Space Marines

A Distributed Online Shooter

Space Marines is a two-dimensional shooter whose aim is to provide a healthy conduit to the pursuit of happiness for all ages. It is implemented using a distributed hybrid peer-to-peer architecture and seeks to abate the role of latency by utilizing concepts found in dead reckoning to extrapolate entity positioning.

Allen Brubaker, Eric Coffey

Contents

Space Marines

A Distributed Online Shooter

Introduction

Space Marines is a two-dimensional shooter game in which the primary objective is to blow other players up. The application is written in C# and leverages the Microsoft Silverlight application framework to enable the game to be played easily online. The underlying network architecture utilizes a simple server-based system to set up games and tracks clients but primarily communicates between clients in a hybrid peer-to-peer fashion with the server simply rerouting packets sent between clients. All this is accomplished with the use of sockets officiated by the transmission control protocol (TCP). The application is able to handle moving and shooting in real time and resolves issues with latency between clients using the dead reckoning technique in order to minimize network traffic (1). Various features implemented enhance gameplay and include player and weapon leveling logic and power-ups instilling sundry boons including stamina and health gain. The game additionally supports a large range of complementary features including chat functionality, GPU utilization, a server polling process to maintain connectedness, all while operating within an internet browser.

Problem

A distributed system can be defined as a system "in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages" (2). Distributed systems bring with them a host of boons including the ability to share resources remotely, allowing clients to dispense with local storage and computational needs as they acquire these by perhaps renting services from a cloud, and even allowing more seamless and collaborative communication between teams of people located in different continents. With the advent and continued proliferation of such systems, it is of paramount concern that distributed system architects remain cognizant of their challenges. These challenges include not only the fact that communication between processes in the system through the channels of perhaps the internet suffer from high latency, reliability concerns, and malicious attacks, but the heterogeneity, scalability, extendibility, security, scalability, partial failure, concurrency, and transparencies to the end user present major design issues as well (2).

Not only do popular applications domains such as finance and commerce, informational society, education, transport and logistics, and the sciences maintain the need for distributed systems, but multiplayer gaming also hinges on the need for distributed systems and complex distributed algorithms and hardware configurations to support player populations in upwards of tens of thousands. Many of the distinct distributed challenges for gaming include "the need for fast response times to preserve the user experience of the game, real time propagation of events to many players, and maintaining a consistent view of the shared world" (2).

The desire to create Space Marines stems uniquely from these challenges. It was desirable to not only investigate how an online game presents a unique challenge to distributed systems in general, but how the unique need for true real-time systems should be handled. One of the most basic concerns in a distributed system is the inability to directly share memory, and this case was no exception. Therefore, it was essential in the presented distributed game to establish a method of passing messages that allows the clients to communicate with one another and to do so in a timely fashion. Space Marines solves this using one of the lowest level solutions - sockets and the TCP protocol. To achieve the best possible performance, all messages are sent as infrequently as possible and carry a minimal amount of data.

The choice of building a game was enticing due to the fact that it would not just be enjoyable to develop and play over the increasingly mundane single-player experience evident in many such games and would test and refine programming ability, but it would give a first-hand perspective how distributed issues are dealt with in many of the current multiplayer games by emulating some of their solutions. A constant in every multiplayer game is latency. Distributed games approach latency resolution in a large variety of ways, but true real-time gaming is a difficult concept to implement properly. Major points of contention include handling concurrency issues as players in real-time concurrently modify the game world, preventing congestion due to the rapid rate of updates, and keeping the global game state properly propagated and updated. Space Marines attempts to solve each of these issues by utilizing a hybrid peer-to-peer methodology and enforcing each peer to keep his/her entities updated on the other peers' systems by sending periodic updates to these cloned entities should projected positions stray too far from the actual positions. This concept is known as *dead reckoning* (1).

Space Marines

Gameplay Mechanics

Space Marines is a dynamic real-time multiplayer game. As users navigate to the website, a user is immediately ushered into the heated gameplay. All players are pitted against each other in a bustling free-for-all arena where the goal is to join alliances to defeat players with the largest levels from vastly overpowering all his/her other opponents.

Each user appears as a small white circle within the game playing field, which is a rectangle encompassing all facets of the game. Once a user is given a unique circle within the game playing field, the user can move the circle in any one of eight directions depending on the key pressed. The normal "WASD" keys are used for movement as is common in standard multiplayer shooter games. The keys W, A, S, and D correspond to north, west, south, and east, respectively. For instance, if the user holds the W key down, the player entity (white circle) moves left at a constant rate until the edges of the game board are reached, whereupon holding this key down yields no effect. Similarly the W and A keys can be simultaneously held down to move at a constant rate in north-west diagonal direction. Again game boundaries always prevent user actions beyond their borders. It is important to note that when none of these keys are pressed the player is immediately made immobile--no acceleration or compounding velocity is incurred the longer a user holds down these keys. In order to aim, the user simply needs to hover his/her mouse pointer over the intended target location on the game board and press a certain key corresponding to one of guns equipped. By holding down this key a steady stream of bullets for the appropriate gun is fired in the target direction. All bullets are removed from the edge of the game board once reached.

Each player is equipped with one of three guns: the sniper, the bomber, and the machine gun. The left mouse button fires the machine gun, the right mouse button fires the sniper, and the E key fires the bomber. Each of these keys can be held down, but bullets are only fired based on the pre-established cool-down rate found on each gun. The purpose of the machine gun is to fire many bullets at low velocity so the goal of this gun is to reward player skill in dodging bullets. To that end, the cool-down rate on this gun is a modest 1 second. The sniper, on the other hand, fires much faster and more powerful bullets. Bullets of this caliber are shot much less often; once per five seconds. Finally, the bomber gun is the most powerful gun. Cool-down rate on this gun is a non-issue since bullets can only be picked up in special cases if a user picks up the appropriate “bomb” power-up off the ground. This type of bullet is by far the largest, fastest, and most damaging bullet in the game; a large ominous ball comes careening at players and causes immediate death upon contact!

As players successfully hit another player with a bullet, the bullet disappears and damage and health loss is incurred on the target player. If the health of a player reaches zero, the player dies and gets resurrected with full health on a random area of the map. When this occurs, two events take place. Firstly, the killer levels up, and secondly, the dead and now resurrected player loses a level. When a player levels up, all aspects of his/her character upgrade. This includes the amount of armor surrounding a player, which more effectively mitigates all future damage in addition to guns leveling up. As guns level up, the amount of bullets emitted from the gun barrel increase, as well as their individual damage, speed, and rate-of-fire. All bullet damage provides immediate feedback to player based on their color. The higher the wavelength of the color the more damage a bullet inflicts. In other words, any bullets that contain a more reddish color do the least amount while those with purple do the most amount of damage. Dead players that lose a level simply lose any additional effects gained from being at that level. All players start at level 0, cannot level down below 0, and there is no maximum level imposed.

