

**UNIVERSIDAD**

**REY JUAN CARLOS**

**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA**

**DOBLE** **GRADO EN INGENIERIA INFORMATICA E INGENIERIA SOFTWARE**

**Curso Académico 2017/2018**

**Trabajo Fin de Grado**

## **SIMULACIÓN DE UN ROBOT SIGUE LINEAS**

### **Autor: David Vacas Miguel**

### **Directores/Tutor: Alberto Herrán González**

Índice

[Resumen 3](#_Toc493508170)

[Capítulo 1 – Introducción 4](#_Toc493508171)

[1.1 Motivación 4](#_Toc493508172)

[1.2 Objetivos 7](#_Toc493508173)

[1.3 Estado del arte 9](#_Toc493508174)

[1.4 Requisitos 10](#_Toc493508175)

[Capítulo 2 - Entorno tecnológico 11](#_Toc493508176)

[2.1 Cauce grafico 11](#_Toc493508177)

[2.2 OpenGL 16](#_Toc493508178)

[introduccion 17](#_Toc493508179)

[3-Objetivos 18](#_Toc493508180)

[4-Funcionamiento mecánico 20](#_Toc493508181)

[5-OpenGL (matrices) 24](#_Toc493508182)

[6-Descripción informática 28](#_Toc493508183)

[7-Interfaz gráfica 42](#_Toc493508184)

[8-Experimentos / validación 45](#_Toc493508185)

[9-Conclusiones 46](#_Toc493508186)

[10-Bibliografía 47](#_Toc493508187)

[11-Apéndices 48](#_Toc493508188)

Resumen

La aplicación realiza la simulación de un robot sigue líneas que, dado los parámetros del robot y un circuito, completará este último sin salirse del mismo.

Mecánicamente el robot se basa en dos sensores situados en la parte delantera del robot que comprueban si están sobrepuestos al circuito, en caso afirmativo la rueda correspondiente se frenará, reducirá su velocidad o, incluso, ira marcha atrás para que el robot realice un giro que le permita continuar su trayectoria dentro del circuito, en nuestro caso la rueda frenará.

Capítulo 1 – Introducción

1.1 Motivación

En una clase de la asignatura robótica, los alumnos, en grupos, debían realizar robots de distintos tipos: sigue líneas, velocistas, etc. Al final del curso el profesor realizaba un circuito y los alumnos usaban los robots que habían construido para ver cuál era el más rápido, creando así una competición de robots entre los alumnos. Lo rápido que el robot realizaba el circuito venía dado por las medidas que los alumnos utilizaban a la hora de construir el mismo, por ejemplo un robot con una separación de sensores mayor realizará menos giros pero más grandes sin embargo un robot con una separación menor realizará más giros pero de menor magnitud y esto con los demás parámetros influirá en la velocidad para realizar el circuito.

El objetivo de este proyecto es desarrollar una aplicación que permita simular y visualizar la dinámica de este robot móvil con direccionamiento diferencial. La aplicación servirá de banco de pruebas con el que analizar el comportamiento del robot sigue líneas antes de su construcción real.

Actualmente se dispone de un desarrollo previo en MATLAB-Simulink, ver figura 1 y figura 2.



**Figura 1.** Modelo del desarrollo hecho en MATLAB-Simulink.



**Figura 2.** Vista de la simulación desarrollada en MATLAB-Simulink.

Sin embargo tiene varios inconvenientes como: a) para poder utilizarlo hay que tener instalado dicho software; b) no permite cambiar los parámetros del escenario de una forma sencilla, estos parámetros se debían escribir en un vector cuya estructura debes conocer, como se puede ver en la figura 3; y c) no ofrece todas las funcionalidades requeridas.



**Figura 3.** Se puede observar el vector en el cual se deben poner los parámetros del robot.

Por ello, se pretende desarrollar dicho software creando una aplicación (integrando las tecnologías C++, Qt y OpenGL) cuyo único requisito para su uso sea el de tener instaladas en el sistema las librerías necesarias.

El destinatario de la aplicación son alumnos de una asignatura en la que se pretende que jueguen con el simulador antes de la construcción del robot real.

1.2 Objetivos

Una vez vistas las carencias de la aplicación que se tenía anteriormente, los objetivos que servirían para subsanarlas, además de otros objetivos adicionales serían los siguientes:

* Eliminar la necesidad de tener software instalado: con esta aplicación no tendrás la necesidad de tener software instalado, más allá de las librerías necesarias para la aplicación, por lo que será más fácil su descarga y utilización inicial.
* Facilidad a la hora de realizar cambios: el mayor problema a la hora de crear el robot era que una vez decidas las medidas y comprado o creado la pieza en cuestión era complicado cambiar esta para utilizar una distinta. Por ejemplo si se decidía que el radio de las ruedas seria de X una vez comprada la rueda deberían comprar una distinta si querían cambiarlo, otro ejemplo es el tamaño del robot, si se decidían porque el robot tuviera unas dimensiones de (Y, Z) pero más tarde se daban cuenta de que sería mejor tener un robot con unas dimensiones mayores, estos deberían volver a crear el soporte del robot que es el que indica las dimensiones.

Esto se soluciona en la aplicación con una simple interfaz user-friendly en la que podrán cambiar tantas veces como quieran las medidas de su robot, de forma sencilla y natural.

* Realización de pruebas con facilidad: debido a lo costoso de realizar un robot y la necesidad de crear un circuito por el cual el robot tenga que correr era complicado poder probar el robot creado, o diferentes opciones para el mismo.

Con esta aplicación se podrán realizar la cantidad de pruebas que se desee puesto que en unos segundos puedes tener tu robot simulado corriendo por el circuito dado sin necesidad de crear físicamente los mismos.

* Mejora del aprendizaje: igual que en el anterior punto, puesto que los alumnos no podían realizar pruebas con el robot de forma sencilla, no podían observar detenidamente los cambios que se creaban en el movimiento del robot utilizando diferentes medidas.

Puesto que aquí tienen una forma rápida de simular el movimiento del robot, pueden experimentar haciendo cambios en las medidas para poder ver qué cambios se realizan en el movimiento del robot y con esto ir aprendiendo con la visualización del mismo.

