GameOD: An Internet Based Game-On-Demand Framework

Frederick W.B. Li[†] Rynson W.H. Lau[‡] Danny Kilis[!]

[†] Department of Computing, The Hong Kong Polytechnic University, Hong Kong
[‡] Department of CEIT, City University of Hong Kong, Hong Kong
[!] PCCW Business eSolutions (HK) Ltd., Hong Kong

ABSTRACT

Multiplayer online 3D games are becoming very popular in recent years. However, existing games require the complete game content to be installed prior to game playing. Since the content is usually large in size, it may be difficult to run these games on a PDA or other handheld devices. It also pushes game companies to distribute their games as CDROMs/DVDROMs rather than online downloading. On the other hand, due to network latency, players may perceive discrepant status of some dynamic game objects. In this paper, we present a *game-on-demand (GameOD)* framework to distribute game content progressively in an on-demand manner. It allows critical contents to be available at the players' machines in a timely fashion. We present a simple distributed synchronization method to allow concurrent players to synchronize their perceived game status. Finally, we show some performance results of the proposed framework.

Categories and Subject Descriptors: I.3.2 [**Computer Graphics**]: Distributed / network graphics
General Terms: Design, Performance.
Keywords: Multiplayer online games, distributed virtual environments, on-demand replication, distributed synchronization.

1. INTRODUCTION

To play a new online game, a player is usually required to install the complete game content, i.e., perform a full replication of the game environment, in the local machine. If the size of the game content is large, it will limit both the type of players' machines and the distribution channels for such games. Hence, handheld or mobile devices, which usually have limited storage space, may not be able to run them. In addition, since downloading the games through the Internet could be time consuming even for players equipped with a broadband network connection, most of the game companies would still prefer to distribute the games in CDROMs/DVDROMs. This seriously restricts the distribution of these games. It also discourages the casual players who just want to try out different games to see which one to buy. On the other hand, there may be a lot of potential players in the developing countries who are more remote from this distribution channel and can only access the Internet with a 56K modem.

To address these problems, we present here a *game-on-demand (GameOD)* framework to support on-demand distribution of game content over the Internet, in which only a very small portion of the game environment is downloaded initially and the rest of the content is progressively downloaded in an on-demand and timely fashion. Although some distributed virtual environment (DVE) systems [8, 10, 4] support similar feature, they are based on either some pre-defined spatial relationship or area of interest (AOI) [8, 18] information of the game objects, which usually require excessive information from the server and hence may not be optimized for use in the Internet environment. In addition, we also present a simple distributed synchronization method to allow concurrent game players to perceived synchronized status of the dynamic game objects, providing more accurate information to the game players for making correct decisions. Our main contributions include:

- a unified data structure for progressive transmission of different types of modeling primitives.
- a content prioritizing scheme to allow visually important content to be delivered with a higher priority, and to adjust the amount of geometry information of the game objects to be sent according to the network bandwidth of the client connection.
- a simple distributed synchronization scheme for minimizing the state discrepancy among the concurrent players to improve the interactivity of the game.

Experiments show that our framework requires a very short initial downloading time before the player may start playing, regardless of whether the player is using a 56K modem or a broadband connection. The remaining game content is then progressively transmitted to the player's machine during game playing. The rest of this paper is organized as follows. Section 2 gives a survey on related work. Section 3 presents the GameOD framework. Section 4 presents our two-level content management scheme. Section 5 presents our prioritized content delivery method. Section 6 describes our distributed synchronization scheme. Section 7 shows some experimental results and an action game that we have developed. Finally, section 8 briefly concludes the paper.

2. RELATED WORK

2.1 On-Demand Content Distribution

Most existing multiplayer online games, such as Quake III Arena [20] and Diablo II [6], have a large game scene and a lot of game objects, resulting in a large game content size. Hence, game companies often store both the game program and content in a CDROM/DVDROM for players to purchase. Although some games like The Sims Online Play Test [24] may be downloaded via the Internet, as its data size is over 1GB, only players equipped with a broadband network connection may obtain the game in this way. Other players still need to purchase the game disc from the local stores. Even after getting the game kit, the players still need to spend a significant amount of time to install it.

We note that researchers working in *distributed virtual environments* (DVE) [22] are facing similar research problems. We have also noted that although a VE may be very large, a user often only visits a small region of it. To save memory space and downloading time, we may transmit the local geometry information to the user machine based on the location of the user inside the DVE. This approach is adopted by several existing DVE systems, which can be generally classified as *region-based* and *interest-based*.

Systems including DIVE [11], CALVIN[15], Spline [25] and VIRTUS[21] have adopted the region-based approach. They divide the whole VE into a number of pre-defined regions. A user may request to connect to any region by downloading the full content of the region before the user starts to work within the region. However, as the content of a region may still be very large in data size, the system may pause for downloading the region content whenever the user switches from one region to another. The interest-based approach uses the area of interest (AOI) [8, 18] to dynamically determine object visibility. Only the scene content and updates within the AOI of the user need to be transmitted to the user machine. Systems adopted this approach include NPSNET [8], MASSIVE [10], and NetEffect [4]. Although this approach may reduce the amount of game content needed to be downloaded during runtime, it may still suffer from some transmission problems. First, within the AOI of a user, the importance of individual objects, i.e., the order of objects, is usually not considered in model transmission. Hence, a user may receive less important objects before the critical ones. While this may be acceptable for walkthrough applications, it is certainly not for the action type of games. Second, traditionally, the geometry information of individual objects is transmitted as a whole to the client. However, a large model can easily use up the available network bandwidth for a considerable period of time, affecting the user interactivity.

2.2 Synchronization

Due to the relatively high network latency of the Internet, game players likely perceive significant delay in receiving updated state information from the server or other players. For example, in Final Fantasy XI [9], a player usually receives position updates of other players with almost a second delay. To reduce the effect of network latency, some restrictions are imposed on the game itself. First, players can only attack enemy objects, but not each other. Second, the enemy objects are designed to move very little while they are under attack by a player. Such game rules significantly limits the type of games that can be developed.

