Università degli Studi di Padova
Dipartimento di Ingegneria Dell'Informazione
Corso di laurea magistrale in Ingegneria Informatica

Analisi e implementazione di algoritmi per la scomposizione gerarchica di superfici 3D in oriented bounding box e conseguenti strategie per il calcolo delle collisioni tra oggetti tridimensionali

Laureando: Fabio Bottero

Relatore: Prof. Enrico Pagello

Anno Accademico 2012/2013

Alla mia famiglia che mi ha sempre aiutato e sostenuto

Indice

Capitolo 1

Introduzione

Euclid Labs è un'azienda che produce software per la simulazione di robot industriali. Questi software servono per verificare il corretto funzionamento del programma da inserire nei robot. In questi applicativi viene definito un mondo (detto scena) in cui si possono importare oggetti ed eseguire operazioni su di essi, come: traslazioni, rotazioni, verifica delle intersezioni...

Il nostro scopo è migliorare l'algoritmo che individua le intersezioni per incrementare le performance del software. Ogni qual volta viene spostato un oggetto nel mondo, questo algoritmo viene utilizzato per verificare se l'oggetto ha colliso e se quindi la sua posizione è valida o meno. Esso si basa sulla creazione di un OBB Tree per ogni oggetto presente nella scena. Questi alberi vengono generati nel momento dell'inserimento nella scena e vengono salvati in RAM. Di conseguenza, dopo la chiusura dell'applicativo, i dati vengono cancellati.

Nel momento in cui si cercano le intersezioni, verranno controllate tutte le possibili coppie di alberi alla ricerca di una collisione e infine, verrà notificato all'utente l'esito dell'operazione.

Nel tentativo di migliorare il tempo d'esecuzione dell'algoritmo, si sperimenteranno i seguenti metodi:

- L'utilizzo dei thread in modo da poter sfruttare appieno le caratteristiche dei nuovi processori che contengono sempre più unità di calcolo.
- Il salvataggio del tree di un oggetto in modo che non sia necessario ricalcolarlo ogni qual volta l'oggetto venga inserito in una scena.
- L'utilizzo della GPU per incrementare la velocità nella generazione dell'albero di un oggetto.

Capitolo 2

Stato dell'arte

Fin dagli albori dell'informatica, la comunità scientifica ha sviluppato algoritmi per l'individuazione delle intersezioni tra solidi non regolari. Questi solidi vengono generalmente descritti tramite un insieme di triangoli, che definiscono la superficie tridimensionale dell'oggetto. Come si può intuire, la quantità di dati da elaborare è elevata e di conseguenza si cercano metodi alternativi per velocizzare il processo. Per far ciò, una delle migliori strategie, è sicuramente l'organizzazione in strutture gerarchiche.

2.1 Bounding Volume Hierarchy

La suddivisione dell'oggetto tramite Bounding Volume Hierarchy ha un enorme vantaggio: se i volumi delle radici di due alberi non hanno intersezioni allora gli oggetti non si intersecano e quindi non sarà necessario fare ulteriori controlli. Se invece i volumi hanno intersezioni si procederà testando tutte le combinazioni dei figli dei nodi radice. Questo procedimento continuerà finché non si arriverà ad un livello in cui non ci sono intersezioni oppure finchè non si raggiungono le foglie in entrambi gli alberi. In quest'ultimo caso, se si ha un'intersezione, gli oggetti collidono. Vedremo successivamente i principali tipi di Bounding Volume Hierarchy e come questi controllino in maniera diversa l'intersezione in base al tipo di volume usato.

Il principale svantaggio di questi algoritmi sta nel fatto che bisogna creare una struttura ad albero per ogni oggetto e aggiornarla nel momento in cui l'oggetto venga traslato, rotato o entrambi.

Codice 2.1: Algoritmo per la creazione dell'albero

```
void createTree(Node *parent){
   if(finish()){
      Node *sons = split(parent);
      createTree(sons[0]);
      createTree(sons[1]);
   }
}
```

Nel processo di creazione dell'albero si dovranno tenere in considerazione le seguenti regole:

1. l'oggetto del nodo padre deve venir racchiuso da un tipo di volume predefinito
2. ogni figlio userà la stessa tipologia di volume del padre per racchiudere una parte dell'oggetto genitore
3. l'unione dei volumi dei figli dovrà contenere l'oggetto racchiuso dal padre
4. il controllo della condizione di fermata nello sviluppo dell'albero, rappresentata nel codice dal metodo *finish*. Questa può essere riferita al box, quindi le dimensioni inferiori ad una soglia, oppure all'oggetto racchiuso, numero di facce inferiore ad un predeterminato valore.

2.1.1 Sphere Bounding Box

La prima versione degli Bounding Volume Hierarchy fu lo Sphere Bounding Box. In questa versione si sviluppa un albero in cui il volume predefinito è la sfera. Questa racchiude l'oggetto intero o una parte di esso, in base al tipo di nodo. Se questo è la radice dell'albero racchiuderà interamente l'oggetto, se invece è un altro nodo solamente una parte. Solitamente si divide l'oggetto in 2 parti usando un piano ortogonale ad un asse. Solitamente, questo piano, passerà per il punto medio dell'asse maggiore del box. In alcuni casi, questa divisione, su quest'asse, interseca tutti i triangoli e di conseguenza non si riescono ad ottenere due sottoinsiemi; la procedura, infatti, prevede di dividere i triangoli in tre insiemi: triangoli sopra l'asse, sotto l'asse e che intersecano l'asse. Quando questo succede si procederà dividendo il box su un altro asse.

Gli Sphere Bounding Volume hanno risultati ottimi per quanto riguarda il tempo per il controllo delle collisioni tra un volume e l'altro e per la quantità di dati che ogni nodo dell'albero dovrà salvare. Infatti ogni nodo dovrà memorizzare 3 valori relativi al centro della sfera ed un quarto per il raggio. Questa quantità è inferiore rispetto agli altri algoritmi che descriveremo successivamente.

Il test di intersezione per verificare l'intersezione di due sfere è:

Codice 2.2: Test d'intersezione tra sfere

```
bool sphereIntersection(Vector3D& c1, Vector3D& c2, double r1, double r2)
{
   if ( sqrt((c1.x - c2.x)^2 + (c1.y - c2.y)^2 + (c1.z - c2.z)^2) <= r1 + r2)
      return true;
   else
      return false;
}
```

dove *c1* e *c2* sono i centri delle due sfere da testare e *r1* e *r2* i relativi raggi.

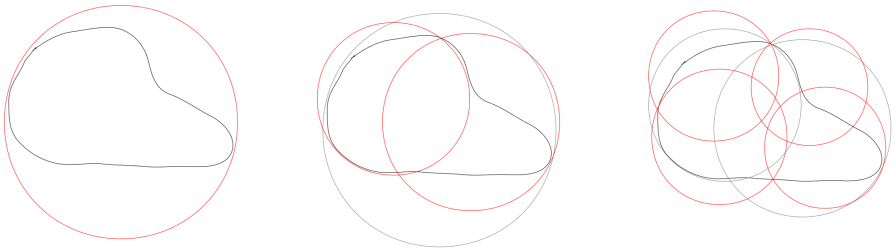


Figura 2.1: Fasi per la costruzione degli Sphere Bounding Box Tree

Se la condizione ritornerà **true** allora le sfere si intersecheranno. Si noti che il numero di operazioni che la CPU eseguirà per ogni controllo sono:

- 7 somme/differenze
- 3 moltiplicazioni (dato che ogni valore è elevato alla seconda, equivale a moltiplicare il valore per se stesso e quindi una sola moltiplicazione sarà necessaria per ogni elevamento a potenza)
- 1 radice quadrata
- 1 confronto

e dato che ogni addizione necessità di un ciclo di CPU, ogni moltiplicazione di 3, un confronto di uno e la radice quadrata di circa 40, il numero totale di cicli per eseguire il metodo è 58.