1.3 Estado del arte

En cuanto a las herramientas que se pueden encontrar en el mercado para simular un robot de este tipo se pueden encontrar de dos tipos:

* Aplicaciones creadas por una persona las cuales muestran cómo realizarlas o dejan la aplicación en un repositorio público (pocas de estas realizan esto último). Estas aplicaciones suelen tener dos problemas: la dificultad para su descarga y utilización, que no están preparadas para su uso puesto que están en las IDE correspondientes y la interfaz no es fácil de usar, en caso de que haya interfaz y no se tenga que cambiar los parámetros por código.
* Aplicaciones creadas por empresas. En este caso muchas de estas aplicaciones tienen una carencia en la UI, no se puede cambiar muy fácilmente los parámetros del robot. Lo bueno que tenía la herramienta encontrada es que no solo servía para un tipo de robot sino que incluía en la aplicación varios tipos de robots.

1.4 Requisitos

Los requisitos que debe tener la aplicación serían los siguientes:

* Facilidad de cambiar los parámetros del robot.
* Visualización de la simulación del robot de forma correcta.
* Integración de la interfaz y de la simulación en la aplicación.
* Interfaz user-friendly, fácil de utilizar para todos.
* Diseño simple de la simulación para poder ver mejor los movimientos de tu robot.

Capítulo 2 - Entorno tecnológico

2.1 Cauce grafico

El cauce grafico es el conjunto de transformaciones y procesados de la imagen que se realiza a los elementos que definen la escena hasta llegar a la imagen resultante final. Actualmente la generación de estas imágenes sigue una serie de pasos ya definidos.

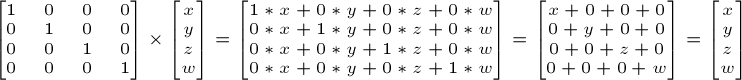
El cauce grafico se divide en varios pasos o etapas que se conectan entre ellas, es decir la salida de la primera será la entrada de la segunda, y así sucesivamente. En la figura 4 podemos observar las diferentes etapas y como se conectan como hemos citado anteriormente.



**Figura 4.** Etapas del cauce gráfico.

Las tres etapas principales del cauce grafico son: transformación, rasterizado y sombreado.

* Transformación: en esta etapa se transforma las coordenadas de los vértices de los objetos de su sistema local a su proyección en 2D. Esto se realiza mediante cálculos de matrices en los que una vez se han realizado todos los cálculos se consiguen las coordenadas en la proyección. A continuación realizare una breve explicación sobre las matrices más relevantes para el proyecto. Si se quiere utilizar en 3D, las matrices que se utilizan para los cálculos son matrices 4x4 siendo la matriz identidad la presentada en la figura 5.

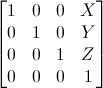


**Figura 5.** Matriz identidad.

Para obtener las coordenadas donde se encuentra cada punto del objeto dentro de la escena es necesario calcular la matriz model-view-projection (MVP), como su nombre indica, esta matriz viene dada por la multiplicación de tres matrices distintas las cuales realizan cálculos para la obtención de diferentes funcionalidades.

* + **Model:** Esta matriz lo que realizará es una transformación de la posición en el modelo a la posición global. Normalmente es una combinación de tres posibles movimientos: trasladar, escalar y rotar. Cada uno de estos movimientos vienen dados por matrices, las cuales se multiplicaran unas sobre otras, comenzando por la matriz identidad, dando así una única matriz que será la matriz model. Las matrices utilizadas para la traslación vienen dadas en la figura 6, para el escalado en la figura 7 y las diferentes matrices para la rotación, que dependen de en qué eje gires, en la figura 8.

Para realizar la traslación lo único que debes hacer es sustituir xyz por la cantidad de unidades que quieras mover en ese eje, así pues si sustituyes x por 5 realizarás una traslación de 5 unidades en el eje X.



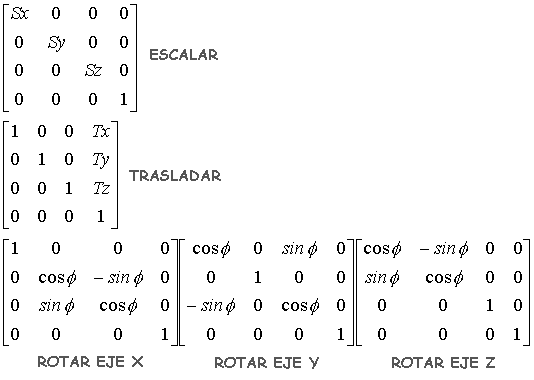
**Figura 6.** Matriz de translación.

En el caso del escalado la sustitución que realices indica el porcentaje de escalado que vas a realizar, esto resulta en que una sustitución por 1 no realiza escalado, una sustitución por 0,5 escala a la mitad de tamaño en el eje correspondiente y una sustitución por 2 escala al doble de tamaño en ese eje.



**Figura 7.** Matriz de escalado.

Para las rotaciones simplemente hay que sustituir *ϕ* por los grados que quieres hacer girar en ese eje el objeto. En caso de que quieras realizar varias rotaciones en distintos ejes, debes crear las diferentes matrices y multiplicarlas a la matriz principal.



**Figura 8.** Matrices de rotación.

* + **View:** Esta matriz hace las funciones de cámara, necesitando para poder usar esta matriz la posición, el punto hacia el que mira y la orientación de la cámara. Además esta matriz no solo sitúa la cámara sino que también realiza los cálculos para simular los movimientos que la cámara realizaría. Puesto que no se puede mover la cámara, lo que se realiza es mover el mundo en concordancia con esto, es decir se aplica al mundo la inversa del movimiento que la cámara realizaría, consiguiendo así el efecto de que la cámara se está moviendo. Por consiguiente, la posición de la cámara también es ilusoria puesto que la cámara siempre está en un punto fijo, pero como en el caso de querer mover la cámara, al decir que la posición de la cámara está en la posición X se está diciendo que el resto de objetos que no son la cámara deben reposicionarse para que al usuario le parezca que la cámara ha tenido un movimiento hacia otro punto del espacio.

