Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2009, Article ID 730894, 7 pages
doi:10.1155/2009/730894

Research Article

Fast and Reliable Mouse Picking Using Graphics Hardware

Hanli Zhao,[1] Xiaogang Jin,[1] Jianbing Shen,[2] and Shufang Lu[1]

[1] *State Key Lab of CAD & CG, Zhejiang University, Hangzhou 310027, China*
[2] *School of Computer Science & Technology, Beijing Institute of Technology, Beijing 10008, China*

Correspondence should be addressed to Xiaogang Jin, jin@cad.zju.edu.cn

Received 15 December 2008; Accepted 4 March 2009

Recommended by Zhongke Wu

Mouse picking is the most commonly used intuitive operation to interact with 3D scenes in a variety of 3D graphics applications. High performance for such operation is necessary in order to provide users with fast responses. This paper proposes a fast and reliable mouse picking algorithm using graphics hardware for 3D triangular scenes. Our approach uses a multi-layer rendering algorithm to perform the picking operation in linear time complexity. The objectspace based ray-triangle intersection test is implemented in a highly parallelized geometry shader. After applying the hardware-supported occlusion queries, only a small number of objects (or sub-objects) are rendered in subsequent layers, which accelerates the picking efficiency. Experimental results demonstrate the high performance of our novel approach. Due to its simplicity, our algorithm can be easily integrated into existing real-time rendering systems.

1. Introduction

Mouse picking, as the most intuitive way to interact with 3D scenes, is ubiquitous in many interactive 3D graphics applications, such as mesh editing, geometry painting and 3D games. In many Massive Multi-player Role Playing Games (MMRPGs), for instance, thousands of players compete against each other, and the picking operation is frequently applied. Such applications require picking to be performed as fast as possible in order to respond to players with a minimum time delay. In recent years, programmable graphics hardware is getting more and more powerful. How to make full use of the co-processors in the picking operation becomes important.

The WYSIWYG method, which takes advantage of graphics hardware to rerender scene objects into an auxiliary frame buffer, was first proposed by Robin Forrest in the mid-1980s and used in 3D painting by Hanrahan and Haeberli [1]. In their method, each polygon is assigned a unique color value which is used as an identifier. Given the cursor position on the screen and the id buffer, the picked position on the surface can be found by retrieving data from the frame buffer. However, this approach has weaknesses for complex scenes in that all objects in the view frustum must be rerendered. This may take a long time for complex scenes and therefore lower the picking performance. By integrating the WYSIWYG method and hardware bilinear interpolation [2], Lander presented a method to calculate the exact intersection information, that is, the barycentric coordinate in the intersected triangle. By setting additional color values with $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$ (normalized with floating-point precisions) to the three triangle vertices respectively, he calculated the barycentric coordinate by interpolation after the rasterization stage. However, the computed barycentric coordinate is in the projected screen-space but not in the object-space, which may restrict its application.

In this paper, we propose a simple, fast and reliable picking algorithm (FRMP) using graphics hardware for 3D triangular scenes. By combining the multi-layer culling approach of Govindaraju et al. [3] with a GPU-based implementation of Möller and Trumbore's ray-intersection test [4], the picking can be performed in linear time complexity. Our approach has the following features.

(1) It is fast—our approach is 2 to 14 times as fast as the traditional GPU-based picking one.

(2) It is reliable—our approach performs the operation in object-space, and the exact intersection information can be computed.

(3) It is parallel—the ray-triangle intersection detection is implemented as a geometry shader.

(4) It is simple—our novel approach operates directly on triangular meshes and can be easily integrated into existing real-time rendering systems.

The rest of the paper is organized as follows. Section 2 reviews some related work. Section 3 describes our new algorithm, whereas experimental results and discussions are presented in Section 4. We conclude the paper and suggest future work in Section 5.

2. Related Work

Intersection detection is widely used in computer graphics. The mouse picking operation can be performed by an ordinary ray-object intersection test and accelerated by lots of schemes for high efficiency.

