Generic Dynamic DataContainer for Rapid Game Development

by

David Z. Whittaker

B.Sc. University of Victoria, 2007

A Thesis Submitted in Partial Fullfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© David Z. Whittaker, 2009

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

Generic Dynamic DataContainer for Rapid Game Development

by

David Z. Whittaker

B.Sc. University of Victoria, 2007

Supervisory Committee

Bruce Gooch (Department of Computer Science)

Supervisor

Amy Gooch (Department of Computer Science)

Departmental Member

Dr. Brian Wyvill (Department of Computer Science)

Departmental Member

Supervisory Committee

Bruce Gooch (Department of Computer Science)

Supervisor

Amy Gooch (Department of Computer Science)

Departmental Member

Dr. Brian Wyvill (Department of Computer Science)

Departmental Member

Abstract

In this thesis I detail an easy to use system for dynamic data storage aimed at use in educational games and game systems. Evaluation of current approaches in use by educators and students suggest a large degree of ad hoc systems in place for data persistence. I evaluate my system against existing techniques including a flat file, XML, and SQL systems. I provide a dynamic, easy to use solution to data persistence in the environment of educational game development.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Educational games, and games for education are an increasingly prevalent topic in computer science [Wolz et al. 2006; Parberry et al. 2006]. With the rise of interest in using games as educational tools, as well as teaching computer science through the design and programming of games more and more students and educators are getting involved in game development. Game development involves a collection of various non related often complex subject areas including graphics, art, sound, design, core logic etc. New technologies such as Microsoft's XNA are making programming many of those components more accessible than ever before [Linhoff and Settle 2008]. What is often overlooked is the need and method for data persistence [Jones 2000; Burns 2008; Coleman et al. 2005]. Whether in the form of a high scores table, save game files, or user performance records data persistence provides important functionality to most if not all game systems.

Data persistence raises several difficult questions. What do you need to store? Where should the data be stored? How should that data be stored? What do we do when the data requirements change? I have seen three common approaches to these problems. The first approach is to take advantage of the built in serialization in the language being used. This approach is quick to implement, but when the structure is changed older data becomes unreadable. The second approach is to define specific purpose data structures for each object that needs to be stored. Then write that data out to a file in a binary format for speed, or a plaintext format for human readability. This approach although functional is limited and rigid requiring specific data writing and reading code for each object. When the data requirements change this approach requires changes to the code in multiple locations, and

is again often no longer compatible with data previously written. The third approach is to connect the game directly to an SQL server [Leutenegger 2006]. This approach is much less rigid and can make development easier but comes with the requirement of having to download, setup and run an SQL server where ever the game is deployed.

Each of these solutions has benefits and shortcomings. In this thesis I present a method for storing data that combines the benefits of each of the above methods and adapts easily to the changing requirements encountered in game development.

Chapter 2

Background and Previous Work

2.1 Data Serialization Methods

Data serialization is the process of converting an object to a form for storage or transmission. There are many common approaches for data serialization, each approach has its pros and cons. In this section I detail some of the common approaches for data serialization.

2.1.1 Built-in Serialization

Languages such as C# and Java are commonly used for educational game development [Adams 1998]. These managed languages provide easy to use built-in object serialization. The C# Serializable allows for any object marked as serializable to be serialized to binary or xml. The serialized object can then be deserialized back into an object when required. The benefits to this method are it's easy to use, very accessible and is a no fuss method of storing almost any datatype. The largest drawback is in order to de-serialize the object types need to match. If you change the definition of an object you will no longer be able to deserialize objects serialized in an earlier format. This can cause a considerable problem if you plan to distribute the end result as any upgrading you do would invalidate all the users existing files.

2.1.2 Custom Binary

A custom binary format is another easy method for storing data, although slightly more advanced than using the built-in serialization. By defining a header, some data type flags,

and data lengths storing your data into a binary file is a fairly straight forward procedure. However it suffers from being difficult to debug, and can be prone to off-by-one errors. A change in data requirements will require changing the storing and loading functions as well as changing the header definitions. This can again suffer from non-backward compatibility unless carefully thought out.

2.1.3 XML

XML is another common format. It comes with 2 major benefits over the built-in serialization and binary approaches. First, it is human-readable and human-editable, this can provide a huge advantage when debugging your data output. Second, it is very easy to implement your XML structure in a way that can still be read even after the data format changes. In other words, backward compatibility is easily supported. This unfortunately comes with two major disadvantages. The XML format is not size or speed efficient. Each opening and closing tag can take up more space than the data it is storing. Also, because it is a text format, all numeric data has to be parsed before it can be used making initial access to data slow. Using XML for data storage is good for small applications but scales it very poorly.

2.1.4 SQL

A less common approach is to forget trying to store your data out to file and connect directly to an SQL server. This approach may be the easiest and most flexible for development. It comes with several distinct advantages. First, manipulating your data structures is done through powerful and robust database management software, and does not require any code changes. Second there are many well written and tested database interface libraries that are guaranteed to work, so hooking your game to the database can be a trivial task. Lastly, SQL provides a powerful way to manipulate and report on your data. The obvious disadvantage

to an SQL approach is the required SQL server, if distribution of your game is the intended result the SQL server would have to be packaged with it.

2.2 Other Available Systems

There are two notable existing database systems that on windows have C# bindings, SQLite, and Berkeley DB. Both systems can be used for data persistence in games, however both suffer from non-trivial short comings when used in the context of educational gaming and student developers.

2.2.1 SQLite

SQLite implements a large subset of the SQL-92 standard for SQL, but leaves out portions of the alter command that make it more difficult to change data types or data structure during development. It also requires that you interface with it using SQL instead of a direct key, value pair mapping that makes more sense from a game development perspective.

2.2.2 Berkeley DB

Berkeley DB provides an incredibly robust database system accessed in a direct key, value pair mapping that makes it more ideal for game usage; however Berkeley DB is a feature complete database system offering a significantly more complicated, and confusing system than necessary for beginner/student developers.

2.3 Previous Work

2.3.1 Art and Science of Game Programming

The paper titled *Art and Science of Game Programming* [Parberry et al. 2006] describes an undergraduate game programming course. Limited detail is provided regarding data persistence, but use of the windows registry is described to store settings and high scores. This approach, while quick and easy, is not suitable for storing more than a small collection of values.

2.3.2 When SQL Makes Sense

In the paper titled *A CS1 to CS2 Bridge Class Using 2D Game Programming* [Leutenegger 2006] a game programming course using Adobe Flash to create web games is described. Although not directly taught in the course some of their students hooked their web based game to a MySQL database for storing global high scores. It is important to note that in this context a central SQL based persistence mechanism makes the most sense as your central server is always in the equation. Even if the game itself is being distributed providing a global scores or ranking mechanism requires a central server. The easiest way to interface with this server is through an SQL based system.

