Using a simple MMORPG to teach multi-user, client-server database development

Greg Wadley
Department of Information Systems
The University of Melbourne
Australia 3010
+613 8344 1586
greg.wadley@unimelb.edu.au

Jason Sobell
Philology Pty Ltd
111 Barry Street
Carlton, Australia 3053
+613 9349 4735
jason@philology.com.au

ABSTRACT

Applications built for undergraduate programming assignments are typically single-user systems, of which the programmer is also the sole user. Real-world information systems differ from this scenario in a number of ways. In particular, they are usually client-server systems within which many users concurrently access the same data. In order to illustrate for our students the benefits and pitfalls of multi-user systems based around a shared database, we asked them to build a simple massively-multiplayer online role-playing game (MMORPG) which stored game-world and player state in a relational database. We provided students with a graphical client written in Visual Basic. As players moved about the game world, interacting with objects and other players, their client programs called procedures in the central database to update game state accordingly. The students' task was to implement database tables and procedures that allowed the clients to work. The system's client-server architecture resembled that of commercial information systems and often occasioned concurrent access to data. In this paper we describe the system, the students' experience of building it, and our perception of its pedagogical pros and cons.

Categories and Subject Descriptors

K.3.2 [Computing Milieu] Computer and Information Science Education – *Information systems education.*

General Terms

Design, Human Factors, Languages, Theory

Keywords

database, multi-user, education, project, MMORPG, RDBMS

1. INTRODUCTION

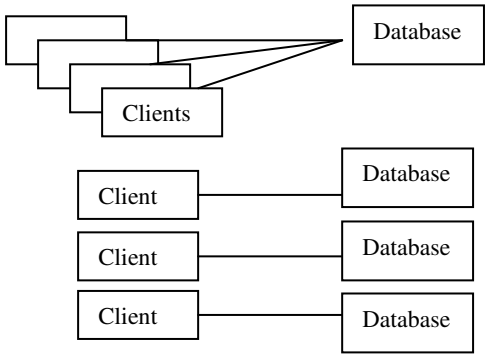
Computer games are used as undergraduate programming assignments for a number of reasons. Many students play games, understand this class of application well, consider games fun to develop and use, and are interested to find out how they work. Games emphasize the user interface of an application, and are well suited to real-time visualization of system state. They can readily illustrate a number of problems from simple program logic to graphics, usability and artificial intelligence. Game development expertise can lead to a career that many students regard highly.

Different genres of games are suited to different courses and competencies. For example, students in introductory programming courses are often asked to build simple puzzle games, while advanced students in game development courses might build 3d games or graphics engines.

The authors teach an advanced database course in a Bachelor of Information Systems degree. Our aim is to expose final-year students, who have completed an introductory course in SQL and E-R modeling, to issues concerning the development, physical implementation and administration of database systems. Lab exercises in our course involve hands-on use of both Oracle and Sql Server database management systems and related tools.

Information systems in industry typically use a client-server architecture, by which many users connect to a central database. Yet projects in database courses are typically single-user systems, of which the student is both the programmer and the only user. While this architecture is easy to program, teach and administer, it obscures the main purpose to which most relational databases are put, which is to store a representation of entities and events that are significant to a group of people, who read and write the data in an ad-hoc way and effectively communicate through it. An understanding of this function of multi-user databases cannot easily be gained by building single-user applications in which the database is simply a convenient disk-based data store.

Figure 1. Typical I.S. architectures used in Industry (above) and Education (below)



2nd Annual Microsoft Academic Days Conference on Game Development, February 22–25, 2007, Florida, USA.

Nor do single-user systems readily illustrate the problems, such as lost updates, uncommitted dependencies, and inconsistent reads, that can arise when several users concurrently access the same data. Without experience of these problems it is difficult for students to understand the techniques that database and DBMS designers use to solve them, such as transactions and locks, or the problems that these techniques in turn can cause, such as deadlock.

We reasoned that in order to better understand client-server database programming, our students should build a multi-user system in which an update to the database by any user affected everyone else's view of the data in an obvious way. Taking into consideration also the benefits of using games in programming assignments, we concluded that building a multi-user game which stored the state of the game-world and players in a relational database would be an informative and motivating project.

