TRANSFORMACIONES Y SHADERS

TRABSFORMACIONES

En esta sección del capítulo se van a programar las transformaciones más utilizadas en motores 3Dque serán la traslación, rotación y escalado. Estas tres transformaciones pueden representarse todas juntas en una matriz 4x4. Pero antes de calcular dicha matriz, debe calcularse la matriz vista (*View Matrix*)

PROGRAMANDO LA VIEW MATRIX

Lo primero que haremos será crear un nuevo método para la clase *Scene()*. Este nuevo método se llamará *Transform()* y se le pasará un objetivo (*target*) y una vista (*eye*).

A primera parte de este método definirá la *View Matrix*.

*float4x4 ModuleScene::Transform(float3 eye, float3 target)*

*{*

*float4x4 resultMatrix;*

*float3 f(target - eye);*

*f.Normalize();*

*float3 s(f.Cross(up));*

*s.Normalize();*

*float3 u(s.Cross(f));*

*viewMatrix[0][0] = s.x; viewMatrix[0][1] = s.y; viewMatrix[0][2] = s.z; viewMatrix[3][0] = 0;*

*viewMatrix[1][0] = u.x; viewMatrix[1][1] = u.y; viewMatrix[1][2] = u.z; viewMatrix[3][1] = 0;*

*viewMatrix[2][0] = -f.x; viewMatrix[2][1] = -f.y; viewMatrix[2][2] = -f.z; viewMatrix[3][2] = 0;*

*viewMatrix[0][3] = -s.Dot(eye); viewMatrix[1][3] = -u.Dot(eye); viewMatrix[2][3] = f.Dot(eye); viewMatrix[3][3] = 1;*

Lo que se está realizando en el primer bloque del nuevo método es definir los tres vectores que después formarán parte de la *view matrix*. Obsérvese que todos los vectores excepto el vector *u* están normalizados.

Una vez se han calculado los vectores *f*, *s* y *u* que compondrán la *View Matrix*, se pasa a definir la mismas en el tercer bloque del método. Al tener que ser una matriz 4x4, se añade una última fila con un vector nulo a excepción de la última columna.

Este método continua con a definición de la matriz de la perspectiva (*Perspective Matrix*). Esta matriz se definirá de la siguiente manera a continuación de la *View Marix*.

*Frustum frustum;*

*float aspect = SCREEN\_WIDTH / SCREEN\_HEIGHT;*

*frustum.type = FrustumType::PerspectiveFrustum;*

MATRIZ DE PROYECCION

Tras la definición de la *Prespective Matrix* nos queda la definición de la matriz de proyección (*Projection Matrix*). Por tanto, tras la definición de la *Perspective Matrix* se añadirán las siguientes líneas con la definición de la *Projection Matrix*:

*frustum.pos = float3::zero;*

*frustum.front = -float3::unitZ;*

*frustum.up = float3::unitY;*

*frustum.nearPlaneDistance = 0.1f;*

*frustum.farPlaneDistance = 100.0f;*

*frustum.verticalFov = math::pi / 4.0f;*

*frustum.horizontalFov = 2.f \* atanf(tanf(frustum.verticalFov \* 0.5f)) \* aspect;*

*math::float4x4 proj = frustum.ProjectionMatrix();*

Tras estas declaraciones sólo nos queda cerrar el método con la devolución del resultado:

*this->viewMatrix = viewMatrix;*

*this->projectionMatrix = proj;*

*resultMatrix = proj \* viewMatrix;*

*return resultMatrix;*

*}*

Lo único que nos queda por realizar, es la multiplicación de los vértices por la matriz de transformación. Para ello incluiremos el siguiente bucle *for* después de la declaración del triángulo:

*for (int i = 0; i < 3; ++i)*

*{*

*float4 res = Transform(eye, target) \* float4(*vertex\_buffer\_data *[i], 1.0f);*

vertex\_buffer\_data *[i] = res.xyz() / res.w;*

*}*

Ahora pasaremos al fichero de cabecera donde lo primero que haremos será realizar el siguiente *include*:

*#include "Math\float4x4.h"*

Tras esto, empezaremos a declarar las variables necesarias para las nuevas variables que hemos creado. Por lo que añadiremos la siguiente línea para la definición del nuevo método.

float4x4 Transform(float3 eye, float3 target);

Tras crear este nuevo método iremos al método *Update()* que es desde donde será llamado. Para ello incuiremos la siguiente línea al inicio del método *Update()*.

