Quick Rendering of Outdoor Small Objects and Persistent Research and Implementation

YE Li-na
Computer center of public course，Wuhan Institute of Shipbuilding Technology
Wuhan，China
e-mail：504292159@qq.com

Abstract—This paper first introduces the method of drawing an efficient outdoor small object group widget batch, then design a representation to the method of creating widget visible set search strategy based on improved BSP tree and its corresponding to the efficient use of memory, and finally achieve the efficient and fast rendering terrain surface small objects in a demo display using this method.

Key words-outdoor small objects; Quick render; persistent；BSP tree; Batch(key words)

Anybody who played 3D games knows that small objects in 3D games are usually immobile objects(they can't participate in the interaction), they exist only to increase the depth of feeling the ground. Although these small objects with the new regional activities can randomly generated scenarios, but astute players in a position to return to the time before may notice some subtle differences. In addition, artists usually want to control the appearance of all levels affect things, so how to define better what type of small objects will appear in the piece of the region, as well as the frequency in which they appear, and in extreme cases, a separate place some small objects to achieve special look for terrain rendering part also can't be ignored. Another problem can't be ignored is that their destructiveness: though small object itself is not moving, but the players hope a scene incident will affect them. For example, the game will be a huge explosion by changing the original terrain texture to the performance of the damage suffered by the ground, so if the grass unharmed in the blast will be very absurd, it can be temporary or permanent removal of small independent from the scene small objects or groups of objects help to increase immersion.

For how to efficiently render small objects, we need to pay attention to two issues:

1) How to make small objects that was generated quickly fall on the visually truncated body;

2) How to draw results list mode efficiently.

I. Introduction

Although our goal is to render highly complex, covered with small objects of the scene, the independent grids that make up the small objects don't have to be complicated. In fact, the most effective models as objects of small objects (we call widget) are usually only a few single-sided polygons and textures. This simplicity allows us to render much more objects than other methods, thereby we can increase the overall complexity of the scene. Fig .1 shows a widget that can be used as a simple grid, which consists of eight-sided triangle and a texture components.

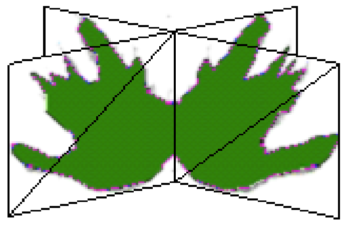

Figure 1. A simple ground cover grid

Although the grid itself is very simple, but it's not easy to draw them. Now, the graphic hardware can render a considerable number of triangles. But only when the unit is rendered block by thousands of vertices can we get the best performance. Rendering thousands of copies of eight different triangular mesh is not the best use of the graphics hardware.

II. Efficient batch rendering of the widget

A．Widget set

Usually, it's ok that we use the pre-conversion widget fill the vertex buffer, in fact, you can find a way to make the graphics hardware with the correct matrix transformation for each widget. This is similar to skin problems, each vertex in time will transform the role of a matrix version of the index and then multiple copies of the same widget join buffer. Each independent widget it contains every vertex must have the same index value, but continuous widget may have different values. Using such an index system, as long as there is a constant matrix vertex space, you can put as many widget into a single vertex buffer. This method introduced by Gosselin[1] et al., Used to render the role of large, but because the skeletal animation, so during one API call, it can only render four characters. For small objects, in one API we call can be rendered more widget. The following structure describes the possible vertex format of the widget:

```
typedef struct
{
	float position[3];
	float uv[2];
	u32 strIndex;
}WIDGETVTX;
```

Because vertex buffer contains multiple widget, so we have to build an index buffer contains a single strip to draw all widget. We must remember that we should increase the number of vertices of a widget an each successive widget base vertex to be addressed to the correct set of vertices.

B. *Draw widget set*

When we completed the vertex and index buffers, we can draw multiple widget in a single draw call a. To achieve this, you first need to generate a set of widget that must be drawn and a transformation matrix corresponding to them, and then the transformation matrix of the first widget is sent to the corresponding vertex shader constant registers, and drawing primitives. If the widget's number to the number that can be sent in a single group, just send many groups that contain as much widget as possible.

