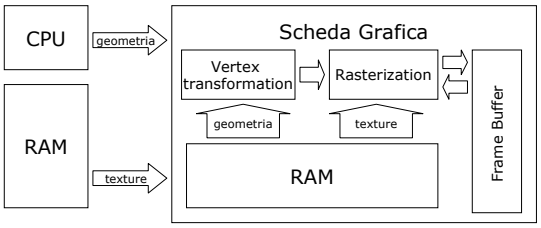
Costruzione di Interfacce
Lezione 23
Scene Graphs, Object Loading

cignoni@iei.pi.cnr.it
http://vcg.iei.pi.cnr.it/~cignoni

Migliorare la performance

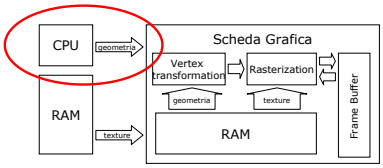
- Se l'obiettivo è l'interattività è necessario riuscire a mantenere un frame rate di almeno 10~20 fps.
- Perché un'applicazione opengl va piano?
- Cercare i possibili colli di bottiglia
  - CPU
  - Trasformazioni
  - Rasterizzazione

Architettura Hw



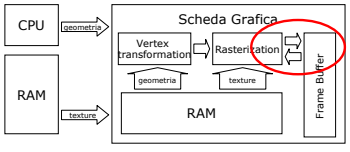
Scenario 1

- Molta geometria mandata con glvertex
- Triangoli piccoli
- No texture
- Indipendente dimensioni finestra



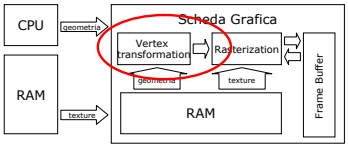
Scenario 2

- Pochi glvertex
- Triangoli grandi
- Con o senza texture
- Dipendente dimensioni finestra



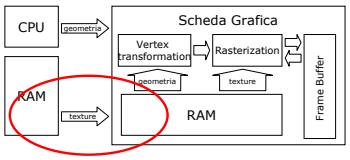
Scenario 3

- Molti Vertici mandati con path preferenziali (display list, compiled vertex array)
- Triangoli piccoli
- poca texture
- Indipendente dimensioni finestra



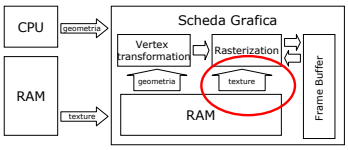
Scenario 4

- Medio numero di triangoli, mandato bene
- Triangoli piccoli
- molta texture >> memoria scheda grafica
- Dipendente dimensioni finestra (poco)



Scenario 5

- Medio numero di triangoli, mandato bene
- Triangoli piccoli
- molta texture << memoria scheda grafica
- Dipendente dimensioni finestra (abbastanza)



Benchmarking

- Tenere traccia del tempo di rendering durante lo sviluppo dell'applicazione.
  - Se va improvvisamente piu' piano cercare di capire perchè subito.
- Tecniche principali per capire come mai va piano:
  - glCullFace(GL_FRONT_AND_BACK)
  - Window sizing
- La fase di rasterizzazione sparisce o pesa meno

Migliorare l'efficenza

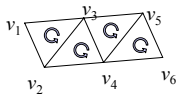
- Non tutti i modi per disegnare sono egualmente veloci.
- Function Call Overhead
  - glColor3f(…); / glTexture2f(…)
  - glNormalf(…);
  - glVertex(…);
- Per specificare un triangolo 9 chiamate di funzione.

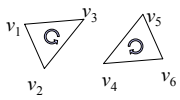
Triangle Strip

- Modo efficiente per disegnare una serie di triangoli:

```
glBegin(GL_TRIANGLE_STRIP);
     glVertex3fv(v1); glVertex3fv(v2);
     glVertex3fv(v3); glVertex3fv(v4);
     glVertex3fv(v5); glVertex3fv(v6);
glEnd();


glBegin(GL_TRIANGLES);
     glVertex3fv(v1); glVertex3fv(v2);
     glVertex3fv(v3); glVertex3fv(v4);
     glVertex3fv(v5); glVertex3fv(v6);
glEnd();
```





- Nel caso ottimo si manda un solo vertice per triangolo.

