IN379 Computer Game Architectures

Lab 2: Data-Driven Design

Data driven design allows game engines to separate the core engine logic with the game specific functionality. In fact, with many engines, you can create a game without writing a single line of C++ or C#. By supplying data to a game engine, we’ll be able to quickly prototype different solutions to a problem and delegate tasks to non-programmers (e.g. art tools for artists, level design tools for designers, etc.). This reduces costs and increases productivity, while encouraging loosely-coupled code design.

In this tutorial we’ll look at two ways of using C# to read data files and represent them appropriately in a game.

# Platformer Sample

We’ll be modifying the Platformer XNA sample. To get started download the package from:

[*http://xbox.create.msdn.com/en-US/education/catalog/sample/platformer*](http://xbox.create.msdn.com/en-US/education/catalog/sample/platformer)

Extract the zip file and open the Windows solution. Make sure it compiles and runs as expected.

The sample showcases a simple 2D platformer with 3 different levels (stored as .txt files). We will be modifying the level loading functionality to demonstrate some different ways we can make use of data files.

Open *Level.cs* in the *Platformer* project. This file contains the *LoadTiles()* function that handles the file loading. We’re going to expand upon this by creating a *Loader* class and using it to load in the levels. Have a look at the *LoadTiles()* function so you understand how it works in the original code.

# Adding the Loader Class

**First, remove the following code from LoadTiles() in *Level.cs*:**

using (StreamReader reader = new StreamReader(fileStream))

{

string line = reader.ReadLine();

width = line.Length;

while (line != null)

{

lines.Add(line);

if (line.Length != width)  
 throw new Exception  
 (String.Format("The length of line {0} is different from all preceeding lines.", lines.Count));

line = reader.ReadLine();

}

}

**Add the following lines where you removed the code, so that you know where to add code back in later (and to avoid a compilation error).**

//StreamReader code removed here

width = 0;

**Now add a new class to the project (Right click the project file and select Add > Class). Name it “*Loader.cs”* and give it following implementation (add the code in bold):**

public class Loader

{

**// Stream we are going to use for file reading**

**private Stream mFileStream = null;**

**// Constructor**

**public Loader(Stream stream)**

**{**

**mFileStream = stream;**

**}**

}

**You will need to add the last “*using”* statement in the list below at the top of the file (System.IO).**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

**using System.IO;**

The *Stream* object is used to give us a handle to the data file that we want to look at. We will be using *StreamReader* objects to read it.

# Loading From Text Files

Reading text files is one of the easiest ways to load data into a game. This data can consist of variables that the engine can read, such as speed of enemies or how much health the player has. The text could also represent a level, or even an output file for writing high scores or save data to.

Before we make use of our *Loader* class, we need to give it some sort of functionality. Let’s try to read the entire stream and return it as a string. Here’s a code snippet that will open and read a text file and return it as a string.

**Add a function called *ReadTextFileComplete()* to the *Loader* class as follows:**

**public string ReadTextFileComplete()**

**{**

**//The "correct" (most efficient) way to concatenate strings in .NET**

**//is to make use of a StringBuilder object instead of using the**

**//addition operator (e.g. string s = s + "foo";)**

**StringBuilder result = new StringBuilder();**

**// Use a try-catch block to make sure any exceptions are handled**

**// E.g. File not found errors etc.**

**try**

**{**

**// The using statement here creates a single StreamReader object**

**// and properly disposes of it after it has finished being used**

**using ( StreamReader reader = new StreamReader( mFileStream ) )**

**{**

**// Add the text of the whole file (ReadToEnd()) to the**

**// resulting string**

**result.Append( reader.ReadToEnd() );**

**}**

**}**

**catch ( Exception e )**

**{**

**// If we've caught an exception, output an error message**

**// describing the error**

**Console.WriteLine( "ERROR: File could not be read!" );**

**Console.WriteLine( "Exception Message: " + e.Message );**

**}**

**// Return the resulting string**

**return result.ToString();**

**}**

This is a straightforward function that makes use of .NET’s existing functionality with *StreamReader* objects.

**Now we have something that we can use, try to add a *Loader* member to the *Level* class. Remember to initialise it with the *Stream* passed into the *Level* constructor. Add a call to the *ReadTextFileComplete()* function to get a string of the whole file immediately after the initialisation. We are not going to do anything with this at the moment, but you can add a break point to make sure the call has worked.**

This is not very useful at the moment. Is it easy to figure out which values represent what? Notice that the *LoadTiles()* function makes use of a *List<string>* collection.

**As an extra task at the end of the lab (not now), you could try to split the string into lines and add it to a *List<string>* collection (you can see the syntax of the collection in the next section). Hint: *strings* have a *Split()* function and the ends of lines on Windows are marked by the character sequence “\r\n”.**

# Reading Text as Lines

A neater method of delivering lines of text is to read the lines one at a time from the file, keeping track of each line as it is read.

**Add another function called *ReadTextFileAsLines()* to the *Loader* class as follows:**

**public List<string> ReadLinesFromTextFile()**

**{**

**// We don't have to worry about string building here, as we are only**

**// reading a line at a time**

**string line = "";**

**// Initialise a list to contain the results**

**List<string> lines = new List<string>();**

**try**

**{**

**using (StreamReader reader = new StreamReader(mFileStream))**

**{**

**// Now we'll keep reading until the end of the file and**

**// store each line in a collection**

**while ( ( line = reader.ReadLine() ) != null )**

**{**

**// Add the line to the collection**

**lines.Add(line);**

**}**

**}**

**}**

**catch (Exception e)**

**{**

**Console.WriteLine("ERROR: File could not be read!");**

**Console.WriteLine("Exception Message: " + e.Message);**

**}**

**return lines;**

**}**

**Now modify the *LoadTiles()* function to make use of this new function so that the tiles are loaded correctly. You will need to add this code where you removed it earlier, replacing the lines you added:**

//StreamReader code removed here  
width = 0;

**Notes:**

1. **You will need to comment out your earlier call to *ReadTextFileComplete()* before running this code.**
2. **Initialise the width variable with the length of the first line after you have read in the lines.**

**Compile and run to see if it worked.**

# Modifying the Data

We should have the level loading successfully and the game running normally. Take a look at the *LoadTiles()* function again and you’ll notice that for each character loaded in the text file, the *LoadTile()* function is called. Within this method you will see a *switch* statement that determines which character was passed into the *tileType* parameter and load the appropriate tile.

**Try changing some of the characters in the *switch* statement and modify the level file(s) to use them (level files are .txt files under *Content > Levels* in the solution explorer). For the moment this can represent a copy of one of the elements that are already in the game.**

# Loading from XML Files

We’ve seen how text files can be read and parsed using some straightforward methods in C#. Another common file format used to represent data is XML (eXtensible Markup Language). XML is human readable, easy to understand, and can be used to represent more complex data.

One of the problems with plain text files is that it’s difficult to represent structured data. For example: an enemy can have complex properties such as a set of weapons it is carrying, what kind of armour it is wearing, what kind of spells it can use, etc. Each of these properties can also have their own properties, which in turn can have their own properties, and so on. It’s difficult to represent this kind of complexity in a text file unless you sacrifice a bit of readability.

Fortunately, C# provides us with built-in XML reader classes that we can make use of to read and parse XML files. The easiest method for reading an XML file into a data structure is to make use of the *XmlSerializer* class. We’ll use an XML file to change some constants in the game. But first let’s do a little preparation.

**Add a new class to the project called “GameInfo.cs”.** **Add *“using Microsoft.Xna.Framework;”* at the top and add the following code within the namespace:**

**public class EnemyInfo**

**{**

**public float Speed = 0.0f;**

**}**

**public class GameInfo**

**{**

**private static GameInfo mInstance = null;**

**public static GameInfo Instance**

**{**

**get**

**{**

**if( mInstance == null )**

**mInstance = new GameInfo();**

**return mInstance;**

**}**

**set { mInstance = value; }**

**}**

**public EnemyInfo EnemyInfo;**

**}**

We’ll use this as a global so that we can easily access various game constants from anywhere in the code. C# doesn’t support global variables, but the static member variables can be used in the same way. This is an example of the Singleton design pattern.

