Exercício Computacional – N-Corpos

Informações

Ambiente de Desenvolvimento
Code::Blocks 16.01

Linguagem
C++

Compilador
gcc (tdm-1) 4.9.2

Bibliotecas
FreeGLUT(OpenGL) e algumas bibliotecas padrão do C++(input,output, etc)

Material Utilizado

- Entendimento/Contexto do Algoritmo

Slides “Subdivisão hierárquica de regiões” – Rudnei
https://en.wikipedia.org/wiki/Barnes–Hut_simulation

- Detalhes Técnicos do Algoritmo

https://jheer.github.io/barnes-hut/

Estruturas

quadtree

```
class quadtree{
public:
    //Função construtora
    quadtree(std::vector <particula> plist,float iwidth);
    quadtree(float iwidth);
    quadtree(quadtree *pai);
    void inserepart(particula p);
    void subdivide();
    bool contains(particula pt);

    quadtree* raiz;
    quadtree* nw;
    quadtree* ne;
    quadtree* sw;
    quadtree* se;
    //aabb caixa;
    float massatot=0.0f;
    Vec2 cent;
    float hwidth=0.0f;
    Vec2 centmassa;
    std::vector<particula> ps;
    bool folha=true;
    //variavel de depuração
    int bleeps=0;

private:

};
```

Pelo código, podemos observar que é um quadtree modificado. Há variáveis extras como "massatot" e "centmassa", variáveis utilizadas na determinação das massas cumulativas dos nodos filhos e assim como o centro de massa.

Há 2 formas de inicialização (uma não é utilizada), podemos inicializar dando um conjunto de partículas e "iwidth/hwidth", essa última variável é a metade do lado do nosso quadtree que no caso é um quadrado. Essa inicialização é a mais utilizada no código, é usada tanto na criação inicial do quadtree quanto a recriação para cada atualização ou etapa de tempo.

```
quadtree::quadtree(std::vector<particula> plist,float iwidth){
    //std::cout<<ps.size();
    hwidth=iwidth;
    cent.u[0]=hwidth; cent.u[1]=hwidth;
    for(int i=0;i<plist.size();i++){
        //Ignorar nodos que ultrapassam os limites espaciais da árvore
        if(plist.at(i).x.u[0]<0 || plist.at(i).x.u[0]>hwidth*2 || plist.at(i).x.u[1]<0 || plist.at(i).x.u[1]>hwidth*2){
            continue;
        }
        inserepart(plist.at(i));
    }
}
```

Inicializamos a posição espacial do nodo, iteramos para cada partícula dentro do nosso conjunto para verificar se elas estão no nosso plano computacional, se não estão, as partículas são deletadas. **OBS:** Em vez de deletar podemos simplesmente mover elas para o nosso plano computacional, mas decidi removê-las quando ultrapassarem os limites definidos pelo nodo raiz.

Repare, aqui já temos um gasto $O(n)$ se ignorarmos a função "inserepart". Vamos descrever essa função com mais detalhe.

```
//Inserindo e calculando centros de massas
void quadtree::inserepart(particula p){
    if(ps.size()>0){
        if(folha){
            subdivide();
        }
        if(p.x.u[0]<=cent.u[0]){
            if(p.x.u[1]<=cent.u[1]){
                sw->inserepart(p);
            }else{
                nw->inserepart(p);
            }
        }else{
            if(p.x.u[1]<=cent.u[1]){
                se->inserepart(p);
            }else{
                ne->inserepart(p);
            }
        }
        ps.push_back(p);
        centmassa.u[0]=(centmassa.u[0]*massatot+p.x.u[0]*p.mass)/(massatot+p.mass);
        centmassa.u[1]=(centmassa.u[1]*massatot+p.x.u[1]*p.mass)/(massatot+p.mass);

        massatot=massatot+p.mass;
    }else{
        massatot=p.mass;
        centmassa=p.x;
        ps.push_back(p);
    }
}
```

Dado uma partícula, inserimos a partícula no nodo do nosso quadtree. Primeiramente verificamos se o conjunto de pontos do nodo é maior do que 0 e depois se é um nodo folha, se for, significa que precisamos subdivir em quadrantes ou, se já está subdivido, inserir no quadrante apropriado utilizando 2 comparações(pense no algoritmo de busca em uma árvore binária). Também atualizamos os centros de massas e massas totais do nodo. Quando não há nenhum ponto no nodo, já sabemos que é um nodo folha e assim não precisamos inserir em quadrante, no caso, o centro de massa e massa é o mesmo da partícula.

