Real-Time Streaming and Rendering of Terrains

Soumyajit Deb[1,2], Shiben Bhattacharjee[1], Suryakant Patidar[1], and P.J. Narayanan[1]

[1] Centre for Visual Information Technology
International Institute of Information Technology Hyderabad
[2] Microsoft Research India, Banglore
sdeb@microsoft.com, {shiben@research., skp@research., pjn@}iiit.ac.in

Abstract. Terrains and other geometric models have been traditionally stored locally. Their remote access presents the characteristics that are a combination of file serving and realtime streaming like audio-visual media. This paper presents a terrain streaming system based upon a client server architecture to handle heterogeneous clients over low-bandwidth networks. We present an efficient representation for handling terrains streaming. We design a client-server system that utilizes this representation to stream virtual environments containing terrains and overlayed geometry efficiently. We handle dynamic entities in environment and the synchronization of the same between multiple clients. We also present a method of sharing and storing terrain annotations for collaboration between multiple users. We conclude by presenting preliminary performance data for the streaming system.

1 Introduction

Traditional graphics applications store all geometry locally in the main memory itself. Geometry can also be stored remotely and received progressively when needed and rendered on the fly. Streaming of geometry of large virtual environments can be beneficial and difficult if the network bandwidth is low. Such systems find applications when data cannot be replicated easily. Dynamic environments such as those used for battlefield visualization involving real terrains and multiple players is an example. Different users may read/update parts of the virtual environment while maintaining a collaborative and consistent system across heterogeneous users connected from client machines with different capabilities and network bandwidths. A similar situation is presented by massive, multi-player online games consisting of dynamic persistent worlds.

Geometry cannot be split into frames or chunks unlike media like audio and video. Some parts of a model cannot be lost unlike video where the loss of a frame may be acceptable. A complete model is necessary to render geometry. Each may be used for rendering several frames and hence need to be stored at the client. In this sense, geometry should be served like files from a file-server. On the other hand, geometric model should reach in time for real-time rendering of the final rendered video. Delay can result in the undesirable effects like the freezing and popping. The real-time constraint makes it possible for us to talk

P. Kalra and S. Peleg (Eds.): ICVGIP 2006, LNCS 4338, pp. 276–288, 2006.

about streaming geometry. The parallel in geometry to the reduced bit-rate encoding of audio and video is *level of detail (LOD)*, which is the representation of the shape at different levels of approximations. The server can send a lower detail model to the client to reduce various resource requirements. Thus, remote access of geometry is an interesting mix of data serving and content streaming.

In this paper, we address the issue of streaming terrain data over networks. Real terrain datasets can span multiple gigabytes in size and are difficult to render interactively using brute force methods. Level of detail methods must be used to reduce the amount of detail to be rendered. However, if we are to render terrains remotely, the biggest bottleneck is the available network bandwidth between the server and the client. We need to optimize the transmitted terrain data accordingly depending upon client type and and available bandwidth. This is a hard problem as the system must achieve performance akin to local rendering at the client end. This is compounded by the fact that the system must keep track of any dynamic entities that exist in the environment and update the clients accordingly. This work extends to terrains an earlier work on geometry streaming [1]. We propose an optimized terrain representation based upon tiles for efficient transmission and rendering in Section 3. In Section 4 we look at the basic requirements of a geometry streaming system and our design and implementation of the same. Section 5 presents efficient ways of adapting these techniques for a terrain rendering system including performance improvements for the system using the ideas of prefetching and caching. Methods to synchronize and render multiple dynamic entities in the environment are discussed in 6 followed by experimental results in 7 and conclusions 8.

2 Related Work

Media streaming over the web has been popular and several standards exist. Most media streamers allow a specific bitrate to be chosen based upon available bandwidth and dynamic changing of bitrate to adjust accurately to client parameters. Google Earth/Maps streams maps and satellite imagery in real-time over the internet. However it does not address individual client characteristics which may lead to lags and freezes.

Djurcilov and Earnshaw added compression of models to VRML and developed an integrated visualization system, where the basic selection of data is done by the user [2,3]. However even after revisions of VRML which included geometry compression (Li et al.) [4], it is not usable for web-based serving since data needs to be transmitted before rendering can begin. Commercial products for remote visualization includes VisServer software from Silicon Graphics which allows rendering of any OpenGL application on remote clients by transmission of individual frames. Funkhouser describes a system based upon client server architecture for multi-user virtual environments [5]. The WireGL/Chromium project is a system that provides the familiar OpenGL API to each node in a cluster, virtualizing multiple graphics accelerators into a sort-first parallel renderer with a parallel interface [6]. [7] uses a crude model at the client for navigation and

streams actual high quality views from the server using viewing parameters to protect high detail content.

