Universidad
de La Laguna

Gráficos 3D en Interfaces Web

3D Graphics on Web Interfaces

Javier Villamonte Pereira

Dpto. de Ingeniería Informática y de Sistemas

Escuela Superior de Ingeniería y Tecnología

Sección de Ingeniería Informática

Trabajo de Fin de Grado

La Laguna, 8 de julio de 2014

Dra. Dª. Gara Miranda Valladares, con N.I.F. 78.563.584-T, profesora Ayudante Doctor, y **Dra. Dª. Coromoto León Hernández**, con N.I.F. 78.605.216-W, profesora Titular, adscritas al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna

C E R T I F I C A N

Que la presente memoria titulada:

"*Gráficos 3D en Interfaces Web*"

ha sido realizada bajo su dirección por D. **Javier Villamonte Pereira**, con N.I.F. 46.244.327-K.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 8 de julio de 2014

Agradecimientos

A mis tutoras Gara Miranda y Coromoto León, así como también a Yanira González por su dedicación, atención y amabilidad durante todo este tiempo.

Resumen

Desde que surgiera la web, a finales de la decada de los 80, su evolución constante ha enriquecido mucho su interactividad y ha convertido una plataforma que en un principio solo podía contener texto en un mundo que sus creadores nunca habrían imaginado. En ese contexto, el objetivo de este trabajo es el estudio del estado del arte en el campo de los gráficos tridimensionales en la web. Para llegar a él se ha propuesto una meta concreta: construir una aplicación web dedicada a la visualización de instancias y soluciones del problema de la carga de contenedores. Durante las fases de análisis previas al desarrollo se han evaluado distintas librerías, con algunas de las cuales se han hecho pruebas, lo que ha permitido extraer sus fortalezas y debilidades. Además, los requerimientos del proceso han servido para saber qué es lo que se puede hacer con este tipo de tecnologías. Por otra parte, esta memoria puede ser vista como la documentación de ese producto, ya que contiene una descripción profunda del diseño, las clases y los métodos utilizados, así como una explicación de lo que ocurre internamente y cómo se relacionan las clases en cada caso de uso existente.

Palabras clave: Gráficos 3D, Web, Interfaces Gráficas de Usuario, Problema de Carga de Contenedores

Abstract

Since the Web appeared, at the end of the 1980s, its constant evolution has improved its own interactivity and has transformed a plataform, which at the beginning was only able to deal with text, in a world that its makers would have never imagined. In this context, the goal of this work is to study the state of 3D Graphics on Web Interfaces. In order to achieve such a goal, a specific objective has been proposed: build a Web interface to visualize Container Loading Problem examples. Some libraries have been analyzed and a few of them have been tested to evaluate their pros and cons. This analysis has helped to choose the implementation framework. Furthermore, the process requirements served to find out what can be done with this kind of technology. Otherwise, this document can be seen as the developer and user documentation, because it contains a description of the design, the classes and methods, as well as a case study.

Keywords: *3D Graphis, Web, Graphical User Interfaces, Container Loading Problem*

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

A lo largo de este primer capítulo se introducirá al lector en el mundo de los gráficos 3D, explicando algunos conceptos básicos de la materia. Posteriormente se seguirá la evolución de las herramientas disponibles para llevarlos a la Web y se revisará el estado actual de las mismas, haciendo un repaso de las principales librerías. En último lugar, se definirán los objetivos de este trabajo y se buscarán ejemplos de lo que se quiere lograr con él.

1.1. Gráficos 3D

Los gráficos 3D elaborados mediante ordenador son «trabajos de arte gráfico creados con ayuda de software» [10]. El proceso de creación de los mismos consta de tres fases [11]:

- **Modelado:** es el proceso de desarrollo de una representación matemática, de cualquier superficie u objeto, utilizando un conjunto de puntos en un espacio tridimensional, conectados por formas geométricas tales como triángulos. En la etapa de modelado se da forma a los objetos que posteriormente se usarán en la escena. Este proceso puede incluir la descripción de la superficie del objeto, por ejemplo con texturas, y las propiedades del material entre otras características. También tiene cabida la preparación para una posterior animación, por ejemplo mediante la asignación de un esqueleto que relaciona las partes del modelo con las del esqueleto, simplificando la definición de movimientos [21].

- **Composición de la escena:** en esta etapa se distribuyen los objetos, luces y cámaras, entre otros, que serán utilizados para producir la imagen o animación. La iluminación [18] es un aspecto clave de la composición de la escena porque contribuye al resultado estético y requiere del entendimiento físico de la luz real para poder recrearla con fidelidad. Existen tres tipos de luces a los que se puede denominar «básicos»: luz global, luz puntual y la luz direccional. La luz global únicamente tiene información de dirección por lo que los rayos que emite son paralelos, se asemeja a como el sol ilumina el planeta tierra. La luz puntual emite luz hacia afuera desde un solo punto del espacio 3D en todas direcciones, tiene el comportamiento de una bombilla. La luz direccional posee información tanto de dirección como de posición por lo

que es útil cuando se separan áreas iluminadas en la escena, tiene el comportamiento de un foco o linterna.

- ***Renderizado*:** es el proceso final que genera la imagen o animación a partir de la escena creada. Esto puede ser comparado con tomar una foto o filmar una escena real. El software puede simular efectos cinematográficos producidos por las imperfecciones mecánicas de la fotografía real, aportando realismo con su presencia debido a la costumbre de los seres humanos a ellos. Esta fase requiere una gran capacidad de cómputo ya que simula procesos físicos complejos, por lo tanto, es natural que con la mejora de la capacidad de los ordenadores a lo largo del tiempo también mejorara el realismo de los renders.

En el ámbito de la Ingeniería Informática este procedimiento es empleado con propósitos muy diferentes: desde la creación de videojuegos hasta la representación de datos de distinta índole. Asimismo, las tecnologías existentes para ello y, en consecuencia, los contextos donde se pueden visualizar interfaces que emplean gráficos tridimensionales son muy variados. Hay numerosos ejemplos de software para el diseño 3D, como *Blender* [1], que es de código libre, o el motor *Unity* [32]. Los modelos generados con ellos pueden, incluso, ser exportados y utilizados en plataformas Web, lo cual ofrece muchas posibilidades a los programadores y enriquece la interactividad de los contenidos. Este trabajo seguirá esa última vertiente, es decir, se centrará en el estudio de las herramientas disponibles para la elaboración de gráficos 3D específicamente en las plataformas Web. En la siguiente sección se hablará sobre ellas y se realizará una relación inicial que compondrá la primera aproximación al estudio que se desea realizar.

1.2. Gráficos en la Web

Antes de abordar los detalles de las librerías disponibles en la actualidad, sería interesante comentar cómo han evolucionado los gráficos en la Web y sus herramientas a lo largo del tiempo para resaltar el increíble crecimiento que han experimentado. Una de las bases actuales de la programación Web, *JavaScript*, surgió en el año 1995 de la mano de Brendan Eich cuando trabajaba en *Netscape* y más tarde, en 1997, fue adoptado como estándar. Originalmente se llamaba *LiveScript* pero antes de lanzar la primera versión *Netscape* se alió con *Sun Microsystems*, creador de *Java*, para seguir desarrollando el lenguaje. Dicha alianza hizo que se diseñara como un hermano pequeño de *Java*, solamente útil dentro de las páginas Web y mucho más fácil de utilizar, de modo que cualquier persona, sin conocimientos de programación pudiese adentrase en el lenguaje y utilizarlo sin ningún problema [14, 15].

Otro actor importante del panorama Web, *CSS* (Hojas de Estilo en Cascada, del inglés, *Cascading Style Sheets*), nació en 1996, a pesar de que antes de esa fecha ya existía el concepto de hojas de estilo, aunque con diferentes propósitos. En la Web, al principio, el propio *HTML* (siglas de *HyperText Markup Language*) era el encargado de proporcionar a los desarrolladores la capacidad de controlar la apariencia de sus sitios, pero las

diferencias entre las implementaciones de los navegadores hacían difícil que las páginas se vieran iguales en todos ellos y por tanto se tenía menos control sobre el diseño. Para solucionar este problema, se quería separar la estructura del estilo, de modo que nueve lenguajes distintos de hojas de estilo fueron propuestos a la *W3C* [33] (*World Wide Web Consortium*) y dos de ellos fueron combinados para crear *CSS*: *CHSS* (*Cascading HTML Style Sheets*), propuesto por Hakon Wium Lie y *SSP* (*Stream-based Style Sheet Proposal*), planteado por Bert Bos. Ambos trabajaron juntos para crear el estándar *CSS*, por el cual el *W3C* decidió apostar para su desarrollo y estandarización, añadiéndolo a su grupo de trabajo de *HTML* [12, 13].

Hasta mediados de la década de los 2000, las descritas anteriormente eran las herramientas de las que se disponía para hacer animaciones y gráficos en la Web de manera nativa. Otra opción era recurrir a plugins como *Adobe Flash* o applets *Java*, que además eran las únicas maneras de aprovechar la aceleración mediante GPU (Unidad de Procesamiento Gráfico, del inglés, *Graphics Processing Unit*). Desde entonces se han ido proponiendo otras tecnologías que permiten los gráficos 3D en Interfaces Web: los gráficos vectoriales, el elemento *Canvas* de *HTML5* y la especificación estándar *WebGL* (*Web Graphics Library*). Este último empezó como un experimento de Vladimir Vukicevic con el *Canvas* 3D en la *Mozilla Foundation* en el año 2006. A finales de 2007 *Mozilla* y *Opera* habían hecho sus propias implementaciones separadas y en 2009 se fundó el *WebGL Working Group* [16, 17].

En la actualidad existen numerosas librerías que trabajan con el elemento *Canvas* y WebGL [19]. Sobre cada uno de ellos hay mucha información, por tanto se hará un breve resumen de lo que se puede encontrar sobre cada una [4].

- **C3DL** [2]: La última actividad de desarrollo fue hace más de dos años (junio de 2011), tiene una buena documentación, presenta signos de mal diseño: la opción de luz ambiental debe moverse a la clase de luz porque no se encuentra en ella.

- **Curve3D** [8]: Este *framework* no tiene documentación, solo el código fuente en GitHub y el desarrollo se detuvo hace tres años.

- **CubicVR 3D** [7]: Tiene ejemplos prometedores, tutoriales útiles y una buena documentación. Se puede trabajar en C++ y en JavaScript.

- **Copperlicht** [5]: Aunque es una solución comercial, es de uso gratuito. La página de inicio ofrece una buena documentación y ejemplos interesantes. Los nombres de clase y método no siempre son autoexplicativos, ya que tienen el nombre de la librería incrustado.

- **GLGE** [9]: Tiene una buena documentación. No tiene ninguna clase para las sombras. Aunque la página de inicio proporciona algunos ejemplos, deberían ser más útiles y estar acompañados de tutoriales.

- **O3D** [22]: El desarrollo se detuvo en 2010 y no tiene documentación.

