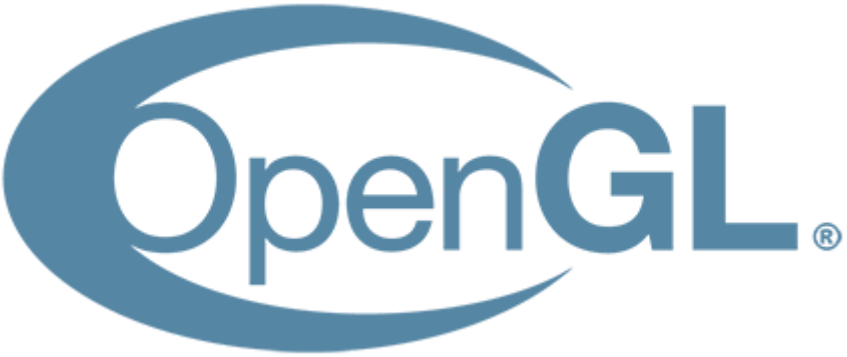


Gráficos por computador

Práctica de OpenGL: CarGL - Segunda revisión

Alejandro Moreno Rodríguez

Índice

Introducción

En esta revisión, exploramos la creación de una aplicación gráfica con OpenGL y GLSL, centrada en la visualización y manipulación de objetos tridimensionales. Destacando la inclusión de un vehículo interactivo y la posibilidad de diseñar modelos propios en Blender, la práctica busca ofrecer una experiencia envolvente y personalizada.

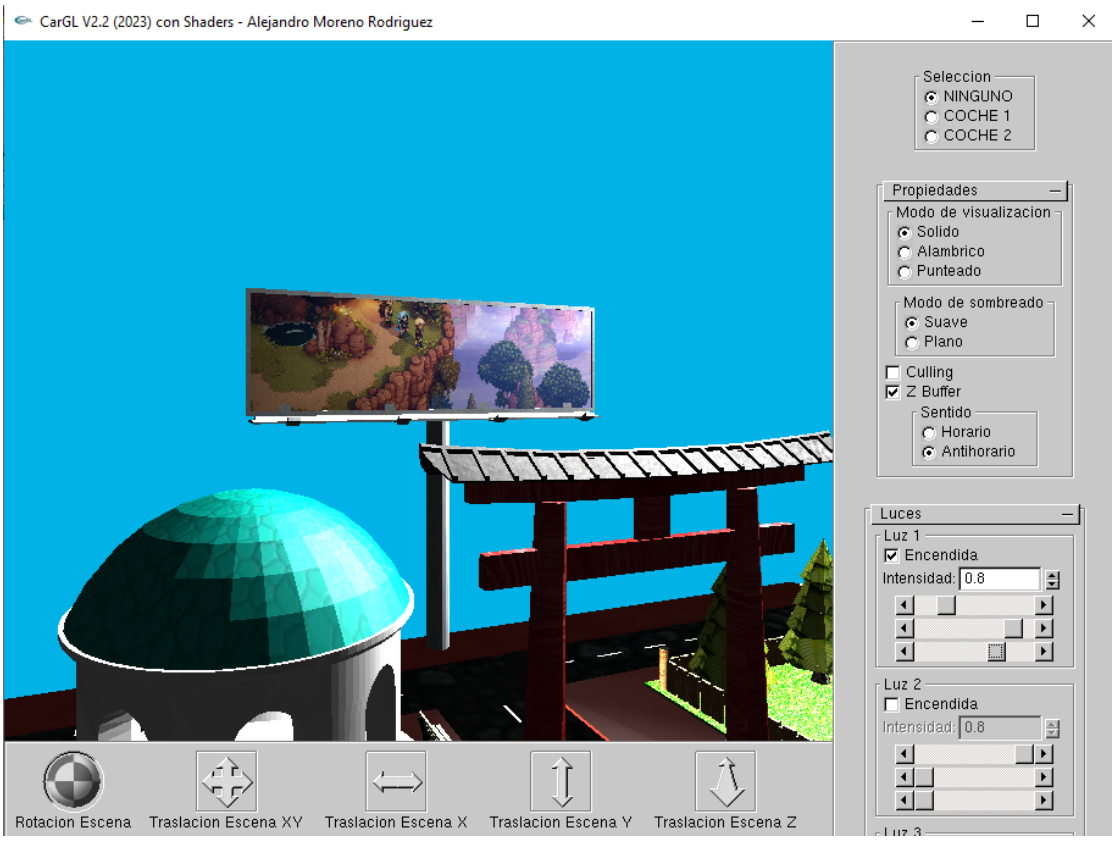
La implementación incluye un vehículo con ruedas que giran y se inclinan al avanzar, brindando una sensación realista de movimiento. Además, se han agregado cámaras adicionales para ofrecer perspectivas diversas, incluyendo una vista aérea y una que sigue dinámicamente al vehículo, además de en primera persona desde el coche.

La interacción se ha mejorado con la introducción de la selección de objetos 3D mediante clics del ratón, marcando visualmente los vehículos seleccionados. Menús interactivos permiten ajustar la proyección, la visualización de polígonos y otras configuraciones.

El control de la cámara se ha simplificado y enriquecido, permitiendo rotación, traslación y escala mediante el ratón y combinaciones de teclas. Además, los shaders han sido mejorados para admitir brillos especulares, luz ambiente y múltiples focos de luz con ajustes personalizables.

Finalmente, se ha integrado el uso básico de texturas para elevar la calidad visual de los modelos, completando así una aplicación versátil y envolvente en el ámbito de los gráficos por computadora.

Interfaz de usuario (GUI)



Esta es la interfaz gráfica de la aplicación. En el panel inferior están disponibles las opciones de movimiento por la escena, aunque se ha implementado un uso más sofisticado con el ratón con la propia escena, que se explicará más adelante.