2.1.2 Axis Aligned Bounding Box

Un'altra versione degli Hierarchy Bounding Volume è quella degli Axis Aligned Bounding Box [4]. Il volume utilizzato in questo caso è un parallelepipedo le cui facce sono parallele ai piani xy, yz e xz. Così facendo si ottiene un albero composto da nodi contenenti parallelepipedi e quindi i dati che dovranno essere salvati in ogni nodo sono 3 valori per il centro del parallelepipedo e altri 3 per metà altezza, lunghezza e larghezza. Vengono salvate le dimensioni dimezzate per semplicità, infatti per individuare il box basterà sommare e sottrarre al centro i valori relativi alle mezze dimensioni.

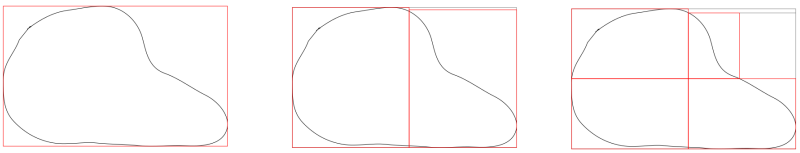


Figura 2.2: Fasi per la costruzione degli Axis Aligned Bounding Box Tree

Il test di intersezione per questi nodi è:

Codice 2.3: Test d'intersezione tra nodi dell'AABB

```
bool AABBIntersection(Vector3D& c1, Vector3D& c2, Vector3D& r1, Vector3D& r2)
{
   if ( Abs(c1.x - c2.x) > (r1.x + r2.x) ) return false;
   if ( Abs(c1.y - c2.y) > (r1.y + r2.y) ) return false;
   if ( Abs(c1.z - c2.z) > (r1.z + r2.z) ) return false;
   return true;
}
```

dove *c1* e *c2* sono i centri dei due parallelepipedi e *r1* e *r2* sono la metà delle dimensioni dei parallelepipedi.

Se la condizione ritornerà **true** allora i due parallelepipedi si intersecheranno. Si noti che il numero di operazioni necessarie in questo caso non è costante, quindi analizzeremo il caso peggiore e il caso migliore.

Il numero totale di operazioni nel caso peggiore è:

- 3 confronti
- 6 somme/differenze
- 3 valori assoluti

Nel caso migliore, invece, è:

- 1 confronti
- 2 somme/differenze
- 1 valore assoluto

Il valore assoluto esegue un confronto e in alcuni casi una moltiplicazione quindi il numero totale di cicli è 21 nel caso peggiore e 4 nel caso migliore.

2.1.3 Oriented Bounding Box

La versione successiva agli Axis Aligned Bounding Box è la Oriented Bounding Box [11]. Questa versione aggiunge ad ogni nodo altri 3 valori per memorizzare la direttrice del box. In questo modo potremo orientare i parallelepipedi rispetto ai tre piani e racchiudere, in maniera migliore, la parte dell'oggetto.

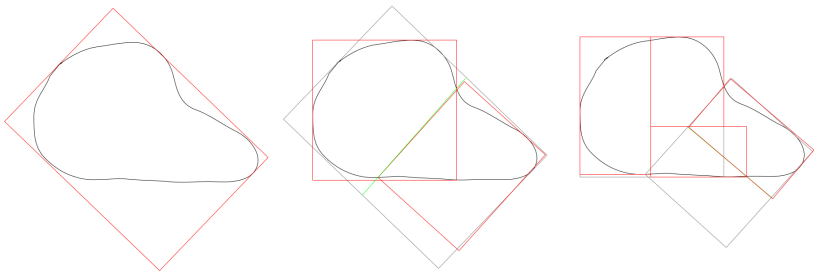


Figura 2.3: Fasi per la costruzione degli Oriented Bounding Box Tree

Il controllo delle collisioni verrà eseguito controllando le intersezioni tra le proiezioni di due box su di un asse. Grazie al teorema Separating Axis Theorem [10], gli assi su cui controllare le intersezioni saranno solamente 15: 3 assi del primo box, 3 assi del secondo box e 9 assi dati dal prodotto vettoriale delle combinazioni degli assi del primo e secondo box.

Di conseguenza, il test di intersezione sarà:

Codice 2.4: Test d'intersezione tra nodi dell'OBB

```
bool OBBIntersection(OBB& obb1, OBB& obb2)
{
   Vector3D T = box1.center - box2.center;
   Vector3D R = box1.rotationMatrix.inverse() * box2.rotationMatrix;

   double a1 = box1.halfSize.X;
   double a2 = box1.halfSize.Y;
   double a3 = box1.halfSize.Z;

   double b1 = box2.halfSize.X;
   double b2 = box2.halfSize.Y;
   double b3 = box2.halfSize.Z;

   double Tx = box1.Translation.X;
   double Ty = box1.Translation.Y;
   double Tz = box1.Translation.Z;

   Vector3D aX = box1.Direction.X;
   Vector3D aY = box1.Direction.Y;
   Vector3D aZ = box1.Direction.Z;

   Vector3D L = box1.Direction.X;
   double maxSum = Math.Abs((Tx * L.X) + (Ty * L.Y) + (Tz * L.Z));
   double radiusSum = a1 + b1 * R[1,1] +
            b2 * R[1,2] + b3 * R[1,3];
   if (radiusSum < maxSum)
      return false;

   L = box1.Direction.Y;
   maxSum = Math.Abs((Tx * L.X) + (Ty * L.Y) + (Tz * L.Z));
   radiusSum = a2 + b1 * R[2,1] + b2 * R[2,2] + b3 * R[2,3];
   if (radiusSum < maxSum)
      return false;

   L = box1.Direction.Z;
   maxSum = Math.Abs((Tx * L.X) + (Ty * L.Y) + (Tz * L.Z));
   radiusSum = a3 + b1 * R[3,1] + b2 * R[3,2] + b3 * R[3,3];
   if (radiusSum < maxSum)
      return false;
```

```
Vector3D bX = box.Rot.XDir;
Vector3D bY = box.Rot.YDir;
Vector3D bZ = box.Rot.ZDir;

L = bX;
maxSum = Math.Abs((Tx * L.X) + (Ty * L.Y) + (Tz * L.Z));
radiusSum = b1 + a1 * R[1,1] + a2 * R[2,1] + a3 * R[3,1];
if (radiusSum < maxSum)
   return false;

L = bY;
maxSum = Math.Abs((Tx * L.X) + (Ty * L.Y) + (Tz * L.Z));
radiusSum = b2 + a1 * R[1,2] + a2 * R[2,2] + a3 * R[3,2];
if (radiusSum < maxSum)
   return false;

L = bZ;
maxSum = Math.Abs((Tx * L.X) + (Ty * L.Y) + (Tz * L.Z));
radiusSum = b3 + a1 * R[1,3] + a2 * R[2,3] + a3 * R[3,3];
if (radiusSum < maxSum)
   return false;


L = Vector3D.Cross(X, bX);
maxSum = Math.Abs((Tx * L.X) + (Ty * L.Y) + (Tz * L.Z));
radiusSum = a2 * R[3,1] + a3 * R[2,1] + b2 * R[1,3] + b3 * R[1,2];
if (radiusSum < maxSum)
   return false;

L = Vector3D.Cross(X, bY);
maxSum = Math.Abs((Tx * L.X) + (Ty * L.Y) + (Tz * L.Z));
radiusSum = a2 * R[3,2] + a3 * R[2,2] + b1 * R[1,3] + b3 * R[1,1];
if (radiusSum < maxSum)
   return false;

L = Vector3D.Cross(X, bZ);
maxSum = Math.Abs((Tx * L.X) + (Ty * L.Y) + (Tz * L.Z));
radiusSum = a2 * R[3,3] + a3 * R[2,3] + b1 * R[1,2] + b2 * R[1,1];
if (radiusSum < maxSum)
   return false;

L = Vector3D.Cross(Y, bX);
maxSum = Math.Abs((Tx * L.X) + (Ty * L.Y) + (Tz * L.Z));
radiusSum = a1 * R[3,1] + a3 * R[1,1] + b2 * R[2,3] + b3 * R[2,2];
if (radiusSum < maxSum)
   return false;

L = Vector3D.Cross(Y, bY);
maxSum = Math.Abs((Tx * L.X) + (Ty * L.Y) + (Tz * L.Z));
radiusSum = a1 * R[3,2] + a3 * R[1,2] + b1 * R[2,3] + b3 * R[2,1];
if (radiusSum < maxSum)
   return false;

L = Vector3D.Cross(Y, bZ);
maxSum = Math.Abs((Tx * L.X) + (Ty * L.Y) + (Tz * L.Z));
radiusSum = a1 * R[3,3] + a3 * R[1,3] + b1 * R[2,2] + b2 * R[2,1];
if (radiusSum < maxSum)
   return false;

L = Vector3D.Cross(Z, bX);
maxSum = Math.Abs((Tx * L.X) + (Ty * L.Y) + (Tz * L.Z));
radiusSum = a1 * R[2,1] + a2 * R[1,1] + b2 * R[3,3] + b3 * R[3,2];
if (radiusSum < maxSum)
   return false;

L = Vector3D.Cross(Z, bY);
```

```
        maxSum = Math.Abs((Tx * L.X) + (Ty * L.Y) + (Tz * L.Z));
        radiusSum = a1 * R[2,2] + a2 * R[1,2] + b1 * R[3,3] + b3 * R[3,1];
        if (radiusSum < maxSum)
            return false;

        L = Vector3D.Cross(Z, bZ);
        maxSum = Math.Abs((Tx * L.X) + (Ty * L.Y) + (Tz * L.Z));
        radiusSum = a1 * R[2,3] + a2 * R[1,3] + b1 * R[3,2] + b2 * R[3,1];
        if (radiusSum < maxSum)
            return false;

        return true;
    }
```

