REAL-TIME IMPULSE-BASED RIGID BODY SIMULATION AND RENDERING

A Thesis

by

CAN YUKSEL

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2007

Major Subject: Visualization Sciences

REAL-TIME IMPULSE-BASED RIGID BODY SIMULATION AND RENDERING

A Thesis

by

CAN YUKSEL

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Donald House
Committee Members,	John Keyser
	Scott Schaefer
Head of Department,	Mark J. Clayton

May 2007

Major Subject: Visualization Sciences

ABSTRACT

Real-time Impulse-based Rigid Body

Simulation and Rendering. (May 2007)

Can Yuksel, B.S., Bogazici University

Chair of Advisory Committee: Dr. Donald House

The purpose of this thesis is to develop and demonstrate a physically based rigid body simulation with a focus on simplifications to achieve real-time performance. This thesis aims to demonstrate that by improving the efficiency with simplified calculations of possible bottlenecks of a real-time rigid body simulation, the accuracy can be improved. A prototype simulation framework is implemented to evaluate the simplifications. Finally, various real-time rendering features are implemented to achieve a realistic look, and also to imitate the game-like environment where real-time rigid body simulations are mostly utilized.

A series of demonstration experiments are used to show that our simulator does, in fact, achieve real-time performance, while maintaining satisfactory accuracy. A direct comparison of this prototype with a commercially available simulator verifies that the simplified approach improves the efficiency and does not damage the accuracy under our test conditions. Integration of rendering elements like advanced shading, shadowing, depth of field and motion blur into our real-time framework also enhanced the perception of simulation outcomes.

To my family

ACKNOWLEDGMENTS

First and foremost, I would like to thank Dr. Donald House for his valuable mentorship, encouragement, and confidence in me. He deeply inspired me not only as a successful academician, but also as a person with a great sense of humor. Secondly, I would like to extend my deepest gratitude to my committee members, Dr. John Keyser and Dr. Scott Schaefer, for their insightful comments and thorough examination. I express a special thank you to Leslie Feigenbaum, my boss, for his immense support and friendship. I also acknowledge my friends and Margaret Lomas from Viz-lab for their assistance and sincere friendship.

I would like to thank my family to whom I dedicate this thesis. I thank my parents, Aysen and Sinasi Yuksel, for their unconditional support throughout my life, and to my wonderful wife, Kamer Yuksel, for her love, support, and inspiration. And last, but not least, I would like to thank my brother, Cem Yuksel, for being a constant source of support throughout my educational journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

The purpose of this thesis is to develop and demonstrate a physically based rigid body simulation with a focus on simplifications to achieve real-time performance. The approach taken in this study was experimental refinement. To efficiently carry out experiments on proposed simulation elements, I implemented a prototype simulator. This was used to develop and refine features like collision detection, collision response and friction. It was also used to evaluate various real-time rendering features such as advanced shading, shadowing, bloom effect, motion blur, and depth of field to achieve a realistic look. The architecture of the prototype simulation exploits the processing capabilities of both the CPU and the GPU. The CPU was utilized for all simulation calculations and the GPU was dedicated for rendering purposes.

The conventional approach in real-time applications of rigid body simulation is to use primitive objects; such as cubes and rectangular solids, as they can be simulated efficiently. Real-time rigid body simulations are commonly used by gaming companies; however their methods for making frameworks fast and efficient are trade secrets. In this study, I propose simplified solutions to the common obstacles that might become a bottleneck for a rigid body simulation framework; both for simulation and rendering. I preferred to use rectangular objects with the intention to imitate the traditional approach, but not all the simplifications presented here are limited to rectangular objects.

Physical accuracy is the most important criterion of an engineering simulation, but in computer graphics visual believability is the most important criterion. Even

The journal model is *IEEE Transactions on Automatic Control.*

though this believability is mostly subjective, it is tightly connected to how well the simulation mimics the real physical model. On the other hand, for any real-time application computational efficiency is crucial. The system has only a small fraction of a second to complete all simulation calculations and to render the resulting frame. This time restriction is one of the main motivations behind the simplifications presented in this thesis. However, these simplifications designed to speed up the computation must not sacrifice the believability of the simulation.

One of the keys to believability is rendering, since human perception of an object's 3D motion depends highly on its rendering and shading. Dynamic object shadows significantly improve the perception of an object's position and orientation in a 3D environment. Since the final result is a 2D image, a 'depth of field' effect can make a significant difference on our perception of depth. Likewise, 'motion blur' helps moving objects stand out from stationary ones. Taken together, these rendering effects not only improve the realism of the final image, but also contribute to the believability of the dynamic simulation.

CHAPTER II

BACKGROUND

This section summarizes some of the existing simulation and rendering methods that are relevant to the thesis. The use of bounding volumes [1] to approximate the shape

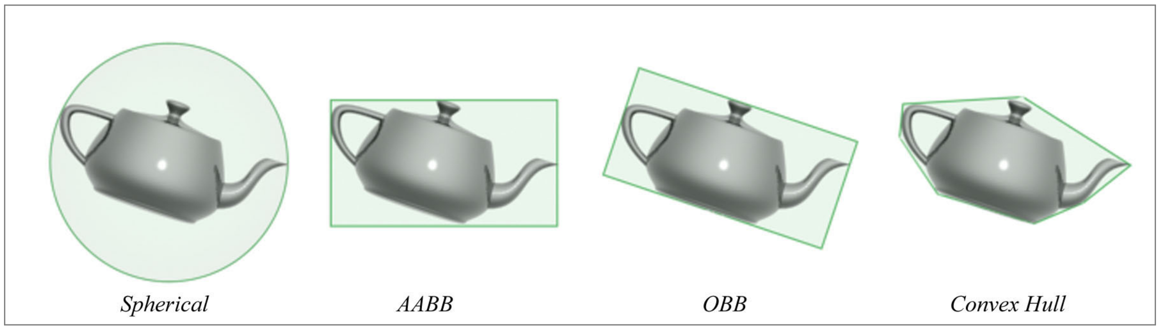
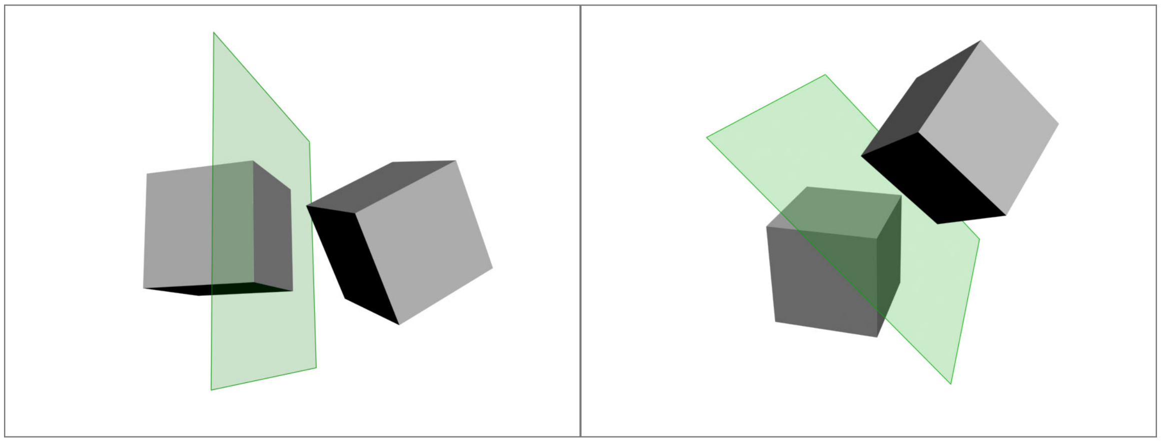


Fig. 1. Some Types of Bounding Volumes. Spherical, AABB (Axis Aligned Bounding Box), OBB (Object-oriented bounding box), Convex Hull

of a polygonal mesh model with a representative primitive object is illustrated in Figure 1. This method has been used for many different applications since the earliest times of computer graphics. For collision detection, the premise is that intersections between bounding volumes are much easier to detect than collisions between the actual surfaces. If the bounding volumes do not intersect, then the surfaces themselves can not intersect. This method is appropriate for fairly simple geometries. Bounding volume hierarchies consist of many bounding volumes and are typically used for complex geometries [1] [2].

Collision detection has two phases: (1) the broad phase, and (2) the narrow phase. The broad phase stands for the cases where two objects's bounding volumes are far separated from each other and no detailed collision check is required. The narrow phase stands for the cases when intersection is detected (at the broad phase) between bounding volumes. This intersection only means that bounding volumes are in penetration, but it does not necessarily mean that a real object-to-object collision is present. If a broad phase intersection is detected, then the narrow phase determines if the objects themselves are in collision. Only after determining that there is a real collision, the collision point is calculated. Compared to the narrow phase, the broad phase computation is much faster, and is used to minimize the number of the narrow phase computations. One of the earliest methods for narrow phase of



Separating Plane on Face Separating Plane on Edge

Fig. 2. Witnessing(Separating) Plane.

collision detection is the polyhedral collision detection algorithm [3]. This method is a brute-force approach since every possible edge-face combination is checked between the objects where bounding boxes are not separating them. On the other hand, Baraff's witnessing plane (separating plane) approach, as illustrated in Figure 2, is a speedup to the brute-force approach [4]. If two convex objects are separate from each other, this means that there is a separating plane that lies between them. This plane is defined on the surfaces or edges of the objects. The method finds that plane and caches it as an initial guess for the following time-steps. Although Gottschalk's separating axis theorem [5] is a similar approach using separating planes, it requires less plane checks. Since Baraff's separating plane is embedded onto the surface of the object, each face and edge combination must be tested. It requires 48 (12 for faces of each object and 36 for edges of one object) checks for the worst case. On the other hand, the separating axis approach requires only 15 (6 for faces and 9 for edges) checks for the worst case.

For finding the exact collision time and point, Hahn processed collisions by chronologically backing up to the time of collision [6]. This back-tracing method becomes quite slow when many objects are in collision at the same time. Mirtich's timewarp algorithm similarly backs up the time, but for only colliding objects [7]. Non-colliding objects evolve forward in time. Since a real-time application requires frequent synchronization of all objects, the potential of this method can not be fully utilized in a real-time rigid body simulation framework. Collision Response is the

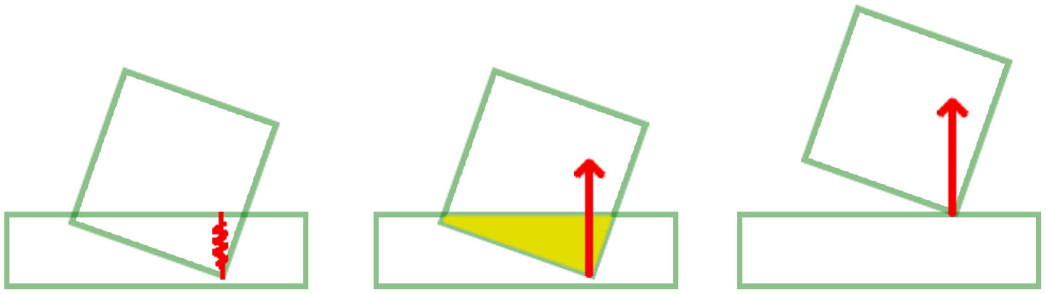


Fig. 3. Examples of Some Collision Response Methods. Illustrated methods are Penalty method using springs, Volumetric penalty method, and Impulse-based method respectively.

phase where contact forces are calculated for the collision points. Baraff [8] proposed the constraint method for collision response. This method determines constraint forces at the contact points and solves them continuously. This method will prevent penetration, but multiple contacts will drastically slow down the simulation. McKenna and Zeltzer calculated contact forces using the penalty method [9]. This analytic constraint method adds temporary springs to the contact points to simulate contact force. The amount of force is proportional to the penetration of the objects in the penalty method. Different penetration criteria differentiate similar methods as illustrated in Figure 3. The volumetric calculation of the penetration is an enhancement to this technique [10]. Mirtich [11] developed the impulse-based approach. This method does not use explicit constraints like the previous methods, where the constraints define forces that are applied to the interacting objects. In the impulse-based approach, impulses are used instead of forces to directly change the velocities of the interacting objects. This method is more stable, robust, and easy to implement.

Stacking of rigid objects is a difficult problem, especially for impulse-based systems. [12] proposes a shock propagation method that transfers momentum through interacting objects, and stabilizes the system for stacking. Their algorithm also supports non-convex objects. [13] introduces a reversible freezing approach that excludes non-interacting objects from the simulation.

Since the priorities for a real-time simulation are different than for an offline simulation, it is important to make proper selections from the existing methods to implement a real-time rigid body simulator. Egan's study focuses on real-time rigid body simulation and evaluates existing methods for their contribution from a real-time perspective [14].

Real-time rendering methods are highly developed and well known. Dynamic shadow computation significantly enhances the user's 3D perception of the scene. Without shadows it is very hard to understand the 3D positions of objects in an environment. There are many ways to compute shadows, but the most popular shadowing method is shadow mapping [15]. This method is fast and robust. Moreover, with the GPU implementation, fragment programming gives precise control over the shadow mapping [16].

The Bloom effect arises from the fact that the intensity of the monitor is limited. Even the absolute white color, which generates the maximum intensity, is not enough to imitate the brightness of a light source. However, this brightness effect can be simulated by spreading bright values to its neighboring pixels [17].

The depth of field phenomenon arises from the fact that the focus of an object depends on its distance from the lens [18] [19]. The precise simulation of this real phenomenon is computationally expensive. Therefore, image based post-filtering techniques approximate this behavior and they are commonly used even for offline applications. Yu [20] explains a post-filtering method to imitate this natural phenomenon in real-time.

The motion blur helps the moving objects to be distinguished from the resting objects. Since a camera's shutter is open for a finite amount of time, a moving object will cause over-writing on the film, causing motion induced blur [19] [21]. Again, the precise simulation of this real phenomenon is computationally expensive and image based post-filtering techniques can be used to approximate this behavior.

CHAPTER III

SIMULATION METHODOLOGY

The implementation of a rigid body simulation system is explained in detail in Baraff and Witkin's course notes [22], and these course notes constitute the basis for my own system. The main purpose of this section is to explain the approaches and simplifications developed for the most common issues of a rigid body simulation.

