2011

Proyecto Final Informática Gráfica

QTron

Implementación de un juego basado en Tron mediante el uso de OpenGL y QT como herramienta de desarrollo.

Miguel Cantón Cortés
miwelc@correo.ugr.es
76655569L

Introducción

Como proyecto final para la asignatura he decidido desarrollar un juego basado en Tron, y más concretamente, en las motos que aparecen en la película.

El juego ha sido desarrollado usando OpenGL mediante QT, por lo que puede ser compilado tanto para Windows como para Linux.

El objetivo ha sido alcanzar el punto en el cual el juego "se pueda jugar", es decir, aunque no tenga gran número de opciones, que al menos sirva a su propósito básico de entretener.

Descripción del Juego

El juego consiste en guiar sobre un tablero cuadrado una moto, la cual va dejando tras de sí una estela, evitando salirse de éste y chocar contra las estelas de los adversarios o la nuestra propia.

Aunque el juego en principio se plantea simple, puede adquirir un poco de complejidad según las estrategias que se usen para intentar vencer, puesto que a diferencia de juegos similares como el antiguo "Snake" de los móviles, uno puede tomar una gran variedad de estrategias; unas más agresivas como intentar acorralar al adversario, por ejemplo poniéndose de acuerdo con otro jugador, o más pasivas limitándose a recorrer el tablero simplemente evitando quedar encerrado.

Menú del Juego

Mediante la herramienta QT se ha integrado un menú básico para elegir modos y cambiar opciones.

- **Juego**
  - ***Empezar Partida Nueva***
  - ***Pausar Partida***
  - ***Finalizar Partida***
- **Opciones**
  - ***Elegir número de jugadores***
  - ***Elegir el tamaño del tablero***
  - ***Elegir velocidad de las motos***
- **Modos de visualización**
  - ***Activar/Desactivar reflejos***
  - ***Mostrar/Ocultar ejes***
  - ***Cámara libre***
  - ***Puntos***
  - ***Líneas***
  - ***Sólido***
  - ***Ajedrez***
  - ***Suavizado Flat***
  - ***Suavizado Smooth***
- **Pantalla**
  - ***Completa***
  - ***Ventana***
- **Extra**
  - ***Ayuda***
  - ***Acerca***
  - ***Demo***