- **OSG.JS** [23]: Esta API está en desarrollo activo, tiene buenos ejemplos, pero una mala documentación.
- **PhiloGL** [25]: En la página oficial ofrece buenos ejemplos y tutoriales, está en desarrollo activo. Está orientada a minimizar la codificación (al estilo jQuery). Tiene una buena documentación.
- **Scene.JS** [27]: Hay ejemplos muy interesantes, pero el código fuente para ejemplos básicos es bastante grande, por lo cual sería lógico suponer que las escenas de mayor tamaño se volvieran complicadas de entender.
- **SpiderGL** [29]: No tiene actividad desde hace seis meses, todo el código está en un solo fichero y no tiene documentación.
- **TDL** [30]: Los ejemplos oficiales son muy prometedores, siendo mejores que los de otros *frameworks*. Casi no tiene documentación, solo una guía de introducción.
- **Three.JS** [31]: Tiene buenos ejemplos en GitHub y un manual de uso. Se pueden realizar escenas con poco código, lo cual ayuda a que sea de fácil lectura.
- **X3DOM** [34]: Esta es la única API para WebGL que sigue fuertemente la especificación X3D. Su página Web proporciona ejemplos básicos y la documentación. Es un *framework* limitado debido a que es un trabajo de investigación.

De lo anterior es posible deducir que las librerías más interesantes son Three.js, Scene.js, PhiloGL, CubicVR 3D y quizás GLGE, aunque la más conocida y sobre la que más documentación se ha encontrado ha sido Three.js. Hay bastantes ejemplos de uso de las librerías nombradas anteriormente y podrían ser de utilidad para el trabajo que se quiere llevar a cabo, pues con cualquiera de ellas resulta factible desarrollar gráficos de gran calidad para la Web.

1.3. Objetivos del trabajo

El objetivo de este trabajo es realizar un estudio del estado de los gráficos 3D en Interfaces Web, comparando las distintas librerías disponibles. Para poder hacer una mejor valoración del estado de los gráficos 3D en Interfaces Web, se hace necesario plantear un problema concreto, una meta práctica en cuya ejecución sea posible comparar y valorar cualitativamente las caracterśticas de cada una de las herramientas.

En este caso, la meta planteada consiste en construir una interfaz con tecnologías Web que permita visualizar instancias y soluciones del Problema de Carga de Contenedores, integrando distintos tipos de interacción para interpretar, de una manera más intuitiva, los resultados que el motor de resolución genera. Para el estudio se ha seleccionado este problema, no sólo por su relevancia a nivel de aplicación práctica en la industria, sino también porque ha sido estudiado y abordado por el Grupo de Algoritmos y Lenguajes Paralelos de la Universidad de La Laguna.

A modo de captura de requisitos, para saber qué funcionalidades son más útiles en las nterfaces dedicadas al problema de carga de contenedores, en la Tabla 1.1 se presenta una comparación de cinco aplicaciones gratuitas similares a lo que se pretende desarrollar.

- **Searates** [28]: solo muestra una foto. El contenedor se delimita por sus aristas y nada más, no tiene rejilla ni transparencias. Las cajas no tienen transparencias. El espacio vacío no se pinta.
- **Calculation Space** [3]: es el más completo de los cinco evaluados. Los controles de la interfaz son menos intuitivos, más molestos y más lentos que en otras páginas. Permite girar alrededor del contenedor en todas las direcciones. Las cajas tienen transparencias y son de distintos colores. Permite zoom. Tiene una máquina de estados, posibilitando pasar de uno en uno para atrás y adelante, también permite ir directamente al fin o el principio. El espacio vacío no se pinta de ningún color. El contenedor se delimita por sus aristas y nada más, no tiene rejilla ni transparencias.
- **Logimar** [20]: no permite hacer zoom ni girar arriba y abajo, solo hacia los costados. Las cajas no tienen transparencias y los colores son parecidos. No tiene máquina de estados. El espacio vacío no se pinta. El contenedor se delimita por sus aristas y nada más, no tiene rejilla ni transparencias.
- **Cube Master** [6]: solo muestra una foto. El contenedor se delimita por sus aristas y nada más, no tiene rejilla ni transparencias, pero se pinta el suelo. Las cajas no tienen transparencias. El espacio vacío no se pinta.
- **Packer 3D** [24]: no sigue la misma idea que el resto. Lo que hace esta aplicación es escoger en primer lugar un medio de transporte y después procede a llenarlo con un determinado tipo de cajas para calcular cuantas pueden ser alojadas. Solo muestran fotos con el alzado, la planta y el perfil. El espacio libre no se pinta. Las cajas no tienen transparencias.

En los proximos capítulos de esta memoria se presentará un estudio comparativo de *frameworks* Web 3D, y se indicarán los motivos por los cuales en este trabajo nos hemos decantado por uno de ellos en concreto. Posteriormente se definirá el problema de la carga de contenedores, explicando el algoritmo de resolución así como de la herramienta utilizada. Luego se describirá el diseño y la implementación de la interfaz gráfica de usuario, especificando las clases y sus relaciones. Se analizarán los casos de uso existentes, desglosando el proceso que ocurre internamente en cada uno de ellos. Por último, se presentarán las conclusiones, tanto en inglés como en español y se estimará el presupuesto necesario para llevar a cabo el trabajo.

Aplicación	**Rotar**	**Zoom**	**Cajas**	**Vacío**	**Contenedor**	**Desactivar cajas**
Searates	No	No	Color sólido	No se pinta	Aristas	No
Calc. Space	Sí	Sí	Transparencias	No se pinta	Aristas	No
Logimar	Lateralmente	No	Color sólido	No se pinta	Aristas	No
Cube Master	No	No	Color sólido	No se pinta	Aristas	No
Packer3D	No	No	Color sólido	No se pinta	Aristas	No

Tabla 1.1: Comparativa entre aplicaciones de carga de contenedores.

Capítulo 2

Herramientas 3D para la Web

En este capítulo se llevan a cabo las pruebas empíricas con las librerías que se han considerado más versátiles y potentes tras el breve estudio presentado en el capítulo anterior. La comparación servirá para determinar cuál es la más apropiada para el propósito de este trabajo. Para ello se ha marcado un objetivo común que ha de ser cumplido con cada una de ellas: se debe dibujar una «caja» ubicada en el interior de un «contenedor». Para que el primero sea visible, se pueden utilizar transparencias en el segundo, o bien que esté formado por una rejilla. Además, para que la interfaz sea amigable de cara al usuario, se intentarán implementar controles con el ratón, para poder ver el conjunto desde diferentes ángulos según convenga. Partiendo de los resultados, se hará un análisis de las ventajas, inconvenientes y, cuando corresponda, observaciones a tener en cuenta.

2.1. CubicVR 3D

2.1.1. Ventajas

Se está ante una librería con la que es bastante fácil familiarizarse, a pesar de que para ello no se puede hacer uso de muchos tutoriales, ayuda el hecho de que existen numerosos ejemplos desarrollados y expuestos en la red.

También resulta útil la documentación existente, es muy buena y se puede encontrar en GitHub junto con el código fuente de la librería y el de los ejemplos que con ella se ofrecen.

En el momento de desarrollar, se agradece que añadir nuevas cajas sea sencillo, sólo hacen falta dos líneas si se tiene una forma ya definida, debido a que seguramente es la operación que más se repetirá en posteriores etapas de este trabajo.

Otra comodidad que proporciona CubicVR es la posibilidad de incorporar controles con el ratón en una sola línea, los cuales se comportan de manera previsible, ayudando a hacer intuitiva la interfaz. Sin embargo, en ocasiones no tiene el comportamiento deseado, realizando movimientos muy veloces. De todos modos esto solo se produce en situaciones aisladas, cuando se mira el techo o la base del contenedor de forma casi perpendicular.

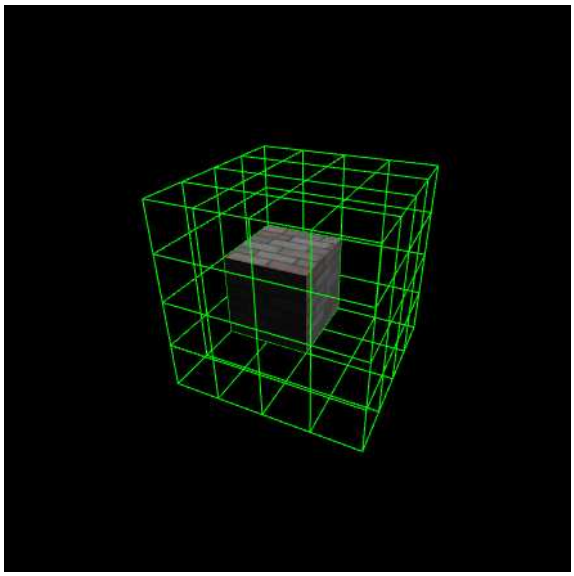

Figura 2.1: Captura de las pruebas con CubicVR 3D.

Por último, después de desarrollar la prueba se comprueba que el código es legible y poco complejo.

2.1.2. Inconvenientes

Un detalle algo negativo es que si bien se tienen a disposición materiales de rejilla y transparencias, en los primeros no es posible regular la cantidad de divisiones para cada par de caras opuestas. Solamente permite establecer un número de divisiones común a todas ellas. La última actualización de la librería fue en mayo de 2013.

2.2. GLGE

2.2.1. Ventajas

GLGE es un *framework* de aprendizaje relativamente sencillo. Para ello se dispone de un número razonable de tutoriales y ejemplos, acompañados de una buena documentación.

En lo referente al código, es fácilmente legible y sencillo, debido en gran parte a que el diseño de la escena se hace en un fichero con formato XML, de modo que se carga en el código JavaScript, se asignan las variables correspondientes (cámara, escena, etc) y se renderiza.

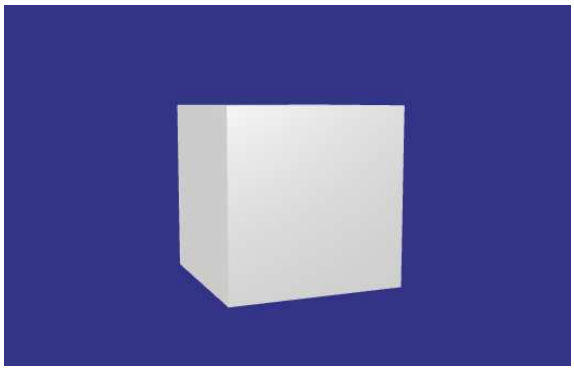

Figura 2.2: Captura de las pruebas con GLGE.

2.2.2. Inconvenientes

Está en estado de abandono, de hecho la última actualización de la librería fue en junio de 2012. Antes de que estuviera así ya había ocurrido que las transparencias dejaban de funcionar por problemas en el desarrollo, lo cual sigue fallando actualmente y por desgracia no hay ningún material de rejilla, debido a todo esto no se puede hacer que el usuario vea las cajas que el contenedor posee en su interior.

Por otro lado, es complicado aumentar la presencia de cajas, no solo por la dificultad de añadirlas, sino también porque GLGE no brinda ayudas para crearlas, con lo cual se tienen que definir todos los puntos, haciendo todos los cálculos pertinentes para situarlos en el lugar adecuado. Para terminar, es importante remarcar que en GLGE es más tedioso controlar la cámara con el ratón que en otras librerías.