Players can quickly assess their health and the health of their opponents by the small health globes that rotate around any player entity in the game. There are 10 health globes and each represents 10% health. As a player is damaged and health is lost, health globes vanish. Once all globes vanish, a player dies and is resurrected.

During gameplay, various power-ups are spawned at random time increments across the map and give a player who runs over them a unique temporary boon. Each unique power-up appears as a rotating ellipse with a different color depending on its type. There are six possible power-ups. These include invisibility, invincibility, stamina, health, level, and bomb power-ups. Invisibility causes all players to lose sight of the current player for an amount of time. Invincibility causes a player to become invincible. Stamina gives a player additional speed for a time. Health grants full health to an ailing player. Level allows a player to instantly level up, and bomb grants exactly one powerful bomb bullet to the bomber gun.

The goal of the game is a simple one yet hard one to attain without extraordinary skill: remain the dominant player vastly overpowering all other players in terms of prowess and level regardless of any makeshift alliances and vain attempts by lesser underlings to control the tide of war.

Enhancements

Various features have been to the game in order to enhance game value and ensure smooth gameplay. The first enhancement to gameplay value is the design of the game board in such a way that the game is tailored to any device that connects. The game board is represented as a rectangle with a static aspect ratio of 16:10 width to height. Depending on the device used to connect to the website and thus the game, the game board dynamically resizes itself to best fit the provided form factor. Because no one player should be given unfair advantage by seeing more of the game board than another, all players see the same game board fully from end to end. If smaller form factors are used, the game board is resized such that all elements on the game board are scaled down. While the game playing field dynamically scales itself, the aspect ratio of all elements including the game board width and length remains constant. Thus all devices connected to the game achieve unparalleled gameplay value as all entities in the game maintain high fidelity as the game is scaled to fully fit the device at hand.

The next important feature implemented is the fact that all entities in the game including processing of translations, scaling, clipping, and animations on these elements, are offloaded to the graphics processing unit. Doing so provides extremely fast processing of all elements on the board with very minimal CPU usage. This keeps the frames per second extremely high, providing a level of fluid seamless gameplay needed in a real time game, and allowing more concurrent users in the system. Because the GPU was used, shadows are added to all entities in the system to aid in providing more realism. It was found that adding shadows to entities while using solely the CPU made gameplay stunted and unplayable. Animations on many entities including the rotating health balls and the rotating power-ups simply would not be possible without the aid of the GPU.

Additionally, the game provides a chatting mechanism to facilitate communication between players in the game. By simply pressing the ENTER key, typing a short message, and again pressing the ENTER key to send the message, players can collaborate their efforts, shun their opponents, and gloat their performance.

The game includes a server polling process to maintain connectivity. This allows the game to automatically check to ensure that the server is indeed online and communication between all clients is kept intact. The instant the server goes down, all entities are removed from the game, and a message conveniently informs the user that the server has crashed or has gone offline. If this is the case, the game constantly polls the server and reconnects when the server goes back online. This is all to give the user the niceties of simply making sure the game is loaded within the browser, while not having to repeatedly press the browser's refresh button.

The last important feature of the game is the dead reckoning smoothing function used to smooth entities between updates sent as a result of dead reckoning. The smoothing utilized is both a linear and a Bezier cubic spline interpolation algorithm, although the latter currently provides the more realistic seamless smoothing and is the one currently utilized in the game. It smoothly transitions between old and new entity by taking into consideration both their current positions, directions, and velocities. Both the need for smoothing and a more detailed look at the interpolation mechanism are discussed below.

Dead Reckoning

The use of the technique known as dead reckoning (1) distinctly governs all facets of the design of Space Marines including its architecture. The following will detail exactly what this concept is, the benefits gained from using such a system, and how Space Marines was designed to cater to the requirements of a dead reckoning system.

The mechanics of dead reckoning is as follows: All clients in the system agree on a predetermined set of algorithms used to extrapolate and predict movement on all entities in the system (1). At any given time each client in the system has a set of all entities in the system, which collectively makes up the global game state; thus, the global game state is duplicated on each client. Of all the entities making up the global game state appearing on one client, say client X, a subset of these entities are those that were created by this client X. Such entities can be called "owned" by the client. Of these entities owned by client X, a duplicate set of entities called "virtual" entities exist—one for each owned entity. It is important to understand that each entity contains a protocol data unit (PDU) that not only contains identification to uniquely identify every entity within the system, but it also contains information important for extrapolating movement by all clients including the position, speed, and direction of the entity (1). The virtual entities contain the exact same PDU as their original entity from which they were conceived. All entities are moved per frame according to their PDU and the pre-determined set of algorithms. The set of virtual entities' are broadcast to all remote clients in the system as they are created. Thus the virtual entities are images or snapshots of what are currently happening on remote clients. The only difference between the set of "owned" entities and their virtual counterparts are that the owned entities are subject to change at any time based on the interactions of the user at client X. As change takes place a new PDU is generated, but the PDU on the virtual counterpart remains the same since no new remote update has been sent yet.

The moment the position of the actual owned entity exceeds the position of the virtual counterpart by a threshold known as the dead reckoning threshold, it means that remote clients currently have a now-deemed “dirty” copy of the actual entity. Thus a new “clean” updated copy of the actual owned entity is created, and the PDU of this virtual entity is propagated to all remote clients in the system and updated accordingly. The system has now been incrementally updated so that all global game states appearing on all clients in the system are back in sync. Thus individual updates are extrapolated and incrementally updated per frame quickly by each client without the need to wait for remote updates from other clients before traversing frame by frame, thus leveraging and being only limited by the power of the processing power of the client computer.

One more important concept in the use of dead reckoning is the need for smoothing algorithms. Notice that when an update is sent it means that virtual copies are out of sync with actual copies. Even while they are in transit to the remote user, the virtual copies become more and more out of sync until they are updated. The moment they need to be updated they are invariably far from where they need to be. If immediate updating commences, a jerkiness or “teleportation” effect occurs and degrades gameplay. Thus smoothing algorithms including such techniques as linear and cubic spline interpolation can be used to smoothly transition.