Display List

- Meccanismo per "cachare" una serie di operazioni di rendering
- Una display list e' una sequenza di comandi opengl che viene memorizzata sulla memoria della scheda grafica per poter poi essere nuovamente eseguita rapidamente.
- Ogni display list e' identificata da opengl con un intero (stesso meccanismo degli identificatori di texture)

Display List

- Allocazione
  - alloca "n" liste consecutive richiamabili con gli interi dli..dli+n-1

```
dli=glGenLists(n);
```

- Disallocazione

```
glDeleteLists(dli,n);
```


Display List

- Generazione

```
glNewList(dli,mode);
..Sequenza Comandi opengl..
glEndList()
```

  - genera la display list `dli`, mode puo essere
    - GL_COMPILE
    - GL_COMPILE_AND_EXECUTE (pericolosa su alcune schede)
- Chiamata

```
glCallList(dli);
glCallLists(dli,n);
```


Display List

- Difetti Display List:
  - sono statiche
  - gli oggetti vengono tenuti in memoria due volte.
  - Se non entrano nella memoria della scheda video possono non essere molto efficienti
- Pregi
  - danno la possibilita' ad opengl di convertire tutti i dati nel formato piu' conveniente

Vertex Array

- In Opengl 1.1 si puo compattare tutte i dati da passare alle varie primitive opengl in un unico vettore.
  - Si deve dichiarare quali vettori si vuole usare
    - vertex
    - color
    - normal
    - texcoord
  - come sono fatti
  - e abilitarli con

```
glEnableClientState (GL_VERTEX_ARRAY);
glEnableClientState (GL_COLOR_ARRAY);
glEnableClientState (GL_NORMAL_ARRAY);
```


Vertex Array

- Per specificare un vettore di vertici

```
glVertexPointer(int n,TYPE,int stride,void *data);
```

  - dove **n** =2,3,4 indica quante coordinate si specifica per ogni vertice
  - **TYPE** puo essere GL_FLOAT, GL_DOUBLE, ecc
  - **stride** indica quanti byte ci sono tra un vertice e il seguente, zero significa che sono paccati strettamente e il vettore contiene solo le coordinate)
  - **data** e' un puntatore al vettore in questione

Vertex Array

- Similarmente si specificano
  - colori

```
glColorPointer(N,TYPE,stride,data);
```

  - normali

```
glNormalPointer(TYPE,stride,data);
```

Vertex Array

- Vertici normali e colori possono anche essere tutti in uno stesso vettore:

```
class Vertex {public:
  float v[3];
  float n[3];
  float c[3];
};
Data Vertex[n];

glVertexPointer(3,GL_FLOAT,sizeof(Vertex),&(Data.v[0]));
glColorPointer(3,GL_FLOAT,sizeof(Vertex), &(Data.c[0]));
glNormalPointer(GL_FLOAT,sizeof(Vertex),&(Data.n[0]));
```


Vertex Array

- Per disegnare utilizzando gli array definiti:
- `glDrawArrays(primitive,int start,int end);`
  - Dove:
  - primitive e' uno dei soliti GL_TRIANGLES, GL_LINES, ecc
  - start e end dicono quali elementi del vettore usare.

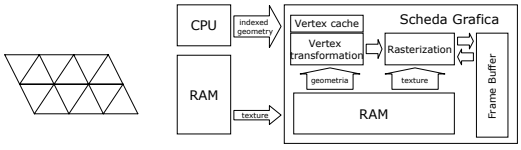
Indexed Vertex Array

- Per disegnare utilizzando gli array in maniera ancora piu' efficiente:
- `glDrawElements(primitive,int count,glenum Type, void *indices);`

  Dove:
  - primitive e' uno dei soliti GL_TRIANGLES, GL_LINES, ecc
  - Count, type e indices definiscono un vettore di indici di vertici relativi ad un array definito precedentemente.

Indexed Vertex Array

- Possono essere MOLTO piu' efficienti perché sfruttano le vertex_cache della gpu
- Coda con gli ultimi 8/16 vertici trasformati
- Peak performance di trasformazione: .5 vertici per triangolo.



Vertex Array

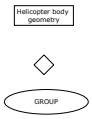
- Difetti
  - Possono, in alcuni casi, essere meno efficienti delle display list
- Pregi
  - Non occupano memoria aggiuntiva
  - l'utente puo' cambiare la mesh continuamente.
  - Possono essere allocati direttamente nella memoria della scheda grafica (usando qualche estensione di opengl...)

Scene Graphs

- Modellazione: processo di definire e organizzare un insieme di geometrie che rappresentano una particolare scena
- Non è facile, specialmente se lo voglio fare dinamicamente (come nei giochi)
- Definire un qualche formalismo che mi permetta di modellare in forma strutturata (e possibilmente OO) significa, di solito definire la sintassi di un qualche scene graph.

Scene Graph

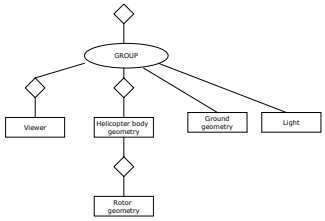
- La scena è strutturata con un albero (o con un dag)
- Tipi principali di nodi
  - Nodi shape
    - Geometry content
    - Appearance content
  - Trasformazioni
  - Group



- Moltissimi altri nodi sono un po' meno comuni (bounding entities, animating entities, agents etc).

