Rigid Body Physics for Synthetic Data Generation

Jens Edhammer
Master of Science Thesis in Electrical Engineering

**Rigid Body Physics for Synthetic Data Generation**

Jens Edhammer

LiTH-ISY-EX--16/4958--SE

**Supervisor:** Jens Ogniewski  
ISY, Linköpings universitet  
Mattias Jansson  
SICK IVP

**Examiner:** Ingemar Ragnemalm  
ISY, Linköpings universitet

Organisatorisk avdelning  
Department of Electrical Engineering  
Linköping University  
SE-581 83 Linköping, Sweden

Copyright © 2016 Jens Edhammer
I would like to thank SICK IVP for giving me the opportunity to work on this project. I thank my supervisors Mattias Jansson and Jens Ogniewski for all their support. And I want to thank Ingemar Ragnemalm for kindling my interest in computer graphics and physics. Lastly I want to thank Conrad Whalén for an excellent implementation of the parallel prefix sum.
Abstract

For synthetic data generation with concave collision objects, two physics simulations techniques are investigated; convex decomposition of mesh models for globally concave collision results, used with the physics simulation library Bullet, and a GPU implemented rigid body solver using spherical decomposition and impulse based physics with a spatial sorting-based collision detection.

Using the GPU solution for rigid body physics suggested in the thesis scenes containing large amounts of bodies results in a rigid body simulation up to 2 times faster than Bullet 2.83.
Contents

1 Introduction 1
  1.1 Purpose .................................................. 1
  1.2 Problem statement ........................................ 2
  1.3 Limitations ............................................... 2
  1.4 Methodology ............................................. 3

2 Theory 5
  2.1 Rigid body dynamics ...................................... 5
    2.1.1 Laws of motion ...................................... 6
    2.1.2 Integrating the laws of motion discretely .......... 7
  2.2 Impulse method .......................................... 7
    2.2.1 Replacing force with impulses ............. 8
    2.2.2 Velocity biasing ................................. 9
    2.2.3 Inertia matrix ................................... 10

3 Related Work 13
  3.1 Spherical decomposition ............................... 13
    3.1.1 Spring-damper model ............................. 14
  3.2 Multiple contacts in sequential solvers ............. 15
  3.3 Mass splitting ......................................... 17
  3.4 Collision detection .................................... 18
    3.4.1 Spatial partitioning .............................. 18
    3.4.2 Parallel Sweep-And-Prune ...................... 18
  3.5 Time-stepping schemes ................................ 18
  3.6 Nonconvex collision with stacking ..................... 18
  3.7 Hierarchical Approximate Convex Decomposition ...... 19

4 Hierarchical Approximate Convex Decomposition 21
  4.1 Implementation .......................................... 21
  4.2 Performance ............................................ 24
  4.3 Remarks .................................................. 25
  4.4 Limitations ............................................... 25
## 5 Parallel Impulse Solver

5.1 Collision matrix ................................................. 27
5.2 Collision detection .............................................. 28
  5.2.1 Grid construction ........................................... 28
  5.2.2 Particles outside the collision grid ....................... 30
5.3 Stabilization iterations ........................................ 31
5.4 Modified normals .............................................. 32
5.5 Tighter voxelization .......................................... 32
5.6 Shader structure .............................................. 33
  5.6.1 Collision detection .......................................... 34
  5.6.2 Impulse calculation ......................................... 35
  5.6.3 Velocity update ............................................. 35
5.7 Optimizations ................................................ 39
  5.7.1 Impulses .................................................... 39
  5.7.2 Velocity update ............................................. 39

## 6 Results

6.1 Performance .................................................. 41
  6.1.1 Increasing number of particles per body ............... 41
  6.1.2 Increasing number of static particles ................. 42
  6.1.3 Increasing number of bodies and dynamic particles ... 43
  6.1.4 Increasing grid size ....................................... 43
  6.1.5 Increasing cell size ....................................... 44
  6.1.6 Modified Normals .......................................... 45
  6.1.7 Tighter voxelization ....................................... 45
  6.1.8 Voxelized teapot ......................................... 46
6.2 Limitations .................................................... 49
  6.2.1 Per body particle limit .................................... 49
  6.2.2 Tunneling .................................................. 49
  6.2.3 Limited domain ............................................ 49
  6.2.4 False alignment ........................................... 50
6.3 Comparison to bullet ......................................... 50
  6.3.1 Concavity .................................................. 50
  6.3.2 Performance ............................................... 50
  6.3.3 Correctness ............................................... 50

## 7 Conclusion and future work

7.1 Conclusion .................................................... 51
  7.1.1 HACD ..................................................... 51
  7.1.2 Spherical decomposition ................................ 51
7.2 Future work .................................................. 52

## A Test data

Bibliography
1

Introduction

Synthetic data generation is a subject of increasingly high relevance. The data can be used for training and validation of self-learning systems, something which is becoming increasingly more popular and useful for vision based algorithms. Solving methods for high speed generation of these datasets can greatly increase the performance and development of these types of algorithms.

1.1 Purpose

For the purpose of generating synthetic data, this thesis will investigate the use of physics simulation for part distribution. The situation investigated in this thesis is when several identical models are ‘poured’ into a bin and simulated to find their final resting distribution. The system in question uses range cameras to detect orientation and position of objects randomly distributed in a bin and guides a robotic arm to grip the object and remove it from the bin. In figure 1.1, the situation the thesis aims to simulate is visualized. Only the distribution of the parts will be simulated, i.e. the robotic arm and gripper used for picking the parts will not be simulated. For these simulations, performance is of great interest as generation of large amounts of data enables the potential of using machine learning for parameter training of these systems. Due to the interest in performance a GPU method will be investigated and compared to a more traditional CPU based solution.
1.2 Problem statement

Graphics processors have access to many cores and can solve problems effectively when the problem can be parallelized. The thesis investigates the use of a graphics processor for rigid body simulations. The suggested implementation will be compared to an already established rigid body physics solver.

1.3 Limitations

Since the system is chaotic in nature, deviations from a high accuracy simulation in exchange for performance is accepted. The simulation should be reasonably accurate and the final distribution must be valid. For instance, a detailed friction model might not grant a better final distribution than a simple one. Additional limitations are hardware. The methods will be evaluated on the hardware provided by SICK IVP. The graphics card used is a GeForce GTX 960 with 4 GB GDDR5 VRAM. The processor used is an Intel Xeon W3530 at 2.8 GHz. The system has a total of 6 GB RAM.
Initially, the rigid-body GPU pipeline of Bullet 3.x was to be investigated. However, it has become apparent that the implementation was more or less abandoned in 2013 when Erwin Coumans started working for Google instead of AMD. While operational, no API for it exists. Due to its state it is excluded from the thesis. A self-implemented GPU rigid body solver will be implemented and compared to Bullet 2.83.

1.4 Methodology

Validation for physics simulations is a very difficult topic. Since the aim for the thesis is to produce realistic enough distributions we do not need to focus on absolutely realistic results, and can instead evaluate the results in terms of a few key properties. Of interest are: performance, for the potential use of machine learning; correctness, as the simulations need to be plausible enough to get close to real life scenarios; concave collision, since the objects might be concave and contain holes or other cavities.

The methods will be evaluated in a comparative fashion against one another.

A CPU method, HACD will be implemented using BulletPhysics 2.83, an open-source, permissively licensed, optimized game physics library. The second method will be a self-implemented GPU based method using voxelized particles with impulses.
This chapter describes the fundamentals of the equations needed to perform physics simulations.

2.1 Rigid body dynamics

According to Baber [2006] there are three major methods for solving rigid body dynamics; penalty methods, impulse methods and constraint based methods. In common with most methods for rigid body dynamics is the need for the governing laws of motion which move the simulation forward in time. In addition methods for collision detection and collision response is needed. The planned use of a GPU collision detection will be limited to a uniform grid/sort-based method, as described by Nguyen [2007]. The methods of using mass splitting and spherical decomposition are very similar to what Macklin et al. [2014] and Coumans [2013a] have presented with regards to constraint based physics. This thesis will describe a method for using it with impulse based physics while keeping the calculations on the GPU. The laws of motion and relevant vectors and variables for updating the states of the bodies are summarized in figure 2.1. Subindices \( a \) and \( b \) refer to which body the variable belong to. The variables used are; velocity, \( v \); rotational velocity, \( \omega \); vector to the collision point, \( r \); mass, \( m \); collision normal, \( n \) and inertia tensor, \( I \).
Figure 2.1: Diagram of the relevant vectors and variables used when resolving a collision. Variables used are; velocity, $v$; rotational velocity, $\omega$; vector to the collision point, $r$; mass, $m$; collision normal, $n$ and inertia tensor, $I$. Subindices $a$ and $b$ refer to which body the variable belong.