Consistency control in distributed applications has been explored in different disciplines, including distributed systems [14], database systems [2] and collaborative editing systems [23]. In these applications, the time gap between two consecutive commands is usually very long as compared with the network latency. Hence, these applications can work well by ensuring only that states updates are presented to the relevant users in a correct order, without considering the exact moment that the state updates are presented to the users. However, this condition could hardly satisfy the requirements of multiplayer online games, where the time gap between two consecutive commands can be very small and network latency becomes significant. To address the network latency problem of multiplayer games, Mauve et al. [19] adopt a local-lag mechanism. When a player (sender) issues an event, the event will be sent and presented to the receivers immediately, but not to the sender itself until after the local-lag period is expired. However, with this method, each sender is limited to have only a single local-lag value. Hence, this method can only be used to synchronize between two players, the sender and the receiver. It cannot be used to maintain consistency among a number of players.

3. THE GAMEOD FRAMEWORK

The GameOD framework adopts the client-server architecture to support progressive game content transmission. The server(s) manages the game environment and handles the interactions among the game clients (or players) and the game objects. It also determines and schedules the required game contents at appropriate details for transmission to the game clients. Figure 1 shows the architecture of GameOD.

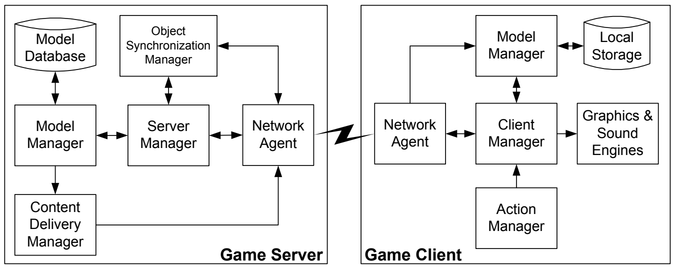

Figure 1: The GameOD architecture.

In summary, the server module has 6 components. The *Server Manager* coordinates all components at the server. It processes all interactions among the objects and sends updates to the clients. It also communicates with the model manager on the status of each client for content delivery. The *Model Manager* determines the required game objects and the appropriate visual qualities for transmission to the game clients. (Refer to section 4 for detail.) The *Model Database* stores the complete game content. All the object models and texture images are already in a format for progressive transmission. The *Content Delivery Manager* manages the references of the selected items for prioritized delivery. (Refer to section 5 for detail.) The *Object Synchronization Manager* is responsible for synchronizing all relevant clients on the states of dynamic objects (including avatars). (Refer to section 6 for detail.) Finally, the *Network Agent* handles all communications between the server and the clients, including game content delivery and status updates.

The client module has 6 components. The *Client Manager* coordinates all components at the client. It processes updates from both the server and the local input devices. The *Model Manager* maintains the object models received from the server for local rendering and performs cache management. The *Local Storage* is the game accessible storage space in the client for the model manager to store the objects received from the server. The *Action Manager* reads control commands from the input devices. It extrapolates the movements of dynamic objects to minimize the amount of update messages sent over the network, and synchronizes dynamic objects to minimize their discrepancy in the client and in the server. (Refer to section 6 for detail.) The *Graphics and Sound Engines* are responsible for rendering visual output and generating the game audio in a time critical manner. Finally, the *Network Agent* handles all communications with the server, including receiving the object models, texture images and game updates from the server. It also sends updates and actions of the client to the server.

4. GAME CONTENT MANAGEMENT

The GameOD framework manages the game content in two levels, *scene-level* and *model-level*. For scene-level, the game contents are managed in a way that the server may efficiently identify appropriate game contents for delivery. For model-level, different types of modeling primitives, including rigid models, deformable models and texture images, are arranged into an unified data structure to support progressive transmission and multiresolution rendering. For transmission, the server determines the optimal geometric details of each game object for delivery based on some dynamic conditions, such as the visual importance of the object to a player or the available network bandwidth of the client's network connection. For rendering, a client may select appropriate resolution of each object to display with any real-time multiresolution rendering method.

4.1 Scene-Level Content Management

To determine the visible objects to a player at a particular moment, we have adopted the viewer/object scope idea from [3], which can be considered as a restricted form of the Aura/Nimbus model [10]. We associate each object in the game with an *object scope* $\bigcirc_o$, which is a circular region defining how far the object can be seen. We also associate each game player with a *viewer scope* $\bigcirc_v$. However, unlike [3], which defines the viewer scope as a single circular region, it is defined here as a circular region with multiple parts as will be described in section 5. An object is visible to a player only if its $\bigcirc_o$ overlaps with $\bigcirc_v$ of the player. Here, we classify the game objects into *static objects* and *dynamic objects*. A dynamic object may change shape or move around in the game environment, e.g., a player or an autonomous object, while a static object may only be passively interacted by a dynamic object, e.g., a tree or a stone.

During run-time, as a player moves around in a game environment W, we need to continuously check which objects are visible to the player, i.e., which objects need to be transmitted to the player. To speedup this process, we partition W regularly into N_C rectangular cells, i.e., $W = \{C_1, C_2, \ldots, C_{N_C}\}$. Each cell C_n may contain a list of references to a number of *shadow objects*, $\{O_{C_n,1}, O_{C_n,2}, \ldots, O_{C_n,m}\}$, where $1 \leq n \leq N_C$, $0 \leq m \leq N_O$, and N_O is the total number of objects in W. These shadow objects are objects which object scopes overlap with C_n. To set up a game, for each game object O, we add a reference of O to all the cells that $\bigcirc_o$ of O overlaps. During run-time, when we need to determine the potentially visible objects to a player, we only need to check all the cells that $\bigcirc_v$ of the player overlaps. The set of potentially visible objects to the player is the disjoint union of all the shadow objects found in all the cells that $\bigcirc_v$ overlaps. For the dynamic objects that may move around in the environment, we would dynamically update the shadow object lists of all the affected cells.

