Programming Assignment #3

Overview

We will now use your knowledge of WPF and your Programming Assignment 2 to make a GUI for your role playing game battle simulation.

Here is an image of my WPF Battle simulation. Some of the tasks which we will need to accomplish are:

- Setup the general layout of the screen and the background
- Create a worker thread to do the battle simulation.
- Enhance the character classes (Wizard, Warrior, Mage, Archer) to have an image that shows the state of the character.
- Output the console text to a window in the GUI for debugging.

I will provide some public domain resources I have found to help you make your program. You may download others as you like. Make sure to provide references to downloaded material. As you complete this assignment, I suggest you compile and get it running after each major step.

Laying out the user Interface

Create a new WPF application as in Worksheet 3. Name it WPFBattle. Add your RPGLibrary project to this solution and make your WPFBattle project reference it. You will need all your classes you created in Programming Assignment 2, so go ahead and add those to the RPGLibrary project if they are not already there.

I have provided some image resources in the location http://web.cse.ohio-

state.edu/~kidwell.53/cse4253_csharp/Labs/Resources. Create a new folder in your WPFBattle project called content and download these now or browse the web to find some of your own images. Just to get a layout of your battle, use some image controls from the toolbox, and one textbox control to lay out the screen similar to what I have above. Your layout does not have to be exact. You may have a different number of characters in each party. You may place your "console" textbox anywhere you like (for example, centered on the top, or centered on the bottom, or as a separate column in the interface). Choose your background by adding an image to the main window's grid and setting its **stretch** property to "UniformToFill".

We will update the textbox to act like a console and replace the character images with a custom control below.

Creating a worker thread

In programming assignment 2, you created a program which ran from beginning to end. WPF has an event-driven programming model. In an event-driven programming model, you must return to the operating system to allow it to handle more events like keyboard input and mouse operations. You cannot have your program do work in the main thread or it will be unresponsive. When we have a long running process in such cases, we either: 1) Create a timer function that does some work each time it is called, or 2) Create a worker thread to do the work and notify the user interface when it is done or when things happen. We will take the latter method.

Create a new class called CombatThread. This class's purpose is create a new thread and take an ICombat and call AutoBattle() on it. The thread does not end until the battle ends.

Create a private variable to store and ICombat. I called mine "encounter". Create a constructor which takes and ICombat and saves it to the class member variable.

Create another private variable to hold the thread:

```
private Thread thread;
```

I will now give you two methods that this class should have. Start and Deactivate:

```
public void Start()
{
    thread = new System.Threading.Thread(() => //give this code
    {
        encounter.AutoBattle();
    });
    thread.Name = "GameThread";

    thread.Start();
}

public void Deactivate()
{
    thread.Abort();
}
```

The first method creates a new thread and initializes it using a lambda expression. The expression

creates a delegate to an anonymous function which just calls AutoBattle. It then starts the thread. The second method gives you a way to deactivate the thread when, for example, your program is exited because the user clicked the exit button in the upper right corner. After the next step, you can test your progress.

Updating the Console

Redirecting to console is pretty simple. We have to create a new StreamWriter class that writes to a text box. We can then redirect console output to this stream writer. First, create a new class called TextBoxStreamWriter. Make it inherit from System.IO.TextWriter.

TextWriter will need a member variable to hold a System.Windows.Controls.Textbox, so create one now. Create a constructor which takes a Textbox control and save the control in the member variable. We will use this textbox for output.

Our class needs two methods. One to return encoding for our stream, and one to write data to the textbox. The following is the encoding function:

```
public override Encoding Encoding
{
    get { return System.Text.Encoding.UTF8; }
}
```

The writing is made a little more difficult in that the textbox control was likely created in another thread besides the one we are calling it from. In this instance, we need to use a special function to invoke the call on its original thread. This again makes use of a lambda expression. Assuming your textbox member variable is “output”, the code is something like this:

```
public override void Write(char value)
{
    base.Write(value);
    output.Dispatcher.BeginInvoke(new Action(() =>
    {
        output.AppendText(value.ToString());
    })
    ); // When character data is written, append it to the text box.
}
```

Now, you can add the following code somewhere to your main window constructor to redirect console output to the text box. (Change the name of txtConsole to whatever name you chose for your text box.)

```
// Redirect console
consoleWriter = new TextBoxStreamWriter(txtConsole);
Console.SetOut(consoleWriter);
```

Test out what you have so far. Craete a player party and enemy party as in program assignment 2 somewhere in your main window constructor. Add an ICombat member variable to your main window. Then add the following two lines.

```
combatThread = new CombatThread(encounter);
combatThread.Start();
```

If you run your program now, you should see your battle simulation from program 2 appearing in the text console.