2.3.3 Absence of Data Persistence

In many of the other recent game programming, and game course papers [Jones 2000; Burns 2008; Coleman et al. 2005] Data persistence is glossed over or ignored completely. The other aspects, object oriented design, graphics, interactivity, sound, networking, and user input are all covered in detail. In *Teaching Game Programming Using XNA* [Linhoff and Settle 2008], focus is placed on the importance of creating many different game resources, such as models, sprites, and sound effects. What is missing is the mechanism to

tie these resources together into a game level. My system provides this missing mechanism.

Chapter 3

System

The intent of my system is to provide the benefits of the Binary, XML, and SQL approaches while minimizing or eliminating their drawbacks. The design of my system focused on three major requirements. First it needed to be able to dynamically change the data definitions without requiring any changes to saving and loading code. This is important during the early stages of development when data structures are more likely to change. Second, data that was previously saved needed to remain backward compatible. Backward compatibility is incredibly important if you want to distribute your program, data should not be lost when users upgrade to a later version. Lastly, data needs to be output in multiple formats, XML for debugging, Binary for performance, and SQL for syncing with a master server.

I achieved these requirements by defining a dynamic data container object that functions similarly to an SQL database table. The dynamic data container has a data definition similar to an SQL table definition. Inside the data container are data records that match that definition and are stored as objects with fieldname to data mappings. What makes the container dynamic is that its data definition can be redefined on the fly. This is done by simply adjusting the data mappings stored in each record. The data structure is designed in such a way that it can be outputted in a naturally to a nested XML format, a nested binary format, and as SQL to store in a local or remote database.

3.1 Components

The core component of the system is the DataRecord. The DataRecord contains a complete definition of the stored data, the data itself, and functionality for parsing, loading, and saving. The DataRecords belong to a DataContainer which is a collection that contains functionality for saving and loading the DataContainer, as well as creating, removing, and retrieving DataRecords. The DataContainer also maintains the primary data definition, and the auto incrementing record ID. At the top level each DataBase contains a collection of DataContainers and contains functionality for starting the save and load processes as well as creating, removing, and retrieving DataContainers.

3.2 Programming Interface

The programming interface is designed to be as simple as possible. Initially a database object would need to be created or loaded. The database object itself is just a container for the container objects and has loading, saving, and code for creating and getting the DataContainer objects:

- LoadDB - Loads all of the containers and records for a given database from ether Binary, XML, or an SQL database.
- SaveDB - Saves the current database structure to Binary, XML, or to an SQL database.
- GetContainerNames - Returns a list of all the containers in the database
- GetContainer - Gets a container by name.
- GetNewContainer - Creates a new container, adds it to the database and returns a reference to it to the calling function.

The database object allows for creating and getting the DataContainer objects that provides the interface for creating and getting individual records. Important functionality in DataContainer includes:

- GetNewRecord - Creates a new record object using the current data definition, adds it to the datacontainer and returns a reference to the record to the calling function.
- GetRecords - Returns a list of all the records ordered by record ID.
- SetDataDefinition - Changes the definition of the container.

The DataRecord objects that are returned from the DataContainer have a very simple get and set interface:

- GetField - Gets the value of a field by name. The return type is dynamic and is controlled by the type passed in as a reference.
- SetField - Sets the value of a field by name.

The DataRecord also offers a [] operator that provides an associative array style interface to the record data. This operator provides simple access and provides for cleaner shorter code.

The functions listed are only a subset of the functionality, but represent the important interfaces to the system.

3.2.1 Creating A Database and DataContainer

Creating a DataBase object in the system is actually trivial. No parameters are required, a simple call to new will create a new empty DataBase system ready to load or to create DataContainers. In order to create a DataContainer you need to define the initial definition of the data as a string similar to how you would create a table with SQL. The data definition

is of the format field_name data_type, field_name data_type, etc. and the current system supports int, float, string, bool, and date. A call to GetNewContainer with an unused name, and correct data definition will create a new DataContainer, add it to the database, and return this container to you for use.

3.2.2 Creating Records

Once you have created a DataContainer creating a new record is done by calling GetNewRecord on the DataContainer. This creates a new record, assigns it a unique record ID, sets the definition of the record to match the DataContainer, adds the record to the container, and returns the record for use. The DataRecord returned is a reference to the record in the container, so updating the records data automatically updates the data in the database. Setting or updating the data is done by indexing the record as though it were an associative array eq. Record["first_name"] = "David". Data is also retrieved in this manor but requires a cast eq. String firstName = (String)Record["first_name"].

3.2.3 Saving

To save the database a call to Save is made on the DataBase object. Save requires a filename, and a type enumeration. The available types are XML, BINARY, and SQL. Each type outputs in the expected format, XML for human readable data, Binary for performance, and SQL for inserting into an another database system.

3.2.4 Loading and Updating Definitions

To load a database a call to Load is made on the DataBase object. Load requires a filename, and a type enumeration (the same as save). In this case the SQL option is currently unavailable. One of the most important features of the system is that the data definitions can be

changed on the fly. When you load a DataBase that is expected to be a specific format the first thing to do is confirm the data definitions are compatible with the current version of your system. This is the case where you have changed definitions by adding or removing fields, or changed the database by adding or removing tables. A call to SetDataDefinition on each of the DataContainers with an up to date definition will update the definitions, adding and removing columns as need. In the case that the definitions have not changed no work will be done. This allows older versions of your database to be loaded without issue, although some fields may end up with default (empty) values.

3.2.5 Error Handling

The system throws a variety of exceptions in the case of error including: InvalidDataDefinition, ColumnNotFound, ContainerDoesNotExist, ContainerNameAlreadyExists, FileNotFound. These cover the primary cases where programmer use of the system can go wrong.

3.3 Data Output

The system outputs to file as XML or Binary. The XML output is a nested XML structure containing each data container, the data definition, and each of the records in the container. By its nature the XML allows for partial or incomplete records. Only the nodes that are set need to be read in which allows for the dynamic changes to data types to only affect new or updated records, and allows old non updated records to still be loaded.

Sample XML output:

```
<XML>
<DataContainer>
<ContainerName>name</ContainerName>
<AutoIncrementValue>1</AutoIncrementValue>
```

```
<DataDefinition>definition </DataDefinition>
<Record>
<RecordID>0</RecordID>
.... record data ....
</Record>
....
</DataContainer>
....
</XML>
```

The Binary output is similar to the XML output, outputting each container and records in chunks. The record field names and types are also included with each record. This is deliberate so the binary data can like the XML data contain partial records. This does make the binary output larger but much more robust and less sensitive to data definition changes.