4.2 Model-Level Content Management

Our game engine supports three types of modeling primitives, rigid models, deformable models and texture images. All these primitives are formatted to support progressive transmission and multiresolution rendering.

4.2.1 Definition of Game Objects

Each game maintains a geometry database D at the server, storing all the geometry models M and all the texture images T used in the game, i.e., $D = \{O, M, T\}$. O is the set of game objects defined as $O = \{O_1, O_2, \ldots, O_{|O|}\}$. Each object O_i in O is composed of a number of models M_{O_i} (where $M_{O_i} \subset M$), a number of texture images T_{O_i} (where $T_{O_i} \subset T$), an animation method A_{O_i}, an object scope $\bigcirc_{o,O_i}$, and a viewer scope $\bigcirc_{v,O_i}$, i.e., $O_i = \{M_{O_i}, T_{O_i}, A_{O_i}, \bigcirc_{o,O_i}, \bigcirc_{v,O_i}\}$. In general, an object is composed of one or more models. For a dynamic object, each moving part is often represented as a separate model. Each model may be associated with zero or more texture images. A_{O_i} is a small program module indicating how O_i should move and react to the environment. Note that since each geometry model, texture image or animation program may be used by more than one object, we consider each of them as a transmission primitive. When an object is to be transmitted to the player, we would check each of its primitives to see which need to be transmitted and which are already transmitted to ensure that only one copy of each primitive is transmitted.

4.2.2 Unification of Modeling Primitives

To simplify the programming interface, we organize all the geometry models and texture images in a unified data structure. We refer to each of these transmission primitives as a modeling unit U, which may be represented as a *base record* U^0 followed by an ordered list P of *progressive records* $\{p_1, p_2, \ldots, p_{|P|}\}$. The base record U^0 contains the minimal information for reconstructing U at its lowest resolution. By applying each of the progressive records p_n in P to U^0 using function $\Omega(u, p)$, a list of approximations of U, $\{U^0, U^1, U^2, \ldots, U^{|U|}\}$, are obtained, where $U^n = \Omega(U^{n-1}, p_n)$. Each U^n in the list improves the quality of U^{n-1} by a small amount and the final $U^{|P|}$ in the list is implicitly equivalent to U, i.e., $U = U^{|P|}$. To transmit a modeling unit U to a client, we first transmit the base record U^0 to help alert the player the existence of the object and its rough shape. Whenever it is necessary, further progressive records may be transmitted progressively to the client to enhance the visual quality of the object. The three types of modeling primitives that we support are described as follows:

Rigid Models: They are triangular models encoded as *progressive meshes* [12]. To encode a model U, a list of Ω^{-1} is applied to U recursively to remove the geometric details from the model until U^0, the most coarsest approximation of U, is obtained. Hence, a list of progressive records $\{p_1, p_2, \ldots, p_{|P|}\}$ are generated. Ω^{-1} is defined as $(U^{n-1}, p_n) = \Omega^{-1}(U^n)$. It is an inverse function of Ω.

Deformable Models: This is a particularly interesting type of objects that we support, which shapes may change in time. They are classified as dynamic objects and represented using NURBS. We apply our efficient technique [17] for updating and rendering these objects. Each deformable object may be composed of one or more NURBS models and optionally some rigid models. Hence, each NURBS model is considered as a modeling primitive. In [17], a polygon model and a set of deformation coefficients are precomputed to represent each NURBS model. This information is maintained in a quad-tree hierarchy. To support progressive transmission, we organize the quad-tree in an ordered list based on the z-ordering indexing scheme [1]. The list begins with a base record U^0, which consists of a surface definition, a node presence list and a root record to represent the deformable NURBS model at the minimal resolution. Subsequent records $\{p_1, p_2, \ldots, p_{|P|}\}$ of the list store information for refining and updating the precomputed polygon model. A Ω function is defined to combine the information stored in each p_n to U^{n-1} to form a slightly refined model U^n.

Texture Images: A straightforward way to handle texture images is to encode them in progressive JPEG format for transmission. However, it requires an expensive inverse discrete cosine transform (DCT) to extract texels from the compressed images. Instead, we extend the color distribution method [13] to support progressive transmission of compressed texture images. The method encodes a texture image by dividing it into uniform grids of texel blocks. Each texel block is encoded by two sets of 32-bit information, a 32-bit representative color in RGBA format and a set of 2-bit color indices for 4×4 texels. For each texel block, a local palette is set up, containing the representative color of the block and the representative colors of three adjacent texel blocks. The color of a texel within the block is then approximated by a 2-bit color index to this palette. This method may compress a normal texture image to about $\frac{1}{8}$ of its size. To allow progressive transmission, we arrange the texture data in the form of an ordered list. The list begins with a base image U^0, which is formed by extracting one bit from each channel of the RGBA of the representative color of each texel block. Each subsequent record p_n of P is formed by alternatively extracting 4-bit information sequentially from the texel color index for each texel block and 4-bit RGBA color value of the representative color. A Ω function is defined to attach the information stored in each p_n to U^{n-1} to recover the details of the texture image.

5. PRIORITIZED CONTENT DELIVERY

5.1 Object Transmission Priority

As mentioned in section 4.1, we make use of the *object scope* and *viewer scope* to identify interested objects to a game player. As shown in figure 2(a), the viewer scope contains three regions, $Q1$, $Q2$ and $Q3$. $Q1$ is the *visible region*. All objects within it are considered as visible objects to the player and have the highest priority for transmission. $Q2$ is the *potential visible region*, composed of $Q2_a$ and $Q2_b$. All objects within it are not immediately visible to the player but will become visible if the player simply moves forward or turns its head around. Hence, these objects should be transmitted to the client machine, once all the objects in $Q1$ are transmitted. $Q3$ is the *prefetching region*. Objects within it will take sometime before they are visible to the player. Hence, we would prefetch them to the client machine if extra network bandwidth is available. To speedup the process of selecting objects for transmission, we maintain an object queue for each of these regions. Hence, there are three queues, $Queue1$, $Queue2$ and $Queue3$ for regions $Q1$, $Q2$ and $Q3$, respectively. All the objects in the queue are sorted according to their transmission priority.