Esempio

- Un elicottero che vola su di un terreno



Scene Graphs

- Il rendering corrisponde alla visita della struttura
- Le trasformazioni relative ad ogni sottoalbero sono isolato da una coppia
  - Pushmatrix
  - Popmatrix
- I nodi appearance sono quelli che modificano lo stato di opengl
  - Materiali
  - texture, glenabling vari ecc

Scene Graphs

- E`un concetto. Non esiste uno standard unico.
- Gli engine dei videogiochi sono, essenzialmente scene graphs.
- La struttura dello scene graph è spesso sfruttata anche da
  - AI agenti in un gioco
  - Simulazione fisica ambiente
  - Collision detection
  - Frustum culling

Scene Graphs

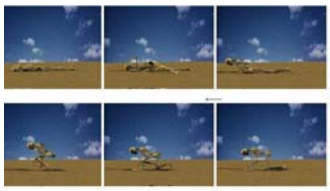
- Nomi
  - Inventor SGI
  - VRML 2.0
  - Java3d
  - Diversi progetti open source
  - Non esiste la soluzione definitiva

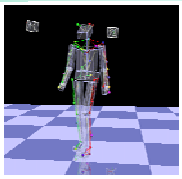
Quake2 format

- Formato molto semplice, tipico esempio di formato da gioco.
- Obiettivi:
  - Oggetti piccoli
  - Textured
  - Animati secondo un certo numero di movimenti predefiniti
  - Sia molto compatto

Animazioni in 3d

- Generazione di animazioni
  - Keyframed
  - Motion Capture
  - Procedurally generated





Quake2 format a grandi linee

- Topologia costante (l'insieme dei triangoli è sempre lo stesso cambiano solo le posizioni dei vertici
- Per ogni frame dell'animazione si salva tutti i vertici (vertici e normali)
- Ogni frame dell'animazione ha un nome ed è relativo ad un piccolo loop (e.g. leggendo I nomi dei frame si capisce che i frame della corsa vanno da 40 a 45)
- Posizioni dei vertici compresse (3byte per vertice)
- Normali compresse 1 byte per vertice (indice in una tabella di normali)

Quake MD2

- Contiene un formato compatto di comandi opengl per fare il rendering in maniera efficiente
  - Sequenze di strip e fan
- Classe CIMd2
  - Load(file, texture);
  - SetAnim(range dei frame da vedere)
  - SetFPS (velocita' di rendering)
  - DrawTimedFrame(currtime);

Loading 3ds

- 3ds formato del 3d studio versione fino alla 4.0 (poi ha iniziato a chiamarsi MAX)
- Molto diffuso, ma con forti vincoli interni (e.g. ogni singolo pezzo non puo' contenere più di 65k vertici)
- Strutturato:
  - rappresenta l'oggetto con una specie di scene graph;
  - Più difficile da usare se voglio importare oggetti complessi.

Libreria L3DS

- Una delle tante. Forse la più piccola in cpp.
- Tutto incapsulato in un oggetto che una volta caricato un file 3ds contiene una lista di mesh con normali e attributi materiale.
- Compito dell'utente gestire il materiale di ogni mesh in maniera corretta…
- Non è robustissima…

L3D Disegno

```
for (uint i= 0; i<pd->mod3ds.GetMeshCount(); i++)
 {
   LMesh &mesh = pd->mod3ds.GetMesh(i);
   glBegin(GL_TRIANGLES);
    for(uint j=0;j<mesh.GetTriangleCount();++j)
    {
     LTriangle2 t=mesh.GetTriangle2(j);
     uint im=mesh.GetMaterial(tt.materialId);
     LColor3 mm= pd->mod3ds.GetMaterial(im).GetDiffuseColor();
     glColor3f(mm.r,mm.g,mm.b);
     glNormal3f(t.vertexNormals[0].x,t.vertexNormals[0].y,t.vertexNormals[0].z);
     glVertex3f(t.vertices[0].x,t.vertices[0].y,t.vertices[0].z);
     glNormal3f(t.vertexNormals[1].x,t.vertexNormals[1].y,t.vertexNormals[1].z);
     glVertex3f(t.vertices[1].x,t.vertices[1].y,t.vertices[1].z);
     glNormal3f(t.vertexNormals[2].x,t.vertexNormals[2].y,t.vertexNormals[2].z);
     glVertex3f(t.vertices[2].x,t.vertices[2].y,t.vertices[2].z);
    }
   glEnd();
 }
```

Conclusioni

- Adesso abbiamo quasi tutti i gli elementi di base che ci servono.
- Occorre strutturare meglio la parte3d (uno scene graph? Forse.)
- La OnDraw è lunga e illeggibile.
- Manca il salvataggio
- Manca molta molta interfaccia, troppe cose hardwired nel codice