Il numero di operazioni da eseguire ad ogni controllo nel caso peggiore sono:

- 15 confonti
- 552 somme/differenze
- 738 moltiplicazioni
- 15 valori assoluti

Invece nel caso migliore:

- 1 confronto
- 26 somme/differenze
- 33 moltiplicazioni
- 1 valore assoluto

Di conseguenza il numero di cicli nel caso peggiore è 2841, invece nel caso migliore 127.

Naturalmente osservando il numero di operazioni necessarie per eseguire il test di intersezione e la quantità di memoria utilizzata non sembrerebbe conveniente utilizzare questo metodo, ma il suo principale punto di forza sta nell'approssimare meglio gli oggetti nello spazio e le loro parti. Grazie a questa caratteristica saranno necessari meno livelli nell'albero e un numero inferiore di controlli rispetto alle altre soluzioni presentate precedentemente.

Inoltre una importante miglioria rispetto agli AABB sta nel fatto che dopo una rotazione non sarà necessario modificare completamente i boxes. Questo, infatti, è necessario per mantenere i box allineati agli assi.

2.2 Tree in GPU

L'algoritmo per la costruzione degli alberi in GPU più utilizzato, e consigliato anche da NVIDIA, è il Linear Bounding Volume Hierarchy (LBVH) [12]. Questo metodo crea una foglia per ogni triangolo e la etichetta con il codice di Morton. A questo punto, le foglie, verranno ordinate e verrà creata la radice con un box che racchiude tutti i triangoli. Si procederà creando i nodi figli della radice dividendo

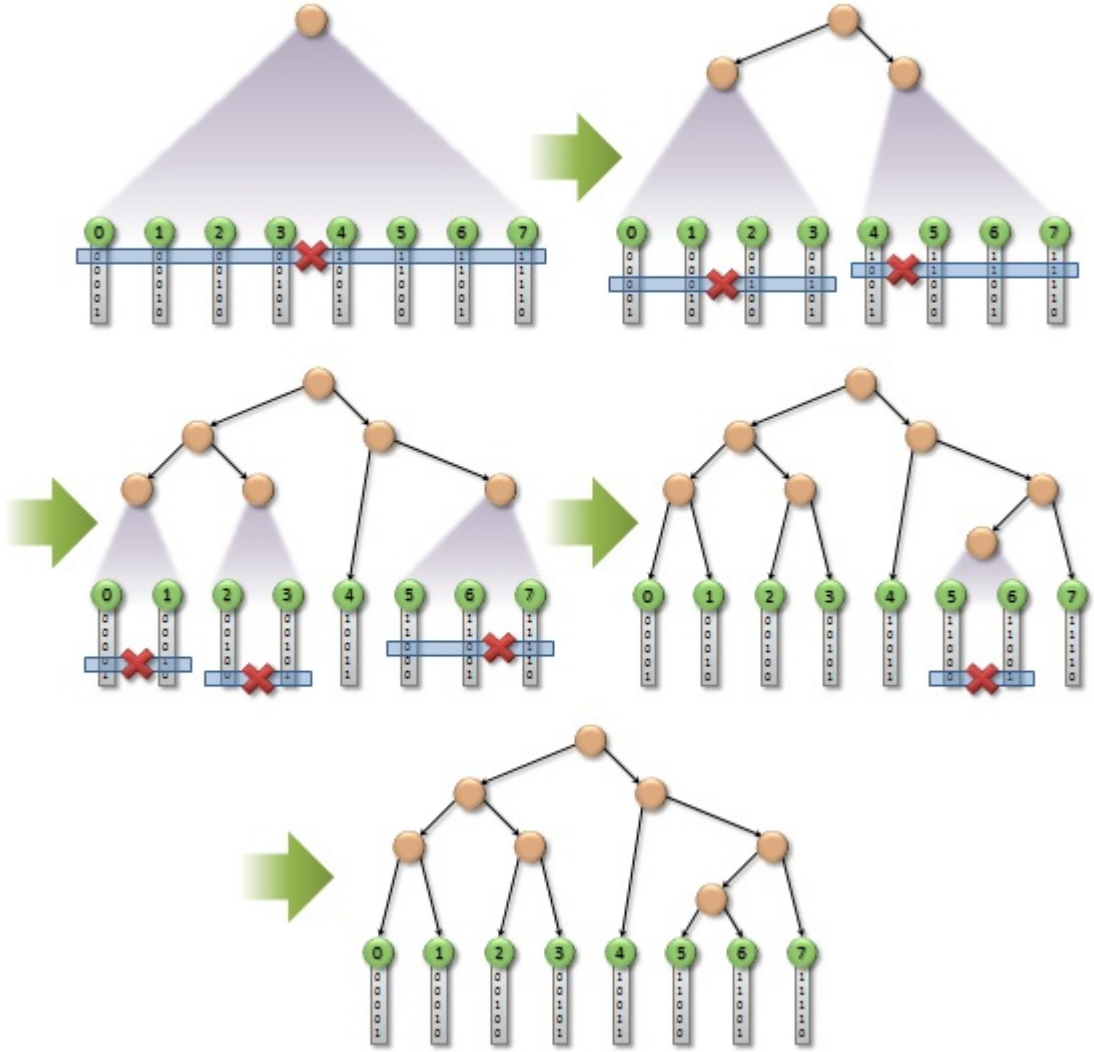


Figura 2.4: Divisione dell'array in base al livello. Fonte: Nvidia

l'array delle foglie quando il bit relativo al livello dell'albero in cui andremo a posizionare i figli, varierà, come si può vedere in figura.

L'algoritmo da modificare crea un OBB-Tree e termina individuando delle superfici nel momento in cui, il box, ha un numero di triangoli inferiore ad una soglia e inserendo queste ultime come dati delle foglie. Di conseguenza non possiamo applicare LBVH perchè il nostro algoritmo funziona con una strategia top-down e termina creando delle superfici, a differenza di questo appena presentato, che usa una strategia diversa e costruisce l'albero basandosi sui singoli triangoli. Se implementassimo quest'ultimo, tutta la parte relativa al controllo delle collisioni non sarebbe più applicabile e quindi il periodo di test diventerebbe troppo lungo.