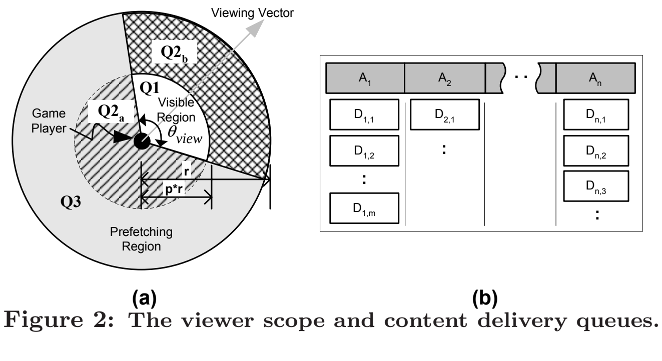

Figure 2: The viewer scope and content delivery queues.

To efficiently determine the region that an object is fallen into, for each player, we set up 4 viewing segments in its local coordinate system as shown in figure 3. We label the segment in which the player's viewing vector is located as $S1$. The segments having the same sign as the x- and y-coordinates of $S1$ are labeled as $S2$ and $S3$, respectively. The segment having opposite signs to both X and Y of $S1$ is labeled as $S4$. The four possible configurations of the viewing segments are shown in figure 3(a) to (d). We refer to the left and right boundary of the visible region as V_L and V_R, respectively, as shown in figure 3(e).

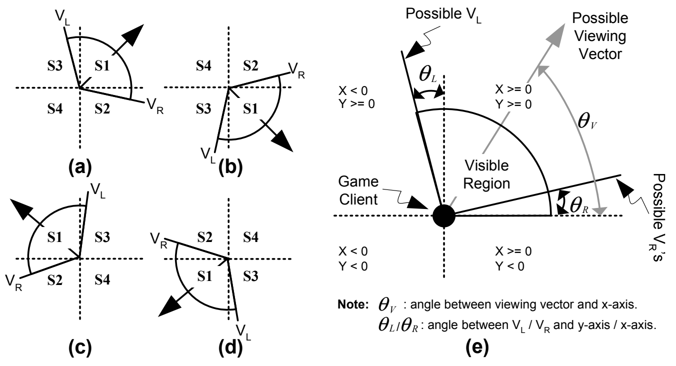

Figure 3: Four viewing segments of a player.

For each object G, we calculate its relative coordinate (x_g, y_g) from the player. Refer to figure 3(e). The signs of x_g and y_g indicate the viewing segment that the object lies. The distance of G from the player is computed as: $D_g = \sqrt{x_g{}^2 + y_g{}^2}$. Assuming that the radius of the viewer scope of the player is r and the radius of the object scope of G is r_g, G is deliverable only if $D_g \leq r + r_g$. If G is deliverable, we put a reference of G into a player specific content delivery queue based on its visual importance to the player. The priorities of the three queues for transmission are: $Queue1 > Queue2 > Queue3$. Objects placed in a queue with lower priority may not be delivered to the player, unless all queues with higher priorities are emptied. Within a queue, objects are first sorted by their angular distances A_n from the player's viewing vector, and then sorted by their distances $D_{n,m}$ from the player. As shown in figure 2(b), we quantize A_n into a number of vertical subqueues and $D_{n,m}$ into a number of vertical bins. When retrieving objects for delivery, we select from the left-most vertical subqueue (with lowest A_n) to the right-most subqueue (with highest A_n) one by one. For a selected subqueue, we transmit objects starting from the top bin. This process is iterated until all subqueues are emptied or the queues need to be updated.

Table 1 summarizes the rules to determine the region that an object G belongs. Column 1 shows the viewing segment that G is located. Column 2 shows the rules to determine the appropriate region for G. Column 3 shows how to compute the angular distance of G from the player's viewing vector. To combine objects from regions $Q2_a$ and $Q2_b$ into a single queue, we need to adjust the angular distances A and euclid distances D of these objects before inserting them into $Queue2$. For objects from $Q2_a$, we set $A = A - \frac{1}{2}\theta_{view}$. This is to normalize its angular distance value to start from zero. For objects from $Q2_b$, we set $D = D - (p \cdot r)$. This is to normalize its distance value to start from zero.

As shown in table 1, when considering the angular distance factor in identifying the order of object delivery, we only need to perform a simple angle comparison instead of evaluating a dot-product for each object against the viewing vector of each player. More specifically, for each player, θ_V, θ_L and θ_R are constants if the player has not changed its viewing direction. To determine the region for an object G, we only need to evaluate either θ_{gx} or θ_{gy}, which are the angular distances of G from the x-axis or y-axis of the viewing segments of the player, respectively. For instance, $\theta_{gx} = \tan^{-1}(\frac{y_g}{x_g})$ and $\theta_{gy} = \tan^{-1}(\frac{x_g}{y_g})$. In practice, the run-time arctangent evaluation may be avoided. Since we only consider objects that are within the outer circle of the prefetching region, we may pre-compute a set of arctangent values, $tan^{-1}(\frac{x}{y})$, with different combinations of x and y, where $x, y \leq r$. In addition, as we have divided the game environment into a grid of small cells, their representative coordinates may well approximate the positions of the objects that they contain. Hence, we may use these coordinates to compute only a finite set of arctangent values to avoid the computation of a continuous series of arctangent values.