Our students were not expert GUI programmers and our course does not focus on writing client programs, especially the kind of graphics-intensive clients used in commercial multi-player computer games. Also, because of the limited power of our server, lab computers and network, a fast-paced game was inappropriate. These restrictions excluded some genres of games from consideration, such as team-based 'shoot-em-ups'.

Inspired by the popularity of massively multiplayer role playing games such as World of Warcraft, we decided to use a simple MMORPG as the project theme. To preserve the course's focus on database rather than GUI programming, we wrote a client program using Visual Basic and gave it to students. We provided also a list of the tables and stored procedures that the client would look for in the supporting database. The students' task was to implement these tables and procedures.

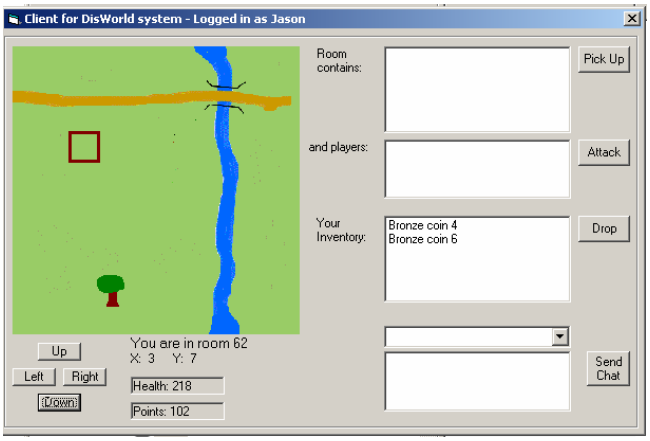
The essence of an MMORPG is that all players act within the same, single virtual world. Each player's actions affect, and are visible to, the other players. Furthermore, the game-world is persistent: world and player state are maintained independently of any particular individual's login sessions. These criteria require properly managed, central, disk-based storage.

We hypothesized that this function of an MMORPG – to maintain a single, persistent representation of a (in this case, virtual) world, which is read and written to by many users – is essentially the same function we wanted to illustrate for RDBMS-based information systems generally.

2. HOW THE GAME WORKED

To simplify the task of programming the client, and to make the game rules and mechanics clear, we designed a simple game that took place in a two-dimensional 10 x 10 'grid world' shaped like a chess board. Players navigated around this grid by moving up, down, left or right, one square ('room') at a time, using buttons on the game client. To illustrate an avatar's location, the client simply highlighted the appropriate square. As well as the players' avatars there were items scattered about the world which players could pick up, hold in their inventory as they moved about, and drop again in a different location. These were displayed to the player in listboxes.

Figure 2. the Game Client (GUI)



Each individual player and item was represented by a row in a table in a central relational database. As the game progressed, the database kept track of players' locations, health and scores, and the locations of items. Location had low resolution, and could be stored as an integer. Items had to be located either in one of the rooms or in a player's inventory. To pick up an item, a player's avatar needed to be located in the same room as the item. The player then pressed a 'pick up' button on his/her game client.

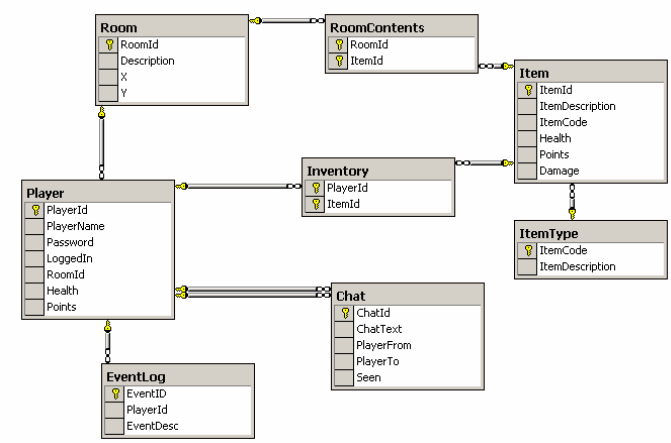
Some items ('weapons') could be used to attack other players. Attacks could only occur when two players occupied the same room. Players accumulated points by picking up items, and lost health when attacked. Health could be restored by finding a health pack. These game actions were effected by the player by pressing a button in the client, triggering execution of a stored procedure which read and wrote data in the database.