The methods for interference detection are typically based on bounding volume data structures and hierarchical spatial decomposition techniques. They are K-d trees [5], sphere trees [6, 7], AABB trees [8, 9], K-DOPs trees [10], and OBB trees [11]. The objects (triangles) are organized in clusters promoting faster intersection detection. The spatial hierarchies are often built in the preprocessing stage and should be updated from frame-to-frame when the scene changes, which is not appropriate in most cases for mouse picking.

Hardware occlusion queries are also used in collision detection for large environments to efficiently compute all the contacts at high frame rates by Govindaraju et al. [3, 12, 13]. These GPU-based algorithms use a linear time multi-pass rendering algorithm to compute the potentially colliding set. They even achieve interactive frame rates for deformable models and breaking objects. In their method, the objects (triangles) list can be traversed from the beginning up to the end and thus no spatial organization (KD and other trees) are required. The WYSIWYG method for mouse picking, which was first proposed by Robin Forrest in the mid-1980s and used in 3D paint by Hanrahan and Haeberli [1] and further studied by Lander [2], Akenine-Möller and Haines [14], belongs to this class. Its efficiency is high in many cases. However, it has limitations as discussed in the introduction section. CPU methods for picking objects were introduced by [15] in the Direct3D platform and by [16] in the OpenGL platform. However, their efficiency decreases dramatically as the number of input primitives increases. Motivated by the multi-layer culling approach of Govindaraju et al., we do not construct a time consuming hierarchy. Instead, we use a multi-layer rendering algorithm to perform a linear time picking operation. In this paper, we perform the exact object-space-based ray-triangle intersection test [4] in a geometry shader by taking advantage of its geometric processing capability. The overall approach makes no assumptions about the object's motion and can be directly applied to all triangulated models.

Some acceleration techniques for real-time rendering need to be applied in our method. Triangle strips and view frustum culling were introduced by [17, 18], respectively. It is possible to triangulate the bounding boxes of objects as strips and to cull away objects that are positioned out of the view frustum. Hardware occlusion queries for visibility culling were studied by [19–21]. GPU-based visibility culling is also important in our algorithm.

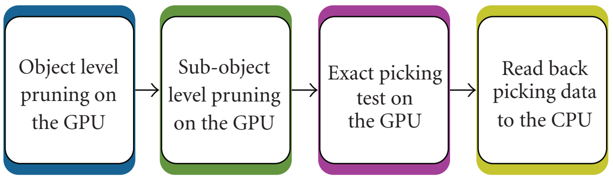
Object level pruning on the GPU
Sub-object level pruning on the GPU
Exact picking test on the GPU
Read back picking data to the CPU

Figure 1: Algorithm workflow.

3. Hardware Accelerated Picking

Our mouse picking operation takes the screen coordinate of the cursor and the scene to be rendered as input, and outputs the intersection information, such as object id, triangle id, and even the barycentric coordinate of the intersection point. In this section, we first present an overview of our algorithm and then we discuss it in detail.

3.1. Algorithm Overview. Our FRMP method exploits the new features of the 4th generation of PC-class programmable graphics processing units [22]. Figure 1 illustrates the algorithm workflow. The overall algorithm is outlined as follows.

(1) Once the user clicks on the screen, compute the picking ray origin and direction in the view coordinate system.

(2) Set the render target with one-pixel size.

(3) Set the render states *DepthClipEnable* and *DepthEnable* to *FALSE*.

(4) After the view frustum culling, render the bounding boxes of the visible objects. We issue a boolean occlusion query for each object during this rendering pass.

(5) Render the bounding boxes of all sub-objects whose corresponding occlusion query returns *TRUE*. Again we issue a boolean occlusion query for each sub-object during this rendering pass.

(6) Reset the states *DepthClipEnable* and *DepthEnable* to *TRUE*.

(7) Render the actual triangles whose corresponding occlusion query returns *TRUE*. Now we only issue one occlusion query for all triangles.

(8) If the occlusion query returns *TRUE*, trivially read back the picking information from the one-pixel-sized render target data; otherwise, no object is picked.

The novel multi-layer rendering pass on programmable graphics shaders is outlined below:

(1) Transform the per-vertex position to the view coordinate system in the vertex shader.

