Universidad de Valladolid

E.T.S. DE INGENIERÍA INFORMÁTICA

Ingeniero Técnico en Informática de Sistemas

Desarrollo de un motor gráfico y bibliotecas de apoyo para la construcción de videojuegos

---

Alumno: Javier Casas Velasco

Tutor: José Manuel Marqués Corral

Índice general

Capítulo 1

Introducción

1.1. Introducción

En el mundo actual el entretenimiento es una industria que ha tenido un gran desarrollo. Con enormes facturaciones anuales, el ocio es, sin duda, una fuente de dinero para quienes lo sepan aprovechar. Y, dentro del ocio, la informática ha buscado su lugar, y lo ha encontrado: los videojuegos. La industria del videojuego ha crecido como la espuma y ya compite con la industria del cine y la música.

El videojuego es un programa informático interactivo, creado para el entretenimiento, y basado en la interacción entre una o varias personas y un equipo informático, como puede ser un ordenador o una videoconsola. Muchas veces, el videojuego recrea un entorno o siuación virtual, donde el jugador controla ciertos personajes o elementos, para conseguir unos objetivos, siguiendo unas reglas determinadas.

Desde la época de los grandes mainframes, las personas ha utilizado los equipos informáticos para divertirse. Los principios fueron sencillos, con juegos pequeños que desarrollaban los programadores para pasar el poco tiempo libre del que disponían aquellos ordenadores. El videojuego como industria no apareció hasta que se popularizó la informática y aparecieron los ordenadores personales. Con el tiempo aparecerían ordenadores específicos para videojuegos, llamados videoconsolas, o vulgarmente, consolas de videojuegos.

Con la aparicion de hardware específico (las videoconsolas) quedó más que claro que el videojuego no era una moda pasajera. Y también estaba claro que el desarrollo de la industria derivaría a la creación de videojuegos más complejos. Posteriormente, el continuo desarrollo tecnológico de la informática hizo que apareciera competición por la creación de videojuegos más amplios, complejos, y con más y más contenido. Todo esto nos lleva al mundo actual, donde el videojuego es el resultado del trabajo de un gran equipo de desarrollo.

El hecho de que el videojuego se haya profesionalizado y convertido en industria ha propiciado también que a su desarrollo se intente aplicar las reglas de la ingeniería. El objetivo es sencillo: crear videojuegos de calidad con un presupuesto ajustado, y en un tiempo concreto. A fin de cuentas, no es más que un tipo muy concreto de aplicación informática.

Uno de los métodos que usa la ingeniería para acelerar el desarrollo y abaratar costes consiste en la creación de piezas genéricas, que puedan ser reutilizadas en gran cantidad

de situaciones con un coste, en conjunto, menor. Uno de los modelos estructurales más utilizados en videojuegos y aplicaciones interactivas, es el patrón arquitectónico Modelo-Vista-Controlador. Las piezas principales de dicho patrón son:

- **El modelo**:
  Contiene la información, datos y reglas del videojuego. En el modelo se procesan las reglas concretas, el comportamiento de los distintos personajes y objetos, y la interacción que tiene el jugador en el videojuego.

- **La vista**:
  Muestra la información contenida en el modelo al jugador, en forma de imágenes en la pantalla y sonido en el altavoz. La vista lee información del modelo, y convierte esa información en gráfico en pantalla y sonido en altavoces.

- **El controlador**:
  Recibe las acciones que el jugador utiliza para controlar el videojuego. El controlador se encarga de organizar el flujo general del videojuego, y de recibir las acciones e información que vengan del exterior.

De estas piezas, el modelo es bastante específico; y es difícil hacer un modelo genérico que sirva para cualquier tipo de videojuego. En cambio, la vista y el controlador son más generalizables. Casi todos los videojuegos de hoy día muestran gráficos tridimensionales en forma de mallas de polígonos. Por tanto, es posible desarrollar un software que pueda ser utilizado para la construcción de muchos juegos modernos distintos. Esto mismo pasa con el sonido y los dispositivos de entrada. Al final, el objetivo es tener gran parte de la funcionalidad del videojuego ya construída; y reducir los costes desarrollando sólo esas partes específicas de cada videojuego concreto.

Por eso, se elige una de esas piezas para el desarrollo de este proyecto. La pieza elegida es el motor gráfico, que irá dentro de la vista. Ese motor gráfico se encargará de dibujar los gráficos del videojuego en pantalla. Sin embargo, es difícil utilizar un motor gráfico directamente, sin recurrir a ningún tipo de estructura de apoyo. Por ello se desarrollará además unas bibliotecas básicas, siguiendo la estructura del videojuego, sobre las que este motor gráfico funciona adecuadamente. Así, en el estado final, este proyecto ofrece la estructura genérica de un videojuego, y un motor gráfico ya disponible.

1.2. Objetivos

- Construir un motor gráfico orientado a las tres dimensiones. Este motor es el que se encargará de dibujar los gráficos de los videojuegos que desarrollemos posteriormente. Será capaz de pintar escenas compuestas por distintos objetos y personajes, con una iluminación adecuada, y mostrada desde un punto de vista determinado por la posición de una o más cámaras. Además, este motor gráfico no olvida el mundo de las 2 dimensiones, sino que dejará espacios y lugares específicos para introducir estos elementos cuando sean necesarios.

- Construir una biblioteca de apoyo para la creación de videojuegos. Esta biblioteca de apoyo se encargará de algunas tareas típicas dentro de videojuegos, como son la organización de los distintos personajes y objetos, y la aplicación de las reglas del juego. Además, esta biblioteca implementará la estructura y arquitectura general de un videojuego.
- Construir un ejemplo básico de aplicación que utilice los sistemas desarrollados. En este caso será un pequeño programa que mostrará un planetario.

Capítulo 2

Requisitos

2.1. Introducción

El sistema desarrollado implementa una estructura general y una colección de componentes pensados para simplificar el desarrollo de un videojuego. El principal componente es el Motor Gráfico, encargado de la tarea de mostrar el juego. Además se han implementado dos bibliotecas de apoyo: una encargada de leer entidades complejas de disco, y otra encargada de manipular y procesar la Lógica del Juego. Para ordenar adecuadamente la ejecución general del sistema, se ha añadido un Controlador General. Las características del sistema están definidas por los Requisitos especificados a continuación:

2.2. Requisitos

- Requisito: **G.1**
  El sistema desarrollado contendrá un Motor Gráfico, que cumplirá las características especificadas por los siguientes requisitos.

  - Requisito: **G.1.1**
    El Motor Gráfico estará preparado para representar gráficos 3D en tiempo real, estando orientado a videojuegos.

  - Requisito: **G.1.2**
    El Motor Gráfico estará pensado y preparado para producir fácilmente al menos 20 fotogramas cada segundo.
    Se llama fotograma a cada imagen completa producida por el Motor Gráfico. Si la escena a representar es muy compleja es posible que esta tasa de rendimiento no sea alcanzable.
  - Requisito: **G.1.3**
    El Motor Gráfico seguirá las órdenes y representará los gráficos que le pidan otros

subsistemas.
Otros subsistemas se encargarán de definir y controlar las escenas dibujadas por el Motor Gráfico.

- Requisito:
  Cuando reciba la orden, el Motor Gráfico pintará todas sus imágenes, a partir del estado y configuración de ese instante.

- Requisito: **G.1.4**
  El Motor Gráfico mostrará sus imágenes en una o varias pantallas, en función de cómo haya sido configurado por otros subsistemas.

  - Requisito: **G.1.4.1**
    Si la aplicación de la cual el Motor Gráfico es parte se ejecuta en un entorno de ventanas, el Motor Gráfico se comportará con dicho entorno de ventanas en función de como sea configurado.

  - Requisito: **G.1.4.2**
    Los modos de comportamiento disponibles en un entorno de ventanas son:

    - **En ventana**: El Motor Gráfico pinta la escena en una ventana.
    - **En pantalla completa**: El Motor Gráfico pinta la escena utilizando toda la pantalla, e ignorando el entorno de ventanas.

  - Requisito: **G.1.4.3**
    Para las pantallas y modos en los que sea posible, el Motor Gráfico ofrecerá mecanismos para modificar la resolución y/o profundidad de color.

- Requisito: **G.1.5**
  Para representar las imágenes, el Motor Gráfico hará uso de la biblioteca OpenGL.

- Requisito: **G.1.6**
  El Motor Gráfico estará preparado para ser portado fácilmente a otra biblioteca gráfica diferente de OpenGL.

- Requisito: **G.1.7**
  A cada pantalla disponible se le podrá asignar una escena que dibujará el Motor Gráfico.

- Requisito: **G.1.8**
  Las escenas estarán compuestas de diferentes capas de pintado, de tal manera que la imagen sea el fruto de la aplicación sucesiva de las capas.

- Requisito: **G.1.9**
  Habrá varios tipos diferentes de capas:

  - Requisito: **G.1.9.1**
    Capa de borrado: esta capa pintará toda la imagen del color especificado.

  - Requisito: **G.1.9.2**
    Capa de borrado del buffer de profundidad: esta capa reiniciará el buffer de profundidad.

  - Requisito: **G.1.9.3**
    Capa de escena 3D: esta capa pintará un conjunto de objetos tridimensionales vistos desde una cámara especificada.

- Requisito: **G.1.10**
  Las escenas que pinta la *Capa de escena 3D* especificada anteriormente tendrán las siguientes características:

  - Requisito: **G.1.10.1**
    Estarán orientadas a utilizar las tres dimensiones: ancho, alto y profundidad; también llamados ejes x,y,z.

  - Requisito: **G.1.10.2**
    Estarán compuestas por diferentes clases de objetos:

    - Requisito: **G.1.10.2.1**
      Cámara de proyección perspectiva: representa lo que observa siguiendo una proyección con perspectiva. Los objetos tendrán un tamaño aparente en función de la distancia a la cámara; cuanto más cercanos, más grandes, y cuanto más lejanos, más pequeños.

    - Requisito: **G.1.10.2.2**
      Cámara de proyección ortográfica: representa lo que observa con proyección ortográfica, sin perspectiva. Los objetos tendrán un tamaño aparente constante, con independencia de su posición en la escena.

    - Requisito: **G.1.10.2.3**
      Iluminador puntual: emite luz desde un punto en todas direcciones.

    - Requisito: **G.1.10.2.4**
      Foco: emite luz desde un punto en un cono.

- ◇ Requisito: **G.1.10.2.5**
  Iluminador desde el infinito: ilumina la escena con rayos de luz paralelos.

- ◇ Requisito: **G.1.10.2.6**
  Objetos geométricos: cada uno tiene asignada una malla de polígonos.

- ○ Requisito: **G.1.10.3**
  Cada objeto tiene una transformada, compuesta por varias partes:

  - ◇ Requisito: **G.1.10.3.1**
    Posición o translación: define dónde está el centro del objeto con respecto al origen.

  - ◇ Requisito: **G.1.10.3.2**
    Orientación o rotación: define el giro que tiene el objeto.

  - ◇ Requisito: **G.1.10.3.3**
    Escala: define el tamaño relativo que tiene el objeto.

- ○ Requisito: **G.1.10.4**
  Cada objeto tiene dos direcciones características asociadas: la dirección frontal, que corresponde al vector (0,0,1) desde el punto de vista del objeto, y la dirección vertical, que corresponde al vector (0,1,0).

  - ◇ Requisito: **G.1.10.4.1**
    Para un objeto concreto, estas direcciones pueden cambiar afectadas por la transformada del objeto.

  - ◇ Requisito: **G.1.10.4.2**
    Las cámaras observan la escena mirando en su dirección frontal. Para una cámara concreta, el punto central de la imagen tomada corresponde a la proyección de su vector frontal.

  - ◇ Requisito: **G.1.10.4.3**
    Las cámaras toman la imagen teniendo en cuenta su vector vertical. Para una cámara concreta, su vector vertical determina la dirección hacia la que está la parte superior de la imagen tomada.

  - ◇ Requisito: **G.1.10.4.4**
    Los focos emiten luz en un cono cuyo eje es la dirección frontal.

  - ◇ Requisito: **G.1.10.4.5**
    Los iluminadores desde el infinito emiten sus rayos en su dirección frontal.

- Requisito: **G.1.10.5**
  Las cámaras de proyección en perspectiva tienen las siguientes características:

  - Requisito: **G.1.10.5.1**
    Tienen definida una distancia máxima y mínima. Las entidades que estén más lejos que la distancia máxima, o más cerca que la distancia mínima a la cámara no serán dibujados por dicha cámara.

  - Requisito: **G.1.10.5.2**
    Tienen definida una proporción en los ejes x e y desde el punto de vista de la cámara. La imagen tomada tendrá una apertura definida en grados por cada píxel en los dos ejes.

  - Requisito: **G.1.10.5.3**
    Las características descritas anteriormente podrán ser modificadas.

- Requisito: **G.1.10.6**
  Las cámaras de proyección ortográfica tienen las siguientes características:

  - Requisito: **G.1.10.6.1**
    Tienen definida una distancia máxima y mínima. Las entidades que estén más lejos que la distancia máxima, o más cerca que la distancia mínima a la cámara no serán dibujados por dicha cámara.

  - Requisito: **G.1.10.6.2**
    Tienen definido un ancho y un alto. La imagen tomada corresponderá al contenido del paralelogramo de ancho y alto indicado y de profundidad definida por la distancia máxima y mínima.

  - Requisito: **G.1.10.6.3**
    Las características descritas anteriormente podrán ser modificadas.

- Requisito: **G.1.10.7**
  Los iluminadores tienen las siguientes características:

  - Requisito: **G.1.10.7.1**
    Emiten luz ambiente. Esta luz alcanza a todos los objetos de la escena por igual.

  - Requisito: **G.1.10.7.2**
    Emiten luz difusa. Esta luz ilumina los objetos en función de cómo apunte el vector normal de cada punto hacia la dirección de la que provienen los rayos.

- Requisito: **G.1.10.7.3**
  Emiten luz especular. Esta luz imita los brillos y reflejos producidos por superficies pulidas o cromadas.

- Requisito: **G.1.10.7.4**
  Tanto la cantidad como el color de las diferentes clases definidas anteriormente pueden ser modificadas.

- Requisito: **G.1.10.7.5**
  La cantidad de luz que alcanza un punto será atenuada en función de la distancia a ese punto, y siguiendo factores lineales y cuadráticos. Los factores de atenuación podrán ser modificados.

- Requisito: **G.1.10.8**
  Los focos tienen un factor de concentración que permite reducir la potencia de la luz emitida en función de lo alejado que esté un rayo concreto del eje del cono de emisión.

- Requisito: **G.1.10.9**
  Será posible cambiar la malla asignada a los objetos geométricos.

- Requisito: **G.1.10.10**
  Los objetos estarán organizadas como un árbol. Para un determinado objeto, su posición, rotación y escala está determinado por su propia transformada y las transformadas de sus objetos antecesores en el árbol hasta llegar a la raíz.

- Requisito: **G.1.11**
  Las mallas que pueden utilizar los objetos geométricos tendrán las siguientes características:

  - Requisito: **G.1.11.1**
    Estarán compuestas de polígonos.

  - Requisito: **G.1.11.2**
    Cada polígono tendrá tres o más vértices.

  - Requisito: **G.1.11.3**
    Cada vértice tendrá una coordenada, una dirección normal, un color y cierta cantidad de coordenadas UV.

  - Requisito: **G.1.11.4**
    La coordenada determinará el punto en el que está situado el vértice.

- Requisito: **G.1.11.5**
  La dirección normal determinará la perpendicular del polígono en ese vértice. Será un vector unitario.

- Requisito: **G.1.11.6**
  El color determinará el tintado del vértice. Estará definido por un triplete RGB y un valor de transparencia.

- Requisito: **G.1.11.7**
  Las coordenadas UV determinarán el punto de la textura asociada al que corresponde el vértice.

- Requisito: **G.1.12**
  Cada polígono podrá tener asignadas varias capas de pintado. Cada capa tendrá asignado un material.

- Requisito: **G.1.13**
  Los materiales definirán el comportamiento de la superficie con respecto a la iluminación. Tendrán varios factores:

  - Requisito: **G.1.13.1**
    Sensibilidad a la iluminación ambiente: determinará cuánto le afecta la iluminación ambiente.

  - Requisito: **G.1.13.2**
    Sensibilidad a la iluminación difusa: determinará cuánto le afecta la iluminación difusa.

  - Requisito: **G.1.13.3**
    Sensibilidad a la iluminación especular: determinará cuánto le afecta la iluminación especular.

  - Requisito: **G.1.13.4**
    Pulido: determinará cómo de pulida está la superficie, y por tanto cómo de reflectiva es la superficie con respecto a la iluminación especular.

  - Requisito: **G.1.13.5**
    Emisión: simulará que el material emite luz.

  - Requisito: **G.1.13.6**
    Los factores de sensibilidad a la luz y el de emisión estarán definidos por un triplete RGB.

- Requisito: **G.1.13.7**
  Todos los factores descritos anteriormente podrán ser modificados.

- Requisito: **G.1.14**
  Los materiales podrán tener asignada una textura.

  - Requisito: **G.1.14.1**
    La textura será una imagen de mapa de bits que se pintará dentro del polígono.

  - Requisito: **G.1.14.2**
    La textura se aplicará al polígono siguiendo las asignaciones de coordenadas UV de los vértices del polígono.

- Requisito: **G.1.15**
  Para un determinado polígono, se definirán varios conjuntos de coordenadas UV, uno para cada capa.

- Requisito: **G.1.16**
  Inicialmente, el sistema utilizará mallas estáticas y sin animación. Pero estará pensado para poder ser extendido y soportar mallas con más características.

- Requisito: **G.1.17**
  El sistema proveerá mecanismos para cargar mallas de polígonos desde ficheros.

- Requisito: **G.1.18**
  El sistema proveerá mecanismos para cargar texturas para las mallas desde ficheros.

- Requisito: **L.1**
  El sistema desarrollado contendrá un mecanismo unificado para cargar objetos y entidades complejas desde ficheros. Dicho mecanismo recibirá el nombre de Gestor de Recursos.

  - Requisito: **L.1.1**
    El Gestor de Recursos organizará sus recursos en bibliotecas.

  - Requisito: **L.1.2**
    Cada biblioteca almacenará recursos de un mismo tipo, asignando un nombre distinto a cada recurso.

- Requisito: **L.1.3**
  Cada recurso tendrá mecanismos para ser cargado en memoria.

- Requisito: **L.1.4**
  Cada recurso tendrá mecanismos para ser descargado de memoria.

- Requisito: **L.1.5**
  Se podrá consultar el estado de un recurso.

- Requisito: **L.1.6**
  Si hubo algún problema al cargar un recurso, el sistema informará de dicho problema a través del estado del recurso.

- Requisito: **L.1.7**
  El uso de algunos recursos será destructivo. El sistema proveerá mecanismos para obtener copias del recurso que puedan ser modificadas sin consecuencias ni efectos secundarios.

- Requisito: **L.1.8**
  Las mallas de polígonos y las texturas del Motor Gráfico se cargarán utilizando el Gestor de Recursos.

- Requisito: **P.1**
  El sistema desarrollado contendrá una estructura básica para implementar las reglas del juego. Dará soporte para las siguientes características:

  - Requisito: **P.1.1**
    Creación de diferentes procesos concurrentes.

  - Requisito: **P.1.2**
    Los procesos tendrán un estado. Los estados posibles son:

    - Requisito: **P.1.2.1**
      Normal. En este estado, el proceso recibirá una ráfaga de CPU en cada fotograma.

    - Requisito: **P.1.2.2**
      Dormido. En este estado, el proceso recibirá una ráfaga de CPU cada ciertos fotogramas.

    - Requisito: **P.1.2.3**
      Congelado. En este estado, el proceso no recibirá ráfagas de CPU.

- Requisito: **P.1.3**
  Los procesos podrán mandarse señales unos a otros y a sí mismos. Las señales disponibles son:

    - Requisito: **P.1.3.1**
      Despertar. Pone el proceso receptor en estado normal.

    - Requisito: **P.1.3.2**
      Dormir. Pone el proceso receptor en estado dormido.

    - Requisito: **P.1.3.3**
      Congelar. Pone el proceso en estado congelado.

    - Requisito: **P.1.3.4**
      Matar. Destruye el proceso.

- Requisito: **P.1.4**
  Los procesos podrán reaccionar a la recepción de las señales.

- Requisito: **P.1.5**
  Los procesos tendrán una prioridad. La prioridad determinará el orden en el que reciben su ráfaga de CPU en cada fotograma. Si la prioridad de un proceso es mayor que la de otro, recibirá su ráfaga de CPU antes.

- Requisito: **S.1**
  El sistema se inicializará de la manera que vea más oportuna, y por último, instanciará un proceso concreto de las reglas del juego. Dicho proceso se llamará Proceso Inicial, y se encargará de inicializar las reglas del juego.

- Requisito: **S.2**
  En cada fotograma, el sistema ejecutará las reglas del juego y después llamará al Motor Gráfico para mostrar el resultado en pantalla.

- Requisito: **S.3**
  El sistema continuará generando fotogramas sin parar hasta que las reglas del juego le ordenen terminar.

2.3. Modelo estructural

El sistema a construir está fuertemente definido por los requisitos, y está compuesto por varios bloques diferentes. Su estructura general está definida de una manera similar al patrón Modelo-Vista-Controlador.

2.3.1. ControlGeneral

Es el controlador en el patrón MVC. Llama a los diferentes componentes del sistema, y recibe los eventos del jugador desde el Sistema Operativo

Dependencias

- **SistemaOperativo**: Recibe los eventos del jugador a través del Sistema Operativo.
- **LogicaDelJuego:EjecutarMotor**: Ordena a la Lógica del Juego aplicar las normas del juego para producir el siguiente fotograma.
- **LogicaDelJuego:CrearProcesoInicial**: Crea el primer proceso que inicializa la Lógica del Juego.
- **MotorGráfico:EjecutarMotor**: Ordena al Motor Gráfico dibujar cada fotograma.

2.3.2. Motor Gráfico

Contiene los mecanismos que utiliza el videojuego para representar imágenes.

Interfaces ofrecidas

- **EjecutarMotor**: Lanza el pintado de un fotograma. El Motor Gráfico recopila el estado general de la escena y lo usa para pintar el fotograma actual.
- **Geometría**: Ofrece accesos y mecanismos para crear y modificar mallas de polígonos y sus componentes.
- **Objetos**: Ofrece accesos y mecanismos para crear y modificar árboles de objetos.
- **Pantalla**: Ofrece accesos y mecanismos para crear y modificar escenas, así como para asignarlas a pantallas.

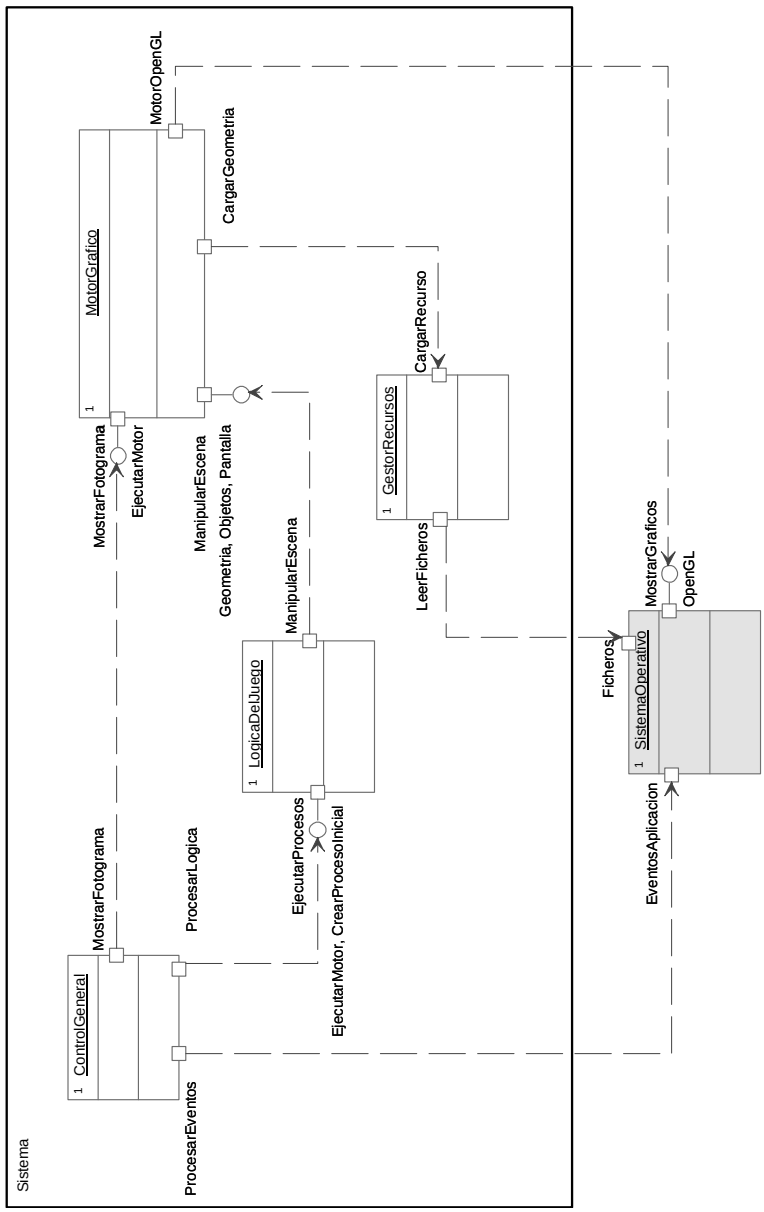

Figura 2.1: Diagrama estructural del sistema

Dependencias

- **SistemaOperativo:OpenGL**: Utiliza la biblioteca OpenGL para dibujar los gráficos.
- **SistemaOperativo**: Utiliza las funciones del Sistema Operativo para manipular el entorno de ventanas.
- **GestorRecursos**: Utiliza el Gestor de Recursos para cargar las mallas y texturas necesarias.

2.3.3. Gestor de Recursos

Implementa mecanismos para organizar y cargar recursos. Proporciona parte de la persistencia del sistema.

Interfaces ofrecidas

- **CargarRecurso**: Ofrece una estructura básica y una organización genérica para manipular recursos.

Dependencias

- **SistemaOperativo:Ficheros**: Utiliza la interfaz del Sistema Operativo encargada de abrir y leer ficheros.

2.3.4. Lógica del Juego

Implementa las reglas de un juego concreto, así como su organización en procesos.

Interfaces ofrecidas

- **EjecutarMotor**: Ejecuta la Lógica del Juego.
- **CrearProcesoInicial**: Instancia el Proceso Inicial.

Dependencias

- **MotorGráfico:Geometría**, **MotorGráfico:Objetos**, **MotorGráfico:Pantalla**: Utiliza las interfaces del Motor Gráfico para construir la imagen que se mostrará en pantalla.

2.3.5. SistemaOperativo

Bajo este nombre se han reunido todas las interfaces y bibliotecas externas a este sistema.

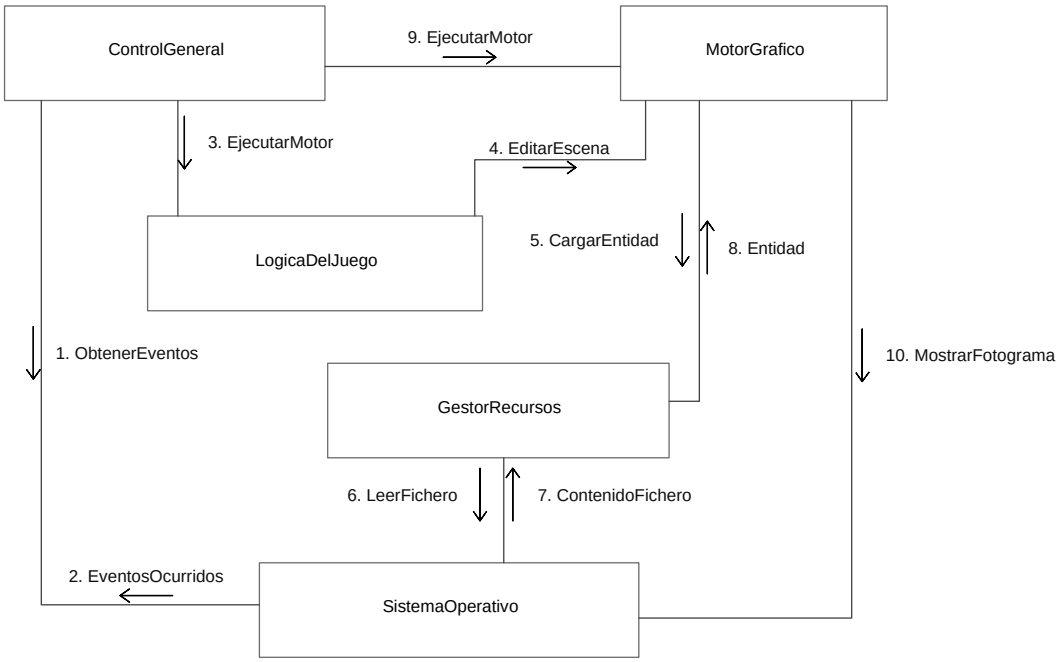

Figura 2.2: Diagrama del flujo de información general del sistema

Interfaces ofrecidas

- **OpenGL**: Ofrece primitivas básicas de dibujado.
- **Ficheros**: Ofrece mecanismos básicos para la lectura y escritura de ficheros.

2.3.6. Flujo de información general

El sistema implementa la estructura general de una aplicación interactiva, con ciertos detalles orientados a la construcción de videojuegos. El sistema se ejecuta en un bucle, de tal manera que en cada ciclo ocurren los siguientes pasos:

1. **ObtenerEventos()** El Controlador General pide al Sistema Operativo los eventos ocurridos desde el último fotograma, tales como movimientos del ratón o pulsaciones de teclas.
2. **EventosOcurridos** El Sistema Operativo devuelve al Controlador General el listado de eventos ocurridos desde el último fotograma.
3. **EjecutarMotor()** El Controlador General ordena aplicar las Reglas del Juego al fotograma actual para producir el fotograma siguiente.

4. **EditarEscena()** Las Reglas del Juego modifican la escena que mostrará el Motor Gráfico, y la preparan para el siguiente fotograma.

5. **CargarEntidad()** A veces es necesario cargar objetos de disco. El Motor Gráfico utiliza el Gestor de Recursos para esta tarea.

6. **LeerFichero()** El Gestor de Recursos lee los ficheros necesarios utilizando el Sistema Operativo.

7. **ContenidoFichero** El Sistema Operativo devuelve al Gestor de Recursos el contenido de los ficheros pedidos.

8. **Entidad** El Gestor de Recursos devuelve al Motor Gráfico la entidad pedida.

9. **EjecutarMotor()** El Controlador General pide al Motor Gráfico pintar la escena.

10. **MostrarFotograma()** El Motor Gráfico utiliza OpenGL para generar y mostrar la imagen al jugador.

De esta manera, en cada ciclo el jugador ve la imagen producida por el videojuego, realiza acciones (que se traducen en eventos de entrada), el videojuego reacciona a dichas acciones y produce la siguiente imagen.

2.4. Definición detallada de interfaces

2.4.1. MotorGráfico:EjecutarMotor

Esta interfaz tiene una única función que consiste en lanzar el pintado de un fotograma. Está compuesta de un único método:

- **ejecutar()**: Ordena pintar el fotograma actual y enviarlo a la pantalla.

2.4.2. MotorGráfico:Geometría

Esta interfaz proporciona acceso a las mallas de polígonos y sus componentes, siguiendo los requisitos especificados. Se ha descrito como una colección de clases, sus relaciones, atributos y operaciones asociados. El uso de esta interfaz consiste en la creación y configuración de instancias de las clases disponibles.

2.4.3. MotorGráfico:Objetos

Esta interfaz proporciona acceso a construcción de escenas, situando las cámaras, iluminadores y objetos en los diferentes lugares, siguiendo los requisitos especificados. Se ha descrito como una colección de clases, sus relaciones, atributos y operaciones asociados. El uso de esta interfaz consiste en la creación y configuración de instancias de las clases disponibles.

2.4.4. MotorGráfico:Pantalla

Esta interfaz proporciona acceso a la asignación de escenas y capas de pintado a las pantallas disponibles en el sistema, siguiendo los requisitos especificados. Se ha descrito como una colección de clases, sus relaciones, atributos y operaciones asociados. El uso de esta interfaz consiste en la creación y configuración de instancias de las clases disponibles.

2.4.5. LógicaDelJuego:Proceso

Esta interfaz proporciona un modelo básico de proceso a partir del cual se pueden definir las Reglas del Juego. El uso consiste en la extensión del modelo básico de proceso a procesos concretos que hagan tareas específicas. El ProcesoInicial es un caso concreto de uso del interfaz, ya que modela en un proceso el método de inicialización de las Reglas del Juego.

2.4.6. LógicaDelJuego:EjecutarMotor

Esta interfaz proporciona mecanismos para ejecutar las Reglas del Juego. Está compuesta por dos operaciones:

- **ejecutar_procesos()**: Llama al método *ejecutar* de todos los procesos despiertos por orden de prioridad.
- **ejecutar_procesos_dormidos()**: Llama al método *ejecutar_dormid* de todos los procesos dormidos por orden de prioridad.

2.4.7. LógicaDelJuego:CrearProcesoInicial

Esta interfaz proporciona un mecanismo para inicializar las Reglas del Juego. Su uso consiste en la instanciación de ProcesoInicial y posterior agregación al Motor de Procesos. Una vez creado dicho proceso, en su operación *ejecutar* debe completar la inicialización de las Reglas del Juego.

2.4.8. GestorRecursos:CargarRecurso

Esta interfaz proporciona un mecanismo para agilizar la implementación de un sistema de carga de entidades de disco. Su uso consiste en la extensión del modelo básico de *Cargador* e *Instanciador*, y posterior uso de una instancia de GestorRecursos.

