#include <glm/glm.hpp>

#include <glm/gtc/matrix\_transform.hpp>

#include <glm/gtc/type\_ptr.hpp>

// FreeGLUT:

#include <GL/freeglut.h>

#include <iostream>

// Rotation angle:

float angleY = 0.0f;

int windowId;

unsigned char faceColor[6][3];

void **displayCube**(float edge){

float size = edge / 2.0f;

// Back:

glBegin(GL\_TRIANGLE\_STRIP);

glColor3ub(faceColor[0][0], faceColor[0][1], faceColor[0][2]);

glVertex3f(size, 0, -size);

glVertex3f(-size, 0, -size);

glVertex3f(size, edge, -size);

glVertex3f(-size, edge, -size);

glEnd();

// Front:

glBegin(GL\_TRIANGLE\_STRIP);

glColor3ub(faceColor[1][0], faceColor[1][1], faceColor[1][2]);

glVertex3f(-size, 0, size);

glVertex3f(size, 0, size);

glVertex3f(-size, edge, size);

glVertex3f(size, edge, size);

glEnd();

// Left:

glBegin(GL\_TRIANGLE\_STRIP);

glColor3ub(faceColor[2][0], faceColor[2][1], faceColor[2][2]);

glVertex3f(-size, edge, -size);

glVertex3f(-size, 0, -size);

glVertex3f(-size, edge, size);

glVertex3f(-size, 0, size);

glEnd();

// Right:

glBegin(GL\_TRIANGLE\_STRIP);

glColor3ub(faceColor[3][0], faceColor[3][1], faceColor[3][2]);

glVertex3f(size, 0, -size);

glVertex3f(size, edge, -size);

glVertex3f(size, 0, size);

glVertex3f(size, edge, size);

glEnd();

// Bottom:

glBegin(GL\_TRIANGLE\_STRIP);

glColor3ub(faceColor[4][0], faceColor[4][1], faceColor[4][2]);

glVertex3f(-size, 0, -size);

glVertex3f(size, 0, -size);

glVertex3f(-size, 0, size);

glVertex3f(size, 0, size);

glEnd();

// Top:

glBegin(GL\_TRIANGLE\_STRIP);

glColor3ub(faceColor[4][0], faceColor[4][1], faceColor[4][2]);

glVertex3f(size, edge, -size);

glVertex3f(-size, edge, -size);

glVertex3f(size, edge, size);

glVertex3f(-size, edge, size);

glEnd();

}

void **displayCallback**(){

// Clear the screen:

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

// Set a matrix to move our triangle:

float cubeSize = 10.0f;

glm::mat4 translation = glm::translate(glm::mat4(), glm::vec3(0.0f, -15.0, -45.0f));

glm::mat4 offset = glm::translate(glm::mat4(), glm::vec3(0.0f, cubeSize, 0.0f));

glm::mat4 rotation = glm::rotate(glm::mat4(), glm::radians(angleY), glm::vec3(0.0f, 1.0f, 0.0f));

glm::mat4 scaling = glm::scale(glm::mat4(), glm::vec3(0.75, 0.75, 0.75));

// Put first cube:

glm::mat4 f = translation \* rotation;

glLoadMatrixf(glm::value\_ptr(f));

displayCube(cubeSize);

// Put more cubes:

for (int c = 0; c < 4; c++)

{

f = f \* offset \* scaling \* rotation;

glLoadMatrixf(glm::value\_ptr(f));

displayCube(cubeSize);

}

// Swap this context's buffer:

glutSwapBuffers();

// Update coords:

angleY += 0.5f;

// Force rendering refresh:

glutPostWindowRedisplay(windowId);

}

void **reshapeCallback**(int width, int height){

std::cout << "[reshape func invoked]" << std::endl;

glViewport(0, 0, width, height);

glMatrixMode(GL\_PROJECTION);

glm::mat4 projection = glm::perspective(glm::radians(45.0f), (float) width / (float) height, 1.0f, 100.0f);

glLoadMatrixf(glm::value\_ptr(projection));

glMatrixMode(GL\_MODELVIEW);

}

void **keyboardCallback**(unsigned char key, int mouseX, int mouseY){

std::cout << "[std key pressed]" << std::endl;

switch (key)

{

case 'r':

for (int c = 0; c < 6; c++)

for (int d = 0; d < 3; d++)

faceColor[c][d] = rand() % 255;

break;

}

// Force rendering refresh:

glutPostWindowRedisplay(windowId);

}

int **main**(int argc, char \*argv[]){

// Init face colors:

for (int c = 0; c < 6; c++)

for (int d = 0; d < 3; d++)

faceColor[c][d] = rand() % 255;