A. Collision Detection

Within a rigid body simulation, collision detection is the component that has the most drastic effect on the accuracy of a rigid body simulation. But at the same time, this phase is the most computationally costly element of the simulation.

Collision Detection has two objectives; finding (a) the colliding objects and (b) their collision points. For finding the objects in collision, the simulation must be robust and accurate because any miss-detection of colliding objects may lead to very unrealistic results. Finding the exact collision point makes the collision detection phase computationally costly. Its robustness is crucial for accuracy. However, as long as the collision response is able to calculate correct impulses, exact collision points are not fundamental for the believability of the simulation. This observation is exploited in the simplifications carried out to make this step faster, which are described in detail in the Narrow Phase (Chapter III.A.2).

1. Broad Phase

The goal of the broad phase is to determine the set of possibly colliding objects and exclude the remaining objects from detailed collision detection in the narrow phase. The list of colliding objects in the broad phase is named the Active Collision List

(ACL). These are not necessarily real collisions, but the ACL strictly satisfies the fact that there can not be any real colliding pair that is not in the ACL. Once the ACL is generated in this step, it will be used in the narrow phase of collision detection.

For the broad phase of collision detection, I chose to use axis-aligned bounding boxes (AABB). Using an axis-aligned box as a representative object has several advantages. It is a very fast and robust method of collision checking due to the fact that it is rotationally invariant. The only downside is that it is not suitable for all geometries. Bounding box tightness depends on both the shape and the orientation of the geometry. For complex shapes, using an axis-aligned box may result in a very poor approximation. Since my simulation uses only rectangular objects, AABB is usually an appropriate method.

By definition, a bounding volume must cover its actual geometry for all times. Although AABB is rotationally invariant, the actual geometry inside an AABB rotates due to interactions with the environment. There are two ways to be sure that the actual geometry will be covered by its AABB. The first method is to use a static-sized AABB that works for the worst-case scenario. In the worst case scenario, the AABB becomes a cube that has sides which are equal to $\sqrt{(a^2 + b^2 + c^2)}$ (for a rectangular object with dimensions a, b, c). When the actual geometry is a cube, the static-sized AABB is 1.7 ($\sqrt{3}$) times bigger than the actual geometry size, which is an acceptable ratio. But, the ratio between actual size and AABB size gets significantly larger for long and thin rectangular objects and it starts to contradict with the notion that a bounding volume should be tight to its representative geometry. The second method is to use a dynamically sized AABB. In this method, the x-y-z extents values of the actual geometry's vertices are calculated for every time step giving the tightest AABB possible. The only drawback of this calculation is that finding extents becomes too costly for a complex object with lots of vertices. Thus, there is a trade off between having tighter AABBs with dynamically sizing and saving computation time using static-sized AABBs. In my simulation, I prefer the dynamically sized method as illustrated in Figure 4. The reason is that the actual geometries in my simulation are simple rectangular objects with 8 vertices. As a result, finding extents is not computationally expensive.

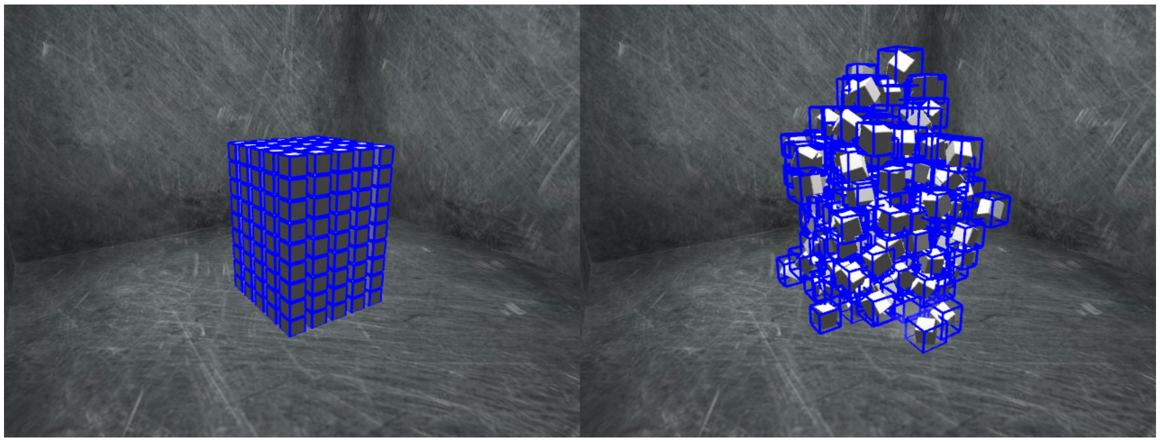


Fig. 4. Dynamically Sized AABB. The blue lines represent the bounding volumes. In the left image, the sizes of the bounding volumes are the same as its representative object because objects lie in axis aligned positions. In the right image, the sizes of the bounding volumes are bigger because the objects are rotated.

Since the AABB is axis-aligned, it is simple to use for collision detection. The AABB values are sorted in the x, y, z directions independently. Two objects can not be in collision unless their AABB min/max values (x, y, z) are overlapping along all three coordinate axes. If any two objects satisfy this condition, they are added into the Active Collision List (ACL) as a pair.

2. Narrow Phase

The goal of the narrow phase is to determine the real collisions using the ACL and to find the collision points where the objects are touching each other. In my simulation, two types of collisions are defined: (a) edge-to-edge and (b) face-to-vertex. The list of colliding objects in the narrow phase is named the Real Collision List (RCL). Once the RCL is generated in this step, it is used in the collision resolution step.

For my real-time simulation, the separating plane (SP) is an appropriate approach to check collisions. Finding the plane that separates two geometries is not computationally cheap, however, most probably, that plane will keep separating those geometries for following time steps. Thus, caching SP is very important for efficiency. The caching of SP starts when the pair is inserted into the ACL and stops when the pair leaves the ACL. This process repeats itself every time a pair of objects moves in and out of the ACL.

In the framework, once a SP is found, it is stored. However, this plane may not separate the pair for the following time step. The most probable reason for this is the rotations of the objects. In the next time step, if the old SP is not separating anymore, the program will look for a new SP. It is important to choose wisely which SP type to start with. In my implementation, I prefer to use an alternating method for seeking the new SP. Since the occurrence of a non-valid old SP is most probably due to rotations, the best guess in terms of SP method would be the other SP method. For example, if the old SP was a face SP and if it is not separating at this time step, the program starts by looking for an Edge SP, and vice-versa.

There are three different situations in which there will be no old SP. The first case is the next time step after a collision. By definition, a collision means there is no possible SP to separate the pair. The second case is that, at the end of the every time step, after object collisions are resolved, the exiting SP is re-checked to be sure that it is still separating. The reason for this double check is the displacements at the end of the collision resolution step and it is explained in detail in the Separation (Chapter III.C.1). During this process, if the existing SP no longer separates the pair, the SP is deleted. The third case is the time when a pair enters or re-enters the ACL, since the stored SP is deleted every time a pair goes out of the ACL. As a result, there may not be an old SP to check for a pair in the ACL. The program starts by looking for a new face SP because an edge SP is computationally more expensive. In any case, if there is no possible new SP, it means collision for the pair. In this situation, the pair in the ACL is moved to the RCL and this finalizes the first goal of the narrow phase collision detection.

The second goal of the narrow phase collision detection is to find collision points where the pair touches each other. Finding the exact collision points is the most computationally demanding part of the collision detection algorithms. Luckily, the SP can also be used for very quickly estimating collision points. My approach of using SP for collision points does not always guarantee finding the exact collision points. However, for most of the cases, the estimated points are very close to the real collision points. The important fact here is that using the SP is a very fast method to detect usable collision points.

In order to find collision points with the SP, I assume that, if there is a real collision and an old SP is present, it indicates that the collision happens on that SP. For example, if the old SP is an edge SP, it indicates that an edge-to-edge collision is present at this time step. This assumption saves time because finding collision points without SP information is not trivial for penetrated objects. This is described in detail in the Separation (Chapter III.C.1). In an edge-to-edge collision case, there exists one collision point and detecting it is straight forward using two edges associated with the old edge SP. However, in the case of face-to-vertex collision, there may be multiple collision points.

My second assumption is that any points penetrating the old SP are treated as collision points in face-to-vertex collisions. With this assumption, the SP replaces the face of the colliding object. Since the SP is an infinite plane, some points penetrating that plane are not necessarily penetrating the actual face of the object. For similar sized objects, the error due to this assumption is negligible. However, when the size difference is big between objects and if the face SP lies on the small object, this assumption causes big deviations from the actual collision points as illustrated in Figure 5.a.

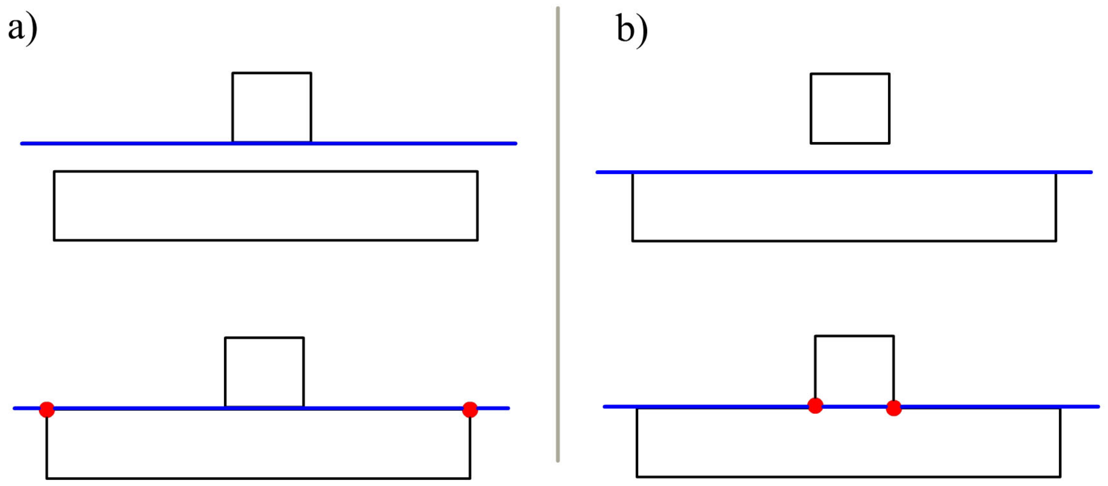


Fig. 5. Finding Collision Point with Separating Plane. (a) illustrates big error in finding the actual collision points. (b) illustrates the correction of this problem by choosing right object for separating plane.

The solution for this problem is to choose the right object for the face SP. For an edge SP, the result plane is unique and it uses the edge information that comes from both objects. For the face SP, the SP lies on one of the objects and is not unique. In many cases, both objects have a face available for a SP as illustrated in Figure 5, and the search stops when the program detects one. In my implementation, the program starts to look for a face SP with the bigger object as illustrated in Figure 5.b. In the case of small-to-big object collision, using an infinite SP is not a bad approximation for the face of the big object. Another benefit of this approach is that the bigger object has a lower tendency to move and rotate compared to the smaller object. This will result in a more stable SP, which will continue to separate the pair for more following time steps.

Another simplification is developed to handle multiple collision points between colliding pairs (only happens in the face-to-vertex collision case). Since my simulation is designed only for rectangular objects, it can be adapted to handle collisions faster for this special case. The common characteristic of rectangular objects is that they are symmetric with respect to their three local coordinate planes. Using this property, multiple collision points can be interpolated to one collision point that will give the same impact. Since the object is symmetric, opposite impacts will cancel out each other and they will be simplified to an impact that can be defined by one collision point, as illustrated in Figure 6. With this method, an object falling straight on the floor (or onto another object) will bounce back straight up due to one collision over one collision point that lies just beneath the object's center of mass, as illustrated in Figure 6.a. To make this approximation more accurate, a weighting component is added to the interpolation. As Figure 6.b illustrates, the penetration depth of the collision point defines the value of the weights.

B. Collision Response

The impulse-based approach is a very robust method for collision response [11]. There are no temporary springs or constraints to maintain under some criteria as in other methods. Moreover, it is easy to implement because all the interactions of the simulation are defined by simple impulses. Since the highest priority of the simulation is to achieve real-time performance, the impulse-based approach is suitable for this

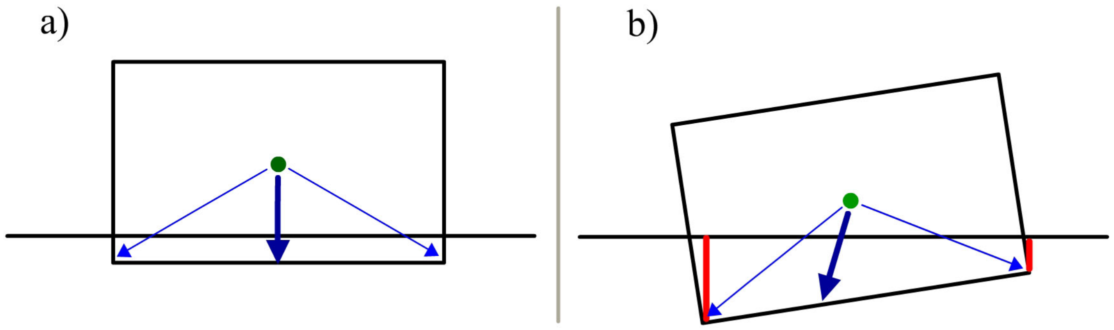


Fig. 6. Multiple Point Collision. (a) Illustrates interpolating multiple collision points to a single collision point. (b) Illustrates the penetration weights.

goal.

After colliding objects are determined with their collision points, the impulses are calculated in the collision response step.

1. Calculating Impulses

The impact (impulse) between objects can be calculated by [11]:

$$\mathbf{J} = \frac{-(1+\epsilon)[\mathbf{v}_A - \mathbf{r}_A\omega_A - \mathbf{v}_B - \mathbf{r}_B\omega_B] \cdot \mathbf{n}}{\frac{1}{m_A} + \frac{1}{m_B} - \mathbf{n} \cdot (\mathbf{r}_A I_A^{-1} \mathbf{r}_A \mathbf{n}) - \mathbf{n} \cdot (\mathbf{r}_B I_B^{-1} \mathbf{r}_B \mathbf{n})} \tag{3.1}$$

where for object i, m_i is its mass, I_i is its inertia tensor, $\mathbf{r}_i$ is its displacement between the center of mass and the collision point, $\mathbf{n}$ is the normal direction of the collision, $\mathbf{v}_i$ is its linear velocity, ω_i is its angular velocity, and ϵ is the coefficient of restitution.