En el panel derecho se han añadido desde `void __fastcall TGui::Init(int main_window)` las opciones presentes a la interfaz, que son:

Luces: gracias a diferentes opciones de la librería GLUI se han añadido spinners y paneles para poder tener la personalización de las luces: intensidad y posición

Modos de visualización: Pudiendo cambiar entre sólido (por defecto), alámbrico (wireframe) y punteado

```
GLUI_Panel *obj_panel_visualizacion = new
GLUI_Panel(obj_panel,"Modo de visualizacion");
```

```
GLUI_RadioGroup *radioGroupModos = new
GLUI_RadioGroup(obj_panel_visualizacion,&escena.modo_wireframe,CO
NTROL_WIREFRAME,controlCallback);
    glui->add_radiobutton_to_group(radioGroupModos,"Solido");
    glui->add_radiobutton_to_group(radioGroupModos,"Alambrico");
    glui->add_radiobutton_to_group(radioGroupModos,"Punteado");
```

Aquí también se han añadido las opciones de activado y desactivado del z-buffer y el culling, además del sentido horario y antihorario de éste.

```
new GLUI_Checkbox( obj_panel, "Culling", &escena.culling,
CULLING, controlCallback );
new GLUI_Checkbox( obj_panel, "Z Buffer", &escena.z_buffer,
ZBUFFER, controlCallback );

GLUI_Panel *obj_panel_sentido = new
GLUI_Panel(obj_panel,"Sentido");

GLUI_RadioGroup *radioGroupSentido = new
GLUI_RadioGroup(obj_panel_sentido,&escena.sentido_horario,HORARIO
,controlCallback);
glui->add_radiobutton_to_group(radioGroupSentido,"Horario");
glui->add_radiobutton_to_group(radioGroupSentido,"Antihorario");
```

Dibujado de objetos: pudiendo visualizar o no los coches y las ruedas, la carretera y los objetos.

```
GLUI_Rollout *options = new GLUI_Rollout(glui, "Opciones", false
);
new GLUI_Checkbox( options, "Dibujar Coche", &escena.show_car );
new GLUI_Checkbox( options, "Dibujar Ruedas", &escena.show_wheels
);
new GLUI_Checkbox( options, "Dibujar Carretera",
&escena.show_road );
new GLUI_Checkbox( options, "Dibujar Objetos",
&escena.show_objetos );
```

Cámara y proyección: Aquí podremos cambiar el tipo de cámara (habiendo 4 posibles) y el tipo de proyección (perspectiva o paralela)

```
GLUI_Panel *panelCamaras = new
GLUI_Panel(obj_panel_opcionescamara,"Camaras");
GLUI_RadioGroup *radioGroupCamaras = new
GLUI_RadioGroup(panelCamaras,&escena.camara,CONTROL_CAMARA,contro
lCallback);
```

```
glui->add_radiobutton_to_group(radioGroupCamaras,"Camara
Principal");
glui->add_radiobutton_to_group(radioGroupCamaras,"Camara
Seguimiento");
glui->add_radiobutton_to_group(radioGroupCamaras,"Camara Aerea");
glui->add_radiobutton_to_group(radioGroupCamaras,"Camara
Conductor");
/*** Radio Group para las proyecciones ***/
GLUI_Panel *panelProyeccion = new
GLUI_Panel(obj_panel_opcionescamara,"Proyeccion");
GLUI_RadioGroup *radioGroupProyeccion = new
GLUI_RadioGroup(panelProyeccion,&escena.proyeccion,CONTROL_PROYEC
CION,controlCallback);

glui->add_radiobutton_to_group(radioGroupProyeccion,"Perspectiva"
);

glui->add_radiobutton_to_group(radioGroupProyeccion,"Paralela");
GLUI_Panel *panelCamaras = new
GLUI_Panel(obj_panel_opcionescamara,"Camaras");
GLUI_RadioGroup *radioGroupCamaras = new
GLUI_RadioGroup(panelCamaras,&escena.camara,CONTROL_CAMARA,contro
lCallback);
glui->add_radiobutton_to_group(radioGroupCamaras,"Camara
Principal");
glui->add_radiobutton_to_group(radioGroupCamaras,"Camara
Seguimiento");
glui->add_radiobutton_to_group(radioGroupCamaras,"Camara Aerea");
glui->add_radiobutton_to_group(radioGroupCamaras,"Camara
Conductor");
/*** Radio Group para las proyecciones ***/
GLUI_Panel *panelProyeccion = new
GLUI_Panel(obj_panel_opcionescamara,"Proyeccion");
GLUI_RadioGroup *radioGroupProyeccion = new
GLUI_RadioGroup(panelProyeccion,&escena.proyeccion,CONTROL_PROYEC
CION,controlCallback);

glui->add_radiobutton_to_group(radioGroupProyeccion,"Perspectiva"
);
glui->add_radiobutton_to_group(radioGroupProyeccion,"Paralela");
```

Giro del vehículo

Para el giro del vehículo se ha modificado el método `void SpecialKey(int key, int x, int y)` en el archivo main.cpp para habilitar el movimiento a los lados y el retroceso o avance del mismo:

```
TPrimitiva *car = escena.GetCar(escena.seleccion);
TPrimitiva *marcador = escena.GetMarcador();
float grados = car->ry*3.14159265358979/180;

        switch (key)
        {
            case GLUT_KEY_UP:   // El coche avanza
                car->rr+=8;
                car->tx += 0.8*sin(grados);
                car->tz += 0.8*cos(grados);
                car->ry += 0.1*car->giro;
                marcador->rr+=8;
                marcador->tx += 0.8*sin(grados);
                marcador->tz += 0.8*cos(grados);
                marcador->ry += 0.1*car->giro;
                break;
            case GLUT_KEY_DOWN:   // El coche retrocede
                car->rr-=8;
                car->tx -= 0.8*sin(grados);
                car->tz -= 0.8*cos(grados);
                car->ry -= 0.1*car->giro;
                marcador->rr-=8;
                marcador->tx -= 0.8*sin(grados);
                marcador->tz -= 0.8*cos(grados);
                marcador->ry -= 0.1*car->giro;
                break;
            case GLUT_KEY_LEFT:  { //El coche gira a la izq
                if(car->giro < 30) car->giro += 2;
                break;
            }
            case GLUT_KEY_RIGHT: { //El coche gira a la dere
                if(car->giro > -30) car->giro -= 2;
                break;
            }
        }
```

Como se puede ver, además de habilitar el movimiento del coche, las coordenadas del marcador también se modifican aquí para que en todo momento sigan al vehículo.

