The 10th International Symposium on Virtual Reality, Archaeology and Cultural Heritage VAST
(2009), pp. 1–9
C. Perlingieri and D. Pitzalis (Editors)

Interactive Remote Exploration of Massive Cityscapes

sub ID 1018

Abstract

*We focus on developing a simple and efficient unified level-of-detail structure for networked urban model viewers. At the core of our approach is a revisitation of the BlockMap [CBG*07] data structure, originally introduced for encoding coarse representations of blocks of buildings to be used as direction-independent impostors when rendering far-away city blocks. The contribution of this paper is manifold: we extend the BlockMap representation to support sloped surfaces and input-sensitive sampling of color; we introduce a novel sampling strategy for building accurate BlockMaps; we show that BlockMaps can be used as a versatile and robust way to parameterize the visible surface of a highly complex model; we improve the expressiveness of urban models rendering by integrating an ambient occlusion term in the representation and describe an efficient method for computing it; we illustrate the design and implementation of a urban models streaming and visualization system and demonstrate its efficiency when browsing large city models in a limited bandwidth setting.*

1. Introduction

Real-time 3D exploration of remote models of our environment has long been one of the most important applications of distributed real-time graphics. Even though, historically, these tools have focused on textured digital elevation models, the interest is now rapidly shifting towards urban environments. Focusing on such environments is extremely important, since a large part of Earth's population lives and works in dense urban areas, and much of our cultural heritage revolves around complex cityscapes. Moreover, recent advances in city acquisition and modeling (e.g., specialized 3D scanning [TAB*03, FJZ05, CCG06], assisted reconstruction from photos and video streams [PVGV*04], or parametric reconstruction [PM01, MWH*06]) lead to a rapidly increasing availability of highly detailed urban models for both modern and ancient cities.

Exploring large detailed urban models, seamlessly going from high altitude flight views to street level views, is extremely challenging. What makes urban models so peculiar is that their geometry is made of many small connected components, i.e., the buildings, which are often similar in shape, adjoining, rich in detail, and unevenly distributed. While multiresolution texturing and geometric levels of detail may provide acceptable solution for buildings near to the viewer and moderate degrees of simplification, major problems arise when rendering clusters of distant buildings, hard to faithfully and compactly represent with simplified textured meshes. Therefore, state-of-the-art solutions propose switching to radically different, often image-based, representations for distant objects having a small, slowly changing on-screen projection. This dual representation approach, however, increases implementation complexity, while introducing hard to manage bandwidth bottlenecks at representation changes.

In this work, we focus on developing a simple and efficient level-of-detail structure that can serve as a basis for an Internet urban model viewer. To reach this goal, we enhance the *BlockMap*, a GPU- friendly data structure for encoding coarse representations of both geometry and textured detail of a small set of buildings [CBG*07]. BlockMaps are stored into small fixed-size texture chunks, and can be efficiently rendered through GPU-based raycasting. However, BlockMaps have been originally designed to replace the geometry-based representation only when the single block of buildings represented by the BlockMap projects on the screen to just a few tens of pixels. Being tuned for this situation, the BlockMap geometry and color encoding is very coarse, basically a set of discretized textured vertical prisms, and exhibits aliasing artifacts when used in close range views. For this reason, a hybrid multiresolution framework had to be employed in [CBG*07], where BlockMaps are used together with a textured polygons representation [BD05]. The contribution of this paper is manifold:

- we extend the BlockMap representation to support sloped surfaces and input-sensitive sampling of color, making it an all-round block of buildings representation;

- we introduce a novel and more elegant sampling strategy that allow us to build more accurate BlockMaps;
- we show that BlockMaps can be used as a versatile and robust way to parameterize the visible surface of a highly complex model;
- we improve the expressiveness of urban models rendering by integrating ambient occlusion in the representation and describe an efficient precomputation method;
- we illustrate the design and implementation of an urban models streaming and visualization system;
- we demonstrate the performance of the approach with the interactive remote visualization of large textured ancient and modern urban environments accessed through limited bandwidth network connections.

Although not all the techniques presented here are novel in themselves, we believe their elaboration and combination in a single unified system is non trivial and represents a substantial enhancement to the state-of-the-art. Using the proposed approach, we can produce a scalable multiresolution representation of a large urban model that can be interactively transmitted and visualized over the network. The overall approach is focused on a particular, but very important, data and application class and should not be intended for the visualization of general 3D datasets. Even though this fact limits the general applicability of the method, such a targeted focus allows us to produce a particularly simple, compact and efficient model and system.

2. Related work

We provide a rapid overview of the approaches more closely related to ours. For a recent survey on the problem of rendering large environments we refer the reader to [YGKM08].