### 2.1.1 Laws of motion

The following equations govern the motions of rigid bodies are summarized by Hansson [2007]. They are repeated here for reference. Here $P$ refer to the the linear moment, $L$ refer to the angular moment and $I$ the inertia tensor of the body, expressed in the mass center of the body.

\[
\frac{dP}{dt} = m \frac{dv}{dt} = f \tag{2.1}
\]

\[
\frac{dL}{dt} = I \frac{d\omega}{dt} = \tau \tag{2.2}
\]

\[
\frac{dx}{dt} = v \tag{2.3}
\]

\[
\frac{dq}{dt} = \frac{1}{2} \omega q \tag{2.4}
\]

\[
I = R I_0 R^T \tag{2.5}
\]
2.1.2 Integrating the laws of motion discretely

The equations in the previous subsection need to be calculated in discrete time steps. For this purpose we need a discrete integration technique. The simplest among them is the Euler step. With a fixed time step, $\Delta t$, equations 2.6 to 2.9 can be used to move the simulation one step forward in time. The update for the quaternions are based on the description given by Fossum [2011].

\[
v_{t+\Delta t} = v_t + \frac{p}{m} \tag{2.6}
\]

\[
L_{t+\Delta t} = L_t + \tau \Delta t \tag{2.7}
\]

\[
\omega_{t+\Delta t} = \omega_t + I^{-1} \tau \Delta t \tag{2.8}
\]

\[
x_{t+\Delta t} = x_t + v_t \Delta t \tag{2.9}
\]

2.2 Impulse method

To use the impulse method we need to solve the impulse $j$. Baraff [2001] explains the details and derivations of the results below, and it is recommended that the interested reader look through his work.

Solving physics using the impulse method has the benefit of having few external tuning parameters. In the suggested implementation tuning parameters could be; $\Delta t$, the time step; $\mu$, the friction coefficient, and the bias factor which is explained further in section 5.3.

The following equation can be used to calculate the impulse created in the collision between body A and body B. Please note that $j = j\hat{n}$ holds for all collisions and that the direction of $j$ is always parallel to the collision normal. Below $\epsilon$ refer to the restitution coefficient and $v_{rel,\hat{n}}$ is the relative velocity between body A and B along the normal vector.

\[
j = \frac{-(1 + \epsilon)v_{rel,\hat{n}}}{m_A^{-1} + m_B^{-1} + \hat{n} \cdot (I_A^{-1}(r_A \times \hat{n}) \times r_A + I_B^{-1}(r_B \times \hat{n}) \times r_B)} \tag{2.10}
\]

The two following equations can be used to update the states of the bodies.

\[
\Delta v_A = j m_A^{-1} \tag{2.11}
\]
\[ \Delta \omega_A = j I_A^{-1}(r_A \times \hat{n}) \]  \hspace{1cm} (2.12)

According to Hansson [2007], the same equations with a simple friction model can be written as follow;

\[ \Delta v_A = j m_A^{-1}(\hat{n} - \mu \hat{t}) \]  \hspace{1cm} (2.13)

\[ \Delta \omega_A = j I_A^{-1}(r_A \times (\hat{n} - \mu \hat{t})) \]  \hspace{1cm} (2.14)

With \( j \) redefined as the following:

\[ j = \frac{-(1 + \epsilon)v_{rel,\hat{n}}}{m_A^{-1} + m_B^{-1} + \hat{n} \cdot (I_A^{-1}(r_A \times (\hat{n} - \mu \hat{t})) \times r_A + I_B^{-1}(r_B \times (\hat{n} - \mu \hat{t})) \times r_B)} \]  \hspace{1cm} (2.15)

In the equations, the tangent can be calculated as described below:

\[ \hat{t} = \frac{v_{rel} - \hat{n}(\hat{n} \cdot v_{rel})}{|v_{rel} - \hat{n}(\hat{n} \cdot v_{rel})|} \]  \hspace{1cm} (2.16)

### 2.2.1 Replacing force with impulses

Assuming that we have the impulse created between body \( A \) and \( B \), the following equations can be used to update the state. The impulse is applied along the collision normal. The equations regarding velocity updates are given below, where \( r \) is referring to the vector to the body’s mass center (also refer to figure 2.1).

\[ \tau = r \times f = r \times \frac{j}{\Delta t} \]  \hspace{1cm} (2.17)

\[ L_{t+\Delta t} = L_t + \tau \Delta t = L_t + (r \times f)\Delta t = L_t + (r \times j) \]  \hspace{1cm} (2.18)

\[ \omega_{t+\Delta t} = RI_0^{-1}R^TL_{t+\Delta t} \]  \hspace{1cm} (2.19)

\[ P_{t+\Delta t} = P_t + f \Delta t = P_t + j \]  \hspace{1cm} (2.20)

\[ v_{t+\Delta t} = \frac{P_{t+\Delta t}}{m} \]  \hspace{1cm} (2.21)
2.2 Impulse method

The equations regarding position and orientation are given below.

\[ x_{t+\Delta t} = x_t + v_{t+\Delta t} \tag{2.22} \]

\[ a = \frac{\omega}{|\omega|} \tag{2.23} \]

\[ \theta = |\omega\Delta t| \tag{2.24} \]

\[ dq = [a \sin\left(\frac{\theta}{2}\right), \cos\left(\frac{\theta}{2}\right)] \tag{2.25} \]

\[ q_{t+\Delta t} = dq \ast q_t \tag{2.26} \]

Note that the last multiplication in equation 2.26 is using a special quaternion multiplication, denoted with the asterisk. With the quaternion defined as in equation 2.27, the multiplication is performed as in equation 2.28.

\[ q = [v, s] = [x, y, z, s] \tag{2.27} \]

\[ q_0 \ast q_1 = [s_0v_1 + s_1v_0 + v_0 \times v_1, s_0s_1 - v_0 \bullet v_1] \tag{2.28} \]

The equations concerning quaternion updates closely follow what is described by Fossum [2011].

2.2.2 Velocity biasing

Using the equations as laid out above with a fixed time-step, a situation might arise where two moving spheres interpenetrate each other. This situation has to either be resolved or somehow avoided. To avoid it, one can calculate the first time of impact in the system globally and resolve the collision at that point in time. This thesis will focus on how to resolve the interpenetrations instead, due to time-backing not being particularly suitable for parallel solvers or performance. The method presented here is what is referred to as velocity biasing as described by Catto et al. [2006]. When there is interpenetration we increase the relative velocity between the bodies to ensure they separate properly. This is a penalty method, and solves a non-physical phenomenon through a non-physical solution. Introducing large energies into the system could potentially destabilize the system, therefore only a fraction of the interpenetration is solved each iteration, and the impulse-velocity steps are iterated several times instead to ensure smooth and careful solving of the interpenetration. In addition a slop variable,
\( \delta_{slop} \) is introduced so that some overlap of geometries is allowed before correction starts. This relaxes the system since some interpenetration is allowed and penalty forces are used to resolve the interpenetration. Please refer to figure 2.2.

\[ \delta_{slop} \text{, is introduced so that some overlap of geometries is allowed before correction starts.} \]

\[ \text{This relaxes the system since some interpenetration is allowed and penalty forces are used to resolve the interpenetration. Please refer to figure 2.2.} \]

\[ \text{Figure 2.2: Diagram of the relevant vectors and variables used when determining } \delta_{slop} \text{ and overlap. The figure is gathered from the presentation by Catto et al. [2006].} \]

\[ v_{bias} = \frac{\beta}{\Delta t} \max(0, \delta - \delta_{slop}) \] (2.29)

In equation 2.29, selecting \( \beta \) between 0.1 and 0.3 are reasonable values according to Catto et al. [2006]. With this, equation 2.15 is modified to read the following.

\[ j = \frac{-(1 + \epsilon)v_{rel,\hat{n}} - v_{bias}}{m_A^{-1} + m_B^{-1} + \hat{n} \bullet (I_A^{-1}(r_A \times (\hat{n} - \mu \hat{t})) \times r_A + I_B^{-1}(r_B \times (\hat{n} - \mu \hat{t})) \times r_B)} \] (2.30)

### 2.2.3 Inertia matrix

To be able to simulate the rotations of the objects properly we need the inherent property of the object to resist rotation, the inertia tensor. In 2D this is simply a single value, at least when considering rotation around the center of mass. However, in 3D we instead use a 3x3 matrix called the inertia tensor. According to Ragnemalm [2015], when the objects initial position is aligned with the objects principal axes, we can always write the inertia tensor as a diagonal matrix.
2.2 Impulse method

