Parallel Decomposition of Three-Dimensional Rotation Space

Mateusz Henc and Joanna Porter-Sobieraj

Warsaw University of Technology, Faculty of Mathematics and Information Science
Plac Politechniki 1, 00-661 Warsaw, Poland
{m.henc,j.porter}@mini.pw.edu.pl

Abstract. The subject of this work is to study the capabilities of using CUDA in decomposing the configuration space of a rigid body. This paper focuses on the subdivision of 3-dimensional rotation space, to identify the collision-free configurations of a body in the presence of obstacles. The results obtained for the worst case of checking the collisions without making use of any hierarchy of the objects, show the potential of the CUDA approach. Parallel GPU computing enhances efficiency from several to several dozen times for higher resolutions of subdivisions, in comparison to CPU calculations.

Keywords: Motion Planning, Parallel Collision Detection, Space Decomposition.

1 Motivation

One of the fundamental problems in VR programming is moving an object in such a way so as to avoid collisions with obstacles on the virtual scene. A typical motion planning task is to find the body a collision-free path from an initial to a target configuration. While moving along this path the body cannot collide with any static obstacles or other moving objects defined in the environment. Motion planning algorithms are used in many areas, e.g. CAD/CAM software, industrial automation, computer games and animation.

There are many methods to determine a collision-free path. The early ways – decomposition [1] or roadmap methods [2, 3] – are based on a combinatorial approach. Such algorithms build an explicit representation of the collision-free configuration space. Regrettably, they are exponential in the number of degrees of freedom available to the moving object and, in practice, are not applicable to configuration spaces of dimension greater than four.

Newer, heuristic methods, like potential field sampling [4, 5], probabilistic roadmaps [6, 7] or adaptive cell decomposition [8], can cope with high time complexity. In contrast to combinatorial methods, they do not transform the obstacles from a three-dimensional workspace to the object configuration space, which is typically of dimension more than three. Consequently, instead of using an explicit collision-free configuration space, the calculation of the collision between the object and obstacles for chosen object positions is performed many times. Such an approach works well when the movement of the body is almost free. Its basic weakness is a decrease in capability in the presence of narrow passages, when the moving object has a low possibility of collision-free movement [9]. In narrow passages, the critical component that influences the performance of the heuristic motion planning algorithms is the procedure of checking the collisions.

A massive number of collision queries is also encountered in follow-up path programming. This technique uses a user-based initial approximation of the object trajectory. Such a path does not have to be perfect and may violate the obstacles. Afterwards, it is refined by local optimization to get the final collision-free movement. During this process the basic operation is, again, the computation of object and obstacle intersections.

The intense development of parallel processing in recent years, mainly because of multicore processors, has created new possibilities for speeding up basic, single-threaded procedures. The collision checking in path-planning algorithms presents an ideal algorithm to be implemented in parallel. To calculate the collision-free movement most methods use a very high number of collision queries in the vicinity of selected positions of the object. These collision queries can be processed independently. Therefore this task may be classified as an embarrassingly parallel problem. Our goal is to examine this hypothesis in practice and answer the question of whether and if yes, by how much, the collision tests may be sped up by executing them on modern graphics hardware using CUDA.

2 Problem Formulation

2.1 Definition of a Scene

Let's consider a scene in a three-dimensional bounded subspace $S \subset \mathbb{R}^3$, called the workspace. Fixed obstacles O_i and a moving solid B are non-deformable polyhedral models defined by sets of triangles. The number of faces describing the boundary of such solids may vary from several dozen to hundreds of thousands of triangles.

The moving object is treated as a rigid body that can rotate in the workspace. It is convenient to represent its position by a point c in the configuration space C, identified with a three-dimensional space SO(3).