* + **Projection:** Esta matriz definirá como se realiza la proyección, poniendo en esta matriz si la proyección será en perspectiva u ortográfica. Además permite realizar el clipping (desactivar la renderización) de los vértices que no son visibles.
  + **MVP:** Una vez tenemos las tres matrices habrá que multiplicarlas, siempre en el mismo orden, junto con el vector de posición del objeto en cuestión (v) y esto dará las nuevas coordenadas (v’), como se muestra en la siguiente ecuación:

v’ = Mprojection \* Mview \* Mmodel \* v

* Rasterizado: En esta etapa se convierten los vértices en fragmentos, estos últimos pueden verse como pixeles en potencia.
* Sombreado: Este paso es el que calcula el color de cada pixel, por lo tanto es la etapa en la que se leen las texturas.

2.2 OpenGL

Para poder implementar el cauce grafico existen diferentes APIs que se pueden utilizar como pueden ser Direct3D (para Windows), Mantle (para tarjetas AMD), Vulkan (basado en Mantle y multiplataforma) o OpenGL. Se decidió utilizar OpenGL puesto que es la librería más conocida, pública y multiplataforma, además es fácil integrarlo en la mayoría de los proyectos.

OpenGL es una API multilenguaje y multiplataforma que se utiliza para el desarrollo de aplicaciones en las que se utilicen gráficos 2D y 3D. Este te proporciona funciones con las cuales podras realizar imágenes, animaciones, juegos, simulaciones, etc. OpenGL te permite realizar entre otras muchas cosas:

* Construir formas geométricas a partir de las primitivas que este te proporciona.
* Ubicar los objetos en la escena.
* Ubicar el punto desde el que se visualiza la escena.
* Poner color o texturas.
* Crear luces.
* Realizar la rasterización (explicada en el punto anterior).

OpenGL tiene diferentes versiones que siguen pudiendo ser utilizadas hoy en dia, a continuación realizare una muy breve explicación de las más relevantes:

* OpenGL 1.X: en las diferentes actualizaciones fueron haciendo extensiones al núcleo de la API.
* OpenGL 2.X: se incorporó GLSL (OpenGL Shading Language), con el cual se podía programar las etapas de transformación y rasterizado del cauce grafico.
* OpenGL 3.X: en la primera etapa (OpenGL 3.0) se nombra ciertas funciones como obsoletas, que serán marcadas para ser eliminadas en futuras versiones (la mayoría de ellas en la versión 3.1).
* OpenGL 4.X:

A pesar de esto OpenGL tiene un problema, no es fácil crear una interfaz con la que un usuario pueda interactuar, por ello y para solucionar este problema se utiliza Qt en este proyecto.

introduccion

En una clase de la asignatura robótica, los alumnos, en grupos, debían realizar robots de distintos tipos: sigue líneas, velocistas, etc. Al final del curso el profesor realizaba un circuito y los alumnos usaban los robots que habían construido para ver cuál era el más rápido, creando así una competición de robots entre los alumnos. Lo rápido que el robot realizaba el circuito venía dado por las medidas que los alumnos utilizaban a la hora de construir el mismo, por ejemplo un robot con una separación de sensores mayor realizará menos giros pero más grandes sin embargo un robot con una separación menor realizará más giros pero de menor magnitud y esto con los demás parámetros influirá en la velocidad para realizar el circuito. Para un robot sigue líneas, que es el que nos concierne, las diferentes medidas que los alumnos debían decidir son:

* El radio, la separación y la distancia al eje de las ruedas.
* La separación y la distancia al eje de los sensores.
* Altura y anchura del robot.
* Potencia del robot (el profesor daba el mismo motor a todos los grupos por lo tanto esto no era algo que lo alumnos tuvieran que decidir).

Una vez decididas estas medidas los alumnos compraban el material o lo construían a partir de materiales y con ello formaban al robot.

La aplicación desarrollada simula el movimiento de un robot de este tipo (sigue líneas). La aplicación consta de dos partes, una zona en la que se visualizara el movimiento del robot en el circuito, y una interfaz en la que se tendrán que introducir los parámetros necesarios para poder realizar la simulación que son: las medidas que los alumnos debían decidir y un circuito.

3-Objetivos

Los objetivos más relevantes a la hora de realizar esta aplicación son:

* Facilidad a la hora de realizar cambios: el mayor problema a la hora de crear el robot era que una vez decidas las medidas y comprado o creado la pieza en cuestión era complicado cambiar esta para utilizar una distinta. Por ejemplo si se decidía que el radio de las ruedas seria de X una vez comprada la rueda deberían comprar una distinta si querían cambiarlo, otro ejemplo es el tamaño del robot, si se decidían porque el robot tuviera unas dimensiones de (Y, Z) pero más tarde se daban cuenta de que sería mejor tener un robot con unas dimensiones mayores, estos deberían volver a crear el soporte del robot que es el que indica las dimensiones.

Esto se soluciona en la aplicación con una simple interfaz user-friendly en la que podrán cambiar tantas veces como quieran las medidas de su robot, de forma sencilla y natural.

* Realización de pruebas con facilidad: debido a lo costoso de realizar un robot y la necesidad de crear un circuito por el cual el robot tenga que correr era complicado poder probar el robot creado, o diferentes opciones para el mismo.

Con esta aplicación se podrán realizar la cantidad de pruebas que se desee puesto que en unos segundos puedes tener tu robot simulado corriendo por el circuito dado sin necesidad de crear físicamente los mismos.

* Mejora del aprendizaje: igual que en el anterior punto, puesto que los alumnos no podían realizar pruebas con el robot de forma sencilla, no podían observar detenidamente los cambios que se creaban en el movimiento del robot utilizando diferentes medidas.

Puesto que aquí tienen una forma rápida de simular el movimiento del robot, pueden experimentar haciendo cambios en las medidas para poder ver qué cambios se realizan en el movimiento del robot y con esto ir aprendiendo con la visualización del mismo.