Among geometry based approaches, Schneider and Martin describe a framework which adapts to the client characteristics including network bandwidth and the client's graphics capabilities [8]. Martin describes an Adaptive Rendering and Transmission Environment (ARTE) framework that facilitates the delivery of 3D models while monitoring the resources available [9]. This uses MPEG4 stream compression which may lead to lag in response to user input. Teler describes a remote rendering system utilizing path prediction and bandwidth based level of detail reduction [10]. This system fails to dynamically change/adapt parameters during the course of the walk-through which may lead to suboptimal performance. Deb and Narayanan develop a system to stream general polygonal models between a server and client in [1]. However this approach is suitable for only tessellated models and may not be the optimal for terrains.

In recent terrain rendering approaches, Lossaso and Hoppe[11] describe how terrains can be broken into geometric clipmaps of varying metric sizes and that these clipmaps can be used as Level of Details. This is however not the most optimal representation for streaming. Their method also calculates the blend/morph factor on a per vertex basis because of inhomoegeneous tile sizes which may slow down lower end clients. Wagner [12] divides the terrain into regular square tiles for rendering. However the view frustum culling approach used by [12] fails in cases when the terrain has large variations in heights. The ground plane is unable to include the projection of tiles, which are near to the camera looking at horizon, since they are out of that projection but inside the view frustum. Pouderoux and Marvie [13] design an out of core terrain rendering system based upon a heuristic metric. However they do not address the problem of network streaming.

In our system, we follow a technique similar to Wagner [12] for dividing the terrain into square tiles to make it easy for the geometry streaming system to select regular data to be transmitted to the client. We can calculate blending factors on a per tile basis because of the use of a regular tile structure thus reducing the amount of computation. Given the tile indices, their object space location is easily computable making query systems on the terrain efficient. However regular tiles become very small at the extremities of the viewing frustum. We take care of this problem by using very low levels of detail for such tiles in view. We tweak Wagner's frustum culling technique by having the projection on realtime average height of terrain in the view and not simply the ground plane.

3 Terrain Rendering

We describe the various steps involved in first creating our terrain representation and then rendering it. Before rendering the tiles, we must store them in a data structure that is suitable for both rendering and also for transmission. View frustum culling is required to select only the necessary entities in the view frustum. VFC is very useful for streaming as it allows us to select only a small portion of the entire terrain for transmission.

3.1 Data Organization

Terrain data consists of a height value for every point x, y on a rectangular grid. We divide it into **tiles** of equal size for rendering. By equal we mean they cover the same rectangular area on the heightmap. To handle levels of detail, we arrange the data in a specific way. For a tile with size $2^n \times 2^n$ height values, we store m number of LODs, $m \leq n$, m is a user defined number based upon characteristics and size of the terrain. We also keep the distance between adjacent heights in x, y as s_x, s_y Fig 1. For an LOD l we have $2^{n-l+1}+1 \times 2^{n-l+1}+1$ $(l > 0)$ number of height values and $2^n + 1 \times 2^n + 1$ for $l = 0$. Note the extra heights at the end corners of the tiles, they are the height values at the starting corners of the next tile; kept as they help in stitching (see Section 3.5). This means $l = 0$ holds highest detail and $l = m$ holds lowest detail as illustrated in Fig. 1. For $l > 0$ we keep original height values h at $(2i, 2j)$ locations, $0 \leq i, j \leq 2^{n-l}$. We replace the height values at $(2i, 2j + 1)$ locations with $avg(h_{2i,2j}, h_{2i,2j+2})$, at $(2i + 1, 2j)$ locations with $avg(h_{2i,2j}, h_{2i+2,2j})$, at $(2i + 1, 2j + 1)$ locations with $avg(h_{2i,2j}, h_{2i+2,2j+2})$; where i, j vary as bounded. This is done so that while rendering when LOD l with alternate height values dropped, we don't see any change in the structure.

