Remote Rendering of Massively Textured 3D Scenes Through Progressive Texture Maps

Article · January 2003

CITATIONS
11

READS
38

2 authors:

Jean-Eudes Marvie
Technicolor
37 PUBLICATIONS **277** CITATIONS
SEE PROFILE

Kadi Bouatouch
Université de Rennes 1
157 PUBLICATIONS **1,487** CITATIONS
SEE PROFILE

Remote Rendering of Massively Textured 3D Scenes Through Progressive Texture Maps

J-E. Marvie and K. Bouatouch
IRISA-INRIA
University of Rennes
Rennes, France
email: {jemarvie,kadi}@irisa.fr

ABSTRACT
In this paper we present a new progressive texture map (PTM) format that encodes the mipmap levels of a texture map into a compact and progressive way. In order to manage these PTMs for a purpose of remote visualisation of architectural 3D scenes, we make use of a space subdivision and a visibility computation algorithm that are performed in a preprocessing step. In addition to this preprocessing, we precompute a metric which is used during the navigation to select the texture mipmap levels to be downloaded and those to be added/removed to/from the graphics hardware memory. Thanks to these mechanisms, we generate and visualize scalable databases for architectural 3D scenes that contain a large number of high resolution texture maps. The visualisation system we propose is based on a client/server architecture and allows for the transmission and the visualization of 3D scenes using either a remote server connected via low bandwidth networks or a local server. In addition our system automatically adapts to the power of the client machine.

KEY WORDS
Rendering, Textures, Walkthrough, Streaming

1 Introduction

Thanks to high performance graphics hardware, texture mapping has become a standard use to increase realism and visual quality of real-time 3D applications. Nevertheless, most common applications such as games or 3D for the web systems always use 3D scenes that are scaled to fit the actual hardware performances. Web 3D applications is an excellent illustration. One can never know how powerful the client's hardware and how fast the network will be. The most commonly used solution is to propose several rendering qualities to the end user which has to choose one before downloading the 3D scene data (geometry and texture maps). Another solution consists in using progressive transmission of data. Such a solution allows the end user to visualize at first a low quality representation of the downloaded content while downloading the additional data needed to refine this representation. When downloading an image, the end user always wants the image to be of highest quality. Conversely, when the user moves rapidly in a 3D model he does not need high resolution texture maps all the time. Finally, depending on the user graphics hardware and on the network bandwidth, the highest resolution and the number of texture maps that are used in a 3D model are variable parameters. By using progressive texture maps and a precomputed metric to select the required and appropriate resolution for texture mipmaps [10], we propose an efficient and scalable system which aims at generating and visualizing remote highly textured 3D scenes.

2 Related work

Progressive image transmission has been studied by many authors, and many different solutions have been developed. Many of today image compression methods propose progressive transmission mechanisms, including row interlacing (GIF), scan-based progressive encoding (JPEG), and hierarchical progressive encoding (Zerotree [6] or SPIHT [5]). For textures, S3 compression (S3TC) provides good compression ratio for texture maps (6:1) and recent graphics hardware are able to decompress these texture maps on the fly. Thus, this compression scheme is usefull to reduce the AGP bus transfers as well as the required graphics hardware memory when using high resolution texture maps. Although these last techniques are very powerful for image compression and transmission, they are neither (most of time) lossless nor adapted to encode texture mipmaps.

The progressive transmission of textures gets interesting if one is able to select and download *only* the mipmap levels needed to produce the best visual representation for a given viewpoint. In their streaming system [7] Teler et al. propose to use the number of pixels (called pixel coverage from now on) that the projection of an object's bounding rectangle covers on the screen, as a metric to select its required visual quality. Consequently, some objects can be inside the view frustum but occluded by some other objects (occluders). Therefore, occluded objects might have large pixel coverages whereas they are not visible. Dumont [2] applies a two pass algorithm to update the texture mipmap levels in the graphics hardware. The result of the first rendering pass is analysed to compute the pixel coverage of

each visible object that is then used to update the texture mipmap levels used during the second rendering pass. This algorithm copes well with the problem of occluded objects and provides the best visual quality. On the other hand, using a two pass algorithm and scanning the frame buffer entails many computations that can drastically reduce the frame rate.

