Distributed Spatial Database for use with 3D Graphics Engines

DAVID A. HEITBRINK, SAM K. MAKKI, DEMETRIOS KAZAKOS
Department of Electrical Engineering & Computer Science
University of Toledo
Toledo, Ohio
U.S.A

Abstract: - Many modern applications employ a spatial database system to perform management of the large amount of spatial or non linear data. These systems are employed in both real-time and non-real time applications, such as GPS/GIS systems, video games, and scientific databases. Most of these applications are three dimensional, but some can be extended to higher dimensions. The goal is to distribute the responsibilities of a real time three dimensional game server over multiple servers. By distributing the game server we can split the responsibilities of a single server among many servers. This will allow for larger world spaces, with more entities. An entity could be any dynamic object in the three dimension world space. The goal is to make it possible for a series of desktop computers with broadband service to be able to host massive on-line games.

Key-words: - real-time, graphics, database, spatial, server, client

1 Introduction

The spatial databases are suitable storage for many modern 3D applications, since traditional relational databases are not suitable for these applications. Due to that fact that the data in these applications are non-linear and we can not simply store them in a sorted list. This is because distances in multidimensional datasets are relative to positions. In addition a good number of these applications are real time as well, such as simulators, 3D video games, and certain GPS applications and many scientific databases.

Most multidimensional storage systems rely on trying to partition the universe of discourse into sub-nodes, and organize it into a tree based structure. The tree structure employed by these systems presents a natural point at which to divide the system database for distribution. The tree structure also allows a relatively quick selection of the related data from the portion of data, based on their locations. Besides by assigning branches of the tree structures to different servers we can distribute the system in a fairly seamless manner. This also presents us with the possibility of preserving the hierarchal nature of our dataset over multiple servers.

Furthermore the tree based structures of spatial databases allow for the efficient selection and culling of unneeded data for a given area in 3D world space. The primary use is for frustum culling, where only those objects that the user can see are returned by the database, and then only those objects are drawn. Also the data should be returned in partial Z order, in other words the objects are sorted by the depth at which they would appear in a scene. Without this sort of system it would be impossible to support large world spaces, as a vast majority of the programs time would be spent drawing things that the user cannot see, and the cost of Z-ordering would be vastly more expensive. This is also employed on the server side to cull updates to entities the client would not be interested, in other words entities that the client cannot see, or hear. This is slightly different for the server as it is not only eliminates those entities that the client will most likely not able to see in a certain time frame, as the client predicts the movement of entities between updates from the server.

Another related use of spatial database is for an on-line game server. Spatial databases are also employed on the servers to cull updates to entities the client would not be interested, in other words entities that the client cannot see, or hear. This is slightly different for the server as in addition to eliminating those entities the client will most likely not be visible in a certain time frame. The client predicts the movement of entities between updates from the server, so just because something is not immediately visible to the client, they may see the object based on prediction.

The two popular examples of these tree structures are the R+ and Binary Spatial Partitions (BSP) trees. The R+ tree is based on minimum bounding boxes, whereas BSP tree is based on arbitrarily subdividing space. However the BSP tree is generally considered simpler in most respects than R+ tree, since BSP is unbalanced tree while R+ is a balanced tree.

In this paper we propose to develop a system where a three dimensional spatial database can be distributed among multiple computers. This system should be applicable across other special database applications such as warehouse management and GPS systems. These systems tend to employ more complicated tree structures such as R+ tree, and have ability to perform spatial joins and other database operations. Applying a distributed system allows to reduce the per server bandwidth requirements for network usage, and reduces the computational requirements for each server. This should also be extendable to peer-to-peer systems.

The remainder of this section will deal with the background information needed to understand the environment of a multi-player game server. Section 2, will deal with issues that need to be dealt with distributing a real time game server. Section 3 will outline the proposed network topology for the distributed database and current problems. Section 4 will outline a functional specification for the distributed system, and outline the network communications that will be used in the system. Section 5 concludes a short summary of our findings and discusses different applications that using spatial database systems.

