Содержание

Гравитационная задача N тел

Гравитационная задача N тел является классической проблемой небесной механики и гравитационной динамики Ньютона.
Она формулируется следующим образом:
В пустоте находится N материальных точек, массы которых известны $\{m_i\}$. Пусть попарное взаимодействие точек подчинено закону тяготения Ньютона, и пусть силы гравитации аддитивны. Пусть известны начальные на момент времени $t = 0$ положения и скорости каждой точки $\mathbf{r}_i|_{t=0} = \mathbf{v}_{i0},\ \mathbf{v}_i|_{t=0} = \mathbf{v}_{i0}$. Требуется найти положения точек для всех последующих моментов времени.

Числительные методы

Мы используем гравитационный потенциал для иллюстрации основной формы вычислений в моделировании всех пар N-тел. В следующих вычислениях мы используем жирный шрифт для обозначения векторов (обычно в 3D). Учитывая N тел с начальным положением $\mathbf{x}_i$ и скоростью $\mathbf{v}_i$ для $1 \leqslant i \leqslant N$, вектор силы $\mathbf{f}_{ij}$ на теле i, вызванной его гравитационным притяжением к телу j, задается следующим образом:

$$\mathbf{f}_{ij} = G\frac{m_i m_j}{||\mathbf{r}||^2} \cdot \frac{\mathbf{r}_{ij}}{||\mathbf{r}_{ij}||}$$

где m_i и m_j — массы тел i и j, соответственно; $\mathbf{r}_{ij} = \mathbf{x}_j - \mathbf{x}_i$ — вектор от тела i к телу j ; G — гравитационная постоянная. Левый фактор — величина силы, пропорционален произведению масс и уменьшается с квадратом расстояния между телами i и j. Правый фактор — направление силы,

единичный вектор от тела i в направлении тела j (поскольку гравитация является притягивающей силой).

Полная сила $\mathbf{F}_i$, действующая на тело i в результате его взаимодействия с другими $N-1$ телами, получается суммированием всех взаимодействий:

$$\mathbf{F}_i = \sum_{\substack{1 \leqslant j \leqslant N \\ j \neq i}} \mathbf{f}_i = G m_i \sum_{\substack{1 \leqslant j \leqslant N \\ j \neq i}} \frac{m_j \mathbf{r}_{ij}}{||\mathbf{r}_{ij}||^3}$$

По мере приближения тел друг к другу сила между ними неограниченно растет, что является нежелательной ситуацией для численного интегрирования. В астрофизических симуляциях столкновения между телами обычно исключаются; это разумно, если тела представляют собой галактики, которые могут проходить прямо друг через друга. Поэтому добавляется коэффициент смягчения $\varepsilon^2 > 0$, и знаменатель переписывается следующим образом:

$$\mathbf{F}_i \approx G m_i \sum_{1 \leqslant j \leqslant N} \frac{m_j \mathbf{r}_{ij}}{\left(||\mathbf{r}_{ij}||^2 + \varepsilon^2\right)^{\frac{3}{2}}}$$

Обратите внимание, что условие $j \neq i$ больше не нужно в сумме, потому что $\mathbf{f}_{ii} = 0$, когда $\varepsilon^2 > 0$. Фактор смягчения моделирует взаимодействие между двумя точечными массами Пламмера: массами, которые ведут себя так, как если бы они были сферическими галактиками (Aarseth 2003, Dyer and Ip 1993). По сути, фактор смягчения ограничивает величину силы между телами, что желательно для численного интегрирования состояния системы.

Для интегрирования по времени нам необходимо ускорение $\mathbf{a}_i = \dfrac{\mathbf{F}_i}{m_i}$ для обновления положения и скорости тела i, поэтому мы упрощаем

вычисления до этого:

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

Предлагается такой код (синтакс *glsl*):

```
vec3 interact(vec3 a, vec3 b, float m) {
  vec3 r = a - b;
  float ds = dot(r, r) + eps2;
  return r * m * inversesqrt(ds * ds * ds);
}
```

Интегратор, используемый для обновления положений и скоростей, является интегратором скачка *Verlet*, поскольку он применим к данной задаче и является вычислительно эффективным (имеет высокое отношение точности к вычислительным затратам). Выбор метода интегрирования в задачах N-тел обычно зависит от природы изучаемой системы. Интегратор включен в наши тайминги, но обсуждение его реализации опущено, поскольку его сложность составляет $O(N)$ и он не влияет на общий производительность.

