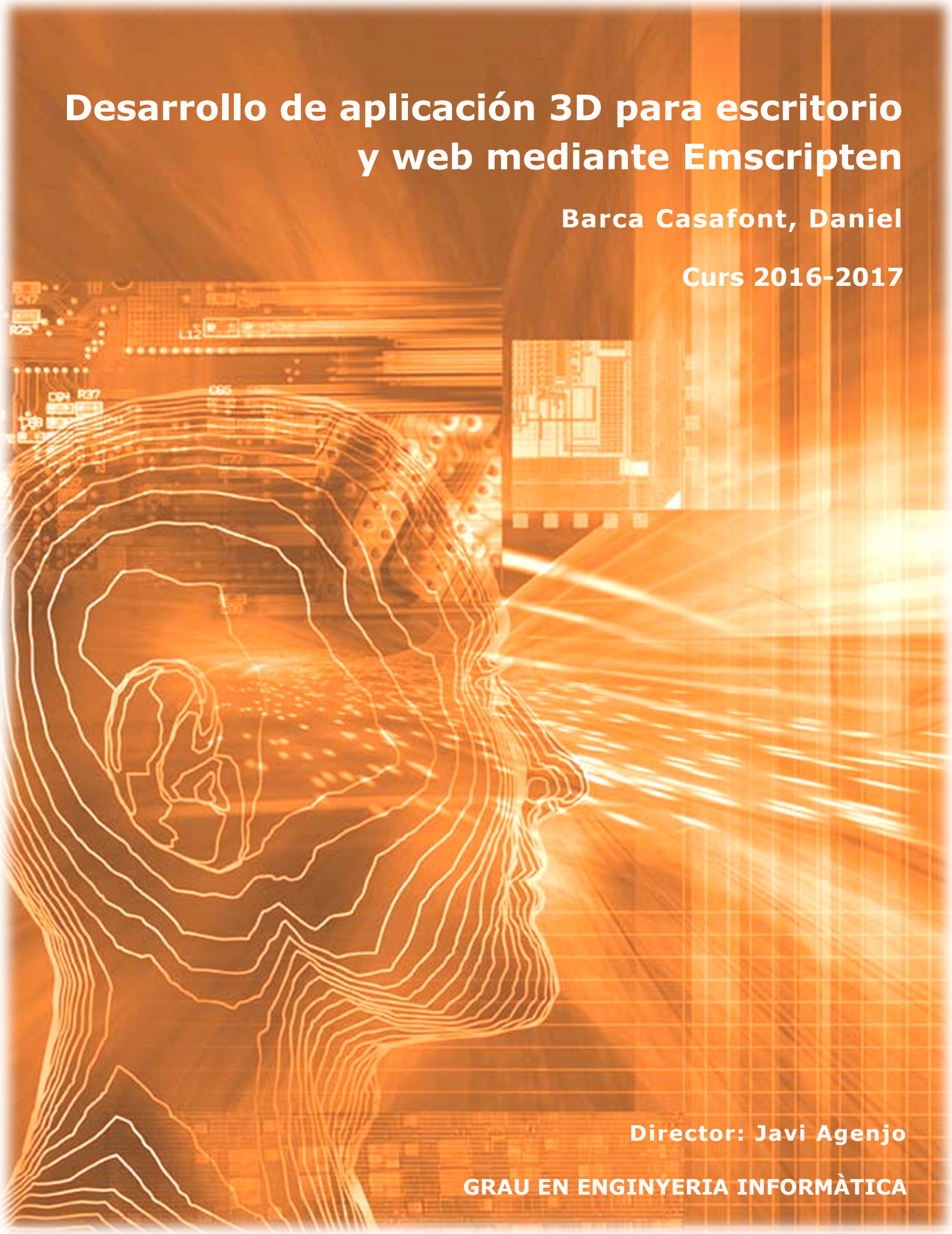
Desarrollo de aplicación 3D para escritorio y web mediante Emscripten

Barca Casafont, Daniel

Curs 2016-2017



Director: Javi Agenjo

GRAU EN ENGINYERIA INFORMÀTICA

upf. Universitat Pompeu Fabra Barcelona
Escola Superior Politècnica

Treball de Fi de Grau

Desarrollo de aplicación 3D para escritorio y web mediante Emscripten

Daniel Barca Casafont

Junio de 2017

Índice

Abstract

En este documento se recoge el proceso de desarrollo de una aplicación 3D multiplataforma para la empresa Interiorvista, la cual permite configurar y visualizar estancias con el lenguaje de programación C++ dados unos requerimientos inciales. Haciendo uso de Emscripten, el código ha sido adaptado a las limitaciones de las plataformas web que no encontramos en aplicaciones de escritorio tradicionales, tales como la existencia de un único hilo de ejecución o la ausencia de un sistema de ficheros. También se deberá adaptar el motor gráfico de la empresa para que funcione mediante WebGL o Vulkan en función de la plataforma donde se ejecute. Además, se discuten detalles sobre el diseño de software y el proceso de desarrollo de la aplicación en sí.

This document describes the process of developing a 3D multiplatform application for the company Interiorvista, that allows users to configure and visualize rooms with the C++ programming language given some initial requirements. Using Emscripten, the code has been adapted to the limitations of web platforms that are not found in traditional desktop applications, such as the existence of only one single execution thread or the absence of a file system. The company's graphics engine must also be adapted in order to work using WebGL or Vulkan depending on the platform where it's running. In addition, this document discusses details about the software design and the development process of the application itself.

Lista de Figuras

[1] Nicholas Haemel Graham Sellers Richard S. Wright. *OpenGL SuperBible*. Addison Wesley, 2015.

Quisiera agradecer a mi familia, amigos y compañeros el haberme dado empuje cuando a mí me ha faltado. Especialmente a Marc Fernández Vanaclocha, Tomás Banzas Illa y Hermann Plass Pórtulas por apoyarme y aconsejarme en todo momento.

1 Introducción

1.1 Motivación

Interiorvista es una empresa especializada en la generación de imágenes por computador (mucho más baratas, rápidas y de igual o mejor calidad que las que pueden obtenerse con un plató y un fotógrafo) y en el desarrollo de aplicaciones web (que requieren personal muy especializado y una gran inversión de tiempo).

Entre estas aplicaciones se encuentran los *Interiorvista Planner*, un conjunto de aplicaciones que tienen el objetivo de permitir a los usuarios generar habitaciones tridimensionales y poblarlas con los productos que los clientes ofrecen en su catálogo. Esto plantea una serie de retos a varios niveles.

En su estado actual, las aplicaciones desarrolladas tienen problemas que hacen cada vez más difícil el mantenimiento y la mejora de estas. Muchas de sus características han sido desarrolladas sin llevar a cabo ningún diseño previo, o incluso sin una especificación previa de los requerimientos de la aplicación, provocando que estos requerimientos surjan a lo largo del desarrollo.

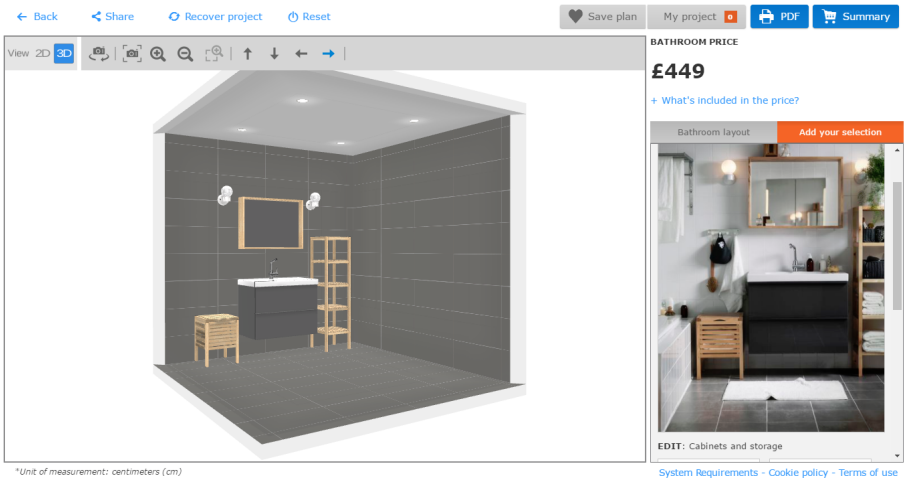


Fig. 1.1: Versión actual del planificador Bathroom Vista en vista 3D.

Las compañías de interiorismo suelen tener una serie de normas por las cuales ciertos elementos estructurales no pueden introducirse en ciertas combinaciones o en ciertas posiciones. Esto ha llevado a un código demasiado especializado en el que se introducen excepciones y condiciones arbitrarias sin mucho orden.

Se suelen requerir diversas aplicaciones muy similares para los distintos ambientes que ofrecen en su catálogo: habitaciones, comedores, baños, cocinas, etc. Aunque cada caso tiene sus particularidades, en general la mayoría de planificadores tienen suficientes características comunes como para poder tener un núcleo común, cosa que no está ocurriendo en estos momentos.

Generalmente casi siempre vamos a tener una habitación con ventanas, puertas, y una serie de elementos

interiores que podemos distribuir por esta. Por ello, con un diseño efectivo debe ser posible reducir la especialización de cada una de estas aplicaciones. En el futuro, una posibilidad con la que se ha soñado en Interiorvista es la de hacer un planificador completo de una planta, con todas sus habitaciones, algo que no resultaría sencillo de conseguir con los desarrollos de que disponemos actualmente.

