Corso di
Grafica Computazionale

STL & VCG

Docente:

Massimiliano Corsini

Laurea Specialistica in Ing. Informatica

Facoltà di Ingegneria
Università degli Studi di Siena

Introduzione

- Alla fine degli anni 70 Alexander Stepanov fu uno dei primi esperti di linguaggi di programmazione ad osservare che molti algoritmi non dipendono dalle specifiche strutture dati su cui lavorano ma da poche proprietà semantiche dei dati.
- Ad esempio per implementare un algoritmo di ordinamento non è essenziale il fatto che esso operi su un vettore o su una lista di elementi.
- Stepanov esaminò un certo numero di algoritmi e trovò che molti di essi potevano essere astratti in modo da non perdere l’efficenza. .

Nascita dell'STL

- Qualche anno dopo, Alexander Stepanov and Meng Lee svilupparono un enorme libreria - la *Standard Template Library (STL)* – con l'intenzione di dimostrare che era possibile sviluppare algoritmi generici senza perdere in efficenza.
- La libreria STL fu inserita nello standard ANSI/ISO C++ nel 1994 dallo Standards Committee del linguaggio..

Funzioni Template

- Consideriamo una funzione che scambia interi:
  - ```
    void swap (int& a, int& b) {
      int tmp = a;
      a = b;
      b = tmp;
    }
    ```
- E supponiamo di volere scrivere una funzione per scambiare floating point, una per scambiare double, ecc → dovremmo duplicare un sacco di codice.
    ```

Funzioni Template

- I templates ci permettono di scrivere una funzione di swap generica astraendo dal tipo di dato da scambiare:
  - ```
    template <class T>
      void swap (T& a, T& b) {
      T tmp = a;
      a = b;
      b = tmp;
    }
    ```
- "T" è il parametro del template. Avrei potuto indifferentemente scrivere "Dato", "TipoDato", ecc.
    ```

Funzioni Template

- Un altro esempio di funzione template:
  - ```
    template <class T>
    T& min (T& a, T&b)
    {
      return a < b ? a : b;
    }
    ```
    ```

Classi Template

- **Analogamente a quanto visto per le funzioni è possibile parametrizzare secondo il tipo di dato anche le classi.**

- Ad esempio:

```
template <class T>
class vector {
  T* v;
  int sz;
  public:
  vector (int s) { v = new T [sz = s]; }
  ~vector () { delete[] v; }
  T& operator[] (int i) { return v[i]; }
  int get_size() { return sz; }
};
```

Classi Template

- Non ci interessano i dettagli implementativi sul come scrivere una classe template.
- Il nostro obiettivo è imparare ad utilizzare classi con il tipo di dato parametrizzato per poter usare alcune parti dell'STL (come i vettori) e capire la VCG almeno a livello di sintassi.

Standard Template Library

- STL è una libreria costituita da diverse componenti che ci permettono, grazie alla generalizzazione, di scrivere molto meno codice per sviluppare algoritmi complessi.
- Le componenti della libreria STL sono:
  - Algoritmi
  - Contenitori
  - Iteratori
  - Oggetti Funzione
  - Adattatori

Contenitori

- STL mette a disposizione due tipi di contenitori: *contenitori di sequenze* e *contenitori associativi*.
- Contenitori di Sequenze:
  - Vector
  - List
  - Deque (Double Ended Queue)
- Contenitori Associativi:
  - Set
  - Map

Contenitori - Vettori

- Per utilizzare un contenitore ***vector*** si deve includere **`#include <vector>`**
- Esempio:
  - **`vector<float> v1;`**
  - **`vector<MyObject> v2;`**
  - **`vector<MyObject *> v3;`**
- Inserimento nel vettore:
  - **`v1.push_back(2.3f);`**
  - **`MyObject obj;`**
    `v2.push_back(obj);`
  - **`v3.push_back(new MyObject());`**

Contenitori - Vettori

- Posso sapere quanti elementi il vettore contiene utilizzando vector::size().
- Esempio:
  - `int numelements = v1.size();`
- Posso accedere ad un qualsiasi elemento del vettore utilizzando l'operatore [].
- Esempio:
  - ```
    MyObject *element;
    element = v3[2];
    ```
    ```

Contenitori - Liste

- Se anzichè organizzare i miei dati in un vettore volessi utilizzare una lista per velocizzare gli inserimenti e le cancellazioni (a scapito della velocità di accesso agli elementi) potrei utilizzare il contenitore ***list***.
- In questo caso per l'accesso non ho a disposizione l'operatore [] ma devo utilizzare gli ***iteratori***.

