

RESEARCH ARTICLE

Visibility-Based Fast Collision Detection of a Large Number of Moving Objects on GPU

MANKYU SUNG 

Department of Game Software, Keimyung University, Daegu 42601, Republic of Korea

e-mail: mksung@kmu.ac.kr


This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean Government [Ministry of Science and ICT (MSIT)] under Grant 2021R1A2C1012316.

ABSTRACT This paper proposes a simple and efficient collision detection algorithm for a large number of moving objects. The basic idea is to minimise the number of moving objects that go through the complicated the collision checking process, which can improve the overall performance. To this end, we propose a visibility-based culling technique that identifies substantially small or hidden objects that do not cause any visual artifact even if we ignore them. This paper also develops a variable-size Morton codes to speed up the construction time of the Linear Bounding Volume Hierarchy (LBVH), which is used to efficiently check the proximity between objects efficiently. The visibility-based culling technique is based on the so called *visibility map* on the top of the g-Buffer technique. This map is a texture that contains the ID of the moving object in the screen space. The number of fragments of each object on the map is then counted in parallel manner on the GPU. If the number of fragments is less than a predefined threshold value, the algorithm does not include the object in the LBVH constructions step. Although the performance depends on the camera view point, we have verified through several experiments that the proposed algorithm improves the overall performance at least 70% even when the number of moving objects is more than 10,000.

INDEX TERMS Collision detection, linear bounding volume hierarchy, visibility.

I. INTRODUCTION

When we want to animate or simulate a large number of 3D objects in real-time applications such as video games, the most nagging problem we have to solve is how to prevent them from inter-penetrating with each other. Many different approaches have been proposed to achieve this goal. A few survey papers [1], [2], [3] explain the evolution of key important techniques. Today, the advent of programmable hardware allows us to build a high-performance algorithm that exploits the highly parallel nature of the GPU. Although GPU is mostly used to accelerate gaming graphics. Today, General Purpose Graphics Processing Units (GPGPUs) are the hardware of choice for accelerating computational workloads in the modern High Performance Computing (HPC) landscape. This gives researchers a new way to solve collision detection problems on GPUs.

The associate editor coordinating the review of this manuscript and approving it for publication was Lei Wei .

For the collision detection problem, the BVH(Bounding Volume Hierarchy) is the most popular acceleration structure to represent a hierarchical abstraction of complex 3D geometry due to its small memory requirement and simplicity. Parallel construction of BVH algorithms on the GPU is able to maximize their performance. Although the original BVH algorithms were used for the ray-object intersection problem, [4], [5], [6] shows that they also be used for collision detection between many 3D moving objects as well, which is called *linear BVH(LBVH)*. The key idea is to first sort the objects along a space-filling curve, also known as Morton codes, and then recursively partition them so that each node ends up representing a linear range of objects. Of all the steps in constructing the LBVH, the most time consuming step is sorting the objects based on their Morton codes, which are generated from the 3D position of the object. Since the Radix sorting method is generally used, the number of bits and the layout of the bits representing the Morton codes are the critical for improving the performance. Several methods

have been proposed to construct the tree in memory-efficient manner [7], [8], [9] because memory usage was the problem for a large number of objects. Efficient ordering of Morton codes using locally-ordered clustering with spatial sorting also proposed [10], [11]. In this paper, inspired by the extended Morton code proposed by [12], we have proposed a variable-size Morton code technique in which the number of bits is changing adaptively per frame depending on the situation. That is, depending on the visible number of objects in the scene, the number of bits changes from minimum of 16 bits to a maximum 64 bits dynamically. Through several experiments, we have found out that this adaptive approach speeds up the sorting time.

One major disadvantage of the LBVH is that it does not take into account the camera's point of view. For example, if we apply this technique to the video game, all objects have to go through the LBVH construction process, even though many of them are hidden by the other objects or barely visible on the screen because they are far away from the camera.

In this paper, we propose a visibility-based collision detection algorithm for a large number of moving objects. In this algorithm, we assume that each moving object is encompassed by a bounding sphere. The basic idea is to generate a *visibility map* that represents how many pixels are required for a particular bounding sphere on the final rendered image given a camera position and direction. If an object is substantially occluded or completely hidden by another object, they it has a small number of pixels on the visibility map. If the object is in this case, then the algorithm ignore it. This makes sense because we don't need to worry about the collisions of visually insignificant objects. To improve the performance, it is necessary to read the pixels of the visibility map in parallel. We used GPU-supported Atomic counting for this. In this way, we can reduce the number of objects entering the collision detection. We found that the number of visible objects entering the collision detection process is significantly smaller than the original number of objects, which can improve the overall performance.

A. CONTRIBUTIONS

The contributions can be summarized as following:

- Visibility based collision detection : we proposed a visibility map on the top of g-buffer algorithm where the pixel of the map represents the id numbers of bounding spheres. This map provides cue how visible the object from the current camera setting, which can reduce the total number of objects entering time consuming collision detection stages by clipping out objects barely visible.
- GPU based counting method : Given a visibility map, we applied GPU-based pixel counting method. This process is based on Atomic counting mechanism where many threads are running simultaneously without interfering each other to count the number of pixels having particular id numbers.

