Texture Data Storage Method based on Hierarchical Spatial Index

ZHOU Dongbo, ZHU Qing*, ZHANG Yeting
State Key Laboratory of Information Engineering in Surveying, Mapping and Remote Sensing, Wuhan University
{zhoudongbo, zhu66, zhangyeting}@263.net

Abstract

Aiming at the massive data organization and management of textures with various sizes for real-time rendering of large-scale 3D city models, this paper proposes a new method: a texture spatial index tree is built up based on the hierarchical spatial index of 3D geometries, and then the texture data is written into mapping files structure through depth-first and hierarchical iteration for the clustering of data blocks, which facilitates the flexible size-changed disk I/O operations. Experimental results prove such kind of storage is more compact and more effective than single file storage method, and enables the fast access of texture data.

1. Introduction

Texture data influence the sense of reality and the ability of fast perception and discrimination in virtual environment directly. Specially in large-scale 3D city models, the complexity of variety and high quality of facade textures of buildings result in extremely large amount of storage volume which makes the data organization a critical issue in real-time rendering in 3D City GIS and virtual geographic environments(VGEs) [1]. However, in general usage, the texture objects are not considered as the independent one, but as an affiliation of the material attributes of geometry for the geographic features, linked by a file name, saved as a picture in the file system or database. In practical applications, the texture data has to be loaded after the geometry data fetched to memory and further interpreted by the linked material information. If there are texture objects, the texture data would be fetched based on the linked file name in the mode of one photo per disk I/O.

The organization structure and the management of texture data for the real-time visualization of large-scale terrain surface models become mature. For example, Google Earth and Virtual Earth have become popular navigation tools of global digital elevation models (DEM) and digital orthophotos map (DOM). Most of the texture data processing methods in these applications adapt the Clipmap[2], which needs the support of special hardware and the pyramid images technology[3]. The texture images are split into regular cells or tiles and the data blocks are generated with the same size. Each block corresponds to one disk I/O operation.

Texture-Atlases[4] technology takes the reduction of texture state switching counts into consideration. For the discrete surface and various sizes of texture objects, this method combines all the textures of the same building into a big photo, and recalculates the texture coordinate for every vertex. Texture-Atlases achieves less switch numbers than Clipmap and others methods which using individual files during the render processing, but still one picture per disk I/O operation, furthermore there will generate pixels of non-use.

A BlockMap[5] represents a set of textured vertical prisms with a bounded on-screen foot-print compactly, and the BlockMaps are stored into small fixed size texture chunks and efficiently rendered through GPU ray casting. This method reduces the disk I/Os because of using less texture objects, but it is limited by the size of BlockMap object and only acceptable for objects far away from view point in special direction. The OGC CityGML[6] standard and KML[7] standard link the independent texture data of files in XML language by the file name or geo-reference code. Each texture data has to be fetched after the geometry data has been interpreted and will cause one disk I/O operation.

Making a conclusion based on above analysis, the loading time and the loading method become the key issues in real-time rendering in large-scale 3D city models. In this paper, the hierarchical organization method and compact texture data storage format are combined to load the texture data in asynchronism and flexible size-changed disk I/O operations.

2. Texture data organization method base on hierarchical spatial index

For the relationship between the texture images and the geometry surface of the 3D city models, the texture data should be considered as an object independently and the spatial index should be built the same way as the geometry data [8].

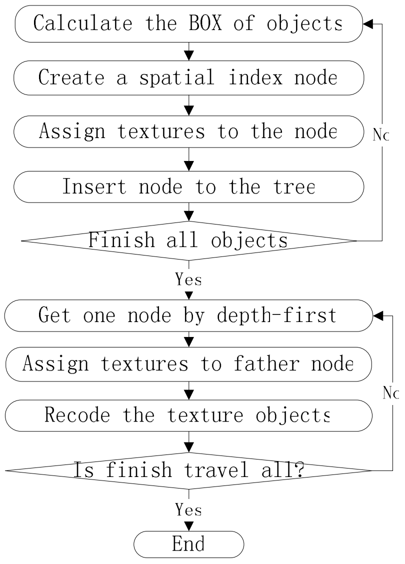
Calculate the BOX of objects
Create a spatial index node
Assign textures to the node
Insert node to the tree
Finish all objects
Nc
Yes
Get one node by depth-first
Assign textures to father node
Recode the texture objects
Is finish travel all?
Nc
Yes
End

Fig. 1 Steps of hierarchical spatial index of texture data

When using a bounding box to query the objects through the spatial index, the texture data and the geometry data should be return back at the same time. All of the texture data of one geo-feature object should be saved as a block, thus it can remove the needless disk seeking I/Os. Furthermore the efficient storage strategy should be implemented to reduce the abundant texture data which share the same material attribution.

