Realidad Virtual. Práctica 7.

CUBEMAPS

Índice

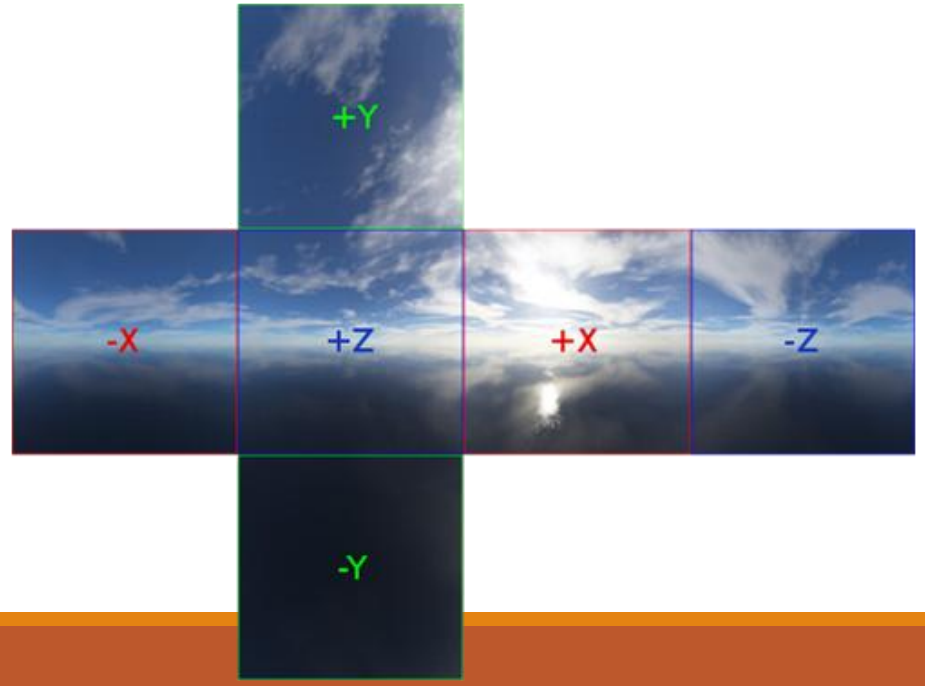
1. Objetivos

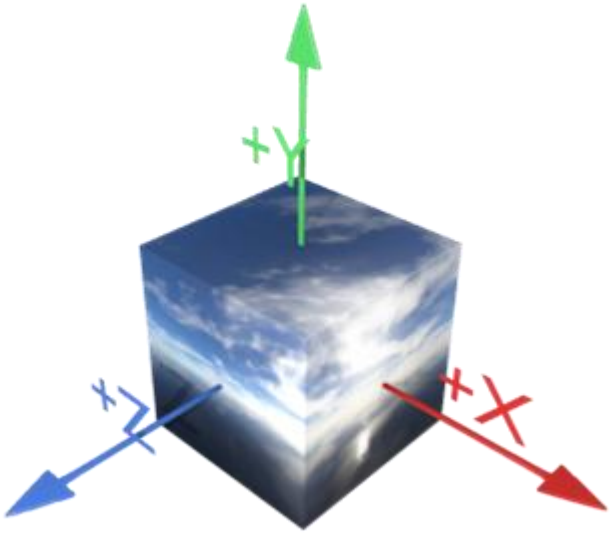
- El objetivo de esta práctica es describir la forma de definir una textura de fondo que muestre el entorno en 360º.
- Para ello se utiliza un tipo de textura especial denominado CubeMap.
- El objeto utilizado para desarrollar este entorno se conoce como “SkyBox”.

- Código de la práctica:
  https://www.uhu.es/francisco.moreno/gii_rv/practicas/practica07/practica07.rar

2. La textura CubeMap

- La textura CubeMap es un tipo de textura diseñada para ocupar todo el espacio.
- Está formada por 6 imágenes en 2D que corresponden a las proyecciones sobre un cubo.
- Las imágenes deben estar diseñadas para ofrecer una imagen continua entre las caras del cubo siempre que se observen desde el centro del cubo.