(2) Perform the object-space-based ray-triangle intersection test in the geometry shader, output a point with picking information if the triangle is intersected. The x- and y-components of the intersection point are set to 0, and the z-component is assigned as the depth value of the point. Then the point is passed to the rasterization stage.

(3) Output the picking information directly in the pixel shader.

3.2. New Features in the Shader Model 4.0 Pipeline. The *Shader Model* 4.0 fully supports 32-bit floating-point data format, which meets the appropriate precision requirement for general purpose GPU computing (GPGPU). The occlusion query can return the number of pixels that pass the z-testing, or just a boolean value indicating whether or not any pixel passes the z-testing. In our case, we only need the boolean result that whether some objects are rendered or none are rendered.

The *Geometry Shader*, which is first introduced into the shader model 4.0 pipeline, takes the vertices of a single primitive (point, line segment, or triangle) as input and generates the vertices of zero or more primitives. The input and output primitive types need not match but they are fixed for the shader program. We use a triangle as the input primitive, as the ray-triangle intersection detection needs to be implemented here. We get a point as output. If the intersection test is passed, a point primitive with intersection information is returned. If the test is failed, no point is output.

3.3. Intersection Test in the Geometry Shader. In this section, we present the ray-intersection test introduced by Möller and Trumbore [4]. We implement the algorithm in a geometry shader by taking advantage of its geometric processing capability.

A ray, $\mathbf{r}(t)$, is defined by an origin point, $\mathbf{o}$, and a normalized direction vector, $\mathbf{d}$. Its mathematical formula is shown in (1):

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}. \tag{1}$$

Here the scalar, t, is a variable that is used to generate different points on the ray, where t-values of greater than zero are said to lie in front of the ray origin and so are a part of the ray and negative t-values lie behind it. Also, since the ray direction is normalized, a t-value generates a point on the ray that is t distance units away from the ray origin.

When the user clicks the mouse, the screen coordinates of the cursor are transformed through the projection matrix into a view-space ray that goes from the eye-point through the point clicked on the screen and into the screen.A point, $\mathbf{t}(u, v)$, on a triangle is given by the explicit formula (2).

$$\mathbf{t}(u, v) = (1 - u - v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2, \tag{2}$$

where (u, v) is the barycentric coordinate, which satisfies $u \geq 0$, $v \geq 0$ and $u + v \leq 1$. The point of intersection

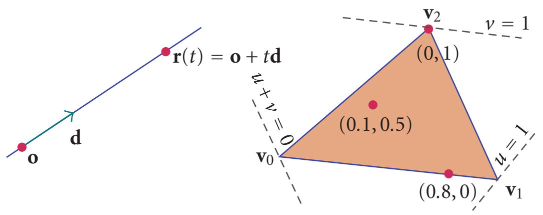

Figure 2: (Left) a simple ray and its parameters. (Right) barycentric coordinate for a triangle, along with some example point values.

between the picking ray, $\mathbf{r}(t)$, and the triangle, $\mathbf{t}(u, v)$, satisfies the equation $\mathbf{r}(t) = \mathbf{t}(u, v)$, which yields:

$$\mathbf{o} + t\mathbf{d} = (1 - u - v)\mathbf{v}_0 + u\mathbf{v}_1 + v\mathbf{v}_2. \tag{3}$$

An illustration of a ray and the barycentric coordinate for a triangle are shown in Figure 2. Denoting $\mathbf{e}_1 = \mathbf{v}_1 - \mathbf{v}_0$, $\mathbf{e}_2 = \mathbf{v}_2 - \mathbf{v}_0$, and $\mathbf{s} = \mathbf{o} - \mathbf{v}_0$, the solution to (3) can be easily obtained by using Cramer's rule [23]:

$$\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \frac{1}{\det(-\mathbf{d}, \mathbf{e}_1, \mathbf{e}_2)} \begin{pmatrix} \det(\mathbf{s}, \mathbf{e}_1, \mathbf{e}_2) \\ \det(-\mathbf{d}, \mathbf{s}, \mathbf{e}_2) \\ \det(-\mathbf{d}, \mathbf{e}_1, \mathbf{s}) \end{pmatrix}. \tag{4}$$