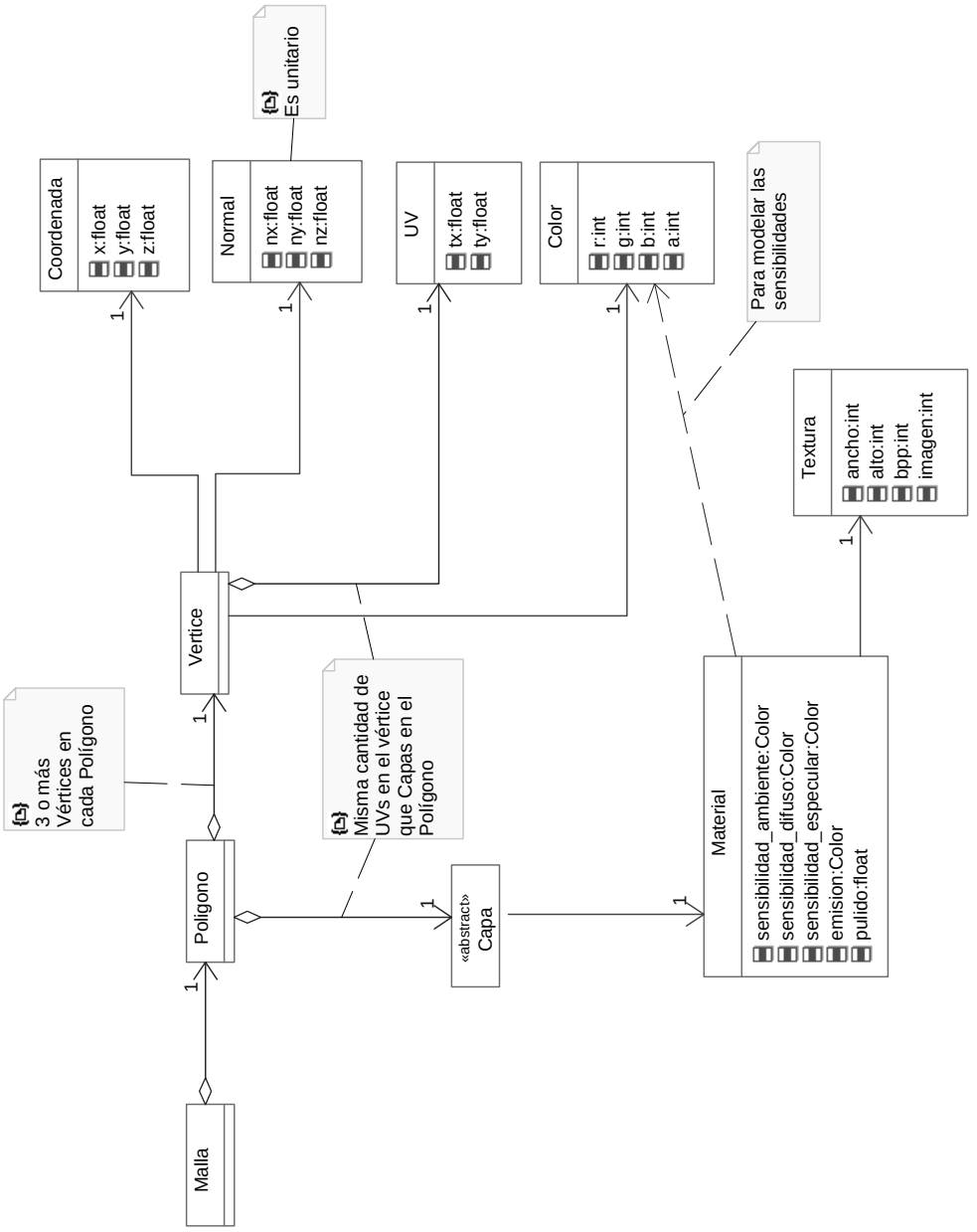

Figura 2.3: Diagrama de Clases del Interfaz de Geometría del Motor Gráfico

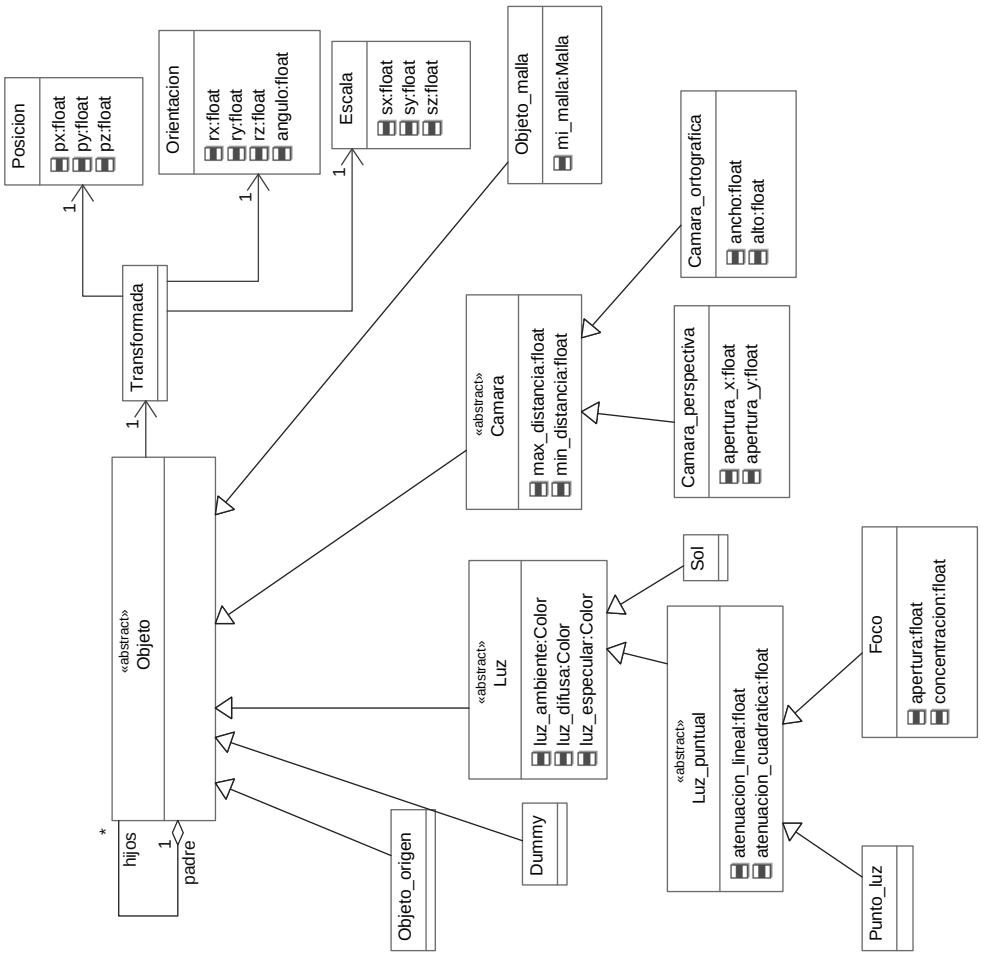

Figura 2.4: Diagrama de Clases del Interfaz de Objetos del Motor Gráfico

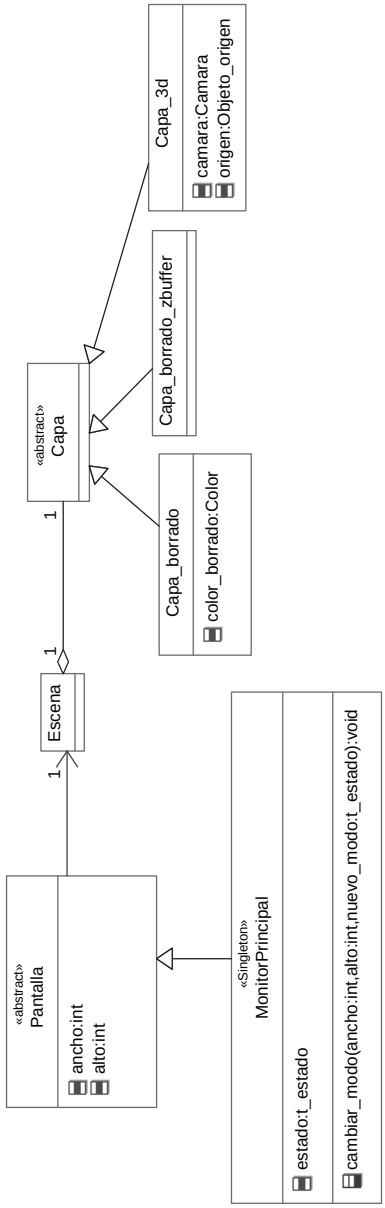

Figura 2.5: Diagrama de Clases del Interfaz de Pantalla del Motor Gráfico

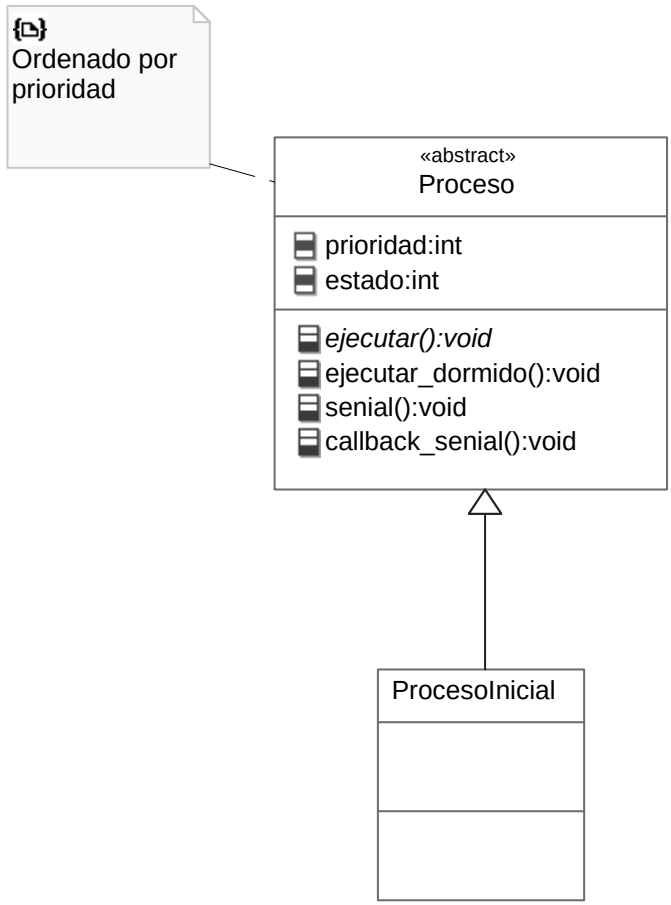

Figura 2.6: Diagrama de Clases del Modelo de Proceso

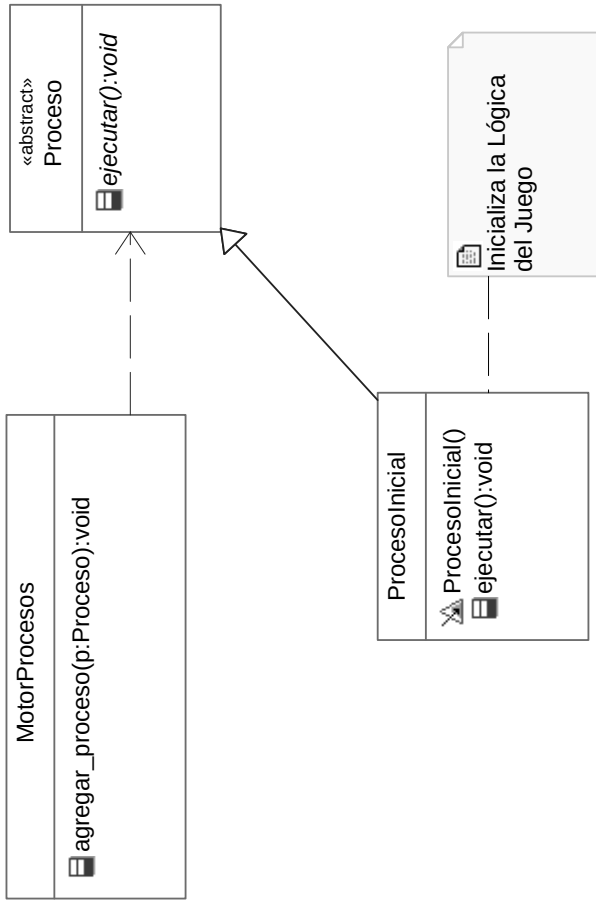

Figura 2.7: Diagrama de Clases del Interfaz del Motor de Procesos, en su uso para inicializar las Reglas del Juego

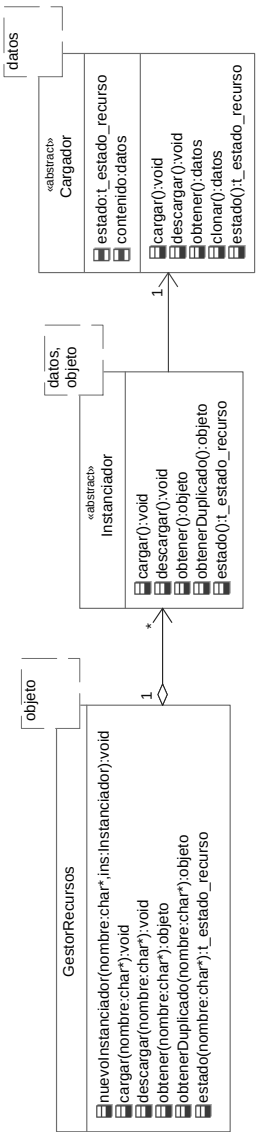

Figura 2.8: Diagrama de Clases del Interfaz del Gestor de Recursos

Capítulo 3

Diseño del Motor Gráfico

3.1. Modelo Estructural y Arquitectónico

El Motor Gráfico desempeña la tarea de proveer un mecanismo a la Lógica del Juego para mostrar imágenes al jugador. Por su ámbito y requisitos, se trata de un subsistema complejo, y como tal, ha sido dividido en varias partes. Se distinguen dos bloques principales: las interfaces, pensadas para construir la imagen; y el Motor OpenGL, pensado para dibujar la imagen. Esta división responde al requisito **G.1.6**, que exige separación entre el modelo de imagen y el mecanismo de dibujado.

3.1.1. Modelo general

En el diagrama estructural se muestra el concepto básico de diseño del Motor Gráfico. Esencialmente, hay una serie de clases genéricas (definidas por los subsistemas Pantalla, Objetos y Geometría) que son los que reciben las órdenes y son manipulados por otros sistemas; y hay un Motor OpenGL, que se encarga de aplicar las clases genéricas a un entorno de dibujado basado en OpenGL.

3.1.2. Geometría

Contiene las clases necesarias para construir mallas de polígonos.

Interfaces ofrecidas

- **ManipularEscena:Geometría**: Ofrece mecanismos para crear y modificar mallas de polígonos.
- **Acceso interno desde el Motor OpenGL**: Ofrece al Motor OpenGL un acceso para la lectura de entidades.
- **Acceso a Malla desde Objetos**: Ofrece al subsistema de Objetos la posibilidad de utilizar Mallas.

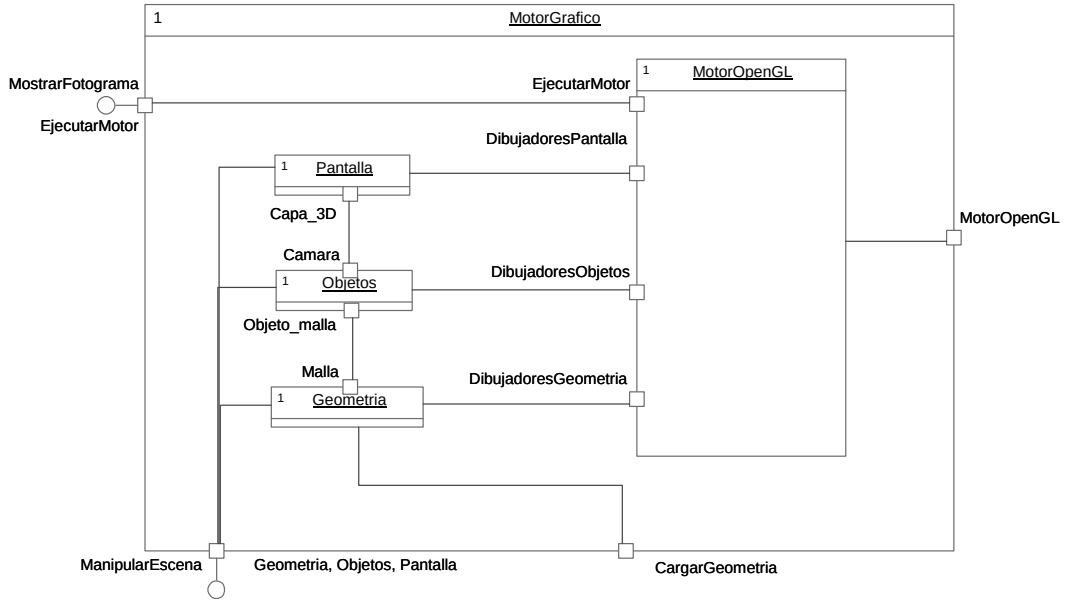

Figura 3.1: Diagrama estructural del Motor Gráfico

Interfaces utilizadas

- **GestorRecursos**: Utiliza el Gestor de Recursos para cargar las mallas y bitmaps requeridos.

3.1.3. Objetos

Contiene las clases necesarias para construir una escena como un árbol de objetos relacionados entre sí.

Interfaces ofrecidas

- **ManipularEscena:Objetos**: Ofrece mecanismos para crear y modificar árboles de objetos y sus componentes.
- **Acceso interno desde el Motor OpenGL**: Ofrece al Motor OpenGL un acceso para la lectura de entidades.
- **Acceso a Objeto_origen y *Camara* desde Pantalla**: Ofrece al subsistema de Pantalla la posibilidad de utilizar Objeto_origen y *Camara*.

Interfaces utilizadas

- **Geometría::Malla**: Utiliza las mallas del Nivel de Geometría.

3.1.4. Pantalla

Contiene las clases necesarias para organizar escenas como composición de capas, y asignárselas a las pantallas disponibles.

Interfaces ofrecidas

- **ManipularEscena:Pantalla**: Ofrece mecanismos para crear y modificar escenas.
- **Acceso interno desde el Motor OpenGL**: Ofrece al Motor OpenGL un acceso para la lectura de entidades.
- **Acceso a Objeto_origen y *Camara* desde Pantalla**: Ofrece al subsistema de Pantalla la posibilidad de utilizar Objeto_origen y *Camara*.

Interfaces utilizadas

- **Objetos::Camara, Objetos::Objeto_origen**: Utiliza el objeto origen y las cámaras para definir el contenido de una **Capa3d**.

3.1.5. Motor OpenGL

Contiene los mecanismos y clases para dibujar en pantalla las entidades definidas en los subsistemas de Geometría, Objetos y Pantalla utilizando las bibliotecas SDL y OpenGL.

Interfaces ofrecidas

- **EjecutarMotor**: Ofrece mecanismos para inicializar, terminar y mostrar fotogramas.

Interfaces utilizadas

- **Geometría, Objetos, Pantalla**: Accede al contenido de las instancias para dibujarlas en pantalla.
- **SDL, OpenGL**: Utiliza SDL y OpenGL para dibujar los gráficos.

3.2. Diseño detallado

Para la implementación del proyecto se ha decidido utilizar el lenguaje de programación OCaML. Este lenguaje tiene algunas características interesantes:

- **Estáticamente tipado**: Reduce significativamente las probabilidades de errores generales.
- **Alta velocidad**: Debido a la característica anterior y a que se trata de un lenguaje de alto nivel, el compilador puede asumir muchas cosas. Esto permite optimizar significativamente el rendimiento en tiempo de ejecución.

- **Multiplataforma**: El entorno de compilación puede producir binarios para muchas plataformas diferentes.
- **Imposible usar sin haber declarado y construido antes**: Para una determinada instrucción, todos los operandos deben haber sido declarados y asignados antes, lo cual significa que a efectos prácticos es casi imposible construir dependencias circulares.

Pero estas características nos imponen ciertos inconvenientes. Los más significativos, y que han impuesto ciertas restricciones en el diseño son los siguientes:

- **Todos los atributos son privados**: No hay manera de declarar un atributo como protegido o público, lo cual significa que todos aquellos atributos que conceptualmente sean públicos deberán tener métodos de acceso (Getter/Setter o similares).
- **Imposible hacer downcasting**: No es posible tratar de determinar si una instancia concreta es de un subtipo del declarado. Por ello, cuando es necesario acceder a una instancia de clase correspondiente a un subtipo, hay que hacerlo a través de métodos de búsqueda y/o asociadores.
- **Variables estáticas de módulo**: Durante la declaración de un módulo, es posible asignar valores a variables globales dentro del módulo. Esto se ha utilizado para construir las clases con estereotipo «singleton». Se ha asignado a una variable del módulo una instancia de la clase especificada.

Para acceder a las funciones del sistema de ventanas y la lectura de eventos de entrada, también es necesario el uso de una biblioteca, ya que OCaML no trae directamente primitivas para estas tareas. En este proyecto se ha decidido utilizar SDL, que tiene las características de ser multiplataforma y de código abierto. SDL tiene otras funciones y subbibliotecas, de las cuales se ha utilizado SDLImage para cargar imágenes de disco.

Puesto que desde OCaML no se puede acceder directamente a bibliotecas externas, es necesario el uso de asignaciones o *bindings*. Estas entidades conectan el lenguaje de programación con las bibliotecas concretas, permitiendo hacer llamadas y recibir resultados. En este proyecto se han utilizado los siguientes *bindings*:

- **OcamlSDL**: Para acceder a las bibliotecas SDL.
- **LablGL**: Para acceder a OpenGL.

Como consecuencia de las características anteriores, los diagramas tienen ciertos detalles que deben ser considerados durante su interpretación:

- **Donde hay atributos públicos, en realidad hay getters y/o setters**: En los diagramas se muestran como atributos públicos, pero en realidad son atributos privados con sus métodos asociados para asignar el valor. Dichos getters y setters no se muestran en los diagramas, excepto cuando su función es más compleja que la de un getter o setter.
- **Scaffolding oculto**: Al igual que los getters y setters, los métodos para acceder a las clases relacionadas han sido ocultados de los diagramas.

- **Multiples vistas de una clase**: En ocasiones, una clase está mostrada en varios diagramas, ya que se pretende mostrar diferentes aspectos de la misma. Cuando ocurra eso, las características globales de la clase serán la unión de las características mostradas en cada diagrama.

3.2.1. Geometría

El nivel de Geometría estaba compuesto por clases abstractas, que proporcionan un modelo muy básico. Para poder utilizar dichas clases es necesario crear modelos instanciables de las mismas, y siguiendo el requisito **G.1.16** se ha construido un conjunto de clases que implementan un modelo básico de malla: la **Malla_simple**. La característica principal de la **Malla_simple** es que es inmutable, es decir, una **Malla_simple** o cualquiera de sus componentes no puede ser modificado o reconfigurado una vez creada.

Puesto que la **Malla_simple** está compuesta de otros objetos, que a su vez están compuestos de otros sucesivamente, se propone un mecanismo avanzado para su instanciación, utilizando el Gestor de Recursos. Este mecanismo se encargará de leer ficheros y traducir su contenido a instancias de **Malla_simple**.

La **Malla_simple** utiliza instancias de **Bitmap** como texturas. Para hacer completa la instanciación de **Malla_simple**, es necesario instanciar también algún tipo de **Bitmap**. Por esa razón, también se ha creado un mecanismo que lee ficheros y construye **Bitmap_RGBA** a partir de los mismos.

Estructuras de datos

En los diagramas de clases se observan las relaciones entre ***Malla*** y **Malla_simple**, así como la estructura y organización de los cargadores.

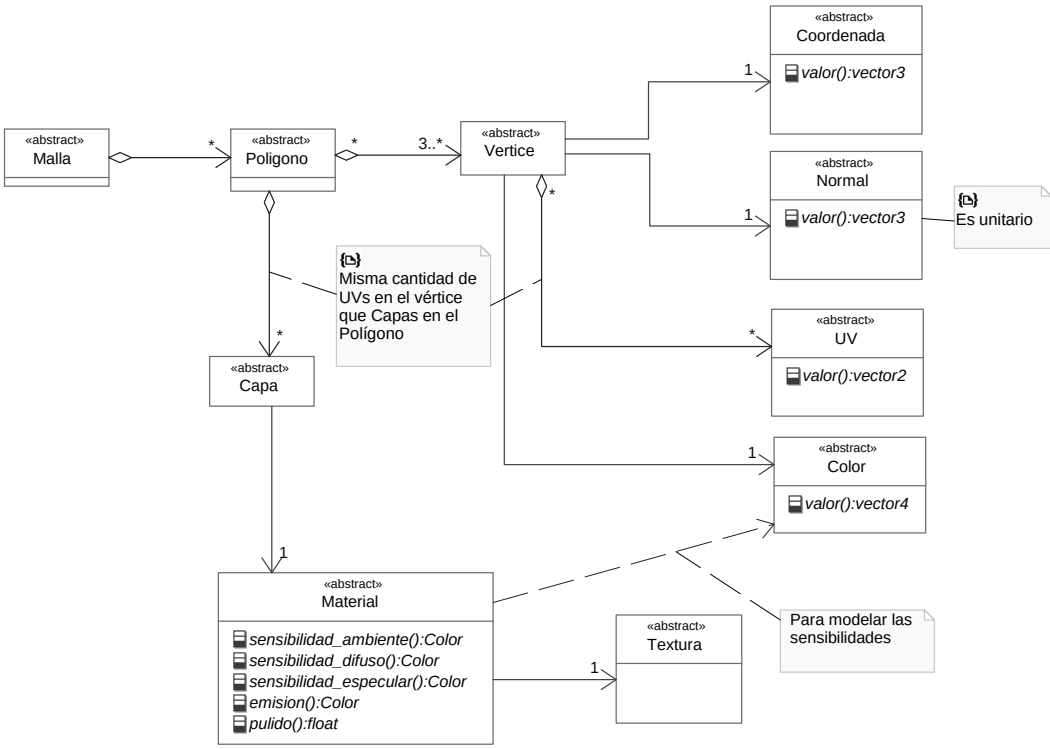

Figura 3.2: Diagrama de las clases abstractas que componen el Nivel de Geometría del Motor Gráfico

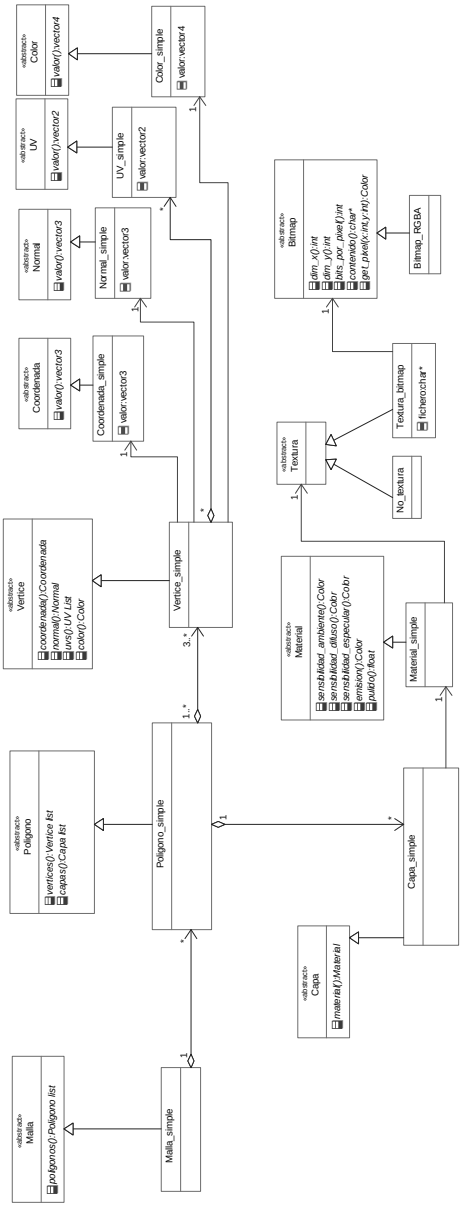

Figura 3.3: Diagrama de las clases que componen la **Malla_simple** del Nivel de Geometría del Motor Gráfico

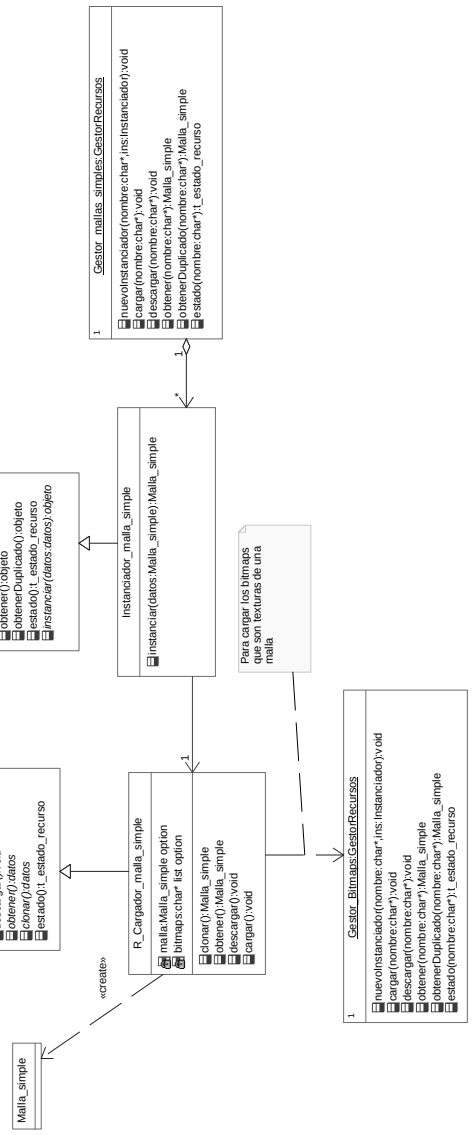

Figura 3.4: Diagrama de las clases que componen el cargador de **Malla_simple** del Nivel de Geometría del Motor Gráfico

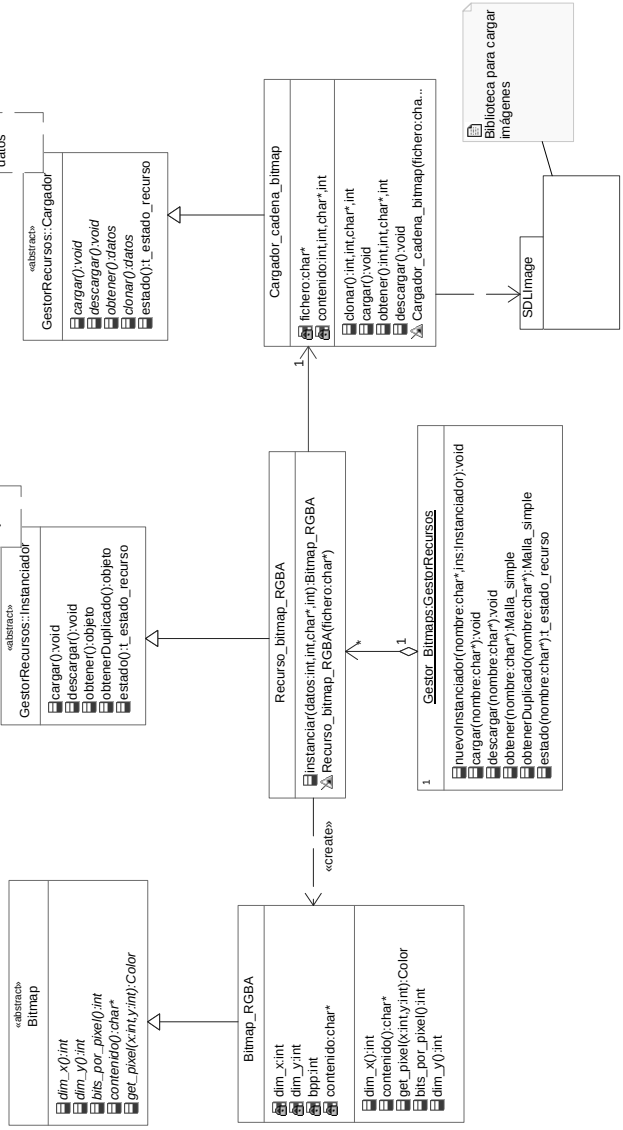

Figura 3.5: Diagrama de las clases que componen el cargador de **Bitmap_RGBA** del Nivel de Geometría del Motor Gráfico

Algoritmos

La principal funcionalidad avanzada de este nivel corresponde a dos funciones de carga, que leen ficheros y los transforman en instancias de **Malla_simple** y de **Bitmap_RGBA**.

La carga de **Malla_simple** es compleja debido a la propia complejidad de **Malla_simple**. Para resolver esta tarea se ha creado un formato de fichero específico, y a partir de ese formato se ha construido un lexer y un parser que hacen la tarea. El comportamiento general es el siguiente:

1. El cliente llama a la operación *cargar* del Gestor_mallas_simples con el nombre concreto de malla.
2. El Gestor_mallas_simples llama a la operación *cargar* del **Instanciador_malla_simple** correspondiente.
3. El **Instanciador_malla_simple** llama al método *cargar* de su **R_Cargador_malla_simple**.
4. El **R_Cargador_malla_simple** abre su fichero.
5. El **R_Cargador_malla_simple** llama al lexer/parser, y le pasa como parámetro el descriptor del fichero.
6. El lexer/parser procesa el fichero.
7. Si durante el procesado encuentra descripciones de materiales que requieran textura, añade un nuevo cargador de bitmap correspondiente al Gestor_Bitmaps.
8. Si se encontró algún problema en el fichero, el lexer/parser falla con una excepción *Parsing.Parse_error*
9. El lexer/parser instancia la **Malla_simple** y todas las clases que necesite (**Poligono_simple**, **Vertice_simple**, **Capa_simple**, etc).
10. El lexer/parser devuelve la **Malla_simple** y una lista de los nombres de los bitmaps que encontró durante la lectura del fichero.
11. Para cada nombre de bitmap recibido del lexer/parser, el El **R_Cargador_malla_simple** llama al método *cargar* del Gestor_Bitmaps pasándole como parámetro el nombre del bitmap.
12. La carga de la **Malla_simple** finaliza.

Tras este proceso, puede obtenerse la **Malla_simple** por los métodos normales de un **Gestor_recursos**.

La carga de **Bitmap_RGBA** es relativamente sencilla, ya que la biblioteca SDLImage hace gran parte del trabajo. El comportamiento general es el siguiente:

1. El cliente llama a la operación *cargar* del Gestor_bitmaps con el nombre concreto del bitmap.
2. El Gestor_bitmaps llama a la operación *cargar* del **Recurso_bitmap_RGBA** correspondiente.

3. El **Recurso_bitmap_RGBA** llama al método *cargar* de su **Cargador_cadena_bitmap**.
4. El **Cargador_cadena_bitmap** llama a SDL y le ordena cargar el fichero como imagen.
5. El **Cargador_cadena_bitmap** extrae los píxeles de la imagen:
    6. Si la imagen es de 32 bits por píxel, **Cargador_cadena_bitmap** reorganiza los colores para que estén en orden RGBA, ocupando cada componente un byte.
    7. Si la imagen es de 24 bits por píxel, **Cargador_cadena_bitmap** reorganiza los colores para que estén en orden RGB, ocupando cada componente un byte.
8. La carga del recurso finaliza.

La posterior instanciación es sencilla, y sigue el patrón general del Gestor de Recursos:

1. El cliente llama a la operación *obtener* del Gestor_bitmaps con el nombre concreto del bitmap.
2. El Gestor_bitmaps llama a la operación *obtener* del **Recurso_bitmap_RGBA** correspondiente.
3. El **Recurso_bitmap_RGBA** llama al método *obtener* de su **Cargador_cadena_bitmap**.
4. El **Cargador_cadena_bitmap** devuelve la cadena de caracteres que contiene los píxeles de la imagen, junto a la descripción de ancho, alto y profundidad de color.
5. El **Recurso_bitmap_RGBA** llama a su método *instanciar* y le pasa como parámetro los datos recibidos de **Cargador_cadena_bitmap**.
6. El **Recurso_bitmap_RGBA** crea una instancia de **Bitmap_RGBA** con las características especificadas.
7. El **Recurso_bitmap_RGBA** devuelve la instancia creada.
8. El Gestor_bitmaps devuelve la instancia al cliente.

3.2.2. Objetos

El nivel de Objetos es similar a lo visto en el interfaz, pero añade algunas funcionalidades a ciertos objetos, con el fin de manipularlos mejor. Dichas funcionalidades son:

- Posición:
  - **mover**: suma el vector especificado a la posición.
- Orientación:
  - **rotar**: aplica una rotación sucesiva a la orientación.
  - **frontal**: devuelve el resultado de rotar el vector frontal.
  - **vertical**: devuelve el resultado de rotar el vector vertical.
- Escala:
  - **escalar**: multiplica la escala por vector especificado.
- Objeto:
  - **emparentar**: convierte el objeto especificado en el padre del objeto actual.
  - **origen**: devuelve el objeto origen del árbol de objetos.
  - **transformada_final**: aplica todas las transformadas que afectan al objeto, y las devuelve como una transformada.
- Objeto_origen:
  - **origen**: se devuelve a sí mismo.

También se han cambiado algunas cosas con respecto a las interfaces vistas, siendo el cambio más importante en las clases **Posición**, **Orientación** y **Escala**, que ahora están orientadas al uso de vectores, en vez de trabajar con las componentes por separado.