Morton code Il codice di Morton [2] [3] è una funzione che mappa dati multidimensionali in una dimensione. Questa funzione permette di far ciò intrecciando

i valori binari di ogni coordinata dell'oggetto da descrivere. Per esempio, se il nostro spazio è stato diviso in quattro in ogni dimensione e abbiamo un triangolo nel punto descritto in figura, possiamo etichettarlo con il codice 111110. In questo caso il triangolo si troverà nella posizione 11 in x, 11 in y e 10 in z. Il primo e il quarto uno stanno a rappresentare la posizione del triangolo nella matrice cubica rispetto l'asse x, quelli nella seconda e quinta posizione la coordinata y e il terzo e sesto valore la coordinata z.

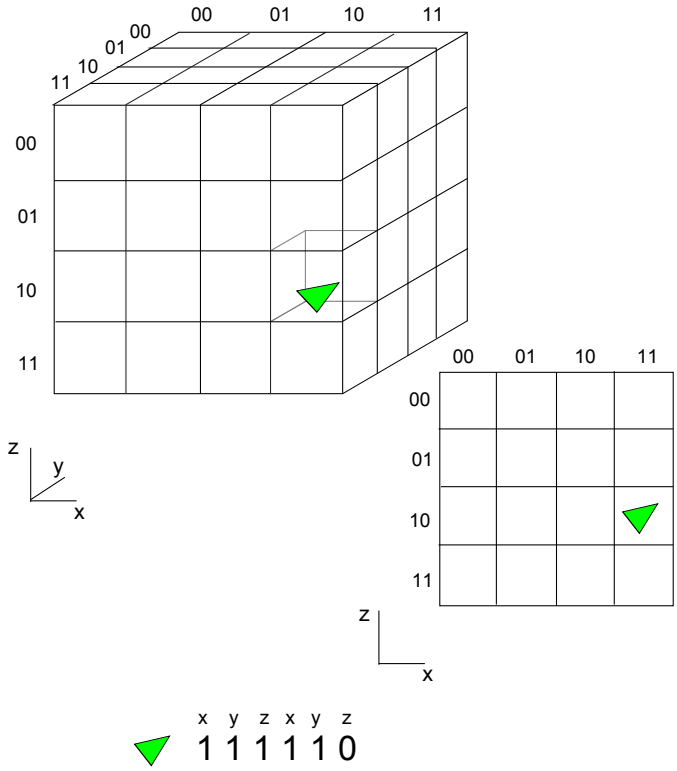


Figura 2.5: Codice di Morton

La tabella a destra si riferisce alla sezione di cubo con y pari a 11. In questo modo per cercare se ci sono oggetti all'interno di un volume, basterà cercare se ci sono degli oggetti etichettati con il codice del volume.

Capitolo 3

Situazione iniziale

I software di Euclid Labs si basano su molteplici librerie e tra queste troviamo World e CollisionDetection. La prima serve per definire la scena all'interno del software dove si trovano tutti gli oggetti con le relative posizioni spaziali; la seconda, invece, serve per creare gli alberi e controllare se nella scena ci sono delle collisioni. Quest'ultima contiene l'algoritmo che si andrà a modificare per incrementare le performance globali.

Nel momento in cui viene importato un oggetto, CollisionDetection creerà un OBB Tree in modo da cercare velocemente le collisioni tra questo e gli altri oggetti quando lo verrà richiesto. Questa fase necessita di molte risorse e tempo, perciò si cercherà di parallelizzare l'algoritmo in modo da ridurre il tempo necessario alla creazione dell'albero. Questo è un albero binario e ogni nodo verrà ricavato seguendo la procedura descritta precedentemente. L'unica particolarità rispetto ad un OBB Tree standard sta nel fatto che nelle foglie vengono descritte delle superfici complanari. Questo perchè il controllo tra superfici è più veloce del controllo tra singoli triangoli [8].

Il codice sorgente fornitomi è scritto in linguaggio C# e si dispone del Framework .NET 4.5 e delle librerie DirectX 11, 10, 9c e SlimDX. Queste librerie vengono massivamente utilizzate nel resto del programma e di conseguenza eventuali ulteriori librerie che si vorranno aggiungere al progetto non dovranno collidere con quelle presenti.

3.1 Framework .NET 4.5

Questo framework è utilizzato da tutta la suite di linguaggi .NET (Visual Basic, Visual C++, C#, ...). In questa libreria è possibile trovare una serie di classi che implementano i maggiori tipi di strutture dati, classi per la creazione dell'interfaccia grafica, per la creazione di servizi web e la comunicazione con il sistema operativo e con dispositivi esterni tramite porte USB, seriali e parallele. Le librerie dinamiche (DLL) possono essere create con qualsiasi linguaggio .NET e utilizzate all'interno della suite. Di conseguenza posso creare una libreria in un linguaggio e utilizzarla con un altro.

3.2 DirectX

Microsoft fin dal 1995 [7] ha messo a disposizione questa libreria per da coloro che utilizzavano massivamente la grafica. Questa è stata sviluppata negli anni fino ad arrivare alla versione attuale. L'unico linguaggio che gli fa concorrenza è OpenGL [14], che viene sviluppato da un consorzio di produttori tra cui Intel.

Attualmente, la scheda video, non viene più considerata solamente come un device per la renderizzazione, ma anche per il calcolo parallelo. La svolta è arrivata da Nvidia, la quale, creando la libreria Cuda [6], ha permesso la creazione di applicazioni che eseguono parti di codice in scheda grafica sfruttandone il parallelismo e la pipeline. Dopo un primo periodo, anche Microsoft e OpenGL si sono mossi in questa direzione creando DirectCompute [5] e C++ AMP [1], la prima, e OpenCl [13], la seconda.

I linguaggi Microsoft per la programmazione in scheda grafica permettono di creare un sistema totalmente integrabile con .NET e DirectX. In questo modo gli applicativi possono utilizzare sia CPU, che GPU. Naturalmente la seconda verrà usata solamente nel momento in cui il calcolo possa essere parallelizzato altrimenti le performance sarebbero peggiori rispetto al codice eseguito in CPU.

3.3 SlimDX

SlimDX [15] è un wrapper della libreria DirectX in C#. Questo permette di interagire con la libreria direttamente da C# semplificando la programmazione della parte grafica.

Capitolo 4

Parallelizzazione in CPU

Il primo tentativo è stato quello di parallelizzare il lavoro svolto dalla CPU utilizzando più thread, in modo da poter utilizzare al meglio la potenza di calcolo dei nuovi processori che montano più unità di calcolo nel singolo chip.

4.1 Descrizione Algoritmi

Tutte le versioni sfruttano la caratteristica dell'albero binario di avere due figli per nodo. Nel momento in cui il padre genera i figli, uno di questi verrà assegnato ad un nuovo thread in modo da espandere i sottoalberi utilizzando thread differenti. Questa strategia verrà interrotta nel momento in cui i thread avviati raggiungeranno il numero prefissato e, dopodiché, ogni processo si occuperà della generazione di entrambi i sottoalberi.

Di seguito presenteremo varie versioni parallele per creare l'albero descritto precedentemente.

4.1.1 Multilock

In questa versione saranno necessari alcuni semafori per evitare dirty-read o premature-write sui dati che devono essere modificati da più soggetti. Per far ciò si è utilizzata la primitiva **lock**, la quale permette di bloccare un'oggetto e, di conseguenza, tutti gli altri processi che cercano di accedere a quella risorsa verranno sospesi finchè il lock non verrà rilasciato. Quando questo accadrà, verrà notificato ai processi bloccati, i quali concorreranno per ottenere la risorsa.

Questa primitiva permette quindi di eseguire in modo sicuro porzioni di codice. Nel nostro caso ci permetterà di isolare le sezioni di codice in cui è necessario:

- assegnare un'indice incrementale al nodo
- controllare se c'è la possibilità di rilasciare ulteriori thread

Nel primo caso ci serve un'indice incrementale per identificare il nodo all'interno dell'albero.

Nel secondo caso dobbiamo avviare un numero limitato di thread in modo che questi non rallentino l'esecuzione a causa del loro overhead al momento della creazione e della distruzione, da un lato, e, al tempo per il trasferimento dati da RAM a cache, nell'altro.

