RELAZIONE DEL PROGETTO DI ELEMENTI DI INFORMATICA GRAFICA DELL'APPELLO DI NOVEMBRE 2012

DI SIMONE CELIA (722920)

UNIVERSITA' DEGLI STUDI DI MILANO BICOCCA

Indice generale

UNIVERSITA' DEGLI STUDI DI MILANO BICOCCA

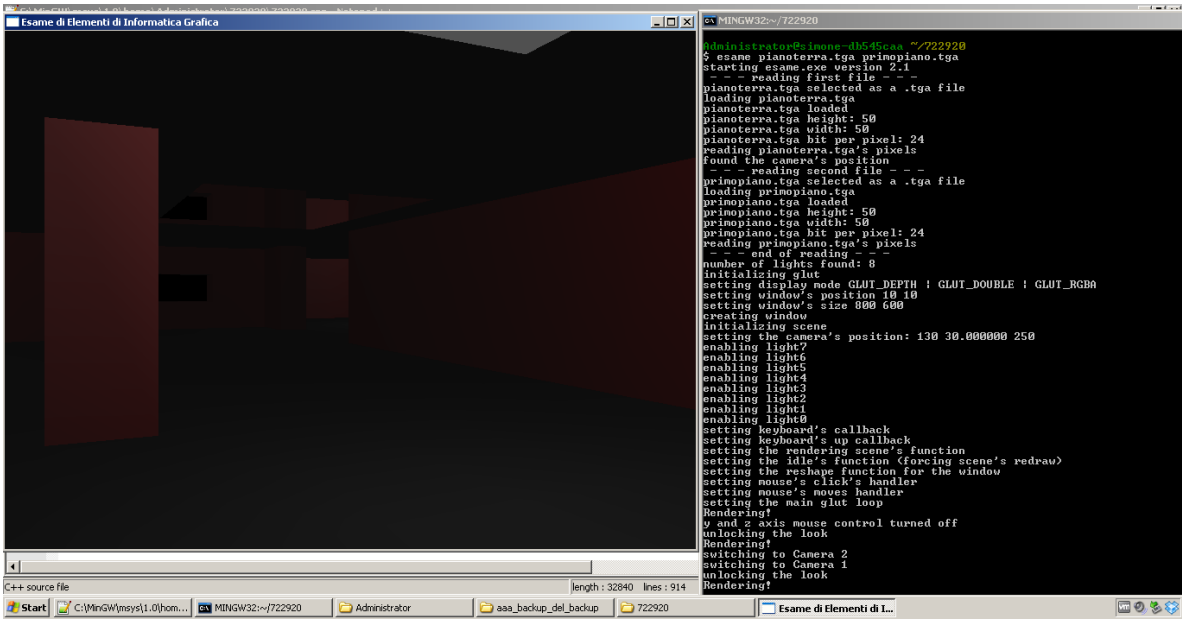
INTRODUZIONE

Anche questa volta il progetto è stato realizzato solamente dal sottoscritto, quindi senza altri componenti di un gruppo di studio; l'exe che verrà prodotto funzionerà solo su sistemi windows a 32bit (principalmente per via della presenza del sorgente "tga_io.cpp"), è stato sviluppato su una macchina virtuale con windows XP a 32bit su un ospite con windows 7 a 64bit. Ho fatto altri test su altre macchine a 32bit con sistemi operativi più aggiornati di XP come Vista e Seven, e ha funzionato, bastava che fosse presente la libreria freeglut.dll se la suit di MinGW fosse installata, altrimenti sarebbero necessarie altre dll come libgcc_s_dw2-1.dll e libstdc++-6.dll (dando per scontato che le librerie openGL siano installate, ma in tutte le versioni di window per default è così).

Un particolare fastidioso è che su certe macchine virtualizzate i movimenti del mouse sono catturati dall'applicazione in modo troppo sensibile per valutare in modo comodo l'ambiente virtuale ricostruito, per questo è stato inserito un comando per disabilitare parzialmente alcuni assi di rotazione gestiti dal mouse.

Per rendere più leggibile il codice ho lasciato parecchi commenti e molte printf() che esternano alcune varibili e stati del programma. Si noti che ho fatto uso di funzionalità deprecate di OpenGL&CO (come ad esempio GL_QUADS), e che non ho gestito le eccezioni, neanche quelle più banali come errori di lettura da file ecc.

