XNA Screen Manager
Part Two
Creating Game Screens

This is part two of a two part tutorial about creating screen management system with XNA. 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. This tutorial uses some common object-oriented programming principles. If you do not have an understanding of object-oriented programming you may have a hard time understanding parts of this tutorial.

To get started you will want to open your project from the previous tutorial I wrote. If you haven't completed that tutorial go now and read it. You can find the link to the tutorial at this link: http://xna.jtmbooks.com/xnatutorials.html

Now that you have the tutorial loaded you are ready to start. To handle the different screens in the game what I am going to do is to create an abstract base class called **GameScreen** that other screens in your game will inherit from. If you are unsure what an abstract class is, an abstract class is a class that you must inherit from. Any methods, properties, fields, indexers, etc that belong to the abstract class can be used in the inherited class. The inherited class can use any protected or public items of an abstract class but not private. You can also use the abstract keyword inside the abstract class to define properties and methods that must be implemented in the base class. You can not create an instance of an abstract class. An object of an abstract class however can be assigned objects of derived classes. This is an object-oriented principle called polymorphism. Polymorphism is the ability of a base class to act as an inherited class at run time. This is one of the key three principles of object-oriented programming: **Encapsulation**, **Inheritance**, and **Polymorphism**. Polymorphism is probably the hardest to understand. I hope that showing you an example of polymorphism will help you understand the principle better.

I also decided to have both of the screens have a background image. You can select these two images in the link below or you can find two of your own. Add the two images to the **Content** folder by right clicking it and selecting **Add** and then **Existing Item**. Select your two images and they will be added to the **Content** folder and you can load them using the **Content.Load** method.

http://xna.jtmbooks.com/Downloads/screenmanager.zip

What you will want to do next is add a new game component to the project. Right click your project in the solution explorer, select **Add**, and then select **New Item**. Make sure that the **XNA Game Studio** node is selected on the left and choose **Game Component**. Name this **Game Component GameScreen**. I will explain the code after you have read it.

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

XNA Screen Manager

```
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 ScreenManager
{
    public abstract class GameScreen :
        Microsoft.Xna.Framework.DrawableGameComponent
    {
        List<GameComponent> components = new List<GameComponent>();
        protected Game game;
        protected SpriteBatch spriteBatch;

        public List<GameComponent> Components
        {
            get { return components; }
        }

        public GameScreen(Game game, SpriteBatch spriteBatch)
            : base(game)
        {
            this.game = game;
            this.spriteBatch = spriteBatch;
        }

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

        public override void Update(GameTime gameTime)
        {
            base.Update(gameTime);
            foreach (GameComponent component in components)
                if (component.Enabled == true)
                    component.Update(gameTime);
        }

        public override void Draw(GameTime gameTime)
        {
            base.Draw(gameTime);
            foreach (GameComponent component in components)
                if (component is DrawableGameComponent &&
                    ((DrawableGameComponent) component).Visible)
                    ((DrawableGameComponent) component).Draw(gameTime);
        }

        public virtual void Show()
        {
            this.Visible = true;
            this.Enabled = true;
            foreach (GameComponent component in components)
            {
                component.Enabled = true;
                if (component is DrawableGameComponent)
                    ((DrawableGameComponent)component).Visible = true;
            }
        }

        public virtual void Hide()
        {
            this.Visible = false;
            this.Enabled = false;
            foreach (GameComponent component in components)
            {
                component.Enabled = false;
                if (component is DrawableGameComponent)
                    ((DrawableGameComponent)component).Visible = false;
            }
        }

    }
}
```

The first thing that you will see is that between **public** and **class** there is the keyword **abstract**. This makes this an **abstract** class that you can not create an object with. In order to use it you must inherit from it. Since your game screens will have visual components I inherited the class from **DrawableGameComponent** instead of just **GameComponent**. There are three fields in this class: **components**, **game**, and **spriteBatch**. The first one, **components**, is a **List<GameComponent>** that will hold any child components of the screen. **game** is an object of type **Game** and will hold the current **Game** object. Finally, **spriteBatch** is a **SpriteBatch** object that you will be able to use to draw in your inherited classes. There is also a get only property, **Components**, that you can use in child classes and classes that use your screen to get the list of components of your screen.

