Ray tracing collision detection based on GPU pipeline reorganization

Sining Cheng¹, Huiyan Qu²*, Xianjun Chen¹

¹Haikou College of Economics, Haikou, Hainan Province, 571127, China
²School of Information Technology, Jilin Agricultural University, Changchun, Jilin Province, 130118, China

*Corresponding author e-mail: quhuiyan68@163.com

Abstract. This paper proposes a parallel collision detection algorithm based on ray tracing to improve GPU performance. The algorithm uses parallel ray tracing pipeline technology to improve the performance of the graphics processing unit (GPU). When the GPU combines multiple ray tracing, it will cause the work group threads to diverge, resulting in too many idle threads, which will reduce the performance of the system. In order to avoid thread divergence, a new collision detection technology for the detailed detection stage of ray tracing using a pipeline structure is proposed. By optimizing the addition of buffers, the ray tracing is divided into three stages, and the pipeline is designed to integrate multiple rays tracking to avoid unnecessary synchronization issues between the CPU and GPU, thereby improving system performance. The comparison experiment results show that the proposed algorithm improves the system performance by 2.7 times.

1. Introduction
Collision detection is an important task in virtual environment simulation. Especially in large-scale virtual scenes, we often need to know which objects are likely to collide with each other. Due to the complexity of virtual environment and the real time constraints required by virtual reality, collision detection is one of the main bottlenecks in VR application. We need to simulate the environment of millions of objects. This environment includes a variety of complex objects that can deform or undergo complex topological or fluid transformations.

The current common approach is to use GPGPU (general purpose computing on graphics processing units). The computation process is completed by the high-performance computing power of GPU. Parallel scanning pruning algorithm is adopted by GPU in the initial detection stage, which can be referred to reference [1]. Detailed detection stage can also be implemented on GPU, which can be referred to reference [2].

In this chapter, a detailed collision detection algorithm is proposed to detect the collision process through image rendering technology. Because GPU has advantages in time-consuming rendering, the image-based approach is well suited to GPGPU implementations. In particular, a combination of multiple ray tracing techniques can be used to improve system performance [3]. When implementing multiple ray tracing on GPU, multi-branch threads will lead to the decrease of GPU utilization and system performance. In order to avoid this problem, a new collision detection algorithm using pipelined structure for ray tracing detailed detection stage is proposed. The goal of pipelining is to minimize branches as threads to improve system performance. Pipelining is designed to integrate
multiple ray tracing processes without overhead. The prediction algorithm can avoid the synchronization of CPUGPU between pipeline and reduce the system performance.

In the above ways, you can think of a GPU as a stream processor. Since GPU is a massively parallel processor, the computing process is parallel computing [12]. When code is running on the GPU, the same function (called the kernel) is called by multiple threads on different data items. Work items do not run independently but in a group. In a workgroup, work items run as a single instruction stream with multiple data streams. On a conditional branch, each branch runs continuously if work items diverge between workgroups. This phenomenon is called thread divergence. Thread divergence can degrade system performance. An example of thread divergence is sparse input, in which some work items have nothing to do and are idle in a working group. Conflicting flows use stream compression to avoid kernel execution on sparse input. Collision detection is divided into several steps that use stream compression between steps to ensure dense input for each kernel.

2. Ray tracing collision detection pipeline
The first complete algorithm is the stackless bounding box hierarchy (BVH) traversal. The algorithm adopts acceleration structure and can be applied to rigid objects. The second complete algorithm is basic ray tracing. The algorithm does not use any additional acceleration structures and can be used to deform objects. If you use stackless BVH traversal algorithms or deform objects, you need to update the accelerated structure at each time step.

The third ray tracing algorithm is incremental and needs to update the previous time step. This algorithm cannot be used at any time because it requires additional input that refers to the data of the last triangle hit in the previous time step (also known as temporal data). The iterative algorithm requires a full ray tracing algorithm to create temporal data, which can be used to replace the full algorithm. An iterative algorithm can be used as long as the relative displacement between the ray and the target remains small. When temporal data is available, iterative algorithms are faster than the first two, and should be preferred in applications. The three basic algorithms are shown in Figure 1.

Figure 1. Classification of ray tracing collision detection pipeline algorithm

2.1 Pipelining structure
In order to realize collision detection, multiple tasks are performed in the detailed detection stage, and displacement is applied on a pair of objects to consider internal deformation. A pair of objects classified with a small displacement will use an iterative ray tracing algorithm.

Vertex cull uses the delete criteria to remove vertices that do not need ray tracing tests because a simple test can determine that there are no collisions between the removed vertices. For vertices classified as likely to collide, check if the vertices are inside the intersection of the enclosing bodies of the two objects, and if so, perform an iterative algorithm. If outside, use ray tracing algorithms.