Adaptive streaming and rendering. Massively textured urban models require the management of very large amounts of textures. The classic solution for managing large textured urban models is the texture-atlas tree [BD05], which, however, considers multi-resolution textures but single resolution geometry, and does not take into account network streaming issues. At the core of our approach is a compact hierarchical representation of both geometry and texture of urban models. Using a simpler geometric representations for far geometry is the core idea of *LOD* techniques, while switching to a radically different image based representation characterizes *impostor* approaches. Impostor-like techniques, introduced a decade ago (see, e.g., *Textured Depth Meshes* [SDB97,DSV97], *Layered Environment-map Impostors* [JWS02,JW02], and *Layered Depth Images* [SGwHS98, WWS01]), are enjoying a renewed interest, because of the evolution of graphics hardware, which is more and more programmable and oriented toward massively parallel rasterization. The *Relief Mapping* approach was introduced in [OBM00], where a warping approach is used to find the final position of an orthogonally displaced texel over a given flat texture. An approach to the problem of rendering generalized displacement mapped surfaces by GPU raycasting was proposed in [WWT*03, WTL*04]. In these methods, the results of all possible ray intersection queries within a three-dimensional volume are precomputed and stored in a compressed five-dimensional map. While general, this approach implies a substantial storage overhead. Other generalizations involve replacing the orthogonal displacement with inverse perspective [BD06], replacing the texture plane with a quadric [MM05], handling self shadowing in general meshes [POC05]. In [ABB*07] the scene is approximated with a cloud of relief maps, renamed *omnidirectional relief impostors*, and a GPU raycasting is done for each frame on the subset of such cloud which is optimal with respect to the point of view. The approach was subsequently implemented in a hierarchical fashion and tailored to urban dataset visualization [AB08]. In this work, we follow instead the BlockMap approach [CBG*07] of exploiting a small number of precomputed maps to efficiently encode and render urban models as a set of compactly encoded textured prisms. This representation is more similar to LODs than to impostors, since it encodes a discretization of the original geometry. The original simple encoding, however, limits its applicability to only coarse levels of details.

Parameterization and perceptual improvements. Since we aim to enhance shape perception during visualization, we include in our level-of-detail structure information that allows us to adopt a shading model based on the Ambient Occlusion term [Lan02], which approximates a uniformly diffusing lighting environment. From a perceptual point of view this choice has a grounded foundation, as there is evidence that it improves shape perception over direct lighting [LB99]. In the context of geo-viewing, non-local shading approaches to enhance image readability have been applied in terrain visualization [Ste98], and off-line city model rendering [WMW07], but not in an interactive streaming and rendering context. It should be noted that precomputation of an ambient occlusion term or more sophisticated radiance transfer information [KLS05] require either a good uniform texture parametrization of the surface or a good meshing of the whole scene, something that usually is difficult to be achieved for general large models like the one used for urban environments. Encoding shape and appearance into a texture is the goal of *geometry images* [GGH02], which enable the powerful GPU rasterization architecture to process geometry in addition to images. Geometry images focus on reparametrizations of meshes onto regular grids, while we focus on a specialized representation of urban structures.

3. The *BlockMap* concept

The BlockMap representation has been introduced as a way to compactly represent a small city portion (i.e. a group of nearby buildings) as seen from a distance. In the following we briefly review the concepts behind it. Please refer to the original paper [CBG*07] for a more detailed description.

The BlockMap representation stores in a rectangular portion of texture memory all the data required to represent a

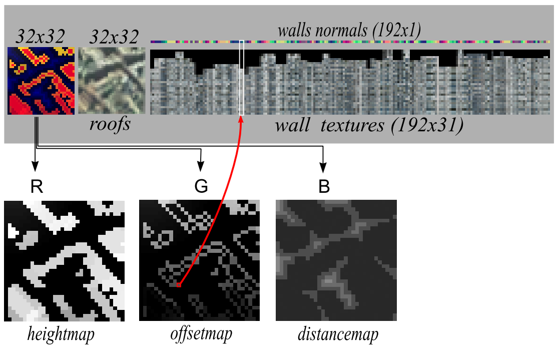

Figure 1: *A single node of the BlockMap data representation encodes the geometry and texture detail of a small set of nearby buildings in a small texture chunk.*

group of textured vertical prisms defined and discretized on a square grid (see an example in Figure 1). The left-most 32x32 texture region gives a top view of the represented region and stores with each color channel: the *heightmap* (it encodes explicit discretized geometry information on buildings heights, and implicitly the discretized buildings plans), the *offsetmap* (storing for each texel the u coordinate of the corresponding column in the right most portion of the BlockMap, where textures for the vertical sides of the prisms and relative normals are stored), and a *distancemap* (used to speed up GPU raycasting of the BlockMaps by space leaping). Resolutions mentioned in Figure 1 refer to the BlockMap version used to render block of buildings far from the viewpoint, as proposed in [CBG*07]. In this representation, all the roofs are assumed to be flat, or considered as such when seen from a distance. Moreover, the amount of texture stored for each of the prisms is fixed, and thus not depends on a prism height.