2.3. PhiloGL

2.3.1. Ventajas

Probablemente sus puntos fuertes sean la gran cantidad de ejemplos que hay en la red y la gran calidad de su documentación, aunque los tutoriales sean escasos. El otro aspecto favorable es la sencillez de incluir controles con el ratón.

2.3.2. Inconvenientes

Desafortunadamente, esta librería presenta más inconvenientes que ventajas para este trabajo, pues no existe un material de rejilla y las transparencias no funcionan del todo bien: es sorprendente ver cómo se aplican a todos los elementos de la escena y no a uno en particular.

La última actualización fue en el año 2012 y tampoco favorece el que sea la librería con la

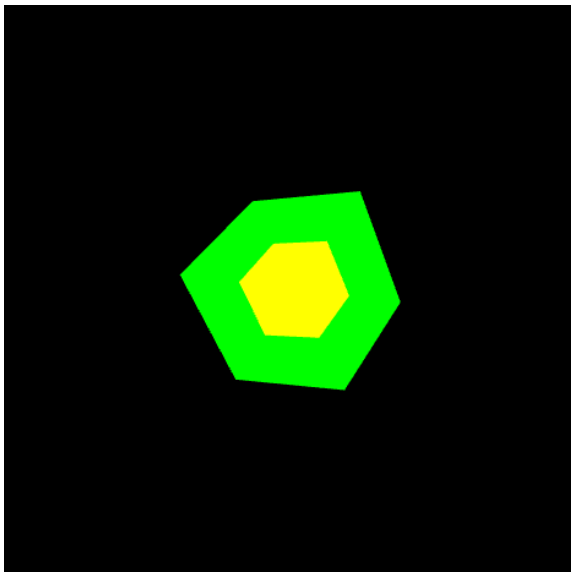

Figura 2.3: Captura de las pruebas con PhiloGL.

peor curva de aprendizaje de las cinco que se han estudiado. Además el código producido es más complejo que el de las otras y menos legible. Como comentario final, hay que indicar que insertar nuevas cajas no es del todo fácil.

2.4. SceneJS

2.4.1. Ventajas

Es agradable saber que SceneJS se ha actualizado de manera reciente, pues se trata de una librería con la que no es difícil familiarizarse. Cuenta con tutoriales y ejemplos, sin embargo, algunos de ellos y su documentación están incompletos o mal explicados. A pesar de todo, no resultó complicado conseguir los objetivos de la prueba, incluso proporcionar controles con el ratón fue relativamente fácil.

2.4.2. Inconvenientes

En primer lugar, debería ser más fácil introducir nuevas cajas. Por otro lado, el código es bastante legible, pero es algo complejo a la hora de manipular porque esta librería se basa en la idea de que cada elemento es un nodo y estos se van anidando para formar la escena, por tanto, con ejemplos simples como el de la prueba que se realizó, se vuelve complicado orientarse entre las numerosas llaves que se ha hecho necesario insertar. Para finalizar, se ha de decir que se pueden usar transparencias correctamente, pero al usar un material de rejilla no se permite regular las divisiones.

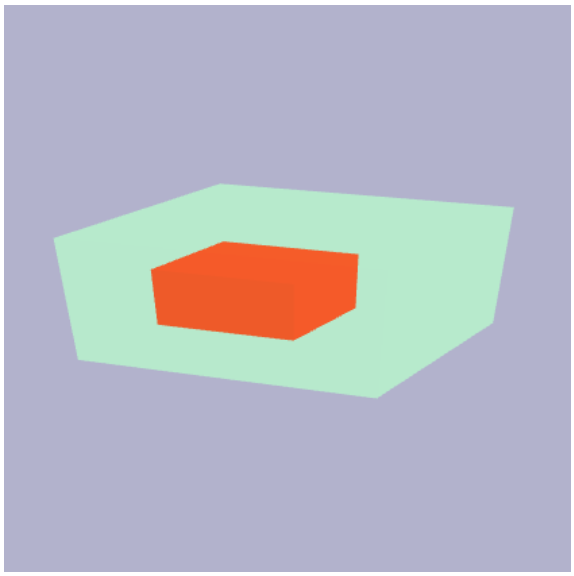

Figura 2.4: Captura de las pruebas con SceneJS.

2.5. ThreeJS

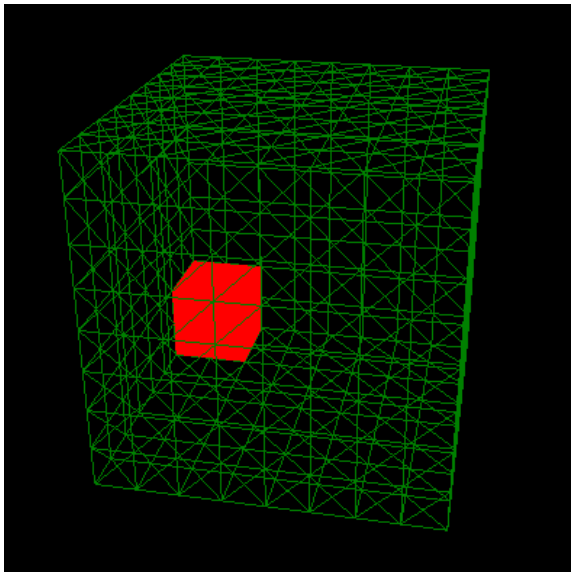

Figura 2.5: Captura de las pruebas con ThreeJS.

2.5.1. Ventajas

ThreeJS fue la librería más completa de todas y está en pleno desarrollo. Es muy fácil de aprender, en ese sentido se pueden encontrar buenos tutoriales, incluso en español, y muchos ejemplos, además de contar con una documentación francamente buena.

El código es legible y poco complejo, si va acompañado de comentarios se hace sencilla su comprensión. Otras grandes ventajas son las facilidades que ofrece este *framework* para colocar nuevas cajas en la escena y controles con el ratón, los cuales se comportan de manera excelente en todas las situaciones y son muy intuitivos.

Otra funcionalidad genial de ThreeJS es que resulta posible utilizar transparencias y materiales de rejilla, permitiendo controlar las divisiones de estos últimos de manera separada para cada par de caras opuestas.

2.5.2. Inconvenientes

Para este trabajo en concreto no se han encontrado inconvenientes.

2.6. Resumen de la comparación

En la Tabla 2.1 se presenta un resumen del estudio comparativo realizado entre las diferentes herramientas. Los criterios que se tendrán en cuenta en esta comparativa serán los siguientes:

- Tiempo transcurrido desde la última actualización (TT).
- Facilidad de aprendizaje (FA).
- Cantidad de tutoriales y de ejemplos disponibles (TyE).
- Calidad de la documentación (CD).
- Legibilidad y complejidad del código producido (LyC).
- Facilidad para añadir cajas nuevas (ACJ).
- Facilidad para añadir controles con el ratón (ACTRL).
- Existencia de transparencias y/o materiales de rejilla (TyR).

Se valorarán estos atributos con una nota del uno al cinco, donde uno significa «muy en desacuerdo» y cinco quiere decir «muy de acuerdo» (a excepción de la fecha). Por simplificación, para hacer referencia a cada atributo en la tabla se usará el código especificado entre paréntesis en la lista anterior.

Librería	**TT**	**FA**	**TyE**	**CD**	**LyC**	**ACJ**	**ACTRL**	**TyR**
Cubic VR	6 meses	5	4	4	4	5	5	4
GLGE	1 año	3	3	4	4	2	2	1
PhiloGL	1 año	1	3	4	1	2	3	1
SceneJS	23 dás	3	4	3	3	2	4	4
ThreeJS	1 semana	5	4	4	5	5	4	5

Tabla 2.1: Comparativa entre opciones para diseño 3D en la web.

Las primeras librerías descartadas son GLGE y PhiloGL por sus graves inconvenientes, que dificultan el desarrollo de este trabajo. De las restantes, SceneJS es la tercera opción, porque si bien es válida, presenta más problemas que las otras. Finalmente se ha escogido ThreeJS, frente a CubicVR debido a su mayor sencillez y la mejor legibilidad del código producido. Esto último se puede observar en la tabla, que además refleja una mejor puntuación para ThreeJS.

Capítulo 3

El Problema de Carga de Contenedores

En este capítulo se describirá el Problema de Carga de Contenedores con un enfoque descendente, es decir, en primer lugar se hará una introducción al mismo en términos generales, viendo sus aplicaciones reales y algunas variantes. A continuación se definirá formalmente, explicando los objetivos de la aproximación concreta que se utilizará. Esto dará paso a los detalles del algoritmo de resolución y los formatos de los ficheros de entrada/salida que utiliza. Por tanto, el recorrido se realizará partiendo de lo general para llegar a lo concreto.

3.1. Introducción al problema

El Problema de Carga de Contenedores (en inglés *Container Loading Problem* o *CLP*) pertenece a un área de investigación activa y tiene numerosas aplicaciones en el mundo real, particularmente en el transporte de contenedores, la logística y la distribución. Cuando se resuelve el *CLP*, normalmente, el objetivo es distribuir un conjunto de piezas con forma de prisma (cajas) en otro objeto de la misma forma pero de mayores dimensiones (el contenedor) para maximizar el volumen total de cajas empaquetadas y que, por tanto, serán transportadas en el contenedor.

Sin embargo, en muchas situaciones del mundo real hay otros elementos a tener en cuenta [37]. A veces hay que tener cuidado con la orientación de los paquetes dentro del contenedor si lo que llevan dentro es frágil. Otras veces, es necesario tener en cuenta, a la hora de colocar las cajas, que algunas de ellas pueden tener que ser retiradas en un lugar anterior a su destino, es decir, en una escala, para ser movidas a otro contenedor. A pesar de lo anterior, un aspecto común en el alcance de este problema es el límite de peso de los contenedores, porque normalmente no puede exceder cierta cifra para que su transporte sea posible. Los camiones que transportan el envío cobran en función del peso total que pueden transportar, sin importar el volumen. Así, se prefiere cargar y enviar un cargamento con un alto peso en vez de bajo.

En la literatura, a pesar de haber algunos trabajos aislados que usan algoritmos exactos para tratar el *CLP*, la mayoría de estudios se enfocan en generar soluciones usando métodos heurísticos y metaheurísticos, porque cumputacionalmente se trata de un problema catalogado como *NP-Hard* [41]. Algunas formulaciones del problema consideran la posibilidad de tener múltiples contenedores. En otros casos una de las dimensiones del contenedor es infinita o está abierta, de modo que todas las piezas pueden ser introducidas [35, 36, 42]. En cualquier caso, la mayoría de los enfoques tratan con formulaciones monoobjetivo del problema, siendo los trabajos que versan sobre el CLP multiobjetivo casi inexistentes [37].