Posteriormente, para la rotación del coche y las ruedas a la hora de renderizar el mismo se ha añadido la siguiente línea:

```
modelMatrix = glm::rotate(modelMatrix, (float) glm::radians(ry),
glm::vec3(0,1,0)); // en radianes
```

Para las ruedas, además de su rotación ha habido que modificar su posición para que estén situadas justo en los huecos del modelo del coche. Para cada rueda ha sido necesario cambiar las variables tx y ty, de forma que para cada una hay valores distintos. Por ejemplo, la rueda izquierda delantera:

```
modelMatrix = glm::translate(modelMatrix, glm::vec3(tx+1.05,
ty+0.45, tz));
modelMatrix = glm::translate(modelMatrix, glm::vec3(-1.05, 0,
-0.03));
modelMatrix = glm::rotate(modelMatrix, (float) glm::radians(ry),
glm::vec3(0,1,0));
modelMatrix = glm::translate(modelMatrix, glm::vec3(+1.05, 0,
+0.03));
modelMatrix = glm::rotate(modelMatrix, (float)
glm::radians(giro), glm::vec3(0,1,0));
modelMatrix = glm::rotate(modelMatrix, (float) glm::radians(rr),
glm::vec3(1,0,0));
modelMatrix = glm::rotate(modelMatrix, (float)
glm::radians(180.0), glm::vec3(0,0,1));
```

La rueda delantera derecha llevará un código similar, solo que con los valores en el **`glm::translate`** modificados. Las ruedas traseras no tienen dos líneas de código que las delanteras sí: esto es porque las traseras no van a girar en el movimiento de izquierda o derecha, tal y como se mueven los coches de tracción delantera en la realidad.

Luces

Para las luces, se han seguido los siguientes pasos. En primer lugar, desde Objects.h se ha asignado un ID distinto para el habilitado, posición e intensidad de las 3 luces:

```
// IDs para los callbacks de TGui
#define LIGHT0_ENABLED_ID    200
#define LIGHT1_ENABLED_ID    201
#define LIGHT2_ENABLED_ID    202

#define LIGHT0_POSITION_ID   210
#define LIGHT1_POSITION_ID   211
#define LIGHT2_POSITION_ID   212

#define LIGHT0_INTENSITY_ID  220
#define LIGHT1_INTENSITY_ID  221
#define LIGHT2_INTENSITY_ID  222
```

Continuando en el mismo archivo, en la clase TEscena se han sumado las siguientes líneas de código:

```
int u_Position_Luz0Location;
int u_Position_Luz1Location;
int u_Position_Luz2Location;

int u_Intensidad0Location;
int u_Intensidad1Location;
int u_Intensidad2Location;

int uLuz0Location;      // Activada la luz 0
int uLuz1Location;      // Activada la luz 1
int uLuz2Location;      // Activada la luz 2

GLfloat light2_ambient[4];
GLfloat light2_diffuse[4];
GLfloat light2_specular[4];
GLfloat light2_position[4];
```

Y en la clase TGui los diferentes objetos GLUI necesarios para hacer funcionar desde la interfaz de usuario las luces.

En el VertexShader.glsl también se han realizado cambios. En primer lugar, añadir los uniforms necesarios:

```
uniform int u_Luz0; // Indica si la luz 0 está encedida
uniform int u_Luz1; // Indica si la luz 1 está encedida
uniform int u_Luz2; // Indica si la luz 2 está encedida

//Posicionamiento
uniform vec4 u_Position_Luz0;
uniform vec4 u_Position_Luz1;
uniform vec4 u_Position_Luz2;

//Intensidad
uniform float u_Intensidad0;
uniform float u_Intensidad1;
uniform float u_Intensidad2;
```

El método main del vertex shader contiene el código para el cálculo de la distancia y posición de la luz, así como calcular la posición y normal del vértice.

```
vec3 P = vec3(u_MVMatrix * a_Position);
vec3 N = vec3(u_MVMatrix * vec4(a_Normal, 0.0));
vec3 viewVec = normalize(vec3(-P));
```

Posteriormente se declaran las variables para indicar el grado de luz ambiente, difusa y especularidad.

```
float ambient = 0.15;
float diffuse0 = 0.0;
float specular0 = 0.0;
```

Y para cada luz, dentro del condicional, si se cumple que se han activado tenemos el cálculo de la intensidad:

```
diffuse2 = max(dot(N, L2), 0.0); // Cálculo de la int. difusa
// Calculo de la atenuacion
float attenuation2 = 80.0/(0.25+(0.01*d2)+(0.003*d2*d2));
diffuse2 = diffuse2*attenuation2*u_Intensidad2;

//Calculo de la especular.
vec3 lightVec2 = L2;
vec3 reflectVec2 = reflect(-lightVec2, N);
specular2 = clamp(dot(reflectVec2, viewVec),0.0, 1.0);
specular2 = pow(specular2, 5.0);
```

```
specular2 = specular2*u_Intensidad2;
```

