Het n-body probleem

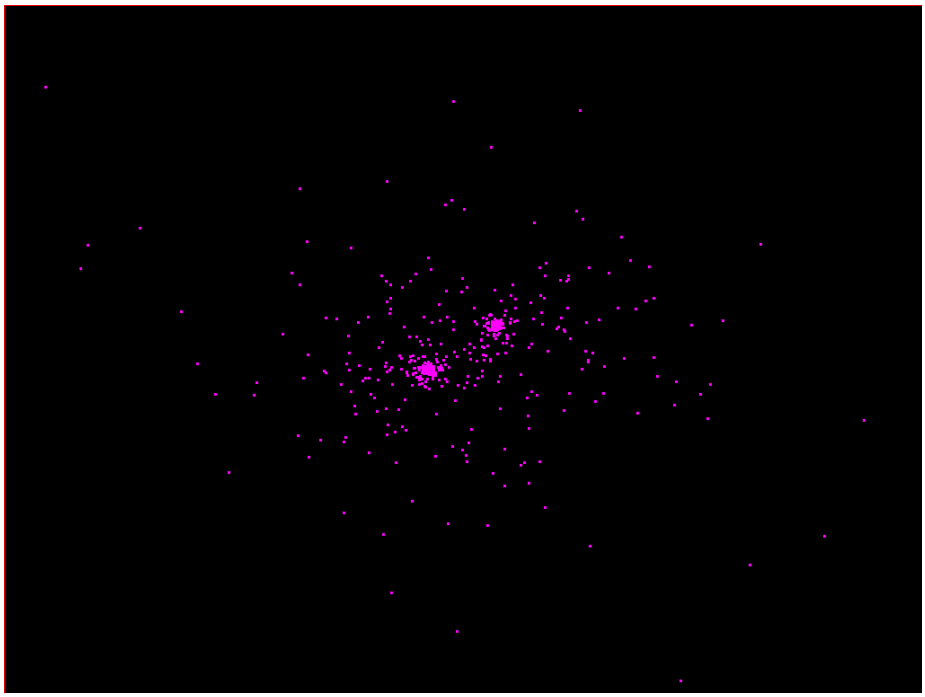

Figure 1: Screenshot van het programma.

Het n-body probleem bestaat er uit de relatieve positie en snelheid van verschillende hemellichamen t.o.v. elkaar te voorspellen. Elk lichaam voert een bepaalde aantrekkingskracht uit op elk ander lichaam.

Een naïeve versie van dit probleem kunnen we dan ook op deze manier implementeren: voor elk lichaam berekenen we de zwaartekracht die elk ander lichaam uitvoert op het lichaam. De zwaartekracht tussen twee lichamen kan met de volgende formule berekend worden:

$$\vec{F}_{ij} = \frac{G m_i m_j (\vec{p}_j - \vec{p}_i)}{\|\vec{p}_j - \vec{p}_i\|^3}$$

Waar G de zwaartekracht constante is, m_i, m_j de massa van beide lichamen, p_i en p_j de positie van de lichamen en $\|p_j - p_i\|$ de euclidische afstand tussen de twee lichamen. Vanuit deze kracht F_{ij} kunnen we dan dmv. $F = ma$ de acceleratie berekenen.

$$\vec{a}_{ij} = \frac{\vec{F}_{ij}}{m_i}$$

De totale acceleraties op een hemellichaam bekomen we dan door alle acceleraties op dit lichaam op te tellen:

$$\vec{a}_i = \sum_{i \neq j}^{n} \vec{a}_{ij}$$

De simulatie wordt uitgevoerd in achtereenvolgende stappen. In elke stap berekenen we telkens voor elk lichaam de acceleratie, de nieuwe snelheid, en de nieuwe positie. De snelheid en de positie moeten we afzonderlijk bijhouden in een buffer, de acceleratie kunnen we berekenen aan de hand van de positie.