Интегратор Verlet

Интегрирование *Verlet* — это численный метод, используемый для интегрирования уравнений движения Ньютона. Он часто используется для расчета траекторий частиц при моделировании молекулярной динамики и в компьютерной графике.

Интегратор *Verlet* обеспечивает хорошую численную устойчивость, а также другие свойства, важные для физических систем, такие как обратимость во времени и сохранение симплектической формы на фазовом пространстве, без существенных дополнительных вычислительных затрат по сравнению с простым методом Эйлера.
Если метод Эйлера использует прямое разностное приближение к первой производной в дифференциальных уравнениях первого порядка, то интегрирование Verlet можно рассматривать как использование центрального разностного приближения ко второй производной:

Интеграция *Verlet*

$$\begin{aligned}\frac{\Delta^2 \mathbf{x}_n}{\Delta^2 t} &= \frac{\frac{\mathbf{x}_{n+1}-\mathbf{x}_n}{\Delta t} - \frac{\mathbf{x}_n-\mathbf{x}_{n-1}}{\Delta t}}{\Delta t} \\ &= \frac{\mathbf{x}_{n+1} - 2\mathbf{x}_n + \mathbf{x}_{n-1}}{\Delta^2 t} = \mathbf{a}_n = \mathbf{A}(\mathbf{x}_n)\end{aligned}$$

$$\mathbf{x}_{n+1} = 2\mathbf{x}_n - \mathbf{x}_{n-1} + \mathbf{a}_n \Delta^2 t$$

Интеграция скоростей *Verlet*

$$\begin{aligned}\mathbf{x}_{n+1} &= \mathbf{x}_n + \mathbf{v}_n \Delta t + \frac{1}{2}\mathbf{a}_n \Delta t^2 \\ \mathbf{v}_{n+1} &= \mathbf{v}_n + \frac{\mathbf{a}_{n+1} + \mathbf{a}_n}{2}\Delta t\end{aligned}$$

Картинка (1) показывает как интеграция *Verlet* имеет гораздо меншую расходимость от изначальной функции, нежели чем методом Ейлера.

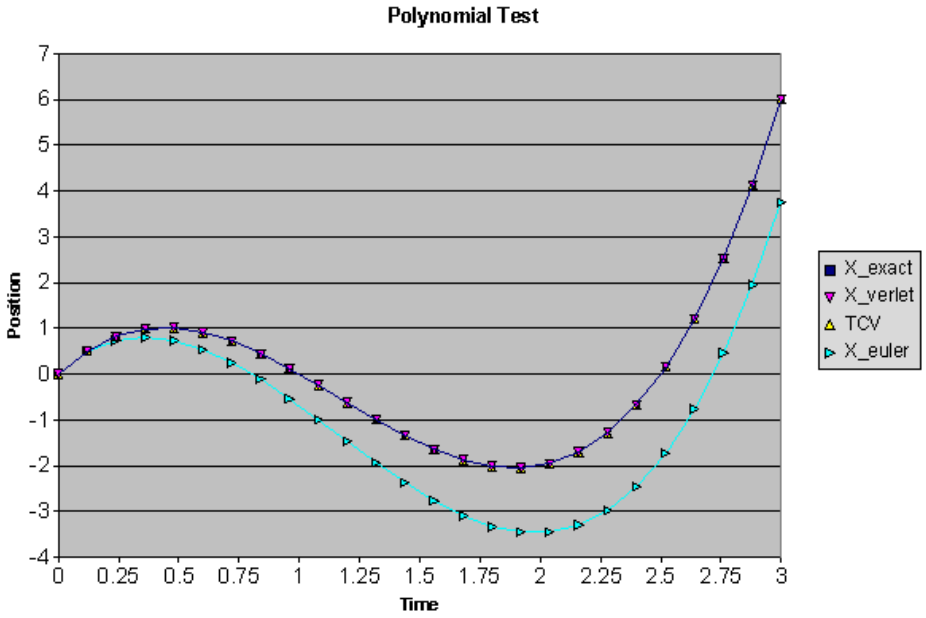


Рис. 1: Сравнение

Предлагается такой код (синтакс *glsl*):

```
void calc_particle(int i, float dt) {
  pos[i] += vel[i] * dt + 0.5 * acc[i] * dt * dt;
  // create the tree, based on the new positions
  vec3 new_acc = calc_accelleration(pos[i]);
  vel[i] += 0.5 * (new_acc + acc[i]) * dt;
  acc[i] = new_acc;
}
```

