Database Research Opportunities in Computer Games

Walker White, Christoph Koch, Nitin Gupta, Johannes Gehrke, and Alan Demers
Cornell University
Ithaca, NY 14853, USA
{wmwhite,koch,niting,johannes,ademers}@cs.cornell.edu

ABSTRACT

In this paper, we outline several ways in which the database community can contribute to the development of technology for computer games. We outline the architecture of different types of computer games, and show how database technology plays a role in their design. From this, we identify several new research directions to improve the utilization of this technology in computer games.

1. INTRODUCTION

Games touch the lives of many people. They are a big portion of the entertainment of the average person; the typical American teenager spends at least an hour of every day playing a computer game [26]. In addition to leisure, games can be used in such areas as training and education [28] or modeling and simulation [21]. Furthermore, computer games are big business, rivaling the movie industry in revenues and profits. The Entertainment Software Association estimates that computer and video game software sales in 2006 were $7.4 billion dollars [1]. The game *World of Warcraft* alone generated revenues of $471 million dollars [13]. Clearly, people spend much of both their time and money on games.

Unfortunately, outside of computer graphics, there has been little academic impact on the development of computer games. Even in the area of machine learning and artificial intelligence, much of the technology used in games was developed in the 1980s, and the newer research does not adequately address current needs [32].

We believe that the database community has unique capabilities that it can bring to this area. However, the ways in which it can contribute are not immediately clear. In particular, obvious directions such as spatial indexing are already being used by the industry, and so work in these areas is likely to yield only incremental benefit. Instead, we as a community need to understand both the state of the art and future needs of game developers. At the recent Austin Game Developers Conference, which is dedicated to the design of massively multiplayer online (MMO) games, several game studios highlighted the difficulties that they encounter with commercial database software [7, 20, 27]. In addition, the authors of the present article have presented a unique way in which database technology can help even non-networked games [33] at this year's SIGMOD conference.

In this paper, we outline some of the directions where the database community could contribute. The directions are of course high-level, but they are motivated by real needs in the game industry. Furthermore they result in fascinating research problems for the database community. In particular, the confluence of expertise in large data, systems, and languages from the database community is crucial to advances in the research directions that we outline.

We do not believe that these are all the problems. In particular, some of the problems that the database community has been notoriously bad at, such as high-level interfaces to the data, are central to the success of the endeavors that we outline. However, we believe that by undertaking this research, we have the means to make a significant impact on the game community, and thus indirectly, society at large.

We wrote this article in order to familiarize the database community with computer games and their unique challenges. In the most general sense, a computer game is a virtual environment in which players interact with digital objects or each other for entertainment. This includes everything from casual single-player games such as *Solitaire* or *Minesweeper*, to immersive massively multiplayer experiences like *World of Warcraft* or *Second Life*. While in the future, data management techniques will certainly reach out to other computer game genres, the present article focuses on games that have an important simulation component. In particular this includes (first-person) shooter games, real-time strategy games, massively multiplayer online games, and various types of games for training and education. These groups form a major share of the computer games market.

The rest of the paper is organized as follows. In Section 2 we outline, at a very high level, the system architecture of various computer games. Within this architecture we pay special attention to the ways in which databases and database technology play a role. In Section 3 we identify several broad research directions in which the database community can improve these architectures. We conclude in Section 4.

2. GAME ARCHITECTURE

To understand how database research can help to improve game technology, we must first understand modern game ar-

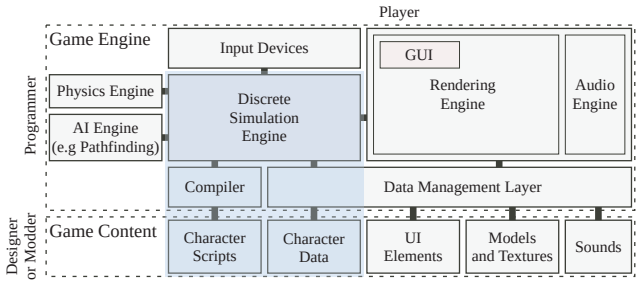

Figure 1: Non-Networked Game Architecture

chitecture. In some cases, such as MMO games, databases are already an integral part of the design, but may not be leveraged in the most effective way. There are other cases in which database technology may not currently be used but its introduction could be greatly beneficial. Scripting character behavior in single player games, as demonstrated by the scripting language SGL, is one such example [33].

