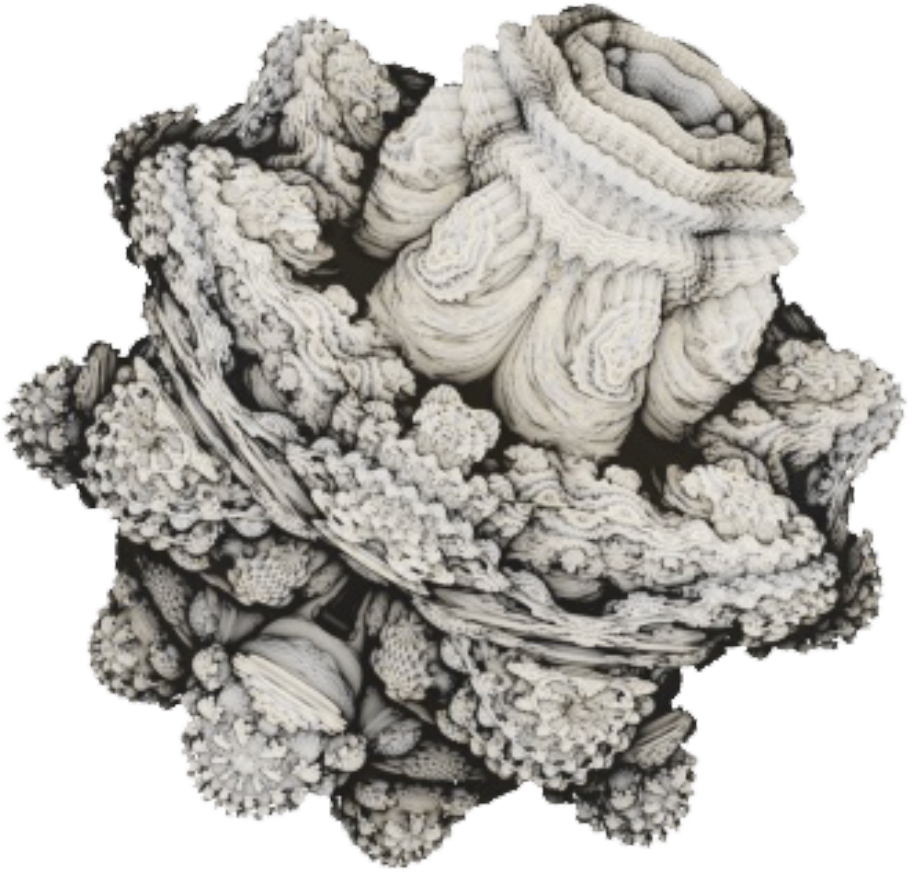
Report Progetto GPU Computing

Ray-marching in CUDA: rendering di frattali 3D



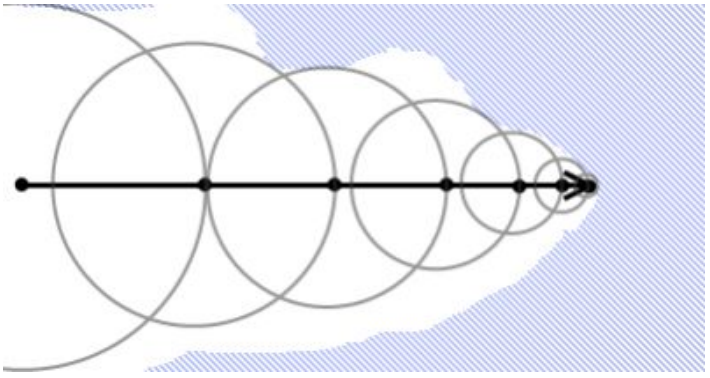
Stefano Pittalis, Manuel Salvadori 2017/2018

1. Descrizione generale del problema

L'obiettivo di questo progetto è la realizzazione di un motore di rendering per la visualizzazione di frattali 3D. A questo scopo abbiamo sfruttato la tecnica del ***ray-marching***, la quale permette di renderizzare oggetti a partire dalla loro forma analitica, a differenza delle tecniche di rendering tradizione le quali si appoggiano invece a delle mesh. Questo fatto rende possibile renderizzare oggetti complessi come i frattali, i quali hanno un livello di dettaglio infinito, che non può essere rappresentato da delle mesh di triangoli.

