Creating a Role Playing Game with XNA Game Studio
Part 53
Moving to a Party Based Game

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 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://xnagpa.net/rpgtutorials/New2DRPG52.zip. You can download the graphics from this link: Graphics.zip

This tutorial is about moving from a single player character to having a party based game with multiple characters. Adding this to the game will be a little complicated as the game, up until this point, was meant to be a single player character game. In this tutorial I will concentrate on changing the game to work with a dynamic party. What I mean is that the characters in the party can change as the game plays, much like in Final Fantasy 13. Each part of the game will be a chapter. The different chapters can have different characters in them. This will allow you to have side quests where the character the player chooses at the start of the game won't be in the party. There will be a main list of characters to choose from. The party is made up of those characters that are in the current chapter. The player will control one character at a time, the party leader.

The first step is to create a wrapper class that has an **AnimatedSprite** and **PlayerCharacter** field for the different characters that will be in the game. These are special NPCs in the game that will join the party. There will be other NPCs that the party leader can interact with but won't join the party. I will be creating an editor that will create the different members that can be in the party and other NPCs in the game.

To the **PlayerCharacter** folder add a new class called **Character**. The code for this class follows next.

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

namespace New2DRPG
{
    public class Character
    {
        PlayerCharacter character;
        AnimatedSprite sprite;

        internal PlayerCharacter PlayerCharacter
        {
            get { return character; }
            private set { character = value; }
        }
```

```
        public AnimatedSprite Sprite
        {
            get { return sprite; }
            private set { sprite = value; }
        }

        internal Character(PlayerCharacter character, AnimatedSprite sprite)
        {
            PlayerCharacter = character;
            Sprite = sprite;
        }

        public void Update(GameTime gameTime)
        {
            character.Update(gameTime);
            sprite.Update(gameTime);
        }

        public void Draw(GameTime gameTime)
        {
            character.Draw(gameTime);
            sprite.Draw(gameTime);
        }
    }
}
```

This class is just a wrapper that will make working with the player characters easier. Everything that you will need can be accessed from this class. There is a using statement for the **SpriteClasses** here because the class uses the **AnimatedSprite** class and a using statement for the XNA Framework because the Draw and Update methods require the **GameTime** class for their parameters.

For now there are just two fields **character** and **sprite**. They are of type **PlayerCharacter** and **AnimatedSprite**. There is an internal and a public property. The internal property means that the property is only usable inside the assembly. Doing this meant not having to mess with all the access modifiers of other classes. The internal property is for getting the **character** field. The set part for both properties is private so that they can only be set in the constructor. Because the fields are classes, they are references so any changes made will change instances. The constructor just sets the fields using the private set properties. The **Update** and **Draw** methods just call the **Update** and **Draw** methods of the fields.

The next step is to create a class that represents the party. Remember that this class is for characters currently in the party, not all characters. To the **PlayerCharacter** folder in your game project, add a new class called **Party**. This class will hold the different characters in the party. The code for the **Party** class follows.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace New2DRPG
{
    public class Party
```

```
    {
        const int maxCharacters = 6;
        List<Character> characters;

        public int PartyMembers
        {
            get { return characters.Count; }
        }

        public Character this[int index]
        {
            get { return GetCharacater(index); }
            set { SetCharacter(index, value); }
        }

        public Party()
        {
            characters = new List<Character>(maxCharacters);
        }

        private Character GetCharacater(int index)
        {
            if (index > characters.Count)
                throw new IndexOutOfRangeException();
            return characters[index];
        }

        private void SetCharacter(int index, Character value)
        {
            if (index > characters.Count)
                throw new IndexOutOfRangeException();
            characters[index] = value;
        }

        public void Add(Character character)
        {
            if (characters.Count >= 6)
                throw new IndexOutOfRangeException();
            characters.Add(character);
        }

        public void RemoveAt(int index)
        {
            if (index >= characters.Count)
                throw new IndexOutOfRangeException();

            characters.RemoveAt(index);
        }

        public void Remove(Character character)
        {
            characters.Remove(character);
        }
    }
}
```

This class is for keeping track of the characters in the party. I decided to go with a party that has a maximum of six characters. The party size is dynamic though, there can be anywhere from one to six

characters at a time. I could have just used a **List<Character>** in the **Game1** class but this allows better control and can easily be expanded.

There is a private constant in here that defines the size of the party. You can change this to have a smaller or larger party size. I keep track of the characters in a **List<Character>**. The constructor of the class just creates a new list of characters with a size of the maximum number of characters.