The reasons for using dead reckoning are manifold. Dead reckoning stipulates that game states are duplicated and simulated by each client in the system. This is especially beneficial to real-time systems that require a fluid system. Fluidity cannot occur if movements must at every instant be updated by a remote client because of latency concerns. Gameplay of around a hundred frames per second is needed for smooth gameplay, but if one were to solely update entities based on remote packets, frames per second, or more appropriately “actions-per-second” would be limited to how many packets could be sent per second, which is at best only around 10-20. However, because each client moves all entities per frame based on these pre-determined algorithms – extrapolated based on entity characteristics (PDU), which is only limited by the speed of the processor and video card—processing is much more fluid, realistic, and gameplay quality is preserved.

Secondly, the alternative to using a dead reckoning system is to cause the game-state to be kept by the central server with all clients sending updates and constantly polling the server for updates of the global game state as often is as feasible—normally ten or twenty times per second. Clearly requiring that the server send the entire game state each update drastically increases traffic and latency to clients retarding communication speed. To combat this problem, the server might potentially only send updates as is needed, but this requires that the server maintain information on each client as to what has been updated, the updates that have been sent, which have been received, etc. It needlessly complicates design, increases implementation costs, and stunts scalability—all of the same issues which plague using a client-server distributed architecture over a more robust peer-to-peer design.

Let us now briefly introduce how dead reckoning is used in the point of view of Space Marines. Because dead reckoning is used, Space Marines uses a hybrid peer-to-peer architecture where each client maintains all entities in the system—the global game state. Each entity, whether it is a player, bullet, or power-up, is simulated or incrementally updated every frame by each client. Each entity contains a PDU consisting of the unique identifier to distinguish the entity (made up of client ID concatenated to local ID), and a ray, which contains other important information including entity position, speed, and direction.

Let us proceed from the point of a single client X in the system keeping in mind that all clients perform the same set of actions. Client X has a set of all entities in the system which makes up the game state. Every frame of the game, entities are moved a small amount based on the PDU. Let us now assume that client X needs to create a new player entity Y (this only happens once upon game-load). Firstly, the player white circle is created along with its PDU and a virtual copy Y' is made. Because the virtual copy previously did not exist, the virtual copy is remotely broadcast to all other clients in the system. Now let us assume that the user on client X now presses the W key to move north on the game board and that the threshold is 1 pixel. A new PDU is created for the actual player entity that has a positive velocity and direction of north. The virtual player entity remains in an inert state while the actual player continues to move upward at constant velocity per frame. Eventually the actual exceeds the virtual by 1 pixel, the virtual is updated to the current PDU with same position, velocity, and direction information as the actual, and all remote clients are updated with the new virtual PDU. On reception of the update Y' on the remote client Z, his/her current view of player entity Y needs to be updated to the PDU of Y'. To combat jerkiness Bezier cubic spline interpolation is employed. Thus Y smoothly interpolates to Y' position; the transition between the old and the new global game state is seamless.

By the mechanisms as dictated by dead reckoning, all peers collectively donate their resources to the game's functioning and further take part in maintaining the global game state. The injurious effects of high latency are mitigated by depending on each peer to extrapolate or predict other peers' actions. Unprecedented quality and seamless responsiveness is promulgated in the thrilling game ubiquitously known as Space Marines: The Final Frontier.

C# and Silverlight

A primary component of the project was the choice of programming language and the framework used to deliver the graphical effects. After some research into our various options, it was decided to use Microsoft's C# programming language and their Silverlight framework.

C# is a multi-paradigm language that "provides support for software engineering principles such as strong type checking, array bounds checking, detection of attempts to use uninitialized variables, and automatic garbage collection." (3) The language was designed with programmer productivity in mind, and to that end made a good choice for Space Marines, which needed to be implemented in a fairly short period of time. It boasted a number of useful methods and constructs which sped the development cycle. In addition, C# has fairly extensive support for writing applications in both hosted and embedded systems; it was important to consider if successful, Space marines may be able to be ported to systems other than Windows, which it was designed to work on.

In addition, Silverlight was chosen as the method to deliver Space Marines to the user. Silverlight is Microsoft's framework for rich Internet applications, and its runtime is executed as a plug-in for web browsers, making it accessible and easily installed by the average user. The application UI code is written in XAML (Extensible Application Markup Language), which provides support for advanced animations and vector graphics. It also supports a number of common video content, and has a wide range of powerful development tools available. Silverlight delivers content to its users by first executing some JavaScript when a user visits the page hosting the Silverlight application. The application code is then downloaded to the client's computer and unpacked from a .XAP file, which is essentially a compiled Silverlight application compressed to transfer more efficiently. Using these two as the base of the project, Space Marines was developed as detailed below.

Distributed System Implementation

An important discourse on the specifics of the gameplay and various features has been discussed, including the basic shooter mechanics, various power-up logic, and additional enhancements. In addition to this, adding to the gameplay are features such as dynamic resizing and graphics processing capabilities, along with a chat feature facilitating user communication. Also discussed was the motivation for why the game was created—not the least of which was the fact that it presented an excellent conduit in which to synthesize and tackle challenges present in distributed systems. Consequently, a careful discussion on the inner working s of dead reckoning, its benefits, and its actual usage in Space Marines was provided. What follows is an exposition on the inner workings of the system including architecture, communication design, and the core game engine logic.

Communication Primitives

Communication between the different components or processes in the distributed system of Space Marines all takes place by the use of sockets. Sockets provide a low level mechanism to pass various packets from process A to process B via pre-established ports. The reason such a mechanism was used over remote procedure calls, for instance, was because all aspects of the communication procedure could be controlled. This included the design of message passing objects, the marshaling process including serialization, and the protocol used, whether connection-oriented or connectionless (TCP or UDP). The TCP protocol was utilized due to its built in ability to prevent omission failures such as dropped packets or re-ordered packets; however, future refinements of the program might replace TCP with UDP due to the relatively small packets needed within the game and speed UDP offers due to less overhead.

Packets

The *Packet* object was designed to act as the message unit sent between processes in the system. It was designed to facilitate both efficiency and compactness. It contains two important components. The first component consists of a command enumeration specifying the type of packet it was contained in while at the same time implicitly describing the type of contents that were appended. The different commands utilized in the system included: ClientConnect, ClientDisconnect, ClientElected, ServerDisconnect, EntityCreate, EntityCreateMany, EntityUpdate, PlayerResurrect, and LogMessage. As one can see, typical commands allowed one process to communicate to another process in the system its intentions. For instance ClientConnect was used when a client connected for the first time while EntityCreate and EntityUpdate commands were used to send packet updates in response to dead reckoning. The contents were the second component of the packet object. This is simply of type *Object* from which all other C# objects derive. Thus contents of any sort could be assigned to this field, but in practice *Strings* were used for LogMessage, while *Entities* were used for any of the entity commands. Lastly, no values were used when the command was a simple directive such as ClientConnect or ServerDisconnect.

