XNA Screen Manager

XNA Screen Manager
Part Four
Creating a Pop Up Screen

This is a forth tutorial in my XNA Screen Manager tutorial. This tutorial is generic enough to work in both XNA 3.0 and XNA 3.1 so you can use either one of them for the tutorial. Pop up screens are common in games, especially when the player wants to exit the game. I will show you a way to create a pop up screen and how to use it so that the rest of the game will continue to render but it will not update the game. You could also use this concept to pause the game, in case the player needs to make a run to the fridge for a snack.

You should probably finish the first three tutorials in the series. You can find links to the tutorials on the XNA Tutorials page of my web site.

To get started you will want to open your project from the last tutorial I wrote. Now that you have the tutorial loaded you are ready to start. What I am going to do is create a new class that inherits from **GameScreen** and have the image centered on the screen. There will also be a menu with two entries Yes and No. You can use this image for the pop up screen, or you can use one of your own but you should probably have the file name, with out the extension, as quitscreen to reduce confusion with the code I will use later. It is totally up to you. Once you have the image you will want to add it to the **Content** folder. Right click the **Content** folder, select **Add** and then **Existing Item** and navigate to your image.

To do this I will first make a few changes to the **MenuComponent** class. All I am going to do is add in three properties to the class. The first one will be a get and set property that can get or set the position of the menu on the screen. The other two are get only properties and will get the width and the height of the menu. I will use these properties to center to place the menu in the pop up window. Add these three properties to the **MenuComponent** class.

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

public float Width
{
    get { return width; }
}

public float Height
{
    get { return height; }
}
```

XNA Screen Manager

Now we can code the **PopUpScreen** class. I will give you the code for the **PopUpScreen** class and then explain it after you have read it. Add a new class to your project called **PopUpScreen**. This is the code for the **PopUpScreen** class.

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

namespace ScreenManager
{
    class PopUpScreen : GameScreen
    {
        MenuComponent menuComponent;
        Texture2D image;
        Rectangle imageRectangle;

        public int SelectedIndex
        {
            get { return menuComponent.SelectedIndex; }
            set { menuComponent.SelectedIndex = value; }
        }

        public PopUpScreen(Game game, SpriteBatch spriteBatch, SpriteFont
spriteFont, Texture2D image)
            : base(game, spriteBatch)
        {
            string[] menuItems = { "Yes", "No" };
            menuComponent = new MenuComponent(game,
                spriteBatch,
                spriteFont,
                menuItems);
            Components.Add(menuComponent);
            this.image = image;

            imageRectangle = new Rectangle(
                (Game.Window.ClientBounds.Width - this.image.Width) / 2,
                (Game.Window.ClientBounds.Height - this.image.Height) / 2,
                this.image.Width,
                this.image.Height);

            menuComponent.Position = new Vector2(
                (imageRectangle.Width - menuComponent.Width) / 2,
                imageRectangle.Bottom - menuComponent.Height - 10);
        }

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

```
        public override void Draw(GameTime gameTime)
        {
            spriteBatch.Draw(image, imageRectangle, Color.White);
            base.Draw(gameTime);
        }
    }
}
```

The code is basically identical to the **StartScreen** class. The differences are the constructor of the classes. The first difference is the menu options. The **PopUpScreen** class has two options: **Yes** and **No**. Also, in the constructor of the **PopUpScreen** class instead of having the image fill the screen I center the image on the screen. I center the image on the screen by taking the width of the screen, subtracting the width of the image and dividing that by 2. That centers the image horizontally. To center the image vertically you take the height of the screen, subtract the height of the image and divide that value by 2.

The other difference is that I position the menu relative to the position of the image on the screen. I center the menu horizontally like I centered the menu on the start screen. I first take the **X** property of the **Rectangle** and the I take the width of the image, subtract the width of the menu and divide that by 2 and add it the **X** property of the **Rectangle**. I didn't think centering the menu vertically was a good idea. Instead, I positioned the menu relative to the bottom of the image. Since I used a **Rectangle** for the image I know where the bottom of the **Rectangle** is using the **Bottom** property. As you move up the screen the values of **Y** decrease so I need to subtract from that value. I first subtracted the height of the menu from the bottom of the rectangle and then an addition 10 pixels to give it a little padding.

Those were the only changes in the class. The rest of the changes will be in the **Game1** class. The first thing you will need to do of course is add in a field to hold an instance of the screen. Add the follow field to the **Game1** class near the other screen fields.