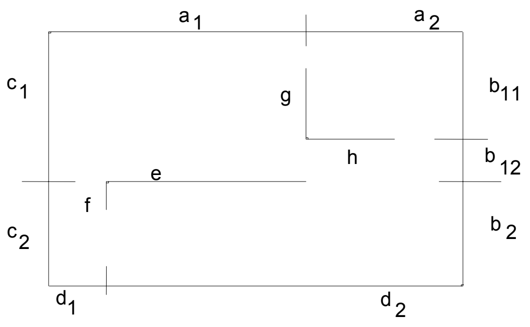

Figure 1

1.1 Binary Spatial Partitioning Tree

A Binary Spatial Partitioning (BSP) tree is a method of spatial partitioning, and the most popular tree type for the spatial database systems for use with real-time 3D graphics [Fuchs]. It is useful as it provides an efficient method for determining interest and data can be returned in a partial z order at the same time, for almost no additional overhead. It is a binary tree structure that splits space into nodes based on if an object is in front of, or behind some arbitrary line, or plane, depending on dimensionality. All polygons that are intersected by this plane are then subdivided. All the polygons that lie along the plane are added to the node at that level. All nodes contain data that can be seen as concave in nature, so that when the objects at that node are drawn the order the objects are drawn in will not effect z-ordering. Figure 1shows a room that a BSP tree has been constructed for. Note that no one polygon can be spanned in two separated nodes. In the case where a polygon spans multiple nodes the polygon is divided. For example in Figure 1, line "b" is divided into multiple sections; otherwise the line would be contained in three nodes.

BSP trees have been used in a number of applications, such as in many modern video games. As an example Quake video game uses BSP trees not only on the client end to properly render a scene, but also on the server end to determine what the client would be interested in. This is also a good example of an application of a spatial database system for a number of reasons. The first it is simple, most games use a BSP trees, or a Quad or Octal trees. These are all non-balanced trees and are relatively easy to implement and understand. Secondly there have been many different implementations of these systems, and are readily available to evaluate. The third reason is that there are a few open-source game engines available which will allow to sort of a "look under the hood" and to see how these are implemented, and what can we change to distribute such a system. Communications between server and client consist of unreliable communication through a channel. The bandwidth of the channel is limited as not to overload with more updates then it can process, and as not to let the client use more then its fair share of bandwidth. The server then is connected to a master server. This master server just keeps a list of game servers that are running, and clients query this server in order to find out what servers are available to connect to. The server provides a "heartbeat" to the master server to indicate that it is still running, and/or update its stats. Often the client queries the individual server to find out what is its status, i.e. who is playing, what are the scores, what is the current map, what is the next map and so on.

The Quake II game server is a good example of an Internet game server. The game runs in a continuous loop. The server first receives all the packets that are available and pre-processes them. The second thing the server does is to calculate all the ping times of all the connected clients, then sends the ping times to the clients. After that the server runs the "game frame". The first step is to increment the current frame number and the current time. Next in this process is to update server controlled entities, such as monster with artificial intelligence, and projectiles. The next step is to loop through all the players, running any event the client has sent to the server, such as a fire, or move forward. The next step is to determine the player's interest, i.e. what sound can the client hear, and what can he see. Next the client message is sent. This consists of adding the data that is to be multi-casted with the data that is specific to the client, i.e. the data that the client is interested in. The next step is to send a heartbeat to the master server if needed, then reset the current state for next frame, i.e. reset various flags and so on.

The data that is sent from client to server consist of chat message to players, events such as move forward, fire, and jump. Also service messages are sent such as a disconnect message, change request to change teams, or administrative commands, the server will check the client privilege level for administration commands. The server then sends the client's information about what is in its area of interest, plus global information. The client is sent only what it is interested in, and only data on entities. Entities are defined as every thing that is essentially moving. Also this data is sent as delta updates, meaning no update will be given for something that has not changed in the previous frame and the difference of the previous position is given as an update. This makes it vitally important to keep a consistent time, since delta updates are time dependent. Also updates to the client's current state are given, i.e. where the client is, what is his status and so forth. All of this data is packaged into a single datagram and sent to the client.

The data being communicated between a client and server can be broken down into two groups for real-time non-critical applications, such as many simulators or video games. The first is high priority time dependent data. This data is absolutely critical that it is handled quickly. This mostly consists of events generated by the client, i.e. the client fires a gun, client moves forward. Movement and firing are critical in most games. They have important role in determining if something was hit or not, besides if movement is not smooth, the game will appear shaky and degrade the user perceived speed of the connection. However there are several non-time critical events. Messages in a game are a key example, rarely a user notices or even care if there are messages, it take up to 500ms to be posted, the only thing that is important here is that, the messages be kept in order. Also collisions and damage determination while not as time insensitive as messages, they can be done in a less timely manner with out degrading user performance.