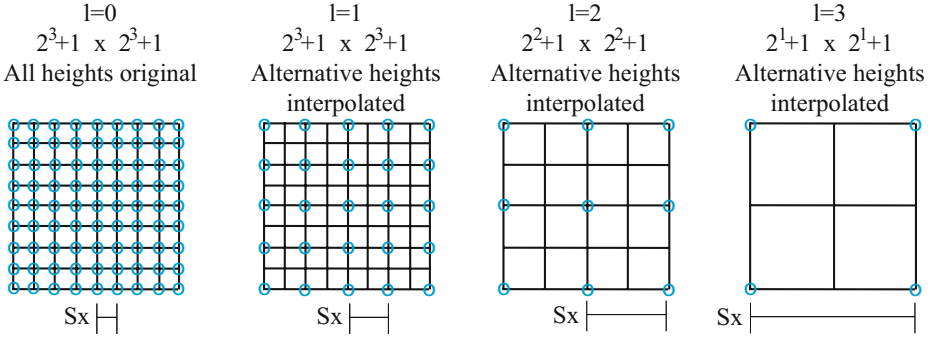


Fig. 1. Data organization: An e.g. with $n = 3$ and $m = 3$, blue circled height values are original, rest are interpolated. Note that, they occupy the same area on ground.

3.2 View Frustum Culling

In each frame, we query the graphics API for view frustum equations and calculate the projection P of the frustum (generally a trapezoid) on the base plain. This base plain is $z = a_h$, a_h is the approximated average height of the terrain in view of previous frame. This is because we haven't accessed the terrain data yet and thus will be using the data from previous frame assuming that the view hasn't changed much. We then calculate orthogonal bounding rectangle of P. We can directly map the coordinates of the bounding rectangle to tile indices. Using these tile indices, we find other tiles that are inside P (Fig.2). We keep the indices that return positive in a tile buffer B_t for use in rendering. We do not

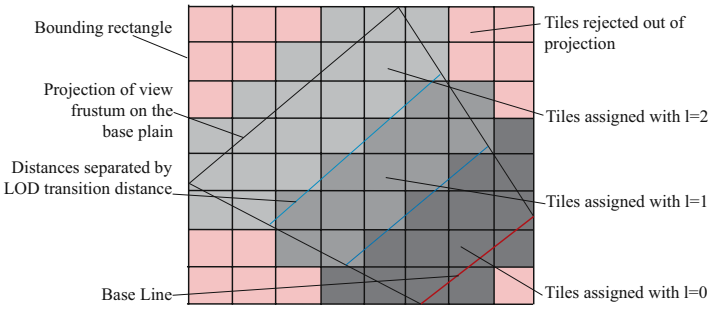


Fig. 2. View frustum culling and LOD assignment

need to do 3D view frustum culling as terrains are injective functions on x, y, and thus can be reduced to 2D in turn to reduce number of required calculations.

3.3 LOD and Blending Factor Calculation

Using the camera parameters we calculate a base line, that is perpendicular to the view vector and parallel to the ground plane. For each tile in B_t, we calculate the perpendicular distance d of its mid point from this line (Fig.2). This distance d is used to calculate LOD l as $\lfloor d/l_t \rfloor$ where l_t is the LOD transition distance. We choose this distance d instead of the direct distance of the tile from the camera because if the field of view of the camera is high, we shall end up rendering tiles at the corner of screen that are actually close to camera in screen space but far in object space in very low level of detail. The value $frac(d/l_t)$ is the blending factor α. α is used for smooth level of detail changes of tiles as explained in Section 3.4. We save l and α in B_t along with the tile indices.

3.4 Rendering

With all data in place, the tiles can be rendered from B_t. For all tile indices in B_t, we load the level l and $l+1$ of that tile. The index is clamped to m to avoid memory exceptions. The distance between adjacent heights for l can be calculated as $(s_{x_l}, s_{y_l}) = (s_x, s_y)2^l$ Fig 1. We calculate the heights h for $l > 0$ as

$$h = h_{(2i,2j)_l}(1-\alpha) + h_{(i,j)_{(l+1)}}\alpha$$

$l = 0$ is a special case: $h = h_{(i,j)_0}(1-\alpha) + h_{(i,j)_1}\alpha$, i, j vary as bounded. We can now see that when α is 0, $h = h_{(2i,2j)_l}$, and when α is 1, $h = h_{(i,j)_{(l+1)}}$. Thus this blending factor is able to smoothly change between the two height values of 2 different LODs of the same tile as we move the camera. On the fly, we also calculate the average of the heights at the mid point of these tiles, a_h, which will be used in the next frame for view frustum culling (See Section 3.2).