- Adoptive Morton codes : For testing collision among a large number of bounding spheres, the LBVH has been proved to be an efficient algorithm. Instead of using a fixed number of bits for Morton code, which is assigned to each moving object based its positions, we proposed a method that changing the number of bits for Morton codes adaptively that improves the overall performance.

II. RELATED WORK

A. COLLISION DETECTION

Given a large number of objects represented as bounding spheres, the goal is to find a set of spheres that intersect each other. This is called the N-body collision detection problem. Among all the different approaches to solve the problem, sweep-based algorithms have been proposed continuously. Basically, the sweep method finds overlaps between AABBs(Axis-Aligned Bounding Boxes) by sorting the projected extents of boxes on the Cartesian axis [13]. The sweep and prune method improves on the original method by exploiting temporal coherence, as object positions change little over time [14]. However, it is not efficient for large numbers of objects because it requires all pairs to be tested separately.

Spatial subdivision is another way of improving the speed of the check. Instead of comparing the collision for each pair, this technique is able to limit the number of objects to be compared by dividing up a continuous space into several discrete areas based on a few simple rules. These techniques include QuadTree, BSP Tree and bins/spatial Grids [15]. All techniques have their own advantages and disadvantages [16]. One thing they have in common, however, is that they do not consider how objects are looked on the final image plane. No matter where the objects are, a collision test is always performed. If an object is turns out to be negligible in the final rendered image, we can cull it out at the earlier stage instead. In this way, we can limit the number of objects that enter to the collision test.

This paper introduces a perception-based visibility map that contains object ID numbers for all fragments. By combining hardware-supported Atomic counting capability with the multi-pass rendering technique, the proposed algorithm is able to detect which moving objects are negligible in real-time by referencing the visibility map. Through a series of experiments, we have verified that this leads to an improvement in overall performance.

B. BOUNDING VOLUME HIERARCHY(BVH)

BVH-based techniques have been applied to ray-tracing problems for many years. The original BVH idea is to construct a hierarchical structure of an object and in each tree node store a bounding volume for the geometry of that sub tree [3], [17], [18], [19]. Then, instead of doing the intersection test of a ray against the complicate geometry itself, it performs the test on the BVH structure instead by traversing from the top node of the hierarchy, which is much simpler and faster.

TABLE 1. BVH construction time split.

Steps	Avg time(μ s)	Percentage
Morton code assignment	3.47	1.67
Radix sort with Prefix scan on the codes	191.1	92.3
Generation of leaf and internal nodes	10.2	4.92
And another entry	2.21	1.06

Further testing is not required if the upper level of the sub-tree does not cause an intersection [4], [6]. However, for animated scenes where many objects are moving at the same time, reconstructing the BVH structure takes a significant amount of time [20]. Re-fitting is a standard way for updating the BVH efficiently [21].

On the top of the original BVH, researchers have proposed several ways to construct the BVH in parallel manner, which leads to significant performance improvement [17], [22], [23], [24]. The linear BVH(LBVH) is the most promising option because it fits well with the GPU architecture [15]. The LBVH gives the order of leaf nodes, which is the object geometry, of the hierarchy tree and then internal nodes can be built in parallel way. This order is called the space-filling curve, also known as Morton codes [25]. In general, the Morton codes are assigned from the 3D position of objects. This means that the X, Y and Z coordinates are interleaved bit-by-bit to obtain the Morton codes. An extension of the Morton code has been proposed by [12]. In the extended Morton codes, three axes can be re-ordered or size information is embedded for efficient traversing of the tree [12].

In this paper, by analysing the environment, we use the *adaptive* Morton codes where the number of bits of the Morton codes change dynamically depending on the number of *active* objects. After the Morton codes are assigned, they are sorted out by the Radix sorting method [26]. Although the prefix-sum step of the Radix sorting can be done in parallel [27], this sorting step is the most time consuming because all the codes have to be sorted by bit-by-bit. When we check the overall computation time of the BVH construction, this sorting step takes almost 92% of the total computation time. Table 1 shows the computation time of all steps for 2000 moving objects when we use 32bits for Morton codes. To reduce the construction time, the number of bits for Morton codes must be as small as possible. We use the re-ordered axis version of the Morton code as suggested in [12], but we let the number of bits change dynamically depending on the viewpoint. The logic behind this is follows: If the average size of each object in the final rendered image is not large, collisions between objects are almost not noticeable. In this case, we can allocate a small number of bits to the Morton codes so that they can check for collision very quickly, although they may miss some of the unimportant collisions.