2. La textura CubeMap

- La aplicación de las texturas de tipo CubeMap utiliza tres coordenadas espaciales que corresponden a la posición del punto sobre el cubo en el sistema de coordenadas del centro del cubo.
- A partir de las tres coordenadas (rx, ry, rz) se calcula a cuál de las seis imágenes pertenece el punto y las coordenadas de textura 2D correspondientes en dicha imagen.

Major axis	Target	sc	tc
+rx	GL_TEXTURE_CUBE_MAP_POSITIVE_X	-rz	-ry
-rx	GL_TEXTURE_CUBE_MAP_NEGATIVE_X	+rz	-ry
+ry	GL_TEXTURE_CUBE_MAP_POSITIVE_Y	+rx	+rz
-ry	GL_TEXTURE_CUBE_MAP_NEGATIVE_Y	+rx	-rz
+rz	GL_TEXTURE_CUBE_MAP_POSITIVE_Z	+rx	-ry
-rz	GL_TEXTURE_CUBE_MAP_NEGATIVE_Z	-rx	-ry

$$s = (sc/|ma| + 1) / 2$$

$$t = (tc/|ma| + 1) / 2$$

2. La textura CubeMap

- La ecuación anterior provoca un efecto curioso.
  - Supongamos que queremos calcular el texel asociado a las coordenadas (0.0, 0.5, -1)
  - Este punto se encuentra en la cara –Z en una posición centrada en el eje X y por encima del centro en el eje Y.
  - Al aplicar la ecuación se obtienen las coordendas de textura (0.5, 0.25), es decir, un punto de la imagen centrado en el eje X pero por debajo del centro en el eje Y.
  - Dicho de otra manera, la ecuación anterior provoca que las imágenes de los ejes +X, -X, +Z y –Z se muestren invertidas
  - Por esa razón debemos invertir verticalmente las imágenes utilizadas en estos ejes.

3. La clase CGSkybox

- Para dibujar el Skybox necesitamos definir una figura que describa un telón de fondo.
- Para ello solo se necesitan cuatro vértices para generar un rectángulo.
- La figura contendrá también la referencia a la textura CubeMap y métodos para cargar las imágenes.
- El comportamiento de esta figura es diferente de los objetos que incluimos en los modelos, así que necesitamos definir una nueva clase que no es subclase de CGFigure.

3. La clase CGSkybox

- Cabecera Skybox.h

```
#pragma once

#include <GL/glew.h>
#include <glm/glm.hpp>
#include "CGShaderProgram.h"

class CGSkybox {
public:
  CGSkybox();
  ~CGSkybox();
  void Draw(CGShaderProgram * program, glm::mat4 projection, glm::mat4 view);

private:
  GLuint cubemap;
  GLuint VBO[2];
  GLuint VAO;
  void InitCube();
  void InitCubemap();
  void InitTexture(GLuint target, const char* filename);
  void InitTexture(GLuint target, int idr);
};
```

3. La clase CGSkybox

- El constructor de la clase realiza dos acciones: cargar la textura CubeMap y crear el VAO que describe el telón de fondo.
  - El método InitCubemap() se encarga de crear la textura y cargar las seis imágenes que la forman.
  - Para cargar las imágenes se utiliza el método InitTexture() con un contenido similar al que utiliza la clase CGMaterial.
  - Se han incluido dos versiones de este método para poder cargar las imágenes desde ficheros externos o desde recursos de la aplicación.
  - El método InitCube() crea el Vertex Array Object y los buffers que describen el rectángulo utilizado como telón de fondo.
  - Las posiciones de los vértices están descritas directamente en coordenadas Clip.
  - La forma en que se transforman los atributos del Skybox y se pintan los píxeles es diferente a la utilizada en las figuras normales, por lo que debemos desarrollar una versión diferente del VertexShader y del FragmentShader.