Describing modern game architecture is difficult because there is no one way to design a particular game. While a large part of game design lies in creating the game content, most game studios do quite a bit of custom architecture development. Even in cases where a game studio licenses its software technology from others, such as Epic Games' *Unreal Engine 3* [16], the studio is likely to modify or extend the system in order to adequately tune performance for their game. Furthermore, architecture often varies widely across game genres. Therefore, our discussion of game architecture is at best an idealization that attempts to highlight the most important aspects of game design.

In this discussion, we classify games according to how they are connected to each other over the network. In our categorization, there are three types of games. *Non-networked games* are those that can be played locally and do not interact with other instances over the network. In a non-networked game, if the player wants to play with another human, then that person must be local, acting through a second input device. *Non-persistent games* are networked games in which the game state exists for only a single session. In such games, when players start a new session they return to their initial state and so all of their previous actions are lost. Finally, *persistent games* are those that provide an environment that preserves the actions of its players. Persistent games often provide living worlds that exist and evolve beyond the game session of any single player. Each of these types of games can exploit database technology in different ways.

2.1 Non-Networked Games

Modern commercial games are rarely designed without network capability, as this is an important part of game value and longevity. However, non-networked functionality is an important part of any game, and its architecture is typically a subset of any of the others.

One of the most important aspects of modern game design is that it be *data-driven*. Loosely defined, a data-driven design is one in which the game content is separate from the game code [10]. This design style has several advantages. It allows the game studio to separate development between programmers and game designers, two groups with essential but not necessarily overlapping skills. It also allows the studio to reuse the code, typically referred to as the *game engine*, for other games, or even license the engine to other studios. Finally, it allows the game content to be modified by users. "Modders" are after-market designers who replace old game content with new content in order to keep a game fresh and interesting. Allowing users to create their own content can significantly increase the appeal and life-span of a game.

With that said, "game content" is a fairly vague term. Obviously, it includes media such as character models, textures, or sounds. It also includes the data used to define the story-line or initial starting state of the game objects [31]. It even includes scripting languages that define character behavior [12, 8]. As character designers often have a different set of skills from programmers, these scripts are usually developed using high-level tool-sets like Simbionic [15].

Figure 1 represents the architecture of an idealized non-networked game. It illustrates the separation of the game content from the game engine. The former is created by game designers or by modders, while the latter is the purview of programmers. Given the volume and diversity of game content, games obviously need a sophisticated data management layer in order to handle this data. However, it is not immediately clear that this is a problem of interest to database researchers. Objects are simply loaded in memory as needed; games do not need particularly complex query processing to handle things like sound or artwork.

One easily identifiable area in which databases can help non-networked games is the *discrete simulation engine* [33]. To understand this part of a game engine, we must first understand the event-loop architecture of games. Every game has a main loop that animates the game. Each pass through the loop corresponds to a frame of animation on screen. During a single pass, the game engine does the following:

- computes the behavior of all the game objects by *querying* the current state of the world,
- *updates* the state of all the game objects according to this computed behavior, and
- draws the new state of the world to the screen.

The query-update part of this animation loop is what we term the discrete simulation engine. In basic game frameworks like XNA [22], the simulation-engine is processed lock-step with the graphics engine, so nothing can change on screen without an explicit update from the simulation engine. More sophisticated engines run the simulation and graphics engines as separate threads, with the simulation engine running at a slower rate [17]. In between updates from the simulation engine, the graphics engine thread interpolates the world state in order to provide smoother animated behavior.

The query-update model of the simulation engine makes it an obvious candidate for application of database technology.

However, to do this, we need to understand how the simulation engine fits with the other components of the game engine. For example, the graphics engine needs to know the state of the world in order to render it. If the world state is represented in the database, should this engine access it using a database API? Or, since graphics programmers rarely know SQL or other database languages, should they access it through an object-oriented API, with the game engine automatically handling the conversion? Similarly, the AI engine needs to access the world state so that it can perform long-term planning (often asynchronously from the animation loop). As AI queries are often much more complex than those of the graphics engine, it is not clear that the same API is appropriate for both.

Conversely, there is the issue of how the simulation thread itself receives data from other parts of the system. For example, the physics engine often handles such issues as collision detection, which are a part of the query-phase of the simulation engine. Should collision detection be treated as a black-box operator, or is there a way to integrate it into the query plan? As another example, the simulation thread needs to react to commands sent to it by the player through the input devices. It is possible for multiple commands to queue up during a single pass of the loop, and the query plan must adjust itself accordingly. All of these are important software engineering questions, and there is not one simple answer.