Estructuras de datos

En el diagrama de clases se aprecia la organización definitiva del nivel de Objetos del Motor Gráfico.

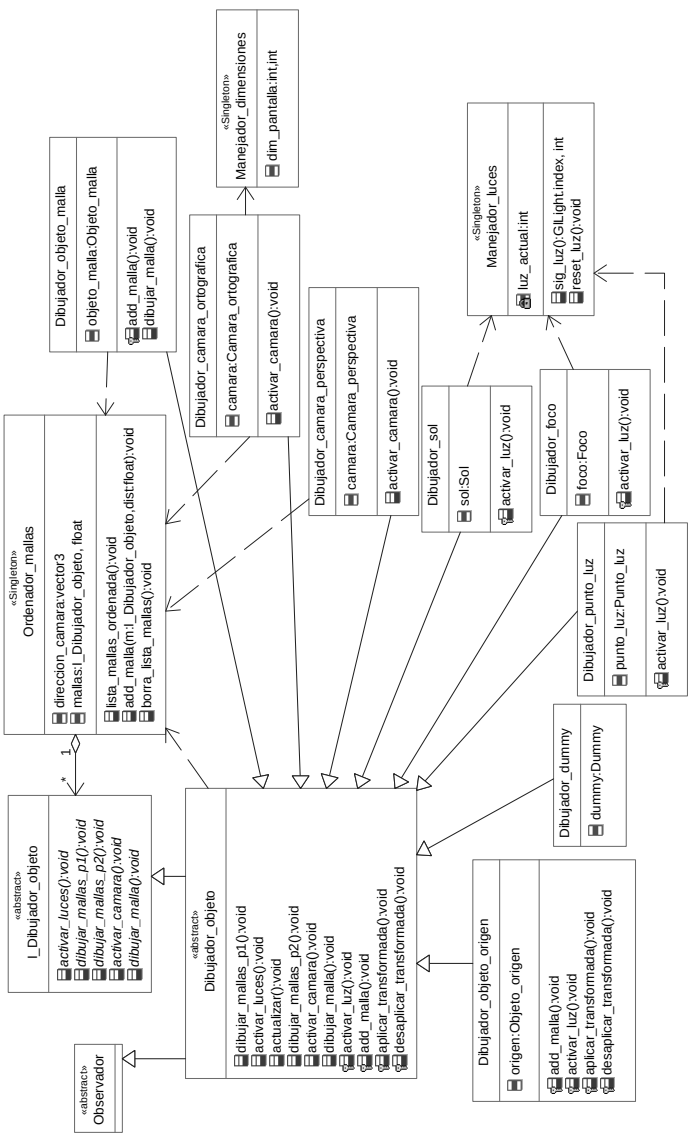

Figura 3.6: Diagrama de las clases que componen la revisión del nivel de Objetos del Motor Gráfico

Algoritmos

La funcionalidad más relevante de este nivel corresponde a la operación *emparentar*. En dicha operación se asigna un objeto como padre de otro, y se corrigen adecuadamente las dependencias.

La operación *emparentar* está compuesta de dos partes: la desasignación del padre anterior y la asignación al nuevo padre.

1. El cliente ordena a **a:Objeto** emparentar con **b:Objeto**.
2. Si **a:Objeto** tiene un padre asignado:
   3. **a:Objeto** llama a *quitar_hijo* de su objeto padre, y se pasa a sí mismo como parámetro.
4. **a:Objeto** asigna como su padre a **b:Objeto**.
5. **a:Objeto** llama a *añadir_hijo* de **b:Objeto**, y se pasa a sí mismo como parámetro.

Al terminar la interacción el árbol de objetos ha sido modificado adecuadamente, y **b:Objeto** es el nuevo padre de **a:Objeto**.

3.2.3. Pantalla

El nivel de Pantalla es similar al interfaz, pero complementa algunos puntos.

Estructuras de datos

En el diagrama de clases se aprecia la organización definitiva del nivel de Pantalla del Motor Gráfico.

Figura 3.7: Diagrama de las clases que componen la revisión del nivel de Pantalla del Motor Gráfico

3.2.4. Motor OpenGL

El Motor OpenGL agrupa y esconde todos los detalles que son necesarios para que la escena construida utilizando los niveles de Geometría, Objetos y Pantalla puedan ser dibujados utilizando la biblioteca OpenGL. Este motor fue ideado pensando en todo momento en que fuera lo más independiente y compacto posible. De esta manera es posible desconectarlo y cambiarlo por otro diferente.

OpenGL tiene unas características concretas que hacen que el diseño del Motor OpenGL siga cierta organización. Algunas de dichas características son:

- **Máquina de estados**: OpenGL funciona como una máquina de estados. Para pintar un polígono primero hay que establecer el estado de todas las características globales y locales, tales como texturas, funciones de blending, posición del observador, luces y matrices de transformada.
- **Muchas llamadas**: cada cambio de estado y cada primitiva que aplicar significa una o varias llamadas a OpenGL. Al final, inevitablemente, el conjunto de llamadas necesarias crece muy rápidamente. Para mitigar este problema, las sucesivas versiones de OpenGL han añadido diferentes mecanismos para condensar grupos de llamadas similares o muy relacionadas en unas pocas.

Observador-dibujador

El mecanismo principal que se utiliza para conectar el Motor OpenGL con el resto del sistema es el patrón Observador-Dibujador. Este patrón tiene la finalidad de controlar las instancias de una clase (el dibujable) y crear instancias de otra automáticamente (el dibujador). De esta manera se pretende que el Motor OpenGL tenga un inventario adecuado de las clases que tiene que dibujar en pantalla. Las clases que lo componen son:

- **C_Observado**: La clase dibujable que pretende ser vigilada por el patrón.
- **Observado**: Variante de la clase **C_Observado** que implementa llamadas a **Contador_observado** cuando es instanciada o destruída.
- **Contador_observado**: Agrega todas las instancias de **C_observado** y atiende sus avisos de instanciación o destrucción. Es el sujeto en el patrón Observador.
- **Vigilante_observado**: Recibe las notificaciones del **Contador_observado** e instancia o destruye el **Dibujador** correspondiente. Es el observador enel patrón Observador.
- **Dibujador**: Implementa los mecanismos concretos para dibujar el **C_Observado**.

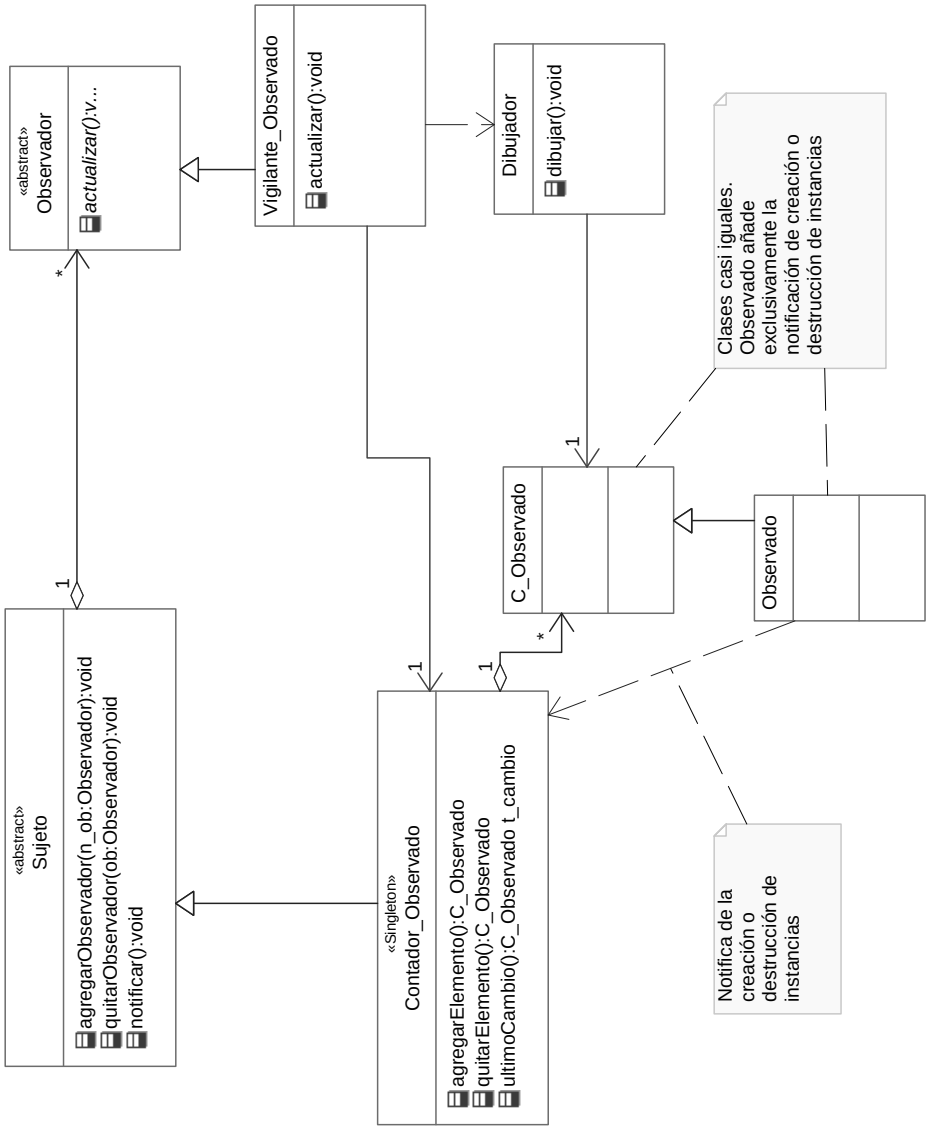

Figura 3.8: Diagrama de las clases que componen el patrón Observador-Dibujador

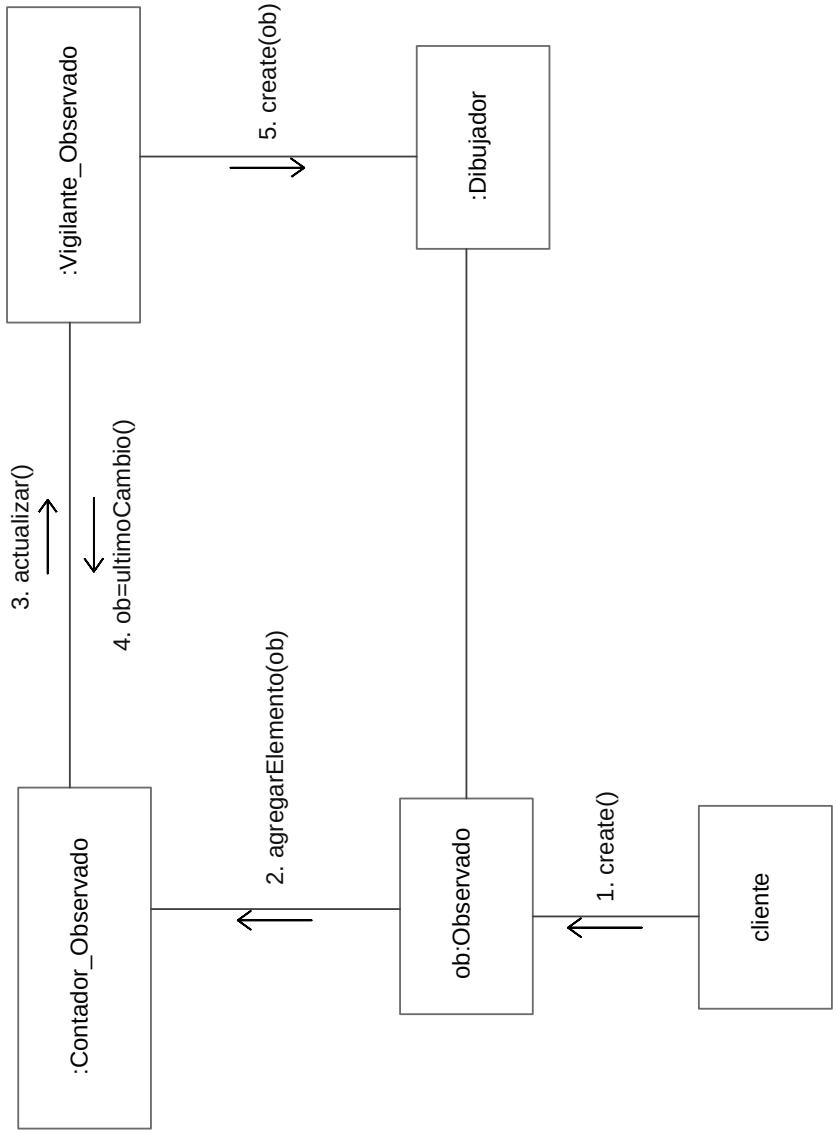

Figura 3.9: Diagrama de Colaboración de los eventos que ocurren al crear una instancia de **Observado**

El proceso general que ocurre en una instanciación es el siguiente:

1. El cliente crea una instancia de **Observado**.
2. El **Observado** envía a **Contador_Observado** una notificación de su creación.
   El **Contador_Observado** agrega la instancia de **Observado**.
3. El **Contador_Observado** llama a su método *notificar*, siguiendo el patrón Observador.
   El método *notificar* llama al método *actualizar* de **Vigilante_Observado**.
4. El **Vigilante_Observado** consulta a **Contador_Observado** el último cambio ocurrido.
   Contador_Observado devuelve la información de que se ha creado un **Observado**.
5. El **Vigilante_Observado** crea una instancia de **Dibujador** para el **Observado**.

En una destrucción, el proceso general es similar:

1. El cliente destruye una instancia de **Observado**.
2. El **Observado** envía a **Contador_Observado** una notificación de su destrucción.
   El **Contador_Observado** desagrega la instancia de **Observado**.
3. El **Contador_Observado** llama a su método *notificar*, siguiendo el patrón Observador.
   El método *notificar* llama al método *actualizar* de **Vigilante_Observado**.
4. El **Vigilante_Observado** consulta a **Contador_Observado** el último cambio ocurrido.
   Contador_Observado devuelve la información de que se ha destruido un **Observado**.
5. El **Vigilante_Observado** destruye la instancia de **Dibujador** correspondiente al **Observado**.

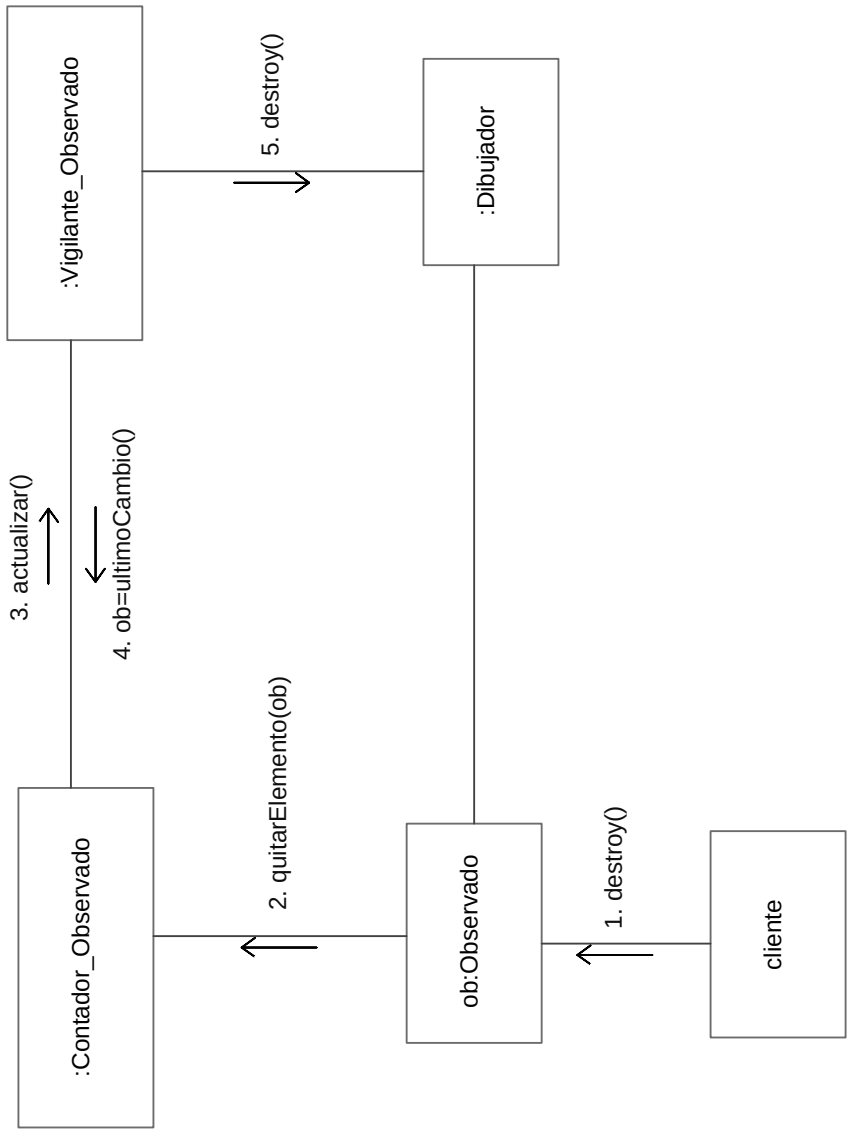

Figura 3.10: Diagrama de Colaboración de los eventos que ocurren al destruir una instancia de **Observado**

Manejo general de instancias

El modelo de organización basado en Observador-Dibujador tiene un inconveniente. Debido a que el Dibujable no conoce a su Dibujador es difícil encontrar el Dibujador correspondiente a un cierto Dibujable. Esto ocurre, por ejemplo, en el Dibujador_objeto_malla. Dibujador_objeto_malla conoce a su Objeto_malla, que a su vez conoce a su Malla_simple correspondiente. Pero no hay ninguna manera de encontrar el Dibujador_malla_simple correspondiente a la Malla_simple. Es necesario encontrar dicho Dibujador, puesto que será llamado para dibujar la ***Malla_simple***. Por ello, se crean los Asociadores. Estas clases se encargan de conectar dentro del Motor OpenGL cada Dibujable con su Dibujador correspondiente.

Los diagramas de ejemplo muestran la conexión real entre el **Dibujador_objeto_malla** y el ***Dibujador_malla*** correspondiente. Para que el patrón sea posible, es necesario que el **Asociador_mallas_dibujadores** tenga ciertas características:

- Conecta instancias de ***Malla_simple*** a ***Dibujador_malla***, de tal manera que a cada instancia de ***Malla_simple*** le corresponde una instancia de ***Dibujador_malla***.
- Debe tener una única instancia accesible globalmente. Es muy recomendable utilizar un patrón Singleton o equivalente.

En la interacción, los pasos que ocurren son los siguientes:

1. El **Dibujador_objeto_malla** recibe la orden de dibujarse.
2. El **Dibujador_objeto_malla** pide a su **Objeto_malla** su ***Malla*** asociada.
3. El **Dibujador_objeto_malla** pide al **Asociador_mallas_dibujadores** el ***Dibujador_malla*** correspondiente a la ***Malla*** recibida.
4. El **Dibujador_objeto_malla** ordena dibujar al ***Dibujador_malla***.

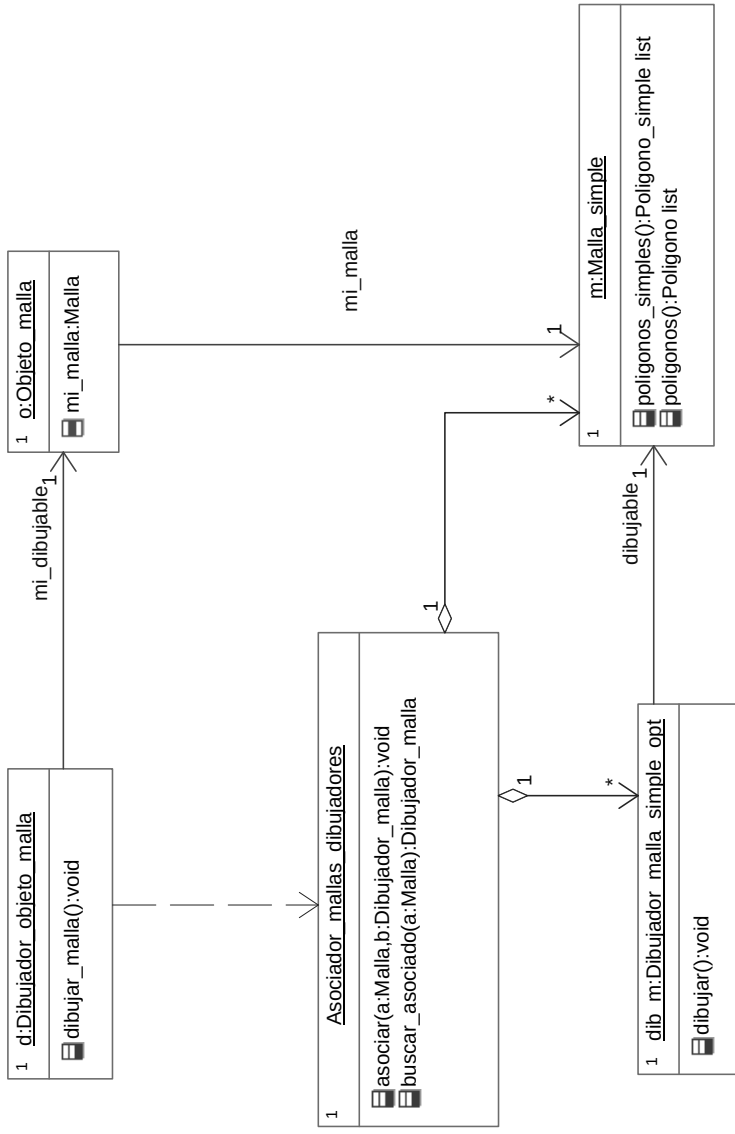

Figura 3.11: Diagrama de Clases del Asociador que conecta la clase ***Malla*** con la clase ***Dibujador_malla***

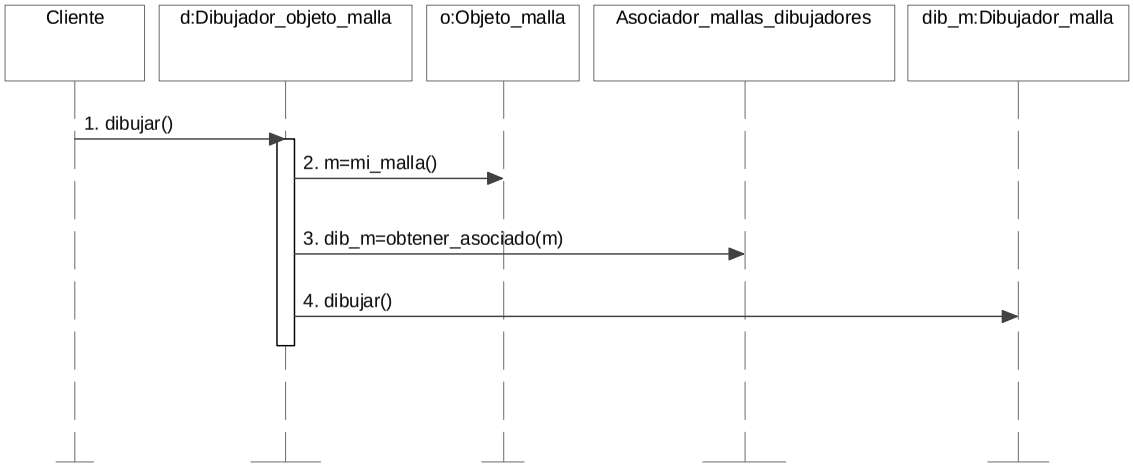

Figura 3.12: Diagrama de Secuencia de los pasos que necesita tomar un **Dibujador_objeto_malla** para acceder a su **Dibujador_malla_simple_opt** correspondiente

Resumen general de Contadores, Vigilantes y Asociadores

En el diseño del Motor OpenGL se han utilizado muchos Contadores, Vigilantes y Asociadores. Estos objetos no se incluyen en los diagramas, ya que no muestran funcionalidad nueva que no esté descrita. A continuación se detalla una lista de los mismos:

- **Asociador_bitmaps_dibujadores**: conecta las instancias de ***Bitmap*** con sus dibujadores.
  - **Contador_bitmap_RGBA** y **Vigilante_bitmap_RGBA**: Observan **Bitmap_RGBA** y crean instancias de **Dibujador_bitmap_RGBA**.
- **Asociador_texturas_dibujadores**: conecta las instancias de ***Textura*** con sus dibujadores.
  - **Contador_textura_bitmap** y **Vigilante_textura_bitmap**: Observan **Textura_bitmap** y crean instancias de **Dibujador_textura_bitmap**.
  - **Contador_no_textura** y **Vigilante_no_textura**: Observan **No_textura** y crean instancias de **Dibujador_no_textura**.
- **Asociador_mallas_dibujadores**: conecta las instancias de ***Malla*** con sus dibujadores.
  - **Contador_malla_simple** y **Vigilante_malla_simple**: Observan **Malla_simple** y crean instancias de **Dibujador_malla_simple_opt**.
- **Asociador_objetos_dibujadores**: conecta las instancias de ***Objeto*** con sus dibujadores.
  - **Contador_objeto_origen** y **Vigilante_objeto_origen**: Observan **Objeto_origen** y crean instancias de **Dibujador_objeto_origen**.
  - **Contador_dummy** y **Vigilante_dummy**: Observan **Dummy** y crean instancias de **Dibujador_dummy**.
  - **Contador_punto_luz** y **Vigilante_punto_luz**: Observan **Punto_luz** y crean instancias de **Dibujador_punto_luz**.
  - **Contador_foco** y **Vigilante_foco**: Observan **Foco** y crean instancias de **Dibujador_foco**.
  - **Contador_sol** y **Vigilante_sol**: Observan **Sol** y crean instancias de **Dibujador_sol**.
  - **Contador_camara_perspectiva** y **Vigilante_camara_perspectiva**: Observan **Camara_perspectiva** y crean instancias de **Dibujador_camara_perspectiva**.
  - **Contador_camara_ortografica** y **Vigilante_camara_ortografica**: Observan **Camara_ortografica** y crean instancias de **Dibujador_camara_ortografica**.
  - **Contador_objeto_malla** y **Vigilante_objeto_malla**: Observan **Objeto_malla** y crean instancias de **Dibujador_objeto_malla**.
- **Asociador_capas_dibujadores**: conecta las instancias de ***Capa*** con sus dibujadores.

  - **Contador_capa_nula** y **Vigilante_capa_nula**: Observan **Capa_nula** y crean instancias de **Dibujador_capa_nula**.
  - **Contador_capa_3d** y **Vigilante_capa_3d**: Observan **Capa3d** y crean instancias de **Dibujador_capa_3d**.
  - **Contador_capa_borrado** y **Vigilante_capa_borrado**: Observan **Capa_borrado** y crean instancias de **Dibujador_capa_borrado**.
  - **Contador_capa_borrado_zbuffer** y **Vigilante_capa_borrado_zbuffer**: Observan **Capa_borrado_zbuffer** y crean instancias de **Dibujador_capa_borrado_zbuffer**.

- **Asociador_escenas_dibujadores**: conecta las instancias de **Escena** con sus dibujadores.
  - **Contador_escena** y **Vigilante_escena**: Observan **Escena** y crean instancias de **Dibujador_escena**.

3.2.5. Dibujadores del nivel de Geometría

Los Dibujadores del nivel de Geometría implementan los mecanismos para cargar las diferentes texturas dentro de OpenGL y para dibujar las mallas de polígonos.

El problema del exceso de llamadas a OpenGL se aprecia especialmente en el dibujado de una malla de polígonos. Si se utilizan llamadas convencionales (GlNormal, GlColor, GlTexCoord y GlVertex) es muy común terminar con al menos 4 llamadas a OpenGL por cada vértice, más otras tantas para establecer el entorno de dibujado de cada polígono. Este modelo clásico de llamadas está implementado en la clase **Dibujador_malla_simple_basico**. Además del modelo de llamadas convencionales, las sucesivas revisiones de OpenGL añadieron mejoras y optimizaciones que permiten implementar mecanismos de dibujado más efectivos. En la clase **Dibujador_malla_simple_opt** se implementa el método de paso de geometría mediante arrays de vértices. Este método optimiza notablemente el sistema de dibujado, y se basa en guardar todas las coordenadas, normales, colores y coordenadas UV en arrays, y pasarle dichos arrays a OpenGL. Para pintar cada polígono, basta con indicar el índice, y OpenGL selecciona la coordenada, la normal, el color y la UV de los arrays.

Estructuras de datos

Dibujadores de bitmaps La función principal del **Dibujador_bitmap_RGBA** consiste en introducir y administrar instancias de **Bitmap_RGBA** dentro de OpenGL, usándolas como texturas. Esta clase tiene dos operaciones principales:

- **+enviar_a_opengl()**: Lee los datos del **Bitmap_RGBA** asociado y los introduce en OpenGL como una textura.
- **+activar()**: Selecciona el **Bitmap_RGBA** asociado como la textura actual en OpenGL. Si el **Bitmap_RGBA** no está disponible en OpenGL, lo introduce usando el método *enviar_a_opengl*.

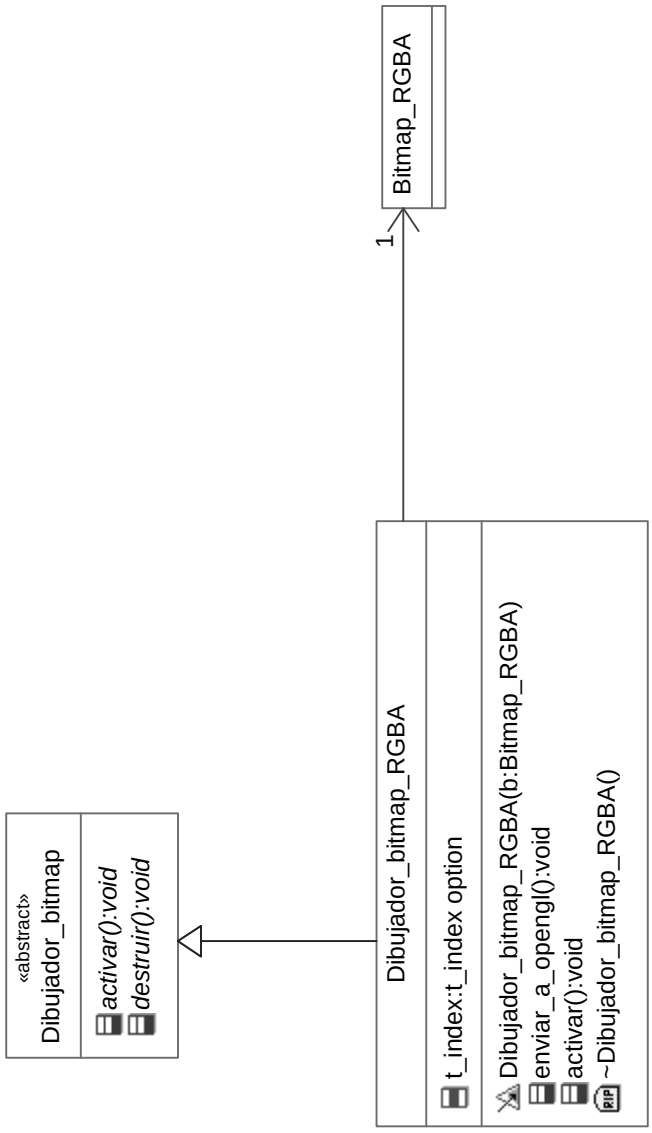

Figura 3.13: Diagrama de Clases del Dibujador de **Bitmap_RGBA**

Dibujadores de texturas La función principal del **Dibujador_textura_bitmap** consiste en activar el modo de texturizado de OpenGL y buscar y activar el **Dibujador_bitmap** adecuado. Esta clase también tiene una función de seguridad, que se encarga de cargar un bitmap si es requerido por una malla, pero no se encuentra cargado.

Al igual que existe la clase **No_textura**, también debe existir su dibujador correspondiente, en este caso **Dibujador_no_textura**. La función de activación de este dibujador es muy sencilla, ya que únicamente desactiva en OpenGL el modo de texturizado.

Dibujadores de mallas Las clases **Dibujador_malla_simple_basico** y **Dibujador_malla_simple_opt** implementan la misma funcionalidad, pero con dos enfoques distintos. La diferencia fundamental consiste en que **Dibujador_malla_simple_basico** consulta continuamente a su **Malla_simple** y a sus clases relacionadas acerca de los datos que tiene que enviar a OpenGL. En cambio, **Dibujador_malla_simple_opt** cambia el enfoque. Durante su instanciación lee toda la información posible de su **Malla_simple** y la prepara para OpenGL, teniéndo así todo listo para cada fotograma. De esta manera, **Dibujador_malla_simple_opt** obtiene un mayor rendimiento en cada fotograma, pagando con un mayor tiempo de instanciación.

Una segunda optimización que añade consiste en minimizar las llamadas de configuración de material. Para ello se ha creado un segundo método: *activar_material_opt*. Este método guarda información del material que está activado actualmente en OpenGL, y sólo llama a *activar_material* si se requiere un cambio.

En la versión actual, ambos tipos de dibujadores están preparados para dibujar únicamente la primera capa de cada polígono, de tal manera que si un polígono tiene más capas, éstas serán ignoradas.

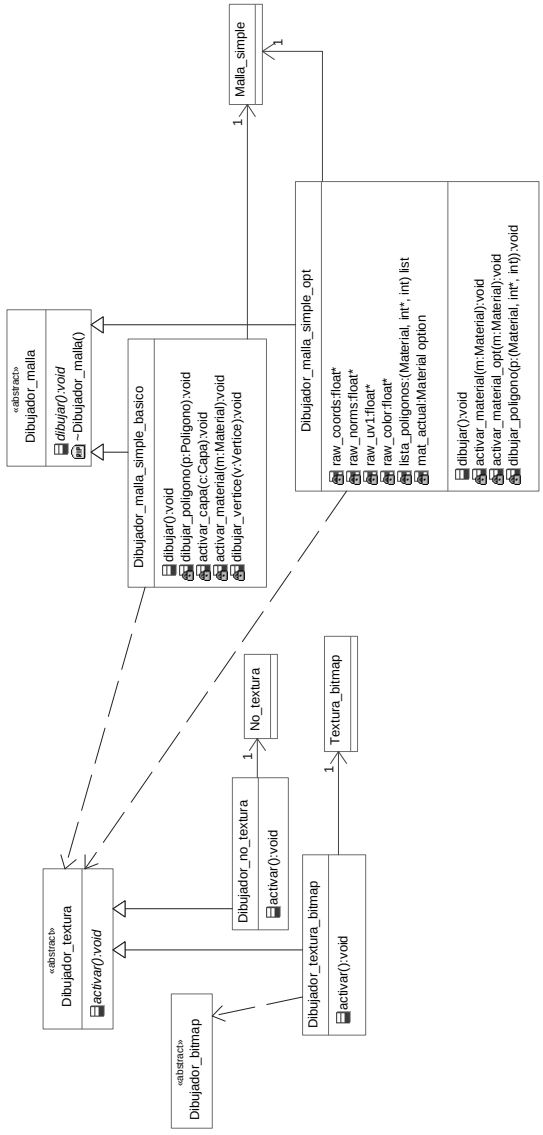

Figura 3.14: Diagrama de Clases de los Dibujadores de texturas y de **Malla_simple**

Algoritmos

El mecanismo que implementa **Dibujador_bitmap_RGBA** es básicamente una manipulación de tipos hasta construir un tipo aceptable por OpenGL, después el envío de información a OpenGL y por último la activación de algunos parámetros. La descripción detallada de su funcionamiento está disponible en el DVD adjunto, en el documento `MotorGrafico/dibujador_bitmap_rgba.dibujar.pdf`.