To cope with the problem of occluded objects, a visibility preprocessing [1, 9, 8, 4] between a set of cells resulting from the subdivision of architectural scenes can be performed. More precisely, the model is subdivided into a set of cells (rooms, corridors, etc.) using a BSP technique. Moreover cell-to-cell visibility relationships are determined. Another advantage of this method is that, during remote navigation, only the viewcell (the cell that contains the viewpoint) together with its potentially visible set of cells (PVS) need to be transmitted through the network to perform the rendering on the client side. Furthermore, this kind of scene partitionning allows to perform data prefetching and memory management [3].

3 Overview

With our system several client machines can connect to a server to visualize static 3D scenes stored in scalable databases. The main goal of our system is to minimize the transmission of texture map data from the server to the client memory (RAM) and from client RAM to the OpenGL graphics hardware. In order to allow for a progressive transmission of architectural scenes during the visualisation process, they are first subdivided into cells [8] and a cell-to-cell visibility relationship is established.

The texture maps of the original scene are then converted into the progressive texture map (PTM) format we present in Section 4. This file format encodes the mipmap levels of a texture using a differential mechanism. Thus, the mipmap levels can be transmitted separately starting from the lowest mipmap resolution. Though storing the mipmap levels of a square texture map entails an extra storage size of more than 30% compared to the original texture map, the differential method we propose increases the size of the original map only by 6%.

Instead of computing the pixel coverage of each visible cell during navigation [7, 2] we precomputes, for each cell, the average pixel coverage value of each cell within its PVS. These values, valid for any viewpoint lying in the convex hull of the cell for whitch they are computed, will be called ACHs (Average Coverage Hints). We will see in section 5 how to precompute these ACHs using an OpenGL graphics hardware.

In Section 6 we explain how the client makes use of the preprocessing results to download geometry and to select the mipmap levels needed for the best visual representation. We show how we manage these selected levels to reduce the AGP transfers by filling the graphics hardware memory in a way better than that of a classical LRU policy, to minimize the network transfers and to optimize the client RAM occupation. Finally, we present some results before we conclude.

4 Progressive Texture Maps

The aim of the PTM format we propose is to encode the mipmap representation of a texture map in a compact way that allows the transmission of the mipmaps, level by level. A first solution would be to encode the sequence of mipmap levels in a raw manner into a single file. Although this method is simple, it entails data redundancy. Actually, a texture map with a resolution of 1024x1024x24bits requires 3MB of memory space whereas the associated mipmaps requires 4MB. The solution we propose consists in storing, for each level, only a portion of this level, whereas the rest can be reconstructed using few additional data and the level just below as seen hereafter.

4.1 Floating point solution

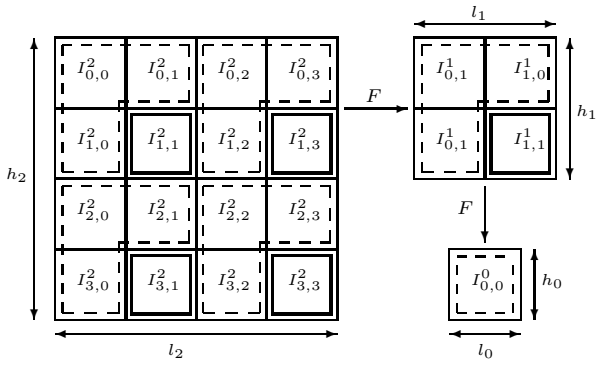

Figure 1. The three lower levels of a square PTM. For each level, the dotted lines surround the pixels that are transmitted from the server to the client (partial mipmap level) and the bold lines surround the pixels that are computed on the client side. F is the low-pass filter that transforms a level n into a level $n-1$.