2.1. Hierarchical spatial index of texture data

There are two kinds of methods to build the spatial index. The first method is about space decompose, which decomposes the space into small parts from the top to bottom, such methods including Quad-tree, Oct-tree, BSP tree or k-D tree. The second one uses the bounding box of the geo-feature to create the leaf node, and then cluster them from the bottom to top into the tree. This method includes bounding volume hierarchies (BVH), hierarchical bounding sphere (HBS) and R-tree. Each geo-feature object should be regarded as an independent semantic object in the large-scale 3D models environments. Therefore, the spatial index should be built based on the bounding box of geometry of the geo-feature object. As for one building, all of its texture data and the geometry data share the same bounding box, so they should share the same spatial index. In fact, this paper build the texture objects spatial index the same as that of the geometry objects. The detail steps are shown one Fig. 1.

Firstly, the whole scene should be traveled to get every 3D city model and the bounding box is calculated. Then the bounding box is used to create a spatial index node and after that the texture data is assigned to the node. Further, the node is inserted into the spatial index tree. When all of the objects are visited and processed, the spatial index tree will be traveled in a depth-first mode to cluster the texture data in the leaf node to its father node.

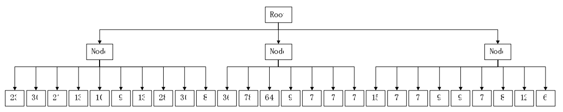

Fig. 2 Texture tree structure based on R-tree(number means texture images in node)

R-tree[9] spatial index is chosen to organize the 3D city models. Buildings, trees, city facilities and other geo-features are calculated the bounding box to create the leaf node and reassigned the texture objects and then inserted to the spatial index tree. Fig. 2 shows the result for the texture data organization of a small urban area.

2.2. Algorithm of processing shared textures

Surfaces with the same texture information always link to the same texture photo through the file name in the 3D city models. So each texture photo has just one file in the file system or database system. But when a texture spatial index tree was built, the texture object belonging to different nodes will be stored as a copy of the photo in every node, so it will generate the large data redundancy and will cause a large number of abundant and repeat disk read operations for the same data, which will dramatically reduce the efficiency of the disk I/O. Algorithm must make sure that every texture data stores once in the tree. To take advantage of the hierarchical pattern of the spatial index tree, shared texture objects in the leaf node are promoted to the father node and the copies in the children nodes are removed.

Fig. 3 shows us the steps to remove the repeat copies of the shared texture data. In the left part, the leaf nodes in blue share the same texture data and the same for the red nodes. The right figure shows the result that the shared texture data in blue nodes is clustered to the non-leaf node and that in red nodes is cluster to the root node.

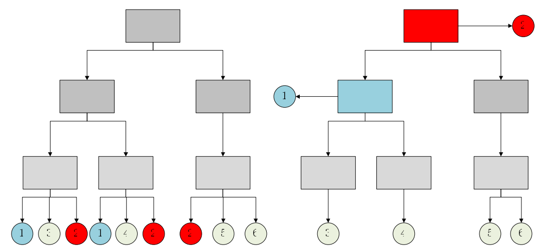

Fig. 3 The procedure of cluster shared texture to father nodes (same color node show same texture shared)

According to this algorithm, the texture shared in the children's node in the Fig2, should be promote to the father node show in Fig4.

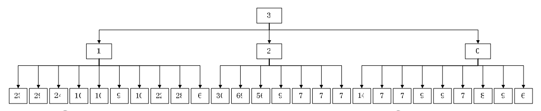

Fig. 4 Cluster the texture to the father's node, number in the node means the textures in this node, totally to 438

3. Texture data storage method based on texture spatial index tree

In general, the texture data always stores as a picture file. The format can be identified though the extension. So there will be a serial of disk I/Os for every texture file especially in a mode of loading then parsing. For example, when reading a BMP format file from disk, *fread* operation first gets the head information to memory, and another *fread* operation will get the pixel data of the BMP file according to the head information. In this way, every *fread* operation may need more than one disk seeking, rotating and transferring I/O operations.

