Computación Gráfica - TP: F1

1. Resumen de tareas

1. Ensamblar correctamente el auto a partir de transformar cada pieza (en la función `renderCar`).
2. Posicionar y orientar el auto como corresponda en la pista.
3. Modificar las cámaras (en la función `setViewAndProjectionMatrixes`):
    1. En el caso de la vista superior la cámara debe girar con el auto de modo que el auto siempre apunte hacia arriba.
    2. En la vista de persecución debe perseguir al auto siempre desde atrás.

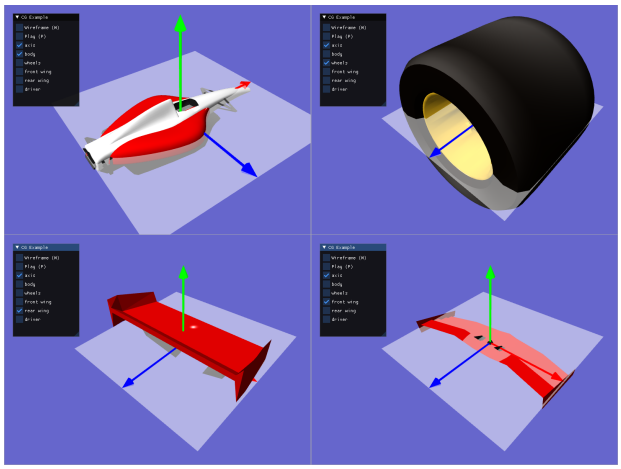
2. Consigna detallada

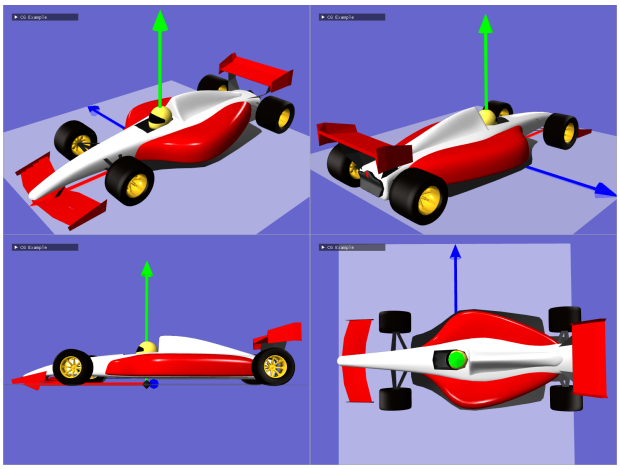
En este práctico deberá saber construir las matrices adecuadas para primero ensamblar el modelo del auto a partir de sus piezas, luego posicionarlo sobre la pista para que se mueva como corresponde, y finalmente acomodar las cámaras para que lo siguan y giren con él.

2.1. Ensamblando el auto

La función `main` carga un conjunto de modelos en el vector `parts`. Los modelos son (listados en el orden en que aparecen en el arreglo):

0. los ejes de referencia,
1. el chasis del auto (*body*)
2. una rueda (*wheel*)
3. el alerón delantero (*front wing*)
4. el alerón trasero (*rear wing*)
5. el casco del piloto (*driver*)





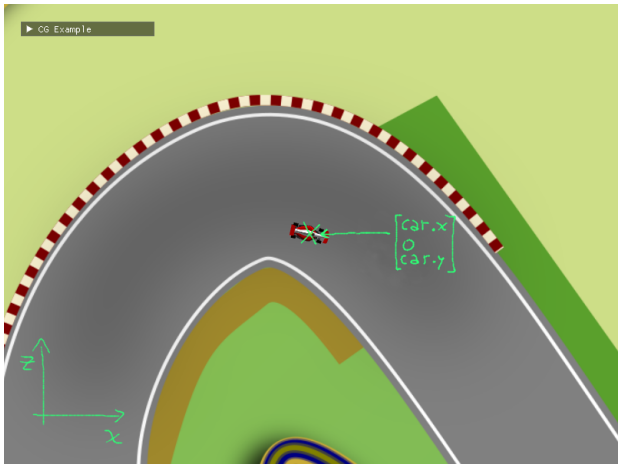
Los modelos incialmente caben en un cubo que va desde -1 a $+1$ en todos los ejes. Puede utilizar los ejes del model 0 como referencia visual. Cada vector mide una unidad, parten del origen y los colores rojo, verde y azul corresponden a los ejes x, y, y z respectivamente.