III. FAST RENDERING WIDGET AND LASTING WAY TO ACHIEVE

After mastering how to draw a lot of widget efficiently, followed by the method that is to solve how to generate the corresponding instance, according to one point of view which is always changing, and this will be discussed in detail in the next section of a position in the scene all the widget by precomputed and to establish an efficient method BSP tree representation method to generate visible widget set quickly.

We call the method that is used to store the scene widget location BSP tree. Use BSP tree, rather than quadtree or octree, making the scene become irregular shape, and spend no extra memory storing the empty node. BSP tree is a hierarchical composition of the surface, each plane would cut scene into two pieces, each object on the scene surface is at each side or the other side[2]. Fig .2, which shows an example of the first two planes of BSP tree.

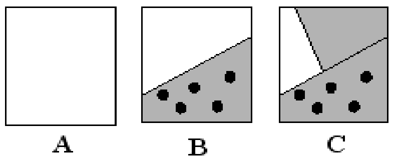

Figure 2. BSP tree structure

As can be seen from the figure, figure A shows the entire scene, a panel divides the entire scene into upper and lower ports in the figure B, panel C and B in another plane perpendicular to the plane of division of the whole turn the upper part of the scene being further subdivided into more detail about the two parts, this method divides the objects in the scene into a binary tree.

A. *Improvement strategies based approach represents efficient use of memory BSP tree*

Enhance the utilization of the CPU cache is to reduce the size of the BSP node structure and the use of some type of memory predictable access patterns. Structure defined node checks the beginning, which shows that it is divided into two roughly equal parts: the first part contains the division planes, while the second part contains the position of the child node.

To ensure that each selected plane is axis aligned with one of the three axes, we can greatly reduce the storage required for each node divided plane memory. This plane can be reduced to the desired 16 bytes to 4 bytes (to save the plane from the origin), and two bits (for which is represented by the axis), the remaining number of 30 to store a pointer to the tree . In addition, in order to save memory consumption for storing pointers, we also need to use some of the memory access patterns for tree query. If we can ensure a child node of given node can be saved in the parent node, we don’t have to need preceding node pointer. Finally, in order to reduce the memory that was saved to point next nodes, you can save it as an absolute displacement distance of the current node, these changes make the definition of BSP node only need 8 bytes total, its structure is as follows:

```
typedef struct
{
	u32 axie:2;
	u32 numFrontLeaves:4;
	u32 numBackLeaves:4;
	u32 backNodeOffset:22;
}WIDGETNODE;
```

B. *Improved build BSP tree*

After creating an array of all the widget scene, the system will calculate an axis-aligned bounding box for them, and then be able to find the longest axis of the box on each node. This axis will become division planes in the node ordering their location and distance between the axes follow after, then place on the plane between two middle widget. For example: There are 63 scene widget, split plane will be placed between the first 31 and second 32 widget. In this manner continue to build nodes, until each node has only a small amount of widget, finally, this node is called leaf node. Here is the structure, said:

```
    typedef struct
    {
            float position[3];// widget position
            s8 sinAngle;// Toward the sin (* 127)
            s8 cosAngle;//_Toward the cos(*127)
            scale;//_widget zoom ratio * 32
            pad;
    }WIDGETLEAF;
```

