

# GPU Project Proposal

## For Planetary N Body Problem

### By

Shahriyar Sharifi

Amir Hossein Maghsoudi

### Core idea

Newtonian n body problem is embarrassingly parallel. It can easily be accelerated via GPU parallelism. Furthermore, due to the traditional use of GPUs aka actual graphical applications, using OpenGL it is possible to share data between the physics engine and the render engine. So the CPU will only play the role of controller for our application. This will basically be something inspired by “Universe Sandbox”.

### Newtonian n body implementation

In this problem we have 3 things we need to keep track of. Planetary positions (Position table  $P$ ), velocity (Velocity table  $V$ ) and acceleration (Acceleration table  $A$ ). The formulas for the former two are as follows:

$$P(t + \varepsilon) = \varepsilon \cdot V(t) + P(t)$$

$$V(t + \varepsilon) = \varepsilon \cdot A(t) + V(t)$$

With  $\varepsilon$  being the time step.

At time  $t$ , each vector  $a_i$  is derived from the following sum:

$$a_i = \sum_{j=0}^{n-1} \frac{m_j \cdot (p_j - p_i)}{|p_j - p_i|^3}$$

Additionally, while constant, a mass table is required for our calculations.

### Parallelism strategy

Incrementing velocity and position tables is trivially done just by assigning each element or set of elements to a thread. Each element

needs to be loaded only once so shared memory isn't useful here. However by using "structure of arrays" for our tables we can coalesce memory reads and writes.

As for calculating the forces, we do need shared memory. The strategy is for each thread to find the new acceleration vector for its index ( $a_i$ ). The position table and mass table should be tiled and loaded into shared memory dynamically so that the memory bottle neck may be hidden.

Alternatively, the load per thread could be split between blocks and reduced to increase occupancy but that requires testing.

## **Double computation**

Newton's third law states for each action is an equal and opposite reaction. This would be pretty important for a serial implementation as it would reduce force vector calculations by half. However in a parallel version we aren't constrained by processing power instead we are limited by memory latency. Additionally, adding the force for both objects would require memory protection or reduction which would have too much overhead.

## **Initial conditions**

Initial conditions include location and velocity tables. There could also be a utility to save current progress and use it as the initial conditions later.

## **User Interface**

To save on development time and keep the project somewhat platform independent, the application will start as a console app accepting the file with the initial conditions and taking user input. When simulation starts a window will open which will render the simulation and allow the user to move the camera and see what's happening.

## **Rendering technique**

As said before, the rendering engine and physics engine if run on the same device may share memory. This means after updating the position table, it alongside the mass table may be copied to another gpu block in the “array of structures” format which can be used by OpenGL in vertex shader computation.

This means the CPU only needs to keep the initial conditions on start up and may free them during simulation.

## **Collision detection**

If two bodies are too close to each other they get flagged as colliding and this will be done during acceleration calculation. This will be handled later by either the CPU or another kernel.

## **Collision handling**

When two bodies collide they become a new body with the combined mass of the two and velocity proportional to the sum of momentum.

## **Performance goal**

I wish to simulate around 1000 bodies targeting 60 frames per second on my device (2060 super) . Rendering the bodies themselves will be done bare minimum and all the objects will be using the same texture just at the correct location and scaled based on mass. The setup will scale linearly until the GPU saturates. After that it'll go back to being an  $O(n^2)$  problem.

## **Additional functionality**

This simulation engine could also be extended to support dynamic timescale and dynamic object addition by the user. This won't be covered in the project but it's definitely something to consider for a future project.