In this section, we show how to reconstruct a mipmap level n knowing $\frac{3}{4}$ of its content (this fraction will be called partial mipmap level) and its lower level $n-1$. To compute a mipmap level $n-1$, we apply the 2x2 low pass filter F to level n for each color component. Figure 1 shows the three lower mipmap levels of a square texture map.

$$F = \begin{pmatrix} \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} \end{pmatrix}$$

The color component $I^{n-1}_{i,j}$ of a texture pixel $P^{n-1}_{i,j}$ of the mipmap level level $n-1$ is obtained using Equation (1), where $I^n_{i,j}$ is the intensity of the color component of pixel $P^n_{i,j}$ of the mipmap level n having a width and a height

equal to l_n and h_n respectively.

$$I_{i,j}^{n-1} = \frac{1}{4} \sum_{\substack{0 \le l \le 1 \\ 0 \le m \le 1}} I_{i\cdot 2+l, j\cdot 2+m}^{n} \qquad \begin{cases} \forall i,j \in \mathbb{Z} \\ i < h_{n-1} \\ j < l_{n-1} \end{cases} \quad (1)$$

A partial mipmap level n contains the color components of only three pixels (i.e. generated with the filter F) for each of its $2x2$ pixel blocks. The missing color component of the fourth pixel of a pixel block can be computed using the three generated color components of this block and that of lower level $n-1$, say $I_{i,j}^{n-1}$ (see Equation (2)). In Figure 1, the three generated color components are outlined with dotted lines while the computed missing color components are outlined with bold lines.

$$\begin{cases} I_{i\cdot 2+1, j\cdot 2+1}^{n} = 4\cdot I_{i,j}^{n-1} - I_{i\cdot 2, j\cdot 2}^{n} - I_{i\cdot 2+1, j\cdot 2}^{n} - I_{i\cdot 2, j\cdot 2+1}^{n} \\ \forall i,j \in \mathbb{Z} \qquad i < h_{n-1} \qquad j < l_{n-1} \end{cases} \quad (2)$$

Having the level $n-1$ in the client memory, we only need to download $\frac{3}{4}$ of the level n, the rest can be computed at the client side using Equation (2). Consequently, instead of transmitting a $4MB$ texture, we only transmit $3MB$, which corresponds to the original size of the texture map. This solution is optimal in floating point representation but it induces some residual errors when using integer representation.

4.2 Integer solution

Usually, texture maps are encoded using 24 bits or 32 bits per pixel and each color component has to be coded using unsigned integers. However, in Equation (1), the division by 4 generates a remainder $r_{i,j}^{n-1}$ ranging within the discrete set $\{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}\}$. It generates, in Equation (2), a residual error $\varepsilon_{i,j}^{n-1}$ within the discrete set $\{0, 1, 2, 3\}$. Therefore, when downloading the mipmap level n we also need to download the residual errors for this level to reconstruct the missing color components. In this way, the reconstruction of the level n is given by the modified Equation (2):

$$\begin{cases} I_{i\cdot 2+1, j\cdot 2+1}^{n} = 4\cdot I_{i,j}^{n-1} + \varepsilon_{i,j}^{n-1} - I_{i\cdot 2, j\cdot 2}^{n} - I_{i\cdot 2+1, j\cdot 2}^{n} - I_{i\cdot 2, j\cdot 2+1}^{n} \\ \forall i,j \in \mathbb{Z} \qquad i < h_{n-1} \qquad j < l_{n-1} \end{cases} \quad (3)$$

The following section shows how we encode the partial mipmap levels and the residual errors.