Equation 3.1 is for collision between two objects and it can be simplified for the case when an object makes a collision with a fixed object. Since the fixed objects are assumed to have infinite mass, we have:

$$\mathbf{J} = \frac{-(1+\epsilon)[\mathbf{v}_A - \mathbf{r}_A\omega_A].n}{\frac{1}{m_A} - \mathbf{n}.(\mathbf{r}_A I_A^{-1} \mathbf{r}_A \mathbf{n})}$$

2. Friction

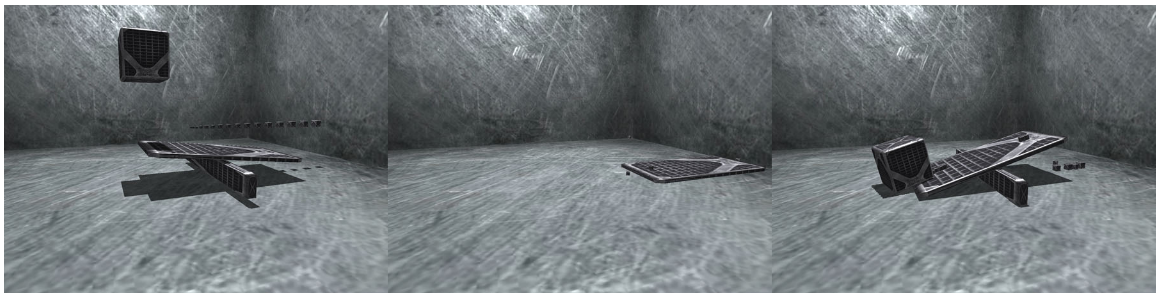


Fig. 7. Friction. The left image illustrates the initial state of the simulation. The other two images illustrate the final states. Without (middle) and with (right) friction.

Friction is a very important component for believability. In real world, there is not only surface friction but also air friction. Moreover, in the absence of friction, objects do not keep interacting as illustrated in Figure 7. This is not desirable, since the main purpose of building a mathematical framework beneath any simulation is the desire to observe these interactions.

I used Coulomb's friction model for my simulation and the direction of the friction force opposes the direction of the motion.

$$\mathbf{F} \leq \mu \mathbf{N}$$

where $\mathbf{F}$ is frictional force, μ is friction coefficient, and $\mathbf{N}$ is the force in the direction of the normal.

There are two types of friction: static and dynamic (kinetic). Static friction refers to the friction when the object is not moving. The dynamic friction is the friction when the object slides over a surface. The only difference in these friction types is magnitude (friction coefficient) and, usually, the static friction is bigger than the dynamic friction. In my implementation, I simply define a very low threshold for speed that differentiates the static and dynamic friction.

C. Collision Resolution

Collision resolution is the step that finalizes the whole collision operation. In my implementation, instead of resolving every pair's collision sequentially, all collision outcomes are calculated and stored for all colliding objects. Then, these stored outcomes are applied at the same time to the states of these colliding objects. The main reason why I prefer not to resolve collisions sequentially is that an object may be in collision with multiple objects and the outcomes would be different depending upon their resolving order. Another reason is that, after collisions are resolved, it is important to check if an object becomes frozen as explained in detail in Freezing (Chapter III.D). In a sequential resolving process, an object may get in and out of the frozen state multiple times in the collision resolution process.

1. Separation

In an iterative simulation of rigid bodies, penetration is inevitable. Some collision response methods even use this penetration to calculate impact forces. However, the impulse-based method enforces non-penetration. Avoiding penetration is a difficult problem. Most simulation methods use a back-tracing technique to avoid penetration by finding the exact collision point and time. The back-tracing method backs up time recursively to find the actual collision time and point till the penetration is detected. This method is not always guaranteed to converge and is computationally very costly.

In my simulation, I used a simple and robust method to separate penetrating objects. My method is to externally apply instantaneous displacements to separate penetrated objects at the end of the collision resolution step. To calculate these displacements, my approach is to use the old SP as a reference for separation and to move the objects away from each other in the direction of that SP's normal. I also add a weighting term to these displacements. The amount by which objects are pushed away from each other varies inversely with their masses. The reason for weighting by masses is to ensure that these displacements do not inject external potential energy to the system.

The problem becomes complicated when an object is in contact with multiple objects. Any separation due to one contact may cause deeper penetrations between other contacts. In my implementation, I used displacement vectors to handle this problem. Since all the collision calculations are carried out at the same time, it would not be wise to simply add these displacement vectors, as illustrated in Figure 8.a. Averaging the sum of the displacement values with the number of colliding objects may not result in total separation even for the simple case as illustrated in Figure 8.b. Since final displacements represent instantaneous position changes of the objects, it

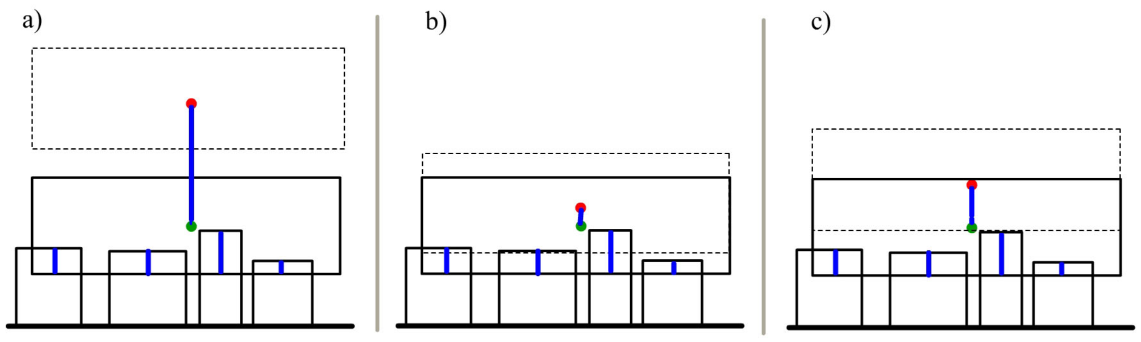


Fig. 8. Separation Methods in a Simple Penetration Case. (a) Using the sum of displacements causes too much separation. (b) Averaging the total sum may not result in total separation. (c) My approach.

is very important to calculate them properly. If the displacements are bigger than they are supposed to be, the result will look very strange. In a simulation, these inaccurate instantaneous jumps would draw the viewer's attention, detracting from believability. On the other hand, if they are too small to separate the penetrating objects, the objects will remain in penetration for the following time steps. This will cause an excessive call of computationally costly collision functions, slowing down in the simulation and jeopardizing its accuracy.

As illustrated in Figure 9.b, my final approach is to use the first possible displacement as a reference. Then the algorithm separates other displacement vectors according to their directions with respect to the reference displacement. In the first option, if they lie in the same direction with the reference displacement, they contribute to the reference displacement. The contribution is done by adding its perpendicular component to the reference displacement. For the horizontal component, it is compared with the reference displacement and the biggest value is selected to be the horizontal component of the new reference displacement as illustrated in Figure 9.b. In the second option, if they lie in the opposite direction of the reference displacement, they are simply added to the reference displacement as illustrated in Figure 9.b. With these steps, the reference displacement will transform to a final displacement that will cover all other displacements. This approach, similar to other methods, does not guarantee a total separation of colliding objects in the scene as illustrated in Figure 9. However, it has the potential to work for most probable simple cases as illustrated in Figure 9.c and, most importantly, the computational burden of this algorithm is negligible.

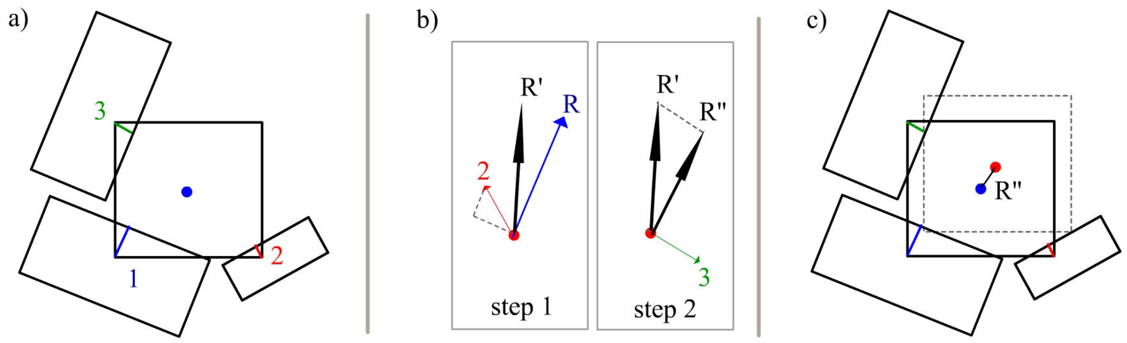


Fig. 9. My Separation Approach in a Complex Penetration Scenario. (a) Labeling penetrations randomly. (b) Updating reference displacement. (c) Applying the final state of the reference displacement.

D. Freezing

Resting contact, where objects are colliding but sliding on each other, is not defined in my simulation. However, I do use the concept of freezing [13]. A frozen object does not move and can be removed from the dynamic calculations. To become frozen, an object must satisfy three criteria:

- Its kinetic energy must be below a certain threshold.
- It must continue in this low energy state over a certain time.
- It must have a real collision with another object in that time step. This criterion assures that the object is touching another object.

The total kinetic energy can be calculated as:

$$\frac{1}{2}M\left|\mathbf{v}\right|^2 + \frac{1}{2}\omega^T I\omega$$

The freezing method saves computation time and stabilizes the simulation. Due to the gravitational field, combined with numerical instabilities and frictional forces, objects have the tendency to keep moving indefinitely. Even when objects should stop completely, discrete time steps cause the objects to vibrate over the surface due to these factors. The bigger the time steps become, the bigger these errors are. The 'freezing' approach helps the simulation to ignore almost stopped objects and use the computation time for dynamically interacting objects as illustrated in Figure 10. When an object becomes frozen, its velocities (linear and angular) are set to zero

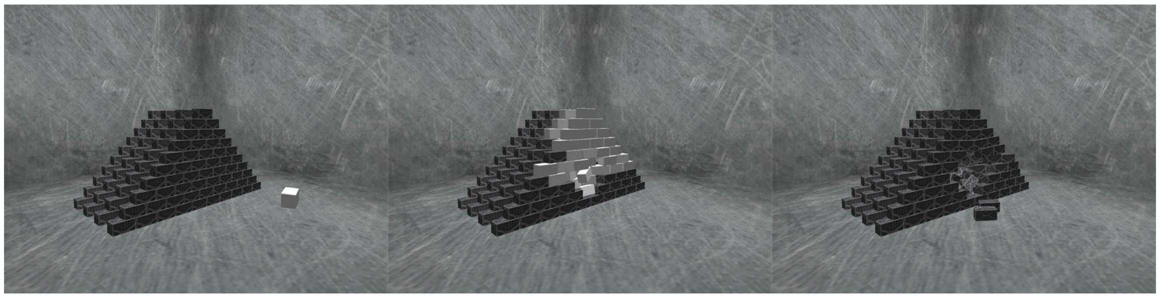


Fig. 10. Freezing of the Objects. In the figures, the textured objects are frozen; white objects are active. In the left image, the stacked objects are frozen in their initial state. As illustrated in middle image, with the impact of the object thrown, some of the stacked objects become active. In the right image, all objects are again frozen in their final state.

and it is not iterated by the simulation. This frozen state is reversible; there are two cases that cause an object to get out of the frozen state. In the first case, if an object is pushed by a force that would result in a kinetic energy more than a certain threshold, the object escapes from the frozen state. Since escaping from the frozen state is instantaneous while entering requires some time, the threshold is different for entering and escaping. In my simulation, I set the escaping threshold to be twice that of the entering threshold. The second case of escaping from the frozen state is due to a contact object (CO). As stated in the third criteria of becoming frozen, every frozen object must have a CO. If object A is in collision with object B and if object B has a lower z position than object A, object B is the CO for object A. If the object A has multiple collisions, the colliding object that has the lowest z position is the CO for object A. The purpose of CO is to decide a contact object that the actual object may rest upon. In the simulation, frozen objects are neglected from iteration unless their CO is frozen at the same time. With this CO method, it is possible to activate a frozen object sitting at the top of another object when the bottom object becomes activated.

In my simulation, the only external force is gravity. Consequently, my CO approach as implemented works only in the z direction. If needed, this approach could easily be adapted for any arbitrary external net force by changing the lowest value direction of CO to be the external net force direction.

E. Energy Conservation

In my simulation, objects are perfectly rigid and there is only surface friction. However, in real life, there are many dissipative forces causing energy losses, which are difficult to simulate. These include air friction, deformation and fracturing. Hence, it is always wise to stabilize the system by some methods that decrease the total energy. Although this section is named 'energy conservation', the main purpose is to ensure that external energy is not pumped into the system.

In my object separation approach, explained in detail in the Separation (Chapter III.C.1), objects to be separated are moved away from each other. The most obvious energy pumping occurs when objects are separated from the ground plane. Objects moved up to be separated from the ground plane gain external elevation (potential energy) that will be converted to kinetic energy by the gravitational force in following time steps. In my implementation, the system calculates height displacement and uses this displacement to calculate external pumped energy. Then, the equivalent amount of kinetic energy is removed from the vertical component of the object's velocity.

1. Stacking

Figure 11 illustrates the stacked configurations that my simulation is able to stably generate. The stacking is a difficult problem, especially with an impulse-based approach. Since the impulses can be transferred from one object to another one at a time, the simulation becomes very unstable when there are many objects on the top of one another. The methods explained in the Freezing and Energy Conservation sections help stacking but are not sufficient to handle all stacking cases. Even with these methods, stacking is not perfectly stable.