Sample Binary Output Format:

```
Int32 containerCount
Int32 containerSize
Int32 containerNameLength
Char[] containerName
Int32 dataDefinitionLength
Char[] dataDefinition
Int32 recordCount
Int32 recordSize
... Record Data ...
Int32 fieldNameLength
Char[] fieldname
```

```
Int32 fieldType
Int32 fieldLength
Byte[] fieldData
...
...Record Data...
```

3.4 Internal Workings

3.4.1 Data Storage Retrieval

Internally a data record contains two maps. Maps are string to value data containers, in C++ they are referred to as maps, and in C# they are referred to as Dictionaries both terms describe a data container that has a string key mapping to a typed data value. The two maps used inside the data record are a data type map containing enumerated values representing the type of data to be stored in each field, and the data map, which contains a string converted value of the actual data being stored. When a field is set the type is cast to a string using the built in string operations and stored in the data map as a string. When a field is requested the type is checked from the type map and then a specific string-to-type function is called to convert the data from a string back to its original type. In C# most of the built in data types such as int, float, date, bool etc. have built in parse functions for parsing strings. The string-to-type functions largely take advantage of the built in parsing to parse back to the original data type.

3.4.2 Definition Updates Loading Out of Date Files

One of the most powerful features of the system is the ability to load without issue older definition tables and update them to match your current definition. The definition of record

is stored in its type map, by adjusting the type map the way the data is accessed is changed. When an old file is loaded its type map is also loaded, but then the system can update that map to match its new definition by simply changing the type map. This change will cause the parsing of that field to change. For example if you had a field that was previously an int and you changed the type to float the parsing function that is called would now try to parse that field as float. Keeping in mind that the data itself is stored as a string and parsing from int to float is no trouble for the system. If an incompatible assignment has occurred a field request would result in invalid data being detected and a default value being returned. All systems using the db and taking advantage of type changes have to be aware of the possible default return.

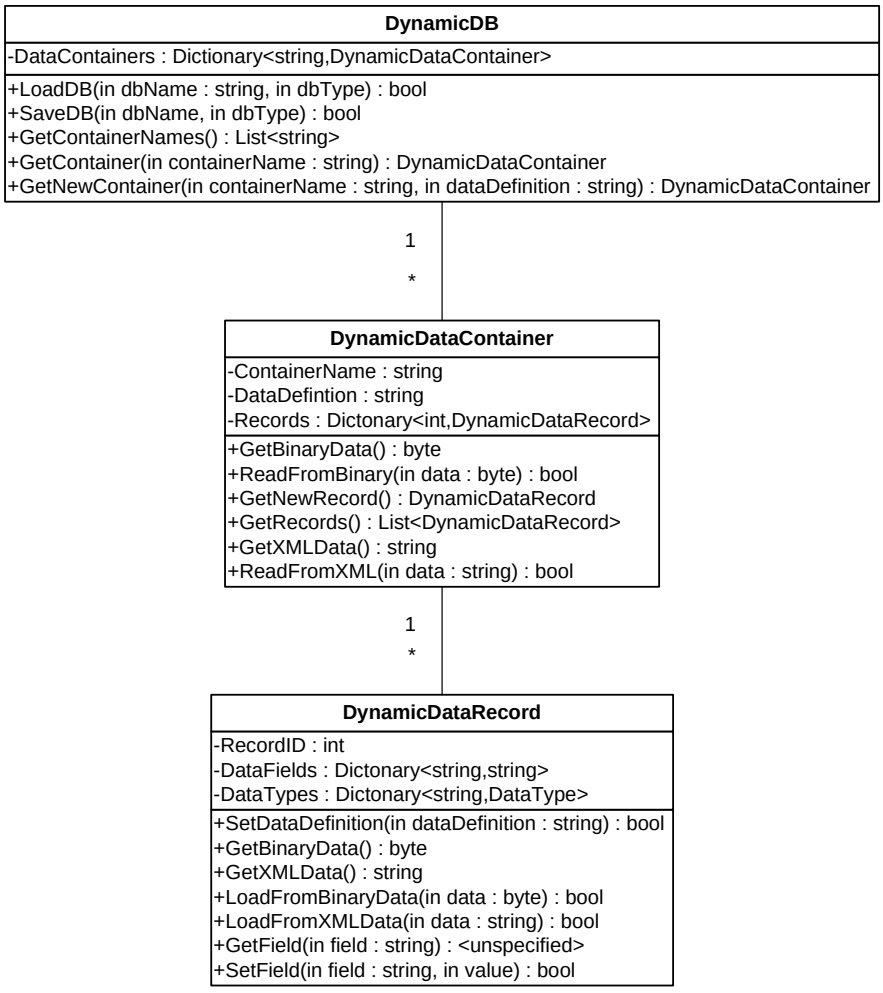

Figure 3.1: Basic class structure of the database system (only important properties and functions listed).

