Game-On-Demand: An Online Game Engine based on Geometry Streaming

FREDERICK W. B. LI, Durham University
RYNSON W. H. LAU, City University of Hong Kong
DANNY KILIS, Morgan Stanley Information Technology (Shanghai)
LEWIS W. F. LI, City University of Hong Kong

In recent years, online gaming has become very popular. In contrast to stand-alone games, online games tend to be large-scale and typically support interactions among users. However, due to the high network latency of the Internet, smooth interactions among the users are often difficult. The huge and dynamic geometry data sets also make it difficult for some machines, such as handheld devices, to run those games. These constraints have stimulated some research interests on online gaming, which may be broadly categorized into two areas: *technological support* and *user-perceived visual quality*. Technological support concerns the performance issues while user-perceived visual quality concerns the presentation quality and accuracy of the game. In this article, we propose a game-on-demand engine that addresses both research areas. The engine distributes game content progressively to each client based on the player's location in the game scene. It comprises a two-level content management scheme and a prioritized content delivery scheme to help identify and deliver relevant game content at appropriate quality to each client dynamically. To improve the effectiveness of the prioritized content delivery scheme, it also includes a synchronization scheme to minimize the location discrepancy of avatars (game players). We demonstrate the performance of the proposed engine through numerous experiments.

Categories and Subject Descriptors: I.3.2 [**Computer Graphics**]: Graphics Systems—*Distributed/network graphics*

General Terms: Design, Performance

Additional Key Words and Phrases: Geometry streaming, multiplayer online games, game engine, continuous synchronization

ACM Reference Format:

Li, F. W. B., Lau, R. W. H., Kilis, D., and Li, L. W. F. 2011. Game-on-demand: An online game engine based on geometry streaming. ACM Trans. Multimedia Comput. Commun. Appl. 7, 3, Article 19 (August 2011), 22 pages.
DOI = 10.1145/2000486.2000493 http://doi.acm.org/10.1145/2000486.2000493

1. INTRODUCTION

Supporting online gaming is very challenging. One important issue is how to replicate relevant game content to client machines, which may at times not be trivial. As an example, [Second Life] owns

This work was partially supported by a GRF grant from the Research Grants Council of Hong Kong (RGC Reference Number: CityU 116008).

Authors' addresses: F. Li, School of Engineering and Computing Sciences, Durham University, Durham, U.K; email: frederick.li@durham.ac.uk; R. Lau, Department of Computer Science, City University of Hong Kong, H.K; email: rynson@cs.cityu.edu.hk; D. Kilis, Morgan Stanley Information Technology (Shanghai) Limited, Shanghai 200002, China; email: kilisd@gmail.com; L. Li, Department of Computer Science, City University of Hong Kong, H.K.

DOI 10.1145/2000486.2000493 http://doi.acm.org/10.1145/2000486.2000493

a growing set of game content, which is more than 270 terabytes at this moment, and much of the content is created dynamically by users. It may not be straightforward to distribute all of this content by currently available methods such as DVD and Blue-ray. Most computers and handheld devices may even have problems storing a small portion of this content. On the other hand, the network latency of the Internet causes motion discrepancy among the client machines. Hence, when a player moves around in the game scene, other players may not know it until after some delay. Such delay affects the visual quality perceived by the game players.

Our approach is for the game company to host the entire game scene on game servers. Once the client machine of a game player is accepted into the game, a relevant portion of the game scene is transmitted to the client so that the game may be started immediately. Additional game content will be transmitted to the client machine dynamically in an on-demand and timely fashion as the player moves around in the game scene [Li et al. 2004]. Our aim here is to maximize the user-perceived visual quality [Gulliver and Ghinea 2006], which concerns the overall presentation quality and accuracy of the game. A critical issue of the geometry-streaming approach is how we may efficiently determine which part of the game scene, and the amount of geometry data, is to be transmitted dynamically so that the visual quality can be maximized to support user interactivity. However, this can be challenging, as different client machines may have different rendering and network bandwidth capacities. There are some distributed virtual environment (DVE) systems [Falby et al. 1993; Greenhalgh and Benford 1995; Das et al. 1997] that support similar features. They are usually based on some predefined spatial information or area of interest (AOI) [Falby et al. 1993; Macedonia et al. 1995] of the game objects. Some concurrent work to ours also proposes to use some form of geometry-streaming techniques [Cavagna et al. 2006; Hu 2006]. However, we are not aware of any work that considers prioritizing the geometry models for transmission. In this article, we propose a method that would efficiently determine which game objects and their appropriate qualities to send for transmission dynamically, based on the available bandwidth.

On the other hand, as the player (or avatar) moves around in the game scene, its position changes continuously. However, due to network latency, there is likely a positional discrepancy of the avatar between the client and the server, resulting in view discrepancy. This causes two main problems. First, it affects user interaction, as the user may be reacting to some outdated information. Second, which is more relevant to our work here, it may affect the server in identifying and sending appropriate geometry data to the client. Hence, we need a way to minimize view discrepancy, but without causing discontinuous movements. We have developed a continuous synchronization scheme to improve the consistency of dynamic objects presented to the clients.

The main contributions of this article are summarized as follows.

(1) We propose a shadow object reference list to enable fast retrieval of dynamic objects.
(2) To have fine-grain control on the perceived visual quality, we propose a unified data structure for progressive transmission of different types of modeling primitives.
(3) We propose a prioritized content delivery scheme to allow visually important objects to be delivered with higher priority, and to adjust their qualities, that is, the amount of game data, to be delivered according to the given network bandwidth, the rendering capacity, and the available local memory space of the client machines.
(4) We propose a continuous synchronization scheme to minimize the state discrepancy between each client machine and the server.

The main idea of this work is to achieve two related objectives: to decouple scene complexity from interactivity (or response time) and to provide a graceful tradeoff between visual quality and download time. These objectives are similar to those of JPEG, which are to decouple image size from response time and to provide a graceful tradeoff between visual quality and download time.

The rest of this article is organized as follows. Section 2 gives a survey on related work. Section 3 presents the architecture of the game-on-demand engine. Section 4 introduces our two-level content management scheme, while Section 5 introduces our prioritized content delivery scheme. Section 6 describes our continuous synchronization scheme. Section 7 presents a number of experiments on our game engine prototype and evaluates the results. Finally, Section 8 concludes.

2. RELATED WORK

In this section we provide a survey on relevant work. We look at four areas of work that are related to ours communication architectures, content distribution, motion synchronization, and user-perceived visual quality.

2.1 Communication Architecture

Communication architecture defines how machines are connected. Common choices include *client-server*, *peer-to-peer* (P2P), and *hybrid* models. Many online games such as “Second Life” and “World of Warcraft” adopt the client-server model, as it offers better security control and data persistency by hosting and distributing game content using proprietary servers. Recently, the P2P model [Cavagna et al. 2006; Hu 2006] in which machines communicate directly with each other for data transmission without involving some forms of centralized servers has been explored. Technically, the client-server and the P2P models are very different in machine connections and task allocation. However, for content distribution, given that a suitable content-discovery mechanism is in place, e.g., Hu et al. [2004], with the P2P model, the peer (client) machines may act like mini-game servers to distribute downloaded game content to other peers, despite the fact that these “servers” may possess only a limited and dynamic subset of the game content. To enhance data persistency of a P2P-based game, the hybrid model [Botev et al. 2008], which employs separate servers on top of the P2P model to maintain and distribute game content, may be considered. Our opinion is that the client-server architecture provides central control of resources and game states. This is more important for commercial games. Although the P2P architecture allows resource sharing, the resources are distributed, and it is more difficult to impose some centralized resource/object management without involving a separate client-server layer. As the main purpose of the server process in our game engine is for information distribution, and the operations involved in the game engine are designed to be independent of the type of physical machine connections, our work does not restrict the choice of communication architecture for implementation.

2.2 On-Demand Content Distribution in DVEs

Existing 3D distribution methods can be broadly classified into *video streaming* and *geometry transmission*. In video streaming [Chang and Ger 2002; Pazzi et al. 2008], the servers render the scene and stream the rendered videos to the client machines for display. This approach is developed with the assumption that some client machines may not be powerful enough to render 3D objects. However, such concern is becoming less significant, as even mobile phones are now equipped with 3D rendering capability. In addition, since VEs are becoming more complex, with a lot of objects, and different users may have different views of the VE with different quality requirements, rendering images for all clients will impose very high workload on the servers.