For a given point $p \in B$, $p(c)$ means the new position of point p in the workspace after applying the rotation described by configuration c. The notation $B(c)$ denotes a set $\{p(c) : p \in B\}$, in other words – the solid after the transformation. Let a set O be defined as $\{q : q \in Q_i\}$. Then configuration c is collision-free if and only if $B(c) \cap O = \emptyset$. The set of all collision-free rotations is denoted by C_{free}, and a set $C_{obs} = C \backslash C_{free}$ represents the obstacles in the configuration space.

We should note, that defining the configuration space is a general approach that does not utilize the fact that the solids in the scene are described by triangles. Therefore the results obtained here may be generalized into other representations of solids.

The collision-free path is constructed with the use of collision-free configurations only. Therefore, all the motion planning algorithms need a representation of C_{free} or a method verifying if a given configuration c belongs to C_{free}. We calculate here an approximation of C_{free} and C_{obs} using a well-known uniform decomposition method applied to rotation space C. Such an approximation may be used not only in motion planning, but also in order to get a deeper knowledge of the shape and complexity of C_{free} and C_{obs} sets, e.g. by visualizing the rotation space or analyzing the boundaries of these sets. This will be a convenient tool to be used in future research.

2.2 Decomposition of the Rotation Space

There are several methods of representing rotations of the object in $\mathbb{R}^3$, e.g. axis-angle pair, rotation matrix, Euler angles and quaternions. We describe a rotation R as a composition of the three elementary rotations around the axes of the body frame $F = (p_0; X, Y, Z)$, this means by yaw γ, pitch β, and roll α angles:

$$R = R_Z(\alpha) \cdot R_Y(\beta) \cdot R_X(\gamma) \ . \tag{1}$$

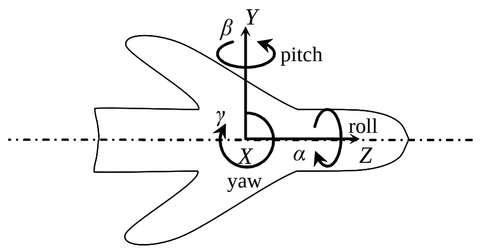

Fig. 1. Yaw, pitch, and roll rotations.

These angles are natural, simple to understand and convenient for performing transformations, e.g. in aircraft positioning or deriving Denavit-Hartenberg parameters for 3D kinematic chains. Obviously, there is a continuum of values for yaw, pitch and roll (RPY) angles that provide the same rotation. This may cause problems, because the topology of SO(3) is not determined precisely. However, the time needed for the decomposition of the parameter space, which we are interested in, depends mainly on the number of its dimensions. Therefore, the choice of a parameterization is a negligible part of the task.

Next, we define a three-dimensional box $A = [-\pi, \pi]^3$ representing RPY angles. The RPY box is then partitioned into a finite collection of cubic cells, regularly in each direction. These are arranged into a three-dimensional array of cubes, called the RPY array. Each array cell represents a configuration point corresponding to the cube vertex with the lowest RPY-coordinates. These configuration samples are then classified into C_{free} and C_{obs} sets.

Such a simple array structure is not memory-efficient, but in contrast to adaptive, tree structures, it is convenient for GPU parallel processing. Moreover, we concentrate on worst-case complexity for the approximation of a configuration space in narrow passages, assuming that the approximation accuracy of decomposition is sufficiently high. In such a case a slight change in the orientation of the body may cause it to cross the boundary of C_{free} and C_{obs} subspaces. Therefore, no approximation adjustment (decreasing) is analyzed in this paper.

On the other hand, the results presented in the next sections are also valuable for the randomized sampling of configuration points or adaptive structures of a rotation space, when a regular decomposition is defined only in subareas, such as an array of three-dimensional arrays or an octree. In such a case the total number of cubes, corresponding to the number of independent collision queries, is a variable in the task.

2.3 Classification of the Configuration Points

Verification, if configuration c is collision-free, is based on the test of whether a solid $B(c)$ intersects obstacles O_i. The objects are defined by sets of triangles. Therefore, the collision test utilizes a triangle-triangle intersection procedure, together with checking the relative location of the triangles. Our goal is to investigate the effectiveness of the pure approach. Therefore, no hierarchy for the object's triangles (like bounding solids or BSP trees) is imposed.

The implementation of the triangle-triangle intersection is based on a robust sequential algorithm given in [10].

Such classification may optionally include a signed scalar value corresponding to the distance of the moving body to the obstacles, or the depth of the penetration.

3 Implementation Notes

3.1 GPU Computing Overview

In recent years programmers have acquired technology called CUDA, which has allowed them to make numerical calculations on a GPU. In this section we introduce the main concepts of the CUDA parallel programming model. An extensive description is given in the NVIDIA CUDA Programming Guide [11].

The most modern GPUs contain hundreds of cores, able to execute tasks in parallel. Therefore, they have performances far higher than common CPUs. However, a GPU's architecture is different from a CPU's, and requires a tailor-made approach for designing algorithms. A multiprocessor executes hundreds of threads concurrently, and employs a SIMT (Single-Instruction, Multiple-Thread) architecture, very similar to the well-known SIMD (Single-Instruction, Multiple-Data) approach. The main difference is that SIMT also supports writing thread-level parallel code for independent, scalar threads.

A software environment provides, among other things, CUDA C language, which is simple to learn and understandable for programmers familiar with any standard high-level programming language. To exploit GPU parallelism, the programmer has only to define a special function, called a kernel, that is then executed in many copies by different CUDA threads. Threads are logically grouped in a grid of blocks. A particular thread can be identified by two built-in variables: a 2-dimensional block index *blockIdx* (*blockIdx.x*, *blockIdx.y*) and a 3-component vector *threadIdx* (*threadIdx.x*, *threadIdx.y*, *threadIdx.z*) handling the index of the thread within its block. Together with the kernel, the programmer defines its execution configuration, i.e. the number of blocks and the number of threads. The maximum number of threads per block is 512 or 1024, depending on the compute capability of a particular device. Threads in a block are executed on the same multiprocessor. Tasks have to be divided in such a way so as to utilize all cores, bearing in mind that the threads in different blocks cannot cooperate.

The threads are executed in chunks of 32 parallel threads called warps. Each thread in the same warp executes the same instruction path, so for conditionals or loops, if any of them steps into the branch, the whole warp must perform it. Therefore, GPU algorithms should be designed in such a way so as to limit inconsistent branching decisions in warps.

A GPU has its own memory hierarchy. The largest, global memory has the highest access latency. Because it is global, threads from different blocks can exchange information using it. The second memory type is shared memory. This memory is common for the whole block, faster than the global memory, but very limited in size. When many threads try to access the same memory bank, their requests are serialized, affecting performance. Therefore, to improve performance, limiting access to the global memory by using the shared memory for calculations is recommended.

3.2 Data Processing

The scene for the computation contains two types of models. The first, called a body, rotates in the scene, and the second one, called an obstacle, is fixed and may intersect with the body. The prepared triangle meshes are stored as OBJ files. When the procedure begins, all the objects are loaded and represented as lists of triangles with repeated vertices.

To verify whether there is a collision for a discrete configuration, every triangle from the body is checked against every triangle from the obstacle to check whether they intersect. The pseudo-code is presented in Algorithm 1 below.

Algorithm 1. The procedure for classification of configuration points.

```
for (each discrete configuration q:=(alpha,beta,gamma)) do
   for (each triangle tri_b from body) do
      Rotate tri_b by q;
      for (each triangle tri_o from obstacle) do
         if (tri_b intersects tri_o) then
            Tag q as collision;
            break;
         end if
      end for
   end for
end for
```

In GPU parallel implementation, each discrete configuration is processed by a different CUDA thread. We define a two-dimensional grid of blocks of threads. The size of the grid is equal to the resolution of the subdivisions of the RPY box in (α, β)-plane. The number of threads in a block is equal to the number of subdivisions for an angle γ. In other words, each block processes a fibre of configurations with angles α and β fixed, and one thread processes only one configuration $q = (\alpha, \beta, \gamma)$. Consequently, the configuration q for the thread is uniquely identified by the indices of the block (by both *blockIdx.x* and *blockIdx.y* components) and the thread (only the first component *threadIdx.x* is used).

Obviously, the code of the kernel (i.e. the function checking the collision for a thread) is common for all configuration points and, roughly speaking, corresponds to lines 2 to 10 of the sequential procedure in Algorithm 1. In the CUDA program, at first, all the triangles have to be copied into the GPU's global memory. Then each thread checks for intersections between every triangle from the body in the configuration q and every triangle from the obstacle. As global memory is slower than shared, in order to accelerate memory access, we divide the triangle array into smaller blocks (tiles) that fit into shared memory. The threads copy these parts of the triangle data from global to shared memory and perform computations in it. Algorithm 2 shows a pseudo-code for loop tiling to obtain more efficient memory usage.

Algorithm 2. Loop tiling for enhanced memory access carried out by a single thread on GPU.

```
for (each triangle's tile tile_b from body) do
   Load tile_b into shared memory s_body;
   for (each triangle's tile tile_o from obstacle) do
      Load tile_o into shared memory s_obs;
      for (each triangle tri_b from s_body) do
         Rotate tri_b by q;
         for (each triangle tri_o from s_obs) do
            if (tri_b intersects tri_o) then
               Tag q as collision;
               break;
            end if
         end for
      end for
   end for
end for
```

Shared memory is common for all threads in a block. Therefore, loading a tile from global to shared memory is done not for each thread separately, but by using multiple threads to exploit parallelism and to avoid redundant copying.

A three-dimensional RPY array of a required size is allocated in the GPU global memory. After computations each thread writes its result to the RPY array in an appropriate cell.

4 Results and Analysis

We tested the efficiency of GPU computing for a different number of collision queries, depending on the resolution of the RPY box discretization. We generated different scenes with various numbers of triangles (from one hundred to several thousand). The rotating body was always located near the narrow passage and almost half of its configurations collided with obstacles. The tests were performed on an Intel Core 2 Duo E8400 3GHz processor with an NVIDIA GeForce GTX 295 graphics card with 480 CUDA cores.

Table 1 and Table 2 show the times for processing the configuration space, defined with 16^3, 32^3 and 64^3 elements, by a GPU and a CPU respectively. We observe the expected linear dependency between the time and the number of triangle pairs. The time is in practice linear in the sizes of arrays as well. An eight-fold increase in the number of configuration samples results in a four to eight-fold increase in the amount of time taken.

Table 1. Timing (in seconds) for the classification of configuration points on a GPU for different numbers of samples in 3D arrays (*size*), number of triangles in the body (*Bn*) and in the obstacles (*On*).

Size	16×16×16			32×32×32			64×64×64		
	B 128	B 896	B 8964	B 128	B 896	B 8964	B 128	B 896	B 8964
O 86	0.4	1.0	9.1	0.8	4.6	40.5	5.6	32.2	300.1
O 1004	1.7	10.7	98.2	7.9	49.3	408.8	55.4	350.2	3261.0
O 9932	16.6	103.0	957.5	75.3	483.1	4039.8	544.9	3465.5	33437.5

GPU implementation is at least several times faster than a CPU procedure. The advantage of a GPU is clearly visible in the case of higher resolutions of the RPY box, when a GPU computes the result over twenty times faster.

Similar dependencies were also observed in the case of 128^3, 256^3 and 512^3 elements. For the biggest array (of over $1.3 \cdot 10^8$ samples giving an accuracy of angle discretization of a range $0.7°$), in the simplest case of a body of 128 and obstacles of 86 triangles, when almost $1.5 \cdot 10^{12}$ triangle-triangle intersections are calculated, the GPU procedure took 35 minutes, and for a CPU – it exceeded 14 hours.

The results presented confirm that a three-dimensional configuration space may be processed with higher efficiency on GPUs than CPUs. The biggest gain comes from the parallel nature of the given problem.

Table 2. Timing (in seconds) for the classification of configuration points on a CPU for different numbers of samples in 3D arrays (*size*), number of triangles in the body (*Bn*) and in the obstacles (*On*). A star ⋆ denotes interrupting the calculations when the estimated time exceeded 10 hours.

Size	16×16×16			32×32×32			64×64×64		
	B 128	B 896	B 8964	B 128	B 896	B 8964	B 128	B 896	B 8964
O 86	1.6	10.5	75.3	12.4	83.0	589.4	99.2	664.8	4656.2
O 1004	15.4	103.9	729.5	120.5	823.2	5668.6	962.2	6448.2	⋆
O 9932	152.3	1005.4	7139.6	1191.7	8223.1	⋆	9786.9	⋆	⋆

5 Conclusion and Future Work

This paper presented the parallel implementation of an algorithm for calculating the collision-free configurations of a rigid body. To calculate the collision information and to obtain a potentially efficient parallelization, separate sets of configurations are assigned to different GPU cores. The effectiveness of the implementation was verified by numerical experiments for scenes with different geometric complexities. The experimental results obtained, show the potential of the CUDA approach, especially for massive collision queries where CPU times are unacceptably long, even several dozen times longer than our preliminary version on the GPU.

There are many possible ways to improve our implementation. We are going to parallelize triangle-triangle tests and examine the gains from a triangle hierarchy that limits the number of processed elements. The next feature is to include in the RPY array elements information about the distance between the body and the obstacles, instead of simple information regarding whether this configuration is in free space or not. This will give a deeper knowledge about the structure of the configuration space and be especially usable in optimal path programming.

References

1. Schwartz, J.T., Sharir, M.: On the piano movers' problem: II. General techinques for computing topological properties of real algebraic manifolds. In: Advances in Applied Mathematics, vol. 4, pp. 298–351 (1983)
2. Canny, J.F.: The complexity of robot motion planning. MIT Press (1988)
3. Basu, S., Pollack, R., Roy, M.-F.: Computing roadmaps of semi-algebraic sets on a variety. In: Journal of the AMS, vol. 3, no. 1, pp. 55–82 (1999)
4. Barraquand, J., Latombe, J.-C.: A Monte-Carlo algorithm for path planning with many degrees of freedom. In: IEEE International Conference on Robotics and Automation, vol. 3, pp. 1712–1717 (1990)
5. Hwang, Y.K., Ahuja, N.: A potential field approach to path planning. In: IEEE Transactions on Robotics and Automation, vol. 8, no. 1 (1992)

6. Kavraki, L. E., Švestka, P., Latombe, J.-C., Overmars, M.H.: Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In: IEEE Transactions on Robotics and Automation, vol. 12, no. 4, pp. 566–580 (1996)
7. Kuffner, J.J., Jr., LaValle, S.M.: RRT-connect: an efficient approach to single-query path planning. In: IEEE International Conference on Robotics and Automation, vol. 2, pp. 995–1001 (2000)
8. Zhang, L., Kim, Y.J., Manocha, D.: A simple path non-existence algorithm using c-obstacle query. In: Springer Tracts in Advanced Robotics, vol. 47, pp. 269–284 (2008)
9. Hsu, D., Kavraki, L.E., Latombe, J.-C., Motwani, R., Sorkin, S.: On finding narrow passages with probabilistic roadmap planners. In: Workshop on the Algorithmic Foundations of Robotics, pp. 141–153 (1998)
10. Devillers, O., Guigue, P.: Faster Triangle-Triangle Intersection Tests. Research Report 4488, INRIA (2002)
11. NVIDIA Corporation: NVIDIA CUDA Programming Guide Version 3.0 (2010)