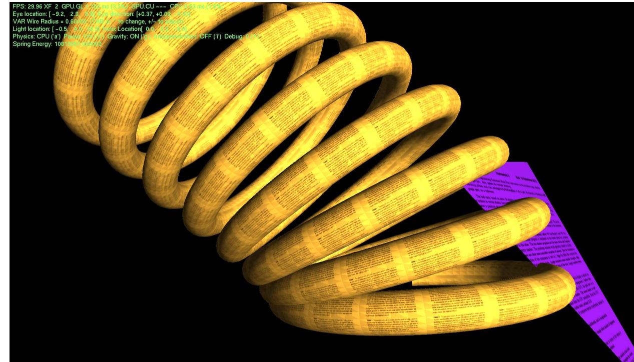
LSU EE 4702-1 Homework 5 Solution Due: 30 November 2012

Follow the Programming Homework Work Flow *instructions on the procedures page, substituting* `hw5` *for* `hw1`*. Also, update the include directory.*
Use `http://www.ece.lsu.edu/gp/refs/CUDA_C_Programming_Guide.pdf`*, the NVIDIA CUDA Compute Unified Device Architecture Programming Guide, as a reference. The code in this assignment requires OpenGL 4.3 and CUDA, which should be installed on* `snow`*,* `ice`*, and* `frost`*.*

See `hw5-sol-cuda.cu` in the repository for the solution, search for "Problem 1" or "Problem 2". The code is also available at `http://www.ece.lsu.edu/koppel/gpup/2012/hw5-sol-cuda.cu.html`



Problem 0: Follow the class procedures for homework but substitute hw05 for the assignment. Compile and run the code. It should display our familiar helix, but this time it's a simulated spring, and it should be drooping under gravity, see the illustration to the right. The physics can be performed by both CPU and GPU, by default it runs on the CPU, pressing 'a' will toggle it.

The CPU code can prevent interpenetration, this option is initially off, it can be toggled by pressing 'i'. The option was turned off when the image above was captured, notice that the bottom loops of the spring interpenetrate. The code for interpenetration can be found in `time_step_cpu`, it uses a brute-force approach, comparing every segment of the helix to every other, making it very slow. The goal of this assignment is **not** to use a clever broad-phase collision detection scheme to avoid $O(n^2)$ comparisons, but instead use the GPU to do the calculations faster. (In "real life" one might use both broad-scale collision detection and a GPU to do calculations.)

Problem 1: The routine `time_step_cpu` contains the CPU interpenetration code, clearly identified by comments. For this problem put the interpenetration code in CUDA kernel `time_step`, in file `hw5-cuda.cu`. The `time_step` kernel is launched in a configuration with one thread per helix segment. Each thread should test "its" segment against other segments. A correct solution will be faster than the CPU, but should still be slow.

See `hw5-sol-cuda.cu` in the repository for the solution, search for "Problem 1".

Problem 2: By GPU parallelism standards, the number of threads launched for the `time_step` routine is not large. In the default configuration there are 160 segments, just five warps. That will underutilize any GPU.

For this problem, add a new kernel to perform the interpenetration test. The new kernel should be launched with more threads than segments, taking advantage of more parallelism. Modify the existing `time_step` kernel so that it does not modify `helix_position` (but it still updates `helix_velocity` and the other two state variables). The new kernel should compute forces based on interpenetration tests, use that to update velocity, and then update the position.

A good solution of this problem requires the use of shared memory, atomic operations, and synchronization. That will be covered next week.

See `hw5-sol-cuda.cu` in the repository for the solution, search for "Problem 2".

A new kernel `time_step_intersect` is added. This kernel is launched with one *block* per helix segment (the `time_step` kernel is launched with one thread per helix segment). Each block compares one segment, call it the a

segment, against most of the others, call them the b segments. The index of the a segment is set to `blockIdx.x` and the index of the first b segment to look at is set to `threadIdx.x`. Each block uses a shared variable to hold the force, `threadIdx.x` is responsible for initializing it. When a thread detects an intersection it updates `force` using an atomic add, which is convenient but slow. This is acceptable because even when the helix is compressed each segment will be in contact with about four others, much less than the $80 \times 6 = 480$ segments that need to be examined (excluding near-neighbors).

After the loop `threadIdx.x` 0 will use the force to update velocity.

Two small changes were made to "help" the compiler optimize. The distance between the segments in the CPU code was found using a pNorm object, which needs to compute a square root in order to normalize. This object also provides the distance between the objects squared. It is the distance squared which is used to detect intersection, and if there is no intersection there is no need to perform a square root (which is used for finding the force). The NVIDIA compiler seemed to be computing the square root before the branch which usually makes sense for a GPU since that would help if even only one thread per warp would need it. But in our case the square root was only rarely needed. So, the tweak is to compute the distance squared separately.

The second tweak was moving the neighbor branch, the one that checks `min_idx_dist`, next to the distance check. That forces the GPU to load `helix_position` even when its not needed (which is rarely). Perhaps because there is no gap in the pattern of accesses performance is better.