Para terminar, calculamos el color final del píxel

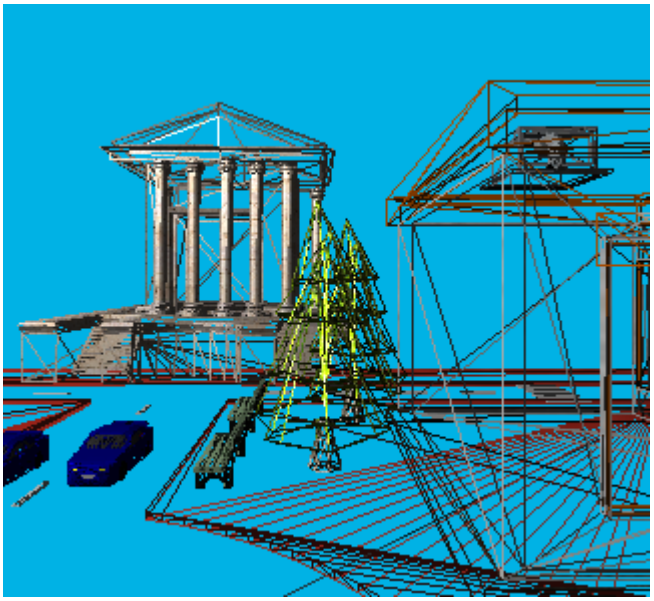
```
// Calculo final de la luz
float sumDiffuse = diffuse0 + diffuse1 + diffuse2;
v_Color = u_Color * (v_Luz + sumDiffuse);
```

Modos de visualización y opciones (Alámbrico, punteado y sólido)

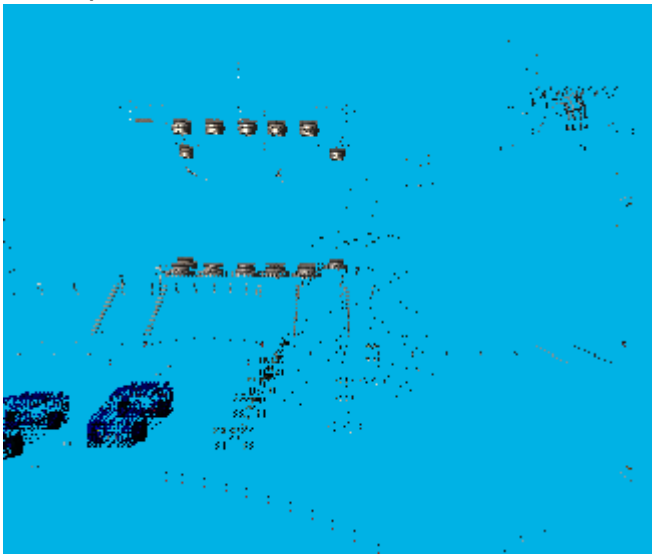
Dentro del método Render de TEscena se ha implementado el código necesario para que funcione el modo de vista wireframe, punteado y default, además de la opción de renderizar la escena con o sin zbuffer, y en caso de activar el culling, establecer el sentido como horario o antihorario:

```
switch (escena.modo_wireframe){
	case 0:{
		printf("Modo wireframe: Solido \n");
		glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
		break;
	}
	case 1:{
		printf("Modo wireframe: Alambrico \n");
		glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
		break;
	}
	case 2:{
		printf("Modo wireframe: Punteado \n");
		glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
		break;
	}
}
```

Modo wireframe



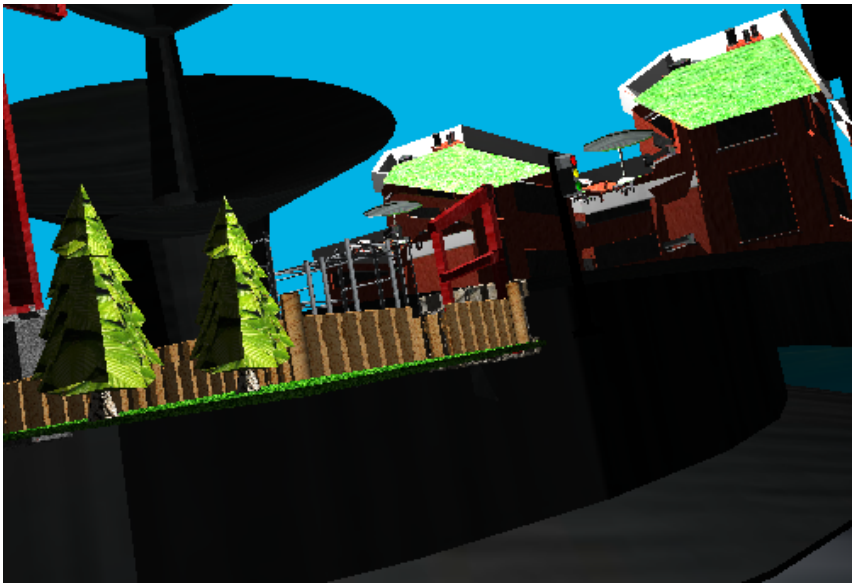
Modo punteado



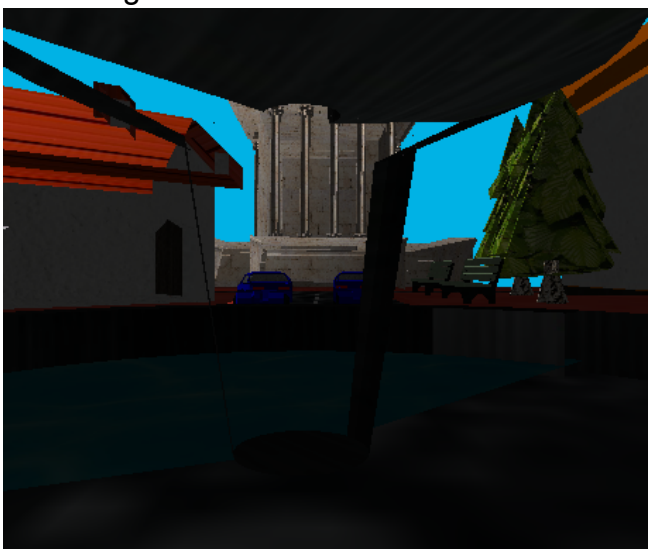
Para el culling y el ZBuffer:

```
case CULLING: {
	if(escena.culling==1) {
		glEnable(GL_CULL_FACE);
		glCullFace(GL_FRONT);
	}
	else {
		glDisable(GL_CULL_FACE);
	}
	break;
}
case ZBUFFER: {
	if(escena.z_buffer==1) {
		glEnable(GL_DEPTH_TEST);
	}else{
		glDisable(GL_DEPTH_TEST);
	}
	break;
}
```