O custo da inserção acima é $O(\log n)$, típico de uma árvore binária de busca. Já podemos determinar o custo resultante da inicialização/atualização do nosso quadtree: $O(n \log n)$. Ou seja, para cada etapa de tempo, depois de integrarmos o movimento de todas as partículas, sempre teremos esse custo. **OBS:** Podemos melhorar o custo total sabendo que em um dt pequeno, muitas partículas ainda vão estar situadas nos seus nodos quadtree, não há necessidade de atualizar e assim não precisamos fazer para cada etapa de tempo. Resolvi não implementar o método acima para não aumentar ainda mais o erro da simulação por quadtrees.

Ignoramos uma função, “subdivide”, talvez ela pode possuir um custo às escondidas.

```
//Subdivisão da árvore
void quadtree::subdivide(void){
    std::vector<particula> tmp;
    nw = new quadtree(this);
    nw->cent.u[0]=cent.u[0]-hwidth/2;
    nw->cent.u[1]=cent.u[1]+hwidth/2;
    nw->bleeps = bleeps+1;
    //std::cout<<nw->bleeps<<std::endl;

    ne = new quadtree(this);
    ne->cent.u[0]=cent.u[0]+hwidth/2;
    ne->cent.u[1]=cent.u[1]+hwidth/2;
    ne->bleeps = bleeps+1;

    sw = new quadtree(this);
    sw->cent.u[0]=cent.u[0]-hwidth/2;
    sw->cent.u[1]=cent.u[1]-hwidth/2;
    sw->bleeps = bleeps+1;

    se = new quadtree(this);
    se->cent.u[0]=cent.u[0]+hwidth/2;
    se->cent.u[1]=cent.u[1]-hwidth/2;
    se->bleeps = bleeps+1;

    tmp = ps;
    ps.clear();
```

Aqui usamos a convenção: nw – nodo noroeste, ne – nodo nordeste, sw – nodo sudoeste, se – nodo sudeste. Nessa função utilizamos a segunda inicialização, a inicialização quadtree(quadtree* pai).

```
//Inicializando árvore com um nodo pai
quadtree::quadtree(quadtree *pai){
    raiz = pai;
    hwidth=pai->hwidth/2;

}
```

Na função subdivide também definimos as posições dos centros geométricos dos nodos usando uma relação simples e a nossa convenção descrita acima assim como clonar a lista de partículas do nodo e depois zerar a lista original.

```
    //reinserir pontos que já estavam na árvore caso tenha mais de um
    for(int i=0;i<tmp.size();i++){
        if(tmp.at(i).x.u[0]<=cent.u[0]){
            if(tmp.at(i).x.u[1]<=cent.u[1]){
                sw->inserepart(tmp.at(i));
            }else{
                nw->inserepart(tmp.at(i));
            }
        }else{
            if(tmp.at(i).x.u[1]<=cent.u[1]){
                se->inserepart(tmp.at(i));
            }else{
                ne->inserepart(tmp.at(i));
            }
        }
        ps.push_back(tmp.at(i));
    }
    folha = false;
}
```

Vamos lembrar quando a função “subdivide” é chamada, ela é chamada quando a lista de partículas do nodo não for vazia e for um nodo folha(não houve subdivisão). Repare, o nodo que

estamos subdividindo não necessariamente possui somente uma partícula por ser um nodo folha, caso exista mais de uma partícula em um nodo folha, precisamos reorganizar tais partículas para que eles se situem nos quadrantes apropriados. Isso explica o laço, mas na minha implementação eu resolvi fazer com que, um nodo sendo um nodo folha, possui no máximo uma partícula e assim fazemos a pergunta: Será que esse laço cria um custo extra adicional? Sim, mas o custo é desprezível, se for só uma partícula por nodo folha, teremos $O(2)$ comparações, ou simplesmente complexidade constante. **OBS:** Eu deixei esse laço caso eu queira deixar mais de uma partícula por nodo, mas acabei não seguindo essa linha.

