Creating a Role Playing Game with XNA Game Studio
Part 46
Minor Fixes

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/New2DRPG44.zip You can download the graphics from this link: Graphics.zip

There are a few things that I want to fix. They are minor things but they have been bugging me for a while now. The first is I don't like the HUD on the bottom of the screen. I'd much rather display messages in a pop up window. So, I'm going to remove that. The first thing you need to do is go to the **TileEngine** class. You will need to update the **viewPortHieght** field to fill the entire screen. Change it to the following.

```
static int viewPortHeight = 768;
```

With that done there are a few small changes to make to the **ActionScreen** class. What you will want to do is remove everything that has to do with the HUD. At the start of the class there is the field **characterHUDTexture**. Go a head and delete that. That will cause a few errors. One in the **LoadContent** method and one in the **Draw** method. You can change those methods to the following.

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

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

What I want to next is handle the input of the game in the **Game1** class. What I mean is that instead of having **CheckKey** and **CheckButton** methods littered all over the game have static methods and fields in the **Game1** class that can be called from anywhere else in the game. The first thing to do is to change all of the **KeyboardState** and **GamePad** states to static.

```
static KeyboardState newState;
static KeyboardState oldState;
static GamePadState newPadState;
static GamePadState oldPadState;
```

I also want to have these available anywhere in the game where they might be needed. To do that I will create some public static get only properties. I would have made the fields public but that would have meant making quite a few changes to the game and this seemed to be the best solution. Add

the following properties to the **Game1** class.

```
public static KeyboardState NewState
{
    get { return newState; }
}

public static KeyboardState OldState
{
    get { return oldState; }
}

public static GamePadState NewPadState
{
    get { return newPadState; }
}

public static GamePadState OldPadState
{
    get { return oldPadState; }
}
```

I needed to make a small change to the **Update** method for this to work. I need to move where I set the old state fields to the new state fields after the call to **base.Update**. If you don't when you try and call the static methods from a component outside of the class the new and old state fields will be the same as the update methods of components are called during the call to **base.Update**. This is the code for the new **Update** method.

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

    if (!dialog.Enabled)
        inDialog = false;

    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();
        }

        base.Update(gameTime);

        oldState = newState;
        oldPadState = newPadState;
    }
```

The next thing to do is to change the classes that use the **CheckKey** and **CheckButton** methods. I will handle the **ButtonMenu** first because I have another change that I want to do in the **DialogComponent** class. The first thing to do is to delete the fields that handle the state of the keyboard and the game pad. Remove the **newState**, **oldState**, **newPadState**, and **oldPadState** fields from the class. You can also remove the **CheckKey** and **CheckButton** methods from the class. The last thing to do is to update the **Update** method. You will need to remove anything that uses the fields and methods that were removed and for the **CheckKey** and **CheckButton** reference them with **Game1**. This is the code for the new **Update** method.

```
public override void Update(GameTime gameTime)
{
    if (Game1.CheckKey(Keys.Down) || Game1.CheckButton(Buttons.DPadDown))
    {
        selectedIndex++;
```

```
            if (selectedIndex == menuItems.Count)
                selectedIndex = 0;
        }
        if (Game1.CheckKey(Keys.Up) || Game1.CheckButton(Buttons.DPadUp))
        {
            selectedIndex--;
            if (selectedIndex == -1)
            {
                selectedIndex = menuItems.Count - 1;
            }
        }
        base.Update(gameTime);
    }
```

Before I get to the **DialogComponent** class I created a few new interface items. You can find all of the graphics that I used in the Content.zip file on my web site. You don't need to use them all but you should at least use the one for the **DialogComponent**. I also converted most graphics from JPEG to PNG format. Though JPEGs take less space, they are compressed and there is some loss of detail in them. Using a format that does not lose detail is always a good idea when it comes to making games. The people playing your game will love you for it even if it means that it takes up more space. After you have unzipped all of the files you can add them to the **GUI** folder in the **Content** folder of the game by right clicking the **GUI** folder, selecting **Add**, and then **Existing item**. You will now have a lot of duplicate items in the **GUI** folder. To fix this click right click on any of the files that end with **.jpg** and select **Delete**. You should at least replace the **conversationbox.png** file with the new version. The reason is that it is bigger and it will hold the text of the conversation better.

Now that you have the new graphic it is time to make some changes to the **DialogComponent** class. The first thing to do is to remove the **newState**, **oldState**, **newPadState**, and **oldPadState** fields. Also remove the **CheckKey** and **CheckButton** methods. Now you can change the **Update** method so that you reference **Game1** for the **CheckKey** and **CheckButton** methods. This is the new code for the **Update** method.

