Creating a Role Playing Game with XNA Game Studio 3.0
Part 40
Refactoring - Part 3
NPC Dialogs - Part 1

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/New2DRPG39.zip You can download the graphics from this link: Graphics.zip

In this tutorial I will do a little more refactoring and I will also get started on conversations with NPCs in the game. You will need a need graphic that I create for the conversations with the NPCs in the game. You can download all of the graphics needed for the game in the graphics.zip file referenced above. After you have downloaded the file extract it and add the **conversationbox.png** file to the **GUI** folder in the **Content** folder.

The first thing to do is to create a new folder in the game project to hold classes related to dialogs with NPCs. Right click your game and add a new folder called **DialogClasses**. Now right click that folder then select **Add** and then **New Item**. Make sure that the **XNA Game Studio** node is checked and select **Game Component** and call it **DialogComponent**. This is the code for the **DialogComponent**.

```
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;

namespace New2DRPG.DialogClasses
{
    public class DialogComponent : Microsoft.Xna.Framework.DrawableGameComponent
    {
        SpriteBatch spriteBatch;
        ContentManager Content;
        Texture2D dialogTexture;
        Rectangle position;
```

```
        public DialogComponent(Game game)
            : base(game)
        {
            spriteBatch =
                (SpriteBatch)Game.Services.GetService(typeof(SpriteBatch));
            Content =
                (ContentManager)Game.Services.GetService(typeof(ContentManager));
            LoadContent();
        }

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

        protected override void LoadContent()
        {
            base.LoadContent();
            dialogTexture = Content.Load<Texture2D>(@"GUI\conversationbox");
            position = new Rectangle((1024 - dialogTexture.Width) / 2,
                (768 - dialogTexture.Height) / 2,
                dialogTexture.Width,
                dialogTexture.Height);
        }

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

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);
            spriteBatch.Draw(dialogTexture, position, Color.White);
        }

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

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

Right now the class doesn't do much. Eventually it will be responsible for writing out when the NPC has to say and respond to the player's input. The first thing is that I inherit the class from **DrawableGameComponent** rather than just **GameComponent** because it will be rendering. There are at the moment four fields in the class: a **SpriteBatch** object, a **ContentManager** object, a **Texture2D** and a **Rectangle**. The **Texture2D** and **Rectangle** will be used in drawing the dialog.

The constructor just gets the **SpriteBatch** and **ContentManager** objects that were added to the

list of services for the game and then call the **LoadContent** method. The **LoadContent** method reads in the image for the dialog and then creates a rectangle for drawing the image for the dialog. It centers the dialog in the view port by taking the width of the view port, subtracting the width of the image and then dividing the result by 2. Same idea for the height. You take the height of the view port subtract the height of the image and then divide it by 2. In the **Draw** method I draw the image using the rectangle I created as the destination rectangle. There are also methods to show and hide the dialog.

Now you will want to add a property to the **Sprite** class called **Origin**. This property will be used to find the origin of a sprite. The origin of a sprite is the position of the sprite plus its center. The reason I added it to the **Sprite** class is so that any of the sprite classes you create will have this property available to them. It is a handy property to have. This is the code for the **Origin** property.

```
public Vector2 Origin
{
    get { return position + center; }
}
```

Next I created a new sprite class called **NPC**. Right click the **SpriteClasses** folder in the game and add a new class called **NPC** to the folder. This is the code for the **NPC** class.

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

namespace New2DRPG.SpriteClasses
{
    class NPC : AnimatedSprite
    {
        DialogComponent dialog;
        float speakingRadius;
        bool isTalking;

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

        public bool IsTalking
        {
            get { return isTalking; }
        }

        public float SpeakingRadius
        {
            get { return speakingRadius; }
        }

        public override void Update(GameTime gameTime)
        {
```

```
                base.Update(gameTime);
                if (dialog.Enabled)
                    dialog.Update(gameTime);
            }

            public override void Draw(GameTime gameTime)
            {
                base.Draw(gameTime);
                if (dialog.Enabled)
                    dialog.Draw(gameTime);
            }

            public void StartDialog()
            {
                dialog.Show();
                isTalking = true;
            }

            public void StopDialog()
            {
                dialog.Hide();
                isTalking = false;
            }
        }
    }
```

This class uses part of the XNA framework and it uses a **Texture2D** object so there are using statements for the XNA framework on the Graphics classes in the XNA framework. NPCs will have dialogs attached to them so there is a using statement for the **DialogClasses** namespace.