El **Dibujador_textura_bitmap** actua como un mecanismo de seguridad, además de sistema de activación de texturizado. La descripción detallada de su funcionamiento está disponible en el DVD adjunto, en el documento `MotorGrafico/dibujador_textura_bitmap.activar.pdf`.

El **Dibujador_malla_simple_basico** implementa un modelo básico para introducir geometría en OpenGL. La descripción detallada de su funcionamiento está disponible en el DVD adjunto, en el documento `MotorGrafico/dibujador_malla_simple_basico.dibujar.pdf`.

El **Dibujador_malla_simple_opt** implementa un método más efectivo para enviar la geometría a OpenGL. Se compone de dos partes:

- Inicialización: recorre **Malla_simple** acumulando y ordenando sus vértices en arrays.
- Ejecución: envía los arrays de vértices a OpenGL.

La fase de inicialización corresponde a los siguientes pasos:

1. **Dibujador_malla_simple_opt** genera una lista de vértices.

   Para cada polígono de la malla asociada:

   Para cada vértice del polígono:

   2. **Dibujador_malla_simple_opt** añade el vértice a la lista de vértices.

   Para cada vértice de la lista:

   3. **Dibujador_malla_simple_opt** genera un índice para el vértice.

4. **Dibujador_malla_simple_opt** genera un array para guardar las coordenadas.
5. **Dibujador_malla_simple_opt** genera un array para guardar las normales.
6. **Dibujador_malla_simple_opt** genera un array para guardar los colores.
7. **Dibujador_malla_simple_opt** genera un array para guardar las coordenadas UV.

   Para cada vértice de la lista:

   8. **Dibujador_malla_simple_opt** guarda las coordenadas en el array de coordenadas.
   9. **Dibujador_malla_simple_opt** guarda la normal en el array de normales.
   10. **Dibujador_malla_simple_opt** guarda el color en el array de colores.

11. **Dibujador_malla_simple_opt** guarda las coordenadas UV en el array de coordenadas UV.

Para cada polígono de la malla:

Para cada vértice del polígono:

12. **Dibujador_malla_simple_opt** busca el índice asociado al vértice.

13. **Dibujador_malla_simple_opt** crea una lista de índices de los vértices.
14. **Dibujador_malla_simple_opt** convierte la lista en un array.
15. **Dibujador_malla_simple_opt** convierte el array en un raw.

16. **Dibujador_malla_simple_opt** guarda los raws de cada polígono junto con su material y número de vértices.

17. **Dibujador_malla_simple_opt** convierte el array de coordenadas en un raw.

18. **Dibujador_malla_simple_opt** convierte el array de normales en un raw.

19. **Dibujador_malla_simple_opt** convierte el array de colores en un raw.

20. **Dibujador_malla_simple_opt** convierte el array de coordenadas UV en un raw.

Con la geometría convertida en raws, transferir dicha información a OpenGL es más rápido, y requiere menos llamadas. El envío de geometría a OpenGL ocurre en los siguientes pasos:

1. **Dibujador_malla_simple_opt** activa los arrays de vértices en OpenGL.

2. **Dibujador_malla_simple_opt** envía a OpenGL el raw de coordenadas.

3. **Dibujador_malla_simple_opt** activa los arrays de normales en OpenGL.

4. **Dibujador_malla_simple_opt** envía a OpenGL el raw de normales.

5. **Dibujador_malla_simple_opt** activa los arrays de colores en OpenGL.

6. **Dibujador_malla_simple_opt** envía a OpenGL el raw de colores.

7. **Dibujador_malla_simple_opt** activa los arrays de coordenadas de texturizado en OpenGL.

8. **Dibujador_malla_simple_opt** envía a OpenGL el raw de coordenadas de texturizado.
   Para cada polígono:

   9. **Dibujador_malla_simple_opt** llama a su método *dibujar_poligono* con el polígono.
   10. **Dibujador_malla_simple_opt** llama a su método *activar_material_opt* con el material del polígono.
   Si el material no está activado:

11. **Dibujador_malla_simple_opt** llama a su método *activar_material* con el material.
12. **Dibujador_malla_simple_opt** llama al método *activar* del **Dibujador_textura** correspondiente al material.
13. **Dibujador_malla_simple_opt** pide al **Material** su factor de sensibilidad a la luz ambiente.
14. **Dibujador_malla_simple_opt** pide al **Material** su factor de sensibilidad a la luz difusa.
15. **Dibujador_malla_simple_opt** pide al **Material** su factor de sensibilidad a la luz especular.
16. **Dibujador_malla_simple_opt** pide al **Material** su factor de brillo.
17. **Dibujador_malla_simple_opt** pide al **Material** su factor de emisión.
18. **Dibujador_malla_simple_opt** envía a OpenGL el factor de sensibilidad a la luz ambiente.
19. **Dibujador_malla_simple_opt** envía a OpenGL el factor de sensibilidad a la luz difusa.
20. **Dibujador_malla_simple_opt** envía a OpenGL el factor de sensibilidad a la luz especular.
21. **Dibujador_malla_simple_opt** envía a OpenGL el factor de emisión.
22. **Dibujador_malla_simple_opt** envía a OpenGL el factor de brillo.
23. **Dibujador_malla_simple_opt** envía a OpenGL los índices de los vértices del polígono.

3.2.6. Dibujadores del Nivel de Objetos

Los dibujadores del nivel de Objetos se encargan de implementar las manipulaciones y transformadas, y resolver algunas limitaciones de OpenGL. Para ello, se ha creado una organización en varias pasadas y se han añadido ciertos objetos, encargados de manipular el estado de OpenGL.

Estructuras de datos

Los Dibujadores del Nivel de Objetos imitan y utilizan la estructura del propio árbol de objetos para organizarse. Gracias al patrón Observador-Dibujador y a los Vigilantes, se crean instancias automáticamente para cada objeto que se instancia. Puesto que los propios dibujadores utilizan las referencias de los objetos para organizarse, el resultado es que conforman un árbol similar al árbol de objetos que dibujan.

Como era de esperar, cada clase del Nivel de Objetos tiene su dibujador correspondiente, pero hay algunas clases nuevas:

- ***I_Dibujador_Objeto***: Interfaz para los Dibujadores del Nivel de Objetos. Proporciona unas primitivas básicas a las que otras clases pueden llamar durante el dibujado. Sus operaciones son:
  - ***+activar_luces()***: Recorre el árbol de objetos introduciendo las luces encontradas en OpenGL.

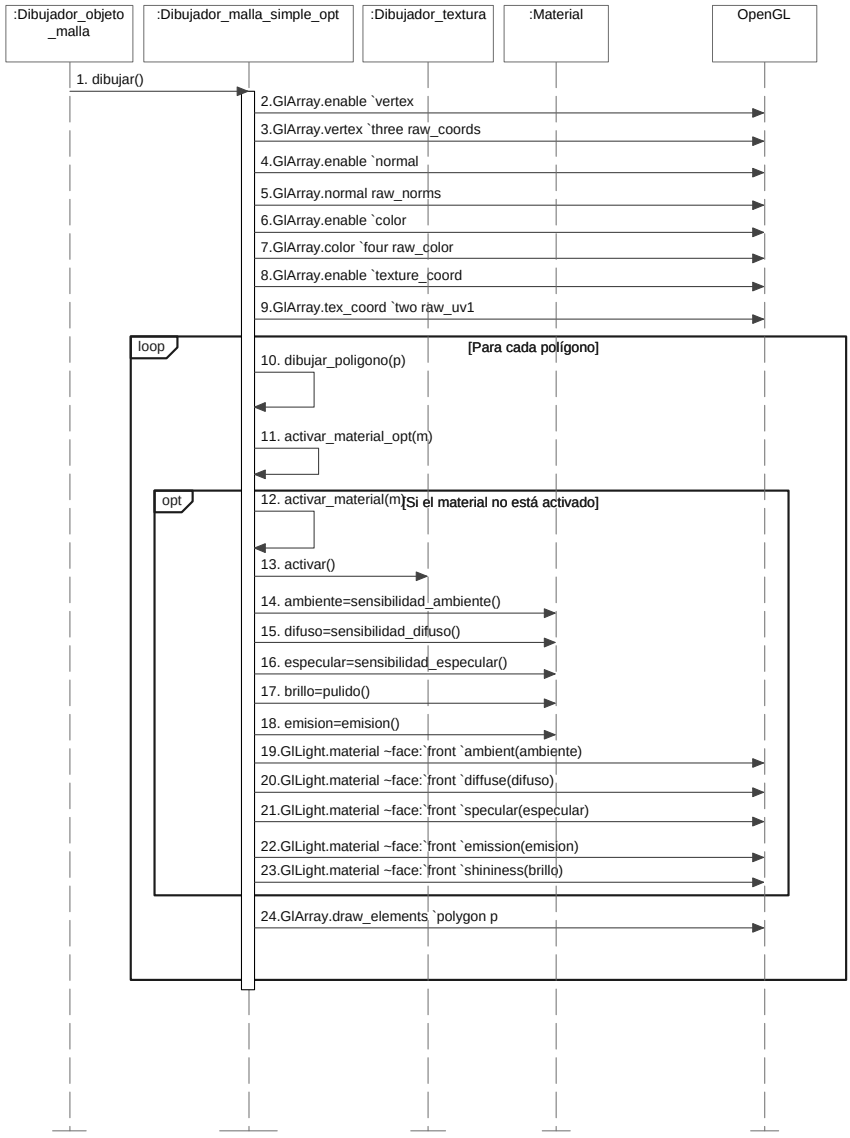

Figura 3.15: Diagrama de Secuencia del mecanismo utilizado para dibujar **Malla_simple** de una manera optimizada en OpenGL

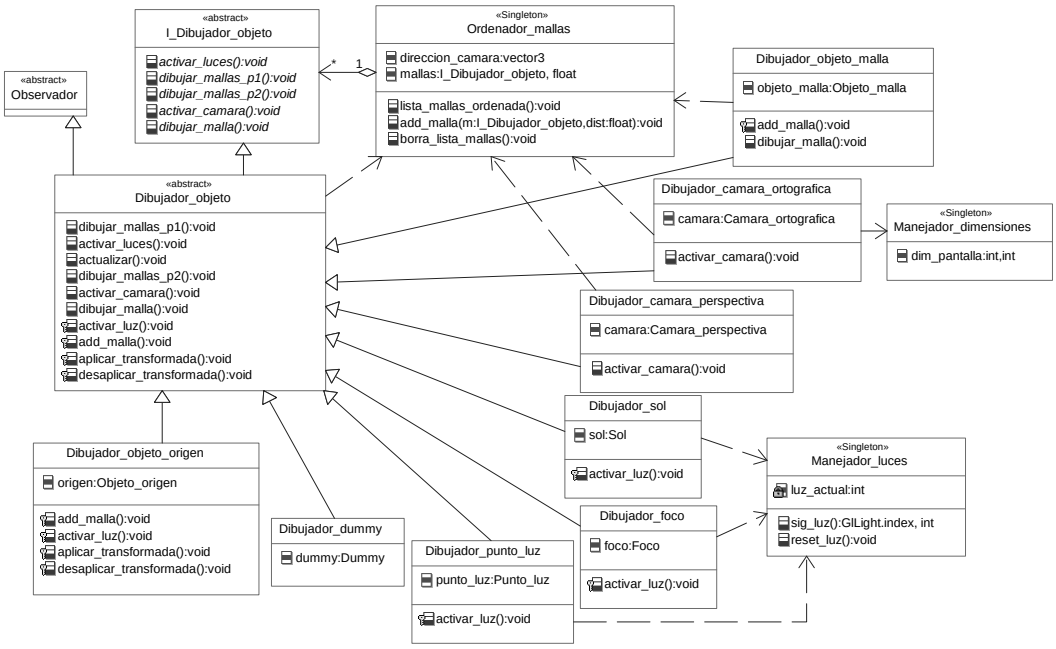

Figura 3.16: Diagrama de Clases de los Dibujadores del Nivel de Objetos

- ***+dibujar_mallas_p1()***: Recorre el árbol de objetos introduciendo las mallas encontradas en el ordenador de mallas.
- ***+dibujar_mallas_p2()***: Dibuja las mallas del ordenador de mallas.
- ***+activar_camara()***: Si es un dibujador de una cámara, introduce su información en OpenGL.
- ***+dibujar_malla()***: Si es una malla, llama a su dibujador correspondiente.

- ***Dibujador_objeto***: Implementa la funcionalidad básica de un dibujador del Nivel de Objetos. Es además un Observador, que vigila las operaciones de su ***Objeto*** asociado. Algunas de sus operaciones no hacen nada por defecto ya que serán redefinidas en sus subclases correspondientes. Sus operaciones son:
  - **#aplicar_transformada()**: Obtiene la transformada de su objeto asociado y la introduce en OpenGL.
  - **#desaplicar_transformada()**: Deshace los efectos de "aplicar_transformada".
  - **+activar_luces()**: Llama a "aplicar_transformada", a "activar_luz", a los dibujadores correspondientes a los hijos del objeto, y por último a "desaplicar_transformada".
  - **+dibujar_mallas_p1()**: Llama a "aplicar_transformada", a "add_malla", a los dibujadores correspondientes a los hijos del objeto, y por último a "desaplicar_transformada".

  - **+dibujar_mallas_p2()**: Obtiene la lista de dibujadores de mallas ordenados del ordenador de mallas, y llama a "dibujar_malla" de cada dibujador en orden.
  - **+actualizar()**: Por defecto no hace nada.
  - **+activar_camara()**: Por defecto no hace nada.
  - **+activar_luz()**: Por defecto no hace nada.
  - **+add_malla()**: Por defecto no hace nada.
  - **+dibujar_malla()**: Por defecto no hace nada.

- **Manejador_luces**: Este «singleton» se encarga de sortear la limitación de OpenGL de que sólo hay disponibles 8 luces. Actualmente contiene una implementación básica, y su tarea consiste en ir entregando los identificadores de las diferentes luces a medida que se los vayan pidiendo. Sus operaciones son:
  - **+sig_luz(): (GlLight.index, int)**: Devuelve el identificador de la siguiente luz disponible, junto con su índice. Si no quedan luces disponibles continua devolviendo la última.
  - **+reset_luz()**: Reinicia el contador de luces.

- **Ordenador_mallas**: Este «singleton» se encarga de sortear el problema de las transparencias en OpenGL. A la hora de pintar entidades transparentes, se obtiene el mejor efecto si se pintan "de atrás a adelante", es decir, de más lejanas a más cercanas a la cámara. Esta clase se encarga de recopilar todas las mallas de una escena y ordenarlas por distancia a la cámara. Sus operaciones son:
  - **+add_malla(m:I_Dibujador_objeto, dist:float)**: Añade un dibujador de mallas al ordenador de mallas.
  - **+borra_lista_mallas()**: Elimina todos los dibujadores de mallas del ordenador de mallas.
  - **+lista_mallas_ordenadas():I_Dibujador_objeto list**: Devuelve todos los dibujadores que tiene asociados ordenados por distancia a la cámara.
  - **+set_direccion_camara(v:Vector3)**: Establece la dirección hacia la que apunta la cámara.

- **Dibujador_objeto_origen**: Llama a los mecanismos de reinicio que ponen a OpenGL en un estado adecuado para dibujar otros objetos. Sus operaciones son:
  - **#add_malla()**: Reinicia el **Ordenador_mallas**.
  - **#activar_luz()**: Reinicia el **Manejador_luces** y los valores de las luces en OpenGL.
  - **#aplicar_transformada()**: No hace nada.
  - **#desaplicar_transformada()**: No hace nada.

- **Dibujador_dummy**: Puesto que el **Dummy** no se pinta en pantalla y su única forma de influencia en la escena es a través de su transformada, sus operaciones de dibujado ya están implementadas a través de ***Dibujador_objeto***.

- **Dibujador_punto_luz**: Implementa el mecanismo para enviar a OpenGL la información de un **Punto_luz**. Su operación principal es:

  - **#activar_luz()**: Obtiene un nuevo identificador de luz del **Manejador_luces** y asigna a dicho identificador los parámetros de su **Punto_luz** asociado.

- **Dibujador_foco**: Implementa el mecanismo para enviar a OpenGL la información de un **Foco**. Su operación principal es:

  - **#activar_luz()**: Obtiene un nuevo identificador de luz del **Manejador_luces** y asigna a dicho identificador los parámetros de su **Foco** asociado.

- **Dibujador_sol**: Implementa el mecanismo para enviar a OpenGL la información de un **Sol**. Su operación principal es:

  - **#activar_luz()**: Obtiene un nuevo identificador de luz del **Manejador_luces** y asigna a dicho identificador los parámetros de su **Sol** asociado.

- **Manejador_dimensiones**: Este «singleton» implementa un mecanismo por el cual se informa a una **Camara_perspectiva** del tamaño real de la pantalla. La **Camara_perspectiva** necesita esta información puesto que mide su apertura en grados/píxel, y necesita conocer su apertura total.

- **Dibujador_camara_perspectiva**: Implementa el mecanismo para enviar a OpenGL la información de una **Camara_perspectiva**. Su operación principal es:

  - **#activar_camara()**: Obtiene el tamaño de la pantalla del **Manejador_dimensiones**, calcula las matrices de proyección y de visión del modelo, y las introduce en OpenGL. Por último, envía su vector frontal al **Ordenador_mallas**.

- **Dibujador_camara_ortografica**: Implementa el mecanismo para enviar a OpenGL la información de una **Camara_ortografica**. Su operación principal es:

  - **#activar_camara()**: Calcula las matrices de proyección y de visión del modelo, y las introduce en OpenGL. Por último, envía su vector frontal al **Ordenador_mallas**.

- **Dibujador_objeto_malla**: Implementa el mecanismo para enviar a OpenGL la información de un **Objeto_malla**. Sus operaciones principales son:

  - **#add_malla()**: Guarda la transformada acumulada, calcula su profundidad con respecto al vector frontal de la cámara y se añade al **Ordenador_mallas**.
  - **#dibujar_malla()**: Recupera la transformada guardada anteriormente y llama al dibujador correspondiente a la malla.

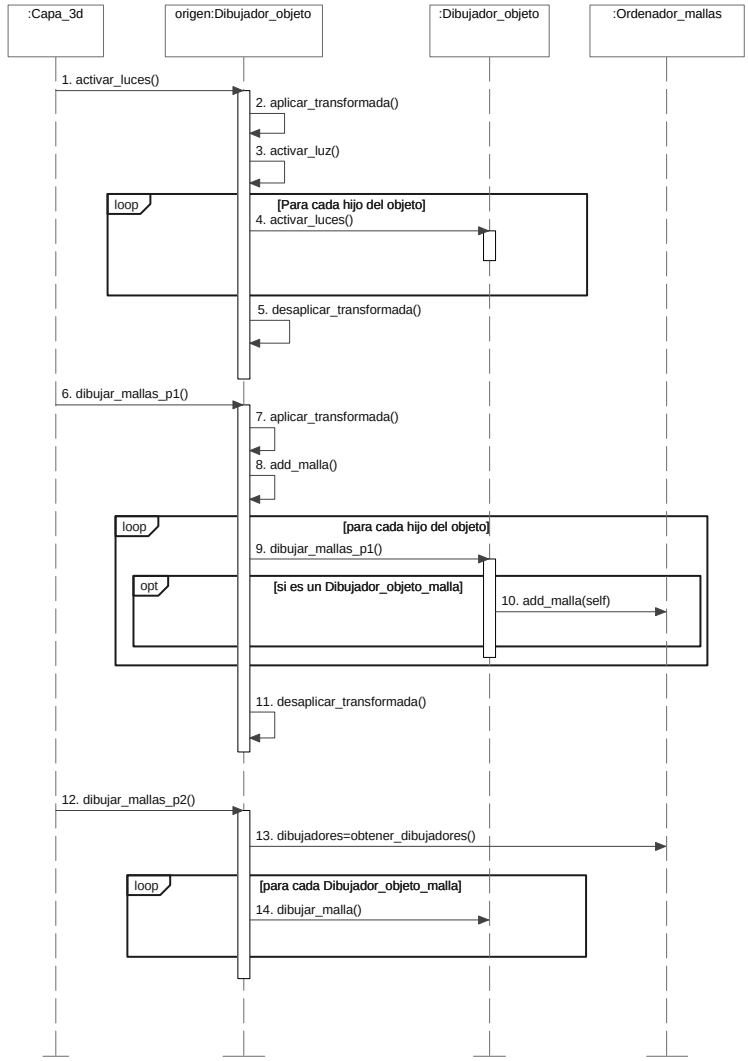

Figura 3.17: Diagrama de Secuencia del mecanismo utilizado para dibujar árboles de objetos en OpenGL

Algoritmos

Puesto que los objetos están organizados en árbol, se ha implementado un algoritmo de dibujado que recorre dicho árbol. El modelo de recorrido es un preorden, con las peculiaridades específicas para este caso concreto. El algoritmo general aplicado a nuestro caso es:

1. El objeto padre llama a la operación global: se procede al recorrido del sub-árbol llamando a las operaciones locales.
   2. Aplicar transformada: la transformada del objeto se introduce en OpenGL, combinándose con la transformada que ya estuviera.
   3. Llamar a la operación local: se realiza la operación, tal como activar la luz o introducir la malla en el ordenador de mallas.
   4. Llamar a la operación global para cada uno de los objetos hijo: el recorrido de los sub-árboles (uno por cada hijo) continúa.
   5. Desaplicar transformada: la transformada del objeto se anula, puesto que OpenGL ya la ha almacenado durante la llamada a la operación local.

Cabe destacar los siguientes puntos:

- El objetivo final es recorrer todo el árbol, y que en cada instante cada objeto tenga su transformada adecuada.
- Las operaciones globales (activar_luces, dibujar_mallas_p1, dibujar_mallas_p2) han sido separadas de las operaciones locales (activar_luz, add_malla, dibujar_malla) con el fin de facilitar la implementación de las operaciones locales.
- Puesto que la primera llamada es "aplicar transformada" y la última es "desaplicar_transformada", los objetos hijo se encuentran bajo la influencia de la transformada local, justo como se pretende en las interfaces.

El algoritmo general de dibujado se formula en 4 llamadas a los dibujadores:

1. Activar cámara: los datos de la cámara desde la que se observa la escena son enviados a OpenGL.
2. Activar luces: los datos de las luces de la escena son enviados a OpenGL.
3. Dibujar mallas pasada 1: se recopilan todas las mallas, y se guarda su profundidad desde el punto de vista de la cámara.
4. Dibujar mallas pasada 2: se ordenan las mallas por profundidad y se pintan de más lejanas a más cercanas.

Los métodos "aplicar_transformada" y "desaplicar_transformada" tienen la función de introducir y cancelar en OpenGL la transformada de un objeto. La descripción detallada de su funcionamiento está disponible en el DVD adjunto, en el documento `MotorGrafico/dibujador_objeto.aplicar_transformada.pdf`.

Los dibujadores encargados de aplicar las luces funcionan de una manera similar. Aprovechan su llamada a *activar_luz* para leer los datos correspondientes a su luz, y los envían a OpenGL. El funcionamiento de los mismos se puede consultar en el DVD adjunto, en los documentos `MotorGrafico/dibujador_punto_luz.activar_luz.pdf`, `MotorGrafico/dibujador_foco.activar_luz.pdf`, y `MotorGrafico/dibujador_sol.activar_luz.pdf`.

El **dibujador_camara_perspectiva** y el **dibujador_camara_ortografica** funcionan de una manera similar. Utilizan su llamada a *activar_camara* para leer los datos correspondientes a su cámara, y los envían a OpenGL. El funcionamiento de los mismos se puede consultar en el DVD adjunto, en los documentos `MotorGrafico/dibujador_camara_perspectiva.activar_camara.pdf` y `MotorGrafico/dibujador_camara_ortografica.activar_camara.pdf`.

El **Dibujador_objeto_malla** se encarga de trabajar con el **Ordenador_mallas** para preparar el entorno para un ***Dibujador_malla***. El dibujado ocurre en dos fases, como está mostrado en el diagrama. Primero el **Dibujador_objeto_malla** extrae la posición absoluta de su **Objeto_malla** asociado y la introduce en el **Ordenador_mallas**. Después, cuando es llamado a dibujar en la segunda pasada, recupera su transformada y llama al ***Dibujador_malla*** adecuado. La secuencia de llamadas es la siguiente:

Durante la primera pasada:

1. El **Dibujador_objeto_malla** llama a su método "add_malla".
2. El **Dibujador_objeto_malla** guarda la transformada actual de OpenGL.
3. El **Dibujador_objeto_malla** extrae la posición absoluta de la transformada.
4. El **Dibujador_objeto_malla** pide al **Ordenador_mallas** el vector frontal de la cámara.
5. El **Dibujador_objeto_malla** calcula la profundidad de la malla desde el punto de vista de la cámara.
6. El **Dibujador_objeto_malla** se agrega al **Ordenador_mallas** con su distancia.

Durante la segunda pasada:

7. El **Dibujador_objeto_malla** llama a su método "dibujar_malla".
8. El **Dibujador_objeto_malla** introduce la transformada guardada en OpenGL.
9. El **Dibujador_objeto_malla** llama al dibujador correspondiente a su malla.

3.2.7. Dibujadores del Nivel de Pantalla

Los Dibujadores del Nivel de Pantalla se encargan de implementar los mecanismos para dibujar escenas y capas, además de para cambiar de resolución y modo de pantalla.

Estructuras de datos

Los Dibujadores del Nivel de Pantalla mantienen la misma organización que el Nivel de Pantalla, creando un dibujador por cada clase que debe ser dibujada. Las clases tienen las siguientes funciones:

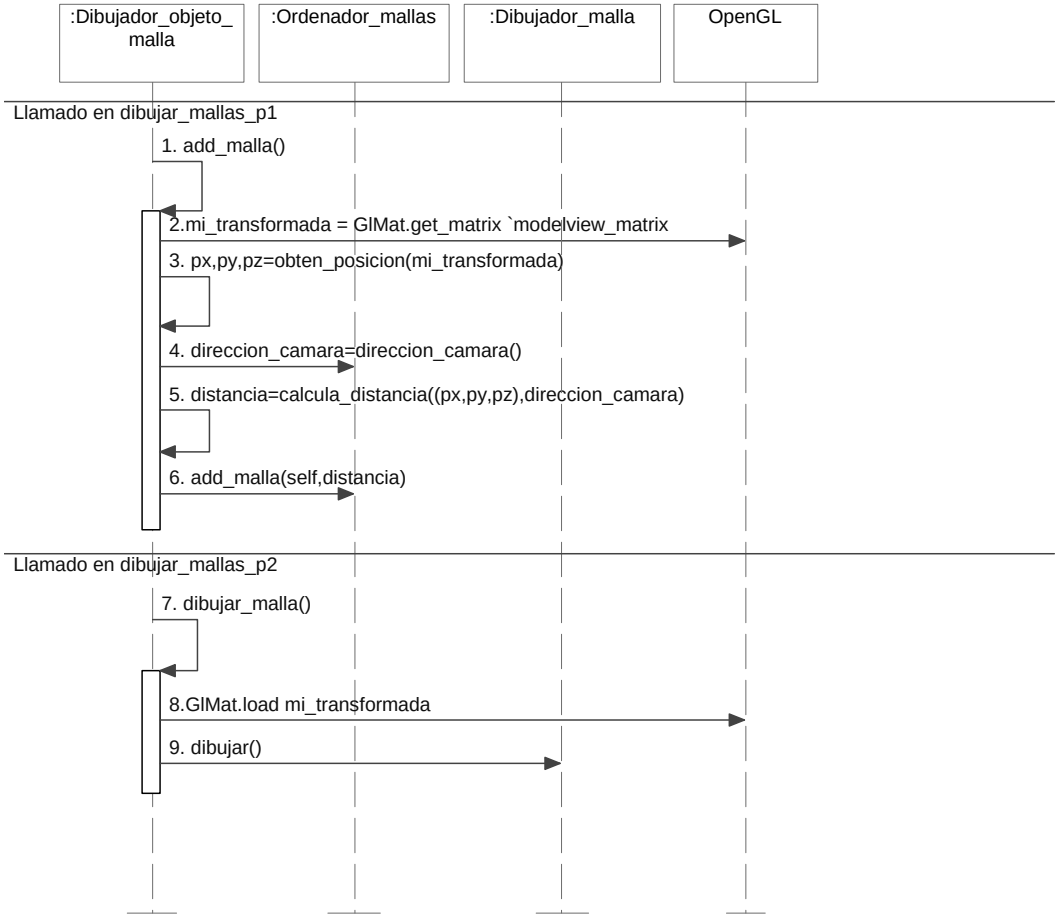

Figura 3.18: Diagrama de Secuencia del mecanismo utilizado para procesar y enviar a OpenGL un **Objeto_malla**

- ***Dibujador_capa***: Es un modelo básico para implementar dibujadores de capas.
  - ***dibujar*()**: Aplica la capa a OpenGL.
- **Dibujador_capa_3d**: Implementa un dibujador para dibujar un árbol de objetos como una capa.
  - **dibujar()**: Llama a "activar_camara" de la cámara desde la que se ve el árbol de objetos, y después a "activar_luces", "dibujar_mallas_p1" y "dibujar_mallas_p2" siguiendo el modelo de dibujado en tres pasadas.
- **Dibujador_capa_borrado**: Implementa un dibujador que pinta la pantalla de un color.
  - **dibujar()**: Ordena a OpenGL borrar la pantalla con el color especificado.
- **Dibujador_capa_borrado_zbuffer**: Implementa un dibujador que reinicia el z-buffer.
  - **dibujar()**: Ordena a OpenGL reiniciar el z-buffer.
- **Dibujador_capa_nula**: Implementa un dibujador para la capa nula, que no hace nada.
  - **dibujar()**: No hace nada.
- **Dibujador_escena**: Implementa un dibujador que pinta una escena.
  - **dibujar()**: Llama a los dibujadores de las capas en orden.
- ***Dibujador_pantalla***: Implementa un dibujador que controla y pinta la escena de una pantalla.
  - **dibujar()**: Si la pantalla tiene asignada una escena, llama al dibujador de la escena.
- **Dibujador_monitor**: Implementa un ***Dibujador_pantalla*** pensado para controlar el monitor principal de un ordenador.
  - **actualizar()**: Recupera los datos de cambio de modo o resolución y establece dicha resolución en el monitor.

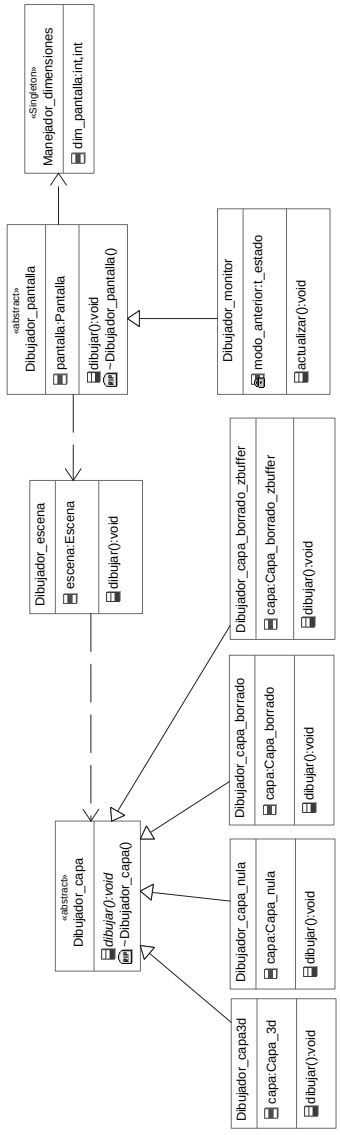

Figura 3.19: Diagrama de Clases de los Dibujadores del Nivel de Pantalla

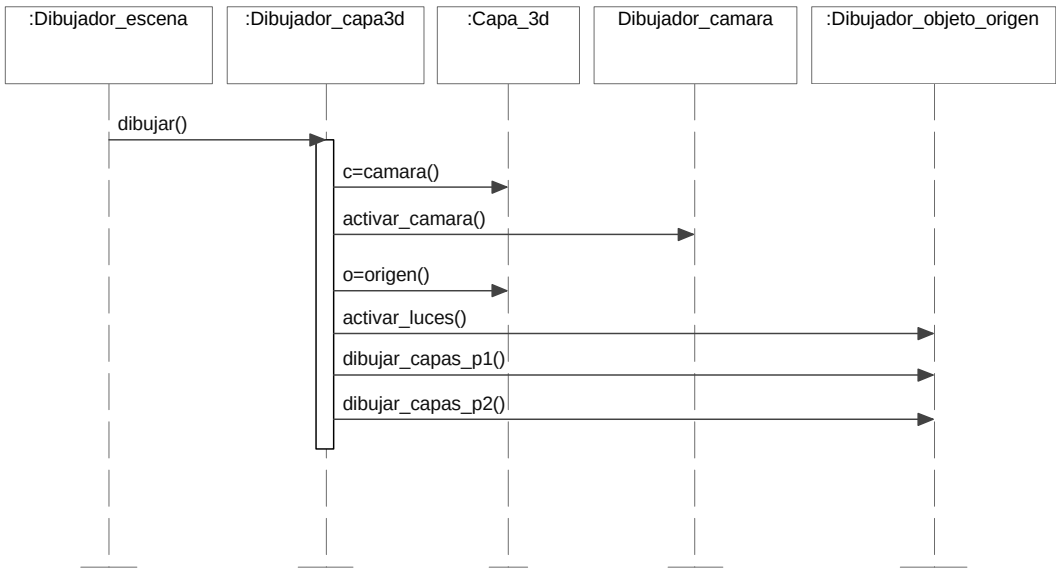

Figura 3.20: Diagrama de Secuencia de la operación "dibujar" de la clase **Dibujador_capa_3d**

Algoritmos

El **Dibujador_capa_3d** implementa el algoritmo de las 3 pasadas, explicado anteriormente en la sección de los Dibujadores del Nivel de Objetos. La secuencia de llamadas es la siguientes:

1. El **Dibujador_escena** llama al método "dibujar" del **Dibujador_capa_3d**.
2. El **Dibujador_capa_3d** pide la ***Camara*** asociada a su **Capa3d**.
3. El **Dibujador_capa_3d** llama al método "activar_camara" del dibujador de la cámara.
4. El **Dibujador_capa_3d** pide el **Objeto_origen** asociado a su **Capa3d**.
5. El **Dibujador_capa_3d** llama al método "activar_luces" del dibujador del **Objeto_origen**.
6. El **Dibujador_capa_3d** llama al método "dibujar_mallas_p1" del dibujador del **Objeto_origen**.
7. El **Dibujador_capa_3d** llama al método "dibujar_mallas_p2" del dibujador del **Objeto_origen**.

El **dibujador_capa_borrado** y el **dibujador_capa_borrado_zbuffer** implementan mecanismos para reiniciar el *framebuffer*. El funcionamiento de los mismos se puede consultar en el

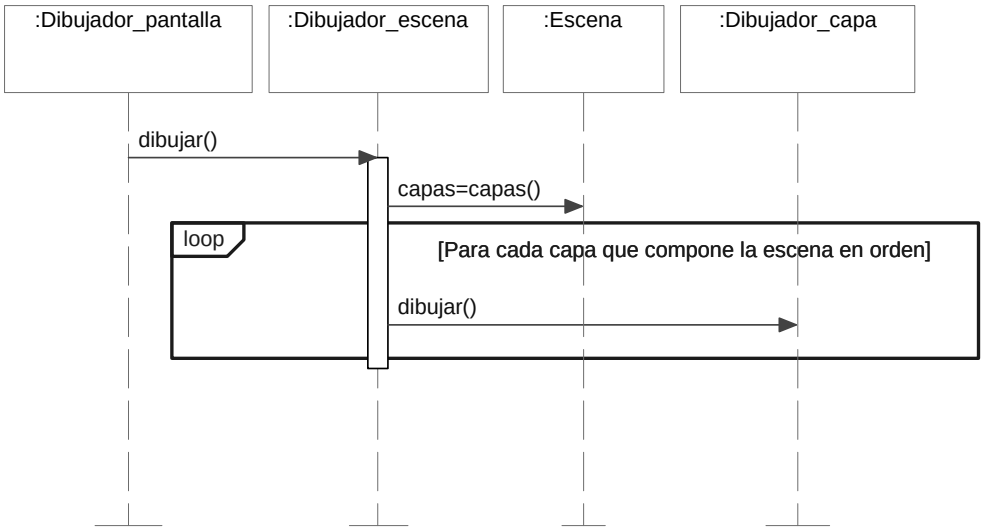

Figura 3.21: Diagrama de Secuencia de la operación "dibujar" de la clase **Dibujador_escena**

DVD adjunto, en los documentos `MotorGrafico/dibujador_capa_borrado.dibujar.pdf` y `MotorGrafico/dibujador_capa_borrado_zbuffer.dibujar.pdf`.

