How to take screenshots in your game

Contributed by Mark Thomas
Wednesday, January 17 2007
Last Updated Thursday, January 18 2007

One handy feature in a lot of games is the ability to take screenshots. This tutorial walks you through the process of doing just that, using a component that you can download at the end of the tutorial. IMPORTANT: This only works for Windows games.

Before we get started, I'll say this again- this only works for Windows games. The component uses code that collects the startup path of the application. My testing has shown no problems with it, but your mileage may vary. Hot Seat, Ltd is not liable for any problems that may occur as a result of you using this code.

I've created a simple game library called hsScreenshot. As of this writing, it only contains one method, TakeScreenshot(). Before we get into the creation of the component, let's take a look at just how easy it is to use. All you'll need is a functioning XNA game. I'll be using the tutorial game that I modified in my first tutorial that covered keyboard and mouse input.

After opening your project, add a reference to hsScreenshot.dll, which you can download at the end of the tutorial. To add a reference, right-click on the project name in the Solution Explorer. Choose Add Reference... and select the Browse tab. Navigate to hsScreenshot.dll and click OK.

Now we need to set up code that will cause a screenshot to be taken. I want to do this when the user presses the F12 key. The tutorial game uses a routine called UpdateInput() to collect input. We just need to add a check for the F12 key, like so: //get the keyboard state

```
  KeyboardState curKbState = Keyboard.GetState();
//other input collection code removed for brevity
  //see if we want to take a screenshot
  if (curKbState.IsKeyDown(Keys.F12))
  {
   hs.Screenshot screenshot = new hs.Screenshot(graphics);
   System.Diagnostics.Trace.WriteLine(screenshot.TakeScreenshot("myScreen",".tga", ImageFileFormat.Tga));
  }
```

We instantiate an instance of our Screenshot object, sending our GraphicsDeviceManager to the constructor. Note that if you intend to take a lot of screenshots in your game, it may be advantageous to declare a Screenshot object that won't be created and destroyed every time. Also note that if the user holds the F12 key down, many screenshots will be generated. The code automatically increments the filename if the file already exists.

The TakeScreenshot() method has two overloads. If you call it with no parameters, you will get a jpg file called screen.jpg. Alternatively you can specify a filename, extension, and a file format. The method returns a string that, upon success, contains a message indicating the file was created, along with the filename. If there is an error it will return the Exception message. The screenshot(s) will be located in the folder in which your executable is located.

That's all there is to using this component! Now let's see how it's made.

I'll start by printing the contents of the class, and then will explain the code. public class Screenshot

```
{
 private GraphicsDeviceManager graphics;

 ///
 /// We need the GraphicsDeviceManager from our game class
 ///
 /// GraphicsDeviceManager 'graphics' from the Game class
 public Screenshot(GraphicsDeviceManager _graphics)
 {
  graphics = _graphics;
 }

 ///
```



```
 /// Takes a jpg screenshot with the default name of screen.jpg
 ///
 ///
 public string TakeScreenshot()
 {
  return TakeScreenshot("screen", ".jpg", ImageFileFormat.Jpg);
 }

   ///
   /// Captures the game window to a file. If the file exists, a number will be added to the name.
   ///
   /// Main part of the filename, example "screen"
   /// File extension with period, example ".jpg"
   /// ImageFileFormat, example ImageFileFormat.Jpg
   ///
 public string TakeScreenshot(string baseFilename, string fileExtension, ImageFileFormat imageFileFormat)
 {
  string retval = "";

  string filename = GetFilename(baseFilename, fileExtension);
  try
  {
   Viewport vp = graphics.GraphicsDevice.Viewport;

    Texture2D tex = new Texture2D(graphics.GraphicsDevice, vp.Width, vp.Height, 1, ResourceUsage.ResolveTarget,
SurfaceFormat.Color, ResourceManagementMode.Manual);

    graphics.GraphicsDevice.ResolveBackBuffer(tex);
    //note that jpg is very lossy.
    tex.Save(filename, imageFileFormat);
    retval = "Screen captured to " + filename;
  }
  catch (System.Security.SecurityException ex)
  {
   //most likely a privileges issue
   retval = ex.Message + "; Please verify that you have the proper privileges.";
  }
  catch (Exception ex)
  {
   retval = ex.Message;
  }
  finally { }

  return retval;
 }

 ///
 /// returns you the filename for your new screenshot.
 ///
 ///
 private string GetFilename(string baseFilename, string fileExtension)
 {
  //get our current folder (add a reference to System.Windows.Forms)
  string basepath = System.Windows.Forms.Application.StartupPath;
  //add a trailing slash if we dont already have one
  if (!basepath.EndsWith(@"\"))
   basepath += @"\";
  //filename will be the complete path + filename, e.g. c:\temp\screen.jpg
  string filename = basepath + baseFilename + fileExtension;
  // Make sure a duplicate file doesn&rsquo;t exist.  If it does, keep on appending an incremental numeric until it is unique
  int file_append = 0;
  while (System.IO.File.Exists(filename))
  {
   file_append++;
   //make a new filename including our integer. e.g. c:\temp\test1.jpg
```



```
   filename = basepath + baseFilename + file_append.ToString() + fileExtension;
  }

  return filename;
 }
}
```

At the top, we declare a private instance of the GraphicsDeviceManager. We'll need that to collect the screen information. Then we have our simplified overload of TakeScreenshot(), but let's get right to the main code. TakeScreenshot() requires 3 parameters: the base filename, the extension (INCLUDING the period!) and the ImageFileFormat. The format you choose is important; jpgs will take up less space, but are lossy and tend to look bad. Experiment to see what works for you.

First we call a helper function, GetFilename() that handles the dirty work of collecting our path and ensuring that we don't overwrite any existing files. Note that it uses code from System.Windows.Forms, so if you are recreating this component, you must make a reference to that library. A hint for those of you new to C# : putting the @ character in front of a string tells C# to treat the entire string as literal characters. Otherwise, it would treat the \ character as an escape character, and your path would have to have \\ for each \ that you wanted.

After we return with our filename, we get a reference to the Viewport. This will let us know the height and width of our image. Then we declare an instance of the Texture2D class. There are many arguments and options available when creating this class, so consult the MSDN documentation if you want details on all of them. Note that we are using ResourceUsage.ResolveTarget , which we will use to collect data from the BackBuffer. Yes, that's right: we aren't technically getting a screen capture here; we are getting a capture of the next screen. For most uses, this should suffice, but if you need something else, hopefully this points you in the right direction.

After calling the ResolveBackBuffer() method of our GraphicsDevice and sending it our Texture2D object, we call the Save() method of the Texture2D object, sending our filename and format. Finally, we return a string indicating succes or failure. That's it!

Download the compiled dll: hsScreenshot.dll

Download the source code: hsScreenshot source