By voxelizing the object for which we seek the inertia tensor of and treating each voxel as a point mass during the calculation, we can sum over all the point masses and reach an approximation of the inertia tensor. The method is described in ‘So How Can We Make Them Scream’ by Ragnemalm [2015]. \( J \) can be calculated as following.

\[
J = \sum_i \begin{bmatrix}
  m_i (r_{iy}^2 + r_{iz}^2) & -m_i r_{ix} r_{iy} & -m_i r_{ix} r_{iz} \\
  -m_i r_{ix} r_{iy} & m_i (r_{ix}^2 + r_{iz}^2) & -m_i r_{iy} r_{iz} \\
  -m_i r_{ix} r_{iz} & -m_i r_{iy} r_{iz} & m_i (r_{ix}^2 + r_{iy}^2)
\end{bmatrix}
\]  

(2.31)

This method gets quite good results but relies on how finely decomposed the object is. Below is a comparison of theoretical versus approximated inertia tensor of a cube. Only one of the three diagonal components are presented since the object is symmetrical along all three axes, thus the inertia tensor will have the same value along the diagonal. One can note that the inertia reaches a reasonable approximation quite quickly.

![Figure 2.3: Theoretical inertia versus approximated with increasing amounts of voxels.](image)

**Figure 2.3:** Theoretical inertia versus approximated with increasing amounts of voxels.
In this chapter work related to physics simulations and GPU based solutions are presented.

3.1 Spherical decomposition

Nguyen [2007] presents a method for collision detection and response using spherical decomposition. By subdividing the object into a collection of spheres one gets an approximation of the shape. Since each new collision shape is locally convex, collision detection is much simplified, but the global result of the decomposition still has concave properties. Since each new collision shape is the same primitive, only one type of collision detection and response is needed; in this case spheres. For an example of objects decomposed in this manner, see figure 3.1.
3.1.1 Spring-damper model

The spring-damper model is a model for collision response described in Nguyen [2007] and by Fossum [2011]. This model has the benefit of being quite easy to implement on both CPU and GPU. However, since there are two design parameters, the spring constant and the damping constant, the system can be difficult to tune. Often a parameter constellation that works well for one system will fail for another.

The equations below describe how the force and torque can be derived for this type of model. The equations describe the forces for collision between sphere $i$ and $j$, where $k$ is the spring constant, $c$ the damping constant and $d$ the diameter of the sphere. In equation 3.4, $\hat{r}$ refers to the position of the particle (or sphere) relative to the center of the body.

\[
\begin{align*}
 f_{i,spring} &= -k(d - |r_{ij}|) \frac{r_{ij}}{|r_{ij}|} \\
 f_{i,damping} &= -c v_{ij} \\
 v_{ij} &= v_{particle} + v_{tangential} \\
 v_{tangential} &= \omega \times (r - \omega \cdot \frac{\omega}{|\omega|^2})
\end{align*}
\]
Note that the equations above are mass independent.

The parameters must be tuned for the specific system for stability. Often these parameters are determined through testing and the tests need to be redone for new setups.

### 3.2 Multiple contacts in sequential solvers

When solving multiple contacts between a body pair one can rewind time to find the first collision that occurred in the system (a single body pair, or globally), process it first then step forward until the next collision and so forth. However this method can be time consuming and is not trivial to parallelize since the steps in the solver are inherently serial. If we do not revert the system in time, to the initial collision, and instead simply solve all collisions we have currently and take some special care towards interpenetration we would end up with an incorrect simulation. The problem becomes apparent with the following example. In figure 3.2 we see the expected result and in figure 3.3 we see the result if a naive implementation of parallel impulses is used.

---

**Figure 3.2:** The expected result of the collision between the rigid body consisting of the red balls and the rigid body consisting of the green balls. In the figure $M$ is the mass of the jointly rigid body consisting of the two red and green balls respectively. The two bodies collide and energy and momenta is preserved.
Figure 3.3: Calculating all impulses in parallel can lead to too high end velocities. In the figure $M$ is the mass of the jointly rigid body consisting of the two red and green balls respectively. The two bodies collide and energy and momenta is not preserved.

This problem can be remedied by only solving one impulse between a body pair at a time. This can be achieved as stated above by either backing up time until there is only one or a few collision(s), and then solve one of them, chosen arbitrarily. This gives rise to an order dependent system, and is something which makes for difficult realizations in parallel algorithms. Another approach popularized by Erin Catto in his Box2D is Sequential Impulses which solves the worst interpenetration on each respective body pair, updates the velocities and reiterates the solution a few times. One interesting aspect here is that negative impulses are allowed in intermediate results. However, as the name suggests, it is most suitable for a sequential implementation. Both of the methods above suggest that only one collision between a body pair should be solved at a time, which leads to sequential processing. For more details see Catto [2009]. This method could arguably be parallelized in the sense that each body could be processed in parallel. However, a different approach is used for this project since concave collisions is one of the aims for this thesis and extending this method to handle concave collisions is non-trivial.
3.3 Mass splitting

Tonge et al. [2012] describes a parallel constraint based solver using mass splitting, where objects are split and solved independently and superpositioned for the final result.

A similar approach would be to split the mass prior to calculating the impulses by the number of collisions between the two bodies. The calculated impulses are then applied to the original object. The idea is visualized in figure 3.4, where we get the expected result.

![Figure 3.4](image)

*Figure 3.4: By mass splitting the objects by the number of interbody collisions and applying the resulting impulses to the non-split object we get the expected result.*

For this method to be implementable we need to count the number of collisions between every body pair. This is described in further detail in section 5.1.

The impulse, defined in equation 2.30, is divided by the number of collisions between the bodies, here defined as k. The final modification to $j$ before applying it to the bodies then becomes the following:

$$j_{masssplit} = \frac{j}{k}$$

(3.5)

While inspired by the mass splitting technique from Tonge et al. [2012] this is simply an application of the superposition-principle of forces.
3.4 Collision detection

3.4.1 Spatial partitioning

Nguyen [2007] suggests spatial partitioning by ways of a uniform grid for collision detection. This method is especially suitable for the spherical decomposition since we can select an optimized grid size for a fixed sphere radius. In addition Nguyen [2007], Green [2010] and Hoetzlein [2014] claim that sorting the collisions by bin index increases memory coherence and overall performance.

3.4.2 Parallel Sweep-And-Prune

Parallel Sweep-And-Prune as investigated by Coumans [2013b], which in turn is based on Liu et al. [2010], discusses a single axis Sweep-And-Prune algorithm on the GPU for broadphase collision detection using bounding boxes and the Separating-Axis-Theorem. For more details concerning the basics of Sweep-And-Prune, turn to the work of Terdiman [2007].

3.5 Time-stepping schemes

Lembcke [2006] describes using Guendelman’s time stepping algorithm. By detecting when the velocity is very small the contact impulses can be calculated by setting $\epsilon = 0$ in equation 2.30. Guendelman’s time stepping algorithm is provided in pseudocode below, in accordance to what is described in Guendelman et al. [2003].

```
1: DetectCollisions
2: for all collisions with $|v| > 0$ do
3:     calculateImpulses
4: end for
5: applyGravity
6: updateVelocity
7: for all collisions with $|v| \leq \epsilon_v$ do
8:     calculateImpulse (with elasticity = 0)
9: end for
```

3.6 Nonconvex collision with stacking

Guendelman et al. [2003] present in their paper "Nonconvex Rigid Bodies with Stacking" a scenario very similar to what this thesis aim to produce. While the end results reach high physical accuracy the runtime is not suitable for realtime applications. Dropping 500 rings consisting of 320 triangles averaged 3 minutes per frame.
In the same paper Guendelman et al. [2003] present the Shock Propagation algorithm. By constructing a contact graph from the static objects in the scene (see figure 3.5) then solving the objects in contact with the static objects first, followed by changing their mass making them static and solve the next layer and so forth. This lead to much higher stability in simulations, but does however introduce a sequential dependency since each level of the graph needs to be processed before the next.

![Diagram of contact graph traversing from static ground upwards. Each box represents an object and the large rectangular block at the bottom of the image represents a static ground. The distance is measured, as in how many collisions away from the static ground each object is. Each object is assigned a level based on this distance.](image)

**Figure 3.5:** Contact graph traversing from the static ground upwards. Each box represents an object and the large rectangular block at the bottom of the image represents a static ground. The distance is measured, as in how many collisions away from the static ground each object is. Each object is assigned a level based on this distance.