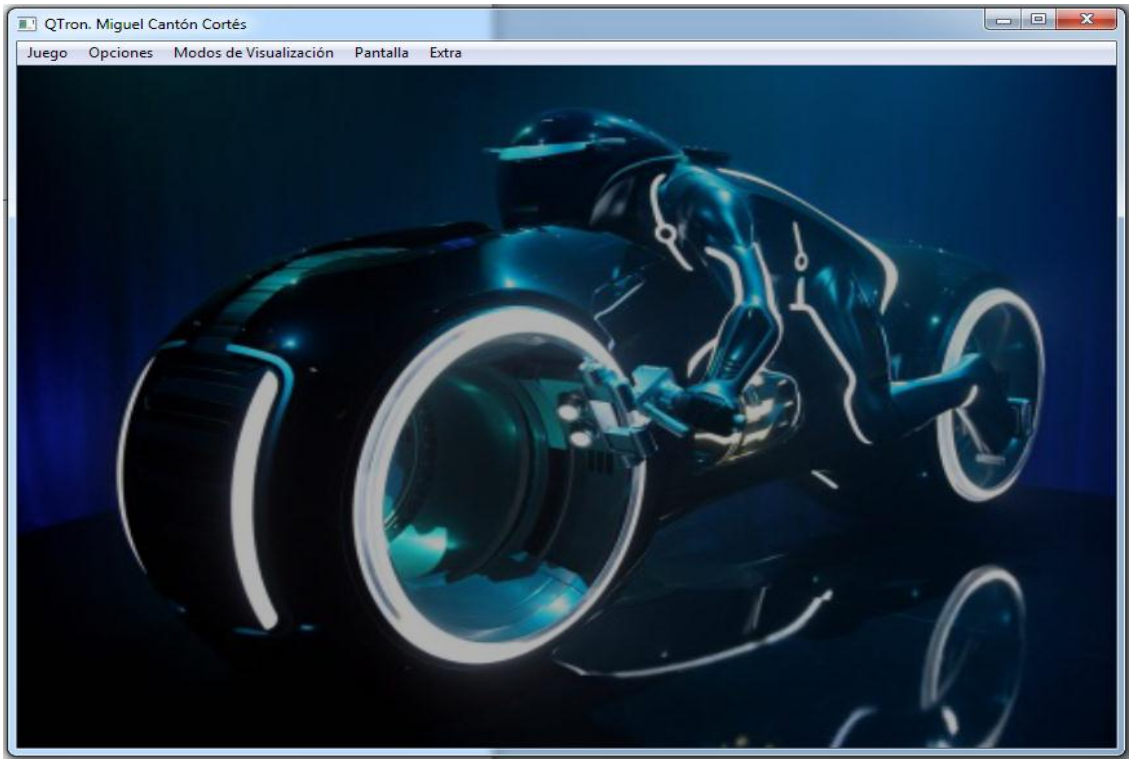
Controles

Los controles son muy simples dada la naturaleza del juego. Cada jugador dispone de una tecla para girar a la izquierda y otra para girar a la derecha, en adición a otra para cambiar entre varios tipos de cámaras. Son las siguientes:

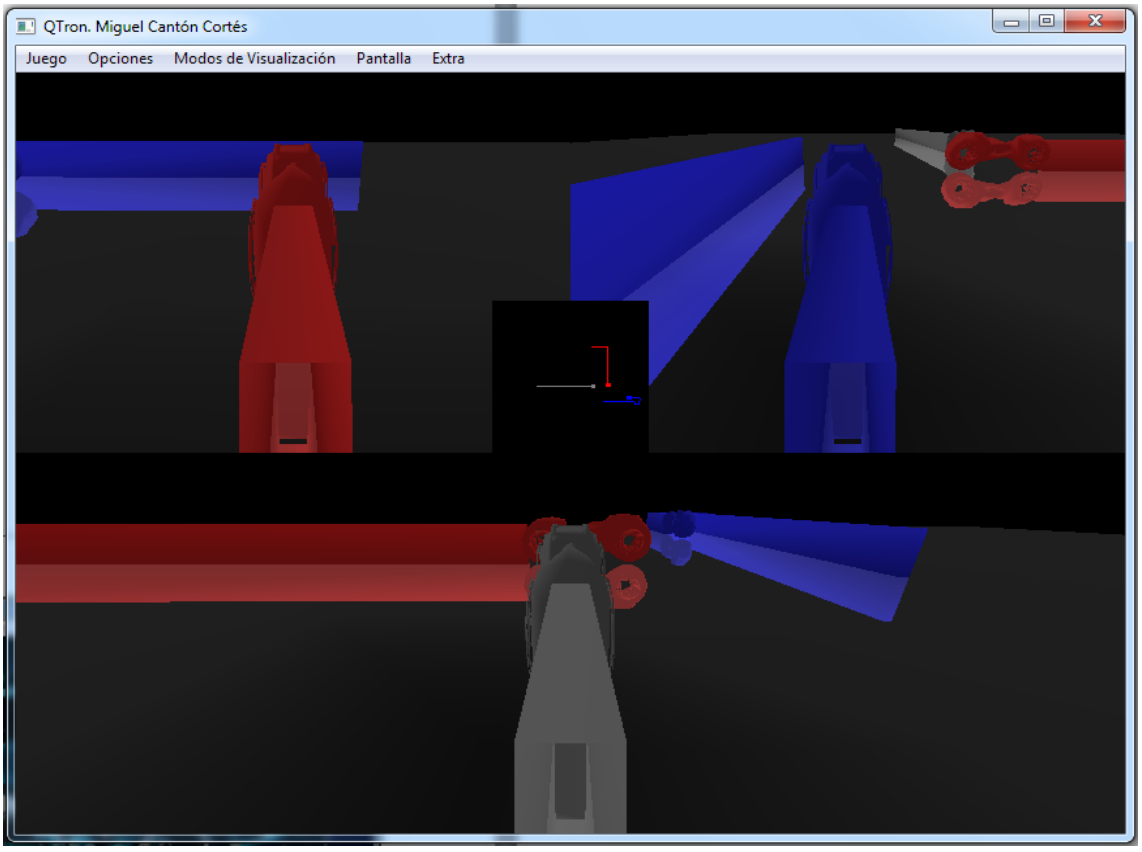
	Izquierda	Derecha	Cambiar cámara
Jugador 1	A	S	Z
Jugador 2	G	H	B
Jugador 3	L	Ñ	.
Jugador 4	←	→	↑