Viewing Segment	Rules for Determining the Region	Angular Distance

S1	If $\lvert\theta_v - \theta_{gx}\rvert \leq \frac{1}{2}\theta_{view}$, if $D_g \leq (r \cdot p + r_g)$, assign **G** to **Q1**;
	else, assign **G** to **Q2**$_\mathbf{b}$.
	else, assign **G** to **Q3.**
		$\lvert\theta_v - \theta_{gx}\rvert$

S2	If V$_R$ in S1, assign **G** to **Q3**;
	If V$_R$ in S2,
	if $\theta_{gx} \leq \theta_R$, if $D_g \leq (r \cdot p + r_g)$, assign **G** to **Q1**;
	else, assign **G** to **Q2**$_\mathbf{b}$.
	else, if $D_g \leq (r \cdot p + r_g)$, assign **G** to **Q2**$_\mathbf{a}$;
	else, assign **G** to **Q3**.
		$\theta_{gx} + \theta_v$

S3	If V$_L$ in S1, assign **G** to **Q3**;
	If V$_L$ in S3,
	if $\theta_{gy} \leq \theta_L$, if $D_g \leq (r \cdot p + r_g)$, assign **G** to **Q1**;
	else, assign **G** to **Q2**$_\mathbf{b}$.
	else, if $D_g \leq (r \cdot p + r_g)$, assign **G** to **Q2**$_\mathbf{a}$;
	else, assign **G** to **Q3**.
		$\theta_{gy} + (90^\circ - \theta_v)$

S4	if $D_g \leq (r \cdot p + r_g)$, assign **G** to **Q2**$_\mathbf{a}$;
	else, assign **G** to **Q3**.
		If $x_g \leq y_g$, $180^\circ + (\theta_{gx} - \theta_v)$;
		else, $180^\circ - (\theta_{gx} - \theta_v)$

Table 1: Assigning objects to appropriate regions.

5.2 Object Quality Determination

After prioritized the delivery order of the objects, we need to determine the amount of geometric information, i.e., the quality, of each object for progressive transmission. Since each object is composed of some transmission primitives, the server would map an object to the corresponding transmission primitives when it is considered for transmission. To do this, for each player, the server maintains *deliverable lists* for the geometric models and texture images. For each primitive in the lists, we attach two parameters, P_{sent} and P_{opti}, to control its delivery quality. They represent the number of progressive records sent and the preferred optimal number of progressive records, respectively. P_{sent} is updated whenever some part of the primitive is sent. P_{opti} is determined based on some real-time factors as follows:

$$P_{opti} = P_{max} \times (\gamma B + \tau R + \alpha(1 - A) + \beta(1 - D))$$

where P_{max} is the maximum number of progressive records that the primitive has. Parameters A, B, R and D are normalized by their maximum possible quantities and they range from 0 to 1. B and R represent the available network bandwidth and the rendering capacity of a client, respectively. To simplify the computation, we may assume that they are constants through the session. A and D are the run-time angular distance of the object from the player's viewing vector and distance of the object from the player, respectively. For each of the primitives, we assign it with the smallest possible values of A and D based on all visible objects that are using this primitive. This would maximize P_{sent} to fulfill the maximum requirement of those visible objects. Finally, α, β, γ and τ are application dependent weighting scalars, where $\alpha + \beta + \gamma + \tau = 1$. For example, if a game needs to transmit a lot of messages among all clients and the server, the performance of the network connections would have a high impact on the performance of the game. Hence we may use a higher γ to allow network bandwidth to be a dominating factor in determining the object quality.

5.3 Object Transmission

For object transmission, each object in the content delivery queues is retrieved based on its priority as described in section 5.1. If its corresponding transmission primitives exist in the deliverable lists, the next progressive record of each of these primitives is transmitted to the player. We then increment P_{sent}'s of these primitives by 1. If P_{sent} of a primitive reaches its P_{opti}, we remove this primitive from the *deliverable list* and add it to the *sent list*, which stores primitives that are already sent to the client at the optimal quality. In addition, if all primitives of an object have satisfied this condition, we will remove the object from the content delivery queue.

6. DISTRIBUTED SYNCHRONIZATION

To perform synchronization, we regard the server copy of each dynamic object as a *reference simulator* of the object. Regardless of whether the object is a player controlled character (avatar) or an autonomous object, its motion must be synchronized according to its reference simulator. As our framework adopts a client-server architecture, state information from a player typically needs to be collected at the server before propagating to other players. This leads to a double round-trip delay for a remote client to receive a state update. However, having the reference simulator at the server, a system could effectively reduce the network latency among the clients to a single round-trip delay, as a client is now only needed to synchronize with the reference simulators of any dynamic objects stored at the server.

During game playing, a player may control the movement of its own avatar by issuing motion commands with a keyboard or a game pad. Motion vectors can then be computed based on these commands to form the navigation vectors of the avatar. The motion of the avatar immediately after receiving a navigation vector can be derived from a motion predictor. Here, we use the first-order predictor as follows:

$$p_{new} = p + t \times V \qquad (1)$$

where p is the current position, t is the motion timer difference between p and p_{new}, and V is the motion vector of the avatar. Other motion predictors may also be used [7].

6.1 Client-Server Synchronization

Figure 4 shows the interactions between the server and a client in order to synchronize the two motion timers, T_s and T_c, which are maintained at the server and the client, respectively. When the player issues a motion command to drive its avatar to move in certain direction (state A_c), the command is first buffered for a short period, about 50ms in our implementation. All motion commands received during this period are combined and only the resultant motion vector is sent to the server at the end of the buffering period. This buffering period acts as a filtering process to limit the sampling frequency of the input device. When such period expires, T_c is reset to 0. The current position and the velocity of the avatar are sent to the server to update its reference simulator. As soon as the server has received this information, T_s is reset to 0 and the reference simulator begins to run. Due to network latency, the reference simulator is expected to start later than the actual motion of the avatar. While the command button is still pressed (states A to D), the avatar would continue to move at the same velocity until the command button is released (state E). During this period, the position of the avatar is extrapolated using the motion predictor and the client's local T_c.

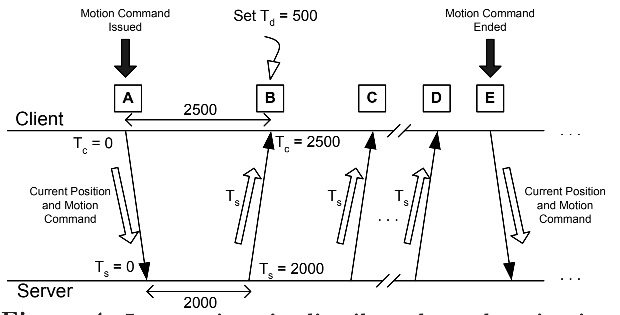

Figure 4: Interactions in distributed synchronization.