1.1 Ray-marching e funzione DE (distance estimator)

La tecnica del ray-marching consiste nello sparare, per ogni pixel dell'immagine finale, un raggio nella scena. Questo raggio si muoverà ad ogni iterazione di un valore pari alla distanza dall'oggetto più vicino. Questa distanza è restituita dalla funzione DE, che descrive in maniera analitica la scena nella sua interezza. Quando questa distanza sarà minore di un certa soglia fissata, avremo che il raggio ha colpito un oggetto.



1.2 Colorazione

Il metodo più semplice per colorare un oggetto, è quello di assegnare ad ogni pixel dell'immagine un colore scalato sul numero di iterazione necessarie all'hit. Questo tipo di calcolo del colore simula un ambient occlusion.

1.3 Calcolo delle normali e modello di illuminazione

E' anche possibile applicare un modello di illuminazione standard, che aumenta enormemente il grado di realismo dell'immagine. Per poterlo implementare occorre calcolare le normali della superficie dell'oggetto. Una tecnica è quella del metodo delle differenze finite, che consiste nel trovare un gradiente ottenuto tramite il calcolo della funzione DE in un intorno del punto considerato lungo i tre assi cartesiani. La normale infatti punterà nella direzione dove la DE cresce più rapidamente.

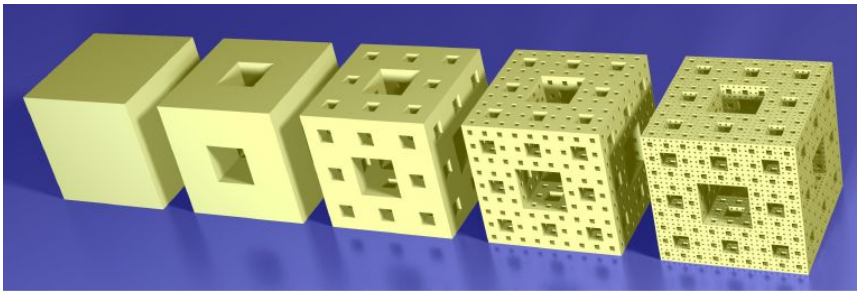
$$n = normalize(DE(p \pm \varepsilon x),\ DE(p \pm \varepsilon y),\ DE(p \pm \varepsilon z))$$

Dopo aver ottenuto le normali è possibile quindi applicare il modello di illuminazione. Nello specifico, abbiamo scelto il semplice Blinn-Phong.

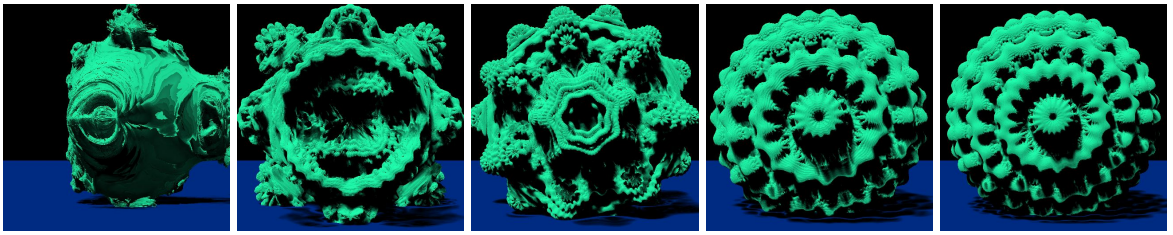
1.4 Tipi di frattali

Nella nostra applicazione è possibile rappresentare un qualsiasi tipo di frattale, basta conoscere la sua formula analitica. Concretamente abbiamo inserito nel programma due tipi di frattali molto conosciuti.

- *Spugna di Menger*
  E' la generalizzazione tridimensionale del tappeto di Sierpinsky. Si tratta di un frattale costruito ricorsivamente partendo da un cubo. Ogni sua faccia viene divisa in una griglia 3x3, formando complessivamente 27 cubi. Vengono poi rimossi i cubi centrali; il processo è poi ripetuto per ogni sotto-cubo ricorsivamente.