Com Isso terminamos o "grosso" da estrutura "quadtree", há funções que não abordamos como "contains", utilizada para ver se uma partícula está dentro de um nodo quadtree.

particulas

```
class particula{
public:
    //Função construtora
    particula(Vec2 xi, Vec2 vi, float massp);

    //"Getters"
    /*
    void pegaPos(float (&xr)[2]);
    Vec2 pegaPos();
    void pegaVel(float (&vr)[2]);
    Vec2 pegaVel();
    void pegaMass(float &ms);
    float pegaMass();
    */
    float mass=0.0f;
    Vec2 x;
    Vec2 v;
    GLfloat cor[3];

    //Aplicação de impulsos/forças em um tempo dt
    void aplicafor(Vec2 force,float dt);
    /*
private:
    //Variáveis
    float mass;
    Vec2 x;
    Vec2 v;
    */
};
```

A estrutura "particula" é bem simples, temos uma função construtora que inicializa a posição inicial, velocidade inicial e a massa da partícula. Na estrutura partícula também temos um arranjo de 3 "GLfloat/float" usada na renderização das partículas – cor(R,G,B). A única função desta estrutura é "aplicafor", a função que integra o movimento em um tempo dt usando o método de Euler.

```
void particula::aplicafor(Vec2 force,float dt){
    //std::cout<<"("<<force.u[0]<<","<<force.u[1]<<")"<<std::endl;
    for(int i=0;i<2; i++){
        v.u[i] = v.u[i] + force.u[i] * dt/mass;
        x.u[i] = x.u[i] + v.u[i]*dt;
    }
}
```

Em partículas.cpp também temos uma subestrutura chamada Vec2.

```
struct Vec2{
    float u[2]={0,0};
};

float dotProd(Vec2 a,Vec2 b);
float magVec(Vec2 a);
float distVec(Vec2 a,Vec2 b);
Vec2 dirVec(Vec2 to, Vec2 from);
Vec2 difVec(Vec2 a,Vec2 b);
Vec2 normalize(Vec2 a);
float clamp(float x,float lo,float hi);
```

OBS: Revisando o código e podíamos ter feito ele inteiro sem a necessidade desse Vec2, a introdução dele até torna o código um pouco ilegível.

Barneshut

Por último, chegamos na estrutura “mestre”. Esta estrutura utiliza a nossa estrutura “quadtree” e “partícula”.

```
class barneshut{
public:
    //Função construtora
    barneshut(std::vector<particula> ps,float iwidth);

    void atualiza(float theta,float dt);
    std::vector<quadtree> pegaForcas(particula pt,float theta,quadtree* nd);

    quadtree* pai;
    float time=0.0f;

};
```

A estrutura armazena somente 2 dados, um ponteiro do nodo raiz do quadtree e o tempo percorrido na simulação (não foi utilizado em nenhum ponto no código).

```
//Constante gravitacional com um valor grande para não ter que criar partículas de massas planetárias
float G = 1e6;
//Epsilon para evitar divisões por 0
float eps=1e-2;

//o sistema Barnes-Hut é inicializado por uma árvore
barneshut::barneshut(std::vector<particula> ps,float iwidth){
    pai = new quadtree(ps,iwidth);
}
```

Em barneshut.cpp também definimos a constante gravitacional e um épsilon para não acontecer erros de divisão por 0. Na inicialização, barneshut simplesmente chama o construtor de quadtree passando a lista de partículas e a nossa dimensão espacial.