In contrast, geometry transmission [Das et al. 1997; Falby et al. 1993; Hagsand 1996; Leigh et al. 1996; Saar 1999; Teler and Lischinski 2001; Waters et al. 1997] delivers geometric objects to the clients and relies on them to render the received objects. This approach is employed in some of the *distributed virtual environments* (DVEs) [Singhal and Zyda 1999] and forms the basis of our game engine. In a typical DVE, although the virtual environment (VE) may be very large, a user often visits only a small region of it. To save memory space and download time, a DVE system may transmit only the geometry of the VE that is visible to the user to the client and then dynamically transmit extra geometry to the client as the user moves around in the VE. Existing DVEs based on this approach can be roughly divided into *region-based* and *interest-based* categories. Systems including DIVE [Hagsand 1996], CALVIN [Leigh et al. 1996], Spline [Waters et al. 1997] and VIRTUS [Saar 1999] have adopted the region-based approach. They divide the whole VE into a number of predefined 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 be very large in data size, the system may need to pause to download the region content whenever the user switches from one region to another.

The interest-based approach typically uses the area of interest (AOI) [Falby et al. 1993; Macedonia et al. 1995] to determine object visibility. Only the scene content and updates within the AOI of the user need to be transmitted to the clients. Systems that have adopted this approach include NPSNET [Falby et al. 1993], MASSIVE [Greenhalgh and Benford 1995], and NetEffect [Das et al. 1997]. Although this approach may reduce the amount of game content for downloading, existing systems do not provide any mechanisms to control or guarantee the visual quality. In addition, it may still suffer from a long download time as with the region-based approach; particularly in high-quality games there may be too many objects inside the AOI or some objects inside the AOI may be very large in data size.

To improve the performance of the interest-based approach, *Second Life* constructs objects using CSG (constructive solid geometry) primitives and allows a primitive to be transmitted with a higher priority if it is a component of multiple visible objects. This allows the visible region of the game scene to be built up very quickly. Cavagna et al. [2006] represents each object using several levels of detail (LODs). When an object is selected for transmission, the lowest LOD will go first, followed by the higher LODs if they are requested. This method is simple, but it increases the network bandwidth consumption, as more than one LOD of each object may need to be sent. Hu [2006] represents each object as a progressive mesh [Hoppe 1996] for progressive transmission. It is similar to Cavagna et al. [2006], but surpasses it by sending out only one copy of each object. All these methods aim at shortening the time for the visible objects to be presented to the player. Unfortunately, due to the limited network bandwidth of the Internet, it may still be difficult for these methods to ensure that every visible object can be sent to the clients fast enough to support interactivity. To our knowledge, none of these systems consider prioritizing objects for transmission. Hence, it is possible that less important objects are transmitted before important ones.

2.3 Synchronization

As the Internet has relatively high network latency, an online game player may suffer from significant delay in receiving state updates from other players as they move around in the game scene. For example, in *Final Fantasy Online*, a player usually receives position updates of other players with almost a second delay. To reduce the effect of such delay, some restrictions are imposed on the game. 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. Such game rules significantly limit the type of games that can be developed.

Synchronization techniques have been developed for different applications, including distributed systems [Lamport 1978], database systems [Bernstein and Goodman 1981] and collaborative editing systems [Sun and Chen 2002]. These systems generally regard state updates as discrete events. Hence, they only need to ensure that state updates are presented to the relevant users in the correct order, without the need to consider the exact moment when the updates are presented to the users. However, this approach may not satisfy the requirements of online games or DVEs [Chim et al. 2003], where events are usually continuous [Mauve et al. 2004] and must be presented to the users within an accepted period of time in order to maintain the interactivity of the application. Unfortunately, it is not trivial to synchronize continuous object states. In addition to the network latency problem, there is also the time-space inconsistence problem [Zhou et al. 2004]. Existing methods mainly use time stamping to work out the "true" state of a remote object in online games. However, as timestamps are based on the senders' clocks but are typically interpreted based on the receivers' clocks, there is a need to align the clocks of all the machines (e.g., using the Network Time Protocol (NTP)) [Mills 1991].

To address the network latency problem, there are two main approaches: *forward recovery* and *backward recovery*. For forward recovery, *Unreal Engine* maintains a reference simulator at the server for each dynamic object and treats the local copies of the dynamic object running on the client machines as proxies. In general, the reference simulators and the proxies perform state changes independently. A reference simulator may broadcast its state to update all proxies if a critical change occurs. However, there is no mechanism to synchronize such an update. Mauve et al. [2004] proposes to delay presenting a new event to all participants with a "local-lag" period. Although this method is simple, it requires a sufficiently large local-lag in order to enforce synchronization among multiple players. In a client-server-based game, this method induces a 2-trip network delay for every single event, as each event must be sent to and propagated through a server. This significantly affects the game interactivity.

For backward recovery, [Cronin et al. 2002; Mauve et al. 2004] suggest a time-warp mechanism, which uses a separate buffer to record time-stamped object states during run-time. Any state inconsistency can be resolved offline, as the interactivity constraint does not apply there. If a significant game state problem is detected, a rollback operation will take place by undoing all incorrect updates and applying the corrected object states stored in the buffer. However, this is an undesirable action for many game players.

2.4 User-Perceived Visual Quality

The concern for user-perceived quality in distributed multimedia applications has been studied at the *media*, *network* and *content* levels [Gulliver and Ghinea 2006]. The media level focuses on the quality of individual objects. Typical examples include the quadric error metric for progressive meshes Garland and Heckbert [1997] and the object quality metrics devised in Teler and Lischinski [2001]. Such indicators work fairly well if network latency is ignorable. However, it is not the case in the Internet environment, as game players may perceive discrepant views of the game scene due to transmission delay. In contrast, the network level examines the user-perceived visual quality in the existence of network latency. It complements the media level to offer a better visual quality in a networked environment. For example, Pazzi et al. [2008] proposes a scheduling mechanism to allow the rendered videos to be streamed to the clients by observing the network, level quality requirement. As another example, our earlier work on DVE caching and prefetching [Chan et al. 2005] observes such a quality requirement and uses mouse-driven motion prediction to assist geometry transmission. Finally, the content level concerns the overall presentation quality of the application, which corresponds to the visual quality and accuracy of the output images in the case of online gaming. This is the prime indicator to measure how good a game is perceived by the game players. Our proposed content distribution and synchronization methods form an integrated solution to address these quality requirements.

3. THE ARCHITECTURE OF THE GAME-ON-DEMAND ENGINE

The game-on-demand engine adopts the client-server architecture to support progressive geometry delivery and synchronization. The server(s) maintains the game scene and handles the interactions among the clients (or game players) and the game objects. It also determines and schedules the required content at appropriate detail for delivery to the clients. Figure 1 shows the architecture of the

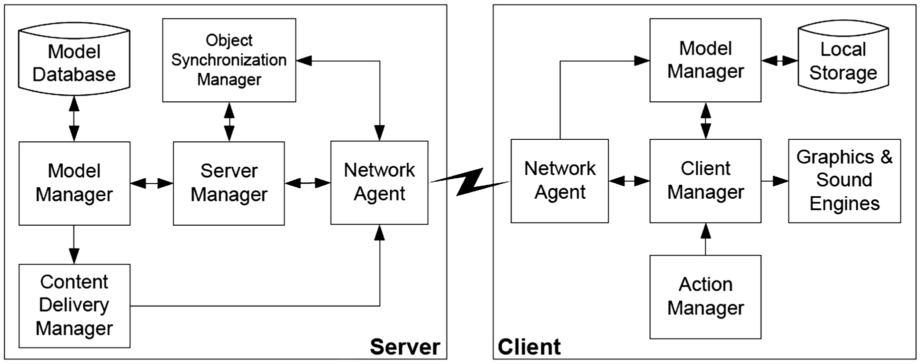

Fig. 1. The architecture of the game-on-demand engine.

game-on-demand engine. Note that if the peer-to-peer model [Cavagna et al. 2006; Hu 2006] is employed here, a peer-selection mechanism [Hu 2006] will need to be employed.

In summary, the server module has six components. The *Server Manager* coordinates all components at the server. It processes all interactions among game objects and sends updates to the clients. It also communicates with the model manager on the state of each game client. The *Model Manager* determines the required game objects and their appropriate visual qualities for delivering to the game clients (refer to Section 4 for details). The *Model Database* stores the complete game content, in which all the object models, texture images, and motion-captured data are kept in a progressive transmission ready format. The *Content Delivery Manager* manages the references of the selected items for prioritized delivery (refer to Section 5 for details). The *Object Synchronization Manager* is responsible for synchronizing the clients on the states of dynamic objects. (refer to Section 6 for details). Finally, the *Network Agent* handles all communication between the server and the clients, including content delivery and status updates.