The communication between the client and server is as follows: 1) The client sends the server a heartbeat, this is a continuously signal, and the purpose of this is, to provide the server a signal that the client still exist. 2) The client sends events to the server. The events are messages such as, move commands, fire commands, and so on. 3) The server then sends the client at regular intervals the updates to the current game state. The server however does not send the whole state to the client; rather it sends, what is determined as being interesting to the client. These updates should be sent every 100ms at least. The state provides information on the location and status of every entity. The status of an entity could include things such as the current pose of an object, or any other attributes it may have.

Another example of communication for client server communication is for GIS/GPS systems. These tend to be more complicated in many ways than either games or simulator type applications. A typical web based GIS system can be divided into five components, 1) the client, 2) the proxy server, 3) the database, 4) information server, and 5) the sockets. The client in GIS system will issue queries to the proxy server. For example the client may query a five mile by five mile block of Manhattan. The client will be expecting to receive a satellite photo of the five mile by five mile region. The client then would have a number of functions to process the image, such as an ability to perform a fly-over a section, or perform some feature extraction on the image.

The queries are then received by the proxy server. The proxy server stands a gateway between the server, and one or a number of databases that may need to be queried. The server once it determines which database should be queried; it constructs the query in format required by the database server, and sends the query to the database server.

The database server is where the data is actually stored. This tends to be based on an R based trees. The server receives simple queries and returns the results.

The client can also connect to an information server. This server is responsible for processing systematic queries such as: "All hospitals in 20 mile radius of 2456 Dorr St. Toledo OH" or a nearest neighbor query such as "Closest hospital to 2456 Dorr St Toledo OH". This type of server is also often multi-threaded as to process multiple requests at one time.

The differences between the client server relations in these two systems are substantial. In the real-time environment the server usually processes queries in an iterative manner. For each frame the server receives updates from the client, then processes the said request from the client. The server can often assume what the client wants every frame, and so are rarely any real queries in this type of system. Also queries processed in a non-real time system can be processed in a non-synchronous manner, the order in which they are processed is not that important in this type of application. While as the real time application the order in which things are processed can have great effect on environment.

2. Issues

Beyond distributing our spatial database we most worry about several specific issues dealing with the fact that we are operating in a real time environment. These primarily deal with the fact that we must maintain a somewhat consistent state among all the clients/servers, and we must also be concerned with the fact that we have limited bandwidth available to each server.

2.1 Consistency

A major issue with real time distributed systems is consistency. The main problem is that we are not really dealing with data in real time but is delayed over the network connections at varying time, anywhere from 2ms to 500ms. It is important that we keep time over the system consistent, that every system must have the same time. With the clients sending data that is delayed as much as 500ms, or is lost in transmission all together.

What we must do when we have an inconsistent state is to roll back the system to a consistent state. Also we must have the system state consistent as well. When a user character dies, every one must know he is dead, and all the servers must agree that, the character is dead. If this is not the case, then the user may be spawned at a new location, and then still be alive at its old location.

2.2 Latency

The primary goal of the systems is to provide a system with low latency; it has been found that ideally the latency of the client connection to server should not exceed 100 ms, although latency of up to 350 ms can be considered tolerable. High latency can be caused by two main factors that we can at least control. First, is that demand on bandwidth needed to communicate to all clients is greater then the amount of bandwidth we have available. Second cause is that the server cannot handle the computational load required to make a timely response to the client's request.

2.3 Interest

Interest is the data that is pertinent to a client, and only that data. If another player is hidden behind a wall, and out of what would be visual to the end client user, the client does not care, and does not need to have the data about the player that is obscured from the view of the client. The well known Software product Quaketm was able to reduce the amount of data that would needed to be communicated to the client by about 70% by determining what data the client would be interested in. In a distributed system where each server has an area that it is responsible for, the client now has to determine which servers are interested in its generated data. With out interest determination the distributed system would simply be just multiplying the amount of data that is being communicated.

2.4 Bandwidth

It is hoped that by distributing the regions responsibility to specific servers; we can distribute the bandwidth usage of a number of servers, therefore reducing the bandwidth needs and computational requirements for each server. The bandwidth requirement for a multi player game grows at the square number of joining players to the game. This is because we have to send all data about every client to every other client. With interest management this can be reduced considerably, but still the bandwidth requirements for a game server can be quite high, and grow at vary high rates.