3.2. Definición del problema

La definición formal del problema aquí tratado es la siguiente: se tiene un contenedor con una anchura W, longitud L, alto H y peso máximo P_{max} conocidos, y un conjunto de N cajas. Estas cajas pertenecen a uno de los tipos existentes $\mathcal{D} = \{T_1 \dots T_m\}$. Cada tipo T_i tiene un peso $p_i < P_{max}$ y volumen v_i asociado tal que $v_i <= W * L * H$. El objetivo es encontrar un subconjunto de cajas que maximicen el peso y volumen utilizado en el contenedor:

$$\text{máx} \sum_{i=1}^{m} x_i v_i \quad \text{y} \quad \text{máx} \sum_{i=1}^{m} x_i p_i$$

Donde x_i indica el número de cajas del tipo i que serán introducidas en el contenedor. El objetivo de la herramienta de resolución utilizada es cargar los items (cajas) que permitan obtener el mejor aprovechamiento tanto del volumen como del peso máximo admitido por el contenedor. Estos dos objetivos entran en conflicto debido a que el volumen de una caja, normalmente, no es proporcional a su peso. Por esta razón, se considera que el problema trata de maximizar simultáneamente la utilización del peso y el volumen. En este sentido, el problema puede ser catalogado como de optimización multiobjetivo, porque trata de optimizar la disposición de las piezas dentro del contenedor para que el volumen sea maximizado a la vez que el espacio, sin exceder los límites del mismo.

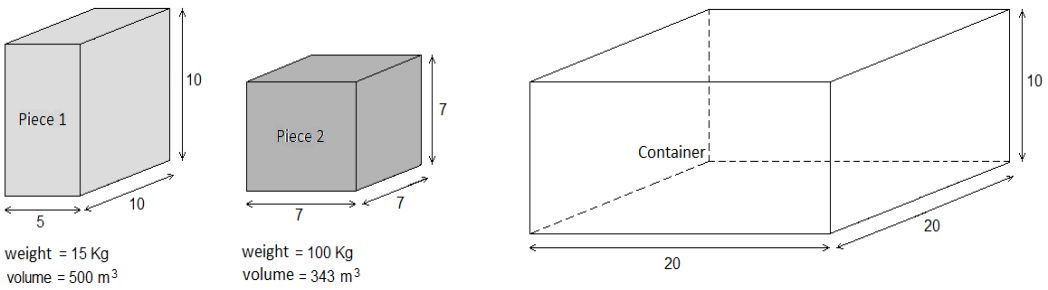

Figura 3.1: Ejemplo de problema

3.3. Algoritmo de resolución

Para decidir dónde colocar exactamente cada una de las piezas, se han propuesto dos heurísticas [39] basadas en una estrategia de llenado por niveles, en las cuales las piezas son almacenadas en el contenedor: la heurística de llenado de un solo nivel (*SLFH*, del inglés *Single-Level Filling Heuristic*), y la heurística de llenado multinivel (*MLFH*, del inglés *Multiple-Level Filling Heuristic*). Ambas están basadas en la creación y gestión de niveles de piezas o capas dentro del contenedor. Dichos niveles o capas identifican espacios vacíos dentro, y así, representan áreas donde colocar items. La heurística de llenado multinivel funciona del siguiente modo:

- La primera pieza es introducida en la esquina del fondo izquierdo inferior del contenedor. Ésta determinará las dimensiones de los espacios a generar: uno frente a la caja, otro encima y el último al costado.

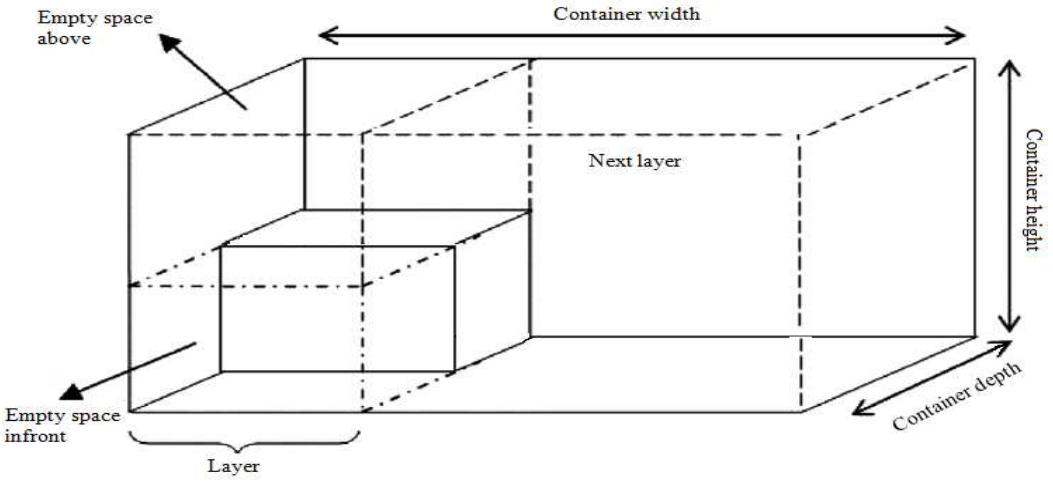

Figura 3.2: MLFH: Espacios creados

- La siguiente pieza es escogida y situada en el contenedor siguiendo estas reglas:
  - Primero, se trata de llenar el espacio creado frente a la caja. Si no hay espacio disponible para la pieza actual se comprueba en el espacio que está encima. Si aún así no cabe, se trata de ubicar en el espacio al costado.
  - Cuando la pieza encaja en un nivel, la caja es ubicada en la esquina inferior izquierda del mismo. En ese caso, debe comprobarse si la caja entra sin dejar espacio libre. Si es así, el nivel está completo y se puede eliminar de la lista de niveles disponibles. Si no, significa que todavía hay algo de espacio libre y es necesario crear un nuevo conjunto de niveles (delante, arriba y al costado) para la nueva caja.
  - En cualquier momento, al comprobar un nivel dado, se debe comprobar el creado más recientemente.

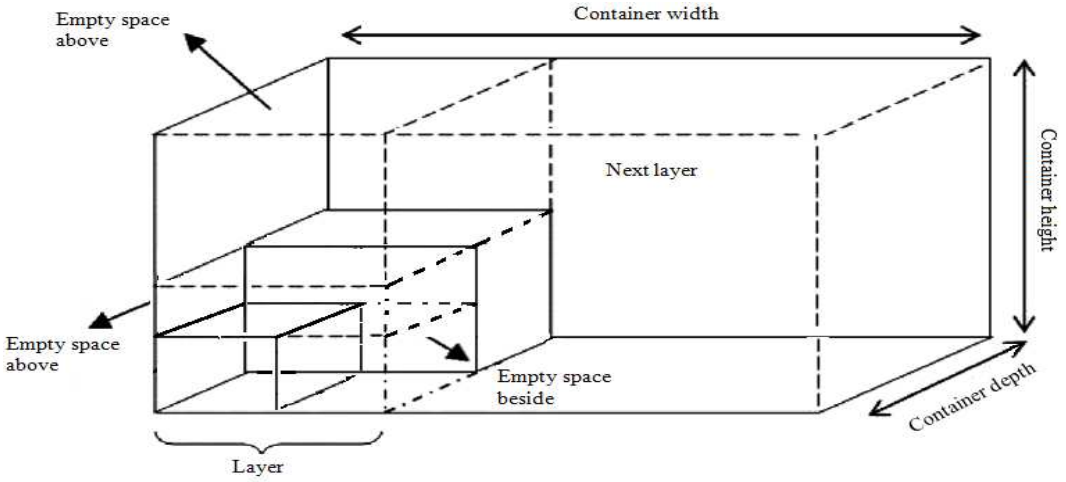

Figura 3.3: MLFH: Segunda iteración

- Cuando la caja analizada no entra en ninguno de los niveles disponibles, el procedimiento termina, y no se colocan más piezas en el contenedor. En ese momento, es posible calcular el valor de los objetivos (volumen y peso totales) sumando el volumen y el peso de las piezas introducidas.

La diferencia entre las heurísticas *SLFH* y *MLFH* consiste en que la heurística *SLFH* nunca usa la capa ubicada encima de la caja colocada si hay otro espacio vacío que pueda alojar items; mientras que en la heurística *MLFH* todas las capas están disponibles.

Estos algoritmos han sido testeados con el *framework* METCO [40], que permite al usuario final incorporar sus propios problemas y algoritmos evolutivos de manera sencilla. La herramienta permite realizar personalizaciones muy flexibles a través de la implementación de plugins adicionales que incorporen nuevas funcionalidades. El *framework* también proporciona un conjunto de librerías con operaciones estándar en computación evolutiva.

3.4. Entrada y salida del algoritmo

Las aproximaciones desarrolladas hasta el momento trabajan con un fichero de entrada en texto plano, a través del cual se especifican las características del problema a resolver. A continuación se muestra un ejemplo:

```
20 10 10
200
2
5 0 20 0 5 0 3 10
10 0 5 0 5 0 3 10
```

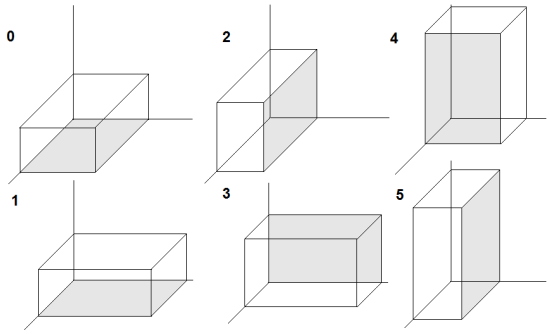

Figura 3.4: Tipos de rotación de las cajas.

En la primera línea se describe el tamaño del contenedor, correspondiéndose el primer parámetro con el largo, el segundo con el ancho y el tercero con el alto. En la segunda línea se especifica el peso máximo que puede transportar el contenedor. En la tercera línea se especifica el número de items o cajas distintas que se desean alojar o transportar en el contenedor. Posteriormente hay una línea por cada tipo de caja, en las que el primer, tercer y quinto número son el largo, ancho y alto respectivamente. En esas mismas líneas, el segundo, cuarto y sexto valor solo pueden ser cero o uno y señalan si, al ser introducidas en el contenedor, las cajas pueden ser rotadas en cada uno de los ejes o dimensiones. Por último, el séptimo y octavo parámetro definen el peso y el número de cajas de ese tipo que tendrá el problema. De forma general, se puede decir que la especificación de un fichero de entrada (que define un problema en concreto) tendrá el formato siguiente:

```
Largo | Ancho | Alto
Peso Admitido
Número de tipos de cajas
Largo | Rotación | Ancho | Rotación | Alto| Rotación | Peso | Número de cajas
```

Tras aplicar el mecanismo de resolución mencionado, se genera un fichero, en texto plano, que representa la solución propuesta para el problema de entrada. A continuación se muestra un ejemplo:

```
0 1 0 0 0
1 4 5 0 0
0 1 0 5 0
1 4 5 0 5
1 4 5 0 10
1 4 5 0 15
```

Como se puede observar, todas las líneas siguen el mismo formato, habiendo una línea por cada caja de la solución (no por cada tipo, sino por cada una de las cajas concretas

que han sido introducidas en el contenedor). En cada una de ellas se nombra el tipo de caja con un valor numérico entre uno y el número de tipos de caja; la rotación, que en esta ocasión se establece mediante una cifra entre cero y cinco, como se puede observar en la Figura 3.4; y por último las coordenadas en las que se ubicará la caja. En este caso, se generaliza el formato de la siguiente manera:

```
Tipo de caja | Tipo de Rotación | Posición X | Posición Y | Posición Z
```

Dado el problema y la especificación de ficheros presentados, se diseñará una interfaz que permita al usuario visualizar gráficamente tanto las instancias como las soluciones del problema. Además tendrá múltiples opciones interactivas con el objetivo de facilitar la comprensión de la solución.