As a result, the intersection information is obtained by solving (4). As this process is independent of the triangles, we can parallelize it in graphics hardware. This equation is adapted with optimizations since the *determinant* of a matrix is an intrinsic function in the High Level Shading Language (HLSL). The intersection test is conducted in the view space and if it is passed, we output a point primitive. The x- and y-components of its position coordinate are 0 because the render target used in our algorithm is only one-pixel in size. The z-component is the depth value which is obtained by transforming the distance value into the projection space. The GPU will automatically add a primitive id as the triangle identifier in the *Input Assembler Stage*. In addition, the barycentric coordinate value (u, v) and the object id are also obtained from the picking information. The pseudo-code in the geometry shader is presented in Algorithm 1.

3.4. Multi-Layer Visibility Queries. We use a multi-layer rendering algorithm to perform linear time intersection tests, taking advantage of the 4th generation of PC-class programmable graphics processing units. The overall approach makes no assumption about the object's motion and is directly applicable to all triangulated models.

First of all, we set a 1×1 sized texture as a render target after the view frustum culling. Instead of rendering the actual triangles, we then render the bounding boxes of the visible objects. We issue a boolean occlusion query for each object during this rendering pass. As we know, the render state *DepthClipEnable* controls whether to clip primitives whose depth values are not in the range of $[0, 1]$ or not; the render state *DepthEnable* determines whether to perform the depth testing or not. After the view frustum

```
(1)  float 3 × 3 edge; float4 Picker
(2)  edge[0] = input[1] − input[0]
(3)  edge[1] = input[2] − input[0]
(4)  Picker.w = det(float3 × 3(−Ray, edge[0], edge[1]))
(5)  if Picker.w == 0 then
(6)      return
(7)  end if
(8)  if Picker.w < 0 then
(9)      Picker.w = −Picker.w
(10)     edge[2] = input[0] − float3(0, 0, 0)
(11) else
(12)     edge[2] = float3(0, 0, 0) − input[0]
(13)     Picker.x = det(float3 × 3(−Ray, edge[2], edge[1]))
(14) end if
(15) if Picker.x < 0 || Picker.x > Picker.w
(16)     return
(17) end if
(18) Picker.y = det(float3 × 3(−Ray, edge[0], edge[2]))
(19) if Picker.y < 0 || Picker.x + Picker.y > Picker.w
(20)     return
(21) end if
(22) // get the distance in the view-space
(23) Picker.z = det(float3 × 3(edge[2], edge[0], edge[1]))
(24) Picker.z = Picker.z / Picker.w ∗ PickingRay.z
(25) PICKING_GS_OUTPUT output
(26) output.Pos = float4(0, 0, Picker.z, 1)
(27) // transform the distance value into projection space
(28) output.Pos.zw = mul(output.Pos.zw,
         float2 × 2(mxProj[2].yz, mxProj[3].yz))
(29) output.Info = float4(Picker.xy/Picker.w, FacetID,
         ObjectID)
(30) outStream.Append(output)
```

Algorithm 1: Object-based intersection test.

culling, there are some objects intersected with the near-plane or the far-plane of view frustum. The depth values of some vertices may not be in the range of $[0, 1]$. In order to collect all the possible intersected objects for the next layer, we set *DepthClipEnable* and *DepthEnable* to *FALSE*. If any occlusion query is passed, the corresponding object may intersected with the picking ray and thus its actual triangles will be rendered; otherwise, it is pruned. Since a large number of objects are not intersected during this step, we can greatly reduce the rendering time compared with the WYSIWYG method, which requires us to render all the objects.

Second, we render the bounding boxes of all sub-objects whose corresponding occlusion query returns *TRUE*. Again we issue a boolean occlusion query for each sub-object during this rendering pass. Since some systems need to handle large models, which may not fit entirely into the GPU memory, we group adjacent local triangles to form a sub-object and prune the potential regions considerably as suggested in [3].