4. A unified approach for far and near geometry

The advantage of the BlockMaps is that they encode the large scale features of a urban-like dataset, i.e. the vertical flat walls of the buildings, with little memory footprint. Thanks to these characteristics, the BlockMaps have been successfully integrated in a hybrid multiresolution scheme employing them for the coarse level of the hierarchy and textured meshes for the fine ones. While this dual representation approach is satisfactory when used locally, it is not as much suitable for remote browsing of large cities, because, ultimately, a large amount of geometry and texture will need to be transmitted for the textured geometry in the near field. Due to constraints in semi-automatic urban acquisition methods, the very large scale urban databases targeted by internet viewing, consists of many small connected components (single or small group of buildings), each one represented with a relatively small number of polygons but a large amount of color information, typically one or more photographs for each façade of the building. While the original BlockMap representation is too crude for close views, a full unstructured mesh representation can be considered an overkill. We therefore propose to improve the BlockMaps in terms of geometry content and visualization quality at the point that they can be used both for far-away and for near geometry, and to build a multiresolution representation entirely based on this data structure. To reach this goal, we extend the BlockMap representation to support sloped surfaces and input-sensitive sampling of color, we introduce a novel and more elegant sampling strategy that allow us to build more accurate BlockMaps, and we improve the expressiveness of urban models rendering by supporting ambient occlusion. Figure 2 illustrates our new blockmap datastructures. The rest of this section details the modification made to the blockmap datastructure. The next section illustrates an efficient method for attribute sampling.

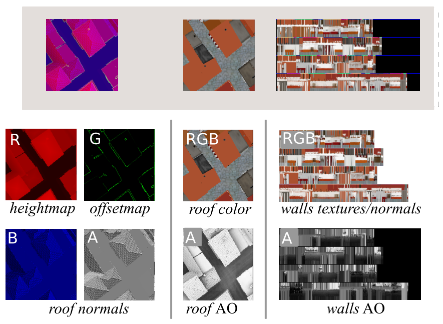

Figure 2: *Improved version of BlockMap data structure.*

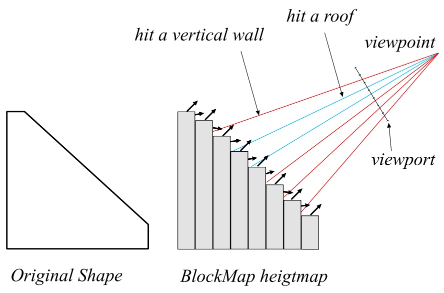

Figure 3: *Left: original shape. Right: BlockMap encoding. When a vertical wall is hit, the previous texel along the ray direction is tested to estimate slope.*

Sloped rooftops. One of the assumptions taken in the blockmap representation is that the roof's shape of most buildings is negligible in far distance views. If a BlockMap covers a small region of the domain, a single rooftop spans over more than one texel. In the case shown in Figure 3, for instance, a sloped rooftop of a building (left most picture) is sampled to produce a heightmap (right most picture). When ray-casting this BlockMap, some rays will hit the sides of the prisms (the red rays in the figure) and others will hit the rooftops (the cyan rays). As a result, the sloped rooftop silhouette will have a step-like appearance, and shading will incorrectly ignore slopes. We overcome this problem with two modifications. First, instead of considering the normal for the rooftops as constantly pointing upward, we sample the normal from the geometry and store it in the BlockMap (see *map* in Figure 2) to obtain a corret lighting. We also modify the ray-casting algorithm to compute the partial derivative of the height value along the ray projection into the ground and to use it to distinguish walls from the rooftops. More precisely, if the computed derivative is smaller than the maximum slope a roof can have, the ray is considered to have hit a roof and the values for the shading are chosen accordingly. In the example in Figure 3, all the rays shown in red hit a vertical wall but, because of the detected small increment with respect to the previous texel, the normal and color are fetched from the *roof color* and *roof normal* respectively.

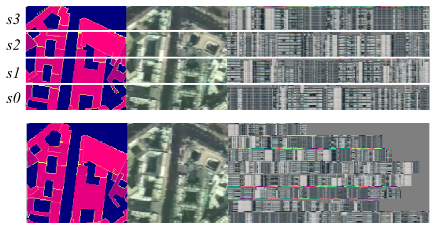

Figure 4: *A BlockMap sliced in four parts, each one with its own parameterization. In this example the alpha channel of the BlocMap has been removed for clarity.*

