A Compact Random-Access Representation for Urban Modeling and Rendering

Zhengzheng Kuang Bin Chan Yizhou Yu Wenping Wang
The University of Hong Kong

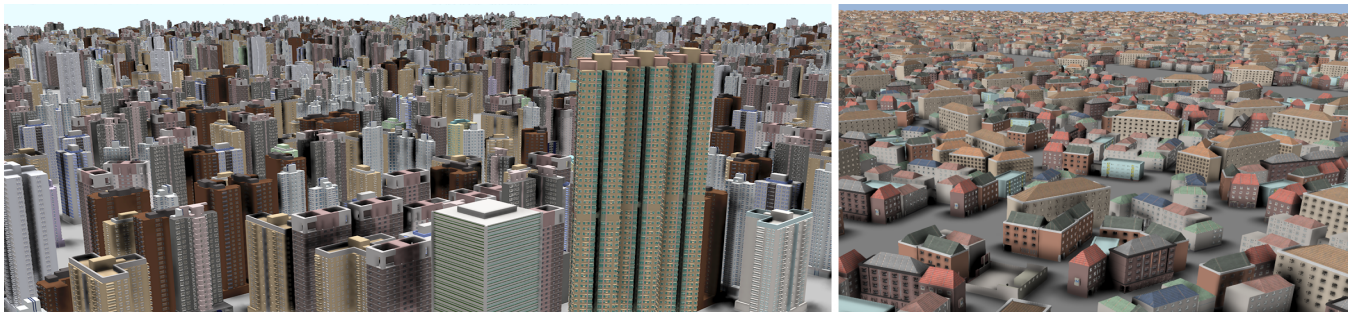

Figure 1: *Left: a city scene with 5,061 distinct buildings represented using our proposed Non-Uniform Textures (NUTs). The size of this digital model is 42 MB, compressible to 10.6MB PNG files. This model is rendered at 49 fps @ 1280×720 on an NVidia GTX580 GPU. An equivalent mesh-based model has 671 million triangles (37GB). Right: a much larger European-style city with 40,400 distinct buildings. This model consists of NUTs (229MB) and a complementary mesh (300MB) for roof structures. The total PNG-compressed NUT file size is 60.5MB. It is rendered at 30 fps. An equivalent mesh model has 1.36 billion triangles, or 61GB.*

Abstract

We propose a highly memory-efficient representation for modeling and rendering urban buildings composed predominantly of rectangular block structures, which can be used to completely or partially represent most modern buildings. With the proposed representation, the data size required for modeling most buildings is more than two orders of magnitude less than using the conventional mesh representation. In addition, it substantially reduces the dependency on conventional texture maps, which are not space-efficient for defining visual details of building facades. The proposed representation can be stored and transmitted as images and can be rendered directly without any mesh reconstruction. A ray-casting based shader has been developed to render buildings thus represented on the GPU with a high frame rate to support interactive fly-by as well as street-level walk-through. Comparisons with standard geometric representations and recent urban modeling techniques indicate the proposed representation performs well when viewed from a short and long distance.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

Keywords: urban modeling, texture mapping, ray casting

Links: DL PDF

1 Introduction

Urban visualization is a challenging task because handling the large number of buildings involved demands high storage capacity, transmission speed and rendering performance. Although systems such as Google Earth enable interactive browsing of urban environments, the visual quality is limited by coarse 3D models and low-resolution textures, making it inadequate for many applications that require highly detailed models and high-resolution rendering.

While detailed building models are desired for an improved viewing experience, the required data size can be prohibitively large. For instance, a detailed model of a multi-story building can easily consume tens of megabytes of memory. Representing a moderate-sized city model with thousands of such buildings would require tens or even hundreds of gigabytes. The conventional mesh representation is commonly used for 3D models because it can approximate arbitrary shapes and be rendered efficiently on the GPU. Nevertheless, it incurs excessive memory consumption.

Urban buildings are often composed of rectangular elements, corners with a right angle, horizontal or vertical edges and faces as well as chamfers and fillets. We wish to explore these characteristics to develop an efficient representation to drastically reduce the amount of data that needs to be stored, processed or transmitted for large-scale urban visualization.

Many existing procedural urban modeling techniques assume repeated instances of the same set of building elements or facade textures across a city. Such an assumption is often invalid in reality. In this paper we adopt a more realistic setting where there exist a large number of distinct buildings in a city, a condition under which many other methods would not perform well. We propose a new building representation, called *Non-Uniform Texture* , or *NUT* for short, that facilitates the individual representation of each building. The NUT completely relies on an image-based format without involving any mesh data structures. A building or a block of aligned buildings can be represented as a NUT in a very compact manner. The NUT can be directly transmitted to a client's GPU to be rendered efficiently without being converted to any other format first. The memory footprint of a NUT is typically more than two orders of magnitude smaller than that of its mesh counterpart. Our exper-

iments indicate that detailed city models can be quickly transmitted over the Internet and rendered in high resolution at a real-time frame rate for smooth navigation in a cluttered urban environment. In addition, an existing mesh-based building model can be easily converted to a NUT representation using a semi-automatic tool.

Major contributions of this paper are summarized as follows.

1. We propose a memory efficient building representation called NUTs. It has powerful modeling capabilities and is suitable for both Manhattan-style buildings and European-style houses. It is highly effective for viewing from a short and long distance. NUTs are deliverable through a network as images. Therefore it is also compatible with the current web infrastructure.
2. We develop an efficient method for rendering NUTs directly on the GPU in a way compatible with the existing graphics pipeline.
3. We develop a semi-automatic tool that enables the user to convert a building in the mesh format to a NUT in a matter of minutes.

The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 introduces the NUT structure. Section 4 extends NUT to a hierarchical structure for building representation. Sections 5 and 6 presents a modeling tool and a scheme for rendering NUTs on the GPU, respectively. Section 7 compares NUTs with other existing techniques. Section 8 discusses conclusions and future work.

2 Related Work

Urban visualization covers a few related research topics including representation, city modeling, urban acquisition and rendering. Related work on these topics is summarized as follows.

Representations Image-based modeling (e.g. Chen [1995] and Dykes [2000]) are the most explored representations before the emergence of programmable hardware. Those methods suffer from the problem that the viewpoint is seriously confined to small regions from which the images are captured. In the GPU era, many hybrid image-and-geometry based representations are proposed. There are two branches of works. One focuses on enhancing traditional texture mapped building models by adding accurate geometric information to the building models without using mesh data. These methods can be used to render realistic buildings individually and are suitable for close inspection. Ali et al. [2009] uses a displacement map to add indentation to texture mapped facades. Haegler et al. [2010] uses a grammar-based encoding of facades to significantly reduce the data needed to represent similar facades with indentations.

The other branch focuses on quick rendering of large-scale city-wide models suitable for viewing from a distance. Andujar et al. [2010] uses multi-level relief impostors to produce a hierarchical representation of buildings suitable for virtual globe applications. Cignoni et al. [2007] converts city models to compact image-based representations for quick rendering of large-scale distant buildings. These representations sacrifice accuracy for speed and therefore the models they represent are not suitable for close viewing such as street-level walk-through but are very efficient for high-level city viewing.

City modeling Urban models are usually generated by procedural modeling or model acquisition. Shape-grammar-based building modeling, such as CityGML by Kolbe et al. [2005], CLOQ by Döllner et al. [2005], and CGA Shape in [Müller et al. 2006; Müller et al. 2006; Kelly and Wonka 2011], provides powerful tools to generate architectural models procedurally. Marvie et al. [2012] uses the GPU to directly interpret shape grammars and avoids mesh generation on the CPU.

Urban acquisition Model acquisition is usually done by reconstruction from images or laser scanner data. An automatic approach is proposed in [Xiao et al. 2009] to generate 3D photo-realistic models from images captured along the streets at the ground level. It is discussed in [Shen et al. 2011; Xiao et al. 2008; Müller et al. 2007] how to extract and use building facades from images to reconstruct 3D building models. Nan et al. [2010] proposes a method to interactively snap building blocks to point cloud data with simple user manipulations. Another method is proposed in [Nebiker et al. 2010] that uses dense 3D point clouds, coupled with comprehensive geometric, radiometric and semantic information, to reconstruct a 3D scene.