The constructor of the class requires two parameters: a **Game** object, **game**, and a **SpriteBatch** object, **spriteBatch**. Since this is a game component there is also a call to the constructor of the base class with **base(game)**. The constructor then sets the **game** and **spriteBatch** fields with the parameters passed in to the constructor.

In the override of the **Update** method I have a foreach loop that will loop through all of the game components in the **components** field. Inside this loop I check to see if the **Enabled** property of the current component is true. If it is I call the **Update** method of that component passing in **gameTime**.

In the override of the **Draw** method there is a foreach loop that will loop through all of the game components in the **components** field. This is an example of polymorphism in action. When the game is running you don't know if the component is a **DrawableGameComponent** or a **GameComponent**. In C# you can use the **is** keyword to text if an object belongs to a class. I check to see if **component** is a **DrawableGameComponent**. You might think that C# will through an error in the and part of the condition. It will not. C# is smart enough to know that if the first part is false there is no need to check the second part. In the second part in paratheneses I cast **component** to a **DrawableGameComponent** and check the **Visible** property of **component**. This is the syntax you need to use when you are using polymorphism of a inherited class of a base class that implements fields, properties, methods, etc that are not in the base class. If this is a **DrawableGameComponent** and the **Visible** propery is true I call the draw method, again casting it, of **component**.

Next there is a virtual method called **Show**. Virtual methods are methods that you can override in inherited classes to change their behavior. This method will be used to set the component to be enabled and visible. It sets the **Enable** and **Visible** properties to true so the component will be updated and drawn. Then in a foreach loop I loop through all of the game components and set their **Enabled** property to true. Then, after checking if this component is a **DrawableGameComponent**, I set the **Visible** property to true.

There is another virtual method called **Hide**. This method works in the opposite method of **Show**. It sets the **Enable** and **Visible** properties to false so the component will not be updated or drawn. Again in a foreach loop I loop through all of the game components. If they are **DrawableGameComponents** I set their **Visible** property to false.

Now I will create a new class that inherits from **GameScreen** that will act as the main menu for the game so the player can choose to play the game or exit the game. To do this right click your project in the solution explorer and then select **Add** and then **Class**. Name this class **StartScreen**. I will again explain the code after you have read it.

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

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

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

        public StartScreen(Game game,
            SpriteBatch spriteBatch,
            SpriteFont spriteFont,
            Texture2D image)
            : base(game, spriteBatch)
        {
            string[] menuItems = { "Start Game", "End Game" };
            menuComponent = new MenuComponent(game,
                spriteBatch,
                spriteFont,
                menuItems);
            Components.Add(menuComponent);
            this.image = image;
            imageRectangle = new Rectangle(
                0,
                0,
                Game.Window.ClientBounds.Width,
                Game.Window.ClientBounds.Height);
        }

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

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

This class uses parts of the XNA framework so there are using statements for the framework and the Graphics classes. This class inherits from **GameScreen** so it has all of the functionality of the **GameScreen** class and it can override parts of it that it wants to act differently than the base class.

The class will handle the menu for the game so there is a **MenuCompontent** field. Because I will have a background image I have a **Texture2D** field called **image** and a **Rectangle** field called **imageRectangle**. I will be using this rectangle to have the image fill the entire game window. The game will be acting on the player's menu choice so there is a property called **SelectedIndex** that just gets and sets the **SelectedIndex** property of the **MenuComponent**.

The constructor for this class takes four parameters: a **Game** object, a **SpriteBatch** object, a

SpriteFont object and a **Texture2D**. The constructor also calls the constructor of the base class that requires a **Game** object and a **SpriteBatch** object. The constructor creates a new menu with two options: **Start Game** and **End Game**. It then adds the menu to the list of components for the component. It then sets the **image** field to be the **image** being passed in. It then creates a rectangle that will fill the entire window. There is a property of the **GameComponent** class called **Game** that gets the **Game** object associated with the component. The **Game** object has a property **Window** that holds information about the window. I used the **Width** and **Height** properties of the **ClientBounds** property to get the width and height of the window. I created a rectangle with **X** and **Y** values of 0, the upper left hand corner of the window and with the width and height of the window. You will see why when I get to the **Draw** method.

