Documentacion interna

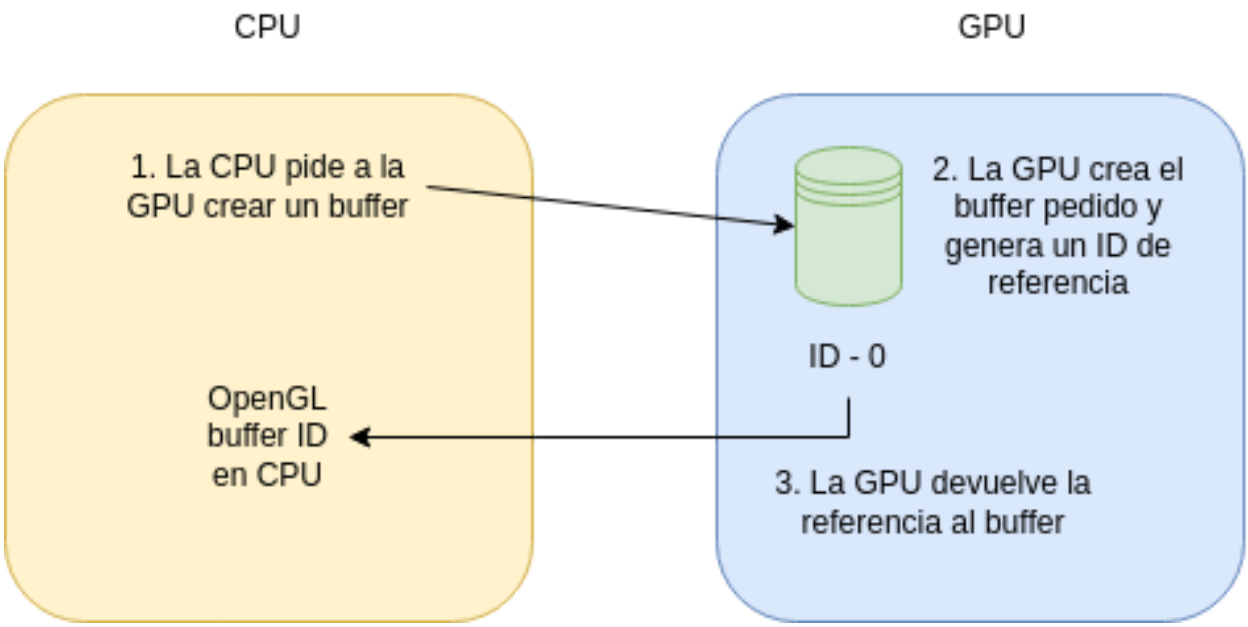
Sistema de renderizado

El sistema de renderizado esta diseñado para agrupar el mayor número de vértices posible y así ejecutar el mínimo número de draw calls*. A cada conjunto de vertices y otros datos que veremos más adelante se les denomina batch.

El sistema de renderizado se basa, en su más alto nivel, en un sistema de capas. Podemos entender estas capas como escenas que podemos ir acumulando una encima de otra.

El la parte de mas bajo nivel, el sistema esta compuesto por texturas, shaders, sprite batch, fuentes de texto y el frame buffer. Estos conceptos luego se usan en clases de más alto nivel, como los sprites o los shapes y en manejadores, como el textureAtlasManager, el fontManager... Los veremos uno por uno.

Para el renderizado vamos a usar una libreria llamada OpenGL, que nos permite trabajar a bastante bajo nivel con la GPU y pintar cosas por pantalla. OpenGL, a muy grandes rasgos, es una maquina de estados que nos permite crear buffers, decirle que tipo de dato va en el buffer, llenar el buffer, usar el buffer y vaciar el buffer. Un buffer no es mas que una region de memoria en la que almacenar datos, referenciada a partir de un ID para luego poder acceder a ese mismo buffer.



Texture

API

Tipo	Metodo	Descripcion
Construc	Texture(const char*)	carga una imagen con loadFromFile
Construc	Texture(Texture*, IntRect)	carga una subtextura a partir de una textura ya existente
bool	loadFromFile(const char*)	carga una imagen almacenada en el pc
bool	loadFromMemory(unsigned char*, int)	carga una imagen generada en tiempo de ejecucion
bool	loadTextTexture(int, int)	crea una textura de fuente de texto
bool	loadTextSubTexture(Vec2I, Vec2I, unsigned char*)	crea subtexturas para un glyph
float	getKb()	devuelve lo que pesa la imagen
IntRect	getRegion()	devuelve la region de imagen a renderizar
uint	getOpenGLTextureID()	devuleve el puntero de la imagen en la GPU

La textura es el elemento basico para poder renderizar imágenes. Sin ella solo podemos renderizar formas geométricas de uno o varios colores en función del shader que usemos, pero para poder renderizar imágenes necesitamos de texturas.

Para crear la textura usamos una librería llamada stb_image, una libreria sencilla de usar que nos devuelve, de una ruta hacia una imagen, un array con todos los datos de todos los pixeles de la imagen (en formato unsigned char), además de otros valores interesantes como el ancho y alto, si usa RGB o RGBA...

```
stbi_set_flip_vertically_on_load(1);
texturePixels = stbi_load(_path, &width, &height, &channels, 0);
GLenum _internalFormat = 0, _dataFormat = 0;
if (channels == 4) {
    _internalFormat = GL_RGBA8;
    _dataFormat = GL_RGBA;
} else if (channels == 3) {
    _internalFormat = GL_RGB8;
    _dataFormat = GL_RGB;
} else {
    LOG_E("Not supported format image")
}
```

Una vez que tenemos estos datos, pasamos a la parte interesante, que es subir los datos a la GPU a través de OpenGL.

Primero de todo tenemos que entender que OpenGL funciona como una máquina de estados, es decir, nosotros le vamos diciendo que debe hacer, pero las cosas no se ejecutan hasta el momento de dibujar en pantalla.

El proceso para crear una textura es el siguiente:

```
glCreateTextures(GL_TEXTURE_2D, 1, &openGLTextureID);

glTextureStorage2D(openGLTextureID, 1, internalFormat, width, height);

glTextureParameteri(openGLTextureID, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTextureParameteri(openGLTextureID, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

glTextureParameteri(openGLTextureID, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTextureParameteri(openGLTextureID, GL_TEXTURE_WRAP_T, GL_REPEAT);

glTextureSubImage2D(openGLTextureID, 0, 0, 0, width, height, dataFormat, GL_UNSIGNED_BYTE,
texturePixels);
```

Primero debemos indicarle a OpenGL que queremos crear una textura mediante glCreateTextures, lo que genera un buffer en la GPU para esta imagen. La variable openGLTextureID es la más importante de todas ya que contiene la referencia dentro de la GPU de donde se encuentra la textura almacenada y es la única forma que tenemos de localizarla para usarla y finalmente limpiarla de memoria.

