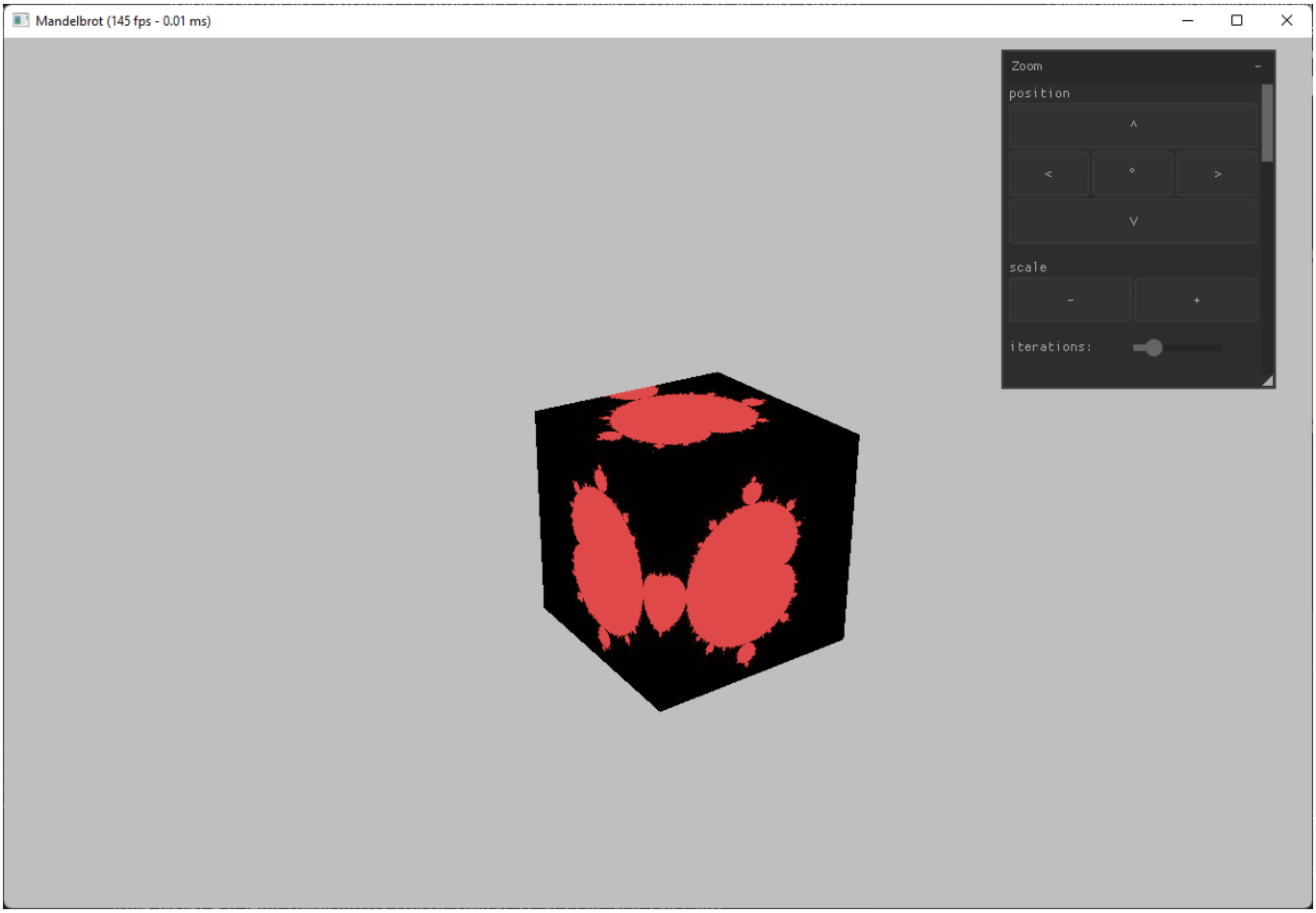
Visualizzatore del frattale di Mandelbrot

Lavoro per Tecnologie
Francesco Viciguerra Classe 4^AIA

Premessa

Questo progetto è un visualizzatore del frattale di Mandelbrot scritto totalmente in C99.
Esso utilizza solo la scheda grafica per determinare i colori dei pixel, quindi evita di di fare calcoli utilizzando la CPU.
Nell'applicazione è presente una modalità di visualizzazione in 2d ed una modalità in 3d, mappando il frattale ad un cubo, sfruttando la potenza della GPU.
Sebbene possa sembrare che il visualizzatore sia la parte principale del progetto, in realtà quest'ultimo serve solo per dimostrare il funzionamento del framework custom con il quale è stato costruito.
Il lavoro è stato pertanto incentrato in tale libreria.



