Parallelizzazione dell'algoritmo di intersezione di MöllerTrumbore

Gallina Roberto

06/11/2023

Indice

1 Algoritmo di MöllerTrumbore

L'algoritmo di intersezione di MöllerTrumbore è un metodo usato per capire se un raggio (linea con un punto d'inzio e una direzione) intersecano un triangolo nello spazio 3D. è usato nella Computer Graphics per determinare quali parti di modelli 3D sono visibili da un certo punto di vista.

```
bool RayIntersectsTriangle(
    Vector3D rayOrigin,
    Vector3D rayVector,
    Triangle* inTriangle,
    Vector3D& outIntersectionPoint
){
    const float EPSILON = 0.0000001;
    Vector3D vertex0 = inTriangle->vertex0;
    Vector3D vertex1 = inTriangle->vertex1;
    Vector3D vertex2 = inTriangle->vertex2;

    Vector3D edge1, edge2, h, s, q;
    edge1 = vertex1 - vertex0;
    edge2 = vertex2 - vertex0;

    float a, f, u, v;
    h = rayVector.crossProduct(edge2);
    a = edge1.dotProduct(h);

    if (a > -EPSILON && a < EPSILON)
        return false;

    f = 1.0 / a;
    s = rayOrigin - vertex0;
    u = f * s.dotProduct(h);

    if (u < 0.0 || u > 1.0)
        return false;

    q = s.crossProduct(edge1);
    v = f * rayVector.dotProduct(q);

    if (v < 0.0 || u + v > 1.0)
        return false;

    float t = f * edge2.dotProduct(q);

    if (t > EPSILON){
        // ray intersection
        outIntersectionPoint = rayOrigin + rayVector * t;
        return true;
    }else
        return false;
}
```

Figura 1: Algoritmo di MöllerTrumbore da Wikipedia

2 Prima semplice implementazione

Prima di procedere con la prima implementazione, va do a creare le strutture e le funzioni basilari che serviranno nei vari calcoli.

2.1 Struttura Point3D

```
struct Point3D{
    double x;
    double y;
    double z;
};
```

2.2 Struttura Triangle

```
struct Triangle {
    Point3D p1;
    Point3D p2;
    Point3D p3;
};
```

2.3 Differenza tra vettori

```
Point3D difference(Point3D a, Point3D b){
    return {a.x - b.x, a.y - b.y, a.z - b.z};
}
```

2.4 Dot product

```
double dotProduct(Point3D &v1, Point3D &v2) {
    return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
}
```

2.5 Cross product

```
Point3D crossProduct(Point3D &v1, Point3D &v2) {
    return {
        v1.y * v2.z - v1.z * v2.y,
        v1.z * v2.x - v1.x * v2.z,
        v1.x * v2.y - v1.y * v2.x
    };
}
```

2.6 Codice

Vado quindi ad implementare la versione del codice vista precedentemente

```
bool rayIntersectsTriangle(
    Point3D rayOrigin,
    Point3D rayVector,
    Triangle inTriangle
){
    const float EPSILON = 0.0000001;
    Point3D vertex0 = inTriangle.p1;
    Point3D vertex1 = inTriangle.p2;
    Point3D vertex2 = inTriangle.p3;

    Point3D edge1, edge2, h, s, q;
    double a, f, u, v;

    edge1 = difference(vertex1, vertex0);
    edge2 = difference(vertex2, vertex0);

    h = crossProduct(rayVector, edge2);
    a = dotProduct(edge1, h);

    if (a > -EPSILON && a < EPSILON)
        return false;

    f = 1.0 / a;
    s = difference(rayOrigin, vertex0);
    u = f * dotProduct(s, h);

    if (u < 0.0 || u > 1.0)
        return false;

    q = crossProduct(s, edge1);
    v = f * dotProduct(rayVector, q);
    if (v < 0.0 || u + v > 1.0)
        return false;

    double t = f * dotProduct(edge2, q);

    if (t > EPSILON)
        return true;

    return false;
}
```

Implemento ora il ‘main‘ verificando che l’algoritmo funzioni correttamente sui casi base

2.6.1 Raggio interseca triangolo

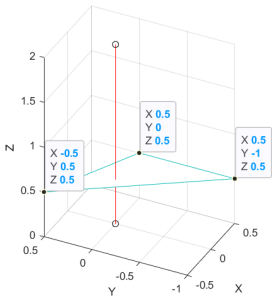
```
int main() {
    Point3D rayOrigin = {0.0, 0.0, 0.0};
    Point3D rayDirection = {0.0, 0.0, 2.0};

    Triangle triangle = {
        {-0.5, 0.5, 0.5},
        {0.5, 0.0, 0.5},
        {0.5, -1.0, 0.5}
    };

    cout << rayIntersectsTriangle(rayOrigin, rayDirection,
        triangle);

    return 0;
}
```