Алгоритм Barnes-Hut

Основы алгоритма

Важнейшей идеей ускорения алгоритма грубой силы для n тел является группировка близлежащих тел и аппроксимация их как одного единого

тела.
Если группа находится достаточно далеко, мы можем аппроксимировать её гравитационное воздействие, используя её центр масс.
Центр масс группы тел — это среднее положение тела в этой группе, взвешенное по массе.
Формально, если два тела имеют положения $\mathbf{x}_1$ и $\mathbf{x}_2$, массы m_1 и m_2, то их общая масса и центр масс $\mathbf{x}$ определяются следующим образом:

$$m = m_1 + m_2$$
$$\overline{\mathbf{x}} = (m_1\mathbf{x}_1 + m_2\mathbf{x}_2)/m$$

Для n тел:

$$m = \sum_{i=1}^{n} m_i$$
$$\overline{\mathbf{x}} = \frac{1}{m}\sum_{i=1}^{n} m_i\mathbf{x}_i$$

Для вычисления центр масс не листового узла, мы использоваем центр масс дочерных узлов, чтобы упростить расчёты, не пренебрегая точности.

Алгоритм *Barnes-Hut* — это умная схема для группировки тел, которые находятся достаточно близко друг к другу.
Он рекурсивно делит множество тел на группы, сохраняя их в пространственном дереве. В двухмерном пространстве используется *Quadtree*, а в трёхмерном *Octree*. Они похожи на бинарное дерево, за исключением того, что каждый узел имеет 4/8 дочерных узлов (некоторые из которых могут быть пустыми).
Каждый узел представляет собой область квадрата/куба.

В нашей иплементации не выделяется память для пустых потомков.

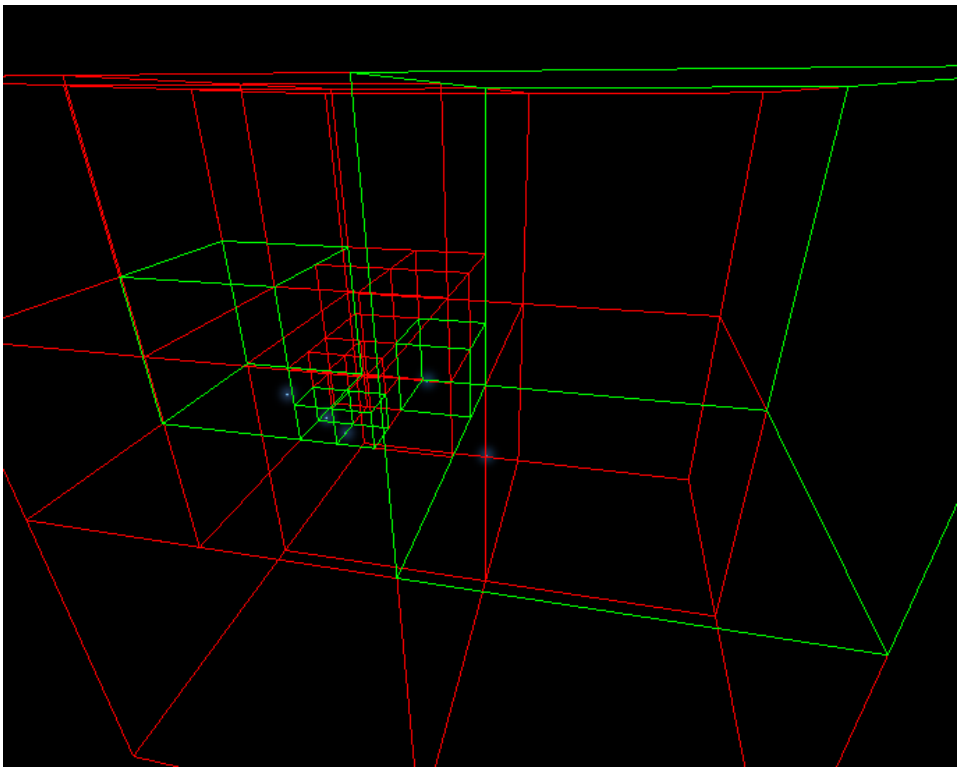


Рис. 2: *Octree*

Вычисления силы действущий на тело

Для вычисления силы, действующей на тело b, используют следующую рекурсивную процедуру, начиная с корня:

A) Если текущий узел является внешним узлом (и это не тело b), вычисляют силу, действующую на b со стороны текущего узла, и добавляют эту величину к силе b.

