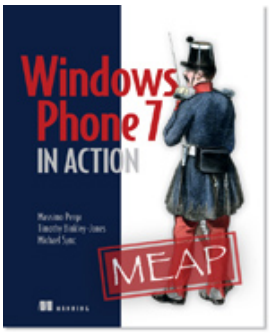
MANNING PUBLICATIONS



Windows Phone 7 in Action
Massimo Perga, Timothy Binkley-Jones, and Michael Sync

In this article, based on chapter 11 of Windows Phone 7 in Action, the authors explain how to implement starting, pausing, and resuming game play.

Implementing the Game Menu

The game menu provides two options—Choose Input and Play. Choose Input prompts the player to pick between controlling the game with a thumbstick, a button pad, gestures, or with the accelerometer. Play enables the game play component and allows the player to move around in our 3D world. The options are selected when the player touches the corresponding text.

> **Tip** The game in this article presents a simple game with a menu and pause/resume functionality. A sample game with more realistic transitions between menus, loading screens, and gameplay can be found on the AppHub. The Game State Management sample can be downloaded from http://create.msdn.com/en-US/education/catalog/sample/game_state_management.

Our `GameMenuComponent` doesn't draw text to the screen yet nor does it respond to touches. To accomplish drawing and menu selection we need to implement the `Draw` and `Update` methods.

Drawing the menu

Before we can draw text to the game screen, we need to add a font asset to our content project. Add a new `SpriteFont` item and name it `NormalFont`. We need a field to hold the loaded font content, and we need to override the `LoadContent` method to initialize the font.

```
private SpriteFont font;

protected override void LoadContent()
{
    font = Game.Content.Load<SpriteFont>("NormalFont");
}
```

Now that we have loaded the font, we can draw the words *Choose Input* and *Play* to the screen.

```
public override void Draw(GameTime gameTime)
{
    SpriteBatch.Begin();

    SpriteBatch.DrawString(font, "Choose Input",
        new Vector2(30, 30), Color.DarkRed);
    SpriteBatch.DrawString(font, "Play",
        new Vector2(30, 110), Color.DarkRed);

    SpriteBatch.End();
```

```
        base.Draw(gameTime);
    }
```

When the player taps the Play option, the `GameMenuComponent` needs to inform the `InputGame` to start game play. To accomplish this feat, the InputGame needs to provide a callback or delegate method that can be called to start play. We will implement this functionality by declaring a field of type `Action`. An Action represents a parameterless method that does not return no a value. An `Action` can be passed as a parameter to a method. We are going to modify `GameMenuComponent's` constructor to take an `Action` parameter will store the action in a field named playStarted.

```
    private Action playStarted;

    public GameMenuComponent(Game game, Action playCallback) : base(game)
    {
        playStarted = playCallback;
    }
```

We now have all the pieces in place to implement starting, pausing, and resuming game play.

Implementing Pause and Resume

When the game is launched, we create the game play component paused or disabled and the menu is created in an enabled and visible state. Starting and resuming are really the same operation, and can be implemented by enabling the game play component and disabling and hiding the menu component. We put these operations into a method named `ResumeGamePlay`, which we add to `InputGame`.

```
    private void ResumeGamePlay()
    {
        gameplay.Enabled = true;
        menu.Enabled = false;
        menu.Visible = false;
    }
```

We need to update `InputGame's Initialize` method to pass the `ResumeGamePlay` method as the action parameter.

```
    menu = new GameMenuComponent(this, ResumeGamePlay);
```

Now, we can return to `GameMenuComponent` and respond to the player's selecting the Play option in the menu. We will use the Touch API. For the menu, we want to call the playStarted action when the player releases a touch on the Play option. Listing 1 shows how the Touch API is used in the menu component's `Update` method.

Listing 1 Detecting when the player taps the play option

```
    using Microsoft.Xna.Framework.Input.Touch;

    public override void Update(GameTime gameTime)
    {
        TouchCollection touches = TouchPanel.GetState();
        for (int index = 0; index < touches.Count; index++)
        {
            TouchLocation location = touches[index];
            if (location.State == TouchLocationState.Released)
            {
                if (location.Position.Y > 110 && location.Position.Y < 150)
                    playStarted();
            }
        }
        base.Update(gameTime);
    }
```

Now that the game is running, how do we pause it? Game play can be paused by disabling the game play component, and enabling and showing the menu component. We will perform these operations in an `InputGame` method named `PauseGamePlay`.

```
private void PauseGamePlay()
{
    gameplay.Enabled = false;
    menu.Enabled = true;
    menu.Visible = true;
}
```

The *Windows Phone Marketplace Requirements* state that games should be paused with the hardware Back button. `InputGame.Update` already looks for Back button presses and we need to add the logic shown in listing 2 to pause the game instead of exiting.