Here we describe CWidgetBSP bsp.h class through widgets bsp.cpp and widgets to be responsible for establishing and managing a widget BSP tree. The parameter of number function in code is a WIDGETLEAF structure array, the system will create a BSP tree based on them. Once the trees are established, we can efficiently search to find a list of widget that must be drawn. Here is the core algorithm of the process:

```
class CWidgetBSP
{
    public:
        CWidgetBSP();
       ~CWidgetBSP();
    void   CreateTree( CWidgetMesh *pMesh, WIDGETLEAF
*pLeaves, u32 uNumLeaves );
     void  DestroyTree();
    // Construct an observation box
     void  Draw( const D3DXMATRIX &rViewMtx, const float
fov[2], float fFar=50.0f );
protected:
     void  DrawNode( const WIDGETNODE *pNode );
     void  DrawLeaves( const WIDGETLEAF *pLeaves, u32
uNumLeaves );
     void  CalculateViewBox( const D3DXMATRIX
&rViewMatrix, const float fFov[2], float fFar );
            u32   m_uTreeLength;
     CWidgetMesh    *m_pMesh;
     WIDGETNODE *m_pTree;
     float  m_viewBox[3][2];
};
#endif
……
void CWidgetBSP::CreateTree( CWidgetMesh *pMesh,
WIDGETLEAF *pLeaves, u32 uNumLeaves )
{
     u32   uNumNodes,uLength;
     void  *ptr;
     assert(sizeof(WIDGETNODE)==8);
     assert(sizeof(WIDGETLEAF)==16);
     DestroyTree();
     // Here you must create at least two branches for a tree
     if( uNumLeaves>1 )
     {
            uNumNodes =
(uNumLeaves+(LEAVESPERNODE-1))/LEAVESPERNODE;
            uNumNodes = uNumNodes*2;
            m_uTreeLength =
uNumNodes*sizeof(WIDGETNODE) +
uNumLeaves*sizeof(WIDGETLEAF);
            m_pTree =
(WIDGETNODE*)malloc(m_uTreeLength);
            if( m_pTree )
            {
                   m_pMesh = pMesh;
                   // Create a tree
                   ptr =
_CreateTree((void*)m_pTree,pLeaves,uNumLeaves);
                   uLength = ((u8*)ptr) - ((u8*)m_pTree);
                   assert(uLength<=m_uTreeLength);
            }
     }
    }
    …….
CWidgetBSP::~CWidgetBSP()
{
     DestroyTree();
}
void CWidgetBSP::DestroyTree()
{
     FREE(m_pTree);
     m_uTreeLength = 0;
     m_pMesh = 0;
}
……
```

C. BSP tree search strategy

When we have a compact, efficient memory access BSP tree definition, in relation to the study of how to calculate the rendering widget. For this reason, all the cones must be within a certain distance and the viewpoint of the widget from a BSP tree. We can use the size of the viewport, the camera field of view to construct a matrix and the axis-aligned box to encompass all such areas, as shown in Fig .3.

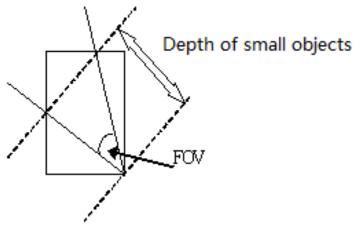

Figure 3. An axis-aligned box observation

With this box, search BSP tree becomes simpler. At each node, we can put the distance associated with it and the corresponding observation axis cassette precomputed maximum and minimum values are compared, in order to ensure access to which one or more child nodes. Note that, in order to maintain cache coherence, to access this first node of the child node we must accessed before the next child node.

After passing the observation parameters and maximum drawing distance, CWidgetBSP Draw class member function will construct an observation box, and then perform a search based on the viewpoint, for each widget found in the observation box will call the appropriate class of widget BSP tree CWidgetMesh the AddInstance member functions.

Widget will eventually render quickly and permanently. The following is the core of this part of the algorithm:

```
    void CWidgetBSP::DrawLeaves( const WIDGETLEAF
*pLeaves, u32 uNumLeaves )
     {
           while( uNumLeaves-- )
      {
            m_pMesh->AddInstance(     pLeaves->position,
                                      pLeaves->scale/32.0f,
                                      Leaves->sinAngle/127.0f,
                                    pLeaves->cosAngle/127.0f);
            pLeaves++;
      }
     }
     ……
```

IV. Implementation and operating results of the widget

Through the above exposition we can basically achieve a fast rendering and persistent examples of small objects, operating conditions can execute vertex shader and pixel shader in PC graphics hardware to version 2.1 or higher on the basis of this article using the above method to achieve a demo example, its operating results as shown in Fig .4, the scope of this example to draw grass on the ground can be flexibly designated or randomly generated by the program can be.