*transformationMatrix = Transform(eye, target);*

Si todo ha salido bien, deberías tener algo muy similar al primer triángulo.

SHADERS

Ahora pasaremos a la creación de los *shaders*. Para ello lo primero será rear dos ficheros donde se almacenarán los *shaders*. Para crear los *shaders* los más sencillo es abrir nuestro editor de texto, en mi caso utilizaré el *NotePadX*, y crear un fichero que denominaremos *Default.vs* y contendrá el siguiente código:

*#version 330*

*layout(loaction = 0) in vec3 vertex position;*

*void main()*

*{*

*gl\_Position = vec4(vertex\_position, 1.0);*

*}*

Una vez creado este fichero se procederá a a creación de default.fs. Para su creación se seguirá el procedimiento ya descrito y el código que se escribirá será:

*#version 330 core*

*out vec4 color;*

*void main()*

*{*

*color= vec4(1.0, 1.0, 1.0);*

*}*

Este fichero también se tendrá que importar a nuestro proyecto.

Una vez de vuelta al proyecto y con los *shaders* ya incorporados al mismo, deberá crearse una nueva clase que denominaremos: *Program()*. Esta clase será utilizada para cargar y compilar los *shaders* que acabamos de incorporar.

Para cargar los shaders crearemos un nuevo método que denominares LoadShaderFile y contendrá el siguiente código:

*char\* ModuleProgram::readShaderFile(const char\* shaderPath)*

*{*

*FILE\* file = nullptr;*

*errno\_t err = fopen\_s(&file, shaderPath, "rb");*

*if (file)*

*{*

*fseek(file, 0, SEEK\_END);*

*int size = ftell(file);*

*rewind(file);*

*char\* data = (char\*)malloc(size + 1);*

*fread(data, 1, size, file);*

*data[size] = 0;*

*fclose(file);*

*return data;*

*}*

*return nullptr;*

*}*

No olvidemos que también deben incluirse los *include* necesarios, que para esta clase serán los siguientes:

*#include "ModuleProgram.h"*

*#include <errno.h>*

*#include "glew-2.1.0\include\GL\glew.h"*

El siguiente paso será la creación del método *Init()* que será el encargado de crear y compilar los *shaders*.

El primer bloque de este método será como sigue:

*bool ModuleProgram::Init()*

*{*

*bool ret = true;*

*GLuint vertShader = glCreateShader(GL\_VERTEX\_SHADER);*

*GLuint fragShader = glCreateShader(GL\_FRAGMENT\_SHADER);*

*GLint compileStatus = GL\_FALSE;*

*int logLength = 0;*

En este bloque se crean los dos *shaders*, uno para los vértices y otro para los fragmentos.

Tras este bloque insertaremos el segundo bloque que tendrá el siguiente código:

*char\* vertShaderStr = readShaderFile("default.vs");*

*char\* fragShaderStr = readShaderFile("default.fs");*

Lo que se está haciendo en este bloque es enlazar los ficheros con las definiciones de los *shaders*.

Tras realizar el enlace, se incluirá un bucle *if* para comprobar que el enlace es correcto. Esto se realizará de la siguiente foram:

*if (vertShaderStr == nullptr || fragShaderStr == nullptr)*

*{*

*LOG("Error: Reading shaders failed");*

*return GL\_FALSE;*

*}*

El siguente bloque a incluir será en el que se realice la compilación de los *shaders* y se compruebe si ha habido algún error durante la misma.