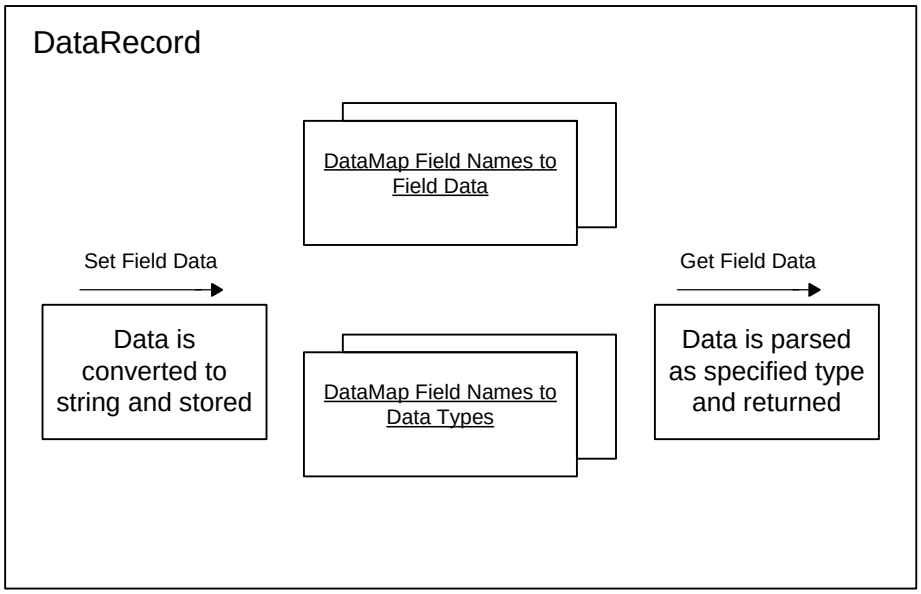

Figure 3.2: The DataRecord

Chapter 4

Results and Discussion

4.1 User Evaluation

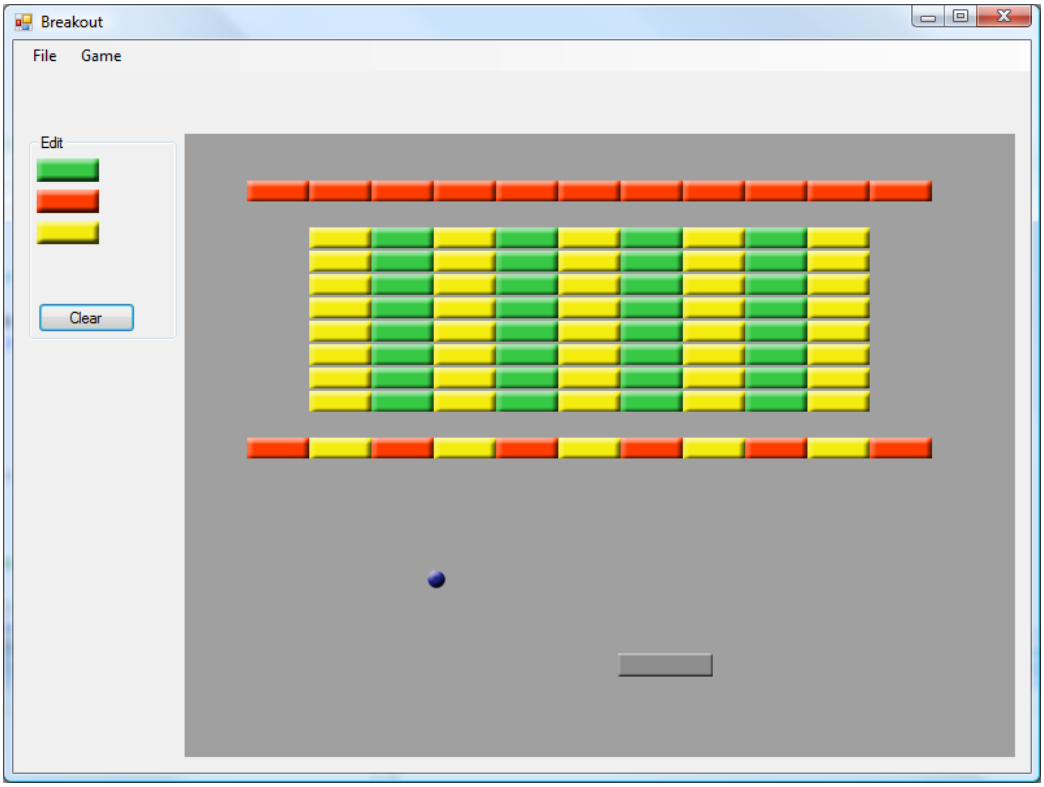

Figure 4.1: Evaluation Game

The system is designed to be an easy to use data persistence tool for student game developers. To evaluate the system a data persistence task was performed by 10 computer science students using XML, Binary, and the database system. The purpose of the evaluation was to determine how easy, and how fast each method of data persistence was to implement.

4.1.1 User Evaluation Task

The evaluation task presented the students with a simple breakout style game (bricks, ball, and paddle). The game had a built in editor to facilitate the creation of new levels. The save and load functionality of the editor was presented as code stubs to be filled in by the students as XML, Binary, and using my database system. The students were also presented with examples of data persistence in all 3 formats. When the coding task was completed the students completed a survey with questions regarding the perceived difficulty and time required to complete each data persistence method, as well as method preference and previous experience. The instructions and survey given can be found in appendix A.

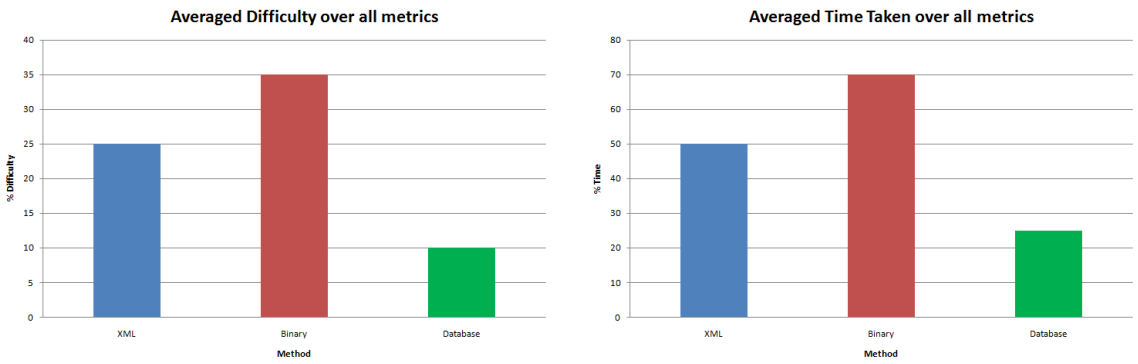

Figure 4.2: Left average difficulty rating, right average time rating

4.1.2 Results

The average time to complete the entire exercise was 45 minutes, with 20 minutes and an hour and 30 minutes being the outliers. Largely the discrepancy in completion times lined up with previous experience. The students ranged from 3rd year computer science to graduate level computer science students. The slowest student was a 3rd year, and the fastest was a graduate student.

The average test user had minimal experience in all three methods of data output as shown in Table 4.1. All of the test users completed the evaluation successfully.

The data entry and retrieval results shown in Table 4.2 and Table 4.3 and summarized in Figure 4.2 generally rated XML the hardest and requiring the most time. Comments suggested this was largely due to an unfamiliarity with how XML worked and more thought being required to lay out the data structure. The database system was generally rated the easiest and requiring the least time. Binary was generally in the middle for both difficulty and time.

In the follow up questions binary was generally rated as thought to take the most time and to be the hardest to change after the fact. The database system was rated as thought to be the easiest and to take the least amount of time to change.

When asked to rank the systems for future use the resulting ordering was my database system then XML and lastly binary. Comments indicated binary was no fun to debug when done incorrectly.

4.1.3 Typical Saving and Loading Code

Below are Listings 4.1, 4.2, 4.3, 4.4, 4.5, and 4.6 these are the typical results for each of the functions the students were asked to write in the evaluation.