Once the leaf nodes have been constructed by sorting the Morton codes, the remaining internal nodes are built by checking the common bits between two adjacent codes. An efficient parallel algorithm for this has been introduced

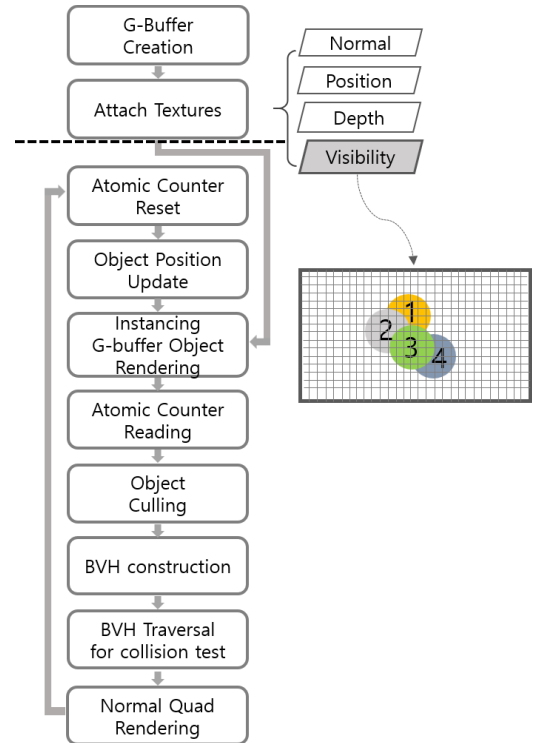


FIGURE 1. Overview of the algorithm.

by [6] for this. We also used this technique in our research as well.

In summary, all other previous work did not consider how important the objects are in the scene. That is, regardless of the size of the object in the final rendered image, the collision test was performed. In this paper, however, the proposed algorithm first checks how many pixels are needed for the object in the final rendered images, which is encoded in the visibility map. If the object is small enough to be negligible in the scene, or hidden by other objects, it is discarded at an early stage. The number of bits for Morton codes in LBVH, which significantly impacts the overall performance, is also changed dynamically according to the camera setting in this algorithm. The proposed method has an advantage in the performance improvement.

III. ALGORITHM

The whole algorithm is divided into a pre-processing step and a rendering step. Figure 1 illustrates the whole steps. The pre-processing step initializes the g-buffer [28]. The g-buffer is generally used for multi-pass rendering [29], [30]. The first render pass creates all lighting-relevant data as textures including normal vectors, depth and world positions of the vertices. Then, the second pass uses all the data from the first pass and computes the shading or lighting of all pixels. In addition to the general g-buffer structure, we propose to add another texture called the visibility map. The visibility map simply contains the ID numbers of objects in the screen space. The right side of Figure 1 shows an example of the

visibility map where each pixel contains the ID number from 1 to 4 of the bounding spheres. An example of use of visibility map is also shown in Figure 2 as well. Note that the background pixel has an ID number of 0. The visibility map, in our implementation, is an unsigned integer 2D texture whose size is equal to the size of the rendered image for simplicity. The more details about the creation of frame buffer and textures can be found in the [31].

At the first render pass, the ID number of the object for the visibility buffer can be easily set from the GPU instancing where the instanced draw call has its own ID number in the Vertex Shader. The GPU instancing is a powerful technique for reducing draw calls when we went to similar geometry data multiple times. It fits our case because the bounding spheres have similar geometry even though they may have different sizes.

The Fragment Shader then sets the same ID number for all fragments obtained from the Rasterization. Listing 1 shows the code snippet of the Vertex Shader and Fragment Shader for the visibility buffer. Note that line number 40 of the Listing 1 is where the ID number is set in Fragment Shader.

The model matrix in the listing can be built separately for each object from the buffer containing the current world position of the object. Note that we used the GLSL for the implementation.

The second pass then counts the number of pixels for each ID number in the visibility map. Due to the depth test, occlusions between objects are naturally reflected on the visibility map. Therefore, if the number of pixels for a particular object is 0, then it means that it is completely hidden by other objects or out of view frustum of the camera.

Once the visibility is constructed, the only remaining task is to find a way to count the number of pixels efficiently. In our approach, the counting of fragments is done by using GPU-based Atomic operations [32]. Atomic operations allow different threads to safely manipulate shared variables, and in turn, allow synchronization and work sharing between threads on the GPU. This is necessary because all invocations of Fragment Shader must update the same counter exclusively without any racing conditions. Current 3D graphics APIs such as OpenGL, Vulkan, or DirectX support the Atomic operations. Listing 2 shows the snippet of the Atomic counting codes. Note that line number 15 of listing 2 is where the counter is incremented.

If the number of fragments of a particular object is below a predefined threshold value, λ , the object can be ignored for collision detection. We think of these objects as invisible. This selective culling minimizes the number of objects entering to the complicated LBVH construction step. Figure 3 illustrates the culling process where the objects that pass the threshold value are pushed into the visible object buffer. In order to compact the visible object buffer, coordination between threads is required so that the index is increased exclusively without a racing condition. In order to do this, Atomic counter buffer must be created before hand. The buffer can contain only unsigned integers and be incremented

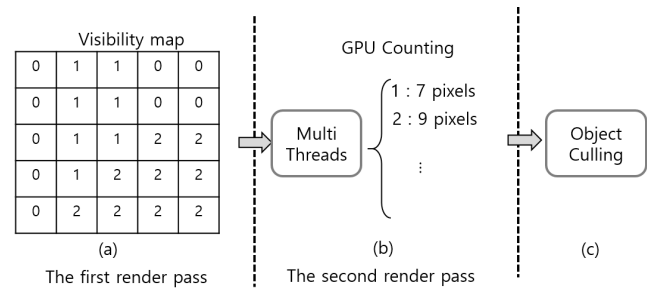


FIGURE 2. An example of visibility map and its GPU counting: (a) The visibility map contains the ID number of object. (b) Those numbers are counted in parallel manner in GPU. (c) Those information is used to cull out negligible objects that do not enter collision test.