3. Proposed network topology

The initial proposed network is fully connected. Each server will have a portion of the BSP tree that it is responsible for. This will complicate client as it will have to determine what servers will contain data that it is interested in, and conversely determine what servers would be interested in its events.

The BSP tree will be subdivided splitting branches of the tree amongst the servers, with each server being responsible for all children beneath the root node. The root node of the server will be the node that is considered its root, and it will be responsible for all children nodes of its root. Also servers may have servers under its root node; it will not be responsible for the children nodes of this server, and its root. The server may have some responsibility of its children servers. Also no server will branch across two limbs of the BSP tree, and will not be responsible for any nodes that are not children of its root node. One question that remains now is; how do the servers communicate among themselves? Should the hierarchal model used to assign the areas of responsibility from the BSP tree to be preserved, or should the network be completely connected? At this point a completely interconnected network seems to be the better alternative as it would be faster to eliminate the hops. As a message would have to make to just only one hop, however for administration purposes using a hierarchal model may be ideal.

One problem that persists is that, events may bridge over multiple servers. For example a client fire a bullet that traverses multiple areas of responsibility. The client would be responsible for sending the bullet entity to each server to inform them, that the bullet crosses its area of responsibility. However this now presents us with a problem of collision detection. If a bullet fired across multiple areas of responsibility and an event is sent to each server, and there is a collision in servers. Then how and where is it determined which collision occurred first? One possible solution is to use a mirror server. The mirror server will contain the entire game/simulation state. The mirror server will determine which collision occurred first, and then report back to the servers and client or clients involved which hit occurred first. This will of course take a bit longer to determine if a hit occurred but in general damage determination is considered a lower time priority, i.e. lag is more tolerated here. The mirror server will also be responsible for maintaining some degree of fault tolerance. If a server shuts down it will be responsible for re-assigning its area of responsibility, and sending the state to the effected servers for their new regions.

3.1 Boundaries

Some entitles create splash effects, for example an explosion. Although the explosion may occur in one region it may affect other entities on other servers. Another issue is when an entity may be bisected by boundaries of two regions of control.

Possible Solutions: Have a mirror server make final decision on what happens to said entity. If on both servers at the same time an advent occurred involving the entity, the mirror server would "brake the tie", and determine which advent has priority if they are in conflict. Another way would be to arbitrarily assign control to one of the servers, or classify different events with a priority level higher priority wins, same level of priority randomly pick winner.

Another possible solution is to have a shared virtual leaf, which the server maintains for boundary leafs. The leaf would be shared between the two servers, and they would only pass the control of the entities in the boundary node. Each server would then maintain a copy of this virtual node

3.2 Determining Tree Distribution

Another issue is how to distribute the BSP tree among the distributed game servers. Issues include deterring how much area is given out per branch assigned to each server, how much of an area is used, i.e. how many users will be querying it. Also servers should be distributed so that their neighbors have the smallest latency to them. A few algorithms will have to be developed that take into account latency between servers, popularity of regions, i.e. how often it is accessed.

Clustering - Clustering should be able to determine areas with a high density of clients. By determining these clusters and making sure that these clusters are not split between two servers we can reduce the amount of hand-over that has to be done between servers.

The two popular techniques with strong promise for applications to the need for clustering in real-time spatial databases are fuzzy c-means clustering [2], and BIRCH [3]. Fuzzy c-means clustering is a modification of c-means clustering with the addition of fuzzy clusters in which members can have partial membership to multiple clusters. BIRCH is a hierarchical approach to clustering. It provides a method for culling data points that will not likely have a major effect on clustering, and data points that far enough away from clusters centroids that they are extremely unlikely to effect clustering.

Another work on clustering that specifically deals with spatial data was developed at the Universities of Vechta, and Bonn for use in their GeoToolKit. This clustering technique used the Euclidean distance between objects and centroids of clusters in an iterative manner, for determining clusters.

Two specific things that have not been dealt with the previous works, which have large implications on the proposed database system, are time and collisions. We are dealing with temporal data which change rapidly with time. It is in our interest not only to determine where clusters of entities occur but also to try to determine where they may move to, and determine clusters that are not relevant to our intended purpose. Another issue to deal with, is the distance. Euclidean distance is usually used in clustering. The problem with using Euclidean distance without respect to collision detection is that two entities that may be close together using Euclidean distance can be rather far apart from each other when the shortest path between the two entities is found. Also calculating the shortest distance between entities and centroids may be too costly to perform in a relatively short time frame. A more cost effective approximation may be needed.