4.3 Compact representation

The residual error $\varepsilon_{i,j}^{n}$ for a color component can be encoded using 2 bits. The division by 4 is done by shifting the dividend value by 2 bits on the right and the 2 bits of the remainder are retrieved using a logical mask. Therefore, the algorithm for encoding-decoding is very fast, and storing the errors increases the size of the texture file only by 6%. Furthermore, each partial mipmap level can be encoded using JPEG loseless compression which leads to an aproximate compression ratio of 2.5:1 instead of 2:1 when using JPEG hierachical encoding and loseless compression.

Once the partial mipmap levels and the residual errors are computed, they are written into a file in a compact format which allows an optimized and progressive access to this file. The file first contains a header describing the resolution of the highest mipmap level, the number of components per pixel and the number of mipmap levels. When reading the file, one must compute the resolution for each mipmap level using the header values. Note that the number of levels allows to encode a lowest mipmap level of any resolution. If the lowest encoded level is not reduced to one pixel, the lower mipmap levels are computed by the client with the Equation (1) after having downloaded the lowest level. After this short header, the lowest level is entirely written into the file using a line major order. This level is then followed by a set of groups, each one containing the data needed to reconstruct one mipmap level. These groups are written into the file starting with the lowest level and ending with the highest one. Each group starts with the array of residual errors needed to reconstruct the level, followed by the array of color components that make up the partial mipmap level.

5 ACH Preprocessing

Recall that the architectural scenes are first subdivided into a set of cells for which a cell-to-cell visibility relationship is precomputed. For each cell, the ACH values associated with its potentially visible cells are computed by summing a set of ACHs samples computed for different points of the cell. An ACHs sample is computed in screen space by rendering the potentially visible cells for six cameras having the same center of projection (COP). The view direction of each camera is perpendicular to one face of a box. Such a box will be called rendering box from now on. The COP shared by the six cameras is the center of the rendering box and the FOV (field of view) of each camera is equal to 90 degrees. The projection plane of a camera is a face of the rendering box. In our implementation we generally use eight ACHs samples per cell whose COPs are uniformly distributed inside the cell.

For each camera of the rendering boxes of a cell, all the objects of its PVS are rendered. In order to accelerate the rendering we use an OpenGL graphics hardware and we perform a frustum culling on the bounding box of each object. Each object is displayed with a unique color which is assigned to the memory pointer pointing to its parent cell. So the contents of the image directly gives the memory pointers to all the cells visible from the COP of the associated camera. Then, for each camera C_i (of the N_c cameras associated with a cell), we count the number of pixels N_{ij} covered by each visible cell I_j. The total number of pixels N_j^{total} covered by a visible cell I_j is then:

$$N_j^{total} = \sum_{i=1}^{N_c} N_{ij}$$

Let N_{cells} be the number of visible cells for the N_c cam-

eras. The total number of covered pixels N_{pixels}^{total} for the N_c cameras is then:

$$N_{pixels}^{total} = \sum_{j=1}^{N_{cells}} N_j^{total}$$

The ACH (Average Coverage Hint), denoted ACH_j, associated with each visible cell I_j is computed as:

$$ACH_j = \frac{N_j^{total}}{N_{pixels}^{total}}$$

The properties of the ACH values are the following:

$$\begin{cases} \forall j \in [1, N_{nodes}], ACH_j \in [0, 1] \\ \sum_{j \in [1, N_{nodes}]} ACH_j = 1 \end{cases}$$

6 Visualization

Once the visibility and the ACHs have been computed, the database (described in VRML97) is stored on the server using a root file, a set of cell files and the set of PTM files. The root file contains a set of viewpoints from which the navigation can start and each viewpoint refer to the file url of the cell in which it is placed. Each cell file contains the description of one cell: its convex hull description, its child geometry, the Url list of its adjacent and potentially visible cells and the list of their associated ACH values.