Marshaling

Once a *Packet* was populated with the required command and contents it was propagated throughout the system to remote clients. However, clearly the packet object itself could not simply be sent as it existed in its current form. A process of marshaling was needed to encode the message into bits of 0's and 1's. These strings of bits were then sent via the network to the destination process. On reception of the bits, a process called un-marshaling was needed to decode these seemingly unintelligible strings of bits back into its original message or *Packet*. The actual process of marshaling used consisted of three distinct parts.

Step 1 of the marshaling procedure implemented in Space Marines was the serialization and deserialization. A C# library entitled *DataContractSerializer* (4) was used here to aid in the process of converting not only a *Packet* to a string of bits but any object populating the contents field (second component of *Packet)*. It handled the dirty work of automatically converting first the object to its xml representation where various xml elements represented field names and attributes represented types and values populating those fields. It secondly converted the xml representation to a string of bits. The reason for using said library was its excellent ability to allow developers to explicitly control the serialization/deserialization process by explicitly labeling fields that needed to be serialized. Further the *DataContractSerializer*, when deserializing and constructing a class from its xml representation does not call the class's constructor. It assumes that all fields have already been correctly populated since they were once constructed by the sending process/client. It properly handles deserialization of all public fields marked *serializable*, and additionally calls OnDeserializing and OnDeserialized methods before and after the deserialization process. This was found to be pivotal in the design of the game and provided the needed decoupling between normal instantiation logic and deserialization logic.

Previous implementations of SpaceMarines consisted of the design where the entire entity wasn't serialized but an internal EntityInfo was serialized and propagated instead. An entire entity could be populated based off this EntityInfo class from the way the entity class was designed. One can think of the EntityInfo class as the PDU (refer to dead reckoning). The entity class manually converted all its containing fields to their xml equivalent and converted them back into objects on deserialization. Thus a manual encode and decode method had to be overridden on each entity type (Bullet, Player, and Pickup) in order for EntityInfo to know how to serialize them. Needless to say, the process was cumbersome and definitely did not facilitate future scalability of the system as new fields or entities are created. Thus the DataContractSerializer was used, theoretically giving the same performance gains (all information is converted to xml first and only on fields needed) while automatically handling serialization/deserialization logic.

Step 2 of the marshaling procedure entailed the use of a popular, fast, and optimized compression algorithm known as QuickLZ (5). It utilizes the LZ77 encoding and is “a fast lossless compression algorithm code in 386 compatible assembly” (6). As discussed by its creator: “QuickLZ is the world’s fastest compression library, reaching 308 Mbyte/s per core. [It is] simple to use and easy to integrate [and contains a] streaming mode for optimal compression ratio of small packets [along with] auto-detection and fast treatment of incompressible data” (5).

Once the bits were compressed so that messages were as small as possible, step 3 of the marshaling process included construction and prepending a header which simply included the length of the string of bits output after compression. The header was stored in 2 bytes and prepended to the string of bits allowing for a maximum bit length of 2^16. The reason for using a header was to properly distinguish one string of bits belonging to one packet from another string of bits belonging to another packet upon un-marshaling at the receiving end. C# and its TCP handling logic uses Nagle’s algorithm and waits a length of 200 milliseconds to see if any other packets arrive in the outgoing buffer at a given client location (7). It then concatenates all such packets together and sends them. The reasoning behind using Nagle’s algorithm is to reduce network traffic by concatenating packets sent in rapid succession to each other all in a single large message. Because each TCP packet consists of 40 bytes of header for each packet, concatenating all these packets means only one header is needed for n packets instead of n headers (7). Thus a method for distinguishing packets from another was very important to the marshaling process.

System Architecture

Now that a proper understanding of the underlying message passing primitives used in the distributed system has been discussed, along with the procedure for marshaling the messages between the processes, it is now important to discuss the architecture of the system in terms of how the different processes in the system interact and are related to one another.

Hybrid Peer-to-Peer

The technology used to design Space Marines is known as Silverlight. Although Silverlight added the ability to design the entire game in the browser, it has stricter security restrictions, adherence to which forced redesign of pivotal components in the system. Furthermore, it lacks the depth of libraries provided by the full .NET framework since its goal was brevity—to use Silverlight, clients must download a plugin similar to the way adobe flash requires a plugin. The smaller the plugin is, the less obtrusive the technology. Thus, instead of implementing the peer-to-peer solution which was originally intended for the project, a hybrid peer-to-peer architecture was designed. This was to cope with the challenges invoked by Silverlight and specifically in this case the inability to directly communicate between individual peers in the system. A method of indirect communication by means of UDP multicast is supported but has yet to be investigated in the design of Space Marines. The hybrid peer-to-peer architecture consists of peers talking as directly as possible with other peers but with the added restriction that all messages must be routed through the server first. The server performs no un-marshaling of the string of bits received nor has any concept of any facet of the game, game objects, or game state, but simply broadcasts the bits unmodified to every other client in the system.

Server View

As discussed above the server plays as minimal a role in the hybrid peer-to-peer architecture as possible. It was desirable that clients maintain full control and processing of the game state without being inhibited by the server by any means. This was to stray as much as possible from the disadvantages of a full-fledged client-server solution due to its inability to scale properly and maintenance costs. Another limitation of Silverlight was its ornery focus on security. It requires that in addition to implementing a normal game server for the functioning of the game, a policy server needs to be implemented. Upon connecting to a remote server for the first time, a policy server on port 943 is sent a message discreetly requesting a small policy file. The file acts as a contract made between server and client so that all future communication can be done with trust. Because of these limitations a scalable system design was created such that any arbitrary amount of servers could be created, listen on any pre-specified Silverlight enabled port (ports 4502-4534), handle an arbitrary number of clients, include custom logic called on client connect, receive, and disconnect, and spawn a new graphical user interface complete with logging information. In its current state, the game includes both the requisite policy server and game server.