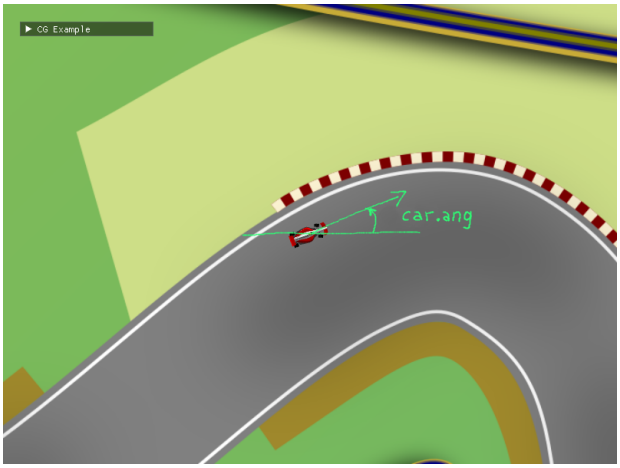
El alumno deberá acomodar todas las partes (a excepción de los ejes de referencia) para ensamblar el auto. El chasis debe mantener su tamaño (aunque puede variar su posición para despegarlo del suelo[1]), mientras que las demás piezas deben escalarse, rotarse y trasladarse para acomodarse en torno al chasis.

La función `drawPart` recibe una parte y una matriz de transformación. **Es la función `drawCar` la que invoca a `drawPart` pasándole cada parte y cada matriz [2].** En el código inicial, cada parte va acompañada de la matriz identidad. **El alumno deberá reemplazar esas matrices, generando las correctas[3] para cada pieza**. Estas funciones de encuentran definidas en el archivo *Render.cpp*.

2.2. Aplicando el movimiento

El código inicial ya simula cómo debería moverse el auto cuando el usuario utiliza el teclado para conducirlo. De esto se encargan la función `getInput`(de sensar el teclado, o el joystick si está presente) y la función `Car::Move` (de calcular el movimiento). Como resultado, una instancia `car` del struct `Car` guarda en cada frame la posición (`car.x` y `car.y`) y orientación del mismo en la pista (`car.ang`). Deberá utilizar estas variables para **agregar al programa las transformaciones necesarias para ubicar todo el auto ya ensamblado en la pista**.





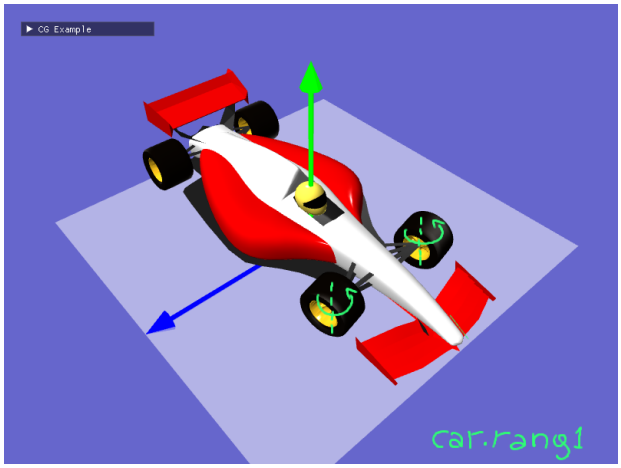
Inicie el juego (opción *play*, o la tecla *P*) y utilice la vista superior para verificar si el auto se mueve correctamente. Dado que la cámara de vista superior ya tiene implementado el seguimiento del auto, si se mueve correctamente, siempre debería verlo en el centro de la ventana.

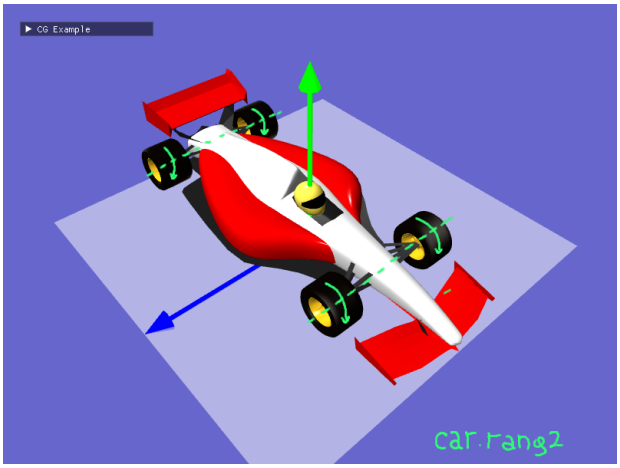
Notar que si bien las variables de posición del auto se denominan `x` e `y` (porque la física del auto es 2D, para la clase `Car` no existe z), en el mundo 3D estas coordenadas deben asociarse a los ejes x y z, ya que la pista está en el plano $y = 0$.

[1] La pista sobre la cual se moverá coincide con el plano $y = 0$

[2] Hay un solo modelo para las ruedas, que modela una de ellas, y deberá reutilizarse invocando 4 veces a `drawPart` con ese mismo modelo pero variando la matriz

[3] No se proveen los tamaños y las ubicaciones exactas de cada pieza, sino que se deja a criterio del alumno. Se espera que el auto se ”vea” razonablemente bien ensamblado.





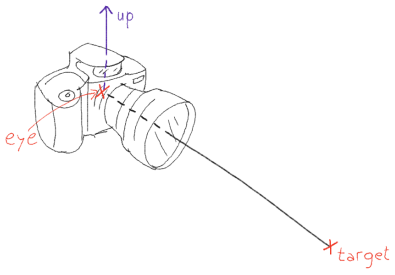
Pero además también guarda dos ángulos adicionales para el movimiento de las ruedas: `car.rang1` guarda una rotación extra a aplicar a las 2 ruedas delanteras para reaccionar al giro del ”volante”, y `car.rang2` otra rotación adicional para reflejar el girar de las 4 ruedas a medida que el auto avanza. Todos **los ángulos están almacenados en radianes. Deberá modificar las matrices de transformaciones de las ruedas en la función `drawCar` para reflejar estos dos giros adicionales.

2.3. Ajustando las cámaras

Este *juego* permite alternar entre 2 cámaras o vistas: una superior (o también llamada aérea) y una trasera (o también llamada en muchos juegos *chase-cam*, de persecución).

Es habitual utilizar una función denominada `lookAt` (antes parte de *OpenGL*, ahora provista por *glm*) para generar la matriz adecuda para ”posicionar” una cámara. Esta función recibe tres argumentos:

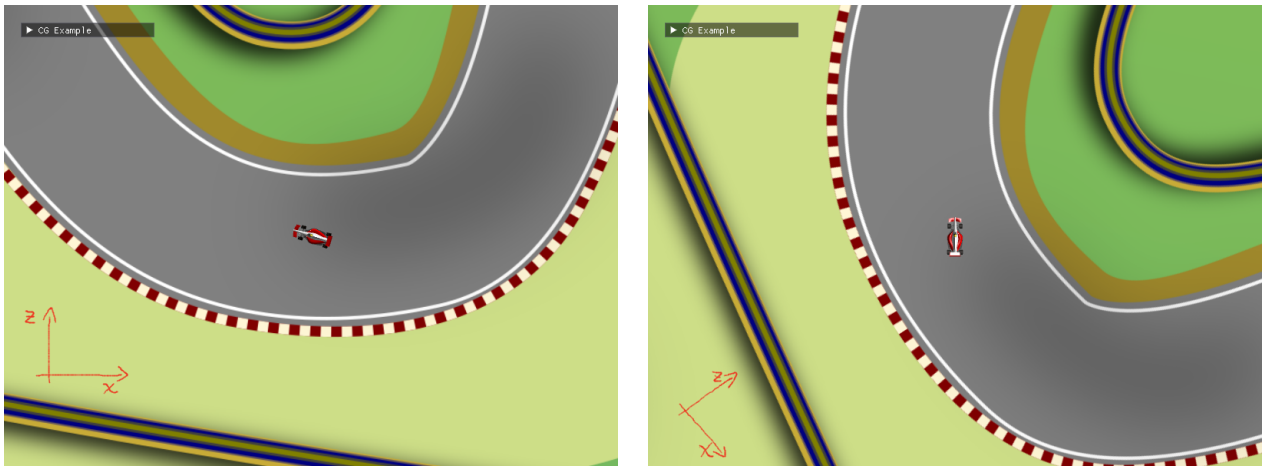
- *eye*: la posición de la cámara/ojo
- *target*: un punto al que la camara apunta (entre eye y target definen un vector con la dirección hacia la que se ”mira”)
- *up*: un vector que indica para dónde es ”arriba” para la cámara (dada una dirección a la cual mirar, la cámara todavía puede girar usando el vector *target-up* como eje, por lo que es necesario este tercer argumento para terminar de definirla).



La definición de la matriz de la vista se implementa en la función `setViewAndProjectionMatrixes` del archivo *Render.cpp*. El alumno deberá modificar esta implementación de acuerdo a los siguientes critetrios:

2.3.1. Cámara de vista superior

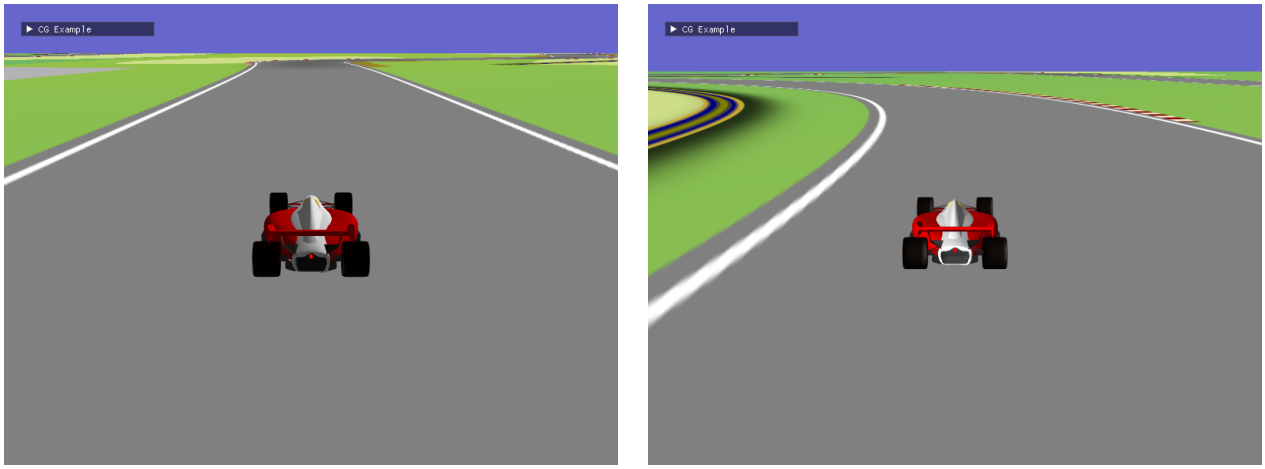
La cámara de vista superior está implementada en el código inicial del práctico, pero no gira solidariamente con el auto. La tarea aquí es corregir la definición de esta cámara para que lo haga.



La figura de la izquierda corresponde a la implementación inicial, la de la derecha a la corrección que debe hacer el alumno.

2.3.2. Cámara trasera

En el caso de la cámara trasera, el código inicial no tiene ninguna implementación previa por lo que el alumno deberá defirnirla completamente. El objetivo es que esta cámara ”persiga” al auto, y siempre le apunte desde ”atrás” del mismo.



2.4. Matrices en la biblioteca `glm`

En este código las matrices 4D se representan mediante la clase `glm::mat4` [4]. La biblioteca provee de funciones y sobrecarga para generar y manipular estas matrices:

[4] Las clases como `glm::mat4`, `glm::vec3`, `glm::vec4`, etc son en realidad especializaciones de un *template* para el tipo de dato `float`. Las funciones libres de la biblioteca también son *templates*. Por esto, en muchos casos es necesario agregrar el `.f` a las constantes (ej: `0.5f` en

Se puede construir la matriz identidad con `glm::mat4(1.f)`.

- Se puede construir una a partir de los 16 coeficientes pasándo los mismos como argumentos al constructor:

```
glm::mat4 m( 0.5f, 0.0f, 0.0f, 0.0f,  // nuevo eje x
             0.0f, 0.5f, 0.0f, 0.0f,  // nuevo eje y
             0.0f, 0.0f, 0.5f, 0.0f,  // nuevo eje z
            -1.0f, 0.0f, 0.0f, 1.0f ) // desplazamiento
renderPart(parts[4].models, m, shader);
```

  Este código por ejemplo aplica una matriz que escala uniformemente la pieza a la mitad y la mueve 1 unidad hacia atrás en el eje x. Notar que en el código **la matriz se escribe transpuesta** respecto a como la escribimos habitualmente fuera del mismo. Esto se debe a que *OpenGL* espera matrices definidas por columna, pero en C/C++ las matrices se almacenan por fila[5].

- Se pueden sumar, restar o multiplicar las matrices con las sobrecargas de los operadores `+`, `-` y `*`.
- Se pueden construir matrices especiales con funciones de `glm` como [6]:
  - `glm::translate`, `glm::rotate`, `glm::scale`: construyen las matrices para las transformaciones básicas que indican sus nombres[7][8].
  - `glm::perspective`, `glm::frustrum`, `glm::ortho`: funciones para armar las matrices de proyección.
  - `glm::lookAt`: función que arma la matriz que "ubica la cámara" (que pasa del espacio del mundo al del ojo).

lugar de `0.5`), para que el compilador lo tome como `float`. Si no sería `double`, y la deducción de tipos del template podría fallar (por ej, por que al invocar una función de *glm*, la mezcla de argumentos entre `float`s y con `double`s le impida decidir con cual de los dos especializar).

[5] Es habitual que una API gráfica reciba a las matrices de 4x4 como arreglos lineales de 16 elementos, y cada api gráfica determina si el orden de esos 16 elementos deber ser por columna (*column-major*, como por ejemplo en *OpenGL*) o por fila (*row-major* como por ejemplo en *DirectX*).

[6] Estas funcionalidades eran parte de *OpenGL* (en versiones previas a la 3.2, o cuando se usan los perfiles de compatibilidad). La mayoría de las funciones de *glm* para armar matrices imitan a las originales de *OpenGL* y se usan en su reemplazo.

[7] Estas funciones además reciben una matriz previa como primer argumento, para multiplicar por la generada y permitir así encadenar fácilmente más de una transformación. Si no es necesario, se puede pasar la identidad como primer argumento ($glm :: mat4(1.0)$).

[8] Las funciones de *glm* que reciben ángulos (como `glm::rotate`) esperan que los mismos estén expresados en radianes. Pero dispone además de las funciones `glm::radians` y `glm::degrees` que convierten de un sistema a otro (el nombre de la función indica a qué unidad convertir, el argumento debe estar expresado en la otra).