



Grafica al calcolatore

Laboratorio - 2

Andrea Giachetti andrea.giachetti@univr.it

Fabio Marco Caputo

Department of Computer Science, University of Verona, Italy



Reminder:



- Ricordare il path per le librerie GLFW
 - `export LD_LIBRARY_PATH=lib/lin`
- Per usare l'emulazione mesa
 - `export LIBGL_ALWAYS_SOFTWARE=1`
- Per far funzionare i codici su alcune macchine con scheda video intel su ubuntu
 - `export MESA_GLSL_VERSION_OVERRIDE=130`
 - Potrebbe essere provato in alternativa al comando precedente



Es 1,2 da ricordare

- Come passiamo gli attributi dei vertici

es1

Nome variabile

```
GLint posAttrib = glGetAttribLocation(shaderProgram, "position");  
glEnableVertexAttribArray(posAttrib);  
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(GLfloat), 0);
```


DIMENSIONE

Salto tra successivi

Offset iniziale

es2

```
GLint posAttrib = glGetAttribLocation(shaderProgram, "position");  
glEnableVertexAttribArray(posAttrib);  
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat), 0);  
  
GLint colAttrib = glGetAttribLocation(shaderProgram, "color");  
glEnableVertexAttribArray(colAttrib);  
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(GLfloat), (void*)(2 *  
sizeof(GLfloat)));
```



```
const GLchar* vertexSource =
#ifdef __APPLE_CC__
    "#version 150 core\n"
#else
    "#version 130\n"
#endif
```

es1

```
"in vec2 position;"
"void main() {"
"    gl_Position = vec4(position, 0.0, 1.0);"
"}";
```

```
/*
    applichiamo un colore uniforme
*/
```

```
const GLchar* fragmentSource =
#ifdef __APPLE_CC__
    "#version 150 core\n"
#else
    "#version 130\n"
#endif
"out vec4 outColor;"
"void main() {"
"    outColor = vec4(1.0);"
"}";
```

```
const GLchar* vertexSource =
#ifdef __APPLE_CC__
    "#version 150 core\n"
#else
    "#version 130\n"
#endif
```

es2

```
"in vec2 position;"
"in vec3 color;"
"out vec3 Color;"
"void main() {"
"    Color = color;"
"    gl_Position = vec4(position,0.0,1.0);"
"}";
```

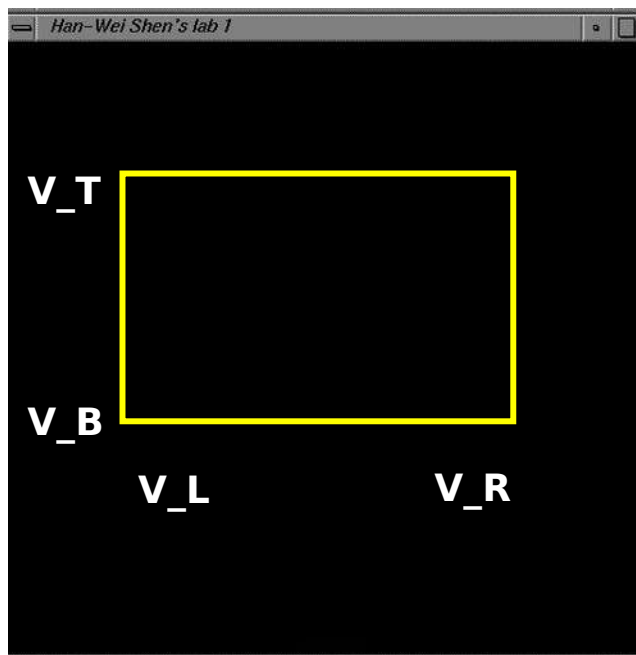
```
/*settiamo il colore interpolato */
const GLchar* fragmentSource =
#ifdef __APPLE_CC__
    "#version 150 core\n"
#else
```

```
    "#version 130\n"
#endif
"in vec3 Color;"
"out vec4 outColor;"
"void main() {"
"    outColor = vec4(Color, 1.0);"
"}";
```