Se gli input sono corretti il programma crea una finestra di 800x600 nell'angolo in alto a sinistra (in [10,10]) ridimensionabile, e se lanciato da ad esempio una consolle come quella di minGW fornisce in output informazioni utili per la comprensione di quello che sta avvenendo; per questo motivo suggerisco di compilare ed eseguire il programma da minGW ed affiancare se possibile la finestra alla consolle, come nella figura della pagina successiva:


Esame di Elementi di Informatica Grafica
MINGW32:~/722920
Administrator@simone-db545caa ~/722920
$ esame pianoterra.tga primopiano.tga
starting esame.exe version 2.1
- - - reading first file - - -
pianoterra.tga selected as a .tga file
loading pianoterra.tga
pianoterra.tga loaded
pianoterra.tga height: 50
pianoterra.tga width: 50
pianoterra.tga bit per pixel: 24
reading pianoterra.tga's pixels
found the camera's position
- - - reading second file - - -
primopiano.tga selected as a .tga file
loading primopiano.tga
primopiano.tga loaded
primopiano.tga height: 50
primopiano.tga width: 50
primopiano.tga bit per pixel: 24
reading primopiano.tga's pixels
- - - end of reading - - -
number of lights found: 8
initializing glut
setting display mode GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA
setting window's position 10 10
setting window's size 800 600
creating window
initializing scene
setting the camera's position: 130 30.000000 250
enabling light7
enabling light6
enabling light5
enabling light4
enabling light3
enabling light2
enabling light1
enabling light0
setting keyboard's callback
setting keyboard's up callback
setting the rendering scene's function
setting the idle's function (forcing scene's redraw)
setting the reshape function for the window
setting mouse's click's handler
setting mouse's moves handler
setting the main glut loop
Rendering!
y and z axis mouse control turned off
unlocking the look
Rendering!
switching to Camera 2
switching to Camera 1
unlocking the look
Rendering!
C++ source file
length : 32840 lines : 914
Start
C:\MinGW\msys\1.0\hom...
MINGW32:~/722920
Administrator
aaa_backup_del_backup
722920
Esame di Elementi di I...

UNIVERSITA' DEGLI STUDI DI MILANO BICOCCA

SCOPO DEL PROGETTO

Il pdf dell'esame mi chiedeva di creare un programma che leggesse in input due file .tga che rappresentano le piantine di un'abitazione a due piani e, tramite rendering, creasse un modello tridimensionale di quest'ultima. In aggiunta c'è da gestire l'illuminazione e una telecamera che si muove all'interno di questo mondo. Più precisamente le immagini tga in ingresso sono a 24 bit (anche se credo che con l'implementazione che ho scritto vadino bene anche immagini a 32 bit visto che il puntatore che legge i dati avanza in modo intelligente) e ogni pixel dell'immagine rappresenta, in base al colore, un elemento tridimensionale differente.

Nello specifico:

- un pixel nero (0, 0, 0) rappresenta il vuoto/l'esterno dell'edificio;
- un pixel bianco (255, 255, 255) rappresenta una zona interna. Sarà compito del visualizzatore rappresentarla con una coppia di elementi soffitto/pavimento;
- un pixel rosso (255, 0, 0) rappresenta una parte di “muro”. Anche in questo caso, il visualizzatore dovrà rappresentare l'elemento in maniera opportuna;
- un pixel verde (0, 255, 0) rappresenta il punto di partenza in cui va posizionata la camera. E' consentito solo un pixel di questo tipo. Si suppone che in questa area ci sia una coppia pavimento/soffitto. Se sono presenti più punti verdi faccio terminare il programma. Se non è presente, faccio terminare il programma;
- un pixel giallo (255, 255, 0) rappresenta un coppia pavimento/soffitto in cui è presente una luce sul soffitto. Sono consentiti solo un massimo di 8 pixel gialli. Se sono presenti più di 8 punti gialli faccio terminare il programma;
- un pixel blu (0, 0, 255) rappresenta una parte di “finestra”. I pixel blu si trovano su una striscia di pixel rossi (muro). Ho scelto di rappresentare la finestra come un muro con una fessura che comunque non è in contatto nè co il soffitto nè con il pavimento;
- Vi è poi una zona di pixel magenta (255, 0, 255) (che si trovano nella stessa posizione sulle due piantine, cioè con le stesse coordinate): in questa zona (che possiamo considerare zona “ascensore” – ma l’ascensore non va disegnato!) non c'è il soffitto del piano terra e non c'è il pavimento del primo piano (quindi non vanno disegnati): i due piani comunicano tra loro. Questa zona è sempre rettangolare (non saranno possibili altre forme). Nel progetto base la camera (a cui è permesso un movimento verticale) può attraversare il soffitto del piano terra in qualunque posizione. Per avere una valutazione più alta fare in modo che la camera passi da un piano all'altro nella zona magenta (cioè dove “c'è l'ascensore”), che è poi la scelta per cui ho optato.