2.2 Steps of detecting each pair of objects
The detection step for each object (object) pair begins with the list of object pairs provided by the preliminary detection. In this step, one thread corresponds to one object pair.
First, pairs are divided into two categories: rigid objects and deformable objects only. This division is to judge the basis of using complete or iterative algorithms.

For pairs with only rigid body objects, a displacement criterion is applied at this stage to separate those with high displacement, which will use the full ray tracing algorithm and the small displacement algorithm using the iterative algorithm. At this stage, the displacement criterion cannot be included in the application of a deformed object pair. In this case, the displacement needs to be measured locally to account for the internal deformation.

2.3 The steps of detecting each vertex

Each vertex step starts with a list of each pair built by each vertex, in which a thread is executed once for each vertex of each object in each pair.

For a pair of deformed objects, displacement measurements can be applied at the vertices to separate small displacements with the iterative algorithm from other displacements using the full algorithm.

Then, drop criteria are applied to the vertices to eliminate those that can be deleted. Using iterative algorithms and vertex rules can give different results for different vertices. For vertices using the full ray tracing algorithm, check that the vertices are within the hierarchical bounding box intersection of the two objects. If not in the intersection, delete this vertex as a non-colliding vertex. For vertices using the iterative algorithm, determine if the test time data can be obtained from the previous step. If no data is available, the vertex is deleted.

3. Pipelining algorithm for ray tracing

In order to be able to schedule a kernel as input to an additional buffer, you need to know the completion of the buffer to determine how many threads are executed. This completion value is in GPU memory, and kernel execution is scheduled by CPU. There are two solutions to manage this data dependency.

Step0: Initialize. Set of contact points \( T = \{T_1, T_2, \ldots, T_n\} \) , detection point set \( S = \emptyset \);

Step1: Get the preliminary test results from the preliminary test stage and from the contact point set \( T = \{T_1, T_2, \ldots, T_s\} \) and take the first point \( T_1 \) put in the set of test points \( S \);

Step2: Judge whether it is a rigid body model, if not go to the Step7;

Step3: Judge whether the relative displacement is high displacement, if not Step6;

Step4: Judge whether it is a deformation object according to the high displacement table, if so, adopt the hierarchical bounding box (BVH) ray tracing algorithm. Return contact point set Step10;

Step5: Use the basic ray tracing algorithm to return the contact point set Step10;

Step6: Judge available data through low displacement table, use iterative ray tracing algorithm, and return contact point set Step10;

Step7: Delay the test point pair, judge whether the relative displacement is high displacement, if not Step9;

Step8: Use the hierarchical bounding box (BVH) ray tracing algorithm to return the contact point set Step10;

Step9: Adopt iterative ray tracing algorithm;

Step10: Take the set of contact points \( T = \{T_1, T_2, \ldots, T_n\} \) The next point is put into the detection point set \( S = \{T_1, T_2, \ldots, T_s\} \) The algorithm is repeated until the contact point set is empty and the algorithm is finished.
4. Performance evaluation

This section describes the experimental environment and performance results. The performance of ray tracing pipeline algorithm is demonstrated through the experimental environment given. The performance optimization, optimization test and error caused by prediction are given.

4.1 Scene Experiment

In order to measure the ray tracing algorithm in the detailed detection stage proposed in this chapter, the experimental scenes of rigid bodies, rigid bodies and deformed objects within a given experimental time \( t \) seconds were tested respectively.

![Figure 2. Collision detection of rigid objects falling to the ground](image)

The circle is made up of 106 triangles, and the cartoon is made up of 720 triangles.

Since the rigid body does not deform during collision detection, it is not necessary to update the bounding box, so the collision detection time is very short. Figure 2 shows two different objects, with more than 1,000 pairs of objects being tested during the detailed detection phase.

![Figure 3. Collision detection of deformed objects](image)

The four deformation objects are composed of 1656, 2890, 3264 and 2650 triangles respectively. As shown in Figure 3, the four deformed objects are respectively composed of 1656, 2890, 3264 and 2650 triangles, and more than 23,780 pairs of objects are tested in the detailed detection stage. Detailed detection phase using OpenCL in GPU implementation. All scenarios were executed with 3.19ghz Intel Core i7CPU, 6GB DDR RAM, and GPU with NVIDIA Geforce GTX780 graphics card.

4.2 Pipeline performance

GPU collision detection and pipeline reorganization of rigid-body objects in the scene used a complete ray tracing algorithm, with an average acceleration of 1.27 times, and a complete iterative ray tracing algorithm, with an average acceleration of 2.66 times.