Rendering Urban models are mostly rendered using meshes. Sometimes ray-casting, which is primarily used in volume rendering, is used for urban rendering, for example, Crassin et al. [2009] and Zhou et al. [2008]. GPU-based ray casting has also been used for rendering height fields [Mantler and Jeschke 2006; Dick et al. 2009]; for adding surface details [Jeschke et al. 2007]; and for rendering urban models [Ali et al. 2009; Marvie et al. 2011; Cignoni et al. 2007].

3 Hierarchical Grid-Based Modeling

We have the following observations of modern buildings. First, orthogonality is a commonly shared attribute. Second, there exist many repetitions of the same elements. Such repetitions can be grouped in a few different levels. For example, there are repetitive windows in a floor and floor repetitions in a building. Third, elements are often aligned horizontally or vertically. Finally, when the geometry is sufficiently detailed to represent every individual surface of a building, the appearance details we find on the surfaces are usually homogeneous, e.g. a marble pattern or latex paint. Therefore, the required texture maps are often material-dependent instead of being structure-dependent. This is important because material-dependent textures are highly reusable in a city while structure-dependent textures are not.

We propose a new modeling scheme for buildings. Our goal is to drastically reduce the amount of data, including geometric data and texture maps, necessary for representing a city accurately. First, we locate the rectangular regions of a building and identify the sets of parallel lines that align those rectangular regions. Then, in a divide-and-conquer manner, we break down hierarchically the entire volume into multiple blocks by fitting 3D grids of variable cell sizes over the building. See Figure 2 for a 2D illustration of the process. The grid boundaries are then aligned to match the surface boundaries or material boundaries of the building. The process is recursively repeated for all cells until each cell in the 3D grids is either an empty cell or a non-empty cell with the same material properties.

The adaptiveness of the grid cell boundaries and the hierarchical structure dramatically reduce the dimensions of the 3D grids at each level. All sub-grids are clipped by the parent cell when rendered but there is no bound on the size of a sub-grid. In other words, the parent cell is just a window to a possibly much larger sub-grid. Such a design can increase the modeling efficiency of a grid as it is possible to have many similar underlying components that share common attributes (e.g. modeling several windows of similar dimensions together needs less data than modeling them individually as some cell dimensions are shared). In order to support that kind of component reuse, the position of a sub-grid can be translated

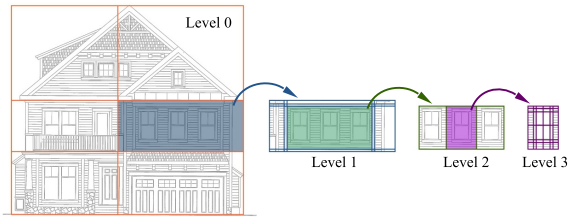

Figure 2: *2D illustration of hierarchical grid-based modeling*

and rotated freely within its parent grid. Allowing size differences between parent cells and sub-grids also enables easy repetition of sub-structures, which is similar to texture wrapping when texture coordinates are beyond the range of [0,1] (Figure 3).

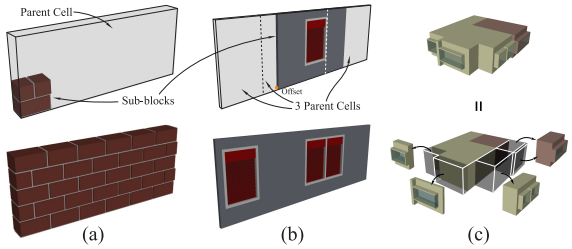

Figure 3: *(a) Object wrapping achieved by extending the size of the parent cell. It is convenient for modeling repeated patterns. (b) Arbitrary redirection, for reusing sub-blocks or parts of the sub-blocks, allows an irregular arrangement of components by controlling the offset of each redirection. (c) Redirection at a higher level.*

A 3D grid in essence represents a solid model of a building without recording the vertex positions, normals or texture coordinates. Although a NUT is based on an axis-aligned grid, by supporting rotational redirection and quadric surface solids, its modeling capability extends beyond box-looking Manhattan-world building models. Figure 4 shows some structures that can be modeled and rendered together with other parts represented as NUTs.

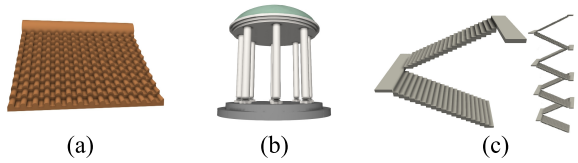

Figure 4: *Building structures modeled entirely using NUTs. (a) An imbrex roof model represented using 612 bytes. (b) A rotunda model represented using 2.4KB. (c) A four-story staircase model represented using 3.4KB with rotational redirection.*

3.1 Non-Uniform Texture (NUT)

The proposed 3D grid described above consolidates to the design of a new data structure called Non-Uniform Texture (NUT), encoded in 2D images. Such 2D-image-encoded 3D data is similar to Geometry Images by Gu et. al. [2002] however the main difference is that NUT is used to encode internal solid information of axis-aligned models instead of representing surface details of arbitrary models. The data stored in NUT are coordinates, dimensions and material properties organized in a special layout. Please refer to Figure 12b for the appearance of a hierarchical NUT image. Hence, it is called a texture only because 2D images are used as its representation, so it should not be confused with conventional 2D or 3D texture maps.

The intuition behind the definition of the NUT is best explained with the help of a 2D diagram. Consider a pattern shown in Figure 5 that we want to represent by an ordinary uniform texture (see Figure 5b). Obviously, the image has to be of sufficiently high resolution (21×30 in this case) in order to make sure the boundary lines of the shapes confirm with the columns and rows of the images. NUT, on the other hand, allows adaptive pixel sizes so that the pixel boundaries can move freely to align with color boundaries, hence *non-uniform*. Therefore, the same pattern in Figure 5a can be represented using much fewer pixels in NUT than in an ordinary texture, see Figure 5c. Note that the NUT is not intended to represent general images but discrete axis-aligned color patterns often appearing on architecture models.

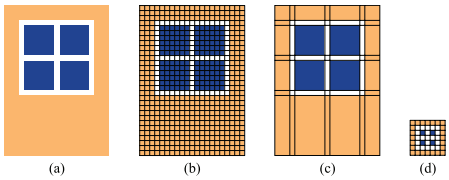

Figure 5: *(a) Original window structure. (b) A conventional texture representing (a) using 21×30 pixels. (c) A NUT representing (a) using 7×7 pixels. The NUT has 45 pixels of metadata. The data size is reduced by 6.7 times. (d) The NUT in (c) is stored into uniform pixel size.*

3.2 Random Access and Metadata Array

Since GPU supports the lookup of ordinary textures only, a customized lookup mechanism is needed to simulate NUT lookup. Some extra data (*metadata*), one set for each dimension, besides the color component, is needed for NUT lookup. A straightforward choice of the metadata would be a series of pixel widths and heights for binary searches in the shader. However, that implies multiple dependent GPU texture lookups, each taking hundreds of clock cycles to complete. That would make NUT lookup significantly slower than ordinary texture lookup. Therefore, a heuristic approach is developed. The idea is to allow a small error in pixel sizes such that exact pixel sizes is not needed. Instead, an array of indices that guides the shader to pick the correct pixel is used. This is similar to random access of run-length encoding [Crivat et al. 2011].

Let's consider a single dimension for simplicity. For a NUT of W pixels wide, as shown in Figure 6, a metadata texture of size $W+w$ by 1 pixels is needed, where $w \geq 0$. The number w is chosen such that each metadata pixel contains at most 1 NUT pixel boundary when the metadata pixels and the NUT pixels are stretched to the same width and aligned. In other words, each metadata pixels touches at most 2 NUT pixels. The value of w depends on the distribution of the underlying pixels and there are some constraints for the value of w and pixel sizes representable by NUT. Let X_T, X_w and X_n be the total width of the pixels, the width of the widest pixel and the width of the narrowest pixels, w could go from 0 up to $X_T/X_n - W$. Since the j numbers are stored in 8-bit color components, X_w/X_n must be ≤ 255 and therefore $X_T \leq 255 X_n W$ or $X_T/X_n - W < 255W$. Combining the equations together we get the possible range of w goes from 0 to $254W$. In practice, $W + w$ is limited to 256 for easier handling during compilation to images so whenever the 256-pixel size constraint is violated, the modeling tool will prompt the user to break a block into smaller ones. We think it is not easy to reach the limit in practise as we have not

encountered such problem in any of our building models.