The client module has six 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 application accessible storage space in the client for the model manager to store objects received from the server. The *Action Manager* reads control commands from input devices. It extrapolates the movements of dynamic objects to minimize the number of update messages sent over the network, and synchronizes dynamic objects to minimize the discrepancy between the client and the server (refer to Section 6 for details). The *Graphics and Sound Engines* are responsible for rendering visual and audio outputs in a time-critical manner. Finally, the *Network Agent* handles all communication with the server, including receiving the game models, texture images, and state updates from the server. It also sends updates and actions of the client to the server.

4. GAME CONTENT MANAGEMENT

The game-on-demand engine manages the game content at *scene* and *model levels* to support content selection. At the scene level, the content is stored in a way that the server may efficiently identify appropriate game objects for delivery. At the model level, different types of modeling primitives, including deformable models, rigid models, and texture images, are arranged in a unified data structure for progressive transmission and multiresolution rendering. For transmission, the server determines the optimal resolution of each object for delivery based on some dynamic conditions, such as object distance from the player or the available network bandwidth of the client machine. For rendering, a client may select appropriate resolution of each object to display using any real-time rendering method [To et al. 2001].

4.1 Scene-Level Content Management

To determine the visible objects to a particular game player, we have adopted the viewer/object scope idea from Chim et al. [1998], which can be considered a restricted form of the Focus/Nimbus model [Greenhalgh and Benford 1995]. We associate each object in the game with an *object scope* O_o, which is a circular region defining how far the object can be seen. We also associate each player with a *viewer scope* O_v, which is a circular region defining how far the player can see. However, unlike Chim et al. [1998], which defines the viewer scope as a single region, here we define it as a circular region with multiple parts, as described in Section 5. An object is visible to a player only if its O_o overlaps with the O_v of the player. Here, we classify the game objects into *static objects* and *dynamic objects*. A dynamic object, such as an avatar of a player or an autonomous object, may change shape or move around in the game scene while a static object, such as a building, is not supposed to move at all.

During runtime, as a player moves around in the game scene W, we need to continuously check for objects that are visible to the player, that is, objects to be transmitted to the client. To speed this process up, we partition W regularly into $|W|$ rectangular cells, that is, $W = \{C_1, C_2, \ldots, C_{|W|}\}$. Each cell C_n may contain a list of references to $|C_n|$ *shadow objects*, that is, $C_n = \{O_{Cn,1}, O_{Cn,2}, \ldots, O_{Cn,|C_n|}\}$, where $1 \le n \le |W|$. These shadow objects are objects, where object scopes overlap with C_n. To set up a game, for each object O_i, we add a reference of O_i to all the cells that object scope $O_{o,i}$ of O_i overlaps. During runtime, when we determine the potentially visible objects to a player, we only need to check all the cells that the O_v of the player overlaps. The set of potentially visible objects to the player is the union of all the shadow objects found in all the cells that O_v overlaps. For the dynamic objects that may move around in the scene, we dynamically update the shadow object lists of all the affected cells. To speed-up the searching time and the update cost of dynamic objects, we may split the shadow object list in each cell into two, one for static objects and the other for dynamic objects.

4.2 Model-Level Content Management

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

4.2.1 *Definitions of Game Objects.* Each game maintains a game object database Φ at the server(s), storing all the geometry models M, texture images T, and motion data A used in the application, that is, $\Phi = \{O, M, T, A\}$. 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 $T_{O_i} \subset T$); a number of motion data A_{O_i} (where $A_{O_i} \subset A$); an object scope $O_{o,i}$; and a viewer scope $O_{v,i}$, that is, $O_i = \{M_{O_i}, T_{O_i}, A_{O_i}, O_{o,i}, O_{v,i}\}$. In general, an object is composed of one or more models. Each model may include zero or more texture images. If O_i is a dynamic object, it may be assigned with a set of motion data or a program module indicating how O_i should move and react to the environment. Note that since each geometry model, texture image, or motion data 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 a player, we would check each of its primitives to see which were already transmitted or need to be transmitted to ensure that each primitive is transmitted to the same client once.

4.2.2 *Unification of Modeling Primitives.* To simplify the programming interface, we organize the geometry models, texture images, and motion data in a unified data structure. We refer to each of these transmission primitives as a modeling primitive 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

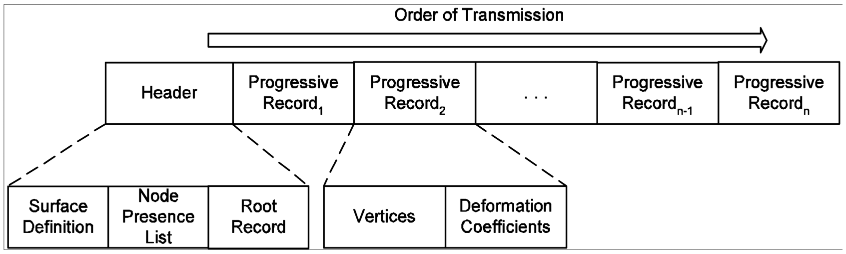

Fig. 2. Progressive transmission of a deformable NURBS model.

contains information for reconstructing U at its lowest resolution. This record is critical, as it is the baseline information for a game player to perceive the existence of this modeling primitive. Progressive records, in contrast, are used to improve the resolution of the modeling primitive. Delaying or even abandoning the transmission of these records may affect the perceived quality of the modeling primitive, but typically has, limited long-term effect. If we apply each of the progressive records p_n in P to U^0 using function $\Omega(u, p)$, a list of approximations of U, $\{U^0, U^1, U^2, \ldots, U^{|U|}\}$, is 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, until reaching the maximum resolution $U^{|P|}$, where $U^{|P|} = U$. To transmit U to a client, we first transmit the base record U^0 to help alert the player of the existence of a modeling primitive. Given more time, we may progressively transmit the progressive records to the client to enhance the visual quality of the modeling primitive. The four types of modeling primitives that we support are described as follows:

Rigid Models. These are triangular models encoded in a format similar to the *progressive meshes* [Hoppe 1996]. 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 coarsest approximation of U, is obtained. Hence, a list of progressive records $\{p_1, p_2, \ldots, p_{|P|}\}$ is 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 category of object that we support. The shapes of deformable objects may change over time. They are classified as dynamic objects and represented using NURBS. Each deformable object may be composed of one or more NURBS models, and optionally some rigid models, and each NURBS model is considered as a modeling primitive. In Li et al. [1997], we propose to represent each NURBS model by a polygon model and a set of deformation coefficients to support efficient rendering. This information is maintained in a quad-tree hierarchy. To support progressive transmission, we organize the quad-tree into an ordered list based on the z-ordering indexing scheme [Balmelli et al. 1999]. The list begins with a base record U^0, which consists of the surface definition, a node presence list, and a root record to represent the deformable NURBS model at the minimal resolution as shown in Figure 2. Subsequent records $\{p_1, p_2, \ldots, p_{|P|}\}$ of the list store information for refining the precomputed polygon model. The 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 a progressive JPEG format for transmission. However, JPEG requires an expensive inverse discrete cosine transform operation to extract texels from the compressed images. Instead, we extend the color distribution method [Ivanov and Kuzmin 2000] to support progressive transmission of compressed texture images. This method encodes a texture image by dividing it into uniform grids of texel blocks. Each texel block is encoded as two sets of 32-bit datum, 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

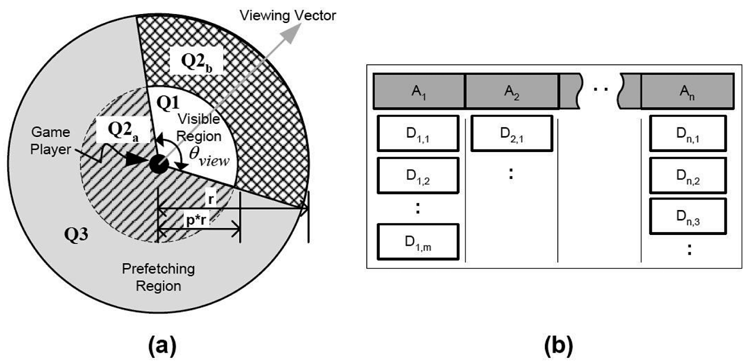

Fig. 3. The viewer scope and the object delivery queues.

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 1/8 of its size. To support 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 Geometry Transmission Priority

We make use of the *object scope* and *viewer scope* to identify interested objects to each player. In Figure 3(a), the viewer scope contains three regions, $Q1$, $Q2$, and $Q3$. $Q1$ is the *visible region*. All objects within it are considered as visible 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 may be transmitted to the client once all the objects in $Q1$ have been transmitted. $Q3$ is the *prefetching region*. Normally, it will take some time before objects within $Q3$ become visible to the player. Hence, we would prefetch them to the client machine if extra network bandwidth is available. To speed-up the process of selecting objects for transmission, we maintain a delivery queue for each of these regions. Hence, there are three queues, *Queue* 1, *Queue* 2, and *Queue* 3 for regions $Q1$, $Q2$, and $Q3$, respectively. All the objects in the delivery queues are sorted according to their transmission priority.