- *Mandelbulb*
  E' la generalizzazione tridimensionale del celebre insieme di Mandelbrot:
  $$z = z^n + c$$
  dove z e c sono vettori 3D. L'insieme dei punti 3D che appartengono all'insieme di Mandelbulb sono tutti i c tali per cui z converge a un valore finito dopo un numero fissato di iterazioni. La potenza n definisce il numero di bulbi autosimilari presenti nel frattale:



$n = 2$ $n = 5$ $n = 8$ $n = 16$ $n = 20$

2. Descrizione delle classi/codice

File header:

Application.h	header contenente la definizione della classe Application, che si occupa della gestione di tutta la logica
Bitmap.h	header contenente la definizione della classe Bitmap, che si occupa di salvare un frame in un immagine su disco
Fract.h	header contenente la definizione della classe Fract, che si occupa della computazione del singolo frame
Shapes.h	header contenente le definizioni dei metodi che implementano le DE delle forme geometriche di base
cutil_math.h	libreria *header-only* che contiene funzioni per la manipolazioni di vettori
sdf_util.hpp	header contenente le funzioni DE dei frattali
type_useful.h	header contenente la dichiarazione dei tipi utilizzato e tutte le *#define*

File sorgenti:

Application.cpp	implementazione della classe Application
Bitmap.cpp	implementazione della classe Bitmap
Fract.cu	implementazione della classe Fract
main.cu	file che contiene il main

3. Parallelizzazione CUDA

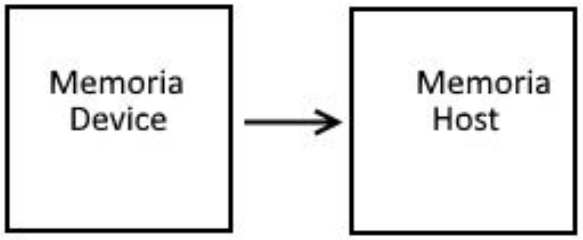
Poiché il ray-marching permette di calcolare il risultato di ogni pixel in maniera indipendente, si presta molto bene ad una parallelizzazione massiccia.

All'interno del nostro progetto abbiamo utilizzato diverse tecniche per migliorare le performance complessive del ray-marching, cercando di utilizzare al meglio le funzionalità offerte da CUDA.

3.1 Allocazione memoria

3.1.1 Memoria Globale

Non essendo necessario tenere in memoria informazioni relative a una texture o una mesh (che non vengono usate in questo approccio), all'inizio dell'applicazione possiamo limitarci ad allocare nella memoria globale del device lo spazio necessario a contenere il risultato della computazione di un singolo frame, senza aver bisogno di trasferire dei dati da host a device; il trasferimento è infatti solo ***unidirezionale***, da device a host.



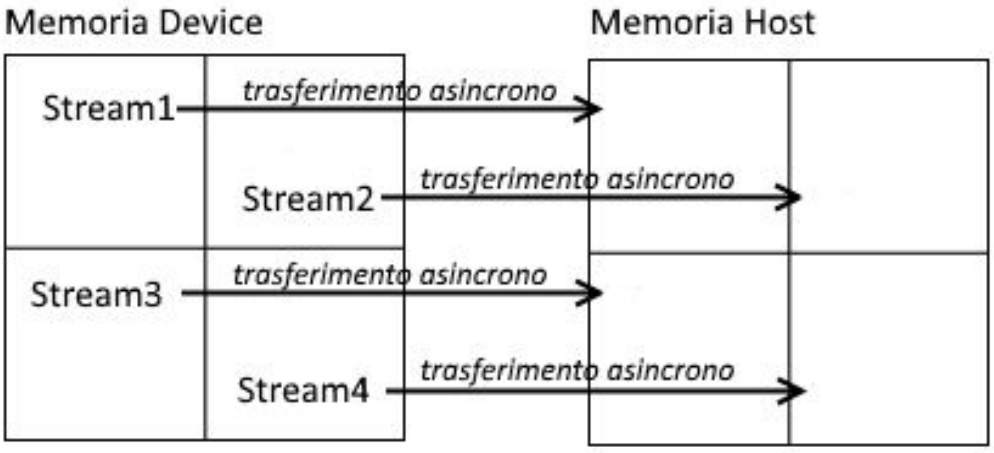
Il trasferimento avviene solo da device a host, e non viceversa