> Nota: Una vez subida la imagen a la GPU, esta NO ES MODIFICABLE, tendríamos que volver a subirla a la GPU

Después le indicamos el tipo de imágen que vamos a almacenar y qué puntero la contiene (en este caso openGLTextureID) y con los valores que hemos sacado gracias a stb_image. El 1 es para el numero de mipmaps.

Mipmap

Cuando queremos renderizar una imagen, podemos hacerlo de 3 formas diferentes.

1. En su tamaño original
2. Maximificada
3. Minimizada

Claro es que no renderizar una imagen en su tamaño original, hara que de una u otra forma, pierda calidad. Por ello aparecen los mipmaps. Un mipmap es una copia optimizada de la textura original en un tamaño diferente al original, de esta forma no tenemos que estar escalando las imagenes en tiempo de ejecución.

En el caso de arriba, la imagen original es la segunda mayor y hemos creado varios mipmaps de la imagen original, para poder usarlas en tiempo de ejecución.

Después de indicar el tipo de imagen que vamos a almacenar en GPU y de decirle la cantidad de mipmaps, los canales (RGB o RGBA en nuestro caso) y el tamaño de la imagen, le indicamos las funciones que aplicar a la hora de escalar la imagen ya sea a menor o a mayor. En nuestro caso usaremos GL_LINEAR y GL_NEAREST, aunque podriamos usar los dos iguales tanto para escalado hacia arriba como para abajo.

Wrap seria para repetir la imagen (estilo un tile map) en altura o en anchura y usamos el mismo valor para ambas.

Finalmente viene la parte mas interesante, que es la de subir los datos a la GPU. Lo hacemos con la función glTextureSubImage2D, cuyos parametros son

1. Puntero a la imagen en GPU (openGLTextureID)
2. Mipmap a usar (0, o sea el primero)
3. OffsetX, 0 si queremos que la imagen empiece en origen en X
4. OffsetY, 0 si queremos que la imagen empiece en origen en Y
5. Anchura de la imagen
6. Altura de la imagen
7. Formato usado
8. Tipo de dato en que se mandan los datos GL_UNSIGNED_BYTE o lo que es lo mismo unsigned char
9. El array conteniendo todos los pixels de la imagen

Con esto ya habriamos subido la imagen a la GPU y ya seria localizable.

Esta es la primera forma de crear una textura. Tenemos otras formas, como generar una textura en tiempo de ejecucción mediante loadFromMemory o mediante una imagen ya existente con Texture(Texture*, IntRect) que es el segundo constructor de esta clase. Su uso lo veremos cuando documentemos el Texture Atlas Manager.

Finalmente para liberar la memoria de la textura y limpiar usamos:

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

Con 1 los niveles de mipmap y openGLTextureID el puntero al buffer de la GPU que contiene la textura.

Vertex Buffer

Este es uno de los buffers mas importantes e imprescindibles de todos, ya que es por donde vamos a enviar datos a la GPU para despues poder usarlos en los Shaders y asi poder pintar correctamente nuestras texturas por pantalla. Este tipo de buffer es un poco diferente del de la textura, ya que aqui podemos enviar tantos bloques de datos como queramos, pero tenemos que especificarle muy bien como, en que orden y cuantos datos le estamos mandando para que OpenGL sepa interpretarlos sin problema. Vamos con ello.

Lo primero de todo es generar el buffer en la GPU, y para generar un buffer de vertices tenemos la funcion:

```
glGenBuffers(1, &vbo);
```

El primer parametro indica el numero de buffers que queremos crear y el segundo parametro es la referencia que OpenGL nos devuelve hacia dicho buffer, para que podamos acceder a el siempre que queramos.

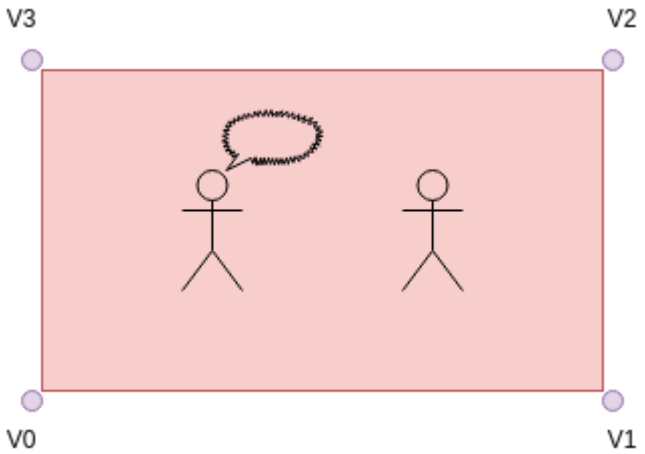
Como ya sabemos, OpenGL es una maquina de estados, por lo que generar el buffer no implica que lo hayamos seleccionado para trabajar con el. Por ello para comunicarle a OpenGL que queremos trabajar con un buffer usamos:

```
glBindBuffer(GL_ARRAY_BUFFER, vbo);
```

Con esta funcion OpenGL ya sabe que las proximas operaciones se ejecutaran sobre el buffer que acabamos de crear. GL_ARRAY_BUFFER quiere decir que los datos que vamos a mandarle a la GPU estan en formato de un array, es decir, todos los atributos que vamos a enviarle se los mandaremos embutidos en un mismo array.

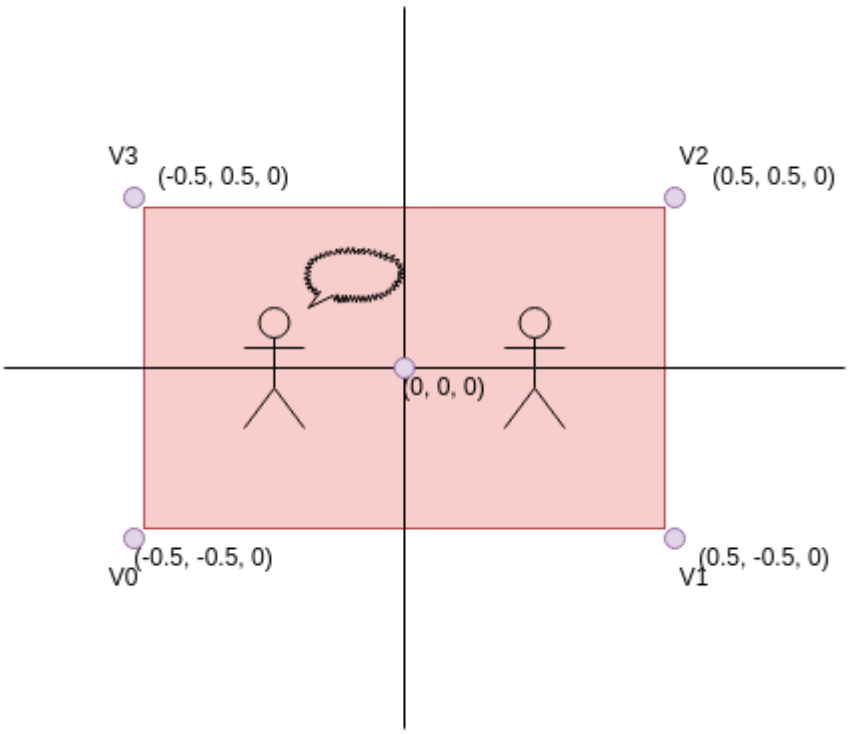
Pero, que es eso de atributos y que tipo podemos mandar? Un atributo al final de cuentas es un bloque de informacion acerca de un vertice, vamos a ver un ejemplo.