2.2 Non-persistent Games

Most games played over a network are not persistent. In games like *Half-Life 2* or *Halo 3*, a player cannot save a game during network play. If the player leaves the game during the session, then all of her state is lost and she must be initialized again when she rejoins. This feature is acceptable because the games are designed with short term goals that can be completed in a single session. There are some non-persistent games, like *Diablo 2*, that allow players to keep some very limited local state between sessions, such as their abilities or their equipment. However, the state of the complete game environment is never saved.

The defining characteristic of a non-persistent game is that there is no single authority for the game state. As such, it is common for these games to be designed peer-to-peer, especially on a LAN where latencies are low. Sometimes these games will connect to a initial broker server which can match up games looking to network with one another. But once the game instances are connected, no central server is involved.

Architecturally, non-persistent games are similar to non-networked games except that they have an additional network layer. The additional challenge with these types of games is concurrency control. As there is no one authoritative repository for the game state, it is a challenge to keep the states consistent in real time. There are many different solutions to this problem. Older games use lock-step or pessimistic synchronization protocols [2]. More modern games use optimistic synchronization protocols. In these designs,

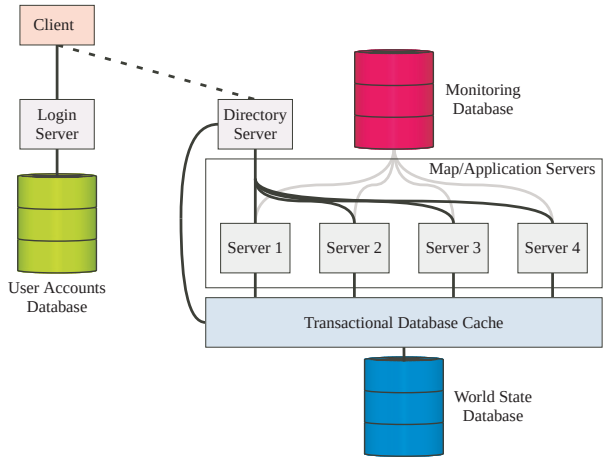

Figure 2: Persistent Game Architecture

the games partition their data so that each game instance is "authoritative" on a particular set. On data on which a game instance is not authorative, it simulates the other instances between network messages, and rolls back any simulations that are not correct [24].

In addition to concurrency control issues, another way in which database technology can fit into non-persistent games is game monitoring. Game companies are increasingly interested in gathering information about demographics and play-behavior from their users. This allows them to evaluate what features of the game are successful and should be incorporated into the design of future games. It also helps them to understand their market for purposes of advertising or localizing game content. Monitoring is used extensively in MMOs [18], but it is possible with any game connected to the Internet. For example, the gathering of this data may be part of the terms of service of the broker used to match up game instances.

Monitoring game data is a challenging problem because there is so much of it. In its most primitive form, monitoring consists of a log of the history of the game's world state over time. However, this level of detail is unnecessary and very difficult to analyze. Furthermore, the designers may want to break up the data for use by different groups in the company. The data needed by the marketing department is not the same data that the designers need to analyze gameplay.

2.3 Persistent Games

Persistent games always have an authoritative store of the current game state. Therefore, they almost always have a client-server architecture. As such, the game is sold as a service, because the game studio must manage a data center in order to keep the game running. Therefore, persistent games have much more in common with traditional database applications than other games. Indeed, databases already play a central role in all existing MMO games, which are the most common example of a persistent game.

There are a variety of persistent architectures in use, but we can summarize them at a high level with the illustration

in Figure 2. The primary difference between this architecture and that of the previous two categories is that no one machine is responsible for all of the components of the game engine. For example, no server runs a copy of the graphics engine, as this is clearly unnecessary. On the other hand, the client almost never does any arbitration or make any authoritative decision other than computing local game state and rendering it. Industry aversion to client-side computation is very strong, as clients are subject to being reverse-engineered and hacked. Hacked clients could be modified to "cheat" at the game, ruining the experience for other players, and thus devaluing the service provided by the game studio. Therefore, in a modern MMO game design, no client computation is ever allowed to determine the outcome of an interaction of a player with a game object.