Capítulo 4

Diseño e Implementación de la Interfaz Gráfica

En este capítulo se explicará cómo está implementada la interfaz: las clases que la forman, así como sus métodos, y las relaciones entre ellas, que constituyen el diseño. Posiblemente sea el que necesite de mayor atención por parte del lector, puesto que le permitirá entender lo que sucede internamente en los casos de uso que se expondrán más adelante. Para facilitar la comprensión, primero se describirá la interfaz gráfica de usuario.

4.1. La interfaz

Como se puede apreciar en la Figura 4.1, la aplicación tiene dos «zonas» bien diferenciadas: la zona de la solución, ubicada en la parte superior de la imagen, que engloba los elementos del uno al cinco; y la zona de tipos de caja, en la parte inferior, donde hay un *Canvas* o controlador por cada tipo y un *slider* que posibilita hacer zoom en todos ellos a la vez.

Antes de seguir con esta sección, es importante hablar del *modo libre*, cuya interfaz se puede observar en la Figura 4.2. Básicamente es un estado del controlador principal en el que se habilita la interacción con las cajas, pudiendo arrastrarlas a cualquier posición del espacio. Su principal utilidad es ampliar la funcionalidad de la *máquina de estados* (de la que se hablará posteriormente) para poder explorar todos los rincones sin estar obligado a ocultar las cajas o seguir un orden preestablecido al tratar con ellas.

A continuación se describen los elementos de la interfaz remarcados en la Figura 4.1:

1. Controlador principal: en él se puede observar la solución dentro del contenedor. Cuando el usuario entra en el *modo libre* da respuesta a eventos del ratón para desplazar las cajas.

2. Etiquetas: reflejan datos acerca de lo que se está mostrando en el controlador asociado. En el caso de las etiquetas del controlador principal, además de informar sobre

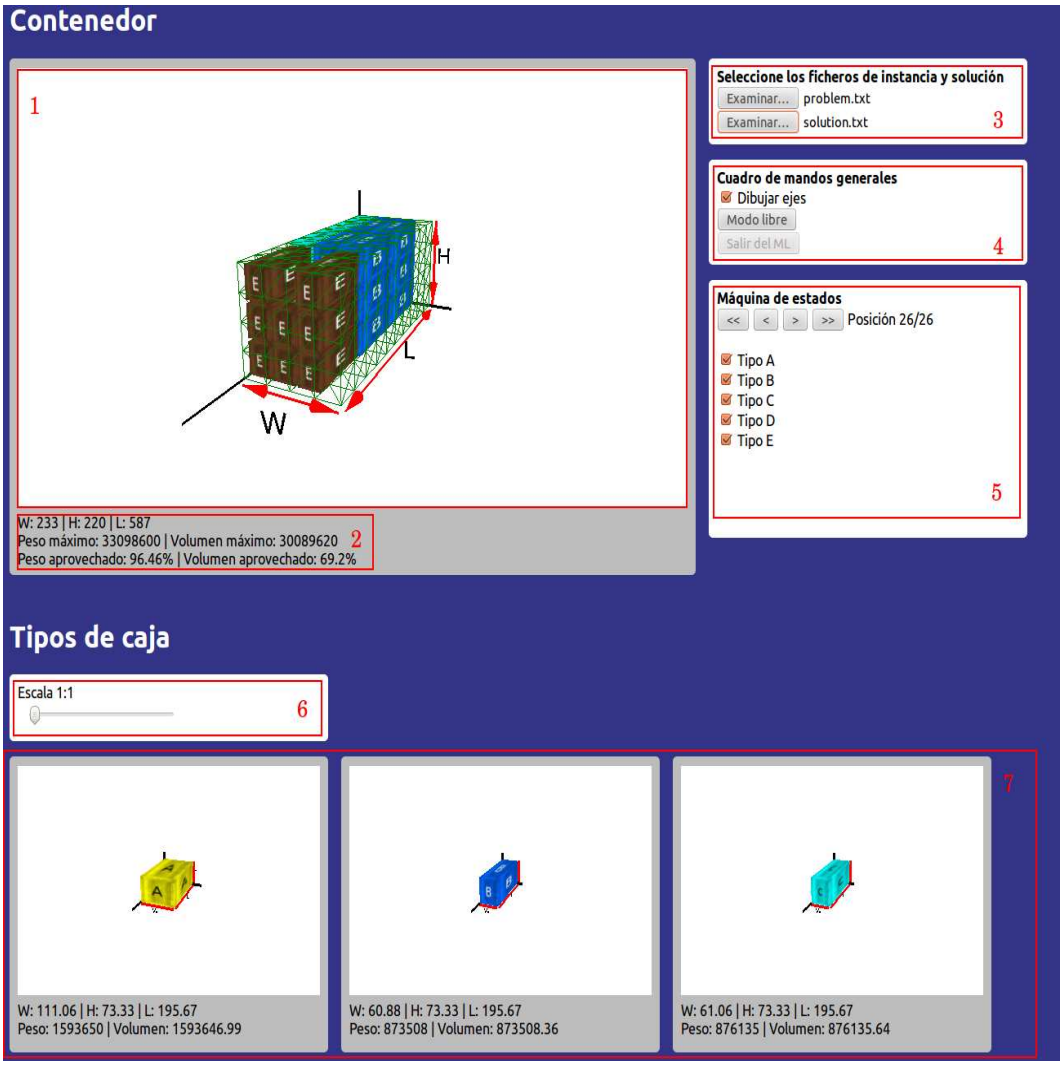

Figura 4.1: Imagen de la interfaz

datos propios del contenedor (como las dimensiones, peso y volumen máximos), también permiten al usuario saber qué porcentaje de peso y de volumen se aprovecha con las cajas visibles en ese momento.

3. Panel de carga de ficheros: los botones de este panel permiten cargar los ficheros de problema y solución.

4. Panel de mandos generales: contiene elementos que disparan eventos de diversa naturaleza. El *checkbox* sirve para mostrar u ocultar los ejes de coordenadas en todos los controladores de la interfaz, incluidos los de la zona de tipos de caja. Los dos botones constituyen la salida y entrada al *modo libre*.

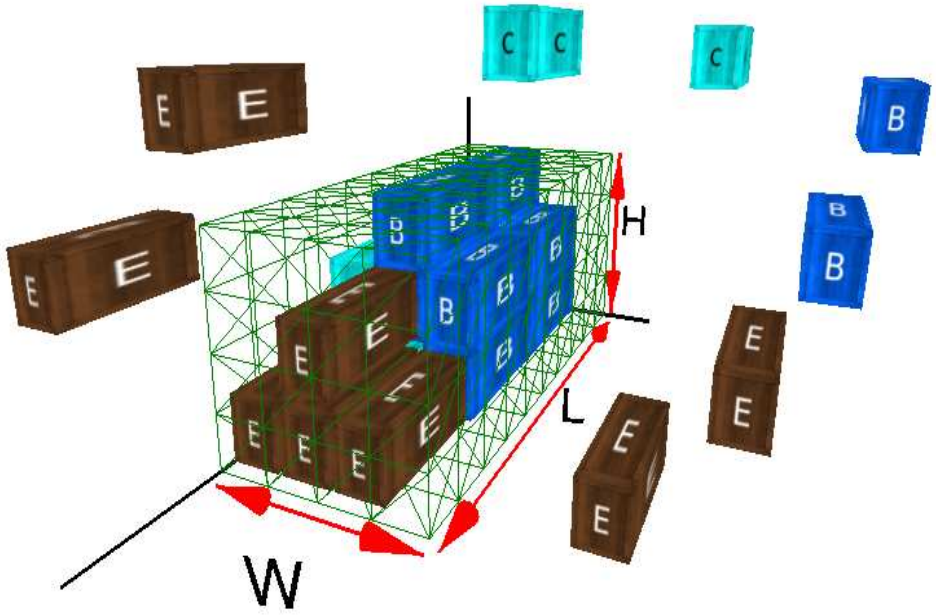

Figura 4.2: Captura del *modo libre*

5. Panel de la *máquina de estados*: una vez cargada la solución, al usuario puede interesarle esconder ciertas cajas para apreciar mejor los detalles de la distribución. Para ello se ponen a su disposición dos métodos: el primero es la *máquina de estados*, mediante la cual se puede hacer un recorrido, iteración a iteración, de cómo se ha formado la solución. En el ejemplo de la imagen, la *máquina* se encuentra justo al final, con todos los items ubicados. La segunda forma es esconder las cajas de un tipo concreto. Esto se puede llevar a cabo pulsando el *checkbox* del tipo deseado.

6. *Slider* de la escala: aproxima o aleja la posición de la cámara de las figuras en todos los controladores de la zona de tipos de caja a la vez.

7. Controladores secundarios: contienen un tipo de caja cada uno. Sus cámaras se encuentran en la misma posición, permitiendo captar las diferencias de tamaño.

4.2. El Diseño

En primer lugar, si se contempla el conjunto de la interfaz y el algoritmo, se concluye que puede ser dividido en tres capas, como se deduce de la Figura 4.3. La más baja estaría formada por el motor de resolución; la capa intermedia quedaría compuesta por los ficheros que el motor tiene como entrada y salida; por último, la capa superior, más cercana al usuario, sería la interfaz, que tomaría los ficheros procedentes del motor para presentar la información contenida en ellos de manera gráfica, facilitando su interpretación.

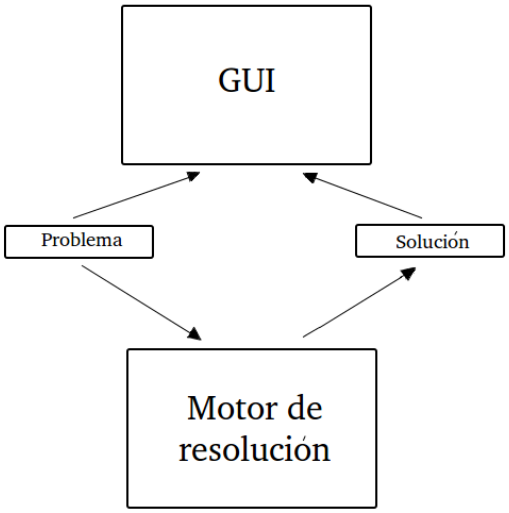

Figura 4.3: Esquema del conjunto.

Por otra parte, cuando se habla específicamente de la interfaz de usuario, resulta interesante analizar el esquema que representa las relaciones entre las diferentes clases que la integran. Para hacerlo más comprensible se ha divido en tres.

