Creating a Role Playing Game with XNA Game Studio 3.0
Part 24
Adding a Sprite – Part 2

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: Eyes of the Dragon - Version 23 You can download the graphics from this link: Graphics.zip

In today's tutorial I will be continuing with adding the sprite for the player to control. You have noticed that when ever I want to draw something that scrolls with the tile map I am always having to subtract the position of the camera from the object. In today's tutorial I will be removing that restriction by using a transformation matrix.

Many are probably unaware of what a transformation matrix is. Transformation matricies are used all the time in 3D game programming. They are used to scale, rotate and move objects in 3D. Those aren't there only uses but they are a few of them. In order to do this I will have to create a second **SpriteBatch** object. The reason is that when I call the **Begin** method I will be using the transformation matrix to position the camera instead of always subtracting its position. If I was to try and do that with the current **SpriteBatch** object everything in the game will be transformed so if you are on the action screen and you pressed V to bring up the character infromation screen it would be draw relative to the position of the camera.

I will be using a static **SpriteBatch** object and expose it to the rest of the game using a get only property. In your **Game1** class, near where you declare the **Camera** object add the following code to hold the **SpriteBatch** object and the get only property.

```
static SpriteBatch tileSpriteBatch;

public static SpriteBatch TileSpriteBatch
{
    get { return tileSpriteBatch; }
}
```

You can't create a new **SpriteBatch** object before **Initialize** has been called. **Initialize** is called just before **LoadContent** when you are executing an XNA game. That is why the **SpriteBatch** object is created in the **LoadContent** method. To create a **SpriteBatch** object you need to pass it a **GraphicsDevice** object. In XNA games there is a property **GraphicsDevice** that holds the current object used by the game. All we have to do is create **tileSpriteBatch** the same way the other **SpriteBatch** object is created. Add this line to the **LoadContent** method just after the line where the first **SpriteBatch** object is created.

```
tileSpriteBatch = new SpriteBatch(GraphicsDevice);
```

Now we need to create the transformation matrix. I will do this in the **Camera** class because the transformation is created relative to the camera's position. I will also be removing the **speed** field from the **Camera** class because instead of scrolling the map relative to the camera I will be scrolling the map relative to the player's sprite. I will give you the new code and then go over the changes.

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

namespace New2DRPG
{
    public class Camera
    {
        public Vector2 Position;

        public Camera(Vector2 position)
        {
            this.Position = position;
        }

        public Camera()
        {
            this.Position = Vector2.Zero;
        }

        public Matrix TransformMatrix
        {
            get
            {
                return Matrix.CreateTranslation(new Vector3(-Position, 0f));
            }
        }

        public void LockCamera()
        {
            Position.X = MathHelper.Clamp(
                Position.X,
                0,
                TileMap.WidthInPixels - TileEngine.ViewPortWidth);
            Position.Y = MathHelper.Clamp(
                Position.Y,
                0,
                TileMap.HeightInPixels - TileEngine.ViewPortHeight);
        }
    }
}
```

So what I did here was remove everything that is related to the **speed** field. I still have the two constructors for the class. One that takes the position of the camera and the other that sets the position of the camera to **Vector2.Zero** or (0f, 0f).

I added in a property that returns a **Matrix**. The **Matrix** class is a 4 by 4 matrix. Using a 4 by 4 matrix you can preform transformation to objects in 3D space. In this case I will be creating a **translation** matrix the **CreateTranslation** method of the **Matrix**. What a **translation** matrix does is move objects in 3D space. If you preform a translation you are shifting objects. We are working in 2D space but the same idea applies. In 3D space the last coordinate, the Z coordinate, represents depth. If you are looking at your screen you can thing of the Z axis as coming out of the screen at you. By always using 0 for the Z coordinate we are saying that there is no depth involed. To create the translation we need to pass the **CreateTranslation** matrix a **Vector3**. A **Vector3** is like a **Vector2** it just

has a third coordinate involved, the **Z** coordinate. There is an overload of the constructor for **Vector3** that takes a **Vector2** as a parameter plus the Z coordinate so I passed in **-Position** and 0f. Why did I pass in **-Position** and not just **Position**? It is because when we were adjusting for the camera we were always subtracting the position of the camera from what we were drawing.

To use the translation matrix and the new **SpriteBatch** object I need to make a change to the **TileMap** constructor. Instead of using **GetService** to get the **SpriteBatch** object I can use the static property **TileSpriteBatch** to get the **SpriteBatch** object. This is the code for the new **TileMap** constructor.

```
public TileMap(int tileMapWidth, int tileMapHeight, Tileset tileset, Game game)
    : base(game)
{
    spriteBatch = Game1.TileSpriteBatch;
    Content =
        (ContentManager)Game.Services.GetService(typeof(ContentManager));

    this.tileset = tileset;

    LoadContent();

    mapWidth = tileMapWidth;
    mapHeight = tileMapHeight;

    TileMapLayer layer = new TileMapLayer(mapWidth, mapHeight);

    for (int x = 0; x < tileMapWidth; x++)
        for (int y = 0; y < tileMapHeight; y++)
            layer.SetTile(x, y, 0);

    tileMapLayers.Add(layer);

    layer = new TileMapLayer(mapWidth, mapHeight);

    for (int x = 0; x < tileMapWidth; x++)
        for (int y = 0; y < tileMapHeight; y++)
        {
            layer.SetTile(x, y, -1);
            if (random.Next(0, 50) < 5)
            {
                layer.SetTile(x, y, tileset.Tiles.Count - 1);
            }
        }

    tileMapLayers.Add(layer);
}
```

Now I have to make a few changes to the **Draw** method in the **TileMapLayer** class to use the new transformation matrix. The first thing is I now have to call **Begin** and **End** on the **SpriteBatch** object because we are not using the same **SpriteBatch** object we were using before. The overload for the **Begin** method I used takes 4 parameters: **SpriteBlendMode**, **SpriteSortMode**, **SaveStateMode** and a **Matrix**. The first one you are familiar with. It determines how sprites will be blended. Since we will be drawing transparent sprites I chose **SpriteBlendMode.AplhaBlend**. The next parameter might be new to many. It describes how the sprites drawn will be sorted. For this case I decided to use the **SpriteSortMode.Deferred** option. What this option does is that sprites will not be drawn to the screen

until the **End** method is called. There are other options but for what I am rendering there was no need to consider other options. The next parameter is a little complicated. It allows you to save the state of the graphics device before and after an effect is called. I will not be doing this so I used **SaveStateMode.None**. The last parameter allows you to choose a **Matrix** to transform what is rendered. This is where I apply the **TransformMatrix** of the **Camera** class. I also needed to change how I calculated the **X** and **Y** values of the destination rectangles. Because I used the transformation matrix I no longer have to subtract the position of the camera. The last change was I needed to add in a call to the **End** method. This is the updated **Draw** method for the **TileMapLayer** class.

```
public void Draw(SpriteBatch sBatch, Texture2D texture, Tileset tileset)
{
    Point cameraPoint = VectorToCell(Game1.Camera.Position);
    Point viewPoint = VectorToCell(Game1.Camera.Position +
                        TileEngine.ViewPortVector);
    Point min = new Point();
    Point max = new Point();
    min.X = cameraPoint.X;
    min.Y = cameraPoint.Y;
    max.X = (int)Math.Min(viewPoint.X, map.GetLength(1));
    max.Y = (int)Math.Min(viewPoint.Y, map.GetLength(0));

    Rectangle tileRectangle = new Rectangle(
            0,
            0,
            TileEngine.TileWidth,
            TileEngine.TileHeight);

    sBatch.Begin(SpriteBlendMode.AlphaBlend,
        SpriteSortMode.Deferred,
        SaveStateMode.None,
        Game1.Camera.TransformMatrix);
    for (int x = min.X; x < max.X; x++)
    {
        for (int y = min.Y; y < max.Y; y++)
        {
            if (map[y, x] != -1)
            {
                tileRectangle.X = x * TileEngine.TileWidth;
                tileRectangle.Y = y * TileEngine.TileHeight;

                sBatch.Draw(texture,
                    tileRectangle,
                    tileset.Tiles[map[y, x]],
                    Color.White);
            }
        }
    }
    sBatch.End();
}
```

Because the sprite for the player is related to the camera class I need to change the way that I draw the sprite. I need to use the **SpriteBatch** object to render the sprite and I need to now call **Begin** and **End** on the **SpriteBatch** object the same way I did for the **TileMapLayer** class. I also no longer need to calculate the position of the sprite by subtracting the position of the camera. So in the constructor of the **AnimatedSprite** class I get the new **SpriteBatch object using the TileSpriteBatch** property of the **Game1 class. In the Draw** method of the **AnimatedSprite** class I call **Begin** and **End**

of the **SpriteBatch** object like I did in the **TileMapLayer**. This is the code for the new constructor of the **AnimatedSprite** class as well as the **Draw** method.

```
public AnimatedSprite(Game game, Texture2D texture, List<Animation> animations)
    : base(game, texture)
{
    spriteBatch = Game1.TileSpriteBatch;
    this.animations = animations;
    currentAnimation = AnimationKey.Down;
    isAnimating = false;
}

public override void Draw(GameTime gameTime)
{
    base.Draw(gameTime);

    spriteBatch.Begin(SpriteBlendMode.AlphaBlend,
        SpriteSortMode.Deferred,
        SaveStateMode.None,
        Game1.Camera.TransformMatrix);

    spriteBatch.Draw(
        texture,
        Position,
        animations[(int)currentAnimation].CurrentFrameRect,
        Color.White);

    spriteBatch.End();
}
```

Okay, that is all well and good but at the moment the game will not compile or run because I removed the **speed** field and **Speed** property from the **Camera** class as the camera is no longer responsible for scrolling the map. What I will work on now is getting the sprite to move according to the keyboard input and have the sprite animate only when the player is moving. This will take place in the **HandleActionScreenInput** method. The first thing I will need to do is make a change to the **Sprite** class. I could have done an override of the **Position** property in the **AnimatedSprite** class but I decided instead to change the **Sprite** class. I just added in a simple set part for the class. This is the new **Position** property for the **Sprite** class.

```
public virtual Vector2 Position
{
    get { return position; }
    set { position = value; }
}
```

Now I will do the work on the **HandleActionScreenInput** method. There are quite a few changes so I will give you the code and then explain it. The code for the **HandleActionScreenInput** method follows.

```
private void HandleActionScreeenInput()
{
    if (CheckKey(Keys.Escape))
    {
        this.Exit();
    }
```

```
else if (CheckKey(Keys.V))
{
    playerSprite.IsAnimating = false;
    activeScreen.Enabled = false;
    activeScreen = viewCharacterScreen;
    viewCharacterScreen.SetPlayerCharacter(playerCharacter);
    activeScreen.Show();
}
else
{
    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();
```

```
            }
            else
            {
                playerSprite.IsAnimating = false;
            }
            playerSprite.Position += motion * playerSprite.Speed;

            camera.LockCamera();
        }
    }
```

There are many ways to determine how to determine which animation to choose to draw. The way I decided to do it, for this tutorial, is that if the sprite is moving diagonally the left and right animations take precendence over the up and down animations. It just looks better in my opinion. In another tutorial I will go over a different method for determining which direction the sprite will animate.

The first thing you will see is that I decided that if the view character screen is visible the sprite will not animate. I thought this was reasonable. So if the V key has been pressed I set the **IsAnimating** property of the **PlayerSprite** object to false.

There were many changes to the way I handle the input for determining if the sprite is moving. The first thing that I do is set the **IsAnimating** property for the **PlayerSprite** object to true. I changed the order of the if statements. The Up and Down direction is handled first. Since the Left and Right are checked after if either is pressed those animations will take precedence over the Up and Down directon. The other if statements are in the same order as before.

Besides just changing the motion vector for each of the key presses I also set the animation. If the Up key or 8 on the keypad are being pressed I set **CurrentAnimation** to **Animation.Up**. If the Down key or the 2 on the keypad are being pressed I set **CurrentAnimation** to **Animation.Down**. Next if the Right key or 6 on the keypad are being pressed I set **CurrentAnimation** to **Animation.Right**. For the Left key or 4 on the keypad I set **CurrentAnimation** to **Animation.Left**. For 9 on the keypad I set **CurrentAnimation** to **Animation.Right**. The same is true for 3 on the keypad. For 1 on the keypad I set **CurrentAnimation** to **Animation.Left** and I do the same if 7 on the keypad is being pressed.

I added in an else to where I check to see if there is any motion for the sprite. If there is no motion the sprite is not moving so I set **IsAnimating** to false. Then instead of adding the motion vector times the speed of the camera to the position of the camera I add the motion vector times the speed of the sprite to the position of the sprite.

If you compile and run the game now the sprite will move around the screen and animate. At the moment though the sprite can walk off the screen and the map doesn't scroll with the sprite. I will get to that in the next tutorail.

I want to add one two more things to this tutorial. I want to add in a couple item sprites to the map. The reason I want to do this is that the items were all rendered using the same method as the tiles. The position of the camera was subtracted to find out where to draw them. I was also asked if there is a way to have a message pop up when the escape key was pressed in the action screen to ask if the player would like to exit the game. That isn't very hard so I thought I would add that in as well.

I will start with the changes to the **ItemSprite** class. All that I had to do was in the constructor get the **TileSpriteBatch** from the **Game1** class and change the **Draw** method to use the **SpriteBatch** object. I also move the call to the **Draw** method after the call to **base.Draw** to make sure they will be rendered after other objects. This is the new **ItemSprite** class.

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

namespace New2DRPG.SpriteClasses
{
    class ItemSprite : Sprite
    {
        public ItemSprite(Game game, Texture2D texture, Vector2 position)
            : base(game, texture)
        {
            spriteBatch = Game1.TileSpriteBatch;
            this.position = position;
        }

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);

            spriteBatch.Begin(SpriteBlendMode.AlphaBlend,
                SpriteSortMode.Deferred,
                SaveStateMode.None,
                Game1.Camera.TransformMatrix);

            spriteBatch.Draw(texture,
                new Rectangle((int)position.X * TileEngine.TileWidth,
                    (int)position.Y * TileEngine.TileHeight,
                    TileEngine.TileWidth,
                    TileEngine.TileHeight),
                    Color.White);
            spriteBatch.End();
        }

    }
}
```

So now I need to add a couple items to draw. What I will do that in the **TileMap** class. The first thing you will need to do is add a using statement to the **TileMap** class for **New2DRPG.SpriteClasses** to be able to use the **ItemSprite** class.

```
using New2DRPG.SpriteClasses;
```

Now you will need a field to hold some **ItemSprites**. I chose to make a **List<ItemSprite>** to hold the items. You will also need a texture for the items. Add these fields to the **TileMap** class.

```
Texture2D itemTexture;
List<ItemSprite> items = new List<ItemSprite>();
```

You will have to load in the texture for the items. You will do that in the **LoadContent** method. This is the new **LoadContent** method.

```
protected override void LoadContent()
{
    texture = Content.Load<Texture2D>(@"TileSets\" + tileset.TextureName);
    itemTexture = Content.Load<Texture2D>(@"Items\chest");
    base.LoadContent();
}
```

Now that you have the texture you can create a few items. I chose to add 2 items to the **List** of items. You will do this in the constructor of the **TileMap** class. At the end of the constructor I added in a for loop. In the loop I create an **ItemSprite** object passing in the **Game** object, **itemTexture** and finally a **Vector2** that will hold, in tiles, where the item will appear on the screen. After creating the item you need to add it to the list of items. This is the updated constructor for the **TileMap** class.

```
public TileMap(int tileMapWidth, int tileMapHeight, Tileset tileset, Game game)
    : base(game)
{
    spriteBatch = Game1.TileSpriteBatch;
    Content =
        (ContentManager)Game.Services.GetService(typeof(ContentManager));

    this.tileset = tileset;

    LoadContent();

    mapWidth = tileMapWidth;
    mapHeight = tileMapHeight;

    TileMapLayer layer = new TileMapLayer(mapWidth, mapHeight);

    for (int x = 0; x < tileMapWidth; x++)
        for (int y = 0; y < tileMapHeight; y++)
            layer.SetTile(x, y, 0);

    tileMapLayers.Add(layer);

    layer = new TileMapLayer(mapWidth, mapHeight);

    for (int x = 0; x < tileMapWidth; x++)
        for (int y = 0; y < tileMapHeight; y++)
        {
            layer.SetTile(x, y, -1);
            if (random.Next(0, 50) < 5)
            {
                layer.SetTile(x, y, tileset.Tiles.Count - 1);
            }
        }

    tileMapLayers.Add(layer);

    for (int i = 0; i < 2; i++)
    {
        ItemSprite item =
            new ItemSprite(game,
```

```
                itemTexture,
                new Vector2(random.Next(0, 8), random.Next(0, 8)));
            items.Add(item);
        }
    }
```