3.1.2 Memoria Pinned

Per rendere il trasferimento da memoria device a host più veloce abbiamo scelto di utilizzare la ***memoria pinned***.
Solitamente quando si trasferiscono i dati dalla memoria host alla memoria del dispositivo, il driver CUDA dapprima alloca temporaneamente la pinned memory nell'host, copia i dati dell'host di origine in pinned memory e quindi trasferisce i dati dalla pinned memory alla memoria del dispositivo.
Allocando direttamente la memoria come pinned saltiamo questo passaggio, garantendo dei trasferimenti più veloci ottenendo un maggiore throughput durante i trasferimenti dell'immagine (la quale, essendo di grandi dimensioni, trae un maggiore beneficio da questa scelta).

3.2 Stream

Per cercare di nascondere il tempo necessario per il trasferimento da device a host abbiamo suddiviso l'area dell'immagine in sotto-blocchi di uguali dimensioni, che sono stati computati da ***stream*** differenti.
In questo modo, non appena un singolo stream ha terminato la computazione, siamo stati in grado di iniziare il trasferimento per quel singolo blocco, ottenendo una sovrapposizione tra la fase di trasferimento e di computazione.



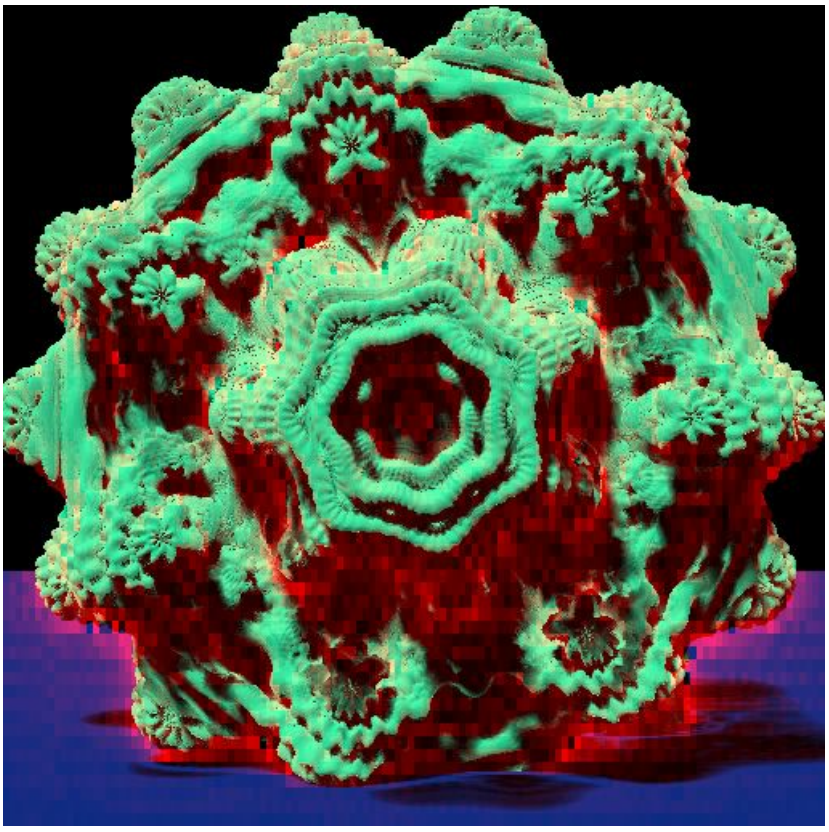
Esempio di trasferimento asincrono usando 4 stream

In ogni stream sono quindi state inserite due operazioni: prima di tutto viene chiamata la funzione *rayMarching*, che si occupa di effettuare i calcoli veri e propri per poter disegnare il frattale. In seguito, sullo stesso stream viene eseguita una *cudaMemcpyAsync*, in modo da poter effettuare il trasferimento in memoria host non appena la computazione giunge al termine.