- La Figura 4.4 está dedicada a la clase `Init.js`. Tiene una relación de asociación con las clases `Controller.js` y `Box.js`, pues almacena como atributos un vector de controladores que da soporte, por ejemplo, a las acciones de ocultar y dibujar los ejes de coordenadas, y otro vector con los tipos de cajas del problema. En cambio, con las clases `ProblemFile.js`, `SolutionFile.js` y `Container.js` solo mantiene dependencia, porque las utiliza en sus métodos.

- La Figura 4.5 muestra que la clase `Controller.js` está asociada a las clases `Container.js`, `Box.js`, `DrawableContainer.js` y `DrawableBox.js`. Esto no siempre es así: al no ser JavaScript un lenguaje de tipado fuerte, el atributo que contiene la figura principal puede alojar una instancia de `Controller.js` o `Box.js` según cuál sea la figura principal del controlador y, como consecuencia, ocurrirá esta misma variación con el atributo que contiene la figura dibujable. En el caso de que se trate del controlador principal, se tiene la certeza de que habrá un vector de elementos de la clase `Box.js` y otro de `DrawableBox.js`, pues son necesarios para mostrar y operar con las cajas.

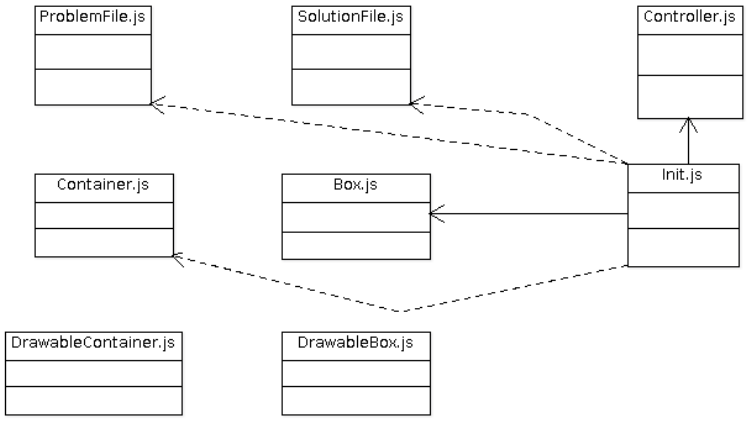

Figura 4.4: Relaciones de la clase `Init.js` con el resto.

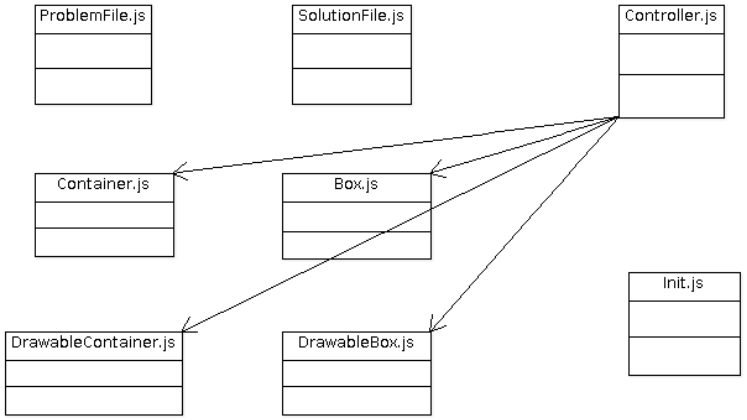

Figura 4.5: Relaciones de la clase `Controller.js` con el resto.

- Finalmente, la Figura 4.6 indica que hay dependencia de las clases `Container.js` y `Box.js` frente a `DrawableContainer.js` y `DrawableBox.js` ya que en el método «`replicate`» de las primeras se crean instancias de las segundas.

4.3. Las clases y sus métodos públicos

`DrawableContainer.js` y `DrawableBox.js`: constituyen las figuras que el usuario puede ver en la pantalla y con las que puede interactuar en el *modo libre*. Durante este trabajo

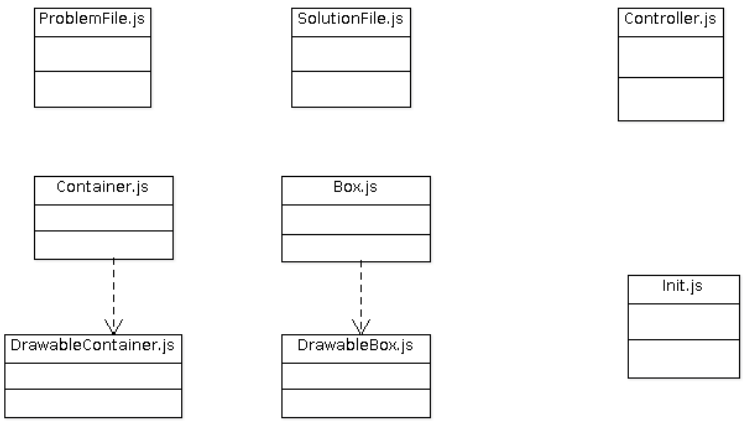

Figura 4.6: Relaciones de las clases `Container.js` y `Box.js` con sus respectivas *drawables*.

las llamaremos *figuras, objetos o clases dibujables*. Son clases muy similares pero la diferencia fundamental está constituida por los métodos exclusivos que tiene `DrawableBox.js` para dar soporte a la *máquina de estados* y el *modo libre*.

- `show` y `hide`: estos métodos están presentes en ambas clases, sirven para dibujar y borrar la figura en la escena que se le pase por parámetro.
- `clone`: solo está presente en la clase «`DrawableBox.js`», crea una copia exacta de la caja. Este método es utilizado para hacer la copia del vector de figuras secundarias.
- `position`: solo está presente en la clase «`DrawableBox.js`», devuelve la posición del dibujo en el espacio.

`Container.js` y `Box.js`: son clases que representan a las figuras en sí mismas, de manera general, mediante su alto, ancho, largo, peso, volumen, tipo y textura, pero no se pueden dibujar. A simple vista, la existencia de estas clases y las dos anteriores puede dar la sensación de duplicidad, pero este diseño evita la redundancia de datos y reduce la complejidad del código. Si no se tuvieran `DrawableContainer.js` y `DrawableBox.js`, cada figura mostrada en la interfaz tendría todos los datos de su tipo, obviamente repitiéndose en las del mismo e incluso almacenando algunos datos innecesarios para la visualización. Otro aspecto relevante es la facilidad para dibujar cajas que ofrece esta división, pues `Box.js` almacena y utiliza automáticamente la información del fichero del problema, evitando esta tarea al programador, de modo que al crear una caja dibujable solo hacen falta los datos del fichero de solución y un vector con los objetos `Box.js`.

- `replicate`: mediante este método se produce un objeto dibujable, teniendo en cuenta las coordenadas en el espacio y la rotación de la pieza en concreto. Está presente

en las dos clases. La figura tiene un nombre que permite un borrado posterior.

`Controller.js`: es la clase más extensa de la aplicación, se encarga de controlar todo lo que el usuario ve en el *Canvas*. Muchos de sus métodos son imprescindibles en el *Canvas* principal, porque posibilitan la interacción del usuario dando respuesta a sus peticiones. También se utiliza en los *Canvas* de la zona de tipos de caja, pero en esa situación su participación es reducida debido a que se requiere un menor grado de interactividad.

- `drawElement` y `deleteElement`: son métodos simples, que se utilizan al inicializar la interfaz para mostrar al usuario un mensaje diciéndole que debe seleccionar un fichero. La primera añade un elemento de cualquier tipo a la escena y la segunda recibe un parámetro con el nombre de la figura a borrar. Con ellas se dibuja directamente sobre el *Canvas*. Se usa solamente para el mostrar el mensaje inicial.

- `addMainFigure`: recibe un objeto de la clase «`Container.js`» o «`Box.js`», que almacena en un atributo, y a partir de él crea uno de la clase dibujable que corresponda. Posteriormente utiliza métodos privados para dibujar las cotas e imprimir las etiquetas. Solo habilita los controles con el ratón si el elemento recibido es un contenedor, lo cual significa que es el *Canvas* principal.

- `deleteMainFigure`: borra la figura principal, sus cotas y los ejes.

- `moveCamera`: recibe las coordenadas de la nueva posición de la cámara y hace que mire al centro de la escena.

- `setAxes`: si existe una figura principal, entonces dibuja o borra los ejes de coordenadas según corresponda. Esto se sabe mediante un atributo, al igual que se hace en la clase «`Init.js`».

- `addSecondaryFigures`: recibe una matriz con los datos del fichero en la que cada fila se corresponde con una caja dibujable. Invoca a la función «`replicate`» de cada tipo de caja con los datos recibidos para crear cajas dibujables, éstas luego son insertadas en un vector de figuras secundarias que da soporte a múltiples acciones interactivas. Por último, inicializa la *máquina de estados*.

- `deleteSecondaryFigures`: borra las figuras de la escena y elimina todos los objetos del vector de figuras secundarias.

- `goBack`, `goForward`, `toTheBeginning` y `toTheEnd`: son funciones que se llaman desde los listeners de la clase «`Init.js`» y que realizan operaciones relativas a la *máquina de estados*. Lo único que hacen es variar el atributo que almacena el estado, y por cada variación (suma o resta) se dibuja o esconde una caja. Las dos primeras solamente hacen esto una vez, mientras que las otras dos lo hacen cuantas veces sea posible, hasta que el estado no pueda ser mayor («`toTheEnd`») o menor («`toTheBeginning`»). Cada vez que una de ellas es ejecutada, se actualizan los datos de aprovechamiento en las etiquetas.

- `deactivateGroup` y `activateGroup`: recorren el vector de figuras secundarias, con elementos de la clase `DrawableBox.js`, y ocultan o dibujan, respectivamente, las cajas que sean del tipo que se les especifica por parámetro. Cada vez que una de ellas es ejecutada, se actualizan los datos de aprovechamiento en las etiquetas.
- `labelCleaner`: borra todas las etiquetas desde el árbol del documento HTML.
- `activateFm`: declara los cuatro listeners, que se describirán a continuación, que captan los eventos del ratón dentro del *Canvas*. Esconde el panel de la *máquina de estados*, puesto que en el *modo libre* no es posible interactuar con él, oculta las etiquetas de aprovechamiento momentáneamente y crea una copia del vector de figuras secundarias que se utilizará al salir para devolver la interfaz exactamente al mismo estado en el que se encontraba antes de entrar. Para poder determinar qué objetos se seleccionan con el ratón, se crea un plano y se añade a la escena.
- `deactivateFm`: en rasgos generales hace todo lo contrario que el método anterior, es decir, remueve los listeners y el plano, vuelve a mostrar el panel de la *máquina de estados* y recupera la copia del vector de figuras secundarias para dejar todo como se encontraba antes.
- `OnDocumentMouseDown`: este evento se dispara cuando se hace clic (sin soltar). Se utiliza el plano para averiguar qué objeto hay debajo del puntero y se selecciona.
- `OnDocumentMouseMove`: evento que se produce cuando el ratón se mueve, y que asigna a la figura seleccionada actualmente, si la hay, dicha posición.
- `OnDocumentMouseUp`: es el contrario de `onDocumentMouseDown`, se activa cuando se suelta el botón izquierdo del ratón. Deshace la selección.
- `OnDoubleClick`: cuando el usuario hace doble clic, si hay una figura debajo del puntero, se busca en la copia del vector de figuras secundarias su posición original y se coloca ahí.
- `cleanVectors`: borra la copia del vector de figuras secundarias y otro que se utiliza para la selección de objetos, no interviene en la interfaz, es solamente para evitar errores.