### 3.7 Hierarchical Approximate Convex Decomposition

A popular method for collision detection of complex concave objects are convex decomposition. Reasons for the popularity of the method may be its relative simplicity in theory. Dividing the concave model into convex subsets can give composite shapes that are concave, yet still work with the traditional collision detection and response models since they work on convex sub-shapes. In figure 3.6 an initially concave shape has been subdivided in smaller convex shapes. Creating composite shapes in a physics engine is fairly straightforward as each submodel contributes to the inertia and mass of the object. The previous collision shape of each object can be used, or a new collision shape for the whole object can be recalculated. The forces of each submodel contribute to the movement of the whole model instead of the submodel itself. There are some automatic methods for convex decompositions, among them hierarchical approximate convex decomposition (HACD) developed by Mamou and Ghorble [2009].
Bäcklund and Neijman [2014] investigated the use of HACD and found it produced fewer hulls than previous methods while still approximating the original objects well. Mamou is currently working on a newer version of HACD called V-HACD, which among other things includes GPU acceleration.
Hierarchical approximate convex decomposition (HACD) can be used to decompose concave objects into a collection of convex objects. These objects can be used to simulate the scene instead.

4.1 Implementation

Bullet 2.83 contains a composite (or compound) shape, a shape container which contains sub-shapes and acts as a joint rigid body. Bullet updates the inertia of the object internally as more sub-shapes are added, as well as the bounding box needed for the Sweep-And-Prune (SAP).

An implementation of HACD is included with Bullet’s source code and the code is developed by Mamou, based on the work of Mamou and Ghorble [2009]. Some input parameters are available. These have been left as the recommended parameters. The parameters control for instance concavity penalty weights, (i.e if it is possible to disregard a small concavity) and the maximum amount of points per convex hull. While effective in terms of convex decomposition, this method is not particularly effective in terms of performance. For instance, a Utah Teapot of around 4300 vertices takes approximately 25 seconds to decompose, see figure 4.1 for the result. In addition, the method is rotationally variant, so results may vary depending on the rotation of the object which is to be decomposed.
Figure 4.1: The decomposition from HACD method, model left in initial orientation

The models are added in a three dimensional grid above the bin before the simulation is started. Figure 4.2 shows the start position.
During a simulation, much more time is spent in the last part of the simulation when most of the models are quite stationary and only small changes are made. The increased frametime in the later parts of the simulation is due to the many contacts between the objects. A lot of small, rolling motions, likely due to the round shape of the Utah Teapot, can be observed. Without the small rolling effect, Bullet’s sleeping would take effect earlier and increase performance. One can expect that more rectangular objects will have better performance.

The result of the simulation can be seen in figure 4.3 and 4.4.
4.2 Performance

For testing, Utah teapots consisting of twelve submodels were used. Each submodel has between 50 and 100 vertices. A timestep of 0.01 seconds was used. All timings presented are without rendering. All tests measure simulation time and average the result across 2000 iterations. For the first test, Bullets’ sleeping was disabled, and for the second, the sleeping was left enabled. The results are presented in table 4.1.
Table 4.1: Testing of Bullet with HACD

<table>
<thead>
<tr>
<th>Sleeping Enabled</th>
<th>Number of objects</th>
<th>Time/frame [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>1024</td>
<td>94.665</td>
</tr>
<tr>
<td>Yes</td>
<td>1024</td>
<td>78.317</td>
</tr>
<tr>
<td>No</td>
<td>512</td>
<td>51.9156</td>
</tr>
<tr>
<td>Yes</td>
<td>512</td>
<td>30.5515</td>
</tr>
<tr>
<td>No</td>
<td>256</td>
<td>22.3175</td>
</tr>
<tr>
<td>Yes</td>
<td>256</td>
<td>13.298</td>
</tr>
</tbody>
</table>

4.3 Remarks

While the time necessary for the decomposition renders it implausible for online uses, it is still an excellent tool for a preprocessing step. The models can be decomposed at the start and copies of the same shape can be used, i.e. a single decomposition at the start of the program. For this thesis the results were also saved to an Wavefront Object (.obj) file so that simple and quick loading of the decomposed models could be used while testing, i.e. a single decomposition was performed and the results reused, even if the program was restarted.

The performance of the whole system simulation, while not phenomenal, is fast enough for the generation of synthetic data, assuming that suitable simulation parameters are selected.

The method does handle concave results through the approximate decomposition.

In terms of accuracy the approximate decomposition, while good, does not reproduce the objects with perfect accuracy.

4.4 Limitations

The limitations of this method mainly concern performance. The fact that the parameters of the HACD might affect the decomposition makes the system on the whole less robust.
The parallel impulse solver was implemented using spherical decomposition as described in section 3.1, using the grid based and sorted output collision detection described in section 3.4. Implementation details concerning the collision detection can be found in section 5.2. However, in this chapter we suggest solving rigid body dynamics on the GPU using spherical decomposition instead of using a spring-dampner model as in Nguyen [2007], or constraint based physics as Macklin et al. [2014] suggests, we investigate the use of impulses.

5.1 Collision matrix

As previously mentioned in section 3.3 we need to count the collisions between all collision pairs to acquire correct impulses. One way to do this is to have a matrix $C$ keeping count of all collisions. $C_{i,j}$ would contain the number of collisions between body $i$ and body $j$. This matrix would be zero throughout the diagonal and symmetric. Thus, $C_{i,j}=C_{j,i}$ would be satisfied and therefore more data than required is allocated by letting $C$ be an $M \times M$ matrix where only $\frac{M^2}{2} - M$ elements are required (where $M$ is the number of bodies). This reduction in size has not been implemented, and instead the full $M \times M$ matrix is used.

A reasonable location in the shader stages for adding the collisions is during the collision detection shader, each detected collision increments $C_{i,j}$ through an atomic increment. A drawback of the atomic operations is that they may affect the performance of shaders negatively. In other words, the benefits of using atomic operations has to outweigh the potentially slow atomic operations.
5.2 Collision detection

Initially, a brute force collision detection was tested. With $O(n^2)$, it was too slow and had poor scaling as expected. Instead, spatial partitioning with sorting was implemented. Each cell in the grid has the size $d\times d\times d$ where $d$ is the diameter of the largest sphere present in the system. The grid represents a 3D space, and for simplicity the grid is implemented with a size of $m\times m\times m$.

5.2.1 Grid construction

Initially we must count the number of spheres in each bin in the grid and then create the exclusive prefix sum for the sorting step. The exclusive prefix sum is calculated on the GPU as described in Nguyen [2007]. With the exclusive prefix sum we can reset the bin counters and start the sorting. The pseudocode below describes the process.

```
1: binIndex = hashFunc(pos);
2: gInd = thisGlobalThreadIndex;
3: index = prefixSum[binIndex] + atomicAdd(binCount[gInd],1);
4: outPutSpheres[index] = inputSpheres[gInd];
```

The hash function used for this thesis is a simple rounding down of the $x,y,z$ coordinates, a division by the cell size ($d$), and an offset to center the grid over our specified domain. Domain here refers to the coordinates which the grid covers, i.e. the range of valid positional ids in the grid. For calculating the ids see equation 5.1.

\[
id = \lfloor (p - \text{offset})/d \rfloor \tag{5.1}\]

Accessing the correct bin, given $m$ bins per dimension and $id = [idx, idy, idz]$, becomes:

\[
binIndex = idx + idy \times m + idz \times m^2 \tag{5.2}
\]

The prefix sum ensures we know beforehand how much space (or number of particles) each bin needs. The atomic add allows us to index the correct subspace of the array. The two following figures describe the process of sorting through prefix summation.
5.2 Collision detection

Figure 5.1: By sorting the spheres we can get more coherent reads. Image inspired by Hoetzlein [2014]. At the top of the figure the original particle order is displayed. Each bin has a line connecting it to the particles which lie within that bin, for visualization purposes only. The particles are then reordered in bin-order. Particles that lie in bin 50 are placed before particles that lie in bin 51.

Figure 5.2: The sorting can be performed through an exclusive prefix sum. Image inspired by Hoetzlein [2014]. Particle one when reordering, is placed at the position indicated by the prefix sum of its bin. Particle two is placed at the first available location after particle one, i.e. position one. Particle five is placed at position two as indicated by the prefix sum of the bin which that particle belong to.
The overall algorithm of the collision detection can be summarized as below.

1: gridCounting with atomicAdd
2: exPrefixSum through reduction
3: reset gridCount
4: reorder particles with exPrefixSum and atomicAdd into gridCount to offset.