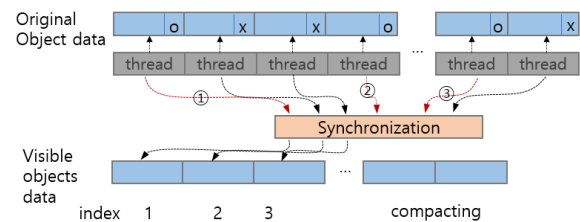


FIGURE 3. Selective Object Culling : After all threads check the visibility of objects simultaneously, they compact the object's data in serial manner without conflict, if the object's pixel count is greater than the threshold. Note that (o) mark means the object is visible and (x) mark means that object is not visible.

and decremented by one. Please refer to [31] for more details.

Only objects in the visible object buffer can be sent to the next collision checking step. This culling job was implemented as a Compute Shader in our case.

As the algorithm is figuring out the visible objects, it also calculates statistical information such as the average number of fragments. This statistical information is used to determine the number of Morton codes. Inspired by the extended Morton codes proposed in [12], we use a similar but modified version of the algorithm in which the three different sets of the number of bits are applied for depending on the situation.

The entire LBVH construction step follows the parallel algorithm proposed in [15]. Figure 4 illustrates this step. First, at the bottom of the figure, Morton codes are assigned to all objects in the visible buffer. The Morton codes are built from the global position of the object's bounding sphere. We have used the *modified* extended Morton code technique in our algorithm. The extended Morton code method changes the order in which the bits corresponding to different spatial axes are used in the code [12]. In particular, ordering the axes by putting the ones with the largest extent at the beginning of the code had a positive effect because splitting the larger axes first can make the center of the clusters are closer to the cuboid shape and thus more spatially compact than always keeping a fixed order [12]. Furthermore, instead of setting the same number of bits for each axis, we can assign a different number of bits depending on the extent of the axes. For example,

```

1 //
2 //Vertex Shader
3 //
4 in vec3 VS_IN_Position; //local coord
5 in vec3 VS_IN_Normal;
6
7 out vec3 FS_IN_WorldPos; //global coord
8 out vec3 FS_IN_Normal;
9 out uint FS_IN_instanceID; //object ID number
10
11 buffer pos {
12     vec3 Pos[] //Buffer containing the position of all
13     objects
14 };
15 void main() {
16     //Instance ID
17     FS_IN_instanceID = uint(gl_InstanceID);
18     //Build a model matrix for each object
19     mat4 model = build_transform(Pos[gl_InstanceID]);
20     //global vertex positions
21     FS_IN_WorldPos = model * Position;
22     //vertex normals
23     FS_IN_Normal = VS_IN_Normal;
24 }
25 //
26 //Fragment Shader
27 //
28 //Textures
29 out vec3 FS_OUT_Normal; //A texture of vertex normal
30 out vec3 FS_OUT_WorldPos; //A texture of vertex position
31 out uint FS_OUT_vis; //A texture for visibility
32
33 in vec3 FS_IN_WorldPos;
34 in vec3 FS_IN_Normal;
35 in uint FS_IN_instanceID;
36
37 void main() {
38     //for each pixel,
39     //write its ID number in the visibilt map
40     FS_OUT_vis = FS_IN_instanceID;
41     FS_OUT_Normal = FS_IN_Normal;
42     FS_OUT_WorldPos = FS_IN_WorldPos;
43 }

```

Listing 1. Vertex shader/fragment shader snippet.

```

1 //visibility map
2 uniform sampler2D s_Vis;
3 //texture coordinates
4 in vec2 FS_IN_TexCoord;
5
6 //Atomic counter buffer, n:number of objects
7 uniform atomic_uint ac1[n];
8
9 void main() {
10     float v = texture(s_Vis, FS_IN_TexCoord).r
11     //if not a background
12     if (v > 0.0) {
13         int idx = int(v); //object ID
14         //increase the counter
15         uint a = atomicCounterIncrement(ac1[idx - 1]);
16     }
17 }

```

Listing 2. Atomic counting codes in GPU.

if the environment is flat, then the y coordinate of all object would be almost the same during animation. In this case, we can assign the smallest number of bits for the y axis. As we show in the table 1, the most time-consuming step in building LBVH is the radix sorting. Radix sorting rearranges the objects by bit-by-bit comparison. Therefore, the number of bits has a significant effect on the overall performance. Table 2 compares three cases when we use 16, 32 bits or 64 bits for Morton codes. Note that the configuration column of the table explains the order of the axes and the number of bits allocated to each axis as well.