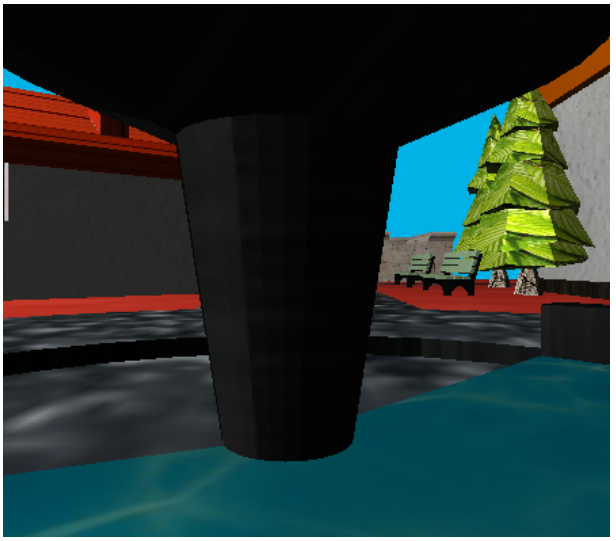
ZBuffer desactivado:



Culling horario



Culling antihorario



Cámaras y proyección

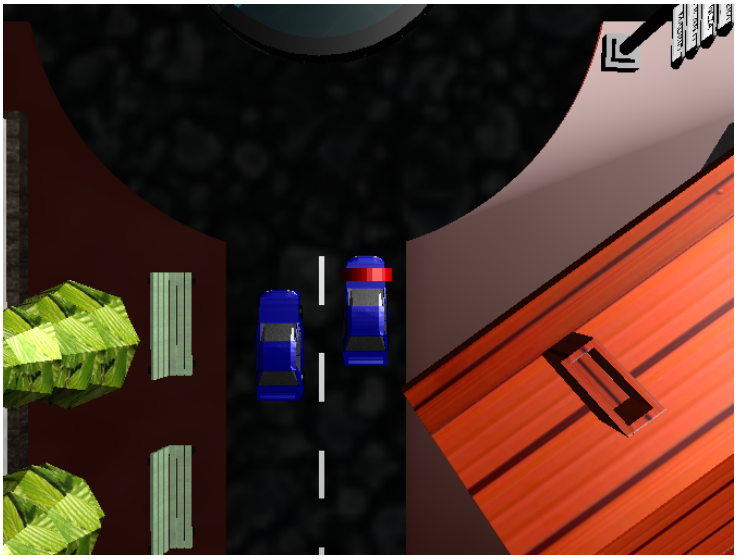
Además de la cámara por defecto, hay disponibles 3 tipos de cámaras nuevas. Estas son la aérea, la de seguimiento y la cámara en primera persona.

El código necesario para las cámaras se encuentra en la función Render de TEscena, y es el siguiente:

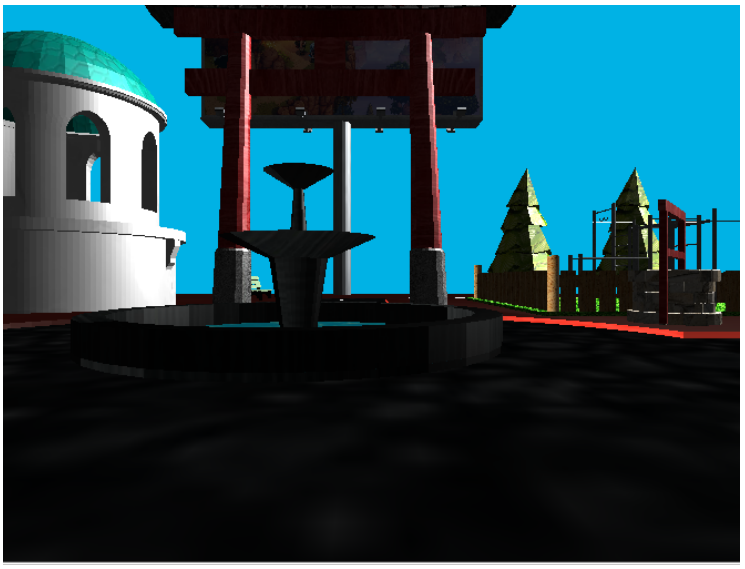
```
// Cálculo de la vista (cámara)
switch(escena.modo_camara){
    //Camara general
    case 0:{
        viewMatrix = glm::mat4(1.0f);
        rotateMatrix = glm::make_mat4(view_rotate);
        viewMatrix =
glm::translate(viewMatrix,glm::vec3(view_position[0],
view_position[1], view_position[2]));
        viewMatrix = viewMatrix*rotateMatrix;
        viewMatrix = glm::scale(viewMatrix,glm::vec3(scale/100.0,
scale/100.0, scale/100.0));
        break;
    }
    case 1:{
        //Camara de seguimiento
        TPrimitiva *coche = GetCar(seleccion);
        if(coche){
            float grados = coche->ry*3.14159265358979/180.0;
            viewMatrix =
glm::lookAt(glm::vec3(coche->tx-10.0*sin(grados),5,coche->tz-10.0*cos(
grados)), glm::vec3(coche->tx,coche->ty,coche->tz),glm::vec3(0,1,0));
        }
        break;
    }
    case 2: {
        //Camara aerea
        TPrimitiva *coche = GetCar(seleccion);
        if(coche) viewMatrix =
glm::lookAt(glm::vec3(coche->tx,30,coche->tz),glm::vec3(coche->tx,0,co
che->tz),glm::vec3(0,0,1));
        break;
    }
    case 3: {
        // Camara conductor
        TPrimitiva *coche = GetCar(seleccion);
        if(coche){
            float grados = coche->ry*3.14159265358979/180.0;
```

```
            viewMatrix =
glm::lookAt(glm::vec3(coche->tx,2.5,coche->tz),glm::vec3((coche->tx)+2
*sin(grados),2.5,(coche->tz)+2*cos(grados)),glm::vec3(0,1,0));
        }
        break;
    }
}
```