4.1.2 Separated Array

In questa versione si è cercato di ridurre al minimo il numero di lock necessari. Per far ciò si è indicato ad ogni thread un range di indici da poter assegnare ai nuovi nodi. Non si è comunque riusciti ad eliminare il controllo per il numero di thread e di conseguenza bisognerà utilizzare il lock in questa sezione di codice. Alla fine ogni thread dovrà scrivere il numero di nodi creati e la massima profondità su un array così il processo principale ricaverà i nodi totali e la profondità massima dell'albero.

4.1.3 Thread Pool

ThreadPool è una classe messa a disposizione dalla libreria .NET che rende trasparente al programmatore la gestione dei thread. Questa contiene il metodo *QueueUserWorkItem* al quale si dovrà passare la sezione di codice da eseguire con gli eventuali parametri e, ThreadPool, autonomamente avvierà questa procedura su un thread. Così facendo useremo i lock solo nel momento in cui i nuovi nodi necessiteranno degli indici. In questo modo cercheremo di ridurre al minimo le sezioni critiche nel codice e il numero di operazioni in esse dato che la sezione critica per il controllo del numero di thread contiene più operazioni rispetto a quella per ottenere l'indice del nodo.

4.1.4 One node, One thread

La versione "One node, one thread" si comporta come multilock, ma su questa non verrà settato un limite al numero di thread avviabili. Questa versione, come la Thread Pool, necessita dei lock nel momento in cui verranno creati nuovi nodi per assegnare loro un indice. Dato che la struttura dati è un albero binario sarà il padre dei nuovi nodi colui che farà la richiesta dei due indici per i figli. In questo modo si allungherà leggermente il tempo per eseguire la sezione critica, ma si dimezzeranno il numero di lock durante l'esecuzione riducendo l'overhead dovuto alla loro gestione.

4.1.5 My Pool Thread

La versione My Pool Thread, ha al suo interno la classe MyPoolThread, che si comporta come la classe ThreadPool solamente che a differenza di quest'ultima in MyPoolThread si deve inserire il numero massimo di thread avviabili. Questa particolarità serve per riutilizzare i thread assegnando loro il lavoro in modo dinamico, in più, riutilizzandoli, si riuscirà a diminuire l'overhead dovuto alla loro

generazione e distruzione. Per gestire la sincronizzazione sarà necessario introdurre un semaforo numerico in modo da conoscere il numero di processi disponibili per l'esecuzione. Quindi il processo principale si occuperà di controllare i signal sul semaforo in modo che nel momento in cui un thread termini venga subito riavviato se ci sono dei nodi da sviluppare. Per l'assegnazione degli indici ai nodi si è utilizzata la stessa politica di "One node, One thread".

Limited depth

Questa versione di My Pool Thread sviluppa il sottoalbero del nodo assegnatogli per un numero di livelli predeterminato. Dopo aver espanso il sottoalbero, ogni foglia che dovrà essere sviluppata ulteriormente verrà inserita in uno stack. Nel momento in cui un thread terminerà il lavoro, estrarrà il primo nodo dallo stack e continuerà il processo di costruzione. In questo modo si limita il carico di ogni singolo thread cercando di distribuirlo in maniera migliore nel momento in cui l'albero sia sbilanciato. Infatti, limitando l'altezza del sottoalbero, il processo genererà più velocemente i nodi da inserire nello stack e, così facendo, più thread potranno essere avviati ed eseguire contemporaneamente.

Maximum threads utilization

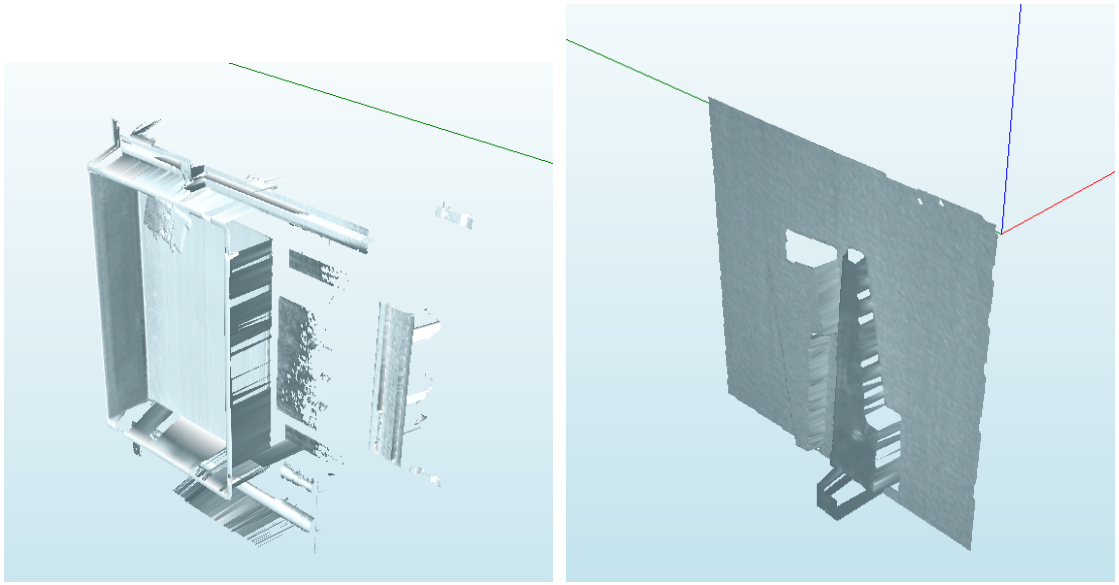
A differenza della versione precedente, questa non inserirà nessun limite sul massimo numero di nodi da sviluppare, ma bensì seguirà la seguente regola:

Regola: *Ogni processo durante la generazione dei figli controllerà se tutti i thread stanno eseguendo, in caso contrario questo inserirà uno dei figli nello stack*

Ogni thread continuerà ad eseguire finchè non terminerà a causa dell'assenza di nodi all'interno dello stack. Da notare come la regola verrà utilizzata all'inizio, per fornire lavoro ad ogni thread, e durante l'esecuzione, in modo da bilanciare il carico su tutti i thread. Per evitare che i thread terminino per assenza di nodi nello stack si avrà un semaforo su cui verrà fatto il signal ad ogni nuovo rilascio. In questo modo ogni thread controllerà lo stack solo dopo un signal e non potrà terminare anticipatamente.

Slow start Questa versione del "Maximum threads utilization" inibirà la regola nella parte iniziale dell'algoritmo in modo che il processo principale generi un numero di nodi maggiore o uguale al numero di thread massimi che si potranno avviare. A questo punto si renderà attiva la regola e, così facendo, ogni thread avrà del lavoro e potrà iniziare l'esecuzione.

Start if there is work Questa versione del "Maximum threads utilization" avvierà i thread solamente se quelli avviati sono inferiori a quelli avviabili. Ogni qualvolta un thread genera i figli controllerà questo numero e, se è inferiore, avvierà un thread. Così facendo ogni thread eseguirà finchè non terminerà il suo sottoalbero e poi riceverà altro lavoro da un altro thread.



(a) Calib1 (b) AfterGrid

Figura 4.1: I benchmark utilizzati

4.2 Valutazione delle prestazioni

Utilizzando i thread il tempo di esecuzione dovrebbe diminuire rispetto alla versione seriale, questo però non è vero in tutti i casi, come si è visto sperimentalmente.

Alcune soluzioni hanno performance migliori:

- Multilock
- Separated Array
- Slow start

alcune peggiori:

- Thread Pool
- My Pool Thread
- Limited thread
- Maximum thread utilization
- Start if there is work

e alcune non funzionano:

- One node, One Thread

One node, One Thread non funziona perchè il sistema operativo lo blocca a causa dell'eccessivo numero di thread avviati.

La causa dell'andamento non lineare degli algoritmi e la conseguente perdita di performance è da ricercare negli accessi alla RAM. Infatti questo algoritmo necessita di grandi quantità di dati residenti in memoria centrale e, perchè questi siano disponibili per il processore, dovranno essere trasferiti in cache. Avendo questa, dimensioni molto limitate, necessita di continui swap dalla memoria centrale, riducendo di molto le performance.