Introduzione

Come spiegato nella premessa, l'applicazione è stata costruita utilizzando una libreria custom. Verranno quindi spiegati prima di tutto le scelta del linguaggio e delle dipendenze utilizzate ed il funzionamento del framework.
Si ricorda che il visualizzatore utilizza la scheda grafica per i propri calcoli, perciò verrà anche fatta una "*veloce*" spiegazione sul come quest'ultimo è renderizzato.
Infine sarà possibile analizzare cosa sono i frattali, per poi analizzare l'app vera e propria.
Si fa notare infine che il codice è *per la maggior parte* commentato (in special modo i file *header*, quindi se si hanno questioni sul funzionamento del programma è possibile sempre consultare il codice sorgente a questo link).

Attraverso lo Specchio

Filosofie e Linguaggi

Prima di tutto è importante sapere che sono sempre stato interessato a linguaggi di programmazione a basso livello (quindi compilati), specificamente con allocazione manuale di memoria. E'sempre stato importante per me sapere esattamente come vengono convertite le informazioni e le funzioni dal codice al linguaggio macchina, dove e come vengono salvati i dati in memoria. Sebbene al giorno d'oggi la maggior parte di linguaggi sono interpretati o utilizzano metodi ottusi per gestire la memoria, ne rimangono sempre alcuni.
Molti utenti spesso pensano che più il linguaggio sia grande e complicato, più sia una migliore soluzione per i loro progetto, ma spesso capita di farsi trasportare dalle features *astratte* dei linguaggi di programmazione più complessi, aggiungendo al proprio codice dettagli e funzionamenti che portano solo più complicatezza ed illeggibilità, in cambio di una soluzione che può sembrare più *intelligente o astratta*.
Un ovvio esempio è l'operator overloading. Si consideri questo codice scritto in cpp, preso da un progetto scolastico:

```
NumeroNaturale f(1);
std::cout << !f << std::endl;
```

Ebbene, sapendo che la classe `NumeroNaturale` rappresenta un numero e che quest'ultima è inizializzata a `1`, quale dovrebbe essere il risultato di `!f`? Secondo pura logica si dedurrebbe che `!1` = `0`. Questa assunzione è errata. L'operatore `!` in questo codice ritorna il fattoriale del numero. Questo comportamento è impossibile da determinare senza precedente conoscenza del programma. C'è una chiara mancanza di leggibilità.
Questo non è il solo problema: non è chiaro se sia in primo luogo chiamata un funzione che possa modificare i valori nel codice. Secondo la mia filosofia il linguaggio di programmazione perfetto avrebbe bisogno delle seguenti caratteristiche:

- **semplicità**. L'intero linguaggio deve poter essere imparabile da un comune mortale;
- **chiarezza nella chiamata delle funzioni**: se un codice non sembra che debba chiamare una procedura non deve chiamarla;
- **chiarezza e consistenza nello stile**: un linguaggio dovrebbe essere bello da vedere e non ci si dovrebbe mai chiedere cosa un determinato simbolo o espressione faccia.
- **un programma serve solo a trasformare dati in altri dati. Un dato non è altro che uno spazio in memoria.**

Di tutti i linguaggi da me analizzati il più vicino che si avvicina a questa ideologia è il linguaggio C. Si aggiunge però che quest'ultimo è lontano dall'essere perfetto. Il suo più grande problema è la mancanza dei template (che tutta via può essere ottenuto con l'utilizzo delle macro).

Sono stati considerati e scartati i seguenti linguaggi:

- `BetterC subset of D`: la migliore alternativa, ma, come D, è molto incentrato sul concetto di RAII (Resource Acquisition Is Initialization), che non considero necessario per il mio stile di programmazione.
- `Zig`: un linguaggio con grande enfasi sull'esecuzione di codice al tempo di compilazione e sull'integrazione con il linguaggio C. Tuttavia utilizza una sintassi ed un module-system non ottimali e pieni di piccole stranezze. Inoltre è ancora un progetto immaturo, non ancora adatto a grandi lavori.
- `Odin`: il linguaggio più simile al C. Utilizza una sintassi molto innovativa e semplice. Tuttavia manca di un sistema che possa possa sostituire o porre una viabile alternativa alle macro del linguaggio C. Infine, come Zig, è ancora immaturo.
- `C3`: evoluzione spirituale del linguaggio C. Prova ad espanderlo mantenendone la stessa filosofia. Tuttavia penso che si sia spinto troppo lontano dai concetti originari. Come Zig ed Odin è ancora immaturo.
- `Jai`: il linguaggio che più si avvicina alle mie caratteristiche ideali. Utilizza una sintassi molto simile ad Odin e permette di eseguire grandi parti di codice al tempo della compilazione (addirittura un intero gioco). Tuttavia è ancora in sviluppo ed è in closed beta.