3. La clase CGSkybox

- El método Draw() lanza el proceso de dibujo del telón de fondo y requiere asignar las variables específicas del programa gráfico, incluyendo la asignación de la textura CubeMap como textura activa 0.
  - Como veremos a continuación, el VertexShader en este caso utiliza una matriz llamada Inverse que corresponde a la transformación entre el sistema de coordenadas Clip y el sistema de coordenadas del modelo.
  - Para obtener esta transformación se calcula primero la transformación entre el modelo y el observador (solo la orientación) y entre el observador y el volumen Clip (matriz projection).
  - La transformación que necesitamos (de Clip a modelo) es la inversa de ésta (de modelo a Clip).
  - El método glDepthMask() se utiliza para activar y desactivar la escritura del buffer de profundidad.

3. La clase CGSkybox

- Contenido CGSkybox.cpp

```
#include "CGSkybox.h"
#include <GL/glew.h>
#include <FreeImage.h>
#include "CGFigure.h"

CGSkybox::CGSkybox()
{
  InitCubemap();
  InitCube();
}

CGSkybox::~CGSkybox()
{
  glDeleteBuffers(2, VBO);
  glDeleteVertexArrays(1, &VAO);
  glDeleteTextures(1, &cubemap);
}
```

3. La clase CGSkybox

```
void CGSkybox::InitCube()
{
  GLfloat vertices[12] = {
    -1.0f, -1.0f, -1.0f,
    1.0f, -1.0f, -1.0f,
    1.0f, 1.0f, -1.0f,
    -1.0f, 1.0f, -1.0f
  };

  GLushort indexes[6] = {
    0,1,2,
    0,2,3
  };

  // Create the Vertex Array Object
  glGenVertexArrays(1, &VAO);
  glBindVertexArray(VAO);

  // Create the Vertex Buffer Objects
  glGenBuffers(2, VBO);

  // Copy data to video memory
  // Vertex data
  glBindBuffer(GL_ARRAY_BUFFER, VBO[VERTEX_DATA]);
  glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*12, vertices, GL_STATIC_DRAW);

  // Indexes
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, VBO[INDEX_DATA]);
  glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLushort)*6, indexes, GL_STATIC_DRAW);

  glEnableVertexAttribArray(0); // Vertex position
  glBindBuffer(GL_ARRAY_BUFFER, VBO[VERTEX_DATA]);
  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
}
```

3. La clase CGSkybox

```
void CGSkybox::InitCubemap()
{
 glActiveTexture(GL_TEXTURE1);

 glGenTextures(1, &cubemap);

glBindTexture(GL_TEXTURE_CUBE_MAP, cubemap);

 // Versión con recursos

InitTexture(GL_TEXTURE_CUBE_MAP_POSITIVE_X, IDR_IMAGE4);

InitTexture(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, IDR_IMAGE5);

InitTexture(GL_TEXTURE_CUBE_MAP_POSITIVE_Y, IDR_IMAGE6);

InitTexture(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, IDR_IMAGE7);

InitTexture(GL_TEXTURE_CUBE_MAP_POSITIVE_Z, IDR_IMAGE8);

InitTexture(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, IDR_IMAGE9);

 // Versión con ficheros

 //InitTexture(GL_TEXTURE_CUBE_MAP_POSITIVE_X, "textures/posx.jpg");

 //InitTexture(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, "textures/negx.jpg");

 //InitTexture(GL_TEXTURE_CUBE_MAP_POSITIVE_Y, "textures/posy.jpg");

 //InitTexture(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, "textures/negy.jpg");

 //InitTexture(GL_TEXTURE_CUBE_MAP_POSITIVE_Z, "textures/posz.jpg");

 //InitTexture(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, "textures/negz.jpg");

 // Typical cube map settings

glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER,
                GL_LINEAR);

glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER,
                GL_LINEAR);

glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,
                GL_CLAMP_TO_EDGE);

glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,
                GL_CLAMP_TO_EDGE);

glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R,
                GL_CLAMP_TO_EDGE);

}
```