Quindi dalle immagini in ingresso andrò a creare una zona di quadro dove ogni tot di unità equivalgono ad un pixel letto dall'immagine. L'immagine campione che mi è stata fornita ha dimensione quadrata, ma l'implementazione supporta anche immagini di lati diversi quindi rettangolari.

Modificando due semplici #define si possono modificare le dimensioni massime dell'immagini in ingresso, anche se il vincolo delle dimesioni dipende dalla dimesioni massime che lo specifico compilatore c++ impone su matrici di char a tre dimensioni.

Dopo aver acquisito l'immagine, creato tutta l'abitazione, abilitato le eventuali luci, creato e posizionato la camera, dovrò gestire il rendering in tempo reale e i vari input da tastiera e mouse.

A livello di interfaccia utente, si richiede che l'applicazione consenta tramite mouse e tastiera di navigare all'interno del piano simulando una persona che cammina nell'abitazione.

In particolare:

- tramite gli spostamenti orizzontali del mouse sarà possibile scegliere la direzione sul piano X-Z (rotazione su se stessi);
- tramite gli spostamenti verticali del mouse sarà possibile alzare/abbassare lo sguardo.
- tramite i tasti “e” e “d”, sarà possibile, rispettivamente, avanzare o indietreggiare lungo la direzione sul piano X-Z;
- tramite i tasti “s” e “f” sarà possibile spostarsi lateralmente a sinistra e destra. Il movimento è ortogonale rispetto alla direzione in cui la camera sta osservando la scena;
- tramite i tasti “a” e “z” sarà possibile lo spostamento verticale della camera, per consentire gli spostamenti da un piano all'alto.

Il programma dovrà ricevere a linea di comando i nomi dei file TGA relativo alla due piantine.

Esempio: "esame pianoterra.tga primopiano.tga".

DESCRIZIONE DELL'IMPLEMENTAZIONE

Il codice sorgente è tutto contenuto nell'unico file 722920.cpp. All'interno vi si trova il main il quale gestisce la corretta sintassi degli argomenti passati, eventuali errori nella lettura dei file e crea l'ambiente openGL/GLUT per i fini del progetto.

Nella cartella 722920 deve anche essere present il file logo_unimib.tga e il Makefile (e in alcuni casi freeglut.dll); inoltre la struttura directory deve essere la seguente:

Oltre ad aver usato funzionalità deprecate di OpenGL, penso che ci sia stato insegnato così per motivi didattici, anche il livello della qualità del c++ non è ingegnerizzato al meglio, ma ho cercato di rendere il tutto molto semplice da leggere.

Per spiegare il funzionamento di questo progetto credo non vi sia metodo migliore di passare in rassegna ogni singola funzione di questo file. Iniziando dal main, ovviamente.

int main(int argc, char argv)**

Come prima cosa viene fatto un controllo sul numero degli argomenti passati, il programma va avanti se e solo se la sintassi risulta essere "esame <file1.tga> <file2.tga> dove <file#.tga> sono stringhe di testo senza spazi che rappresentano il nome di file o path assoluti + nome.

Successivamente vengono lette le due immagini tramite le funzioni *readPiantina()* e gestiti gli eventuali errori rilevate da queste due chiamate. Poi vi è un controllo se effettivamente è stato trovato un pixel verde corrispondente alla posizione iniziale della telecamera, se non viene trovato il programma termina.

Il controllo successiva verifica se l'area di pixel rettangolare magenta è allineata correttamente su ogni piano.

Da qui in poi se tutte le condizioni sono verificate e non ci sono errori si passa alle chiamate OpenGL / GLUT vere e proprie:

glutInit() : l'inizializzazione per glut;

glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA) : imposta modalità in double buffering e alloca il depth buffer;

glutInitWindowPosition(10, 10) : seleziona la posizione della finestra;

glutInitWindowSize(800, 600) : imposta la dimensione della finestra;

glutCreateWindow("Esame di Elementi di Informatica Grafica") : crea la finestra;

initGL() : inizializzo la scena (chiamate openGL);

glutKeyboardFunc(keyboard) : setta la callback per la tastiera;

glutKeyboardUpFunc(keyboardUp) : setta la callback per la tastiera per quando vengono rilasciati i tasti;

glutDisplayFunc(renderScene) : setta la funzione di disegno della scena;

glutIdleFunc(idle) : setta la funzione di idle che forza il ridisegno della scena;

glutReshapeFunc(reshape) : funzione per il reshape (utilizzando quando viene modificata la finestra);

glutMouseFunc(processMouse) : setta l'handler per i click del mouse;

glutPassiveMotionFunc(processMousePassiveMotion) : setta l'handler per i movimenti del mouse;

glutMainLoop() : richiama il main loop bloccante di glut.

int readPiantina(int iPiano, char* sFileName)

Questa funzione legge il contenuto dei file tga e riempie una matrice di caratteri dichiarata

static nel corpo della classe. Ho pensato che il modo più semplice per rappresentare le informazioni sui pixel dei file fosse una matrice cubica di caratteri dove la terza dimensione rappresenta il numero del piano e le prime due gli assi 2D delle piantine.

iPiano è un intero che suggerisce alla funzione se il file in ingresso è il piano terra [0], o il primo piano [1], mentre *sFileName* viene dato in pasto alla *img.load()* presa dalle librerie compilate per questa materia e in una variabile viene caricata l'intera immagine tga.

Ricordo che in caso di errore viene sollevata un'eccezione non gestita ma dalla shell minGW si capisce tutto l'avvenuto.

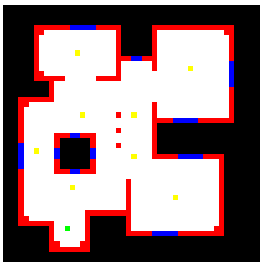
Tramite i metodi *.height()* e *.width()* ricavo la dimensione in pixel dell'immagine, e con il metodo *.BPP()* i byte per pixel, informazione molto utile che mi serve per incrementare correttamente il puntatore che uso per trovare le informazioni sui colori dei pixel.

Queste informazioni vengono mappate in memoria tramite la matrice *cMatrix*, quindi accessibile da tutte le funzioni del programma (ok, non è il metodo migliore!).

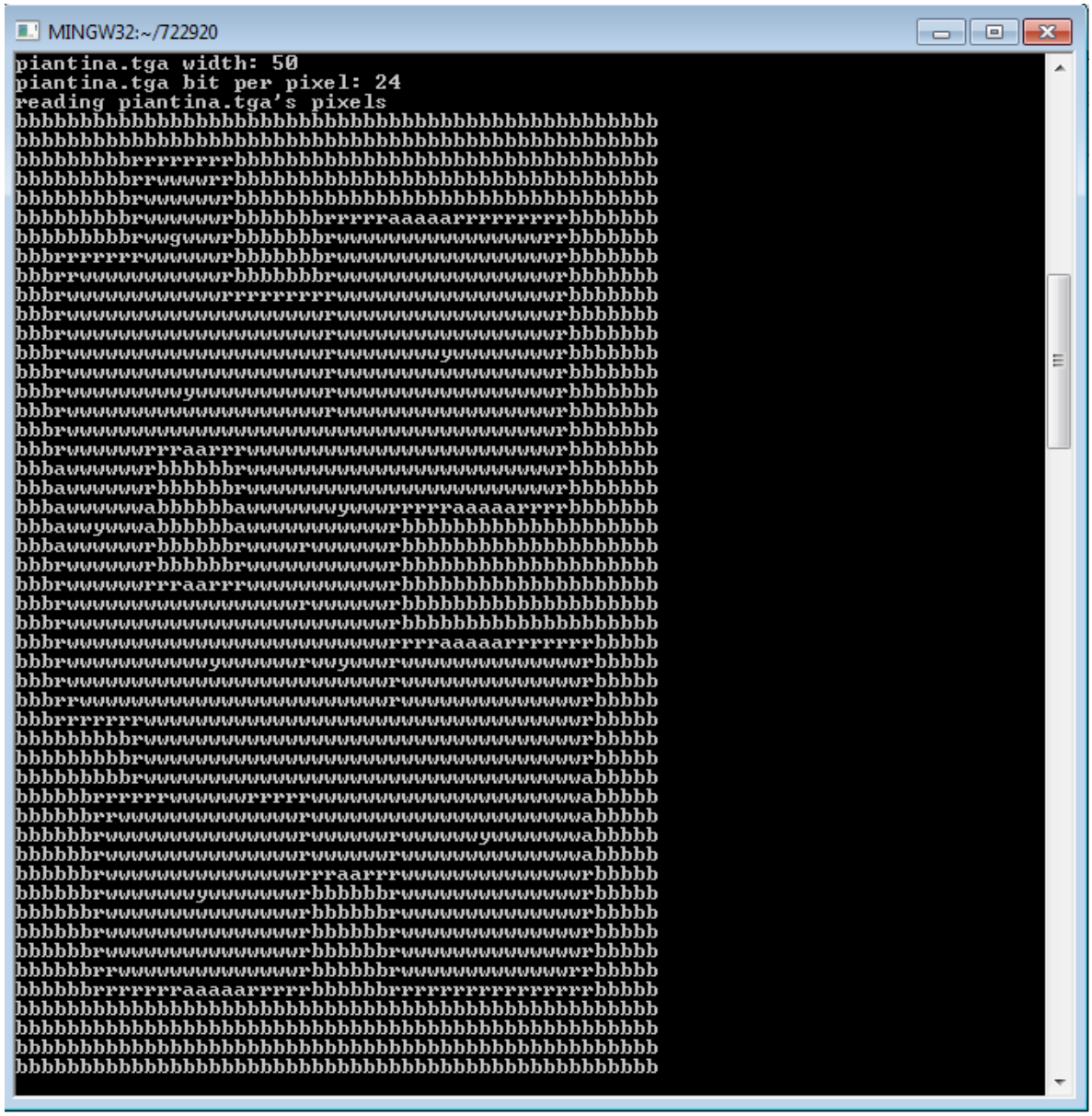
Tramite due for annidati riempio questa matrice secondo le seguenti regole:
l'iesimoxjesimo valore della matrice sarà "b" se il colore del pixel corrispondente è nero, "w" se bianco, "a" se blu, "r" se rosso, "y" se giallo, "g" se verde, "m" se magenta.

Sempre da questi for vengono fatti i controlli sul numero delle luci e sulla presenza della posizione della telecamera.

Per esempio questo file tga



genera la seguente (porzione di) matrice:



Inoltre viene riempito l'array *locLuci[][]* (sempre static) con le posizioni di ogni sorgente luminosa.

void imposta_materiale_parete()
void imposta_materiale_soffitto()
void imposta_materiale_pavimento()
void imposta_materiale_lampada()

Queste funzioni vengono chiamate dalla *renderScene()* mentre si disegnano i poligoni che compongono la scena, tramite la *glMaterialfv()* settano in che modo i materiali si comportano in presenza della luce; ad esempio le pareti risultano rosse.

void initGL(void)

Questa lunga funzione si occupa dell'inizializzazione degli stati opengl, flags e di molte variabili di sistema. Vengono resettate le variabili che gestiscono la rotazione del mouse, i booleani associati ai tasti di movimento e il flag del freelook.

Viene posizionata la camera nella scena, a seconda di dove si trova il carattere "g" all'interno della matrice. Come si può notare dalla formula ogni pixel dell'immagine tga l'ho fatto corrispondere a 20 unità, e si parte a contare dallo spigolo (0, 0, 0); in pratica il primo elemento della matrice corrisponde al primo cubo di lato 20 con spigolo in basso in (0, 0, 0) e si procede verso coordinate positive. Un piano è alto 60 unità.

Con la direttiva *glEnable(GL_LIGHTING)* viene abilitato il meccanismo di opengl dell'illuminazione, una sola chiamata per tutto il programma.

A seconda del numero di pixel gialli trovati vengono abilitate le luci e settate le loro caratteristiche; ho scelto luci con attenuazione lineare di 0,001 di tipo spot con normale verticale che illuminano il materiale della lampada da fuori verso il basso.

Viene riempito anche il vettore *cameraPos[][]* per memorizzare varie posizioni della camera.

Inoltre viene caricata l'immagine tga del logo dell'università e settata come texture da utilizzare durante il rendering.

void renderScene()

Questa è la funzione più complicata, lunga e dove ho dovuto dedicarci più tempo. E' qui che tre for annidati (due per le dimensioni x y, uno per decidere in quale piano modellare)

vanno a creare i poligoni che definiscono la scena, seguendo sempre la matrice di caratteri riempita in precedenza. Ogni poligono è un rettangolo o quadrato con le relative proprietà del materiale e per ogni vertice ho settato la normale utile per l'illuminazione. Per rendermi conto come far disegnare la scena dai for, ho dovuto ragionarci parecchio con carta e penna disegnando i poligoni su assi cartesiani in tre dimensioni.

Dopo i for vengono collocate anche le luci nelle corrette posizioni.

Alla fine tutte le chiamate incodate vengono forzate ad essere eseguite con la direttiva *glFlush()*; ma prima dei for che disegnano la scena c'è la gestione dei movimenti della camera da parte del mouse e della tastiera; interessante notare come, dato che glut non mette a disposizioni funzioni per leggere il movemento relativo del mouse, sia necessario forzare il puntatore al centro della finestra prima di settare la *gluPerspective()*.

Viene anche disegnata la texture del logo dell'università degli studi Bicocca, in una posizione determinata dall'utente.

void idle (void)

Contiene solo l'istruzione *glutPostRedisplay()* che serve per forzare il redrawing.

void keyboard (unsigned char key, int x, int y)

E' la funzione handler per la tastiera, per tasti "normali", ad esempio i tasti F1-F12 non vengono intercettati. Ad esempio, come chiesto dal testo del progetto, vengono rilevati i tasti "e", "s", "f", "d" per i movimenti, tramite variabili booleane, il tasto "q" per terminare il processo ed il tasto "u" che fa da switcer per la funzione che libera il mouse dalla finestra.

void keyboardUp (unsigned char key, int x, int y)

Funzione handler della tastiera per quando i tasti vengono rilasciati, settata solamente per i tasti di movimento; servono per far cessare il "moto" nella render scene quando non si preme più su un tasto direzionale.

void processMousePassiveMotion (int x, int y)

Funzione di callback per il movimento passivo del mouse; il movimento relativo lo ricavo come scostamento dal centro della finestra. Ad ogni chiamata di rendering, il mouse viene forzato al centro della finestra. Questo è un artificio "brutto ma necessario" poichè glut non

fornisce funzioni che forniscono scostamenti relativi del mouse.

void processMouse (int button, int state, int x, int y)

Callback per la gestione dei click del mouse.

void reshape (int w, int h)

Handler per gli eventi di resizing della finestra: è necessaria perchè modifica i valori della viewport per la corretta visualizzazione della scena a qualunque dimensione si setti la finestra.

int converter (float x)

Funzione di supporto per convertire coordinate mondo in interi indici della matrice di caratteri, creata per rendere più leggibili i meccanismi di collision control con i poligoni della scena.

int LD (float x)

(Level Dectector), anche questa è una funzione molto semplice che mi è servita sempre per i meccanismi di collision control, si suppone che il numero a virgola mobile in ingresso sia la coordinata z della scenza, quindi l'altezza della camera, viene restituito o zero o uno a seconda di che piano si trova dando per scontato che i piani siano alti 60.0 unità.

bool collisionControl (float x, float y, float z)

Questo metodo è quello che si occupa delle collisioni con le pareti della scena con la telecamera, quindi non riguarda soffitti e pareti; è richiamato nella renderscene se vi è stato rilevato movimento tramite i tasti direzionali del piano x y. Restituisce true se ci si può muovere in una data posizione date le coordinate della camera, false altrimenti e come risultato da la sensazione di collisione con i poligoni della scena. Ovviamente fa pesante uso della matrice di caratteri, quindi non vi è nessun calcolo geometrico.

Il ragionamento che ho seguito, anche per non rendere troppo pesante l'algoritmo, è quello di rilevare se il punto in cui ci si troverà andando in quella direzione conoscendo la velocità, sarà un punto non appartenente alle pareti o finestre chiuse della scena.

Eventuali errori sono dovuti al casting da float a integer per consultare la matrice (anche se

ho cercato di migliorare la formula), ma ritengo che siano accettabili ed il calcolo computazionale si riduce a semplici confronti effettuati solamente durante il moto.

bool collisionControlUpDown (float x, float y, float z, bool d)

Questo invece è il metodo che si occupa della gestione delle collisioni tra la camera ed il soffitto, conoscendo la posizione in float della camera e la direzione d del movimento sull'asse z (true se verso alto, false altrimenti).

Inoltre gestisce la zona magenta, consentendo all'utente di cambiare piano solamente in questa zona, andando ad emulare il sistema ascensoristico.

Nella pratica sono un po di if annidati ma di facile comprensione, vengono prima gestiti i movimenti se si è nella zona magenta, altrimenti si considera la posizione del piano e la direzione; ad esempio se si è in una zona non magenta del secondo piano l'altezza della camera dovrà essere nel range di 60.0-120.0 unità.

DESCRIZIONE FEATURES ADDIZIONALI

Come features addizionali ho aggiunto la possibilità di gestire fino a 4 camere, la collisione con i poligoni della scena, la gestione di una superficie alla quale è stata applicata una texture e qualche hotkey che cambia lo stato interno del programma.

La gestione di 4 telecamere è "finta", in realtà vengono salvate le coordinate di 4 punti in una matrice e tramite i tasti 1-4 ci si può spostare nelle posizioni salvate in precedenza (posizione di default è il punto verde nelle piantine); notare che viene salvato solamente il vettore della posizione e non quello del "look at", quindi non avviene nessuna trasformazione di vista in modo esplicito.

La gestione delle collisioni è stata spiegata in precedenza nell"implementazione dei vari metodi, qui riporto che si può abilitare e disablitare in tempo reale tramite il tasto "o".

Con il tasto "u" si può "liberare" il mouse dalla finestra (il focus ritorna al primo click).

Invece con il tasto "p" (print) viene stampato un poligodo (quadrato) sul quale è applicato la texture del logo dell'università nella posizione attuale della camera; anche qui questo poligono esiste fin dall'inizio del programma, ma all'inizio si trova in una posizione nascosta indipendentemente dalle piantine lette in ingresso, ad ogni chiamata viene semplicemente spostato.

Con il tasto "q" si terminia il programma.

Il tasto "x" funge da trigger per la gestione degli assi y e z del mouse, utili per navigare in modo più lineare all'interno dell'ambiente.

CONCLUSIONI

Come al solito è stato un progetto piacevole da sviluppare, ricordo la sintassi del programma e riporto una tabella riassuntiva dei comandi.

esame <file1.tga> <file2.tga>

dove file1.tga è interpretato come pian terreno e file2.tga come primo piano.

e	spostamento in avanti lungo la direzione sul piano x-z
d	spostamento in indietro lungo la direzione sul piano x-z
s	spostamento laterale a sinistra
f	spostamento laterale a destra
a	spostamento verticale verso l'alto della camera
z	spostamento verticale verso il basso della camera
mouse movements	si modifica la direzione sul piano x-z e si può alzare/abbassare lo sguardo
u	toglie il focus alla window
o	abilita/disabilita il collision control
p	"stampa" una texture campione nella posizione attuale
x	abilita/disabilita il controllo del mouse sugli assi di rotazione y e z
1	abilità la telecamera 1
2	abilità la telecamera 2
3	abilità la telecamera 3
4	abilità la telecamera 4
q	termina il programma