El **Dibujador_escena** aplica las capas de dibujado por orden. La secuencia de llamadas es la siguientes:

1. El **Dibujador_pantalla** llama al método "dibujar" del **Dibujador_escena**.
2. El **Dibujador_escena** pide a su **Escena** asociada las capas que la componen.
   Para cada capa de la escena:
3. El **Dibujador_escena** llama al método "dibujar" del ***Dibujador_capa*** correspondiente.

El **Dibujador_pantalla** inicializa la clase **Manejador_dimensiones** y llama a su **Dibujador_escena**. La secuencia de llamadas es la siguientes:

1. El Motor OpenGL llama al método "dibujar" del **Dibujador_pantalla**.
2. El **Dibujador_pantalla** pide a su ***Pantalla*** su tamaño.
3. El **Dibujador_pantalla** asigna el tamaño al **Manejador_dimensiones**.
4. El **Dibujador_pantalla** pide a su ***Pantalla*** su escena asociada.
   Si la ***Pantalla*** tiene asignada una escena:
5. El **Dibujador_pantalla** llama al método "dibujar" del **Dibujador_escena** correspondiente.

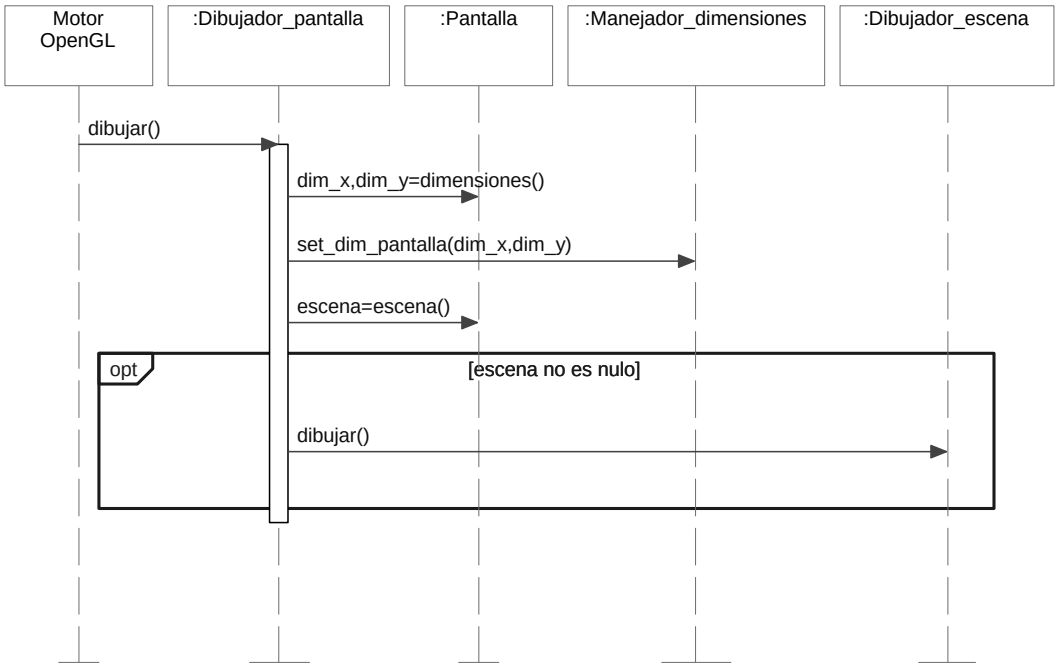

Figura 3.22: Diagrama de Secuencia de la operación "dibujar" de la clase **Dibujador_pantalla**

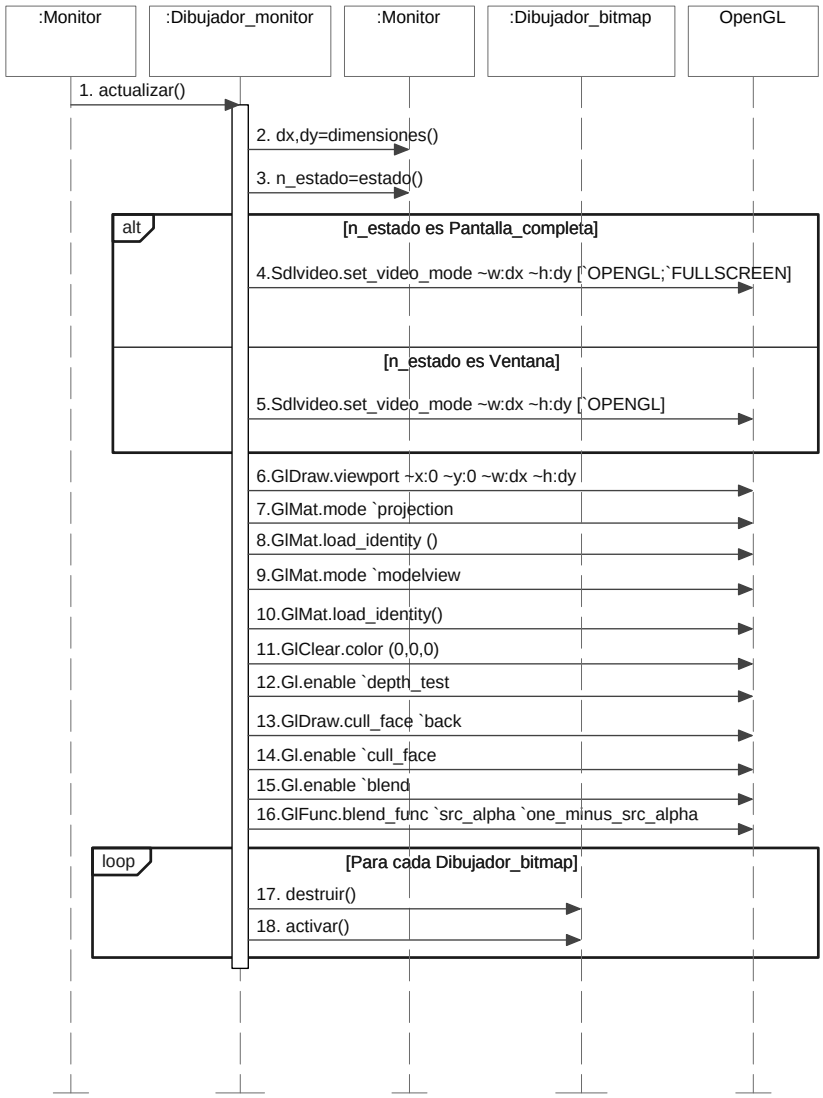

Figura 3.23: Diagrama de Secuencia de la operación "actualizar" de la clase **Dibujador_monitor**

El **Dibujador_monitor** implementa el mecanismo general para cambiar el modo de pantalla en un monitor. Las llamadas se encargan de reiniciar los parámetros generales de OpenGL, puesto que SDL destruye y crea un contexto nuevo al cambiar de modo. Además se reenvían las texturas, ya que con la destrucción y re-creación del contexto, también se pierden. La secuencia de llamadas es la siguientes:

1. El **Monitor** sufre cambios y genera un evento "actualizar" para "**Dibujador_monitor**, siguiendo el patrón Observador.
2. El **Dibujador_monitor** recupera la nueva resolución de su **Monitor** asociado.
3. El **Dibujador_monitor** recupera el nuevo modo de su **Monitor** asociado.
4. El **Dibujador_monitor** recupera las nuevas dimensiones de su **Monitor** asociado.
   Si el nuevo modo es "Pantalla_completa":
   a) El **Dibujador_monitor** ordena a SDL cambiar a pantalla completa con la resolución especificada.

   Si el nuevo modo es "Ventana":
   a) El **Dibujador_monitor** ordena a SDL cambiar a ventana con la resolución especificada.
5. El **Dibujador_monitor** establece el puerto de visualización como el total de la pantalla.
6. El **Dibujador_monitor** selecciona la matriz de proyección.
7. El **Dibujador_monitor** reinicia la matriz de proyección.
8. El **Dibujador_monitor** selecciona la matriz de visión del modelo.
9. El **Dibujador_monitor** reinicia la matriz de visión del modelo.
10. El **Dibujador_monitor** establece el color de borrado como (0,0,0).
11. El **Dibujador_monitor** establece el uso de buffer de profundidad.
12. El **Dibujador_monitor** establece el modo de eliminación de caras traseras.
13. El **Dibujador_monitor** activa la eliminación de caras.
14. El **Dibujador_monitor** establece el uso de transparencias.
15. El **Dibujador_monitor** establece el modo de transparencias.
    Para cada **Dibujador_bitmap**:
    a) El **Dibujador_monitor** ordena al **Dibujador_bitmap** eliminar los datos correspondientes al anterior contexto de OpenGL.
    b) El **Dibujador_monitor** ordena al **Dibujador_bitmap** enviar los bitmaps a OpenGL.

3.2.8. Control general del Motor OpenGL

Todos los dibujadores y su estructura y control general son preparados desde una clase central, cuyas funciones consisten en inicializar y terminar adecuadamente el Motor OpenGL y servir de punto de entrada para dibujar el fotograma. Dicha clase recibe el nombre de **Motor_OpenGL**.

Estructuras de datos

En el patrón Observador-Dibujador y los patrones Asociadores se definió un mecanismo general que permite conectar las clases del Motor Gráfico con los dibujadores del Motor OpenGL. Para hacer funcionar correctamente todo el mecanismo es necesario que alguien se encargue de inicializar y controlar las clases de las diferentes instancias del patrón. Es especialmente necesario crear e inicializar los Vigilantes y los Asociadores, ya que el resto puede inicializarse por sí mismo. La clase **Motor_OpenGL** se encarga de esta tarea, y sus operaciones son:

- **preparar_dibujadores_bitmap_RGBA()**
- **preparar_dibujadores_textura_bitmap()**
- **preparar_dibujadores_no_textura()**
- **preparar_dibujadores_malla_simple()**
- **preparar_dibujadores_objeto_origen()**
- **preparar_dibujadores_dummy()**
- **preparar_dibujadores_punto_luz()**
- **preparar_dibujadores_foco()**
- **preparar_dibujadores_sol()**
- **preparar_dibujadores_camara_perspectiva()**
- **preparar_dibujadores_camara_ortografica()**
- **preparar_dibujadores_objeto_malla()**
- **preparar_dibujadores_capa3d()**
- **preparar_dibujadores_capa_nula()**
- **preparar_dibujadores_capa_borrado()**
- **preparar_dibujadores_capa_borrado_zbuffer()**
- **preparar_dibujadores_escena()**: Crea un vigilante que vigila al contador correspondiente a la entidad y crea dibujadores para las instancias de la entidad.

- **inicializar()**: Crea un **Dibujador_monitor** para el monitor principal, pone el modo de pantalla por defecto y llama a las operaciones "preparar_dibujadores".
- **dibujar_fotograma()**: Llama a los dibujadores de todas las pantallas disponibles, y ordena a OpenGL mostrar la imagen resultado.
- **destruir_dibujadores_bitmap_RGBA()**
- **destruir_dibujadores_textura_bitmap()**
- **destruir_dibujadores_no_textura()**
- **destruir_dibujadores_malla_simple()**
- **destruir_dibujadores_objeto_origen()**
- **destruir_dibujadores_dummy()**
- **destruir_dibujadores_punto_luz()**
- **destruir_dibujadores_foco()**
- **destruir_dibujadores_sol()**
- **destruir_dibujadores_camara_perspectiva()**
- **destruir_dibujadores_camara_ortografica()**
- **destruir_dibujadores_objeto_malla()**
- **destruir_dibujadores_capa3d()**
- **destruir_dibujadores_capa_nula()**
- **destruir_dibujadores_capa_borrado()**
- **destruir_dibujadores_capa_borrado_zbuffer()**
- **destruir_dibujadores_escena()**: Elimina todos los dibujadores de la entidad correspondiente, y después al vigilante de dicha entidad.
- **terminar()**: Llama a las operaciones "destruir_dibujadores".

Algoritmos

La inicialización del Motor OpenGL está compuesta de dos partes: primero el sistema crea un dibujador para el **Monitor_principal** y lo inicializa, de tal manera que se crea un contexto de OpenGL. Después el sistema crea vigilantes y dibujadores para todas las entidades del Motor Gráfico. Los detalles concretos se pueden consultar en el DVD adjunto, en el documento `MotorGrafico/motor_opengl.inicializar.pdf`.

Cuando el Motor OpenGL ya no es necesario se debe llamar a su operación de terminación y limpieza general. Esta operación, llamada "terminar", llama a las operaciones

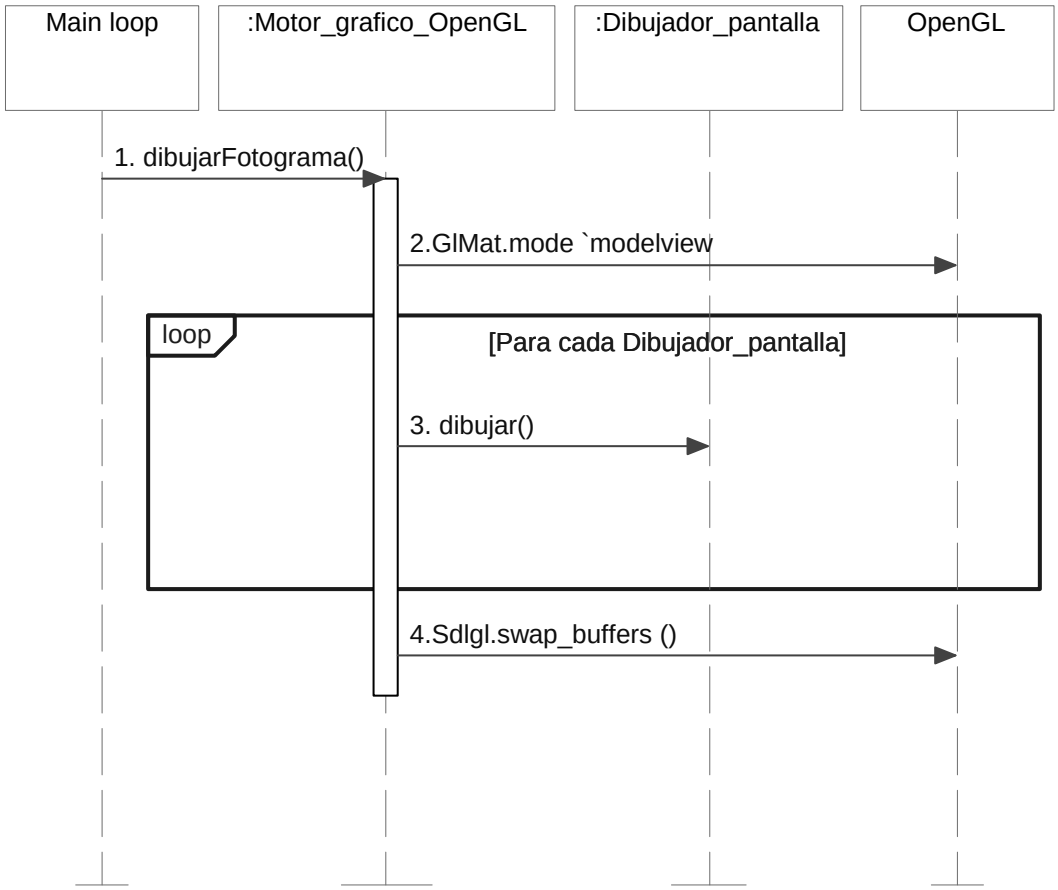

Figura 3.24: Diagrama de Secuencia de la operación “dibujarFotograma” del Motor OpenGL

"destruir_dibujadores". Los detalles concretos se pueden consultar en el DVD adjunto, en el documento `MotorGrafico/motor_opengl.terminar.pdf`.

La operación "dibujarFotograma" es el punto de llamada desde el Controlador, y se encarga de tomar toda la escena tal como está en un determinado instante y dibujarla en OpenGL. Los pasos que ocurren son los siguientes:

1. El Controlador llama a "dibujarFotograma" del Motor OpenGL.
2. El Motor OpenGL selecciona en OpenGL para ser usada la matriz de visión del modelo. Para cada ***Dibujador_pantalla***:
    3. El Motor OpenGL llama al método "dibujar" del ***Dibujador_pantalla***.
4. El Motor OpenGL ordena a OpenGL mostrar la imagen resultado.

Por último, se muestra una vista general del funcionamiento del Motor OpenGL en un uso convencional. En este caso concreto, el sistema se ha configurado de tal manera que sólo se utiliza una pantalla, el **Monitor_principal**. En él, se muestra una **Escena** compuesta por tres capas: una **Capa_borrado**, una **Capa_borrado_zbuffer**, y una **Capa3d**. La secuencia de operaciones que ocurre es la siguiente:

1. El Controlador llama al método "dibujarFotograma" del Motor OpenGL.
2. El Motor OpenGL ordena dibujar el **Monitor_principal**.
3. El **Dibujador_monitor_principal** ordena dibujar su **Escena** asociada.
4. El **Dibujador_escena** ordena dibujar su primera capa: una **Capa_borrado**.
5. El **Dibujador_capa_borrado** ordena a OpenGL borrar la pantalla.
6. El **Dibujador_escena** ordena dibujar su segunda capa: una **Capa_borrado_zbuffer**.
7. El **Dibujador_capa_borrado_zbuffer** ordena a OpenGL borrar el buffer de profundidad.
8. El **Dibujador_escena** ordena dibujar su tercera capa: una **Capa3d**.
9. El **Dibujador_capa_3d** ordena seleccionar su cámara.
10. El **Dibujador_camara** toma los datos de su ***Camara*** asociada y los introduce en OpenGL.
11. El **Dibujador_capa_3d** ordena activar las luces de la escena al objeto origen. Para cada luz de la escena:
    12. El dibujador correspondiente a la ***Luz*** introduce sus datos en OpenGL.
13. El **Dibujador_capa_3d** ordena la primera pasada para dibujar los **Objeto_malla**. Los **Objeto_malla** se ordenan por distancia a la cámara.

14. El **Dibujador_capa_3d** ordena la segunda pasada para dibujar los **Objeto_malla**. Para cada **Objeto_malla** y por orden de distancia:
    15. El dibujador correpondiente al **Objeto_malla** ordena dibujar la ***Malla*** asociada. Para cada ***Textura*** de la ***Malla***:
        16. El dibujador correpondiente a la ***Textura*** introduce los datos de la misma en OpenGL.
    17. El ***Dibujador_malla*** envía a OpenGL la geometría de dicha malla.
18. El Motor OpenGL ordena a OpenGL mostrar el resultado de las operaciones.

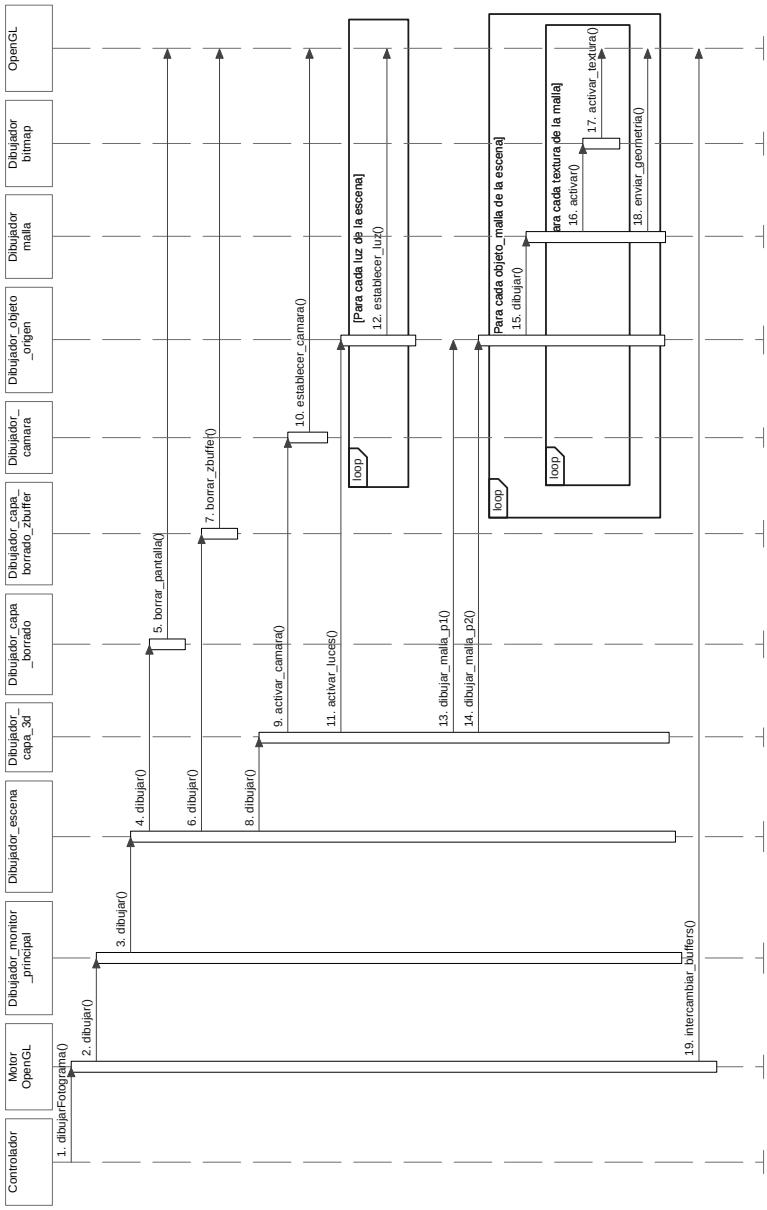

Figura 3.25: Vista general del funcionamiento del Motor OpenGL

Capítulo 4

Diseño del Gestor de Recursos

4.1. Modelo Estructural y Arquitectónico

El Gestor de Recursos desempeña la tarea de proveer un mecanismo unificado para acceder a entidades que estén guardadas en disco u otros medios de almacenamiento masivo, y cargarlas en memoria. Debido a la relativa simplicidad del subsistema, no se ha visto adecuado subdividirlo en partes.

El modelo estructural del Gestor de Recursos es similar a su modelo de diseño. Se basa en el principio de dividir la carga de recursos en tres pasos:

- **Cargador**: Lectura de los datos del recurso desde archivos de disco.
- **Instanciador**: Creación de la instancia del recurso a partir de los datos.
- **Gestor**: Organización del recurso dentro de una biblioteca.

4.1.1. Cargador

Lee los archivos de disco y carga los datos necesarios para el recurso.

Interfaces ofrecidas

- **Uso desde el Instanciador**: Ofrece un interfaz para la carga, descarga y duplicado de los datos necesarios para un recurso.
- **Cargadores concretos**: Ofrece una estructura genérica para construir cargadores para cualquier tipo de recurso.

Interfaces utilizadas

- **LeerFicheros**: Acceso a las funciones del Sistema Operativo para lectura de ficheros.

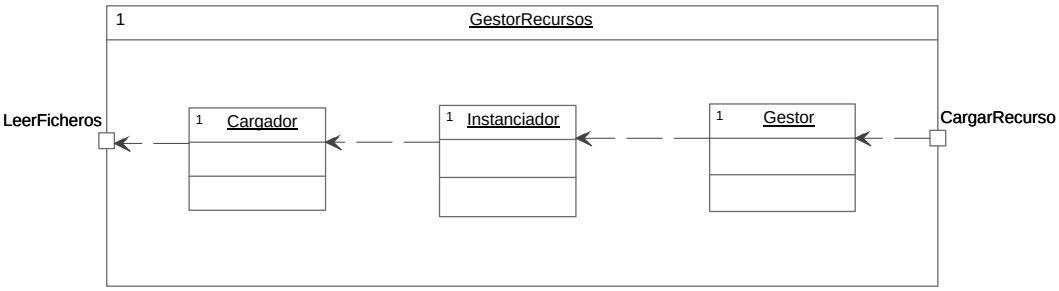

Figura 4.1: Diagrama estructural del Gestor de Recursos

4.1.2. Instanciador

Crea instancias del recurso que se quiere cargar a partir de los datos que lo componen.

Interfaces ofrecidas

- **Uso desde el Gestor**: Ofrece un interfaz para la carga, descarga, instanciado y duplicado de un recurso.
- **Instanciadores concretos**: Ofrece una estructura genérica para construir instanciadores para cualquier tipo de recurso.

Interfaces utilizadas

- **Cargador**: Acceso al Cargador para cargar los datos necesarios para un recurso.

4.1.3. Gestor

Organiza los recursos en bibliotecas, asignando un nombre a cada recurso.

Interfaces ofrecidas

- **Obtención de recursos**: Ofrece un interfaz para la carga, descarga, instanciado y duplicado de cualquier recurso asociado al gestor.

Interfaces utilizadas

- **Instanciador**: Acceso al Instanciador concreto para cada recurso que se quiera tratar.

4.2. Diseño detallado

El Gestor de Recursos ha sido pensado y organizado para ser genérico y reutilizable, además de para soportar grandes bibliotecas de entidades. Sus enfoques principales son los siguientes:

- **Extensión por herencia**: El Gestor de Recursos provee un interfaz básico con ciertas funciones predefinidas, pensadas para apoyar la creación de gestores específicos para cargar entidades de tipos concretos. El mecanismo pensado para la utilización de este subsistema consiste en la herencia. Se deben crear cargadores e instanciadores derivados de los abstractos provistos, e implementar en ellos los mecanismos concretos de carga e instanciación de entidades.
- **Parcialmente descentralizado**: Puesto que es necesaria la creación de nuevos instanciadores y cargadores para cada tipo de recurso, es muy difícil hacer que el funcionamiento del Gestor esté centralizado en un subsistema. Por ello se ha decidido que cada subistema que requiera cargar entidades de disco tendrá que crear sus propias instancias del Gestor de Recursos.
- **Carga en segundo plano**: La carga de entidades puede ser muy lenta, y no es recomendable hacer esperar al jugador ese tiempo. Por ello se piensa en cargar dichas entidades en un segundo hilo de ejecución, de tal manera que el sistema pueda ir precargando las entidades necesarias y estén listas cuando sean necesarias.

Para ello se divide la tarea de creación de entidades en varias partes:

- **Carga**: Consiste en la lectura de los datos necesarios para el recurso de algún dispositivo, normalmente de un disco duro. Es una tarea potencialmente lenta, especialmente para recursos que estén compuestos por muchos archivos o archivos muy grandes. Esta operación puede ser ejecutada en segundo plano.
- **Instanciación**: Una vez que los datos estén cargados en memoria y organizados adecuadamente, el recurso se puede instanciar. Esta operación toma dichos datos y los convierte en un objeto. Es de esperar que esta operación sea mucho más rápida, casi instantánea, y originalmente está pensada para crear un objeto envoltorio para los datos, de tal manera que todos los objetos comparten datos, mejorando el aprovechamiento de memoria.
- **Creación de un duplicado**: A veces es necesario modificar un objeto. El problema reside en que todos los objetos comparten los datos subyacentes, de tal manera que al modificar uno se modificarían todos. Por ello, se añade la operación de creación de un duplicado. Esta operación duplica los datos subyacentes, y permite la modificación

Además de estas operaciones se añade una de consulta del estado, que nos será útil para saber cuándo ha terminado la carga de un recurso y está listo para su instanciación.

4.2.1. Estructuras de datos

El Gestor de Recursos está pensado para ser genérico y poder cargar cualquier tipo de recurso. Por ello, ha sido organizado de la manera más abstracta y genérica. Así, el ***Cargador*** y el ***Instanciador*** son clases abstractas, que deben ser extendidas para cargar tipos concretos de recurso. Las clases ***Datos*** y ***Objeto*** en realidad son parámetros. El objetivo es disponer de un conjunto de clases que se pueda modificar fácilmente para cargar cualquier tipo de entidad, manteniendo un interfaz concreto. Las operaciones de cada clase son las siguientes:

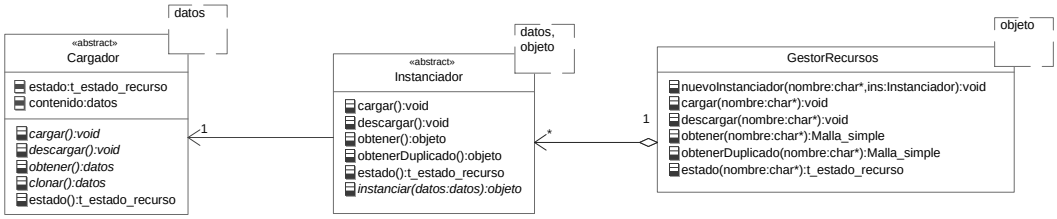

Figura 4.2: Diagrama de Clases del Gestor de Recursos

- ***Cargador***:
  - ***Cargar*()**: Carga los datos del recurso de uno o más ficheros de disco.
  - ***Descargar*()**: Libera la memoria ocupada por los datos del recurso.
  - **Obtener():datos**: Si el recurso está cargado, devuelve los datos que se han cargado.
  - ***Clonar*():datos**: Genera una copia de los datos.
  - ***Estado*():t_estado**: Devuelve el estado de carga del recurso.
- ***Instanciador***:
  - **Cargar()**: Envía a su ***Cargador*** la orden "Cargar".
  - **Descargar()**: Envía a su ***Cargador*** la orden "Descargar".
  - **Obtener() : objeto**: Pide los datos a su ***Cargador***, instancia una copia del objeto y la devuelve.
  - **Obtener_duplicado() : objeto**: Pide un duplicado de los datos a su ***Cargador***, instancia una copia del objeto y la devuelve.
  - ***Instanciar*(d:datos):objeto**: Crea una instancia del objeto a partir de los datos.
  - **Estado():t_estado**: Consulta a su ***Cargador*** el estado.
- **GestorRecursos**:
  - **Cargar(nombre:string)**: Comienza la carga del recurso con el nombre especificado.
  - **Descargar(nombre:string)**: Comienza la descarga del recurso con el nombre especificado.
  - **Estado(nombre:string):t_estado**: Consulta el estado del recurso con el nombre especificado.
  - **Obtener(nombre:string):objeto**: Devuelve una instancia del recurso.
  - **Obtener_duplicado(nombre:string):objeto**: Devuelve un duplicado de la instancia del recurso.

- **Forzar_obtener(nombre:string):objeto**: Si el recurso no está cargado, lo carga. Cuando termine la carga, lo obtiene.
- **Nuevo_instanciador(nombre:string, ins:Instanciador)**: Agrega un nuevo ***Instanciador*** al Gestor asignándole el nombre especificado.

Como es esperable, no se puede crear instancias de un objeto sin que sus datos hayan sido cargados, por ello se define para cada recurso un estado, que puede ser uno de los siguientes:

- **No_cargado**: El recurso no se encuentra cargado.
- **Cargando**: El recurso se está cargando.
- **Error_carga(string)**: Durante la carga ocurrió un error, descrito por la cadena de caracteres.
- **Cargado**: El recurso se encuentra cargado y disponible.
- **Descargando**: El recurso se está descargando.
- **Error_descarga(string)**: Durante la descarga ocurrió un error, descrito por la cadena de caracteres.

Con cada estado se definen las operaciones posibles que pueden ser llamadas estando el recurso en dicho estado:

- **No_cargado**:
  - **Cargar**: Comienza la carga del recurso.
- **Cargando**: El recurso está siendo cargado, no se puede hacer nada. Hay que esperar a que termine la fase de cargado y cambie el estado.
- **Error_carga(string)**:
  - **Cargar**: Reintenta la carga del recurso.
- **Cargado**:
  - **Descargar**: Comienza la descarga del recurso.
  - **Obtener**: Instancia y devuelve el objeto correspondiente al recurso.
  - **Obtener_duplicado**: Hace una copia profunda del recurso, instancia y devuelve un objeto creado con la copia.
- **Descargando**: El recurso está siendo descargando, no se puede hacer nada. Hay que esperar que termine la fase de descargado y cambie el estado.
- **Error_descarga(string)**:
  - **Descargar**: Reintenta la descarga del recurso.

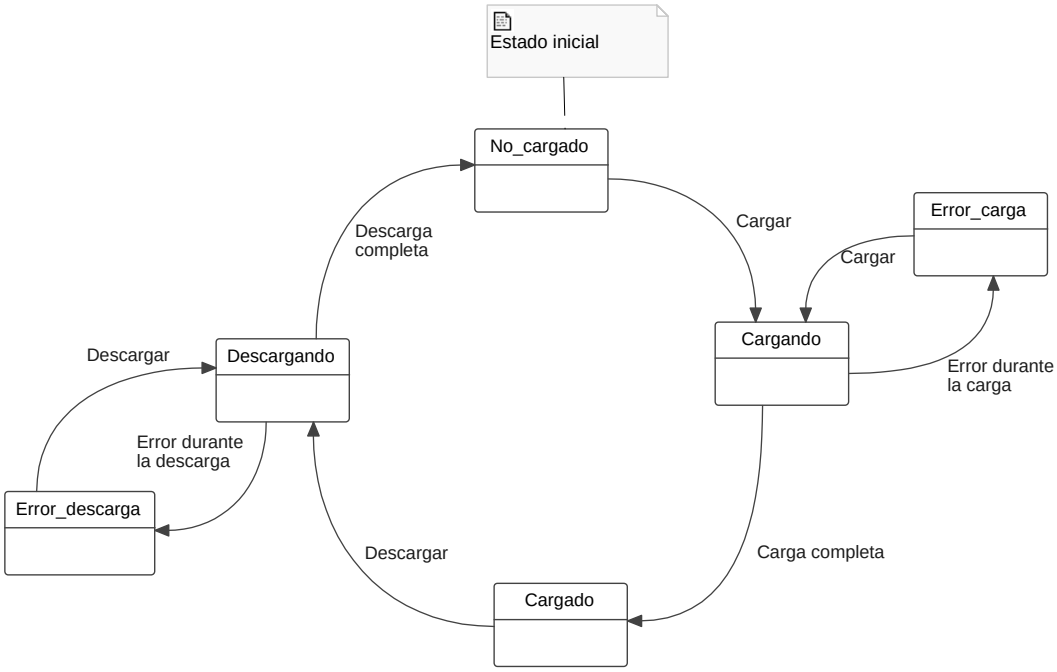

Figura 4.3: Diagrama de Estados de un recurso

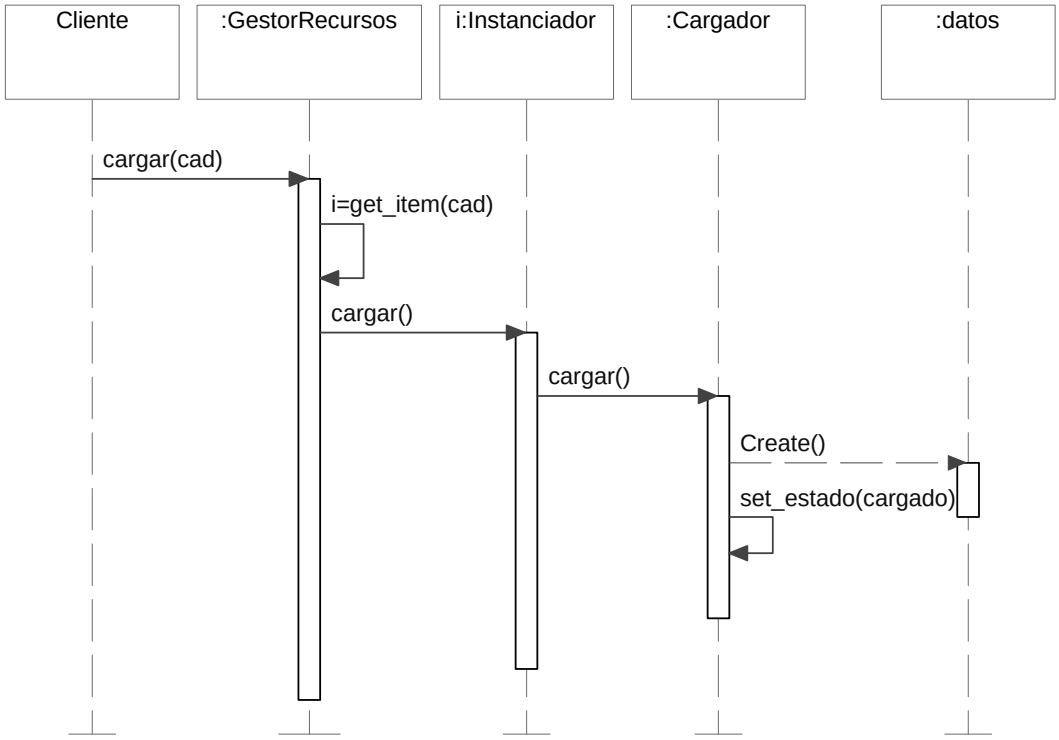

Figura 4.4: Diagrama de Secuencia de la operación de carga del Gestor de Recursos

4.2.2. Algoritmos

La operación de carga de un recurso es relativamente sencilla. Consiste esencialmente en el paso del mensaje de carga hasta su llegada al ***Cargador***, que procesa la petición.