3. La clase CGSkybox

```
void CGSkybox::InitTexture(GLuint target, const char* filename)
{
  FREE_IMAGE_FORMAT format = FreeImage_GetFileType(filename, 0);
  FIBITMAP* bitmap = FreeImage_Load(format, filename);
  FIBITMAP *pImage = FreeImage_ConvertTo32Bits(bitmap);
  int nWidth = FreeImage_GetWidth(pImage);
  int nHeight = FreeImage_GetHeight(pImage);
  glTexImage2D(target, 0, GL_RGBA8, nWidth, nHeight,
    0, GL_BGRA, GL_UNSIGNED_BYTE, (void*)FreeImage_GetBits(pImage));
  FreeImage_Unload(pImage);
}
```

3. La clase CGSkybox

```
void CGSkybox::InitTexture(GLuint target, int idr)

{

  HRSRC handle = FindResource(NULL, MAKEINTRESOURCE(idr), L"IMAGE");

  HGLOBAL hGlobal = LoadResource(NULL, handle);

  LPCTSTR rsc_ptr = static_cast<LPCTSTR>(LockResource(hGlobal));

  DWORD mem_size = SizeofResource(NULL, handle);

  BYTE* mem_buffer = (BYTE*)malloc((mem_size) * sizeof(BYTE));

  memcpy(mem_buffer, rsc_ptr, mem_size * sizeof(BYTE));

  FreeResource(hGlobal);

  FIMEMORY* hmem = FreeImage_OpenMemory(mem_buffer, mem_size * sizeof(BYTE));

  FREE_IMAGE_FORMAT fif = FreeImage_GetFileTypeFromMemory(hmem, 0);

  FIBITMAP* check = FreeImage_LoadFromMemory(fif, hmem, 0);

  FIBITMAP* pImage = FreeImage_ConvertTo32Bits(check);

  int nWidth = FreeImage_GetWidth(pImage);

  int nHeight = FreeImage_GetHeight(pImage);

  glTexImage2D(target, 0, GL_RGBA8, nWidth, nHeight,

    0, GL_BGRA, GL_UNSIGNED_BYTE, (void*)FreeImage_GetBits(pImage));

FreeImage_Unload(pImage);

FreeImage_CloseMemory(hmem);

  free(mem_buffer);

}
```

3. La clase CGSkybox

```
void Skybox::Draw(CGShaderProgram * program, glm::mat4 projection, glm::mat4 view)
{
  glm::mat3 rot3 = glm::mat3(view); // Parte rotacional de la matriz View
  glm::mat4 rot4 = glm::mat4(rot3);
  glm::mat4 mvp = projection * rot4; // Transformación del Skybox a coordenadas Clip
  glm::mat4 inv = glm::inverse(mvp); // Transformación de coordenadas Clip a coordenadas de modelo del Skybox
  program->SetUniformMatrix4("Inverse", inv);
  program->SetUniformI("CubemapTex", 0);
  glDepthMask(GL_FALSE);
  glActiveTexture(GL_TEXTURE0);
  glBindTexture(GL_TEXTURE_CUBE_MAP, cubemap);
  glBindVertexArray(VAO);
  glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, NULL);
  glDepthMask(GL_TRUE);
}
```

4. El programa gráfico

- El dibujo del Skybox requiere un tratamiento diferente del que utilizamos para el resto de objetos que forman la escena, por lo que utilizaremos un programa gráfico específico para dibujarlo.
- Con respecto al código del VertexShader, el Skybox va a recibir como atributo VertexPosition la posición de los vértices en coordenadas Clip, así que la variable de salida gl_Position se copia directamente del atributo.
- Las coordendas de textura para el CubeMap requieren tres componentes (rx, ry, rz) y se van a almacenar en la variable de salida Position.
- Estas coordenadas se calculan como las posiciones del telón de fondo expresadas en coordenadas del modelo.
- Esto requiere una transformación entre el sistema de coordenadas Clip y el sistema de coordenadas del modelo que debe estar recogida en la matriz Inverse.