I benchmark sono stati eseguiti utilizzando come input i files Calib1 (21 MB) e AfterGrid (5 MB). Dai dati acquisiti si è notato che con processori della famiglia Intel i3 si ottengono risultati migliori se il numero di threads è 12-16 e, su processori i7, se il numero di threads è uguale o doppio al numero di processori virtuali presenti sul chip(12 e 16 in questo caso). Si può inoltre notare come le performance di *Slow Start* e *Multilock* siano molto simili usando il numero di processori ottimale o vicino ad esso.

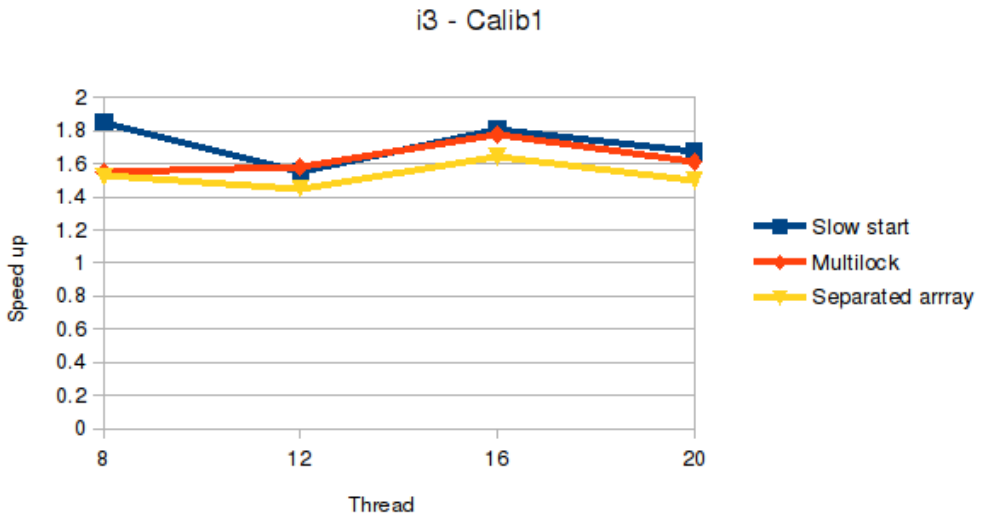


Tabella 4.1: Intel i3 performance (i3-3220 4 processori virtuali) - Calib1

Num Thread	Slow Start	Multilock	Separated Array
8	47 (54%)	56 (74%)	57 (65%)
12	48 (55%)	44 (50%)	55 (63%)
16	50 (57%)	49 (56%)	52 (59%)
20	54 (62%)	45 (51%)	50 (57%)

Risultati ottenuti con gli algoritmi che garantiscono performance migliori. Tutti i valori riportati sono espressi in secondi e i valori nelle parentesi sono le percentuali di tempo impiegato rispetto all'algoritmo attuale, il quale esegue in 87 secondi su questo processore

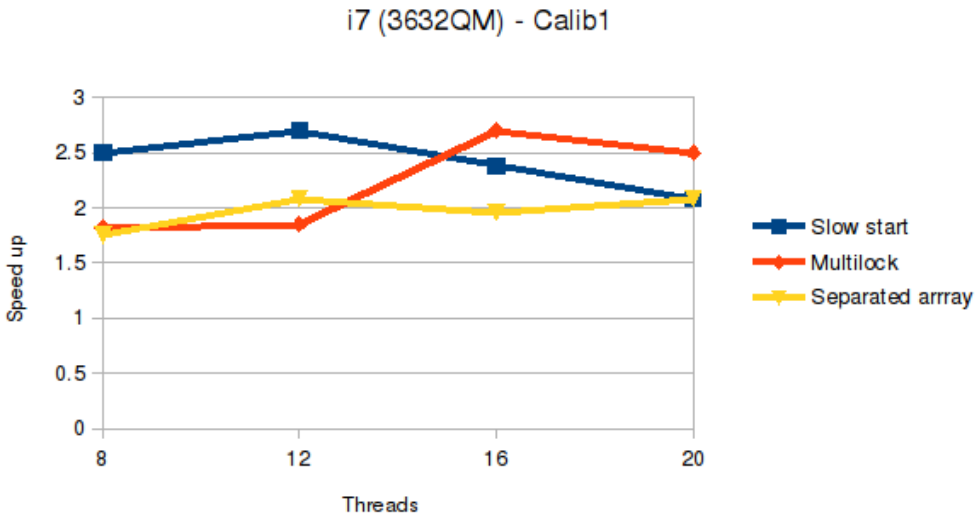


Tabella 4.2: i7 performance (i7-3632QM 8 processori virtuali) - Calib1

Num Thread	Slow Start	Multilock	Separated Array
8	40 (40%)	55 (55%)	57 (57%)
12	37 (37%)	54 (54%)	48 (48%)
16	42 (42%)	37 (37%)	51 (51%)
20	48 (48%)	40 (40%)	48 (48%)

Risultati ottenuti con gli algoritmi che garantiscono performance migliori. Tutti i valori riportati sono espressi in secondi e i valori nelle parentesi sono le percentuali di tempo impiegato rispetto all'algoritmo attuale, il quale esegue in 100 secondi su questo processore.

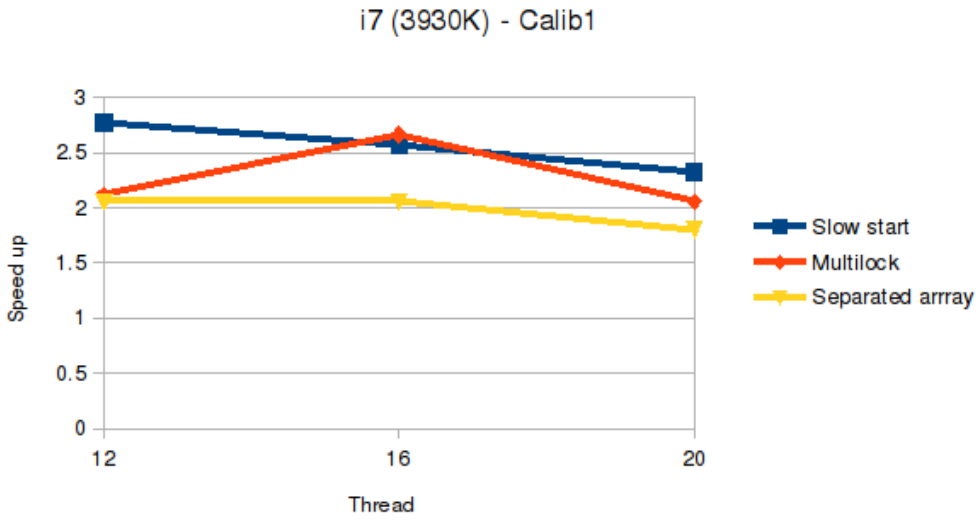


Tabella 4.3: i7 performance (i7-3930K 12 processori virtuali) - Calib1

Num Thread	Slow Start	Multilock	Separated Array
12	26 (36%)	34 (47%)	35 (48%)
16	28 (38%)	27 (37%)	35 (48%)
20	31 (43%)	35 (48%)	40 (55%)

Risultati ottenuti con gli algoritmi che garantiscono performance migliori. Tutti i valori riportati sono espressi in secondi e i valori nelle parentesi sono le percentuali di tempo impiegato rispetto all'algoritmo attuale, il quale esegue in 72 secondi su questo processore.