Capturas

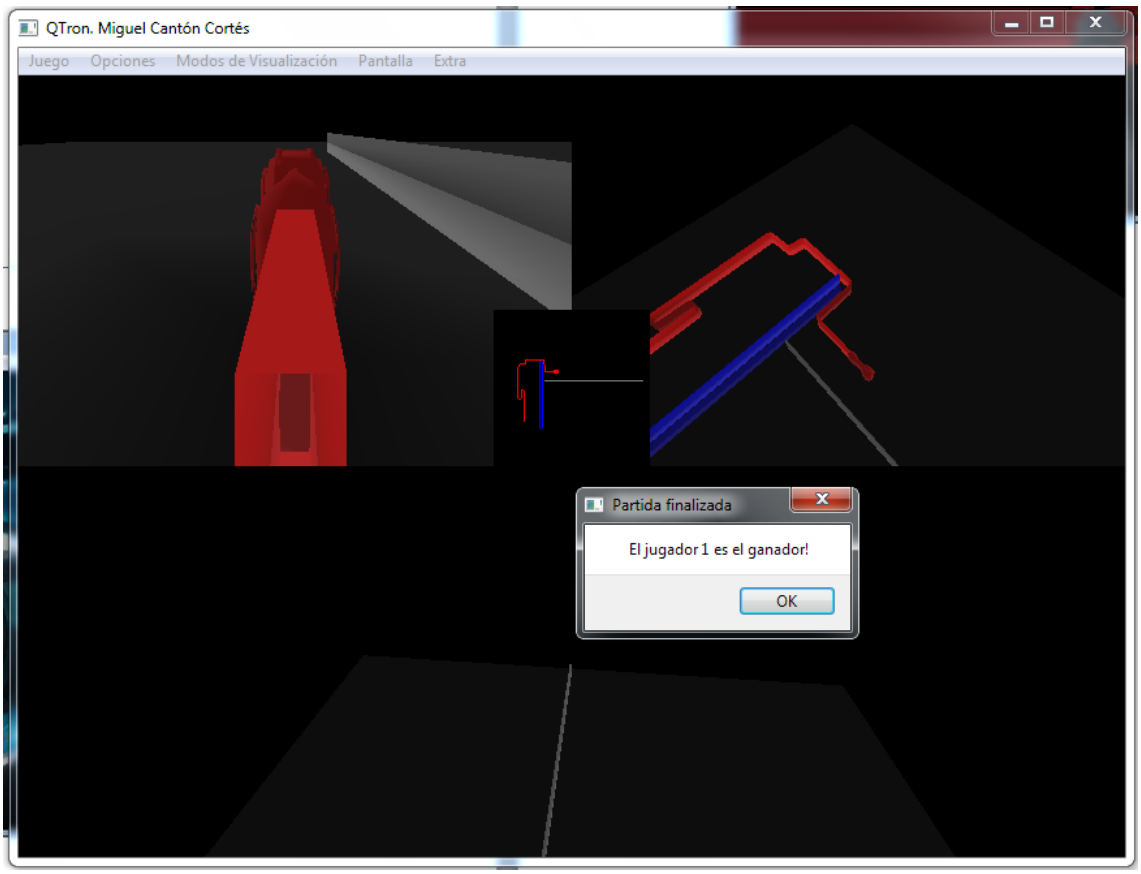
- **Menu**



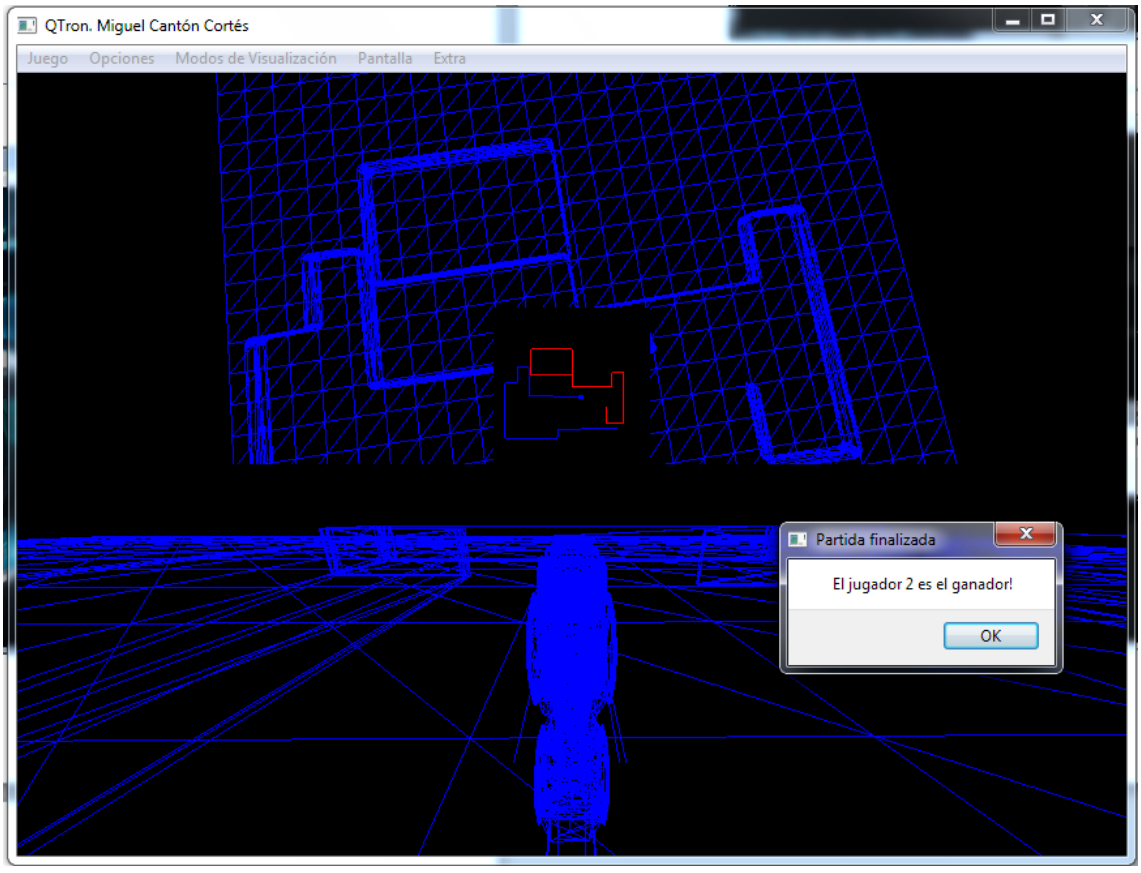
- **Partida Empezada**



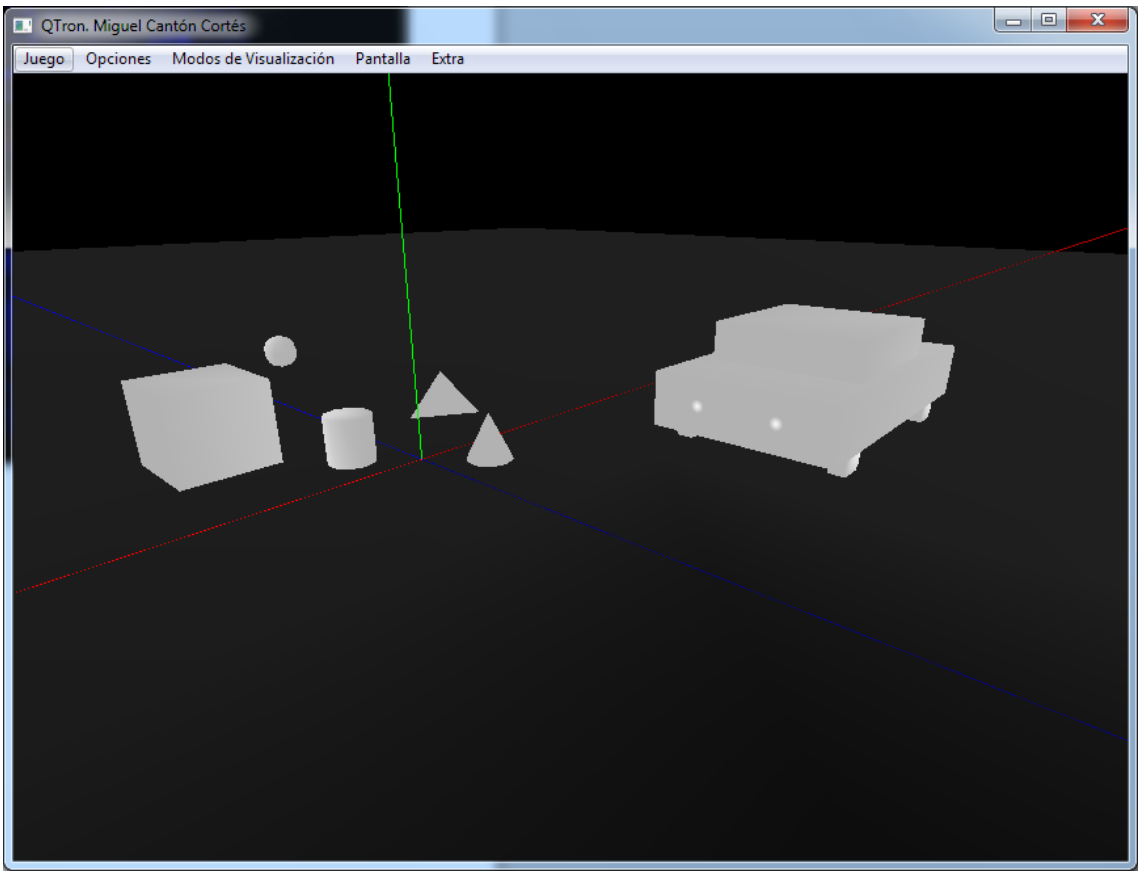
- **Partida Finalizada**



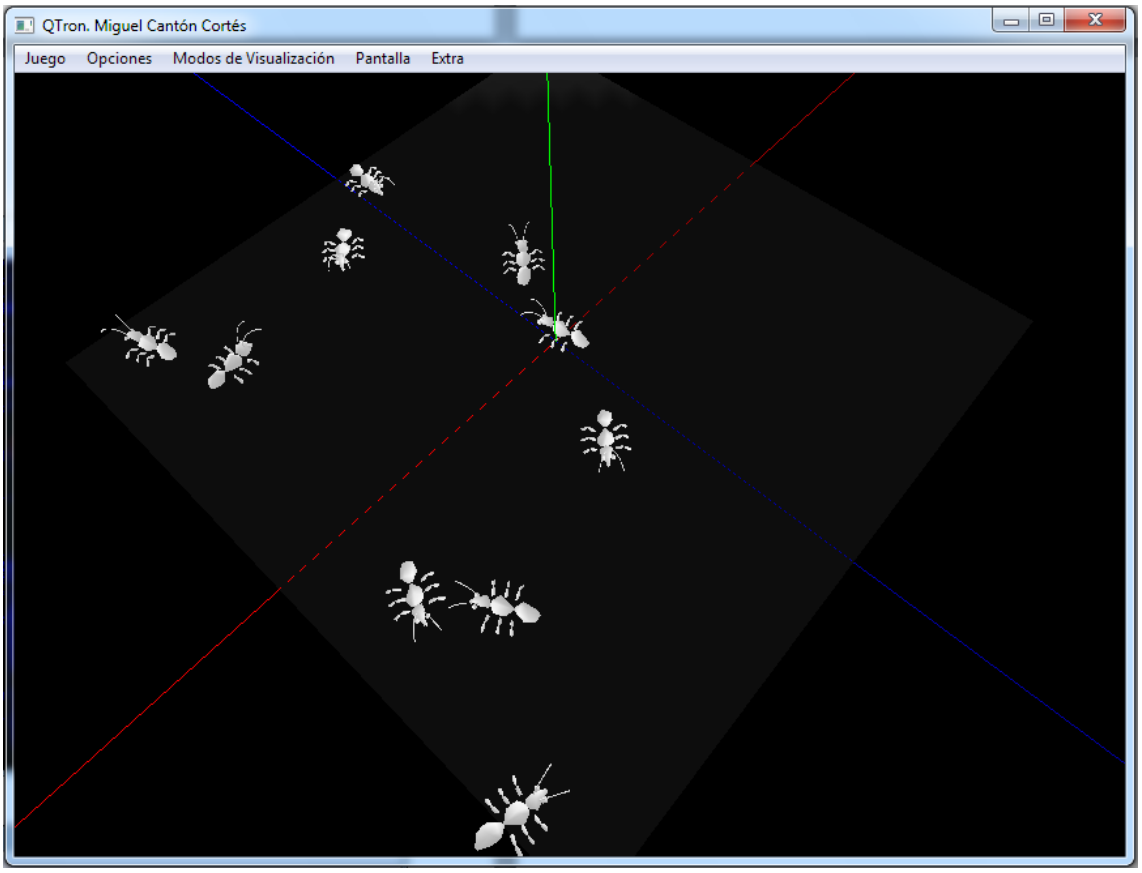
- **Modo líneas**



- **Modo Demo (Figuras)**



- **Modo Demo (Hormigas)**



Implementación General

La implementación se ha llevado a cabo en C++ usando QT, es decir, programación orientada a objetos, dirigida por eventos.

Gracias al uso de C++ se ha podido modularizar adecuadamente en clases el programa.

Las clases principales son:

- **Objeto3D**: se encarga de gestionar la geometría de los modelos, su carga, adición de nuevos vértices y caras a los ya existentes, así como su dibujado, independientemente de la geometría.
- **Entidad3D**: lleva asociado mediante un puntero un *Objeto3D*. Se encarga de gestionar las características externas a la geometría tales como posición, rotación, colores, animaciones etc. Varias *entidades* pueden llevar asociado un mismo *objeto3d*.
- **Particulas**: proporciona funcionalidad para poder gestionar conjuntos de *entidades* con facilidad.
- **Camara**: gestiona todo lo relativo a una cámara.
- **Luz**: permite posicionar y configurar una luz de manera sencilla.
- **Textura**: se encarga de cargar imágenes y mostrarlas como un objeto 3d más.
- **Vértice**: la implementación de esta clase ha sido fundamental tanto para definir, gestionar y operar con vectores y puntos.