3.3 Network Map

One issue is, how to let every one know what the current topology of the network is, what servers are mapped to what leaves of the BSP tree, and if the server topology is dynamic what is the most effective way of directing change in topology and communicating with all parties.

Solutions: One solution is maintaining a "blackboard" solution on the master server that is available to every one. Another way is to have servers send change of address messages to clients. A third method is to have the master game server send messages directly to all parties, requiring them to maintain an open connection to the server. Also a protocol will have to be developed to describe the topology.

3.4 Convergence

What happens if all of the clients converge at one area in our world space? This looks to be an inherit drawback of the proposed. It may be necessary construct a world space and/or game rules that would lessen the likelihood of this happening.

Solutions: One solution is to have cache available on other server with low load; the cache server would be given a copy of the game state, and a list of clients to send packets to. The draw back of this is that it adds an extra step to the end client, this can double the time needed to communicate with the client. Another solution would be to further partition the region with highest load.

3.5 Regions of Interest

One problem is how we can communicate with the servers that a client is interested in their data. The client's location might not be in the given servers regions of control. The server would need to know the location of the client in order to properly do a traversal of it BSP tree to determine what type of data it has that a client could be interested in.

Solutions: The client could multi-cast its position to all the servers which would be interested in, or simply all servers. Another suggestion is to have the server that contains the location of the client to multi-cast the client's position to all other servers, or to just the server it would be interested in. Another suggestion is to have the clients make requests to all the servers.

3.6 State Consistency

One major problem is trying to keep the state among the servers consistent they should all be on the same frame. It should be remembered that there are two different times; the wall time and the simulation time, although both the real time and simulation time must be similar. Another problem is that everything is delayed by a certain amount of latency due to latency involved with network communication.

Solutions: One solution is to have the master server maintain a heartbeat informing all the servers and clients the current frame. Multiple network protocols exist to address these problems; these should be looked in great detail.

3.7 Multi-Server Collision Detection

One problem is determining collision detection for events that span over multiple servers. For instance if we have a bullet that is fired, that crosses multiple regions of control. The problem is the first collision for the bullet (assuming the bullet does not penetrate what ever it hits), is the only collision that counts, and other collision would be voided.

Solutions: One solution is to have a centralized mirror server make the decision, although this would be add some time delay due to the additional communication required. A second solution is to have the server relay messages stating that there was no collision that took place, and propagating the no collision messages until a collision occurred. A second method is instead of having the bullet event multicast across all the servers; simply pass the bullet from server to another server.

3.8 Fault Tolerance

One problem with maintaining multi-server environments is that when more servers are added, this increases the probability of one of the servers failing. In addition game servers are often taken down for multiple reasons. The problem is we must keep a back up of the server state, and reassign various responsibilities.

Solutions: There are multiple ways of address this issue. One way is to maintain a single backup server that keeps a back up of the entire game state. Another issue is what happens when the game master server goes down. The various game servers must all agree that the master game server is down, then pick one of the game servers to run a new master game server on it, it also may be necessary to reduce the load on that game server.

3.9 Delta Updates

Delta Updates are used to a great deal in multiplayer games in order to reduce the amount of communications required. The problem is that the current frame number and timing information most now be maintained over multiple servers to properly form the delta update. This will most likely not pose a major problem as this has already been solved for the client to server communications, and server to server should just be an extension of this.

4. Functional Specification

The functional specification for the program calls for the development of three processes, client, game server, and master server. The client is responsible for interacting with the user and displaying the game state to the user. The game server is responsible for updating the client with the current game state, and receiving from the client its updates. The master server is responsible for setting up the distributed environment. It is responsible for determining what game server is responsible for what region of the world space, and many other tasked that need to be centralized in nature.

4.1 Client

The client is the end user program. The client program is responsible for displaying graphics, taking in input, playing sound and so on. The client receives updated to the game state. The client also must figure out what servers to send its updates to. The client contains the following processes:

Listener: listens for incoming packets from servers, 1 for each server

Broadcaster: Sends constructed packets out to respective server

Packet preprocessor: This is an important process. It figures out what data various servers would be interested in, constructs the data packets from this data, and then sends the pre-constructed packets to the Broadcaster.