Instead of client-side computation, the simulation engine is run on an application server. This application server may be anything from a single server thread on one machine to a distributed application spread across many machines. Application servers are often termed "map servers" because each server instance typically corresponds to a geographic zone. In this implementation, when a player crosses from one geographic zone in the game to another, her client must change servers. This hand-off is typically achieved through a directory server, which can also redirect the client in case of server failure. This design is favored because players can only influence others in their geographic vicinity, so this provides a natural way to reduce the communication bandwidth between server instances. The treatment of zone borders leads to complex concurrency control problems, including real-time consistent propagation of visible effects across zone boundaries as well as the atomic client hand-off operation mentioned above. These issues are handled differently in every architecture.

Another technical challenge with persistent games is transaction management. In non-persistent games, data corruption can easily be handled just by ending the session. Persistent games, on the other hand, must react to such issues by rolling back incomplete or failed transactions. They do this by managing the world state entirely within a database, which may be disk resident or in-memory. In addition to guaranteeing transactional behavior, this also provides the world state with an opportunity to persist in the case of server failure.

The problem with this approach is that disk-resident databases are not designed to handle the load that these types of games require, and in-memory databases that may be able to handle such loads do not provide the required persistence. Game transactions are highly interdependent, with the outcome of each update likely to affect the next query. As a result, even using high-end commercial products such as Oracle, Microsoft SQL Server or BerkeleyDB, today's MMO games struggle to achieve much more than 500 transactions per second [7], and this transaction rate cannot be improved by simply adding more machines. In order to deal with this problem, most architectures have a database cache that sits in front of the database. This custom designed application handles the transactions in main memory and only periodically writes to the actual database for persistence.

The overhead of all this transaction management comes at a price. Even the fastest MMO games cannot handle more than about 10 frames per second [7] in their simulation engine. As with graphics threading in non-networked games, game designers handle this problem via interpolation. State changes in the simulation are represented at a very coarse level, which are then interpolated in the client in order to produce smooth results. For example, in *World of Warcraft*, if a player starts dancing, the world state database only notes that the player is dancing and for how long. The exact animation of the dance is entirely up to the client and may be visibly different between two different clients.

In addition to the world state database, persistent games have two other important databases. One is the monitoring database which serves the same role as for non-persistent games. The other is the accounts database. This is a standard database that acts as a gateway to make sure that the player is authorized to join the game.

To put all of the pieces together, let us walk through a sample session in an MMO using the architecture in Figure 2. The player starts playing by connecting her client to a login server, which checks that the player has a valid and current account with this game service. If so, the client is handed off to the directory server, which will assign her an application server. To do this, the directory server will load the player's state from the world state database to determine her physical location. Once assigned to an application server, the player interacts with all players on her server, which includes all those in her immediate vicinity. All computation of this interaction occurs either on the application server or in the database cache, but is rendered locally at the client. Should the player change zones or lose touch with the server, she is assigned a new one by the directory server. When the player quits the session, her final state is stored to the world state database so she can begin the game there when she returns.

3. RESEARCH DIRECTIONS

The presence of database technology in games opens up several opportunities for research. Many of these research possibilities, like query processing, query optimization, and indexing are traditional topics that need to be explored again with new assumptions. Others, like motion steering [25, 29, 30], introduce new problems to the database community. We survey here three broad research directions of which we have the best understanding so far.

3.1 Database Engines for Games Workloads

3.1.1 Query Processing

The most immediate problem for the community to address is that of new query processing requirements. The vast majority of database research is devoted to optimizing

queries for disk I/O. Games, on the other hand, need to process database queries at about the graphics frame rate, and therefore cannot afford to access the disk. Currently, high speed SATA drives like the WD Raptors have a uniform sustained transfer rate that tops out at 85 MB per second [14]. This means that a typical game that wants to run its simulation engine at 20 frames per second [17] can access only a little more than 4 MB per query-cycle. With gaming-grade PCs currently shipping with gigabytes of RAM, it makes much more sense in this context to focus optimizing query performance in memory.