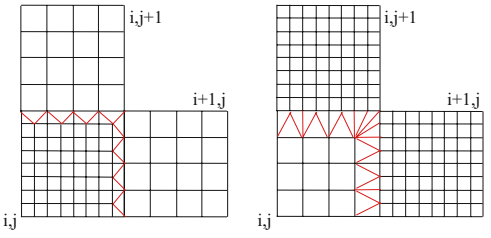


Fig. 3. Tile Stitching: tile i, j is stitched only to $i, j+1$ and $i+1, j$

3.5 Tile Stitching

Since every tile is getting assigned l and α independently, we find un-tessellated areas near the corner of each of the tiles. We assume that a tile on the ground with LOD l can have a nearby tile whose LOD can be only $l-1$ or $l+1$. This makes tile stitching easy and smooth blending of LODs works perfectly. Our assumption remains true iff l_t is always more than the maximum distance a tile can extend on the ground, i.e., the tile is never able to skip an LOD in between. So for a tile index t_i, t_j in B_t, we get the l and α of $t_{(i+1)}, t_j$ and $t_i, t_{(j+1)}$, and use them for the corner heights of t_i, t_j Fig 3. Note that we are not looking at $(i-1, j), (i, j-1)$ indices of tiles since those corners are already stitched by earlier tiles.

4 General Geometry Streaming

The basic objective of a geometry streamer is to provide each client with data appropriate to it as quickly and efficiently as possible. The server must allow the highest quality rendered output possible for the client and transmit geometry and assets that allow the client to maintain an acceptable frame-rate. Changing latencies should not cause the system to freeze or hang for long durations during the walkthrough. The server should adapt to the different client parameters such as graphics capability, network bandwidth and connection latency. Ideally, these must be met strictly. We briefly outline the basic requirements of a Client-Server geometry streaming system. The basic architecture of the system is similar to the system in [1] which may be referred to for further information.

Transparency: A transparent streaming system treats remote and local objects without distinction. The architecture of the system allows a user program to include remote models from multiple servers into its local virtual environment. The client API will handle the necessary tasks such as server interaction, data caching and management etc., transparently. The streamed data will match with the client machine's capabilities and the network properties.

Support for varying clients and networks: No client should receive a model that it cannot handle at interactive rates. A suitable level of detail is sent to each client based on capabilities of rendering hardware. Multiple levels of

detail may be used for improved performance on low-end clients. The heightmap and model detail can be reduced to handle different connection speeds to avoid freezing. A model matching the client's capabilities may be sent subsequently by progressively refining the original heightmap or model. Frequent updates to the model at the client can be avoided by sending the client more information than immediately necessary. Continuous connection monitoring and adjustment of detail is essential for good streaming performance.

Support for dynamic objects and local modifications: The server module should keep track of the static and dynamic objects transmitted to the client for each of the connected clients. Changes to an existing model in the virtual environment are notified to all clients possessing the same. All clients must have access to dynamic objects and their state information. User programs can have local control of transmitted model. It can mix and match remote models with local models, and can modify local copies of remote models.

The system consists of the Server Module, the Client Module and the Terrain Renderer (User Program) as shown in shown in Fig 4.

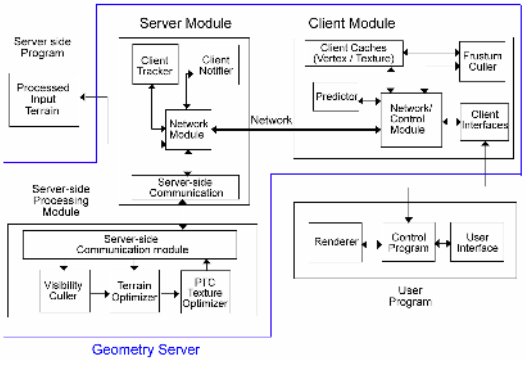


Fig. 4. Geometry Streaming System Block Diagram

The basic functions of the server module include managing a database of heightmaps and models, accepting incoming connections from different clients, serving the clients appropriately and quickly and handling dynamic objects in the virtual environment. Server receives requests for transmission from clients. In response, it generates and transmits a representation of the model suitable for the client. Each client request is translated into a model optimized based on available bandwidth, client capabilities and viewer speed. Low quality models will suffice when the user is moving fast, which may be progressively refined when the user slows down.