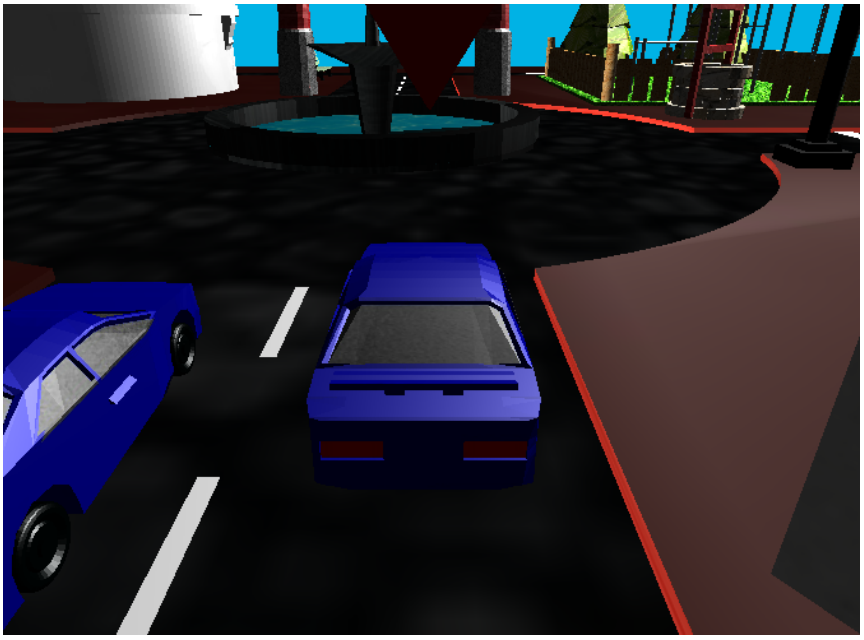
Cámara aérea:



Cámara en primera persona:



Cámara de seguimiento:



Para la proyección paralela:

```
//Proyeccion
int tx, ty, tw, th;
GLUI_Master.get_viewport_area( &tx, &ty, &tw, &th );
glViewport( tx, ty, tw, th );
escena.xy_aspect = (float)tw / (float)th;
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

if(proyeccion==0){
    projectionMatrix = glm::perspective(45.0f, xy_aspect, 0.1f,
1000.0f);
    glUniformMatrix4fv(uProjectionMatrixLocation, 1, GL_FALSE,
glm::value_ptr(projectionMatrix));
}
else if(proyeccion==1){
    projectionMatrix = glm::ortho(-50.0f,(float)tw/10, (float)-th/10,
50.0f, -1000.0f, 1000.0f);
    glUniformMatrix4fv(uProjectionMatrixLocation, 1, GL_FALSE,
glm::value_ptr(projectionMatrix));
}
```

Proyección paralela:

Selección del vehículo con el ratón (Pick 3D)

El primer paso ha sido declarar en el vertex shader el uniform para habilitar la selección:

```
uniform int u_selection_enabled;
```

En el Objects.h también se ha definido la selección:

```
//Seleccion.
#define U_SELECTION_ENABLED     "u_selection_enabled"
```

Ahora para la correcta selección con el ratón se ha habilitado el stencil buffer con esta línea en el método InitGL de TEscena:

```
glEnable(GL_STENCIL_TEST);
```

Luego, para todos los objetos, se han incluido estas dos líneas:

```
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
glStencilFunc(GL_ALWAYS, 0, 0xFF);
```

Esto conseguirá pintar todos los objetos del mismo color que el fondo dentro del stencil buffer (no se muestra en pantalla), lo que hace que no se puedan seleccionar esos objetos.
Para que se puedan seleccionar los coches, tendremos que sustituir la segunda línea del paso anterior por la siguiente:

```
glStencilFunc(GL_ALWAYS, ID, 0xFF);
```

Esto hará que el coche se pinte (en el stencil buffer) del color del id del coche. Ya que el esténcil buffer se pinta en una escala de grises esto quiere decir que podemos seleccionar hasta 254 coches diferentes.
Ahora modificaremos la función pick3D para que quede de la siguiente forma:

```
void __fastcall TEscena::Pick3D(int mouse_x, int mouse_y)
{
    GLint   viewport[4];
//buffer[0] = numero de objetos
    GLuint  buffer[2048];
//buffer[1] = profundidad minima
    GLint   hits;
//buffer[2] = profuncidad maxima
    int     profundidad, tx, ty, tw, th;
//buffer[3] = nombre de la pila
    char    cad[80];
```

```
    //seleccion = 0;
    GLUI_Master.get_viewport_area( &tx, &ty, &tw, &th );
    glViewport( tx, ty, tw, th );

    int index;
    glReadPixels(mouse_x, th-mouse_y+81, 1, 1, GL_STENCIL_INDEX,
GL_UNSIGNED_INT, &index);

    gui.sel=index;
    seleccion= index;

    switch(seleccion){
        case 1: { // Coche 1
            if(escena.show_marcador == 0){
                escena.show_marcador = 1;
            }
            gui.sel=1;
            seleccion=1;
            gui.radioGroup->set_selected(seleccion);
            TPrimitiva *marcador = escena.GetMarcador();
            TPrimitiva *coche = escena.GetCar(seleccion);
            marcador->ry = coche->ry;
            marcador->tx = coche->tx;
            marcador->tz = coche->tz;
            marcador->ty = coche->ty;
            break;
        }
        case 2: {
            if(escena.show_marcador == 0){
                escena.show_marcador = 1;
            }
            gui.sel=2;
            seleccion=2;
            gui.radioGroup->set_selected(seleccion);
            TPrimitiva *marcador = escena.GetMarcador();
            TPrimitiva *coche = escena.GetCar(seleccion);
            marcador->ry = coche->ry;
            marcador->tx = coche->tx;
            marcador->tz = coche->tz;
            marcador->ty = coche->ty;
            break;
        }
```

```
            default: {
                //printf("No has seleccionado ningun coche \n");
                escena.show_marcador = 0;
                gui.radioGroup->set_selected(0);
                break;
            }
        }
    }
```

Con el switch de la selección controlaremos qué sucede si pulsamos en un coche u otro. La lógica es que se renderice el marcador justo encima del vehículo seleccionado.