To synchronize the motion of the avatar at the client with its reference simulator at the server, the server sends a motion time stamp to the client regularly. (In our implementation, the server does it once per second.) When the client receives the time stamp (state B), it evaluates a time difference $T_d = T_c - T_s$, which is used to approximate the round-trip delay between the client and the server. If we consider the example shown in figure 4, $T_d = 500$ means that the reference simulator has moved 500 time units less than the avatar at the client. Let us assume that the client would update the avatar's position every T_δ (of a second) to render the most updated frame, i.e., T_δ may be considered as the duration between two consecutive frames. Before the client receives the next motion time stamp from the server (states B to C), whenever the client updates the avatar's position, t in the motion predictor is adjusted as follows:

$$t = \begin{cases} T_\delta & \text{if } |T_d| \leq \epsilon \\ \frac{T_\delta}{2} & \text{if } T_d > \epsilon \\ 2 \cdot T_\delta & \text{if } T_d < -\epsilon \end{cases} \tag{2}$$

$$T_d = T_d - t \tag{3}$$

where ϵ is just a very small number. Equation 2 progressively adjusts the motion period of the avatar. This has the effect of modifying the motion timer of the avatar, T_c, to that of the server, T_s, and therefore, synchronizing the two timers. Equation 3 acts as a counter so that the adjustment process will continue until the current motion command is ended, when the player issues a new motion command to the server to initialize a new synchronization cycle, or when the two timers are synchronized, i.e., when $|T_d| \leq \epsilon$.

6.2 Client-Client Synchronization

Our reference simulator approach natively handles the client-client synchronization problem. Given an avatar of player **A**, its reference simulator is again run at the server. The avatar's motion of **A** would be adjusted gradually to synchronize against its reference simulator at the server. If another player **B** needs to interact with **A**, **B** will send a request to the server to gather the motion information of the avatar of **A** to start its own simulator of the avatar locally. Consequently, the gradual synchronization process will be performed in a similar way as presented in section 6.1. Finally, since the avatar's motion at any other client, such as **B**, would be synchronized against the reference simulator at the server, motion of any avatar could ultimately be synchronized among all relevant clients.

7. RESULTS AND DISCUSSIONS

We have implemented the proposed framework in C++, and developed a first person fighting multiplayer online game, *Diminisher*, based on this framework. It allows multiple game players to navigate in a shared game environment to fight with each other and some automated opponents. By connecting to the game server, an initial content package, which contains the geometry information of the objects surrounding the player, is transmitted to the client. After receiving the package, the player may start to play the game. Additional content will then be progressively streamed to the client based on the location of the player in the game.

7.1 Video Demo

A video showing a brief game playing session can be found in [5]; the network bandwidth was set at 1.5Mbps. Figure 5 shows the corresponding rendering frame rate and data transmission rate of the session. Our measurements were started after the client had received the initial content package, decoded it and begun rendering the scene. We observe that there was a high

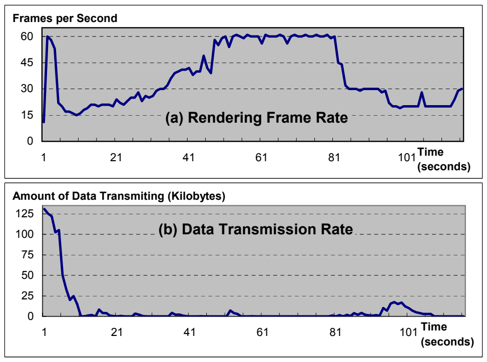

Figure 5: Performance of the game demo.

data transmission rate at the beginning. This is because the initial content package only contains minimal amount of geometry information for the client to render a low visual quality scene. Hence, the server will continue to stream additional game content to the client to improve the visual quality to an optimal value. It is also because the objects are in low visual quality, i.e., containing only a small number of primitives, the frame rate can be much higher. As the visual quality of the visible objects increases, the data transmission rate began to drop. Concurrently, the player turned and faced a scene with a lot of characters and objects, resulting in a significant drop in frame rate. The frame rate rose again as most of the opponents were killed by the player. Finally, as the player moved to the scene with a large number of easter island sculptures, the player received the corresponding geometric information and the frame rate dropped again due to the increase in the number of primitives needed to be rendered. The pulses appeared approximately 15s after the start of the game as shown in figure 5(b) were due to the progressive content transmission. As the player moved around in the game environment, the client received new content progressively from the server.

7.2 Experiment on Game Startup Time

In this experiment, we test the performance of the game prototype under different network connection speeds, ranging from 56Kbps modem speed to 10Mbps LAN speed. We measured the startup time for downloading the initial content package of the game in order for the player to start playing the game. As shown in figure 6, if a player has a broadband network connection, i.e., network speed $\geq$ 1.5Mbps, the initial downloading time is less than 5s. If a game player connects to the game using a 56Kbps modem, the initial downloading time is only about 30*s*. This startup time is considered as acceptable by most players.

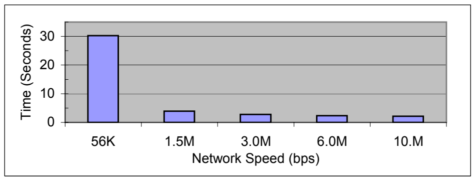

Figure 6: Startup time vs. network connection speeds.

7.3 Experiment on Content Delivery

In this experiment, we test the efficiency of the on-demand content distribution method. During the experiment, a game player navigated in a game environment with a circular path, and looked around freely to visualize interested game objects throughout the navigation. There were about 150 visible game objects located around the player's navigation path. They are all in compressed format with an average size of 124KBytes. We define the *perceived visual quality* as the percentage of the required model data received by the player in terms of the data size. We compare the perceived visual quality by the player during the navigation using different model transmission methods: Method **A** is our method, Method **B** is our method but without progressive transmission, Method **C** is the progressive transmission method but without prioritized content delivery, and Method **D** only transmits the base record of each model to the player.