Listing 4.1: Typical XML Saving Code

```
public void LoadLevelXML(string fileName)
{
        //clear the bricks
        currentBricks = new List<Brick>();

        XmlTextReader reader = new XmlTextReader(fileName);
        while (reader.Read())
        {
                switch (reader.NodeType)
                {
                        case XmlNodeType.Element: // The node is an element.
                        if (reader.Name == "Brick")
                        {
                                int brick_x = int.Parse(reader[0]);
                                int brick_y = int.Parse(reader[1]);
                                int brick_hits = int.Parse(reader[2]);
                                Brick toAdd = new Brick(brick_x, brick_y, brick_hits);
                                currentBricks.Add(toAdd);
                        }
                        break;
                }
        }

        reader.Close();
        RefreshVisibleBricks();
}
```

Listing 4.2: Typical XML Loading Code

```
public void SaveLevelXML(string fileName)
{
        FileStream fs = File.Create(fileName);
        TextWriter tw = new StreamWriter(fs);


        tw.WriteLine("<XML>\n");
        for (int i = 0; i < currentBricks.Count; i++)
        {
                string toWrite = "<Brick X= '" + currentBricks[i].X + "'
                Y= '" + currentBricks[i].Y + "' BrickHits= '" +
                currentBricks[i].BrickHits + "'></Brick>";

                tw.WriteLine(toWrite);
        }
        tw.WriteLine("</XML>\n");

        tw.Close();
        fs.Close();

}
```

Listing 4.3: Typical Binary Saving Code

```
public void SaveLevelBinary(string fileName)
{
        FileStream fs = File.Create(fileName);
        BinaryWriter bw = new BinaryWriter(fs);

        //Write the number of records that we are writing out
        bw.Write(currentBricks.Count);

        //Output binary data here
        for (int i = 0; i < currentBricks.Count; i++)
        {
                int brick_x = currentBricks[i].X;
                int brick_y = currentBricks[i].Y;
                int brick_hits = currentBricks[i].BrickHits;

                //write the brick x coord
                bw.Write(brick_x);

                //write the brick y coord
                bw.Write(brick_y);

                //write the brick hits
                bw.Write(brick_hits);
        }

        //close the file
        bw.Close();
        fs.Close();

}
```

Listing 4.4: Typical Binary Loading Code

```
public void LoadLevelBinary(string fileName)
{
        //clear the bricks
        currentBricks = new List<Brick>();

        FileStream fs = File.Open(fileName, FileMode.Open);
        BinaryReader reader = new BinaryReader(fs);

        //Read in binary data
        int brickCount = reader.ReadInt32();

        //for the length of the data
        for (int i = 0; i < brickCount; i++)
        {
                int brick_x = reader.ReadInt32();
                int brick_y = reader.ReadInt32();
                int brick_hits = reader.ReadInt32();

                Brick toAdd = new Brick(brick_x, brick_y, brick_hits);
                currentBricks.Add(toAdd);
        }


        //Close the file
        reader.Close();
        fs.Close();

        RefreshVisibleBricks();
}
```

Listing 4.5: Typical Database Saving Code

```
public void SaveLevelDB(string fileName)
{
        //Create a new database
        DynamicDB db = new DynamicDB();

        DynamicDataContainer container = db.GetNewContainer("Bricks",


        for (int i = 0; i < currentBricks.Count; i++)
        {
                //Create a new record
                DynamicDataRecord brickRecord = container.GetNewRecord();

                //Set the fields in the record
                brickRecord["X"] = currentBricks[i].X;
                brickRecord["Y"] = currentBricks[i].Y;
                brickRecord["BrickHits"] = currentBricks[i].BrickHits;
        }

        //Save the Database in a binary format
        db.SaveDB(fileName, DynamicDB.DBType.BINARY);
}
```

Listing 4.6: Typical Database Loading Code

```
public void LoadLevelDB(string fileName)
{
        //clear the bricks
        currentBricks = new List<Brick>();

        //Create a new database object
        DynamicDB db = new DynamicDB();

        //Load the database from a binary format
        db.LoadDB(fileName, DynamicDB.DBType.BINARY);

        //Get the data container named "Bricks"
        DynamicDataContainer container = db.GetContainer("Bricks");

        //Get the records from the container
        List<DynamicDataRecord> records = container.GetRecords();

        //Itterate over each record
        for (int i = 0; i < records.Count; i++)
        {
                //Load the data into local variables result must
                //be cast, as it returns type object.
                int brick_x = (int)records[i]["X"];
                int brick_y = (int)records[i]["Y"];
                int brick_hits = (int)records[i]["BrickHits"];

                Brick toAdd = new Brick(brick_x, brick_y, brick_hits);
                currentBricks.Add(toAdd);
        }

                RefreshVisibleBricks();
}
```

4.2 Performance

4.2.1 Testing Platform

All tests were performed on my lab machine which is an Intel Core 2 Duo 3.0GHZ with 3GB of ram.

	None	Minimal	Moderate	Expert
SQL	0%	20%	40%	40%
Plain Text	0%	0%	60%	40%
XML	20%	20%	20%	40%
Binary	60%	0%	0%	40%

Table 4.1: Prior Experiance With Data Persistence Methods

	XML		Binary		Database	
	Easy	Hard	Easy	Hard	Easy	Hard
Setup	80%	20%	40%	60%	100%	0%
Data Entry	60%	40%	80%	20%	80%	20%
Data Retrieval	80%	20%	80%	20%	100%	0%
Structure Adjustment	80%	20%	60%	40%	80%	20%

Table 4.2: Difficulty Rankings for Each Method of Data Persistence

	XML		Binary		Database	
	Short	Long	Short	Long	Short	Long
Setup	60%	40%	20%	80%	60%	40%
Data Entry	40%	60%	40%	60%	80%	20%
Data Retrieval	60%	40%	20%	80%	80%	20%
Structure Adjustment	40%	60%	40%	60%	80%	20%

Table 4.3: Time Rankings for Each Method of Data Persistence

Rank	XML	Binary	Database
1	20%	20%	40%
2	20%	40%	60%
3	60%	40%	0%

Table 4.4: Future Project Usage for Each Method of Data Persistence

Record Count	Create Time (Average of 5 Trials)
10,000	58ms
100,000	507ms
1,000,000	5,858ms

Table 4.5: Record Creation Time - Time to generate and insert records into the database.

4.2.2 Performance Metrics

There are many different metrics for measuring performance of a database system. For this system the important metrics are speed of record creation and insertion, speed of record retrieval, speed of saving the system, speed of loading the system, file size of the stored system, and size of the system in memory.

4.2.3 Generating Random Records

To test the system three sets of random records are generated, the randomly generated records represent a user description and consist of three string fields, two number fields and one date field. Sets of 10,000, 100,000 and 1,000,000 records are generated and inserted into the database. On my test platform it took just over five seconds to generate and insert 1,000,000 records as shown in Table 4.5 which is 0.005ms per record.