Packet post-processor: This process processes all the received packets, and takes necessary actions based on that, this could be from updating the topology map, updating the game state, and so on.

Game Engine: This runs the game; it is in a loop of processing data than updating the games state, handles user input, and displays the game. For the most part this process is the game.

4.2 Server

The purpose of this program is to enable multi-player game play. Its job is to take in user input, processes a game state, then update the users on this game state. These servers are intended to be distributed where they are only responsible for a portion of the game state, more specifically they are responsible for a portion of a BSP tree, that which is used to store the entirety of the whole game state. This program will responsible for all A.I., and computer controlled entities. Also the server will have to pass entities on to other servers when a computer controlled entity passes on to another region of control. The following are the main components:
Listeners: listens for incoming packets from various sources, other server clients, and master game server.
Broadcaster: Sends the packets out to various sources mostly to clients in its area of influence, but also to other servers, and master server.
Post-processor: This processes the incoming packets, including making decisions based on multi-server events.
Game Loop: This runs the games, it's in a loop updating the game state, determining the user interest, and constructing packets to be sent.

4.3 Master Game Server

The Master Game server is responsible for managing the entirety of the game environment, and may reside on one of the servers but there will be only one per game. The responsibility of this program is two insure consistency across game, and excepting new servers and clients. The main components are:
Server manager: This program accepts new servers. This process also tracks server load, and possibly directs caching systems. This would involve instructing servers where they can cache some data. This function may be distributed among the servers. This program also decides when repartitioning the world space is required. Then it sends a request to the topology manager to have this done.

Topology manager: This process is responsible for partitioning the BSP tree, and assigning the areas of responsibility.

Client manager: This process is responsible for handling the new clients, client disconnect events, and assigning client spawns, i.e. when a client begins a new life, i.e. deciding where the new client will begin his new life.

4.4 Network Communications

The Quake II engine already contains much of the communication facilities that will be needed for the actual IP communication. Work will have to be done on actually creating the application layer of the system, in knowing where to send data, and which data to send. Also where the facilities do not already exist; systems for packaging certain data to be sent, than reassembled must be created. The actual communications will occur using both reliable and unreliable communication systems over an IP network. Certain data must be sent through reliable means, other data can be sent using data grams. The advantage of using data grams is that it uses less bandwidth as no acknowledge has to be sent to the sender. The downside is that it is unreliable, data will get lost. For most data in the game environment this is not a real big problem. An entity may appear out of place to a client for a few frames but this is acceptable if the occurrence is rare.

There will be three types of communications. The first will be client to game server, this will not differ a whole lot than what is seen in a single server environment and will change a little. The second will be game server to game server. The server will have to communicate to other servers the locations of clients on the server, that the client would be interested in the data on its server. For instance, if a client is on server A, but can see part of the region located on server B, server A would forward the location of the given client to Server B. This can be with unreliable communication, using the delta update system already employed by the quake engine.

Entities will have to be passed between servers these entities will have to be sent with reliable communication, as when they are passed to the new server, that server is responsible for that entity. If that entity is lost in the communication process the entity will either dropped from the game, or become a zombie.

Another level of communication will be from server to master server. This will include instructions as to new spawn points (where a new client is initially placed, or after a new life) for clients, new clients in the game, and new information on the server zone of responsibility for the BSP tree. This information will be kept in a two dimensional array with each node number, and its corresponding server that is interested in the data. A node labeling system must be implemented for diversion of the BSP tree. There are many systems preexisting systems available for this the labeling of tree structures. All communication on this level will have to be done in a reliable manner. The master server will also communicate and relay certain administration commands to all servers. This will not change from what is already implemented in Quake II engine; the master server will just relay such commands to all game servers.

The final type of communication that will have to happen is from client to master server. This type of communication will be reliable for the most part. Chat messages will be communicated, and notices that the map is changing, and the score will be communicated between client, and master server. The only thing that is different from the current single server configuration is that the server will now have to communicate a lookup table for the node-server pairs. This will be updated, and will have to happen in reliable communication.

4.5 Tracking of Objects

One issue when dealing with objects that move in space is, how do we track individual objects. This brakes down into two separate issues, one where we have some control of the objects in the environment, and another where we can not control the objects.