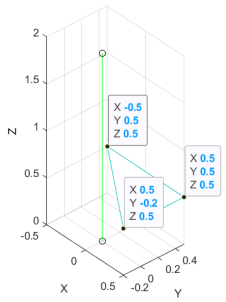
```
1
```

2.6.2 Raggio non interseca triangolo

```
..
Triangle triangle = {
    {-0.5, 0.5, 0.5},
    {0.5, 0.5, 0.5},
    {0.5, -0.2, 0.5}
};
..
```

```
0
```





3 Esempio complesso

3.1 Lettura dei file

Vado ora a implementare delle funzioni per poter leggere i file dei punti e delle mesh, sappiamo infatti che il file *verts.csv* è composto da tre numeri con virgola scritti in notazione scentifica.

Es. -6.195182353258132935e-02,-5.577753782272338867e-01,5.792263150215148926e-01

Vado quindi a creare una funzione che dato il nome del file genera un vettore di punti.

```
vector<Point3D> readPoints(string filename) {
    vector<Point3D> punti;
    string linea;
    ifstream file(filename);
    if (file.is_open()) {
        while (getline(file, linea)) {
            Point3D punto;
            stringstream ss(linea);
            string valore;
            if (getline(ss, valore, ',')) {
                punto.x = stod(valore);
            }
            if (getline(ss, valore, ',')) {
                punto.y = stod(valore);
            }
            if (getline(ss, valore)) {
            punto.z = stod(valore);
            }
            punti.push_back(punto);
        }
        file.close();
    }
    return punti;
}
```

Allo stesso modo sappiamo che il file *mesh.csv* è composto da tre numeri scritti in notazione scentifica, corrispondenti al numero (indice) del punto.

Es. 1.00000000000000000e+00,2.00000000000000000e+00,0.00000000000000000e+00

Vado quindi a creare una funzione che dato il nome del file e un vettore di punti genera un vettore di triangoli (mesh).

```
vector<Triangle> readTriangles(string filename, vector<Point3D>
     punti) {
    vector<Triangle> triangoli;
    string linea;
    ifstream file(filename);
    if (file.is_open()) {
        while (getline(file, linea)) {
            Triangle t;
            stringstream ss(linea);
            string valore;
            if (getline(ss, valore, ',')) {
                index = (int) stod(valore);
                t.p1 = punti[index];
            }
            if (getline(ss, valore, ',')) {
                index = (int) stod(valore);
                t.p2 = punti[index];
            }
            if (getline(ss, valore)) {
                index = (int) stod(valore);
                t.p3 = punti[index];
            }
            triangoli.push_back(t);
        }
        file.close();
    }
    return triangoli;
}
```

Nel main uso queste funzioni per caricare i dati.

```
int main() {
    vector<Point3D> punti = readPoints("verts.csv");
    vector<Triangle> triangoli = readTriangles("meshes.csv",
        punti);
    .
}
```

3.2 Implementazione

Vado ora a implementare una funzione che data l'origine, la direzzione e un vettore di triangoli verifica se quel raggio interseca almeno un triangolo

```
bool rayIntersectsAnyTriangle(Point3D origin, Point3D dir,
    vector<Triangle> triangles) {
    for (const Triangle &triangle : triangles) {
        bool r = rayIntersectsTriangle(origin, dir, triangle);
        if (r) {
            return true;
        }
    }
    return false;
}
```

Nel main vado quindi a verificare se ogni punto interserca almeno un triangolo

```
int main() {
    vector<Point3D> punti = readPoints("verts.csv");
    vector<Triangle> triangoli = readTriangles("meshes.csv",
        punti);

    Point3D rayOrigin = {0.0, 0.0, 0.0};

    for (const Point3D &punto : punti) {
        if (punto.x > 0 || punto.x <= 0)
            cout << rayIntersectsAnyTriangle(rayOrigin, punto,
                triangoli) << endl;
    }

    return 0;
}
```

3.3 Output su file

Non rimane che aggiungere la stampa direttamente su un file.

```
int main(){
    ofstream oFile("out.txt");
    vector<Point3D> punti = readPoints("verts.csv");
    vector<Triangle> triangoli = readTriangles("meshes.csv",
        punti);

    Point3D rayOrigin = {0.0, 0.0, 0.0};

    for (const Point3D &punto : punti)
    {
        if (punto.x > 0 || punto.x <= 0)
            oFile << rayIntersectsAnyTriangle(rayOrigin, punto,
                triangoli) << endl;
    }
    oFile.close();

    return 0;
}
```

4 Parallelizzazione

4.1 Introduzione a CUDA

La prima cosa da parallelizzare è il ciclo che viene fatto su punti, ricordiamo infatti che per ogni punto viene verificato se il segmento che unisce l'origine con esso interseca qualche triangolo.

