CS195V Project 3 - NBody

March 10, 2012

Part I
Solar

Overview

This project will have you building a large scale particle simulation system. You will keep track of and update your particles using the GPU, specifically, OpenGL image load/stores.

This project is due Tuesday, March 13.

There will be an updated version of the support code available soon with instructions on how to set it up. Keep your eyes and ears open!

Requirements

For simulation, you must store and operate on particle data using OpenGL image units. You cannot only use multiple passes with framebuffers for your simulation; you must make use of the atomic image operations extension. You will probably need one pass to construct your hash/data structure and one to actually perform your physics calculations. Your system must be able to support at least 10,000 particles (this should not be difficult). Your particles must behave as gravitational bodies under a modified version of Newton's law of universal gravitation (provided below).

$$\mathbf{a}_i \approx G \cdot \sum_{1 \le j \le N} \frac{m_j \mathbf{r}_{ij}}{\left(\left|\mathbf{r}_{ij}\right|^2 + \epsilon^2\right)^{3/2}}$$

Where ϵ is a small "softening" factor which affects interactions at very close distances (read the link below to find out more).

You should use *Verlet integration* (not Euler integration) to keep track of your particles. Your particles do not have to combine or collide like they did in CS32 Solar. Below is the Verlet integration formula:

$$\mathbf{x}(t + \Delta t) = 2 \cdot \mathbf{x}(t) - \mathbf{x}(t - \Delta t) + \mathbf{a}(t) \cdot (\Delta t^2)$$

Make sure to choose a small enough time step Δt such that your integration / simulation does not blow up.

For 3D visualization, we recommend alpha-blended point sprites. Your program must run with at least 30fps with 10,000 particles. You can also tessellate some geometry for your bodies, which might look cooler, but has a greater performance hit. In this case, we will relax the performance requirement.

You must use some sort of spatial partitioning to speed up the simulation. We recommend the cell hashing method from lecture, but you can try other things (like octrees!) if you feel adventurous. For hashing, you may find a method similar to the linked list method in lecture to be helpful (hint: use an image to keep track of how many particles are in each hash cell).

You must implement at least *two* post-process effects shaders. One (or both) of these can be the same effects you used in previous projects. The performance requirement only applies to the base simulation before any post processing, so please include a way to turn off all extraneous effects so that we can just test the simulation alone.

Technique

You will need two framebuffers (or one framebuffer with two color attachments) to keep track of the two positions necessary for Verlet integration. All of the physics computations (velocitiy, acceleration, positions) should be done in a single shader. You may need additional shaders to manage your data structure.

To visualize, you will need to make some kind of point primitive to pass into your draw call (either with a large VBO or with instanced drawing).

To use your hash to speed up the physics calculations, for faraway hash cells you want to do you physics calculations with the total mass of the cell and its center rather than each individual particle inside. This is the same technique you used for your Octree in CS32's Galaxy (see Barnes-Hut Simulation).

Tips

Make sure that you properly declare the extensions you are using in each shader. The shader compiler will usually yell at you if you are missing something, but not always. Check the lecture slides for some of the specific extensions you will need.

The main new things for this project will be image units/operations and memory barriers. Try something simple at first just to make sure that you are correctly binding, modifying, and reading from image units before you start going wild.

For point sprites, call glEnable(GL_POINT_SPRITE) and glEnable(GL_VERTEX_PROGRAM_POINT_SIZE), which will allow you to adjust the size of a point sprite in the vertex shader by assigning gl_PointSize (you may, for example, want to scale the size based on the distance to the camera). When texturing the point sprite in the fragment shader, you can use gl_PointCoord which is equivalent to a texture coordinate for that particular point.

If you don't want an in/out value to vary between fragments, declare it as flat (i.e. flat out int v_id;).

To set the initial positions of the particles, you can call Texture::setData(). To generate some random values, use something like...

```
#include <random>

// setup the random number generators (uniform 0..1 and normal random)
std::mt19937 mtwist(1337);  // the prng seed is 1337
std::uniform_real<float> uniform;
std::normal_distribution<float> normal;
std::variate_generator<std::mt19937,std::uniform_real<float>> randu(mtwist,uniform);
std::variate_generator<std::mt19937, std::normal_distribution<float>> randn(mtwist,normal);

// get some random numbers
float normalDraw = randn();
float uniformDraw = randu();
```

Note: if you're not using the support code and do not have C++0x enabled, the random number generation is part of tr1.

Link

GPU Gems chapter on NBody simulation More information about the equations and such.

Part II
Fluids

Coming soon! We will put out a handout for this later. It will be due Tuesday, March 13.