The client module interfaces with the user program on one side and the server module on the other. It provides a client API to the user program through which, the user program communicates with the client module, provides an initial set of client parameters and receives data. The amount of data transmitted is to be minimized by the system to avoid wasteful use of available bandwidth.

A local model and a remotely served heightmap/model should appear the same to the user for transparent streaming. A handle to the remote heightmap or model is returned to the user program by the client module. This is used by the Terrain Renderer (User Program) directly. The user program is responsible for the interaction with the user and the navigation control in the virtual environment. The user program passes the motion parameters to the client module on user movement in the virtual environment.

5 Terrain Streaming

In our system, the terrain data exists on the server and must be transmitted to the client on demand. The renderer described in Section 3 is completely based on the client side. Instead of loading the data from local storage, the renderer issues an API call *LoadRemoteTile()* to the client module. This call requests for a particular tile in the terrain at a desired level of detail. The client module maintains a local cache of tiles which is typically much larger than the number of tiles in the viewing frustum. The cache maintains tiles at varying levels of detail depending upon initially negotiated client parameters and available network bandwidth. If the tile does not exist in the client cache, the client module streams it from the server. Until the tile is actually received, an upsampled version of the existing data for the same location in the heightmap is used if available.

Tile Transmission: At the start of the walkthrough, the system transmits a very low resolution heightmap for the entire terrain dataset. As the viewer moves around over the terrain, higher quality data is streamed to the client depending upon viewer position. The renderer will have a bare minimum representation of the entire terrain available to it. The renderer requests for newer data as and when required. To transmit the tile, the last transmitted resolution of the tile is checked and only the residue between the high resolution tile and the supersampled version of the low resolution tile is transmitted to the client by the server. The residue is compressed using the wavelet based PTC codec [14] before transmission. At the client end, the client module decompresses the representation and generates the high resolution tile. We use geomorphing to smoothly blend across tiles without any visible artifacts.

Tile Selection: Selecting the optimal set of tiles to be streamed is difficult problem. We need to not only select the tiles to stream but also the level of detail of the tiles to be streamed. We only need to stream tiles that are visible or would become potentially visible in the near future. This is done by taking multiple square sets of tiles around the viewing frustum. The inner squares have the higher levels of detail than the outer squares. Once the frustum moves, newer higher resolution tiles must be streamed to the client.

Object Selection: Objects present on the tiles are selected in a similar manner as the tiles. The objects are anchored to a particular point on the terrain. The discrete levels of detail of the object are precomputed. When selecting a particular level of detail of an object, we check the level of detail of the underlying

tile and select the LOD of the object accordingly. The entire model must be transmitted as there is no easy way of creating a general polygonal model from a residue and lower level of detail in real-time. We maintain a list of transmitted objects on the server and never retransmit the same or lower level of detail.

The client module performs caching and prediction needed for better performance and interfaces with the server. The client module prefetches data based on predicted motion depending on the latency between the server and the client. The client caches already transmitted data so that requests can be avoided when the viewer retraces the navigation path. A balance is established so that the amount of data prefetched is enough to cover the potential areas in the virtual environment that the viewer might visit until the time of the next request. Caching and prefetching are transparent to the renderer. The organization of the cache is important as a cache miss is extremely expensive as data must be fetched from the remote server before it can be rendered in full detail. Each heightmap and its corresponding geometry is timestamped when the cache is updated. The object tracker logs the objects moving in and out of the cache along with their LOD.