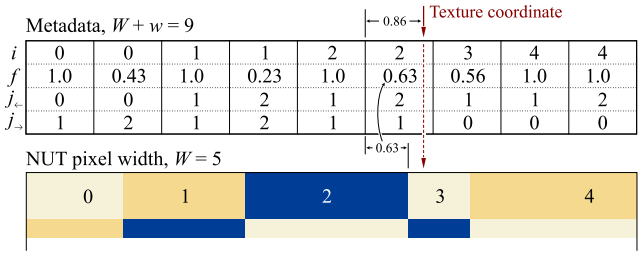

Figure 6: *Heuristic NUT lookup in a NUT 5 pixels wide (i.e. $W = 5$ and $w = 4$ in this case).*

Each pixel of the metadata array stores four components: index i, fraction f, and two j numbers. Here i is the index of the left-most NUT pixel that each metadata pixel touches, and f is an integer-coded fraction used for metadata pixels that intersect more than 1 NUT pixels. It represents the approximate position of the NUT pixel boundary in the metadata pixel. When the GPU receives the texture coordinate, it looks up the metadata uniformly and uses the index i to find the NUT coordinate directly if $f = 1$, otherwise the fractional position of the texture coordinate in the metadata pixel is compared with f to decided if NUT pixel i or $i + 1$ is to be used. The numbers $j_{\leftarrow}$ and $j_{\rightarrow}$ represent the number of metadata pixels to skip to reach the ones that touch the adjacent pixel boundaries for both directions. The nearest pixel boundary can therefore be reached with just 1 extra texture lookup of the metadata texture following the hint of the j numbers. Note that the j numbers are not used in random access but it is useful in ray casting where adjacent pixel boundaries search is needed.

The error E caused by heuristic lookup depends on the size of the NUT in world coordinate and how the f component is coded. Normally, an 8-bit integer is used and in that case $E \leq \frac{D}{512(W+w)}$, where D is the world-space size of the NUT's width. E is usually small enough to avoid any perceivable problem in a city modeling setting.

With such a metadata design, only 1 extra lookup of the metadata is required for each dimension besides the lookup of the color value. Table 1 shows the relative speed comparison between ordinary texture lookup, NUT lookup based on binary search and NUT lookup based on heuristics. There is on average a 30% slow down for NUT heuristics lookup when compared with ordinary texture lookup.

	Texture dimensions			
Texture Lookup	4×4	8×8	16×16	32×32
GPU Native	19.61	19.55	19.54	19.59
NUT Binary Search	19.52	8.99	3.32	1.21
NUT Heuristic	13.91	13.89	13.96	13.98

Table 1: *Comparison between native ordinary texture lookup, NUT lookup using binary search and NUT lookup using heuristics with different texture sizes. All frame rates are measured by issuing 1 billion texture lookups / frame, on a Nvidia GTX480. Typically, NUT dimension of 8-16 is used when modeling building elements.*

4 Hierarchical NUT for Building Modeling

When extending NUT to a hierarchical structure, each redirection has a rotation and a 3D offset that controls the positioning of the sub-block. NUT hierarchy is designed to be a directed acyclic graph to allow flexible reuse of building components. All NUTs in a hierarchy are eventually compiled into a 2D 32-bit RGBA Hierarchical NUT Image (HNI), see Figure 12. The upper left 256^2 pixels square is defined as the *metadata region* where small entities such as NUT headers and metadata arrays are stored so that items located in the metadata region can be addressed by 8-bit integer coordinates for compactness. Multiple buildings can be encoded in a single HNI to share common sub-blocks. Besides the Metadata Arrays described in the previous section, each NUT contains two other components: the NUT Header and the Data Component. Each HNI also contains a Material Array and a Texture Information Array.

4.1 The NUT Header

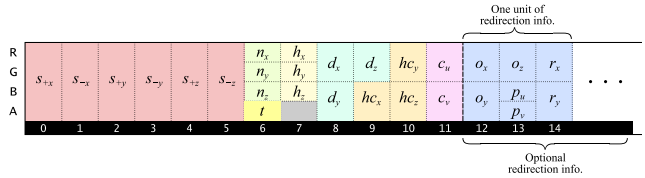

Figure 7: *The NUT header. Gray color represents unused space. The first 12 pixels are mandatory while the redirection info., 3 pixels each, is optional. $S(s_{\pm x}, s_{\pm y}, s_{\pm z})$ are the* visual summary *which summarizes the color of each side of the block by a single color. They are used for level-of-detail rendering of the buildings. $N(n_x, n_y, n_z)$ are the NUT size in number of pixels. $H(h_x, h_y, h_z)$ are the size of the metadata array of the 3 dimensions. $D(d_x, d_y, d_z)$ are the physical dimension of the NUT in cm. $Hc(hc_x(u, v), hc_y(u, v), hc_z(u, v))$ are the coordinates of the 3 metadata array. $C(c_u, c_v)$ are the coordinates of the corners of the data component of the NUT. $O(o_x, o_y, o_z)$ are the offset distance of each redirection. $P(p_u, p_v)$ are the coordinate of the header of the redirected NUT. $R(r_x, r_y)$ are the rotational angles about the x, y axes in a redirection. t stores the flags that control the tiling behavior of the NUT.*

The NUT header stores vital information such as NUT size and redirection information. Figure 7 shows the byte arrangement and the size of each element. Each redirection requires a copy of the three elements O, P and R.

4.2 Data Component

The data component is used to describe the shape, material and texture mapping of each NUT cell. Each cell can take the form of a wedge, a pyramid or a quadric surfaces or their inverses like those in Figure 8. There are altogether 69 types of shapes to choose from so that chamfers, fillets, rods and dome structures can be modeled directly by NUT without resorting to mesh surfaces.

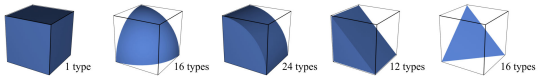

Figure 8: *Each NUT cell can have geometry from these 5 groups. Different solids within each group differ only in their orientation within the bounding cube. Inverse solids, i.e. those with the shape subtracted from a full cube, are also included in these groups.*

Each cell can also have its own material property and texture mapping. Since many material properties and textures are shared among the cells and different NUTs in the hierarchy, the material Array and the texture Information Array, both defined as horizontal rows of pixels, are used for references. Figure 9 shows the pixel format of the data component, material array and texture information array.

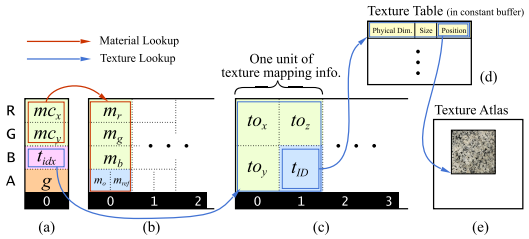

Figure 9: *(a) Format of data component pixel, for each cell. (b) Material Array. (c) Texture Information Array. (d) Texture Table in GPU's constant buffer. (e) Global texture atlas.* $Mc(mc_x, mc_y)$ *are the texture coordinates of the material pixel.* t_{idx} *is the index of the texture mapping info. in the texture information array.* g *is the geometric descriptor that indicates if the cell is an empty cell, redirection cell or a solid cell.* $M(m_r, m_g, m_b)$ *is the diffuse color of the material.* m_o *is the opacity and* m_{ref} *is the reflectance.* $To(to_x, to_y, to_z)$ *is the 3D texture offset and* t_{ID} *is the index of the texture image. The index is used to fetch data from the texture table in the GPU's constant memory to obtain the location of the mapped texture in the global texture atlas.*

In our design, when g equals to 1, it is used as a redirection cell, where the R and G component of the data component pixel is interpreted as the coordinate of the redirection info. block. Figure 10 shows how redirection is done. Each NUT cell requires one pixel in the data component. The 3D data component is eventually flatten to 2D by tiling the z dimension along the x dimension, for compilation into the final 2D HNI described in section 4.4.