Next, the actual triangles of the unpruned sub-objects are rendered. We only issue one occlusion query for all the triangles during this step. We would like to get the exact

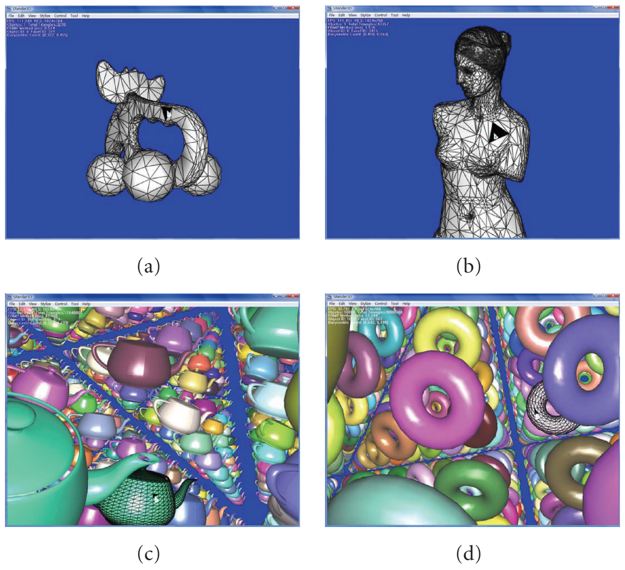

Figure 3: The four test scenes: the toy elk (upper left), Venus (upper right) the teapots (lower left) and the tori (lower right). Note that the picked objects are shown in wireframe and the picked triangles are shown in black, whereas other objects are shaded normally.

intersection result after this step. Triangles outside the view frustum are discarded, and only the closest triangle is needed. Thus the render states *DepthClipEnable* and *DepthEnable* are reset to *TRUE*.

Lastly, if the occlusion query passes, the triangle with the minimal distance from the eye-point is picked and its intersection information can be retrieved from the 1×1 sized render target texture. This causes an additional delay while reading back data from the graphics memory to the system memory. In the WYSIWYG method, we need to lock the window-sized texture to get the picking information but this is slow when the window size is large. Actually our novel algorithm only needs to store the information in the smallest sized texture. If the occlusion query fails, we need not read the data from the render target because we know that nothing has been picked. In the WYSIWYG method, however, one cannot know if anything has been picked until one reads the corresponding data from the texture.

4. Experimental Results and Discussion

Our algorithm takes the screen coordinates of the cursor and the scene to be rendered as the input, and outputs intersection information, such as object id, triangle id, and even the barycentric coordinate of the intersection point. Now our algorithm can be used with platforms which support Direct3D 10 APIs. We have incorporated our FRMP method into a Direct3D 10-based scene graph library and tested it on four scenes in order to evaluate its efficiency for different scene types. All tests were conducted on a PC with a 1.83 GHz Intel Core 2 Duo 6320 CPU, 2 GB main memory, an NVIDIA Geforce 8800 GTS GPU, 320 MB graphics memory, and Windows Vista 64bit Operating System.