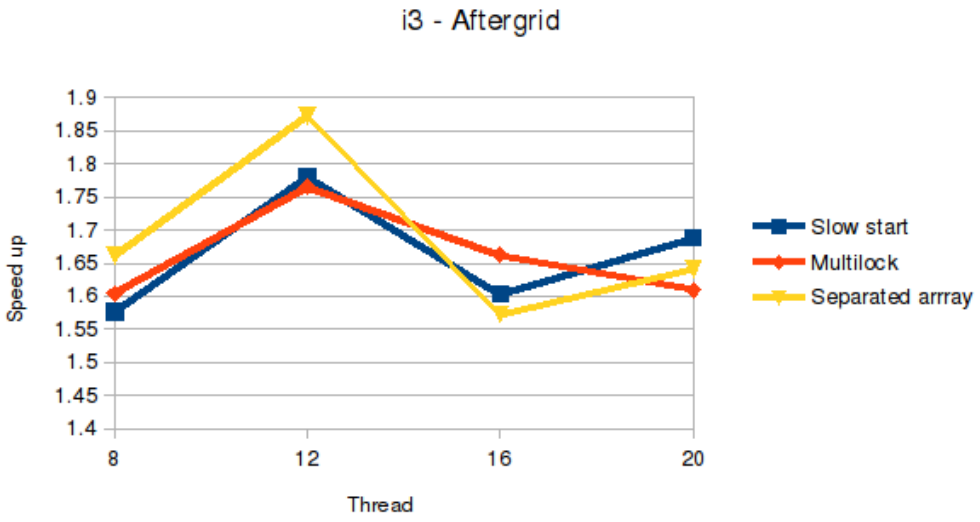


Tabella 4.4: i3 performance (i3-3220 4 processori virtuali) - AfterGrid

Num Thread	Multilock	Separated Array	Slow Start
8	13,06 (63%)	12,84 (62%)	12,4 (60%)
12	11,57 (56%)	11,67 (56%)	11,0 (53%)
16	12,85 (62%)	12,39 (60%)	13,1 (63%)
20	12,2 (59%)	12,8 (62%)	12,55 (60%)

Risultati ottenuti con gli algoritmi che garantiscono performance migliori. Tutti i valori riportati sono espressi in secondi e i valori nelle parentesi sono le percentuali di tempo impiegato rispetto all'algoritmo attuale, il quale esegue in 20,6 secondi su questo processore.

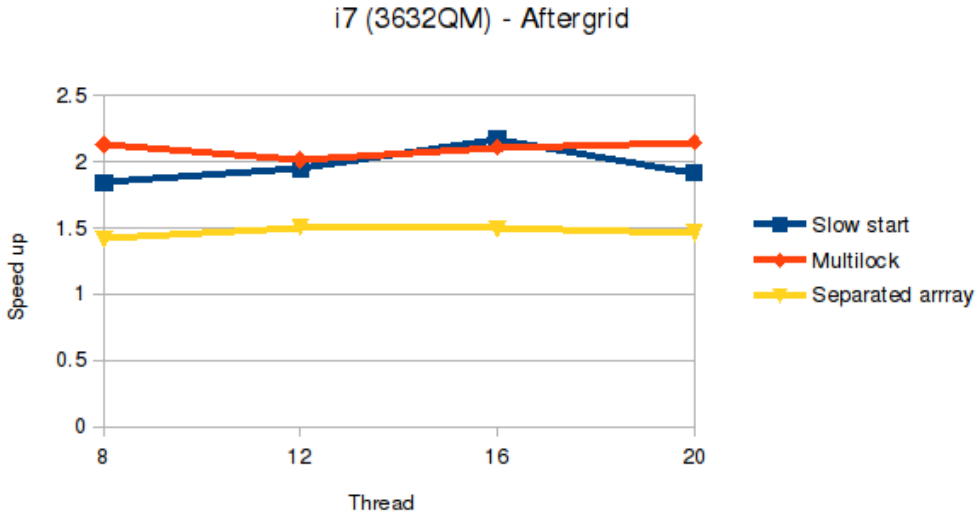


Tabella 4.5: i7 performance (i7-3632QM 8 processori virtuali) - AfterGrid

Num Thread	Multilock	Separated Array	Slow Start
8	12,47 (54%)	10,8 (46%)	16,2 (70%)
12	11,8 (51%)	11,4 (49%)	15,26 (66%)
16	10,6 (46%)	10,9 (47%)	15,37 (66%)
20	12 (52%)	10,7 (46%)	15,72 (66%)

Risultati ottenuti con gli algoritmi che garantiscono performance migliori. Tutti i valori riportati sono espressi in secondi e i valori nelle parentesi rappresentano le percentuali di tempo impiegato rispetto all'algoritmo attuale, il quale esegue in 23 secondi su questo processore.

Capitolo 5

Salvataggio delle strutture dati

Per il salvataggio di strutture dati in un file, il Framework .NET, mette a disposizione due librerie: XML e Binary. Queste librerie contengono varie classi per il salvataggio e la lettura di strutture dati da file. Le classi che utilizzeremo sono XmlFormatter e BinaryFormatter che permettono di salvare e leggere istanze di oggetti direttamente dal disco senza dover scrivere metodi specifici per ottenere o immagazzinare questi dati. Questo processo, infatti, sarà automatico.

Queste classi necessitano di alcuni caratteri o stringhe speciali che permettano il riconoscimento dell'attributo descritto nella relativa sezione del file e per questo motivo i file non contengono solamente i dati, ma anche dei tag per definire il tipo di dato, il nome della classe e a quale attributo, il dato, si riferisce. Questi "supplementi" incrementano la dimensione complessiva del file e, per questa ragione, si sono sviluppati dei benchmark per individuare quelli che riducono le dimensioni totali del file. Dopo un'attenta analisi si è preferito usare il BinaryFormatter perché l'overload è minore rispetto a quello di XmlFormatter.

La struttura dati che dobbiamo salvare è un albero e di conseguenza dobbiamo mantenere la relazione padre-figlio tra i nodi. Dato che il nostro albero è binario, lo si può mappare in un array e per mantenere la gerarchia delle superfici presenti si aggiunge ad ognuna l'id della foglia a cui appartiene. In questo modo si è ridotto al minimo l'utilizzo di memoria e si è aumentata la velocità nella scrittura, lettura e creazione delle strutture dati in RAM.

I risultati ottenuti con il file AfterGrid (5MB) sono:

- tempo di scrittura su file: 42 secondi
- tempo di acquisizione e creazione struttura dati: 5 minuti e 12 secondi
- spazio occupato: 320 MB

La fase di acquisizione dati impiega circa 20 volte il tempo utilizzato per la creazione dell'albero in CPU, di conseguenza questa strategia non ha portato ai risultati sperati.

Capitolo 6

Parallelizzazione in GPU

Il termine GPGPU sta a significare General-Purpose computing on Graphics Processing Units. Questa branchia della computazione parallela permette l'esecuzione di algoritmi su scheda video. Questo dispositivo ha un elevato numero di processori che possono essere utilizzati in parallelo. In questo modo, il codice altamente parallelizzabile, può essere eseguito su questo device incrementando le performance e, sfruttando anche la pipeline, è possibile diminuire ulteriormente il tempo d'esecuzione. La pipeline viene utilizzata nelle sezioni di codice dove si devono eseguire le stesse operazioni su più insiemi di dati.

Grazie a queste caratteristiche, la parallelizzazione su GPU si è sviluppata molto in questi ultimi anni e di conseguenza sono stati sviluppate molteplici linguaggi, i più famosi sono sicuramente CUDA e OpenCl.

Anche se l'algoritmo usato per la costruzione dell'albero non è efficientemente parallelizzabile, si sono studiate comunque le sue performance in GPU. Lo scopo è capire se ci sono dei miglioramenti rispetto alla versione attuale. Dato che questo algoritmo dovrà essere integrato nel software di Euclid Labs, ci sono dei vincoli riguardo i linguaggi da scegliere:

- devono essere indipendenti da modello e casa produttrice di GPU
- devono essere eseguibili da C#
- preferibile, se sviluppati da Microsoft

Di conseguenza gli unici linguaggi che permettono di far ciò sono DirectCompute e C++ AMP, i quali sono sviluppate da Microsoft. Il punto di forza di questi linguaggi è che si basano su DirectX, che sono delle API già implementate nei driver delle schede video da parecchi anni e di conseguenza non necessitano di modifiche ai driver delle schede video da parte dei produttori.