At the begining of the navigation, the clients downloads the root file and select one of the proposed viewpoints. Once selected, the cell associated with the view point is downloaded as well as its potentially visible cells. At this point, all the texture map nodes (PTM nodes) that are used by a shape contained in one of the cells are registred in a table used to manage the resident nodes. A resident node is a node that manipluates some data that are stored in the OpenGL graphics hardware (i.e. texture maps or display lists). When the user moves inside the scene, its trajectory is extrapolated to find and prefetch the next visited cell. When the current cell changes, all the PTM nodes that are not used any more are removed from the resident nodes table and their associated mipmap levels are removed from the graphics hardware memory. Then, the new PTM nodes are added to this same table.

The rendering algorithm is divided into three consecutive steps. In the first step, the PVS of the current cell is analyzed to compute the set of visible objects. This is performed by the nodes themselves and is implementented in their *computeVS* method that takes the rendering options as parameters. When an object finds its bounding box to be inside the view frustum, it registers into a display table. In the second step, the *refreshResident* methods of the resident nodes are invoked so that they can refresh the graphics hardware memory according to the parameters computed during the current PVS analysis. Finally, in the third step, the *display* methods of the objects stored into the rendering table are invoked to perform the rendering.

6.1 Memory budget allocation

During the first step, a frustum culling is first performed using the convex hulls of the cells within the PVS of the current cell. The ACHs are then used to make the visible cells share out the memory budget M^{mem} that gives the total amount of graphics hardware memory that can be used for storing the texture mipmaps. The value for the M^{mem} parameter is either user defined, or benchmarked for the given graphics hardware. For each visible cell (including the current cell), its ACH is normalized using the number of cells that are visible. We compute the memory budget M_i^{mem} to assign to each visible cell i using its normalized ACH denoted ACH_i as follows: $M_i^{mem} = ACH_i \cdot M^{mem}$. Each visible cell shares out its budget among its child nodes that use it to allow their child PTM nodes to select their best suitable highest mipmap level, and returns M_{used}^{mem} which is the exact amount of memory used to store all the mipmap levels referred to by its child nodes. The budget of memory available for the next visible cell is now $M^{mem} - M_{used}^{mem}$. This process is repeated for each visible cell, starting from the cell having the highest ACH and ending with the one having the lowest one.

6.2 PTMs update

Recall that the *computeVS* method of a PTM is invoked by its parent object if this latter is visible. Since a PTM node can be shared out among different objects, it does not update its mipmap levels each time its *computeVS* method is invoked. Instead, it saves the memory budget allocated to it at each call to this method according to the following rules. Each time the *computeVS* method is invoked, the PTM node has to decide if it keeps or not the memory budget N_{alloc}, given as a parameter, to store its own mipmap levels. Let L be the index of the current highest mipmap level and N the current total amount of memory budget saved during the previous calls to the *computeVS* method of the PTM node. If the level $L+1$ exists and its memory size $S(L+1)$ is higher than N+N_{alloc}, the memory budget N_{alloc} is added to the current amount of memory N otherwise the unused memory is returned to the parent node. If the level $L+1$ does not exists, the same tests are performed using the current highest mipmap level L.

Then, during the resident node update, the *refreshResident* method uses the total amount of memory budget N to add/remove a mipmap level into/from graphics hardware memory or to download a new level. Let L_{max} be the highest level already downloaded and L_{total} is the total number of levels that are stored on the server side. If $S(L+1) \leq N$ and $L+1 \leq L_{max}$ and $L+1 \leq L_{total}$, the level $L+1$ is added into the graphics hardware memory. Otherwise, if $S(L) > N$ the level L is removed from the graphics hardware memory. Else, if we are not downloading a level and $S(L+1) \leq N$ and $L+1 > L_{max}$ and $L+1 \neq L_{total}$, a request for downloading the next level has to be sent.