Creating a custom control to display character state

Characters in our game have 4 states: Attacking, Defending, Idle, and Dead. We would like to display these graphically. There are many ways to do this. For example, we could simply use 4 images for each character and make the appropriate one visible when the character changes state. Instead, to show how to extend an existing control, we will create a class called CharacterImage which extends System.Windows.Controls.Image.

Create a public enum CharacterState for the 4 image states: Attacking, Defending, Idle, and Dead. Now create a member variable state to hold the character state and a public property State with a get and set. For now, fill in the default behavior of get and set to return/set the value of state.

Let us create 4 public members of type System.Windows.Media.ImageSource for the images as follows:

```
public ImageSource IdleImageSource { get; set; }
public ImageSource AttackingImageSource { get; set; }
public ImageSource TakeDamageImageSource { get; set; }
public ImageSource DeadImageSource { get; set; }
```

These images will be set in the xaml. Now lets add a protected method to update our Source member. (Remember, we are an Image, so we have a source just like System.Windows.Controls.Image does.)

```
protected void UpdateImageSource()
{
    switch (State)
    {
        case CharacterState.Attacking:
            this.Source = AttackingImageSource;
            break;
        case CharacterState.TakeDamage:
            this.Source = TakeDamageImageSource;
            break;
        case CharacterState.Dead:
            this.Source = DeadImageSource;
            break;
        case CharacterState.Idle:
        default:
            this.Source = IdleImageSource;
            break;
    }
}
```

Just to be sure the image source is up to date before rendering, lets override the OnRender method:

```
protected override void OnRender(DrawingContext dc)
{
    UpdateImageSource();
    base.OnRender(dc);
}
```

Finally, lets make sure when our state is changed (through the State property), the image that is displayed is changed as well. Modify your state property set method to do the following:

```
public CharacterState State {
    get { return state; }
    set
    {
        state = value;
        this.Dispatcher.Invoke((Action)(() =>
        {
            UpdateImageSource();
        }));
    }
}
```

Notice that we made it thread safe to change the State property. We could have had the caller do this instead.

Now compile your application and make sure there are no errors. Open the main window in the designer. Click on the toolbox. You should see a tab like "WPFBattle Controls". Under this, you should see your new CharacterImage control! You can drag it into the interface and use it just like any other control. Look at the properties. Under "Miscellaneous", you have IdleImageSource, AttackingImageSource, TakeDamageImageSource, and DeadImageSource automatically available to you. If you instead type in your xaml directly, these will be available in Intellisence. Here is the xaml for for a CharacterImage from my own implementation:

```
<custom:CharacterImage Tag="e1" x:Name="imgEnemy1" HorizontalAlignment="Left"
Height="128" Margin="117,367,0,0" VerticalAlignment="Top" Width="128"
IdleImageSource="Content/Riku_idle.png" AttackingImageSource="Content/Riku_attack.png"
TakeDamageImageSource="Content/Riku_hurt.png" DeadImageSource="Content/Riku_dead.png"/>
```

Decorating the Character classes

Ok, now you have all the tools you should need to make the GUI show character state! So far, I have given you much of a walkthrough like the worksheets, but here is where you have to do your own work.

We want all the behavior of the old character classes, but when you attack, now we would like to change the state of the character to reflect your current action. When the character is dead, it should also be appropraitely reflected in the interface.

1. Create new subclasses for each character class you have. These new classes should have the following functionality.
   a. They should have a constructor that takes a character image. The class will control the image to show the state of the character.
   b. You should override the ReceiveAttack function. When you receive an attack, change your character's image state to "Defending". Sleep for a small period of time, so you can graphically see the update. Call the base class implementation to resolve the attack. When done, if your health is zero change to the Dead state. Otherwise, change to the Idle state.
2. Create a new subclass for each attack type. The new character classes will use these new attack types. This new attack should also have a reference to the character's image.

   a. Override the attack function to update the image. When the character is attacking, change the image state to Attack. Sleep a period of time, so you can see the update.
   b. Call the base class method to resolve the attack.
   c. Finally, change the image back to Idle state.
3. Make sure your new character classes use these new attack types.
4. Create a CharacterImage object for each character and enemy in your combat simulation. Place each image on the interface on one side or the other.
5. Use the new enhanced Character classes for your characters when you create your encounter. (You should have to change none of the combat code because you programmed to Intefaces ICharacter and Iattack).
6. Build and run your sample combat application.