Entre las características comunes de los planificadores encontramos que la mayoría disponen de un modo visualización en 2 dimensiones, pensado para configurar la estructura de una habitación, y otro en 3 dimensiones, pensado para visualizar el resultado y realizar retoques sobre este. En estos momentos estos modos se han programado como dos programas distintos con una parte 2D hecha con tecnologías web, y una 3D hecha con un motor gráfico exportado a WebGL. Es una duplicidad de esfuerzos que puede evitarse utilizando una cámara ortogonal en 3D y algunas modificaciones visuales.

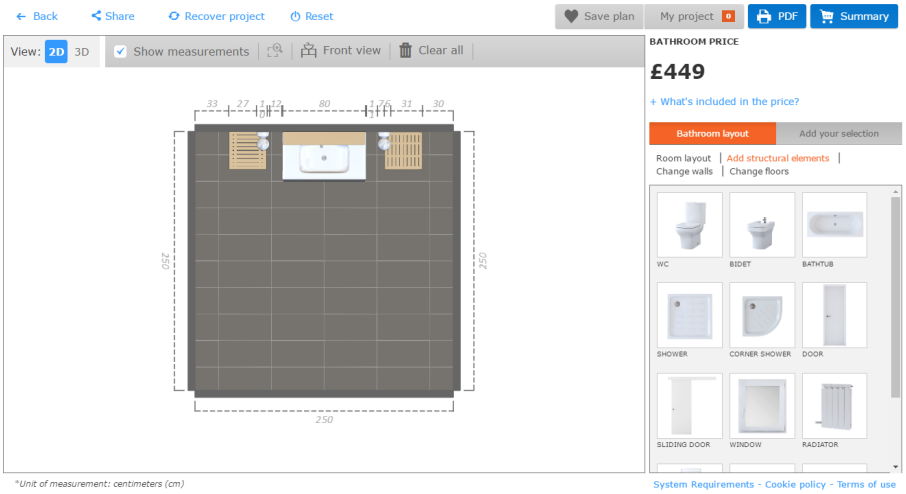


Fig. 1.2: Bathroom Vista, versión en dos dimensiones del baño en la figura 1.1.

A pesar de las similitudes siempre hay elementos que hacen único a cada Planner: algunos baños o cocinas tienen diferentes texturas combinables para las paredes, algunas habitaciones tienen un techo inclinado o algunos productos de son altamente configurables y requieren más atención. Diferentes planificadores pueden tener un enfoque distinto: desde completas herramientas que han de poder generar y visualizar todas las configuraciones posibles de una estancia hasta aplicaciones que buscan un enfoque más emocional que atraiga a los usuarios, lo cual implica sacrificar funcionalidad en favor de la estética. Incluso pueden llegar a existir dos aplicaciones para un mismo conjunto de productos, con el objetivo de cubrir ambos puntos de vista.

El hecho de estar utilizando motores gráficos propietarios nos ha provocado problemas en el pasado. Las necesidades de la empresa son muy específicas y la imposibilidad de controlar el funcionamiento del motor ha hecho que no podamos solucionar efectivamente muchos problemas, llevándonos incluso a tener que esperar a que los desarrolladores lancen actualizaciones que arreglen nuestros problemas, o a mantener versiones abandonadas porque las versiones nuevas no son compatibles con ciertos requerimientos.

La falta de control sobre el motor gráfico también ha hecho que no pudiéramos realizar ciertas mejoras de eficiencia, visualización, o reducir el peso de la aplicación (por ejemplo, las versiones actuales incluyen toda una API de audio que no es necesaria).

1.2 Objetivos

La aplicación debe ser lo bastante genérica como para aplicarse a diferentes casos, de ahí que digamos que se trata de un conjunto de aplicaciones; y al mismo tiempo también ha de ser flexible como para dar cabida a todas estas características.

Debe contar con un visualizador en 2 y 3 dimensiones. Según el modo de visualización la interacción y

las opciones son diferentes, pero el estado y la lógica de la aplicación debe mantenerse el máximo posible.

A largo plazo, la aplicación debe poder ejecutarse sobre distintas plataformas como web, escritorio, móviles o tabletas. Esto tiene muchas implicaciones a nivel de software e interacción: distintas plataformas cuentan con distintos drivers y APIs de ejecución y cada una tiene un funcionamiento y un modo de uso muy distinto. Dado que la aplicación va a estar completamente programada en C++, trasladar el código a otras plataformas (especialmente web) puede suponer un reto.

El desarrollo se realizará sobre un motor gráfico propio de la empresa, el cual está pensado para funcionar con la API gráfica Vulkan en escritorio, y debe ser adaptado para poder utilizarse con otras APIs en otras plataformas como Web o plataformas móviles. El hecho de disponer de un motor gráfico propio nos da un gran control sobre lo que ocurra dentro de este, en contraste con otras alternativas propietarias que no podemos controlar. En el pasado se han tenido muchos problemas haciendo funcionar las aplicaciones en distintas plataformas.

A nivel de diseño de software, esta es una oportunidad para repensar y reorganizar los problemas que ya conocemos. Aplicar correctamente diversas técnicas de diseño de software hará que no sólo sea más sencillo desarrollar la aplicación sino que sea más fácil de mantener y ampliar en el futuro. Algunas características son muy difíciles de implementar si no se ha seguido un cierto diseño desde el principio.

1.3 Estado del arte

Aunque existen diversos planificadores de estancias en el mercado, a día de hoy la mayoría tienen serias deficiencias y prácticamente ninguno está asociado a marcas importantes del modo en que lo está Interiorvista. Sin embargo, eso no impide que aprendamos de las alternativas existentes.

Entre los fallos más comunes se encuentran:

- La necesidad de descargar aplicaciones de escritorio, o un gran número de assets que no necesariamente van a utilizarse.
- El uso de tecnologías obsoletas, especialmente Adobe Flash (muy popular durante la última década pero en desuso hoy en día), o motores web que requieren la instalación de plugins o extensiones.
- Sólo modo en 2 dimensiones o 3 dimensiones, sin la posibilidad de cambiar.
- Mala calidad gráfica.
- Interacción y/o diseño pobre.

Por supuesto, tenemos como precedente los anteriores planificadores hechos en Interiorvista, que aunque están bien situados en el mercado sufren de algunos de los fallos ya mencionados. La alternativa más sólida para lo que queremos realizar es Planner5D[1], que cumple buena parte de los requerimientos que queremos cumplir; sin embargo a día de hoy también tiene defectos en estabilidad e interacción, como que los elementos interiores no se adhieren a las paredes (a excepción de elementos estructurales de las paredes, como puertas y ventanas), o que en 2D pueden estropearse las paredes de forma relativamente fácil.

[1] Planner5D. *Planner5D*. `https://planner5d.com/`.

2 Descripción del proyecto

Interiorvista Planner es una herramienta que permite crear de forma rápida y fácil el diseño de una sala a medida. Está pensada para ser genérica, de modo que después el proyecto se subdivide en otras aplicaciones.

La aplicación debe permitir configurar las dimensiones y forma de una sala para después introducir elementos propios de cada proyecto en esta, para finalmente obtener un listado de los productos introducidos, el precio de comprar dicha configuración, y un código que permite acceder al proyecto desde una tienda física para realizar la compra.

2.1 Clientes

IKEA[1] es una archiconocida multinacional especializada en la venta de muebles de bajo coste. Se gestó en Suecia en el año 1943 como una tienda de venta de productos varios para el día a día a un precio reducido, y cuenta hoy con 314 tiendas repartidas en 38 países, siendo el icono más reconocible en el mundo de los muebles.

Una de las claves del éxito de IKEA es su famoso catálogo, donde los potenciales clientes podían ver los muebles que se ofertan y sus posibles distribuciones. Durante los años 2000 IKEA ha puesto su catálogo a disposición de los clientes también a través de Internet, y les ha ofrecido nuevas herramientas con las que poder imaginar cómo van a quedar los productos que compren en su hogar.

ROCA[2] es el principal proveedor de productos para baños del mundo. Se gestó en Gavá en 1917 como una compañía de radiadores, pero su relación con el agua hizo que se interesara rápidamente en la fabricación de porcelana en 1936 y grifería en 1954.

Al igual que IKEA, ROCA ha encontrado en internet nuevas formas de llegar a sus clientes, creando catálogos online y herramientas de configuración y visualización.

Aunque estos son los dos principales clientes de Interiorvista, la naturaleza genérica del planificador hace que cualquier empresa especializada en el interiorismo sea un potencial cliente.

2.2 Objetivo del producto

Los productos de interiorismo destacan por ser altamente configurables y modulares, para adaptarse a los gustos y necesidades de cada comprador. Esto hace que sea complejo crear una aplicación que tenga en cuenta todas las peculiaridades de los productos. Con los *Interiorvista Planner* los compradores deben poder probar y visualizar las diferentes configuraciones de los productos y realizar la compra (a través del código) si así lo deciden.

Por lo tanto, el objetivo es conseguir que un máximo número de clientes generen un código y lo recu-

[1] IKEA. *Historia de IKEA – cómo empezó todo*. `http://www.ikea.com/ms/es_ES/about_ikea/the_ikea_way/history`.