The solution to tracking objects that we have some control over, or they transmit heartbeats is simple enough. With this we must come up with an ID numbering scheme. We can either have a master server that assigns and manages these addresses, or assign address ranges to servers. In a simulation or a game as master server can assign the ID, and assign the initial location of the object.

For scenarios where we do not control the objects, but they can send a unique ID, we either assign the ID to the object by either assigning a ID from a range of ID numbers that a given server is assigned, or by requesting a new ID number from a master server, and retransmit the ID to the object to transmit.

The second scenario for tracking the objects is when we can only observe the location of objects and not have any actual communications between the objects, and servers. This leads to a difficult problem, how do we track these objects in time as they move. The first thing we must do is to assign the objects ID numbers when we first encounter them, this varies little from when we assign ID numbers to objects that can send heartbeats to our servers.

The hard part is how to differentiate the objects after a time interval in which the objects have moved. In order to differentiate between the objects we need a time resolution which is high enough than the distance an object can move in the given time interval. For this we must be able to assume that the object closest to another object in a previous time frame is the same object. We can further refine this by taking into account from the maximum acceleration of an object, its maximum rate of deceleration, and any other given physical attributes that would describe the motion of the object. We would have to solve a series of equations to try to match objects between frames. To find out which object could be in a given location from its previous location in a given time interval. This of course leads to some problems, what happens if the set of equations we solve to describe where an object is in the previous frame to the current frame produces multiple possible answers. In such cases we may not be able to track the individual objects and we would have to choose from the list of possible solutions. We may be able to increase the possibility of a correct identification if we can also use some attributes of objects not related to the movement of the objects such as size color or any other attributes. Another way of possible increase to the probability of a correct identification is to apply knowledge base on usual movement patterns of the objects. An example of this is, if we have two objects that could possible fit for two objects in the next frame. If one object is in the same spot as one in the previous frame, and the other is elsewhere close by, we could possibly assume that the two objects in the same spot in the two different frames are the same object.

A continued problem is how do we pass the tracking of objects between servers. This comes as an issue in that we need to know the previous state of an object in order to try to determine its assigned ID number. A solution for this is for servers to share a boarder area and use this data to match up ID numbers and objects. An additional problem when we have the possibility of multiple matches to objects and ID numbers. The problem is when the two servers make different ID/object matches. With this we must come up with a method for the two servers to agree on the possible matches.

5 Conclusions

The focus of this paper was development of a distributed spatial database system for a three dimensional game server. A video game such as Quake II was chosen for this application due to the relative simplicity of the database system that it uses, and availability of open source. There are a number of applications of spatial databases; such applications include GPS systems, satellite imaging systems, CAD/CAM, VLSI, visual perception, robotics, and autonomous navigation. The bulk of the effort of this system is to track moving entities in a world space, and passing these real-time objects. Although most spatial database applications do not have real-time objects, they do have the passing of objects from different areas of control. The system developed to partition dataset among different servers in one that can be applied to different spatial database systems, and many of the same concerns exist across most of the different spatial database systems.

References:

[1] H. Fuchs, Z.M. Kedem, B. Naylor, On visible surface generation by a priori tree structures, Comput. Graph. 14(3) 124-133, 1980.

[2] D.E. Gustafson and W.C. Kessel., Fuzzy clustering with a fuzzy covariance matrix. Proc. IEEE CDC, pp 761-766, 1979.

[3] T. Zhang, R. Ramarkrishan, M. Livny BIRCH: An Efficient Data Clustering Method for Vary Large Databases, 1996.

[4] M. Bruenig, A.B. Cremers, W.Muller, J.Siebeck New Methods forTopological Clustering and Spatial Access in Object-Oriented 3D Databases, 1996.

[5] Ralf hartmut Guting, An Introduction to spatial Database systems, VLDB Journal, 3, 357-399, 1994.

[6] D. Pullar, M. Egenhofer, Towards formal definitions of topological relations among spatial objects, proceddings of the third international Symposium on Spatial Data Handling, Sydney, pp 230-236, 1988.

[7] J. Goldstein, R. Ramakrishnan, U.Shaft, and J. B. Yu., Processing Queries by Linear Constraints, in proceedings of the PODS conference, pp 257-267, 1997.

[8] O. Gunter and E. Wong, A Dual approach to detect polyhedral Intersections in Arbitrary Dimensions. BIT, 31(1), pp 3-14, 1991.