**In our *Loader* class add the following *using* statements at the top:**

**using System.Xml;**

**using System.Xml.Serialization;**

**Next, create a new function called *ReadXML()* in Loader, and add the following code:**

**public void ReadXML(string filename)**

**{**

**try**

**{**

**using (StreamReader reader = new StreamReader(filename))**

**{**

**GameInfo.Instance = (GameInfo)new XmlSerializer(typeof(GameInfo)).Deserialize(reader.BaseStream);**

**}**

**}**

**catch (Exception e)**

**{**

**// If we've caught an exception, output an error message**

**// describing the error**

**Console.WriteLine("ERROR: XML File could not be deserialized!");**

**Console.WriteLine("Exception Message: " + e.Message);**

**}**

**}**

Let’s take a look at the code. Most of it we have seen before. The bit that does the really interesting stuff is:

GameInfo.Instance = (GameInfo)new XmlSerializer(typeof(GameInfo)).Deserialize(reader.BaseStream);

This creates an *XmlSerializer* object and specifies the type of object it wants to create as a result (the **typeof** operator). It then loads the file stream and “deserializes” it – i.e. translates it to a machine readable state. After a simple cast to the *GameInfo* object, we have our resulting info. Of course, we need some XML to load, so let’s create some.

**Add a new XML file to the Content project (not the game) by selecting Add > New Item... and select XML File. Name it “info.xml”**

We’re not going to be compiling this file, but it needs to be in the content project because we need it to be copied to our output directory after compiling the game.

**Select info.xml and go to its Properties (right click > Properties) and do the following under the Advanced heading:**

1. **Change “Build Action” to None**
2. **Change “Copy to Output Directory” to Copy Always**

We’ve added an *EnemyInfo* member inside the *GameInfo* class, which contains a value for *Speed*. As you might guess, this will control how fast enemies move in the game. Let’s add the XML first which will set the speed to 1000 (very fast).

**Replace the contents of info.xml with the following:**

<?xml version="1.0" encoding="utf-8" ?>

<GameInfo>

  <EnemyInfo>

    <Speed>1000</Speed>

  </EnemyInfo>

</GameInfo>

NOTE: The tags in the XML match the names of the member variables. So if the member “WaitTime” was added to *EnemyInfo*, it would have a corresponding <WaitTime> tag in the XML.

**Now that we have created an XML reading function, modify the *Level* constructor to load the XML file. Note: you will need to define the path as** "Content/info.xml" **because the xml file is contained in the Content directory (we could add a new directory for xml files to neaten things up).**

**Once this has been loaded, modify the *Enemy* class by removing the MoveSpeed constant and making use of the Speed variable loaded in through the XML. GameInfo uses the “Singleton” design pattern, so you will need to access the Speed variable as follows:** GameInfo.Instance.EnemyInfo.Speed.

**Extra challenge - Create the appropriate class for the following XML (in bold) and modify the game to make use of it (also change the enemy speed to something sensible):**

<?xml version="1.0" encoding="utf-8" ?>

<GameInfo>

  <EnemyInfo>

    <Speed>64</Speed>

  </EnemyInfo>

**<GemInfo>**

**<BounceHeight>0.18</BounceHeight>**

**<BounceRate>3.0</BounceRate>**

**<Color>**

**<R>255</R>**

**<G>0</G>**

**<B>0</B>**

**<A>127</A>**

**</Color>**

**</GemInfo>**

</GameInfo>

We’ve seen two ways of reading and interpreting data files for use within our games now. There are other file formats available (JSON, binary files), but this tutorial has supplied you with the basics of how you can make use of a data-driven approach within your games.