To efficiently determine the region that an object belongs to, we set up four viewing segments for each player in the player's local coordinate system as shown in Figure 4. We label the segment where 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-coordinates of $S1$ is labeled as $S4$. The four possible configurations of the viewing segments are shown in Figures 4(a) to 4(d). We refer to the left and right boundary of the visible region as V_L and V_R, respectively, as shown in Figure 4(e).

For each object G, we calculate its relative coordinate (x_g, y_g) from the player. Referring to Figure 4(e), the signs of x_g and y_g indicate the viewing segment that the object belongs to. The distance of G from

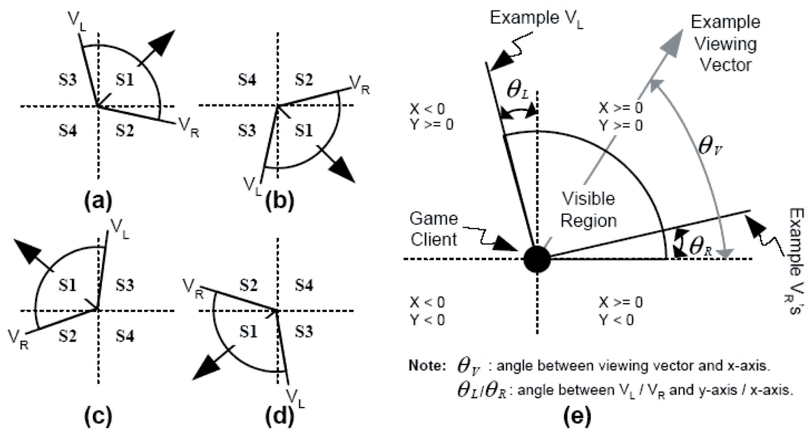

Fig. 4. Four viewing segments of a player.

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 $D_g < r + r_g$. If G is deliverable, we put a reference of G into the player's corresponding object delivery queue based on its visual importance to the player. In our implementation, the priorities of the three queues for transmission are: *Queue* 1 > *Queue* 2 > *Queue* 3. Objects placed in a queue with lower priority may be delivered to the player if all queues with higher priorities are emptied. (Of course, we may also implement it in a different way such that a different percentage of data would be sent from each queue, with the highest percentage from *Queue 1* and lowest from *Queue 3*.) Within a delivery 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 3(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 leftmost vertical subqueue (with lowest A_n) to the rightmost subqueue (with highest A_n) one by one. For each 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 I summarizes the rules to determine the region where an object G belongs. Column 1 shows the viewing segment where G belongs. 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 *Queue* 2. 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 I, 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. If the player has not changed its viewing direction, θ_V, θ_L, and θ_R as defined in Figure 4(e) can be considered constants. To determine the region of object G, we only need to evaluate either θ_{gx} or θ_{gy}, which is the angular distance of G from the x-axis or y-axis of the viewing segment of the player, respectively. For example, $\theta_{gx} = \tan^{-1}(y_g/x_g)$ and $\theta_{gy} = \tan^{-1}(x_g/y_g)$. In practice, the runtime arctangent evaluation may be avoided. Since we only consider objects that are within the outer circle of the prefetching region, we may precompute a set of arctangent values, $\tan^{-1}(x/y)$, with different combinations of x and y, where $x, y \leq r$. In addition, as we

Table I. Assigning Objects to Appropriate Regions

Viewing Segment	Rules for Determining the Region	Angular Distance
S1	If $\lvert\theta_v - \theta_{gx}\rvert \le \frac{1}{2}\theta_{view}$, if $D_g \le (r \cdot p + r_g)$, assign **G** to **Q1**; else, assign **G** to $\mathbf{Q2_b}$. 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 $\theta_{gx} \le \theta_R$, if $D_g \le (r \cdot p + r_g)$, assign **G** to **Q1**; else, assign **G** to $\mathbf{Q2_b}$. else, if $D_g \le (r \cdot p + r_g)$, assign **G** to $\mathbf{Q2_a}$; 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 $\theta_{gy} \le \theta_L$, if $D_g \le (r \cdot p + r_g)$, assign **G** to **Q1**; else, assign **G** to $\mathbf{Q2_b}$. else, if $D_g \le (r \cdot p + r_g)$, assign **G** to $\mathbf{Q2_a}$; else, assign **G** to **Q3**.	$\theta_{gy} + (90^\circ - \theta_v)$
S4	if $D_g \le (r \cdot p + r_g)$, assign **G** to $\mathbf{Q2_a}$; else, assign **G** to **Q3**.	If $x_g \le y_g$, $180^\circ + (\theta_{gx} - \theta_v)$; else, $180^\circ - (\theta_{gx} - \theta_v)$

have divided the game scene into a grid of small cells, the representative coordinates of each cell may well approximate the positions of all the objects in it. 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.

5.2 Object Quality Determination

After we have prioritized the objects for delivery, we need to determine the amount of geometric information, that is, 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 the object is considered for transmission. For each player, the server maintains a deliverable list of the transmission primitives. Each entry of the list stores two parameters of a primitive, P_{sent} and P_{opti}, which indicate the number of progressive records sent and the preferred optimal number of progressive records, respectively. P_{sent} is updated whenever some progressive records of the primitive are sent. P_{opti} is determined based on some real-time factors, as follows:

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

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 the client machine, respectively. To simplify the computation, we may assume that they are constants throughout the session. D and A are two dynamic factors, indicating the distance of the object from the player and the angular distance of the object from the player's viewing vector, respectively. If there is more than one visible object using a primitive, we would assign the smallest distance and angular distance values as A and D, respectively. This would increase P_{opti} to fulfill the maximum requirement of all the relevant 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.

6. CONTINUOUS SYNCHRONIZATION

The objective of our continuous synchronization scheme is to minimize the state discrepancy of each dynamic object (in particular the avatar) between its client host and the server. This has two advantages. First, it minimizes the user-perceived visual quality degradation caused by inconsistent object states among game players. Second, minimizing the discrepancy of the avatar between the client host and server allows the server to identify and send appropriate geometry data to the client.

Similar to the local-lag method [Mauve et al. 2004], our synchronization scheme attempts to align object states among the clients, providing a sufficiently correct condition for a motion predictor to perform dead reckoning [DIS98]. Unlike the local-lag method, we do not resort to pausing events as the tactic, as this introduces unacceptable delay. Instead, we proactively perform corrective actions to minimize state discrepancy. We use time duration instead of a timestamp as the key parameter in the synchronization process. This implicitly avoids the time-space inconsistency problem [Zhou et al. 2004]. We prolong the corrective action for a short period of time to avoid generating artificial state changes to trigger undesired user responses. Note that our synchronization scheme is designed to deal with continuous events. Discrete events, such as shooting and virtual item collection, can be handled by traditional causality control [Sun and Chen 2002].

In this scheme, we consider the server copy of each dynamic object as a *reference simulator* of the object. Regardless of whether the object is an avatar (representing the player) or an autonomous object, its motion must be synchronized according to its reference simulator. During a game play, a player may move around in the game scene by issuing motion commands with a keyboard or a control device. Motion vectors can then be computed based on these commands to form the navigation vectors of the avatar. The motion of the object between motion commands can be computed with a motion predictor. Here, we use the first-order predictor as follows:

$$p_{new} = p + t \times V \tag{2}$$

where p and V are respectively, are the position and velocity of the object from the last motion command received t is the time difference between p_{new} and p. Note that other predictors [DIS98] may be used instead.

6.1 Client-Server Synchronization

To illustrate the interactions between a client, *P1*, and the server, S, we consider the motion of the avatar on *P1*. There is a reference simulator of the avatar running in S. As shown in Figure 5, two motion timers T_s and T_c are maintained in S and *P1*, respectively. They are the virtual clocks indicating how long the avatar has been performing certain movements as perceived by the server and by the client. To maintain the integrity of the synchronization process, we allow only one motion command, which can also be a combined motion command, to be processed before T_s and T_c are synchronized. Hence, both the avatar and its reference simulator will be moving with the same motion vector during synchronization. Based on this, t in Eq. (2) becomes the only variable in the motion predictor, since we expect that both the avatar and its reference simulator will start to move from a synchronized position. As such, their motions are expected to be synchronized if $T_s = T_c$. When the avatar in *P1* issues a motion command (state **I**), the motion command is first buffered for a very short period: about 50ms in our implementation. All the motion commands received during this period are combined to produce a resultant motion vector to be sent to the server at the end of the buffering period (state **II**). Note that this buffering period serves as a low-pass filtering process to reduce noise from the keyboard or the game pad. Some devices may have already included a noise removal mechanism, so this buffering period may not be needed.