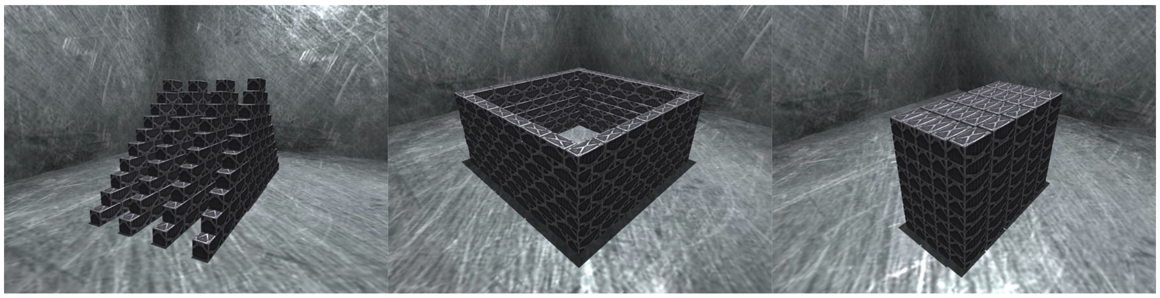


Fig. 11. Stacking Examples. These are some stacking examples that my simulation is able to generate stable configurations.

CHAPTER IV

RENDERING METHODOLOGY

Rendering plays an essential supporting role in the simulation by enhancing the viewer's perception of the final result. The rendering part of this thesis supports per-pixel shading, shadowing, bloom effect, depth of field, and motion blur. Most of these features are natural optical effects, and their accurate computation is expensive.

In this section, I will explain some techniques for simplifying these complex methods to preserve real-time performance. Since rendering plays only a supporting role in this framework, efficiency of the rendering and post-filtering techniques has the highest priority. Most of these rendering effects can be implemented more accurately in an offline simulation and rendering framework. However, here I present highly simplified techniques to get the most visual improvement out of these rendering effects, while keeping the computational cost minimal. All these rendering features are designed to work on the GPU and they are implemented using the CG language.

Some of the rendering features require multiple rendering passes. These rendering passes are processed in a display buffer and their results are stored in textures on the graphics hardware. Some passes are directly used in the final composition and some of them are only used for intermediate states. The final image, to be displayed on the screen, is a composition of these rendering passes.

A. Fragment Lighting

In Figure 12, the left image is rendered using basic OpenGL lighting and shading, and the right image is rendered using fragment lighting. The most obvious difference between these two images is the specular term of shading. The specular term is avoided in the OpenGL version. Although OpenGL is capable of calculating the

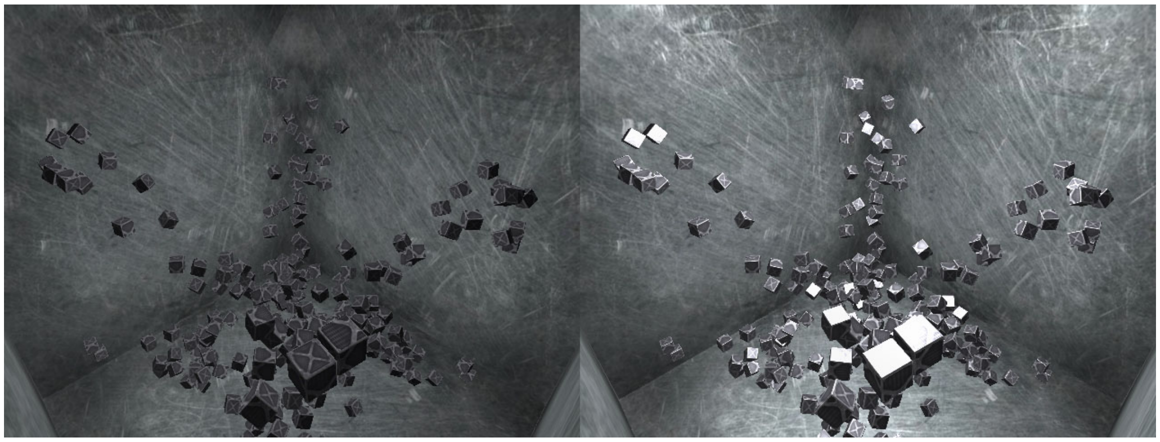


Fig. 12. Advanced Shading. Basic OpenGL (left) and fragment shading (right).

specular term, the result is not pleasing due to the fact that OpenGL handles lighting and shading computations only on the vertices. Since the specular component is a very rapidly changing term over the surface, the basic vertex point interpolation is not capable of catching these high frequency changes. On the other hand, the fragment shader calculates lighting and shading per pixel. Especially for low-poly models, fragment shading significantly improves lighting and shading results. Another improvement for the specular term of shading is the use of 'specular maps'. Parts of an object may have different material properties. Since the diffuse term can be taken care of with the color of the applied texture, the specular term typically requires an additional texture. However, this extra texture is a burden on the graphics hardware, and can be avoided. Since the diffuse texture has an alpha channel that is not used for the diffuse color computation, it can be used to store the specular mask. As illustrated in Figure 13, a specular mask can be used to make the metallic parts of the boxes shinier than the other parts.

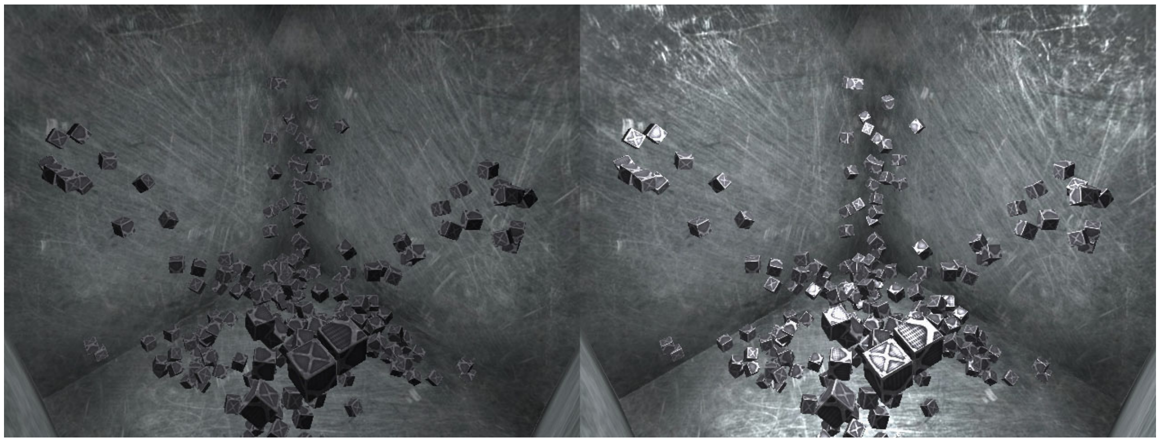


Fig. 13. Specular Maps. Basic OpenGL (left) and fragment shading with specular maps (right).

B. Shadow Mapping

In my implementation, I used the basic shadow mapping technique, giving results as in Figure 14. This method requires additional render passes per light. For performance reasons, I prefer to use only one light source, which requires only one more rendering pass.

The first step for shadow mapping is rendering the scene from the light's view. This rendered image is called the 'depth map'. The values of this image are the distances between the light source and the objects as illustrated in Figure 15. However, GPU implementation has its difficulties. Since a standard texture has discrete integer values between 0-255, the precision is not enough for depth values. I used two different methods to solve this problem. The first method is to encode the floating-point depth value using the three channels of a texture. Even though this improves the results compared to regular integer values, depth values still lack precision. The second method is to implement 'Render to Texture' application using another OpenGL

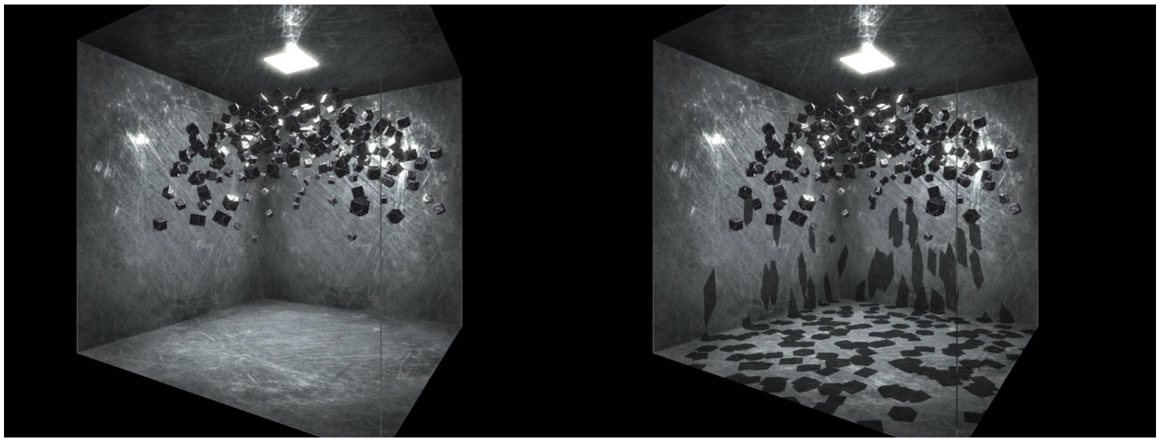


Fig. 14. Shadow Mapping. Without (left) and with shadows (right).

context. With the 'Render to Texture' feature, it is possible to bypass the display buffer and render directly to a texture. Moreover, rendering to a texture gives the ability to use floating-point textures. Even though using a floating-point texture is generally slower, using render to texture performs better since it does not include encoding and decoding of floating-point values. For these reasons, I chose to use the render to texture method for all render passes used in this simulation framework.

The result image after this step is named 'Base Image'. This Base Image will later be used as input data for other effects before it will be used in the final composition.

C. Bloom Effect

The Bloom effect is the only rendering effect that is not designed to enhance perception, but simply to give a more appealing and realistic final image. The true Bloom effect can only be simulated using HDRI (High Dynamic Range Image) with floating point textures to store the render outcome. There are two reasons why I did not chose to use floating-point precision textures in my implementation. The first reason is that

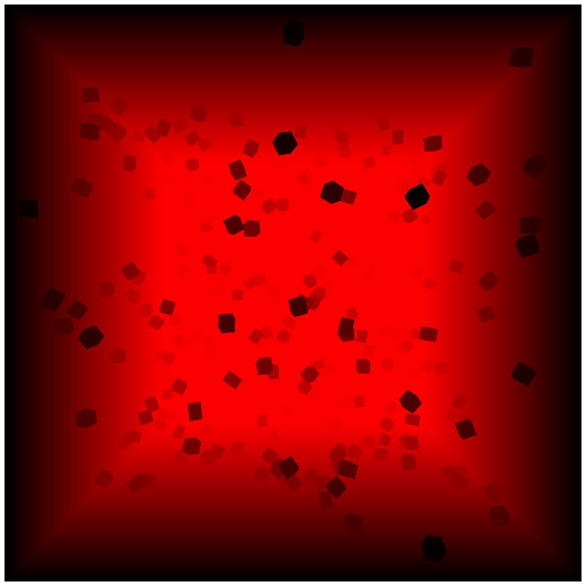


Fig. 15. Depth Map. The values of this texture are the distances between the light source and the objects.

supporting floating-point precision in the graphic pipeline is hardware specific and, even on hardware that supports it, it slows down the application. The second reason is that my simulation environment has a simple lighting setup. Since the only very bright object in the scene is the light source that has absolute white color, it is not necessary to use floating-point precision to detect the bright values generated from this light source. In my framework, I prefer to use standard textures. To catch the bright values, I assume that absolute white in the Base Image indicates that upper values are cut and these values should be enhanced with Bloom effect as illustrated in Figure 16.

The first step is to detect the white values in the Base Image. The texture of these selected values is called the HDR Mask. A simple threshold approach, by selecting only absolute white values, results in flickering of the Bloom effect because of the discontinuity over the HDR Mask. To create a smooth and continuous HDR

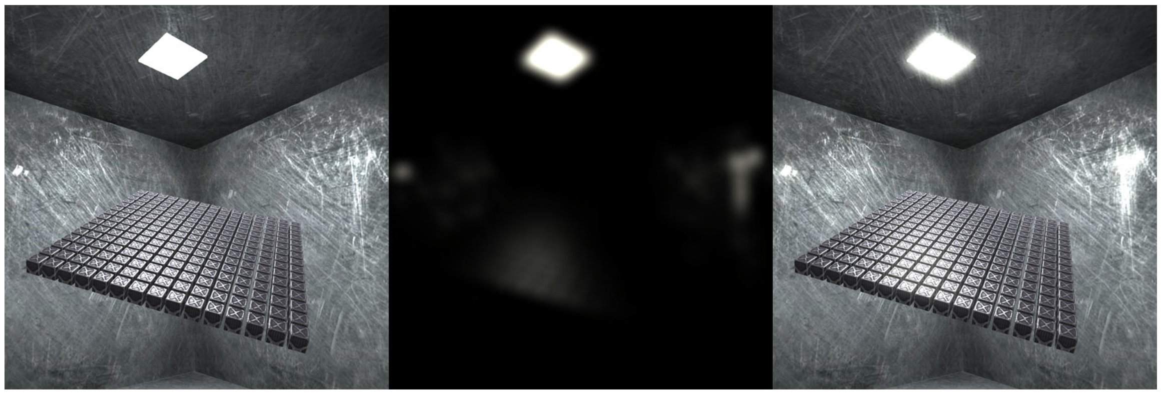


Fig. 16. Bloom Effect. Without bloom effect (left), HDR Mask (middle), and with bloom effect (right).

Mask, I prefer to use 'powers of values' to create the mask instead of the threshold approach. With this method, values lower than 1 rapidly converge to zero and create a smooth transition between 0 and 1. This method is similar to that used when calculating a rapidly changing specular term for shading.

The second step is to spread these values over the HDR Mask to simulate the bloom effect. A two-dimensional convolution filter is applied over the HDR Mask to spread these values. For filtering, I prefer to use a linear kernel instead of the Gaussian kernel, which is commonly preferred by offline applications. Since the Gaussian distribution is curved, for a small sized kernel, the transition is significantly sharper than the linear distribution. Moreover, the linear filter requires a smaller sized kernel than the Gaussian to achieve a similar spreading look.