There has been considerable work in the community on streaming [5, 23, 6, 11] and in-memory databases [3, 4]. However, games have unique workloads that present several new optimization challenges. In particular, games perform a fairly even mixture of queries and updates, organized in bursts of $O(n)$ queries followed by $O(n)$ updates, corresponding to the simulator frame rate. This means that any index structures developed for games must support an extremely high rate of churn. Solving this problem can open up whole new areas of query processing. For example, one of the more interesting discoveries in the development of the SGL language [33] was that, in some cases, it may be cheaper to completely annihilate an index and rebuild it, rather than to support the rebalancing behavior of the index. For example, suppose we have a game with n characters interacting with each other. In order to process a particular query efficiently, we need a d-dimensional orthogonal range tree. We can build a non-dynamic tree from scratch in $O(n \log^{d-1} n)$ time. A dynamic tree does not need to be rebuilt every frame; we can just remove and reinsert the character at a new position. If only k out of the n characters need to be updated, then this sequence of removal and updates costs $O(k \log^{d-1} n \log \log n)$ time [9]. The overhead of this dynamic structure can be significant if k is very close to n, reducing the number of frames per second that we can achieve. Hence, if we can batch the read-only part of game actions, as SGL does, to amortize the cost of rebuilding the index, the static index may be cheaper than the dynamic one.

3.1.2 Indexing

Games also present the opportunity for the development of a wide array of new index structures. The single greatest computational problem for games [19] is the $O(n^2)$ problem: if the action of each game object depends on every other game object, then this action takes $O(n^2)$ steps to compute, where n is the number of game objects. SGL identified aggregate indices as a straight-forward means to solve this problem [33]. However, it only provided indices for a limited class of aggregates. There are many important aggregates not covered by this work. For example, all of the range-dependent aggregates in SGL assumed that the players were out on a battlefield with no obstacles blocking visibility. In order to take obstacles into account, we need to develop aggregate indices that are compatible with visibility graph structures, like binary space partition trees.

In developing these new indices, the research needs to be aware of their memory footprint as well as their performance. Just as it is too expensive to read game data from the disk each frame, the indices must reside entirely in memory. This can be nontrivial for high-dimensional index structures such as the orthogonal range trees used by SGL. A d-dimensional tree with fractional cascading takes $O(n \log^{d-1} n)$ storage. In a 32-bit address space where the aggregates being computed are all doubles, an index for a 4-dimensional orthogonal range query (e.g. x-position, y-position, armor strength, and health) in a game with 10,000 objects would require a structure nearly 0.5 GB in size. This is 1/8th the address space of a 32-bit machine, and thus supporting multiple such indices is clearly untenable. While some of this problem can be solved by moving to a 64 bit machine, this causes the pointer size to double and so even more memory is necessary.

3.1.3 Engine Support for New Languages

In [33] the authors showed that translating game AI scripts into relational database queries and optimizing them allows for increased scalability. These game scripts, however, call for a few extensions of the query language in order to support such new features as randomness or combining the effects of simultaneous actions. We need to develop efficient query processing and optimization techniques for these extended query languages.

3.2 Adaptation of Game-Specific Algorithms

3.2.1 Steering

We need to adapt game-specific algorithms to be compatible with these query engines. Motion steering is a canonical example of this. To understand what we mean by motion steering, we need to understand how motion planning works in games. Traditionally this planning is achieved at two levels [25]. In the first level, classic pathfinding algorithms such as $A^\star$ are used to find the shortest path through a collection of static, unmoving objects; this path is computed once, before the character begins to move. However, once the character starts moving along this path, we need to worry about collisions with other characters. This is achieved using artificial potential fields, which gently push a character away before a collision happens [30]. Potential fields calculations must be queried every frame, and therefore should be processed by the query engine. As the field must be computed anew for each game object, in a game with n moving objects this computation is $O(n^2)$ – and thus expensive – when these objects become crowded together.

Existing potential field algorithms are not amenable to traditional aggregate indexing techniques. For example, if a game object is at position p_u, then the computation of the potential field may require us to evaluate the function

$$F(p_u) = \sum_{u' \neq u} \frac{1}{||p_u - p_{u'}||}$$

for each object u. We cannot do this with a sum index, be-

cause the values that we want to sum are different for each location p_u. Fortunately, computing these algorithms exactly is not important; it is only important that the behavior "looks correct" onscreen. Thus, if the algorithms can be altered in such a way that they are amenable to aggregate indexing, this can help greatly with game performance.