[2] Roca. *Empresa global - 100 años de historia que nos han convertido en referente a nivel mundial*. `http://www.roca.es/nuestra-empresa/sobre-nosotros/una-empresa-global`.

peren desde una tienda (signo de que han acabado comprado el producto). Cuanto más satisfactorio sea el proceso de configuración, más probable es que dichos clientes lleguen hasta el final, es por esto que la usabilidad y los tiempos de carga son clave.

Otro objetivo indirecto es hacer que la aplicación sea lo suficientemente genérica como para poder adaptarse, con poco esfuerzo, a los diferentes sub-proyectos.

2.3 Usuarios

Hay dos posibles usuarios de la aplicación: los compradores, que pueden acceder a esta a través de la página web del cliente, desde los ordenadores disponibles en las tiendas o bien desde las aplicaciones móviles; y los empleados del cliente (también conocidos como coworkers), que se encuentran en las tiendas vendiendo productos y ayudando a los clientes.

El enfoque para cada cliente es algo distinto. En el caso de los compradores se busca algo más rápido y emocional, que le lleve lo más rápido posible a la compra del producto. Sin embargo, para los trabajadores esta aplicación es una completa y precisa herramienta de trabajo, que ha de ser capaz de poder reflejar cualquier posible configuración.

A pesar de esto la aplicación será razonablemente similar en ambos casos, a excepción de algunos "atajos" con los que los trabajadores podrán llegar más rápido a las secciones que les interesan.

3 Motor gráfico

Como se ha mencionado anteriormente, el motor gráfico con el que trabajaremos para crear esta aplicación es propio de la empresa: Manta. Manta está programado en C++ y pretende ser un motor multipropósito, aunque el hecho de estar programado en la misma empresa nos permite prestar especial atención a los usos específicos que le demos dentro de esta. En este apartado discutiremos algunas de las características más relevantes del motor para con la aplicación. No se pretende pues crear una documentación completa de este sino tan sólo una visión general, como referencia para el desarrollo de la aplicación.

3.1 Diseño y estructura del motor

Antes de empezar a trabajar es necesario conocer cómo comunicar la aplicación con Manta, el motor gráfico que será utilizado para el desarrollo, y tener al menos una buena idea del modo en que el motor tratará la información que le proporcionemos.

La aplicación final estará dividida en tres partes importantes: la aplicación en sí, el framework, y el núcleo o core (fig. 3.1).

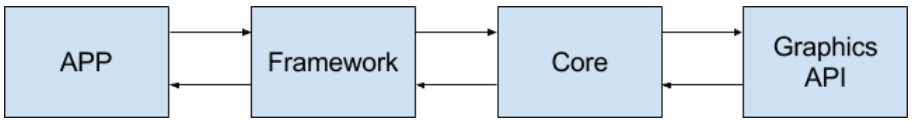


Fig. 3.1: Estructura de la aplicación.

En el momento de manejar la escena, la aplicación en ningún momento se comunicará directamente con el núcleo del motor, sino que lo hará a través del framework. El framework está diseñado para facilitar la usabilidad, mientras que el núcleo prioriza la eficiencia, de modo que utilizando este sistema no se renuncia a ninguna de las dos cosas.

El framework también se encarga de gestionar las colisiones y las normas de la aplicación (cómo qué interiores pueden ir en un sitio determinado). Otros elementos más sencillos del núcleo sí son accesibles desde la aplicación, como el gestor de memoria o la carga de materiales y texturas.

3.1.1 Ratio de fotogramas por segundo y bucle de ejecución

Aunque puede funcionar para la generación de imágenes estáticas, Manta es sobretodo un motor gráfico en tiempo real. Esto significa que su código se ejecuta múltiples veces para generar imágenes distintas y mostrarlas en pantalla, con lo cual se logra una ilusión de movimiento. De manera óptima suele considerarse como estándar los 60 fotogramas por segundo[1] (FPS o frames en adelante), aunque 24-30 FPS suele considerarse el mínimo aceptable. Debe tenerse en cuenta que aunque estos son los valores habituales, la

[1] GeForce. *Adaptive VSync Technology*. `http://www.geforce.com/hardware/technology/adaptive-vsync/technology`.

tolerancia varía según el tipo de aplicación: en nuestro caso aunque buscamos una respuesta fluida, puede llegar a ser aceptable una bajada de los FPS sin afectar gravemente la experiencia de usuario.

Como implicación está que para una buena experiencia en tiempo real todos los aspectos de la aplicación, incluyendo los cálculos de la propia aplicación como del propio motor y el renderizado a través de la GPU, deben calcularse como mínimo en unos 40 milisegundos segundos y óptimamente en 16 milisegundos (los mencionados 24 y 60 FPS). En contraste, un render de alta calidad con ray tracing (una técnica que simula la física detrás de la interacción entre la luz y las superficies de un espacio) puede tardar varios minutos u horas[2]. Es de esperar por tanto que el motor sacrifique gran parte de ese realismo para reducir el tiempo de ejecución.

Para hacer esto repetidamente de manera indefinida se encapsula todo el código del programa en un bucle de infinito que sólo puede detenerse manualmente y que contiene, en orden:

- El cálculo del tiempo de ejecución del frame: las variaciones de FPS que puedan producirse a lo largo de la ejecución pueden provocar un efecto de aceleración y deceleración que empeora notablemente la experiencia. Además no se puede predecir cual será el ratio de FPS al que se ejecutará la aplicación cuando no conocemos en qué máquina se ejecutará y cual es su potencia. Para evitar este tipo de indeterminación se calcula el tiempo que transcurre entre un frame y el siguiente, el cual puede utilizarse en los cálculos a la hora de actualizar para compensar.
- La actualización de la aplicación: se llama a una función desde la cual debemos actualizar la escena en función de los cálculos que se realicen. Cuando la aplicación alcance cierta complejidad, este puede llegar a ser el punto del bucle que absorba una mayor carga de trabajo, por lo que la eficiencia debe ser tenida en cuenta al actualizar para no ralentizar demasiado el programa.
- La actualización de la escena: Como se mencionará en el apartado 3.1.3, el motor dispone de una colección de elementos que le hemos ordenado mostrar en pantalla. En este punto se actualiza la escena para que refleje los cambios hechos en el proceso de actualización de la aplicación. Entre otras cosas, se calculan las transformaciones de las entidades (3.1.2) y se prepara toda la información que pueda necesitarse para renderizar.
- Renderizado: En el momento de renderizar la escena, junto con la orden de renderizado se envía toda la información necesaria a la GPU (incluyendo la cámara, las luces y los elementos de la escena). Aquí predomina el funcionamiento de la API gráfica, por lo que según la plataforma en que nos encontremos, este paso será realizado por una API diferente (véase el apartado 3.2).

Previamente al bucle se ejecutan las funciones de inicialización tanto del motor como de la aplicación, mientras que al detener la ejecución se libera la memoria reservada (normalmente durante esta primera inicialización).

[2]Phill Miller (Nvidia). *NVIDIA Brings AI to Ray Tracing to Speed Graphics Workloads*. `https://blogs.nvidia.com/blog/2017/05/10/ai-for-ray-tracing`.

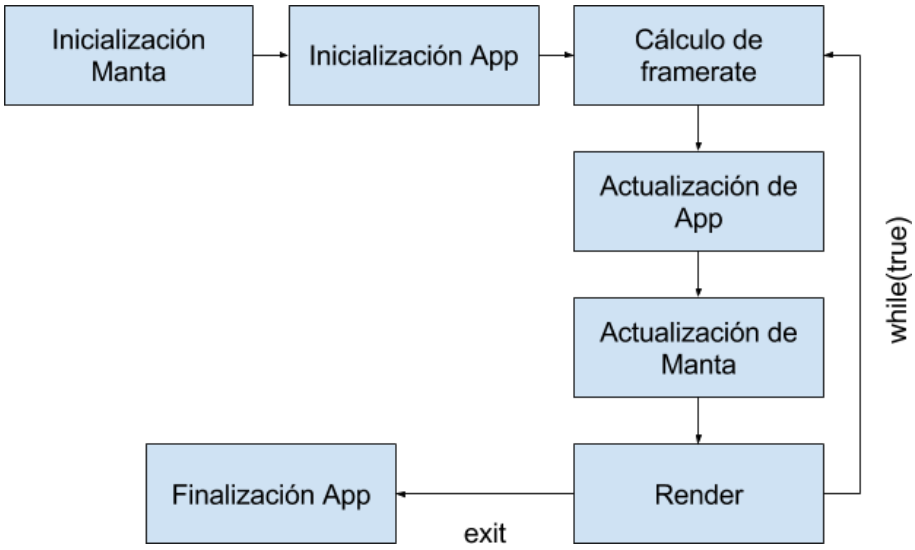


Fig. 3.2: Esquema del bucle de ejecución.

3.1.2 Patrón de entidad-componente

En una aplicación 3D, puede entenderse como entidad cualquier elemento que se encuentre en la escena. Por lo general todas tienen un elemento en común: tienen una transformación que indica su traslación, rotación y escalado. Sin embargo, cada entidad puede tener propósitos distintos: algunas son luces, otras cámaras, o objetos tridimensionales con malla y material, etc.

Dentro de cada categoría pueden haber varios tipos, o tal vez se necesita que una entidad cumpla unas propiedades determinadas como emitir audio o verse afectada por un motor de físicas; y también se podrían querer combinaciones de estas como que un elemento sea una luz y al mismo tiempo se balancee mediante un motor de físicas.