```
public override void Update(GameTime gameTime)
{
    if (dialog != null && npc != null)
    {
        if (Game1.CheckKey(Keys.Up) || Game1.CheckButton(Buttons.DPadUp))
        {
            currentHandler--;
            if (currentHandler < 0)
                currentHandler = dialog.HandlerCount - 1;
        }
        if (Game1.CheckKey(Keys.Down) || Game1.CheckButton(Buttons.DPadDown))
        {
            currentHandler++;
            if (currentHandler == dialog.HandlerCount)
                currentHandler = 0;
        }
        if (Game1.CheckKey(Keys.Enter) || Game1.CheckButton(Buttons.B))
        {
            dialog.InvokeHandler(npc, currentHandler);
            currentHandler = 0;
        }
    }
    base.Update(gameTime);
}
```

What I will do next is use the new image for the dialogs to appear in. The first thing to know is that the border around the image that I created is about 10 pixels in size so you will need to pad things a little when you are drawing them. After a bit of trial and error I found values that gave a bit of padding and things seemed to look good in the dialog. In the **Draw** method change the line that initialize the **textPosition** variable to the following.

```
Vector2 textPosition = new Vector2(position.X + 12, position.Y + 12);
```

The padding for right side of the pop up was again fixed by trial and error. In the **WrapText** method there is an if statement. Changing it to the following will fix the padding for the right side of the window.

```
if (length + size < position.Width - 36)
```

The one other that I want to fix is a small problem with starting and handling dialogs with sprites. I've been trying things with the Xbox controller and it doesn't feel right to me. Having to press A to start a conversation and then B for handling the options just feels weird. I wanted to change it so that B would start a conversation and handle the options for the conversation. I will also change it so that Enter is used for starting conversations. This isn't going to be as easy as the other fixes. Part of the problem is when I checked if there is an NPC in speaking range I started the dialog right there. If you do that with the new input system the first option in the dialog will be selected automatically. I believe I found a better solution handling starting a dialog. The first thing to do is to add a new field to the game that will hold which NPC the player character is in a dialog with. Add this field to the **Game1** class near the other NPC fields.

```
NPC dialogNPC = null;
```

I will be using **null** to help in deciding when the game is in dialog mode and when to start a new dialog. The next thing to do is to change the **HandlePlayerInput** method. What you want to do there is check if the Enter key or the B button have been pressed instead of A or the Space key. This is the new **HandlePlayerInput** method.

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

        if (CheckAttackRadius(gameTime))
            return;

        if (CheckKey(Keys.Enter) || CheckButton(Buttons.B))
            CheckSpeakingRadius();
    }
    if (!inDialog)
        HandlePlayerMovement();
}
```

Next I needed to modify the **CheckSpeakingRadius** method to work with the new field I added

in. What I did was in the if statement where I checked to see if the player and the NPC are within range is set the **dialogNPC** field to be the current **NPC**, **inDialog** to true, and then break out of the loop. This is the new code for the **CheckSpeakingRadius** method.

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

That leaves making a quick change to the **Update** method. What you want to do is first check to see if **inDialog** is true and **dialogNPC** is not null. If both conditions are true it is safe to start a new dialog with the **dialogNPC**. You can then check to see if the **inDialog** is false and the rest of what the **Update** method did. This is the code for the **Update** method.

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

    if (inDialog && dialogNPC != null)
    {
        dialog.Show();
        dialogNPC.StartDialog(dialogNPC.DialogName);
        dialogNPC = null;
    }

    if (!dialog.Enabled)
    {
        inDialog = false;
        dialogNPC = null;
    }

    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();
        }

        base.Update(gameTime);

        oldState = newState;
        oldPadState = newPadState;
    }
```

Well, that is it for this tutorial. I am planning on adding in more and more functionality that you would find in a complete role playing game. 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.