Adaptive parametrization by slicing BlockMaps. In typical urban datasets, the ratio between the total perimeter of a block of buildings and their maximum height varies from block to block, typically increasing with the number of buildings. In contrast, in the BlockMaps the number of texture samples in the horizontal and vertical direction is fixed. The parameterization made by the BlockMaps (i.e., the mapping from the boundary texel to the *wall textures/normals*) assigns the same number of vertical samples to all the prisms, independently from their height, leading to over- and undersampling. We greatly improve sampling by supporting a variable perimeter/height ratio within the same BlockMap. Instead of unrolling all the building boundaries in a single strip of columns, we subdivide the BlockMap in slices of equal size and parameterize each slice individually. Figure 4 shows an example of a 4-sliced BlockMap. Note that we could, alternatively, use BlockMaps of different sizes to adapt to the characteristics of the specific region of the model. Doing so would, however, lose the advantage of having memory chunks of fixed size when handling fetching and chaching of data at rendering time. By slicing the BlockMaps, we conserve all the advantages of fixed-memory management, and only have to make a minor modification to the ray-casting algorithm. We recall that when a ray hits the vertical side of a prism, the u coordinate used to fetch its color is taken from the *offsetmap*, while the v coordinate is given by the height of the intersection point. The only modification needed to support sliced BlockMaps is to change the computation of the v coordinate, by scaling and translating the height value depending on the slice s_i the texel is in. We choose to slice the BlockMaps regularly, so that the value s_i can be quickly computed by the ray casting algorithm without having to store it. The number of slices in a BlockMap is chosen to provide a perimeter/height ratio as close as possible to the value found in the original dataset. The construction algorithms repeatedly constructs BlockMaps with an increased number of slices until a satisfactory value is found. For each block, we start trying to build a single-slice BlockMap. If the number of boundary texels exceeds the number of available columns, we try with 2 slices and so on, until either no slice exceeds the number of columns or the maximum number of slices has been reached. In a $m \times n$ BlockMap, this number cannot exceed $n/2$, with a slice made of exactly one row of colors and one row of normals. However, we experimentally determined that having at least 3 texels/slice provides better results, so we stop slicing when the number of slices is $n/4$.

Ambient Occlusion. The original blockmap stores as material attributes a single diffuse color and a normal, forcing the real-time renderer to use a local shading model. Since we aim to enhance shape perception during visualization, we include in our level-of-detail structure information that allows us to adopt more a sophisticated shading model that approximates the light coming from an uniformly diffusing lighting environment. Ambient occlusion is a technique to account for the accessiblity of the surface points in the local shading models. It consists of computing the percentage of the hemisphere above each surface point not occluded by geometry and to use this value to weight the ambient term of the lighting equation [Lan02]. Given the fact that our approach offer an unique parametrization of each visible point of the surface, the modification to the BlockMaps data structure to include ambient occlusion term is quite simple: it consists of storing its value in the alpha channel (not used) of the texture color. Figure 5 show the same view with or without using ambient occlusion.

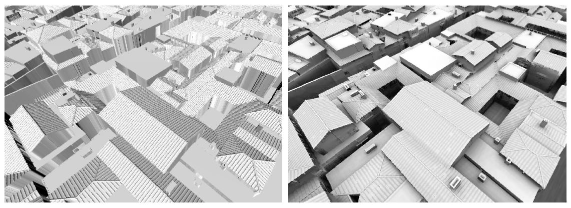

Figure 5: *Close view of a block without (left) or with (right) ambient occlusion.*

5. Multi-view sampling of shading attributes

The sampling of shading attributes for vertical walls is the most critical step in the process of building a BlockMap. In the original BlockMap version, for each visible side of a prism, a texture column was computed by simply performing orthographic renderings of the correspoding textured geometry and copying the result in the assigned texture column. This simple process potentially leads to two kinds of sampling artifacts. We illustrate the problem with an example.

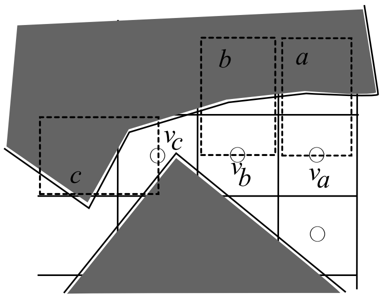

Figure 6: *Top view of few pixels partially included in the boundary of two buildings. The drawing shows a manifactured example of how the sampling process may generate rendering artifacts.*

Sampling strategies. Figure 6 shows three boundary texels and respective columns. The geometry inside texels a and b is sampled independently by taking orthogonal renderings from positions v_a and v_b. These renderings can be performed with arbitrarily large viewports, but the result will ultimately be resampled to at most one-pixel-large adjacent columns, causing a resampling artifact at the boundary between the columns. A more severe issue can be observed in the sampling of the texel c. If the rendering is done from the position v_c (the center of the adjacent texel), a whole portion of the geometry will be missed. On the other hand, placing the point of view further away would include unwanted geometry from texel d. To solve this problem we would need to flatten the geometry with respect to the sides of the boundary texels. In this work, we develop a more elegant and effective sampling strategy. We take views of the whole geometry from uniformly distributed directions in the hemisphere centered at the BlockMap center, and reproject the result of each view onto the BlockMap's prisms. Since each surface point can be seen from several directions, the final value is obtained by gathering all the contributions and combining them to a single value per point.