Entramos agora para a função que é sempre chamada para cada etapa de tempo e assim a mais custosa.

```
//Loop central para do sistema físico
//O(n*f(n,theta))
//se theta=0.0, O(n^2)
//se theta=1.0, ~O(n*log(n))
void barneshut::atualiza(float theta,float dt){
    std::vector<quadtree> corpos;
    float force;
    Vec2 forceres,dir;
    for(int i=0;i<pai->ps.size();i++){
        forceres.u[0] = 0.0f; forceres.u[1] = 0.0f;
        //std::cout<<i<<":"<<std::endl;
        particula pt = pai->ps.at(i);
        corpos = pegaForcas(pt,theta,pai);
        //std::cout<<corpos.size()<<std::endl;
        for(int j=0;j<corpos.size();j++){
            quadtree corpo = corpos.at(j);
            force = (G*pt.mass*corpo.massatot)/pow(distVec(corpo.centmassa,pt.x)+eps,2);
            if(std::isnan(force) || force>1000){
                force = 1000;
            }
            dir = dirVec(corpo.centmassa,pt.x);
            forceres.u[0] = forceres.u[0]+dir.u[0]*force;
            forceres.u[1] = forceres.u[1]+dir.u[1]*force;
            //std::cout<<corpo.centmassa.u[0]<<std::endl;
            //std::cout<<"("<<dir.u[0]<<","<<dir.u[1]<<")"<<std::endl;
            //std::cout<<magVec(forceres)<<std::endl;
        }
        //std::cout<<"("<<pt.x.u[0]<<","<<pt.x.u[1]<<")"<<std::endl;
        //std::cout<<magVec(forceres)<<std::endl;
        pai->ps.at(i).aplicafor(forceres,dt);

        //std::cout<<"("<<pt.x.u[0]<<","<<pt.x.u[1]<<")"<<std::endl;

    }
    pai = new quadtree(pai->ps,pai->hwidth);
}
```

Há dois laços, o laço que itera sobre cada partícula descrita pelo nodo raiz do quadtree $O(n)$ e um laço que itera sobre cada corpo descrita pela função “pegaForcas”, uma função que retorna uma lista de corpos que satisfazem um critério, o critério θ.

Assim teremos o custo total desta forma: $O(n\, f(n,\theta))$, uma função f que depende do número de partículas e uma constante θ. Experimentalmente – **OBS:** não realizei o experimento –, para $\theta = 0.0$, o algoritmo se degenera para um algoritmo de força bruta, já para $\theta = 1.0$, o algoritmo é aproximadamente $O(n \log n)$. Com um θ maior teremos erros maiores, porém ele obtém as forças cada vez mais rápido. Com um $\theta = 10.0$, por exemplo, o único corpo que será avaliado na determinação de forças é o nodo raiz na maior parte.

Nessa função também aplicamos a fórmula da força gravitacional e assim como um limite de força pra evitar acelerações extremamente grandes. Em seguida, aplicamos a função “aplicafor” em cada partícula. Quando terminarmos de aplicar todas as forças nas partículas, recriamos o quadtree.

Para concluir, chegamos na função final, “pegaForcas”. Primeiro vamos entrar em detalhe o que significa o critério θ, por que valores maiores torna o algoritmo mais rápido? Definimos o critério θ deste jeito:

$$\theta = \frac{s}{d}$$

Em que s é o lado do nosso nodo quadtree e d a distância da partícula que queremos determinar suas forças ao centro geométrico do nodo quadtree. Uma partícula que está muito longe de um nodo vai possuir um θ muito pequeno, enquanto que uma partícula que está muito perto, um θ muito grande. Prosseguimos na função “pegaForcas”.

```
//Função que analisa nodos do quadtree que satisfazem um criterio e retornando-os
std::vector<quadtree> barneshut::pegaForcas(particula pt,float theta,quadtree* nd){
    std::vector<quadtree> lista,resto;
    if((nd->ps.empty() == false)){
        if(nd->folha){
            if(!(nd->contains(pt))){
                lista.push_back(*nd);
            }
        }else{
            if((2*(nd->hwidth))/(distVec(pt.x,nd->cent)+eps)<=theta){
                lista.push_back(*nd);
            }else{
                resto = pegaForcas(pt,theta,nd->nw);
                lista.insert(lista.end(),resto.begin(),resto.end());
                resto = pegaForcas(pt,theta,nd->ne);
                lista.insert(lista.end(),resto.begin(),resto.end());
                resto = pegaForcas(pt,theta,nd->sw);
                lista.insert(lista.end(),resto.begin(),resto.end());
                resto = pegaForcas(pt,theta,nd->se);
                lista.insert(lista.end(),resto.begin(),resto.end());
            }
        }
    }
    return lista;
}
```