Each room in the game-world was represented by a row in a database table. A room was marked 'out of bounds' by omitting that row from the table (we represented non-traversable rooms graphically in the client as a river and a tree). Movement by a player required that the client perform a Select against the database to check whether the desired destination room was traversable, and if it was, to read which items and players were in the room: these were then listed in the client. Finally movement required an Update to change the player's recorded position.

Players could send text messages to each other. The messages were displayed in a list box in the game client. The sending player's client inserted a row in the Chat table, and the receiving player's client eventually selected it for display.

Players had to identify themselves to the game (ie log in) by supplying a name and password already recorded in the Player table. Thereafter a player's game client was able to supply the player's identifier to the database when sending or retrieving data. Following the 'persistent world' approach used in MMORPGs, a player's location and state was preserved between any logout and the next login. Logged-out players did not appear in game clients and could not be attacked. Logins, logouts and other game events were recorded by inserting rows into an event-logging table.

Figure 3. the Database Schema



The database schema is illustrated in figure 3. Several of the kinds of tables typically implemented in relational databases are present in this schema, including:

- *entities* (people, objects, places) - the Player, Item, and Room tables
- *associative entities* (relationships between other entities) – the RoomContents and Inventory tables
- *logged events* - the EventLog and Chat tables
- *lookup tables* – the ItemType table

The individual tables were:

Player: contained one row for each individual person who was registered to play the game (ie students and teachers taking the course). Like a typical 'person' table, it recorded identifying information (a player's name and password), some dynamic properties (location, health, points), and whether the player was currently logged in.

Room: contained one row for each of the game world's rooms (except for non-traversable ones), identified by the integers 1 to 100 and by x-y coordinates. Players and items in the game world were situated in exactly one room at any given time. We programmed the client in such a way that it would be relatively straightforward to re-implement the game with more or fewer rooms.

Item: contained one row for each individual object in the game world. Items were classified into four categories - Points, Health-packs, Weapons and Fixed - using the lookup table *ItemType*. While health items were consumed when picked up, players could carry weapons and points items in an Inventory, implemented as a table pairing items with players. We arbitrarily restricted the size of an inventory to three items, to reduce hoarding by players. Players could not interact with Fixed items. At any given time an Item had to belong either to a player's inventory or to a room. This method of representing items made it relatively straightforward for game administrators to insert new items into the game.

The *Inventory* and *RoomContents* tables associated Items with Players and Rooms respectively.

EventLog: recorded actions by players for (fictitious) auditing purposes. For simplicity we described events with a string of text rather than categorizing them with a lookup table.

Chat: contained one row for each text message sent by a player to another player. When the receiving player's game client fetched a message for display, it marked the message as 'seen' rather than deleting it from the table.

When a player carried out a game action, their game client made one or more calls to the database. The students' task was to enable these calls using stored procedures, according to the specifications listed in table 1. For each procedure we gave students the procedure name, a description of its behaviour, the inputs that a client would provide, and the outputs a client would expect. The client was programmed to call the appropriate procedure in the database when the user pressed a button. The 'getter' procedures were also called every few seconds by a timer in the client, to check whether the room's contents had changed, check incoming messages, and allow the screen to be refreshed.

Table 1. Stored procedures that supported game actions

Procedure	**Inputs**	**Outputs**
spLogin: See if there is a row in Player that matches this name and password. If there is, change that row's LoggedIn field to 1, write a row to the Event table, and return the row from Player. Otherwise, return nothing.	PlayerName, Password	1 row from Player table
spMovePlayerTo: Select from Room to check that desired room exists (is traversable). If it is, update player's location.	PlayerId, X, Y	'Success' = 1 or 0
spGetPlayer: Join Player and Room tables to select all information about this player and current room	PlayerId	Player and room details
spGetItems: Join Item and RoomContents tables to get information about each item in this room.	RoomId	list from Item table
spGetPlayers: Return info about all players in this room, except this player	PlayerId, RoomId	list from Player table
spGetInventory: Join Item and Inventory tables to get information about each item in this player's inventory	PlayerId	list from Item table
spPickUpItem: Delete item from RoomContents and add it to Inventory. Update player's point score.	PlayerId, ItemId	(none)
spDropItem: Delete one row from Inventory, add one row to RoomContents.	PlayerId, ItemId	(none)