Implementing multiple-view sampling. We developed a GPU accelerated technique that harnesses the performance and programmability of current graphics hardware to efficiently perform multi-view sampling. The sampling process is performed by iterating the followig steps for each view:

1. render the geometry from the viewpoint with an orthogonal projection to a $n \times n$ texture (called *GeoBuffer* in Figure 7);
2. ray-trace the BlockMap with the same viewing settings, targeting rendering to a $n \times n$ texture (called *Address-Buffer* in Figure 7); whenever a ray hits a prism, the corresponding address is written to the data that should be fetched for shading, i.e. a pointer to a texel of the *roof* or to a column in the *wall textures*.
3. set the BlockMap as rendering target and draw $n \times n$ point primitives, one for each pixel of the *GeoBuffer*. In the vertex shader, the color corresponding to pixel i, j is fetched from the *GeoBuffer*, while the vertex position is read from the *AddressBuffer* at the same coordinates.

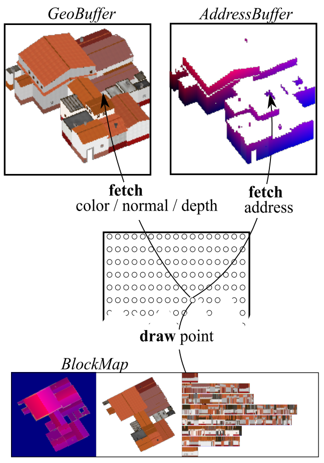

Figure 7: *Scheme of the computation of the contribution of a single view to the BlockMap construction.*

The result of these steps is to build the BlockMap that, when ray-traced with the same viewing parameters, would produce the same result as rendering the geometry. This approach is used to sample all the shading attributes in the blockmap, i.e., *color*, *normal* and *accessibility*. For the color, the rendering of geometry at step 1 is done with texture mapping alone (i.e. without lighting), while the value written in the BlockMap at step 3 is blended with the value possibly present in the BlockMap because of other samples fallen in the same texel (either in the same or in an earlier view). The same is done for the normals, encoded as color for the sampling. In this manner, colors and normals stored on each texel of the BlockMap are averaged among all the views from which the corresponding portion of surface is visible. For the accessibility, we also need the geometry outside the BlockMap because it may occlude part of the geometry inside the BlockMap. Here, we conservatively assume to use the entire geometry as potential occluder, although we verified that, in practical cases, using only the geometry closer than a fixed distance also leads to good visual results. Step 1 is modified by rendering all the potentially occluding geometry and saving the depth buffer. Step 3 is modified by checking, for each point, is the depth value of the corresponding fragment in the GeoBuffer is greater than the depth value stored at step 1. If not, the contribute of the point is ignored (the sample is occluded), otherwise the value written into the BlockMaps is the value already present plus one. With this approach, the final value per texel corresponds to the number of views from which the corresponding portion of the surface is visible. The accessibility is then obtained by dividing that result by the total number of views.

6. The remote rendering architecture

The major application of our system is networked browsing of very large urban environments. We implemented a prototype client-server architecture enabling multiple clients to explore city models stored on a remote server. Since our enhanced BlockMaps are able to provide approximate representations of blocks of buildings which are visually valid for a wide range of viewpoints, we can create a full quadtree hierarchy of levels of detail by associating BlockMaps to larger and larger areas. Thanks to the constant footprint of BlockMaps, memory management is particularly simple and effective. In particular, no fragmentation effects occur throughout the memory hierarchy, and data transfers at all levels can be optimized by grouping BlockMaps for tuning message sizes. Upon connection, the node hierarchy structure is transmitted to the client, which stores it in main memory. The hierarchy (few Kilobytes) is the only data structure permanently kept in client memory, while all the BlockMaps reside only on the server and are transmitted only on demand. Each client maintains a 2-level cache of recently used BlockMaps in video RAM and local storage, and requests data to the server only at cache misses.

The main client thread, called *RenderThread* from now on, is responsible for visiting the hierarchy to determine the level-of-detail required for all parts of the scenes and performing the rendering. Other two threads, *VRAMThread* and *NetworkThread*, transparently handle the BlockMap requests made by the RenderThread to the remote data server. The VRAM cache is organized as an array of fixed size chunks of texture memory, where the BlockMaps to be rendered are copied from local storage. The *RenderThread* works in three stages: *RequestingVisit*, *LockingVisit*, and *Rendering*.

In the *RequestingVisit* stage, the RenderThread performs a top down traversal of the hierarchy according to the current viewing parameters and issues a request for each visited node. If the node is present in the local database, then the request is queued to the *VRAMRequestsQueue*, meaning that the BlockMap is waiting to be loaded in VRAM, otherwise the request is queued in the *NetworkRequestsQueue*. In order to prioritize requests, we distinguish among different request classes: class **A** requests, if the requested node is inside the view frustum, class **B** requests if the node is not inside the view frustum but is a neighbor of a class **A** node, or class **C** requests if its error is below the accepted error. Note that requests of class **B** and **C** define the neighborhood of the set of nodes required for rendering the current view.