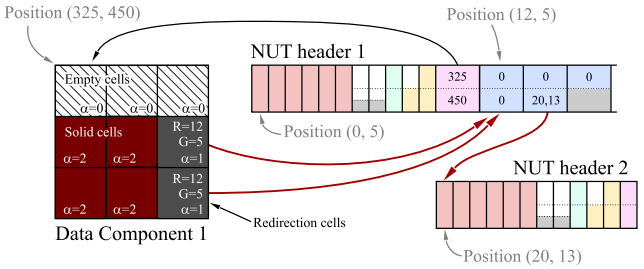

Figure 10: *Redirection is represented by setting the alpha value to 1, in which case the red and green components of the pixel are taken as the coordinates of the redirection info. block, which contains the offset of the redirected block and the coordinates of the redirected block's NUT header.*

4.3 Texture Mapping

Ordinary textures are used to enhance the realism of single color materials. NUT provides limited support of ordinary texture mapping. In our modeling scheme, textures in NUT are used for two primary purposes: as decals or as material textures. In order to minimize the data needed to support texture mapping, we have made two assumptions, based on the two purposes. First, since each NUT cell should have already been divided sufficiently to represent a region of a single material, the texture of all faces of the cell should be the same. Therefore, we limit each NUT cell to have only one texture, which is mapped onto all faces of the cell to reduce the complexity of the NUT data structure. Second, material textures such as wood or granite patterns do not appear to be natural when resized. Decals that represent signs or complex patterns usually do not have a smaller or bigger counterpart. Therefore, all texture images used in a NUT are associated with a fixed physical dimension and a default orientation so that they cannot be resized or rotated. By doing so, we can avoid storing texture coordinates explicitly for the corners of each texture mapped cell. Instead, a single 3D offset suffices for fixing the texture coordinates for any point on the solid surface via simple transforms.

Methods for mapping texture images onto the faces of different solid geometry groups are defined differently. The general rule treats texture maps as tilable stickers on a solid surface. Specifically, for all non-horizontal faces, when they are being looked at, the right direction is always taken as the positive horizontal direction and the up direction is the positive vertical direction. For flat surfaces facing up and down, the mapping direction is chosen depending on the z direction. For non-horizontal flat faces, both the horizontal and vertical texture coordinates are mapped linearly. For vertically aligned cylinders and spheres, the horizontal texture coordinate is cylindrically mapped starting from the most clockwise corner, which is set as the origin, and increases in the counterclockwise direction (Figure 11). For horizontal cylinders, texture coordinates in both directions are also mapped linearly. Such a definition can ensure seamless texture mapping over the surface of chamfers, fillets and round corners when all associated cells are set to the same texture offset. For both quadric cases, the vertical direction is also mapped linearly.

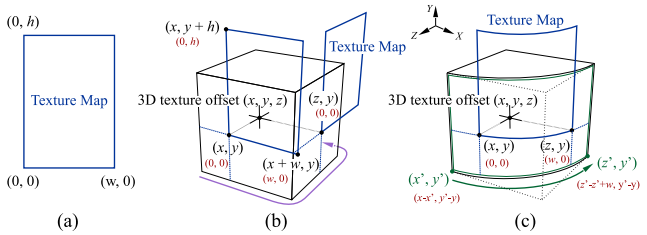

Figure 11: *(a) A texture map is associated with physical dimensions and its coordinates are defined in the following ranges:* $x \in [0, w]$ *and* $y \in [0, h]$. *(b) 2D texture coordinates (in red) are computed from the surface coordinates and the 3D texture offset. Different faces use different coordinate components from the offset. The purple arrow indicates textures are wrapped counterclockwise around the box. The blue lines indicate texture borders and show how it is tiled. (c) Quadric surfaces are cylindrically mapped. The texture coordinates are computed from all three coordinate components of the offset.*

During rendering, the texture images themselves are stored in a global atlas. The positions and dimensions of the texture images in the global atlas are stored in a fixed buffer on the GPU.

4.4 HNI Compilation

During compilation, all NUT headers, metadata arrays, data components, the material array and the texture information array in the entire hierarchy are passed through a bin packing solver. The bin packing algorithm we use is based on the MaxRects method with the best area fit heuristic described in [Jylänki 2010]. In order to find the best image dimension as well as tile pattern, the width of the HNI is iterated from the maximum NUT data component width, i.e. $max(n_{x_1} n_{z_1}, n_{x_2} n_{z_2}, ..., n_{x_M} n_{z_M})$, where M is the total number of NUTs in the hierarchy, up to 256. The chosen HNI is the one with the least waste percentage, which depends heavily on the data component size variations and the number of NUTs. When the number of NUTs in an HNI is large, the waste percentage is typically below 2%.

For a typical residential building with detailed structure of every

(a) (b)

Figure 12: *(a) A building made up of 40 NUTs. An equivalent mesh has 70K triangles. (b) is the compiled HNI file of (a). With* 101×26 *pixels, it uses 10.3KB, reduced to 3.1KB when compressed as a PNG file. The alpha channel of the image is stripped for printing. The dark brownish region is the data component while the remaining region contains all other arrays. For this model the waste percentage happens to be 0%.*

windows on every floor, the generated uncompressed HNI is usually less than 40KB. That means for modern GPUs, it can store tens of thousands of buildings in core, which is enough to cover most cities in the world. HNI image is also highly compressible. Saving as PNG has a typical 4:1 compression ratio. That means each building can be saved in less than 10KB.

5 Modeling Tool

A NUT can be either generated manually or converted from mesh models. We have created a custom tool with building modeling and city layout capabilities. The time needed to model a building using this tool ranges from minutes to several hours, depending on the building complexity, which is comparable to a mesh-based modeling system.

There is much literature, e.g. [Shen et al. 2011], on the extraction of symmetry from point cloud data of buildings. But we are not aware of any work that supports automatic hierarchical extraction that balances the element size and the number of levels, which is vital for effective NUT representation. Instead, we developed a semi-automatic mesh converter similar to that in [Musialski et al. 2012]. Unlike their method, we use a bottom-up approach instead of a top-down approach to allow building elements to span across high-level boundaries. Our method involves three main steps, solid conversion, block extraction and simplification.

Solid conversion Given a closed surface mesh of a building (assuming most of its faces are already axis-aligned for convenience), the x, y and z coordinates of the intersections of all axis-aligned planes are recorded. These coordinates are grouped into clusters X_i, Y_j and Z_k where $i = 1...n_x$, $j = 1...n_y$ and $k = 1...n_z$ are the indices of clusters in each dimension. The weighted averages, x_i, y_j and z_k, of the clusters are computed to represent the clusters where mesh face areas are used as weights. Then the bounding box of the mesh is partitioned according to x_i, y_j and z_k to form a 3D non-uniform grid. All mesh faces contributing to X_i, Y_j and Z_k are associated with the faces of the corresponding cells. Each cell's occupancy is obtained by tracing rays from one end to the other along one of the dimensions and counting the number (even or odd) of intersections with mesh faces. The result is a solid representation G_0 of the original mesh. All non-axis-aligned mesh faces are collected and saved as a complementary mesh.

Block extraction User interaction is required to select a rectangular 3D region S_0 of cells that represents a basic building block, such as windows or balconies, in G_0. Similar structures in the building can be identified automatically. The number of cells along each dimension of the block, w_p, w_q and w_r where $p = 1...m_x$, $q = 1...m_y$ and $r = 1...m_z$ in S, and the occupancies within the block, $O_{p,q,r}$, are used to test against the entire grid. The test compares the width pattern in each dimension of the block against cell width sequences starting at all positions in that dimension. The process is somewhat like string search except that a cell c_i with width w_i can be matched to multiple successive cells from c_j to c_{j+n} with widths w_j to w_{j+n} if w_i is within $(\sum_{k=0}^{n} w_{j+k}) \pm \epsilon$, where ϵ is a predefined tolerance value. A block match is reported only if the cell width patterns of the selected block in three dimensions all match the cell width patterns of another block and their corresponding cell occupancies are all identical.