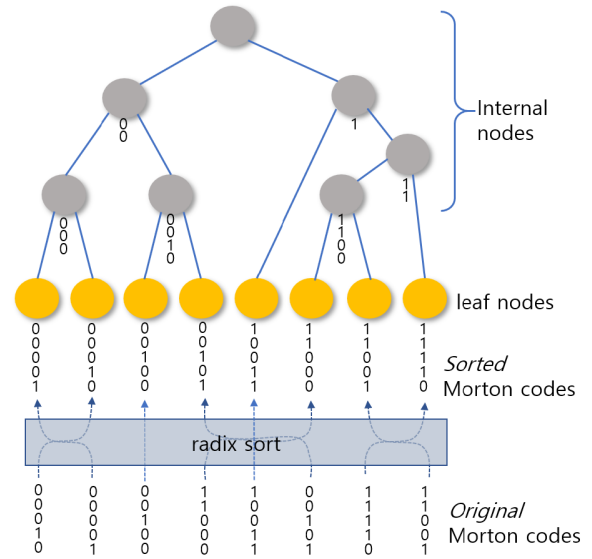


FIGURE 4. Construction of linear bounding volume hierarchy: Grey nodes are internal nodes and yellow nodes are leaf nodes.

TABLE 2. Sorting time comparison (Note that configuration states the order of the axes and the number of bits for the axis as well).

# of bits	Sorting time(μ s)	Configuration
16	98.13	x(7) \rightarrow z(6) \rightarrow y(3)
32	191.1	x(15) \rightarrow z(14) \rightarrow y(3)
64	403.3	x(31) \rightarrow z(30) \rightarrow y(3)

If we increase the number of bits for the Morton code, detailed collision detection is possible because each object occupies a separate leaf node. Otherwise, there is a chance that two objects can have a exactly the same Morton code if not enough bits are allocated. In this paper, we propose to use the different sets of Morton codes depending on the average number of pixels of all objects. The average number of pixels is calculated in the visibility map counting step. The logic behind this is similar to that of the visibility map. If the camera is positioned far from the objects, then collisions are almost invisible. Precise collision tests are not necessary in this case. Given the average number of pixels of all objects, say μ , the number of bits for the Morton codes, denoted as m , is determined by the formula 1, where a and b are the predefined thresholds.

$$\begin{cases} 0 \leq \mu < a \Rightarrow m = 16 \\ a \leq \mu < b \Rightarrow m = 32 \\ \geq b \Rightarrow m = 64 \end{cases} \quad (1)$$

After adding leaf nodes corresponding to all objects sorted by Morton codes, internal nodes of the LBVH are constructed by recursively finding common bits between adjacent nodes. This process can be optimized by applying the parallel algorithm proposed in [6]. See this paper for more details.

Once the LBVH is constructed, the actual collision test is done by traversing the tree starting from the root internal node

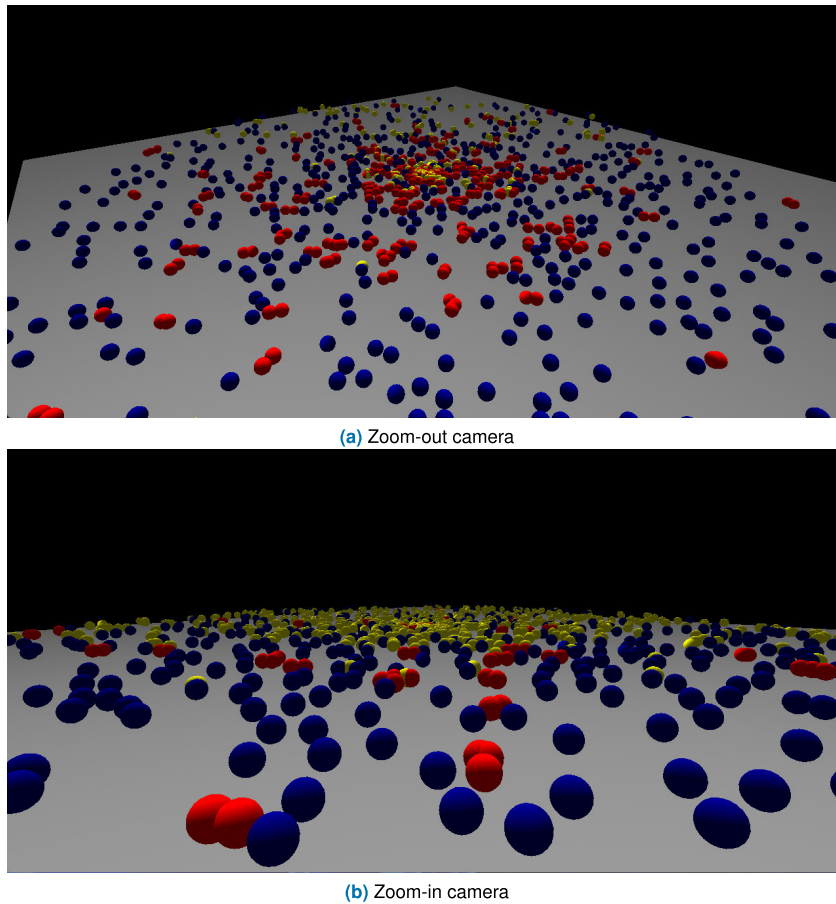


FIGURE 5. Collision Testing (number of objects = 2000).