Para la realización de este proyecto se tenían dos bases importantes en cuanto a desarrollo: una interfaz user-friendly y poder ver la simulación del robot. Basándonos en esto se decidieron utilizar las siguientes tecnologías para el desarrollo de la aplicación:

* Qt 5.7.1: Se decidió por este SDK (kit de desarrollo de software) debido a la facilidad para realizar interfaces y la integración de OpenGL en el mismo.
* OpenGL: Se utilizó esta API para realizar la simulación del robot puesto que es una API conocida y se integraba bien con Qt.
* Qt Creator: Este IDE es el recomendado por Qt puesto que es el suyo propio, por lo tanto para tener una mayor facilidad con el mismo se decantó por este IDE.
* Mingw: Compilador utilizado.
* Glm 0.9.8: Librería utilizada para la realización de cálculos en la simulación del robot sigue líneas.
* Freeglut: Librería que se utilizara para dar forma al robot y los sensores.

4-Funcionamiento mecánico

4.1-Modelo cinemático

Según se ve en la Figura 1, suponiendo un sistema de referencia {G} y un sistema {L} con origen en el punto de guiado del vehículo y eje YL en la dirección del eje longitudinal del vehículo.



**Figura 1.** Modelo cinemático.

Por lo tanto, si el vehículo tiene una velocidad de desplazamiento y de rotación con respecto a {L}, con respecto a {G} la velocidad es:

4.2-Modelo discreto

La derivada de una función puede aproximarse por el cociente incremental:

Esto se conoce como derivada discreta hacia adelante, pero también puede aproximarse con el cociente incremental hacia atrás, la aproximación centrada, u otras aproximaciones más complicadas.

El resultado es que con una ecuación de este tipo, la nueva coordenada x {en t+1} se puede calcular a partir de la anterior (en t) mediante la ecuación:

Aplicando el mismo resultado al resto de coordenadas del modelo cinemático directo, se obtiene:

Por tanto, si se dispone de las coordenadas en cada instante de un determinado horizonte temporal, se puede calcular la nueva posición y orientación del robot en dicho horizonte.

4.3-Direccionamiento diferencial

El direccionamiento diferencial viene dado por la diferencia de velocidades de las ruedas laterales. La tracción se consigue con esas mismas ruedas. Adicionalmente, existen una o más ruedas para el soporte. En la figura 2 se muestra una imagen de dicho esquema. Nótese que para especificar la configuración hay que indicar los valores de las tres variables (x, y, ), siendo las variables de control las velocidades de las ruedas laterales.



**Figura 2.** Locomoción mediante guiado diferencial.

Sean y las velocidades de giro de las ruedas izquierda y derecha, respectivamente. Si el radio de la rueda es , las velocidades lineales correspondientes son y . Es este caso, la velocidad lineal y velocidad angular correspondientes en el modelo vienen dadas por:

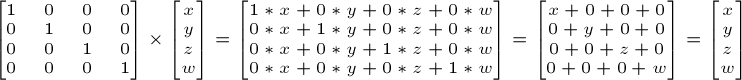
Sustituyendo estas expresiones en las obtenidas a partir de la Figura 1, se obtienen las velocidades de las coordenadas del robot en el sistema {G} a partir de la velocidad de giro de cada rueda:

Finalmente, utilizando el modelo discreto, se obtiene:

5-OpenGL (matrices)

Antes de comenzar con la explicación del código desarrollado, realizaré una pequeña explicación sobre el funcionamiento de OpenGL.

OpenGL es una API que realiza cálculos matemáticos relativamente complejos en funciones para que el usuario pueda utilizarlas como cajas negras sin necesidad de saber mucho del tema. Básicamente OpenGL realiza cálculos de matrices que una vez realizados todos estos cálculos se acaba obteniendo el punto en el espacio donde se visualizarán los objetos, además de estos cálculos OpenGL realiza la renderización de los objetos para que se puedan visualizar. A continuación realizare una breve explicación sobre las matrices más relevantes para el proyecto (es decir las matrices utilizadas para el cálculo de la matriz model). Si se quiere utilizar en 3D, las matrices que se utilizan para los cálculos son matrices 4x4 siendo la matriz identidad la presentada en la figura 3.



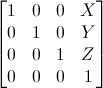
**Figura 3.** Matriz identidad.

Para obtener las coordenadas donde se encuentra cada punto del objeto dentro de la escena es necesario calcular la matriz model-view-projection (MVP), como su nombre indica, esta matriz viene dada por la multiplicación de tres matrices distintas las cuales realizan cálculos para la obtención de diferentes funcionalidades.

5.1-Model

Esta matriz lo que realizará es una transformación de la posición en el modelo a la posición global. Normalmente es una combinación de tres posibles movimientos: trasladar, escalar y rotar. Cada uno de estos movimientos vienen dados por matrices, las cuales se multiplicaran unas sobre otras, comenzando por la matriz identidad, dando así una única matriz que será la matriz model. Las matrices utilizadas para la traslación vienen dadas en la figura 4, para el escalado en la figura 5 y las diferentes matrices para la rotación, que dependen de en qué eje gires, en la figura 6.

Para realizar la traslación lo único que debes hacer es sustituir xyz por la cantidad de unidades que quieras mover en ese eje, así pues si sustituyes x por 5 realizarás una traslación de 5 unidades en el eje X.



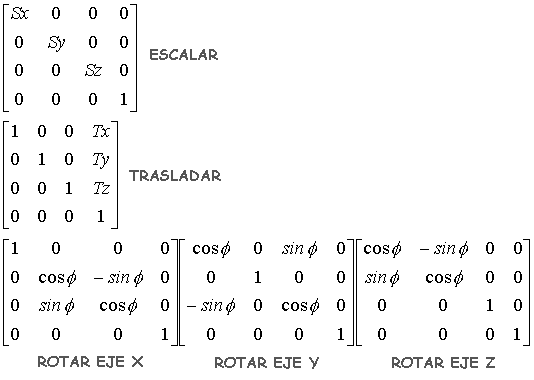
**Figura 4.** Matriz de translación.

En el caso del escalado la sustitución que realices indica el porcentaje de escalado que vas a realizar, esto resulta en que una sustitución por 1 no realiza escalado, una sustitución por 0,5 escala a la mitad de tamaño en el eje correspondiente y una sustitución por 2 escala al doble de tamaño en ese eje.



**Figura 5.** Matriz de escalado.

Para las rotaciones simplemente hay que sustituir *ϕ* por los grados que quieres hacer girar en ese eje el objeto. En caso de que quieras realizar varias rotaciones en distintos ejes, debes crear las diferentes matrices y multiplicarlas a la matriz principal.