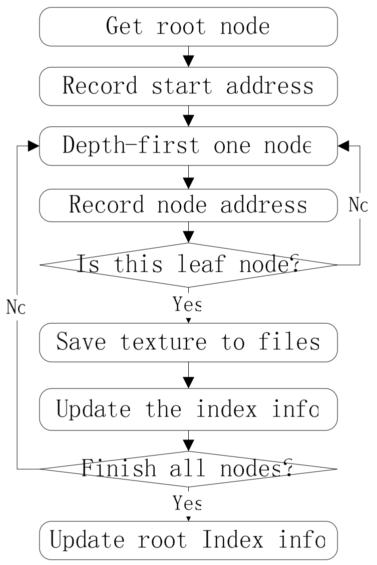

Fig. 5 Texture data memory mapping file storage

This paper proposes a storage method which combines all texture files in the spatial index node into a data block, and saves the address information of the data block in the spatial index. When querying objects though the spatial index, the data block address information can be get from the answered nodes, which will be used to load the data in one disk I/O. All of the texture objects can be parsed in memory. The spatial index tree records the address information of the texture data in leaf node, of the block data in middle node and of the whole data in root node. So the disk loading I/O is changeable. The data loading can be in object granularity, block data granularity and whole data block granularity disk I/O. Which one should be chosen is based on the efficiency of the data loading in the application. Fig.5 shows the detail steps of the algorithm. When reaching a leaf node by traveling from the root node in depth-first mode, a data block is created to include all the texture objects in the node. After that the data block will be written into a memory-mapping file and the address information of block in the file will be assigned to the index node. The memory-mapping file that holds the texture data will be closed after visiting all the nodes of the spatial index tree.

For supporting the changeable disk I/O granularity during loading, the texture datum and blocks are written into the file in the sequence of visiting the spatial index tree as the same way of querying. So the data blocks of the children's will be continuous in physical storage. For the sake of parallel loading, the spatial index node with information of location address of nodes will be saved as a compact file. We defined some structures to store the index information for nodes. The *CTexNodeItems* in the memory corresponds to a texture data in the disk, and which also records the type, ID and the start address and offset value in the texture memory-mapping file. Then the *CTexIndeNode* is equal to the block data, which holds the start address, the offset value, the counts of the texture data. The start address and the offset value of the father node including all the texture data of its children in the memory-mapping file make it suitable to load the data in a changeable disk I/O granularity.

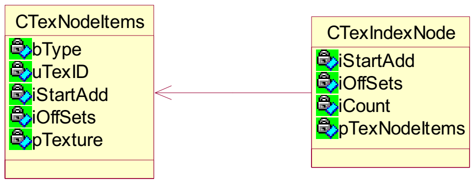

Fig. 6 Texture Item and Texture data block structure

Fig. 7 shows us the mapping of the texture spatial index tree to the memory-mapping file. Firstly, the root node including 3 texture data has been saved into the first part of the file, and then the nodes are traveled in depth-first. The data blocks of the node in the tree are saved on the same order of traveling. During the navigation, the visual node can be queried by spatial index and then the *iStartAdd* and *iOffSets* stored in the index node can be used to load the texture data in one disk I/O. Otherwise, only one single texture data in the first node was needed, the *iStartAdd* and *iOffSets* value of the it will be set to load by one smaller disk I/O granularity.

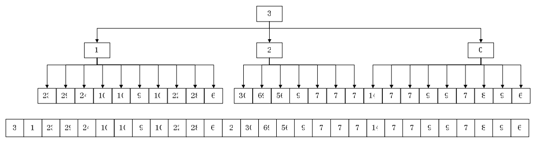

Fig.7 The mapping of texture spatial index tree to texture memory-mapping file

4. Test and Analyses

4.1. Test environment

The test environment is as below: windows XP (sp3) operating system, a 3GB memory, a CPU of Intel Core2 Duo with 2.53GHz and a hard disk with size of 320GB (7200rpm).

The test data include a typical urban area with hundreds of buildings, trees and city facilities. Geometry data size is 27.6MB in total and texture data size is 342.3MB that are stored as 438 photos.

4.2. Data preprocessing

The original geometry data comes from *.obj format, and the texture photo stores as *.bmp files. During the preprocessing we parse the *.obj file to get every object then using the box of the object to construct the spatial index. Texture data included in each geo-feature object is extracted and assigned to the node created by the bounding box information.

4.3. Test results

Table 1. Texture data loading time

Load mode	I/Os	Time（s）
Texture Geometry one I/O	1	6.20
Node per I/O	29	6.32
Per bmp file	438	11.9

Table 1 shows us the loading time of the texture data. When load all 438 photos in BMP format, the total time is 11.9 seconds. Otherwise, when we use the new method of this paper, we design two situations, the first is that all of the data loading using the largest disk I/O granularity within one disk I/O, it need 6.2 seconds. And the second is that in the disk I/O granularity of node data block, totally 29 blocks, it needs 6.32 second. The methods of this paper can achieve the reduction of 5.7 second using the largest disk I/O granularity 5.58 for a smaller disk I/O granularity. It is obvious show us the improved effect.