To make this effect visible on the screen, the kernel size should not be smaller than 1/15 of the input size. Let N denotes the width and height of the bloom texture. The computational cost of the bloom effect is:

- Filtering: N^2 texture units

- Convolution filter per texture unit: $(N/15)^2$
- Total processing: $N^2 \cdot (N/15)^2 = N^4/225$

This process will be significantly faster with the following simplifications. Since the spreading application is a blurring operation over the source image, the high-frequency features will automatically smooth out. The first simplification is to use a smaller input image for Bloom effect calculation by scaling down the Base Image by 1/4. Using a smaller version of the source image will further support the goal of achieving a soft bloom effect and it requires significantly smaller computation time. I called the scaled-down version of the Base Image the Blurred Image. It is a smaller and blurred version of the Base image and its alpha channel is used for the HDR Mask.

The second simplification is to use a separable convolution filter. Separable convolution stands for the special case when a convolution kernel can be separated into two independent elements for each axis. Since the linear kernel is a separable convolution filter, the simulation applies the filter (kernel) sequentially two times in the x and y direction separately. This step requires one more filtering pass, however the time complexity improves from $O(n^2)$ to $O(n)$. The whole computation of the Bloom effect with this simplified version is:

- Scaling down the Base Image: N^2 texture units
- Filtering: $(N/4)^2$ texture units
- Convolution filter per texture unit: $(N/4)/15$
- Total processing: $N^2 + 2 \cdot ((N/4)^2 \cdot (N/60)) = N^2 + (N^3/480)$

For example, if the Base Image is chosen to be 1024x1024, these simplifications speed up the computations more than 1,500 times. Using a smaller Base Image sacrifices some quality; however, the speed-up is very significant.

D. Depth of Field

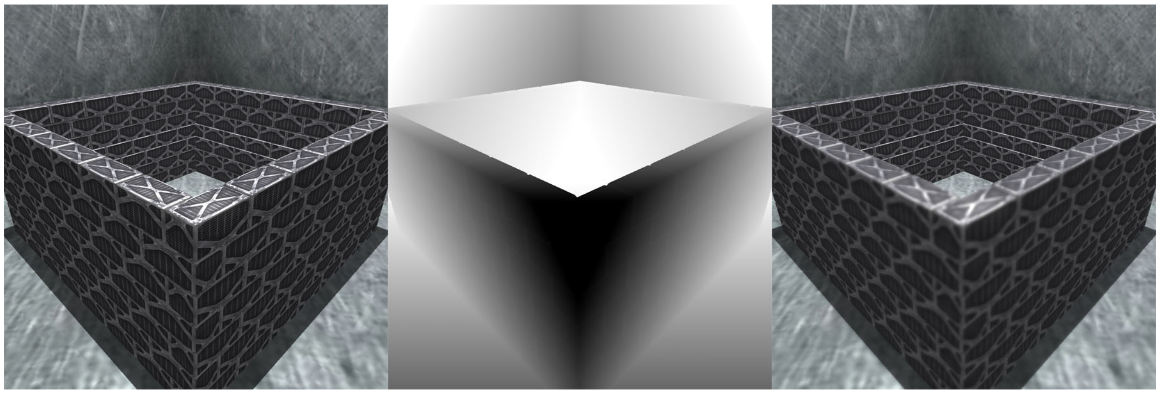


Fig. 17. Depth of Field. Base Image (left), Depth Mask (middle) and final Image with Depth of Field (right).

Depth of field (DoF) is another feature that enhances the 3D perception. In my implementation, I prefer to use a post-filtering technique because the resulting quality of this method is acceptable and the computation time is minimal.

The first step of the DoF implementation is to gather the depth values. These values are calculated in advance of the shading step using camera space z positions. By definition, both the background and the foreground are out of focus in a DoF effect. The next step is to set the 'plane of focus' that lies in between foreground and background. The plane of focus is an imaginary plane that lies perpendicular to the camera view and it represents the region that is in focus. After setting the plane of focus, the other regions are interpolated to have a smooth transition between in and out of focus. The finalized mask is then stored in the alpha channel of the Base Image. This one channel mask is called the 'Depth Mask'. For the second step of the implementation, a blurred version of the Base Image is needed.

In the previous Bloom effect (Chapter IV.C), a filtering operation (sequential over x and y) is applied over the HDR Mask to spread its values. The same technique (with a smaller sized kernel) is applied to the Blurred Image. After this operation, the Blurred Image becomes not only smaller, but also a uniformly blurred version of the Base Image. Since the HDR Mask is in the alpha channel of the Blurred Image, both filtering operations are applied at the same time. In the final composition step, the Base Image and Blurred Image are blended using the Depth Mask as illustrated in Figure 17.

E. Motion Blur

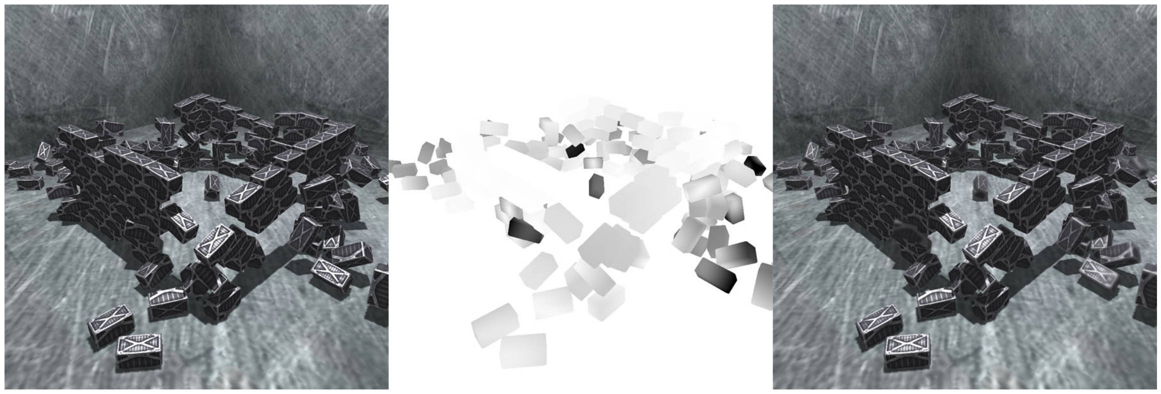


Fig. 18. Motion Blur. Base Image (left), Motion Mask (middle) and final Image with Motion Blur (right).

The motion blur is an effect that helps the user to distinguish moving objects from resting objects. Since fast moving objects should look blurred compared to slower moving objects, this effect can be achieved with a similar post-filtering method as described in the Depth of Field (Chapter IV.D).

The first thing needed is the velocities of the moving objects. Since the objects have angular velocities, the relative velocity is not constant over an object. In the

Advanced Shading step, the angular and linear velocities are explicitly sent to the vertex shader and these vertex-based values are then interpolated in the fragment shader to generate a mask, as illustrated in Figure 18. This mask is called the 'Motion Mask'. After the Motion Mask is generated, it is stored in the Base Image's alpha channel.

Since the Base Image has only one alpha channel, the Motion Mask is in conflict with the Depth Mask. In my implementation, I prefer to simply add these two masks. Since the DoF and Motion Blur effect are using the same post-filtered image to blur, it may become confusing to the viewer to distinguish which effect causes the blurred look. Since the depth of field effect is emphasizing the depth and the motion blur effect is emphasizing the motion, it is usually easy to perceptually differentiate these two effects.

In a true motion blur effect, the blur should be in the direction of the motion covering the path of the object, however, in my implementation, the blur occurs only on the fast moving object. Although inaccurate, it is computationally efficient and creates an acceptable effect. After the DoF effect is embedded in the framework, adding motion blur using a similar approach, has little extra computation cost.

F. Final Result

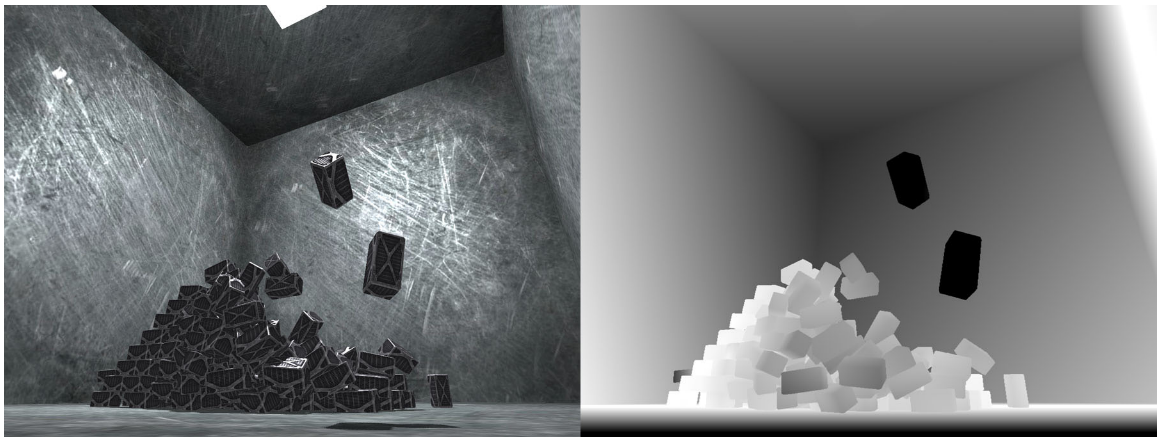
The final rendering result is accomplished after 6 rendering passes in the following order.

- 1st pass: Rendering from light's view to generate the Depth Map.
- 2nd pass: Per pixel rendering to generate the Base Image.
- 3rd pass: Scaling down the Base Image to generate the Blurred Image.

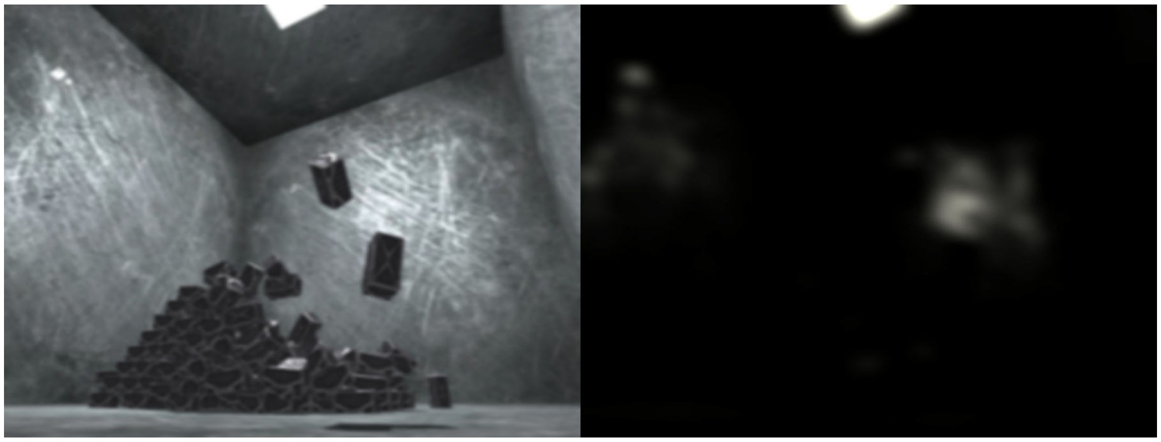
- 4th pass: X pass of filtering of the Blurred Image.
- 5th pass: Y pass of filtering of the Blurred Image.
- 6th pass: Compositing

The final compositing step uses the Base Image and the Blurred Image. The Base Image's RGB components consist of the per-pixel rendering result and its alpha channel has the depth and motion masks. The Blurred Image's RGB components consist of the scaled down and blurred version of the Input Image to be used for Depth of Field/Motion Blur. Its alpha channel has the HDR Mask. This compact representation of the rendering elements allows the framework to run in real-time.

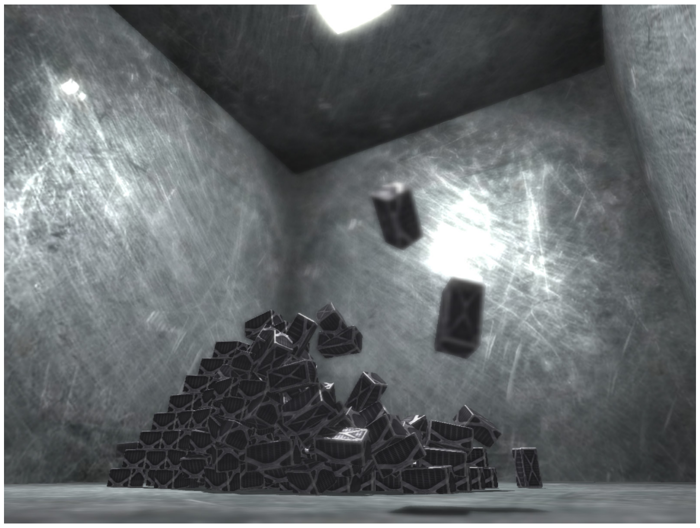
The compositing step uses the Base Image and the Blurred Image to generate the final result as illustrated in Figure 19.



Base Image (left) and the sum of Depth and Motion Masks in its alpha channel (right)



Blurred Image (left) and the HDR mask in its alpha channel (right)



Final Image

Fig. 19. Construction of the Final Composted Image.

CHAPTER V

RESULTS

A number of example scenarios, shown in Figure 20, were developed to test the simulation. These images were captured from the real-time version of the simulation. In Figure 20.a, 20.b and 20.c, the configurations are stable and all objects are in a frozen state except one as illustrated in the initial states of figures. The motion is externally triggered by throwing this separated object into the stacked objects. In Figure 20.d, the configuration is not stable and objects fall due to gravity.

An important feature of my simulation is its capability for handling different sized objects in the same environment without adjusting any parameters. In penalty-based methods, the simulation requires careful tweaking of parameters when there is a change in the size of simulated objects. Moreover, even after tweaking, these methods may still not be able to handle greatly different sized objects in the same environment. Figure 20.d shows that the impulse-based simulation is able to simulate variable sized objects robustly in the same environment. As illustrated in the same figure, the biggest box is more than 15,000 times bigger than the smallest one and the simulation performs without a problem.