`ProblemFile.js` y `SolutionFile.js`: son clases muy similares, que cumplen un objetivo muy parecido, aunque debido a que leen ficheros con formatos distintos, difieren un poco entre sí.

- `complyFormat`: este método se encarga de revisar la totalidad del fichero, comprobando línea por línea si el formato es correcto. Para este fin dispone de una serie de constantes con expresiones regulares específicas para cada tipo de línea.
- `readLine`: no recibe ningún parámetro de entrada. Devuelve la primera línea en la primera llamada y así sucesivamente.

- `reset`: establece a cero el valor del atributo usado por el método anterior para llevar la cuenta de la línea que corresponde devolver.
- `size`: solo está presente en la clase «`SolutionFile.js`», devuelve el número de líneas del fichero.

`Init.js`: esta clase actúa como *listener* de la mayoría de eventos que el usuario genera durante su interacción con la interfaz.

- `init`: es el método que se ejecuta cuando el usuario carga la página. Crea el *Canvas* principal, adaptándolo al tamaño del monitor. Declara los eventos necesarios para poder cargar los ficheros de entrada y salida, aunque el botón que dispara el segundo no esté visible en este punto del flujo de trabajo.
- `problemFileEvent`: es el encargado de gestionar la carga del fichero que contiene el problema y lleva a cabo una tarea de limpieza de la interfaz cuando hay otro fichero abierto previamente. Crea un objeto de la clase «`ProblemFile.js`» y mediante él lee línea a línea. Aquí se crea el contenedor, que es pasado al controlador de la zona de solución como figura principal. En base al tamaño del contenedor se establece una posición para la cámara, que compartirán todos los *Canvas* de la interfaz. La tarea de crear un controlador para cada tipo de caja es delegada al método «`createBox`».
- `createBox`: recibe las características de la caja y la posición de la cámara. Crea un controlador, que se ubicará en la zona de tipos de caja, y le asigna la caja como figura principal. Luego guarda el controlador y la caja en un vector que servirá posteriormente para dar soporte a diferentes acciones, como por ejemplo el cambio en la escala que se produce con el método «`scaleEvent`».
- `axesEvent`: capta el evento de activación o desactivación de los ejes de coordenadas, para saber en qué caso está, tiene una variable. Recorre todos los controladores que haya en ese momento y utiliza la función «`setAxes`» pasándole como parámetro un 0 si hay que borrar o un 1 si hay que dibujar.
- `scaleEvent`: escucha los eventos del *slider* dedicado a la escala, simplemente establece la distancia de la cámara con la escena, siendo la nueva una división de la original entre el número escogido.
- `solutionFileEvent`: es el método que carga el segundo fichero de entrada, es decir, el de la solución. En primer lugar limpia los vectores utilizados en la interacción del *modo libre* y borra las figuras secundarias. Con la ayuda de una instancia de la clase «`SolutionFile.js`» lee las líneas y crea una matriz que es dada al controlador directamente, que es quien realiza la mayor parte del trabajo.
- `activateType` y `deactivateType`: una vez cargada la solución, en un momento dado al usuario le puede interesar ocultar un tipo concreto de cajas. Estos eventos se encargan de ello, para lo cual reciben el tipo de caja y simplemente se lo pasa

al controlador principal, que se encarga de recorrer el vector de cajas dibujadas, comprobar su tipo y si coincide esconderlas o mostrarlas.

- `backBtn`, `forwardBtn`, `startBtn`, `endBtn`, `activateFmBtn`, `deactivateFmBtn`: si bien estos seis listeners recogen eventos de distinta clase, todos ellos tienen una cosa en común: no ejecutan ningún cambio sobre la interfaz, solo invocan a una función del controlador (distinta en cada caso).

Capítulo 5

Casos de uso

En este capítulo se explicará qué ocurre internamente en la aplicación durante los diferentes casos de uso. Es muy importante tener en cuenta la información del capítulo anterior para poder comprender exactamente el flujo de trabajo.

Caso de uso 1: Cargar la pantalla

1. Lo primero que ocurre al entrar en la página es que se carga la clase «`Init.js`» y se ejecuta el código que se encuentra fuera de sus funciones. En él se crean vectores que serán utilizados más adelante, por ejemplo el de controladores, y se inicializan algunas variables.

2. Lo siguiente que sucede es la ejecución del método «`init`», donde se crea el controlador principal y se dibuja el mensaje inicial.

Figura 5.1: Botones de carga de ficheros.

Caso de uso 2: Cargar el fichero del problema

1. El usuario inicia la acción pulsando el botón «`Examinar`», que aparece en el recuadro 1 de la Figura 5.1, y escogiendo un archivo. Ese evento es captado por la clase «`Init.js`».

2. Se crea un objeto de la clase «`ProblemFile.js`» al que se le da el fichero que se lee. Comprueba que el formato es correcto, si no lo es lanza un mensaje de error.

3. Se borran los objetos que pudiera haber en el *Canvas* principal, tanto figuras principales como secundarias.

4. A partir del objeto anterior, si el formato es correcto, se lee cada línea con el método «`readLine`» desde la clase «`Init.js`».

5. Con los datos obtenidos se crea una instancia de la clase «`Container.js`» y se pasa al controlador principal, que se encarga de conseguir un objeto dibujable a partir de ella. Tomando como referencia las medidas del contenedor se reubica la cámara. Esta posición se utilizará también en los *Canvas* de la zona de tipos de caja. Esto es así para que pueda apreciarse la diferencia te tamaño entre los elementos que forman parte del problema.

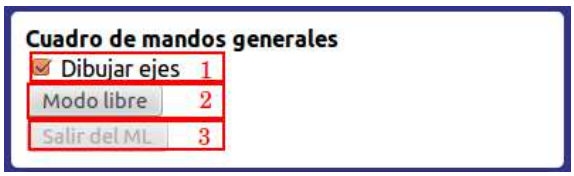

Figura 5.2: *Checkbox* de los ejes y botones del *modo libre*.

Caso de uso 3: Dibujar y borrar los ejes de coordenadas

1. El usuario cambia el estado del *checkbox* (recuadro 1 en la Figura 5.2). Este evento es captado por la clase «`Init.js`».

2. El método «`checkboxEvent`» distingue si corresponde dibujar o borrar los ejes y recorre el vector de controladores, utilizando el método «`setAxes`» de cada uno de ellos.

Figura 5.3: *Slider* de la escala.

Caso de uso 4: Cambiar la escala en los tipos de caja

1. El funcionamiento de este caso de uso es idéntico al anterior. El usuario cambia la escala, con el *slider* que se puede ver en la Figura 5.3, y el evento que se emite es captado por la clase «`Init.js`».

2. En el método «`scaleEvent`» se calcula la nueva distancia de la cámara respecto al punto (0,0) y se recorre el vector de controladores, utilizando el método «`moveCamera`» en cada elemento.

Caso de uso 5: Cargar el fichero de la solución

1. El usuario pulsa el botón «`Examinar`», que aparece en el recuadro 2 de la Figura 5.1, y escoge un archivo. Ese evento es captado por la clase «`Init.js`».

2. Se crea un objeto de la clase «`SolutionFile.js`» al que se le pasa el fichero que se lee. Comprueba que el formato es correcto, si no lo es lanza un mensaje de error.

3. A partir del objeto anterior, si el formato es correcto, se obtiene cada línea con el método «`readLine`» desde la clase «`Init.js`».

4. Se borran todas las figuras secundarias que pudieran existir con anterioridad.

5. Una vez obtenidas todas las líneas, éstas son asignadas al controlador en forma de matriz, en la cual cada fila se corresponde con una línea del fichero.

6. El controlador, que tiene un vector con los tipos de cajas, utiliza el método «`replicate`» del que corresponda para generar los objetos dibujables de acuerdo a los datos leídos.

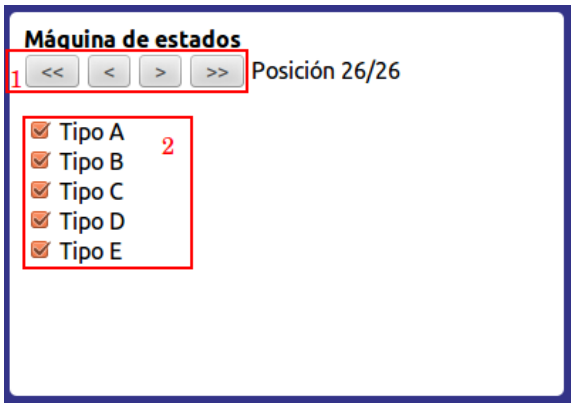

Figura 5.4: Botones de la *máquina de estados* y *checkbox* de los grupos.

Caso de uso 6: Pulsar los botones de la *máquina de estados*

1. El usuario pulsa un de los botones de la máquna, de los cuatro que se pueden observar en el recuadro 1 de la Figura 5.4. El evento es captado por la clase «`Init.js`».

2. Esta clase solamente llama al método correspondiente del controlador.

3. El controlador, dependiendo de si la orden es de retrasar o adelantar posiciones de la *máquina*, utiliza el método «`hide`» o «`show`» de las cajas dibujables para realizar la acción que corresponda.

Caso de uso 7: Activar y desactivar tipos de cajas

1. El usuario pulsa un *checkbox* (Figura 5.4, recuadro 2). El evento es captado por la clase «`Init.js`».
2. Esta clase solamente llama al método «`deactivateGroup`» o «`activateGroup`» del controlador, que recorre el vector de cajas dibujables.
3. Si el tipo de una caja coincide con el que se recibió por parámetro, se utiliza el método «`hide`» o «`show`», según corresponda, para realizar la acción.

Caso de uso 8: Entrar y salir del *modo libre*

1. El usuario pulsa uno de los botones que se pueden apreciar en los recuadros 2 y 3 de la Figura 5.2. El evento es captado por la clase «`Init.js`».
2. Esta clase solamente llama al método «`activateFm`» o «`deactivateFm`» del controlador.
3. Cuando se entra al *modo libre*, el controlador inicializa los objetos necesarios para averiguar el punto que señala el ratón y guarda una copia del vector de cajas materiales, para poder recuperar la *máquina de estados* exactamente como el usuario la deja al entrar. Cuando se sale, el controlador borra el vector de cajas y lo sustituye por la copia guardada. Borra los objetos que creó antes.