After the game server is running as a process on the internet assigned a specific IP address and listening to one of the specified Silverlight ports, assume that a client connects to it. The moment a client connects, firstly a client ID counter is incremented by one and assigned to the client. Secondly, the server determines if the client should be the sole coordinator in the system by checking if it is the only client in the system at this point. Thirdly, the server instantiates a new client object, which in turn spawns a new thread and listens for messages from that specific client via its unique socket. Thirdly, the client object is added to the client table, a list of all clients currently in the system (game). Immediately the client is sent the following packet: Packet(Command.ClientConnect, “ClientID IsCoordinator”) which the sending client is currently synchronously waiting for.

When a new message is received via the socket, its contents remain unaltered and broadcast to all other clients in the system—this information is found by iterating the client table. Recall messages in this hybrid peer-to-peer architecture are simply rerouted without any processing and adheres to the original peer-to-peer design sought when Space Marines was first conceived.

When a client disconnects either because it crashed or properly shutdown and severed its socket connection, a 0 bit message is sent. On reception of this empty message, the server understands that the client has shut down; therefore, the server broadcasts a ClientDisconnect packet to all other clients in the system, whereupon they remove the associated entities from their game simulations. The server-side client object corresponding to this disconnecting client is gracefully shutdown and cleaned—the listening thread is aborted, sockets are shut down, and the client is removed from the client table. If this client was the sole coordinator in the system, a new coordinator is chosen as the client with the smallest id remaining in the client table. This new client is then sent a ClientElected packet.

Client View

Assume that a new user enters the game on his local machine and is now the new client entering the system. Immediately, the client connects to the system and the following series of events take place: A new socket is created with the destination IP address and port that the game server is waiting on. The connect method on the socket is invoked and behind the scenes the policy server is queried and the policy file is downloaded. Once the contract is established between the new client and server, the socket connection call proceeds to call the game server. The client immediately synchronously waits until it receives a response from the server, which includes its client ID within the system and if it is indeed the sole coordinator client in the system. Subsequently, a new thread is spawned to listen to any messages sent through the socket from the server.

When any message is received through this port in the listener thread, who is at all times synchronously waiting for receives from the server, the thread handles breaking up the messages into distinct segments depending on the length ascertained from the header as discussed above. Each array of bits is then un-marshaled and a collection of Packet objects are obtained. Each packet is then sent to the appropriate method and processed according to its command. In the case of the EntityUpdate command, the EntityUpdate(Entity) method is called and sent the EntityUpdate packet contents (type *Entity* in this case). It then locks the list of entities (recall this is a multi-threaded environment and care must be taken to provide mutual exclusive access to shared resources between threads), removes the old entity from the list, and adds the new one. After each packet in this iteration is processed, the listener thread again synchronously waits for any new string of bits from the server, whereupon the entire procedure is repeated.

No special process is done client side if the client desires to disconnect because the server can properly update all other clients on the disconnection status of the client even if the client shuts down. This is done to ensure robustness within the system.

Game Implementation

Space Marines supports a wide array of features over the base mechanics of the shooter style gameplay. All of these in concert provide a smooth and entertaining gameplay experience for all users in the game. In order to support such features a properly designed system should support the underlying mechanics of the game in such a way that maintainability is realized, finesse in interaction between the various components exists, and errors in logic are all carefully expunged. The following details the implementation details of the game and how it works together to supplies the game play features detailed above. Firstly, the server polling mechanism is discussed. This then gives way to a discussion on the main game engine loop that is called every frame of the game. Finally, a look at the various objects existing in the game including players, bullets, and power-ups is given.

Game Manager

The game manager provides immediate feedback to a user when the remote game server is shut down thus inhibiting any further processing. It also facilitates automatic reconnection to the server if it goes back online. It handles connection logic and ensuring that the game is at all times either functioning properly or should be gracefully shut down due to communication to the server being severed. Its process is as follows: The game manager runs on its own thread and every second checks to make sure that the socket is still connected to the server. If all the sudden it finds that it is no longer connected, it stops the main game object, which in turn clears the main game board and displays a server offline message. At this point the game manager tries to connect with the socket to the server every second. The moment a successful connection is made, it sits and waits for the ClientConnect packet including its ClientID and IsCoordinator string. On reception, the game manager passes these arguments to the game and restarts the game. Additionally, it restarts the listener thread to listen to incoming messages from the server. When the game is restarted, all entities are created from scratch and no memory is maintained between the sessions. In this way the game at all times is either connected or gracefully cleared based on the socket connection status, and the game manager is the object that makes this possible. It is implemented so that the user need not continuously press the browser's refresh button to check to check if the server has indeed come back online.

Game Engine

The game object acts as the engine of the game and provides the nuts and bolts for its major functioning including the frame by frame logic needed to progress its state. What follows is an exposition on the major components or methods of the object including the methods called by the game engine to both start and stop the game and the event handler called frame by frame by the Silverlight renderer.

Start(ClientID, IsCoordinator)

The start method is called by the game manager as detailed above. Notice the client ID and IsCoordinator parameters that are passed to the method providing the vital components to proper functioning of the game engine. Without the client ID, entities would not be adequately differentiable from each other in the entire distributed system spanning multiple remote clients. Further, dead-reckoning would simply not function correctly since the concept of which client owns which entity would be impossible. Once the parameters are populated in their class fields, gun objects are created and attached to the appropriate keys.

Next, the player entity is created and assigned the client ID along with a client wide entity ID being incremented as well. Once it is instantiated it is placed in a random position on the game board and is remotely propagated to the other clients via the EntityCreate packet. The newly created player entity is added to the list of entities, henceforth labeled EntityList. Finally the Render() event handler is attached to the Silverlight renderer to start receiving per frame invocations.

Stop()

When it is time to stop the game due to socket disconnections and server crashes, the game manager calls the stop method. Basically this method removes all entities from the game board and the EntityList, displays a “Server Offline” message conveniently in the entities’ place, and the renderer is suspended. Recall that all entities are deleted and thus no memory is kept between sessions of successive method calls to Start and Stop. No remote packets need to be sent since the central point of communication between entities is down; implicitly all clients perform the same actions, and global game state is preserved and in this case cleared.

Render()

As discussed previously the render is responsible for handling the core of the game engine logic since it is called every frame. Indeed it handles not only entity movement, bullet constructions upon a successful fire, but handles spawning power-ups, collision detection, and game board cleaning (removing entities that moved out of bounds). The set of actions taken each frame is as follows: A delta time value Delta is calculated consisting of the number of seconds since the last frame was called. This delta value is used to calculate the frames per second of the game and other movement calculations.