Las métodos de los que disponen estas clases son:

```
class Objeto3D {
   […]
   public:
      void generarRevolucion(float *puntos, int nPuntos, int nSecciones, float altura);
      void anadirVertices(Vertice *Vertices, int nVertices);
      void anadirVertices(QVector<Vertice> Vertices);
      void setVertice(int nVertice, Vertice valor);
      Vertice getVertice(int nVertice) { return vertices[nVertice]; }
      void anadirCaras(Cara *Caras, int nCaras);
      void cargarDatos(QVector<Vertice> Vertices, QVector<Cara> Faces);
      void cargarDatos(QVector<float> Vertices, QVector<int> Faces);
      virtual void draw(int modoDibujo, GLenum modoSuavizado, GLfloat
colorSolido[3], GLfloat coloresAjedrez[2][3]);
};
```

```
class Entidad3D {
   […]
   public:
```

```
        void asociarObjeto(Objeto3D *objasc);
        Vertice getVertice(int n) { return obj3d->getVertice(n); }
        void draw();
        void drawReflection();
        void drawTranslucido();
        void setModoDibujo(int modo);
        void setModoSuavizado(GLenum modo);
        void setColorSolido(GLfloat r, GLfloat g, GLfloat b);
        void setColorSolido(const GLfloat col[3]);
        void getColorSolido(GLfloat col[3]);
        void setColoresAjedrez(GLfloat r1, GLfloat g1, GLfloat b1, GLfloat r2, GLfloat
g2, GLfloat b2);
        void setEscala(Vertice esc);
        void setEscala(float x, float y, float z) ;
        void setEscala(float esc);
        void setFrente(Vertice fre);
        void setFrente(float x, float y, float z) ;
        void setCentro(Vertice cent);
        void setCentro(float x, float y, float z);
        Vertice getCentro();
        Vertice getPosicion();
        void setElevacion(float ele);
        void setPosicion(Vertice pos) ;
        void setPosicion(float x, float y, float z);
        Vertice getOrientacion();
        void setOrientacion(Vertice ori);
        void setOrientacion(float x, float y, float z);

        //Animación
        void animarMov(Vertice destino);
        void animarMov(Vertice destino, float vel);
        void animarAvanzar(float vel);
        void animarGiro(Vertice destino);
        void animarGiro(Vertice destino, float vel);
        Vertice getPosicionAleatoria(int margenX, int margenY, int margenZ);
        void animacionAleatoria();
        void animacionAleatoria(int margenX, int margenY, int margenZ);
        void pararAnimacion();
        void pararMover();
        void pararGirar();
        void pararAvanzar();
        void cambiarVelocidadMov(float vel);
        void cambiarVelocidadGiro(float vel);
        void setTipoAceleracion(TipoAceleracion acel);
};
```

```
class Particulas {
   […]
   public:
      void crearConjunto(Objeto3D *obj3d, int num);
```

```
        void setFrente(Vertice fren);
        void setPosicion(Vertice pos);
        void setPosicion(int n, Vertice pos);
        Vertice getPosicion(int n);
        void draw();
        void setModoDibujo(int modo);
        void setModoSuavizado(GLenum modo);
        void setColorSolido(GLfloat r, GLfloat g, GLfloat b);
        void setEscala(float esc);

        void animacionAleatoria();
        void animacionAleatoria(int margenX, int margenY, int margenZ);
        void cambiarVelocidadMov(float vel);
        void cambiarVelocidadGiro(float vel);
        void setTipoAceleracion(TipoAceleracion acel);