Caso de uso 9: El *modo libre* y sus acciones

1. En esta ocasión los eventos se generan debido a la interacción del usuario con el controlador, y es él mismo quien se encarga de captarlos.
2. Hay cuatro casos:
   - Si el usuario presiona el clic izquierdo (sin soltarlo): el controlador identifica el elemento seleccionado (si lo hay) y lo marca.
   - Si el usuario arrastra el ratón con el clic izquierdo presionado: el controlador modifica la posición del elemento seleccionado para situarlo allí donde vaya el puntero.
   - Si el usuario suelta el clic izquierdo: el controlador quita la marca de seleccionado al elemento, por tanto, aunque se mueva el ratón, la caja no lo hará.
   - Si el usuario realiza un doble clic izquierdo: cuando esta acción se produce sobre una caja, el controlador le asigna su posición original.

Capítulo 6

Conclusiones y trabajos futuros

Desde que surgiera la Web, a finales de la decada de los 80, su evolución constante ha enriquecido mucho su interactividad y ha convertido una plataforma que en un principio solo podía contener texto en un mundo que sus creadores nunca habrían imaginado. En ese contexto, el objetivo de este trabajo es el estudio del estado del arte en el campo de los gráficos tridimensionales en la web. Para llegar a él se ha propuesto una meta concreta: construir una aplicación web dedicada a la visualización de instancias y soluciones del problema de la carga de contenedores. Durante las fases de análisis previas al desarrollo se han evaluado distintas librerías, con algunas de las cuales se han hecho pruebas, lo que ha permitido extraer sus fortalezas y debilidades. Además, los requerimientos del proceso han servido para saber qué es lo que se puede hacer con este tipo de tecnologías. Por otra parte, esta memoria puede ser vista como la documentación de ese producto, ya que contiene una descripción profunda del diseño, las clases y los métodos utilizados, así como una explicación de lo que ocurre internamente y cómo se relacionan las clases en cada caso de uso.

Durante el desarrollo de la aplicación han surgido diversos contratiempos. Un buen ejemplo de ello son algunos rediseños de las clases, que se tuvieron que hacer en determinadas fases del proceso, como la separación de las cajas y los contenedores en dos clases, las de los objetos y sus *drawables*. También fue compleja la tarea de desarrollar y a la misma vez efectuar pruebas, porque no es fácil abstraerse, es decir, cuando una persona se encarga de la programación, fija su atención en ciertos aspectos y, si tiene que realizar tests, son los que más va a mirar. Siempre es bueno que haya una segunda persona que pruebe, sin tener esos modelos mentales previos. Esto último es, quizás, lo peor de trabajar de manera individual, ya que las ideas de otras personas enriquecen mucho el producto final.

Lo anterior está muy relacionado con algunas de las competencias que se desarrollan con el Trabajo de Fin de Grado. La capacidad para resolver problemas con iniciativa y de actuar autónomamente han estado muy presentes por la naturaleza propia del TFG. Y a pesar de lo dicho, sí se ha tenido la oportunidad de evaluar soluciones alternativas y argumentar las decisiones tomadas, pues las tutoras han hecho propuestas de cambio y sugerido nuevas ideas.

Por otra parte, este trabajo ha dado lugar a una comunicación en el congreso *Interacción 2014* [38], en la categoría de visualización de datos.

Finalmente, son diversos los trabajos que se podrían llevar a cabo con el resultado obtenido, por ejemplo:

- Implementar animaciones con la *máquina de estados*, para que el usuario no necesite hacer clic tantas veces como cajas haya, sino que el proceso de llenado del contenedor se muestre automáticamente.
- Permitir que el usuario expanda y contraiga las distintas secciones de la , a modo de *tiles*, para que pueda ver mejor lo que le resulte más importante en cada momento.
- Integrar el motor de resolución con la interfaz, para evitar la selección de ficheros.
- Mejorar la calidad gráfica, buscando otras tecnologías de distinto tipo.

Capítulo 7

Summary and Conclusions

Since the Web appeared, at the end of the 1980s, its constant evolution has improved its own interactivity and has transformed a plataform, which at the beginning was only able to deal with text, in a world that its makers would have never imagined. In this context, the goal of this work is to study the state of 3D Graphics on Web Interfaces. In order to achieve such a goal, a specific objective has been proposed: build a Web interface to visualize Container Loading Problem examples. Some libraries have been analyzed and a few of them have been tested to evaluate their pros and cons. This analysis has helped to choose the implementation framework. Furthermore, the process requirements served to find out what can be done with this kind of technology. Otherwise, this document can be seen as the developer and user documentation, because it contains a description of the design, the classes and methods, as well as a case study.

During the development, different setbacks have emerged. A good example is the redesign of the classes that divided `Box.js` and `Container.js` into two classes each one. Development and testing were a complex task. When only one person takes care of programming, he/she must pay attention to many aspects. Then, it is very difficult to maintain all of them under control. It is always better to work with someone else who tries out the software. The latter is the worst of working alone, since another person could contribute with new ideas.

This is a point which is related to some of the degree skills. The capacity to solve problems with initiative has been present due to the work nature. Despite the latter, this work has brought the opportunity to evaluate alternatives and defend decisions, because supervisors have posed changes and suggestions.

The result of this work has been published as a communication in the *Interacción 2014* Conference [38].

Finally, the software produced in this work could be extended in the following ways:

- Animations can be implemented through the *state machine*, in order to avoid clicks,

it would not be necessary one click per box.

- Expansion and conctraction of the interface can be implemented, so that users can focus on specific areas.
- Integration of the resolution engine and the user interface can be implemented to avoid selecting files.
- Improvement of the graphics quality through the use of other technologies.

Capítulo 8

Presupuesto

En la Figura 8.1 se pueden observar las tareas realizadas a lo largo de este trabajo, así como su distribución temporal en este curso 2013/2014. Un hecho a destacar es que se ha generado documentación durante todo el proceso, es decir, esta tarea ha sido paralela a todas las demás.

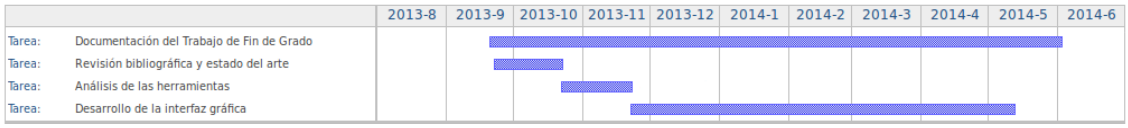

Figura 8.1: Diagrama de Gantt.

Como complemento, en la Tabla 8.1 se especifican las horas invertidas aproximadamente en cada tarea. Si se toma como referencia un precio de 10 euros la hora, se puede concluir que el coste estimado de este trabajo es de 3150 euros.

Tareas	**Horas**
Revisión bibliográfica y estado del arte	35 horas
Análisis de las herramientas	65 horas
Diseño y desarrollo de la interfaz gráfica	150 horas
Documentación del trabajo	65 horas
TOTAL	**315 horas**

Tabla 8.1: Estimación de horas invertidas en cada tarea

Un hecho a comentar es que para la planificación, la gestión y el control de las tareas vinculadas al proyecto se ha utilizado el gestor de proyectos *Redmine* [26].

Bibliografía

[1] Blender. http://www.blender.org/.

[2] C3DL, Canvas 3D Library. http://www.c3dl.org/.

[3] Calculation Space. http://www.calculationspace.com/container-packing-software/.

[4] Comparación de librerías. http://weeraphan.com/?p=406.

[5] Copperlicht. http://www.ambiera.com/copperlicht/.

[6] Cube Master. http://www.cubemaster.net/Subscription/home.asp.

[7] CubicVR 3D. http://www.cubicvr.org/.

[8] Curve3D. https://github.com/sunetos/curve3d.

[9] GLGE. http://www.glge.org/.

[10] Gráficos 3D. es.wikipedia.org/wiki/Gr %C3 %A1ficos_3D_por_computadora.

[11] Gráficos 3D. www.ecured.cu/index.php/Gr %C3 %A1ficos_3D_por_computadora.

[12] Historia de CSS. http://en.wikipedia.org/wiki/Cascading_Style_Sheets#History.

[13] Historia de CSS. https://maicoleslomejor.wordpress.com/2012/05/29/historia-de-css/.

[14] Historia de JS. http://es.wikipedia.org/wiki/Javascript.

[15] Historia de JS. http://www.desarrolloweb.com/articulos/491.php.

[16] Historia de WebGL. http://es.wikipedia.org/wiki/WebGL#Historia.

[17] Historia de WebGL. http://www.students.science.uu.nl/∼3685632/content/history.html.

[18] Iluminación. www.neopixel.com.mx.

[19] Lista de librerías. http://webglframeworks.org.

[20] Logimar. http://logimar.it/binpack/index.php/it.

[21] Modelado 3D. en.wikipedia.org/wiki/3D_modeling.

[22] O3D. https://code.google.com/p/o3d/.

[23] OpenSceneGraph JS. http://osgjs.org/.

[24] Packer 3D. http://www.packer3d.com/.

[25] PhiloGL. http://www.senchalabs.org/philogl/.

[26] Redmine. http://www.redmine.org/.

[27] SceneJS. http://scenejs.org/.

[28] Searates. http://www.searates.com/reference/stuffing/.

[29] SpiderGL. http://spidergl.org/.

[30] TDL. https://github.com/greggman/tdl.

[31] ThreeJS. http://threejs.org/.

[32] Unity 3D. http://unity3d.com/.

[33] W3C. http://www.w3.org/.

[34] X3DOM. http://www.x3dom.org/.

[35] S.D. Allen, E.K. Burke, and G. Kendall. A hybrid placement strategy for the three-dimensional strip packing problem. *European Journal of Operational Research*, 209(3):219–227, 2011.

[36] A. Bortfeldt and D. Mack. A heuristic for the three-dimensional strip packing problem. *European Journal of Operational Research*, 183(3):1267–1279, 2007.

[37] Andreas Bortfeldt and Gerhard Wäscher. Container Loading Problems - A State-of-the-Art Review. Working Paper 1, Otto-von-Guericke-Universität Magdeburg, April 2012.

[38] Yanira González, Coromoto León, Gara Miranda, and Javier Villamonte. Graphical user interface for the container loading problem. In *Proceedings of the 15th ACM/AIPO International Conference on Human Computer Interaction, INTERACCION '2014, Puerto de la Cruz, Tenerife, Spain - September 10 - 12, 2014*, page to appear. ACM, 2014.

[39] Yanira González, Gara Miranda, and Coromoto León. A multi-level filling heuristic for the multi-objective container loading problem. In *SOCO-CISIS-ICEUTE*, pages 11–20, 2013.

[40] Coromoto León, Gara Miranda, and Carlos Segura. Metco: a parallel plugin-based framework for multi-objective optimization. *International Journal on Artificial Intelligence Tools*, 18(4):569–588, 2009.

[41] Guntram Scheithauer. Algorithms for the Container Loading Problem. In *Operations Research Proceedings 1991*, pages 445–452. Springer-Verlag, 1992.

[42] Yong Wu, Wenkai Li, Mark Goh, and Robert de Souza. Three-dimensional bin packing problem with variable bin height. *European Journal of Operational Research*, 202(2):347–355, 2010.