**Figura 6.** Matrices de rotación.

5.2-View

Esta matriz hace las funciones de cámara, necesitando para poder usar esta matriz la posición, el punto hacia el que mira y la orientación de la cámara. Además esta matriz no solo sitúa la cámara sino que también realiza los cálculos para simular los movimientos que la cámara realizaría. Puesto que en OpenGL no se puede mover la cámara, lo que se realiza es mover el mundo en concordancia con esto, es decir se aplica al mundo la inversa del movimiento que la cámara realizaría, consiguiendo así el efecto de que la cámara se está moviendo. Por consiguiente, la posición de la cámara también es ilusoria puesto que la cámara siempre está en un punto fijo, pero como en el caso de querer mover la cámara, al decir que la posición de la cámara está en la posición X se está diciendo que el resto de objetos que no son la cámara deben reposicionarse para que al usuario le parezca que la cámara ha tenido un movimiento hacia otro punto del espacio.

5.3-Projection

Esta matriz definirá como se realiza la proyección, poniendo en esta matriz si la proyección será en perspectiva u ortográfica. Además permite realizar el clipping (desactivar la renderización) de los vértices que no son visibles.

5.4-MVP

Una vez tenemos las tres matrices habrá que multiplicarlas, siempre en el mismo orden, junto con el vector de posición del objeto en cuestión (v) y esto dará las nuevas coordenadas (v’), como se muestra en la siguiente ecuación:

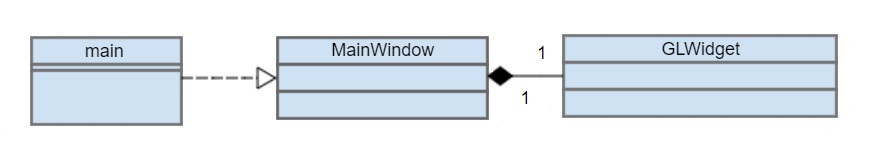
v’ = Mprojection \* Mview \* Mmodel \* v

6-Descripción informática

Puesto que es una aplicación desarrollada en Qt, el cual trabaja sobre C++, en el proyecto se encontraran archivos de 4 tipos:

* .pro: Solo habrá un archivo .pro en el proyecto. Este archivo contiene toda la información necesaria para realizar la build de la aplicación.
* .h: Estos archivos incluyen las variables y las cabeceras de las funciones de la clase correspondiente, tanto públicas como privadas.
* .cpp: Son los archivos en los cuales está situado el código fuente de la clase.
* .ui: Este archivo es el correspondiente a la interfaz gráfica.

Esta aplicación consta de un diseño simple de 3 clases. La clase ‘main’ utiliza la clase ‘MainWindow’ la cual contiene a la clase ‘GLWidget’ como se muestra en la figura 7.



**Figura 7.** UML sin variables ni métodos.

A continuación se explicara en detalle el funcionamiento de las diferentes clases desarrolladas, comenzando por la más simple hasta llegar a la más compleja:

6.1-main

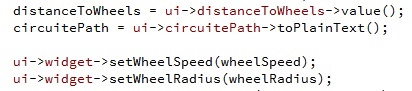
A pesar de ser esta la clase más sencilla y con menos codigo de todas, es la clase principal de la aplicación sobre la que se ejecutan el resto puesto que es la primera en ejecutarse al iniciar la aplicación.

Esta clase lo que realiza es la inicialización de freeglut y la creación de una instancia de la clase MainWindow. A continuación utiliza el método show de esta última que mostrará el widget principal (“MainWindow”) y sus widgets hijos.

6.2-MainWindow

Esta clase se divide en tres archivos: ‘MainWindow.h’ que contiene las variables y las cabeceras de las funciones de esta clase (como se dijo anteriormente), ‘MainWindow.ui’ la cual contiene toda la información sobre la estructura de la interfaz, la cual dejaremos para más adelante y ‘MainWindow.cpp’ que pasaremos a explicar su funcionamiento a continuación, que es el que contiene el código fuente de la clase.

El funcionamiento de esta clase es básico, funciona como conector entre la clase ‘GLWidget’ y la interfaz gráfica. Esta conexión se realiza a partir del método on\_insertParameters\_clicked la cual se llamara en caso de que el usuario pulse el botón con el texto “Insert Parameters”. Este método accederá a los datos escritos en los widget de tipo input situados en la interfaz y se los enviara por medio de métodos de la clase ‘GLWidget’ a esta misma clase, como se muestra en la figura 8.



**Figura 8.** Recogida de datos y su paso a la clase ‘GLWidget’.

Además esta clase tiene un método público llamado setTimer el cual llamará la clase “GLWidget” y que actualizara el reloj de la interfaz que lleva el tiempo que lleva activa la simulación.

6.3-GLWidget

En esta clase se desarrollan todos los cálculos relacionados con la simulación del movimiento del robot y la visualización por medio de OpenGL.

La clase ‘GLWidget’ hereda de la clase ‘QOpenGLWidget’, la cual es propia de Qt y es un widget que se utiliza para renderizar gráficos en OpenGL. Esta te proporciona funciones básicas para la realización de proyectos con OpenGL. Las tres funciones más importantes que se heredan son:

* initializeGL(): Esta función se ejecuta una sola vez y antes que las otras dos funciones. Por lo tanto se utiliza para inicializar y configurar todo lo necesario para la utilización de OpenGL u otros.
* paintGL(): Esta función es la que renderiza la escena de OpenGL y se llama siempre que el widget necesite ser actualizado, en este caso, puesto que se necesita de una animación para simular el movimiento del robot, se llamará cada X milisegundos para poder visualizar la animación. Además este método será en el cual se modificará la posición de la cámara y la posición del robot, es decir, las matrices view y model. Gracias a los cambios en esta última matriz se realiza la animación de movimiento del robot.
* resizeGL(int w, int h): En este método se debe configurar el viewport (los parámetros de entrada w y h son el ancho y el alto respectivamente de la zona donde se podrá visualizar el código desarrollado en OpenGL), el tipo de vista (perspectiva u ortográfica), etc. Es llamado por primera vez cuando el widget se crea (siempre después de initializeGL) y siempre que el widget sea reescalado. Este método es el responsable de crear la matriz projection.