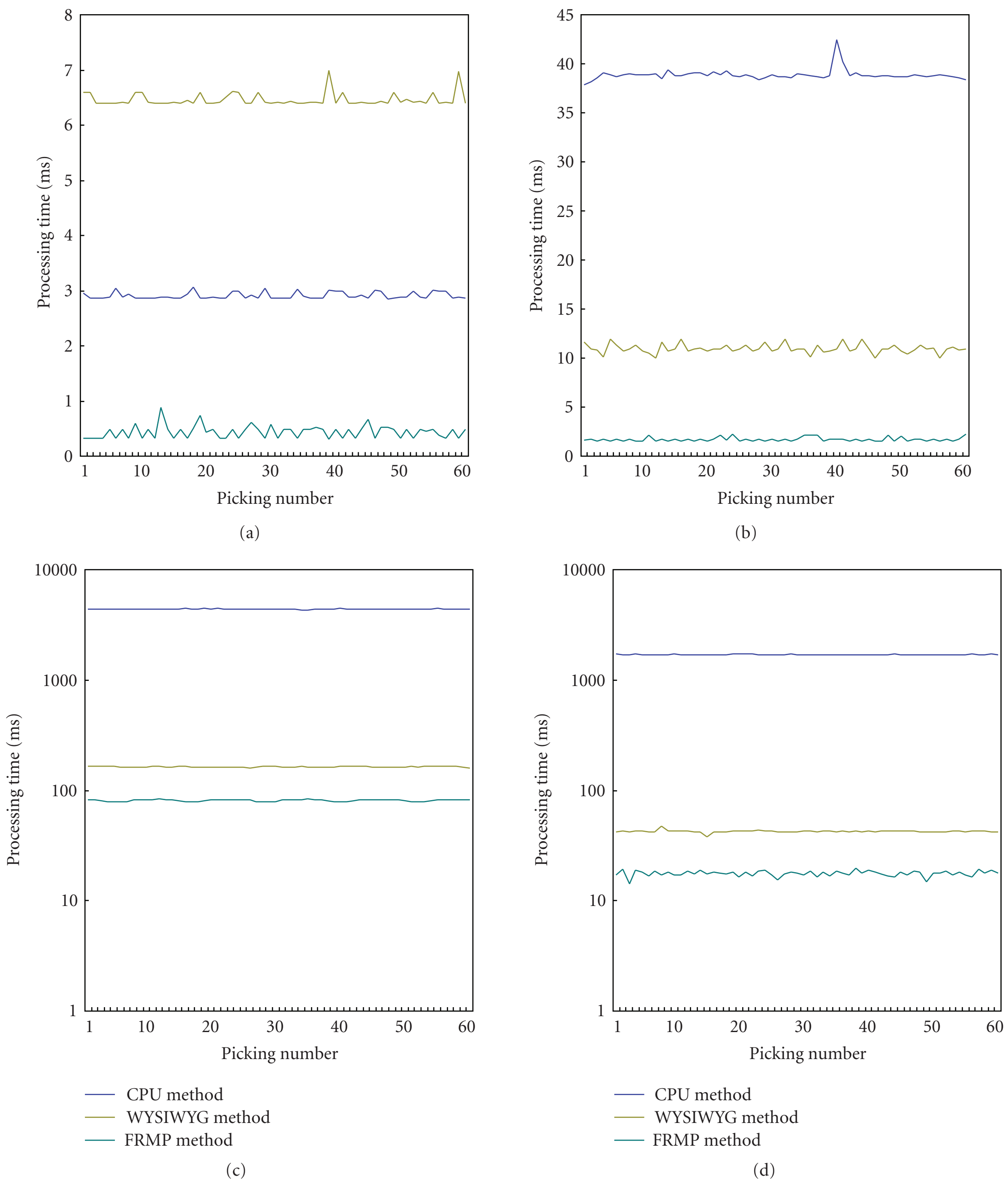

Figure 4: Processing time comparisons for the toy elk (upper left), the Venus (upper right), the teapots (lower left) and the tori (lower right). Note that the lower two scenes use a logarithmic scale to capture the high variations in processing times. Note that if no object is picked, the processing times of our method will even be faster because we picked one object on purpose to perform these tests.

4.1. The Test Scenes. The four test scenes comprise of an arrangement of a toy elk model (3290 polygons), a Venus model (43 357 polygons), 2000 randomly rotated teapots (12.64 M polygons) and 10 000 randomly rotated tori (8 M polygons), all are in resolution of 1024 × 768 pixels. The test scenes are depicted in Figure 3.

The toy elk scene only has 3290 triangles, while the Venus scene consists of large number of triangles. Both are simple cases to handle for the picking operation as only one object is used and is not occlusion culled. These two scenes were tested in order to evaluate the efficiencies in simple cases. Such cases may occur in mesh editing or geometry painting applications.

The teapots scene with 12.64 M triangles and the tori scene with 8 M triangles are complex cases and are designed to rotate randomly from frame-to-frame. They can offer

Table 1: Statistics for the four test scenes. The processing times are in (miliseconds).