No es deseable que el código mezcle cosas tan dispares, tener las físicas y el audio en el mismo sitio resultaría en un código imposible de mantener a largo plazo; por lo que se precisa buscar un modo de modularizar estas características. Para ello existe el patrón entidad-componente[3], que nos permite crear componentes que aglomeran ciertas propiedades y comportamientos. Después podemos añadir cuantos componentes queramos a una entidad, permitiendo hacer que esta cumpla dichas propiedades sin aglomerar todo el código de estas.

Para implementar este patrón existen dos clases principales: “Entity” y “Component” (fig. 3.3); las cuales pueden extenderse como se desee (no sólo desde el framework sino también desde la propia aplicación) para crear distintos tipos de cada una. Una entidad por defecto contiene una lista de componentes, que en el fondo son instancias de los diferentes tipos de componentes que creemos. Component es una clase abstracta y sus clases derivadas deben implementar uno o varios métodos que puedan ser llamados desde la entidad.

[3]Robert Nystrom. *Game Programming Patterns*. Lightning Source Inc, 2014.

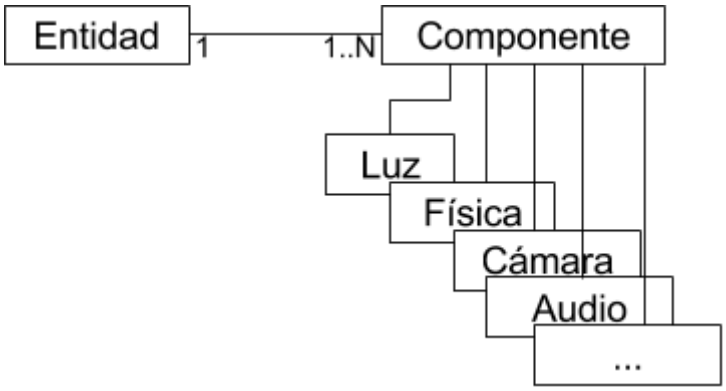


Fig. 3.3: Esquema simplificado del patrón entidad-componente.

De este modo, cada vez que la entidad necesite delegar una funcionalidad a sus componentes, iterará sobre ellos y llamará dichos métodos, que tendrán un comportamiento distinto según se requiera. En nuestro caso, los métodos que implementa Component son los siguientes:

- AddedToEntity: Llamado en el momento en que se incorpora el componente a la entidad.
- RemovedFromEntity: Llamado en el momento en que este se borra de la entidad (lo cual normalmente significa que estamos eliminando la entidad, pero no necesariamente).
- AddedComponent: Se llama a todos los componentes que tenga la entidad cada vez que se añade uno nuevo, e incluye una referencia por parámetro del componente añadido.
- RemovedComponent: Al igual que la anterior, cuando se elimina un componente también se hace saber al resto de componentes aún existentes.

Aunque no ha sido utilizado, es bastante típico en el patrón Component tener un método "update" que se llama en cada actualización de la aplicación (3.1.1) y hace que las propias entidades se encarguen de su propio comportamiento en tiempo real, a través de los componentes. En esta aplicación se ha decidido que sean funciones externas quienes controlen dicho comportamiento, y no las propias entidades.

Para acceder externamente a los componentes, en la clase Entity se hace uso de "templates" de C++, sobre los cuales no se profundizará. Los templates permiten implementar un método independientemente del tipo de las variables con las que trabaja, de modo que este tipo se especifica en el momento de llamar al método. Para pedirle a una entidad que un componente específico, se debe especificar el tipo del componente que se desea.

3.1.3 Escena, jerarquía y pasos previos al renderizado

Como se ha mencionado en el apartado 3.1.2, existen una serie de entidades distribuidas por el espacio con una traslación, rotación y escalado. Este espacio se conoce como la escena.

Al crear una entidad, esta automáticamente se registra a sí misma en la jerarquía de la escena. La jerarquía contiene por tanto un listado con punteros a cada una de las entidades que le permite acceder a sus componentes. El motor trabaja directamente con este listado de objetos y hará automáticamente todo lo que se requiera con ellos en función de sus componentes. Por lo tanto, el usuario no debe pedir en ningún momento que se rendericen los objetos: basta con crear una entidad que tenga un componente de malla (véase el apartado 3.1.4) para que el motor entienda que deberá renderizarlo.

Las entidades están organizadas en la jerarquía de tal modo que una entidad puede ser padre de otras entidades, formando una estructura de árbol. Las transformaciones de las entidades se acumulan desde la raíz de la jerarquía hacia abajo, es decir: si una entidad está desplazada o rotada, sus entidades "hijas" tendrán una traslación y rotación respecto a la primera. Esto permite crear diferentes estructuras dentro de

la escena y trabajar con estas sin tener que controlar la posición de todos los elementos que la componen. Por ejemplo: dada una entidad "coche" y otra entidad "asientos" asignar el primero como padre de los segundos hará que se mantengan en su sitio, pegados a la estructura del coche, dado que su transformación es relativa a este.

Una consecuencia de este sistema es que las transformaciones que hay en cada entidad no son realmente la transformación respecto al centro de la escena, que es la que necesitamos en el momento de renderizar. Por lo que se debe realizar un cálculo previo al renderizado que consiste en multiplicar de forma acumulativa las transformaciones en cada una de las ramas de la escena y asignarlas a cada uno de los elementos.

Aunque desde fuera la información se nos presente de en forma de árbol, internamente los datos están estructurados en arrays. El objetivo de esta estructura es mejorar la eficiencia del código (haciendo uso de conceptos como Data Oriented Design o SIMD, sobre los que no se profundizará en este documento pero pueden ser una buena lectura complementaria) sin sacrificar usabilidad.

3.1.4 Mallas, luces y cámara

Se trata de los ejemplos más importantes de componentes (véase el apartado 3.1.2) que se utilizan en el motor.

Mallas

Entendemos por malla (o mesh) un listado de vértices conectados por otro listado de índices. Los vértices contienen información de la posición, normal, tangente, bitangente y coordenadas UV (véase 3.1.5) mientras que los índices simplemente ordenan los vértices de 3 en 3 creando triángulos. Los componentes "MeshComponent" y "MeshDynamicComponent" recogen esta información para que el motor renderice el resultado posteriormente. Estos componentes también incluyen los materiales de las mallas.

La diferencia principal entre ellos es que "MeshComponent" lee los datos de un fichero al crearse y es inmutable, mientras que "MeshDynamicComponent" se crea asignando una cantidad máxima de vértices e índices y estos se asignan programáticamente, pudiendo modificarse en cualquier momento (será muy importante para la generación dinámica de paredes y ventanas en la sección 5.1). Otra diferencia importante es que "MeshComponent" puede contener diversas sub-mallas que comparten transformación pero pueden tener distintos materiales, mientras que "MeshDynamicComponent" solo puede contener una.

Luces

Las luces son el elemento más complejo de un motor gráfico. La calidad de la luz es el elemento que más influye en el realismo de la imagen generada, y es uno de los puntos que más coste computacional requiere. En las técnicas de renderizado de alta calidad, se trata de simular la física de la luz para conseguir un gran realismo, pero esto es impracticable en un motor en tiempo real como se ha dicho en el apartado 3.1.1.

En el momento de renderizar se utilizan luces para saber con qué intensidad debe renderizarse cada elemento de la escena, o los diferentes puntos de su superficie. Conociendo en qué dirección incide la luz y la normal de la superficie en el punto que queremos pintar, mediante el producto escalar de estos vectores podemos saber si la luz incide sobre este punto y con qué intensidad. Además las luces pueden tener color (normalmente será luz blanca pero no tiene por qué ser así) que se refleja mezclando el color de la luz con el de la superficie.

En la realidad las luces pueden tener formas muy variadas, pero en el motor se reducen a: puntos de luz, luces direccionales, focos, luces de área, luces esféricas y luces cilíndricas. El tipo de luz hará variar el modo en que incide sobre la escena. Las propiedades de esta se asignan en el momento de crear el componente y pueden cambiarse en cualquier momento.

Cámara

La cámara indica el punto de vista desde el que se realizará el renderizado de la escena. Existen dos tipos (fig. 3.4):

- Cámara ortogonal: se caracteriza por mostrar todos los elementos con la misma escala, sin importar la distancia en que estén. Es especialmente útil para comparar tamaños de distintos elementos, y la utilizaremos para vistas superior y frontales de la habitación, pensados como modos de edición de esta. Sus propiedades son el tamaño, en alto y ancho, de su campo de visión.
- Cámara en perspectiva: imita el modo en que vemos los objetos en la realidad, teniendo en cuenta la distancia en que se encuentran. Da resultados mucho más satisfactorios para visualizar la habitación, y transmite más información espacial. Sus propiedades son el ángulo de su campo de visión, más conocido como FOV (field of view), y los planos cercano y lejano de este, comúnmente conocidos como "near plane" y "far plane".