Si queremos renderizar una imagen, es decir, un rectangulo, tenemos que decirle a la GPU como hacerlo exactamente y esto se hace mediante los atributos:



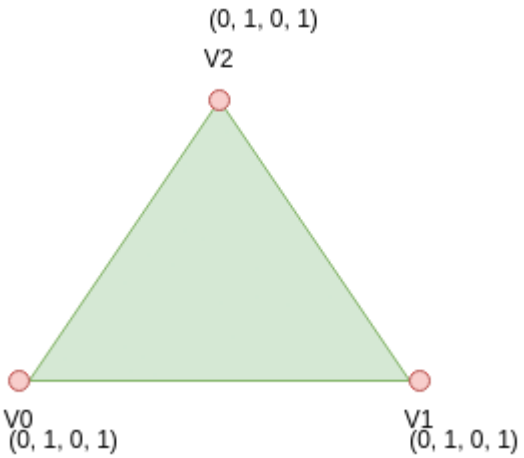
Tenemos la imagen de arriba, lo primero es identificar sus vertices, que son v_0, v_1, v_2 y v_3. Cada uno de estos vertices va a tener unos atributos, que pueden ser, entre otros, las coordenadas que ocupan dentro de la pantalla (x, y, z), el color que va a tener el vertice (r, g, b, a), las coordenadas de la textura (x, y)... Podriamos incluir mas si fuesen necesarios.

Las coordenadas son claras, pero veamos un dibujo rapido:



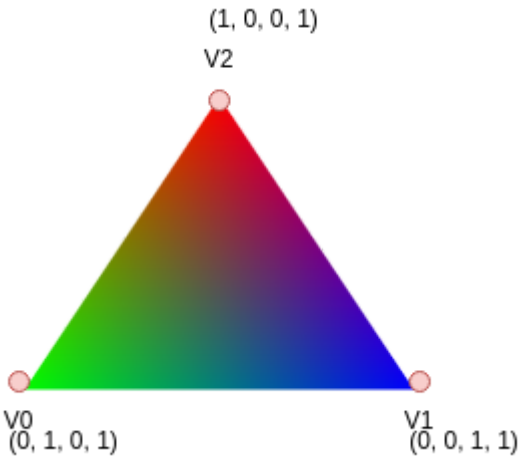
Como vemos las coordenadas simplemente nos indican donde esta cada vertice en el espacio, sencillo.

El color es algo mas complicado. Vamos a ver como funciona en un simple triangulo sin textura, un triangulo que solo tiene color de fondo. El formato de color en el motor es RGBA, es decir, Red-Green-Blue-Alpha y sus valores estan normalizados, es decir, [0, 1].



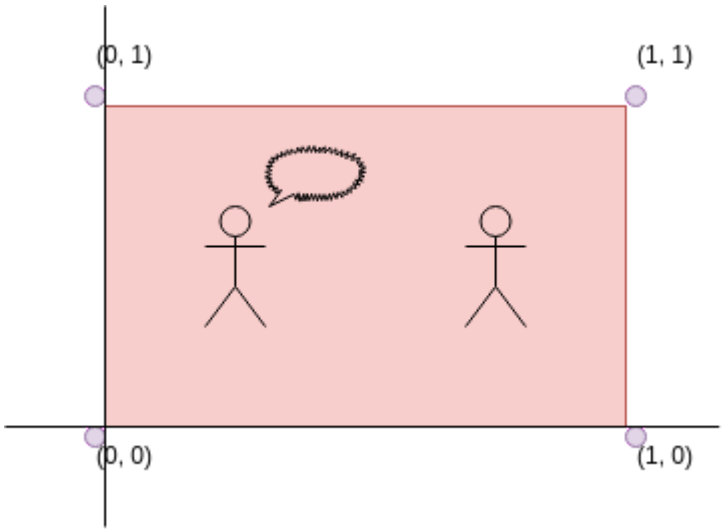
Este triangulo es verde y lo es porque hemos definido que v_0, v_1 y v_2 tengan como color (0, 1, 0, 1), es decir, color verde.

Si a cada vertice le damos un color diferente, lo que obtendremos son combinaciones de los colores de todos los vertices por el triangulo:

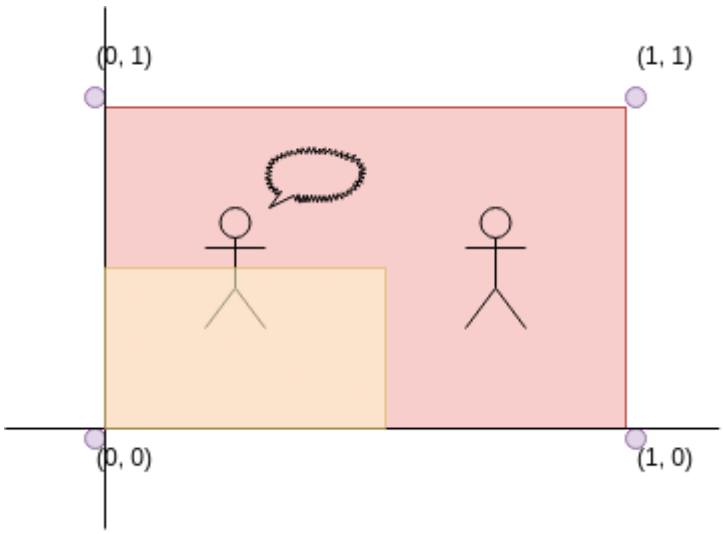


Si usamos una textura, para que tenga su color natural usaremos blanco en todos los vertices, es decir, (1, 1, 1, 1). Pero si queremos tintar la textura de algun color, pondremos los vertices del color que queramos tintar.