3.2.2 Set-At-A-Time Processing

Another way in which game algorithms need to be redesigned to take advantage of database technology is to make them more amenable to set-at-a-time processing. If one action depends on the result of another (e.g. a character cannot steal gold from a chest if another character steals it first) then it is difficult to process these actions as a single query. SGL handles this problem through a computational model that supports simultaneous actions [33]. While games already use this computational model for a large part of their design, some actions are easier to model than others. For example, steering algorithms are not always perfect, and collisions may occur. When this happens, the simulation engine needs to resolve the collision before the results are rendered onscreen. The problem then, is to determine how to resolve these collisions in a set-at-a-time fashion. In traditional game engines, objects move one at a time, so if there is a collision, the object can be moved back to its original location. If we move all of the objects at once, we need to be concerned about a second object moving into its original location, thus preventing us from undoing the move.

3.3 Consistency in Networked Games

Players in a networked game may be geographically dispersed, so speed-of-light round trip times are on the order of several tens of milliseconds. Actual measured RTTs can be significantly longer, exceeding the simulation frame rate. These delays make consistency a serious problem. So far we have seen two design points for networked games:

Non-persistent Games. A sophisticated non-persistent game uses a P2P architecture and optimistic concurrency control. Each instance is authoritative about the player and game objects that it holds. For the remaining objects, the instance optimistically simulates operations and rolls back if the result eventually received from the authoritative instance is in conflict. This means every instance is either executing (authoritatively) or simulating (optimistically) the entire game state. Note the number of network connections grows as the square of the number of instances. So this approach generally cannot scale beyond a handful of instances.

Persistent Games. In a modern persistent game, all game computation is done at a central server, which is usually divided into zone servers internally as discussed above. This approach does not simplify computation at the client very much. Effectively, the client is not authoritative about *anything*, but because of the long RTT between client and central server, each client must execute its own actions optimistically (in order to show them to the user in near-real time) and then resolve conflicts as they are received from the server. Moreover, the central server is a scaling bottleneck.

We need to research ways to integrate these two designs to achieve greater scalability and potentially lower average latency. One possible approach is to use optimistic concurrency control against a central server (as in the persistent case) but allow clients to be authoritative about objects they hold (as in the non-persistent case). Making clients authoritative allows a design in which the central server makes serialization decisions by ordering events in the virtual world, but the server is not required to execute any of the game logic, thus making it more scalable. We could then use locality properties of game moves to limit the amount of speculative computation required at each client to only those computations needed to display relevant nearby state to the user. This is roughly equivalent to the division of a central server into zone servers as discussed earlier. Using the zone assignments, the server could route to each client only the moves and game state data needed for its local computation.

4. CONCLUSIONS

In this paper, we have illustrated how databases fit into computer games and research areas in which the database community can contribute. As a final word, aside from being an interesting area for new research problems, we think there are two important reasons why the database community should embrace games.

The first reason has to do with multicore architectures. The computer science research community has been riding Moore's law for the last four decades, and the exponential increase of clock frequency has translated into exponential performance growth for years. Worries that Moore's law cannot continue have been looming on the horizon, and the computer architecture community has developed ingenious ways to address some of these ailments. For example, relative slow memory speeds have been addressed with caches, out-of-order execution, and speculative execution; low utilization of processor resources has been addressed with simultaneous multithreading, and so on. However, recently clock frequency growth stalled because the power dissipation trends became unmanageable. We now hope that we can ride Moore's law again by doubling the number of cores in every new generation of processors, resulting in the next wave of exponential performance growth through massive parallelism. The database community has much expertise with parallelization of database queries. Injecting database processing models into computer games may thus also be an investment into the future for an easy transition to a highly parallel programming model.

Second, recent years have witnessed a significant drop in enrollments in computer science at American universities. Many universities are currently developing programs in computer games. For example, at Cornell we have an undergraduate minor in computer games that is associated with the Game Design Initiative at Cornell University (GDIAC; `http://gdiac.cis.cornell.edu`) in which several

of the authors of this paper have become heavily involved. In GDIAC courses, students work cooperatively across disciplines and years. A typical GDIAC student group involves artists, writers, musicians, and programmers, all working together to make an original game. All students engage in the game design process, planning and refining game rule systems, mechanics, and interfaces. Attracting students to the field of computer science through computer games is one way to convey the excitement of our field [34], and we believe that the database community is poised to contribute.

Acknowledgments. This work is supported by the National Science Foundation under Grant IIS-0725260, the Air Force under Grant FA9550-07-1-0437, and a grant from Microsoft Corporation. Any opinions, findings, conclusions or recommendations expressed herein are those of the author(s) and do not necessarily reflect the views of the sponsors.

5. REFERENCES

[1] Entertainment Software Association. 2006 sales, demographic and usage data: Essential facts about the computer and video game industry. http://www.theesa.com/.