Vado quindi a creare un Kernel CUDA che presa l'origine, un punto e il vettore di triangoli (e una indirizzo dove dalvare il risultato) verifichi la condizione.

Ricordiamo che le parti principali per usare un Kernel Cuda sono:

- Allocazione memoria del device
- Passaggio dati dall'host al device
- Passaggio dati dal device all'host
- Deallocazione memoria del device

4.1.1 Passaggio dati

La prima versione del kernel servirà per verificare il correttto passaggio dei dati tra l'*host* e il *device* pertanto restituirà sempre *true*.

Vado quindi a preparare i parametri da passare: il punto origine, un array di punti, un array di trinagoli, il numero di triangoli e un array di booleani che serve per salvare i valori di ritorno. Per fare ciò definisco alcuni puntatori per l'array di punti e di triangoli e l'array di ritorno.

```
Point3D *d_points;
Triangle *d_triangles;
bool *d_result;
```

Vado poi ad allocarne lo spazio.

```
cudaMalloc(&d_points, Np * sizeof(Point3D) );
cudaMalloc(&d_triangles, Nt * sizeof(Triangle) );
cudaMalloc(&d_result, Np * sizeof(bool) );
```

E copio i dati.

```
cudaMemcpy(
    d_points, h_punti,
    Np * sizeof(Point3D),
    cudaMemcpyHostToDevice
);
cudaMemcpy(
    d_triangles,
    h_triangoli,
    Nt * sizeof(Triangle),
    cudaMemcpyHostToDevice
);
```

Andrò poi ad eseguire il kernel (spiegato successivamente), prima però calcolo la dimensione dei blocchi

```
dim3 DimGrid(Np/256, 1, 1);
if (Np % 256)
    DimGrid.x++;

dim3 DimBlock(256, 1, 1);

rayIntersectsAnyTrianglesKernel<<<DimGrid, DimBlock>>>(
    origin,
    d_points,
    d_triangles,
    Nt,
    d_result
);
```

E coppiare i risultati.

```
cudaMemcpy(
    result,
    d_result,
    Np * sizeof(bool),
    cudaMemcpyDeviceToHost
);
```

E infine libero lo spazio allocato.

```
cudaFree( d_points );
cudaFree( d_triangles );
cudaFree( d_result );
```

Il kernel sarà una funzione void e dovrà ricevere i parametri precedentemente preparato, ogni thread analizzerà un unico punto, per fare ciò calcola l'indice del punto che deve analizzare.

```
__global__
void rayIntersectsAnyTrianglesKernel(
    Point3D rayOrigin,
    Point3D *ps,
    Triangle *ts,
    int Nt,
    bool *result
){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    result[idx] = true;
}
```

4.2 Implementazione algoritmo nel Kernel

Per definizione devo restituire *true* se almeno un triangolo è intersecato dal raggio che unisce l'origine al punto, pertanto il kernel sarà un cilo su tutti i triangoli e se interseca restituirà true altrimenti a fine ciclo restituirà false.

```
__global__
void rayIntersectsAnyTrianglesKernel(
    Point3D rayOrigin,
    Point3D *ps,
    Triangle *ts,
    int Nt,
    bool *result
){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    Point3D point = ps[idx];
    Point3D dir = point;

    result[idx] = false;

    for(int i = 0; i < Nt; i++){
        Triangle t = ts[i];

        bool r = rayIntersectsTriangle(rayOrigin, dir, t);
        if(r){
            result[idx] = true;
        }
    }
}
```

Implemento quindi la funzione che verifica línsersezione di un raggio in un triangolo, devo quindi definire le funzioni dotProduct, crossProduct e difference come funzioni $\_\_device\_\_$ in modo che i thread possono usarle.

```
__device__
Point3D difference(Point3D a, Point3D b) {
    return {a.x - b.x, a.y - b.y, a.z - b.z};
}

__device__
Point3D crossProduct(Point3D &v1, Point3D &v2){
    return {    v1.y * v2.z - v1.z * v2.y,
        v1.z * v2.x - v1.x * v2.z,
        v1.x * v2.y - v1.y * v2.x};
}

__device__
double dotProduct(Point3D &v1, Point3D &v2){
    return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
}
```