Finalmente vamos con las coordenadas de la textura. Estas coordenadas indican que cantidad de la textura vamos a dibujar. Funcionan de la siguiente manera:



Es decir, si a v_0 le asignamos la coordenada de textura (0, 0), a v_1 (1, 0), a v_2 (1, 1) y a v_3 (0, 1) renderizaremos la imagen entera. Si por ejemplo las coordenadas son v_0 (0, 0), v_1 (0.5, 0), v_2 (0.5, 0.5) y v_3 (0, 0.5) entonces solo se renderizara la parte marcada en amarillo en la imagen de abajo:



Vista la teoria, vamos a ver como hariamos esto con codigo. Ya hemos creado anteriormente el buffer y lo hemos asignado para trabajar con el, ahora vamos a indicarle los atributos que vamos a usar, para ello:

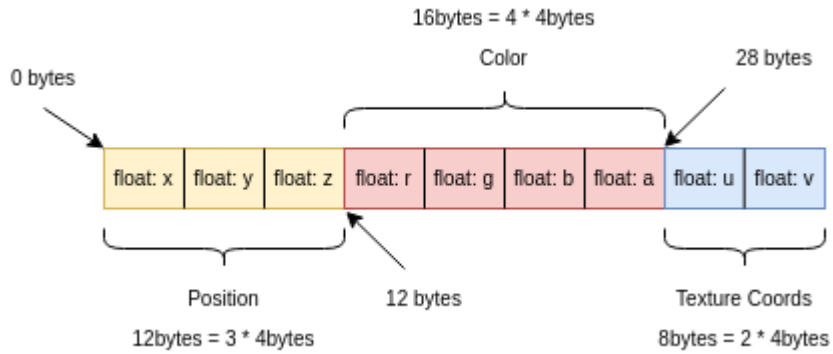
```
struct Vertex2dUVColor {
    glm::vec3 position;
    glm::vec4 color;
    glm::vec2 texCoord;
};

GLsizei _structSize = sizeof(Vertex2dUVColor);

glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, _structSize, (void*) nullptr);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, _structSize, (void*)(4 * 3));
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, _structSize, (void*)(4 * 3 + 4 * 4));
```

Muchas cosas nuevas, vamos una por una. Primero, lo mas normal es agrupar todos los atributos que vamos a enviar a la GPU en un struct, de esta forma es mas facil trabajar al crear los atributos de un vertice. En nuestro caso primero tenemos un vector de 3 coordenadas que es la posicion, otro de 4 que es el color y otro de 2 que son las coordenadas de la textura.

La funcion glVertexAttribPointer tiene muchos parametros:

1. Location en el Shader.
2. Numero de elementos que tiene el atributo (vec2 -> 2, vec3 -> 3, mat4 -> 16...)
3. El tipo de valor que tiene cada elemento, en la mayoria de los casos son floats
4. Si esta o no normalizado, es decir, entre [0,1]
5. El tamaño de la estructura de datos
6. Offset desde el que empieza

Creo que el valor mas complicado de entender es el sexto, vamos con ello. La disposicion en memoria de nuestra estructura de datos Vertex2dUvColor es la siguiente:



Como podemos ver, la posicion tiene 3 elementos de tipo float y con offset 0 bytes, es decir, el punto donde empiezan en memoria los datos de la posicion es 0, por eso:

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, _structSize, (void*) nullptr);
```

Tiene en el segundo parametro con 3, porque tiene 3 elementos, el tercer parametro es un float y el sexto es nullptr, o lo que es lo mismo, 0.

```
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, _structSize, (void*)(4 * 3));
```

En su sexto parametro ponemos 4 * 3, ya que cada float ocupa 4 bytes y por ello la informacion del color empieza despues de los 3 primeros 4 bytes.

Lo mismo ocurre para el atributo de las coordenadas de la textura, solo que en este caso hay que sumar los 3 * 4bytes de la posicion + los 4 * 4bytes del color.

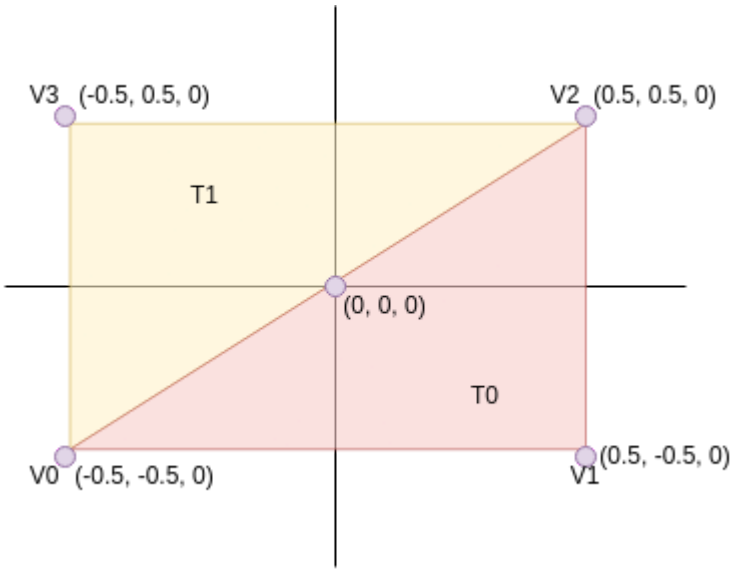
Con esto ya tendriamos nuestro buffer mas que listo. Por ultimo, tenemos que decirle a OpenGL que por el momento no vamos a usar mas este buffer y lo hacemos asi:

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Renderizar nuestro primer rectangulo, o casi

Por el momento ya hemos generado un buffer para la textura, la hemos subido a la GPu, hemos creado el buffer para los vertices y ahora tenemos que rellenarlo para poder pintar la imagen por pantalla.

Lo cierto es que a la GPU no le gusta pintar rectangulos formas geometricas con mas vertices de forma directa, a la GPU le gusta dibujar trinagulos, y hay que reconocer que es muy buena haciendolo. Lo que quiero decir con esto es que la imagen que vimos en el apartado anterior, ahora la tendremos que dividir en dos triangulos que estaran pegados y que formaran la imagen final, por lo que tendriamos lo siguiente:



Asi quedarian los dos triangulos, T0 siendo el rojo con vertices v_0, v_1, v_2 y el triangulo amarillo T1 con vertices v_0, v_2 y v_3. Como veis, al estar pegados se renderizaran como un solo rectangulo, ahi esta el truco.

Vamos primero a dibujar un rectangulo sin textura, que es mas sencillo y luego dibujaremos un trinagulo con textura. Cuales son los pasos?

```
Vertex2dUVColor _vertices[6] = {
//        position          color       texture coords
    {{-0.5f, -0.5f, 0}, {0, 0, 1, 1}, {0, 0}},
    {{ 0.5f, -0.5f, 0}, {0, 0, 1, 1}, {0, 0}}, // T0
    {{ 0.5f,  0.5f, 0}, {0, 0, 1, 1}, {0, 0}},