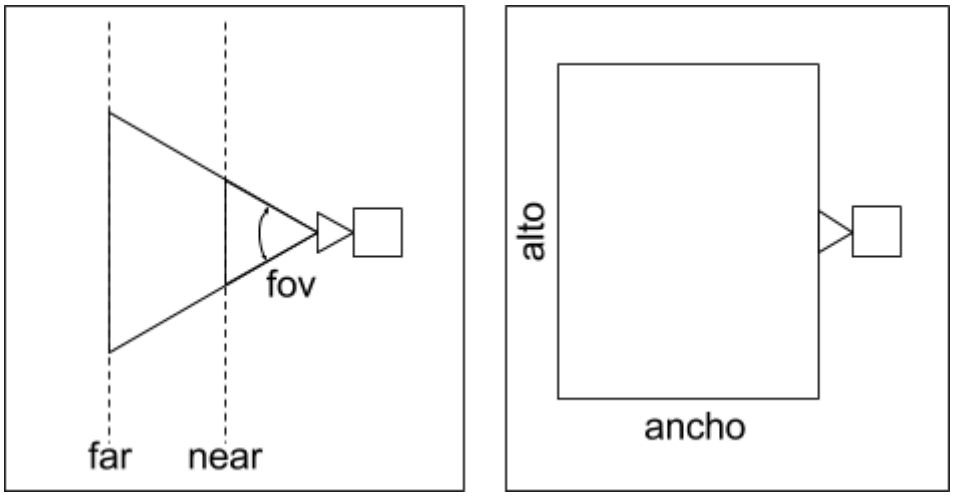


Fig. 3.4: Diferencias entre una cámara en perspectiva y ortogonal, respectivamente.

3.1.5 Materiales

El material de un objeto 3D describe el aspecto que ha de tener la superficie de este en el momento de renderizarlo. Para conseguirlo se asocian una serie de texturas del mismo tamaño y se asocia cada uno de los vértices con coordenadas de dos componentes dentro de estas texturas (conocidas como UV, véase 3.1.4). Como se explica en el apartado 3.3, el shader interpola los datos entre dos vértices a cada uno de los píxeles intermedio. De este modo sabiendo la UV que corresponde a cada uno de los vértices podemos saber la que corresponde a cada píxel del objeto 3D.

Las texturas que forman el material pueden ser un color único (que equivale a aplicar una textura de ese color o una imagen importada desde un fichero. El material está formado por los siguientes tipos de texturas o mapas:

- Albedo: En el terreno de la física, el albedo es la cantidad de luz que rebota sobre una superficie tras incidir sobre estaMaría Merino Julián Pérez Porto. *Definición de albedo*. `http://definicion.de/albedo/`. Puede entenderse como la textura que expresa el color que ha de tener la superficie en cada punto.

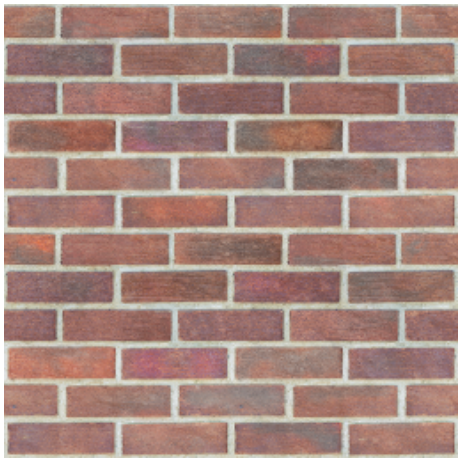


Fig. 3.5: Ejemplo de AlbedoFree PBR. *Free PBR*. `http://freepbr.com`.

- Mapa de normales: Habitualmente, la normal de la superficie se expresa a partir de la propia geometría de esta, pero a veces se necesitan ciertos detalles que requerirían aumentar mucho la resolución de la malla, aumentando con ello también el peso del modelo 3D y el coste computacional de procesar y renderizar dicha información. Para evitar esto se puede utilizar una textura de normales: en cada píxel, a la normal de la superficie interpolada a partir de los vértices próximos, le sumamos la normal correspondiente extraída del mapa de normales. Esto permite crear un efecto de profundidad o rugosidad sin modificar la geometría del objeto.

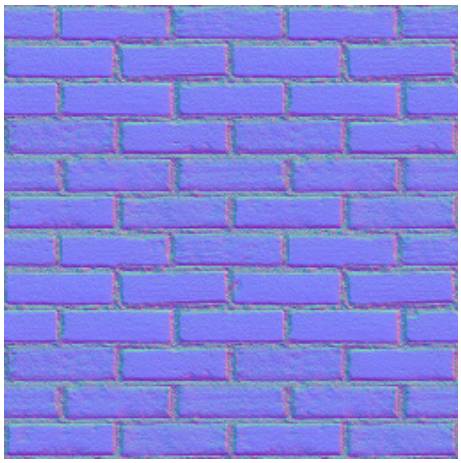


Fig. 3.6: Ejemplo de Mapa de NormalesFree PBR. *Free PBR*. `http://freepbr.com`.

- Aspereza (roughness): Se produce cuando la superficie tiene micro-imperfecciones que afectan a los reflejos, haciendo que se difuminen. Cuanto mayor sea la aspereza, más se dispersará la luz al rebotar contra la superficie, y más se difuminará el reflejo; mientras que en una superficie sin imperfecciones la luz se reflejará uniformemente.

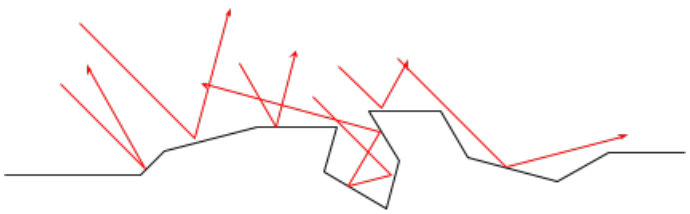


Fig. 3.7: El roughness simula las micro-imperfecciones de la superficie.

- Fresnel[4]: El reflejo Fresnel (nombre que hace referencia a su descubridor, Augustin-Jean Fresnel) es la propiedad según la cual los materiales reflejan más cuanto más grande es el ángulo entre la normal de la superfície y la dirección de la luz.
- Metalizado[5]: Los metales tienden a reflejar la luz, razón por la que se define un valor de metalizado para especificar la cantidad de luz que refleja una superficie. Con el metalizado también se puede definir un tinte para la luz reflejada (por ejemplo, reflejos amarillentos en la superficie del oro).
- Oclusión ambiental[6]: Se trata de un efecto del que a menudo se abusa, dado que no existe en la realidad. El mapa de oclusión permite oscurecer ciertas zonas de la geometría, normalmente los puntos que forman ángulos más cerrados. Resulta efectivo para imitar ciertas sombras que se producen en estos puntos, pero en exceso puede resultar artificial.

Fig. 3.8: Ejemplo de mapa de oclusión ambiental.

3.1.6 Gestión de memoria

Una de las operaciones que más tiempo demandan en C++ es solicitar nueva memoria al sistema operativo. Debido a eso, abusar de las asignaciones de memoria, por ejemplo haciéndolas dentro de bucles (como el bucle de Manta), puede hacer que el programa se ralentice considerablemente.

Para minimizar la cantidad de asignaciones que realizamos, manta tiene un único "bloque de memoria" (más conocido como Memory Pool). El Memory Pool asigna una gran cantidad de memoria al inicializar el programa y nos da acceso a esta cada vez que lo necesitemos. Hacer esto es más barato computacionalmente, de modo que no necesitamos preocuparnos de las asignaciones que necesitemos hacer. También nos proporciona un tiempo constante de reserva y liberación de memoria.

Dada la cantidad de memoria que se necesita para una variable, el Memory Pool automáticamente busca una región del bloque no esté siendo utilizada en ese momento y devuelve un puntero a esa región, marcándola como "usada" desde ese momento. Cuando la variable deje de ser necesaria, podemos liberarla para futuras operaciones. Eso provoca que a lo largo de la ejecución puedan quedar pequeños fragmentos del bloque de memoria sin utilizar, que no siempre serán útiles para nuevos usos; algunas implementaciones de Memory Pool pueden reorganizar la memoria utilizada para evitar que queden estos fragmentos, pero hacerlo tiene un mayor coste computacional e impide que el programador trabaje directamente con punteros al bloque de memoria (dado que al reorganizar dichos punteros dejarían de ser válidos), obligando a usar sistemas más complicados de punteros.

Sin embargo, existen dos desventajas importantes:

[4] Jeff Russell (Marmoset). *Basic theory of physically-based rendering*. `https://www.marmoset.co/posts/basic-theory-of-physically-based-rendering`.

[5] Ibid.

[6] David Lenaert. *Screen-Space Ambient Occlusion: Battling your Contrast Bias*. `http://www.derschmale.com/2013/12/12/screen-space-ambient-occlusion-battling-your-contrast-bias`.

Por un lado necesitamos saber desde el principio cuánta memoria podemos llegar a necesitar dado que esta solo se va a asignar una vez; en caso de superar ese límite de memoria tendremos un error de segmentación que detendrá el programa.

Por otro lado, también necesitamos tener mucho más cuidado con la memoria que solicitemos al Memory Pool: si nos equivocamos al manejar la memoria asignada, C++ suele responder con un error de segmentación; pero si hacemos lo mismo con la memoria del Memory Pool, ese error no se producirá y estaremos modificando datos que corresponden a otras variables. Al fin y al cabo, el puntero que recibimos apunta a una posición desconocida del Memory Pool, y nada nos impide desplazarnos por este sin tener en cuenta el tamaño de la memoria que nos ha asignado.