In the *LockingVisit* stage, the RenderThread performs a second top-down traversal stopping at nodes of class **A**. The goal of the visit is to lock all the nodes that satisfy the error criterion and add them to the *RenderList*, i.e. the list of nodes that will be rendered in the current frame. A node can be locked if the associated BlockMap is in the VRAM. Locking a node sets a flag that prevents its unloading. The visit is implemented as a recursive procedure which starts by trying to lock the current node. If locking is successful, but the error is above the user specified threshold, then children are visited. If the error is within the threshold, the node is added to the *RenderList*. In case all children of a locked node have been locked, their parent is unlocked. Otherwise, the region occupied by the children that cannot be locked will be drawn using the corresponding portion of the parent node, as show in Figure 8. This is done by inserting the parent node in the *RenderList* with a code that specifies which of the four portions of the BlockMap must be used for rendering.

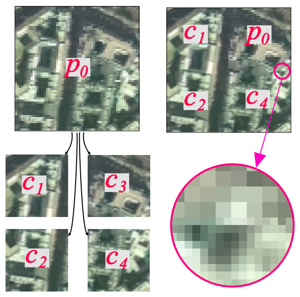

Figure 8: *On the left side an example of a node with its four children (top-view). In the right side the representation used when only nodes c_1, c_2 and c_3 can be locked, with a zoomed image of the border region between the two resolutions.*

The final stage is *Rendering*, which simply consists of scanning the *RenderList* and sending the BlockMaps to the GPU raycasting process. To minimize the number of texture switches, the *RenderList* is ordered in buckets on the base of the texture id containing the BlockMap and processed front to back within each bucket, thus reducing pixels overdraw and enabling the use of occlusion strategies (see [CBG*07]).

Prioritization of requests is crucial for interactive rendering. Both the requests to load BlockMaps from local storage to the VRAM and from the network to local storage are kept in dedicated priority queues (*VRAMResquestsQueue* and *NetworkRequestsQueue*, respectively). The comparison operator to sort the requests is implemented by combining three criteria. The first criterion is the *time* the request is issued. At every frame all the necessary nodes that are not present in local memory (in VRAM) are enqueued in the *NetworkRequestsQueue* (*VRAMResquestsQueue*), while the queue is purged of the requests older than a predefined *life time*. This means that after having satisfied all the requests of the last frame are satisfied, recent unsatisfied requests can be processed. In this manner we avoid to continuously delete/reinsert in the queue the nodes close to the boundary of the view frustum. If the time does not discriminate the request and prefetching is enabled, the *class* of the requests is used as criterion, giving higher priority to class **A** requests.

Finally, if both previous criteria fail, which happens for all the nodes in the view frustum, the screen space error is used.

7. Implementation and results

We implemented the described system on Windows Vista platform using C++, OpenGL, and GLSL shaders. We have extensively tested our system with a number of urban models. The quantitative and qualitative results discussed here are for the Paris urban environment, which shows an example of large scale models created from cadastral maps, and the ancient Pompeii environment, representative of a smaller scale but detailed model created by procedural means. The Paris model is created from the cadastral maps, containing a vector representation of 80,414 buildings described with 3.7M triangles. The original dataset has no texture information for the building façades, so, for the sake of testing, we have created and stored for each building a different 512^2 procedural texture. Roof textures were taken from aerial photographs composed in 64 $2K \times 2K$ tiles. Overall, the texture information for the façades is composed by 20G texels (almost 60GB of uncompressed data). The Pompeii model represents a reconstructive hypothesis of the ancient Pompeii [Pro07] described by 30M triangles and 30M of PNG compressed texture data.

Preprocessing. For the Paris dataset the creation of all the geometry from the cadastral profiles, the geometric partitioning and the quadtree construction took less than a couple of minutes and generated a tree of 20,229 nodes with 15,072 leaves, with maximum depth of 10. The recursive partitioning of the tree targeted less than 10000 polygons and buildings for each leaf and a set of textures that could be arranged in a 2048×2048 atlas. The construction of the atlas-tree, used to speed-up sampling, took approximatively four hours starting from the original 80,414 512^2 facade textures. Once the data is reorganized this way, in the case of 128:512 BlockMaps used in our experiments, the creation of a tree of 3,104 BlockMaps took 140 mins.

Streaming and rendering. The reported times were obtained on a dual core Pentium IV @ 3 GHz PC equipped with 1 GB Ram, two HD 160 GB SATA and a NVIDIA GeForce 8800 GTX with 768 MB running the client application and a Pentium IV @ 3 GHz, 2 GB RAM, IDE hard disk for the server side. The goal of our tests was to show how our system can provide an efficient solution for remote visualization of large urban models. The reported results were recorded during a fly-through over the Paris model performed with a controlled bandwidth of 100, 20, 8 and 4 Mb/s on a 640×480 viewport. As expected the frame rate is is always above 50 fps, since the system is multi-threaded and the rendering cycle does not have to wait for BlockMaps to be received from the server. A more interesting data is the number of network cache misses shown in the graph in Figure 9, i.e. the number of nodes that should have been rendered according to the screen space error threshold and which has not been rendered because was not available locally.