Vado ora a creare una funzione partendo da quando visto sulla Wiki, usando le funzioni precedentemente create.

```
__device__
bool rayIntersectsTriangle(
    Point3D rayOrigin,
    Point3D rayVector,
    Triangle inTriangle
){
    const float EPSILON = 0.0000001;
    Point3D vertex0 = inTriangle.p1;
    Point3D vertex1 = inTriangle.p2;
    Point3D vertex2 = inTriangle.p3;

    Point3D edge1, edge2, h, s, q;
    double a, f, u, v;

    edge1 = difference(vertex1, vertex0);
    edge2 = difference(vertex2, vertex0);

    h = crossProduct(rayVector, edge2);
    a = dotProduct(edge1, h);

    if (a > -EPSILON && a < EPSILON)
        return false;

    f = 1.0 / a;
    s = difference(rayOrigin, vertex0);
    u = f * dotProduct(s, h);

    if (u < 0.0 || u > 1.0)
        return false;

    q = crossProduct(s, edge1);
    v = f * dotProduct(rayVector, q);
    if (v < 0.0 || u + v > 1.0)
        return false;

    double t = f * dotProduct(edge2, q);

    if (t > EPSILON)
        return true;
    return false;
}
```

4.3 Uso della shared memory

Attualmente ogni thread, per eseguire il calcolo, deve ciclare su tutti i trinagoli, che risiedono in memoria globale; questo comporta un notevole tempo d'accesso. Vado quindi ad introdurre una memoria condivisa (shared) tra i vari thread, essa memorizzerrà alcuni triangoli in modo che ci siano meno accessi in memoria globale.

Aggiungo la dimensione della shared memory, e dentro la funzione eseguita dai vari thread aggiungo un array di triangoli condiviso.

```
#define SH_SIZE 512
```

```
__global__
void rayIntersectsAnyTrianglesKernel(
    ..
){
    ..
    __shared__ Triangle sh_triangle[SH_SIZE];
    ..
}
```

Essendo la dimensione della shared minore del numero totale di triangoli, sarà necessario caricare una parte di triangoli, attendere che tutti i thread finiscano di analizzarli, per poi caricare i successivi

```
__global__
void rayIntersectsAnyTrianglesKernel(
    Point3D rayOrigin,
    Point3D *ps,
    Triangle *ts,
    int Nt,
    bool *result
){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    Point3D point = ps[idx];
    Point3D dir = point;
    result[idx] = false;

    __shared__ Triangle sh_triangle[SH_SIZE];

    bool res = false;

    for(int m = 0; m < Nt/SH_SIZE; m++){
        if(idx < SH_SIZE){
            sh_triangle[idx] = ts[m * SH_SIZE + idx];
        }
        __syncthreads();

        for(int i = 0; i < SH_SIZE; i++){
            Triangle t = sh_triangle[i];

            bool r = rayIntersectsTriangle(rayOrigin, dir, t);
            if(r){
                res = true;
            }
        }
        __syncthreads();
    }
    result[idx] = res;
}
```

5 Metriche

5.1 Specifiche

Tutte le esecuzioni sono state fatte su un pc con:

- CPU: AMD Ryzen 7 5800x 8-Core (3.80GHz)
- GPU: NVIDIA GeForce RTX 3060 (circa 3500 CUDA Core da 1.70 GHz)

5.2 Metriche

Tipo di codice	**Tempo**
Sequenziale	4391245824ns = 4391245µs 4391ms = 4.3s
Parallelo senza SH	536918784ns = 536918µs 536ms = 0.5s
Parallelo con SH (Size 16)	362736128ns = 362736µs 362ms = 0.3s
Parallelo con SH (Size 32)	412990208ns = 412990µs 412ms = 0.4s
Parallelo con SH (Size 64)	392447744ns = 392447µs 392ms = 0.4s
Parallelo con SH (Size 128)	411473152ns = 411473µs 411ms = 0.4s
Parallelo con SH (Size 256)	299524300ns = 299524µs 299ms = 0.3s

5.3 Conclusioni

Anche se siamo su un caso di studio abbastanza piccolo, l'esecuzione tramite CPU risulta essere circa 10 volte più lenta rispetto all'uso della GPU; inoltre implmentando la shared memory si vedono dei leggeri benefici, credo queste differenze verrebbero amplificate in un caso di studio più grande.