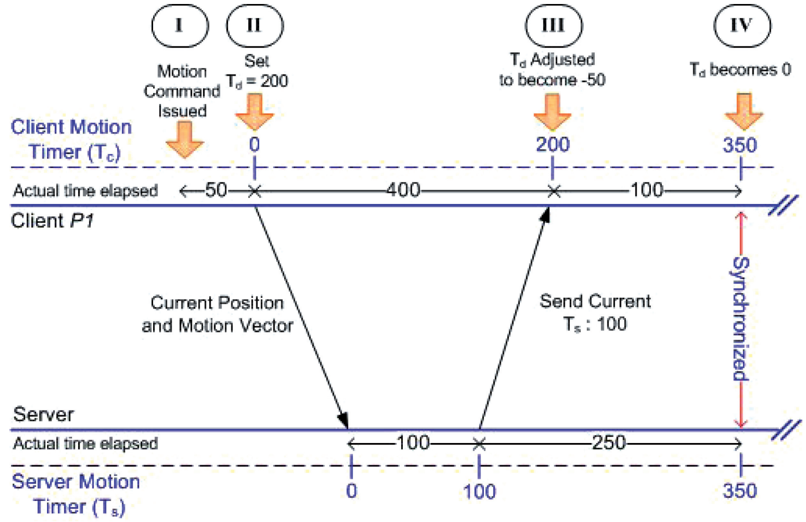

Fig. 5. Client-server interactions. (All values are in milliseconds.)

When a new motion command is issued from *P1*, this motion command will take a single-trip time to arrive at *S*. If we do nothing but just let *P1* execute the new motion command immediately, there will be a state discrepancy between *P1* and *S*, and the maximum discrepancy is a single-trip delay when the motion command has just arrived at *S*. With the continuous synchronization scheme, we slow down the motion at *P1* by half from the moment when a motion command is generated until the motion command has arrived at *S*. This effectively reduces the state discrepancy between *P1* and *S* by half to half of a single-trip delay. After *S* has received the motion command, the state discrepancy will gradually drop from half of a single-trip delay down to zero when *P1* and *S* are synchronized. By gradually aligning *P1* and *S*, we avoid visual artifacts caused by forcing an immediate state corrective action on the avatar.

Our synchronization algorithm begins at state **II**. We set T_d, the estimated time difference between T_c and T_s, to T_{S-P1}^{accum}, which is the accumulated weighted average of the single-trip network latency between *P1* and *S*. When computing T_{S-P1}^{accum}, we assume that the delay from *S* to *P1* is the same as from *P1* to *S*. Hence, T_{S-P1}^{accum} is equal to half of the round-trip delay. Let us assume that the client updates the position of the avatar every Δt, which is the duration between two consecutive frames (e.g., $\Delta t = 40$ms if a game renders 25 frames per second), through updating the following variables:

$$t = \Delta t - sgn(T_d) \times \varepsilon \tag{3}$$

$$T_d = T_d - sgn(T_d) \times \varepsilon \tag{4}$$

$$T_c = T_c + t \tag{5}$$

where t is the effective time increment for every Δt, $sgn()$ is the sign function; T_d is the estimated time difference between T_c and T_s; and ε is computed as $\min(|T_d|, \Delta t/2)$. Eq. (4) serves as a counter, so that the adjustment process will stop when T_d becomes 0ms when synchronization is achieved (i.e., state **IV** in Figure 5).

At state **III** of Figure 5, T_c is increased to 200ms while the actual time elapsed is 400ms since state **II**. This is because our synchronization scheme reduces the increment of motion timer T_c by half with Eqs. (3) and (5). At the same time, *P1* receives the value of T_s from *S*, which is 100ms. We then

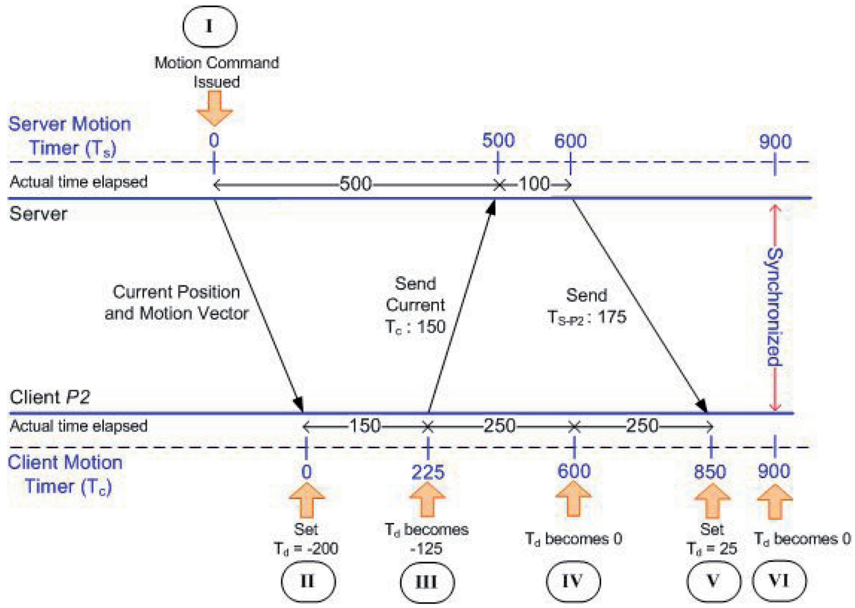

Fig. 6. Server-client interactions in the client-client synchronization process.

use it to estimate the single-trip network latency, T_{S-P1}, which is equal to approximately half of the absolute difference between T_s and the actual time elapsed from state **II** to state **III**. Whenever *P1* obtains an updated T_{S-P1}, T_d and T_{S-P1}^{accum} need to be adjusted (in addition to Eq. (4)) as follows:

$$T_d = T_d + \left(T_{S-P1} - T_{S-P1}^{accum}\right) \tag{6}$$

$$T_{S-P1}^{accum} = \lambda \cdot T_{S-P1}^{accum} + (1-\lambda) \cdot T_{S-P1} \tag{7}$$

where λ is an application-dependent weight for estimating future T_{S-P1}. In our experiment, setting λ to 0.5 gives reasonably good predictions most of the time. As T_d is 0ms at state **III** and the updated T_{S-P1} is 150ms, T_d is then adjusted to become -50ms, which indicates that the original T_{S-P1}^{accum} is overestimated by 50ms. The synchronization process will continue by repeating Eq. (3) – (5) until T_d becomes 0ms (state **IV**).

6.2 Client-Client Synchronization

Our reference simulator approach handles the client-client synchronization problem. As our synchronization scheme assumes that the server runs the reference simulators for all objects, each client machine only needs to synchronize itself with the server, instead of synchronizing directly with all other clients as in Mauve et al. [2004]. Synchronization among multiple clients can thus be resolved as independent client-server and server-client synchronization operations. Hence, our synchronization scheme can be run independently to the number of clients participating in the synchronization process.

With the scenario described in Section 6.1, we now have another player, *P2*, who wants to interact with *P1*. *P2* will need to send a request to *S* to obtain the motion information of *P1*, and, at the same time, create a simulator locally to model the motion of *P1*. Figure 6 depicts this process.

When *P2* receives the motion information of *P1* from *S* (state **II**), it sets $T_d = -T_{S-P2}^{accum}$ to start the simulation locally, where T_{S-P2}^{accum} is the accumulated weighted average of the latency between *S* and *P2*, and we assume it to be 200ms here. T_d is set to a negative value, since we anticipate that the

Fig. 7. A snapshot of *Diminisher*.

simulator running in *P2* starts with some delay relative to S. As such, t in Eq. (3) now becomes $1.5\Delta t$, which causes *P2* to speed up in order to catch up with S. Similar to the client-server synchronization process, *P2* updates the relevant variables in Eqs. (3) – (5), until it is synchronized with S.

In Figure 6, after *P2* has received the motion information of *P1* from S (state **II**) and some event processing delay, *P2* returns the time duration between states **II** and **III** to S (state **III**). Upon receiving the time duration, S estimates the single-trip network latency between *P2* and S, T_{S-P2}. T_c will continue to speed up until T_d becomes 0ms (stat **IV**). At state **V**, *P2* receives the updated T_{S-P2}. It then adjusts T_d and T^{accum}_{S-P2} with Eqs. (6) and (7). Since T_d has been set to 0ms at state **IV** and the updated T_{S-P2} is 175ms, T_d is then updated to 25ms, which indicates that the original T^{accum}_{S-P2} was overestimated. The synchronization process will continue by applying Eqs. (3) – (5) until T_d becomes 0ms (state **VI**). Finally, since any client interested in the position of the avatar only needs to synchronize it with the server, the motion of the avatar will ultimately be synchronized among all relevant clients.