Un'altra alternativa è OpenCl, però quest'ultima viene implementata lentamente all'interno dei driver NVIDIA, di conseguenza, per evitare performance molto diverse tra schede video di marche differenti, si è preferito usare i linguaggi Microsoft.

6.1 DirectCompute

Questo linguaggio è di proprietà di Microsoft e permette l'esecuzione di script in GPU caricabili in runtime. Questa caratteristica permette di modificare il codice da eseguire in scheda video direttamente da C# nel caso in cui:

- si debba caricare uno script diverso in base ai dati restituiti da uno stadio precedente dell'algoritmo
- si debba modificare il numero di thread che devono eseguire lo script
- non si conosca a priori la massima grandezza delle strutture dati

Questo linguaggio non permette l'allocazione dinamica della memoria e di conseguenza la grandezza delle strutture dati deve essere definita nella fase di compilazione.

Per poter eseguire gli script bisognerà definire il numero di gruppi e il numero di thread in ognuno di essi. Così facendo si potrà suddividere l'algoritmo in modo che sia eseguibile da più gruppi e da più thread contemporaneamente.

Ogni gruppo contiene al massimo 1024 thread organizzati in matrici cubiche. I gruppi non possono accedere alla memoria puntata da un altro gruppo e la memoria totale assegnata ad ognuno è esigua. Per questo motivo il numero di gruppi all'interno degli script, solitamente, è molto basso.

I thread saranno suddivisi in gruppi e hanno queste particolarità:

- Il numero di thread massimo nella dimensione X e Y della matrice cubica è 1024 e nella Z è 64.
- Il numero di thread per gruppo è al più 1024.
- La memoria per ogni gruppo è limitata a 16 KB
- La memoria locale per ogni thread è limitata a 256 B e la somma di queste deve essere inferiore a 16 KB
- C'è la possibilità di eseguire operazioni atomiche
- Il massimo numero di thread avviabili è 65535

Il programma sviluppato con questo linguaggio non esegue correttamente a causa della quantità di dati necessaria ad ogni thread e la quantità di dati che genera. L'esempio qui di seguito si riferisce ad un file di 5 MB.

Dati necessari:

- 20000 vertici → 60000 double → 480000 byte
- 44000 facce → 132000 int → 528000 byte

La memoria necessaria, ad ogni nodo, per accedere ai dati di input è circa 1 MB, a differenza della massima allocabile ad un gruppo che è 16 KB. Di conseguenza l'algoritmo non è stato implementato a causa del limite appena descritto.

6.2 C++ AMP

C++ AMP è un linguaggio specifico per il GPGPU. Infatti Microsoft, che ne è il proprietario, lo ha sviluppato cercando di nascondere o semplificare al programmatore tutte le problematiche relative a:

- gestione del flusso dati;
- suddivisione dei gruppi;
- suddivisione dei thread;

Questo linguaggio permette di sviluppare applicativi molto più semplicemente di DirectCompute grazie ad una ampia documentazione, al forum e al blog Native Concurrency in cui vengono presentati molti esempi di applicazioni.

Gli svantaggi in C++ AMP sono:

- non si possono lasciare dati in memoria video tra più esecuzioni
- mancanza di allocazione dinamica della memoria
- mancanza di metodi per conoscere il processore in cui si sta eseguendo il codice
- mancanza di metodi per conoscere il numero totale di processori nella GPU

Per permettere l'esecuzione degli script che usano C++ AMP in C# sarà necessario creare un progetto in Visual C++ e compilarlo come libreria dinamica (DLL) [9]. Questo progetto preparerà il contesto in modo da rendere possibile lo scambio dati tra RAM e GPU-RAM e conterrà lo script da eseguire in GPU. Grazie all'interoperabilità delle DLL, sarà possibile usare questa libreria come se fosse scritta in C#. Per far ciò basterà linkare la libreria in fase di compilazione e il compilatore creerà un eseguibile, in linguaggio macchina, che unirà il codice di entrambi i progetti.

Come primo tentativo si è proceduto sviluppando la suddivisione del nodo in GPU però, ogni qual volta la funzione ritorna i risultati, questa termina e i dati relativi alle facce e ai vertici vengono persi. Di conseguenza il tempo di esecuzione aumenta drasticamente perchè ad ogni chiamata bisognerà ricaricare in memoria video queste informazioni. Infatti l'esecuzione di questo algoritmo con il file AfterGrid impiega 50 secondi a differenza dei 20 usati attualmente.

C++ AMP non ha dei limiti sulla massima memoria allocabile per thread, però deve essere predefinita al momento della compilazione o, per avere un elemento dinamico, deve essere passato al metodo tramite un array nel momento dell'esecuzione. Nel nostro caso la memoria necessaria è conosciuta a priori per la fase di splitting, ma non nella fase di decomposizione in superfici perchè dipende dal numero di facce nel box e dalla loro disposizione. Per questo motivo si voleva creare una matrice con un numero di righe uguale al numero di processori e passarla a runtime allo script. In questo modo ogni processore accedeva alla sua riga

e poteva usare questo spazio per l'algoritmo per l'individuazione delle superfici. Però, in C++ AMP, non c'è la possibilità di conoscere il processore sul quale si sta eseguendo e nemmeno il numero totale di unità di calcolo. Di conseguenza anche questa opzione è stata scartata.

Capitolo 7

Conclusioni

Al termine di questa analisi possiamo affermare che l'unico metodo che ha dato un esito positivo è la parallelizzazione in CPU. Questa infatti ha permesso di arrivare ad un algoritmo con un tempo di esecuzione inferiore del 60% rispetto a quello precedente.

Il salvataggio su dispositivo di memoria secondaria non ha dato esito positivo perchè il tempo di elaborazione per rendere l'albero fruibile dal programma è di qualche ordine di grandezza maggiore rispetto al tempo per la costruzione tramite CPU. Questo è dovuto principalmente all'enorme quantità di dati da leggere e da organizzare in memoria.

Infine lo sviluppo dell'albero tramite GPU non ha le performance sperate a causa della mancanza dell'allocazione dinamica da parte dei linguaggi presi in considerazione. Anche i metodi alternativi hanno portato ad un fallimento a causa dell'impossibilità di conoscere il numero totale di processori presenti sulla scheda e su quale di essi stia eseguendo lo script.

In conclusione, si ha un miglioramento del 60% rispetto all'algoritmo attuale grazie all'utilizzo del multithreading in CPU.

Bibliografia

[1] C++ AMP. http://msdn.microsoft.com/en–us/library/vstudio/hh265137.aspx.

[2] Morton code. en.wikipedia.org/wiki/z-order_curve.

[3] Morton code. https://developer.nvidia.com/content/thinking-parallel-part-iii-tree-construction-gpu.

[4] Jonathan D Cohen, Ming C Lin, Dinesh Manocha, and Madhav Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 189–ff. ACM, 1995.

[5] ComputeDirect. http://msdn.microsoft.com/en–us/library/windows/desktop/ff476331(v=vs.85).aspx.

[6] CUDA. https://developer.nvidia.com/category/zone/cuda–zone.

[7] DirectX. https://it.wikipedia.org/wiki/directx.

[8] Stephen A Ehmann and Ming C Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. In *Computer Graphics Forum*, volume 20, pages 500–511. Wiley Online Library, 2001.

[9] Using C++ AMP from a C# app. http://blogs.msdn.com/b/pfxteam/archive/2011/09/21/10214538.aspx.

[10] S. Gottschalk. Separating axis theorem. Technical Report TR98-024, Department of Computer Science, UNC Chapel Hill, 1996.

[11] Stefan Gottschalk, Ming C Lin, and Dinesh Manocha. Obbtree: a hierarchical structure for rapid interference detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 171–180. ACM, 1996.

[12] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast bvh construction on gpus. In *Computer Graphics Forum*, volume 28, pages 375–384. Wiley Online Library, 2009.

[13] OpenCl. http://www.khronos.org/opencl/.

[14] OpenGl. http://www.opengl.org/.

[15] SlimDX. http://slimdx.org/.