4.2.4 Xml Vs Binary

Twelve performance tests were performed three saving and three loading the database as XML and then three saving and three loading the database as Binary. Each test was preformed five times and the results averaged. For each test the file size and the time in ms was recorded.

The saving tests found for smaller record counts (10,000, and 100,000) the XML speed performance to be very close to the Binary performance as shown in Table 4.6. In the larger test of 1,000,000 the XML had significantly worse speed performance (just over twice as long as the binary). This speed difference for the higher record count is accounted for by the difference in the file sizes. In the larger test the XML file is 85MB larger than the binary file, which makes for a more significant difference than the 8.5MB difference in the smaller 100,000 record case.

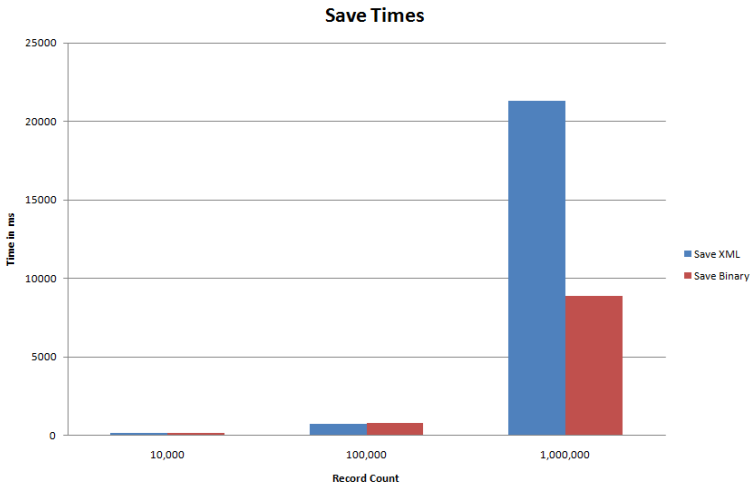

Figure 4.3: Save times for XML and binary output.

Record Count	Save Time XML	File Size XML	Save Time Binary	File Size Binary
10,000	141ms	2.17MB	106ms	1.8MB
100,000	691ms	21.9MB	750ms	13.6MB
1,000,000	21,288ms	221MB	8,858ms	136MB

Table 4.6: Record Save Times and Sizes as XML and Binary

The loading tests found the XML to be on average 8% faster, in all cases than the binary loading. While I do not have a definitive answer as to why this is, the XML is loaded

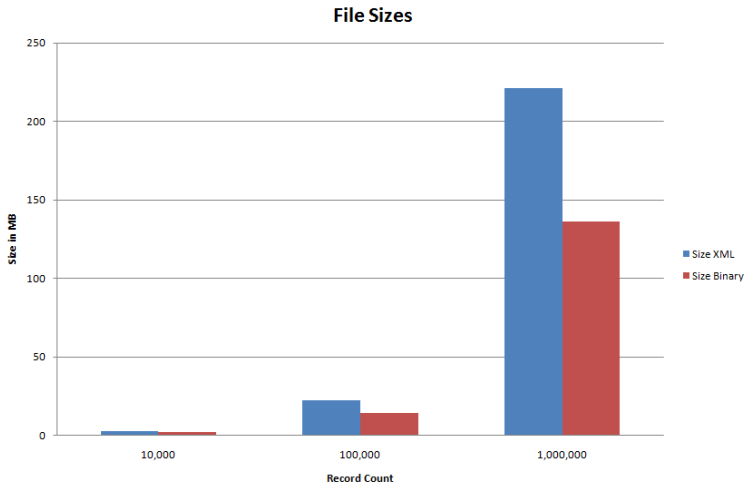

Figure 4.4: File sizes for XML and binary output.

through the .Net XML reader while the binary loading is read directly out of a C# MemoryStream object using a BinaryReader. I would theorize that there are some additional optimizations in the XML reader, or some unfortunately slow components in the BinaryReader that are adversely affecting performance as shown in Table 4.7.

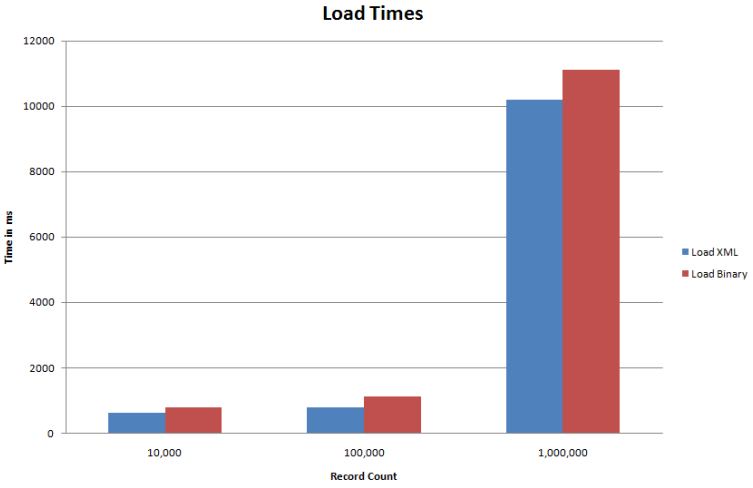

Figure 4.6: Load times for XML and binary output.

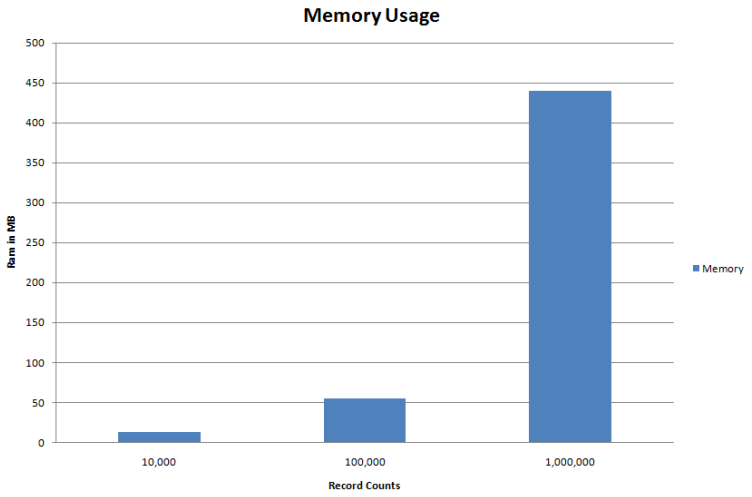

Figure 4.5: Memory usage for loaded database.