For the client-client synchronization, when the user in *P1* issues a motion command, *P1* will send the motion command to S and start the continuous synchronization process immediately. Hence, the maximum discrepancy, which occurred when the motion command has just arrived at S, is a 0.5-trip delay. When S receives the motion command from *P1*, it propagates it to *P2*. The maximum discrepancy between S and *P2*, which occurs when the motion command has just arrived at *P2*, is a 1-trip delay. However, at this moment, the discrepancy between *P1* and S has been significantly reduced. Hence, the maximum discrepancy between *P1* and *P2* is between a 1-trip and 1.5-trip delay, depending on the relative network latency between *P1* – S and S – *P2*. Here, it may be interesting to compare it with other methods. With sudden convergence, the maximum state discrepancy between *P1* and *P2* is a single-trip delay. However, it suffers from discontinuous motion. With the local-lag method [Mauve et al. 2004], the maximum state discrepancy can be higher than a 2-trip delay in a client-server environment.

7. RESULTS AND DISCUSSIONS

We have implemented the proposed game-on-demand engine in C++ for the PC. Based on this engine, we have developed an online first-person fighting game called *Diminisher*. It allows multiple players to navigate in a shared game scene and to fight with each other or with some other automated opponents. Figure 7 shows a snapshot of the game. The file size of the client program is less than 600KB, which is small enough for game players to download from the Internet. With the client program, a player

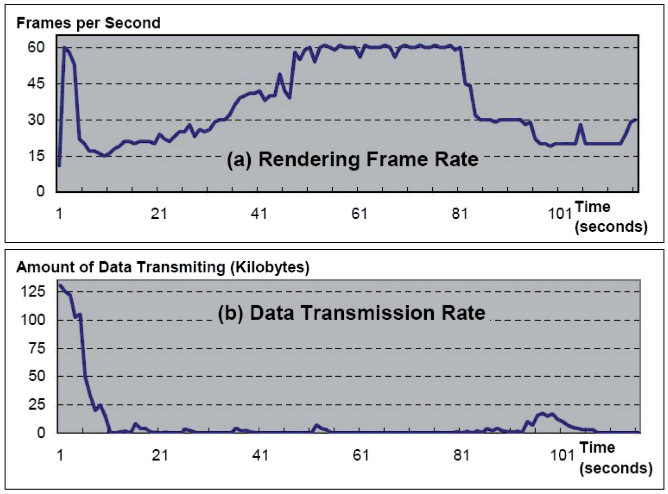

Fig. 8. Performance of the game demo.

may connect to the game server to obtain an initial content package, which contains the geometry information of the objects surrounding the player. After receiving the package, the player may start to play the game. Additional content is then progressively sent to the client based on the location of the player in the game. In this section, we conduct a number of experiments to evaluate the performance of the game-on-demand engine using the prototype game and the performance of the synchronization scheme.

Note that our main purpose in building the game prototype is to study the performance of the game-on-demand engine. We have not devoted too much effort in designing the game content. Nevertheless, this should not affect the validity of the experimental results, since with the proposed game engine, the size and the complexity of the game scene no longer determine the downloading time. This is made possible with the proposed game engine by adjusting the optimal resolutions of visible objects in proportion to the available network bandwidth.

7.1 Video Demo

A video showing a brief game-play session of *Diminisher* is included in this submission, and can also be downloaded at www.cs.cityu.edu.hk/∼rynson/projects/ict/shortdemo.mpg. The network bandwidth was set at 1.5Mbps. Figure 8 shows the rendering frame rate and the 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 data transmission rate at the beginning. This is because the initial content package only contains a minimal amount of geometry information for the client to render a low visual quality scene. To improve the visual quality, the server needs to send additional game content to the client. Due to the low visual quality (i.e., low resolution) of visible objects at the beginning, the frame rate was much higher. As the visual quality of the visible objects increased, 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 the frame rate. The frame rate rose again as most of the opponents were killed by the player. Finally, as the player moved to a scene with a large number of Easter Island sculptures, the client received a lot of geometry information and the frame rate dropped again due to the increase in the number of primitives that had to be

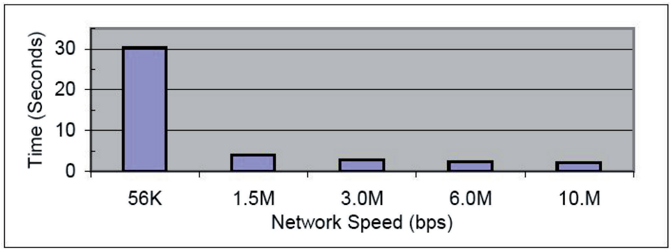

Fig. 9. Startup time vs. network connection speeds.

rendered. The pulses appeared approximately 15*s* after the start of the game, as shown in Figure 8(b), were due to the progressive content transmission.

7.2 Experiment with 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 so that the player could start playing the game. As shown in Figure 9, if a player has a broadband network connection, that is, network speed $\geq$ 1.5Mbps, the initial downloading time is less than 5*s*. If a game player connects to the game using a 56Kbps modem, the initial downloading time is only about 30*s*. Which is considered acceptable by most players.

7.3 Experiment with Content Delivery

Here, we test the performance of our geometry-streaming method. In the experiment, a player navigated the game scene in a circular path and looked around freely to visualize interested game objects. This allows the player to see different parts of the game scene, where each part has a different number of objects. 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. The user-perceived visual quality is defined as the weighted sum of the percentage of the required model data received by each visible object, $\mathbf{P}_i$, normalized by the sum of all weights, $\Sigma\omega_i$, where in this experiment the weight for the *ith* visible object is determined by a simplified version of Eq. (1), with $\rho = \varphi = 0.5$ and $\gamma = \tau = 0$. Hence, the visual quality can be written as

$$\Sigma(\omega_i \times \mathbf{P}_i)/\Sigma\omega_i \tag{8}$$

We compare the visual quality during the navigation using the following model transmission methods:

—Method A is our method;

—Method B uses the prioritized content delivery scheme only;

—Method C uses progressive transmission only; and

—Method D transmits the base record of each object model only.

We performed the experiment using a 56K modem (Figure 10(a)) and a 1.5Mbps connection (Figure 10(b)). (However, we allocated only 60% of the bandwidth for the game engine and the rest for collecting the statistics by the test program.) For methods with progressive transmission, that is, methods **A** and **C**, the game server would transmit the current visible objects to the client up to their optimal resolutions, as described in Section 5.2. A player is said to perceive 100% visual quality if the client receives all such objects in their optimal resolutions. For method B, a player could visualize

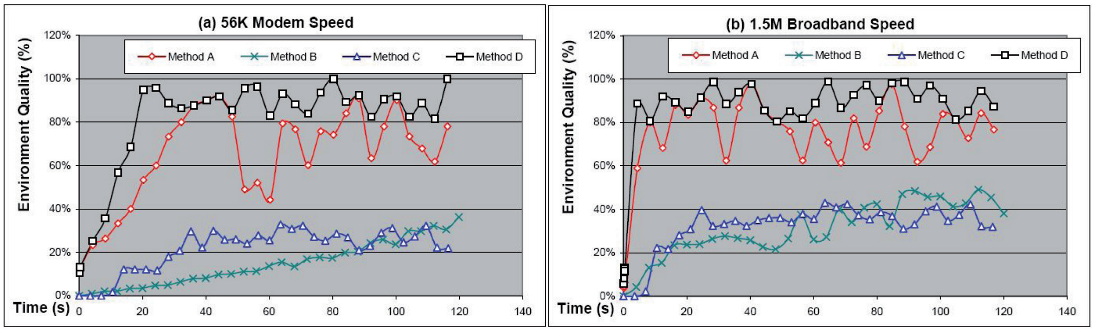

Fig. 10. Efficiency of various content delivery methods.

a game object only if the complete object model was received by the client. Hence, the player could perceive 100% visual quality of an object only if the model was completely transmitted to the client; quality was 0% otherwise. As a reference, we performed an additional test by setting the server to transmit only the base records of the objects requested (method D).

From Figure 10, we observe that our method (method A) offers the player a significantly better perceived visual quality than other methods do on both network conditions. Note that the fluctuation in the visual quality is due to the fact that the player keeps looking around in the game scene as it moves.