The organization method of texture data and the storage structure can not only use to load data in changeable disk I/O granularity for a smart fetching, but also loading the geometry data and texture data in the same time. Table 2 gives the contract of two methods for loading geometry and texture data synchronism. With the origin method, only when the geometry data read to the memory and finish parsing, we can get the texture information then to load them. So the total time is all of the geometry data and texture data loading time.

Table 2. Loading time for geometry and texture

		1	2	3	4	5	6
Geo MB		1.8	3.9	5.1	4.6	1.9	1.4
Tex MB		23.6	60.6	44.1	17.3	19.9	22.8
Tex num		30	69	56	22	28	29
01	¬ Geo s	0.041	0.078	0.11 1	0.112	0.02 8	0.06 4

	Tex s	0.411	1.093	0.784	0.277	0.391	0.407
	Time s	0.452	1.171	0.98	0.38	0.41	0.47
New	Geo s	0.039	0.078	0.111	0.112	0.028	0.064
	Tex s	0.411	1.093	0.784	0.277	0.391	0.407
	Time s	0.411	1.093	0.78	0.277	0.39	0.40

In our method, when the spatial index node is identified, the geometry data and the texture data can be loaded at the same time when they location on different physical hard disks. So the total time is the larger one between the geometry and texture data.

Table 3. Comparison of loading different scenes

	Geometry	texture	Tex num	Geometry times	Texture time
Scene 1	159MB	0	0	2.377 s	0
Scene 2	9.10MB	61MB	126	1.442 s	0.788 s

Table 3 gives us the comparison results of the geometry and texture data loading time, in scene 2, the spatial index has 126 nodes with the data size about 80MB, but it only smaller for 0.147 second. The scene 2 has a high efficiency in loading texture data.

Table 4. Comparison of shared texture data loading

	count	Data size	Fetch time	Tree nodes	Repeated texture	Fetch time
1	572	62.2MB	1.43s	126	7076	0.788s
2	438	342MB	11.9s	29	42	6.32s

Table 4 gives us the share texture in different scene. In scene 2, the shared texture more than 7000 times, because of the same kinds of tree. Using our method, we get only 126 nodes, and cluster 25 texture data to root node and all of other non-leaf nodes store all other share photo. Results show that our method is suitable for the scene with large numbers of shared texture.

5. Conclusion

This paper proposes a texture data organization structure and storage method for real-time rendering for large-scale 3D city models. The spatial index is built by using the city model geometry information and then extracting the texture data from the model. A changeable disk I/O granularity block data is built based on the spatial index by traveling in depth-first. Companying with frustum to calculate the visible spatial index node, we can use the multi-disks, multi-threads to get the texture and geometry in synchronism. The experimental results prove the good performance of this method.

Acknowledgments

This work was supported by 973 program (2010CB731801), 863 program (2008AA121600), the leading academic discipline project of Shanghai Educational Committee (No.J50104), NSFC (40701144 and 40871212)

References

[1] Q. Zhu and H. Lin, CyberCity GIS: 3D city models for virtual city environment, Wuhan: Wuhan University Press, 2004, p. 282.
[2] C. C. Tanner, C. J. Migdal and M. T. Jones, "The clipmap : a virtual mipmap," in Proc. 1998 Computer graphics and interactive techniques, pp. 151-158.
[3] D. J. Heeger and J. R. Bergen, "Pyramid-based texture analysis/synthesis," in Proc. 1995 Computer graphics and interactive techniques, pp. 229-238.
[4] H. Buchholz and J. Dollner, "View-Dependent Rendering of Multiresolution Texture-Atlases," in Proc. 2005 Visualization, 2005. VIS 05. IEEE, pp. 215-222.
[5] P. Cignoni, M. Di Benedetto, F. Ganovelli, E. Gobbetti, F. Marton and R. Scopigno, "Ray-Casted BlockMaps for Large Urban Models Visualization," Computer Graphics Forum, vol.26, 2007.
[6] OpenGIS City Geography Markup Language (CityGML) Encoding Standard, OGC 08-007r1, 2008.
[7] OGC® KML, OGC 07-147r2, 2008.
[8] S. Zlatanova, "3D GIS for Urban Development," Doctor of the Technical Sciences dissertation, Graz University of Technology, Graz, Austria, 2000.
[9] M. Kofler, M. Gervautz and M. Gruber, "R-trees for organizing and visualizing 3D GIS databases," The Journal of Visualization and Computer Animation, vol.11, pp. 129-143, 2000.