A. Comparison of the Results with Another Engine

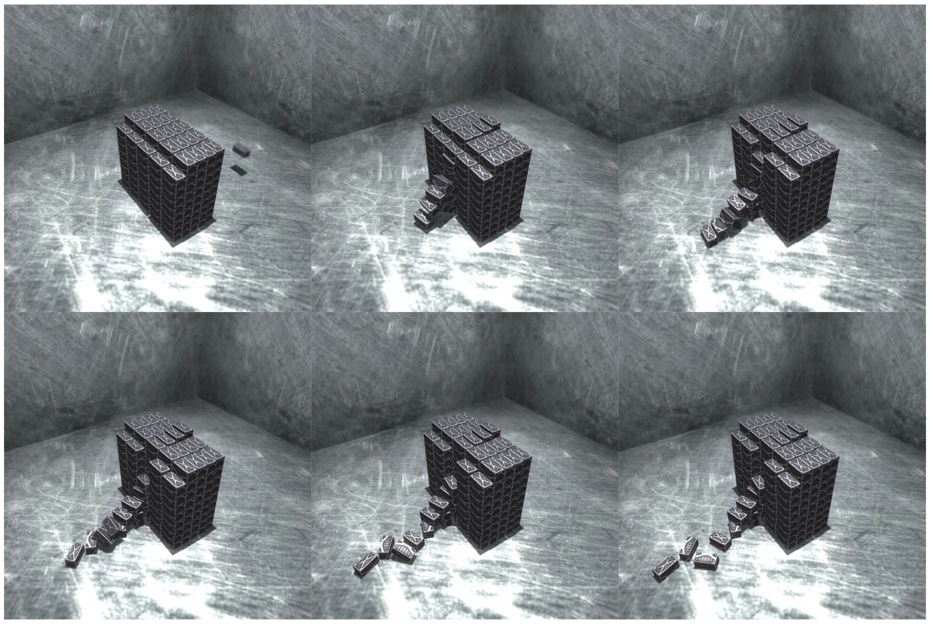
In this section, the results of my simulation are compared with 3ds Studio Max®7.0 simulation plug-in which is a built-in simulation environment developed by Havok®. For comparison purposes, the simulation parameters are set to be the same. I compared the outcomes in terms of performance and accuracy. As stated in the introduction (Chapter I), the goal of my simulation is to achieve believability. Since evaluating the believability of a simulation is subjective, I assumed that the performance and accuracy would reflect the believability of the simulation.

The term 'sub-time step' as used by Havok implies dividing the constant time step into smaller steps, in order to achieve plausible results. In order to match this feature when doing comparisons, I also integrated a sub-time step application into my framework. My simulation's sub-time step algorithm works adaptively by counting the number of collisions. The number of sub-time steps determines the maximum depth of recursion in my framework. Since Havok's simulation is a trade secret, the implementation details of their sub-time step method are unknown. For the test, I designed a scene in which the outcomes are predictable. For most cases, it is not easy to predict the outcomes of any simulation because the interactions are complicated. In the test scene, the objects (boxes) are aligned and, at the same time they lie on a plane that is 45 degrees with the ground plane, as illustrated in Figure 21. Given this alignment, it is easy to predict the general course of a simulation outcome. When simulation starts, objects will fall due to the gravity.

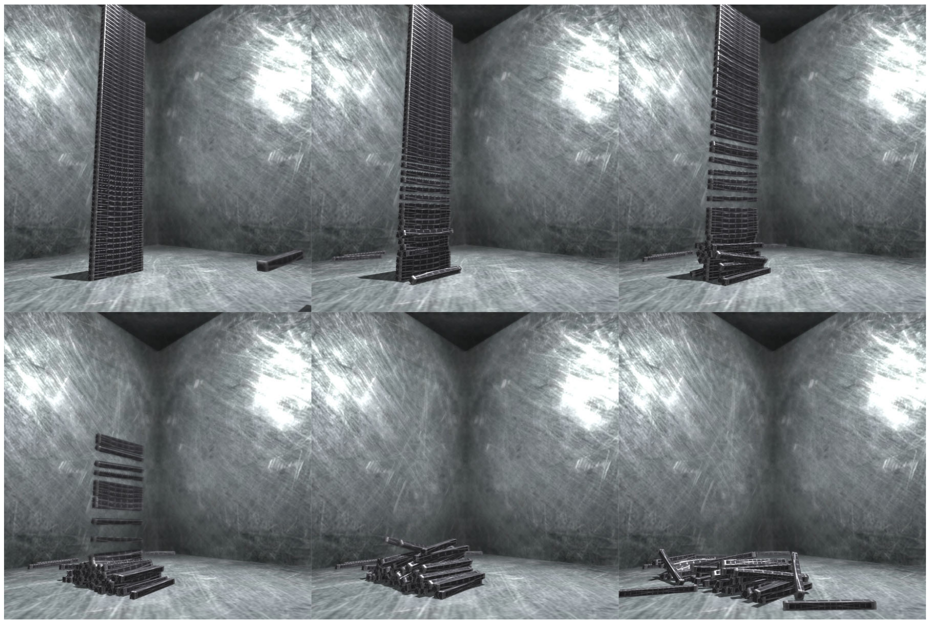
The tests were conducted in two groups by the magnitude of the constant time steps. In order not to crash Havok's simulation, the first group of tests was done using Havok's simulation default time step settings (0.0333 seconds). Even with sufficient sub-time steps, both simulations suffered from accuracy problems with this time step. Hence, the second group of simulations was carried out with a 0.0033 seconds (one tenth of the first test) constant time step. For both test groups, the results represent the final states after 5 seconds (simulation time) of the simulation. For all of the result images, the black background results represent my simulation, and the white background results represent Havok's simulation. The expected result from the test scene is a symmetric distribution. Since all the objects are perfectly aligned and lie on a 45 degrees plane at their initial state, the final distribution of objects should be aligned and symmetric on the ground plane. Moreover, the whole displacement should lie on the vertical direction when seen in top view, due to the object's initial configuration. Any horizontal displacement is a calculation error. From this point of view, none of the results presented in Figure 22 is perfect. However, Havok's simulation has different issues other than symmetry. As illustrated in Figure 22.a, Havok's simulation failed to perform for 1, 4 and 10 sub-time steps. Many objects totally penetrated into the ground plane and fell through the floor to infinity. The objects that appear smaller in Havok's simulation results indicate these falling objects. Since Havok's simulation is a trade secret, the underlying framework is unknown.

Table I summarizes the performance comparisons between the Havok simulation and my simulation. It shows clearly that my simulation is significantly faster than Havok's simulation for all cases in which a comparison was possible. For 400 objects, my simulation's slowest result (0.0033 seconds constant time steps with 200 sub-time steps) takes about the same amount of time as Havok's fastest acceptable result with only 4 sub-time steps. In my test scenario, distribution and symmetry are the main criteria used for accuracy comparisons. For this purpose, both simulations' best results (0.0033 constant time steps with 200 sub-time steps) were compared. Although this accuracy comparison is by nature subjective, it is apparent that my simulation gives no less accurate results than Havok's while being more than 20 times faster.

Figure 23 demonstrates another very important and surprising fact that the only result satisfying accuracy expectations is generated with the real-time version of my simulation. It is surprising because the real-time version calculated the same scene more accurately than the slowest 200 sub-time step versions. Figure 24 illustrates the test results with 1,600, 3,600 and 6,400 cubes. Havok's simulation could not perform the tests for 3,600 and 6,400 objects, due to memory limitations. The complexity of the simulation is $O(n)$, except for the sorting algorithm for the overlapping tests which has $O(n. \log n)$ complexity. The scalability graph shown in Figure 25 supports this fact, indicating that the curve is close to linear.

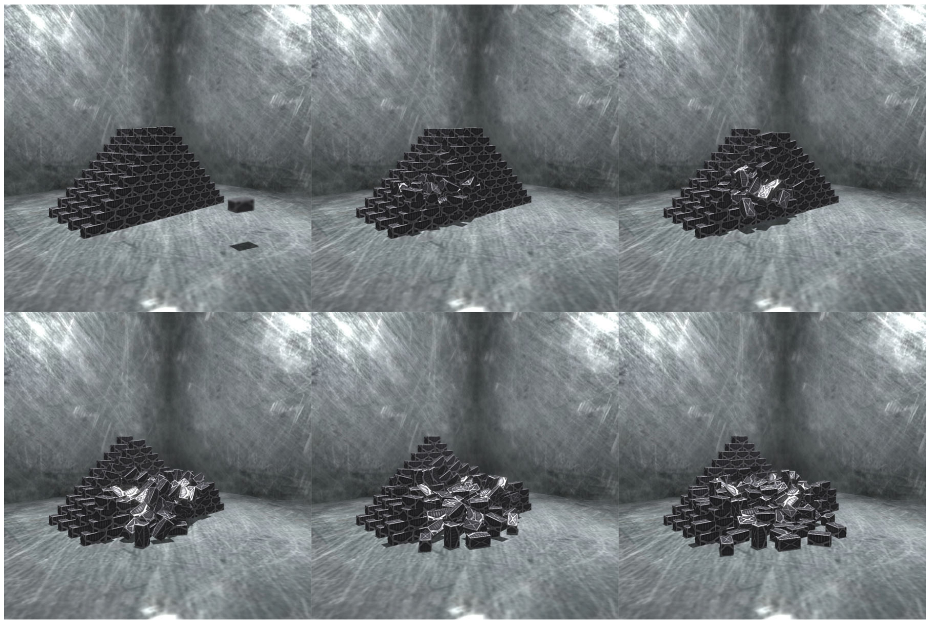


(a) - Simulation of 221 brick objects rendered with all rendering effects.



(b) - Simulation of 60 very thin and long objects rendered with all rendering effects.

Fig. 20. Results from Various Test Scenarios.


(c) - Simulation of 221 brick objects rendered with all rendering effects.

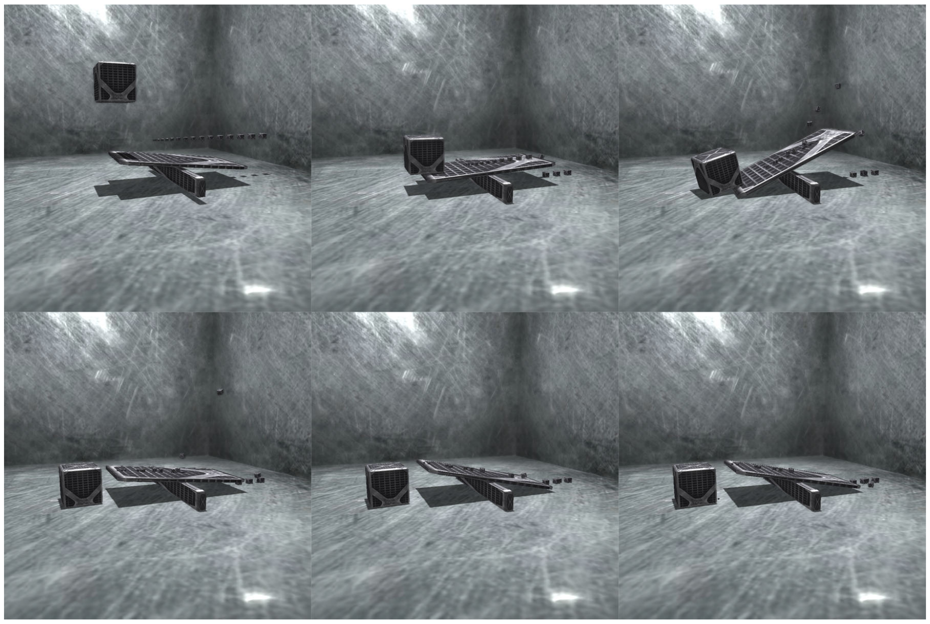

(d) - Simulation of 19 variable sized objects rendered with all rendering effects.