En la figura 3.9 se puede ver un ejemplo de como podría quedar la memoria en mitad de una ejecución mientras se solicita un bloque del tamaño de un float. Cada cuadrado representa un byte de información.

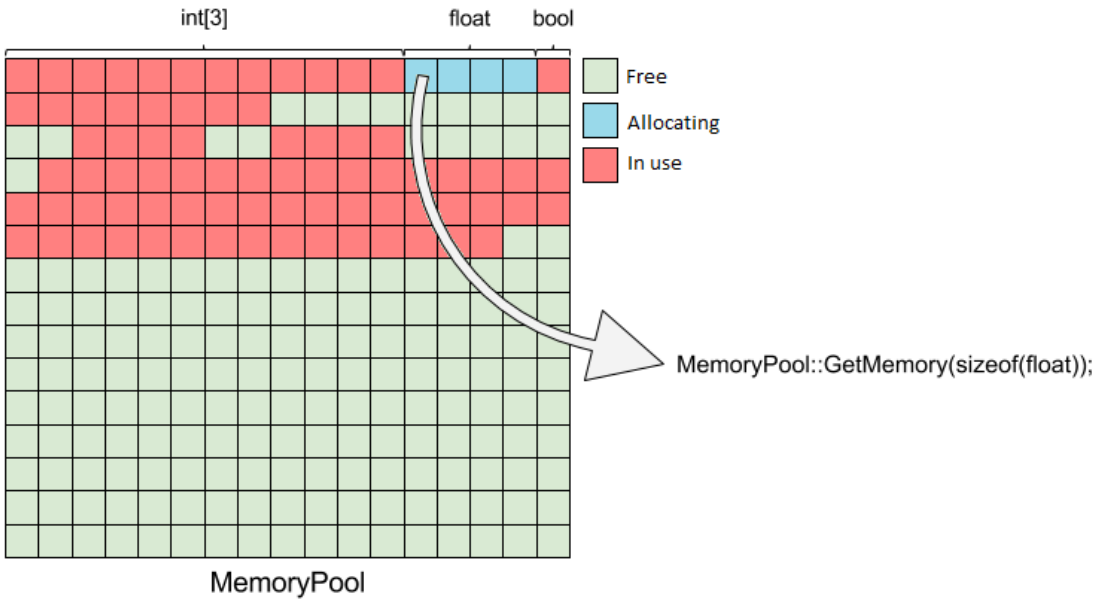


Fig. 3.9: Ejemplo de una posible distribución del Memory Pool durante una ejecución.

3.1.7 Gestión de identificadores

Entender como funcionan los identificadores en el motor Manta será un detalle importante en algunos puntos de este documento. Cada vez que se ordena a la API gráfica cargar un asset, esta devuelve un identificador que permite indicar posteriormente lo que se quiere hacer con los datos cargados en memoria. Sin embargo, este identificador no es el que se obtiene desde fuera del motor.

Manta crea una capa de abstracción por encima de los identificadores de la API gráfica, para evitar que sean manipulados desde fuera del motor. Desde Manta se espera tener constancia del estado de todos los assets por lo dar el identificador que proporciona la API gráfica permitiría al programador realizar ciertos "hacks" que harían que Manta perdiera el rastro de los assets.

Por otro lado, generar los propios identificadores de Manta permite mantener la consistencia entre diferentes APIs gráficas y otros elementos del motor. No sólo se busca impedir al programador manipular la API gráfica sino que también se espera que no sea necesario en ningún caso y el mantener esta consistencia hace más sencillo trabajar con el motor.

3.2 APIs gráficas

Según la plataforma en que se esté ejecutando la aplicación, se hará uso de una API gráfica u otra. Por defecto Manta ha sido desarrollado para funcionar en escritorio haciendo uso de Vulkan, pero esta API no está disponible en web, por lo que tendremos que utilizar OpenGL ES 2 y WebGL en este caso. Con este capítulo se pretende mostrar una vista general y con poco detalle de las APIs gráficas consideradas para hacer funcionar el motor en cada plataforma.

3.2.1 Vulkan

Entre las diferentes APIs gráficas que existen, la tendencia actual es la de dar cada vez más control al desarrollador sobre lo que ocurre entre la aplicación y la gráfica, dando acceso de bajo nivel al hardware. Vulkan es la respuesta de software libre a esta tendencia, y una de las APIs que más tracción está recogiendo últimamente.

Al ofrecer control de bajo nivel, con Vulkan pueden realizarse muchas optimizaciones que en una API de alto nivel no sería posible realizar[7]. Vulkan facilita el uso de múltiples dispositivos de hardware con diversos propósitos (la GPU puede utilizarse también para realizar cálculos en paralelo, sin estar necesariamente renderizando) y el uso de multithreading.

Para ello Vulkan puede detectar y listar los dispositivos de hardware que se encuentran disponibles en la máquina, así como sus capacidades y especificaciones. Por cada dispositivo se crea una cola de comandos, que describen las acciones a realizar por el hardware.

Los comandos pueden tener distintos estados que permiten controlar el flujo de la aplicación, especialmente en aplicaciones multinúcleo. Dicho estado indica si el comando se ha ejecutado, está a la espera de ejecutarse o está disponible para ser ejecutado de nuevo. Por supuesto, la pipeline gráfica se ejecuta a través de dichos comandos.

En cuanto a la gestión de memoria, Vulkan asigna dos tipos de memoria al dispositivo: una mayor cantidad para los elementos con los que ha de trabajar constantemente, que cambiará lo mínimo posible, y que es inaccesible desde fuera del dispositivo; y otra menor conocida como "staging" que permite hacer una copia de fragmentos de la memoria principal y modificarlos desde el exterior, para subirlos de nuevo a la principal posteriormente. Este paso se realiza porque es muy ineficiente que un dispositivo acceda a la memoria de la CPU y vice-versa, tanto la GPU como la CPU deben trabajar con su propia memoria y reducir al mínimo la transmisión de datos.

Al contrario que otras APIs, en Vulkan es posible utilizar multithreading. En caso de utilizarse, esto implica la necesidad de controlar la sincronía de cuanto ocurre en la aplicación, pero resulta una gran ventaja si se está dispuesto a realizar el esfuerzo, y supone un mucho mejor aprovechamiento de la CPU teniendo en cuenta que cada vez tienen más núcleos de procesador.

Como puede verse, en Vulkan la aplicación es responsable de la mayor parte de gestión en cuanto a qué debe hacer el dispositivo en cada momento. Es algo que contrasta, como se ha dicho, con la tendencia que ha existido hasta el momento de abstraer los elementos descritos en este apartado.

Aunque a priori el tener más control resulta beneficioso, se debe tener especial cuidado pues la dificultad de hacer funcionar Vulkan es muy superior que con otras APIs (como OpenGL en el apartado 3.2.2), llegando al extremo en que una mala implementación en Vulkan puede dar peores resultados que su equivalente en alternativas que habrían sido mucho más simples de implementar.

3.2.2 OpenGL ES 2 y WebGL

En estos momentos en las tecnologías web, WebGL es la única API 3D estándar y por lo tanto que vale la pena considerar. Por eso Vulkan sólo se va a utilizar en escritorio. En el capítulo 6.1.2 se explicará como

[7] Khronos Group. *Vulkan® 1.0.50 - A Specification*. `https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html`.

hacer que una aplicación programada con Vulkan pueda trabajar con WebGL al cambiar a la plataforma web.

OpenGL nació en 1992 como alternativa a las APIs gráficas propietarias del momento. La especificación y estandarización hicieron que se popularizara hasta ser la API gráfica más popular[8]. OpenGL ES es una versión de OpenGL para dispositivos embebidos[9].

WebGL apareció de la necesidad de tener un estándar de aceleración gráfica dentro de las tecnologías web. Hasta el momento las alternativas pasaban por el uso de plugins como Adobe Flash[10]. Está basado en OpenGL ES 2 y su especificación es prácticamente idéntica, de ahí que resulte relativamente sencillo convertir llamadas a OpenGL ES en llamadas a WebGL.

Al contrario de lo que ocurría con Vulkan en el apartado 3.2.1, OpenGL está pensado para proporcionar la mayor sencillez de uso posible, en detrimento del control que posee el desarrollador sobre lo que ocurre a bajo nivel. En OpenGL no es necesario especificar qué dispositivo utilizar (de hecho es prácticamente imposible escoger el dispositivo, lo cual puede ser una desventaja), gestionar la memoria, o manejar los comandos ejecutados por el dispositivo.

Aunque técnicamente es posible hacer programas con múltiples hilos de ejecución con OpenGL, hacerlo no supone ninguna mejora en términos de eficiencia y supone un esfuerzo fútil[11]. En cualquier caso, la imposibilidad de usar hilos de ejecución en Javascript hace que no podamos aprovechar multithreading de ningún modo con WebGL (véase el apartado 6.1.4 donde se explican, entre otras cosas, algunas características de Javascript).

El resultado es una mayor rapidez de desarrollo y un código mucho más ligero y sencillo de estructurar en comparación con Vulkan, en consecuencia más fácil de mantener. A pesar de sus limitaciones, por lo general es suficiente para la mayoría de aplicaciones 3D. Para aplicaciones en las cuales el rendimiento sea crítico, sus limitaciones pueden llegar a ser un problema, o al menos una desventaja importante.