I added an indexer that takes an integer. This is so you can access the characters in the party like an array. The indexer calls private methods that I will get to shortly. There is also a property, **PartyMemebers**, that returns the number of characters in the party.

GetCharacter first checks to see if the index of the character is greater than or equal to the number of characters in the list of characters. If it is, I throw and IndexOutOfRangeException. I then return the character at that index. The **SetCharacter** method also checks the index and will throw an exception if it is out of bounds. It then sets the character to the value passed in.

There are three static methods that work like the methods of the **List<T>** class. The **Add** method adds a character to the party. The **RemoveAt** method removes the character at the index from the party. It checks to make sure that the index isn't greater than or equal to the number of characters in the party. The **Remove** method will remove a specific character from the party.

Changing the game to a party based system will be a bit of a pain. I will start with the **PlayerComponent**. Open up the code for the **PlayerComponent** and change it to the following.

```
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 : DrawableGameComponent
    {
        Character character;
        Game game;
        bool inCombat = false;
        SpriteBatch combatSpriteBatch;

        public PlayerComponent(Game game, Character character)
            : base(game)
        {
            this.character = character;
            this.game = game;
            combatSpriteBatch =
```

```
            (SpriteBatch)Game.Services.GetService(typeof(SpriteBatch));
    }

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

    public Vector2 Origin
    {
        get { return character.Sprite.Origin; }
    }

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

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

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

    public int SpriteWidth
    {
        get { return character.Sprite.Width; }
    }

    public int SpriteHeight
    {
        get { return character.Sprite.Height; }
    }

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

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

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

```
            }
            else
            {
                character.PlayerCharacter.Draw(gameTime);
                combatSpriteBatch.Draw(
                    character.Sprite.Texture,
                    character.Sprite.Position,
                    character.Sprite.CurrentRectangle,
                    Color.White);
            }
        }

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

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

The first thing that I did was change from having **AnimatedSprite** and **PlayerCharacter** fields to having a **Character** field. In the constructor I create a new **Character**. I also added a second constructor that takes a **Character** parameter. I change the properties to use the new field instead of the old fields. In the **Update** method I call the **Update** method of the **Character** class. In the **Draw** method I now call **base.Draw** and then inside the if statement that checks if the game is in combat mode call the **Draw** method of the **Character** class. In the else part I call the **Draw** method of the **PlayerCharacter** and **Sprite** properties of the **Character** class.

You can build and run the game now and the game will play like before. The party system isn't implemented yet though. There are a few more things that need to be added in first. The one thing you will need is a **Party** field in the **Game1** class. You can add the following field to the **Game1** class near the **PlayerComponent**and **PlayerCharacter** fields

```
Party party = new Party();
```

The party leader will always be at index zero in the list of characters. The **PlayerComponent** calls the **Update** and **Draw** methods of that character. You will also want to call the **Update** and **Draw** methods of the other party members. In the **HandlePlayerInput** method is the best place to call the **Update** methods of the other party members. You can change that method to the following.

```
private void HandlePlayerInput(GameTime gameTime)
{
    player.Update(gameTime);

    if (!inDialog)
    {
        foreach (NPC npc in npcs)
            npc.Update(gameTime);
```

```
            if (CheckAttackRadius(gameTime))
                return;

            itemManager.Update(gameTime);
            itemManager.CheckChestCollision(player.Origin);

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

            for (int i = 1; i < party.PartyMembers; i++)
                party[i].Update(gameTime);

        }
    }
```

You will want the other characters in the party to respond to the player character's actions so it is best to call their **Update** methods after handling the input for the player character. Since the character the player is controlling is at index zero you start the loop to loop through the characters and one. I used the indexer to use the party field as an array to call the **Update** method of the **Character** class passing in the **gameTime** parameter.

You will also want to draw the characters in the party and that will be done in the **Draw** method of the **Game1** class. You can change the **Draw** method to the following.

```
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);

        for (int i = 1; i < party.PartyMembers; i++)
            party[i].Draw(gameTime);

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

I just added a for loop like I did in the **HandlePlayerInput** method and call the **Draw** method of the party member using the indexer. I did it inside the if statement that checks if the current screen is the action screen of the quit action screen.

I think that is as far as I will go in this tutorial. I know that adding in party support isn't finished quite yet. The combat screen is just going to be a little problematic as well as having characters join the party. I will get to the combat screen in the next tutorial, which will be done very soon.

I will be working on these tutorials and I will be adding in more and more functionality to the game that you would expect to find in a role playing game. I encourage you to keep either visiting my site http://xnagpa.net or my blog, XNA Game Programming Adventures Blog for the latest news on my tutorials.