Fig. 20. Continued

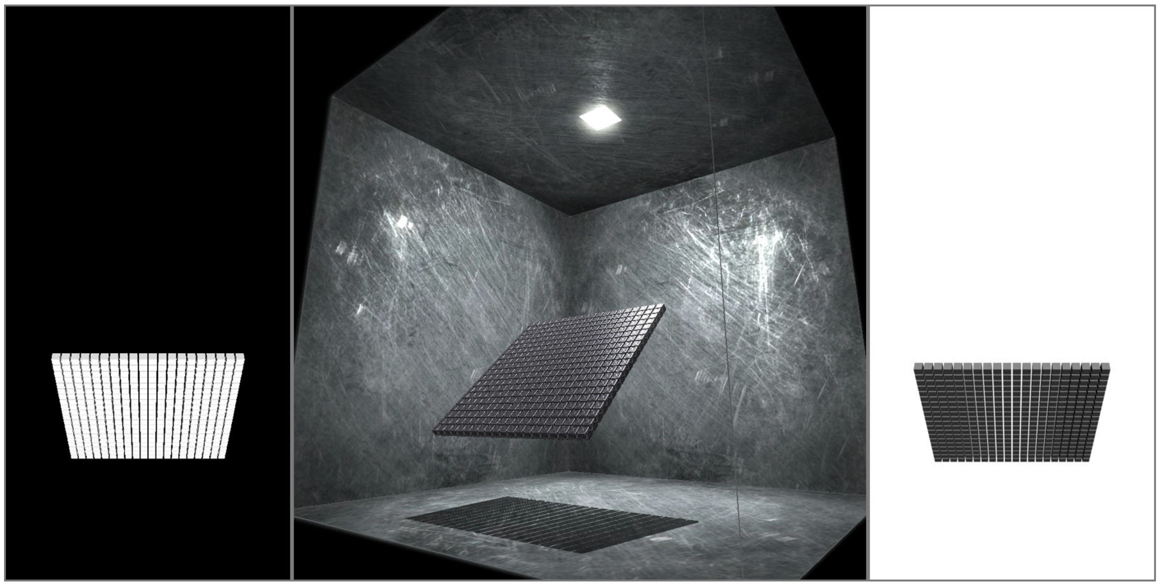
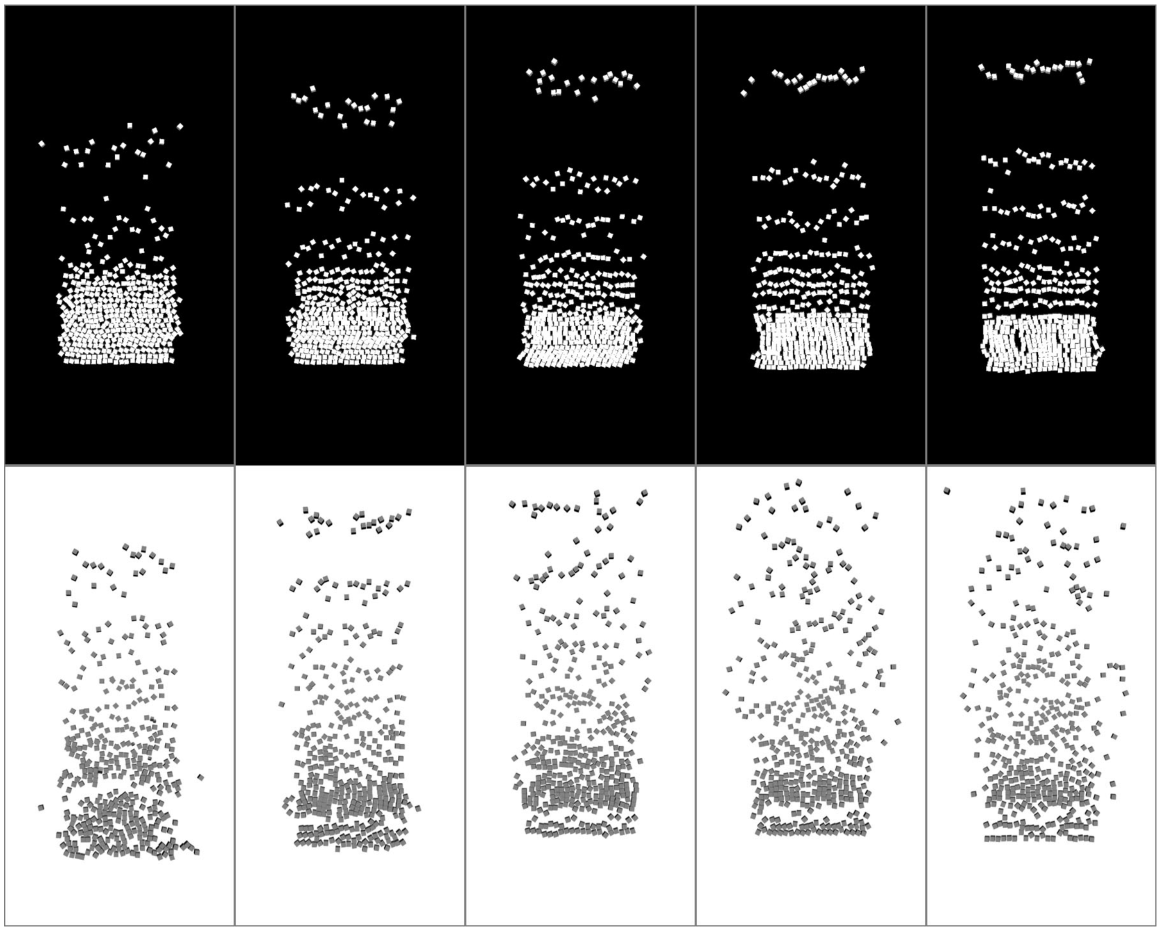


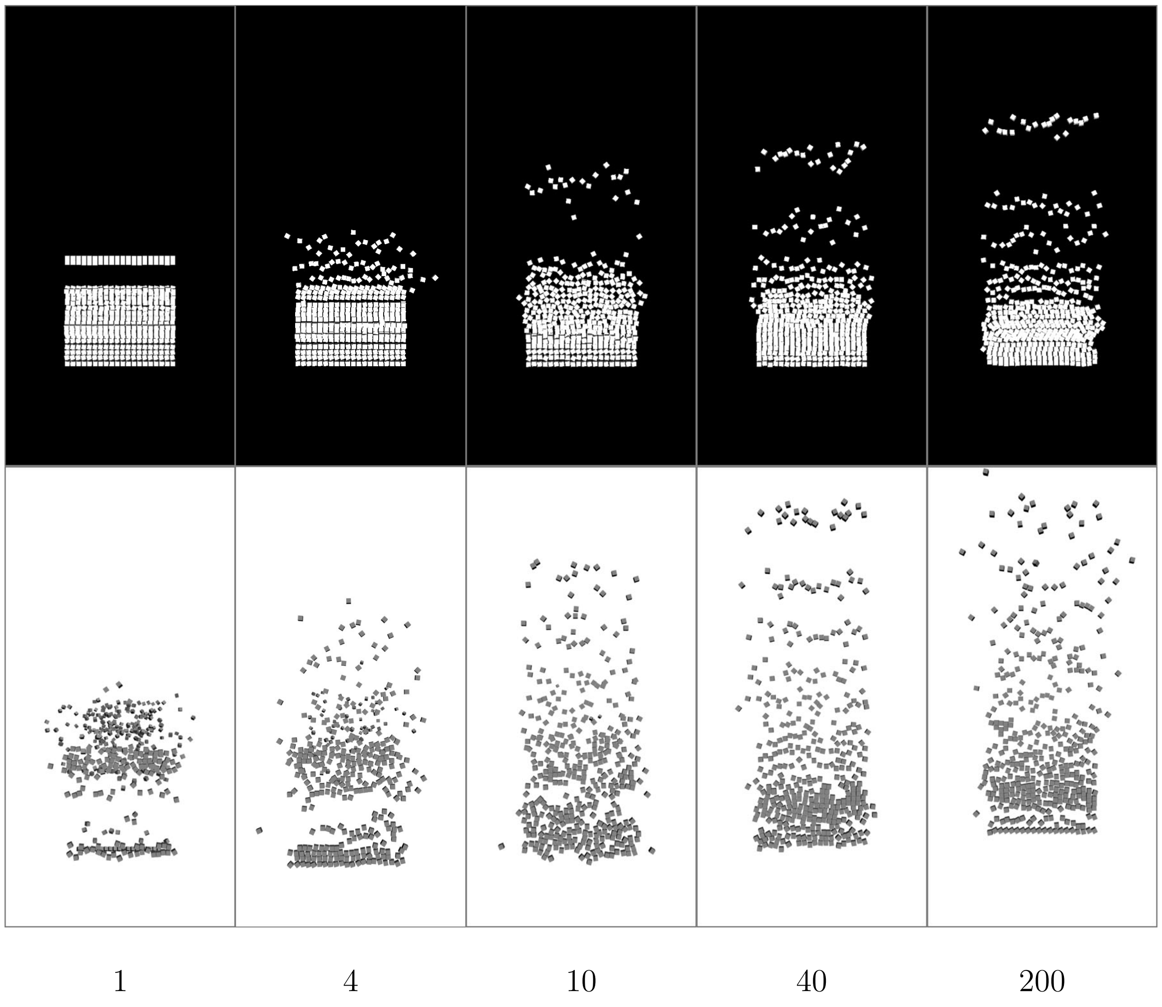
Fig. 21. Comparison Test Environment. The left image is captured from the top view from my simulation and the right image is from Havok's simulation. The middle is captured from the perspective view of my simulation to give an idea about the test environment.



1 4 10 40 200

(a) Constant time is 0.0033 seconds.

Fig. 22. Results of 400 cubes compared with 3ds Max simulation plug-in developed by Havok. The top images are the results of my simulation and the bottom images are results of Havok's simulation. The numbers below the images represent the sub-time step iterations.



(b) Constant time is 0.0333 seconds.

Fig. 22. Continued

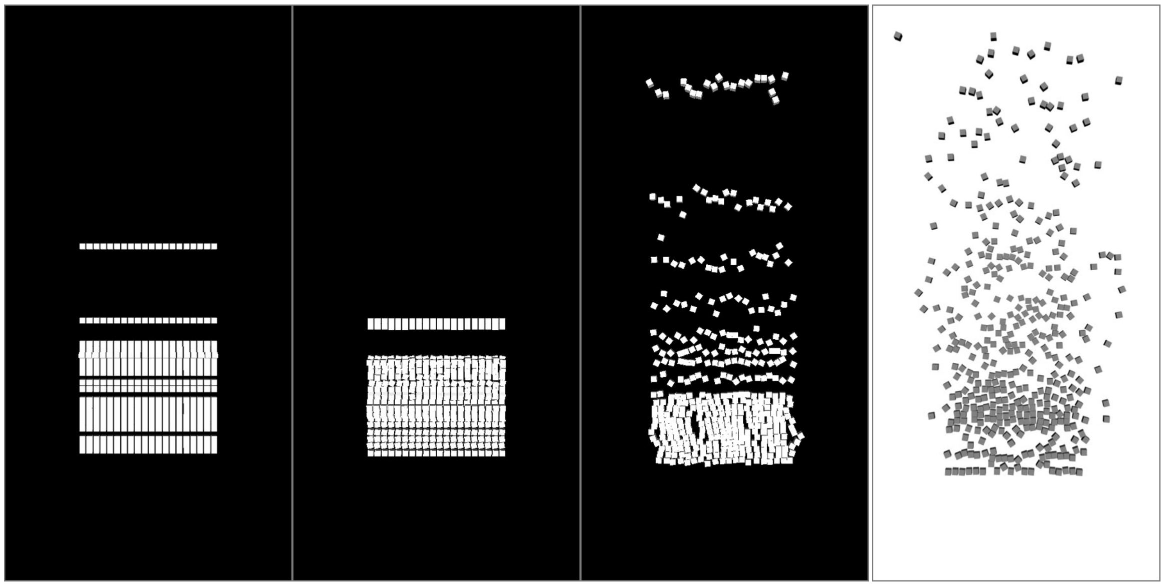
Table I. Comparison of Running Times. Time results indicate the computation time of 5 seconds sequence. 'F' indicates 'failure in calculations'. 'N/A' indicates that 3ds Max couldn't run the test due to the memory limitations. All tests are done with a 1.8 GHz (AMD) CPU and 1 Gb RAM computer.

# of objects		36	100	225	400	1600	3600	6400
Engine	**# of sub-time steps**							
My Simulation	1	0.14s	0.30s	0.73s	1.55s	11.98s	41.35s	120.17s
	4	0.16s	0.41s	0.79	2.30s	12.17s	43.97s	124.85s
	10	0.20s	0.85s	3.62s	5.94s	60.15s	167.91s	2777.54s
	40	0.25s	0.92s	6.76s	20.16s	94.14s	663.94s	1694.44s
	200	0.32s	1.18s	20.16s	33.58s	461.14s	870.12s	3915.67s
Havok's Simulation	1	5.13s	5.38s **F**	7.01s **F**	13.23s **F**	91.13s **F**	**N/A**	**N/A**
	4	4.94s	5.45s	10.78s **F**	36.88s **F**	323.16s **F**	**N/A**	**N/A**
	10	5.14s	5.57s	13.18s	53.14s **F**	796.13s **F**	**N/A**	**N/A**
	40	5.19s	10.47s	24.31s	73.72s	1156.16s	**N/A**	**N/A**
	200	10.53s	33.81s	91.94s	231.24s	2045.12s	**N/A**	**N/A**

(a) Constant time is 0.0333 seconds.

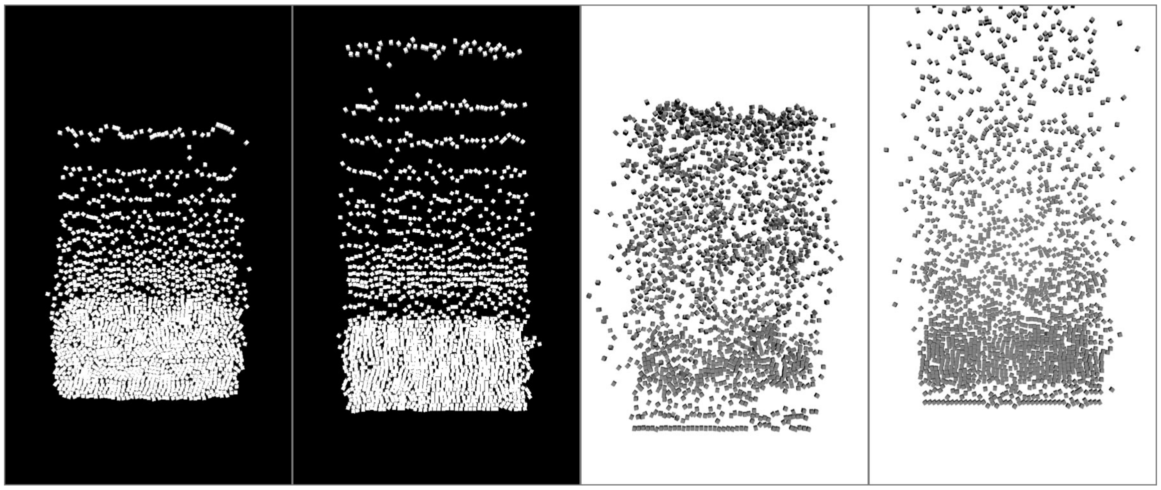
# of objects		36	100	225	400	1600	3600	6400
Engine	**# of sub-time steps**							
My Simulation	1	1.54s	2.12s	8.41s	23.44s	183.53s	608.56s	1763.18s
	4	1.67s	2.24s	10.16s	30.30s	381.71s	1323.53s	4472.28s
	10	1.73s	2.81s	15.12s	45.81s	658.69s	2658.39s	7458.24s
	40	1.86s	3.61s	17.71s	49.21s	1271.32s	5061.96s	20157.72s
	200	1.88s	4.94s	27.58s	86.31s	1922.57s	9951.07s	38309.41s
Havok's Simulation	1	5.32s	6.70s	16.89s	57.68s **F**	829.22s **F**	**N/A**	**N/A**
	4	6.01s	13.16s	30.86s	78.37s	822.75s	**N/A**	**N/A**
	10	8.97s	24.40s	57.64s	131.50s	918.96s	**N/A**	**N/A**
	40	23.80s	70.77s	208.34s	455.96s	2153.03s	**N/A**	**N/A**
	200	69.18s	303.34s	1002.17s	2054.16s	12815.12s	**N/A**	**N/A**

(b) Constant time is 0.0033 seconds.