Viewport

- The rectangular region in the screen that maps to our world window
- Defined in the window's (or control's) coordinate system



```
glViewport(int left, int bottom,  
           int (right-left),  
           int (top-bottom));
```

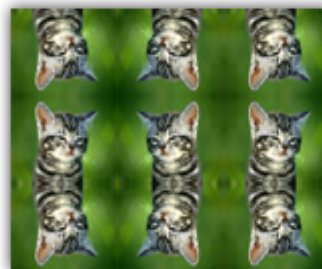


Textures

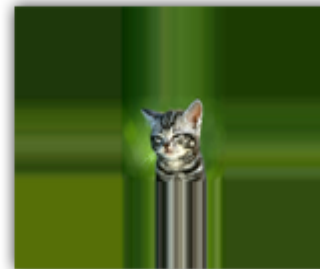
- Abbiamo visto in teoria il meccanismo del texture mapping
- Ora lo vediamo in pratica. Dobbiamo
 - Avere un modo di mettere in memoria l'immagine texture
 - Associare un attributo ai vertici con le coordinate (ricordate che erano teoricamente nell'intervallo [0,1] [0,1] che corrispondevano agli estremi dell'immagine, anche se poi avevamo visto che potevamo mettere coordinate fuori estendendo la parametrizzazione con le modalità
 - Tile: repeat (OpenGL); ripetizione periodica
 - Mirror: ripetizione periodica ma specchiata a ogni ripetizione
 - Clamp to edge – valori esterni sono prolungati dal bordo vicino
 - Clamp to border – tutti i valori esterni sono attribuiti ad un valore a parte



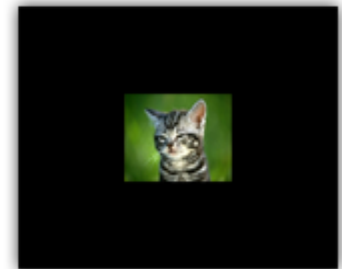
GL_REPEAT



GL_MIRRORED_REPEAT



GL_CLAMP_TO_EDGE



GL_CLAMP_TO_BORDER



Esercizio e03

- Sempre geometria2D come prima, ma mappiamo texture
- Utilizziamo la libreria <http://lodev.org/lodepng/>
- Togliamo il colore e mettiamo le coordinate texture al suo posto: 2 float Le definiamo sui vertici

```
const GLfloat vertices[] = {  
    // Position      Texcoords  
    -0.5f,  0.5f, 0.0f, 0.0f, // Top-left  
    0.5f,  0.5f, 1.0f, 0.0f, // Top-right  
    0.5f, -0.5f, 1.0f, 1.0f, // Bottom-right  
    -0.5f, -0.5f, 0.0f, 1.0f // Bottom-left  
};
```

- Definiamo l'oggetto texture: GLuint texture;

- Nel main si chiama la funzione
 - initialize_texture();

```
glGenTextures(1, &texture);  
std::vector<unsigned char> image;  
unsigned width, height;
```

```
unsigned error = lodepng::decode(image, width, height, "image.png");  
if(error) std::cout << "decode error " << error << ": " << lodepng_error_text(error) <<  
std::endl;
```

```
// il bind della texture avverra' sulla texture unit numero 0 (l'indice che dovra' essere  
passato agli shader)  
glActiveTexture(GL_TEXTURE0);  
glBindTexture(GL_TEXTURE_2D, texture);  
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,  
GL_UNSIGNED_BYTE, image.data());  
// shaderProgram must be already initialized  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);  
}
```

- La funzione collega la texture a un indice e definisce parametri
 - comportamento per i parametri st fuori [0 1]
 - Filtri per magnification e minification
 - (si potrebbe fare mipmapping)



```
glUniform1i(glGetUniformLocation(shaderProgram,  
"textureSampler"), 0);
```

- Nel draw() avviene il collegamento della texture da usare (ce ne potrebbero essere diverse) alla variabile “sampler” di tipo uniform con cui si accede dal GLSL

```
"uniform sampler2D textureSampler;"
```

```
"void main() {"
```

```
" outColor = texture2D(textureSampler, Coord);"
```