Het berekenen van nieuwe posities van lichamen kan bv. met volgende C-code gebeuren:

```
for (int i = 0; i < length; ++i) {
    for (int j = 0; j < length; ++j) {
        if (i == j)
            continue;

        cl_float3 pos_a = host_pos[i];
        cl_float3 pos_b = host_pos[j];

        float dist_x = (pos_a.s[0] - pos_b.s[0]) * distance_to_nearest_star;
        float dist_y = (pos_a.s[1] - pos_b.s[1]) * distance_to_nearest_star;
        float dist_z = (pos_a.s[2] - pos_b.s[2]) * distance_to_nearest_star;

        float distance = sqrt(dist_x * dist_x + dist_y * dist_y + dist_z * dist_z);

        float force_x = -mass_grav * dist_x / (distance * distance * distance);
        float force_y = -mass_grav * dist_y / (distance * distance * distance);
        float force_z = -mass_grav * dist_z / (distance * distance * distance);

        float acc_x = force_x / mass_of_sun;
        float acc_y = force_y / mass_of_sun;
        float acc_z = force_z / mass_of_sun;

        host_speed[i].s[0] += acc_x * delta_time;
        host_speed[i].s[1] += acc_y * delta_time;
        host_speed[i].s[2] += acc_z * delta_time;
    }
}
for (int i = 0; i < length; ++i) {
    host_pos[i].s[0] += (host_speed[i].s[0] * delta_time)
            / distance_to_nearest_star;
    host_pos[i].s[1] += (host_speed[i].s[1] * delta_time)
            / distance_to_nearest_star;
```

```
    host_pos[i].s[2] += (host_speed[i].s[2] * delta_time)
            / distance_to_nearest_star;
}
```

Opdracht

De code C-code van dit programma kan je hier downloaden:

`http://telescript.denayer.wenk.be/~wvr/n-body.tgz`

Versnellen simulatie op GPU

Schrijf een OpenCL programma dat de gegeven C-code versnelt op GPU. Probeer hiervoor verschillende werkwijzen te volgen:

- Bestudeer welke data-afhankelijkheden er zijn in de gegeven C-code, en bestudeer hoe je het gegeven programma het beste kan vertalen naar OpenCL code. Evalueer het resultaat.
- OpenCL heeft ingebouwde vector types (bv. `float3`) en intrinsieke functies om vele wiskundige operaties sneller te laten gaan. Ga na hoe deze in het programma gebruikt kunnen worden, en wat de invloed is op de rekentijd.

Hint: Atomische operaties in OpenCL 1.1

Bij het berekenen van de snelheid kan je eventueel atomische operaties gebruiken (dit is niet noodzakelijk de meest efficiënte manier). Een atomische operatie op een float type zit niet in OpenCL, je kan het wel als volgt implementeren door gebruik te maken van de atomic_cmpxchg operatie (`https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/atomic_cmpxchg.html`).

Atomische operaties op vectors zijn ook niet standaard ondersteund in OpenCL, maar je kan ook dit omzeilen door het type van de buffer te wijzigen naar volgende union:

```
typedef union
{
    float3 vec;
    float arr[3];
} float3_;
```

Voor vector operaties gebruik je dan het “vec” veldje en voor de atomische operaties het arr veldje.

In de gedownloade tarball is hier ook een voorbeeld van te vinden (`atomic_add.cl`).

Berekenen van statistieken

Naast het versnellen van de simulatie is het ook leuk enkele statistieken te kennen. Bereken (in OpenCL) de gemiddelde afstand, de maximum afstand en de minimum afstand tussen verschillende lichamen. Welke methoden presteren hier het beste? (Hint: kijk eens naar de reductie-algorithmes die je gezien hebt in de les).

Verslag en demonstratie

Op **vrijdag 25 mei** zal je de werking van je programma moeten demonstreren en toelichten.

Ten laatste op **maandag 14 mei** moet je een verslag van je werk indienen. In dit verslag moet je volgende elementen opnemen:

- Bespreek de verschillende varianten van je programma die je hebt uitgeprobeerd
- Toon je experimentele resultaten (bv. in de vorm van een aantal grafieken)
- Bespreek je experimentele resultaten: hoe kan je de geobserveerde fenomenen verklaren en welke conclusies trek je hieruit?

Als bijlage bij dit verslag moet je **ook je code** indienen.