[2] P. Bettner and M. Terrano. 1500 archers on a 28.8: Network programming in Age of Empires and beyond. In *Proc. GDC*, 2001.

[3] P. Bohannon et. al. The architecture of the Dalí main-memory storage manager. *Multimedia Tools Appl*, 4(2):115–151, 1997.

[4] P. A. Boncz and M. L. Kersten. Monet: An Impressionist Sketch of an Advanced Database System. In *Proc. Basque Int. Workshop on Information Technology*, San Sebastian, Spain, July 1995.

[5] D. Carney et. al. Monitoring streams — a new class of data management applications. In *Proc. VLDB*, 2002.

[6] S. Chandrasekaran et. al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[7] B. Dalton. Online gaming architecture: Dealing with the real-time data crunch in MMOs. In *Proc. Austin GDC*, Austin, TX, September 2007.

[8] B. Dawson. Game scripting in Python. In *Proc. GDC*, 2002.

[9] M. de Berg et. al. *Computational Geometry: Algorithms and Applications*. Springer Verlag, 2nd edition, 2000.

[10] M. DeLoura, editor. *Game Programming Gems*, volume 1. Charles River Media, 2000.

[11] A. Demers et. al. Cayuga: A general purpose event monitoring system. In *CIDR*, 2007.

[12] M. Dickheiser, editor. *Game Programming Gems*, volume 6. Charles River Media, 2006.

[13] Screen Digest. Western world MMOG market: 2006 review and forecasts to 2011. http://www.screendigest.com/reports, March 2007.

[14] Western Digital. WD Raptor WD740ADFD. http://www.wdc.com/en/products/products.asp?driveid=244.

[15] D. Fu, R. Houlette, and R. Jensen. A visual environment for rapid behavior definition. In *Proc. Conf. on Behavior Representation in Modeling and Simulation*, 2003.

[16] Epic Games. http://www.unrealtechnology.com. Corporate Website, 2007.

[17] Intel. Threading games for performance: A one day hands-on workshop by intel. In *Proc. GDC*, San Francisco, CA, March 2007.

[18] D. Kazemi. Gameplay metrics for a better tomorrow. In *Proc. Austin GDC*, Austin, TX, September 2007.

[19] P. Kruszewski and M. van Lent. Not just for combat training: Using game technology in non-kinetic urban simulations. In *Proc. Serious Game Summit, GDC*, San Francisco, CA, March 2007.

[20] J. Lee, R. Cedeno, and D. Mellencamp. The latest learning - database solutions. In *Proc. Austin GDC*, Austin, TX, September 2007.

[21] D. McGrath, M. Ryan, and D. Hill. Simulation interoperability with a commercial game engine. In *European Sim. Interop. Workshop*, 2005.

[22] Microsoft. XNA developer center. http://msdn2.microsoft.com/en-us/xna/default.aspx.

[23] R. Motwani et. al. Query processing, approximation, and resource management in a data stream management system. In *Proc. CIDR*, 2003.

[24] A. Mulholland and T. Hakala. *Programming Multiplayer Games*. Wordware Publishing, 2004.

[25] J. O'Brien and B. Stout. Embodied agents in dynamic worlds. In *Proc. GDC*, San Francisco, CA, 2007.

[26] Bureau of Labor Statistics. American time use survey. http://www.bls.gov/tus/, 2006.

[27] S. Posniewski. Massively modernized online: MMO technologies for next-gen and beyond. In *Proc. Austin GDC*, Austin, TX, September 2007.

[28] M. Prensky. *Digital Game-Based Learning*. McGraw-Hill, New York, 2001.

[29] C. Reynolds. Steering behaviors for autonomous characters. In *Proc. GDC*, 1999.

[30] B. Stout. Artificial potential fields for navigation and animation. In *Proc. GDC*, 2004.

[31] M. Thamer. Act of mod: Building Sid Meier's Civilization IV for customization. *Game Developer*, August:15–18, 2005.

[32] Various. Artificial intelligence in computer games. Roundtable Discussion at GDC, San Francisco, CA, March 2007.

[33] W. White et. al. Scaling games to epic proportions. In *Proc. SIGMOD*, pages 31–42, 2007.

[34] M. Zyda. Introduction: Creating a science of games. *Communications of the ACM Special Issue: Creating a science of games*, 50(7):26–29, July 2007.