This class inherits from the **AnimatedSprite** class. There are three fields in this class: **dialog**, which is a **DialogComponent** object that will hold the dialog for the NPC, **speaknigRadius** that will tell if the player is close enough to the NPC to talk to them, and **isTalking** which will hold if the player is currently talking to the sprite. Because the **AnimatedSprite** class requires a **Game** object, **Texture2D** object and a **List<Animation>** the constructor for this class has three parameters that match them. The constructor then sets **speakingRadius** to 80. This measures the distance from the origin of the player to the origin of the NPC. The sprites are all 64 pixels wide. That means that if they were aligned perfectly in adjacent tiles their origins would be 64 pixels apart. I added a little padding to that so the player does not have to be exactly right beside them. Just being close will be good enough. I also create an instance of the **DialogComponent** class and call the **Hide** method of that class. There are two get only properties in this class that return the **speakingRadius** and **isTalking** fields.

There are overrides of the **Update** and **Draw** methods of the **AnimatedSprite** class. The **Update** method calls the **Update** method of the parent class and then if the dialog is active calls the **Update** method of the dialog. Same thing with the **Draw** method. It calls the **Draw** method of the parent and if the dialog is active calls the **Draw** method of the dialog. The **StartDialog** method calls the **Show** method of the dialog and sets **isTalking** to true. The **StopDialog** method calls the **Hide** method of the dialog and sets **isTalking** to false.

I am going to do a little refactoring of the **ActionScreen** class now. What I am going to do is pulling everything out of that class that has to do with the animated sprites. I'm instead go to handle all of that in the **Game1** class. There is also no need to have the **PlayerComponent** in the **ActionScreen** class so I removed that as well. I removed a few other things that weren't being used any more as well.

The code is pretty much the same so I will just give you the new code and not explain it.

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

namespace New2DRPG
{
    class ActionScreen : GameScreen
    {
        Texture2D characterHUDTexture;
        string tilesetName;
        Tileset tileset;
        int viewportWidth;
        int viewportHeight;
        TileMapComponent tileMapComponent;
        TileMap tileMap;

        public ActionScreen(Game game, SpriteFont gameFont, TileMap tileMap, string
tilesetName)
            : base(game)
        {
            this.tilesetName = tilesetName;
            this.tileMap = tileMap;
            LoadContent();
            tileMapComponent = new TileMapComponent(tileMap, tileset, game);
            Components.Add(tileMapComponent);
            tileMapComponent.Show();

            viewportWidth = TileEngine.ViewPortWidth;
            viewportHeight = TileEngine.ViewPortHeight;
        }

        protected override void LoadContent()
        {
            base.LoadContent();
            tileset = Content.Load<Tileset>(@"TileSets\" + tilesetName);
            characterHUDTexture =
Content.Load<Texture2D>(@"Backgrounds\characterhud");
        }

        public bool CheckUnWalkableTile(Rectangle bounds, Vector2 motion)
        {
            Rectangle nextRectangle = new Rectangle(
                bounds.X + (int)motion.X,
                bounds.Y + (int)motion.Y,
                bounds.Width,
                bounds.Height);

            if (motion.Y < 0 && motion.X < 0)
            {
                return tileMap.CheckUpAndLeft(nextRectangle);
```