3.3 Memoria Shared

Nonostante per il nostro progetto non sia strettamente necessario condividere informazioni tra i thread che fanno parte dello stesso blocco, l'utilizzo della memoria shared ci ha permesso di cercare di introdurre un ottimizzazione per velocizzare il calcolo dei frattalli, in particolare il frattale MandelBulb.

A causa della particolare forma di questo frattale, alcune delle zone della figura geometrica sono particolarmente ricche di incavi e sporgenze di piccole dimensioni. Poiché l'algoritmo del ray-marching prosegue finché non abbiamo un hit con un particolare oggetto, queste piccole sporgenze possono richiedere un numero maggiore di iterazioni per essere calcolate rispetto ai propri vicini, andando a causare tempi di calcolo maggiori e ***warp divergency***.



Le zone rosse in immagine sono quelle con tempi di computazione maggiore.

Poiché molto spesso questi dettagli sono di piccole dimensioni, è possibile interrompere precocemente la computazione se la maggior parte dei pixel all'interno dello stesso blocco (80%) hanno finito i propri calcoli: dopodiché, utilizzando l'informazione del colore dei pixel nel vicinato del pixel in considerazione, è possibile impostare il colore facendo la media del colore dei pixel del vicinato.

3.4 Constant memory

La constant memory è una memoria per scopi speciali utilizzata per i dati che sono di sola lettura e accessibili in modo uniforme dai thread in un warp.
La constant memory si comporta meglio quando tutti i thread in un warp leggono dallo stesso indirizzo di memoria.
Per questo motivo, informazioni che non variano nel tempo ma devono essere accedute da tutti i thread, come la posizione della camera, i tre vettori forward, up e right e, nelle scene dove la luce rimane ferma, le coordinate della fonte di illuminazione, sono state inserite in constant memory.

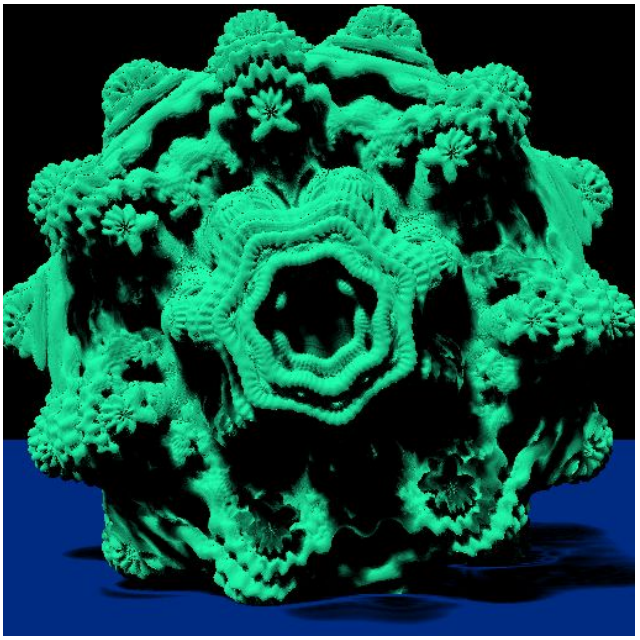
3.5 Kernel Dinamici

Poiché i calcoli necessari per trovare le normali all'interno di un singolo pixel sono tra loro indipendenti, abbiamo pensato di utilizzare i kernel dinamici per cercare di ottenere un ulteriore livello di parallelizzazione intra-pixel.

Purtroppo abbiamo riscontrato che il lancio dei kernel dinamici ha ottenuto l'effetto opposto a quanto ci saremmo aspettati, andando a impattare negativamente il tempo totale di calcolo.
Supponiamo che questo rallentamento sia dovuto all'overhead introdotto con il lancio di un kernel sia maggiore del beneficio che otteniamo parallelizzando le 6 funzioni di calcolo per le normali che, seppur indipendenti tra loro, restano comunque molto veloci da eseguire.