Simplification Once all blocks $\mathbb{S}_0$ of the same type S_0 have been identified, all cells in $\mathbb{S}_0$ are labeled with a unique identifier that represents S_0. Those original cell boundaries x_i, y_j and z_k used for splitting the cells in $\mathbb{S}_0$ (i.e. those not used as solid/space boundaries in any other cells) are removed and relevant cells are merged to produce a simplified building model G_1 which contains a redirection pointer to the basic building block S_0. This forms the basis of the hierarchical NUT. If the blocks in $\mathbb{S}_0$ are found to be aligned and touching each other, the user has the option to merge the repeated blocks together to form a larger block before cell boundary removal and cell merging. Simplification prevents previously extracted blocks from being extracted again and hence prevents redundant representation of the same elements.

Block extraction and simplification are repeated until no more similar elements can be extracted. In each iteration, a new building block S_i is discovered and a new simplified grid G_{i+1} is produced. After n iterations, the solid models G_n, S_0, $S_1...S_{n-1}$ are converted to $n + 1$ NUT blocks. Redirections are added according to the unique identifiers recorded in each cell. See Figure 13 for an example.

This mesh-to-NUT conversion process usually takes just a few minutes to complete. The converter has a certain degree of built-in tolerance and works well with reasonably clean mesh models. However, building models with large holes or many redundant surfaces could negatively affect the result of this solid conversion step. In that case, manual cleanup of the model before conversion is required (refer to the accompanying video for a demonstration of NUT generation and mesh-to-NUT conversion). Note that the solid conversion step described above is a memory-intensive task because it needs to construct the 3D solid in full details before simplification. In practice, a building is seldom processed as a whole. Instead the facades are first processed individually and assembled together later as NUTs to reduce memory consumption. Note that the mesh-to-NUT converter does not support rotated sub-blocks.

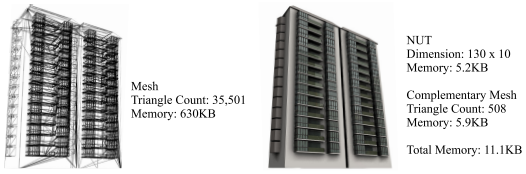

Figure 13: *Sample input and output of the semi-automatic mesh-to-NUT converter. The roof and side balconies are incompatible with NUT and are retained as a complementary mesh to the NUT. There is a 98.2% reduction in data size when NUT is used to represent this model.*

6 Rendering

NUT models can be efficiently rendered without mesh reconstruction. A NUT hierarchy can be seen as an adaptive spatial subdivi-

sion structure with well defined spatial compartments. Therefore it would be efficient to directly perform ray casting on the hierarchy on the GPU without any additional acceleration data structure.

Similar to other GPU ray casting algorithms, HNI ray casting also runs in a pixel shader. The front face of the bounding box is culled as in [Dick et al. 2009] to make sure the contents inside are visible even when the viewpoint is outside the bounding box. Our ray casting algorithm, Algorithm 1, loops over two main steps, *NUT traversal* and *redirection landing*. Each step has different NUT lookup patterns.

Algorithm 1: HNI Ray Casting Pixel Shader

```
Input: BackFacePosition
Output: PixelColor
RayPosition = FindFrontFacePosition(BackFacePosition);
RedirectionLanding(RootNUT);
PixelColor = NUTColorLookup(RayPosition);
while IterationCount < MAX and HasNotHitSurface() do
    AdvanceRay(RayPosition);
    if RayPosition hits RedirectionCell then
        if ProjectedSize(HitCell) < THRESHOLD then
            PixelColor = VisualSummary(RedirectedNUT);
            continue;
        else
            RayPosition=RedirectionLanding(RedirectedNUT);
        end
    end
    else if RayPosition hits EmptyCell or TransparentCell then
        if RayPosition leaves ParentCell then
            RayPosition = ReturnToNUT(ParentNUT);
        else
            RayPosition = NUTTraversal();
        end
        if RayPosition leaves GlobalBoundingBox then
            clip;
        end
    end
    else
        PixelColor=NUTColorLookup(RayPosition);
    end
end
return PixelColor
```

NUT traversal refers to the iterations that extend a ray forward within the same NUT. The shader always keeps track of the last intersections between the ray and three cell boundary planes perpendicular to the coordinate axes. In general, each forward propulsion moves the tip of the ray to the next cell boundary plane perpendicular to one of the three coordinate axes. That means, only one of the recorded ray-plane intersections has to be updated, i.e. *one* next cell boundary query, in each traversal iteration to ascertain the boundary plane where the ray enters the next cell. As shown in Figure 14a, given the precomputed j numbers in the metadata array, each next cell boundary query only performs two native texture lookups of the metadata array, same cost as a random NUT access. Therefore, NUT traversal is very efficient.

Redirection landing is the process that guides the tip of the ray in one NUT to *land* in a redirected NUT. It involves three steps. First, the redirection information (3 pixels) must be fetched from the end of the header of the current higher-level NUT. This is followed by a ray transformation based on the rotation and offset associated with the redirection. Second, the basic information stored in pixel number 6 to 11 of the header of the lower-level NUT must be retrieved. Finally, since the boundary of the lower-level NUT does not necessarily align with the higher-level NUT, the x, y and z indices and the size of the cell where the ray lands in the lower-level NUT must be established. That involves three random NUT lookups and three cell boundary position queries, as shown in Figure 14b. These three steps together make redirection landing a resource-demanding task. When moving down the hierarchy, a stack-like mechanism

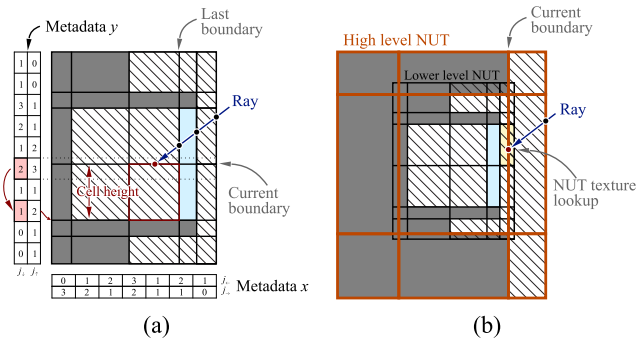

Figure 14: *(a) Same-level NUT traversal. When the ray touches a horizontal cell boundary at the red dot, the height of the next cell is found by looking up the metadata y array. The j numbers tell which metadata pixel contains the next boundary position. The difference between the next boundary and the current one is the cell height. The blue cells are transparent cells that do not stop the ray but only affect the output color. All hashed cells represent empty cells. (b) Redirection landing. After vital information of the lower-level NUT is loaded, the cell in yellow will be found by heuristic lookup followed by retrieving the cell boundary planes for subsequent traversal.*

is maintained to temporarily store the cell boundary information for detecting if a ray has exited the parent cells and to reduce the number of lookups when returning to the upper level. The number of maximum iterations needed depends on the complexity of the building model. Normally a detailed building model would not require more than 10 iterations as most NUTs have small dimensions. However, self-repeating (i.e. object wrapping) often requires much more iterations. Experiments indicate that usually 20-50 iterations are sufficient most of the time.

Note that rendering NUT models that contain many quadric surface cells could be considerably slower than pure rectangular structures. The degree of slowdown depends on many factors. To have a better understanding, we have performed experiments and found that for typical structures where quadric surface cells cover less than 10% of the total projected screen area, the degree of slowdown in rendering is also less than 10%.

6.1 Implementation

The HNI shader is built using DirectX 11 HLSL with Shader Model 5.0. Due to the unique nature of NUT, only *nearest- pixel* texture sampling is needed. Any higher-order filtering is irrelevant because sharp color changes between adjacent pixels are necessary when we represent features such as wall boundaries, material boundaries and sharp edges. In fact, texture filtering is not applied at all for HNI lookup since, otherwise, the shader could slow down by a factor of 10. In practice, all HNIs and texture atlases are tiled into one or more 4096×4096 atlas images to avoid frequent texture switching.

Ray casting runs entirely on the GPU. The CPU performs front-to-back sorting on the bounding boxes every few seconds so that early depth culling can be used to prevent unnecessary computation on invisible objects behind the front rows of buildings. Level-of-detail simplification (using visual summaries, see Figure 7 and Algorithm 1) is used in the ray caster to achieve faster rendering and less aliasing on distant buildings.