When compared with method C, our method could provide 20% to 70% higher perceived visual quality. In method C, progressive transmission allows more objects to be transmitted within a short period of time and to improve their visual quality progressively. It resembles the work in Hu [2006]. However, this method does not account for the fact that certain objects contribute more significantly to the overall visual quality than the others on the basis of some viewing parameters [Lau et al. 1997]. Without a proper scheduler, it cannot guarantee building high visual quality efficiently, as it sometimes spends time on transmitting less important objects before important ones. Our method produces a better result with the prioritized content delivery scheme.

When compared our method with method B; our method may occasionally even provide 80% higher perceived visual quality. Method B tests the effect of using the prioritized content-delivery scheme, but without progressive transmission. This method cannot efficiently build visual quality because an object may contribute to the overall visual quality only if the entire model has been received by the client. Hence, this method may spend a lot of time transmitting data which may not help improve visual quality.

Method D only transmits the base record of each object model, and this base record is defined to contribute 100% of object quality. The performance record collected for this method may serve as a reference. Given that method D only requires minimal geometry information to be transmitted to the client, the quality difference between methods **A** and D indicates that there is room for application designers to refine the value of the optimal perceived visual quality. This provides flexibility for game systems to adapt to various resource limitations. Practically, a game system can set up a performance test to help calibrate real-time factors and application-dependent weighting scalars, as described in Section 5.2.

7.4 Experiment with Synchronization

To determine the performance of the synchronization scheme in our game-on-demand engine, we have tested it with three player movement patterns: *linear*, *zig-zag*, and *circular* paths, under different

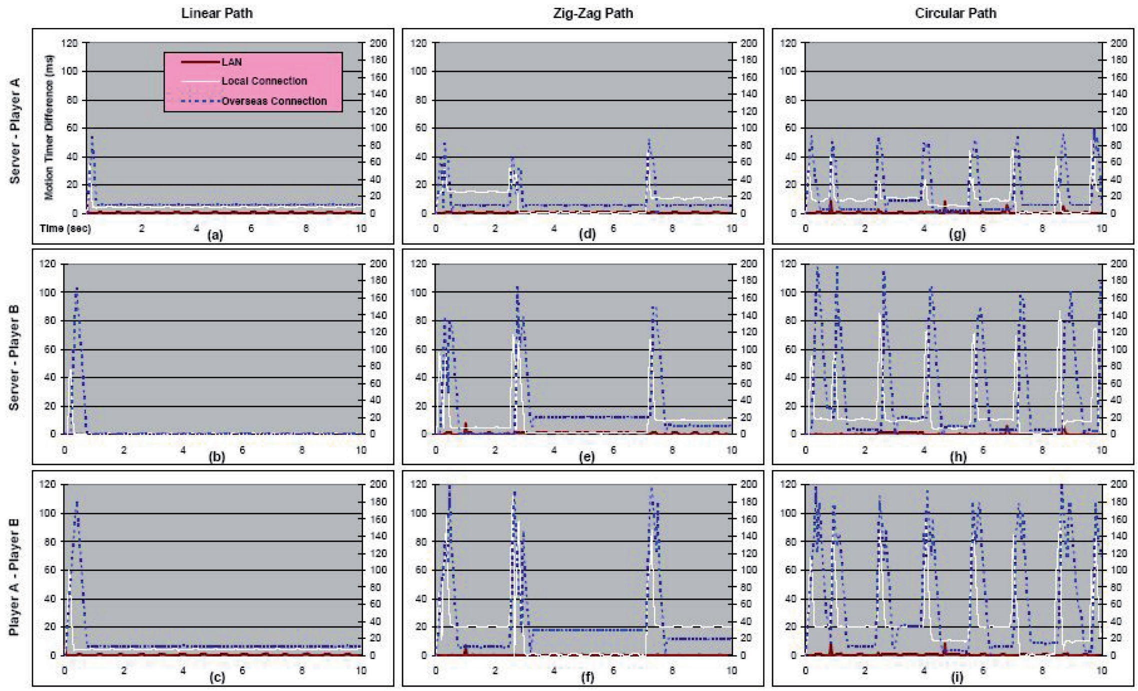

Fig. 11. Experimental results of our synchronization scheme. (All the numbers on the vertical axes are in miliseconds, while all the numbers on the horizontal axes are in seconds.)

network latencies, including 0.64ms (for LAN), 10 ms (for Internet, within a city) and 160ms (for Internet, international). In the experiment, player A navigates in the game scene and reports its states to the server, where the reference simulator of A is being run. Player B is interested in **A**'s motion, and requests A's motion information of A from the server. Since our synchronization scheme only requires a player to align its motion with the reference simulator running on the server, hence synchronization among multiple players can be resolved as independent client-server and server-client synchronization operations. To determine how well our synchronization scheme works, we monitor the motion timers of various parties for a period of time to determine the discrepancies among the relevant parties.

Figure 11 shows the performance of our synchronization scheme. To improve the readability of the results, we use two different scales to depict the state discrepancies under different network environments. Since the state discrepancy values collected from the LAN and from the local Internet connection are generally smaller, they are plotted based on the scale on the primary y-axes (the vertical axes on the left). However, the state discrepancy values collected from the overseas connection are in general much higher, and are plotted based on the scale on the secondary y-axes (the vertical axes on the right). As shown in the diagrams, all the relevant parties would in general experience a large state discrepancy when a motion command is just being sent out.

Consider the first set of results as shown in Figure 11(a) to 11(c). Since the results from different network connections generally exhibit similar behaviors, we focus our discussion on those from the overseas connection, as the effect here is more obvious. Since player A navigates in a linear path, only a single motion command is issued. In Figure 11(a), at around 0.2*s*, when the server has just received a motion command from A, there is a large state (or motion timer) difference between client A and the server, as they are running different motion commands. With the continuous synchronization scheme, we slow the speed at client A by half from the moment when a motion command is generated until the motion command has just arrived at the server. This effectively reduces the state discrepancy between A and the server by half to 0.5-trip delay, since by the time when the server receives the motion command, **A** has effectively been performing the motion command for only a duration of a 0.5-trip delay. After the server has received the motion command (at about 0.2s) and applied it, the state discrepancy will gradually drop from a 0.5-trip delay down to zero (at about 0.45s). At this moment, **A** and the server are said to have synchronized. Figure 11(b) depicts the motion timer difference between the server and player B, who is interested in the motion information of **A**. At 0.4*s*, when B has just received the motion information of A from the server, there is a large motion timer difference between B and the server. Since there is no trivial way to correct the discrepancy at this moment, the amount of discrepancy is equal to a single-trip delay. Finally, A and B are supposed to suffer from a two-trip delay. However, since **A** has started the continuous synchronization process with the server earlier, the discrepancy between **A** and the server would have been much reduced by the time B received the motion command from A. Hence, the discrepancy between A and B is reduced to about a single-trip delay.

Figure 11(d) to 11(f) shows the second set of results. Here, player **A** moves in a zig-zag path by pressing different command buttons alternatively so as to change the direction of movement. From the diagrams, we can see that whenever A issues a new motion command, a discrepancy pulse appears. We can see that the period from the start of a pulse to the start of the next pulse (e.g., the period from 0.3*s* to 2.5*s* in Figure 11(d)) exhibits similar behavior as in the first set of results. This is because during each of these periods, the avatar of **A** essentially moves in a straight line.

Figure 11(g) to 11(i) shows the final set of results. Here, player **A** moves in a circular path by continuously pressing different command buttons. Overall, this set of results exhibits similar behavior to the second set. In fact, both cases are conceptually similar, as they both require the player to continuously issue different motion commands in order for the avatar to move to the desired paths. The difference is that, in this set of results, there are more discrepancy pulses generated within the same period of time, as it requires the player to change the command buttons more frequently in order for the avatar to move circularly.

Finally, we can see from the results that some small discrepancy pulses occur in most of the diagrams, even after our synchronization operations (e.g., the period from 0.5*s* to 2.5*s* in Figure 11(d)). This is because our method attempts to correct the discrepancies based on estimating the network latency from recent network communications. This estimation would sometimes cause small errors when the network traffic fluctuates significantly.

To summarize, the proposed continuous synchronization scheme is simple and effective. The main limitation is that it can only handle one motion command at a time. A new motion command cannot be processed until the current one has been synchronized. It typically takes about one round-trip delay to synchronize a motion command. To reduce the effect of this problem while synchronizing a motion command, we combine all the motion commands received into one, to be processed after the current one is synchronized.

8. CONCLUSION

In this article, we have presented the game-on-demand engine for multiplayer online games based on progressive geometry streaming. The engine allows game clients to download game content progressively, without the need to purchase game disks or wait for a long download time. Our main contributions in this article include a two-level content management scheme for organizing the game objects, a prioritized content-delivery scheme for scheduling game content for delivery on the basis of object importance and network bandwidth, and a global-wise continuous synchronization scheme for synchronizing the motion timers in the client and in the server. They form an integrated solution to deliver good user-perceived visual quality to the game players. We have demonstrated the performances of the framework through a number of experiments.