Using the positions of the viewer in the past, the motion parameters are extracted. These are then used to estimate future motion This method of prediction works when the motion of the user in the world is smooth and continuous. Good performance from the prefetching algorithm is absolutely necessary to maintain a smooth walkthrough. Formally, we average the motion in the last 5 frames to generate the motion parameters for the next frame. We use a prediction scheme similar to [15]. However we assume that the rate of change of acceleration is constant. Assuming a constant acceleration may not be the right thing in case an object is experiencing rotational motion. If a_i is the acceleration, v_i the velocity and P_i the position vector in the ith frame which takes t_i time to render, assuming that the change in acceleration is smooth, we get the following relationships: $\boldsymbol{a_i} - \boldsymbol{a_{i-1}} = \boldsymbol{a_{i+1}} - \boldsymbol{a_i}$ or $\boldsymbol{a_{i+1}} = 2\boldsymbol{a_i} - \boldsymbol{a_{i-1}}$. Knowing that $\boldsymbol{a_i} = \frac{\boldsymbol{v_i} - \boldsymbol{v_{i-1}}}{t_i - t_{i-1}}$ and $\boldsymbol{v_i} = \frac{\boldsymbol{P_i} - \boldsymbol{P_{i-1}}}{t_i - t_{i-1}}$, this reduces to $\boldsymbol{v_{i+1}} = \boldsymbol{v_i} + (t_{i+1} - t_i)(2\boldsymbol{a_i} - \boldsymbol{a_{i-1}})$ and finally to $\boldsymbol{P_{i+1}} = \boldsymbol{P_i} + c_1(t_{i+1} - t_i)\boldsymbol{v_i} + c_2(t_{i+1} - t_i)^2(2\boldsymbol{a_i} - \boldsymbol{a_{i-1}})$, for some constants c_1 and c_2. The right hand side of this equation consists of known quantities other than t_{i+1}. $\boldsymbol{P_{i+1}}$ can be written completely in terms of earlier samples of $\boldsymbol{P}$ and frame time t. We do not reproduce the same here for the sake of brevity. Normalizing $\boldsymbol{P_{i+1}}$ will yield us the position vector of the point for which data needs to be prefetched. Since we do not know the value of t_{i+1}, we must estimate it from older known values of frame times. Once we know the future position of the viewer, we can prefetch data corresponding to that particular position. The amount of data prefetched depends directly on the size of the cache. We can control the bounds of the area of the terrain to be prefetched depending upon the cache size.

6 Dynamic Entities and Collaborative Environments

An environment or mode is defined as dynamic if its objects can change in form, position or appearance or if there is any addition or deletion of objects.

Synchronization of the state of a dynamic object in all clients is essential to avoid inconsistencies. The magnitude of the amount of data to be transmitted depends upon the type of change occurring in the dynamic environment. The different types of dynamic events that may occur in the VE are:

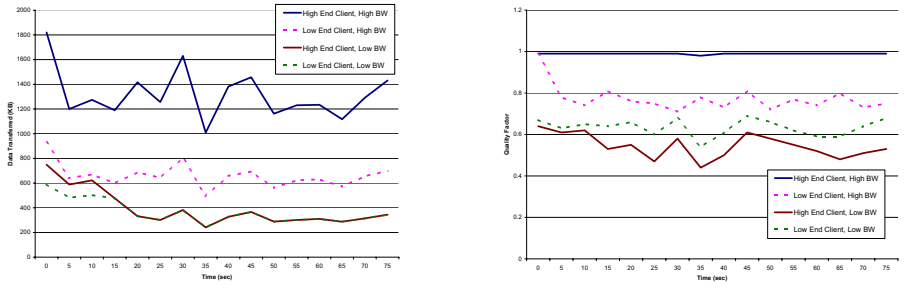
- Change in the position of an object in the VE: This is the simplest case as only the new position needs to be sent. If motion is parametric, the positions can be computed by the user if initialized properly. No data needs to be transmitted since the position of the object maybe calculated by the client provided such a motion model is known to the client.
- Change in form/shape of the object: The model of the newly changed shape needs to be streamed.
- Addition of a new object to the VE: The server needs to calculate which clients need the object based on the view frustums and notify the clients accordingly.
- Deletion of an object from the VE: Notify the clients who possess the object. They in turn can delete the object from their client cache.

To handle dynamic data efficiently, the client must be notified of the changes immediately. There are two ways in which this can done. One way is to send the changed data directly. The other way is to inform the clients about the change and allow them to initiate data transmission. The approach of lazy updates is preferable since data need not be sent unless needed. When the dynamic object comes into view at the client's end, data can be requested for and transmitted. The difference in this scenario over a typical static VE is that the server needs to initiate the transmission of individual objects in the VE without applying visibility calculations. This is an additional requirement to handle dynamic environments. A typical sequence of events during a walk-through of an environment with dynamic objects is as follows: (1) Dynamic Object is introduced into the VE or an existing object changes form. (2) Server Module checks the type of change and the clients affected by it. (3) Server notifies the affected clients of the change. (4) Clients request and download the required information when they need it.