4. El programa gráfico

```
#version 400

layout(location = 0) in vec3 VertexPosition;

uniform mat4 Inverse;

out vec3 Position;

void main()
{
  gl_Position = vec4(VertexPosition,1.0);
  Position = vec3(Inverse * gl_Position);
  Position = normalize(Position);
}
```

4. El programa gráfico

- Por su parte, el código del FragmentShader se limita a buscar el color de la textura CubeMap asociado a las coordenadas del telón de fondo.
- La textura CubeMap se introduce como una variable uniforme (CubemapTex) de tipo samplerCube.
- Las coordendas del telón de fondo corresponden a la variable de entrada Position generada en el VertexShader.

4. El programa gráfico

```
#version 400

in vec3 Position;

out vec4 FragColor;

uniform samplerCube CubemapTex;

void main()
{
  FragColor = texture(CubemapTex,Position);
}
```

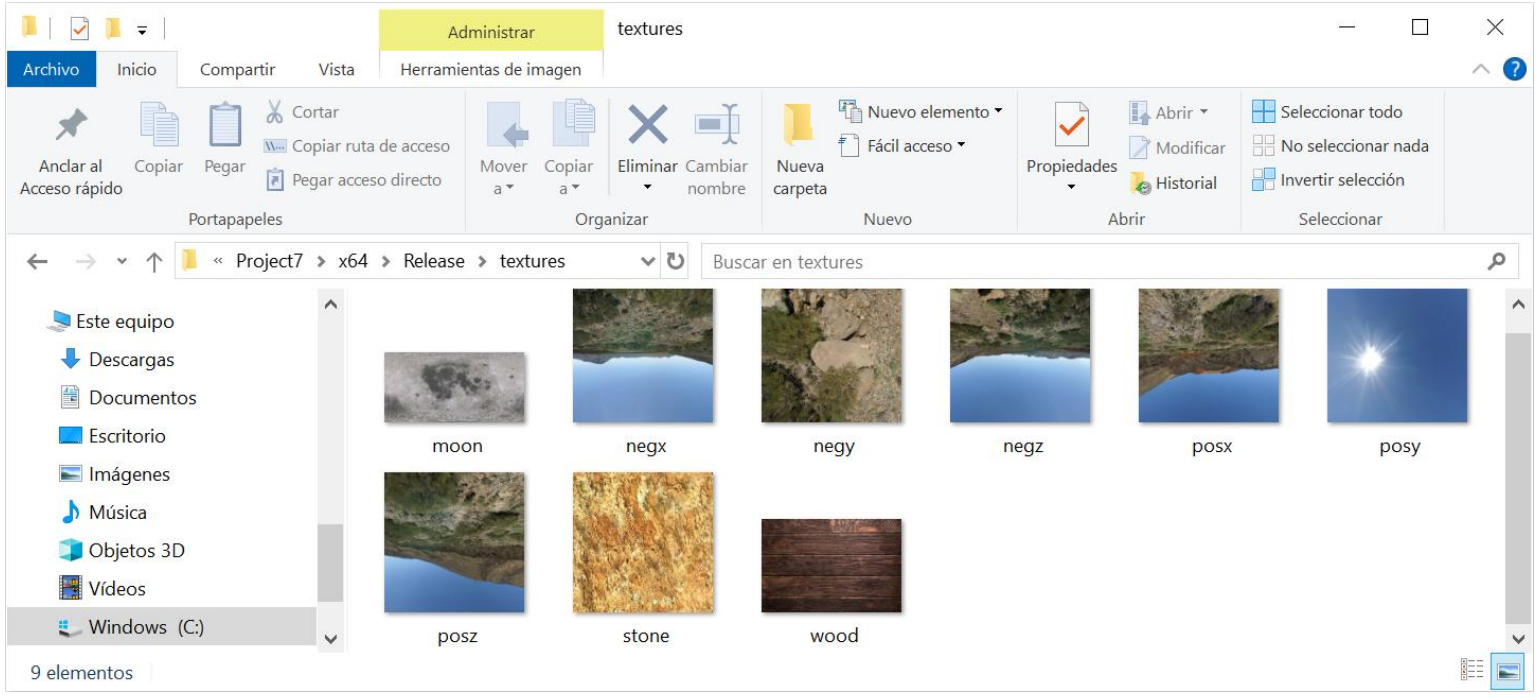
5. Los recursos de la aplicación

- Para introducir el Skybox se han añadido nuevos recursos a la aplicación.
- Por una parte es necesario incluir las 6 imágenes correspondientes al CubeMap.
  - Las imágenes están en el comprimido de la práctica dentro del proyecto Project7/textures
  - Para ello se han definido los recursos IDR_IMAGE4 a IDR_IMAGE9:

IDR_IMAGEX	Imagen
IDR_IMAGE4	posx.jpg
IDR_IMAGE5	negx.jpg
IDR_IMAGE6	posy.jpg
IDR_IMAGE7	negy.jpg
IDR_IMAGE8	posz.jpg
IDR_IMAGE9	negz.jpg

5. Los recursos de la aplicación

- Como puede observarse en la siguiente captura, las imágenes de los ejes X y Z del CubeMap han sido volteadas verticalmente.



5. Los recursos de la aplicación

- Por otra parte, es necesario incluir los dhaders utilizados en el programa gráfico dedicado a mostrar el Skybox.
- Estos shaders se han incluido como los recursos IDR_SHADER3 e IDR_SHADER4.

IDR_SHADERX	Archivo
IDR_SHADER3	SkyboxVertexShader.glsl
IDR_SHADER4	SkyboxFragmentShader.glsl

6. La clase CGModel

- La clase CGModel se ha modificado para incluir el objeto Skybox y el programa gráfico vinculado a él.