    public slots:
        void comprobarColisiones();
};
```

```
class Camara {
    […]
    public:
        void setCamara();
        void configurarCamara(Vertice vrp, Vertice vpn, Vertice vup);
        Vertice getPosicion();
        void setPosicion(GLfloat x, GLfloat y, GLfloat z);
        void setPosicion(Vertice pos);
        void setDireccion(Vertice dir);
        void setDireccion(GLfloat x, GLfloat y, GLfloat z);
        void mover(int tecla);
};
```

```
class Luz{
    […]
    public:
        void encender();
        void apagar();
        void setPosicion(GLfloat x, GLfloat y, GLfloat z, GLfloat a);

        void setComponenteAmbiente(GLfloat r, GLfloat g, GLfloat b, GLfloat a);
        void setColoresLuz(GLfloat *ambiente, GLfloat *difusa, GLfloat *especular);
        void setLuzAmbiente(GLfloat r, GLfloat g, GLfloat b, GLfloat a);
        void setLuzDifusa(GLfloat r, GLfloat g, GLfloat b, GLfloat a);
        void setLuzEspecular(GLfloat r, GLfloat g, GLfloat b, GLfloat a);

        void setColoresMaterial(GLfloat *ambiente, GLfloat *difusa, GLfloat *especular);
        void setMaterialAmbiente(GLfloat r, GLfloat g, GLfloat b, GLfloat a);
        void setMaterialDifusa(GLfloat r, GLfloat g, GLfloat b, GLfloat a);
        void setMaterialEspecular(GLfloat r, GLfloat g, GLfloat b, GLfloat a);
```

```
        void setBrillo(GLfloat br);
};
```

```
class Textura {
    […]
    public:
        void cargarTextura(const char* archivo, int anch, int alt);
        void draw();
        void setPosicion(Vertice pos);
        void setPosicion(float x, float y, float z);
        void setEscala(Vertice esc);
        void setEscala(float x, float y, float z);
        void setEscala(float esc);
};
```

```
class Vertice {
    […]
    public:
        Vertice();
        Vertice(double x, double y, double z);
        Vertice(double x, double y, double z, double w);
        Vertice(const Vertice &vertice);
        void setFromEsfericas(float radio, float phi, float theta);
        float getPhi() const;
        float getTheta() const;
        float *v() { vect[0] = x; vect[1] = y; vect[2] = z; vect[3] = w; return vect; }
        bool operator==(const Vertice &vertice);
        bool operator!=(const Vertice &vertice);
        Vertice &operator=(const Vertice &vertice);
        Vertice &operator-=(const Vertice &vertice);
        Vertice &operator+=(const Vertice &vertice);
        Vertice operator-(const Vertice &vertice) const;
        Vertice operator+(const Vertice &vertice) const;
        Vertice operator*(const Vertice &vertice) const;
        Vertice operator*(const float num) const;
        Vertice operator*(_matrix4<float> &Matrix1) const;
        float productoEscalar(const Vertice &vertice) const;
        Vertice operator/(const float &divisor) const;
        double modulo() const;
        Vertice versor() const;
        Vertice vectorHacia(const Vertice punto) const;
        double anguloV(const Vertice v2) const;
        double anguloX() const;
        double anguloY() const;
        double anguloZ() const;
};
```

Implementación características específicas del juego

Posiblemente la parte más importante del juego propiamente dicho sea la gestión de las motos y sus estelas, es decir, el manejo de las motos, sus colisiones y la generación de las estelas.