NUT-oriented ray casting can seamlessly mix with ordinary mesh rendering because when a ray hits a solid, the depth value is calculated in the same way an ordinary triangle is rendered. The Eu-

ropean city in Figure 1 is such an example. Therefore, NUT-based modeling can be integrated with most existing modeling techniques to produce realistic and memory-efficient urban models.

6.2 Examples

Figure 15 shows renderings of a modern city model (details can be found in the next section). All the building elements are represented geometrically, and every window of every building is represented individually. The NUT representation is also efficient in modeling interior structures. Figure 16a shows the interior of a multi-level car park with clearly visible pillars. Figure 16b shows the interior of a house modeled using NUT. Figure 17 shows screen shots of the European city. Refer to the accompanying video for further details of these building models and a fly-by of the cities.

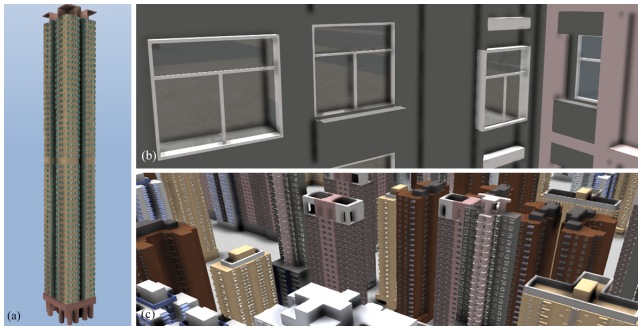

Figure 15: *(a) Model of a 53-story apartment building with NUT resolution* 233×19*, 17.3KB. (b) Closeup view of the windows of a detailed model with NUT resolution* 229×12*, 10.7KB. (b) Distant view of a district.*

Figure 16: *(a) Car park model, same model as in Figure 15b. (b) Exterior and interior of a house represented in a single NUT image with resolution* 214×25*, 20.9KB.*

7 Comparisons

In this section we compare our NUT representation with a few other methods. We use two large-scale city models in our comparisons. The first one is a city with 5,061 high-rise buildings, most of which are over 20 stories, constructed entirely using NUTs. The second one is a European-style city with 40,400 4-6 story buildings. It was modeled using both NUTs and meshes (for parts incompatible with NUT, such as roofs). In fact simple angled roofs can be embedded directly into the NUT structure and it is far more efficient that way but we chose to model them using meshes to demonstrate the interoperability between NUTs and meshes. Although the same buildings are placed repeatedly in the cities, every building instance is treated as a distinct building with data duplicated to simulate a genuinely large-scale city model. Refer to Table 2 for the data size of

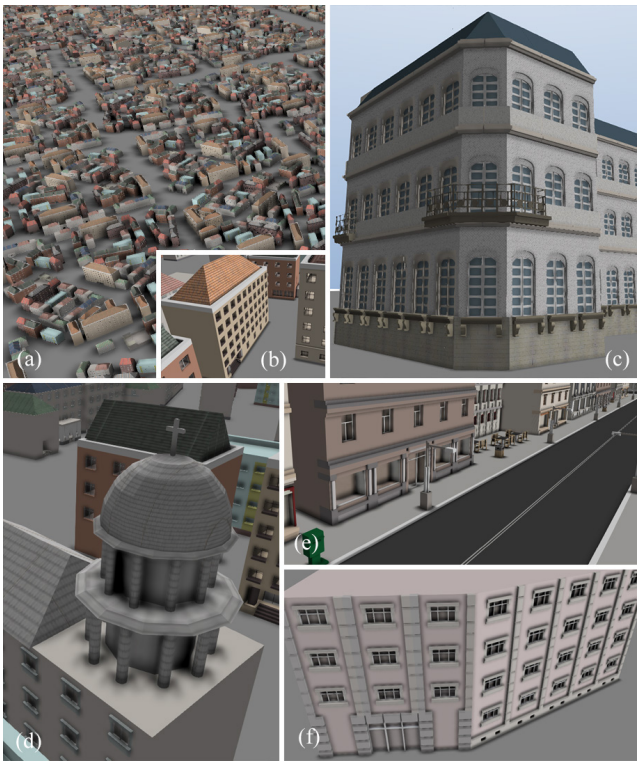

Figure 17: *(a) Distant view of the European city model. (b) Closeup view of (a). Note the details, such as the interior of the buildings, are modeled using NUT with the data size less than 10KB. The roofs are mesh models. (c) A highly detailed European-style house modeled entirely using NUT (including the roof and the curved surfaces) and some material textures. This model uses all features offered by NUT including object tiling, redirection, object rotation, quadric surface cells and texture mapping. The NUT resolution is* 211×13*, and the size of the model is 10.7KB. (d) A hybrid NUT/Mesh church model in the scene. (e) Detailed building models with shops and cafes. All objects in this figure are represented using NUTs. (f) A hospital modeled using NUT with rotational redirection.*

the two city models. Unless otherwise stated, all experiments were conducted on an NVidia GTX580 with 2GB video memory and the screen resolution is 1280×720. The maximum number of iterations in the pixel shader was set to 50 by default. The average frame rates of the modern city and the European city are 49 fps and 30 fps, respectively.

7.1 Comparisons with Standard Representations

We first compare the proposed technique with ordinary mesh-based models as well as object instancing. Both space efficiency and rendering efficiency are compared for the three methods. In the experiments, city models with different numbers of distinct buildings were built using NUT. Each model was then converted to a mesh model and simplified by removing all redundant vertices and triangles. For the model based on instancing, all identical floors at the middle of the building were replaced by floor instances so that the amount of mesh data has been substantially reduced. We instantiated the model at the floor level because a finer level of instantiation would give rise to better compression ratios on models based on instancing or NUT. For the purpose of comparison, instantiation at the floor level is already sufficient.

Cities	Bldgs.	NUT/Mesh Size	PNG	ETC / Size
Modern	5,061	42M / -	10.6M	671 Mil. / 37G
European	40,400	229M / 300M	60.5M	1.36 Bil. / 61G

Table 2: *Cities built using NUTs. PNG denotes the size of PNG-compressed NUT images. ETC means equivalent triangle count, which refers to the total number of triangles in an equivalent mesh model.*

The models were then rendered using a set of viewpoints. Table 3 and Table 4 summarize the results.

No. of Bldgs.	225		900		2116		8464	
Method	VM	TC	VM	TC	VM	TC	VM	TC
NUT	2.3	-	9	-	21	-	85	-
Instancing	27.9	1.5M	111	5.3M	262	12M	1048	50M
Ordinary	148	7.8M	593	28M	1394	65M	5579	261M

Table 3: *Memory consumption comparisons between three modeling methods. VM represents the amount of video memory required for the model, and its unit is megabyte. TC represents the triangle count of the model.*

No. of Bldgs.	225	900	2116	8464
NUT	61.2	54.2	51.6	34
Instancing	71.9	15.9	6.8	-
Ordinary	114.0	34.4	-	-

Table 4: *Frame rate comparisons among the three modeling methods. When the model grows too large, the meshes are not able to fit into the GPU memory. So the frame rates are unknown for some of the above tests.*

It can be seen from Table 3 that the NUT-based model is an order of magnitude more space-efficient than instancing and dozens of times more efficient than using mesh. Note that the testing models used for comparisons are only fairly complex to allow several hundreds of their mesh-based variations to fit into the video memory; otherwise it would be difficult to compare the methods. The degree of saving in the size of memory footprint increases as the model complexity grows. For example, the space saving factors for the models shown in the accompanying video are typically in the range of 300-400.

Table 4 shows that the ordinary mesh representation is extremely fast when the data size is not overly large. Instancing is much slower because it requires extra vertex transforms. When thousands of mesh-based buildings are inside the viewport, most of the rendering time is spent on vertex shaders and the extra vertex transforms become an important negative factor. Due to its complexity, the NUT representation is the slowest among the three under low data load. Nevertheless, the frame rates on both mesh-based models are inversely proportional to the mesh size while the frame rates on NUT stays relatively constant as the city size grows. This is mostly due to the fact that ray casting is an image-space operation independent of the actual triangle count. Although ray-casting itself is slow, with the help of front-to-back sorting and early depth culling, the total number of times ray-surface intersections are computed does not deviate much from the number of screen pixels. Therefore it is advantageous when rendering extremely large environments.