1. El cliente ordena al **GestorRecursos** la carga del recurso con el nombre especificado.
2. El **GestorRecursos** busca el ***Instanciador*** asociado al nombre.
3. El **GestorRecursos** ordena cargar al ***Instanciador***.
4. El ***Instanciador*** ordena cargar a su ***Cargador*** asociado.
5. El ***Cargador*** lee los datos de los ficheros que tiene que leer y los copia a memoria.
6. El ***Cargador*** cambia su estado a "Cargado".

La operación de descarga es similar a la de carga. De nuevo el mensaje de descarga se propaga hasta llegar al ***Cargador***, que procesa la petición. Los detalles de dicha operación se pueden consultar en el DVD adjunto, en el documento `GestorRecursos/gestor_recursos.descargar.pdf`.

Una vez cargado un recurso, se puede obtener el objeto correspondiente al mismo. Dicha operación se realiza de la siguiente manera:

1. El cliente ordena al **GestorRecursos** la obtención del recurso con el nombre especificado.
2. El **GestorRecursos** busca el ***Instanciador*** asociado al nombre.
3. El **GestorRecursos** ordena obtener al ***Instanciador***.
4. El ***Instanciador*** pide los datos a su ***Cargador*** asociado.
5. El ***Instanciador*** ordena crear un objeto con los datos.
6. El ***Instanciador*** crea una instancia del objeto.

La operación "obtener_duplicado" sigue el mismo patrón que el resto de las operaciones. De nuevo el mensaje de obtención se propaga hasta llegar al ***Cargador***, que procesa la petición, fabrica un duplicado, y el instanciador lo convierte en un nuevo objeto. Los detalles de dicha operación se pueden consultar en el DVD adjunto, en el documento `GestorRecursos/gestor_recursos.obtener_duplicado.pdf`.

La consulta del estado sigue el mismo patrón que el resto de las operaciones. La consulta se propaga hasta llegar al ***Cargador***, que la responde. Los detalles de dicha operación se pueden consultar en el DVD adjunto, en el documento `GestorRecursos/gestor_recursos.estado.pdf`.

Puesto que la obtención de un recurso es aparentemente demasiado compleja, ya que requiere cargarlo y comprobar su estado antes de instanciarlo, se ha creado la operación "forzar_obtener" en el **GestorRecursos** que encapsula todas estas tareas. Los detalles de dicha operación se pueden consultar en el DVD adjunto, en el documento `GestorRecursos/gestor_recursos.forzar_obtener.pdf`.

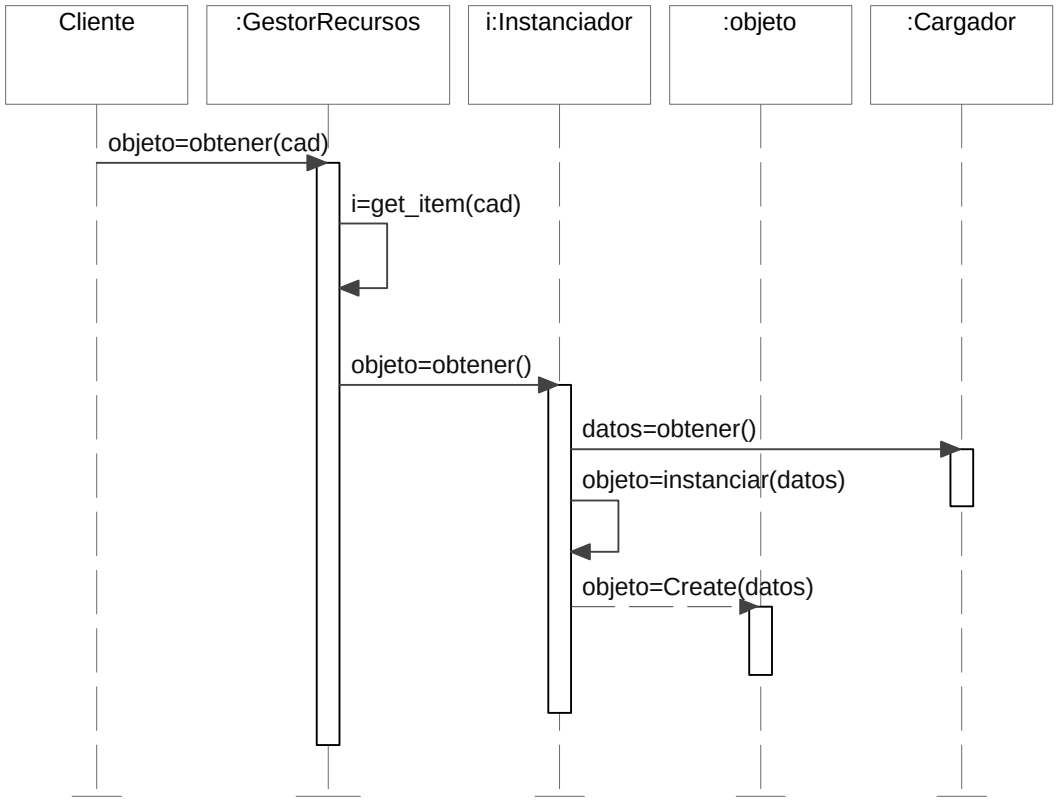

Figura 4.5: Diagrama de Secuencia de la obtención de un Recurso

Capítulo 5

Diseño de la Lógica del Juego

5.1. Modelo Estructural y Arquitectónico

La Lógica del Juego es el subsistema encargado de implementar las reglas del videojuego concreto. Está dividido en dos partes: un meta-sistema pensado para implementar entidades concurrentes, y las propias Reglas del Juego, implementadas usando concurrencia.

5.1.1. Modelo general

En el diagrama estructural se muestra la división de este subsistema en dos partes: una estructura genérica para implementar procesos, y un conjunto de procesos que dependerán del juego concreto.

5.1.2. Motor de Procesos

Provee un modelo básico de Proceso e implementa la lógica necesaria para manipularlos adecuadamente.

Interfaces ofrecidas

- **MotorProcesos**: Ofrece mecanismos para ejecutar los procesos de la Lógica del Juego y añadir nuevos.
- **Proceso**: Ofrece un modelo básico de proceso.

Interfaces utilizadas

- **GestorRecursos**: Utiliza el Gestor de Recursos para cargar las mallas y bitmaps requeridos.

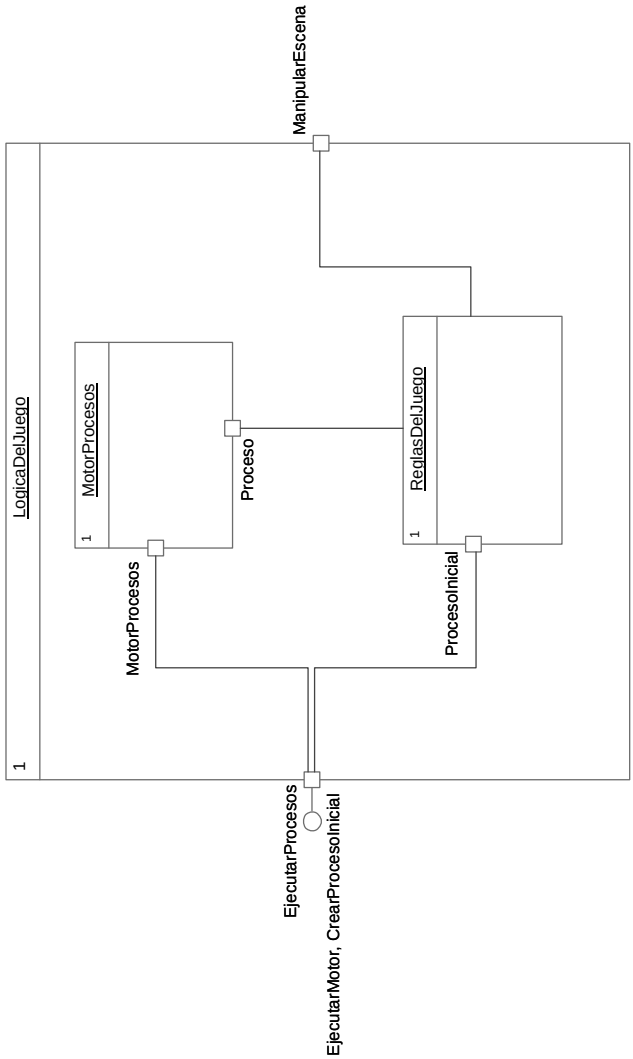

Figura 5.1: Diagrama estructural de la Lógica del Juego

5.1.3. Reglas del Juego

Implementa las reglas y la funcionalidad del juego, y su inicialización. Como tales, las Reglas del Juego no se definen en este punto, si no en cada juego concreto. Aquí se detalla únicamente el mecanismo de inicialización de las mismas.

Interfaces ofrecidas

- **ProcesoInicial**: Mecanismo de inicialización de las Reglas del Juego.

Interfaces utilizadas

- **MotorProcesos::Proceso**: Utiliza los procesos del Motor de Procesos.

5.2. Diseño detallado del Motor de Procesos

Para el Motor de Procesos se han tomado ciertas consideraciones y decisiones de diseño:

- **Agregación indirecta**: Para simplificar la ordenación global se ha decidido que Sistema agregue procesos indirectamente, a través de una nueva clase llamada Prioridad.
- **Siempre ordenado**: Sistema mantendrá todas sus instancias de Prioridad ordenadas en todo momento.
- **Cambios de estado al finalizar la ejecución de los procesos**: Los cambios de estado, de prioridad y las eliminaciones de los procesos muertos serán realizados al final de la ejecución de todos los procesos correspondientes a un fotograma. Esto significa que se tendrán dos listas de procesos: la lista que se utiliza en el fotograma actual y la que se prepara para el siguiente fotograma.
- **Separación por estados**: Con el fin de mejorar el rendimiento de la implementación de Prioridad, se ha decidido separar los procesos por estado, guardándose en diferentes listas.
- **Los procesos eliminados son purgados automáticamente**: No se guarda una lista de procesos eliminados, si no que sus referencias son eliminadas al terminar el fotograma, de tal manera que el colector de basura del lenguaje se encarga automáticamente de ellos.

5.2.1. Estructuras de datos

El Motor de Procesos ha sido separado en dos partes, con el fin de mejorar las dependencias internas. Estas dos partes son:

- Interfaces: Sistema, Prioridad, Proceso.
- Implementaciones: Sistema_impl, Prioridad_impl, Proceso_abstract.

En el diagrama se puede apreciar las varias listas de Procesos de cada Prioridad, así como las dependencias tratadas de tal manera que no son circulares. Las funciones y operaciones de cada clase son las siguientes:

- ***Sistema***:
  - ***ejecutar_procesos*()**: Llama al método "ejecutar" de cada proceso despierto por orden de prioridad.
  - ***ejecutar_dormidos*()**: Llama al método "ejecutar" de cada proceso dormido por orden de prioridad.
  - ***add_proceso*(p:Proceso)**: Agrega un proceso al sistema de procesos.
- ***Prioridad***:
  - ***ejecutar_procesos*()**: Llama al método "ejecutar" de cada proceso despierto.
  - ***ejecutar_dormidos*()**: Llama al método "ejecutar" de cada proceso dormido.
  - ***valor*():int**: Devuelve el valor de prioridad.
  - ***add_proceso*(p:Proceso)**: Agrega un proceso a la prioridad.
  - ***eliminar_proceso*(p:Proceso)**: Elimina un proceso de la prioridad.
  - ***pre_proceso*()**: Llamado antes de llamar a "ejecutar_procesos" o "ejecutar_dormidos".
  - ***post_proceso*()**: Llamado después de llamar a "ejecutar_procesos" o "ejecutar_dormidos".
- ***Proceso***:
  - ***ejecutar*()**: Punto de entrada al código contenido en el proceso.
  - ***ejecutar_dormido*()**: Punto de entrada al código contenido en el proceso, que será ejecutado cuando el proceso esté dormido.
  - ***senial*(s:t_senial)**: Envía una señal al proceso.
  - ***callback_despertar*()**: Llamado cuando el proceso recibe una señal "Despertar".
  - ***callback_dormir*()**: Llamado cuando el proceso recibe una señal "Dormir".
  - ***callback_congelar*()**: Llamado cuando el proceso recibe una señal "Congelar".
  - ***callback_matar*()**: Llamado cuando el proceso recibe una señal "Matar".
- **Sistema_impl**:
  - **ejecutar_procesos()**: Llama a los métodos "pre_proceso" de todas las prioridades, después a los métodos "ejecutar_procesos" y por último a los métodos "post_proceso".
  - **ejecutar_dormidos()**: Llama a los métodos "pre_proceso" de todas las prioridades, después a los métodos "ejecutar_dormidos" y por último a los métodos "post_proceso".
  - **add_proceso(p:Proceso)**: Busca la prioridad adecuada para el proceso, y lo agrega a dicha prioridad.

- **pre_proceso()**: Llama a los métodos "pre_proceso" de todas las prioridades en orden.
- **post_proceso()**: Llama a los métodos "post_proceso" de todas las prioridades en orden.
- **#obten_prioridad(i:int) : Prioridad**: Busca una ***Prioridad*** con el valor indicado. Si no la encuentra, instancia una nueva.

▪ **Prioridad_impl**:

- **valor() : int**: Devuelve el valor de la prioridad.
- **pre_proceso()**: No hace nada.
- **post_proceso()**: Guarda las colas de procesos modificadas en el fotograma actual como fijas para el fotograma siguiente.
- **add_proceso(p:Proceso)**: Consulta el valor de prioridad del proceso. Si dicho valor es el mismo que el de la prioridad, lo añade a una de sus listas de procesos. Si no, pasa la petición a su **Sistema**.
- **eliminar_proceso(p:Proceso)**: Busca y elimina el proceso de sus listas de procesos.
- **ejecutar_procesos()**: Llama al método "ejecutar" de los procesos cuyo estado es "Normal".
- **ejecutar_dormidos()**: Llama al método "ejecutar_dormido" de los procesos cuyo estado es "Dormido".

▪ ***Proceso_abstract***:

- **ejecutar_dormido()**: Por defecto no hace nada.
- **callback_despertar()**: Por defecto no hace nada.
- **callback_dormir()**: Por defecto no hace nada.
- **callback_congelar()**: Por defecto no hace nada.
- **callback_matar()**: Por defecto no hace nada.
- **senial(s:t_senial)**: Envía una señal al proceso.
- **set_prioridad(pr : int)**: Cambia el valor de prioridad del proceso, reconectando a otra ***Prioridad*** si es necesario.

▪ **Proceso_inicial**:

- **ejecutar()**: Inicializa las Reglas del Juego.

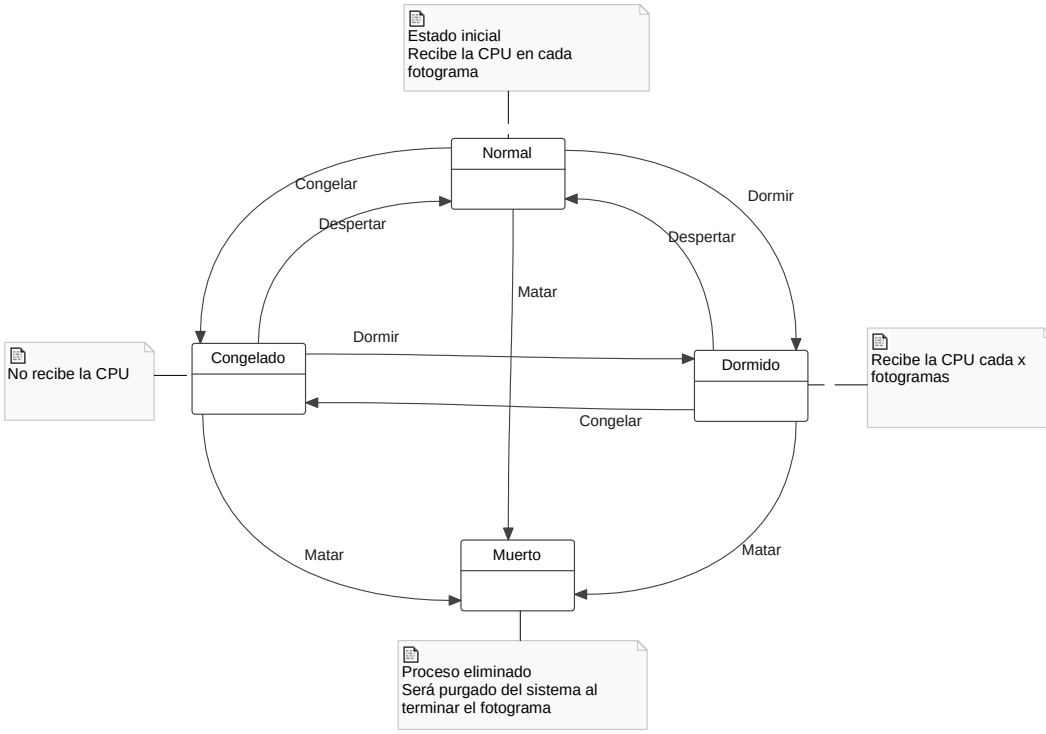

Figura 5.2: Diagrama de los estados posibles de un Proceso

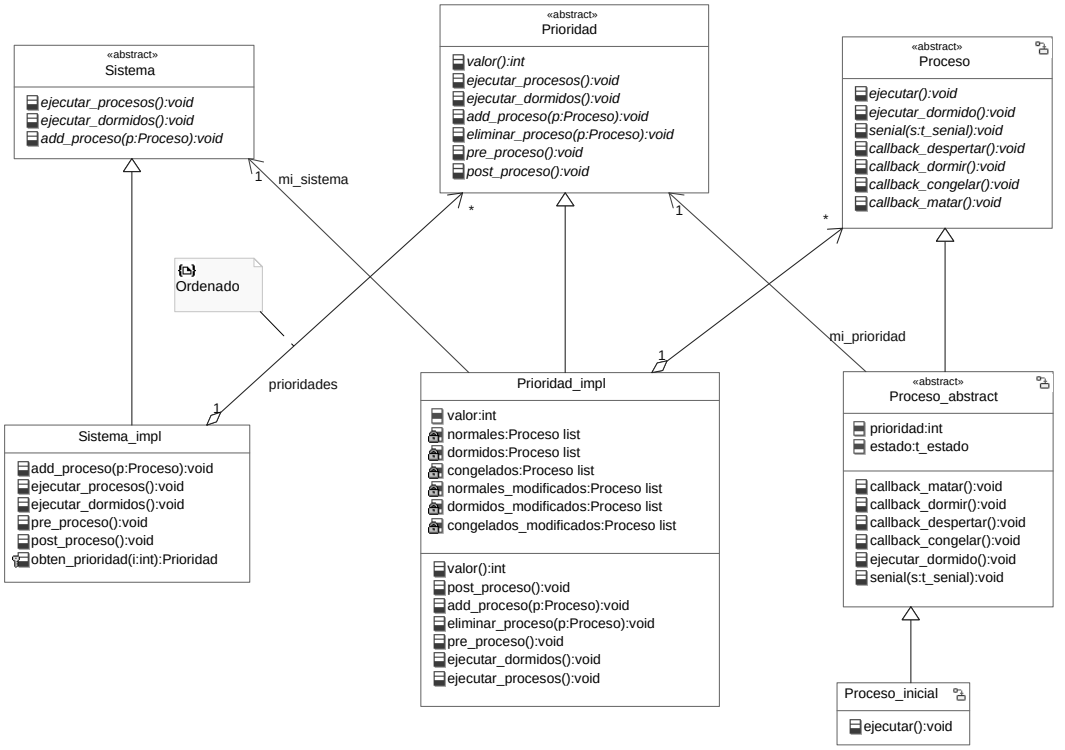

Figura 5.3: Diagrama de Clases del Motor de Procesos

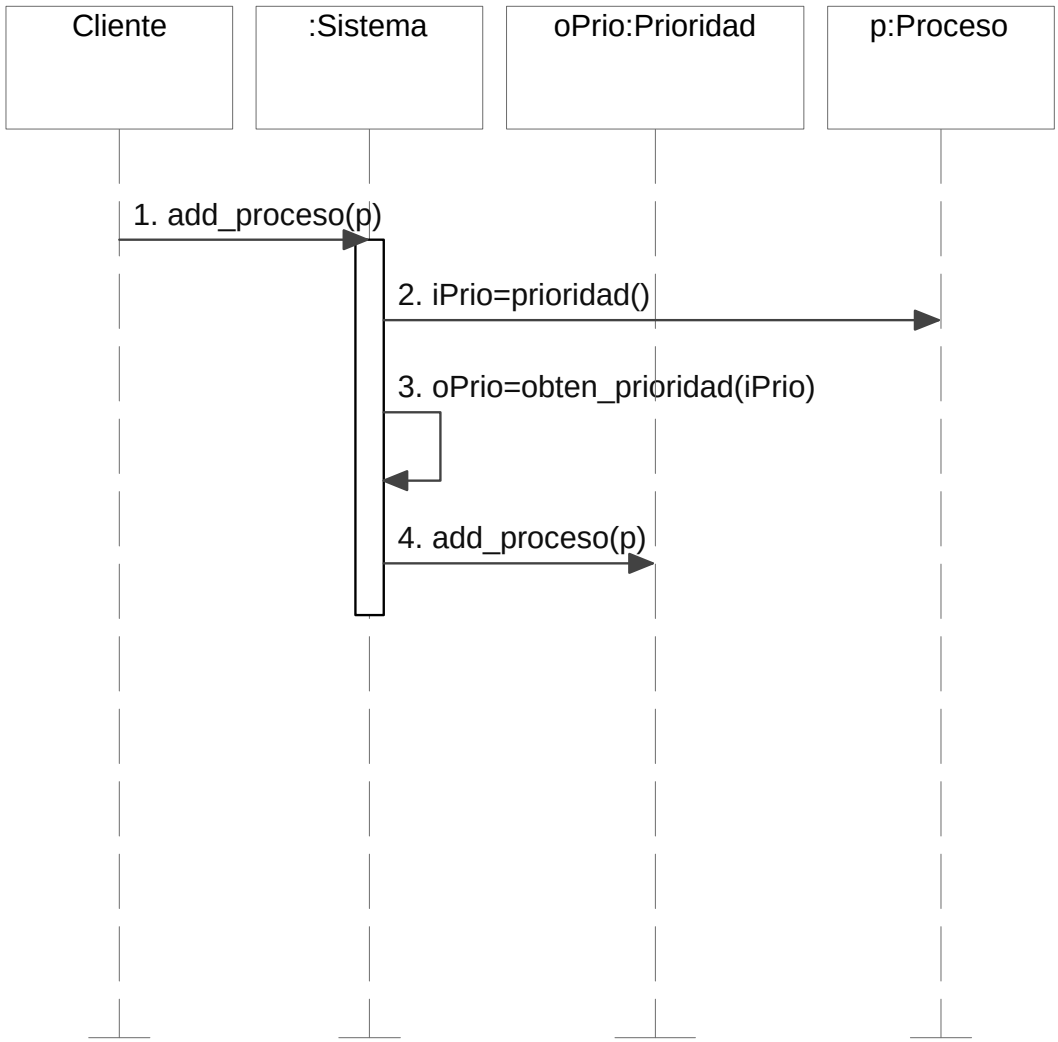

Figura 5.4: Diagrama de Secuencia de la operación "add_proceso" de la clase **Sistema_impl**

5.2.2. Algoritmos

Operaciones de Sistema

La operación "add_proceso" se encarga de introducir un proceso nuevo. Para ello, busca la prioridad adecuada y agrega el proceso a dicha prioridad. Los eventos que ocurren son los siguientes:

1. El cliente ordena al **Sistema** añadir el **Proceso**.
2. El **Sistema** consulta el valor de prioridad del **Proceso**.
3. El **Sistema** busca la **Prioridad** con el valor del **Proceso**.
4. El **Sistema** ordena a la **Prioridad** agregar el **Proceso**.

La operación "ejecutar_procesos" se encarga de dar una ráfaga de CPU a los procesos despiertos por el método de llamar a su operación "ejecutar". Todo ocurre mediante una colaboración entre **Sistema** y **Prioridad**. El flujo de eventos es el siguiente:

1. El Controlador ordena al **Sistema** ejecutar los procesos de la Lógica del Juego.
2. El **Sistema** llama a su método "pre_proceso".
   Para cada **Prioridad** en orden:
   3. El **Sistema** llama a "pre_proceso" de la **Prioridad**.

   Para cada **Prioridad** en orden:
   4. El **Sistema** llama a "ejecutar_procesos" de la **Prioridad**.
      Para cada **Proceso** cuyo estado sea "Normal" de la **Prioridad**:
      5. La **Prioridad** llama a "ejecutar" del **Proceso**.

6. El **Sistema** llama a su método "post_proceso".
   Para cada **Prioridad** en orden:
   7. El **Sistema** llama a "post_proceso" de la **Prioridad**.
   8. La **Prioridad** guarda su nueva lista de procesos normales para el siguiente fotograma.
   9. La **Prioridad** guarda su nueva lista de procesos dormidos para el siguiente fotograma.
   10. La **Prioridad** guarda su nueva lista de procesos congelados para el siguiente fotograma.

La operación "obten_prioridad" se encarga de buscar la **Prioridad** adecuada al valor especificado, creándola si no existe. La descripción detallada de su funcionamiento está disponible en el DVD adjunto, en el documento `MotorProcesos/motor_procesos.obten_prioridad.pdf`.

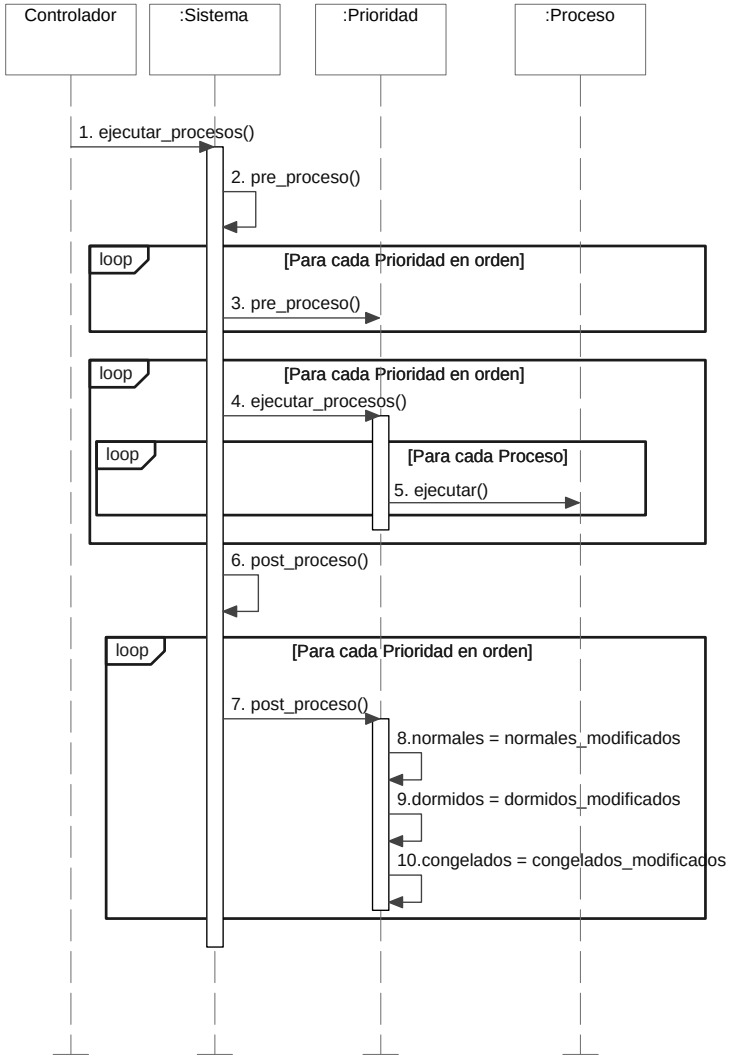

Figura 5.5: Diagrama de Secuencia de la operación "ejecutar_procesos" de la clase **Sistema_impl**

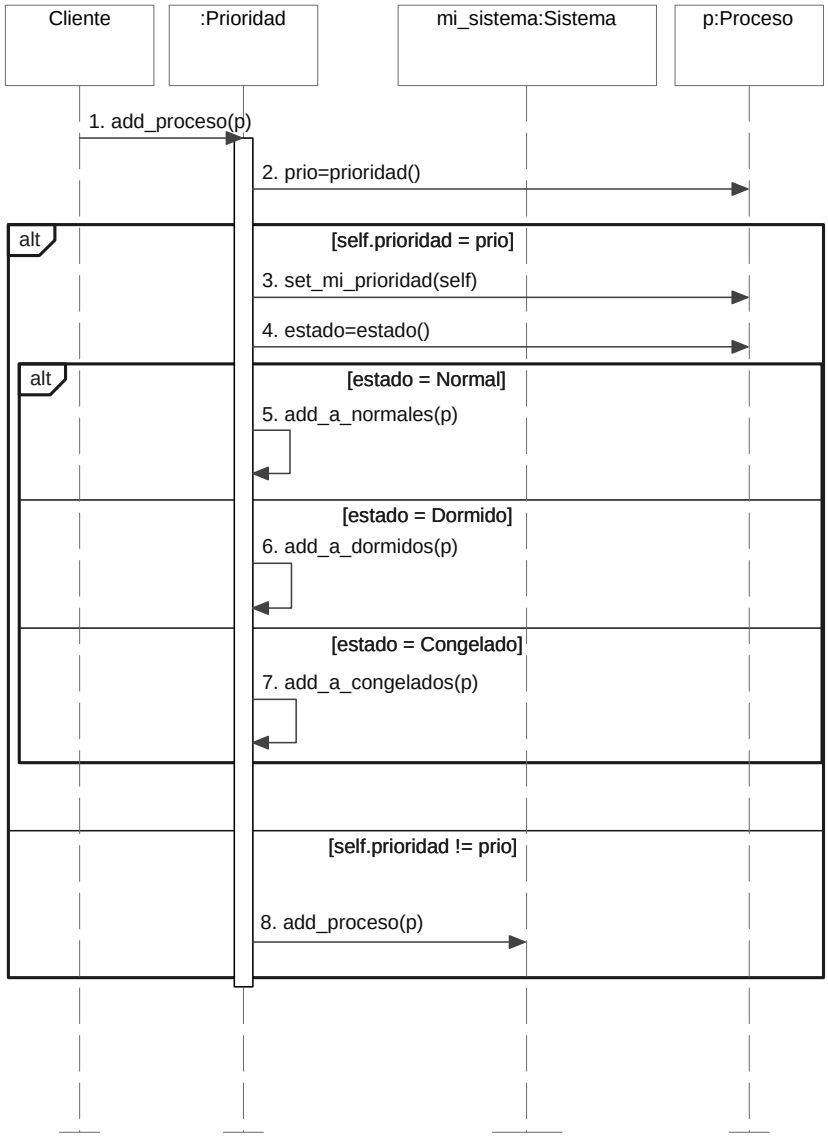

Figura 5.6: Diagrama de Secuencia de la operación "add_proceso" de la clase **Prioridad_impl**

Operaciones de Prioridad

Las dos operaciones principales de **Prioridad** que no han sido descritas indirectamente son las de añadir o eliminar un proceso. La operación "add_proceso" se encarga de añadir dicho proceso, o pasar la llamada a su **Sistema** si el valor de prioridad no coincide. El flujo de eventos es el siguiente:

1. El cliente ordena a la **Prioridad** añadir el **Proceso** especificado.
2. La **Prioridad** consulta el valor de prioridad del **Proceso**.
   Si dicho valor de prioridad coincide con el valor de la propia **Prioridad**:
   3. La **Prioridad** se conecta al **Proceso**.
   4. La **Prioridad** consulta el estado del **Proceso**.
      Si el estado es "Normal":
      5. La **Prioridad** introduce el **Proceso** en su lista modificada de procesos normales.

      Si el estado es "Dormido":
      6. La **Prioridad** introduce el **Proceso** en su lista modificada de procesos dormidos.

      Si el estado es "Congelado":
      7. La **Prioridad** introduce el **Proceso** en su lista modificada de procesos congelados.

   Si el valor de prioridad no coincide con el valor de la propia **Prioridad**:
   8. La **Prioridad** ordena a su **Sistema** agregar el **Proceso**.

La operación "eliminar_proceso" se encarga de la limpieza necesaria para desagregar un proceso de su **Prioridad**. La descripción detallada de su funcionamiento está disponible en el DVD adjunto, en el documento `MotorProcesos/motor_procesos.eliminar_proceso.pdf`.

Operaciones de Proceso

La operación "set_prioridad" se encarga de actuar ante el cambio de valor de prioridad de un proceso, buscando una nueva **Prioridad** adecuada para el mismo. La descripción detallada de su funcionamiento está disponible en el DVD adjunto, en el documento `MotorProcesos/motor_procesos.set_prioridad.pdf`.

La operación "lanzar" simplifica la introducción de nuevos procesos en el sistema de procesos. La descripción detallada de su funcionamiento está disponible en el DVD adjunto, en el documento `MotorProcesos/motor_procesos.lanzar.pdf`.

La operación "senial" se encarga de procesar las señales que reciben los procesos. La descripción detallada de su funcionamiento está disponible en el DVD adjunto, en el documento `MotorProcesos/motor_procesos.senial.pdf`.