4. Tempi di rendering in base a parametri/tecniche

Il confronto tra le varie tecniche è stato effettuato per il calcolo di una scena contenente un frattale MandelBulb e un Cubo.
La scena contiene un semplice modello di illuminazione, e vengono inoltre calcolate le ombre.
Per velocizzare il testing, confronteremo i tempi di calcolo di un singolo frame di dimensioni ***512x512***.
Le performance dell'algoritmo sono fortemente influenzate al variare di due particolari valori, il ***numero massimo di iterazioni*** per la funzione *distanceExtimator* e una variabile ***epsilon*** che regola il livello di precisione del frattale.
In questo test i valori sono stati impostati rispettivamente a ***128*** e ***0.01***.



Scena di test per la valutazione delle performance.

4.1 Dimensioni blocchi e griglia

Dim Block	Dim Grid	Numero Stream	Transf Asincroni	Filtro Shared Memory	Kernel Dinamici	Tot Time (sec)
4x4	64x64	1	true	false	false	7.868
8x8	***32x32***	***1***	***true***	***false***	***false***	***5.471***
16x16	16x16	1	true	false	false	6.559
32x32	8x8	1	true	false	false	9.651

Le dimensioni che hanno dato le migliori performance sono rispettivamente 8x8 per la dimensione dei blocchi, e 32x32 per le dimensioni della griglia.

4.2 Streams

Dim Block	Dim Grid	Numero Stream	Transf Asincroni	Filtro Shared Memory	Kernel Dinamici	Tot Time (sec)
8x8	***32x32***	***1***	***true***	***false***	***false***	***5.471***
8x8	32x32	4	true	false	false	5.487
8x8	32x32	16	true	false	false	5.475
8x8	32x32	64	true	false	false	5.483

Dim Block	Dim Grid	Numero Stream	Transf Asincroni	Filtro Shared Memory	Kernel Dinamici	Tot Time (sec)
16x16	***16x16***	***1***	***true***	***false***	***false***	***6.559***
16x16	16x16	4	true	false	false	6.586
16x16	16x16	16	true	false	false	6.620
16x16	16x16	64	true	false	false	6.572
16x16	16x16	256	true	false	false	6.687

Dim Block	Dim Grid	Numero Stream	Transf Asincroni	Filtro Shared Memory	Kernel Dinamici	Tot Time (sec)
32x32	8x8	1	true	false	false	9.651
32x32	8x8	4	true	false	false	9.681
32x32	8x8	16	true	false	false	9.625
32x32	8x8	64	true	false	false	9.554
32x32	***8x8***	***256***	***true***	***false***	***false***	***9.432***

La versione con le migliori performance resta quella dove le dimensioni di blocco sono 8x8, e l'uso degli stream sembra aver influito in maniera ridotta sulle performance finali.
Abbiamo potuto notare come l'uso degli stream sia invece andato a migliorare le performance nella versione con dimensioni di blocco maggiore (32x32), arrivando a ridurre i tempi di 0.2 secondi.

4.3 Filtro Shared Memory

Dim Block	Dim Grid	Numero Stream	Transf Asincroni	Filtro Shared Memory	Kernel Dinamici	Tot Time (sec)
8x8	***32x32***	***1***	***true***	***false***	***false***	***5.471***
8x8	32x32	1	true	0.8	false	5.496
8x8	32x32	1	true	0.5	false	5.497

Dim Block	Dim Grid	Numero Stream	Transf Asincroni	Filtro Shared Memory	Kernel Dinamici	Tot Time (sec)
16x16	***16x16***	***1***	***true***	***false***	***false***	***6.569***
16x16	16x16	1	true	0.8	false	6.600
16x16	16x16	1	true	0.5	false	6.580

Dim Block	Dim Grid	Numero Stream	Transf Asincroni	Filtro Shared Memory	Kernel Dinamici	Tot Time (sec)
32x32	***8x8***		***true***	***false***	***false***	***9.660***
32x32	8x8	1	true	0.8	false	9.685
32x32	8x8	1	true	0.5	false	9.688

Il filtro in shared memory purtroppo non ha introdotto un miglioramento nelle tempistiche come sperato; i pixel isolati che richiedono un numero maggiore di iterazioni rispetto al suo vicinato non è in numero tale da portare ad uno speed-up sostanziale nelle prestazioni.