Listing 2 Pausing the game on Back button press

```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back
        == ButtonState.Pressed)
    {
        if (gameplay.Enabled)
            PauseGamePlay(); #1
        else
            this.Exit();
    }
    base.Update(gameTime);
}
```

#1 Disables game play

If the Back button is pressed when game play is enabled #1 then call the `PauseGamePlay` method. If we are already paused and showing the menu then exit the game when the Back button is pressed.

We now have the first of the two menu options implemented. The second option will allow the player to switch between different input mechanisms used to control player movement.

Choosing an input type

Our game allows the player to control game play with a thumbstick, a button pad, gestures, or the accelerometer. The Choose Input menu option allows the player to specify which of the mechanisms they prefer. To capture the player's choice, we will display message box, shown in figure 1, prompting the player for a number. If the player inputs an invalid choice, we will display the error in another message box.

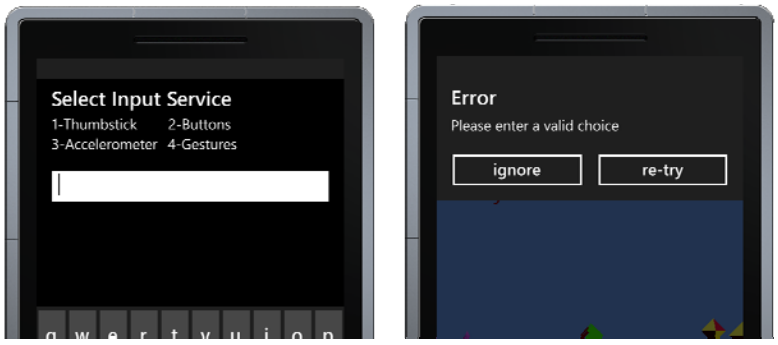


Figure 1 The input selection and error message boxes

An XNA game can display message boxes with the Guide class. The XNA Framework provides the Guide API to display message boxes, retrieve text input, and provide basic information about the run time environment. Table 1 shows the Guide methods supported on Windows Phone and that are available to all developers.

Table 1 Guide members available in Windows Phone

Member	Description
`IsScreenSaverEnabled`	Used to control when the phone enters the user idle mode.
`IsTrialMode`	Used to determine if the game has been purchased from the application marketplace.
`IsVisible`	Used to determine if the Guide is currently visible.
`SimulateTrialMode*`	Sets the value returned by IsTrialMode for debug mode testing.
`BeginShowKeyboardInput`	Displays a prompt and allows the player to input text.
`BeginShowMessageBox`	Displays a message box.
`EndShowKeyboardInput`	Returns the text input by the player.
`EndShowMessageBox`	Returns the index of the clicked message box button.

We will use the `IsVisible` property and the keyboard and message box methods to the Choose Input menu option. `IsScreenSaverEnabled` is used to prevent the phone from entering the user idle state, a power-saving mode intended to conserve the battery when the player is not interacting with the phone. Setting `IsScreenSaverEnabled` also sets the `UserIdleDetection` property of the `PhoneApplicationService` class.

The phone enters the user idle state when the player has not pressed a hardware button or touched the screen after a period of time. This can be important for games that make use of the accelerometer.

> **Note** The `Guide` provides a number of methods used to access Xbox LIVE Services. These methods are either not supported on the Windows Phone or require gamer services. Gamer services are only available to developers who have an Xbox LIVE partnership with Microsoft.

Before writing the code to prompt the player, we need to define an enumeration representing the player's choice. This enumeration will have values for each of the input mechanisms we are going to implement.

```
public enum InputChoice
{
    Thumbstick = 1,
    Buttons,
    Accelerometer,
    Gestures,
}
```

The InputChoice enumeration defines Thumbstick with a value of one to make the processing of user input easier as the player is asked to enter values between 1 and 4. The other enumeration values do not require an explicit number, since the compiler with generate their values sequentially.

We want to remember the user's choice across executions of the game and we will use `IsolatedStorageSettings` to persist the setting. Instead of defining a field variable, we are going to read from isolated storage. Before we can use isolated storage, we need to add a reference to the `System.Windows` assembly and add a using statement to InputGame.cs.

```
using System.IO.IsolatedStorage;
```

The first time the player starts the game, the setting will not exist in isolated storage and we need to initialize the setting in the game's constructor.

```
InputChoice inputChoice;
if (!IsolatedStorageSettings.ApplicationSettings
    .TryGetValue<InputChoice>("inputChoice", out inputChoice))
{
    IsolatedStorageSettings.ApplicationSettings["inputChoice"]
        = InputChoice.Thumbstick;
}
```