```
PopUpScreen quitScreen;
```

What you need to do now is create an instance and add it to the list of components for the game. This will be done in the **LoadContent** method. This is important. You need to create the instance after all of the other screens that you want the screen to be on top of. The reason is that the components of the game are drawn in the order you add them to the list of components. If you add the screen before the action screen it will not appear because we are still drawing the action screen, just not updating it. This is the code for the **LoadContent** method.

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

    startScreen = new StartScreen(
        this,
        spriteBatch,
        Content.Load<SpriteFont>("menufont"),
        Content.Load<Texture2D>("alienmetal"));
    Components.Add(startScreen);
    startScreen.Hide();
```

```
    actionScreen = new ActionScreen(
        this,
        spriteBatch,
        Content.Load<Texture2D>("greenmetal"));
    Components.Add(actionScreen);
    actionScreen.Hide();

    quitScreen = new PopUpScreen(
        this,
        spriteBatch,
        Content.Load<SpriteFont>("menufont"),
        Content.Load<Texture2D>("quitscreen"));
    Components.Add(quitScreen);
    quitScreen.Hide();

    activeScreen = startScreen;
    activeScreen.Show();
}
```

The last thing to do is to handle the input for the new screen. You handle input in the **Update** method. The **Update** method will get very long if you handle the input for all of your screens in it. What I decided to do was to refactor it a little by creating methods to handle the input for the various screens. I created methods: **HandleStartScreen**, **HandleActionScreen** and **HandleQuitScreen**. I will give you the code for all four methods and then explain it.

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

    if (activeScreen == startScreen)
    {
        HandleStartScreen();
    }
    else if (activeScreen == actionScreen)
    {
        HandleActionScreen();
    }
    else if (activeScreen == quitScreen)
    {
        HandleQuitScreen();
    }

    base.Update(gameTime);

    oldKeyboardState = keyboardState;
    oldGamePadState = gamePadState;
}
```

```
private void HandleStartScreen()
{
    if (CheckKey(Keys.Enter) || CheckButton(Buttons.A))
    {
        if (startScreen.SelectedIndex == 0)
        {
            activeScreen.Hide();
            activeScreen = actionScreen;
            activeScreen.Show();
        }
        if (startScreen.SelectedIndex == 1)
        {
            this.Exit();
        }
    }
}

private void HandleActionScreen()
{
    if (CheckButton(Buttons.Back) || CheckKey(Keys.Escape))
    {
        activeScreen.Enabled = false;
        activeScreen = quitScreen;
        activeScreen.Show();
    }
}

private void HandleQuitScreen()
{
    if (CheckKey(Keys.Enter) || CheckButton(Buttons.A))
    {
        if (quitScreen.SelectedIndex == 0)
        {
            activeScreen.Hide();
            activeScreen = startScreen;
            activeScreen.Show();
        }
        if (quitScreen.SelectedIndex == 1)
        {
            activeScreen.Hide();
            activeScreen = actionScreen;
            activeScreen.Show();
        }
    }
}
```

As you can see, the **Update** method has been scaled down to fewer lines of code. It is good to try and keep your methods short, in any sort of programming. Longer methods are harder to debug for one thing and there is a greater chance of introducing bugs when you add code to longer methods. After getting the state of the keyboard and the game pad there is a series of if-else-ifs. The reason is that you only want to process the current screen for that particular frame of the game. If you just did if statements, and you changed from one screen to another there would be a possibility of processing

another screen in that frame of the game. You don't want to do that.

What happens here is I check to see which screen is the active screen. I'm not comparing classes for equivalence here. I'm checking to see if the references to the classes are the same. There is a big difference. What I am saying is that if **activeScreen** and **startScreen** point to the same instance then do this. I'm not saying, is all the data in **activeScreen** the same as **startScreen**. After the call to **base.Update** I set the old states of the keyboard and the game pad to the current states. You might be wondering what happened to allowing the game to exit from the **Update** method. Well, I removed that and placed it in the **HandleActionScreen** method. If I hadn't and the player pressed the back button the game would exit and the screen wouldn't be displayed.

The **HandleStartScreen** method is the code from the original **Update** method. It just checks to see if the enter key or the A button on the game pad of been pressed and released. The **HandleActionScreen** method is where I handle displaying the screen to ask if the player is sure they want to quit. I check to see if the back button on the game pad or the escape key have been pressed once. If they have I just set the **Enabled** property of the active screen to false instead of calling the **Hide** method of the screen. What this does is tell the action screen to not update itself but it can draw itself. So your game is effectively paused until the player chooses an option from the menu. I then set the active screen to be the quit screen and call the Show method of the active screen displaying the quit screen.

The last method is the **HandleQuitScreen** method. This method first checks to see if the **A** button or the enter key have been pressed. If they have there are two if statements that check what the selected index is. If it is zero, I call the **Hide** method of the quit screen. I then set the active screen to be the start screen and finally call the **Show** method of the active screen. This is the behaviour you want in your games. You don't want the game to just exit, you want to take the player back to the menu. If the selected index is one, the no option, I hide the quit screen, set the active screen to be the action screen and call the **Show** method.

This is the last tutorial I'm going to do in the screen management series. You now have all of the tools you will need to manage screens in your game. You are not in anyway limited to the screens I've shown you. You can create any number of screens, each with different functionality. The sky is the limit. These are all good bases from which you can work from.