Dado uma partícula, um θ e um nodo quadtree, obtemos uma lista de nodos que satisfazem a condição: $\frac{s}{d} \leq \theta$ ou o nodo for um nodo folha que não possui a nossa partícula – evita divisões por zero apesar de ter um épsilon prevenindo no código. Caso contrário, $\frac{s}{d} > \theta$, isto é, ele está mais perto, testamos os seus nodos filhos, ou seja, os 4 quadrantes do nodo quadtree e depos unimos tudo e assim retornando a lista final.

Resultados

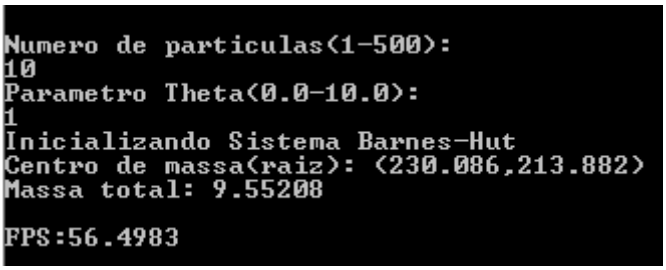
Todas as simulações seguintes foram feitas em um CPU antigo da Intel: Intel i5 da 4ª geração. Em um processador moderno, acredito que os tempos vão ser melhores.

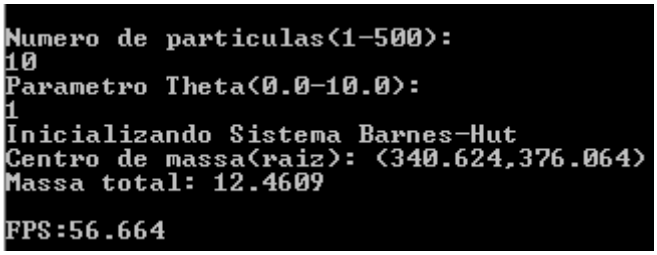
Comparações de Tempo

```
Numero de particulas(1-500):
10
Parametro Theta(0.0-10.0):
0
Inicializando Sistema Barnes-Hut
Centro de massa(raiz): (469.301,397.391)
Massa total: 11.093

FPS:53.7892
FPS:55.7777
```

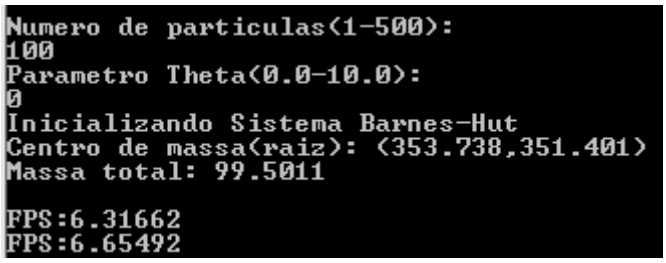
```
Numero de particulas(1-500):
10
Parametro Theta(0.0-10.0):
0
Inicializando Sistema Barnes-Hut
Centro de massa(raiz): (325.567,382.344)
Massa total: 10.9911

FPS:53.6819
FPS:56.1775
```

Testando com 10 partículas e $\theta = 0.0$, obtemos em média 54.85 FPS.