Implementación de texturas

Una vez hemos creado las texturas en Blender gracias a la opción 'Bake' y un correcto mapeado UV, para implementarlas en nuestra práctica se han seguido varios pasos. Para empezar, en el Objects.h se han declarado los siguientes elementos:

```
//Texturas
#define A_TEXTURECORD "a_TextureCoord"
#define U_SAMPLER "u_Sampler"
```

Y en la clase TEscena:

```
public:
    // Texturas
    unsigned char *texturas[100];
    int texturas_width[100];
    int texturas_height[100];
    GLuint texturas_id[100];

public: // Atributos de la clase
//...
int aTextureCoordLocation;
int u_SamplerLocation;
int activadaLocation;
```

Se han creado dos métodos para la carga de texturas en main.cpp. Es importante destacar que ha sido necesario incluir el archivo *loadjpeg.c* en el proyecto, además de su include. Los métodos para la carga de texturas mencionados:

```
void cargarTextura(int id){
    int width, height;
    unsigned char *img;

    width = escena.texturas_width[id];
    height = escena.texturas_height[id];
    img = escena.texturas[id];

    glBindTexture(GL_TEXTURE_2D, escena.texturas_id[id]);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
```

```
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glTexImage2D(GL_TEXTURE_2D, 0, 3, width, height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, img);
}

void cargarTexturas(){

      escena.texturas[0] = LoadJPEG("../../Texturas/Acera.jpg",
                                    &escena.texturas_width[0],
                                    &escena.texturas_height[0]);
      // Para cada textura jpg. . .
      glGenTextures(100, escena.texturas_id);

      std::cout  << "Cargando texturas: " << std::endl;

      for(int i = 0; i < 100; i++){
          cargarTextura(i);
      }
      std::cout << std::endl;
}
```

Después, en el método main se llama al método cargarTexturas para cargarlas. Por último, en el Objects.cpp, para cada objeto en su renderizado se incluyen estas líneas, acordes a cada textura que hemos metido en el array de texturas (es decir, si por ejemplo, escena.texturas[8] corresponde al edificio:)

```
glBindTexture(GL_TEXTURE_2D, escena.texturas_id[8]);
glUniform1i(escena.activadaLocation, 1);
glUniform1i(escena.u_SamplerLocation, 0);
glActiveTexture(GL_TEXTURE0);
```

Cabe destacar que es necesario habilitar el paso de atributos desde el método InitGL:

```
// Habilitamos el paso de attributes
glEnableVertexAttribArray(aPositionLocation);
glEnableVertexAttribArray(aNormalLocation);

// Habilitamos las texturas
glEnableVertexAttribArray(aTextureCoordLocation);
```

Por último, se ha modificado el fragment shader para que todo funcione correctamente.

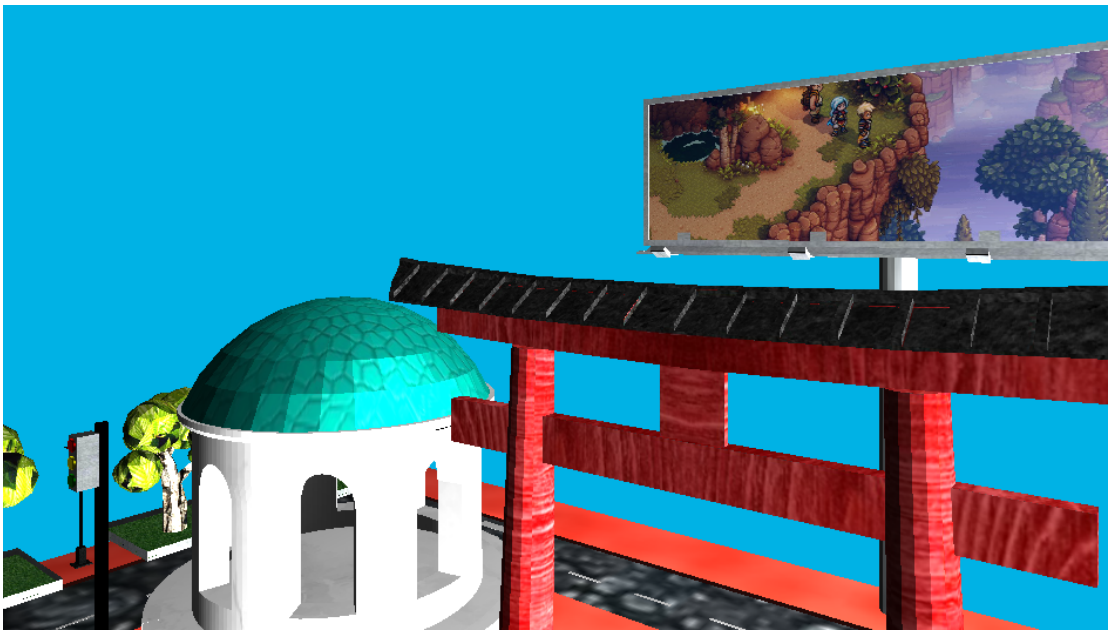
```
precision mediump float;  // Precisión media, en algunas gráficas no se soporta (depende de la versión de GLSL), en ese caso comentar o quitar esta línea

varying vec4 v_Color;

// Texturas

varying vec2 v_TextureCoord;
varying vec4 v_Luz;
uniform sampler2D u_Sampler;
uniform int activada;

void main() {
    if (activada == 0) {
        gl_FragColor = v_Color;
    } else {
            gl_FragColor = texture2D(u_Sampler, v_TextureCoord) * v_Luz;
    }
}
```