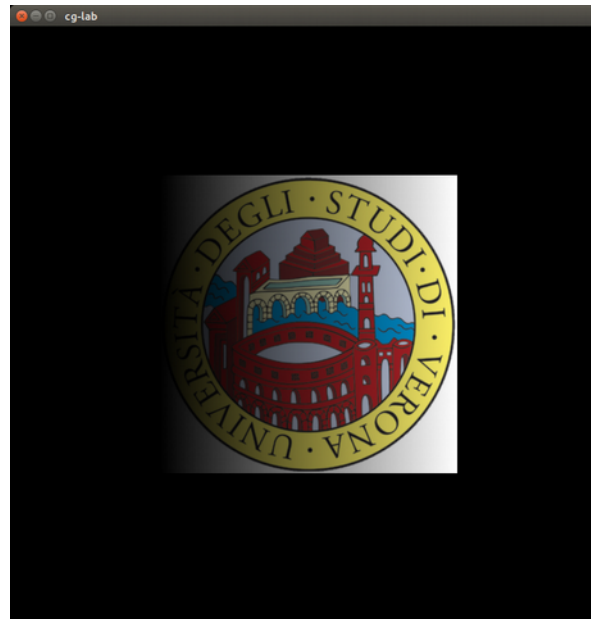
- Alla fine la texture deve essere distrutta

```
glDeleteTextures(1, &texture);
```



Esercizio

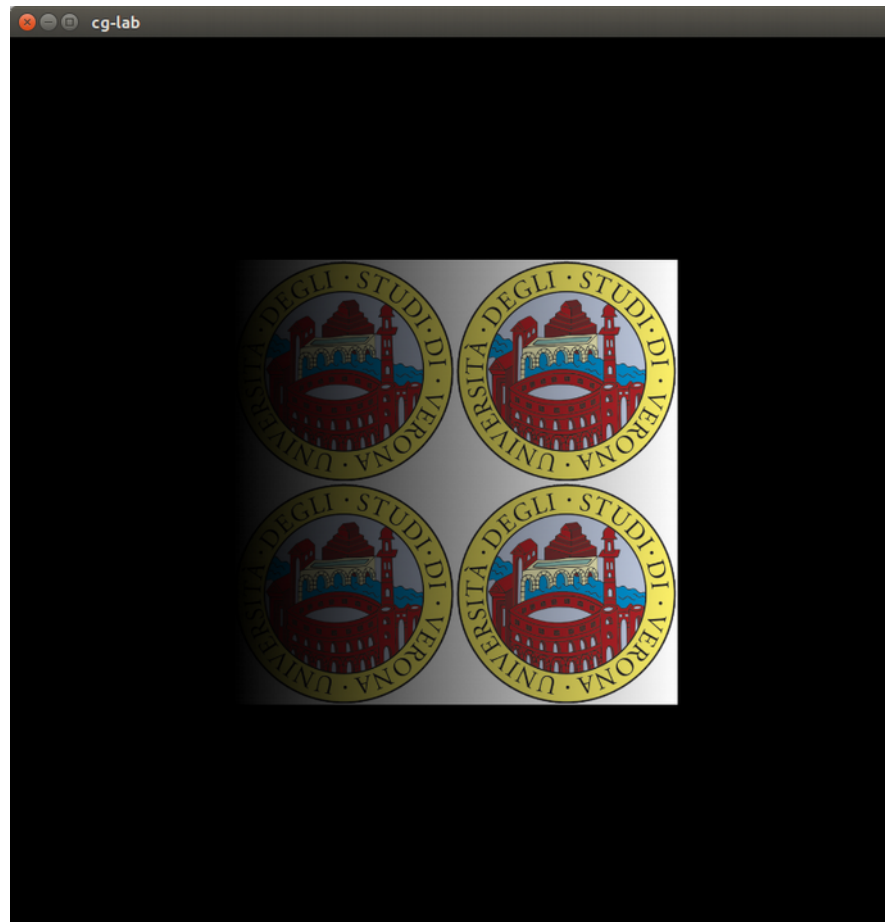
- Caricare ora il file es03b.cpp
- E' come prima, ma si passano allo shader sia coordinate texture e texture sia colore vertici (tutti bianchi)
- L'output è un blending moltiplicativo
`"outColor = vec4(Color, 1.0)*texture(textureSampler, Coord);"`
- Modificare il programma per ottenere l'effetto in figura