// Init context:

glutInitDisplayMode(GLUT\_RGB | GLUT\_DOUBLE | GLUT\_DEPTH);

glutInitWindowPosition(100, 100);

// FreeGLUT can parse command-line params, in case:

glutInit(&argc, argv);

// Set some optional flags:

glutSetOption(GLUT\_ACTION\_ON\_WINDOW\_CLOSE, GLUT\_ACTION\_GLUTMAINLOOP\_RETURNS);

// Create the window with a specific title:

windowId = glutCreateWindow("The tower");

// Set callback functions:

glutDisplayFunc(displayCallback);

glutReshapeFunc(reshapeCallback);

glutKeyboardFunc(keyboardCallback);

// Global OpenGL settings:

glClearColor(1.0f, 0.6f, 0.1f, 1.0f);

glEnable(GL\_DEPTH\_TEST);

// Enter the main FreeGLUT processing loop:

glutMainLoop();

return 0;

}

//Esempio matrici

#include <glm/glm.hpp>

#include <glm/gtc/matrix\_transform.hpp> //

#include <iostream>

#define VEC2STR(v) "[" << v.x << "\t" << v.y << "\t" << v.z << "\t" << v.w << "] --> (" << v.x/v.w << ", " << v.y/v.w << ", " << v.z/v.w << ")"

#define MAT2STR(m) "\t" << m[0][0] << "\t" << m[0][1] << "\t" << m[0][2] << "\t" << m[0][3] << std::endl \

<< "\t" << m[1][0] << "\t" << m[1][1] << "\t" << m[1][2] << "\t" << m[1][3] << std::endl \

<< "\t" << m[2][0] << "\t" << m[2][1] << "\t" << m[2][2] << "\t" << m[2][3] << std::endl \

<< "\t" << m[3][0] << "\t" << m[3][1] << "\t" << m[3][2] << "\t" << m[3][3]

int main(int argc, char \*argv[]){

glm::mat4 t1 = glm::scale(glm::mat4(1.0f), glm::vec3(0.5f));

glm::mat4 t2 = glm::rotate(glm::mat4(1.0f), glm::radians(90.0f), glm::vec3(0.0f, 0.0f, 1.0f));

glm::mat4 t3 = glm::translate(glm::mat4(1.0f), glm::vec3(10.0f, 0.0f, 0.0f));

glm::mat4 perspective = glm::perspective(glm::radians(45.0f), 1.0f, 1.0f, 100.0f);

glm::vec4 vertexIn[3];

vertexIn[0] = glm::vec4(-15.0f, 0.0f, -50.0f, 1.0f);

vertexIn[1] = glm::vec4(15.0, 0.0f, -50.0f, 1.0f);

vertexIn[2] = glm::vec4(0.0, 15.0f, -50.0f, 1.0f);

// Case 1: just perspective

std::cout << "1) Perspective only" << std::endl;

glm::mat4 f = perspective;

std::cout << " f = " << MAT2STR(f) << std::endl;

// Compute clip-space coords:

std::cout << std::endl;

glm::vec4 vertexOut[3];

for (int c = 0; c < 3; c++)

{

vertexOut[c] = f \* vertexIn[c];

std::cout << " " << c << ") " << VEC2STR(vertexOut[c]) << std::endl;

}

// Case 2: scaling

std::cout << std::endl << "2) Scaling" << std::endl;

f = perspective \* t1;

std::cout << " f = " << MAT2STR(f) << std::endl;

// Compute clip-space coords:

std::cout << std::endl;

for (int c = 0; c < 3; c++)

{

vertexOut[c] = f \* vertexIn[c];

std::cout << " " << c << ") " << VEC2STR(vertexOut[c]) << std::endl;

}

// Case 3: rotation

std::cout << std::endl << "3) Rotation" << std::endl;

f = perspective \* t2;

std::cout << " f = " << MAT2STR(f) << std::endl;

// Compute clip-space coords:

std::cout << std::endl;

for (int c = 0; c < 3; c++)

{

vertexOut[c] = f \* vertexIn[c];

std::cout << " " << c << ") " << VEC2STR(vertexOut[c]) << std::endl;

}

//////////////////////

// Case 4: translation

std::cout << std::endl << "4) Translation" << std::endl;

f = perspective \* t3;

std::cout << " f = " << MAT2STR(f) << std::endl;

// Compute clip-space coords:

std::cout << std::endl;

for (int c = 0; c < 3; c++)

{

vertexOut[c] = f \* vertexIn[c];

std::cout << " " << c << ") " << VEC2STR(vertexOut[c]) << std::endl;