*// Compile shaders*

*glShaderSource(vertShader, 1, &vertShaderStr, NULL);*

*glCompileShader(vertShader);*

*glGetShaderiv(vertShader, GL\_COMPILE\_STATUS, &compileStatus);*

*if (!compileStatus)*

*{*

*LOG("Error: Failed compiling vertex shader");*

*CheckCompilationErrors(vertShader);*

*return GL\_FALSE;*

*}*

*glShaderSource(fragShader, 1, &fragShaderStr, NULL);*

*glCompileShader(fragShader);*

*glGetShaderiv(fragShader, GL\_COMPILE\_STATUS, &compileStatus);*

*if (!compileStatus)*

*{*

*LOG("Error: Failed compiling fragment shader");*

*CheckCompilationErrors(fragShader);*

*return GL\_FALSE;*

*}*

El siguient bloque contendrá el código para compilar el módulo:

*// Compile Program*

*program = glCreateProgram();*

*glAttachShader(program, vertShader);*

*glAttachShader(program, fragShader);*

*glLinkProgram(program);*

*glGetProgramiv(program, GL\_LINK\_STATUS, &compileStatus);*

Por último, para finalizar el método se escribirá el bloque para la eliminación de los shaders y el valor de retorno.

*// Remove shaders, we wont need them anymore if they are loaded correctly into Program*

*glDeleteShader(vertShader);*

*glDeleteShader(fragShader);*

*return ret;*

*}*

Tras este médoto añadiremos un méodo *Update()* para que se vaya actualizando. Este méodo será de la siguiente forma:

*update\_status ModuleProgram::Update()*

*{*

*return UPDATE\_CONTINUE;*

*}*

Tras el méodo *Update()* se añadirá un método para la comprobación de erros al que denominares *CheckCompilationErros()*. Dentro de dicho método se escribirá el siguiente código:

*void ModuleProgram::CheckCompilationErrors(GLuint shader) {*

*GLint infoLogLength;*

*glGetShaderiv(shader, GL\_INFO\_LOG\_LENGTH, &infoLogLength);*

*GLchar\* strInfoLog = new GLchar[infoLogLength + 1];*

*glGetShaderInfoLog(shader, infoLogLength, NULL, strInfoLog);*

*LOG(strInfoLog);*

*delete[] strInfoLog;*

*infoLogLength = NULL;*

*glDeleteShader(shader); // Don't leak the shader.*

*}*

Ahora ya, sñolo nos queda incuir el método de limpieza *CleanUp()*, el cual tendrá en su interior el siguiente código:

*bool ModuleProgram::CleanUp()*

*{*

*bool ret=true;*

*return ret;*

*}*

El último paso, como siempre, se realiza en el fihero de cabecera, donde s deberá declara todas las variables y los métodos utilizados.

Nuestro siguiente paso será indicar a *Application* que puede llamar a este módulo para la creación de los *shaders*. Para ello iremos a *Application*, donde incluiremos la siguiente línea en los include:

*#include "ModuleProgram.h"*

También se añadirá la siguiente línea justo antes de la llamada a *Scene*:

*modules.push\_back(program = new ModuleProgram());*

Posteriormente iremos al fichero de cabecera y declararemos nuestra nueva clase añadiendo la siguiente línea justo debajo de la declaración de la última clase:

*class ModuleProgram;*

Lo último que nos queda por realizar en este fichero es la declaración del puntero:

*ModuleProgram\* program = nullptr;*

Una vez efectuado esto, si ejecutamos el motor no se apreciará ningún cambio en el riángulo, eso es por que nuestra clase *Scene()* no está llamando a la clase *Program()*. Así pues, abramos la clase *Scene()* e incluyamos a *ModuleProgram* dentro de los *include*.

Para realizar la llamada a la clase *Program()*, nos situaremos al inicio del método Init() e insertaremos las siguientes líneas:

if (!App->program->program)

{

LOG("Error: Program cannot be compiled");

return false;

}

glUseProgram(App->program->program);

UNIFORMS

En este apartado crearemos los *Uniforms*. Para ello deberemos modificar los ficheros de los *shaders*.

El fichero Default.vs será sustituido por el siguiente:

#version 330

layout(location = 0) in vec3 vertex\_position;

layout(location = 1) in vec2 vertex\_uv0;

uniform mat4 proj;

uniform mat4 view;

uniform mat4 model;

out vec2 uv0;

void main()

{

gl\_Position = proj\*view\*model\*vec4(vertex\_position, 1.0);

uv0 = vertex\_uv0;

}

Y el fichero defult.fs será sustituido por:

#version 330 core

out vec4 color;

in vec2 uv0;

uniform sampler2D texture0;

*void main()*

*{*

*color = texture2D(texture0, uv0);*

*}*

Ahora ya tenemos creado el archivo del *uniform*.

cddf