Esercizio

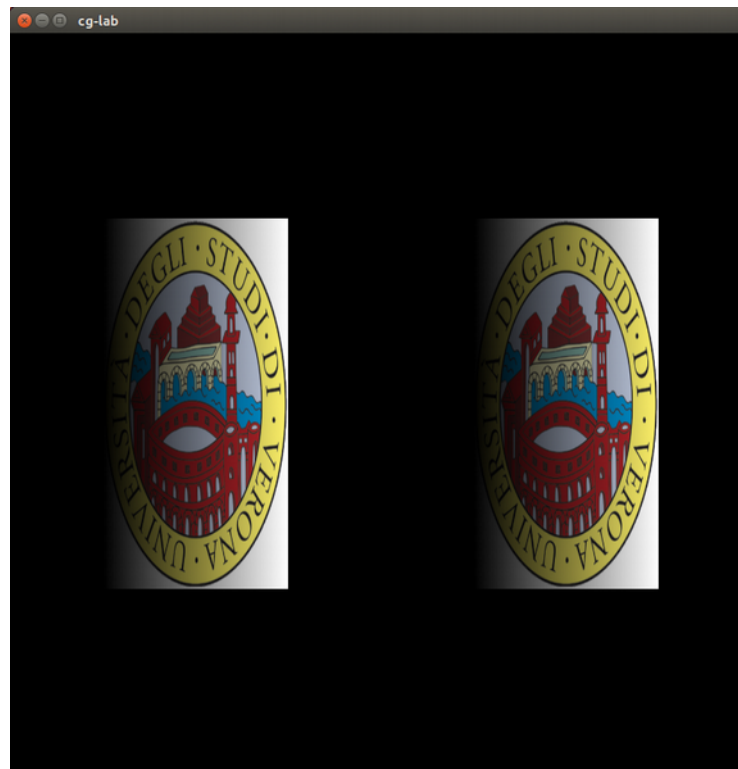
- Modificare ulteriormente il programma per ottenere l'immagine qui sotto





Esercizio

- Usare due viewport per ottenere la figura qui sotto





Passiamo (finalmente) al 3D

- Caricare l'esercizio e04
- Il modello adesso è 3D: vertici e edge di un cubo, ma è fatto allo stesso modo
- Ci appiccichiamo una texture che mappa i lati a regioni di un'immagine creata ad hoc
- Introduciamo le matrici: model matrix, view matrix e proiezione prospettica
 - Sono gestite con la libreria glm. Passate al vertex shader come uniform
 - L'operazione è svolta poi nel vertex shader
- Usiamo l'algebra delle matrici in coordinate omogenee che abbiamo visto in teoria

Il modello del cubo

```
const GLfloat vertices[] = {  
  // Position          Color          Texcoords  
  -0.5f,  0.5f, -0.5f, 1.0f, 1.0f, 1.0f, .25f, 0.0f, // 0  
    0.5f,  0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.5f, 0.0f, // 1  
  
  -0.5f,  0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 1.f/3.f, // 2  
  -0.5f,  0.5f,  0.5f, 1.0f, 1.0f, 1.0f, .25f, 1.f/3.f, // 3  
    0.5f,  0.5f,  0.5f, 1.0f, 1.0f, 1.0f, 0.5f, 1.f/3.f, // 4  
    0.5f,  0.5f, -0.5f, 1.0f, 1.0f, 1.0f, .75f, 1.f/3.f, // 5  
  -0.5f,  0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 1.0f, 1.f/3.f, // 6  
  
  -0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.0f, 2.f/3.f, // 7  
  -0.5f, -0.5f,  0.5f, 1.0f, 1.0f, 1.0f, .25f, 2.f/3.f, // 8  
    0.5f, -0.5f,  0.5f, 1.0f, 1.0f, 1.0f, 0.5f, 2.f/3.f, // 9  
    0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, .75f, 2.f/3.f, // 10  
  -0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 1.0f, 2.f/3.f, // 11  
  
  -0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, .25f, 1.0f, // 12  
    0.5f, -0.5f, -0.5f, 1.0f, 1.0f, 1.0f, 0.5f, 1.0f, // 13  
};
```