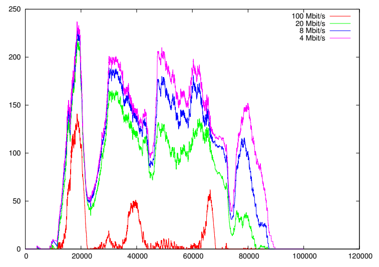

Figure 9: *The graph shows the number of cache misses during the fly-through in the accompanying video.*

By looking at the graph we can notice a peak for all of the four bandwidths, which is simply due to the fact that initially there are no BlockMaps stored locally. After the peak, at about 20 seconds from the beginning of the flythrough ($1m30s$ in the video) it can be seen how the slower clients shows much more cache misses. The ratio between the cache misses tends to be (inversely) proportional to the bandwidth only when the point of view moves at a sufficient speed, so that the set of nodes in the frustum changes rapidly, as it happens around the 50^{th} second ($2m20s$ in the video). For a normal inspection, where the user is not interested to the maximum level of detail while moving from a point to another, but only when stopping to a region of interest, the 4 Mb/s bandwidth provides a comparable result.

Rendering quality is easier to appreciate when looking at the detailed Pompeii model, which contains fine geometric features and many sloped roofs, impossible to faithfully represent with the original BlockMap approach. The attached video shows the improvement of over the original BlockMaps data structure. In the frame shown in Figure 5 it can be appreciated how the ambient occlusion term enhances the perception of small features, for example the details of the roofing, and gives the impression of jutting out roofs.

8. Conclusions

We have proposed an approach for the interactive remote visualization of large urban environments. The proposed framework, simple to implement, is based on a significant enhancement of the BlockMap structure. Our improvements include augmenting the expressiveness and quality of the representation, the readability of the visualized structure and the robustness of the conversion from general possibly ill-formed input datasets. In some sense, our approach can be interpreted as a robust and practical way to create a new textured simplified representation of a urban environment, and, therefore, as a way of finding a unique parameterization of its surface. Most of the sophisticated high quality real-time rendering techniques need a good uniform parameterization of the surface to correctly work, something that is usually quite difficult to obtain for models such as those used in city rendering application, that often exhibit a wide variety of problems, such as high variance in element size and self intersections.

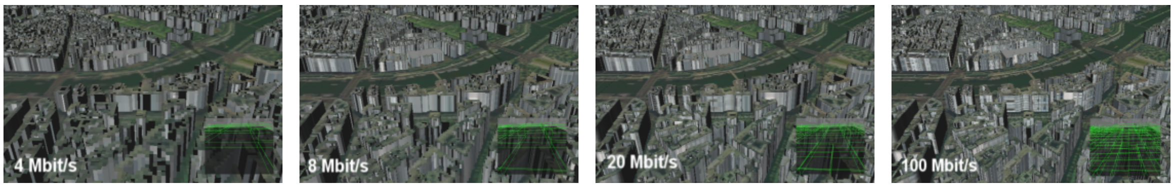

Figure 10: *A snapshot from the accompanying video showing the same flythrough as seen from four different clients with increasing network bandwidth availability.*

Our approach, as a side effect, produces a complete, multiresolution parameterization of the visible surface: in this paper we have used it to store a simple ambient occlusion term, but it is quite trivial to extend it for storing more sophisticated precomputed transfer radiance information [KLS05].

References

[AB08] ANDUJAR C., BRUNET P.: Relief impostor selection for large scale urban rendering. In *IEEE Virtual Reality Workshop on Virtual Citiscapes: Key Research Issues in Modeling Large-Scale Immersive Urban Environments*. 2008.

[ABB*07] ANDUJAR C., BOO J., BRUNET P., FAIRÉN M., NAVAZO I., VÁZQUEZ P., VINACUA À.: Omni-directional relief impostors. *Computer Graphics Forum 26*, 3 (Sept. 2007), 553–560.

[BD05] BUCHHOLZ H., DÖLLNER J.: View-dependent rendering of multiresolution texture-atlases. In *IEEE Visualization* (2005), p. 28.

[BD06] BABOUD L., DÉCORET X.: Rendering geometry with relief textures. In *Graphics Interface* (2006), Gutwin C., Mann S., (Eds.), pp. 195–201.

[CBG*07] CIGNONI P., BENEDETTO M. D., GANOVELLI F., GOBBETTI E., MARTON F., SCOPIGNO R.: Ray-casted blockmaps for large urban visualization. *Computer Graphics Forum 26*, 3 (Sept. 2007).

[CCG06] CORNELIS N., CORNELIS K., GOOL L. V.: Fast compact city modeling for navigation pre-visualization. In *Proc. CVPR* (2006), pp. 1339–1344.

[DSV97] DARSA L., SILVA B. C., VARSHNEY A.: Navigating static environments using image-space simplification and morphing. In *SI3D* (1997), pp. 25–34, 182.