Record Count	Load Time XML	Load Time Binary	Memory Size
10,000	620ms	790ms	12MB
100,000	789ms	1,105ms	55MB
1,000,000	10,206ms	11,123ms	440MB

Table 4.7: Record Load Times XML and Binary, and System Memory Usage

4.2.5 Record Retrieval

The time to retrieve and display a set of 10,000 records from the database takes 270ms. The retrieve and display includes reading and parsing each field to the correct data type. In this case each record had 5 fields to parse. The system resides in memory, so record retrieval is very fast. The bulk of the time is spent parsing the data which is stored as strings into the correct data type for use.

4.3 Discussion

My system is aimed squarely at educator and student developers. The game development industry is very proficient at data management and very likely has developed better approaches than the one offered here. Unfortunately most of their systems are regarded as trade secrets and are information regarding these systems is not freely available.

One of the important elements considered was compatibility with XNA on both Windows and the Xbox 360. XNA does not allow the execution of native code on the Xbox and that limitation makes many if not all of the previously written systems for windows unusable with XNA. My system is written entirely in C# with the intent of being XNA Xbox 360 compatible.

The database currently supports five basic field types: int, float, boolean, DateTime, and string. Support for additional game related formats such as vectors, and doubles, is next in development. A generic serialized object field type would also add useful functionality.

Currently datasets are loaded by record ID in sections. If a specific set of records is required iteration through the returned set is required to find them. The addition of a query system would make this easier and is also under development.

Save and load operations on the database load and save entire databases, therefore the size of the database is limited by the amount of available system memory. A database that cannot be loaded entirely into ram is going to be a problem. On Windows this is not as big a problem as on the Xbox where resources are more limited. Additionally a segmenting system, that loads and stores segments of the database to disk instead of loading the entire database, would allow for a lower memory footprint in cases where resources were needed for other things.

Chapter 5

Case Studies and Usage

5.1 Case Studies

Figure 5.1: Left the platform memory game from Cognitive Carnival, right sustained attention task game from Cognitive Carnival.

5.1.1 Cognitive Carnival

Cognitive Carnival is series of games currently in development in our lab (UVic Computer Graphics Lab). The games are a joint project with the Psychology Department. The primary goal is to design games that exercise critical cognitive functions including working memory, attention, and executive function. The games store detailed play records that track a users progress through each of the cognitive tasks. The game required a database system that would work on both windows, and the Xbox 360, as well as be able to connect to a central database system for central access to the data. In the first iteration of the design, the data

was stored locally in XML files with custom save and load code for each of the tasks. It was the clumsiness of the designed XML approach that lead to the creation of my database system. The switch to the database system has consolidated all of the files, as well as greatly reduced the amount of save and load code required for each task. As many of the tasks are still in early development the data requirements are continually changing the database system has allowed easy changes, additions and removal of fields without loss of already recorded data.

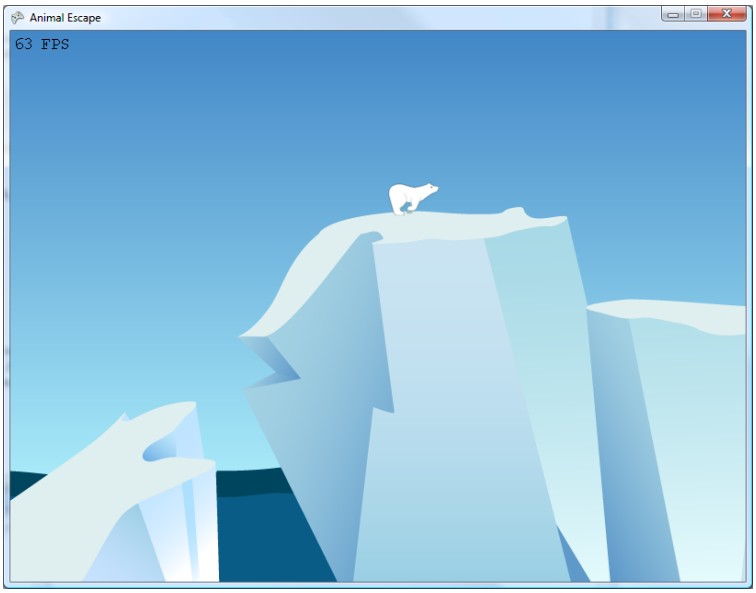

Figure 5.2: Global Warning Imagine Cup Entry.

A recent addition to the game set is the cognitive platform game as shown in Figure 5.1. The gameplay consists of collecting a set of targets in a given random order while avoiding obstacles and remembering the order to collect the targets. This game required a level editor as part of its design. Using the database system all of the level information is stored in a single database. The game is tile based, so along with the level information, descriptors for each tile are also stored in the same database.

5.1.2 Undergraduate Imagine Cup Class

We ran a class this spring for undergraduate students interested in participating in the Microsoft Imagine Cup game development competition. As part of this class, teams of four to five students worked on designing and building games in XNA Game Studio that addressed world issues. One of the teams used my database system in their game "Global Warning" a platformer staring a polar bear as shown in Figure 5.2, to store all of their level data. Their level data included tile sets with polygon based collision boundaries, and level layout. As part of the development a tileset editor and a level editor were built. The database system provided an easy to use interface allowing loading and storing of the same data in the editors and the game without specific save and load code being duplicated in all 3 programs.

5.1.3 Database Manager

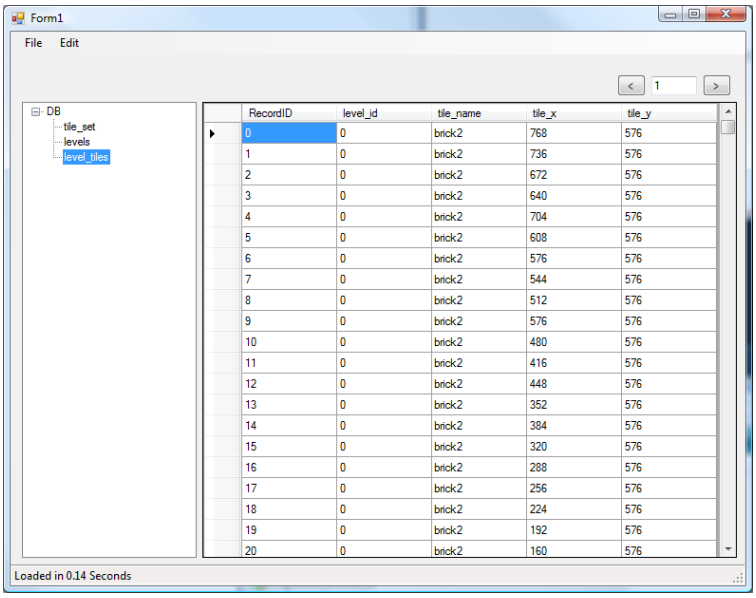

Figure 5.3: Database Management Tool.