Il modello del cubo



```
/* 0--1
   |  |
2--3--4--5--6
|  |  |  |  |
7--8--9-10-11
   |  |
   12--13    */
const GLuint elements[] = {
0, 3, 4, 0, 4, 1,
2, 7, 8, 2, 8, 3,
3, 8, 9, 3, 9, 4,
4, 9, 10, 4, 10, 5,
5, 10, 11, 5, 11, 6,
8, 12, 13, 8, 13, 9
};
```



Il modello del cubo



```
// shaderProgram must be already initialized
GLint posAttrib = glGetAttribLocation(shaderProgram,
"position");
glEnableVertexAttribArray(posAttrib);
glVertexAttribPointer(posAttrib, 3, GL_FLOAT, GL_FALSE, 8
* sizeof(GLfloat), 0);

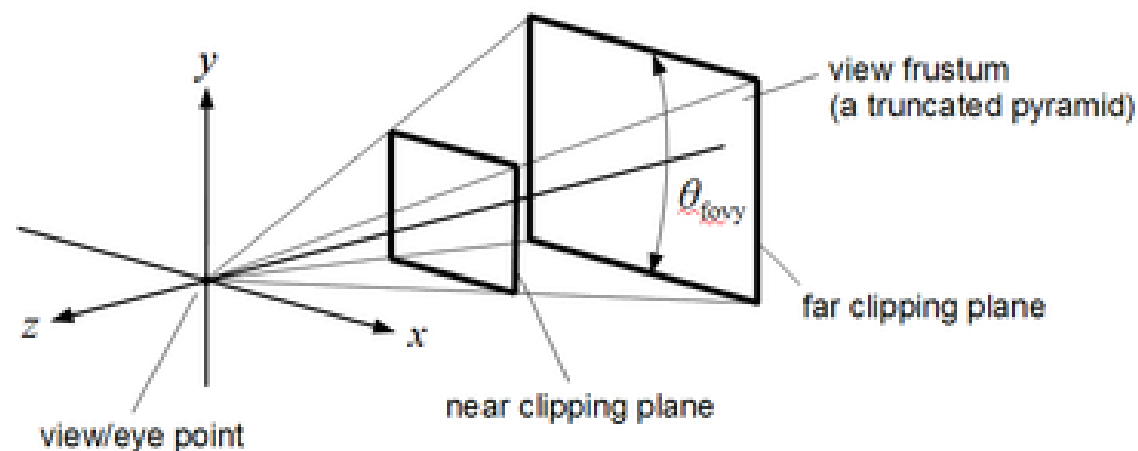
GLint colAttrib = glGetAttribLocation(shaderProgram,
"color");
glEnableVertexAttribArray(colAttrib);
glVertexAttribPointer(colAttrib, 3, GL_FLOAT, GL_FALSE, 8
* sizeof(GLfloat), (void*)(3 * sizeof(GLfloat)));

GLint cooAttrib = glGetAttribLocation(shaderProgram,
"coord");
glEnableVertexAttribArray(cooAttrib);
glVertexAttribPointer(cooAttrib, 2, GL_FLOAT, GL_FALSE, 8
* sizeof(GLfloat), (void*)(6 * sizeof(GLfloat)));
```



Matrici

- Model Matrix: la applichiamo ai modelli per muoverli nella scena.
- View Matrix: muoviamo il sistema di riferimento standard associato con la telecamera
- Projection matrix: applichiamo la proiezione per ottenere le coordinate standardizzate sul piano immagine
- Poi c'è la viewport che abbiamo già visto