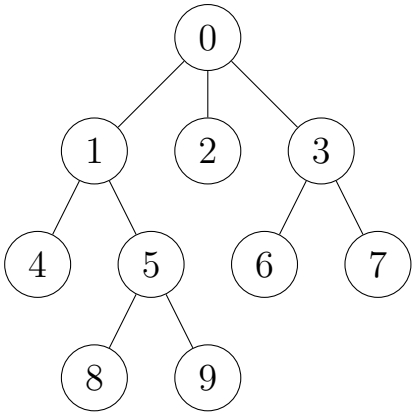
B) В противном случае вычисляют отношение s/d. Если $s/d < \theta$, рассматривают этот внутренний узел как единое тело, вычисляют силу, которую он оказывает на тело b, и добавляют эту величину к силе b.

C) В противном случае выполняют процедуру рекурсивно для каждого

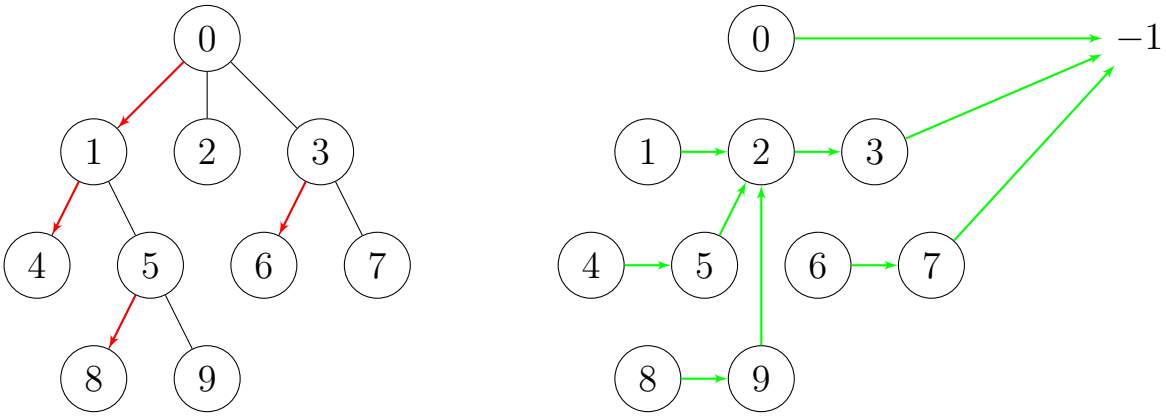
из дочерних узлов текущего узла.

Наша имплементация алгоритма на GPU

Поскольку GPU не поддерживает рекурсию и Стек, надо изменить алгоритм с рекурсионного к итеративному. Мы можем возпользоваться CPU для построения кое-каких функциональных графов (естесвенно упакованных в виде массивов, как и всеобщие данные на GPU) и перемещаться по дереву с помощью их. Представим такое дерево:



На каждом этапе мы проверяем, если не выполнено условие $s/d < \theta$, тогда мы двигаемся вниз по дереву, иначе останавливаемся, вычисляем a_i и двигаемся дальше (к братьям или к братьям излов предков).
Для этого мы использоваем один массив для перемещения к первому дочернему узлу (3a) и другой для продолжения к братьям (своим или предков) (3b).
В обоих случаях несуществование узла обозначается со значением -1.