Model name	Triangles per model	Modelnumber	Method	Longest time(miliseconds)	Shortest time(miliseconds)	Average time(milisecond)	Speedup
Toy elk	3290	1	CPU	3.064	2.852	2.910	1.000
			WYSIWYG	6.993	6.390	6.464	0.450
			FRMP	0.891	0.314	0.441	6.599
Venus	43 357	1	CPU	42.497	37.824	38.859	1.000
			WYSIWYG	11.974	10.010	10.948	3.549
			FRMP	2.249	1.521	1.702	22.831
Teapot	6320	2000	CPU	4500.598	4320.855	4387.013	1.000
			WYSIWYG	165.392	160.398	163.254	26.872
			FRMP	83.334	78.293	80.959	54.188
Torus	800	10 000	CPU	1720.887	1696.976	1706.358	1.000
			WYSIWYG	47.918	38.016	42.411	40.234
			FRMP	19.636	14.120	17.651	96.672

good occlusions as most of their objects are occluded in most instances.

4.2. Comparison of the Results. For each test scene, we report the processing times of our fast and reliable mouse picking (FRMP) algorithm in comparison to the CPU implementation of our algorithm, and to the traditional GPU method (WYSIWYG) (see Figure 4). Note that in our tests we have picked an object. Had we not done so, our algorithm would have performed even better than the competition. This is because when no bounding box intersects with the picking ray, our approach will not render the actual triangles and return *FALSE* directly.

As we can see from a number of scene statistics shown in Table 1, our method can produce a speedup of more than two as compared to the traditional WYSIWYG method. In the toy elk scene, our method was 2469 miliseconds faster than the CPU method, while the WYSIWYG method was 3554 miliseconds slower than the CPU method. That is because the whole window-sized texture data needs to be read back to the main memory to check the intersection even for small models. In the Venus scene, as the triangle number is increased, our method and the WYSIWYG method produce a speedup of 22.831 and 3.549, respectively. Even in the teapot scene and in the torus scene, our method maintained a good speedup over the WYSIWYG method. If a very large model cannot be loaded into the video memory in its entirety, then our GPU-based algorithm seems to be slower than the CPU-based approach. Fortunately such occurrences are rare in many real-time applications.

5. Conclusions and Future Work

We have presented a novel algorithm for intersection tests between a picking ray and multiple objects in an arbitrarily complex 3D environment using some new features of graphics hardware. The algorithm in this paper is fast, more reliable, parallelizable, and simple. Our algorithm is applicable to all triangulated models, making no assumptions about the input primitives and can compute the exact intersection information in object-space. Furthermore, our FRMP picking operation can achieve high efficiency as compared with traditional methods. Due to its simplicity, our algorithm can be easily integrated into existing real-time rendering applications. Our FRMP picking approach is of relevance to interactive graphics applications.The presented approach still leaves some room for improvement and for extensions. For instance, alternative acceleration techniques for real-time rendering may be applied to our FRMP method. Moreover, additional hardware features will be useful with the progress of the graphics hardware. In the future, we would like to extend and to apply our technique to the generic collision detection field.

Acknowledgments

The authors would like to thank the Cybergames '08 conference and special issue reviewers for their dedicated help in improving the paper. Many thanks also to Xiaoyan Luo, Charlie C. L. Wang, and Feifei Wei for their help and their valuable advice. The models used for the test in our paper can be downloaded from http://shapes.aim-at-shape.net/. This work was supported by the National Natural Science Foundation of China (Grant nos. 60533080 and 60833007) and the Key Technology R&D Program (Grant no. 2007BAH11B03).

References

[1] P. Hanrahan and P. Haeberli, "Direct WYSIWYG painting and texturing on 3D shapes," *ACM SIGGRAPH Computer Graphics*, vol. 24, no. 4, pp. 215–223, 1990.

[2] J. Lander, "Haunted trees for halloween," *Game Developer Magazine*, vol. 7, no. 11, pp. 17–21, 2000.

[3] N. K. Govindaraju, S. Redon, M. C. Lin, and D. Manocha, "CULLIDE: interactive collision detection between complex models in large environments using graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS '03)*, pp. 25–32, Eurographics Association, San Diego, Calif, USA, July 2003.