}

/////////////////

// Case 5: all in

std::cout << std::endl << "5) All-in" << std::endl;

f = perspective \* t3 \* t2 \* t1;

std::cout << " f = " << MAT2STR(f) << std::endl;

// Compute clip-space coords:

std::cout << std::endl;

for (int c = 0; c < 3; c++)

{

vertexOut[c] = f \* vertexIn[c];

std::cout << " " << c << ") " << VEC2STR(vertexOut[c]) << std::endl;

}

std::cout << std::endl;

return 0;

}

**Screen Tearing** di solito si verifica quando si dispone di una potente scheda grafica o si sta eseguendo un gioco che non è ad uso intensivo della grafica. Succede perché la tua scheda grafica produce più fotogrammi rispetto alla frequenza di aggiornamento del tuo monitor da gioco. Se il frame buffer swap si verifica tra un aggiornamento del monitor, sul display verranno inviate due o più immagini. Poiché le informazioni per più fotogrammi vengono simultaneamente inviate al display del tuo monitor, può verificarsi un disallineamento dello schermo lacerante come nell'esempio precedente.

**V-Sync**, limita il numero di fotogrammi della tua scheda grafica in uscita per corrispondere alla frequenza di aggiornamento del tuo monitor da gioco. Il suo scopo è quello di fermare il disallineamento dello schermo e fa quello che dice il nome, sincronizzandosi verticalmente. Il metodo riduce la tua scheda grafica per produrre solo un frame ogni volta che il monitor si aggiorna. Se la frequenza fotogrammi cala anche solo di un fotogramma al di sotto della frequenza di aggiornamento del monitor, la frequenza di aggiornamento verrà compensata e temporaneamente ridotta della metà per mantenere la sincronizzazione con la frequenza fotogrammi. Se la scheda grafica non ha generato un nuovo frame, viene utilizzato quello precedente, causando una stuttering.

**G-Sync e FreeSync** invece il monitor modifica automaticamente la frequenza di aggiornamento su cui opera per adattarsi al frame rate della scheda grafica.

**Ray Tracing**, si divide in **light tracing** e **eye tracing**, il primo simula la natura, una fonte di luce manda i raggi che vanno ovunque, i pochi raggi che arrivano alla camera (l’occhio) vengono renderizzati (computazionalmente pesante). È molto meglio l’eye tracing siccome i raggi partono dalla camera per arrivare alla fonte di luce (meno pensante).

**Ray Casting** il raggio parte dalla camera, colpisce l’oggetto e riporta il colore del pixel.

**Recursive Ray Tracing** uguale al Ray Casting ma per ogni collisione genera tre raggi (riflessione, rifrazione e ombra). Il raggio di shadow è quel raggio che va dal punto di intersezione verso la fonte di luce, se incontra un oggetto vuol dire che quel punto è in ombra. Allora viene colorato del colore del punto di interesezione del primo raggio ma scurito (perché in ombra).(guarda immagine slide 8)

**Path Tracing** generati dei raggi a caso, solo quelli che andranno verso l’occhio vengono presi

**Rasterization** (illuminazione diretta) parte dalle world transformation poi fa il passaggio dalle varie coordinate per arrivare a reindirizzare una scena in 3D in 2D, è molto più facile perché utilizza matrici di proiezione e sono di due tipi (ortogonale e prospettiva). la rasterizzazione prende i vertici dei triangoli e vengono copiati in 2D tramite proiezione (ortonale o prospettiva), oltre alla posizione vengono anche salvati il colore e la profondità (nei vertici) successivamente vengono collegati i vari vertici che formeranno l'immagine. Nella rasterizzazione

**Radiosity** permette di usare l’illuminazione globale in partica ad ogni passaggio dell’algoritmo viene calcolata la luce per ogni pixel delle scena (quindi una parte viene assorbita e una riflessa). Durante la prima passata saranno illuminati solo gli oggetti che vedono la sorgente di luce, dopo la seconda invece la scena riceverà la luce causata dal rimbalzo della prima passata. Più passi si fanno maggiore sarà la qualità.

**Camera inversa** (formula per avere le eye coordinate) se si vuol osservare il mondo da un punto di vista più basso, ma non c'è la camera, devi spostare il mondo in su.

**Vertex Shader vs Fragment Shader** la differenza principale è che un vertex shader può manipolare gli attributi dei vertici. quali sono i punti d'angolo dei tuoi poligoni. D'altra parte Fragment Shader si occupa di come appaiono i pixel tra i vertici. Sono interpolati tra i vertici definiti seguendo regole specifiche. se vuoi che il tuo poligono sia completamente rosso, definirai tutti i vertici in rosso. (Vertex Shader) Se vuoi effetti specifici come un gradiente tra i vertici, devi farlo nello Fragment Shader .