5.2.2 Particles outside the collision grid

Particles can collide even when outside the grid due to the indexing being clamped to the grid, see figure 5.3.

![Visualized here is a 5x5 grid. The sixteen central cells are dx$d in size while the cells on the edges extend infinitely due to clamping in the indexing.](image)

However, the cells at the edges are technically infinite in size (limited only by the limits of the used data types) and can therefore contain a very large amount of particles and thus the collision detection step may suffer greatly in performance. Consider the following two situations: About half of the particles are initially...
outside the simulation domain; The grid’s cell size is doubled thus moving most
of the particles previously outside into the domain, but each collision detection
will have to search through more particles. The latter was faster during testing.

This change results in many cells containing more particles to iterate over. This
was still preferable to letting a few cells execute slower due to containing many
more particles. It is not surprising since this is a parallel system. In the best
case the execution time of the whole system would be the slowest path in total
through the system. However, this is not always the case since not all particles
can be executed in parallel.

5.3 Stabilization iterations

The velocity and impulse step are iterated several times to stabilize and solve
interpenetrations by a percentage of the distance as previously described in sec-
tion.

The number of iterations needs to be high enough for the current situation. Fig-
ure 5.4 is a situation presented where the number of iterations (2) is too low.
When too few iterations are used tunneling can occur and in some extreme cases
the objects at the bottom have begun penetrating through the bin.

**Figure 5.4:** Red sphere is a static object. From top to bottom iterations in-
crease. At the last row the positional update is visualized where interpene-
tration happens due to too few iterations.
Unfortunately with higher stacking and increased number of objects in tight proximity, the number of iterations needs to increase or a smaller time step is needed.

### 5.4 Modified normals

Since the spheres can be seen as a type of collision proxy shape for the true object, it is reasonable to investigate what would happen if instead of the sphere’s normals the original object’s normals were used for collision resolution. For the cube only spheres that lie on the faces have their normals modified, the corners and edges are left with their original normals. The result of the modified normals is presented in section 6.1.6. The spheres are assumed to be in contact however the figure leaves some space between them to make the figure more readable.

![Figure 5.5: Red spheres have their normals modified and the normals are rendered as lines.](image)

### 5.5 Tighter voxelization

By tightening the voxelization, while still using the same radius on the particles, i.e. let the particles overlap within the body, we effectively change the surface normals. Overall this will lead to a smoother surface at the cost of increasing the number of particles needed. A smoother surface might reduce the number of stabilization iterations needed. Tighter voxelization is related to modified nor-
mals as the new representation now more accurately models the surface and its normals. In figure 5.5 the spheres have been tightened and overlap. The result of the tightened voxelization is presented in section 6.1.7.

5.6 Shader structure

For this implementation OpenGL 4.5 with Compute Shaders are used to program the GPU. In Compute Shaders there are several tools for workgroup synchronization but to synchronize globally across different workgroups one need to split the program into different shaders and synchronize from the CPU. This is done by ways of barrier bits. Global synchronization is performed between every shader, i.e. between each box in the flow chart, in the figure 5.6. Green boxes in the figure represent shader stages which are dispatched with a thread per sphere, whereas the blue dispatch across bodies instead.
5.6.1 Collision detection

The collision detection is performed through sorting and spatial partitioning. The main problems with this approach is the limited domain which can be simulated. The bins size should, for optimal search conditions, be exactly the diameter of the largest particle in the system, albeit larger bins can be used with the penalty that more particles have to be checked. However, this results in a larger valid simulation domain.

Figure 5.6: Flow chart of the different shaders involved. Green shaders are dispatched over spheres and blue over bodies.
When the collision grid and reordering is done one can find the actual collisions among the candidates. All candidates reside in the 26 surrounding neighboring grid cells and the particles' own cell. The distances are measured between all candidates and the actual collisions are saved to a contact manifold. For this implementation a finite number of contacts can be assigned to the manifold. For completely solid spheres eight is the theoretical maximum of contacts. However, since we use discrete time steps the spheres could theoretically overlap and more information would need to be saved. On the other hand most spheres have neighboring spheres in the same body which cannot cause collisions and potentially fewer collisions could be saved. As the amount of collisions saved affect the performance quite heavily, for this implementation a total of four collisions are saved. This could potentially lead to incorrect results but during all the testing no significant error could be seen.

The collisions are transferred between shader by storing them in a contact manifold. The contact manifold contains space for the particle id of the other colliding particle, the impulse to be applied and the normal along which we should apply it. In the current implementation the maximum amount of collisions transferred for a particle is four. However, this leaves room for error as with interpenetrating particles more than four collision per particle can happen. Testing has shown that this happens rarely and gives only a small impact on the outcome, it does however give a big performance increase when transferring fewer contacts.

In the collision detection the number of collisions between body pairs is also determined.

### 5.6.2 Impulse calculation

The contact manifold contain all the contacts for this sphere and the respective id’s of the spheres. For each id in the manifold the counterpart sphere is read and then the impulse is calculated as described in the theory chapter. The contact manifold also contain enough space to save the impulses and a normal direction to be used in the velocity step. By not simply accumulating the forces in the sphere and keeping both the impulse and the direction we can reconstruct the collision point when updating the velocity. This becomes extra important for low resolution decompositions.

### 5.6.3 Velocity update

This shader step dispatches across all bodies instead of across the particles. This is necessary as we need to sum all the impulses from all the particles in the body. Since no atomicAdd for float is supported in OpenGL 4.5, this is done through a gather scheme. For each particle add the influence from the current particle to the body. Since internal particles have no influence on the simulation those are removed to increase performance of this shader step.

---

1There are ways to exploit atomic add for integers to emulate close to floating point atomic add (fixed point arithmetics), this has not been investigated further.
Initially the workload of calculating the torque and linear moment was left up to the velocity shader, this meant that fewer workgroups were issued and each workgroup had to do an increasing amount of work and the parallelism of the GPU was utilized poorly. A new shader as a pre-velocity step was added were the calculations for the velocity was performed on a per particle basis and the much faster shared memory was utilized to avoid read-write collisions. Each workgroup got a shared array with enough space to save each particles’ torque and linear moments. For the initial implementation the scaling of the velocity step was very poor. For the tests the total amount of particles in the system was kept constant but the amount of particles per body was increased for each test. This results in fewer bodies in total for each new test. The results can be seen in figure 5.7. For the more workload aware implementation, the pre-velocity step, the results can be seen in 5.8. As one can easily see the latter scales much more reasonably.

![Performance with increasing number of particles per body.](image)

**Figure 5.7:** Poor scaling for the velocity step with increasing number of particles per body.
**Figure 5.8:** Improved workload gives reasonable scaling for the velocity step with increasing number of particles per body.

The flowchart is updated accordingly to what can be seen in figure 5.9. The calculations performed in the velocity pre-calculation step could just as easily be implemented into the impulse shader step instead, but has been given a separate shader step for maintainability.
After the velocities have been updated the positions and orientation can be updated through integration as described in the theory chapter.
5.7 Optimizations

5.7.1 Impulses

In the impulse shader we need to fetch all the other particles which have a collision with the current particle. Storing these to a local variable before doing any calculations doubled performance.

1: for all particles in collision with current particle do
   Add particle to local container
2: end for
3: Synchronize, ensure all threads have performed their reads
4: Calculate impulses

5.7.2 Velocity update

When calculating the forces' contributions to the velocities once again local variables are used and synchronizations. This time the aim is two fold: fast writes during calculations to the local variables, and coherent writes to buffer memory once all calculations are done.
In the following chapter the result of simulating physics through spherical decomposition will be discussed and performance testing will be presented.

6.1 Performance

Since different shaders are impacted differently by different problems a few performance test series have been performed and will be discussed below. For all tests the objects being approximated are dropped into a bin consisting of static particles. For the complete data collected and settings used see appendix A.

The times presented for velocity and impulse are the total time for all ten iterations. For the spherical decomposition method, the shapes of the objects do not affect the simulation significantly and simple boxes have been used in all tests, with one exception. In the last test series a Utah Teapot has been voxelized for a more accurate comparison with Bullet 2.83 using decomposed Utah Teapots.

6.1.1 Increasing number of particles per body

Increasingly fine decomposition leads to more particles per body. For this test the number of particles per body was increased for each series and is compensated by adding fewer bodies so that the number of particles in the system in total is kept constant. For all three series some particles from the bin had to be either added or removed to keep their total number equal between the tests.
As we can see the velocity shader time increased with the number of particles within a body. This is not surprising as the shader performs a gather scheme for summation of the impulses, in which it does a number of global reads proportional to the number of particles in the body per thread.