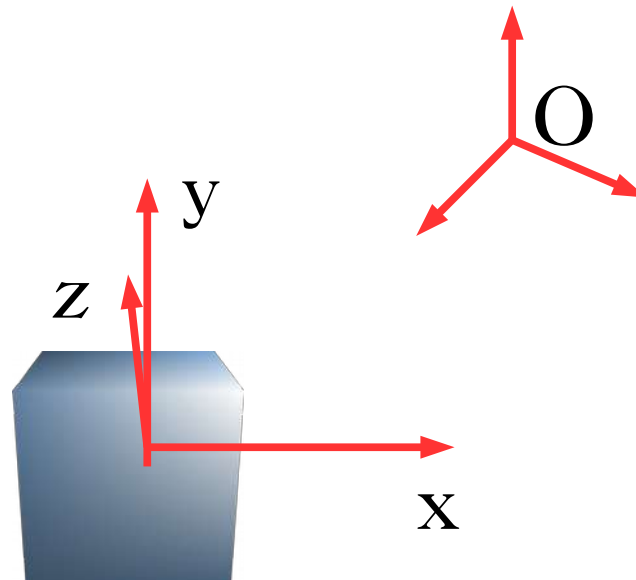
Codice

```
glm::mat4 projection = glm::perspective(PI/4, // vertical FOV  
1.f/1.f, // aspect ratio  
1.0f, // near clipping plane  
10.0f // far clipping plane  
);
```



Codice

```
glm::mat4 view = glm::lookAt(  
    glm::vec3(2.5f, 2.5f, 2.5f), // punto da cui guardo  
    glm::vec3(0.0f, 0.0f, 0.0f), // punto a cui guardo  
    glm::vec3(0.0f, 1.0f, 0.0f) //up vector  
);
```





Codice

```
glUniformMatrix4fv(glGetUniformLocation(shaderProgram,  
"projection"), 1, GL_FALSE, &projection[0][0]);  
glUniformMatrix4fv(glGetUniformLocation(shaderProgram,  
"view"), 1, GL_FALSE, &view[0][0]);  
glUniformMatrix4fv(glGetUniformLocation(shaderProgram,  
"model"), 1, GL_FALSE, &model[0][0]);
```




Funzioni glm



- Si può applicare
 - `mat4 translate(mat4, vec3) // = T*m`
 - `mat4 scale(mat4, vec3)`
 - `mat4 rotate(mat4, float, vec3)`
- Se `mat4` è fornito come primo argomento il risultato è moltiplicato e ritornato dalla funzione
 - `m=glm::translate(m,glm::vec3(1,1,1));`
 - `m=glm::rotate(m,a,glm::vec3(0,0,1));`
 - `m=glm::translate(m,glm::vec3(1,1,1));`
- È più efficiente di
 - `const glm::mat4 i(1.0); //identity matrix`
 - `m=m*glm::translate(i,glm::vec3(1,1,1));`
 - `m=m*glm::rotate(i,a,glm::vec3(0,0,1));`
 - `m=m*glm::translate(i,glm::vec3(1,1,1));`



glm

- Tipicamente comporremo le trasformazioni con le funzioni viste sopra
- Documentazione su <https://glm.g-truc.net/0.9.8/index.html>