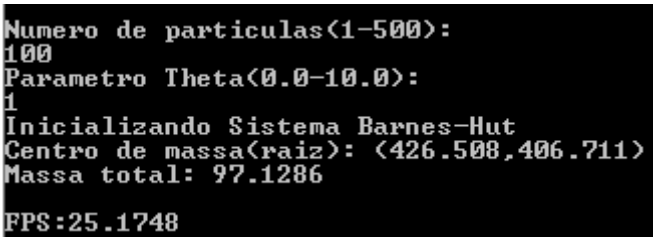




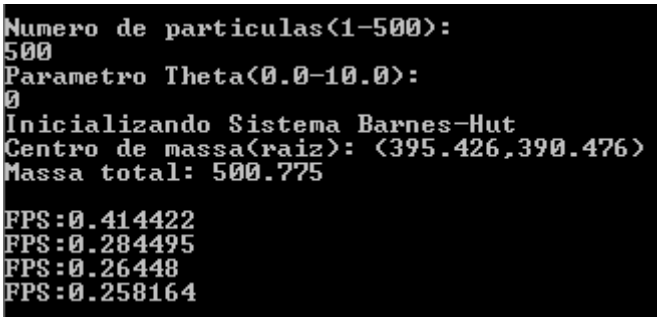
Testando com 10 partículas e $\theta = 1.0$, obtemos em média 56.58 FPS. Não houve uma melhora significante para uma quantidade pequena de partículas. **OBS:** Se eu tivesse implementado um algoritmo de força bruta verdadeiro (não degenerado) pode acontecer, em certos casos, com $\theta = 1.0$ até ser mais lento quando as partículas estiverem muito densas e for uma quantidade pequena. Os custos da recriação da árvore somadas com a determinação das forças sobressaem os custos do algoritmo de força bruta.



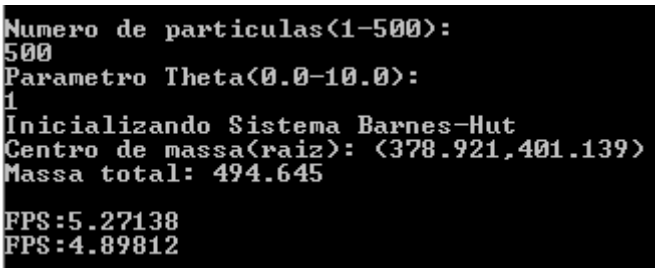
Testando com 100 partículas e $\theta = 0.0$, obtemos em média 6.48 FPS.



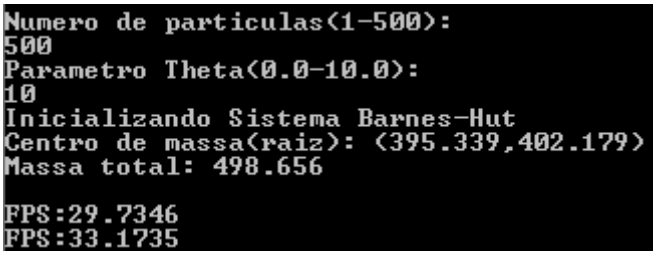
Testando com 100 partículas e $\theta = 1.0$, obtemos em média 25.17 FPS. Houve uma melhora significante, quase 4x mais rápido.



Testando com 500 partículas e $\theta = 0.0$, obtemos em média 0.30 FPS.

