added this one is that the backgrounds for the combat are going to vary according to where the player is fighting the monster. It wouldn't make sense for the player to be fighting on a grass field in a dungeon, would it? So I will be handling the background for the combat a little differently than in other screens. There are two fields to hold the old position and animation of the player so that when control is passed back to the action screen the player will be in the same place and facing the same direction. Finally there are fields for the width and the height of the screen.

The constructor of the class only takes one parameter, the **Game** object. It then sets the **screenHeight** and **screenWidth** fields to the height and width of the screen.

Next there is a method called **Begin** that has as its parameters: a **Texture2D** called **image**, a **PlayerComponent** called **player**, and a **Monster** called **monster**. The **image** parameter will be used to set the background image of the combat. The other parameters are for the player and the monster the player will be fighting. I will eventually add in the ability to have more than one monster for the player to fight at one time.

The method then sets the **player** and **monster** fields of the class using the objects passed in. It then creates a new **BackgroundComponent** passing in true for the **fill** parameter so that it will fill the screen. It then sets the **Visible** and **Enabled** properties of the **BackgroundComponent** to true so that it will be drawn and updated. Next it does something that I haven't done much with the screens I've created so far. It adds it to the list of child components of the **GameScreen** class. The reason is that when I'm done with the combat I will remove it from the list of child components and when I start the next combat the list of child components will be empty. This will save memory in the long run as well as time rendering the combats.

The method then sets the **oldPosition** and **oldAnimation** fields to the player's current position and animation. It then sets the position of the player and the monster on the screen. To set the position of the player I used 25 pixels for the **X** value of the **Vector2** so it would start on the left hand side of the screen. I then centered it vertically on the combat area. The screen is divided in to two areas. There is

the combat area where the combat will take place and then there is the text area where messages will be written. The text area that I used was 100 pixels high. So to center the player's sprite vertically on the combat area I take the height of the screen, subtract the height of the text area and then subtract the height of the sprite. I can then divide that by two to get the position of the sprite vertically on the screen. I then set the position of the monster on the screen. To find the **X** value for the monster I take the width of the screen, subtract the width of the sprite and then subtract 25 pixels so that there is the same distance between the edges of the screen and the sprites.

I then set the **CurrentAnimation** field of the **monster** to **Animation.Left** so that the monster is facing the player. I set the **IsAnimating** property of the sprite to be false so it will not animate and set the **InCombat** property to true so that the sprite is considered to be in combat. For the player I set the **Animation** field to **Animation.Right** so it is facing the left side of the screen, **IsAnimating** to false so the sprite is not animating, and **InCombat** to true so that the player component knows to draw the sprite using **combatSpriteBatch**.

The **End** method removes the **BackgroundComponent** from the list of child components. The **InCombat** property is set to false and the position of the player to the saved position and the animation of the player to the saved animation.

The **Update** method calls **base.Update** to update other components, **player.Update** to update the **PlayerComponent** and finally **monster.Update** to update the monster. The **Draw** method calls **base.Draw** to draw other components that are visible, **player.Draw** to draw the player and **monster.Draw** to draw the monster.

Almost done, all that is left is to add this to the game. The first thing to do is to add a field to hold the **CombatScreen**. Add the following field where the other screen fields are in the game.

```
CombatScreen combatScreen;
```

The next thing that needs to be done is to load in the screen. When I gave you the code for the **LoadContent** method earlier in the tutorial I made reference to the **LoadCombatScreen** method that I would use to load in the screen. All that the method does is create a new instance of the **CombatScreen** class, add it to the list of components of the game and call the **Hide** method to hide the screen. This is the code for that method.

```
private void LoadCombatScreen()
{
    combatScreen = new CombatScreen(this);
    Components.Add(combatScreen);
    combatScreen.Hide();
}
```