[FJZ05] FRUEH C., JAIN S., ZAKHOR A.: Data processing algorithms for generating textured 3d building facade meshes from laser scans and camera images. *International Journal of Computer Vision 61*, 2 (feb 2005), 159–184.

[GGH02] GU X., GORTLER S. J., HOPPE H.: Geometry images. In *Proc. SIGGRAPH* (2002), Hughes J., (Ed.), Annual Conference Series, pp. 335–361.

[JW02] JESCHKE S., WIMMER M.: Textured depth meshes for realtime rendering of arbitrary scenes. In *Proceedings of the 13th Eurographics Workshop on Rendering (RENDERING TECHNIQUES-02)* (Aire-la-Ville, Switzerland, June 26–28 2002), Gibson S., Debevec P., (Eds.), Eurographics Association, pp. 181–190.

[JWS02] JESCHKE S., WIMMER M., SCHUMANN H.: Layered environment-map impostors for arbitrary scenes. In *Graphics Interface* (2002), pp. 1–8.

[KLS05] KAUTZ J., LEHTINEN J., SLOAN P.-P.: Precomputed radiance transfer: Theory and practice. Course Notes of ACM SIGGRAPH, 2005.

[Lan02] LANDIS H.: Production ready global illumination. In *SIGGRAPH 2002 Course Notes* (July 22-26 2002), ACM Press, pp. 331–338.

[LB99] LANGER M. S., BULTHOFF H. H.: *Perception of Shape From Shading on a Cloudy Day*. Tech. Rep. 73, Max-Planck-Institut fur biologische Kybernetik, October 1999.

[MM05] MANUEL M.OLIVEIRA F. P.: *An Efficient Representation for Surface Details*. Tech. Rep. RP 351, Universidade Federal do Rio Grande, January 2005.

[MWH*06] MÜLLER P., WONKA P., HAEGLER S., ULMER A., VAN GOOL L.: Procedural modeling of buildings. *ACM Transactions on Graphics (TOG) 25*, 3 (2006), 614–623.

[OBM00] OLIVEIRA M. M., BISHOP G., MCALLISTER D.: Relief texture mapping. In *Proceedings of the Computer Graphics Conference 2000 (SIGGRAPH-00)* (New York, July 23–28 2000), Hoffmeyer S., (Ed.), ACMPress, pp. 359–368.

[PM01] PARISH Y., MÜLLER P.: Procedural modeling of cities. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), 301–308.

[POC05] POLICARPO F., OLIVEIRA M. M., COMBA J. L. D.: Real-time relief mapping on arbitrary polygonal surfaces. *ACM Trans. Graph 24*, 3 (2005), 935.

[Pro07] PROCEDURAL, INC.: Procedural modeling of cg architecture. http://www.procedural.com/cityengine/production-pipeline/export-samples.html, 2007.

[PVGV*04] POLLEFEYS M., VAN GOOL L., VERGAUWEN M., VERBIEST F., CORNELIS K., TOPS J., KOCH R.: Visual Modeling with a Hand-Held Camera. *International Journal of Computer Vision 59*, 3 (2004), 207–232.

[SDB97] SILLION F., DRETTAKIS G., BODELET B.: Efficient impostor manipulationfor real-time visualization of urban scenery. *Computer Graphics Forum 16*, 3 (Aug. 1997), 207–218. Proceedings of Eurographics '97. ISSN 1067-7055.

[SGwHS98] SHADE J., GORTLER S. J., WEI HE L., SZELISKI R.: Layered depth images. In *SIGGRAPH* (1998), pp. 231–242.

[Ste98] STEWART A. J.: Fast horizon computation at all points of a terrain with visibility and shading applications. *IEEE Trans. Vis. Comput. Graph 4*, 1 (1998), 82–93.

[TAB*03] TELLER S., ANTONE M., BODNAR Z., BOSSE M., COORG S., JETHWA M., , MASTER N.: Calibrated, registered images of an extended urban area. *International Journal of Computer Vision 53*, 1 (June 2003), 93–107.

[WMW07] WATSON B., MUELLER P., WONKA P.: Urban design and procedural modeling. Course Notes of ACM SIGGRAPH, 2007.

[WTL*04] WANG X., TONG X., LIN S., HU S., GUO B., SHUM H.-Y.: Generalized displacement maps. In *Proceedings of the 2004 Eurographics Symposium on Rendering* (June 2004), Fellner D., Spencer S., (Eds.), Eurographics Association, pp. 227–234.

[WWS01] WIMMER M., WONKA P., SILLION F.: Point-based impostors for real-time visualization, May 29 2001.

[WWT*03] WANG L., WANG X., TONG X., LIN S., HU S.-M., GUO B., SHUM H.-Y.: View-dependent displacement mapping. *ACM Trans. Graph. 22*, 3 (2003), 334–339.

[YGKM08] YOON S., GOBBETTI E., KASIK D., MANOCHA D.: *Real-time Massive Model Rendering*, vol. 2 of *Synthesis Lectures on Computer Graphics and Animation*. Morgan and Claypool, August 2008.