spAttack: Check that selected item is a weapon and how much damage it inflicts. Reduce victim's health accordingly.	PlayerId, VictimId, ItemId	Message about how much damage was inflicted
spGetAllPlayers: Return a list of all players in the Player table, to populate chat dropdown.	(none)	list of PlayerId, PlayerName
spAddChat: Write one row to Chat table	Text, PlayerId1, PlayerId2	(none)
spGetChat: Select all chat messages addressed to this player and not yet seen. Mark them as seen.	PlayerId	list of PlayerName, ChatText
spAddEvent: Write one row to Event table	PlayerId, EventDetail	(none)
spLogout: Change this player's LoggedIn field to False.	PlayerId	(none)

3. THE STUDENT EXPERIENCE

We provided each student with a Sql Server database in which to implement his/her tables and procedures. We encouraged students to check the behaviour of their databases by using both the game client and Sql Server *Query Analyzer* to execute procedures. The latter made it easy to display several different SQL statements and their outputs on the same screen, such as 'select *' before and after a procedure execution, which was useful when debugging procedure code.

While each student implemented their own project database, they could, when desired, allow another student's client to log into their database, in order to play their game with the other student.

We also provided a working 'black box' solution to the assignment. This was a database with tables and procedures already implemented by us. It allowed students to play and observe a working version of the game, to better understand their project requirements. We used server permissions to ensure that students' game clients could execute the procedures in this database without being able to read the procedure code or table structures. During a number of lab classes we asked all students to log into the provided database and play a game together. During these sessions we displayed the contents of some tables on a screen in the lab, using the *Enterprise Manager* and *Query Analyzer* tools. Students were able to simultaneously observe their own screen, the screens of other students nearby, and the changing table data on the projection screen, in order to understand how game clients were interacting with and within the shared database.

To submit their work, each student placed their 'create table' scripts and stored procedure code into a text file, and emailed it to the teachers. Code was assessed according to correctness of client behaviour, with bonus marks for elegance and efficiency.

4. WAS THE PROJECT SUCCESSFUL?

In using a multiplayer videogame as a database project our aim was not to teach game design, but to utilize a type of application which we believed undergraduate students would find interesting and intuitive, and which would successfully illustrate the benefits and pitfalls of developing systems based on multi-user access to shared data. Therefore in analyzing whether our project was successful, we need to ask whether students acquired a better understanding of how systems are built around relational databases. That is our 'general' question. We can also analyze our detailed decisions. For example, was the project too easy or hard? Was the database schema too simple or too complex? Was it reasonable to give students a pre-programmed client and ask them to develop the server?

We did not conduct a formal experiment to measure the educational impact of the project. However we can give brief answers based on informal feedback received from students during the project, the University's 'Quality of Teaching' feedback received after the project was over, the work submitted by students, and our subsequent reflections on the course. Overall, we felt that the project was successful in engaging and educating students. However there were some problems, and these are listed below.

4.1 Pros

Our game emphasized user interaction via shared data. The game was multi-user in a way that was easily understandable by students. It emphasized 'computation-as-interaction' over 'computation-as-calculation'; the client-server, networked, interaction-based view that Stein [1] suggests is the best metaphor for understanding modern information technology use.

The project utilized an architecture which is common in business contexts: a client written in Visual Basic, running on a Windows PC, accessing a Sql Server database.

The game client helped students to monitor the changing values stored in the underlying database, realizing some of the pedagogical advantages of visualization [2]. The visual client encouraged students to frequently compare their client screen with the underlying database tables and see more clearly the effect of their procedures on the database.

The system afforded communication among users. Other than tools such as email and instant messaging, it is difficult to think of an application class that lends itself as readily as multiplayer games to having several users communicate through a shared database. The collaborative and visual nature of the system helped to motivate students.