```
            }
            else if (motion.Y < 0 && motion.X == 0)
            {
                return tileMap.CheckUp(nextRectangle);
            }
            else if (motion.Y < 0 && motion.X > 0)
            {
                return tileMap.CheckUpAndRight(nextRectangle);
            }
            else if (motion.Y == 0 && motion.X < 0)
            {
                return tileMap.CheckLeft(nextRectangle);
            }
            else if (motion.Y == 0 && motion.X > 0)
            {
                return tileMap.CheckRight(nextRectangle);
            }
            else if (motion.Y > 0 && motion.X < 0)
            {
                return tileMap.CheckDownAndLeft(nextRectangle);
            }
            else if (motion.Y > 0 && motion.X == 0)
            {
                return tileMap.CheckDown(nextRectangle);
            }
            return tileMap.CheckDownAndRight(nextRectangle);
        }

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

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);
            if (Visible)
            {
                Vector2 position = new Vector2(0, 0);
                position.Y = viewportHeight;
                spriteBatch.Draw(characterHUDTexture, position, Color.White);
            }
        }

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

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

    }
```

```
}
```

The rest of this tutorial will take place in the **Game1** class. The first thing to do is add some fields to the **Game1** class. They are the fields from the **ActionScreen** class plus a bool called **inDialog**. The last one will be used to tell if the game is in a dialog. I change the **List<AnimatedSprites>** to a **List<NPC>** as well as the name of the field.

```
List<NPC> npcs = new List<NPC>();
string[] assetNames =
    {
        @"Sprites\knt1",
        @"Sprites\nja1",
        @"Sprites\skl1" };
Texture2D[] spriteTextures;
Random random = new Random();
bool inDialog;
```

Now you need to load in the sprites and actually create them. That will be done in the **LoadContent** method. I also add a **CreateNPCS** method to **Game1** that is the same as the **CreateSprites** method from the **ActionScreen** class. This is the code for those two method.

```
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;

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

private void CreateNPCS()
{
    List<Animation> listAnimation = new List<Animation>();

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

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

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

    tempAnimation = new Animation(2, 64, 64, 384, 0);
    listAnimation.Add(tempAnimation);

    for (int i = 0; i < assetNames.Length; i++)
    {
        List<Animation> animations = new List<Animation>();

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

        npcs.Add(
            new NPC(this,
                spriteTextures[i],
                animations));

        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;
    }
}
```

You will need to change two methods: the **CreatePlayerCharacter** method and the **HandleStartScreenInput** method. The reason is that I removed the **PlayerComponent** from the action screen so you no longer need to call the **SetPlayerComponent** method. This is the new code.

```
private void CreatePlayerCharacter()
{
    if (createPCScreen.CharacterClass == CharClass.Fighter)
    {
        playerCharacter = new FighterCharacter(
            createPCScreen.CharacterName,
            createPCScreen.CharacterGender,
            createPCScreen.DifficultyLevel, this);
    }
    if (createPCScreen.CharacterClass == CharClass.Priest)
    {
        playerCharacter = new PriestCharacter(
            createPCScreen.CharacterName,
            createPCScreen.CharacterGender,
            createPCScreen.DifficultyLevel, this);
    }
    if (createPCScreen.CharacterClass == CharClass.Thief)
    {
        playerCharacter = new ThiefCharacter(
            createPCScreen.CharacterName,
            createPCScreen.CharacterGender,
            createPCScreen.DifficultyLevel, this);
    }
    if (createPCScreen.CharacterClass == CharClass.Wizard)
    {
        playerCharacter = new WizardCharacter(
            createPCScreen.CharacterName,
            createPCScreen.CharacterGender,
            createPCScreen.DifficultyLevel, this);
    }
    player = new PlayerComponent(this, playerSprite, playerCharacter);
    player.Hide();
}
```

```
private void HandleStartScreenInput()
{
    if (CheckKey(Keys.Enter) || CheckKey(Keys.Space))
    {
        switch (startScreen.SelectedIndex)
        {
            case 0:
                activeScreen.Hide();
                activeScreen = createPCScreen;
                activeScreen.Show();
                break;
            case 1:
                activeScreen.Hide();
                playerCharacter = new FighterCharacter(
                    "Evander",
                    false,
                    Level.Normal,
                    this);
                player = new PlayerComponent(this, playerSprite, playerCharacter);
                activeScreen = actionScreen;
                player.Show();
                actionScreen.Show();
                break;
            case 2:
                activeScreen.Hide();
                activeScreen = helpScreen;
                activeScreen.Show();
                break;
            case 3:
                activeScreen.Hide();
                activeScreen = creditScreen;
                activeScreen.Show();
                break;
            case 4:
                activeScreen.Enabled = false;
                activeScreen = quitPopUpScreen;
                activeScreen.Show();
                break;
        }
    }
}
```

The next thing to do is change the **Draw** method. In the **Draw** method you only want to draw sprites if the current screen is the **ActionScreen** or if the current screen is the pop up screen asking if the player wants to quit the game. You need to do the rendering after the call to **base.Draw** because it is in that call that all of the components of the game are drawn. 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);
```

```
        }
        spriteBatch.End();
    }
```

I modified the **HandleActionScreenInput** method to only handle if the player presses the **Escape** key to end the game or the **V** key to bring up the screen to view the character. This is the code for that 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();
   }
}
```

What is left is to handle the other input when the current screen is the action screen. What I decided to do is in the **Update** method when I check to see if the current screen is the action screen after calling the **HandleActionScreenInput** method was to call a method called **HandlePlayerInput**. That method requires the **GameTime** object from the **Update** method as it calls the update method of a few objects. This is the code for the two methods. I will explain the **HandlePlayerInput** method after you have read the code.

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

        base.Update(gameTime);
    }

    private void HandlePlayerInput(GameTime gameTime)
    {
        player.Update(gameTime);
        if (!inDialog)
        {
            foreach (NPC npc in npcs)
                npc.Update(gameTime);
        }
        if (CheckKey(Keys.Space))
        {
            if (!inDialog)
            {
                foreach (NPC npc in npcs)
                {
                    float distance = Vector2.Distance(
                        npc.Origin,
                        playerSprite.Origin);
                    if (distance < npc.SpeakingRadius)
                    {
```

```
                npc.StartDialog();
                inDialog = true;
                break;
            }
        }
    }
    else
    {
        inDialog = false;
        foreach (NPC npc in npcs)
        {
            if (npc.IsTalking)
                npc.StopDialog();
        }
    }
}
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();
            motion *= playerSprite.Speed;

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

        }
        else
        {
            playerSprite.IsAnimating = false;
        }

        playerSprite.LockToMap();

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

The first thing the **HandlePlayerInput** method does is first cal the **Update** method of the player component to update the player. If there is no dialog active it updates the NPCs by calling their **Update** method in a foreach loop. It then uses the **CheckKey** method to see if the space bar has been pressed and released. If it has it then checks to see if there is a not dialog currently in progress. If there isn't a dialog in progress it loops through all of the NPCs in the list of NPCs in a foreach loop. What happens next is the method calculates the distance between the origins of both sprites. Fortunately there is a **Distance** method of **Vector2** that will calculate the distance between two vectors. It then checks to see if the **speakingRadius** of the NPC is less than the distance between the origins. If it is then there is an NPC with in speaking range of the player. I call the **StartDialog** method of that NPC to start its dialog, set the **inDialog** field to be true so I know that there is a dialog in progress and break out of the loop. The reason I break out of the loop is because there is no need to check for more NPCs in the list of NPCs. If there was already a dialog in progress I set **inDialog** to false so I know there is no dialog in progress. In a foreach loop I then loop through all of the NPCs to see which one was talking and call the **StopTalking** method of that NPC. If the space bar has not been pressed and released I then check to see if the player is trying to move the sprite using the same method as before.

That is it for this tutorial. In the next tutorial I will continue on with adding in dialog for the NPCs 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.