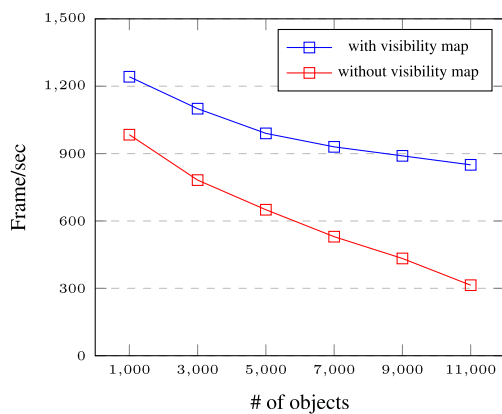


FIGURE 6. Performance graph with/without visibility map.

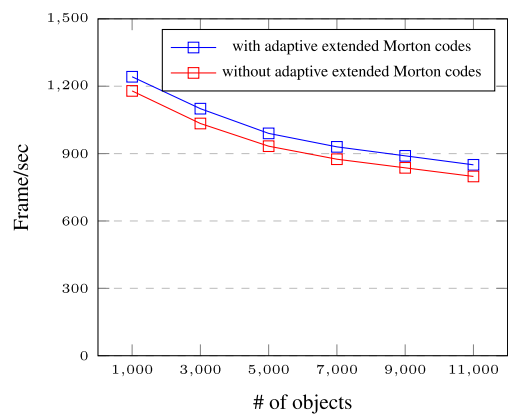


FIGURE 7. Performance graph with/without adaptive extended Morton codes.

and diving through the internal nodes in the tree to find leaf nodes that intersect with the bounding sphere for that thread's object.

IV. EXPERIMENTS

We have conducted experiments to verify the proposed algorithms. The while test system is built with OpenGL and

GLSL language on Windows 11 platform. The hardware specification includes Intel® 11th Gen. i-7-11700 CPU and Nvidia RTX™ 3080 graphics card. Most of the algorithms are implemented using Compute Shader 4.6 version. Figure 5 shows the screenshots of the test system where 2,000 objects move in a circular motion. We set different speeds for all

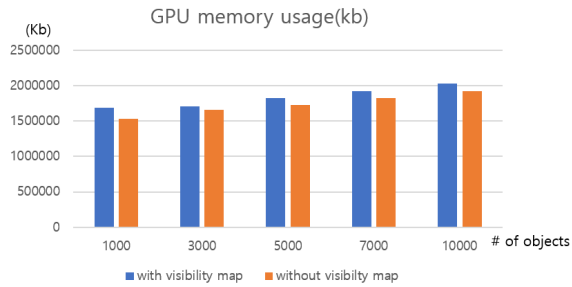


FIGURE 8. GPU memory usage as we increase the number of objects.

objects so that they can easily bump into each other. Note that the final rendered screen has a resolution of 1280×720 .

For constructing the LBVH, we choose the method proposed by [15] because it maximizes the parallel performance of GPU although the original algorithm is modified to support dynamic changing Morton codes.

In the pictures, the red colored objects are in the middle of the collision. The blue colored objects mean that they are not colliding with other objects although they are visible from the current camera. The yellow colored objects mean that their number of pixels is below the threshold in the visibility map.

Figure 6 shows a performance graph where the blue line shows the frame rate trend as we increase the number of objects with the proposed visibility map, while the red line shows the case when we don't use the map. The visibility map also has a resolution of 1280×720 . From the experiment, we found that the proposed method is about 70% faster than the case when we didn't use it in average. This difference becomes larger when we increase the number of objects. For this experiment, we set $\lambda=200$ for the threshold.

Figure 7 shows a performance graph where the blue line shows the frame rate trend when we use the adaptive extended Morton codes explained in chapter III, while the red line shows the trend when we don't use the adaptive extended Morton codes. In this experiment, we set $a = 100$ and $b = 400$ where three levels of number of bits were applied for the Morton code. From this experiment, we knew that the adaptive extended Morton codes has around 5% performance improvement.

V. DISCUSSION AND CONCLUSION

In this paper, we proposed a visibility map based collision detection algorithm for massive number of moving objects. The visibility map is constructed on top of the general g-buffer structure, where each pixel of the map indicates the ID number of moving objects. Thanks to the deferred rendering technique where z-culling is implicitly done with the depth map, counting the number of pixels of the visibility map reflects how visible the object is. We use this information to decide whether or not to send the object to the complicated collision checking process. We found that when we use the visibility information, the number of objects entering the test is significantly reduced, which improves

the overall performance because only objects passing the visibility are used for LBVH construction. Our algorithm used the parallel construction of LBVH algorithm proposed in [15]. However, we proposed an adaptive extended Morton codes where different set of bits are used depending on the overall visibility circumstance. We verified that the proposed methods have a positive effect on the performance through a series of experiments. A disadvantage of the visibility map would be the additional memory consumption. Since we use the same size of visibility map as the final rendered screen, the memory usage would increase when rendering a high resolution image such as 4K. However, the visibility map does not need to be the same size as the final image because the relative number of pixels of each object is important. Instead, we can use the fixed size of the map regardless of the final rendered image, which would be our future work. Also, the additional memory requirement for the visibility map would not be that large because the overall memory usage in LBVH would be reduced due to the early culling. To verify this, we compared the GPU memory usage in Figure 8 as we increase the number of objects. In this case, we used a 1024×768 visibility map. To find out the memory usage, we call a function that returns the current available memory usage and the total amount of memory (`GL_GPU_MEM_INFO_TOTAL_AVAILABLE_MEM_NVX`, `GL_GPU_MEM_INFO_CURRENT_AVAILABLE_MEM_NVX`). As we can see in the figure, there is no significant memory overhead when we use the visibility map.