The project emphasized that the core function of an information system is to represent some interesting subset of the world; in particular some entities, their properties and relationships, categorized into classes, which are relevant to a business problem [3]. Although a game-world is fictitious, using a relational database to represent the properties and behaviour of a game world emphasized the representational function of databases.

Some common database project applications (such as order entry) tend not to excite students. The MMORPG was an unusual project for a database course and was more interesting for many

students. They were happy to explore the system in class and even after hours. While the system was presumably not as engaging as a commercial MMORPG, its relative simplicity made the underlying mechanisms clear to students curious about how MMORPGs might work.

Although this project did not use a business-oriented theme, many of the design issues exposed here are relevant also to business systems. The most important tables in a business database typically represent classes of entities and events. Event tables tend to become large over time. Miscellaneous tables are needed to implement sub-classes and many-to-many relationships. These phenomena were present in our game database.

In common with many real-world applications, our game had to adequately handle user identity (established through a login) to allow the system to work in a meaningful way, and to allow communication between users.

A number of game actions lent themselves to demonstrating problems of concurrent access to data. For example, to pick up an item required a Delete from the RoomContents table, an Insert to the Inventory table, and an Update of points in the Player table. Getting this code to work correctly when two players simultaneously tried to pick up the same item required careful ordering of these statements, and the use of transactions. Game scenarios such as these may demonstrate problems of concurrency in a more visual and dramatic way than do commonly-used teaching examples such as "transfer funds".

4.2 Cons

While many students are interested in games, some are not. Our assignment was not a typical business application. While this was a plus for some students, many of them were business-focused, and some felt a game to be relevant only to recreation.

There were some problems fine-tuning our approach of giving students a finished client and asking them to implement a database to support it. In typical system development the server would be developed before, or at least in conjunction with, the client. It took some redrafting before we specified the inputs and outputs of the client clearly enough for students to write their procedure code. Without access to client code, it was more difficult for students to debug database code (they could not, for example, display client variables), and students were not exposed to methods of data validation in the client. It is probably better for students to program both client and database in an application: however this was not feasible in our project.

4.3 Relevance to game development courses

The aim of our project was to illustrate the design and development of client-server databases rather than games. Our game-world and game-play were not to commercial standards. We did not intend that the game closely resemble a commercially viable MMORPG. However some comments can be made on the relevance of this project to game development.

It is possible that for performance reasons some commercial MMORPGs do not use a relational database to store game-world and player state. However our research indicated that several do, and at least one open-source MMORPG does so [4]. Our two-tiered architecture was much simpler than the complex multi-tier, multi-server, distributed architectures needed to operate a large-scale MMORPG. However we feel that we captured the essence of multiplayer game design in this project. A course oriented to game design or development might make use of a system like this as a starting point to discuss multiplayer game design.

5. CONCLUSION

We found that development of a relational database to support a simple MMORPG worked well as a project in an advanced database course. Multiplayer games are well-suited to illustrating issues of identity, interaction, and concurrent access to data in multi-user information systems. In this paper we have described the game mechanics, the database tables and stored procedures that our students produced, and our experience as teachers using this project in an undergraduate course.

The game is easy to simplify or expand, allowing it to be easily tailored to different pedagogical goals, course levels and class sizes. For example, students could be assigned a simple version of the game in an introductory database course, and a more complex version in an advanced course. Students in a game design course could be given this game and asked to improve its game play, game world, or interaction design. A web interface for game administration could be added. The authors would be pleased to discuss the application with educators interested in using it in their courses.

6. ACKNOWLEDGMENTS

The authors would like to thank the students who undertook the project, and to Graeme Simsion and Darren Skidmore at the University of Melbourne for reviewing a draft of this paper.

7. REFERENCES

[1] Stein, L. *Challenging the computational metaphor: implications for how we think,* in Cybernetics and Systems 30:473-507, 1999

[2] Hundhausen, C. *Integrating algorithm visualization technology into an undergraduate algorithms course: ethnographic studies of a social constructivist approach,* in Computers and Education, 39:237-260, 2002

[3] Weber, R. *Ontological Foundations of Information Systems,* Coopers and Lybrand, 1997

[4] Riddoch, A. and J. Turner. *Technologies For Building Open-Source Massively Multiplayer Games*, Worldforge.org, 2005