(a) `first_child` (b) `next`

Можно тогда выразить алгоритм, как простой *while loop* — предлагается такой код (синтакс *glsl*):

```
vec3 calc_accelleration(vec3 pos) {
  int i = 0;
  vec3 acc = vec3(0, 0, 0);
  while (i != -1) {
    vec3 com = center_of_mass[i].xyz;
    float mass = center_of_mass[i].w;
    float s = sizes[i];
    int fc = first_child[i];
    vec3 d = com - pos;
    bool cond = fc == -1 || th2 > (s * s / dot(d, d));
    if (cond) acc += interact(com, pos, mass);
    i = cond ? next[i] : fc;
  }
  return acc;
}
```

Расхождение потоков и кэширование

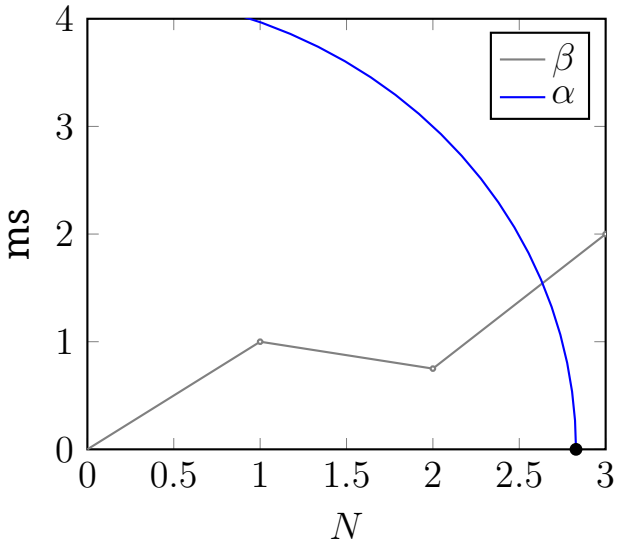
Расхождение потоков

Поскольку потоки, принадлежащие к одному и тому же *warp*, выполняются в *lockstep*, каждому *warp* фактически приходится ждать прохождение префиксов дерева всех потоков в *warp*. Другими словами, всякий раз, когда происходит какая-то развилка, которая не нужна некоторым из потоков, эти потоки из-за расхождения замораживаются, пока эта часть будет пройдена.
Как следствие, крайне важно группировать пространственно близкие тела вместе, чтобы потоки проходили схожие префиксы дерева, то есть, чтобы расхождение префиксов было как можно меньше. В нашей имплемнтации сортирования ускоряет алгоритм в 50%.

Кэширование

Каждая итерация в дереве нуждается многими и различными чтениями из главной памяти GPU с каждого из потока.
Зная, что общая память потоков на несколько порядков быстрее, чем основная память, и что префиксы повторяются, мы можем заставить один поток кэшировать данные дерева и другие потоки читать с кэша.
Есть один нюанс: быстрая память гораздо меньше, чем основная, для этого мы будем использовать *hash table* где префикс `p = prefix % TABLE_SIZE` (конечно выполняется дополнительная проверка столкновения хэшей).
К сожалению, наша имплементация не принесла значительных улучшений.

Производительность



Недостатки и будущие оптимизации

По текущей имплементации, создание дерева просходит на CPU и передаётся на GPU, при каждом шаге симуляции.
После компьютации новых положений частиц, новые точки передаются обратно на CPU для создания нового дерева.
Есть алгоритмы, которые создают дерево на GPU, обходя постоянный обмен данных между CPU и GPU.
С другой стороны, создание дерева с нуля каждый раз неизбежно, и алгоритмов, позволяющих использовать старое дерево для создания нового, нет. Имея это в виду, это не особо влияет на скорость программы.
Сейчас алгоритм выполняется в среднем $O(N \log N)$, при использовании более продвинутого алгоритма FMM (*Fast Multipole Method*) можно достичь сложность $O(N)$.
Не используется эффективно возможности кэшевой памяти. В дальнейшем, предусматривается использование *Vulkan* API, вместо *OpenGL*, для

более низкого упраления GPU.

Список литературы

[1] Гравитационная задача n тел, 2022. *https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D0%B2%D0%B8%D1%82%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%B7%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_N_%D1%82%D0%B5%D0%BB*.

[2] Verlet integration, 2023. *https://en.wikipedia.org/wiki/Verlet_integration*.

[3] Jan Prins Lars Nyland, Mark Harris. Fast n-body simulation with cuda, 2006. *https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda*.

[4] Keshav Pingali Martin Burtscher. An efficient cuda implementation of the tree-based barnes hut n-body algorithm, 2022. *https://iss.oden.utexas.edu/Publications/Papers/burtscher11.pdf*.

[5] Salel. nbody, 2015. *https://github.com/salel/nbody/*.

[6] Kevin Wayne Tom Ventimiglia. The barnes-hut algorithm, 2022. *http://arborjs.org/docs/barnes-hut*.