Next, based on the current keys pressed down, a method calculates the new vector that the user is moving in: recall this can be in one of eight possible directions and is based on the WASD keys and any combination of them. For instance if the keys W and S are held down concurrently special logic determines that since, assume for instance S was held after W, the S key takes precedence over the W key and the vector returned is <0,-velocity>. This is done using a special queue. Assume now for instance that non-conflicting keys W and A are pressed down. The vector returned from the method would be $velocity *$ (velocity * √(2)/2 * <-1, 1>). The length of the vector must be the velocity. Once the player movement vector is calculated the player entity’s ray is updated with the correct vector.

The next method called is the GunFire() method. This handles determining if bullet entities should be spawned and remotely propagated throughout the system. Once bullets are fired all the bullet entities are added to EntityList. This method will be discussed in more detail below.

All entities in the system are incrementally moved based on the local ray object attached to each entity by Delta amount. All aspects of the ray determine the next position of the entity including position, direction, and velocity. Not only are all entities moved but all virtual entities of the actual owned entities of the current client are also incrementally moved.

Dead reckoning then comes into play and determines if the virtual position exceeds the actual position and sends an EntityUpdate packet automatically if needed.

The entire need for a sole coordinator to exist in the system is due to the following component of the rendering logic: spawning pickups. Note that power-ups (pickups) should be spawned and propagated throughout the entire system only once, and this happens once every 30 seconds on the average. Thus one client should be deemed the coordinator and handle spawning the pickups. The server cannot spawn pickups because it has no concept or understanding of any game object. Once a power-up is spawned assuming the current client is the coordinator, the entity is added to EntityList, and the entity is propagated to the server and then throughout the entire system by means of an EntityCreate packet.

Consequently, any entities that have just moved out of the bounds of the game board are removed in the TrimCanvas() method. This is done to prevent any additional actions from being performed on them such as moving them or testing for collisions against them and thus is done to save processing time. Lastly, collision detection handles any collided entities appropriately by lowering their health or removing their entities completely.

GunFire()

This method is called within the main render() method per frame and handles bullet entity creation locally and remotely. A couple of criteria exist before a bullet is created and sent careening across the game board. Any keys corresponding to one of the three guns are checked if they are pressed down. The gun must be fully cooled down based on its pre-determined cool-down rate. Finally, the level of the gun determines the configuration of bullets generated, speed, and damage. Bullets inherit the damage of the gun that they are shot out of, and their speed and direction as well as the position of the player that shot them are all encapsulated in the bullet's ray object. Recall that the level of the gun is the same level of the player and that level increases or decreases dynamically based on the kills over deaths the current player has garnered. The damage of the bullet directly affects the color of the bullet—a lower wavelength color (red) implies lesser damage, while a higher wavelength (purple) denotes greater damage. Finally all bullets are collected into a list and sent via one big CreateEntityList packet. The reasoning behind sending them in one big list over sending multiple CreateEntity packets is because it was found that the bullets were not all sent at exactly the same time so a slight lag occurred between bullet entity packet receptions thus causing games on remote clients to emit bullets in disarray: for example, the arc shot would shoot for instance 4 bullets out of the corresponding player (white circle), but two of the bullets would be emitted a split second before the other and the true arc form of the bullet splaying would be fragmented. A positive side effect of packing them together was the space saved on duplicate TCP headers that would need to be generated for each packet.

CollisionDetection()

Every frame, collision detection is tested between every pair of entities in the game. Because are all objects are conveniently circles, testing collisions between such objects can be done quickly by simply checking if the distance between their centers is smaller than the sum of their radii. Some preprocessing is done in order to filter down the set of objects tested for collisions to save processing time. For example, collisions are only tested between every player entity and every other non-player entity. Furthermore, the list of non-player entities tested against each player can be further whittled down by the fact that players cannot hit themselves with their own bullets.

The moment a collision is detected between a player and a non-owned bullet, the bullet is removed from the game. No remote update is needed because game states by all clients should be synchronized: If all clients remove the same bullet entity due to the same collision logic employed, the game state remains in sync. However, it is important that health of players be truly synchronized and thus no damage calculation and health loss is performed by clients of wounded players if the player is not owned by the client (was created by the client). Instead, to force the fact that health synchronization is of utmost importance, remote clients simply wait for the player entity update to be sent from the owning client. Because any health changes that occur represent a change in state, an entity update must be sent by the owning client to inform all remote clients and synchronize game state.

Previous implementations of collision detection tried to reduce the inherent $O(n^2)$ collisions tested by exploiting the fact that the system was a distributed system. If clients only tested collisions between all objects and their unique player (white circle) the calculations would be reduced to $O(n)$. If a non-owned bullet hit his/her player, the client would send both an EntityDelete packet containing the collided bullet and an EntityUpdate packet to update clients on the status of the player health. Unfortunately, because of lag inherent in a distributed system, bullet deletions were not immediately performed by all remote clients the moment a bullet collided with a player. Instead it seemed as if bullets would travel through the player the time it took for the owning client to send the EntityDelete packet for the remote client to finally remove the bullet from the game board.

If a player dies as a result of the last collision, the player is resurrected (health restored and position randomly changed) by the owning client and its updated status propagated throughout the system. Because of the way health is updated as explained above, the killer of the player that just died doesn't know that he in fact killed him, and even after it receives an entity update it remains hard to tell. Furthermore, once the client owning the killer player levels himself up as a result of the kill, he must send an update to all other clients that his level has increased by one. However, the two steps can be combined into one process by simply embedding information in the player that just died by recording the player who killed him. The resurrected player is then sent within a PlayerResurrected packet, and on reception remote clients retrieve the killer entity and increase its level by 1. In this way the game state stays sync on all clients and the number of rounds of message passing needed to increase the level of the killer on all clients is reduced from 2 to 1.

Lastly, if a player entity collides with an immobile pickup entity (power-up), the pickup is removed immediately by all clients, the boon is applied immediately in the case of the owning client, and the state of the player is propagated throughout the system.

Entity

The entity object is an abstract class that all other classes derive from. These include in totality: Player, abstract Bullet, CommonBullet, SmallBullet, abstract Pickup, InvisibilityPickup, InvincibilityPickup, StaminaPickup, BombPickup, LevelPickup, HealthPickup. One very important concept in the entire game is to establish a means of entity equality that spans multiple clients. Thus, relying on merely memory pointer equality is not adequate because remote clients do not share the same memory. Every time an entity is created/instantiated a static wide variable within the entity class called ID is incremented and assigned to the current entity. This value can be thought of as the unique identifier distinguishing all entities from each other locally. Moreover, a ClientID field is also populated with the current client's ID as assigned by the server on the initial connection. Thus these two fields concatenated together provide a means to distinguish all entities from each other across the expanse of the distributed system.