    {{-0.5f, -0.5f, 0}, {0, 0, 1, 1}, {0, 0}},
    {{ 0.5f,  0.5f, 0}, {0, 0, 1, 1}, {0, 0}}, // T1
    {{-0.5f,  0.5f, 0}, {0, 0, 1, 1}, {0, 0}}
};


glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, (long)(sizeof(Vertex2dUVColor) * 6), &_vertices[0],
GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);


glEnableVertexAttribArray(0); // position
glEnableVertexAttribArray(1); // color
glEnableVertexAttribArray(2); // textureCoords

glDrawArrays(GL_TRIANGLES, 0, 6);

glDisableVertexAttribArray(0);
glDisableVertexAttribArray(1);
glDisableVertexAttribArray(2);
```

Nuevamente, muchas cosas tenemos aqui. Vamos una por una. Lo primero es crear los vertices y como podemos ver creamos 6 de ellos, cada uno de los vertices de los dos triangulos T0, T1 con sus respectivos atributos posicion, color y coordenada de textura. Ahora mismo hemos puesto en todos que la coordenada de la textura sea (0, 0) y es porque ahora mismo no vamos a hacer uso de ese atributo, entonces podemos dejarlo en (0, 0) porque vamos a dibujar un rectangulo con color plano azul.

Lo siguiente es decir en que buffer de vertices (vertex buffer) vamos a meter esta informacion, y para ello hacemos el bind. Una vez "bindeado", le indicamos el tamaño que va a ocupar y le decimos desde que punto del array debe de inicarse. El ultimo valor de la funcion es GL_STATIC_DRAW, de momento vamos a dejarlo asi y ya explicaremos mas adelante por que lo usamos. Otra opcion posible seria GL_DYNAMIC_DRAW, pero como digo, lo veremos mas adelante.

La siguiente parte tiene que ver con la activacion de atributos en el Shader, lo que veremos mas adelante, pero si que teneis que fijaros que los valores 0, 1, 2 se corresponden con la posicion, color y coordenada de textura que indicamos al generar el buffer cuando usamos glVertexAttribPointer.

Una vez hemos activado los atributos para el Shader, podemos dibujar de dos formas, con glDrawArrays y con glDrawElements. En nuestro caso y por no usar un index buffer, usamos glDrawArrays.

El primer parametro es como OpenGL debe dibujar los datos que le hemos pasado y tenemos varias formas:

`GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_LINE_STRIP_ADJACENCY`, `GL_LINES_ADJACENCY`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP_ADJACENCY`, `GL_TRIANGLES_ADJACENCY` and `GL_PATCHES`

Como nosotros queremos triangulos, usamos GL_TRIANGLES. El cero es para ver en que punto del array empezamos, entonces en 0 porque queremos renderizar todos los vertices y finalmente el numero de vertices que hay que renderizar.

Por ultimo, desactivamos las locations que habiamos activado anteriormente. Y con esto estamos listos ya para mostrar por pantalla el rectangulo? Pues no! Para poder renderizar por pantalla el rectangulo necesitamos usar lo que se llama un Shader, que lo veremos a continuacion.

Shaders

Con las versiones viejas de OpenGL y con la textura cargada ya podriamos empezar a renderizar nuestras texturas y rectangulos, pero las versiones viejas tienen ciertos problemas porque tienen un pipeline estatico, frente a las nuevas que tienen uno dinamico gracias a los shaders.

Un shader no es mas que un programa, es decir, es codigo y el lenguaje, en el caso de OpenGL es GLSL. Este codigo tiene la peculiaridad de que en vez de ejecutarse en la CPU como el codigo que habituamos a escribir, lo hace en la GPU, es decir, es una forma de programar nuestra GPU. Y para que exactamente queremos poder programar la GPU? Pues para indicarle a nuestro motor como y donde debe pintar las cosas en la pantalla.

Los shaders no son mi fuerte, ya que no se hacerlos muy complejos, pero si que se como funcionan y es lo que voy a explicar. Pera ello vamos por pasos mostrando primero la API del motor.

API

Tipo	Metodo	Descripcion
Construc	Shader()	
uint	loadFromFiles(const std::string&, const std::string)	carga un shader almacenado en un archivo
uint	loadFromString(const std::string&, const std::string&)	carga un shader escrito en un string
uint	getShaderID()	devuelve el puntero al buffer donde esta almacenado el programa en la GPU

Lo primero de todo, un shader esta formado por varios sub-fragmentos de codigo que se ocupan de una parte del renderizado diferente. Minimo necesita dos, el Vertex Shader y el Fragment Shader. Nos centraremos solo en estos dos ya que son los mas importantes y obligatorios. Cada shader se ejecuta un numero bastante alto de veces y son operaciones todo el rato iguales que pueden paralelizarser. La GPU es muy buena en ejecutar secuencias de instrucciones iguales seguidas, es por eso mismo que el renderizado se ejecuta en la GPU ya que tienen muchos cores y conseguien paralelizar muchos y ejecutarlos muy rapidamente.

Vertex Shader

El vertex shader es el encargado de indicarle a la GPU donde estan los vertices que se van a renderizar. Este shader se ejecuta tantas veces como vertices vayamos a renderizar, es decir, si tenemos un cuadrado, tendremos 4 o 6 vertices (dependiendo del tipo de renderizado implementado). Con este shader podemos pintar en pantalla nuestras imagenes tal cual las vemos en el visor de imagenes de nuestro ordenador o podemos hacer cosas mas interesantes y mover estos vertices segun nos interese para crear efectos, como este:

La estructura del shader es similar al main.cpp de C++:

```
#version 400 core

layout(location = 0) in vec3 in_position;
layout(location = 1) in vec4 in_color;

out vec4 color;

void main(void) {
    color = in_color;
    gl_Position = vec4(in_position, 1);
}
```

Lo importante es el valor que finalmente tenga la variable gl_Position ya que es la posicion final de cada uno de los vertices.

> Nota: gl_Position es un vec4. Veremos por que cuando expliquemos la Model View Projection Matrix.

Este es un ejemplo muy basico para poder renderizar un rectangulo del color que nosotros queramos.

Fragment Shader

Esta parte del shader se encarga de decirle a la GPU como debe pintar todos y cada uno de los pixeles a dibujar en la pantalla. Como podreis imaginar, es un shader que debe ser eficiente, ya que puede llegar a ejecutase una cantidad ingente de veces. En una imagen 100x100px se esta ejecutando 10.000 veces.

Este shader permite desde el mas simple de los renderizados, como es pintar las imagenes con su color original, hasta hacer que esten en escala de grises o en negativo, generar un blur, pintarlas de un solo color, generar un delineado al rededor de la imagen, crear niebla, generar efecto Toon, efecto de pixel art, glow... Tiene mil aplicaciones.

El formato es igual que el del vertex shader, pero calculamos cosas distintas:

```
#version 400 core

in vec4 color;

void main(void) {
    gl_FragColor = color;
}
```

Nuevamente hay mil cosas que explicar y las veremos mas adelante. Lo importante de este shader es el valor final de gl_FragColor que es color final de cada pixel.

Compilacion del shader

Como hemos dicho, un shader es un programa, y si algo sabemos los programadores es que de una u otra forma, antes de ejecutar el codigo hay que comprobar la sintaxis y compilar dicho codigo. Pues los shaders no son una excepcion y esto lo hacemos en tiempo de ejecucion. Para ello lo primero que hacemos es crear un buffer para el shader, despues indicarle el codigo asociado al buffer y compilarlo:

```
GLuint _shaderID = glCreateShader(_shaderType);

const char* _shaderCodePointer = _shaderCode.c_str(); // Esto viene de un string
int _shaderCodeLength = (int)_shaderCode.size();

glShaderSource(_shaderID, 1, &_shaderCodePointer, &_shaderCodeLength);
glCompileShader(_shaderID);
```

_shaderType puede tener varios valores, pero en nuestro caso sera o GL_VERTEX_SHADER o GL_FRAGMENT_SHADER en funcion de cual queramos compilar.

Una vez compilado, tenemos que ver si ha tenido exito o si ha dado algun problema:

```
GLint _isCompiled;
glGetShaderiv(_shaderID, GL_COMPILE_STATUS, &_isCompiled);

if (!_isCompiled) {
    char _infolog[1024];
    glGetShaderInfoLog(_shaderID, 1024, nullptr, _infolog);
    LOG_E("Shader compile failed with error: ", _infolog)
    glDeleteShader(_shaderID);
    _shaderID = 0;
    return false;
}
```

> Nota: LOG_E es una macro que usa el propio motor, se puede quitar sin problemas, es solo para sacar en rojo por pantalla que ha habido un error.

Con glGetShaderiv comprobamos si el resultado esta compilado o no. En el caso de no ser compilado eliminamos el shader y devolvemos false para saber que hay error. Si no da error continuamos.

```
GLuint _bufferShaderID = glCreateProgram();
glAttachShader(_bufferShaderID, _shaderID)
glLinkProgram(_bufferShaderID);
```

Con estos ultimos pasos lo que hacemos primero es generar el buffer para el programa compilado final _bufferShaderID, despues le indicamos que a este puntero al buffer le corresponde el codigo compilado anteriormente en el buffer con ID _shaderID y finalmente el link junta todas las partes compiladas en un solo programa, el shader final.

Como programar los shaders

Lo primero es que cada shader debe programarse en un archivo en separado. Bueno, esto es mentira, pero en mi opinion es la forma mas ordenada de hacerlo. Con que al final tengas un string que contenga todo el codigo del vertex shader y otro con el del fragment es mas que suficiente, pero a mi me gusta separarlo en dos archivos.

Vamos a coger el codigo anterior del vertex shader y analizarlo:

```
#version 400 core

layout(location = 0) in vec3 in_position;
layout(location = 1) in vec4 in_color;
layout(location = 2) in vec2 in_uv;

uniform float dummyUniform;

out vec2 uv;
out vec4 color;

void main(void) {
    uv = in_uv;
    color = in_color;
    gl_Position = vec4(in_position, 1);
}
```

Lo primero es poner siempre la #version y es obligatorio y muy importante ya que no todas las versiones de GLSL soportan las mismas caracteristicas. En general, poniendo de la 400 para arriba, no deberiais tener problemas.

Despues vemos layout, location, in, uniform, mat4, varios vec, out... Vamos, un monton de palabras clave complicadas. Vamos de una en una.

Lo primero que tenemos que saber es que estas variables se pasan desde la CPU a la GPU, es decir, no obtienen sus valores magicamente, si no que primero se calculan en CPU y la GPU los usa. layout, uniform son las diferentes formas en que le podemos enviar datos desde la CPU a la GPU y como podreis imaginar, cada forma de envio es diferente. in remarca que es un valor de entrada para el shader.

El mas facil de todos es uniform. Este tipo de variable se envia en dos pasos mediante funciones de OpenGL:

```
GLint _location = glGetUniformLocation(_bufferShaderID, "dummyUniform");
glUniform1f(_location, 1, GL_FALSE, 3.14f);
```

Primero necesitamos obtener el _location, es decir, donde esta en el shader la variable llamada "dummyUniform". Esta puede o no ser la mejor forma de hacerlo, ya que puede haber equivocaciones al escribir y el string debe coincidir con el nombre de la variable en el shader, pero es la que ofrece OpenGL, que yo sepa por lo menos.

Vale, tenemos que con la primera linea de codigo hemos descubierto donde se encuentra la variable, ahora podemos darle un valor. Para ello usamos la funcion

glUniform1f que nos permite asignar como valor a la variable un float. Hay un monton de funciones diferentes dependiendo del tipo de variable que tenga el shader, aqui link. Para todos los tipos de variables soportadas por GLSL aqui el [link](Data Type (GLSL) - OpenGL Wiki).

Algo muy importante de las variables de tipo uniform es que tienen el mismo valor para todo el frame de renderizado.

La siguiente forma que usamos para enviar datos de la CPU al shader es mediante layout y location. Ahora es donde ponemos en practica lo aprendido en el apartado anterior donde primero usamos glVertexAttribPointer y luego glEnableVertexAttribArray. En cada uno de ellos el primer parametro era un int y como podeis ver, ese valor se corresponde con el location. En el location 0 del shader tenemos un vec3 que es la posicion y con glVertexAttribPointer y glEnableVertexAttribArray le indicamos que en el valor 0 le enviaremos un vec3 y asi sucesivamente con todos (excepto para el vec2 de la coordenada de la textura que lo he quitado ahora para simplificar el ejemplo, lo pondremos de nuevo un poco mas adelante).

Las variables out son valores que van a pasar al siguiente shader de la cadena de shaders, en nuestro caso son valores que pasaran al Fragment Shader, es decir, vamos a pasar los valores de las coordenadas de la textura y del color al fragment shader, donde este shader los utilizara.

Vamos con el codigo del Fragment Shader:

```
#version 400 core

in vec2 uv;
in vec4 color;