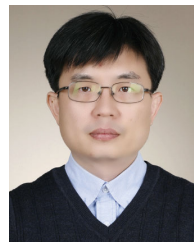
Also, there is a limitation on the number of Atomic counter buffers, which is different from vendor to vendor. We plan to use other buffers to record the pixel count information if the number of objects is larger than the atomic buffer limit.

In conclusion, we proposed a simple and efficient method that culls out the insignificant moving objects for collision testing from the camera setting. This method builds a visibility map on the top of g-buffer structure and lets the GPU read this data in parallel manner. Since only objects that pass the visibility test are used for collision test, it can improve the overall performance when we construct the LBVH. Also, the proposed dynamic Morton codes have a positive effect on the performance as well. From various several experiments, we know that our method does not require a huge memory overhead.

REFERENCES

- [1] P. Jiménez, F. Thomas, and C. Torras, "3D collision detection: A survey," *Comput. Graph.*, vol. 25, no. 2, pp. 269–285, Apr. 2001.
- [2] C. Ericson, *Real-Time Collision Detection*. Boca Raton, FL, USA: CRC Press, 2004.
- [3] D. Meister, S. Ogaki, C. Benthin, M. J. Doyle, M. Guthe, and J. Bittner, "A survey on bounding volume hierarchies for ray tracing," *Comput. Graph. Forum*, vol. 40, no. 2, pp. 683–712, May 2021. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.142662>
- [4] T. Karras. (Dec. 2012). *Thinking Parallel, Part II and III: Tree Traversal on the GPU*. [Online]. Available: <https://developer.nvidia.com/blog/thinking-parallel-part-iii-tree-construction-gpu>