There is an override of the **Update** method so that the menu will update itself. There is also an overload of the **Draw** method. The order of rendering in 2D is important. If you want objects to appear above other objects you need to draw them last. That is why I call the **Draw** method of the **SpriteBatch** object before the call to **base.Draw**. When **base.Draw** is called all child components, if they are visible, are drawn. So, if you draw the image after the call to **base.Draw** the image will be drawn over top of the menu and it won't be seen. The overload of the draw method that I used takes three parameters: the **Texture2D** to draw, the destination **Rectangle**, and the tint color. Because of the way I created the rectangle in the constructor using the height and width of the window it will fill the window.

I am going to add a second screen to the game. This one is where the game will take place. You can modify this screen to work like your game. You can add a **ContentManager** object for instance to load in your content like a regular game and work on everything in that screen. Right click your project and add a new class called **ActionScreen**. This is the code for that screen.

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

namespace ScreenManager
{
    class ActionScreen : GameScreen
    {
        KeyboardState keyboardState;
        Texture2D image;
        Rectangle imageRectangle;

        public ActionScreen(Game game, SpriteBatch spriteBatch, Texture2D image)
            : base(game, spriteBatch)
        {
            this.image = image;
            imageRectangle = new Rectangle(
                0,
                0,
                Game.Window.ClientBounds.Width,
                Game.Window.ClientBounds.Height);
        }

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

            keyboardState = Keyboard.GetState();

            if (keyboardState.IsKeyDown(Keys.Escape))
                game.Exit();
        }

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

There are using statements for the XNA framework, the Graphics classes, and the Input classes. The reason for the Input classes is that I want the game to be able to exit if the player presses the escape key while the action screen is the current screen. This isn't the best behavior for your game but it will work for this sample. This class also inherits from **GameScreen**. There is a **KeyboardState** field to hold the state of the keyboard, a **Texture2D** field for the background image to draw and a **Rectangle** field to hold the rectangle for the window.

The constructor for this class takes three parameters: a **Game** object, a **SpriteBatch** object, and a **Texture2D**. Just like the constructor for the **StartScreen** class this class calls the constructor of the base class passing the **Game** and **SpriteBatch** objects. The constructor just sets the **image** field and creates a rectangle the size of the window like in the **StartScreen** class.

In the **Update** method, after the call to **base.Update**, I get the state of the keyboard. I then check to see if the Escape key is down. If it is I exit the game calling the **Exit** method of the **Game** object. Like I mentioned earlier this isn't good practice but it will be okay for this tutorial. You will want to do something different in your own games. The **Draw** method draws the image for the background and then calls **base.Draw** to draw any child components, even there are none it is a good habit to have.

It is time to implement these screens into the game. The first thing you can do is replace the **MenuComponent** field with the following fields.

XNA Screen Manager

```
KeyboardState keyboardState;
KeyboardState oldKeyboardState;

GameScreen activeScreen;
StartScreen startScreen;
ActionScreen actionScreen;
```

The first two fields, **keyboardState** and **oldKeyboardState**, will hold the current and last states of the keyboard. I will use these to check if the player has selected a menu item when the **StartScreen** is the active screen. The third field is **activeScreen** and it is of type **GameScreen**. Like I mentioned at the start of the tutorial I will be using polymorphism for handling the different screens. In this case **activeScreen** will be set to either **startScreen** or **actionScreen**, which are the objects of type **StartScreen** and **ActionScreen** respectively. Because both of these types inherit from **GameScreen** at run time you can assign them to **GameScreen** and when you call any methods that have been overriden the methods of the appropriate class will be called. I do not have to cast them because the methods that I am using all belong to the base class **GameScreen**.