The proposed representation has also been compared with kd-trees for better understanding of the performance of ray-casting on NUT models. Since our models are complicated, the kd-tree representation of a building has at least tens of megabytes and it would be difficult to compare rendering performance at the scale of a city. Instead we chose to use two buildings from our modern city scene in our comparisons. Kd-tree based ray-casting was implemented using CUDA and was run on the same GPU. Table 5 shows that ray casting the NUT hierarchy is around 30% to 100% more efficient than the kd-tree equivalent mainly due to the fact that a spatial partition in a kd-tree affects the current subspace only while a partition in NUT affects all cells in the same level. Therefore there are a lot more splits and levels in the kd-tree than in the NUT hierarchy.

Model	kd-tree size / fps	NUT dimension / size / fps
Tower (Fig. 15a)	122.7M / 89	233×19 / 17.3K / 122
Gray (Fig. 15b)	36.7M / 87	229×12 / 10.7K / 161

Table 5: *Comparisons between NUT ray casting and kd-tree accelerated ray casting. The frame rates are recorded with a closeup view of the buildings which fully cover the screen.*

7.2 Comparisons with Grammar-Based Facade Models

To demonstrate the modeling efficiency of our representation, it has been compared with [Haegler et al. 2010] (GBEF), a very compact representation specifically for modeling indentations over building facades, which can be rendered efficiently on GPUs. While NUT can be used for modeling far more general 3D structures than facade indentations, for the sake of comparison, a few simple facade models similar to M1, M2 and M8 in their paper were created using NUT (Figure 18). Since there is no information about the exact geometry of the three facades [Haegler et al. 2010], we modeled them with two different indentations.

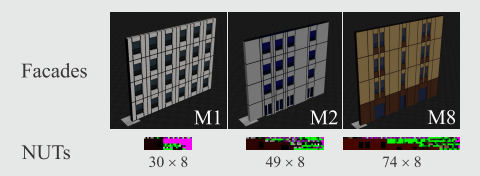

Figure 18: *Facade models used for comparisons.*

There are two alternatives for the construction of the NUT. The first one adopts a texture atlas as what GBEF does and creates simple NUT hierarchies to position the textures. With this configuration, each NUT only requires one level of redirection. The second alternative exploits the color embedding ability of NUT to eliminate the use of a texture atlas at the cost of slightly more complicated NUT hierarchies. In this case, M2 needs one more level of redirection. Table 6 shows a comparison of model size between GBEF and NUT. Note that the size of the texture atlas must be added to the values in the GBEF and NUT (opt.1) columns in the table. For facades like M1, M2 and M8, each patch in the texture atlas should have at least 32×32 pixels, which already cost 3KB. A few such texture patches make the size of the atlas much larger than the geometric data itself. The effective size per facade really depends on the degree of reuse of the texture patches. If the city was procedurally generated and the same component was used repeatedly, the size would be small. On the other hand, if every building in the city was different as in an accurate recreation of an existing city where most buildings are rather unique, the effective size of a facade could be a lot larger than the geometric data alone. NUT-based modeling, using the second alternative, has its advantages under such conditions because it eliminates the need of structure-dependent texture maps and only relies on truly reusable material texture maps.

In terms of rendering efficiency, we found that direct compari-

Facade	GBEF *	NUT (opt. 1) *	NUT (opt. 2)
M1	0.7K	18×3 / 0.21K	30×8 / 0.94K
M2	1.4K	46×3 / 0.54K	49×8 / 1.53K
M8	1.0K	33×6 / 0.77K	74×8 / 2.31K

Table 6: *Size comparisons between NUT and GBEF. Bounding box and triangle data have been omitted because NUT uses only 7 floating point numbers (offset, size and angle of rotation) to define a bounding box and the geometry of the bounding box is shared among all NUTs. * Texture atlas size is not included.*

son with GBEF is hard because our renderer does not run on the hardware they used for testing and we do not have access to their datasets. Nonetheless, we have included our testing results in Table 7 for reference. We have created a simplified version of our European city, which is of similar scale as the Munich model in [Haegler et al. 2010]. In our model, NUT-based facades (opt. 2) similar to M1, M2 and M8 are embedded in the front and back sides of the buildings. There are a total of 40,400 buildings and therefore 80,800 facades in the scene. Note that the mixture of buildings and the complexity of individual buildings in our model could be quite different from the dataset in [Haegler et al. 2010]. Since the facade models thus created are much simpler than the other NUT models used in other scenes, the maximum number of ray-casting iterations is set to 7.

Again, we stress that our NUT representation is capable of modeling general building structures and elements while GBEF was designed for modeling facades only.

Methods	M1	Large city size and frame time
GBEF*	2-5ms	Partial Munich, 10 - 18ms
NUT	0.2-3.8ms†	European city, 40K buildings, 5 - 6.7ms ‡

Table 7: *Rendering speed comparisons between GBEF and NUT. All experiments were tested at the resolution of* 1024×768. **Numbers from their paper (using an Nvidia Quadro4800). The viewing path and the percentage of screen coverage are unknown. † The M1 NUT model is rendered in 3.8ms at 100% screen coverage and 1.1ms at 25% screen coverage. ‡ Frame rate with 70% screen coverage. The European city NUT model uses 120MB. NUT rendering was tested on an NVidia GTX680.*

7.3 Comparisons under Distant Viewing Conditions

NUT relies on its hierarchical structure for controlling the balance of performance and quality. For buildings with small projected screen areas, it is not necessary to descend to the deepest level of the NUT hierarchy but return the *Visual Summary* S stored in NUTs at higher levels. The shader can determine at run time whether it should return early according to an estimated projected area of a redirection cell. Our representation is compared against [Cignoni et al. 2007], which was designed for distant city viewing, to analyze the rendering performance on large-scale city models when they are viewed from a distance.

We implemented their algorithm and converted our modern city model to a BlockMap representation [Cignoni et al. 2007]. Their algorithm can use a wide range of map size to produce different visual effects. We set the size of a BlockMap to 64:256 such that the total size of all BlockMaps for the city is comparable to the size of the NUT model. BlockMaps were designed to represent buildings with a projected height on the screen equal to a few pixels only. At that resolution, it is hard to compare the visual quality. Therefore, we compare the visual quality of the entire scene using the BlockMap algorithm and our NUT-oriented ray-caster but using low quality settings which render the buildings from a distance between 1km and 5km. Figure 19 shows a visual comparison between these two methods. It can be seen from the figure that although BlockMap can capture building height changes rather accurately, the 64-pixel vertical resolution makes it hard to capture vertical color changes over high-rise buildings. In contrast, the NUT model shows the windows clearly even when limited to only two levels of redirection. It can be seen from Table 8 that the NUT model also outperforms the BlockMap model in both data size and rendering speed. In addition, since BlockMap was designed for handling distant viewing conditions only, it needs to be used with another urban model to visualize a whole city and that often results in a visible seam between these two types of models. In contrast, the NUT representation caters for viewing from both short and long distances, and supports a gradual transition between coarse and detailed models.

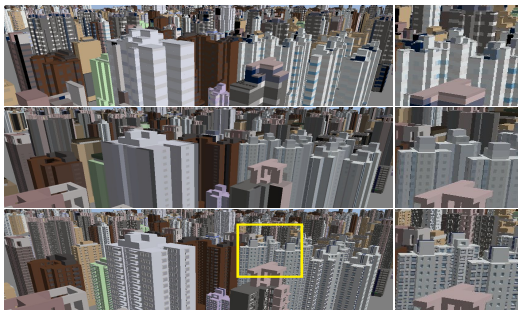

Figure 19: *Visual quality comparisons between BlockMap and low-resolution NUTs at the same viewpoint. Top row: BlockMaps with size 64:256. Middle row: NUT models limited to 2 levels of redirection. Bottom row: The original model. Images in the right column are zoom-in views of those in the left column.*

Methods	Data size	Average fps
BlockMap	72M	75
NUT	42M	156 / 85 †

Table 8: *Comparisons on data size and rendering speed between BlockMap and NUT using the modern city model. The BlockMap city model consist of 1024 64:256 maps. The frame rates were recorded for views with around 80% screen coverage. † NUT renderings with a maximum of 1 and 2 levels of redirection, respectively.*

7.4 Limitations

Designed to reduce the overall data size for modeling large-scale urban scenes, NUT has its limitations in representing certain classes of structures. Most obviously, NUT's axis-aligned and regular nature has limited its application to irregular architectural structures such as the Sydney Opera House, Melbourne's Federation Square or London's the Gherkin. NUT is not very efficient in representing certain kinds of common regular structures found in cities either. For example, supporting structures such as trusses or braced frames can only be modeled using complicated rotational redirections. Additionally, honeycomb structures, sky domes and anticlastic tensile structures often found in contemporary buildings are also difficult to model using NUTs. In all such cases, meshes would be a more effective representation over NUTs.