Moving back to the `GameMenuComponent` code, add the logic to detect the touch on the menu option. The new code goes into the `Update` method and will make a call to a method named `ShowMessageBox`.

```
if (location.State == TouchLocationState.Released)
{
    if (location.Position.Y > 110 && location.Position.Y < 150)
        playStarted();
    else if (location.Position.Y > 30 && location.Position.Y < 70)
        ShowMessageBox();
}
```

`ShowMessageBox` is where we will make a call to the `Guide`'s `BeginShowKeyboardInput` method. The `Guide` class is defined in the `GamerServices` namespace, so we add a using statement to the top of GameMenuComponent.cs.

```
using Microsoft.Xna.Framework.GamerServices;

public void ShowMessageBox()
{
    Guide.BeginShowKeyboardInput(PlayerIndex.One, "Select Input Service",
        "1-Thumbstick\t2-Buttons\r\n3-Accelerometer\t4-Gestures",
        null, ShowKeyboardCompleted, null);
}
```

The `Guide` displays message boxes using an asynchronous pattern. When `BeginShowKeyboardInput` is called, it begins the process of displaying the message box and immediately returns. At some point in the future, the message box is displayed.

> **Note** When BeginShowKeyboardInput or BeginShowMessageBox is called, the game receives an activated event. When a message box is dismissed, the game receives a deactivated event.

When you call `BeginShowKeyboardInput`, you must provide an `AsyncCallback` delegate to be called when the player dismisses the message. Our delegate is the method named `ShowKeyboardCompleted`, shown in listing 3, which validates the player's input. If the player enters a valid choice, we save it into settings and, if not, we display an error.

Listing 3 Validating player input

```
public void ShowKeyboardCompleted(IAsyncResult r)
{
    int result;
    string choice = Guide.EndShowKeyboardInput(r); #1
    if (choice != null)
    {
        if (Int32.TryParse(choice, out result)                #2
            && Enum.IsDefined(typeof(InputChoice), result))   #2
        {
```

```
                IsolatedStorageSettings.
                    ApplicationSettings["inputChoice"] = result;
            }
            else
            {
                while (Guide.IsVisible)                 #3
                    Thread.Sleep(32);                   #3
                Guide.BeginShowMessageBox("Error",
                    "Please enter a valid choice",
                    new[] { "Ignore", "Re-try" }, 0,
                    MessageBoxIcon.Error, ShowMessageBoxCompleted, result);
            }
        }
    }
```

#1 Retrieve player input
#2 Validate
#3 Wait until the message box is closed

Before we can validate the player's choice, we must retrieve the value input into the message box. To retrieve player input from the `Guide`, we call `EndShowKeyboardInput` (#1) and store the value in the variable named choice. If the player presses cancel, the choice variable will be null and we do not need to validate or store the input. Next, we attempt to convert the choice value into an integer and confirm that it is a value defined in the `InputChoice` enumeration (#2). If the choice is valid, we store the value in application settings.

When the player enters an invalid choice, we display an error message using the `BeginShowMessageBox` method. The `Guide` destroys message boxes asynchronously, and we must wait for the `Guide` to finish cleaning up the keyboard input message box (#3) before we display the error message. `BeginShowMessageBox` requires you to specify the text to display on the buttons shown in the message box. On Windows Phone, this is either one or two buttons. `BeginShowMessageBox` also takes an `AsyncCallback` delegate method, so we pass in `ShowMessageBoxCompleted`. `ShowMessageBoxCompleted` redisplays the input message box if the player clicks the retry button:

```
public void ShowMessageBoxCompleted(IAsyncResult r)
{
    int? result = Guide.EndShowMessageBox(r);
    if (result.HasValue && result == 1)
    {
        while (Guide.IsVisible)
            Thread.Sleep(32);
        ShowMessageBox();
    }
}
```

Now that we allow the player to change the game's input mechanism, we have completed our implementation of `GameMenuComponent`. When the player taps the play option on the menu, we enable the `GamePlayComponent`.

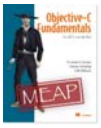
Summary

Supporting pause in your game is important, because the operating system might interrupt a game to handle other tasks such as a phone call. When the game is reactivated from an interruption, the game should restart in a paused state. Pausing the game can be as simple as disabling game play components that are running. The player should be able to pause a running game by pressing the Back button. We showed how to detect a back button press and determine if the game should be paused or if the game should exit.

Here are some other Manning titles you might be interested in:

Android in Action, Second Edition
W. Frank Ableson, Robi Sen, and Chris King

Android in Practice
Charlie Collins, Michael D. Galpin, and Matthias Kaeppler



Objective-C Fundamentals
For iOS4 and iPad
Christopher K. Fairbairn, Johannes Fahrenkrug, and Collin Ruffenach

Last updated: March 3, 2011