In the **LoadContent** method I will create the new screens and add them to the list of components of the game. By adding them to the list of components when the game runs the **Update** and **Draw** method of the components will be called automatically by XNA which in the end simplifies the game and makes all of this work worth while. I will explain the code further after you have read it.

```
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();

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

The code first creates a new instance of the **StartScreen** class passing in: this which is the current **Game** object, **spriteBatch** which is the **SpriteBatch** object for the game, the call to **Content.Load** loads in the font added to the game in the last tutorial, and it loads in the one image that I added to the game called **alienmetal**. If you used a different image you will have to substitute its asset name with **alientmetal**. The **startScreen** field is then added to the list of components for the game and then the **Hide** method is called to hide the screen. You might be screaming that there was no **Hide** method in the **StartScreen** class. There didn't have to be. Because the parent class had a public method **Hide**, the derived class **StartScreen** has access to that method and because it is public it can be called outside of the class.

I do something very similar for creating the **ActionScreen** object. It just required three parameters. I passed in this, the **spriteBatch** field, and the return of **Content.Load** with the asset name **greenmetal**. Again, if you used different assets you will need to use the appropriate asset name. The **ActionScreen** object is added to the list of components of the game and I call the **Hide** method as well. This is where polymorphism comes into play again. The field **activeScreen** is of type **GameScreen** but since **StartScreen** inherits from **GameScreen** I can assign **startScreen** to **activeScreen**. I then call the **Show** method of **activeScreen**. When you do this C# will see of what type **activeScreen** is and then it will go to that type and see if there is a method in that class. If there isn't a method that matches it then checks the parent class. If the parent class doesn't have that method it checks to see if that class has a parent class and if it does it will check that class and so on. If no method is found at compile time you will get an error, the program will not build.

That just leaves handling the input to move between screens. That will be done in the **Update** method. Just like in the **MenuComponent** I want to check if the key was down in the last frame and is up in the current frame. It is a very helpful method so I copied the **CheckKey** method from the **MenuComponent** class. This is the code for the **Update** and **CheckKey** methods.

```
protected override void Update(GameTime gameTime)
{
    keyboardState = Keyboard.GetState();
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
        this.Exit();

    if (activeScreen == startScreen)
    {
        if (CheckKey(Keys.Enter))
        {
            if (startScreen.SelectedIndex == 0)
            {
                activeScreen.Hide();
                activeScreen = actionScreen;
                activeScreen.Show();
            }
            if (startScreen.SelectedIndex == 1)
            {
                this.Exit();
            }
        }
    }
    base.Update(gameTime);
    oldKeyboardState = keyboardState;
}

private bool CheckKey(Keys theKey)
{
    return keyboardState.IsKeyUp(theKey) &&
        oldKeyboardState.IsKeyDown(theKey);
}
```

What I did was at the start of the **Update** method was get the state of the keyboard using the **GetState** method. I kept in the if statement that checks to see if the back button on the XBOX 360 controller was pressed. There is then an if statement that checks to see if **activeScreen** is equal to **startScreen**. This checks to see if the instances are the same, not if the objects are equal. Inside that if statement I check to see if the enter key has been pressed just once. Inside that if statement there are two other if statements. The first one checks to see if the **SelectedIndex** of **startScreen** is 0, which is the first menu item. If that is true I call the **Hide** method of **activeScreen** to hide the screen and its components. I then set **activeScreen** to **actionScreen** and then call the **Show** method of active screen. That effectively changes the game to use **actionScreen** as the current screen instead of **startScreen**. In the next if statement I check to see if the **SelectedIndex** property of **startScreen** is 1, which is the second menu entry. If it is I call the **Exit** method of **Game** class to exit the game. After the call to **base.Update** I set the **oldKeyboardState** field to the **keyboardState** field. The last method **CheckKey** is the same as the **CheckKey** method from the previous tutorial.

This is a simple way to implement screen management in your games to separate the logic of the screens into separate classes. I will write two more tutorials in this series. The next one will be about creating a pop up screen that the player can choose between yes or no answers. The other one will be about implementing a simple text box using this concept.