[4] T. Möller and B. Trumbore, "Fast, minimum storage ray-triangle intersection," in *Proceedings of the 32nd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '05)*, Los Angeles, Calif, USA, July-August 2005.

[5] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer, New York, NY, USA, 1985.

[6] I. J. Palmer and R. L. Grimsdale, "Collision detection for animation using sphere-trees," *Computer Graphics Forum*, vol. 14, no. 2, pp. 105–116, 1995.

[7] P. M. Hubbard, "Approximating polyhedra with spheres for time-critical collision detection," *ACM Transactions on Graphics*, vol. 15, no. 3, pp. 179–210, 1996.

[8] G. van den Bergen, "Efficient collision detection of complex deformable models using AABB trees," *Journal of Graphics Tools*, vol. 2, no. 4, pp. 1–13, 1997.

[9] T. Larsson and T. Akenine-Möller, "Collision detection for continuously deforming bodies," in *Proceedings of the Annual Conference of the European Association for Computer Graphics (EUROGRAPHICS '01)*, pp. 325–333, Manchester, UK, September 2001.

[10] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan, "Efficient collision detection using bounding volume hierarchies of k-DOPs," *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 21–36, 1998.

[11] S. Gottschalk, M. C. Lin, and D. Manocha, "OBBTree: a hierarchical structure for rapid interference detection," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*, pp. 171–180, ACM, New Orleans, La, USA, August 1996.

[12] N. K. Govindaraju, M. C. Lin, and D. Manocha, "Quick-CULLIDE: fast inter- and intra-object collision culling using graphics hardware," in *Proceedings of IEEE Virtual Reality Conference (VR '05)*, pp. 59–66, IEEE Computer Society, Bonn, Germany, March 2005.

[13] N. K. Govindaraju, M. C. Lin, and D. Manocha, "Fast and reliable collision culling using graphics hardware," in *Proceedings of the 11th ACM Symposium on Virtual Reality Software and Technology (VRST '04)*, pp. 2–9, ACM, Hong Kong, November 2004.

[14] T. Akenine-Möller and E. Haines, *Real-Time Rendering*, AK Peters, Natick, Mass, USA, 2nd edition, 2002.

[15] Microsoft Corporation, *DirectX Software Development Kit*, Microsoft Corporation, Redmond, Wass, USA, 2007.

[16] D. Shreiner, Ed., *OpenGL® 1.4 Reference Manual*, Addison Wesley Longman, Redwood City, Calif, USA, 4th edition, 2004.

[17] F. Evans, S. Skiena, and A. Varshney, "Optimizing triangle strips for fast rendering," in *Proceedings of the 7th IEEE Visualization Conference*, pp. 319–326, IEEE Computer Society Press, San Francisco, Calif, USA, October-November 1996.

[18] U. Assarsson and T. Möller, "Optimized view frustum culling algorithms for bounding boxes," *Journal of Graphics Tools*, vol. 5, no. 1, pp. 9–22, 2000.

[19] H. Zhao, X. Jin, and J. Shen, "Simple and fast terrain rendering using graphics hardware," in *Advances in Artificial Reality and Tele-Existence*, vol. 4282 of *Lecture Notes in Computer Science*, pp. 715–723, Springer, Berlin, Germany, 2006.

[20] J. Bittner and V. Havran, "Exploiting temporal and spatial coherence in hierarchical visibility algorithms," in *Proceedings of the 17th Spring Conference on Computer Graphics (SCCG '01)*, p. 156, IEEE Computer Society, Budmerice, Slovakia, April 2001.

[21] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer, "Coherent hierarchical culling: hardware occlusion queries made useful," *Computer Graphics Forum*, vol. 23, no. 3, pp. 615–624, 2004.

[22] D. Blythe, "The direct3D 10 system," *ACM Transactions on Graphics*, vol. 25, no. 3, pp. 724–734, 2006.

[23] E. W. Weisstein, "Cramer's Rule," http://mathworld.wolfram.com/CramersRule.html.