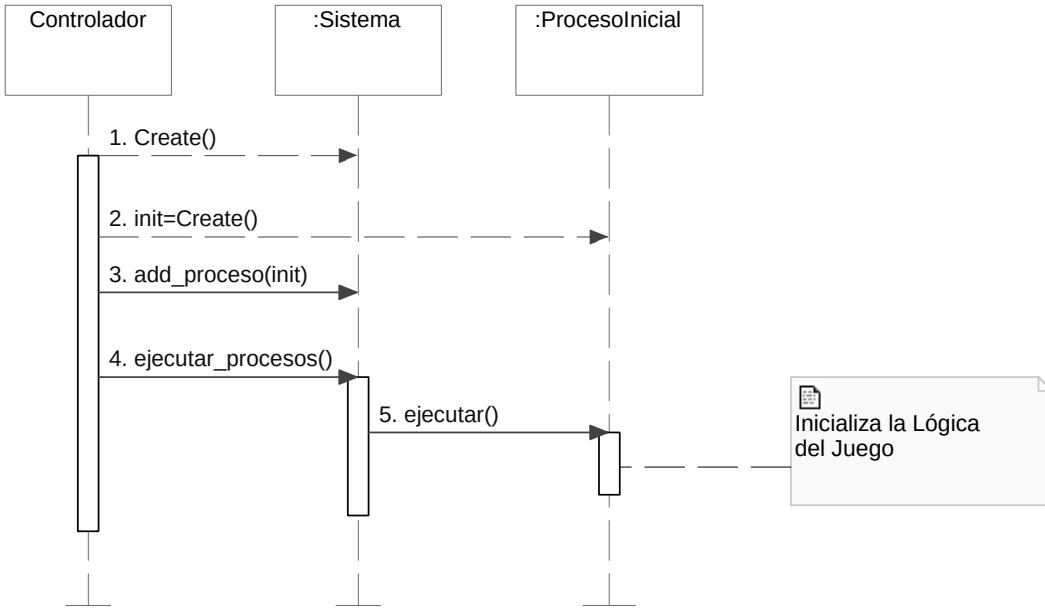

Figura 5.7: Diagrama de Secuencia de la inicialización de la Lógica del Juego

El modelo de inicialización del subsistema consiste en dos pasos: primero se crea un **Sistema**, y después se añade un **Proceso_inicial**. El **Proceso_inicial** se encargará del resto de la inicialización. El flujo de eventos es el siguiente:

1. El Controlador crea un nuevo **Sistema**.
2. El Controlador crea un nuevo **Proceso_inicial**.
3. El Controlador introduce el **Proceso_inicial** dentro del **Sistema**.
4. El Controlador comienza a ejecutar el **Sistema** como si se tratara de un fotograma cualquiera, y el **Proceso_inicial** se encarga del resto de la inicialización.

Capítulo 6

Prueba de concepto

6.1. Introducción

En los capítulos anteriores hemos descrito un conjunto de bibliotecas pensadas para ayudar al desarrollo de videojuegos. Dicha descripción estaría incompleta sin un ejemplo de su uso, que es lo que se describe en este capítulo. El ejemplo de uso que se ha escogido es la construcción de una aplicación informática que muestre un planetario sencillo.

6.1.1. El planetario

Las razones principales de escoger un planetario residen en que es lo suficientemente amplio como para demostrar las capacidades del sistema desarrollado, al tiempo que es suficientemente pequeño como para mantener cierta simplicidad. Dichas razones son las siguientes:

- **Visualización del Sistema Solar**: El planetario mostrará una vista del Sistema Solar, donde se podrá observar los diferentes objetos. Esto nos servirá para demostrar la utilidad del Motor Gráfico a la hora de mostrar gráficos en pantalla.
- **Múltiples objetos simultáneamente**: El planetario tendrá que controlar simultáneamente la órbita de varios planetas y sus satélites. Esto nos servirá para demostrar la utilidad del Motor de Procesos a la hora de manipular muchas entidades.
- **Objetos relativamente complejos, que deberán ser cargados de disco**: Los principales objetos del Sistema Solar son aproximadamente esféricos, y para ello se utilizarán esferas texturizadas para representarlos. Dichas esferas serán preparadas, construidas y texturizadas desde un editor 3D externo, y serán importadas dentro del planetario mediante el Gestor de Recursos.

Con el fin de mantener un planetario sencillo y realizable fácilmente con las bibliotecas desarrolladas, se han impuesto algunas limitaciones:

- **Muy baja interactividad**: Puesto que entre las bibliotecas anteriores no hemos desarrollado mecanismos de entrada de datos del usuario, no disponemos de dichos mecan-

ismos. Por tanto, el planetario se ejecutará sin interacción del usuario, excepto para determinar cuándo debe terminar la ejecución de la aplicación.

- **No a escala**: Ya que la interactividad es muy limitada, el usuario no dispondrá de mecanismos para seleccionar un objeto y observarlo en primer plano. Por ello será necesario mostrar todos los objetos y sus órbitas de tal manera que sean observables en el mismo encuadre. Eso significará que las órbitas estarán más juntas, y los objetos serán mucho más grandes.

6.2. Modelo de análisis del Planetario

6.2.1. Requisitos

- Requisito: **D.1**
  El planetario será un software de ejemplo que muestre cómo utilizar los subsistemas anteriores.

- Requisito: **D.2**
  El planetario representará los objetos siguientes:

  - Sol
    - Mercurio
    - Venus
    - Tierra
      - Luna
    - Marte
    - Júpiter
      - Io
      - Europa
      - Ganímedes
      - Calisto
    - Saturno
      - Titán
    - Urano
    - Neptuno
    - Plutón
      - Caronte

- Requisito: **D.3**
  Los objetos orbitarán siguiendo órbitas circulares, imitando el movimiento de translación de los planetas y satélites. La excepción es el Sol, que estará inmóvil en el centro.

- Requisito: **D.4**
  Los objetos girarán alrededor de su propio centro, imitando el movimiento de rotación de los planetas y satélites.

- Requisito: **D.5**
  Los satélites girarán alrededor de su planeta correspondiente, esto es:

  - La Luna orbita alrededor de la Tierra.
  - Io, Europa, Ganímedes y Calisto orbitan alrededor de Júpiter.
  - Titán orbita alrededor de Saturno.
  - Caronte orbita alrededor de Plutón.

- Requisito: **D.6**
  Todos los objetos serán representados como esferas con textura, con la excepción de Saturno y Urano, en los que además de la esfera estarán representados los anillos.

- Requisito: **D.7**
  El planetario tendrá un carácter ilustrativo. Se permite cierta flexibilidad a la hora de implementar algunas características de los planetas y órbitas:

  - Los objetos tendrán un tamaño relativo siguiendo el espíritu de la realidad, pero sin copiarla. Los objetos *grandes* (Júpiter, Saturno, el Sol) serán significativamente más grandes que los demás, mientras que los objetos *pequeños* (Mercurio, Plutón, satélites) serán significativamente más pequeños que los demás.
  - Los períodos orbitales serán aproximados, y siguen el mismo espíritu que el punto anterior. Se propone que el período orbital de la Tierra sea entre 1 y 5 segundos, y vaya aumentando en función de la distancia al objeto orbitado.
  - La velocidad de rotación de los objetos tampoco es crítica, y seguirá el espíritu del resto de la aplicación.

- Requisito: **D.8**
  Los objetos deben tener un tamaño proporcionado, con el fin de que sean visibles. Por ejemplo, el Sol debe ser razonablemente grande, pero sin ocupar toda la vista, y Plutón debe ser razonablemente pequeño, pero sin que sea casi invisible.

- Requisito: **D.9**
  La aplicación mostrará el planetario con los diferentes objetos girando y orbitando hasta que el usuario pulse la tecla *esc* o cierre el programa.

- Requisito: **DN.1**
  El planetario será implementado usando los subsistemas desarrollados anteriormente.

6.2.2. Modelos del sistema

El planetario simulará dos fenómenos: el movimiento de rotación y el de translación u órbita. Es necesario separarlos ya que el Sol, que actúa como centro, no orbita alrededor de ningún otro objeto; y por tanto, no tiene movimiento de translación asociado. El movimiento de rotación es razonablemente sencillo. Consiste en el giro continuo del objeto alrededor de sí mismo. Puesto que estamos construyendo un planetario sencillo, podemos tomar algunas simplificaciones:

- **Eje y velocidad constantes**: Haremos constantes tanto el eje como la velocidad de rotación.
- **Mismo eje para todos los objetos**: Al contrario de lo que dicen los astrónomos, consideraremos que todos los objetos del planetario tienen el mismo eje de rotación.

El movimiento de translación es más complejo, ya que depende de más factores. En nuestro caso, simplificaremos utilizando órbitas circulares, que estarán todas situadas en el mismo plano. Para calcular la velocidad de desplazamiento de los diferentes objetos, nos basaremos en las ecuaciones de Kepler. En concreto, partiendo de la tercera ecuación, que dicta que "Para cualquier planeta, el cuadrado de su período orbital (tiempo que tarda en dar una vuelta alrededor del Sol) es directamente proporcional al cubo de la longitud del semieje mayor a de la órbita elíptica", y simplificando a órbitas circulares, obtenemos que:
$\frac{T^2}{r^3} = K$ Donde:

- **T**: es el período orbital.
- **r**: es el radio de la órbita.
- **K**: es una constante en nuestro sistema, que depende de la masa del Sol, de la Constante de Gravitación universal y de otras constantes.

Reescribiendo adecuadamente la ecuación, obtenemos:
$v = \frac{1}{T} = \sqrt{r^3 \cdot K}$ Donde:

- **v**: es la velocidad orbital en rad/s

A partir de esta última ecuación podemos construir un esquema de velocidades orbitales realista.

6.2.3. Modelos objeto

Las clases que componen el Diagrama de clases son las siguientes:

- **Objeto**: Representa un lugar en el espacio.
- **Centro_gravedad**: Representa un punto en el espacio alrededor del cual orbitan otros objetos.
- **Planeta**: Representa un objeto espacial, ya sea un planeta, un satélite o el propio Sol.

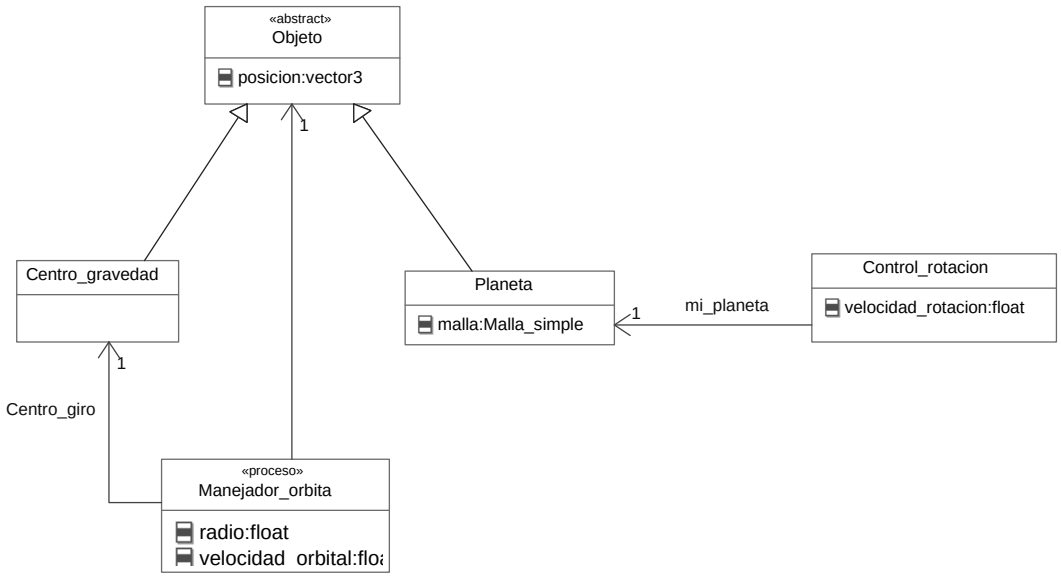

Figura 6.1: Diagrama de Clases del Planetario

- **Manejador_orbita**: Manipula la posición de un **Objeto** para que gire alrededor de un **Centro_gravedad**.

- **Control_rotacion**: Manipula la rotación de un **Orbitante** para que gire alrededor de sí mismo.

El Diagrama de Clases es razonablemente sencillo, pero requiere algunas explicaciones:

- Se ha separado los movimientos de los objetos, ya que hay un objeto (el Sol) que, a diferencia de los demás, no se le puede aplicar el movimiento de translación. Además, esta separación nos permitirá implementar configuraciones más complejas en el futuro, como sistemas duales en los cuales dos objetos de masa similar giran alrededor de un centro de gravedad común (estrellas dobles, o el sistema Plutón-Caronte).

- Se han organizado los objetos siguiendo una jerarquía similar a la del nivel de Objetos del Motor Gráfico (Clases **Objeto**, **Centro_gravedad** y **Orbitante**). Esto nos ayudará a reutilizar esa parte ya desarrollada.

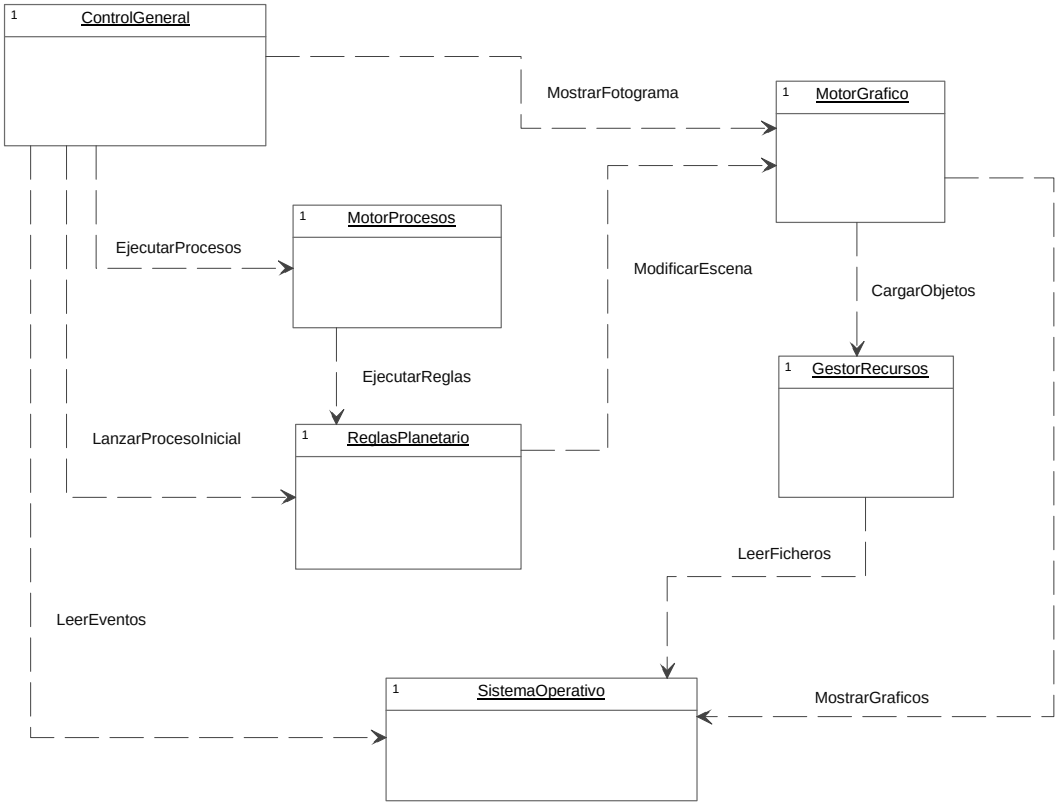

Figura 6.2: Diagrama de la Estructura del Planetario

6.3. Modelo de diseño del Planetario

6.3.1. Modelo estructural y arquitectónico

El diagrama estructural del planetario es similar al visto en el Modelo de Análisis, pero tiene algunas diferencias menores:

- La Lógica del Juego ha sido separada en sus dos componentes:
  - El Motor de Procesos, que ha sido desarrollado en un capítulo anterior.
  - Las Reglas del Planetario, que están descritas en la sección anterior.
- Se han ocultado las bibliotecas e interfaces externas utilizadas. Dichas entidades están reunidas dentro del bloque Sistema Operativo.

Los bloques son los descritos en los capítulos anteriores, con excepción de las Reglas del Planetario y del Controlador, que serán descritos en esta sección.

6.3.2. Reglas del Planetario

Implementa la lógica y las normas concretas del planetario, tales como trayectoria de los planetas, posición y tamaño.

Interfaces utilizadas

- **MotorGrafico::ModificarEscena** : Utiliza el Motor Gráfico para manipular los objetos de la escena y mostrar el planetario.

Interfaces ofrecidas

- **EjecutarReglas** : El Motor de Procesos llama a los procesos del planetario.
- **LanzarProcesoInicial** : El Controlador crea un **Proceso_inicial** que inicializa las Reglas del Planetario.

6.3.3. Controlador General

El Controlador se encarga del control general de la aplicación. Contiene las llamadas para la inicialización general, así como el bucle principal de la aplicación.

Interfaces utilizadas

- **SistemaOperativo::LeerEventos** : El Controlador recibe los eventos de la aplicación a través de esta interfaz.
- **ReglasPlanetario::LanzarProcesoInicial** : El Controlador inicializa las Reglas del Planetario creando un **Proceso_inicial**.

- **MotorProcesos::EjecutarProcesos** : El Controlador ejecuta las Reglas del Planetario utilizando el Motor de Procesos.
- **MotorGrafico::MostrarFotograma** : El Controlador muestra el resultado de las Reglas del Planetario a través del Motor Gráfico.

6.4. Diseño detallado de la Lógica del Juego

Puesto que se reutilizará el sistema desarrollado, y con el fin de simplificar los diagramas de clases, se crea un nuevo estereotipo:

- «**proceso**»: Aplicable a clasificadores. Denota una clase que es una especificación de la clase ***Proceso_abstract*** declarada en el Motor de Procesos.

6.4.1. Estructuras de datos

El Modelo de Diseño muestra unas diferencias significativas. Dichas diferencias son:

- Las clases **Manejador_orbita** y **Control_rotacion** han sido reconvertidas en procesos.
- Las clases ***Objeto***, **Centro_gravedad** y **Planeta** han sido sustituidas por sus equivalentes del Nivel de Objetos del Motor Gráfico:
  - ***Objeto*** es ahora Objetos::***Objeto***.
  - **Centro_gravedad** es ahora Objetos::**Dummy**.
  - **Planeta** es ahora Objetos::**Objeto_malla**.
- **Control_rotacion** ha sido renombrado a **Planeta**.
- Se ha añadido un **Manejador_camara**, que se encargará de manipular la cámara desde la que se observa el planetario.
- Se ha añadido un **Proceso_inicial**, que se encargará de inicializar el planetario.

Con estos cambios la Lógica del Juego utiliza los demás sistemas adecuadamente. Las tareas que realizan los procesos son los siguientes:

- **Manejador_orbita**: Crea un **Dummy** y lo hace girar a velocidad constante alrededor de un objeto.
- **Planeta**: Crea un **Objeto_malla**, lo asigna la malla que recibe como parámetro y lo hace girar a velocidad constante alrededor de sí mismo.
- **Manejador_camara**: Manipula una **Camara_perspectiva**, y hace que se mueva suavemente mientras apunta hacia el centro de la escena.
- **Proceso_base**: Carga todas las mallas necesarias, prepara la escena y crea todas las instancias de **Manejador_orbita**, **Planeta** y **Manejador_camara**. Nos referiremos a este proceso como **Proceso_inicial**.

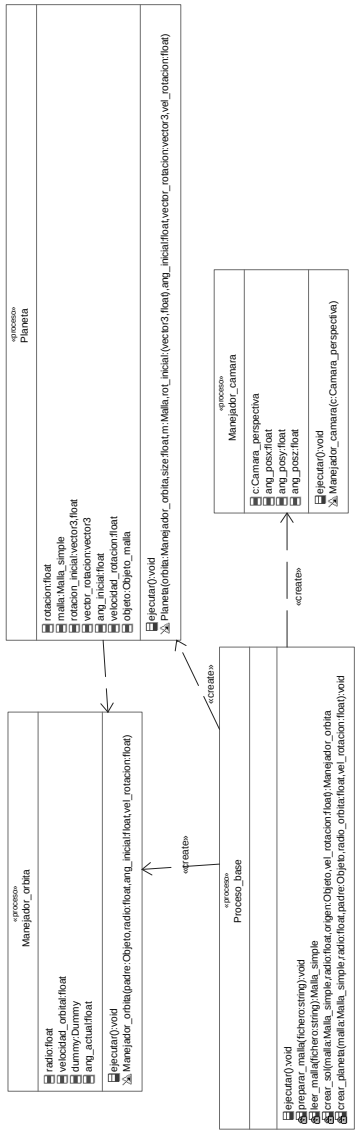

Figura 6.3: Diagrama de Clases de Diseño del Planetario

6.4.2. Valores iniciales

El planetario contiene una gran cantidad de parámetros y constantes, correspondientes a los radios orbitales, tamaños de planeta, velocidad de rotación, etc. Esos valores se detallan a continuación:

Objeto	Radio	Padre	Radio orbital	Velocidad de rotación
Sol	3	Origen	0	0.02
Mercurio	0.3	Sol	4.1	0.04
Venus	0.5	Sol	5.6	0.03
Tierra	1	Sol	9.3	0.3
Luna	0.4	Tierra	1.7	0.3
Marte	0.7	Sol	12.5	0.2
Jupiter	2	Sol	20.54	0.1
Io	0.25	Jupiter	6.54	0.02
Ganímedes	0.3	Jupiter	5,74	0.02
Europa	0.25	Jupiter	5	0.02
Calisto	0.2	Jupiter	4.5	0.02
Saturno	1.5	Sol	33.58	0.1
Titán	0.5	Saturno	4	0.1
Urano	1.2	Sol	38.98	0.1
Neptuno	1.1	Sol	42.18	0.1
Plutón	0.35	Sol	44.88	0.1
Caronte	0.3	Plutón	1.1	0.1

6.4.3. Algoritmos

La implementación de una Lógica del Juego como procesos requiere un reenfoque de la implementación. Los procesos son entidades que reciben el uso de la CPU una vez en cada fotograma; y, por ello, deben hacer sus cálculos con dicha referencia temporal. Los algoritmos son así modificados, de tal manera que típicamente, en cada fotograma, cada proceso realiza las siguientes tareas:

1. Incremento de las variables locales que llevan una referencia temporal: *tiempo_pasado*, *angulo_girado* y similares.

2. Aplicación de los nuevos valores de dichas variables para generar el estado actual del juego.

Con la excepción del **Proceso_inicial**, todos los procesos siguen este patrón.

El método "ejecutar" del **Manejador_camara** se encarga de mover la cámara suavemente en una región del espacio. Para ello, utiliza la función seno, y va aumentando el ángulo a medida que pasa el tiempo. Además, para mantener en el centro de la escena el Sol, se modifica la rotación de la cámara, compensando el movimiento. Las fórmulas utilizadas son:

- $Posicion = (x, y, z)$
- $x = 6 \cdot sin(ang\_posx)$

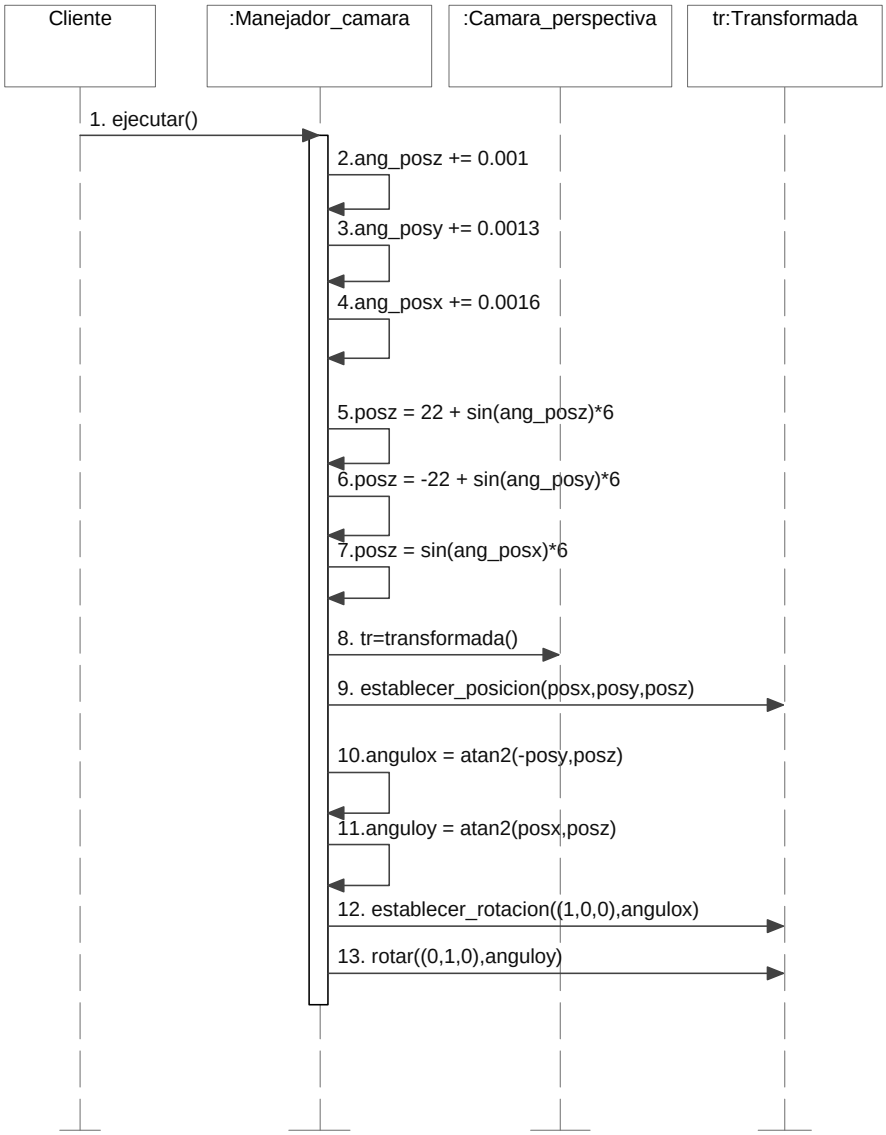

Figura 6.4: Diagrama de Secuencia de la operación "ejecutar" del **Manejador_camara**

- $y = -22 + 6 \cdot sin(ang\_posy)$
- $z = 22 + 6 \cdot sin(ang\_posz)$
- $ang\_posx = 0{,}0016 \cdot t$
- $ang\_posy = 0{,}0013 \cdot t$
- $ang\_posz = 0{,}001 \cdot t$
- $angulox = atan2(\frac{-posy}{posz})$
- $anguloy = atan2(\frac{posx}{posz})$

Donde t es el tiempo medido en fotogramas. Con estas fórmulas, la cámara se mueve en la región delimitada por:

- $-6 <= x <= 6$
- $-28 <= y <= -16$
- $16 <= z <= 28$

En el diagrama de secuencia se puede observar la aplicación de las fórmulas anteriores. Los pasos que ocurren son los siguientes:

1. El cliente ordena la ejecución del proceso **Manejador_camara**.
2. El **Manejador_camara** incrementa su variable "ang_posz" en 0.001.
3. El **Manejador_camara** incrementa su variable "ang_posy" en 0.0013.
4. El **Manejador_camara** incrementa su variable "ang_posx" en 0.0016.
5. El **Manejador_camara** calcula la componente z de la nueva coordenada de la cámara.
6. El **Manejador_camara** calcula la componente y de la nueva coordenada de la cámara.
7. El **Manejador_camara** calcula la componente x de la nueva coordenada de la cámara.
8. El **Manejador_camara** obtiene la transformada de la cámara.
9. El **Manejador_camara** establece la nueva coordenada de la cámara.
10. El **Manejador_camara** calcula la rotación en el eje x de la cámara.
11. El **Manejador_camara** calcula la rotación en el eje y de la cámara.
12. El **Manejador_camara** reinicia la rotación de la cámara estableciéndola por defecto en el valor obtenido anteriormente para el eje x.
13. El **Manejador_camara** aplica la rotación en el eje y a la cámara.

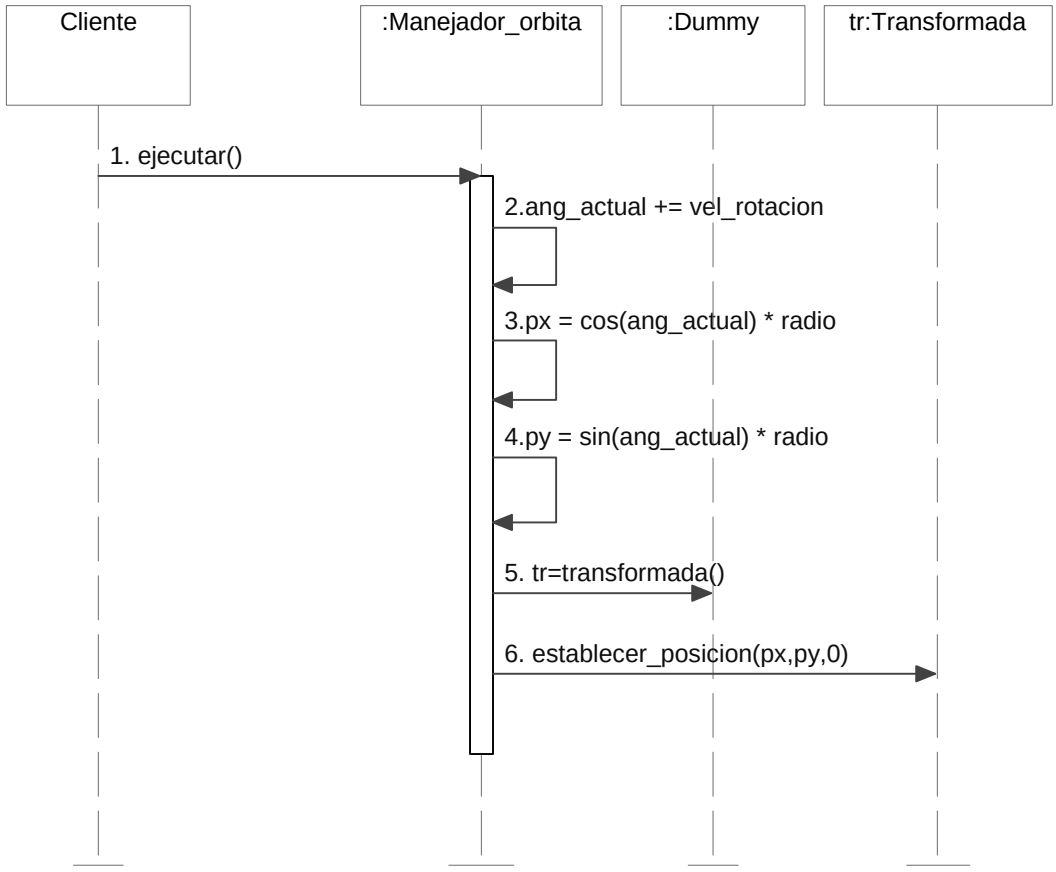

Figura 6.5: Diagrama de Secuencia de la operación "ejecutar" del **Manejador_orbita**

La función principal del **Manejador_orbita** consiste en manipular la posición de un **Dummy** de tal manera que describa una trayectoria circular. Puesto que al emparentar un Objeto con otro, la transformada del objeto hijo se aplica con respecto a la del padre; al emparentar el **Dummy** a otro objeto y siendo controlador por un **Manejador_orbita**, dicho **Dummy** girará alrededor del objeto padre. Las fórmulas utilizadas para la órbita son la aplicación directa de trigonometría básica. Dichas fórmulas son:

- $Posicion = (x, y, 0)$
- $x = cos(ang\_actual) \cdot r$
- $y = sin(ang\_actual) \cdot r$
- $ang\_actual = vel\_rotacion \cdot t$

Donde r es el radio de la órbita y t es el tiempo medido en fotogramas.

En el diagrama de secuencia se puede observar la aplicación de las fórmulas anteriores. Los pasos que ocurren son los siguientes:

1. El cliente ordena la ejecución del proceso **Manejador_orbita**.
2. El **Manejador_orbita** incrementa su variable "ang_actual" en la velocidad de translación.
3. El **Manejador_orbita** calcula el componente x de la nueva coordenada.
4. El **Manejador_orbita** calcula el componente y de la nueva coordenada.
5. El **Manejador_orbita** obtiene la transformada de su **Dummy**.
6. El **Manejador_orbita** establece la nueva posición de su **Dummy**.

La inicialización del **Manejador_orbita** es relativamente sencilla, y consiste en la preparación de un **Dummy**. Los pasos que ocurren son los siguientes:

1. El **Proceso_inicial** crea un nuevo **Manejador_orbita**.
2. El **Manejador_orbita** crea un nuevo **Dummy**.
3. El **Manejador_orbita** emparenta el **Dummy** al objeto padre que recibió como parámetro.

La función principal del **Planeta** consiste en hacer girar su **Objeto_malla** asociado. La fórmula utilizada es trivial, y consiste en incrementar el ángulo girado con el tiempo transcurrido. Los pasos que ocurren en el diagrama de secuencia son:

1. El cliente ordena la ejecución del proceso **Planeta**.
2. El **Planeta** incrementa su variable "ang_actual" en la velocidad de rotación.
3. El **Planeta** obtiene la transformada de su **Objeto_malla**.
4. El **Planeta** reinicia la rotación de su **Objeto_malla**.
5. El **Planeta** rota el **Objeto_malla** en "ang_actual".