At the moment there is no way to exit the screen once it is shown. To handle that in the **Update** method I will check to see if the **activeScreen** field is set to the **combatScreen**. If it is I will call a new method I wrote called **HandleCombatScreenInput**. In that method I will check to see if the escape key has been pressed. If it has been I will call the **Hide** method of the **activeScreen** to hide the **CombatScreen**. I will call the **End** method of the **CombatScreen** to let it know that combat is over and to set the player's fields back to their original values, set the **activeScreen** field to be the **actionScreen** and finally call the **Show** method of the **activeScreen** to show the proper screen. This is the code for the **Update** method and the **HandleCombatScreenInput** methods.

```
protected override void Update(GameTime gameTime)
{
    newState = Keyboard.GetState();

    if (!dialog.Enabled)
        inDialog = false;

    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    if (activeScreen == startScreen)
    {
        HandleStartScreenInput();
    }
    else if (activeScreen == helpScreen)
    {
        HandleHelpScreenInput();
    }
    else if (activeScreen == createPCScreen)
    {
        HandleCreatePCScreenInput();
    }
    else if (activeScreen == quitPopUpScreen)
    {
        HandleQuitPopUpScreenInput();
    }
    else if (activeScreen == genderPopUpScreen)
    {
        HandleGenderPopUpScreenInput();
    }
    else if (activeScreen == classPopUpScreen)
    {
        HandleClassPopUpScreenInput();
    }
    else if (activeScreen == difficultyPopUpScreen)
    {
        HandleDifficultyPopUpScreenInput();
    }
    else if (activeScreen == nameInputScreen)
    {
        HandleNameInputScreenInput();
    }
    else if (activeScreen == introScreen)
    {
        HandleIntroScreenInput();
    }
    else if (activeScreen == creditScreen)
    {
        HandleCreditScreenInput();
    }
    else if (activeScreen == actionScreen)
    {
        HandleActionScreeenInput();
        HandlePlayerInput(gameTime);
    }
    else if (activeScreen == viewCharacterScreen)
    {
        HandleViewCharacterScreenInput();
```

```
    }
    else if (activeScreen == quitActionScreen)
    {
        HandleQuitActionScreenInput();
    }
    else if (activeScreen == combatScreen)
    {
        HandleCombatScreenInput();
    }
    oldState = newState;

    base.Update(gameTime);
}

private void HandleCombatScreenInput()
{
    if (CheckKey(Keys.Escape))
    {
        activeScreen.Hide();
        combatScreen.End();
        activeScreen = actionScreen;
        activeScreen.Show();
    }
}
```

Two more things to do. The first is you need to update the monsters and check to see if the player comes close enough to them for them to attack the player. The other is to draw the monsters on the action screen. The first I will handle in the **HandlePlayerInput** method and the second I will handle in the **Draw** method. First I will do the **HandlePlayerInput** method. This is the code for that method.

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

```
                }
            }
        }
        if (CheckKey(Keys.Space))
        {
            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;
                }
            }
        }
    }
    if (!inDialog)
    {
        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;

            if (!actionScreen.CheckUnWalkableTile(playerSprite.Bounds, motion))
                playerSprite.Position += motion;

        }
        else
        {
            playerSprite.IsAnimating = false;
        }


        playerSprite.LockToMap();

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

After looping through all of the **NPCs** in a foreach loop I loop through all of the monsters. I then check to see if they are not currently in combat. If they are not I call their **Update** method. I then check the distance between the player and the monster to see if the player is with in the **AttackRadius** property of the monster. If the player is I call the **Hide** method of the **activeScreen** to hide the current screen. I then set **activeScreen** to be **combatScreen**. I then call the **Begin** method of the **CombatScreen** passing in the texture that I had you download at the start of the tutorial, the **player** field and the current monster. I then call the **Show** method of **activeScreen** to display the screen. Finally I use the return method to exit the **HandlePlayerInput** method. This is actually an important step. If you don't the rest of the **HandlePlayerInput** method will execute and since the player was holding down one of the keys to move the player's sprite the sprite will still be animating when you get to the combat screen and it may also be facing the wrong direction, depending on what direction the player was facing at the time combat started. The rest of the method works like before.

All that is left is to actually draw the monsters if they are not in combat. Since when I call the **End** method of the combat screen I don't set **InCombat** for the monster to true I can test to see if that is false to know if I need to draw the monster. In a foreach loop I loop through all of the monsters and if their **InCombat** property is false I draw them. This is the code for the **Draw** method.

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);
    spriteBatch.Begin(SpriteBlendMode.AlphaBlend);
    base.Draw(gameTime);
    if (activeScreen == actionScreen || activeScreen == quitActionScreen)
```

```
    {
        player.Draw(gameTime);
        foreach (NPC sprite in npcs)
            sprite.Draw(gameTime);
        foreach (Monster monster in monsters)
        {
            if (!monster.InCombat)
                monster.Draw(gameTime);
        }
    }
    spriteBatch.End();
}
```

Well, that was a bit of a long tutorial. I will be starting the next part of Eyes of the Dragon shortly. The game is getting more and more functional in terms of XNA but it is not really getting more functional in terms of game mechanics. There are many things that go into a role playing game that have little to do with the graphics. I will be working on those and also doing some stuff with XNA as well. 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.