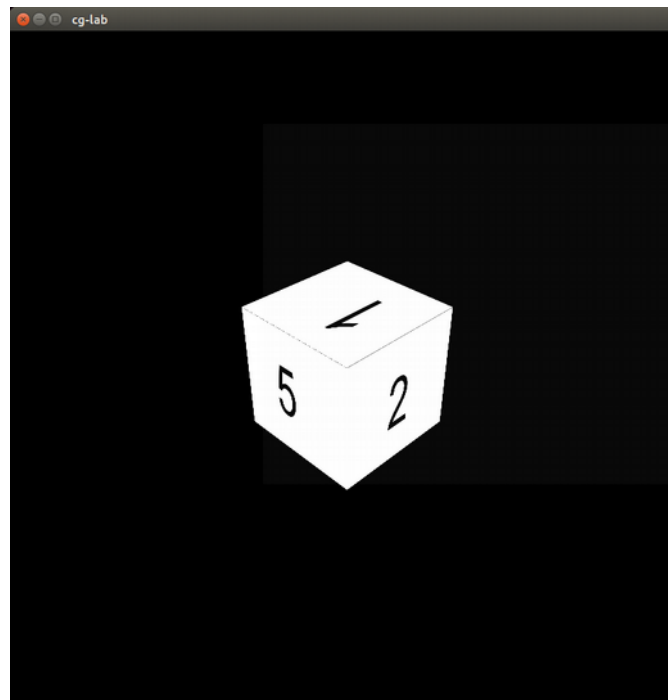
Ricordiamo

- Possiamo per ogni oggetto nella scena memorizzare una model matrix
- Se abbiamo più istanze di oggetti uguali, o con semplicemente attributi da cambiare, usiamo lo stesso buffer e facciamo i vari disegni dopo le rispettive trasformazioni
- La composizione delle trasformazioni segue le regole che abbiamo studiato a teoria
 - Es: verificare che la composizione dipende dall'ordine! Applicare al quadrato fermo una model matrix composta e invertire l'ordine. Che succede?



Esercizio

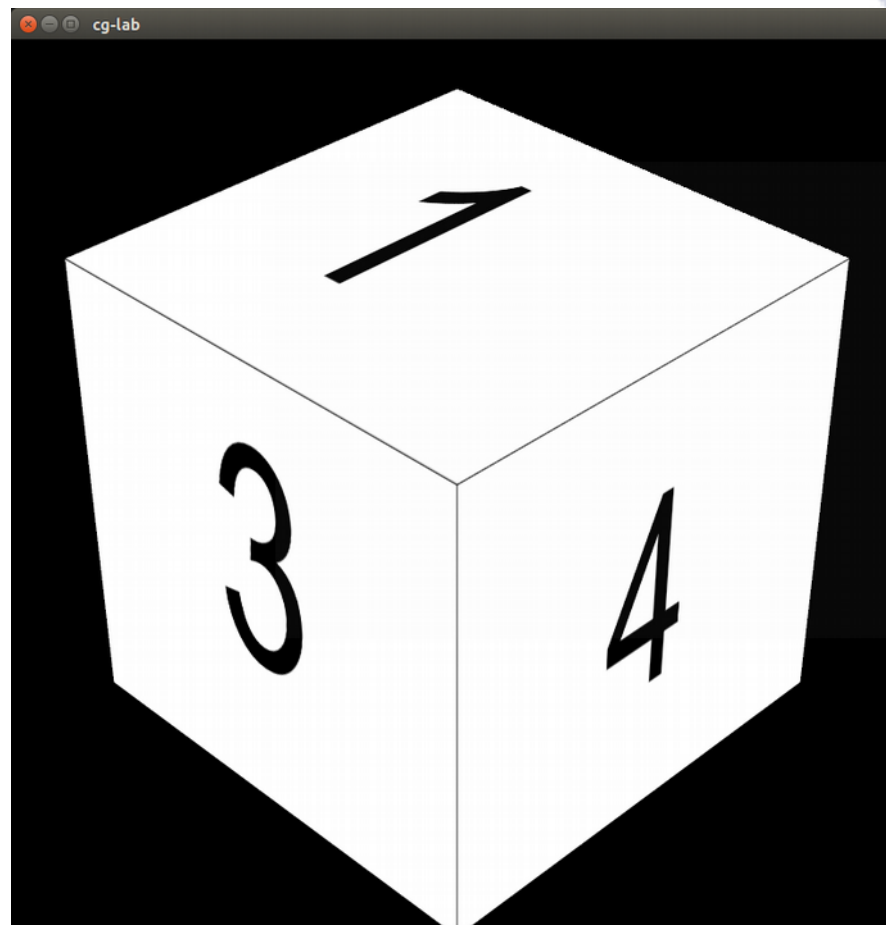
- A partire dal codice originale, modificare la model matrix per ruotare il cubo in modo che mostri le facce 1,2,5 come in figura, quando si preme il tasto "r"
- Nota abbiamo forzato l'uso dell'angolo in radianti