Motos

La clase que controla las motos simplemente es una capa de abstracción extra que hace uso de la clase *Entidad3D* para las cosas más complejas como las animaciones, posicionamiento y apariencia de las motos en general, y se encarga de implementar el manejo de éstas, así como las "reglas" del juego propiamente dichas (las motos giran en ángulos de 90º, las colisiones...).

Estelas

La estela es un elemento básico del juego. Se genera tras la moto, siguiendo la orientación de ésta y queda como un bloque sólido contra el cual pueden estrellarse los jugadores. Es obvio, por tanto, que los elementos que la componen, vértices y caras, deben ser calculados y almacenados durante el juego, puesto que en cada partida las estelas son diferentes. La clase *Estela* cumple este cometido, ya que únicamente indicándole en cada momento el punto nuevo desde el cual se "genera" la estela la actualiza.

Para poder "crear" la estela de una forma eficiente sin necesidad de añadir caras inútiles, la clase *Estela* comprueba si el nuevo punto tiene la misma orientación que el anterior, y por tanto solo tiene que *alargar* la última sección en lugar de añadir nuevas caras y vértices. Para que en las curvas, todas de 90º, no haya más caras de las necesarias, se considera que la orientación a cambiado cuando la orientación anterior y la nueva difieren como mínimo PI/2 / 4 radianes, es decir, se divide en 4 secciones cada giro.

Colisiones

Para las colisiones se han empleado *bounding boxes,* es decir, cajas no visibles que en vuelven al objeto, y que dada su geometría simple (un cubo estirado) facilita detectar las colisiones. Dado que los elementos del juego tienen una forma más parecida a un paralelepípedo que a una esfera, este método resulta más preciso que la detección de colisiones por esfera envolvente; especialmente claro se ve esto en las estelas, que en los

largos tramos rectos, debido a la desproporción entre longitud y anchura, si se usara el otro método la detección de colisiones sería muy imprecisa.

Dado que uno de los objetivos de este juego es encerrar a los rivales para que colisionen contra las estelas, he tenido que tener esto en cuenta a la hora de ajustar algunos elementos del juego (por ejemplo, el centro de las motos está situado en la rueda trasera, de forma que los giros de 90º sean más precisos y no tiendan a *abrir* en las curvas) y en lo que respecta a las colisiones, las estelas son representadas mediante *bounding boxes* y las motos mediente un punto situado ligeramente adelantado a la estela, en la parte trasera de la moto. Esto se ha hecho así por un motivo: si no se permitiera que la moto atravesara ligeramente por la parte frontal una estela perpendicular, los giros no podrían ser lo suficientemente ajustados para poder juntar nuestra estela con otra y dejar *cerrada* un área del tablero.

Cámaras

Pese a que en el menú el número de jugadores seleccionable está limitado a 8 como máximo, el juego está diseñado de tal forma que podría gestionar tantas motos y cámaras como se quieran. Esto se hace dividiendo la pantalla en tantos *viewports* como jugadores haya.

Hay varios tipos de cámaras a disposición del jugador que puede cambiar en cualquier momento de la partida mediante una tecla. Cada cámara sigue a su jugador.

Como extra se ha incluido una cámara libre que se puede activar en el menú *Modos de visualización* y que se maneja mediante los cursores la vista y mediante *A, S, D* y *W* el movimiento.

Mapa

Para facilitar el juego se ha incluido un mapa situado en el centro de la pantalla para poder visualizar rápidamente la situación de cada moto y de la estela que dejan a su paso.

Para hacer más práctica esta opción, las estelas son representadas mediante líneas y las motos mediante cuadrados.

Modo Demo

Se incluye un modo extra para poder ver las posibilidades de las clases comentadas con anterioridad, como la generación de figuras mediante perfil de revolución y montar figuras mediante conjuntos de *Entidades*