3.2.3 Otras plataformas

Aunque en este documento sólo se profundiza en el desarrollo para escritorio y web, en el futuro se espera poder exportar a otras plataformas como sistemas operativos móviles. En este área hay algunas diferencias entre los diferentes proveedores de dispositivos móviles.

A día de hoy, dispositivos con iOS de Apple son compatibles con las versiones de OpenGL "OpenGL ES 1.1", "OpenGL ES 2.0" y "OpenGL ES 3.0"[12] mientras que en Android de Google se acepta "OpenGL ES 1.0", "OpenGL ES 2.0", "OpenGL ES 3.0" y "OpenGL ES 3.1"[13]. Cada versión de OpenGL tiene ciertas características que mejoran a la anterior aunque hacer uso de estas suele dar problemas de incompatibilidad con las anteriores (las versiones previas de cada sistema operativo puede tener menos versiones de OpenGL compatibles).

Android dispone de una versión de Vulkan pensada especialmente para dispositivos móviles[14], por lo que la exportación de código a Android debería ser relativamente directa. iOS en cambio dispone de su propia API para gráficos de bajo nivel: Metal. Metal comparte gran parte de la filosofía de Vulkan, pero realizar la exportación podría implicar bastante más esfuerzo. No hay expectativas de que Apple incluya soporte a Vulkan dentro de iOS a corto plazo.

Microsoft también cuenta con su propia API gráfica, DirectX, aunque los dispositivos móviles con Windows representan hoy en día una parte marginal del mercado, y es poco probable que valga la pena el esfuerzo de portar el código. Al igual que Vulkan y Metal, DirectX tiene un planteamiento de bajo

[8] Khronos Group. *History Of Opengl*. `https://www.khronos.org/opengl/wiki/History_of_OpenGL`.

[9] Khronos Group. *The Standard for Embedded Accelerated 3D Graphics*. `https://www.khronos.org/opengles/`.

[10] Adobe. *Stage3D*. `http://www.adobe.com/devnet/flashplayer/stage3d.html`.

[11] Khronos Group. *OpenGL and multithreading*. `https://www.khronos.org/opengl/wiki/OpenGL_and_multithreading`.

[12] Apple. *Guides and sample code*. `https://developer.apple.com/library/content/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/OpenGLESontheiPhone/OpenGLESontheiPhone.html`.

[13] Google. *OpenGL ES*. `https://developer.android.com/guide/topics/graphics/opengl.html`.

[14] Google. *Vulkan Graphics API*. `https://developer.android.com/ndk/guides/graphics/index.html`.

nivel, aunque su trayectoria es mucho más larga que los dos primeros (su primera versión data de 1995), y sus versiones anteriores no contaban con esta filosofía. DirectX tiene una mayor compatibilidad con dispositivos de escritorio que Vulkan, por lo que podría llegar a ser necesario portar el código a DirectX si en algún momento se requiere ejecutar en un dispositivo Windows que no acepte Vulkan (algunos clientes usan dispositivos muy específicos).

3.3 Shaders

Los shaders son programas diseñados para ejecutarse en la tarjeta gráfica[15]. A pesar de estar muy limitados, tienen la ventaja de aprovechar muy bien la capacidad de procesado paralelo de la GPU. Hoy en día los shaders son el núcleo alrededor del cual gira el renderizado de una aplicación 3D en tiempo real, y ocupan la mayor parte del tiempo de computación.

Aunque no se profundizará en lenguajes de shading, es importante saber que OpenGL ES utiliza el lenguaje de shading GLSL (OpenGL Shading Language)[16]. GLSL es un lenguaje basado en C que se compila en tiempo de ejecución mediante la API de OpenGL, a partir del propio código en texto plano. Vulkan en cambio utiliza el lenguaje intermedio SPIR-V (Standard Portable Intermediate Representation)[17]. SPIR-V es el resultado de compilar GLSL mediante glslang[18][19], y se provee en tiempo de ejecución a la API de Vulkan. Al haber sido procesado previamente SPIR-V carga sensiblemente más rápido en tiempo de ejecución. También es posible utilizar SPIR-V en OpenGL mediante una extensión (refiriéndonos a la API de escritorio, no su versión para sistemas embebidos)[20].

El hecho de que ambos partan del mismo lenguaje de shading será significativo a la hora de adaptar el código a WebGL.

En cada iteración, se ejecuta por cada elemento en escena la pipeline de la API gráfica. La pipeline recibe como parámetro una serie de primitivas (puntos, líneas o polígonos) y ejecuta los shaders pasándoles esa información. En OpenGL, la pipeline puede simplificarse como se ve en la figura 3.10.

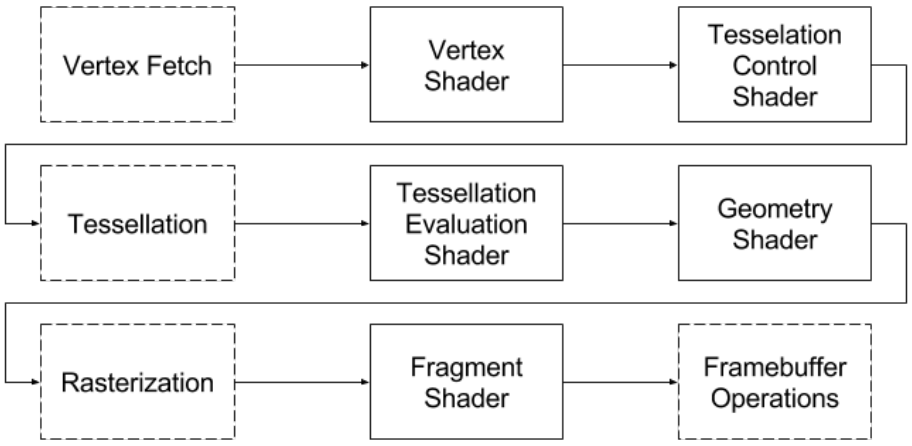


Fig. 3.10: Esquema de la pipeline gráfica de OpenGL, adaptada directamente de OpenGL Superbible[21].

De esta lista, los shaders marcados con una lista discontinua no son programables, sino que tienen

[15] Nicholas Haemel Graham Sellers Richard S. Wright. *OpenGL SuperBible*. Addison Wesley, 2015.

[16] Khronos Group. *OpenGL Shading Language*. https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language.

[17] Khronos Group. *The first open standard intermediate language for parallel compute and graphics*. https://www.khronos.org/spir.

[18] Khronos Group. *Isn't glslang an official GLSL to SPIR-V compiler?* https://www.lunarg.com/faqs/glslang-glsl-spir-v-compiler/.

[19] Neil Henning. *An Introduction to SPIR-V*. http://cdn2.imgtec.com/idc-docs/gdc16/AnIntroductionToSPIR-V.pdf.

[20] Khronos Group. *ARB_gl_spirv*. https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_gl_spirv.txt.

un comportamiento predefinido. A parte de esos, los tipos que se han utilizado para el desarrollo de la aplicación son el Vertex Shader y el Fragment Shader, dado que son compatibles con WebGL.

3.3.1 Vertex Shader

El Vertex Shader[22] se ejecuta independientemente por cada uno de los vértices de la malla a renderizar. Desde este shader se pueden modificar los atributos de cada uno de los vértices, recibidos desde la etapa de "Vertex Fetch", como su posición, color, coordenadas de textura, o cualquier otra propiedad que arbitrariamente le hayamos atribuido. También pueden añadirse nuevos atributos si se desea. Una vez modificados, estos atributos se pasan como outputs del shader para ser utilizados en otros estadios de la pipeline, como el Fragment Shader.

A la hora de programar el Vertex Shader es muy importante tener en cuenta que sus datos de salida van a ser interpolados en el Fragment Shader, y también que desde un Vertex Shader no se puede acceder a los atributos de un vértice distinto al que se está procesando.

3.3.2 Fragment Shader

El Fragment Shader[23] se ejecuta por cada píxel en pantalla que ocupe cada polígono. Sus atributos de entrada son los que se han definido como salida en el Vertex Shader pero están interpolados, es decir, entre los diferentes vértices de un polígono, los píxeles de su interior tienen los atributos intermedios de cada uno de los vértices, según su proximidad a estos. También es posible enviar datos que sean únicos para todos los píxeles, como la posición de la cámara o de las luces.

Desde el Fragment Shader podemos definir el color de un píxel específico, normalmente con la ayuda de los atributos interpolados. Se utiliza especialmente para definir la cantidad y color de luz que recibe cada fragmento del polígono, en función de factores como el ángulo entre la normal de la superficie a pintar y la dirección de la luz o el ángulo respecto a la cámara.

[22] Nicholas Haemel Graham Sellers Richard S. Wright. *OpenGL SuperBible*. Addison Wesley, 2015.
[23] Nicholas Haemel Graham Sellers Richard S. Wright. *OpenGL SuperBible*. Addison Wesley, 2015.

4 Diseño

Antes de empezar el proceso de desarrollo se han tomado una serie de decisiones de diseño. Realizar este paso ayuda a tener una mejor idea de lo que se quiere hacer, cuales serán las necesidades y qué problemas pueden surgir. En este capítulo se describirá el diseño de la aplicación incluyendo los cambios que se han realizado durante el desarrollo.

4.1 Núcleo, incialización y gestión de inputs

El elemento más importante de la aplicación, alrededor del cual girará el resto, es la clase "World". Puede entenderse esta clase como la "puerta de entrada" del flujo del motor hacia la aplicación. Es la primera clase de la aplicación en llamarse en la incialización, actualización y finalización del programa (véase el apartado 3.1.1 sobre el bucle de ejecución).