Si ricorda infine che il linguaggio C ha vari compilatori, varianti e dialetti. Per questo progetto è stato utilizzato il compilatore `gcc`. Ed è stato scelto il dialetto `C99`, in quanto è abbastanza moderno e non ho motivo per utilizzare standard più moderni. L'unica feature utilizzata non conformante agli standard è l'uso dell'operatore `typeof`, che è utilizzato solo per migliore debugging e può essere disabilitato attraverso un `#define`.

Dipendenze

CMake e Ninja

Il linguaggio di programmazione C, in termini di sistema di build, è molto datato.
Senza sistemi esterni creare un eseguibile con file molteplici e librerie esterne è tedioso: è necessario indicare al compilatore le locazioni dei file esterni e delle librerie, per poi compilare ogni singolo sorgente in molteplici file oggetto. Allora bisogna effettuare un processo chiamato linking, prestando attenzione ad includere i giusti file ed utilizzare i giusti parametri.
Molti linguaggi moderni sono in grado di fare questi passaggi in modo trasparente, ma il linguaggio C non ne è in grado.
Viene in aiuto allora un tool esterno chiamato CMake che permette di, attraverso file di configurazione nominati `cmakelists.txt`, compilare facilmente ed in maniera trasparente. Si fa notare inoltre che CMake è in grado di utilizzare molti compilatori e build-system.
Nel progetto è stato utilizzato, grazie a CMake, un build-system chiamato ninja. Quest'ultimo permette di compilare in parallelo il progetto, migliorando abissalmente i tempi di compilazione. Tutto il progetto, comprese librerie esterne, è perciò molto veloce a generare un eseguibile: sono necessari meno di 15 secondi sul mio PC di lavoro.
Si ricorda inoltre che la compilazione progressiva è supportata sia da CMake che da ninja, perciò non è

necessario ricompilare i file che non sono stati cambiati, migliorando ulteriormente i tempi per le compilazioni successive alla prima.

Dare super poteri al linguaggio C: VxUtils

VxUtils è l'unica libreria esterna creata dal sottoscritto.
Quest'ultima serve per ampliare il linguaggio c, con intelligente uso di macro o di features non molto conosciute, ma conformanti agli standard. Oltre a questo offre molteplici funzioni e strutture per facilitare la risoluzione di certi problemi.
La libreria porta le seguenti features:

- migliori tipi, come `u32`, `i16` o `f32`, per poter determinare la grandezza in memoria facilmente;
- funzioni per la facile lettura di file;
- utilities per la gestione della memoria (in particolare versioni safe di `malloc` e `realloc`);
- migliore chiarezza per i puntatori a funzione, insieme a funzioni per renderli più sicure. Per esempio:

```
  i32 add(f32 a, i32 b) {
      return b + a;
  }

  i32 main() {
      /* func è un puntatore a funzione che ritorna un i32 e prende un f32
ed un i32. */
      /* Internamente il tipo viene convertito in `int (*func)()`. */
      VX_CALLBACK(func, i32, f32, i32) = add;

      /* Per maggiore chiarezza è anche possibile nominare i parametri del
callback. */
      VX_CALLBACK(func2, i32, f32 a, i32 b) = add;

      /* Può capitare che un puntatore a funzione sia NULL, come nel
seguente caso: */
      VX_CALLBACK(illegal_fn, void) = NULL;
      /* La seguente funzione farebbe quindi crashare il programma se
lasciata non commentata. */
      // illegal_fn();

      /* `VX_SAFE_FUNC_PTR` è in grado di convertire tale puntatore in uno
che possa essere chiamato. */
      illegal_fn = VX_SAFE_FUNC_PTR(illegal_fn);
      /* Ora `illegal_fn` è ora chiamabile. */
      illegal_fn();
  }
```