Realtime - 5.0s 0.033/1 - 1.55s 0.0033/200 - 86.31s 0.0033/200 - 2054.16s

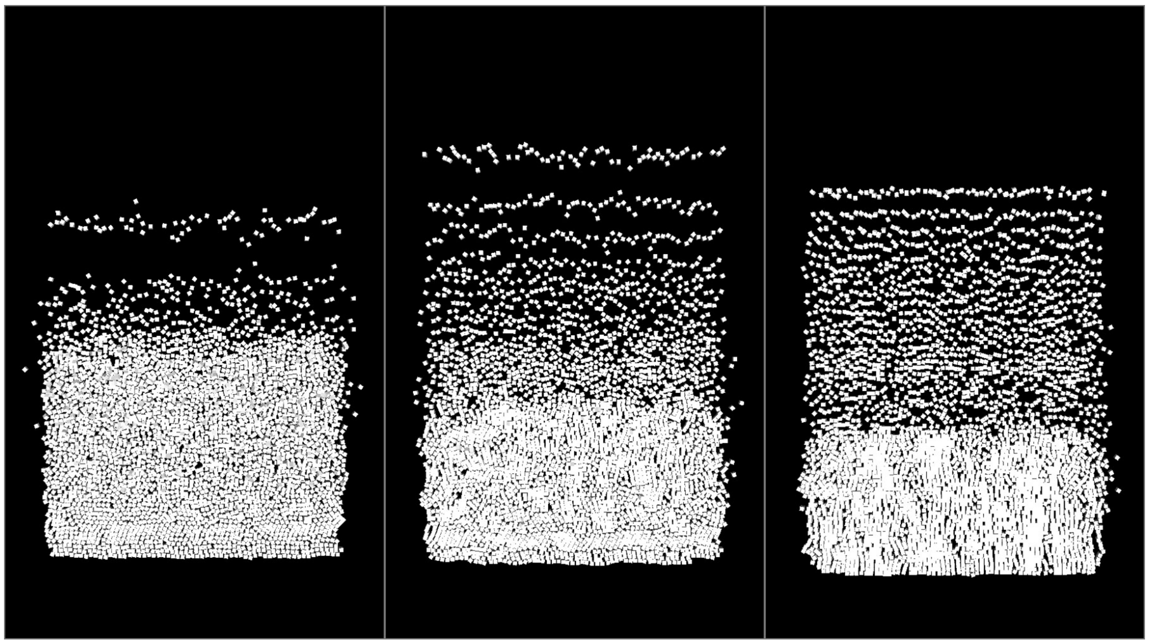
Fig. 23. Real-time Results Comparison. The numbers below the images represent the constant time steps, sub-time step iterations, and the calculation time respectively.



0.0333/200 0.0033/200 0.0033/200 0.0333/200

(a) 1600 cubes. The numbers represent the constant time step, and sub-time step respectively.

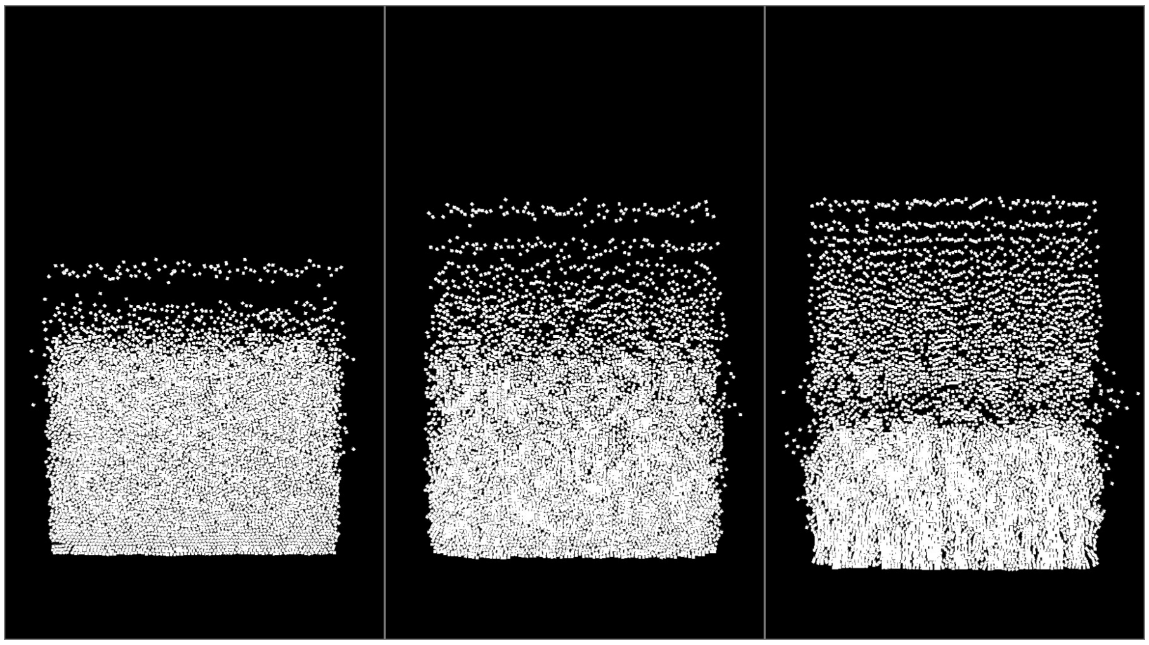
Fig. 24. The Results for Various Numbers of Objects.



1 10 200

(b) 3600 cubes (0.0033 seconds time step). The numbers represent the sub-time step iterations.

Fig. 24. Continued



1 10 200

(c) - 6400 cubes (0.0033 seconds time step). The numbers represent the sub-time step iterations.

Fig. 24. Continued

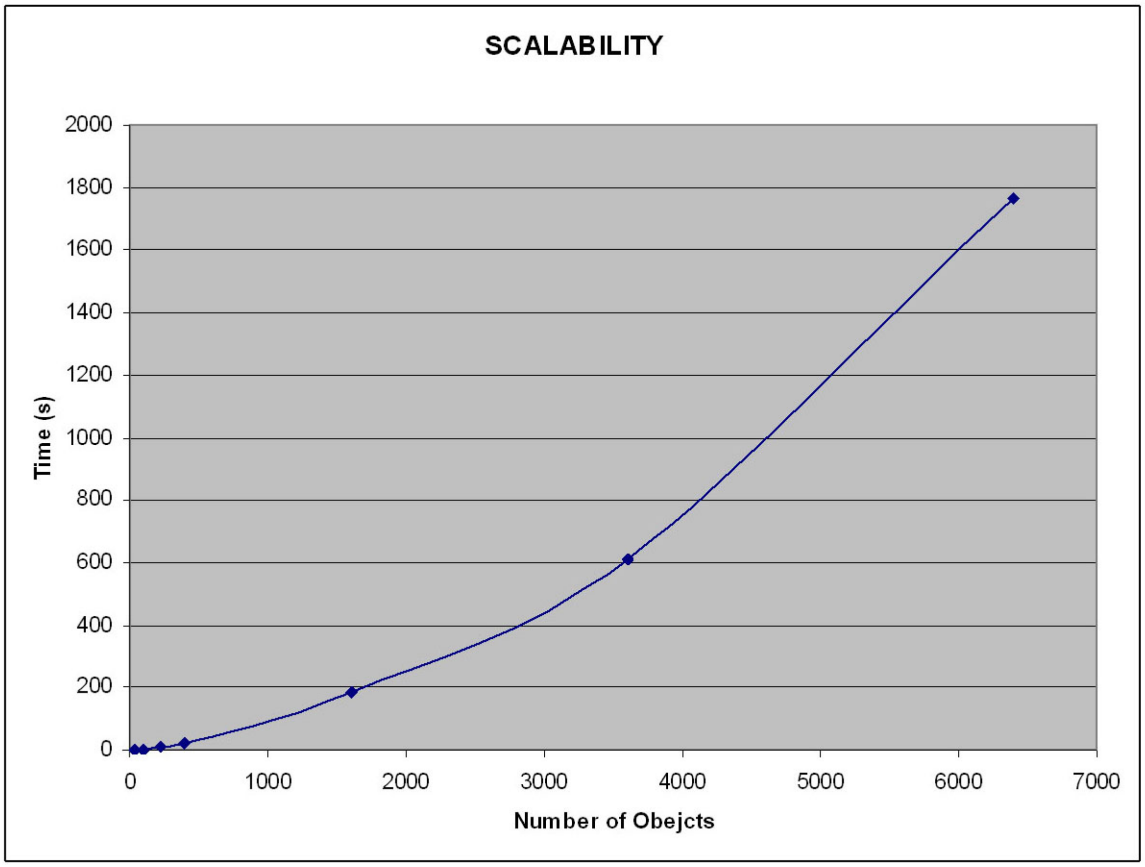


Fig. 25. Scalability of My Simulation.

CHAPTER VI

CONCLUSION

I aimed to design a prototype simulation that works in real-time in a game-like environment with the support of rendering features. To achieve this goal, the features of both simulation and rendering were simplified. Although these simplifications may sometimes cause deviations from exact solutions, they do not generally damage the believability of the simulation. Moreover, by simplifying the calculations of possible simulation bottlenecks, efficiency improves. Since computation time is limited in a real time application, the efficiency improvement may lead to accuracy improvement.

As the comparison section demonstrated, the test results indicate that my simulation outcome is not only more accurate, but also faster than Havok's simulation. Both simulation engines have strong and weak points. Unlike my system, Havok's rigid body simulation is not limited to rectangular objects; it supports a variety of shapes and cases. The more cases a simulation supports, the more performance it sacrifices. Moreover, since Havok's engine is integrated into 3ds Max, performance may not be the highest priority for this offline environment. When the performance comparison is carried out to achieve similarly accurate results, Havok's engine is much slower than my simulation. Another very important fact is that highly optimized implementation and compilation create a big difference for the efficiency and the performance of an application. My simulation environment was implemented to experiment with different ideas and to demonstrate their outcomes. For this reason, when performance is considered, my implementation should not be a match for a highly optimized and developed commercial engine. All these facts aside, the test results still indicate that my simulation is faster and the outcomes are more accurate for the special case of rectangular objects.

My simulation framework is limited to rectangular objects. I used the symmetry property of rectangular objects to approximate multiple collision points as a single collision point. As a future work, this simplification can be generalized to work for any convex shape. The collision resolution step is still a major bottleneck for my framework when many objects are interacting such as in stacking. It might be fruitful to investigate modifying the shock propagation method [12] for use in real-time applications. It might be possible to use this approach to propagate the impact and improve the efficiency for the collision resolution step.

REFERENCES

[1] E. Larsen, “Collision detection for real-time simulation,” Case study, Reseach and Development Department, SCEA, 1999.

[2] R. Oslejsek, “Bounding volume hierarchy analysis,” Case study, Faculty of Informatics, Masaryk University, 2000.

[3] M. Moore and J. Wilhelms, “Collision detection and response for computer animation,” in *SIGGRAPH ’88: Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, 1988, pp. 289–298.

[4] D. Baraff, “Dynamic simulation of non-penetrating rigid bodies,” Ph.D. dissertation, Cornell University, Ithaca, NY, 1992.

[5] S. Gottschalk, M. C. Lin, and D. Manocha, “Obbtree: a hierarchical structure for rapid interference detection,” in *SIGGRAPH ’96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, 1996, pp. 171–180.

[6] J. K. Hahn, “Realistic animation of rigid bodies,” in *SIGGRAPH ’88: Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, 1988, pp. 299–308.

[7] B. Mirtich, “Timewarp rigid body simulation,” in *SIGGRAPH ’00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, 2000, pp. 193–200.

[8] D. Baraff, “Analytical methods for dynamic simulation of non-penetrating rigid bodies,” in *SIGGRAPH ’89: Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, 1989, pp. 223–232.

[9] M. McKenna and D. Zeltzer, “Dynamic simulation of autonomous legged locomotion,” in *SIGGRAPH ’90: Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, 1990, pp. 29–38.

[10] S. Hasegawa, N. Fujii, and M. Sato, “Real-time rigid body simulation based on volumetric penalty method,” in *Proceedings of the 11th Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, Piscataway, N.J, 2003, pp. 326–332.

[11] B. Mirtich, “Impulse-based dynamic simulation of rigid body systems,” Ph.D. dissertation, University of California, Berkeley, Berkeley, CA, 1996.

[12] E. Guendelman, R. Bridson, and R. Fedkiw, “Nonconvex rigid bodies with stacking,” in *SIGGRAPH ’03: ACM SIGGRAPH 2003 Papers*, New York, NY, 2003, pp. 871–878.

[13] H. Schmidl, “Optimization-based animation,” Ph.D. dissertation, University of Miami, Coral Gables, FL, 2002.

[14] K. Egan, “Techniques for real-time rigid body simulation,” M.S. thesis, Brown University, Providence, R.I., May 2003.

[15] L. Williams, “Casting curved shadows on curved surfaces,” in *SIGGRAPH ’78: Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, 1978, pp. 270–274.

[16] R. Fernando and M. J. Kilgard, *The CG Tutorial*, Boston, MA: Addison-Wesley Publishing Company Inc., 2003.

[17] G. James and J. O'Rorke, *GPU Gems*, Boston, MA: Addison-Wesley Publishing Company Inc., 2004.

[18] M. Potmesil and I. Chakravarty, "Synthetic image generation with a lens and aperture camera model," *ACM Trans. Graph.*, vol. 1, no. 2, pp. 85–108, 1982.

[19] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics Principles and Practice*, Boston, MA: Addison-Wesley Publishing Company Inc., 1990.

[20] T. Yu, "Depth of field implementation with OpenGL," *Journal of Computing Sciences in Colleges 20*, vol. 1, pp. 136–146, October 2004.

[21] M. Potmesil and I. Chakravarty, "Modeling motion blur in computer-generated images," in *SIGGRAPH '83: Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, 1983, pp. 389–399.

[22] A. Witkin and D. Baraff, "Physically based modeling: Principles and practice," in *SIGGRAPH '97: ACM SIGGRAPH 1997*, New York, NY, 1997.

VITA

CAN YUKSEL

Texas A&M University

C418 Langford Center

3137 TAMU

College Station, TX 77843-3137

Education:

MS in Visualization Sciences, Texas A&M University, May 2007

BS in Physics, minor in Computer and Mathematics, Bogazici University, May 2004

Experience:

Effects Animator, Dreamworks Animation SKG, 2007

Graduate Assistant, Texas A&M University, 2005-2007