Table 1 compares the collision detection time of the four technologies of rigid body and variable body model.

| Collision Detection Method | Ray Tracing Time (ms) |
|---------------------------|-----------------------|
| Full ray tracing algorithm for rigid objects | 82.33 |
| Iterative ray tracing algorithm for rigid objects | 56.32 |
| Deformable body full ray tracing algorithm | 22.31 |
| Deformable body iterative ray tracing algorithm | 15.56 |
| Collision detection based on bounding box division | 66.27 |
| Collision detection based on parallel pipeline | 35.31 |
| Ray tracing collision detection based on parallel pipeline reorganization | 18.33 |
| Ray tracing collision detection based on parallel pipeline reorganization | 12.28 |

| Collision detection based on bounding box division | 51.25 |
| Collision detection based on parallel pipeline reorganization | 25.83 |
| Collision detection based on parallel pipeline reorganization | 15.51 |
| Ray tracing collision detection based on parallel pipeline reorganization | 9.87 |
| Ray tracing collision detection based on parallel pipeline reorganization | 46.12 |
| Ray tracing collision detection based on parallel pipeline reorganization | 20.21 |
| Ray tracing collision detection based on parallel pipeline reorganization | 13.01 |
| Ray tracing collision detection based on parallel pipeline reorganization | 7.95 |
5. Conclusion
A collision detection method using GPU to improve ray tracing performance is proposed in this paper. Different ray tracing algorithms can be integrated in the detailed detection phase to effectively deal with different properties of objects. By dividing the detailed detection stage into three steps, the whole pipeline can be kept with intensive input to make efficient use of GPU. Experimental results show that the average performance of the proposed method is improved.

6. Acknowledgments
This work was financially supported by Provincial Natural Science Foundation of China, "large-scale complex scenarios based on virtual reality is rapid real-time collision detection technology research" Item number: 619QN247 and Key Project of "The thirteenth-Five" science and technology research of Jilin Provincial Department of Educationon Optical Motion Capture Technology "Rapid image generation and detection technology based on virtual reality", JJKH20190907KJ fund.

References
[1] Liu, F., Harada, T., Lee, Y., and Kim, Y. J. (2010). Real time collision culling of a million bodies on graphics processing units. In ACM Transactions on Graphics (TOG), volume 29, page 154. ACM.
[2] Pabst, S., Koch, A., and Straßer, W. (2010). Fast and scalable cpu/gpu collision detection for rigid and deformable surfaces. In Computer Graphics Forum, volume 29, pages 1605 – 1612. Wiley Online Library.
[3] Lehericey, F., Gouranton, V., and Arnaldi, B. (2013a). New iterative ray-traced collision detection algorithm for gpu architectures. In Proceedings of the 19th ACM Symposium on Virtual Reality Software and Technology, pages 215 – 218. ACM.
[4] Teschner, M., Kimmerle, S., Heidelberger, M., Zachmann, G., Raghupathi, L., Fuhrmann, A., Cani, M., Faure, F., Magenat-Thalmann, N., Strauss, W., et al. (2015). Collision detection for deformable objects. In Computer Graphics Forum, volume 24, pages 61 – 81. Wiley Online Library.
[5] Kockara, S., Halic, T., Iqbal, K., Bayrak, C., and Rowe, R. (2007). Collision detection: A survey. In Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on, pages 4046 – 4051. IEEE.
[6] Lehericey, F., Gouranton, V., Arnaldi, B., et al. (2013b). Ray-traced collision detection: Interpenetration control and multi-gpu performance. JVRC, pages 1–8.
[7] Heidelberger, B., Teschner, M., and Gross, M. H. (2003). Real-time volumetric intersections of deforming objects. In VMV, volume 3, pages 461 – 468.
[8] Huiyan Qu, Wenhui Li and Wei Zhao. (2019). Human-Vehicle Collision Detection Algorithm Based on Image. Processing. International Journal of Pattern Recognition and Artificial Intelligence. Vol. 34(8):205-209.
[9] Wang, B., Faure, F., and Pai, D. (2012). Adaptive image based intersection volume. ACM Transactions on Graphics (TOG), 31(4):97.
[10] Hermann, E., Faure, F., Raififi, B., et al. (2008). Ray-traced collision detection for deformable bodies. In 3rd International Conference on Computer Graphics Theory and Applications, GRAPP 2008.
[11] Lehericey, F., Gouranton, V., and Arnaldi, B. (2013a). New iterative ray-traced collision detection algorithm for gpu architectures. In Proceedings of the 19th ACM Symposium on Virtual Reality Software and Technology, pages 215 – 218. ACM.
[12] Nickolls, J. and Dally, W. J. (2010). The gpu computing era. IEEE micro, 30(2):56 – 69