Esercizio

- Sempre dall'originale e04 cambiare i parametri della proiezione per avere un risultato simile a quello qui sotto

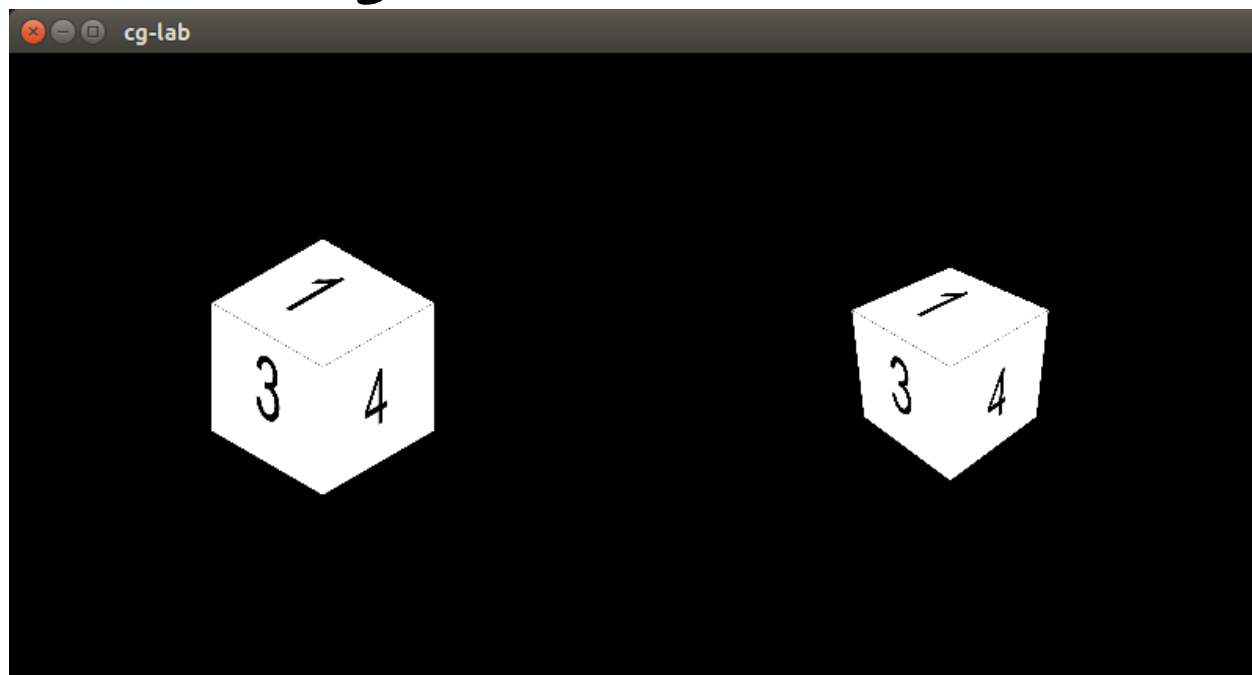


Oranica 2010



Esercizi

- Provare a sostituire la proiezione prospettica con quella ortografica
- `glm::ortho (T const &left, T const &right, T const &bottom, T const &top, T const &zNear, T const &zFar)`
- Provare a mettere in una finestra 800x400 due viewport sui quali mappare un rendering con proiezione ortografica e prospettica come nella figura sotto





Riferimenti

- <http://www.opengl.org>
- <http://www.khronos.org/opengl/>
- <http://www.glfw.org/>
- <http://antongerdelan.net/opengl>
- <http://www.opengl-tutorial.org/>
- <http://www.arcsynthesis.org/gltut/>
- <http://glm.g-truc.net/0.9.5/index.html>