We performed the experiment with a 56K modem (figure 7(a)) as well as with a 1.5Mbps (figure 7(b)) broadband connection. (However, only 60% of the bandwidth was available for the experiment. The rest was used by the test program for collecting the statistics.) The result of the experiment is shown in figure 7. For methods with progressive transmission, i.e., methods **A** and **C**, the game server would transmit the current visible objects to the player up to their optimal resolutions as described in section 5.2. Hence, a player is said to perceive a 100% visual quality if the player receives all such objects in their optimal resolutions. For methods without progressive

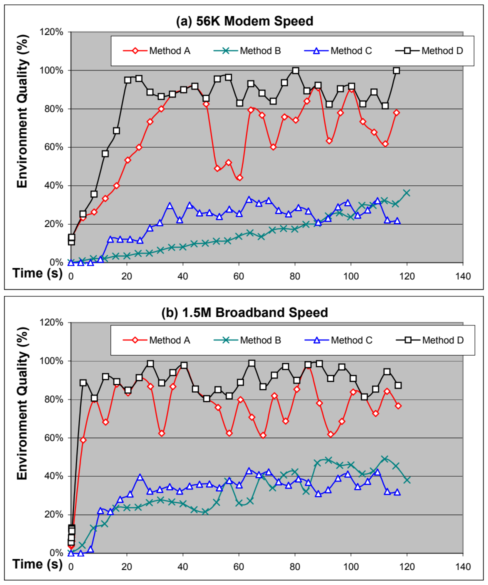

Figure 7: Efficiency of various content delivery methods.

transmission, i.e., method **B**, a player could visualize a game object only if the complete object model is received by the player. Hence, the player could perceive a 100% visual quality only if all current visible objects are completely transmitted to the player. As a reference, we performed an additional test by forcing the server to transmit only the base records of the objects requested (method **D**).

From figure 7, we observe that our method (method **A**) offers a significantly superior perceived visual quality to the players then those of the other methods on both network conditions. When compared to method **C**, our method could provide 20% to 70% higher perceived visual quality. When compared to method **B**, our method may even provide 80% higher perceived visual quality occasionally. The quality difference between our method (method **A**) and method **D** indicates room for application designers to perform further tuning on the value of the optimal perceived visual quality. This provides flexibility for game systems to adapt to the limitations of various resources.

7.4 Experiment on Synchronization

To determine the performance of the proposed distributed synchronization scheme, we have experimented it with three player movement patterns under different network latencies: 0.64ms (for LAN), 10 ms (for Internet - within a city) and 160ms (for Internet - international). They are the *linear path*, the *zig-zag path* and the *circular path*. We have implemented the motion predictor shown in equation 1 of section 6. To measure how well the distributed synchronization scheme works, we compare the error difference between the motion timer in the client with that in the game server. Such difference implicitly indicates the spatial location discrepancy perceived by the two parties. This motion timer is the sole parameter to determine the change in the spatial location of a dynamic object given a motion vector.

Figure 8 shows the performance of our distributed synchronization scheme. For the *linear path*, the player only pressed a command button once to decide the moving direction, and continuously held the button to move in a straight path. From figure 8(a), we can see that at the start of the navigation, the motion timers of the player and of the server were not synchronized and the error was as high as 12% for the one with long network latency. After the timers were synchronized, the error was no more than 2% regardless of the network latency. For the *zig-zag path*, the player pressed different command buttons alternatively to change its navigation direction. From figure 8(b), we can see that there was a sudden increase in error whenever the player pressed the command buttons to change the navigation direction. This was because the change in navigation direction caused the two motion timers out of synchronization. However, this error quickly dropped below 2% as the two timers were synchronized. For the *circular path*, the player pressed different command buttons to move circularly in the game environment. From figure 8(c), we can see that the result was very similar to that in figure 8(b). In summary, our results show that the proposed distributed synchronization scheme can cope with different network latencies well. A more detailed analysis reveals that after a player has pressed a command button to change the moving direction, our method may usually eliminate the discrepancy between the motion of avatar at the client and the reference simulator running at the server before the server sends out the second motion time stamp to the client for synchronization.

Figures 8(d) to (f) show the same set of experiments corresponding to those in figures 8(a) to (c). They use the same motion predictor shown in equation 1 but without using our

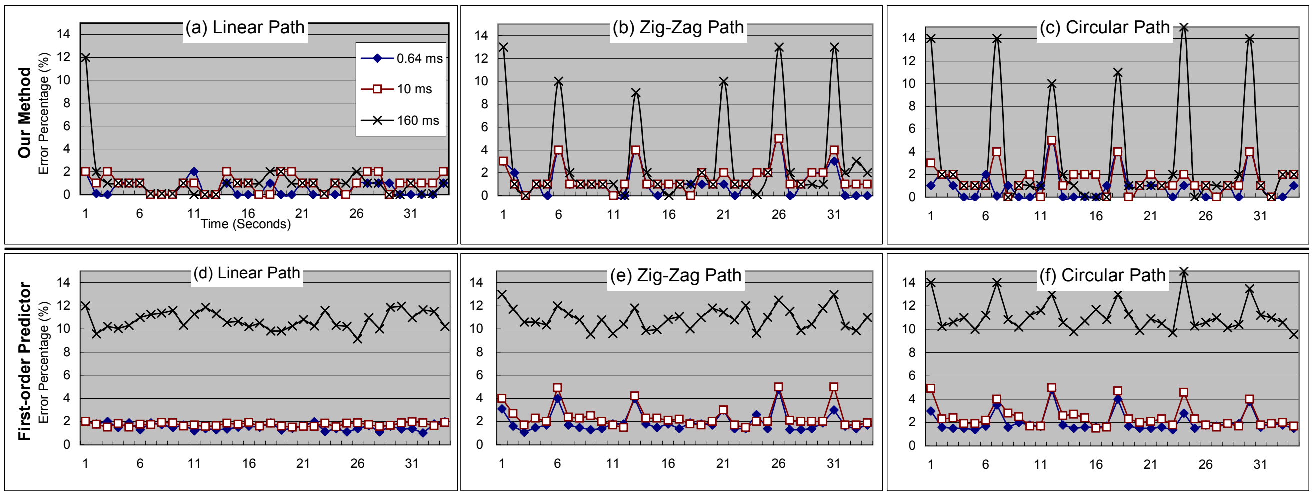

Figure 8: Results of distributed synchronization.