- migliori controlli e asserzioni per la modalità release e per la modalità debug. Per esempio:

```
  typedef struct Stuff {
      i32 a;
  } Stuff;
```

```
  i32 stuff_func(Stuff* stuff) {
      /* In questa funzione stuff non può essere NULL: deve essere
inizializzato. E'possibile utilizzare `VX_NULL_ASSERT` che crea un eventuale
errore a run time se in modalita'di debug. */
      VX_NULL_ASSERT(stuff);
      /* Similarmente se si preferisce che il codice non si fermi è
possibile utilizzare `VX_NULL_CHECK`, che ritornerà eventualmente un valore
di default, non prima di aver inviato un messaggio di warning. */
      VX_NULL_CHECK(stuff, 0);

      return stuff->a;
  }

  void func() {
      /* Questa funzione non è stata completata dal programmatore. Se
chiamata quest'ultima farà cashare il programma. */
      VX_UNIMPLEMENTED();
  }

  i32 main() {
      /* Questo main mostrerà anche controlli più semplici. */
      i32 a = 0;
      a += 1;
      /* Errore condizionale. */
      VX_ASSERT("a non e'1", a == 1);

      a = 3;
      if (a==4) {
          /* Errore non condizionale. */
          VX_PANIC("Non dovremmo essere qui!");
      }

      VX_DBG_PANIC("Questo messaggio verra' mostrato solo in
modalita'debug!");
  }
```

- macro per la *facile* creazione di funzioni e strutture dati che necessitano di un template. Questo viene ottenuto utilizzando l'operatore di concatenazione ##. Alcuni esempi sono:

```
  /* Creazione di una struttura generica. Il nome della struttura generata
sara' `Struct_T` dove T e'il nome del tipo. Per esempio con u32:
`Struct_u32`. */
  #define _STRUCT_ELEM(T) typedef struct VX_TEMPLATE_NAME(T, Struct) {\
      T elem;\
  }

  /* Creazione di una funzione per ottenere elem da Struct. Il nome della
funzione generata sara' `struct_get_elem_T`, dove T e'il nome del tipo. Per
esempio con u32: `struct_get_elem_u32`. */
  #define _STRUCT_GET_ELEM_INL(T) T VX_TEMPLATE_NAME(T, struct_get_elem)
```

```
(VX_TEMPLATE_NAME(T, Struct)* struct) {\
      return struct->elem;
  }

  /* Macro di aiuto per implementare un tipo: */
  #define _GEN_STRUCT_FOR_TYPE(T) _STRUCT_ELEM(T) \
      _STRUCT_GET_ELEM_INL(T)

  /* Implementazione per `u32`. */
  _GEN_STRUCT_FOR_TYPE(u32)

  /* Main per testing. */
  i32 main() {
      /* VX_T e'la versione piu'corta di VX_TEMPLATE_NAME. */
      VX_T(u32, Struct) s;
      s.elem = 30;

      VX_ASSERT("Test!", VX_T(u32, struct_get_elem)(&s) == 30);
  }
```

- un implementazione di un vettore dinamico e di una hashmap entrambi con template. Per esempio:

```
i32 main() {
  /* Creazione di un vettore di interi. */
  VX_T(i32, vx_Vector) vec = VX_T(i32, vx_vector_new)();

  /* Inserimento di dati nel vettore. */
  VX_T(i32, vx_vector_push)(&vec, 5);
  VX_T(i32, vx_vector_push)(&vec, 3);
  VX_T(i32, vx_vector_push)(&vec, 1);

  /* Ciclo foreach. la variabile I viene definita dalla macro. */
  VX_VECTOR_FOREACH(i32, elem, &vec,
    printf("elemento %d: %d\n", I, elem);
  )

  /* Ottenimento del secondo elemento. VX_VD. */
  VX_ASSERT("Test", VX_VD(&vec)[1] == 3);

  /* Deinizializzazione del vettore. */
  VX_T(i32, vx_vector_free)(&vec);

  /* Creazione di un'HashMap di interi. La key e'sempre un intero unsigned a
64 bit.  */
  VX_T(i32, vx_HashMap) map = VX_T(i32, vx_HashMap)();

  /* Inserimento valori. Il primo parametro è il valore, il secondo è una
chiave. */
  VX_T(i32, vx_hashmap_insert)(&map, 100, 0);
  VX_T(i32, vx_hashmap_insert)(&map, 999, 3);

  /* Richiesta valori. */
```

```
    VX_ASSERT("Test!", VX_T(i32, vx_hashmap_get)(&map, 3) == 999);

    /* Rimozione valori. */
    VX_T(i32, vx_hashmap_remove)(&map, 3);

    /* deinizializzazione della memoria. */
    VX_T(i32, vx_hashmap_free)(&map);
}
```

GLFW

GLFW è una libreria che fornisce un'API di basso livello per gestire l'apertura e gli input di una finestra