Para la explicación de esta clase se irá desarrollando en el orden en el que se ejecutan las funciones:

**Creación de la clase**

A la hora de crear la clase se inicializan ciertos parámetros como la distancia a la cual el sensor detecta que está lo suficientemente cerca del circuito y hace parar a la rueda y el tiempo con el que se actualiza la función paintGL, como se explica a continuación. Además lo más importante es la inicialización de un “timer” que se conecta a la función update haciendo que cada vez que se acaba el tiempo del “timer” se ejecute la función update. La función update lo que hace es una llamada a la función paintGL. Por lo tanto lo que estamos consiguiendo con esto es ejecutar la función paintGL cada ‘X’ms que nosotros queramos y por lo tanto actualizar la posición del robot cada ‘X’ms.

**InitializeGL**

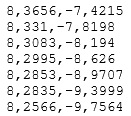
En este método simplemente se colorea el fondo de blanco para que haya contraste con el robot y los sensores.

**Recepción de parámetros de la UI**

Antes de la recepción de parámetros se ejecutan también resizeGL y paintGL pero puesto que no está fijada ni el robot, ni el circuito, ni la forma en la que se proyectará la imagen, el código de estas funciones están congeladas por un parámetro que se activara en el momento que sea necesario.

La recepción de parámetros desde la UI empieza en el momento en que el botón “Insert Parameters” es pulsado, en ese momento la clase “MainWindow” llama a los métodos setter y startRace, de los que cabe destacar sensorPosition, setCircuite y startRace.

* **sensorPosition** es un método que, además de guardar en las variables correspondientes los datos que se le introducen como entrada (los datos del sensor), calcula la posición ‘X’ del vector de posición de los sensores gracias a que uno de los datos que entran en el parámetro es la distancia entre los sensores, por lo tanto (sabiendo que el robot siempre mirara hacia arriba al principio), la ‘X’ de cada sensor será la mitad de la separación en negativo para el sensor izquierdo y la mitad de la separación del sensor para el sensor derecho.
* **setCircuite** es un método en el que entra el path del archivo .txt que contiene las coordenadas de los puntos del circuito y se utiliza para buscar el archivo en el equipo. Este archivo debe tener un formato específico que se muestra en la figura 9.



**Figura 9.** Formato del archivo del circuito, formado por las

coordenadas x z de los puntos del circuito.

En caso de que se encuentre el archivo se leerá y se guardarán los puntos en una variable para más tarde poder utilizarlo para pintar el circuito. Sabiendo que la cámara se sitúa mirando hacia y=0 y la parte de arriba de la cámara está situada hacia –z, se buscan los puntos del circuito más externos, es decir, el punto situado más a la izquierda (x menor), más a la derecha (x mayor), más arriba (z mayor) y más abajo (z menor). Una vez encontrados se crea un vector de 3 posiciones en el cual guardaremos la posición donde más adelante situaremos la cámara. En este vector se guarda en la primera posición (x) el punto intermedio entre el punto situado más a la izquierda y el punto situado más a la derecha, a la hora de guardar la posición 3 (z) se realiza exactamente lo mismo pero con la parte superior e inferior, es decir se calcula y se guarda el punto medio entre el punto más superior y el punto más inferior. Para poder saber la altura a la que se debe situar la cámara para que pueda ver todo el circuito, primero se comprueba cual contiene una mayor distancia si el eje x (es decir el punto más a la derecha menos el punto más a la izquierda) o el eje z (es decir el punto más arriba menos el punto más abajo). Una vez obtenida la mayor distancia se calcula mediante trigonometría la altura necesaria, como se puede ver en la figura 10.



**Figura 10.** Trigonometría utilizada para el cálculo de la altura de la cámara.

Después de realizar los cálculos pertinentes para situar la cámara en el centro del circuito y que este se pueda ver entero, se guarda la posición del primer punto dado en el circuito como el punto donde el robot tendrá su posición inicial.

* **startRace** es el método que calcula la posición inicial de los sensores respecto al robot, activa la variable de control que permite ejecutar el código situado en resizeGL y paintGL y realiza dos redimensionados, el primero agranda el widget y el segundo vuelve a dejarlo como al principio, con esto se consigue que se ejecute resizeGL de nuevo y con ello todo el código que ya está disponible para ser ejecutado.

**resizeGL**

Lo primero que realiza este método es definir el viewport (los pixeles de la ventana que se van a utilizar para que la imagen final sea dibujada) con el valor inicial 0,0 (es decir la esquina inferior izquierda) y los dos parámetros de entrada (ancho y alto de la ventana del widget), es decir utilizaremos toda la ventana. A continuación se inician los cálculos para obtener la matriz de proyección, para esto se utilizan los métodos de la librería glm: perspective u ortho en función de lo que el usuario haya señalizado en la interfaz. Estos métodos realizan todos los cálculos correspondientes a la matriz de proyección, una vez obtenida esta matriz, la cargamos en la pila de OpenGL.

La diferencia principal entre estos dos tipos de vistas se basa en que la vista en perspectiva mantiene las dimensiones reales de los objetos si nos acercamos o nos alejamos de ellos, por lo que la vista en perspectiva se acerca más a la vista de una persona. En la figura 11 y 12 podemos observar como dos cámaras, una de cada tipo de vista, generan el plano de proyección a partir de los puntos de los objetos del espacio.



**Figura 11.** Vista en perspectiva. **Figura 12.** Vista ortográfica.

Esto se debe a que en una vista ortográfica los puntos de los objetos se proyectan de forma perpendicular al plano de proyección, mientras que en la vista en perspectiva los puntos se dirigen al punto central de la cámara o centro de proyección.