In addition to the ID and ClientID fields, the entity object contains other fields of interest that are serialized. These include Ray and EntityColor. Ray is used for movement and movement extrapolation for dead reckoning, while EntityColor is used to color the entity Recall that when a player obtains a bomb pickup, his little white circle turns black; recall further that all bullets are colored on the fly according to their damage.

The last important facet of the entity object is the fact that all entities contain a Silverlight control which specifies the shape, default color, and size of the object as it appears on the game board (Silverlight canvas control). Because of the way Silverlight controls are designed, any such control cannot and should not be serialized due to both their complexity, the inability to specify which fields should be serialized, and the waste of space to remotely send them since the control structure does not change per instance of the entity that it is connected to (ucPlayer.xaml control does not change its structure depending on the specific Player entity that it is associated with); because of this the control was dynamically created on the UI thread the moment deserialization of the corresponding entity occurred. As a result of this separation of logic between the entity and its actual Silverlight control, when moving the entity, the underlying Canvas.Left and Canvas.Top properties of the control were actually responsible for positioning the corresponding control within the game canvas.

Player

The player entity object inherits from the entity and thus has all the fields and functionality of the base class. The player's associated Silverlight control is constructed as a little white circle. Important serialized fields existing in this class are various fields used by the pickup logic: IsInvincible, IsHyper, IsInvisible; also various fields are used by leveling methods: Kills, Deaths, and Level. The Health field denotes the health of the player from 0 to 1.0. Finally the Armor field is a statistic included within Player to mitigate damage upon receiving a bullet wound. Health loss is calculated as follows: Health = Health - Bullet.Damage * (1-Armor). Armor is increased automatically per level based on an exponential decay function: Armor = a^{-level}. Armor is always a value between 0 and 1.

Another important conceptual design issue faced was the fact that StoryBoards, or sets of Silverlight animations, could not be serialized/deserialized because they were again a Silverlight construct. Thus, as discussed above, they could not and should not be serialized and they are manually created anew after each deserialization process. Recall that animations are used for rotating both pickups and health globes around the player. One important issue stemmed from the fact that each time the animation was created anew (must be given the current health of the player to calculate the number of visible health globes) after deserializing an entity, the animation restarted reset. Thus, the original replaced entity’s animation position was retrieved and the new animation had to seek to that position. In this way the rotating health globes appear to rotate with one seamless continuous motion between multiple updates.

Bullet

Bullet is an abstract class inherited by the other concrete classes CommonBullet and SmallBullet. It contains all the functionality as inherited by its parent class Entity while containing fields of its own specific to its own unique function. Its only serialized field of interest is the Damage field which is inherited from the gun the moment it is shot. This field is used to determine the amount of heath loss is incurred to a player upon colliding with the bullet. Within the constructor of the class, a method is called to set the active color of the bullet based on the damage it inflicts.

Ray

The Ray class encompasses both the Point and Vector classes which collectively determine the position, direction, and velocity of the corresponding entity. Recall that a ray is common to all entities in the system, including players, bullets, and pickups. The ray is used for not only movement of entities per frame but it is pivotally used by dead reckoning to extrapolate movement on virtual entities. In fact in reality there is no such concept of a virtual entity in this system, but a virtual ray, or the copy of the actual ray from which the virtual originated. The moment the position as contained within the virtual ray differs from the ray of the actual entity by the dead reckoning threshold (which is set to a pixel size of 1), the entire entity is remotely propagated via an EntityUpdate packet. Magnitude in this case is the velocity or number of pixels travelled per second by the containing entity. Note that the vector determines both direction and magnitude, and the length of the vector determines the magnitude. Thus if the velocity of the vector is 5 pixels per second, the vector is 5 pixels long. The vector and ray classes contain methods in order to move their internal positions based on the vector’s direction and magnitude as well as helper methods to calculate unit vectors, perform vector operations such as addition and multiplication as well as scalar operations. Position is determined by the essence of a parametric line: Position = Position + t * Vector. T in this case is exactly Delta as passed in from the game engine. Recall Delta is the time between the current frame and the last frame. Once the position is changed as a result of calling Ray.Move(Delta), to actually update the position of the corresponding entity control, the control’s fields Canvas.Left and Canvas.Top are set appropriately to the x and y components of the position variable.

One design issue originated from the style of creating a vector. When instantiating a vector for instance, because the size of the board is subject to change and pixels are not constantly set, it is more intuitive to specify or instantiate a vector's velocity in terms of an amount of board-widths per second. Indeed this was the way it was implemented in practice. In fact if one wanted to specify that the sniper bullet travelled at half the board diagonal per second, the velocity object was instantiated with the parameter of .5.

Interpolation

Bezier cubic spline interpolation is implemented in Space Marines. Recall the need for smoothing within dead reckoning to smooth transitions between old entities and newly updated entities sent by clients owning the entities. Bezier cubic spline interpolation presents one of the most realistic, space appropriate, and efficient smoothing algorithms. This is due to the fact that "they account for the starting position/velocity and ending position/velocity" of the object (8). More rudimentary algorithms for smoothing include linear interpolation, which only takes into consideration the starting and ending positions of the objects. More advanced quadratic interpolation schemes take into account the starting and ending position as well as acceleration values, although Space Marines employs no such acceleration (8).

Bezier cubic spline interpolation, the method of smoothing utilized in Space Marines, is now detailed. The interpolation scheme requires 4 control points (C1-C4) where the first and the fourth specify the starting and target positions, respectively. The second and third specify the influence points that movement is swayed toward. This concept is similar to the way a moving body is affected by the mass of a body and consequential gravitational pull. Assume the old entity is R, while the updated entity (where the old entity should be) is S. Now, a time interval T is predetermined (in our case we found a value of .2 seconds to suffice) and specifies the length of time interpolation is used in order to transition from the old state to the new state. It is quite useless to set the destination point (or C4) or the target of the transition to S.Position because after T seconds, S would have moved. Thus, the true destination point S' is used, or the extrapolated position where S will be after T seconds based on S.Vector and S.Position. Now C1 = R.Position and C4 = S'.Position. C2 and C3 are calculated based on the direction that the entities are moving towards as captured by their vectors. C3 is actually where S' would have been in the past and is thus determined from the negation of its vector. Once the control points are found, the parametric equation for calculating positions over various t values are used to smoothly interpolate (when t=0 the position is at C1, and when t=1 the current position is at C4) (8).