void main(void) {
    gl_FragColor = color;
}
```

Como vemos ahora tenemos in y no out. En este caso las variables uv y color reciben el valor *out* que tenian en el Vertex Shader. gl_FragColor es tambien un vec4 y el color final que va a tener cada uno de los pixeles.

Con esto ya tenemos una nocion basica de como funciona un shader y como poder renderizar nuestro primer rectangulo.

Nuestro primer rectangulo, ahora si

En el motor usamos GLFW como libreria de manejo de la ventana, el codigo que se muestra ahora relacionado con GLFW no sera explicado hasta que lleguemos a la seccion donde expliquemos el sistema de ventana que usa el motor.

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <stdlib.h>
#include <stdio.h>
#include <glm/mat4x4.hpp>

struct Vertex {
    glm::vec3 pos;
    glm::vec4 color;
} _vertices[6] = {
                {{-0.5f, -0.5f, 0}, {1.f, 0.f, 0.f, 1} },
                {{0.5f, -0.5f, 0}, {1.f, 0.f, 0.f, 1} },
                {{0.5f, 0.5f, 0}, {1.f, 0.f, 0.f, 1} },

                {{-0.5f, -0.5f, 0}, {1.f, 0.f, 0.f, 1} },
                {{0.5f, 0.5f, 0}, {1.f, 0.f, 0.f, 1} },
                {{-0.5f, 0.5f, 0}, {1.f, 0.f, 0.f, 1} }
        };

static const char* _vertexShaderSrc =
        "#version 400\n"
        "uniform mat4 MVP;\n"
        "layout (location = 0) in vec3 vPos;\n"
        "layout (location = 1) in vec4 vCol;\n"
        "out vec4 color;\n"
        "void main()\n"
        "{\n"
        "    gl_Position = vec4(vPos, 1.0);\n"
        "    color = vCol;\n"
        "}\n";

static const char* _fragmentShaderSrc =
        "#version 400\n"
        "in vec4 color;\n"
        "void main()\n"
        "{\n"
        "    gl_FragColor = color;\n"
        "}\n";

static void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods) {
    if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
        glfwSetWindowShouldClose(window, GLFW_TRUE);
}
```

```
int main(void) {
    // Saltar hasta la parte de glewInit
    GLFWwindow* window;
    GLuint _vertexBuffer, _vertexShader, _fragmentShader, _program;

    if (!glfwInit())
        exit(EXIT_FAILURE);

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 2);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);

    window = glfwCreateWindow(640, 480, "Simple example", NULL, NULL);
    if (!window) {
        glfwTerminate();
        exit(EXIT_FAILURE);
    }

    glfwSetKeyCallback(window, key_callback);

    glfwMakeContextCurrent(window);
    glfwSwapInterval(1);

    // Aqui ya hay que prestar atencion
    glewInit();
    glGenBuffers(1, &_vertexBuffer);
    glBindBuffer(GL_ARRAY_BUFFER, _vertexBuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(_vertices), _vertices, GL_STATIC_DRAW);

    _vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(_vertexShader, 1, &_vertexShaderSrc, nullptr);
    glCompileShader(_vertexShader);

    _fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(_fragmentShader, 1, &_fragmentShaderSrc, nullptr);
    glCompileShader(_fragmentShader);

    _program = glCreateProgram();
    glAttachShader(_program, _vertexShader);
    glAttachShader(_program, _fragmentShader);
    glLinkProgram(_program);

    glDeleteShader(_vertexShader);
    glDeleteShader(_fragmentShader);

    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(_vertices[0]), (void*) 0);
```

```
    glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, sizeof(_vertices[0]), (void*)
(sizeof(float) * 3));

    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);
    glUseProgram(_program);

    while (!glfwWindowShouldClose(window)) {
        int width, height;

        // Codigo aun no importante
        glfwGetFramebufferSize(window, &width, &height);
        glViewport(0, 0, width, height);
        glClear(GL_COLOR_BUFFER_BIT);
        glClearColor(0.3f, 0.3f, 0.3f, 1.0f);

        // Este si es importante
        glDrawArrays(GL_TRIANGLES, 0, 6);

        // Lo mismo, aun nada
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    // Importante
    glDeleteProgram(_program);
    glDeleteBuffers(1, &_vertexBuffer);

    // Aun no
    glfwDestroyWindow(window);
    glfwTerminate();
    return 0;
}
```

Como podeis ver, hay un moton de codigo aqui. Todo lo que tenga que ver con GLFW de momento lo pasamos. Sin contar eso, todo el codigo ya se ha explicado como funciona en los apartados anteriores, excepto glewInit(). Para usar OpenGL podemos usar varias librerias, como glad, pero en nuestro caso he decidido usar glew, por ninguna razon en particular, podeis usar glad si quereis. glewInit() inicia todo el sistema de OpenGL y debe ejecutarse antes de llamar a cualquier funcion de OpenGL. Devuelve un valor que si es distinto de 0, significa que hay un error, asi que tened cuidado con eso.

Ahora mismo hemos dibujado una imagen sin textura, dibujaremos una con textura un poco mas adelante.

Sistema de coordenadas OpenGL vs Coordenadas de mundo

Este tema en particular no se si todo el mundo lo aborda de la misma manera, pero voy a explicar como lo hago yo.

Como hemos visto en el ejemplo completo anterior, las posiciones de los vertices estaban siempre entre [-1, 1] tanto para x como para y, lo que indica que la posicion mas lejana a la izquierda renderizable es -1, a la derecha 1 (en el eje x esto es mentira, lo veremos en el siguiente apartado), arriba 1 y abajo -1. Como podreis imaginar no es especialmente amigable tener que decir que en pantalla nuestros objetos estan en (0.12, -0.53), si no que si nuestra ventana tiene una dimension de por ejemplo, 1280x720, nos gustaria poder dar las coordenadas entre los puntos [-640, 640] en el eje x y [-360, 360] en el eje y, algo mucho mas conveniente.

Bueno, el caso es que si se las damos asi directamente a OpenGL, no le gusta nada, ya que a el le gustan las coordenadas entre [-1, 1] (siempre que estemos dentro del espacio de la camara, y sin mover esta, lo veremos mas adelante, para nosotros ahora la camara es estatica y solo entran cosas entre [-1, 1]).

Muy bonita la explicacion, pero como hacemos esto? Pues facil, tenemos que normalizar el tipo de coordenadas que a nosotros nos interesa usar para que OpenGL pueda entenderlas, vamos con ello:

```
float windowWidth = 1280;
float windowHeight = 720;

float worldCoordX = 250;
float worldCoordY = -100;

float openGLCoordX = worldCoordX / (windowWidth / 2.f);
float openGLCoordY = worldCoordY / (windowHeight / 2.f);
```

Y con esto ya estaria. Dividimos la altura y anchura entre 2 porque hay que dividir la pantalla en dos trozos iguales, uno positivo y otro negativo, tanto en x como en y.

Aspect Ratio

Aspect ratio es la relacion que hay entre la anchura y la altura de nuestra ventana, es decir:

```
float windowWidth = 1280;
float windowHeight = 720;
float aspectRatio = windowWidth / windowHeight;
```

Y de que nos sirver esto? Pues vamos con ello. Si ejecutamos de nuevo el codigo del ejemplo, veremos que las coordenadas deberian de generar un cuadrado, pero estan generando un rectangulo.

La razon es porque la pantalla que hemos creado no es cuadrada, al igual que practicamente ningun monitor de los que existen actualmente y aun no siendo cuadrada, le estamos diciendo que la distancia en el eje x [-1, 1] es la misma que la que hay en el eje y [-1, 1], cuando es mentira porque la anchura es mayor que la altura. El aspect ratio soluciona este problema, ya que la distancia real en x es [-aspectRatio, aspectRatio].

Esto nos hace tener que refactorizar el codigo del apartado anterior a lo siguiente:

```
float windowWidth = 1280;
float windowHeight = 720;

float worldCoordX = 250;
float worldCoordY = -100;

float openGLCoordX = aspectRatio * worldCoordX / (windowWidth / 2.f);
float openGLCoordY = worldCoordY / (windowHeight / 2.f);
```

Ahora si tenemos las coordenadas correctas tanto en x como en y. Aunque aun no estamos 100% listos para poder representar bien nuestro cuadrado con su resolucion correcta, vamos con el siguiente punto.

Model View Projection Matrix

Este capitulo es intenso, ya que hay muchos conceptos matematicos. Soy ingeniero de software, no matematico, asi que es posible que algunos conceptos no los explique a la

perfeccion, pero creo que es mas importante que entendais el concepto general y ya despues indagais por vuestra cuenta para entenderlo al detalle, asi que vamos con ello.