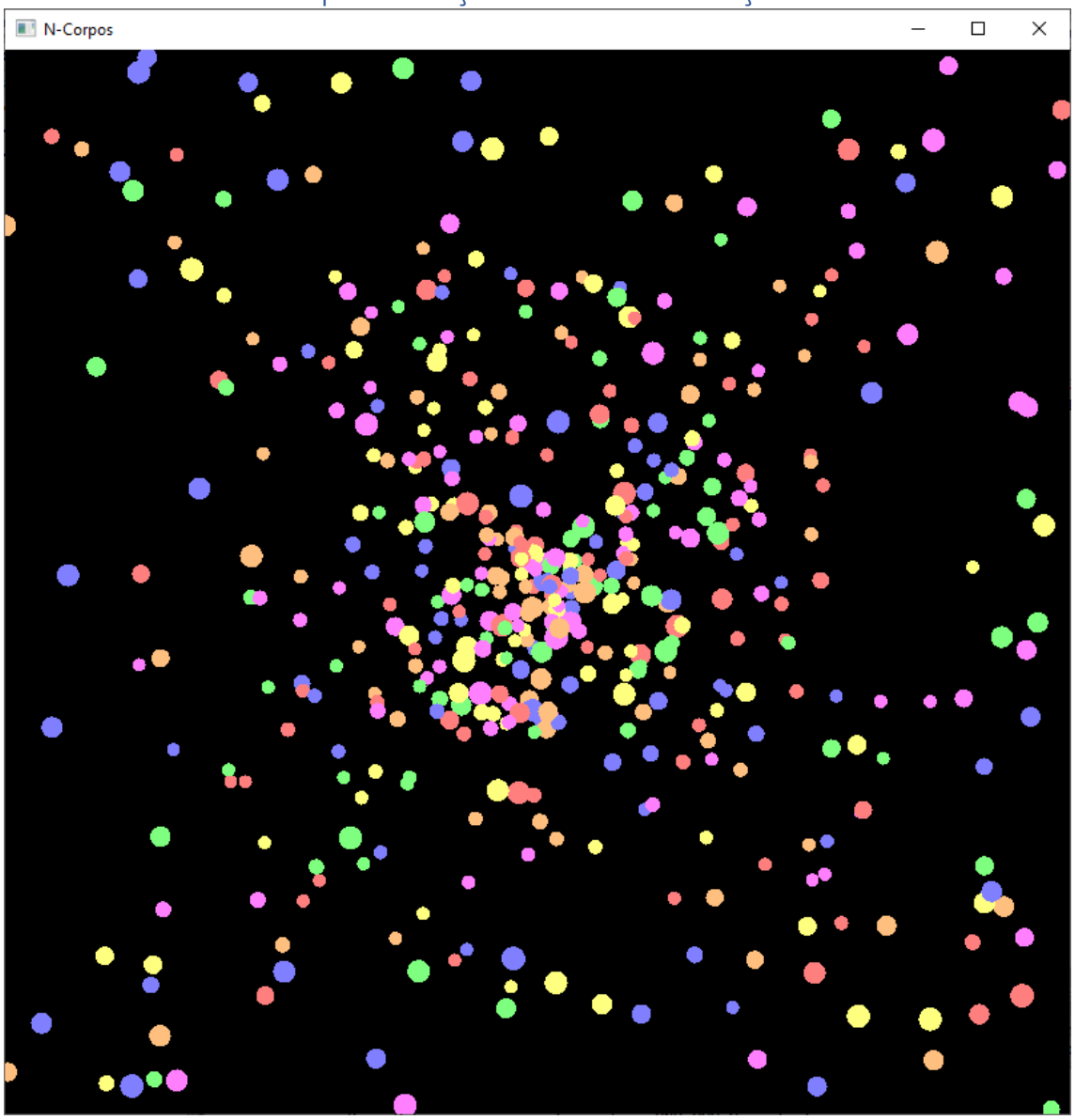


Testando com 500 partículas e $\theta = 1.0$, obtemos em média 5.08 FPS. Houve uma melhora muito significante, quase 17x mais rápido.



Testando com 500 partículas e $\theta = 10.0$, obtemos em média 31.45 FPS. Houve uma melhora muito significante, quase 105x mais rápido, porém o erro da simulação já é aparente graficamente.

Implementação Visual da Simulação



Comentários Extras

O algoritmo de Barnes-Hut é extremamente paralelizável e com um código altamente otimizado podemos simular mais de milhares de partículas em tempo real com a ajuda de uma placa de vídeo potente com vários núcleos. No meu algoritmo eu utilizo apenas um “thread” que é usado tanto para “renderizar” quanto pra computar as forças.

https://www.youtube.com/watch?v=DoLe1c-eokI

https://www.youtube.com/watch?v=BIP_m1SG6p4&

https://www.andrew.cmu.edu/user/wenjayt/proposal.html

Neste trabalho eu também evitei entrar em detalhe em relação à parte gráfica já que não faz parte do assunto e também por não estar no meu domínio.