It was found that calculating C2 and C3 as was advised in (8) by setting C2 to precisely where R.Position would be in 1 second as determined by R.Vector and by setting C3 to precisely where S'.Position *was* 1 second in the past as determined by S'.Vector was error prone. What if the total time it took for traversing from C1 to C4 as specified by T was much less than 1 second? C2 and C3 would so vastly overpower the movement that it appeared to be much less realistic than was desired. Instead, the total distance between C1 and C4 is calculated, and C2 and C3 are calculated as before from the vectors, but this time care is taken to not vastly overpower the movement. A predetermined influence rate from 0-0.5 is established as a game constant. With this influence rate the distances of C2 from C1 and C3 from C4 is precisely the influence rate multiplied by the length between C1 and C4. The actual positions are still determined from the vector's directional component, but the magnitude or length of the vector is ignored.

The last vital implementation detail is how interpolation should be handled should a new update arrive while interpolation is currently in progress. In order to handle this, the current position is used as the new C1'. To calculate the vector, leveraging the Bezier cubic spline mechanics, the vector direction at that specific position C1' is calculated. The magnitude of the vector is the same magnitude between the old C1 and C2 to calculate the new C2'. C3' and C4' are calculated the same way as before: by extrapolating the position of the new packet by T seconds.

Chat

The chat mechanism facilitates communication between different users and clients in the system. After the user presses the enter button, all attached key handler events are severed or suspended temporarily to allow the user to type an arbitrarily long message without unintended side effects such as player movement or bullet fire. Once a sequence of text is entered, the text is trimmed for excess leading and trailing whitespace. The message, if non-empty, is outputted locally in the chat box located on the bottom of the game board. Furthermore, the message is propagated remotely by means of a LogMessage packet, which recall should contain only a single string as its contents. One caveat was the fact that because the contents contained only a single string, embedded within the string should include the originating client who sent the message. Thus, to handle all cases, a simple "[Client #]" or "[Server]" was prepended to the beginning of any messages sent via LogMessage. On the receiving end, a client harvests this information via regular expressions, nicely formats the message along with the originating client, and inserts it into the chat box.

Conclusion

When Space Marines was conceived, its intension was to provide a healthy conduit for users to engage in hearty entertainment. It allowed users to control a player entity which moved smoothly around the game surface, while firing various weapons to destroy enemy players, level up, form unannounced alliances, and refine esoteric strategies. All this was accomplished in an internet browser to reach a much greater audience. Added features over the base shooter mechanics were implemented to augment gameplay. Much of the game processing was offloaded to the powerful graphics processing unit (GPU) so that gameplay was notably smoother and effects such as animations and shadows could be utilized. A chat mechanism was created to give users the ability to communicate with each other. Moreover, pleasant high-fidelity background pictures formed the backdrop to a resizable game board created in such a way that the game was tailored to any device with any form factor.

While ostensibly Space Marines' goal was to engage and entertain users in a riveting shooter game, its truer goal remained to explore the intricacies of distributed system design along with all its inherent problems. These include the most prevalent problem of communication delay along with the inability to directly share memory, thus requiring a robust system of message passing. A robust system was indeed implemented including both the utilization of a properly designed packet object, leveraging socket and TCP protocol mechanics, all implemented in a scalable framework via a hybrid peer-to-peer environment allowing for any number of clients to connect to a variable amount of servers. Game updates and likewise the notion of global game state was maintained by adherence to a design methodology known as dead reckoning (1). This system stipulated that virtual copies should be maintained by any owning entities in order to simulate the remote view of that entity across the system. Should any conspicuous difference occur between the actual and virtual versions, the actual entity is propagated throughout the system, thus preserving game state while at the same time ensuring that message passing traffic was minimized and only performed when absolute needed. Finally, implementation details along with any issues faced during development was addressed in order to aid readers in grasping the reasoning behind how and why such design facets were implemented. Various advanced topics were discussed including the how Bezier cubic spline interpolation was employed to aid in smoothing between packet-updates.

The game is certainly not complete in terms of its potential. Future work might include exploring the use of UDP multicast in order to quickly and indirectly broadcast packets to all members in the group. Because indirect communication is used, peers need not contain a list of the address of all users in the system and further the server may be eliminated allowing for a truly scalable design. Indeed dead reckoning was used in the first place due to its inherent peer-to-peer nature. However, such a system would require additional design overhead in terms of ensuring channel free errors such as packet reordering or drops. Further migration to WPF would prevent interaction with the browser however opening up the possibility of incorporating advanced pixel-shaders to apply neon laser-like effects to all entities in the game. Furthermore, additional guns could be explored, such as the mine gun that lays immobile but ephemeral mines at player locations. An intuitive scheme to distinguish owned bullets from enemy ones might prove useful to gameplay. More polished sprites could replace the rather dull circles currently in use. Perhaps the game can be expanded into a full breath-taking three-dimensional environment.

Bibliography

1. **Aronson, Jesse.** Dead Reckoning: Latency Hiding for Networked Games. *Gamasutra.* [Online] September 19, 1997. [Cited: September 14, 2011.] http://www.gamasutra.com/view/feature/3230/dead_reckoning_latency_hiding_for_.php.

2. **Dollimore, Jean, Kindberg, Tim and Coulouris, George.** *Distributed Systems: Concepts and Design.* s.l. : Addison Wesley, 2011.

3. **ECMA International.** C# Language Specification. *ECMA International.* [Online] December 11, 2011. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf.

4. **Microsoft.** DataContractSerializer Class. *MSDN.* [Online] 2011. http://msdn.microsoft.com/en-us/library/system.runtime.serialization.datacontractserializer.aspx.

5. **Reinhold, Lasse.** QuickLZ. *QuickLZ.* [Online] January 2011. http://www.quicklz.com/.

6. **Garg, Sachin.** QuickLZ, for Really Fast Lossless Compression. *Data Compression News Blog.* [Online] October 10, 2006. http://www.c10n.info/archives/453.

7. **Microsoft.** Socket.NoDelay. *MSDN.* [Online] 2011. http://msdn.microsoft.com/en-us/library/system.net.sockets.socket.nodelay.aspx.

8. **Van Caldwell, Nicholas.** Defeating Lag With Cubic Splines. *Gamedev.net.* [Online] February 14, 2000. http://www.gamedev.net/page/resources/_/technical/multiplayer-and-network-programming/defeating-lag-with-cubic-splines-r914.

9. **MacDonald, Matthew.** *Pro Silverlight 4 in C#.* s.l. : Apress, 2010.

10. **Anderson, Chris.** *Pro Business Applications with Silverlight 4.* s.l. : Apress, 2010.