As handheld devices are becoming ever more popular and because people carry them around, there is an increasing interest in technologies that support mobile games. Due to the small memory space, low network bandwidth, and high network latency of handheld devices, there have been difficulties in putting large games, and in particular multiplayer online games, into handheld devices. We believe that our game-on-demand engine can address these problems very well. As future work, we would like to adapt our engine to the handheld platform to support large-scale multiplayer online games. We would also like to consider simplifying the prioritized content delivery scheme, which is particularly important for mobile devices with lower processing capability. Finally, we would also like to consider different ways of selecting objects from the object delivery queues for transmission.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable and constructive comments on this article.

REFERENCES

BALMELLI, L., KOVAČEVIĆ, J., AND VETTERLI, M. 1999. Quadtrees for embedded surface visualization: constraints and efficient data structures. In *Proceedings of the International Conference on Image Processing* (*ICIP'99*). IEEE, Los Alamitos, CA, vol. 2, 487–491.

BERNSTEIN, P. AND GOODMAN, N. 1981. Concurrency control in distributed database systems. *ACM Comput. Surv.13*, 2, 185–221.

BERNIER, Y. 2001. Latency compensating methods in client/server in-game protocol design and optimization. In *Proceedings of the Game Developers Conference*.

BOTEV, J., HOHFELD, A., SCHLOSS, H., SCHOLTES, I., STURM, P., AND ESCH, M. 2008. The HyperVerse: Concepts for a federated and torrent-based '3D web. *Int. J. Advanced Media Comm. 2*, 4, 331–350.

CHANG, C. AND GER, S. 2002. Enhancing 3D Graphics on Mobile Devices by Image-Based Rendering. In *Proceedings of the IEEE Pacific Rim Conference on Multimedia*. IEEE, Los Alamitos, CA, 1105–1111.

CAVAGNA, R. BOUVILLE, C., AND ROYAN, J. 2006. P2P network for very large virtual environment. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (*VRST'06*). ACM, New York, 269–276.

CHAN, A., LAU, R., AND NG, B. 2005. Motion prediction for caching and prefetching in mouse-driven DVE navigation. *ACM Trans. Internet Technol. 5,* 1, 70–91.

CHIM, J., GREEN, M., LAU, R., LEONG, H., AND SI, A. 1998. On caching and prefetching of virtual objects in distributed virtual environments. In *Proceedings of the 6th ACM International Conference on Multimedia* (*Multimedia'98*). ACM, New York, 171–180.

CHIM, J., LAU, R., LEONG, H., AND SI, A. 2003. CyberWalk: A web-based distributed virtual walkthrough environment. *IEEE Trans. Multimedia 5*. 4, 503–515.

CRONIN, E., FILSTRUP, B., KURC, A.R., AND JAMIN, S. 2002. An efficient synchronization mechanism for mirrored game architectures. In *Proceedings of the Network and System Support for Games*. 67–73.

DAS, T., SINGH, G., MITCHELL, A., KUMAR, P., AND MCGHEE, K. 1997. NetEffect: A Network architecture for large-scale multi-user virtual world. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (*VRST*). ACM, New York, 157–163.

DIS STEERING COMMITTEE. 1998. IEEE standard for distributed interactive simulation—application protocols. IEEE Standard 1278.

FALBY, J., ZYDA, M., PRATT, D., AND MACKEY, R. 1993. NPSNET: Hierarchical data structures for real-time three-dimensional visual simulation. *Comput. Graph.17*, 1, 65–69.

FINAL FANTASY ONLINE. www.finalfantasyxi.com/.

GARLAND M. AND HECKBERT, P. 1997. Surface simplification using quadric error metrics. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques* (*SIGGRAPH'97*). ACM, New York, 209–216.

GREENHALGH, C. AND BENFORD, S. 1995. MASSIVE: A Distributed Virtual Reality System Incorporating Spatial Trading. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems* (*ICDCS'95*). ACM, New York, 27–34.

GULLIVER, S. AND GHINEA, G. 2006. Defining User Perception of Distributed Multimedia Quality. *ACM Trans. Multimedia Comput. Commun. Appl. 2*, 4, 241–257.

HAGSAND, O. 1996. Interactive multiuser VEs in the DIVE system. *IEEE Multimedia 3*, 1, 30–39.

HOPPE, H. 1996. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics* (*SIGGRAPH'96*). ACM, New York, 99–108.

HU, S. AND LIAO, G. 2004. Scalable peer-to-peer networked virtual environment. In *Proceedings of the ACM SIGCOMM Workshop on Network and System Support for Games*. ACM, New York, 129–133.

HU, S. 2006. A case for 3D streaming on peer-to-peer networks. In *Proceedings of the 11th International Conference on 3D Web Technology* (*Web3D'06*). 57–63.

IVANOV, D. AND KUZMIN, Y. 2000. Color distribution—A new approach to texture compression. *Comput. Graph. Forum 19*, 3, 28–290.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM 21*, 7, 558–565.

LAU, R., TO, D., AND GREEN, M. 1997. Adaptive multi-resolution modeling technique based on viewing and animation parameters. In *Proceedings of the IEEE Virtual Reality Annual International Symposium* (*VRAIS'97*), IEEE, Los Alamitos, CA, 20–27.

LEIGH, J., JOHNSON, A., VASILAKIS, C., AND DEFANTI, T. 1996. Multi-perspective collaborative design in persistent networked virtual environments. In *Proceedings of the IEEE Virtual Reality Annual International Symposium* (*VRAIS'96*). IEEE, Los Alamitos, CA, 253–260.

LI, F., LAU, R., AND GREEN, M. 1997. Interactive rendering of deforming NURBS surfaces. *Comput. Graph. Forum 16,* 3, C47-C56.

LI, F., LAU, R., AND KILIS, D. 2004. GameOD: An internet based game-on-demand framework. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (*VRST'04*). ACM, New York, 129–136.

MACEDONIA, M. R., ZYDA, M. J., BARHAM, P. T., PRATT, D. R., AND BRUTZMAN, D. P. 1995. Exploiting reality with multicast groups. *IEEE Comput. Graphics Appl*.

MAUVE, M., VOGEL, J., HILT, V., AND EFFELSBERG, W. 2004. Local-lag and timewarp: Providing consistency for replicated continuous applications. *IEEE Trans. Multimedia 6*, 1, 47–57.

MILLS, D. 1991. Internet time synchronization: The network time protocol. *IEEE Trans. Comm. 39*, 10, 1482–1493.

PAZZI, R., BOUKERCHE, A., AND HUANG, T. 2008. Implementation, measurement, and analysis of an image-based virtual environment streaming protocol for wireless mobile devices. *IEEE Trans. Instrument. Measure. 57*, 9, 1894–1907.

SAAR, K. 1999. VIRTUS: A collaborative multi-user platform. In *Proceedings of the Fourth Symposium on Virtual Reality Modeling Language* (*VRML'99)*. 141–152.

SINGHAL, S. AND CHERITON, D. 1994. Using a position history-based protocol for distributed object visualization. Tech. rep. CS-TR-94–1505, Department of Computer Science, Stanford University.

SINGHAL, S. AND ZYDA, M. 1999. *Networked Virtual Environments: Design and Implementation*. Addison Wesley, Reading, MA.

SECOND LIFE. secondlife.com.

SUN, C. AND CHEN, D. 2002. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Trans. Comput.-Human Interact. 9*, 1, 1–41.

TELER, E. AND LISCHINSKI, D. 2001. Streaming of complex 3D scenes for remote walkthroughs. *Comput. Graph. Forum 20,* 3, 17–25.

TO, D., LAU, R., AND GREEN, M. 2001. An adaptive multi-resolution method for progressive model transmission. *Presence 10*, 1, 62–74.

UNREAL ENGINE. unreal.epicgames.com/Network.htm.

WATERS, R., ANDERSON, D., BARRUS, J., BROGAN, D., CASEY, M., MCKEOWN, S., NITTA, T., STERNS, I., AND YERAZUNIS, W. 1997. Diamond park and spline: A social virtual reality system with 3D animation, spoken interaction, and runtime modifiability. *Presence 6*, 4, 461–480.

WORLD OF WARCRAFT. www.worldofwarcraft.com.

ZHOU, S., CAI, W., LEE, B., AND TURNER, S. J. 2004. Time-space consistency in large-scale distributed virtual environments. *ACM Trans. Model. Comput. Simul.14*, 1, 31–47.

Received July 2009; revised December 2009; accepted February 2010