### 6.1.2 Increasing number of static particles

For this test we let the number of static particles in the simulation increase by increasing the dimensions of the bin.

**Figure 6.2:** Chart of performance of each of the shader steps for increasing number of static particles.
6.1 Performance

We see some time increase in the grid collision detection and the impulse calculations and a small increase in the grid generation. The increase in impulse is not surprising as each particle, even static ones, perform global reads and writes per particle in the impulse shader step. One could potentially use an early exit strategy and not perform the impulse calculations for static particles, however, unless the whole workgroup contains only static particles, no performance gain is guaranteed as returning threads within a workgroup do not ensure that new work can be performed.

6.1.3 Increasing number of bodies and dynamic particles

For this test we increase both the number of bodies and particles in the system.

![Performance with increasing bodies and particles](image)

**Figure 6.3:** Chart of performance of each of the shader steps for increasing number of bodies and particles.

Here we can see an increase in every shader step. Worst scaling can be seen among the impulse and velocity step. Those shader steps are the two involved in the stabilization loop and both are executed ten times each per update, so optimizations to these two steps should be prioritized in the future.

6.1.4 Increasing grid size

This test measures the performance of the grid size. For a M-by-M-by-M grid we can detect collisions efficiently on coordinates between -M/2 and +M/2 in each dimension.
Figure 6.4: Chart of performance of each of the shader steps for increasing number of cells in grid

We see that the grid generation scales poorly with the grid size, most likely cubically as the number of grid cells increased cubically and the generation includes the prefix sum and sorting. We can also note that the grid collision detection does not increase. This is expected since we still read an equal amount of particles no matter the grid size and the particles are sorted from the grid generation shader.

6.1.5 Increasing cell size

For this test we increase the size of the simulation domain by increasing the cell size instead. This leads to more candidate particles, thus more possible collisions have to be evaluated, as more and more will end up in the larger and larger surrounding cells.
6.1 Performance

Figure 6.5: Chart of performance of each of the shader steps for increasing cell size

It is only the grid collision detection shader which takes a penalty to performance while changing the cell size, as expected. Note however that for each new series in this test we get two times the size of the simulation domain in each dimension. Also note that the time increase from cell size 2 to cell size 8 is only 0.34 ms in total for all shader steps. This is less computationally complex than increasing the grid size from 32 to 128, which gives an increase of 0.93 ms.

6.1.6 Modified Normals

While difficult to determine whether the end result is more or less physically correct, it can be tested for how many stabilization iterations the simulation is still stable (i.e. avoids tunneling). The testing was performed on 5x5x5 cubes, so a reasonable amount of particles should have modified normals. Without modified normals at 6 iterations there was severe tunneling between the objects. With replaced normals as low as 4 iterations gave no tunneling. In addition when using too few iterations and tunneling does occur, with modified normals we have a greater chance of the situation resolving itself since the edge particles will always push to separate the objects, much as the signed distance field described by Macklin et al. [2014].

6.1.7 Tighter voxelization

Tighter voxelization led to more accurate collision results since the surface is modeled more accurately. In addition one could decrease the number of iterations needed for tunnel prevention and stability. This is most likely due to the reduced depth of the indentations between particles which led to the normals and resulting forces more often being directed outwards from the object with a lesser angle. With deep indentations between particles the forces will be angled more steeply and some of the force will not be used for separating the objects.
This method will take a penalty to the collision detection shader step in terms of performance since many of the bodies own particles will register as collisions for the tighter voxelization. During testing the benefit of lowering the number of stabilization iterations greatly outweighed the extra time needed in the collision detection. Again testing on the 5x5x5 cubes without tightened voxelization 6 iterations resulted in tunneling, with tightened voxelization as low as 3 iterations still managed without tunneling.

6.1.8 Voxelized teapot

For this final test a Utah Teapot was voxelized into 171 spheres, see figure 6.6. The testing constist of two test series. One with 256 bodies and one with 512 bodies. Each setup tested with 10 and 5 stabilization iterations. These results can be compared to the results from Bullet in chapter 4 and are therefore included in the graphs. In figure 6.7 is a finished simulation with the voxelized teapots rendered with the model instead of spheres and in figure 6.8 is a close up of the simulation.

![Teapot voxelized with 171 spheres](image)

*Figure 6.6: Teapot voxelized with 171 spheres*
At 512 bodies and 5 iterations spherical voxelization reaches a speedup of a factor of 2 compared to Bullet with no sleeping. At 10 iterations a speedup of a factor of 1.3. When Bullets’ sleeping is activated it's slower than spherical voxelization at 5 stabilization iterations but faster than spherical voxelization at 10 stabilization iterations. See figure 6.9.
At 256 bodies the speedup is not as large. Comparing 5 stabilization iterations on the GPU solution to Bullet without sleeping gives a speedup with a factor of 1.6. At 10 stabilization iterations spherical voxelization is somewhat slower than Bullet. With sleeping enabled Bullet is faster than the GPU solution at 5 stabilization iterations, even if only by a small margin. See figure 6.10

At 1024 bodies spherical decomposition is faster even when comparing 10 stabilization iterations to Bullet with sleeping enabled, which is a good indication that
the spherical decomposition system scales better than Bullet with many bodies, see figure 6.11.

![GPU vs CPU 1024 bodies](image)

*Figure 6.11: Chart of performance comparison between spherical voxelization and Bullet for 1024 bodies*

6.2 Limitations

6.2.1 Per body particle limit

Currently in the velocity update shader there is a for-loop across all the spheres belonging to the current body. This is done to gather all the forces from the spheres. To ensure performance on the GPU one uses unrolling, i.e. the for-loops content is duplicated for as many iterations as necessary. However, on the GPU this means two things: for-loops with dynamic length can cause problems or increased runtime and there is an upper bound on the number of iterations that are allowed. For the hardware used for the thesis, this limit is 4096. Therefore the method is currently limited to a maximum of 4096 spheres per body.

6.2.2 Tunneling

Tunneling can occur in the system as no protection against the phenomenon is implemented. For linear movement such a protection is easily implemented, however with rotations the protection is slightly more difficult to implement properly. For long thin objects a smaller time step is recommended.

6.2.3 Limited domain

In addition the domain of the simulation has to be determined before the simulation starts as a uniform spatial partitioning grid is used for the collision detection.
6.2.4 False alignment

Since the method uses spheres as a principal shape the result has a tendency to align with the grid. A single box consisting of 27 sphere fall on a plane, also made up of spheres at a slight rotation. All impulses present will drift towards the cube aligning with the plane at intervals of 90 degrees. The effect would be less apparent with less regularly shaped objects and is remedied somewhat by friction, albeit not completely.

6.3 Comparison to bullet

6.3.1 Concavity

The method can handle concavity through spherical decomposition. The detail of the resulting collision shape is highly dependent on voxelization resolution. Since the collision detection is designed to operate on spheres of equal size objects such as the Utah teapot with both large and small features can quickly produce a high number of spheres. HACD can model these with a single larger hull for the body and smaller hulls for the smaller features.

6.3.2 Performance

In section 6.1.8 we see that when simulating many rigid bodies spherical voxelization shows an improvement in performance over Bullet. However increasing the detail in the decomposition will quickly reduce the performance.

Since the velocity and impulse shader steps are iterated several times for stabilization these steps are quite vital to the performance of the method. More work could be spent on improving the performance of these shaders, but one should also investigate if the stabilization iterations could be exchanged for shock propagation, as indicated by the results achieved by Macklin et al. [2014] with nVidia Flex.

6.3.3 Correctness

Correctness for spherical decomposition depend highly on how finely the model is decomposed, much like how Bullet’s HACD depends on how many vertices are allowed per model and how many models are used. Boxes with as low as 27 particles look convincing, but e.g. a Utah Teapot may need many more particles to produce convincing results, at 171 particles the Utah teapot looks decently decomposed.
Conclusion and future work

With both a CPU and GPU method implemented this chapter will present the findings of this project and discuss future improvements.

7.1 Conclusion

7.1.1 HACD

In this thesis as a baseline comparison a well established open source physics engine (Bullet) has been used to simulate concave objects through Hierarchical approximate convex decomposition. This method has been investigated previously and there were indications that it would produce good results. It has therefore been implemented and tested for the purpose of acting as a baseline comparison.

As suspected Bullet with HACD produces good result with decent performance and the method includes very few design parameters. The approach using HACD gives a robust and easily managed solution, with the only drawback being that the HACD may produce unexpected results for certain meshes. However, when this happen one always has the alternative to design the sub shapes by hand in external applications.

7.1.2 Spherical decomposition

In this thesis we have shown that a physics simulation based on impulses with calculations performed on the GPU is possible. Certain performance optimizations have been implemented, but far from all options have been investigated. In terms of performance the proposed solution can match established solutions (in the form of Bullet 2.83), when an object is decomposed into around 200 particles.
The solution scales better than Bullet with respect to the number of objects in the scene, however quite a large number of bodies need to be in the scene for the GPU solution to outperform Bullet.

The solution shows good promise, in addition to the results presented within this thesis report is the release of Flex by Macklin et al. [2014] at nVidia during the course of the thesis, which uses some of the same core concepts.

Compared to HACD, voxelization is more intuitive and the results are easier to modify when needed, however, HACD has support in design tools such as Blender, which make for easy modifications through it’s design tools.

### 7.2 Future work

The number of iterations needed for stability increases with increased stack sizes. The reasons for this is motivated in section 5.3. Two candidates to replace the stabilization iterations are the approximate shock propagation method described by Macklin et al. [2014] or the original shock propagation method proposed by Guendelman et al. [2003].

Making use of non-uniform structures such as oct-trees or kd-trees for the collision detection, which would make the structure scale better with objects spread across large distances.

Investigating the use of the Parallel-Sweep-And-Prune algorithm instead of the uniform sorted grid for collision detection, a method which would most likely benefit larger objects as they work on bounding boxes and could be performed in a two step fashion. First detecting whole bodies which may have collision. Remove particles which belong to non-colliding bodies and redo the collision detection on all remaining particles. In addition this method would support differently sized particles better than the current one.

The collision carrier structure, the contact manifold, currently only carries 4 collisions per particle. This limits the system in two ways, the system can miss collisions when more than 4 collisions occur simultaneously on a particle and the system is unfit to use differently sized particles. Larger particles would be likely to experience more simultaneous collisions than four and the system would lose needed information. Extending this collision transfer structure to become dynamic in size would be an useful feature. However, it would most likely have to be extended carefully as not to impact performance too heavily.

Solid halfspheres, as described by Macklin et al. [2014], are based on the idea that the spheres that constitute the bodies can have modified normals on the inside of the objects to counteract tunneling. When two bodies become tangled the solid halfspheres’ modified normals on the inside of the object ensure that the objects push apart and separate.

Investigating alternative methods of solving the interpenetrations instead of velocity biasing.
Appendix
Test data
### Number of Particles per body

<table>
<thead>
<tr>
<th>Bodies</th>
<th>1024</th>
<th>222</th>
<th>82</th>
</tr>
</thead>
<tbody>
<tr>
<td>Particles per body</td>
<td>27</td>
<td>125</td>
<td>343</td>
</tr>
<tr>
<td>Gridsize</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Static Spheres</td>
<td>6128</td>
<td>6124</td>
<td>5966</td>
</tr>
<tr>
<td>Dynamic Spheres</td>
<td>27621</td>
<td>27625</td>
<td>27783</td>
</tr>
<tr>
<td>Total Spheres</td>
<td>33749</td>
<td>33749</td>
<td>33749</td>
</tr>
</tbody>
</table>

| Stabilization Iterations | 10 | 10 | 10 |

- **Move to world**: 0.173856, 0.171933, 0.176495
- **Grid Generation**: 1.41836, 1.41551, 1.42996
- **Grid CD**: 1.61042, 1.69701, 1.74039
- **Impulse**: 4.61248, 4.21953, 4.23691
- **Velocity**: 6.89385, 15.4827, 36.7839
- **Position**: 0.070222, 0.021548, 0.014331
- **Total Phys**: 14.77919, 23.00823, 44.38199
- **Rendering**: 48.9705, 49.4565, 44.9628

### Performance with increasing number of particles per body.

![Performance chart](chart.png)

- **Time [ms]**
  - Move to world
  - Grid Generation
  - Grid CD
  - Impulse
  - Velocity
  - Position

- **Particles per body / Number of bodies**
  - 27
  - 125
  - 343
  - 1024
  - 222
  - 82
## Number of Particles per body

### Improved Velocity Step

<table>
<thead>
<tr>
<th>Bodies</th>
<th>1024</th>
<th>222</th>
<th>82</th>
</tr>
</thead>
<tbody>
<tr>
<td>Particles per body</td>
<td>27</td>
<td>125</td>
<td>343</td>
</tr>
<tr>
<td>Gridsize</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Static Spheres</td>
<td>6128</td>
<td>6124</td>
<td>5966</td>
</tr>
<tr>
<td>Dynamic Spheres</td>
<td>27621</td>
<td>27625</td>
<td>27783</td>
</tr>
<tr>
<td>Total Spheres</td>
<td>33749</td>
<td>33749</td>
<td>33749</td>
</tr>
</tbody>
</table>

| Stabilization Iterations | 10 | 10 | 10 |

| Move to world | 0,174129 | 0,161857 | 0,170525 |
| Grid Generation | 1,30746  | 1,30739  | 1,30963  |
| Grid CD | 1,86873  | 1,81506  | 1,92837  |
| Impulse | 4,18692  | 3,74853  | 3,73068  |
| Velocity | 7,41323  | 7,94675  | 9,22635  |
| Position | 0,022917 | 0,01991  | 0,011914 |
| Total Phys | 14,97339 | 14,9995  | 16,37747 |
| Rendering | 49,6672  | 48,7197  | 47,5851  |

![Performance with increasing number of particles per body](image)

**Performance with increasing number of particles per body.**

- Move to world
- Grid Generation
- Grid CD
- Impulse
- Velocity
- Position

Particles per body / Number of bodies:

- 27
- 125
- 343

Particles per body:

- 1024
- 222
- 82
### Increasing number of static particles

<table>
<thead>
<tr>
<th>Particles per body</th>
<th>27</th>
<th>27</th>
<th>27</th>
<th>27</th>
<th>27</th>
<th>27</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gridsize</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Static Spheres</td>
<td>1600</td>
<td>6124</td>
<td>17524</td>
<td>22244</td>
<td>24088</td>
<td>29928</td>
<td>38268</td>
</tr>
<tr>
<td>Dynamic Spheres</td>
<td>3429</td>
<td>3429</td>
<td>3429</td>
<td>3429</td>
<td>3429</td>
<td>3429</td>
<td>3429</td>
</tr>
<tr>
<td>Total Spheres</td>
<td>5029</td>
<td>9553</td>
<td>20953</td>
<td>25673</td>
<td>27517</td>
<td>33357</td>
<td>41697</td>
</tr>
<tr>
<td>Bodies</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Stabilization</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

| Move to world      | 0.032296| 0.062419| 0.125912| 0.137525| 0.150529| 0.176062| 0.137525 |
| Grid Generation    | 1.0785  | 1.13032 | 1.31521 | 1.39307 | 1.43146 | 1.54831 | 1.39307  |
| Grid CD            | 0.250523| 0.462628| 0.974864| 1.19119 | 1.26074 | 1.48446 | 1.19119  |
| Impulse            | 0.87959 | 1.40618 | 2.76055 | 3.32511 | 3.51522 | 4.18639 | 3.32511  |
| Position           | 0.011234| 0.011369| 0.011636| 0.011623| 0.011633| 0.012218| 0.011623 |
| Rendering          | 7.55843 | 13.213 | 25.668 | 24.9235 | 30.8659 | 43.0431 | 44.5063  |

### Performance with increasing number of particles, constant number of bodies (128) and moving particles (3429)

![Performance graph](image)
GPU vs CPU 512

<table>
<thead>
<tr>
<th>Particles per body</th>
<th>171</th>
<th>171</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gridsize</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Static Spheres</td>
<td>16260</td>
<td>16260</td>
</tr>
<tr>
<td>Dynamic Spheres</td>
<td>87552</td>
<td>87552</td>
</tr>
<tr>
<td>Total Spheres</td>
<td>103812</td>
<td>103812</td>
</tr>
<tr>
<td>Bodies</td>
<td>512</td>
<td>512</td>
</tr>
<tr>
<td>Stabilization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterations</td>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>

| Move to world     | 0.354968 | 0.35388 |
| Grid Generation   | 2.42602 | 2.42467 |
| Grid CD           | 4.66109 | 4.6812 |
| Impulse           | 12.7351 | 6.41362 |
| Velocity          | 18.9996 | 9.5629 |
| Position          | 0.016908 | 0.012276 |

<table>
<thead>
<tr>
<th>GPU 10 its</th>
<th>GPU 5 its</th>
<th>CPU Bullet (without sleeping)</th>
<th>CPU Bullet (with sleeping)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Phys</td>
<td>39,19369</td>
<td>23,44855</td>
<td>51,9156</td>
</tr>
<tr>
<td>Rendering*</td>
<td>10,8238</td>
<td>12,9594</td>
<td></td>
</tr>
</tbody>
</table>

*Updates to the rendering pipeline was implemented before this test, but does not affect the simulation since rendering is not included

![GPU vs CPU 512 bodies](chart.png)
### Increasing gridsize

<table>
<thead>
<tr>
<th></th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Particles per body</strong></td>
<td>27</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td><strong>Gridsize</strong></td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td><strong>Static Spheres</strong></td>
<td>4564</td>
<td>4564</td>
<td>4564</td>
</tr>
<tr>
<td><strong>Dynamic Spheres</strong></td>
<td>3429</td>
<td>3429</td>
<td>3429</td>
</tr>
<tr>
<td><strong>Total Spheres</strong></td>
<td>7993</td>
<td>7993</td>
<td>7993</td>
</tr>
<tr>
<td><strong>Bodies</strong></td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td><strong>Stabilization Iterations</strong></td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

### Performance with increasing gridsize

- **Move to world**: 0.046983, 0.047222, 0.053908
- **Grid Generation**: 0.085874, 0.205237, 1.11193
- **Grid CD**: 0.447819, 0.393238, 0.39296
- **Impulse**: 1.20163, 1.21198, 1.21779
- **Velocity**: 3.99334, 3.87185, 3.9285
- **Position**: 0.011389, 0.011284, 0.011465
- **Total Phys**: 5.787035, 5.740811, 6.716553
- **Rendering**: 10.4942, 10.0881, 10.994

![Grid size along one dimension](image_url)
## GPU vs CPU 256

<table>
<thead>
<tr>
<th>Particles per body</th>
<th>171</th>
<th>171</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gridsize</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Static Spheres</td>
<td>16260</td>
<td>16260</td>
</tr>
<tr>
<td>Dynamic Spheres</td>
<td>43776</td>
<td>43776</td>
</tr>
<tr>
<td>Total Spheres</td>
<td>60036</td>
<td>60036</td>
</tr>
<tr>
<td>Bodies</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>Stabilization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterations</td>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>

| Move to world      | 0.224142 | 0.225285 |
| Grid Generation    | 1.86313  | 1.8604  |
| Grid CD            | 2.6756   | 2.67918 |
| Impulse            | 7.45588  | 3.73205 |
| Velocity           | 11.2099  | 5.53585 |
| Position           | 0.013203 | 0.014831 |

**GPU 10 its**  **GPU 5 its**  **CPU Bullet**  **CPU Bullet (with sleeping)**
| Total Phys         | 23,44185 | 14,0476  | 22,3175  | 13,298  |
| Rendering*         | 5,77968  | 6,62077  |          |         |

*Updates to the rendering pipeline was implemented before this test, but does not affect the simulation since rendering is not included.

---

**GPU vs CPU 256 bodies**

![Chart showing time comparison between GPU and CPU for 256 bodies](chart.png)
Increasing number of bodies and dynamic particles

<table>
<thead>
<tr>
<th>Particles per body</th>
<th>27</th>
<th>27</th>
<th>27</th>
<th>27</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gridsize</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Static Spheres</td>
<td>6124</td>
<td>6124</td>
<td>6124</td>
<td>6124</td>
<td>6124</td>
</tr>
<tr>
<td>Dynamic Spheres</td>
<td>3429</td>
<td>6885</td>
<td>13797</td>
<td>27621</td>
<td>55269</td>
</tr>
<tr>
<td>Total Spheres</td>
<td>9553</td>
<td>13009</td>
<td>19921</td>
<td>33745</td>
<td>61393</td>
</tr>
<tr>
<td>Bodies</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>1024</td>
<td>2048</td>
</tr>
</tbody>
</table>

| Stabilization Iterations | 10 | 10 | 10 | 10 | 10 |

| Move to world | 0.064333 | 0.085586 | 0.115471 | 0.183807 | 0.278943 |
| Grid Generation | 1.13499 | 1.17656 | 1.25673 | 1.41463 | 1.66245 |
| Grid CD | 0.46503 | 0.624012 | 0.95638 | 1.61799 | 3.0399 |
| Impulse | 1.40408 | 1.87229 | 2.82382 | 4.65736 | 8.05118 |
| Velocity | 3.97636 | 4.68459 | 5.12763 | 7.15563 | 10.994 |
| Position | 0.011482 | 0.012591 | 0.012113 | 0.016656 | 0.027225 |
| Total Phys | 7.056275 | 8.455628 | 10.29214 | 15.04607 | 24.0577 |
| Rendering | 9.47338 | 18.7482 | 28.4845 | 48.2664 | 87.7795 |

Performance with increasing bodies and particles
### GPU vs CPU 1024

<table>
<thead>
<tr>
<th></th>
<th>GPU 10 its</th>
<th>GPU 5 its</th>
<th>CPU Bullet (without sleeping)</th>
<th>CPU Bullet (with sleeping)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Particles per body</strong></td>
<td>171</td>
<td>171</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Gridsize</strong></td>
<td>128</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Static Spheres</strong></td>
<td>16260</td>
<td>16260</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dynamic Spheres</strong></td>
<td>175104</td>
<td>175104</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total Spheres</strong></td>
<td>191364</td>
<td>191364</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bodies</strong></td>
<td>1024</td>
<td>1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Stabilization</strong></td>
<td>10</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Updates to the rendering pipeline was implemented before this test, but does not affect the simulation since rendering is not included.

<table>
<thead>
<tr>
<th></th>
<th>GPU 10 its</th>
<th>GPU 5 its</th>
<th>CPU Bullet (without sleeping)</th>
<th>CPU Bullet (with sleeping)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Move to world</strong></td>
<td>0.620234</td>
<td>0.620333</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Grid Generation</strong></td>
<td>3.39008</td>
<td>3.3725</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Grid CD</strong></td>
<td>9.67038</td>
<td>9.75827</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Impulse</strong></td>
<td>23.2971</td>
<td>11.7669</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Velocity</strong></td>
<td>34.2611</td>
<td>17.0391</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Position</strong></td>
<td>0.018383</td>
<td>0.02212</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**GPU vs CPU 1024 bodies**

![GPU vs CPU 1024 bodies graph](image-url)

- **Time [ms]**
  - GPU 10 its
  - GPU 5 its
  - CPU Bullet (without sleeping)
  - CPU Bullet (with sleeping)

- **Total Phys**
  - GPU 10 its: 71,25728
  - GPU 5 its: 42,57922
  - CPU Bullet (without sleeping): 94,665
  - CPU Bullet (with sleeping): 78,317

- **Rendering**
  - GPU 10 its: 20,6851
  - GPU 5 its: 20,5978
## Increasing cell size

<table>
<thead>
<tr>
<th>Particles per body</th>
<th>27</th>
<th>27</th>
<th>27</th>
<th>27</th>
<th>27</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell Size</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>Gridsize</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Static Spheres</td>
<td>4564</td>
<td>4564</td>
<td>4564</td>
<td>4564</td>
<td>4564</td>
</tr>
<tr>
<td>Dynamic Spheres</td>
<td>3429</td>
<td>3429</td>
<td>3429</td>
<td>3429</td>
<td>3429</td>
</tr>
<tr>
<td>Total Spheres</td>
<td>7993</td>
<td>7993</td>
<td>7993</td>
<td>7993</td>
<td>7993</td>
</tr>
<tr>
<td>Bodies</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Stabilization</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iterations</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

| Move to world      | 0.053908 | 0.054277 | 0.054255 | 0.054019 | 0.056323 |
| Grid Generation    | 1.11193 | 1.09457 | 1.09291 | 1.09217 | 1.09631 |
| Grid CD            | 0.39296 | 0.438133 | 0.853676 | 2.41577 | 7.24179 |
| Impulse            | 1.21779 | 1.21608 | 1.19425 | 1.19544 | 1.18423 |
| Velocity           | 3.9285 | 3.88855 | 3.85128 | 3.94923 | 3.87158 |
| Position           | 0.011465 | 0.011453 | 0.011462 | 0.011604 | 0.011652 |
| Total Phys         | 6.716553 | 6.703064 | 7.057833 | 8.718232 | 13.46189 |
| Rendering          | 10.994 | 10.9622 | 11.052 | 11.0216 | 11.1365 |

### Performance with increasing cell size

![Performance with increasing cell size](image)
Bibliography