In order to prevent the client from waiting a long time

for a high texture level when the network bandwidth is not high enough, the user can set the parameter δt_{max} that represents the maximum amount of time that should be used to download a level. Using the result of network bandwidth and latency analysis performed for some recent downloadings as well as the size of the requested level, the PTM computes an estimation of the time needed to download the required level. If the estimated time is lower or equal to δt_{max}, the request is sent to the server, otherwise it is discarded. Finally, two other parameters δt^{static}_{max} and M^{mem}_{static} are used if the viewpoint is static for a given number of seconds t_{static} which is a user defined parameter. For example, one can use $M^{mem} = 4MBytes$, $\delta t_{max} = 0.5sec$, $M^{mem}_{static} = 16MBytes$ and $\delta t^{static}_{max} = 20sec$ to obtain a good interactivity when moving the viewpoint and a good quality when focusing on a given object.

7 Results

In this section we provide some results showing the performances of our navigation system regarding the quality of the network transmissions and interactivity during navigation. In the following sections, the tests will be performed using a museum model because of its low number of polygons and its large amount (97.7MB) of high resolution (1024x1024) texture maps.

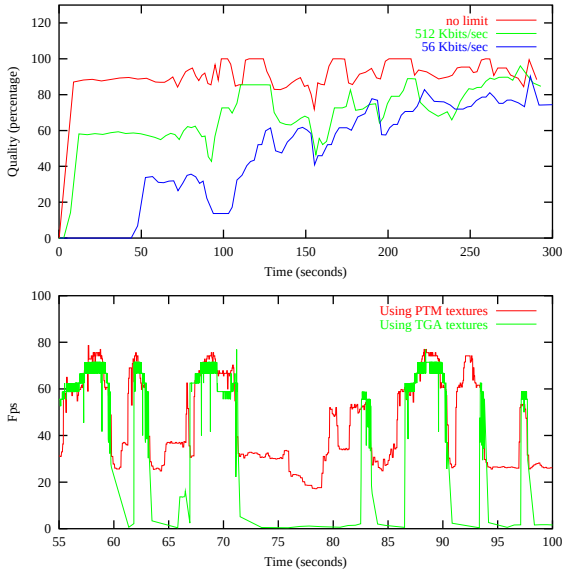

Figure 2. Top: PTM downloading quality and bandwidth adaptation over time. Simulations performed at three different bandwidth limitations. Bottom: Frame rate over time for walkthroughs of museum using high resolution TGA texture maps and PTM texture maps. The tests were performed at a rendering resolution of 1280x1024.

7.1 Transmission quality

In order to analyze the transmission quality, we have recorded a walkthrough path on the client side. This path passes through all the rooms of the museum model. Our system allows to simulate a remote walkthrough, using this path, at different bandwidths. During the simulation, for each used texture map within the PVS of the current cell, we count the amount of downloaded levels that we normalize using the highest number of levels for these textures. This percentage is called progressive downloading quality. The highest this value, the highest the progressive downloading quality. Figure 2 shows how the progressive downloading quality evolve over time for the walkthrough simulated at different bandwidths. The quality plot exhibits very low values at the begining of the walkthrough because the cells (and the geometry) downloading requests are assigned a priority higher than that of mipmap level requests. Then, after this initialisation time we can see that the quality increases rapidly up to a bound controlled by the maximum download latency parameters δt_{max} and δt^{static}_{max}. On the quality plot corresponding to the simulation with no bandwidth limitation, a quality of 100% is reached when the viewpoint is static. We can see on the two other plots that the quality also increases when the viewpoint is static. Figure 3 shows three screen shots for a static viewpoint at different times using a simulated bandwidth of $56Kbits/s$ and one screen shot where texture maps are in the highest resolution, using a local server. The framerate is higher than $29fps$ while it equals $2.7fps$ when the PTM mechanism is not used.