- En este caso se ha incluido además un método CameraConstraint() para impedir que la cámara se pueda alejar demasiado de los objetos de la escena.

6. La clase CGModel

- Cabecera CGModel.h

```
#pragma once

#include <GL/glew.h>
#include "CGShaderProgram.h"
#include "CGScene.h"
#include "CGSkybox.h"
#include "CGCamera.h"

class CGModel
{
public:
  void initialize(int w, int h);
  void finalize();
  void render();
  void update();
  void key_pressed(int key);
  void mouse_button(int button, int action);
  void mouse_move(double xpos, double ypos);
  void resize(int w, int h);
private:
  CGShaderProgram* sceneProgram;
  CGShaderProgram* skyboxProgram;
  CGScene* scene;
  CGCamera* camera;
  CGSkybox* skybox;
  glm::mat4 projection;

  void CameraConstraints();
};
```

6. La clase CGModel

- Contenido CGModel.h

```
#include "CGModel.h"
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <GLFW/glfw3.h>
#include <iostream>
#include "CGCamera.h"
#include "CGScene.h"
#include "CGSkybox.h"
#include "resource.h"
```

6. La clase CGModel

```
void CGModel::initialize(int w, int h)
{
  // Crea el programa gráfico para la escena
  sceneProgram = new CGShaderProgram(IDR_SHADER1, IDR_SHADER2, -1, -1, -1);
  if (sceneProgram->IsLinked() == GL_FALSE) return;

  // Crea el programa gráfico para el entorno
  // skyboxProgram = new CGShaderProgram("shaders/SkyboxVertexShader.glsl",
  //"shaders/SkyboxFragmentShader.glsl", NULL, NULL, NULL);
  skyboxProgram = new CGShaderProgram(IDR_SHADER3, IDR_SHADER4, -1, -1, -1);
  if (skyboxProgram->IsLinked() == GL_FALSE) return;

  // Crea la cámara
  camera = new CGCamera();
  camera->SetPosition(0.0f, 5.0f, 30.0f);

  // Crea el skybox
  skybox = new CGSkybox();

  // Crea la escena
  scene = new CGScene();

  // Asigna el viewport y el clipping volume
  resize(w, h);

  // Opciones de dibujo
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_CULL_FACE);
  glFrontFace(GL_CCW);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
}
```