The last thing to do for adding the items to the game is to draw them. That will be done in the **Draw** method. Since the order you render in 2D is important for the items to be visible on the screen is to draw them after you draw the map. This is the code for the **Draw** method.

```
public override void Draw(GameTime gameTime)
{
    foreach (TileMapLayer layer in tileMapLayers)
    {
        layer.Draw(spriteBatch, texture, tileset);
    }

    foreach (ItemSprite item in items)
    {
        item.Draw(gameTime);
    }

    base.Draw(gameTime);
}
```

Adding the new screen to ask if the player wants to quit the game from the action screen will not be hard. I will just create a second **QuitGameScreen** object and when the player presses the Escape key I will display that screen and create a method to handle the input for that screen if that is the active screen. So add the following field to the **Game1** class.

```
QuitGameScreen quitActionScreen;
```

You will have to create and load in the screen. You will do this in the **LoadContent** method. I just created a new method to do this called **LoadQuitActionScreen** and called this method from the **LoadContent** method. The code for the updated **LoadContent** method and **LoadQuitActoinScreen** follows.

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

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

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

    LoadCreatePCScreen();
    LoadStartScreen();
    LoadHelpScreen();
    LoadActionScreen();
    LoadQuitPopUpScreen();
    LoadGenderPopUpScreen();
    LoadClassPopUpScreen();
    LoadDifficultyPopUpScreen();
    LoadNameInputScreen();
```

```
    LoadCreditScreen();
    LoadIntroScreen();
    LoadViewCharacterScreen();
    LoadQuitActionScreen();

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

    activeScreen = introScreen;
    activeScreen.Show();

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

    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);

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

    playerSprite = new AnimatedSprite(this, playerSpriteTexture, animations);
    playerSprite.CurrentAnimation = AnimationKey.Down;
    playerSprite.IsAnimating = true;
}

private void LoadQuitActionScreen()
{
    quitActionScreen = new QuitGameScreen(this);
    Components.Add(quitActionScreen);
    quitActionScreen.Hide();
}
```

The **LoadQuitActionScreen** method creates a new **QuitGameScreen** object. It then adds the new object to the list of game components. Finally it calls the **Hide** method to hide the screen.

The next thing to do is change the **HandleActionScreenInput** method to call the new screen when the Escape key is pressed. To do that you just need to change the if statement where the Escape key has been pressed. This is the new code for **HandleActionScreenInput** method.

```
private void HandleActionScreeenInput()
{
    if (CheckKey(Keys.Escape))
    {
        playerSprite.IsAnimating = false;
        activeScreen.Enabled = false;
        activeScreen = quitActionScreen;
        activeScreen.Show();
    }
    else if (CheckKey(Keys.V))
```

```
{
    playerSprite.IsAnimating = false;
    activeScreen.Enabled = false;
    activeScreen = viewCharacterScreen;
    viewCharacterScreen.SetPlayerCharacter(playerCharacter);
    activeScreen.Show();
}
else
{
    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();
    }
```

```
            else
            {
                playerSprite.IsAnimating = false;
            }
            playerSprite.Position += motion * playerSprite.Speed;

            camera.LockCamera();
        }
    }
```

All that this does is set **activeScreen.Enabled** to false as well as **playerSprite.IsAnimating** to false so the sprite will not animate while the screen is visible. I set **activeScreen.Enabled** to false so the **actionScreen** will still draw but not update. Then I set **activeScreen** to **quitActionScreen** and finally call the **Show** method of the **activeScreen**.

That leaves two things to do to finish this. The first is in the **Update** method do call a new method that I will write called **HandleQuitActionScreenInput** to handle the input for the new screen if it is the active screen. The code for the **Update** method and the **HandleQuitActionScreenInput** method follows.

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

    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();
        }
        else if (activeScreen == viewCharacterScreen)
        {
            HandleViewCharacterScreenInput();
        }
        else if (activeScreen == quitActionScreen)
        {
            HandleQuitActionScreenInput();
        }
        oldState = newState;

        base.Update(gameTime);
    }

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

The **HandleQuitActionScreenInput** method checks to see if the Space or Enter key has been pressed. If it has it uses the **SelectedIndex** property to determine what to do. If the first option is chosen, the Yes option the game exits. If the second option is chosen, the No Option, control is passed back to the action screen.

Well that is it for this tutorial. I am already working on coding the next part of Eyes of the Dragon so 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 these tutorials.