5. Speed-up rispetto all'algoritmo sequenziale

Riportiamo ora la differenza di prestazioni tra la versione sequenziale dell'algoritmo e la migliore configurazione della versione parallela utilizzando la stessa scena di test usata precedentemente:

	Versione Parallela	Versione Sequenziale
Dim Block	***8x8***	-
Dim Grid	***32x32***	-
Numero Stream	***1***	-
Transf Asincroni	***true***	-
Filtro Shared Memory	***false***	-
Kernel Dinamici	***false***	-
Tot Time (sec)	***5.467***	32.818

La versione dell'algoritmo implementata in CUDA ha quindi ottenuto uno speed-up pari a ***(32.818 / 5.467) = 6*** rispetto alla versione sequenziale.
Per cercare di rendere il confronto il più equo possibile sono state usate le stesse funzioni, compilate sia come codice host che device.

6. Filmati

In questa sezione abbiamo inserito i link ad alcuni video di showcase che abbiamo preparato, riportando le informazioni relative alle impostazioni usate, e i tempi di calcolo medi per un frame, e il tempo totale.

6.1 Mandelbulb

https://drive.google.com/open?id=1uvAgw9hbkjX3O3EdB2BQYuH92NOPRRQp

Num Frames	Dim Block	Dim Grid	Numero Stream	Transf Asincroni	Filtro Shared Memory	Kernel Dinamici	Frame Time (sec)	Tot Time (sec)
360	*8x8*	*64x64*	*1*	*true*	*false*	*false*	*5.209*	*1875.359*

6.2 Menger Sponge

https://drive.google.com/open?id=1SKGQ8NqIFY5_6SJwPAnlWPMH2nxC6SE4

Num Frames	Dim Block	Dim Grid	Numero Stream	Transf Asincroni	Filtro Shared Memory	Kernel Dinamici	Frame Time (sec)	Tot Time (sec)
360	*8x8*	*64x64*	*1*	*true*	*false*	*false*	*4.755*	*1711.897*

6.3 Mandelbulb (variazione potenza)

https://drive.google.com/open?id=1jy5evg-4lcEoBXgLr5ZKGhlTnBiYBu1V

Num Frames	Dim Block	Dim Grid	Numero Stream	Transf Asincroni	Filtro Shared Memory	Kernel Dinamici	Frame Time (sec)	Tot Time (sec)
360	*8x8*	*64x64*	*1*	*true*	*false*	*false*	*5.071*	*1825.668*

6.4 Mandelbulb (zoom - veloce)

https://drive.google.com/open?id=1m1eEqEeBeT-YauweJa51FsbuNQB5ASBJ

Num Frames	Dim Block	Dim Grid	Numero Stream	Transf Asincroni	Filtro Shared Memory	Kernel Dinamici	Frame Time (sec)	Tot Time (sec)
360	*8x8*	*64x64*	*1*	*true*	*false*	*false*	*2.384*	*858.453*

6.5 Menger Sponge (zoom)

https://drive.google.com/open?id=1zDJg6Bs0KtdLdRuejqbXUoB89pWLRHDh

Num Frames	Dim Block	Dim Grid	Numero Stream	Transf Asincroni	Filtro Shared Memory	Kernel Dinamici	Frame Time (sec)	Tot Time (sec)
360	*8x8*	*64x64*	*1*	*true*	*false*	*false*	*9.691*	*3489.015*

6.6 Mandelbulb (zoom - lento)

https://drive.google.com/open?id=1HxBC8THqvqp9WuYY8oZWCsyAIzp_hBho

Num Frames	Dim Block	Dim Grid	Numero Stream	Transf Asincroni	Filtro Shared Memory	Kernel Dinamici	Frame Time (sec)	Tot Time (sec)
5000	*8x8*	*64x64*	*1*	*true*	*false*	*false*	*7.642*	*37566.759*

7. Riferimenti

[1] http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm
[2] http://www.iquilezles.org/www/articles/mandelbulb/mandelbulb.htm
[3] http://blog.hvidtfeldts.net/index.php/2011/06/distance-estimated-3d-fractals-part-i