- [5] C. Lauterbach, Q. Mo, and D. Manocha, "GProximity: Hierarchical GPU-based operations for collision and distance queries," *Comput. Graph. Forum*, vol. 29, no. 2, pp. 419–428, May 2010, doi: [10.1111/j.1467-8659.2009.01611.x](https://doi.org/10.1111/j.1467-8659.2009.01611.x).
- [6] T. Karras and T. Aila, "Fast parallel construction of high-quality bounding volume hierarchies," in *Proc. 5th High-Performance Graph. Conf.*, Jul. 2013, pp. 89–99, doi: [10.1145/2492045.2492055](https://doi.org/10.1145/2492045.2492055).
- [7] F. M. Chitalu, C. Dubach, and T. Komura, "Binary ostensibly-implicit trees for fast collision detection," *Comput. Graph. Forum*, vol. 39, no. 2, pp. 509–521, May 2020, doi: [10.1111/cgf.13948](https://doi.org/10.1111/cgf.13948).
- [8] J. Jakob and M. Guthe, "Optimizing LBVH-construction and hierarchy-traversal to accelerate KNN queries on point clouds using the GPU," *Comput. Graph. Forum*, vol. 40, no. 1, pp. 124–137, Feb. 2021, doi: [10.1111/cgf.14177](https://doi.org/10.1111/cgf.14177).
- [9] X. Wang, M. Tang, D. Manocha, and R. Tong, "Efficient BVH-based collision detection scheme with ordering and restructuring," *Comput. Graph. Forum*, vol. 37, no. 2, pp. 227–237, May 2018, doi: [10.1111/cgf.13356](https://doi.org/10.1111/cgf.13356).
- [10] D. Meister and J. Bittner, "Parallel locally-ordered clustering for bounding volume hierarchy construction," *IEEE Trans. Vis. Comput. Graphics*, vol. 24, no. 3, pp. 1345–1353, Mar. 2018.
- [11] D. Bozidar and T. Dobravec, "Comparison of parallel sorting algorithms," Dept. Comput. Inf. Sci., Univ. Ljubljana, Ljubljana, Slovenia, Tech. Rep., Nov. 2015.
- [12] M. Vinkler, J. Bittner, and V. Havran, "Extended morton codes for high performance bounding volume hierarchy construction," in *Proc. High Perform. Graph.* New York, NY, USA: Association for Computing Machinery, 2017.
- [13] D. J. Tracy, S. R. Buss, and B. M. Woods, "Efficient large-scale sweep and prune methods with AABB insertion and removal," in *Proc. IEEE Virtual Reality Conf.*, Mar. 2009, pp. 191–198.
- [14] F. Liu, T. Harada, Y. Lee, and Y. J. Kim, "Real-time collision culling of a million bodies on graphics processing units," *ACM Trans. Graph.*, vol. 29, no. 6, pp. 1–8, Dec. 2010, doi: [10.1145/1882261.1866180](https://doi.org/10.1145/1882261.1866180).
- [15] T. Karras, "Maximizing parallelism in the construction of BVHs, octrees, and $k-d$ trees," in *Proc. 4th ACM SIGGRAPH/Eurographics Conf. High-Perform. Graph.* Goslar, Germany: Eurographics Association, 2012, pp. 33–37.
- [16] H. Jin, Z. Liu, T. Wu, and Y. Wang, "The research of collision detection algorithm based on spatial subdivision," in *Proc. Int. Conf. Comput. Eng. Technol.*, Jan. 2009, pp. 452–455.
- [17] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, "Fast BVH construction on GPUs," *Comput. Graph. Forum*, vol. 28, no. 2, pp. 375–384, Apr. 2009, doi: [10.1111/j.1467-8659.2009.01377.x](https://doi.org/10.1111/j.1467-8659.2009.01377.x).
- [18] D. Wodniok and M. Goesele, "Construction of bounding volume hierarchies with SAH cost approximation on temporary subtrees," *Comput. Graph.*, vol. 62, pp. 41–52, Feb. 2017. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0097849316301376>
- [19] I. Wald, S. Boulos, and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchies," *ACM Trans. Graph.*, vol. 26, no. 1, p. 6, Jan. 2007, doi: [10.1145/1189762.1206075](https://doi.org/10.1145/1189762.1206075).
- [20] D. Kopta, T. Ize, J. Spjut, E. Brunvand, A. Davis, and A. Kensler, "Fast, effective BVH updates for animated scenes," in *Proc. ACM SIGGRAPH Symp. Interact. 3D Graph. Games*, Mar. 2012, pp. 197–204, doi: [10.1145/2159616.2159649](https://doi.org/10.1145/2159616.2159649).
- [21] M. Yin and S. Li, "Fast BVH construction and refit for ray tracing of dynamic scenes," *Multimedia Tools Appl.*, vol. 72, no. 2, pp. 1823–1839, Sep. 2014, doi: [10.1007/s11042-013-1476-y](https://doi.org/10.1007/s11042-013-1476-y).
- [22] T. Ize, I. Wald, and S. G. Parker, "Asynchronous BVH construction for ray tracing dynamic scenes on parallel multi-core architectures," in *Proc. Eurographics Symp. Parallel Graph. Visualizat.*, J. M. Favre, L. P. Santos, and D. Reiners, Eds. Goslar, Germany: Eurographics Association, 2007.
- [23] J. Pantaleoni and D. Luebke, "HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry," in *Proc. Conf. High Perform. Graph.* Goslar, Germany: Eurographics Association, 2010, pp. 87–95.
- [24] K. Garanzha, J. Pantaleoni, and D. McAllister, "Simpler and faster HLBVH with work queues," in *Proc. ACM SIGGRAPH Symp. High Perform. Graph.* New York, NY, USA: Association for Computing Machinery, Aug. 2011, pp. 59–64, doi: [10.1145/2018323.2018333](https://doi.org/10.1145/2018323.2018333).
- [25] J. A. Orenstein, "Spatial query processing in an object-oriented database system," *ACM SIGMOD Rec.*, vol. 15, no. 2, pp. 326–336, Jun. 1986, doi: [10.1145/16856.16886](https://doi.org/10.1145/16856.16886).
- [26] L. Ha, J. Krueger, and C. T. Silva, "Fast four-way parallel radix sorting on GPUs," *Comput. Graph. Forum*, vol. 28, no. 8, pp. 2368–2378, Dec. 2009.
- [27] S. Sengupta, A. Lefohn, and J. Owens, "A work-efficient step-efficient prefix sum algorithm," *Proc. Workshop Edge Comput. Using New Commodity Architectures*, Jan. 2006, pp. 26–27.
- [28] W. Engel, "Designing a renderer for multiple lights: The light pre-pass renderer," in *ShaderX7: Advanced Rendering Techniques*. Needham, MA, USA: Charles River Media, 2008.
- [29] S. Hargreaves, "Deferred shading," presented at the Game Developers Conf. (GDC), 2004.
- [30] M. Deering, S. Winner, B. Schediwi, C. Duffy, and N. Hunt, "The triangle processor and normal vector shader: A VLSI system for high performance graphics," *ACM SIGGRAPH Comput. Graph.*, vol. 22, no. 4, pp. 21–30, Jun. 1988, doi: [10.1145/378456.378468](https://doi.org/10.1145/378456.378468).
- [31] D. S. John Kessenich and G. Sellers, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 With SPIR-V*. Reading, MA, USA: Addison-Wesley, 2016.
- [32] M. Elteir, H. Lin, and W.-C. Feng, "Performance characterization and optimization of atomic operations on AMD GPUs," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2011, pp. 234–243.



MANKYU SUNG received the B.S. degree in computer sciences from Chungnam National University, Daejeon, Republic of Korea, in 1993, and the M.S. and Ph.D. degrees in computer sciences from the University of Wisconsin–Madison, Madison, WI, USA, in 2005. From January 1995 to July 2012, he was with the Digital Contents Division, ETRI, Daejeon. Since March 2012, he has been an Associate Professor with the Department of Game Software, Keimyung University, Daegu, Republic of Korea. His current research interests include computer graphics, deep learning applications, computer animation, computer games, and human–computer interaction. He is a member of the ACM.

...