**paintGL**

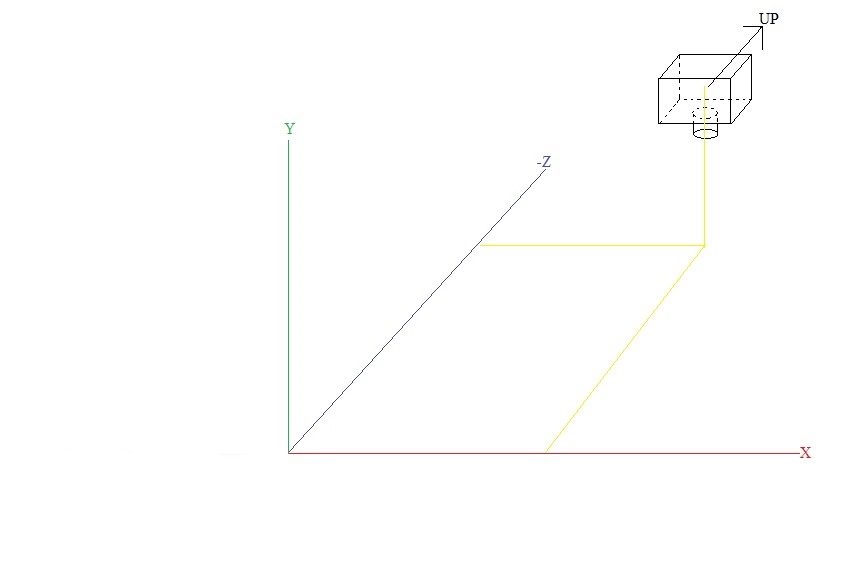
Es este el método que contiene toda la información sobre la simulación del robot, pues en este método es en el que se realizan los cálculos para las matrices model y view. Antes de comenzar hay que recordar que este método será llamado cada ‘X’ms gracias al timer que se creó cuando la clase fue creada.

Para iniciar este método lo primero que se realiza es un clear de los buffer color y de profundidad y a continuación se activa el modo modelview para indicar que las matrices que se añadirán a la pila serán las de view y model. Para decir que estamos en el modo modelview se utiliza el método glMatrixMode(GL\_MODELVIEW), en caso de querer cargar el modo de la matriz projection se realiza igual sustituyendo modelview por projection.

Después de esto se realiza el cálculo del tiempo que lleva la simulación iniciada para enseñarlo por pantalla y poder ver lo que tarda el robot en hacer el circuito. Inmediatamente después se carga en la pila la matriz identidad (figura 3) para poder iniciar los cálculos de la matriz modelview.

* **View matrix**

A continuación se crean las variables que indican la posición y orientación de la cámara, siendo la posición la calculada anteriormente en el método setCircuite, el punto hacia donde mira esa misma posición pero con la y=0, y la parte superior de la cámara en sentido –z, dando un resultado similar al de la figura 13 en la que no tiene en cuenta la posición del circuito.



**Figura 13.** Posición y orientación de la cámara.

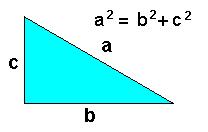
Una vez obtenidos los tres puntos (posición, punto hacia donde mira y orientación de la cámara) se les mete como parámetros al método de glm lookAt que con estos datos te devuelve la matriz view. Por consiguiente una vez obtenida la matriz view se carga en la pila como se hizo anteriormente con la matriz projection con el método glLoadMatrixf.

* **drawCircuite**

Para comenzar se empieza dibujando el circuito, el cual anteriormente habíamos guardado sus puntos en una estructura de datos (una lista en este caso). Para dibujar el circuito se define un grosor de línea de 2 unidades y se itera sobre la lista creando líneas (con glVertex3f) entre cada uno de los puntos de la misma.

* **movementController**

A continuación se pasa a decidir si en ese instante alguna de las dos ruedas debe pararse o deben continuar a su velocidad normal. Para ello se calcula la distancia de cada uno de los sensores a los diferentes puntos del circuito mediante el teorema de Pitágoras, véase en la figura 14.



**Figura 14.** Teorema de Pitágoras.

Fijándose en la figura 14, nuestra ‘b’ seria la coordenada ‘x’ del punto del circuito menos la coordenada ‘x’ de la posición sensor, y la ‘c’ seria la coordenada ‘z’ del punto del circuito menos la coordenada ‘z’ de la posición sensor.

Esto se realizará con cada sensor y todos los puntos del circuito, en caso de que la distancia a uno de estos puntos sea menor que el umbral marcado, la rueda en cuestión se frenara y dejara de realizarse el cálculo de las distancias para ese sensor.

En este momento es cuando se comienza el cálculo de la matriz model y por consiguiente la simulación del robot.

* **Model matrix**

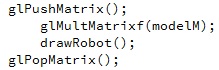
Para comenzar a calcular esta matriz lo primero que debemos hacer es calcular la posición del robot en ese instante mediante las formulas descritas en el apartado “Funcionamiento mecánico”, que adaptadas a las condiciones de este proyecto acaban teniendo la siguiente forma:

Gracias a estas fórmulas podemos obtener la posición del robot en este instante teniendo en cuenta el instante anterior.

Una vez obtenida la nueva posición se hace que la simulación del robot se traslade y rote como se ha calculado mediante los métodos translate y rotate de glm. Estos dos métodos calcularán la matriz model utilizando las matrices de la figura 4 (en el caso de translate) y de la figura 6 (en el caso de rotate) y se creará así la matriz model como se vio en el apartado de OpenGL, es decir, se multiplican entre ellas.

Para calcular la nueva posición de los sensores utilizamos la matriz model, que multiplicándola por el vector inicial que marcaba la posición del sensor respecto al robot, nos dará como resultado las nuevas coordenadas de los sensores.

Después de esto se engloba el cálculo de la matriz model en la pila (creando así la matriz mvp) y el dibujado del robot entre dos funciones (glPushMatrix() al inicio y glPopMatrix() al final como se puede observar en la figura 15) para que los cambios del movimiento del robot (es decir la matriz model) solo le afecten a este y a los sensores. Este paso podría haberse omitido pues al estar en último lugar la carga/multiplicación de la matriz model en la pila solo habría afectado a lo siguiente que en este caso solo se trata del robot, pero se ha hecho así para que en caso de mejora o utilización del proyecto se pueda incluir cosas después o dentro del movimiento así como para aclarar que el movimiento solo se realiza en el robot y los sensores.