In addition to the database system, a database management application was written. The database manager, as shown in Figure 5.3, allows users to create new databases instead

of directly defining them in code. It also has a table listing and a fully editable paginated view of the records in the currently selected table. The database manager is highly useful in debugging data output because it allows you to look directly at records stored in your database file.

Chapter 6

Conclusions

My database system provides an easy to use interface that is at the level of students and educators. The pilot evaluation and case studies indicate that there is little to no difficulty in setting up and working with the database. Users generally indicated the database system to be not only the easiest to work with when compared with the XML, or Binary alternatives, but also the easiest to change. In addition the database system has already been used with great success in several game projects including the Imagine Cup class and our in lab cognitive games collection.

6.1 Contribution

Data persistence is a topic almost completely passed over in the educational games course research. This thesis supports that when compared to standard methods for data persistence a database system provides a cleaner, easier to understand and use system for students. This thesis also suggests there is a benefit to database systems being provided to students working on game projects. There is further research needed to be done in what system provides the most benefit to students, as well as research into why data persistence is overlooked in the majority of the courses.

Bibliography

[Adams 1998]ADAMS, J. C. 1998. Chance-it: an object-oriented capstone project for cs-1. *SIGCSE Bull. 30*, 1, 10–14.

[Burns 2008]BURNS, B. 2008. Teaching the computer science of computer games. *J. Comput. Small Coll. 23*, 3, 154–161.

[Coleman et al. 2005]COLEMAN, R., KREMBS, M., LABOUSEUR, A., AND WEIR, J. 2005. Game design & programming concentration within the computer science curriculum. *SIGCSE Bull. 37*, 1, 545–550.

[Jones 2000]JONES, R. M. 2000. Design and implementation of computer games: a capstone course for undergraduate computer science education. *SIGCSE Bull. 32*, 1, 260–264.

[Leutenegger 2006]LEUTENEGGER, S. T. 2006. A cs1 to cs2 bridge class using 2d game programming. *J. Comput. Small Coll. 21*, 5, 76–83.

[Linhoff and Settle 2008]LINHOFF, J., AND SETTLE, A. 2008. Teaching game programming using xna. In *ITiCSE '08: Proceedings of the 13th annual conference on Innovation and technology in computer science education*, ACM, New York, NY, USA, 250–254.

[Parberry et al. 2006]PARBERRY, I., KAZEMZADEH, M. B., AND RODEN, T. 2006. The art and science of game programming. *SIGCSE Bull. 38*, 1, 510–514.

[Wolz et al. 2006]WOLZ, U., BARNES, T., PARBERRY, I., AND WICK, M. 2006. Digital gaming as a vehicle for learning. *SIGCSE Bull. 38*, 1, 394–395.

Appendix A

Evaluation Instructions Questions

XML, Binary, and Database Evaluation

Purpose:

The purpose of this evaluation is to compare the process of writing game related data out with XML, Binary, and a Dynamic Database System.

Instructions: 1. Download and unzip the breakout project. Run the project to confirm that it works. There are three different color bricks on the left hand side; clicking one selects it as your editing tool. You can now click to place bricks in the gray window. Left clicking on a brick removes it. Clicking the Game/Play menu will let you play your level (not really part of this evaluation). Familiarize yourself with the interface.

2. Open the code for form1.cs. In the region labelled "‘Save and Load Code"’ there are 6 function stubs. These stubs are saving and loading functions for XML, Binary, and the DB System. Familiarize yourself with these saving and loading approaches by viewing the code samples at the end of this document.

3. Complete each stub by coding the correct save and load function. The data you need to save is the member variable currentBricks which is a list of Brick objects. Each Brick object in the list has 3 parameters you need to save, .X, .Y, and .BrickHits. Those are the 3 parameters you need to pass to the constructor to recreate the object when you load it.

4. Test each function by using file load and file save. The 3 file types are selectable in the Save and Load dialog boxes.

5. Fill in the Evaluation Survey and email it to greytone@gmail.com Or print the survey and return it to me directly.

Evaluation Survey

Experience Questions:

How experienced are you with SQL Databases: A) No Experience B) Minimal - I have written at least 1 query C) Novice - I took the database course, or I have used an SQL database for a small website etc. D) Experienced - I have used SQL Databases for larger work/personal projects. E) Expert - I have used SQL Databases for several major projects where I was responsible for database design, performance, or maintenance.

I have written code to read/write data as plain text files: A) Never B) Once C) More than Once, but less than 10 times. D) More than 10 times.

I have written code to read/write data as XML files: A) Never B) Once C) More than Once, but less than 10 times. D) More than 10 times.

I have written code to read/write data as Binary files: A) Never B) Once C) More than Once, but less than 10 times. D) More than 10 times.

Setup:

Rate on a scale of 1 to 10 with 10 being difficult and 1 being easy how difficult it was to come up with your data structure/table definition for saving data as:

XML /10 Binary /10 Database /10

Rank in order of time, how long it took you to come up with your data structure/table definition. 1 for shortest, 3 for longest. You may use the same number more than once if you took the same amount of time for 2 or more of the tasks.

XML Binary Database

Comments (Optional):

Data Entry:

Rate on a scale of 1 to 10 with 10 being difficult and 1 being easy how difficult it was to save data using: XML /10 Binary /10 Database /10

Rank in order of time, how long it took you to write your save data function. 1 for shortest, 3 for longest. You may use the same number more than once if you took the same amount of time for 2 or more of the tasks. XML Binary Database

Comments (Optional):

Data Retrieval:

Rate on a scale of 1 to 10 with 10 being difficult and 1 being easy how difficult it was to load data using: XML /10 Binary /10 Database /10

Rank in order of time, how long it took you to write your load data function. 1 for shortest, 3 for longest. You may use the same number more than once if you took the same amount of time for 2 or more of the tasks. XML Binary Database

Comments (Optional):

Making Changes:

Rate on a scale of 1 to 10 with 10 being difficult and 1 being easy how difficult you think it would be to change your structure to store multiple levels in one file: XML /10 Binary /10 Database /10

Rank in order of time, how long you predict it would take to change each save and load to store multiple levels in one file. 1 for shortest, 3 for longest. You may use the same number more than once if you took the same amount of time for 2 or more of the tasks. XML Binary Database

Comments (Optional):

Relevancy:

Rank in order of preference, if you were developing a game and each system was available to you - 1 for first choice, 2 for second choice, 3 for last choice. XML Binary Database

Comment - Please give a reason for your choice: