I. INTRODUCTION

The simulation of physics phenomena on a computer is a useful tool in many fields. One common type of physics simulation is called rigid body dynamics. These simulations study the movement and interactions of non-deformable bodies. Rigid body simulations are used in many applications, such as video games, film production, virtual prototyping, and robotics.

This paper discusses real-time simulations, as opposed to offline simulations which may take hours to compute. Furthermore, this paper focuses on the simulation of rigid bodies, although many of the techniques and principles are applicable to other types of real-time physics simulation, such as soft bodies or fluids.

Fast CPUs, along with research, have made possible the interactive simulation with thousands of rigid bodies at interactive frame rates. Despite these advances, there is a need for efficient execution of even larger simulations, for example to simulate believable destruction and fracture effects.

The availability of fast graphics processing units (GPUs) capable of massively parallel computation and increased programmability has created a need for new kinds of algorithms in rigid body simulations. The existing research is often focused on algorithms that perform well on a single-threaded CPU, but cannot fully utilize GPUs or even multi-core CPUs. To achieve high performance on a GPU, the algorithms need to be able to execute in parallel, have well-designed memory access patterns, and efficiently handle coordination with the CPU. [1]

In section II, a brief outline of the different stages involved in a rigid body simulation is given. In sections III and IV, the differences of suitable CPU and GPU algorithms are explored.

II. SIMULATION OUTLINE

The state of a rigid body simulation is defined by the initial state of the system and the iterations of the simulation loop. Each iteration of the simulation loop is discrete step forward in time. The simulation loop consists of three different stages: kinematic state update, collision detection, and collision response.

The first stage of the simulation loop updates the kinematic state of the bodies. For each simulated body, numerical integration is used to compute new velocities based on the acceleration affecting the body, and then again to compute the new pose based on the computed velocity. It is trivial to transform this stage to be used efficiently on a GPU, since each simulated body can be updated in parallel.

Updates to the positions in the first stage may have caused bodies to intersect, that is, to collide with each other. The second stage inspects the bodies to find these collisions, and generates contact reports for each colliding pair of bodies. In this stage, it is not possible to simply update each body in parallel, since collisions by definition involve at least two different bodies.

The contact reports generated in the collision detection stage are used in the third stage to form responses to the collisions. This involves pushing bodies apart to avoid intersections and applying friction forces. Collision response is not generally easy to parallelize since the responses of interacting bodies affect each other, for example when multiple bodies are stacked.

III. COLLISION DETECTION

In a simulation of n bodies, each body may in theory collide with every other body, resulting in n(n-1) collision pairs to consider. In practice, there are much fewer actual collisions, since the bodies are sparsely spread around the simulated scene. The collision detection problem is often divided into two successive phases: a broad phase and a narrow phase. In the broad phase, collision culling is performed to quickly reduce the set of n(n-1) collision pairs into a smaller set of potentially colliding bodies. In the narrow phase, the actual collisions are detected, and the position and direction of each collision is calculated.

Sweep and prune, also known as sort and sweep, is a common algorithm for broad phase collision detection. The basic idea of the algorithm is to project the volumes of the bodies in the scene onto a single arbitrarily chosen axis. The order of the volumes along the axis is sorted, and then a sweep is done along the axis to find all the pairs of bodies with overlapping projections. Since a colliding pair must have overlapping projections on any axis, the non-overlapping pairs can be discarded by the algorithm. In a simulation with high temporal coherence for the positions of the objects, there are few changes in the order of the projections along the sweep axis between successive simulation steps. In such cases, it is efficient to do the sorting incrementally using insertion sort.

While the incremental sweep and prune algorithm is a common choice in CPU simulations, it is difficult to parallelize on a GPU, and the assumption of temporal coherence is less valid in large-scale simulations. GPU implementations of sweep and prune perform a new sort at each step using a parallel radix sort. Implementations also exist for performing the sweep step in parallel. [2]

Spatial subdivision techniques are another approach to broad phase collision detection. In one spatial subdivision technique, the three-dimensional space is covered with a uniform grid. For each grid cell a list is kept of those bodies whose volume...
hitting the grid cell. From the grid representation, it is easy to find the bodies which will require a near-phase collision check, since a collision between two bodies is only possible if they share at least one grid cell.

Multiple variations of the uniform grid method exist. Le Grand [3] uses bounding spheres to represent the volumes of the bodies. In Le Grand’s implementation each grid cell can be processed separately, which leads to an efficient GPU implementation.

Harada [4] uses another variation suitable for GPUs. Instead of using a bounding sphere, the shapes are approximated by generating a uniformly spaced voxel representation. In Harada’s GPU implementation, no distinction is made between broad phase and narrow phase collision detection. The information stored in the grid cells is also used to generate an approximate collision report for collision response, where the accuracy depends on the size chosen for the grid cells.

If the bodies in the simulation are spread non-uniformly around the space, or if there is a disparity between the sizes of the bodies, a uniform grid is wasteful. Then, a better data structure would be one that partitions the space non-uniformly, such as a bounding volume hierarchy (BVH) tree. BVH broad phase collision detection has been implemented for GPUs, but updating the BVH tree when bodies move has proven to be hard to parallelize. [5]

IV. COLLISION RESPONSE

The penalty method is a simple algorithm for collision response. For each colliding body, a repulsive damped spring force is applied, which aims to separate the bodies in the next simulation step. The penalty method is simple to calculate and extremely well-suited for GPUs since each body can be processed in parallel. In practice, there are numerous problems in this approach. The parameters of the repulsive force have to be carefully tuned for a realistic and stable simulation, and situations with multiple bodies having multiple contacts are not easily handled. [6]

To overcome the issues of the penalty method, the collision response problem can be modeled to consider the whole set of simulated bodies. The problem is often formulated as a linear complementarity problem (LCP). Numerous different methods exist for iteratively finding a solution to the LCP problem.

In single-threaded CPU implementations, a commonly used method is projected Gauss-Seidel (PGS). The PGS method is a poor fit for the GPU: it cannot process contact pairs in parallel if the pairs affect the same simulated body. It also requires persistent order in the contact generation to avoid jitter between successive simulation frames. Due to these problems, much of the research on GPU accelerated rigid body dynamics has been focused on the development of LCP solvers that can solve a large amount of contacts in parallel.

A Jacobi-based LCP solver is an alternative that does not require persistent ordering and offers better potential for parallelism. The downside of a Jacobi-based solver is that a straightforward implementation converges to a solution much more slowly than when using a PGS solver. Tonge et al. [7] describe a parallel Jacobi-based technique with quick convergence. Their technique also avoids jitter without resorting to persistent ordering.

If it is possible to ensure that none of the contact pairs share colliding bodies, the PGS method can be used to solve the pairs in parallel. This can be achieved by partitioning the full set of contact pairs to separately solvable batches. Partitioning the set is in itself a problem that is difficult to do in parallel. Harada et al. [8] create batches in parallel on the GPU with minimal global communication by using local atomic operations.

Some research has been done on methods that step further from the common PGS solver scheme. Renouf and Alart [9] describe a conjugate gradient method for solving the LCP problem. Tasora et al. [10] parallelize the collision response problem via an alternative to the LCP formulation.

V. SUMMARY

The techniques developed for parallel computation of rigid-body dynamics have enabled the development of physics engines which utilize GPUs. Examples of such physics engines include the proprietary PhysX [11] by NVIDIA, and the open source Bullet [5]. Although the research in parallel techniques is ongoing, the currently available implementations have already shown significant increase in performance compared to CPU implementations.

While it is now possible to compute the entire simulation only on the GPU, there is some research comparing the relative strengths of CPU and GPU simulations. For example, Harada [12] describes a simulation optimized for shared-memory CPU/GPU architectures. In addition to GPU implementations, these parallelizable algorithms are also useful for multi-core CPUs.

REFERENCES