**Figura 15.** Paquete glPushMatrix-glPopMatrix.

Una vez dentro de glPushMatrix se utiliza el método glMultMatrixf para multiplicar la matriz model en la pila y conseguir así la matriz mvp. En este caso se utiliza glMultMatrixf en vez de utilizar glLoadMatrix puesto que OpenGL diferencia las cargas de proyection y modelview y por lo tanto puesto que en la pila se haya la matriz view, si utilizamos glLoadMatrixf, nos cargaríamos la matriz view y pondríamos en su lugar la matriz model, dejando así sin posición de la cámara al proyecto, utilizando glMultMatrixf lo que realizamos es multiplicar la matriz en la pila, en este caso view, con la matriz model realizando así de forma correcta los cálculos pertinentes.

* **drawRobot**

Por último y englobado dentro del paquete glPushMatrix-glPopMatrix llamamos al método creado por nosotros drawRobot el cual dibujara el robot y los sensores.

En primer lugar se dibujaran los sensores a partir de los datos obtenidos por medio de la interfaz según el usuario ha querido. Estos se representan mediante cubos de 0.05 unidades que ya están modelados por la librería freeglut.

Para dibujar el robot se sigue la misma teoría con alguna pequeña diferencia, en primer lugar se baja el robot a la distancia a la que el usuario haya decidido que debe estar el punto donde se colocan las ruedas puesto que desde este punto es desde el cual debe girar el robot. A continuación se realiza un escalado del robot para que este tome las medidas deseadas por el usuario. Finalmente se crea el robot mediante freeglut con unas dimensiones de 1 unidad. Cabe destacar que se hace en este orden puesto que las transformaciones que se realizan antes de crear el objeto se aplicaran a todos los objetos creados después de hacer estas mismas. Al ser el último objeto y estar englobado en el paquete glPushMatrix-glPopMatrix estas transformaciones solo se realizaran al cuerpo del robot. También cabe destacar que las transformaciones realizadas a los sensores para determinar su posición están englobadas en glPushMatrix-glPopMatrix para que no afecten al siguiente sensor y/o el cuerpo del robot.

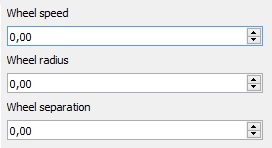
Antes de que este método finalice se resetean las velocidades de las ruedas a su velocidad original para que en caso de que en el siguiente instante no sea necesario parar ninguna rueda estas continúen a la misma velocidad.

7-Interfaz gráfica

En este apartado explicaremos el porqué de las decisiones tomadas sobre la interfaz. El archivo que contiene toda la información sobre la interfaz y sus widgets es “MainWindow.ui”. Para comenzar se decidió que la parte donde se realiza la simulación ocupara cerca del 75% de la aplicación puesto que es la parte más importante, el lugar donde más tiempo se estará observando y donde más detalles hay que fijarse.

Una vez entrada en la parte de interacción con el usuario se decidió que estuviera a la derecha por similitud a la mayor parte de aplicaciones encontradas y por lo tanto que le resultara más natural al usuario el acceso a esta. Además todas las labels estarán en inglés puesto que este es el idioma más hablado y por tanto dota a la aplicación de una mayor usabilidad.

A continuación se decidió por incluir los inputs en grupos de conceptos similares, como el grupo de las ruedas, el de los sensores y el del tamaño del robot. En el caso de las ruedas se decidió por utilizar tres inputs con sus respectivos labels en diferentes niveles, como se observa en la figura 16, puesto que la unión de los 3 parámetros de las ruedas en un mismo nivel no resultaba fácil de visualizar.



**Figura 16.** Paquete de ruedas en tres niveles.

Después de las ruedas viene un parámetro que comparten las ruedas y el robot, que es la distancia a la que se sitúan las ruedas desde el extremo superior del robot. Por lo tanto este parámetro debía estar situado entre los parámetros de las ruedas y los del robot.

Al contrario que para las ruedas, para el tamaño del robot y los parámetros de los sensores, se agruparon en 2 en un mismo nivel puesto que así se asimila a simple vista que esos parámetros están relacionados, como se ve en la figura 17.



**Figura 17.** Paquete de dimensiones del robot en un nivel

Hasta aquí están los parámetros de entrada necesarios para la simulación del robot (a excepción del circuito). Por tanto ahora deberían entrar las opciones de la aplicación que no son directamente relevantes para la simulación de la aplicación, en este caso solo hay una opción que se trata de dos radio button que deciden si el usuario desea la vista perspectiva u ortográfica. Esta opción está colocada aquí puesto que a pesar de que todavía queda introducir el circuito, si se pusiera detrás del input para el circuito esta opción sería poco visible y podría ignorarse, mientras que viniendo de opciones de tamaño reducido se ve claramente.

El último input se trata de un cuadro de texto editable en el que se debe introducir el path del circuito que se desea utilizar.

Por ultimo hay una zona en la que se muestran los segundos pasados desde que empezó la simulación como se muestra en la figura 18, con un tamaño de letra que hace que sobresalga sobre el resto de la interfaz haciendo así que una vez la simulación este comenzada se vea claramente la simulación y los segundos, pudiendo ignorar el resto de la interfaz.



**Figura 18.** Tiempo de simulación.

Por ultimo hay dos botones, el primero cuyo texto pone “Insert Parameters”, el cual cuando pulses se introducirán todos lo parámetros a la aplicación y se iniciara la simulación. Y un último botón que contiene el texto “Quit” que sirve para cerrar la aplicación.

8-Experimentos / validación

No tengo claro que debería incluir aqui

9-Conclusiones

Gracias a este proyecto he obtenido bastante conocimiento sobre C++, además de sobre una tecnología relativamente reciente como es Qt. Para poder realizarlo además ha sido necesario indagar en el funcionamiento más profundo de OpenG, dotándome así de un mayor conocimiento sobre cómo funciona a bajo nivel.

De los objetivos principales propuestos a la hora de realizar la aplicación, han sido todos logrados.

Una de las posibles mejoras a realizar sobre este proyecto podría ser el de incluir una forma de realizar carreras varios robots a la vez, o realizar estas carreras de manera online, una persona realizando de host y el resto poniendo sus robots y viendo la carrera. Además se podría mejorar la forma y figuras del robot y el circuito.

10-Bibliografía

Estoy fuera de mi casa y sin internet en el ordenador para poder buscar los links de la documentación y otras páginas, cuando regrese lo colocaré.

11-Apéndices