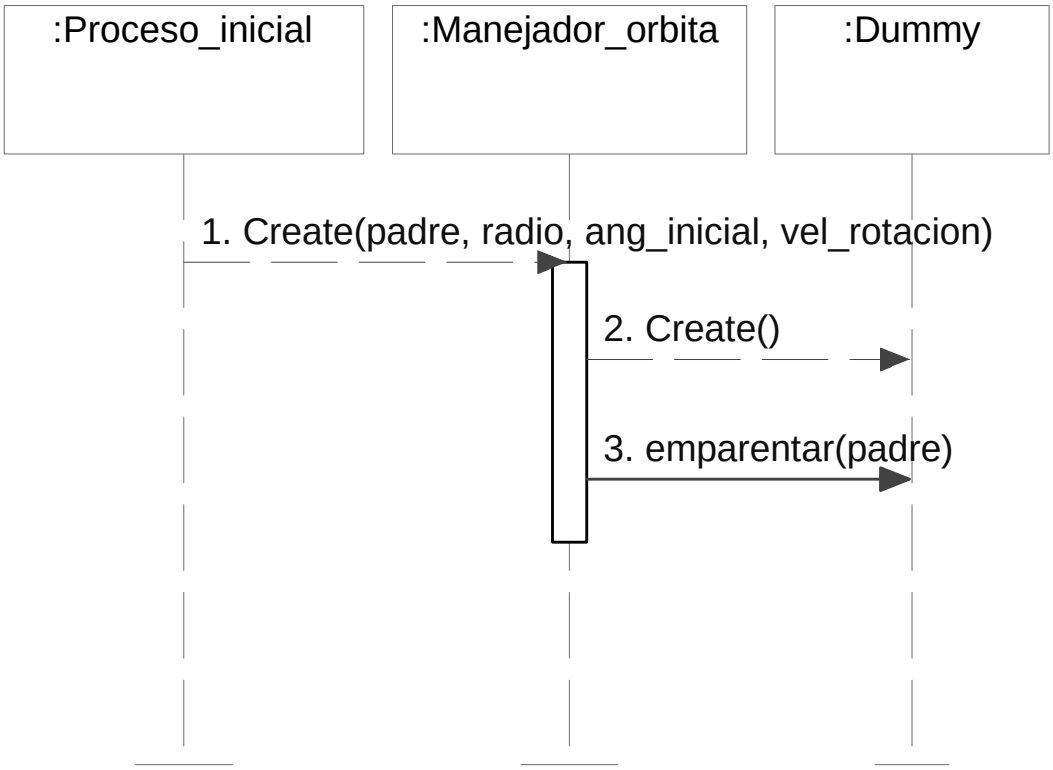

Figura 6.6: Diagrama de Secuencia de la inicialización del **Manejador_orbita**

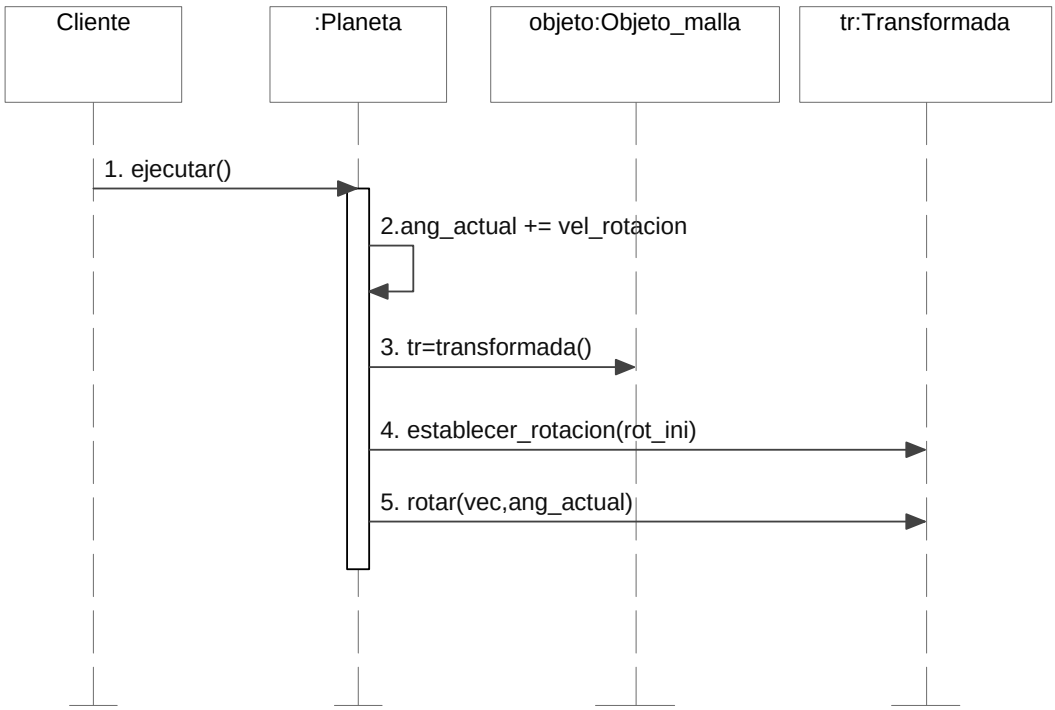

Figura 6.7: Diagrama de Secuencia de la operación "ejecutar" del **Planeta**

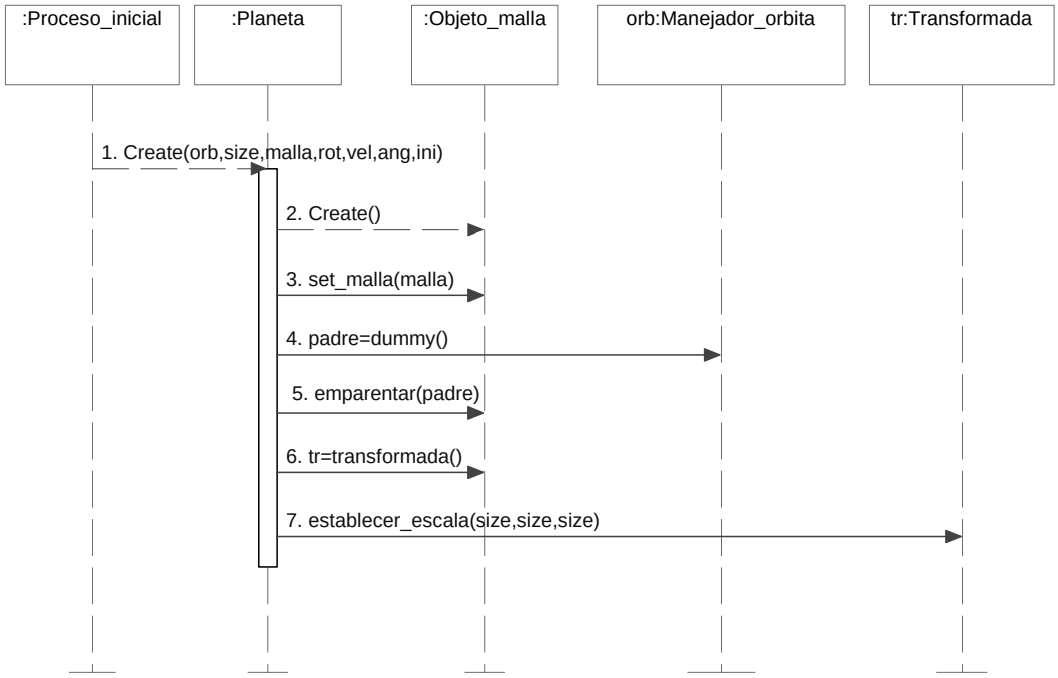

Figura 6.8: Diagrama de Secuencia de la inicialización del **Planeta**

La inicialización de **Planeta** se encarga de crear y configurar su **Objeto_malla** adecuadamente. Los pasos que ocurren son los siguientes:

1. El **Proceso_inicial** crea un nuevo **Planeta**.
2. El **Planeta** crea un nuevo **Objeto_malla**.
3. El **Planeta** asigna la malla del **Objeto_malla** con uno de los parámetros recibidos.
4. El **Planeta** obtiene el **Dummy** de la órbita del planeta.
5. El **Planeta** emparenta su **Objeto_malla** al **Dummy**.
6. El **Planeta** obtiene la transformada de su **Objeto_malla**.
7. El **Planeta** asigna el nuevo tamaño de su **Objeto_malla**.

El **Proceso_inicial** se encarga de la inicialización de la Lógica del Juego, que en este caso consiste en la carga de mallas e instanciación de los demás procesos necesarios. Su tarea principal está recogida en la operación "ejecutar", que utiliza las operaciones "leer_malla", "crear_planeta" y "crear_sol" como sub-operaciones. La descripción detallada de su funcionamiento está disponible en el DVD adjunto, en el documento `Planetario/planetario.proceso_inicial.pdf`.

6.5. Diseño detallado del Controlador General

El Controlador General se encarga de inicializar y dirigir la aplicación desde un punto de vista de alto nivel.

6.5.1. Estructuras de datos

Las clases del Controlador General cumplen la función de inicializar y controlar las llamadas principales de la aplicación. A continuación se describe la función concreta de cada clase, y sus operaciones:

- **Main**: Inicializador general y punto de entrada de la aplicación.
  - **Inicializar_SDL()** : Inicializa la biblioteca SDL.
  - **Detener_SDL()** : Llama a la función de limpieza y terminación de la biblioteca SDL.
  - **inicializar_motor_grafico()**: Llama a la operación "Inicializar" del Motor Gráfico.
  - **detener_motor_grafico()**: Llama a la operación "terminar" del Motor Gráfico.
  - **inicializar_motor_sonido()**: Inicializaría el Motor de Sonido, pero, puesto que aún no hay ninguno implementado, esta operación no hace nada.
  - **detener_motor_sonido()**: Detendría el Motor de Sonido, pero, puesto que aún no hay ninguno implementado, esta operación no hace nada.

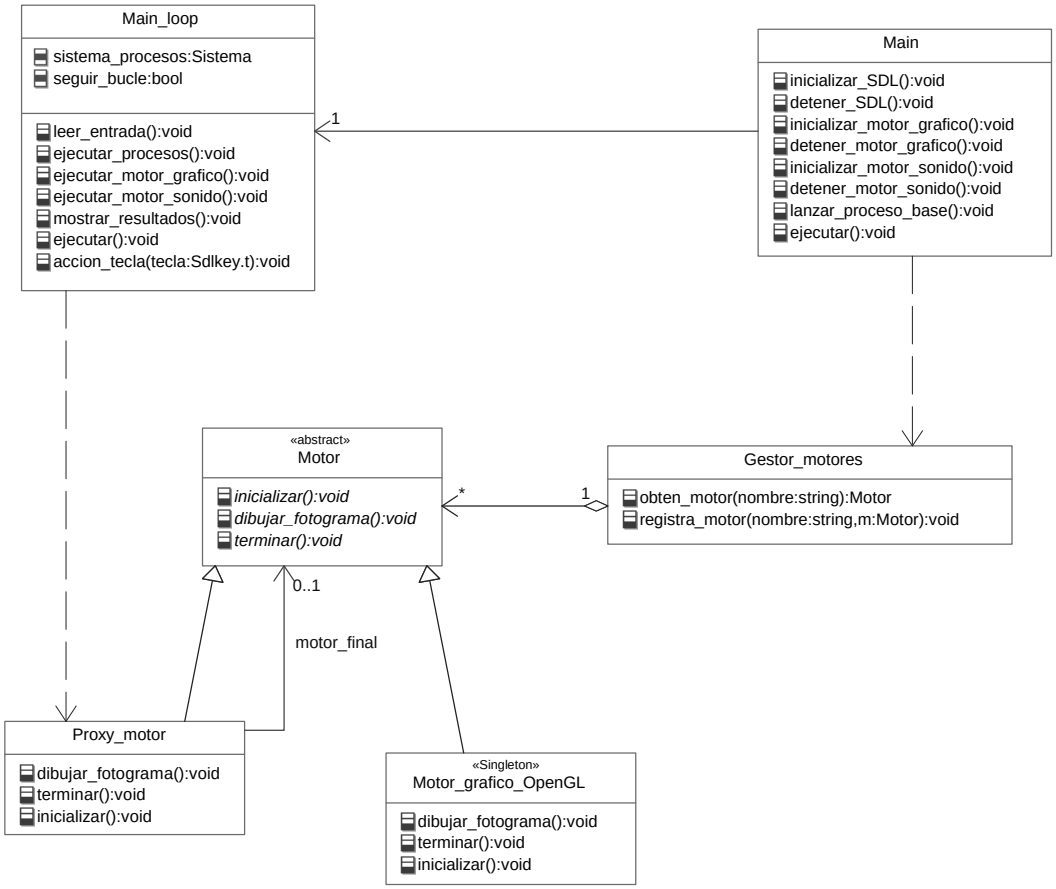

Figura 6.9: Diagrama de Clases de Diseño del Controlador General

- **lanzar_proceso_base()**: Crea un **Proceso_inicial** y lo introduce en el Motor de Procesos.
- **ejecutar()**: Inicializa los diferentes subsistemas, crea un bucle principal, lo ejecuta, y después detiene los subsistemas.

- **Main_loop**: Bucle principal de la aplicación.
  - **leer_entrada()** : Lee todos los eventos recibidos por la aplicación y los procesa.
    - Si recibe un evento "QUIT" ordena finalizar el bucle principal.
    - Si recibe un evento "KEYDOWN" llama a "accion_tecla" con el identificador de la tecla pulsada.
  - **ejecutar_procesos()**: Llama a "ejecutar_procesos" del Motor de Procesos.
  - **ejecutar_motor_grafico()**: Llama a "dibujar_fotograma" del Motor Gráfico.
  - **ejecutar_motor_sonido()**: No hace nada.
  - **mostrar_resultados()**: Llama a "ejecutar_motor_grafico" y a "ejecutar_motor_sonido".
  - **ejecutar()**: Mientras no reciba la orden de terminar, llama a "leer_entrada", "ejecutar_procesos" y "mostrar_resultados".
  - **accion_tecla(tecla : Sdlkey.t)**: Si la tecla pulsada es *Esc* o *Q* ordena finalizar el bucle principal.

- ***Motor***: Provee un interfaz para motores de salida.
  - ***inicializar()***: Inicializa el motor.
  - ***dibujar_fotograma()***: Realiza las operaciones necesarias para el fotograma.
  - ***terminar()***: Detiene el motor y limpia su memoria utilizada.

- **Proxy_motor**: Provee un mecanismo para cambiar dinámicamente un motor.
  - **inicializar()**: Llama a "inicializar" de su motor_final.
  - **dibujar_fotograma()**: Llama a "dibujar_fotograma" de su motor_final.
  - **terminar()**: Llama a "terminar" de su motor_final.

- **Gestor_motores**: Agrega motores de una clase asignando un nombre a cada uno.
  - **obten_motor(nombre:string) : Motor**: Devuelve el motor con el nombre especificado.
  - **registra_motor(nombre:string, m:Motor)**: Asocia el nombre especificado al motor especificado.

- **Motor_grafico_OpenGL**: Implementa un **Motor** que dibuja el resultado utilizando OpenGL. Este **Motor** se desarrolló en un capítulo anterior, y en este capítulo se utiliza.

Para comprender mejor el funcionamiento y la inicialización del sistema se ha añadido un diagrama de objetos que muestra los principales componentes del control general.

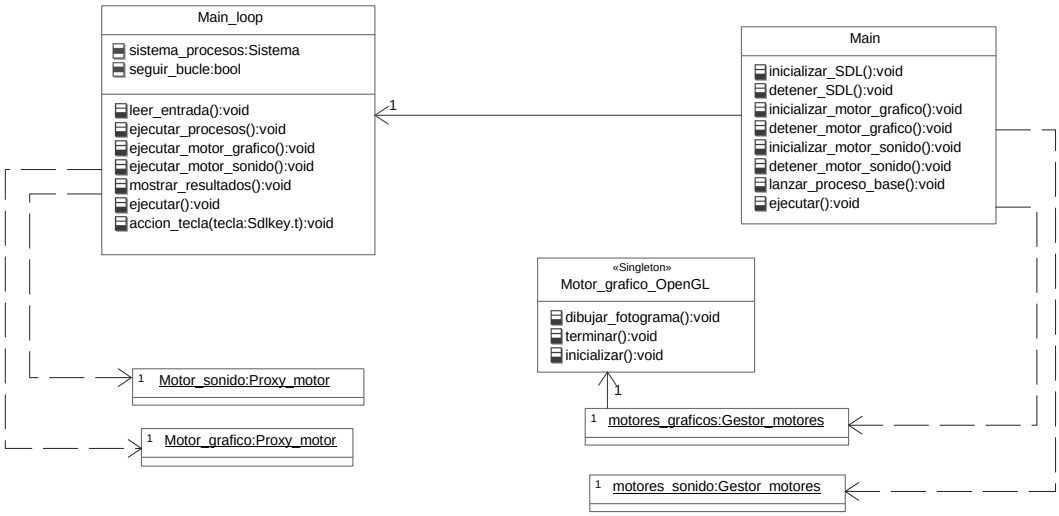

Figura 6.10: Diagrama de Objetos del Controlador General

- Motor_grafico:Proxy_motor: Marca el motor gráfico utilizado actualmente.
- Motor_sonido:Proxy_motor: Marca el motor de sonido utilizado actualmente.
- Motores_graficos:Gestor_motores: Agrega una colección de los motores gráficos disponibles. En la implementación actual tiene una entrada:
  - "OpenGL" : El motor gráfico desarrollado en el proyecto.
- Motores_sonido:Gestor_motores: Agrega una colección de los motores gráficos disponibles.

6.5.2. Algoritmos

El funcionamiento general de la aplicación sigue el patrón típico de aplicación interactiva, que es:

1. Inicializar la aplicación.
2. Mientras no reciba la orden de finalizar:
   3. Leer entrada.
   4. Procesar lógica de la aplicación.
   5. Generar salida.
6. Limpiar recursos utilizados y finalizar la aplicación.

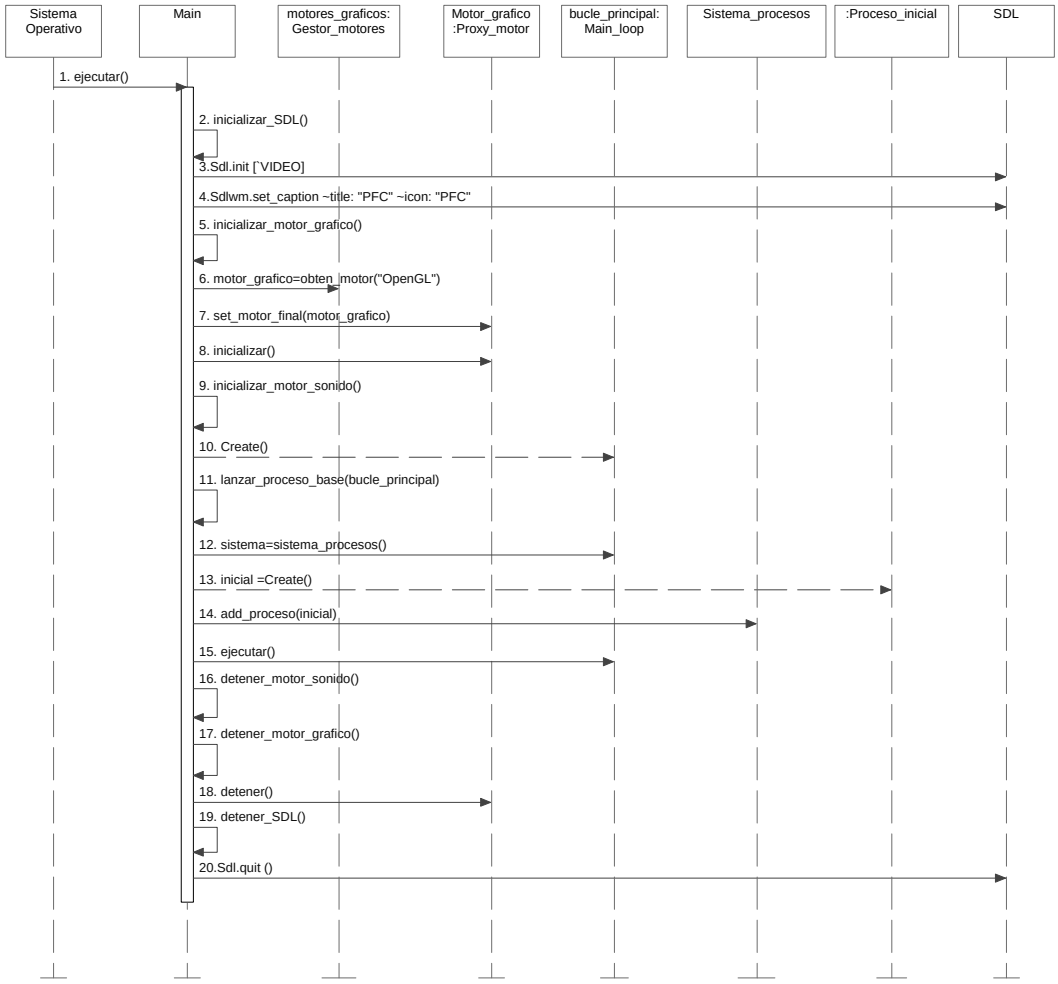

Figura 6.11: Diagrama de Secuencia del punto de entrada de la aplicación

El punto de entrada de la aplicación está localizado en la clase **Main**, en su método "ejecutar". Los pasos que ocurren son los siguientes:

1. El usuario lanza la aplicación a través del Sistema Operativo.
2. **Main** llama a su operación "inicializar_SDL".
3. **Main** inicializa el módulo de vídeo de SDL.
4. **Main** establece el título de la ventana principal.
5. **Main** llama a su operación "inicializar_motor_grafico".
6. **Main** pide al gestor de motores gráficos el motor con el nombre "OpenGL".
7. **Main** establece el motor recibido como motor gráfico.
8. **Main** inicializa el motor gráfico.
9. **Main** llama a su operación "inicializar_motor_sonido".
10. **Main** crea un bucle principal.
11. **Main** llama a su operación "lanzar_proceso_base".
12. **Main** obtiene el sistema de procesos de su bucle principal.
13. **Main** crea un proceso inicial.
14. **Main** agrega el proceso inicial al sistema de procesos.
15. **Main** ordena la ejecución del bucle principal.
16. **Main** llama a su operación "detener_motor_sonido".
17. **Main** llama a su operación "detener_motor_grafico".
18. **Main** detiene el motor gráfico.
19. **Main** llama a su operación "detener_SDL".
20. **Main** llama al método de finalización de SDL.

El bucle principal lee continuamente la entrada, la procesa y muestra el resultado. Puesto que no se ha desarrollado un módulo de entrada adecuado, aquí ha sido sustituido por uno básico, que procesa los eventos de entrada en busca de la señal de finalizar la aplicación. Por cortesía al Sistema Operativo y a las demás aplicaciones, en cada ciclo del bucle principal se hace una llamada de espera, que permite al Sistema Operativo entregar la CPU a otros procesos y realizar otras tareas. Los pasos que ocurren son:

1. **Main** ordena ejecutar el bucle principal.
   Mientras no reciba la orden de terminar:

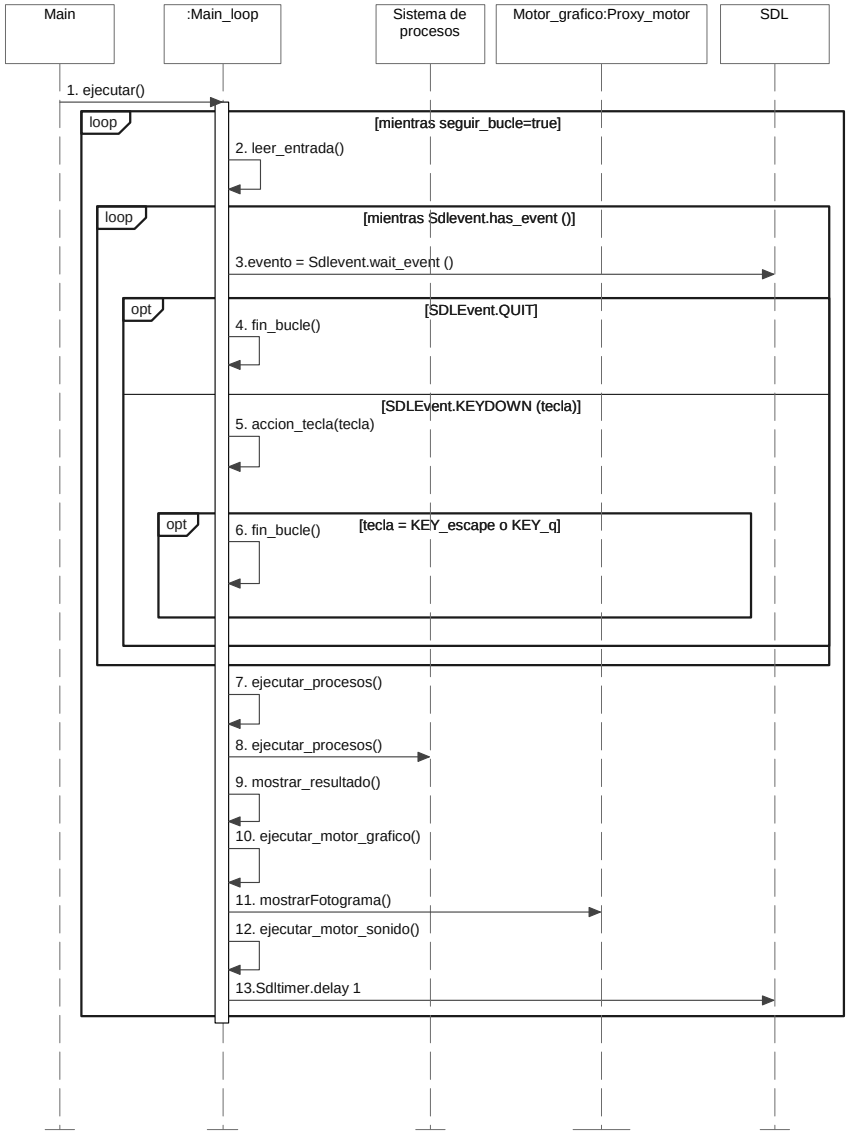

Figura 6.12: Diagrama de Secuencia del bucle principal de la aplicación

2. El bucle principal llama a su operación “leer_entrada”.
   Mientras haya eventos por leer:
   3. El bucle principal lee el siguiente evento disponible.
      Si el evento es de tipo “QUIT”:
      4. El bucle principal establece la orden de terminar.

      Si el evento es de tipo “KEYDOWN”:
      5. El bucle principal llama a su operación “accion_tecla” con la tecla pulsada.
         Si la tecla pulsada es “Esc” o “Q”:
         6. El bucle principal establece la orden de terminar.
7. El bucle principal llama a su operación “ejecutar_procesos”.
8. El bucle principal llama a “ejecutar_procesos” del sistema de procesos.
9. El bucle principal llama a su operación “mostrar_resultado”.
10. El bucle principal llama a su operación “ejecutar_motor_grafico”.
11. El bucle principal llama a “mostrarFotograma” del motor gráfico.
12. El bucle principal llama a su operación “ejecutar_motor_sonido”.
13. El bucle principal ordena pausar la ejecución 1 milisegundo.

Capítulo 7

Conclusión

Alcanzada la fase final del proyecto, se puede afirmar que se han alcanzado los objetivos propuestos:

- Se ha construido un motor gráfico versátil y flexible, que puede usarse directamente o puede ser modificado y adaptado a otras necesidades.
- Se ha utilizado la biblioteca OpenGL para el desarrollo del motor gráfico, y se han aplicado varias de las optimizaciones elementales.
- Se ha construido un sistema básico para la implementación de las reglas del juego.
- Se ha desarrollado un modelo arquitectónico general reutilizable para la creación de videojuegos.
- Se ha comprobado que el sistema funciona adecuadamente.

Durante el desarrollo del proyecto se han observado las características y detalles de varios productos, y se destacan los siguientes aspectos:

- Las características de multiparadigma del lenguaje de programación OCaML son útiles para el desarrollo, pero el continuo cambio de paradigma complica la documentación.
- El lenguaje de programación OCaML es poco conocido. Indirectamente, esto ha significado problemas a la hora de localizar recursos varios, tales como ejemplos avanzados, bibliotecas y bindings.
- Las herramientas disponibles de software libre tienen una potencia sobresaliente en ciertos aspectos, pero son muy débiles en otros. Algunas de las herramientas que se han probado y utilizado o descartado son las siguientes:
    - Creación de diagramas UML: A pesar de haber varias herramientas libres (Umbrello, ArgoUML, Dia, BoUML), todas tenían ciertas carencias; e, incluso, había algunos huecos que no cubría ninguna. Por ello, finalmente fueron sustituidas por IBM Rational Rhapsody Modeler que no es libre, pero, al menos, es gratis.

- Creación de arte para uso en un videojuego: Desde un principio se pensó en el software Blender, que ha estado a la altura de las circunstancias. Cabe destacar el soporte para la creación de extensiones en lenguaje Python, que permitió la construcción de un exportador de mallas de una manera sencilla.
- Escritura de documentación: Desde un principio se pensó en utilizar alguna distribución de LaTeX para la escritura del proyecto, debido a la fama que tiene. Esto ha sido un arma de doble filo, ya que si bien el resultado es excelente, los problemas y dificultades encontrados han sido muy grandes.

Apéndice A

Contenido del CD

En el CD incluído está disponible el resto de la documentación, así como el software utilizado para el desarrollo de este proyecto. A continuación se hace un desglose por directorios.

- **Bibliotecas**
  Contiene las bibliotecas utilizadas por el sistema.
  - **OCaML**
    Contiene el código fuente del compilador y el entorno de ejecución del lenguaje OCaML, así como su documentación básica y referencia de funciones.
  - **OpenGL**
    Contiene los *bindings* de OpenGL para OCaML, permitiendo utilizar de esta manera OpenGL dentro de programas de OCaML. En el desarrollo del proyecto se ha utilizado LablGL. Pero también se incluye la biblioteca alternativa OcamlGL.
  - **SDL**
    Contiene el código fuente de la biblioteca SDL y de varios de sus acompañantes, tales como SDL_image, SDL_mixer y SDL_ttf. Además incluye los *bindings* de SDL para OCaML, y la documentación general de SDL.
- **Documentacion**
  Contiene una copia de este documento, así como de otros que fueron producidos durante el desarrollo del proyecto y se han considerado importantes.
  - **Preparacion_entorno.txt**
    Contiene una lista detallada de instrucciones para construir el entorno donde se puede compilar correctamente el proyecto bajo Windows.
  - **memoria.pdf**
    Contiene este documento.
  - **GestorRecursos**
    Contiene la documentación de los diagramas de secuencia del Gestor de Recursos que no han sido incluídos en este documento.

- **MotorGrafico**
  Contiene la documentación de los diagramas de secuencia del Motor Gráfico que no han sido incluídos en este documento.
- **MotorGrafico**
  Contiene la documentación de los diagramas de secuencia del Motor Gráfico que no han sido incluídos en este documento.
- **MotorProcesos**
  Contiene la documentación de los diagramas de secuencia de la Lógica del Juego que no han sido incluídos en este documento.
- **Planetario**
  Contiene la documentación de los diagramas de secuencia del Planetario que no han sido incluídos en este documento.
- **Exportador_mallas.pdf**
  Contiene la documentación con respecto al script de exportación de mallas para Blender.
- **Formato_malla.pdf**
  Contiene la documentación del formato de fichero de las mallas exportadas desde Blender.
- **Transformada_desde_matriz.pdf**
  Contiene la documentación que explica cómo se puede extraer la translación, la orientación y la escala de una matriz.

- **Implementacion**
  Contiene el código fuente correspondiente a la implementación del proyecto, así como algunos ejecutables pre-compilados que permiten comprobar rápidamente el funcionamiento del sistema.

  - **blender/export_1.0.py**
    Es el script de exportación, que permite convertir mallas de Blender en mallas que acepta el Motor Gráfico.
  - **datos**
    Contiene las mallas y texturas correspondientes al planetario de demostración.
  - **general**
    Contiene bibliotecas de uso general para OCaML.
    - **ficheros.ml**: Contiene la descripción de clases básicas para manipular el contenido de ficheros.
    - **general.ml**: Contiene la descripción de clases para patrones varios, tales como **observador**, **sujeto**, **agregador** y **asociador**.
    - **mat.ml**: Contiene la descripción de clases matemáticas, tales como **vector**, **matriz** y **cuaternion**.
    - **procesos.ml**: Contiene la descripción de las clases construidas para el motor de procesos.

- **recursos.ml**: Contiene la descripción de las clases construídas para el gestor de recursos.
- **registro.ml**: Contiene la descripción de un registro global para recibir y guardar mensajes de advertencia o error producidos por diferentes partes del sistema.
- **vista.ml**: Contiene la descripción de las clases prototipo para motores gráficos y de sonido, así como un gestor global de motores.

- **gfx**
  Contiene el código fuente del Motor Gráfico.
  - **bitmaps.ml**: Contiene las clases y descripciones para manipular bitmaps.
  - **cargador_malla.ml**: Contiene un envoltorio para el sistema de carga de mallas, así como la descripción del gestor de recursos asociado.
  - **cargador_malla_lexer.mll**: Contiene la descripción de items para el parser de mallas.
  - **cargador_malla_parser.mly**: Contiene la descripción del parser que carga mallas.
  - **geo.ml**: Contiene la colección de clases para el nivel de Geometría.
  - **objetos.ml**: Contiene la colección de clases para el nivel de objetos.
  - **render.ml**: Contiene la colección de clases para el nivel de pantalla.
- **opengl/m_opengl.ml**
  Contiene el código fuente de la implementación en OpenGL del Motor Gráfico.
- **logica.ml**
  Contiene la Lógica del Juego correspondiente al Planetario.
- **main.ml**
  Contiene el bucle principal y el punto de entrada del proyecto.
- **Makefile**
  El Makefile con las instrucciones para compilar el proyecto.
- **SDL.dll, SDL_image.dll, SDL_mixer.dll**
  Bibliotecas SDL precompiladas y listas para ser utilizadas.
- **main.exe**
  Ejecutable de demostración del planetario, versión en byte-code con máquina virtual. Construido mediante el comando *make*.
- **main_opt.exe**
  Ejecutable de demostración del planetario, versión optimizada para la arquitectura. Construido mediante el comando *make nc*.

- **Terceros**
  Contiene software variado que ha sido utilizado durante el desarrollo del Proyecto.
  - **Blender**
    Contiene el instalador del software de modelado y animación Blender, así como su código fuente.

- **Cygwin**
  Contiene el instalador y una gran colección de paquetes del entorno de ejecución CygWin, que simula ser un Unix bajo Windows.
- **Dia**
  Contiene el instalador del software de dibujo de diagramas Dia, así como su código fuente.
- **Postscript**
  Contiene los instaladores de Ghostscript y Ghostview para windows, que permiten trabajar con ficheros Postscript, así como su código fuente.
- **SubVersion**
  Contiene el instalador del software de gestión de versiones SubVersion, así como su código fuente.

Apéndice B

Licencias y software de terceros

Este proyecto no hubiera sido posible sin el uso de software de terceros. En este apéndice se hace honor a las licencias que permiten su uso y redistribución.

B.1. Software incluido en el CD

- OcaML: El compilador se distribuye bajo la licencia pública Q, mientras que la biblioteca se distribuye bajo la licencia LGPL. Los detalles pueden consultarse en el archivo LICENSE del paquete ocaml-3.09.3.tar.gz.
- Documentación de OcaML: La documentación se distribuye acorde con la licencia, que puede ser consultada en la sección "Foreword" del documento manual002.html del fichero ocaml-3.09-refman.html.tar.
- LablGL: La biblioteca se distribuye acorde con la licencia, que puede ser consultada en el fichero COPYRIGHT del paquete lablgl-1.02.tar.gz.
- SDL, SDL_ttf, SDL_Mixer, SDL_image, OcamlSDL: Se distribuyen acorde con la licencia LGPL 2.1, que puede ser consultada en el fichero COPYING de los diferentes paquetes.
- Blender: El software Blender se distribuye bajo licencia GPL, que puede ser consultada en el fichero GPL-license.txt del paquete de código fuente de Blender.
- CygWin: El entorno CygWin se distribuye bajo licencia GPL. Los binarios se encuentran en el directorio correspondiente del DVD, y el código fuente en http://www.cygwin.com/.
- Ghostscript: El programa Ghostscript se distribuye bajo licencia GPL, con excepción de algunos elementos. Los detalles concretos se pueden consultar en el fichero LICENSE del paquete ghostscript-8.63.tar.gz.
- Gsview: El programa Gsview se distribuye bajo la licencia pública Alladin. Los detalles concretos de dicha licencia se pueden consultar en el fichero LICENCE del paquete gsv49src.zip.

- Subversion: El sistema de control de versiones Subversion se distribuye bajo una licencia pública concreta, que puede ser consultada en el archivo COPYING del paquete subversion-1.3.0.tar.gz.

B.2. Software de terceros

Además del software incluído en el DVD, se han utilizado otros sistemas.

- Rational Rhapsody Modeller: El editor de diagramas UML Rhapsody tiene una versión gratuita. Dicha versión puede ser obtenida en http://www.ibm.com/developerworks/downloads/r/modeler/.
- MikTeX: Esta variante del entorno TeX contiene las últimas actualizaciones y mejoras para el entorno Windows. Puede ser obtenido en http://miktex.org/.

Bibliografía

[1] The OpenGL Programming Guide - The Redbook `http://www.glprogramming.com/red/`, 1997.

[2] The Objective Caml system. Documentation and user's manual `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`, 2008.

[3] Blender Python API `http://www.blender.org/documentation/242PythonDoc/`, 2006.

[4] The Python Language Reference `http://docs.python.org/reference/`, 2008.

[5] The Matrix and Quaternion FAQ. `http://www.gamedev.net/reference/articles/article1691.asp`, 2003.

[6] The Great Win32 Computer Language Shootout. `http://dada.perl.it/shootout/`, 2003.

[7] The Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/`, 2007.

[8] Cascales Salinas, Bernardo et al. *LaTeX, una imprenta en sus manos.* ADI, 2000.