Cuando desarrollamos juegos, y otros motores conocidos como Unity o Godot, tenemos siempre el concepto de Transform, propiedad que contiene o que nos permite modificar la posicion, rotacion y escala de un objeto en la escena, pero que es exactamente este Transform? Pues es una matrix 4x4 y a fin de cuentas es la matriz Model.

Model

La matriz Model es la que nos indica la posicion, escala y rotacion de nuestros objetos en el espacio. Cuando vamos a calcular esta matriz, empezamos siempre con una matriz identidad 4x4:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Pero que significa esta matriz exactamente? Bueno, para un objeto en el espacio, la matriz identidad significa que su escala es 1 en todos los ejes, que su posicion es el origen (0, 0, 0) y que no tiene rotacion. Los tres primeros elementos de la diagonal (empezando por la esquina superior izquierda) representan la escala en x, y, z y los tres primeros elementos de la ultima columna nos indican la posicion en x, y, z. La rotacion va un poco por otro lado ya que es bastante mas compleja, pero obviando esa parte, podemos escribir de forma generica que una matriz Model es:

$$M = \begin{bmatrix} s_x & 0 & 0 & t_x \\ 0 & s_y & 0 & t_y \\ 0 & 0 & s_z & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Con (s_x, s_y, s_z) la escala y pudiendo ser cualquier valor real y (t_x, t_y, t_z) lo mismo pero para la posicion.

Si queremos trasladar nuestro objeto a una nueva posicion, multiplicaremos $M * T$, siendo M actualmente la matriz identidad.

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Esto hara que nuestro objeto este en la nueva posicion (t_x, t_y, t_z).

Para escalar, multiplicamos $M * S$ siendo S la matriz de escalado:

$$M = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Lo que escalara nuestro objeto al vector (s_x, s_y, s_z).

Finalmente, para rotar hay que elegir primero sobre que eje rotar, ya que un objeto podemos rotarlo en 3 ejes diferentes (x, y, z) o en una combinacion de los mismos:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\alpha) & -sen(\alpha) & 0 \\ 0 & sen(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} cos(\alpha) & 0 & sen(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -sen(\alpha) & 0 & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} cos(\alpha) & -sen(\alpha) & 0 & 0 \\ sen(\alpha) & cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Sabiendo ya las posibles transformaciones que puede tener un objeto, definimos la matriz Model como:

$$M = I$$
$$M = T * R * S$$

Ahora me imagino que estareis algo asustados al pensar que hay que implementar todas estas matrices e implementar las operaciones etc... Bueno, el hecho es que si quereis, podeis

hacerlo, pero hay librerias muy optimizadas que se encargan ya de ello, como es en este caso GLM, la libreria matematica de OpenGL. Esta libreria ya sabe generar matrices, vectores y realizar las operaciones directamente entre ellas.

Tenemos vec2, vec3, vec4, mat2, mat3, mat4, entre otros tipos de daots, pero estos seran los que mas vamos a utilizar de todos. Lo bueno es que podemos operar directamente con ellos, multiplicandolos, sumandoloes... GLM lo hace por nosotros.

Y para trasladar tenemos glm::translate(mat4, vec3), para rotar glm::rotate(mat4, float, vec3) y para escalar glm::scale(mat4, vec3). Veremos mas adelante como usar cada una de ellas.

View

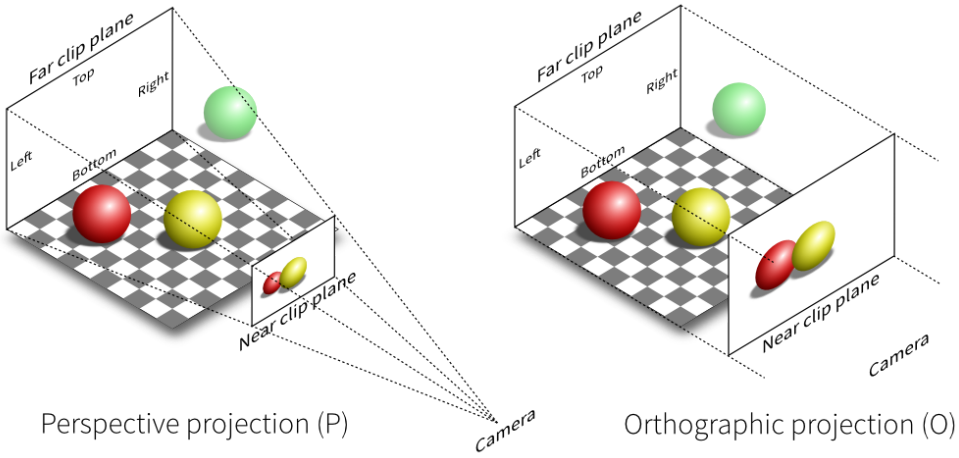
La matriz View a mi me gusta llamarla la "matriz de lo que la camara nos muestra", si, un nombre muy corto, pero que representa muy bien lo que hace. La matriz View representa donde esta la camara y que es lo que ve, pero al final no deja de ser como una matrix Model mas. Esta matriz no tiene ninguna formula chula como en el apartado anterior, es solo una matriz que representa el transform de la camara. En apartados posteriores veremos como hacer que la camara vea mas o menos cosas.

Projection

Ahora vamos a decidir como vamos a enfocar lo que se muestra en pantalla, y tenemos que decidir el tipo de vista que queremos, podemos elegir entre:

- Ortografica
- Perspectiva

Son dos tipos de vistas bastante diferentes, incluso a la hora de crearlas hay diferentes parametros. La principal diferencia es, que para mi, ortografica es mas para juegos 2D y la perspectiva es para juegos 3D, por que? Porque la proyeccion perspectiva nos permite hacer que los objetos mas lejanos se vean menores y los cercanos mayores, entre otras cosas. La siguiente imagen lo define muy bien:



Ortografica

En este tipo de camara hay que indicar lef, right, top, bottom, near y far. Left y right quiere decir la anchura que va a tener la camara [-aspectRatio, aspectRatio], top y bottom la altura que va a tener la camara [-1, 1] y near y far creo que lo mejor es entenderlo como un sandwich que esta de pie, far es la rebanada de atras y near la de delante [-1, 1] y solo se vera lo que esta dentro del sandwich. En 2D no debemos preocuparnos mucho por eso ya que todo esta en el mismo plano, es decir, no es un sandwich con varias capas de jamon, queso y lechuga, es solo el jamon.

Esta si que tiene una matriz de las chulas:

$$O = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

De nuevo, GLM viene al rescate y genera esta matriz por nosotrs mediante

glm::ortho(left, right, bottom, top, near, far)

Como nuestro modelo de motor es solo 2D y solo voy a usar la ortografica, no veremos la de perspectiva.

Camara

Sprite Batch

Texto

Frame Buffer

Texture Atlas Manager

Font Manager

Shader Manager

Glosario

1. Draw Call: Cada vez que se envian datos de la CPU a la GPU para renderizar algun elemento en pantalla.