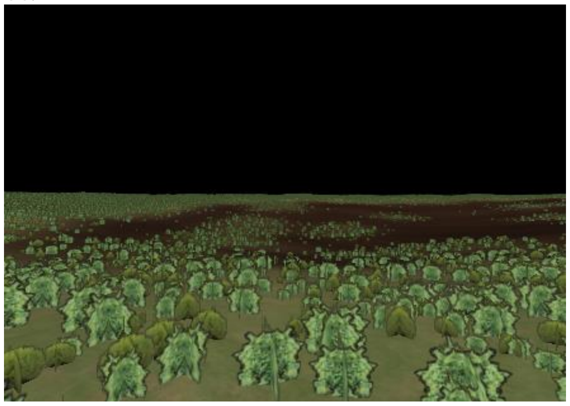
Figure 4. Fast rendering renderings outdoor small objects

References

[1] [Gosselin04]Gossdlin,David,Pedro V.Sander,and Jason L.Mitchell,"Drawing a Crowd:Instancing in Current Hardware."In Shader X3(edited by Wolfgang Engel)[J].Charles River Media,2004.

[2] Samuel Ranta-Eskola.Binary Space Partitioning Trees and Polygon Removal in Real Time 3D Rendering[D].Sweden:Information Technology Computing Science Department Uppsala University.2001.

[3] Heinzle S, GreisenP, Gallup D. Computational stereo camera system with programmable controlloop. ACM Transactionsons on Graphics, 2011, 30(4) : Article No.94.

[4] Liu Wei, Wu Yi-Hong, Hu Zhan-Yi. Survey of 2D to 3D conversion technology for film. Journal of Computer-Aided Design and Graphics, 2014, 24(1): 14-28 (in Chinese).

[5] Northam L, Asente P, Kaplan CS. Consistent stylization and painterly render of sterescopic 3D images//Proceedings of the 10th Non-Photorealistic Animation and Renderng(NRAR'12). Annecy, French, 2012: 47-56

[6] Liu Bo, Wang Zhangye, Wang Liying, et al. Efficient modeling and real-time rendering of large-scale urban scenes [J]. Journal of Computer–Aided Design &Computer Graphics, 2008, 20(9):1153-1162 (in Chinese).

[7] Peng C, Cao Y. A GPU-based approach for massive model rendering with frame-to-frame coherence[J]. Computer Graphics Forum, 2012, 31(2pt2): 393-402.

[8] AlHalawani S, Yang Y L, Liu H, et al. Interactive facades analysis and synthesis of semi-regular facades[J]. Computer Graphics Forum, 2013, 32(2pt2): 215–224.

[9] Ma Chunyong, Chen Yong, Han Yong, et al. A GPU-based rendering acceleration algorithm for urban simulation[J]. Periodical of Ocean University of China: Nature Science Edition, 2010, 40(7), 141-144+158 (in Chinese).

[10] Krecklau L, Born J, Kobbelt L. View-dependent realtime rendering of procedural facades with high geometric detail[J]. Computer Graphics Forum, 2013, 32(2pt4): 479-488.

[11] Hujun Bao and Wei Hua. Real-Time Graphics Rendering Engine[M. Hangzhou: Zhejiang University Press, 2010: 258-260.

[12] Yongli Zhu, Research of Mesh Simplification and Visualization for Terrain[D]. Henan : Henan Polytechnic University, 2012.

[13] Simulation of 3D Fountain Visualization Based on Particle System. MEC2013: The 2013 2nd International Conference on Mechatronic Sciences, Electric Engineering and Computer 2013,12(7):2638~2641.

[14] Okabe M, Anjyo K, Igarashi T, et al. 2009. Animating pictures of fluid using video examples[C]//Computer Graphics Forum. Blackwell Publishing Ltd, 28(2): 677-686.