Uso del mouse en la escena

Se ha añadido mediante los métodos Mouse y Motion un uso sofisticado de navegación por la escena. De forma que, manteniendo distintos botones y usando el ratón tendremos varias opciones de navegación, que son las siguientes:

- **Rotación**: Movimiento del ratón y botón izquierdo sobre la escena. Si mantenemos pulsado el botón CTRL podremos rotar en el eje Z.
- **Traslación**: Movimiento del ratón, botón izquierdo de la escena + botón SHIFT
- **Escalado**: Movimiento del ratón y el botón derecho del mouse.
- **Zoom**: Movimiento del ratón, botón izquierdo del ratón más los botones SHIFT y CTRL a la vez.

El código para el correcto funcionamiento es éste:

```
void __fastcall TGui::Mouse(int button, int button_state, int x, int
y)
{
    if(button_state==GLUT_DOWN && escena.camara_Movil == 0)
    {
        //glutGetModifiers() devuelve GLUT_ACTIVE_CTRL,
GLUT_ACTIVE_SHIFT, y GLUT_ACTIVE_ALT
        int x = glutGetModifiers();

        //FLECHA IZQUIERDA.
        if(button == GLUT_LEFT_BUTTON && glutGetModifiers() == 3)
             escena.raton = 2; //MOVIMIENTO EN Z
                //Izquierdo y shift y control
        else if(button==GLUT_LEFT_BUTTON &&
glutGetModifiers()!=GLUT_ACTIVE_CTRL &&
glutGetModifiers()!=GLUT_ACTIVE_SHIFT)
            escena.raton = 3; //ROTAR EN X Y
            //Solo boton izquierdo
        else if(button==GLUT_RIGHT_BUTTON &&
glutGetModifiers()!=GLUT_ACTIVE_CTRL &&
glutGetModifiers()!=GLUT_ACTIVE_SHIFT)
            escena.raton = 5; //Escalado
            //Solo boton derecho
        else if(button == GLUT_LEFT_BUTTON &&
glutGetModifiers()==GLUT_ACTIVE_CTRL)
            escena.raton = 4; //ROTAR EN Z
            //Boton izquierdo y control
        else if(button == GLUT_RIGHT_BUTTON &&
glutGetModifiers()==GLUT_ACTIVE_CTRL)//No hacer ahora
            escena.raton = 6; //Escalado en Z
            //Boton Derecho y control
```

```
        else if(button == GLUT_LEFT_BUTTON &&
glutGetModifiers()==GLUT_ACTIVE_SHIFT)
            escena.raton = 1; //MOVERSE CON RATON
            //Izquierdo y shift
        escena.last_x=x;
        escena.last_y=y;
    }
    else if(button_state==GLUT_UP) //Liberar tecla
    {
        escena.raton=0;
        if(button==GLUT_LEFT_BUTTON && escena.camara_Movil == 0)
            escena.Pick3D(x, y);
    }
}
```

Controlando en qué situación estamos, desde el método Motion nos moveremos por la escena de una u otra forma en función de lo que queramos.

Rotación:

```
float rotacion[16];
float rotacion_X_dec = x - escena.last_x;
float rotacion_Y_dec = y - escena.last_y;

escena.rotacion_X += rotacion_X_dec;
escena.rotacion_Y += rotacion_Y_dec;

glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();

glRotated(escena.rotacion_X  *  escena.factor_movimiento_camara,  0,  1,
0);
glRotated(escena.rotacion_Y  *  escena.factor_movimiento_camara,  1,  0,
0);

glGetFloatv(GL_MODELVIEW_MATRIX, rotacion);
for(int i=0; i<16; i++){
      if(escena.view_rotate[i] != rotacion[i]){
            escena.view_rotate[i] = rotacion[i];
      }
}
glPopMatrix();
```

Rotación en el eje Z:

```
float rotacion[16];
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();

float rotacion_Z_dec = (y - escena.last_y) - (x - escena.last_x);
escena.rotacion_Z += rotacion_Z_dec;

glRotated(escena.rotacion_Z * escena.factor_movimiento_camara, 0, 1,
0);

glGetFloatv(GL_MODELVIEW_MATRIX, rotacion);

for(int i=0; i<16; i++){
	if(escena.view_rotate[i] != rotacion[i]){
		escena.view_rotate[i] = rotacion[i];
	}
}
```

Traslación:

```
if(x > escena.last_x){          // Nos movemos a la derecha
	escena.view_position[0] += escena.factor_movimiento_camara;
}else if(x < escena.last_x){    // Nos movemos a la izquierda
	escena.view_position[0] -= escena.factor_movimiento_camara;
}

if(y > escena.last_y){          // Nos movemos arriba
	escena.view_position[1] -= escena.factor_movimiento_camara;
}else if(y < escena.last_y){    // Nos movemos abajo
	escena.view_position[1] += escena.factor_movimiento_camara;
}
```

Escalado:

```
if(y < escena.last_y){
	escena.scale += escena.factor_movimiento_camara * 0.55;
	escena.scaleZ += 1.01;
}else if(y > escena.last_y){
	escena.scale -= escena.factor_movimiento_camara * 1.05;
	escena.scaleZ -= 1.01;
```

```
}
```

Zoom:

```
if(y < escena.last_y){
      escena.view_position[2] +=
escena.factor_movimiento_camara*2;
}
else if(y > escena.last_y){
      escena.view_position[2] -=
escena.factor_movimiento_camara*2;
}
```