Online mapping applications are becoming all pervasive. We have witnessed web based mapping applications such as *Windows Live Local* and *Google Maps* gain popularity over the past few years. The next step in evolution of such applications is a real 3D interface with community editing and sharing features. Our current system allows annotations of the terrain as basic collaborative features. Once a user annotates a particular position in the heightmap, the renderer passes this information to the server. Henceforth these annotations are treated in the same manner as dynamic objects. The only difference is that we allow serialization of the annotations at the server side in a database. The entry for each annotation includes the coordinates of anchor point in the heightmap as an index for retrieval. Currently we use flat text files but extending it to a real database instead is easy. The data is preserved across sessions of work. We intend to improve upon this feature and allow multiple options.

7 Results

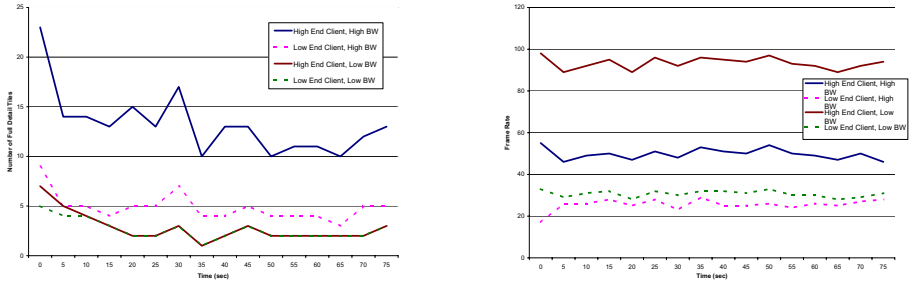
Our test client system consisted of a Pentium M 2GHz laptop with 2GB of memory and 6800 Ultra graphics. For the low end test, we used a Pentium 4 1.5GHz machine with 256MB memory and onboard Intel 845G graphics. The server was an AMD Opteron CPU running at 2.8GHz with 2GB of main memory. Please do note that the clock speeds of the CPUs are not comparable. The laptop CPU is significantly faster than the low end client because of a better architecture. The client and server were connected over a 100BaseT LAN. The lower bandwidth conditions were simulated over this network by limiting network traffic. The terrain was a 10000x10000 heightmap which was around 300MB of data. We use the *Quality factor* metric from [1] to measure the performance of the system. This metric is 1.0 when the client is rendering at its best LOD.



(a) (b)

Fig. 5. (a) Data transferred during Walkthrough and (b) Achieved Quality Factor

The data transfer graph is indicated in Fig.5. We find that the data transfer graph is especially smooth. This implies that the walkthrough is free of hitches and popping. Fig 6 shows the number of high resolution tiles submitted compared to lower resolution tiles. We find that the high resolution tiles account for the majority of the data transmitted to the client. This is because we use a lower compression ratio for high resolution tiles as they are close to the viewer. We find the quality factor 5 to be extremely high as expected with the system achieving a high framerate 6. The quality factor seems to degrade with client type and available bandwidth. The higher end client is worse affected by lower available bandwidth than the lower end client as a larger amount of data needs to be transmitted in case of the higher end client. The low end client, we find that the system is initially unable to cope with the amount of data causing poor frame rates. The system consequently reduces the highest level of detail transmitted to the client and henceforth the walkthrough experience is acceptably smooth. The amount of data transmitted also flattens to a plateau indicating a smooth walkthrough experience without hitches. The quality factor



(a) (b)

Fig. 6. (a) Number of Highest LOD tiles transferred and (b) Achieved Framerate of Walkthrough