distributed synchronization scheme for motion timer adjustment. We can see that their errors are much higher than those in figure 8(a) to (c). We also note that at all those moments when the player just pressed the command button, the errors of the two sets of diagrams are the same. However, when using our distributed synchronization scheme, the errors were significantly reduced almost immediately after the change of motion was detected. This shows that our synchronization scheme is very effective in resolving the synchronization problem.

Because of the importance of this issue, we have recently started a new project to investigate the problem using a different approach. We are developing a more accurate synchronization technique based on a formal analysis of the problem [16].

8. CONCLUSION

In this paper, we have presented the framework of a multiplayer online game engine based on progressive content streaming. The engine allows the game players to download the game contents progressively, without the need to purchase the game CDROMs or to wait for a long downloading time. Our main contributions of this paper include a two-level content management scheme for organizing the game objects, a content delivery scheme for scheduling game content delivery based on object importance and on network bandwidth, and a distributed synchronization scheme for synchronizing between the motion timers in the client and in the server. We have demonstrated the performances of the game engine and of a game built on the game engine through a number of experiments.

Acknowledgments

The work described in this paper was partially supported by two CERG grants from the Research Grants Council of Hong Kong (RGC Reference No.: CityU 1080/10E and PolyU 5188/04E). We would also like to thank the team members of *iCollaborative Technology* for the implementation work.

9. REFERENCES

[1] L. Balmelli, J. Kovačević, and M. Vetterli. Quadtrees for Embedded Surface Visualization: Constraints and Efficient Data Structures. *Proc. IEEE ICIP*, **2**, pp. 487–491, 1999.

[2] P. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, **13**(2):185–221, 1981.

[3] J. Chim, R. Lau, H. Leong, and A. Si. CyberWalk: A Web-based Distributed Virtual Walkthrough Environment. *IEEE Trans. on Multimedia*, **5**(4):503–515, Dec. 2003.

[4] T. Das, G. Singh, A. Mitchell, P. Kumar, and K. McGhee. NetEffect: A Network Architecture for Large-scale Multi-user Virtual World. *Proc. ACM VRST*, pp. 157–163, 1997.

[5] Demo. www.cs.cityu.edu.hk/∼rynson/projects/ict/shortdemo.mpg.

[6] Diablo II. Available at www.blizzard.com.

[7] DIS Steering Committee. IEEE Standard for Distributed Interactive Simulation - Application Protocols, 1998. IEEE Standard 1278.

[8] J. Falby, M. Zyda, D. Pratt, and R. Mackey. NPSNET: Hierarchical Data Structures for Real-Time Three-Dimensional Visual Simulation. *Computers & Graphics*, **17**(1):65–69, 1993.

[9] Final Fantasy XI. Available at www.playonline.com/ff11/home.

[10] C. Greenhalgh and S. Benford. MASSIVE: A Distributed Virtual Reality System Incorporating Spatial Trading. *Proc. ICDCS*, pp. 27–34, 1995.

[11] O. Hagsand. Interactive Multiuser VEs in the DIVE System. *IEEE Multimedia*, **3**(1):30–39, 1996.

[12] H. Hoppe. Progressive Meshes. *Proc. ACM SIGGRAPH*, pp. 99–108, Aug. 1996.

[13] D. Ivanov and Y. Kuzmin. Color Distribution - A New Approach to Texture Compression. *Proc. Eurographics*, pp. 283–289, 2000.

[14] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, **21**(7):558–565, 1978.

[15] J. Leigh, A. Johnson, C. Vasilakis, and T. DeFanti. Multi-perspective Collaborative Design in Persistent Networked Virtual Environments. *Proc. IEEE VRAIS*, pp. 253–260, 1996.

[16] F. Li, L. Li, and R. Lau. Supporting Continuous Consistency in Multiplayer Online Games. Accepted as short paper in *Proc. ACM Multimedia*, 2004.

[17] F. Li, R. Lau, and M. Green. Interactive Rendering of Deforming NURBS Surfaces. *Proc. Eurographics*, pp. 47–56, Sept. 1997.

[18] M. Macedonia, M. Zyda, D. Pratt, D. Brutzman, and P. Barham. Exploiting Reality with Multicast Groups: A Network Architecture for Large-scale Virtual Environments. *Proc. IEEE VRAIS*, pp. 38–45, 1995.

[19] M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-lag and Timewarp: Providing Consistency for Replicated Continuous Applications. *IEEE Trans. on Multimedia*, **6**(1):47–57, Feb. 2004.

[20] Quake. Available at www.idsoftware.com.

[21] K. Saar. VIRTUS: A Collaborative Multi-User Platform. *Proc. VRML*, pp. 141–152, 1999.

[22] S. Singhal and M. Zyda. *Networked Virtual Environments: Design and Implementation*. Addison Wesley, 1999.

[23] C. Sun and D. Chen. Consistency Maintenance in Real-Time Collaborative Graphics Editing Systems. *ACM Trans. on Computer-Human Interaction*, **9**(1):1–41, 2002.

[24] The Sims Online Play Test. Available at www.ea.com.

[25] R. Waters, D. Anderson et al.. Diamond Park and Spline: A Social Virtual Reality System with 3D Animation, Spoken Interaction, and Runtime Modifiability. *Presence*, **6**(4):461–480, Aug. 1997.