7.2 Interactivity

Recall that our system relies on two parameters M^{mem} and M^{mem}_{static} respectively used to limit the texture memory budget for the graphics hardware for a moving and a static viewpoint respectively. To show that our system is capable of rendering scenes with high resolution texture maps at an interactive frame rate, we have simulated a walkthrough using a path with many direction an rotation changes of the viewpoint. Such a path induces many transfers on the AGP bus if the amount of texture maps is too high for the used graphics hardware. For this test, the walkthrough was performed twice for two different models. The first model is the original museum scene with high resolution TGA texture maps and the second one is the preprocessed museum using PTM texture maps. To walk through the second model, we did not use any network bandwidth limitation and the memory bounds were set to $M^{mem} = 4MB$ and $M^{mem}_{static} = 12MB$ which are the best parameters for the Pentium IV 1.7GHz, with a NVidia Quadro 2 Pro 64MB we used for the tests. The first time, the walkthroughs are simulated such that all the needed data are downloaded with the maximum resolution and the TGA texture maps are all loaded into the graphics hardware memory. Then, the second time all the needed data for the walkthroughs

Figure 3. Screen shots from a static viewpoint placed in the museum at different times after the texture downloadings start. Top left: $10s$ after, at $56Kbits/s$. Top right: $35s$ after, at $56Kbits/s$. Bottom left: $1m20s$ after, at $56Kbits/s$. Bottom right: view of highest quality $3s$ after the texture downloadings start using a local server.

are kept into the main and graphics hardware memories. Figure 2 shows the frame rate (fps) analysis for the second time walkthroughs. The fps plots show that between 72 seconds and 83 seconds the frame rate for the walkthrough using TGA texture maps is very low because of the AGP bus bottleneck whereas the frame rate for the walkthrough using PTM texture maps ranges beteween 20fps and 50fps. Consequently, our system allows real time visualization of scenes that could not be visualized in a usual manner.

8 Conclusion

In this paper we have described a system that allows to visualize large 3D scenes that can contain a large amount of high resolution texture maps. To obtain such results we have proposed a compact and progressive representation for texture maps that is fast to encode-decode using integer arithmetic. The precomputation of the ACH values we use to select the level of a PTMs to be downloaded and uploaded into the graphics hardware memory is fast since it is performed by the graphics hardware. Thanks to this mechanism and a small set of parameters, we can ensure an automatic adaptation to the network and to the client machine performances. Our ACH based visualization system is highly scalable, modular and can handle any kind of texture maps.

References

[1] J. Airey, J.H. Rohlf, and F.P. Brook. Toward image realism with interactive update rates in complex virtual building environements. In *Symposium on interactive 3D graphics*, 1990, 41-50.

[2] R. Dumont, F. Pellacini, and J.A. Ferwerda. A perceptually-based texture caching algorithm for hardware-based rendering. In *Proc. 12th Eurographics Workshop on Rendering*, 2001, 246-253.

[3] T.A. Funkhouser. Database management for interactive display of large architectural models. In *Graphics Interface*, 1996, 1-8.

[4] T.A. Funkhouser, C.H. Squin, and S.J. Teller. Management of large amounts of data in interactive building walkthroughs. In *Symposium on Interactive 3D Graphics*, 1992, 11-20.

[5] A. Said and W.A. Pearlman. A new fast and efficient image codec based on set partitionning in hierarchical trees. In *IEEE Trans. Circuits and Systems for Video Technology*, 6(3), 1996, 243-250.

[6] J.M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. In *IEEE Trans. Signal Processing*, 41(12), 1993, 3445-3462.

[7] E. Teler and D. Lischinski. Streaming of complex 3D scenes for remote walkthroughs. In *Computer Graphics Forum*, 20(3), 2001, 15-25.

[8] S.J. Teller and C.H. Sequin. Visibility computations in polyhedral environments. Technical report, University of California at Berkeley, 1992, CSD-92-680.

[9] S.J. Teller and C.H. Squin. Visibility preprocessing for interactive walkthroughs. In *Proc. ACM SIGGRAPH*, 1991, 61-69.

[10] L. Williams. Pyramidal parametrics. In *Proc. ACM SIGGRAPH*, 1983, 1-11.