8 Conclusions and Further Work

We have presented a highly-efficient modeling and rendering method for large-scale urban scenes. This method divides a 3D volumetric space using a non-uniform grid which can be nested within each other to represent detailed structures in the interior and exterior of a building. It offers strong modeling capabilities while much less memory is required than the conventional mesh-based representation. Therefore it enables the transmission of large and detailed city models over the Internet for interactive visualization. Experimental results show that our method is suitable for the visualization of extremely large city models with tens of thousands of buildings.

The proposed method is particularly suitable for Building Information Modeling (BIM) for two reasons. First, its small memory footprint lends itself to modeling shareable buildings over the web for improving collaboration and enhancing productivity. Second, its inherent spatial subdivision makes a building's structural data measurable and searchable.

At present, rendering the NUT structure relies on computationally intensive ray casting. Although interactive frame rates have been achieved on modern GPUs, ray casting leaves little room for the GPU to perform other tasks. Therefore, it is desirable to develop a geometric shader on the GPU to perform on-the-fly NUT-to-mesh conversion.

Similar to other GPU ray casting algorithms, HNI ray casting does not benefit from the GPU's own anti-aliasing capability because it is designed for meshes only. At present, our ray-casting results are passed through a post-rendering image-based anti-aliasing shader FXAA [Lottes 2009] (Figure 15) before being displayed on the screen. An adaptive ray caster for NUTs that directly supports antialiasing would be very useful.

Our semi-automatic mesh-to-NUT converter presently only works for mesh models. However, building the mesh model itself is a labor intensive task and most existing buildings do not have their mesh models already built. Therefore, it will be our future research to develop a method that constructs NUTs directly from photos or laser scans in a way similar to the techniques in [Xiao et al. 2008], [Lin et al. 2011] and [Nan et al. 2010].

Acknowledgements

We would like to thank Li Cao for making building models in our sample cases. We would like to thank Yufei Li for helpful discussions. This research is supported by the State Key Program of NSFC Project (60933008).

References

ALI, S., YE, J., RAZDAN, A., AND WONKA, P. 2009. Compressed facade displacement maps. In *IEEE Transaction on Visualization and Computer Graphics*, 262–273.

ANDUJAR, C., BRUNET, P., CHICA, A., AND NAVAZO, I. 2010. Visualization of large-scale urban models through multi-level relief impostors. *Comput. Graph. Forum*, 2456–2468.

CHEN, S. E. 1995. Quicktime vr: an image-based approach to virtual environment navigation. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, SIGGRAPH '95, 29–38.

CIGNONI, P., DI BENEDETTO, M., GANOVELLI, F., GOBBETTI, E., MARTON, F., AND SCOPIGNO, R. 2007. Ray-casted blockmaps for large urban models visualization. computer graphics forum. 405413.

CRASSIN, C., NEYRET, F., LEFEBVRE, S., AND EISEMANN, E. 2009. Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '09, 15–22.

CRIVAT, B., PETCULESCU, C., AND NETZ, A. 2011. Random access in run-length encoded structures. *US Patent*, US 7952499 (May).

DICK, C., KRÜGER, J., AND WESTERMANN, R. 2009. GPU Ray-Casting for Scalable Terrain Rendering. In *Proceedings of Eurographics 2009*.

DÖLLNER, J., AND BUCHHOLZ, H. 2005. Continuous level-of-detail modeling of buildings in 3d city models. In *Proceedings of the 13th annual ACM international workshop on Geographic information systems*, ACM, New York, NY, USA, GIS '05, 173–181.

DYKES, J. 2000. An approach to virtual environments for visualization using linked geo-referenced panoramic imagery. *Computers, Environment and Urban Systems 24*, 2, 127 – 152.

GU, X., GORTLER, S. J., AND HOPPE, H. 2002. Geometry images. *ACM Trans. Graph. 21*, 3 (July), 355–361.

HAEGLER, S., WONKA, P., ARISONA, S. M., GOOL, L. V., AND MÜLLER, P. 2010. Grammar-based encoding of facades. In *Proceedings of the 21st Eurographics conference on Rendering*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGSR'10, 1479–1487.

JESCHKE, S., MANTLER, S., AND WIMMER, M. 2007. Interactive smooth and curved shell mapping. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGSR'07, 351–360.

JYLÄNKI, J. 2010. A thousand ways to pack the bin - a practical approach to two-dimensional rectangle bin packing. *Online resource at http://clb.demon.fi/files/RectangleBinPack.pdf*.

KELLY, T., AND WONKA, P. 2011. Interactive architectural modeling with procedural extrusions. *ACM Trans. Graph. 30*, 2 (Apr.), 14:1–14:15.

KOLBE, T. H., GRGER, G., AND PLMER, L. 2005. *CityGMLInteroperable access to 3D city models*. No. March. Springer, 2123.

LIN, J., COHEN-OR, D., ZHANG, H., LIANG, C., SHARF, A., DEUSSEN, O., AND CHEN, B. 2011. Structure-preserving retargeting of irregular 3d architecture. *ACM Trans. Graph. 30*, 6 (Dec.), 183:1–183:10.

LOTTES, T., 2009. FXAA. White paper, Nvidia, Febuary.

MANTLER, S., AND JESCHKE, S. 2006. Interactive landscape visualization using gpu ray casting. In *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, ACM, New York, NY, USA, GRAPHITE '06, 117–126.

MARVIE, J.-E., GAUTRON, P., HIRTZLIN, P., AND SOURIMANT, G. 2011. Render-time procedural per-pixel geometry generation. In *Graphics Interface'11*, 167–174.

MARVIE, J., BURON, C., GAUTRON, P., HIRTZLIN, P., AND SOURIMANT, G. 2012. Gpu shape grammars.

MÜLLER, P., WONKA, P., HAEGLER, S., ULMER, A., AND VAN GOOL, L. 2006. Procedural modeling of buildings. *ACM Trans. Graph. 25*, 3 (July), 614–623.

MÜLLER, P., ZENG, G., WONKA, P., AND VAN GOOL, L. 2007. Image-based procedural modeling of facades. *ACM Trans. Graph. 26*, 3 (July).

MUSIALSKI, P., WIMMER, M., AND WONKA, P. 2012. Interactive coherence-based façade modeling. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2012) 31*, 2 (May), 661–670.

NAN, L., SHARF, A., ZHANG, H., COHEN-OR, D., AND CHEN, B. 2010. Smartboxes for interactive urban reconstruction. *ACM Trans. Graph. 29* (July), 93:1–93:10.

NEBIKER, S., BLEISCH, S., AND CHRISTEN, M. 2010. Rich point clouds in virtual globes - a new paradigm in city modeling? *Computers, Environment and Urban Systems*, 508–517.

SHEN, C.-H., HUANG, S.-S., FU, H., AND HU, S.-M. 2011. Adaptive partitioning of urban facades. *ACM Trans. Graph. 30*, 6 (Dec.), 184:1–184:10.

XIAO, J., FANG, T., TAN, P., ZHAO, P., OFEK, E., AND QUAN, L. 2008. Image-based facade modeling. *ACM Trans. Graph. 27*, 5 (Dec.), 161:1–161:10.

XIAO, J., FANG, T., ZHAO, P., LHUILLIER, M., AND QUAN, L. 2009. Image-based street-side city modeling. *ACM Trans. Graph. 28*, 5 (Dec.), 114:1–114:12.

ZHOU, K., REN, Z., LIN, S., BAO, H., GUO, B., AND SHUM, H.-Y. 2008. Real-time smoke rendering using compensated ray marching. *ACM Trans. Graph. 27*, 3 (Aug.), 36:1–36:12.