Iteratori

- Sono dei ***"puntatori generici"*** che permettono di svincolarsi dal tipo di dato puntato.
- Esistono vari tipi di iteratori
  - Random Access iterators
  - Bidirectional iterators
  - Forward iterators

Iteratori (esempio)


```
vector<int> myvett;
vector<int>::iterator itVettInt;
for (itVettInt = myvett.begin();
      itVettInt != myvett.end(); itVettInt++)
{
  std::cout << *itVett << std::endl;
}
-----------------------------------------
list<int> mylist;
list<int>::iterator itList;
for (itListInt = mylist.begin();
      itListInt != mylist.end(); itListInt++)
{
  std::cout << *itList << std::endl;
}
```

Algoritmi

- Solitamente gli algoritmi STL richiedono gli iteratori dei dati su cui devono lavorare.
- Esempio (ordinamento di un vettore):
  - `sort(myvett.begin(), myvett.end())`

Memory Management

- Per ripulire un contenitore dalla memoria si può utilizzare il metodo *::erase(iteratore_inizio, iteratore_fine)*, oppure *::clear()*
- Esempio:
  - ```
    vector<MyObject> v1;
    // ... v1.push_back(...);
    v1.clear();
    ```
  - ```
    vector<MyObject *> v2;
    // ... v2.push_back(new MyObject());
    v2.clear(); // e gli oggetti puntati che fine fanno?
    ```

Libreria VCG

- La VCG è composta interamente da classi template, per poter astrarre dal tipo di mesh (più precisamente dal tipo di complesso simpliciale) su cui si lavora.
- Per esempio voglio poter gestire genericamente mesh con attributi dei vertici diversi.

VCG – Definire una mesh


```
class MyEdge;    // forward declaration (mai usato)
class MyFace;    // forward declaration

// definiamo un vertice passando gli attributi che
   vogliamo che abbia (vedi
   vcg/simplex/vertexplus/component.h per gli attributi
   predefiniti)
class MyVertex : public VertexSimp2< MyVertex,
    MyEdge, MyFace, vert::Coord3f, vert::Normal3f > {};

// Analogamente per le facce
class MyFace: public FaceSimp2 < MyVertex, MyEdge,
    MyFace, face::VertexRef, face::Normal3f > {};
```

VCG – Definire una mesh

- A questo punto si può definire la propria mesh come un contenitore di vertici ed un contenitore di facce:
  - `class CMesh: public`

    `vcg::tri::TriMesh< vector<CVertex>, vector<CFace> > {};`

VCG – Import/Export

- Come caricare un modello 3D generico:
  - ```
    if (vcg::tri::io::Importer<CMesh>::Open(
    mesh, "model.ply") !=0 )
    {
      // Error (!!)
    }
    ```
- Formati supportati: PLY, 3DS, OBJ, STL, OFF, ...
- **N:B:** Per il supporto dei file in formato PLY è necessario aggiungere al progetto il file *plylib.cpp*.
    ```

VCG – Tipi di dato base

- Altri tipi base presenti nella VCG che possono risultare di interesse sono:
  - Point2<...> - Punti in 2D
  - Point3<...> - Punti in 3D
  - Matrix33<...> - Matrici 3x3
  - Matrix44<...> - Matrici 4x4

VCG
Estrazione dati ed invio alla pipeline


```
#include “mymesh.h”
// ...
MyMesh m;

FaceIterator itF;
VertexPointer v1,v2,v3;

glBegin(GL_TRIANGLES);
for (itF = m.faces.begin(); itF != m.faces.end();
itF++)
{
    // (*itF) accedo alla faccia deferenziando l’iteratore
    // V(0) è il primo vertice della faccia, V(1) il secondo, ecc.

    vcg::glTexCoord((*itF).V(0)->T());  // vcg::glTexCoord è una funzione della VCG e non
    vcg::glNormal((*itF).V(0)->N());    // dell’OpenGL!! Essa richiama a sua volta l’OpenGL
    vcg::glVertex((*itF).V(0)->P());    // con i parametri corretti, ad esempio T() ritorna
    vcg::glTexCoord((*itF).V(1)->T());  // le coordinate sotto forma di Point2<> e non ho
    vcg::glNormal((*itF).V(1)->N());    // una glVertex opportuna in OpenGL).
    vcg::glVertex((*itF).V(1)->P());    // Analogamente vcg::glNormal e vcg::glVertex
    vcg::glTexCoord((*itF).V(2)->T());  // sono wrapper delle rispettive funzioni OpenGL.
    vcg::glNormal((*itF).V(2)->N());
    vcg::glVertex((*itF).V(2)->P());
}
glEnd();
```

VCG – Utilizzo base

- Per un'implementazione dell'utilizzo base della VCG fate riferimento al programma ***trimesh_sdl***.

Domande?