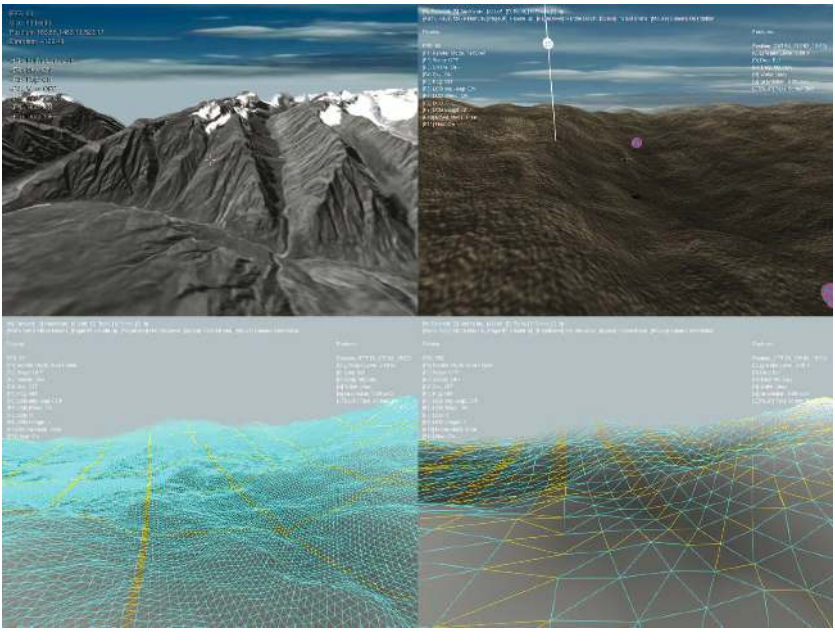


Fig. 7. (a)Output on Real Dataset (b)Dynamic Objects (c) High LOD terrain (d) Low LOD terrain

is lower for the low-end client than the high end client but still quite acceptable. From the low bandwidth tests, we cap the maximum available bandwidth to 100 KB/s. We observe that the network bandwidth is always utilized for progressive refinement.

8 Conclusions and Future Work

We presented a Terrain Streaming and Rendering system which renders data received from a remote server and appropriately adapts to client characteristics and network bandwidth. The system utilizes a tile representation for efficient transmission. It uses a combination of visibility culling, clientside caching, speculative prefetching, motion prediction and deep compression to achieve perfomance similar akin to local rendering. The system supports dynamic entities in the environment allowing the content developer to create collaborative 3D virtual environments. It also supports shared annotations as a preliminary collaborative feature. We intend to include support for realistic terrain deformations at the client end in future versions along with more collaboration support. Streaming systems that serve terrains are especially suitable for applications like Virtual Earth which must transmit large amounts of terrain information. Multiplayer games and flight simulators shall also benefit by utilizing streaming to incorporate new content.

References

1. Deb, S., Narayanan, P.: Design of a geometry streaming system. In: ICVGIP. (2004) 296–301
2. Djurcilov, S., Pang, A.: Visualization products on-demand through the web. In: VRML. (1998) 7–13
3. Earnshaw, R.: The Internet in 3D Information, Images and Interaction. Academic Press (1997)
4. Li, J.: Progressive Compression of 3D graphics. Ph.D Dissertation, USC (1998)
5. Funkhouser, T.A.: Ring: A client-server system for multi-user virtual environments. I3D (1995) 85–92
6. Humphreys, G., Eldridge, M., Buck, I., Stoll, G., Everett, M., Hanrahan, P.: WireGL: A scalable graphics system for clusters. In: SIGGRAPH. (2001) 129–140
7. Koller, D., Turitzin, M., et al, M.L.: Protected interactive 3d graphics via remote rendering. In: SIGGRAPH. (2004) 695–703
8. Schneider, B., Martin., I.M.: An adaptive framework for 3D graphics over networks. Computers and Graphics (1999) 867–874
9. Martin, I.M.: Arte - an adaptive rendering and transmission environment for 3d graphics. In: Eighth ACM international conference on Multimedia. (2000) 413–415
10. Teler, E., Lischinski, D.: Streaming of Complex 3D Scenes for Remote Walkthroughs. EuroGraphics (2001) 17–25
11. Losasso, F., Hoppe, H.: Geometry clipmaps: terrain rendering using nested regular grids. ACM Trans. Graph. **23** (2004) 769–776
12. Wagner, D.: Terrain geomorphing in the vertex shader. ShaderX2, Shader Programming Tips and Tricks with DirectX 9, Wordware Publishing (2003)
13. Pouderoux, J., Marvie, J.E.: Adaptive streaming and rendering of large terrains using strip masks. In: Proceedings of ACM GRAPHITE 2005. (2005) 299–306
14. Malvar, H.S.: Fast progressive image coding without wavelets. In: DCC '00: Proceedings of the Conference on Data Compression, Washington, DC, USA, IEEE Computer Society (2000) 243–252
15. Guthe, M., Borodin, P., Klein, R.: Real-time out-of-core rendering. To appear in the International Journal of Image and Graphics (IJIG) (2006)