6. La clase CGModel

```
void CGModel::finalize()
{
  delete camera;
  delete scene;
  delete skybox;
  delete sceneProgram;
  delete skyboxProgram;
}
```

```
void CGModel::resize(int w, int h)
{
  double fov = glm::radians(15.0);
  double sin_fov = sin(fov);
  double cos_fov = cos(fov);
  if (h == 0) h = 1;
  GLfloat aspectRatio = (GLfloat)w / (GLfloat)h;
  GLfloat wHeight = (GLfloat)(sin_fov * 0.2 / cos_fov);
  GLfloat wWidth = wHeight * aspectRatio;

  glViewport(0, 0, w, h);
  projection = glm::frustum(-wWidth, wWidth, -wHeight, wHeight, 0.2f, 400.0f);
}
```

6. La clase CGModel

```
void CGModel::render()
{
  glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
  glClear(GL_COLOR_BUFFER_BIT |
GL_DEPTH_BUFFER_BIT);

  glm::mat4 view = camera->ViewMatrix();
  skyboxProgram->Use();
  skybox->Draw(skyboxProgram, projection,
view);
  sceneProgram->Use();
  scene->Draw(sceneProgram, projection, view);
}

void CGModel::update()
{
  camera->MoveFront();
  CameraConstraints();
}
```

6. La clase CGModel

```
void CGModel::key_pressed(int key)
{
 switch (key)
 {
 case GLFW_KEY_UP:
  camera->TurnDown();
  break;
 case GLFW_KEY_DOWN:
  camera->TurnUp();
  break;
 case GLFW_KEY_LEFT:
  camera->TurnCCW();
  break;
 case GLFW_KEY_RIGHT:
  camera->TurnCW();
  break;
 case GLFW_KEY_S:
  camera->SetMoveStep(0.0f);
  break;
 case GLFW_KEY_RIGHT_BRACKET:
 case GLFW_KEY_KP_ADD:
  camera->SetMoveStep(camera->GetMoveStep() + 0.1f);
  break;
 case GLFW_KEY_MINUS:
 case GLFW_KEY_KP_SUBTRACT:
  camera->SetMoveStep(camera->GetMoveStep() - 0.1f);
  break;
 case GLFW_KEY_Q:
  camera->SetMoveStep(0.1f);
  camera->MoveUp();
  camera->SetMoveStep(0.0f);
  break;
 case GLFW_KEY_A:
  camera->SetMoveStep(0.1f);
  camera->MoveDown();
  camera->SetMoveStep(0.0f);
  break;
 case GLFW_KEY_O:
  camera->SetMoveStep(0.1f);
  camera->MoveLeft();
  camera->SetMoveStep(0.0f);
  break;
 case GLFW_KEY_P:
  camera->SetMoveStep(0.1f);
  camera->MoveRight();
  camera->SetMoveStep(0.0f);
  break;
 case GLFW_KEY_K:
  camera->TurnLeft();
  break;
 case GLFW_KEY_L:
  camera->TurnRight();
  break;
 }
}
```

6. La clase CGModel

```
} void CGModel::mouse_button(int button, int action)
{
}

void CGModel::mouse_move(double xpos, double ypos)
{
}

void CGModel::CameraConstraints()
{
  glm::vec3 pos = camera->GetPosition();
  int constraint = 0;
  if (pos.y < 1.0f) { pos.y = 1.0f; constraint = 1; }
  if (pos.y > 40.0f) { pos.y = 40.0f; constraint = 1; }
  if (pos.x > 100.0f) { pos.x = 100.0f; constraint = 1; }
  if (pos.x < -100.0f) { pos.x = -100.0f; constraint = 1; }
  if (pos.z > 100.0f) { pos.z = 100.0f; constraint = 1; }
  if (pos.z < -100.0f) { pos.z = -100.0f; constraint = 1; }
  if (constraint == 1)
  {
    camera->SetPosition(pos.x, pos.y, pos.z);
    camera->SetMoveStep(0.0f);
  }
}
```

7. Resultado