Desde "World" se incializa la cámara, y las luces. Un fichero externo `.json` describe el estado inicial de la aplicación y a partir de esta configuración se inicializa el progrma. También es la primera clase de la aplicación en recibir la orden de actualización del motor y los inputs del usario y debe transmitirsela a los diferentes elementos que gestione.

También se encargará de actualizar en cada iteración los diferentes componentes de la aplicación, y gestionar los estados del sistema de estados (explicado en el apartado 4.3).

4.2 Gestión y generación de entidades

Como se ha explicado en la sección 3.1, la aplicación se comunica con el motor a través de la escena y las entidades. Un primer punto importante es mantener el control de dichas entidades para que no acaben esparcidas por el código.

4.2.1 Managers y generadores

Normalmente es suficiente con organizar las entidades en estructuras de datos dentro de la propia clase World (4.1). Sin embargo, algunos casos específicos como las paredes y huecos (cuya generación se explica en el apartado 5.1 o los interiores), son lo suficientemente complejos y numerosos como para tener su propio gestor.

Dentro cada gestor hay una o varias listas de elementos que se organizan según sus índices en la lista, y que se inicializan con un tamaño predefinido. Se ha decidido hacer de este modo para evitar que la lista cambie de tamaño constantemente, provocando reinicializaciones de memoria. Una consecuencia es que debe hacerse una estimación del número de elementos que habrá y controlar mediante aserciones que no se supere ese número.

Como las paredes se definen en forma de conjuntos de paredes, para referirse a ellas se utilizan dos números enteros: la posición en la que empieza la primera pared y el número de paredes en el conjunto.

Si cambia el tamaño de un conjunto de paredes al actualizarlo se reorganiza la lista de paredes para que no haya fragmentación, y se controla en todo momento el número de paredes en uso, puesto que el tamaño de las listas no varía nunca y leer datos no utilizados provocaría errores.

En el caso de elementos que sean atómicos, como las ventanas o los interiores, es suficiente con controlarlos con su índice.

Para evitar que desde fuera del gestor se manipulen los índices, y para tener un identificador más sencillo de manejar, los gestores crean sus propios identificadores en forma de número entero y los relacionan con los elementos de los arrays mediante mapas. Esto permite por ejemplo, que si la posición de una pared cambia (porque se ha borrado una pared anterior) su identificador externo, el que tiene el usuario, siga significando lo mismo. Si se utilizaran índices cambiar el orden de los elementos resultaría en índices incorrectos, de este modo se evita que externamente haya que preocuparse por ello. El sistema es muy similar al que se describe en el apartado 3.1.7 sobre la gestión de identificadores del motor Manta.

Aunque no es estrictamente necesario, los gestores de entidades están pensados para utilizarse a través del gestor de comandos 4.4.

4.2.2 Generación de paredes

El primer paso ha sido crear una definición de los datos que se recibirán para describir cómo ha de ser la pared. Se trata de una lista de puntos en dos dimensiones y un valor booleano que indica si esta lista debe cerrarse conectando el último punto con el primero. Esto último es importante porque, como se puede ver en la figura 4.1, la geometría de una esquina "suelta" es diferente a la de una esquina que conecta dos paredes entre sí.

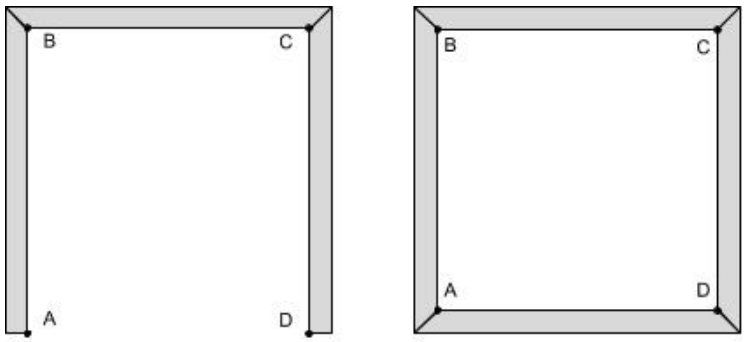


Fig. 4.1: Paredes en vista vertical.

Estos datos nos permiten no sólo crear habitaciones sino también paredes únicas o incluso otros tipos de estructuras, normalmente interiores, similares a una pared. La intención es que en el futuro el programa pueda utilizarse en otras herramientas para hacer diseños más complicados como el plano de una planta completa de un edificio.

Teniendo en cuenta la estructura de una malla, explicada en el apartado 3.1.4, en la figura 4.2 se puede ver la conversión de los datos que se espera conseguir.

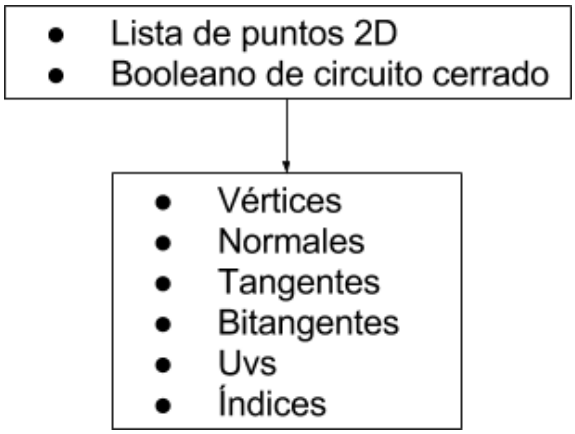


Fig. 4.2: Input y output del generador de paredes.

El output está formado por listas de valores planos. Por ejemplo, la lista de vértices está formada por los valores de posición "x,y,z" de cada uno sucesivamente.

4.2.3 Nomenclatura de los vértices

Para referir a cada uno de los vértices se ha definido la nomenclatura "A, B, A2, B2" como puede verse en la figura 4.3, además de los correspondientes "AH, BH, A2H, B2H" en la parte alta de la pared. Posteriormente, los índices de la pared se extraen de los que habría normalmente en un cubo, aprovechando que sus vértices se conectan del mismo modo.

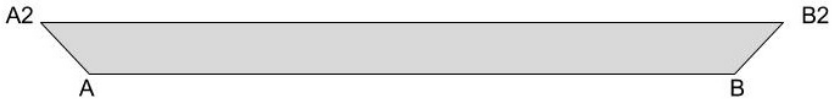


Fig. 4.3: Nomenclatura básica de los vértices.

4.2.4 Generación de huecos

El siguiente paso es implementar la posibilidad de añadir puertas y ventanas a la estancia. Insertar el modelo correspondiente en cada caso no será un problema, pero para que el efecto sea convincente es necesario poder ver a través de estos. Eso implica que se debe que modificar la geometría de las paredes para que incluya huecos donde tengan que ir dichas ventanas y puertas.

Del mismo modo que con las paredes inicialmente, se define un input: por cada hueco existe un punto 3D (que como se verá a continuación, no necesariamente debe colisionar con una pared), una altura y un ancho. El input/output del algoritmo completo para generar paredes quedaría como se puede ver en la figura 4.4.

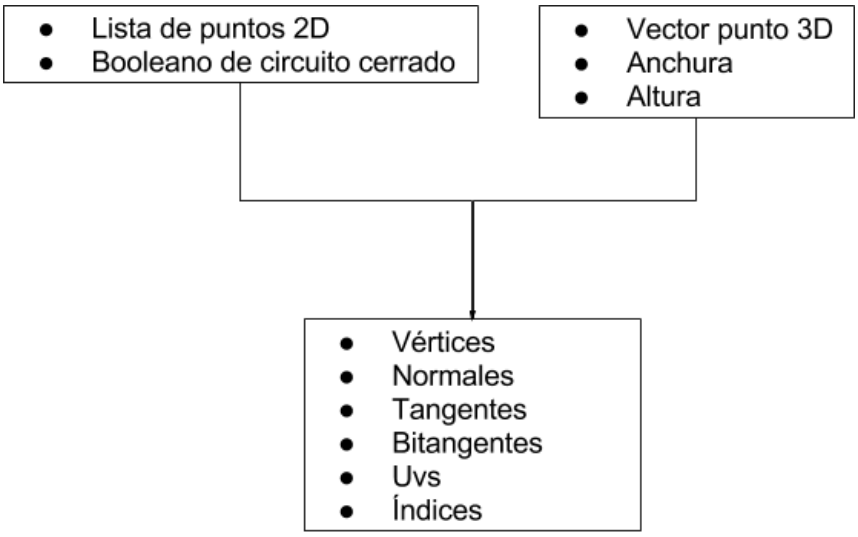


Fig. 4.4: Nuevo input y output de GenerateWalls, incluyendo ventanas.

Para empezar a introducir huecos en las paredes, estas se deben adaptar previamente. Es conveniente que la generación de ventanas se limite a trabajar sobre un solo plano, de modo que se eliminarán los planos anterior y posterior de la pared para añadirlos después con la nueva geometría. Aunque intuitivamente pueda parecer que esto supone simplificar la geometría respecto a lo que se ha hecho hasta ahora, en realidad se complica sensiblemente. Los planos anterior y posterior de la pared van a ser siempre idénticos, pero el plano posterior es algo más alargado debido a la geometría de las esquinas que se puede apreciar en la figura 4.3.

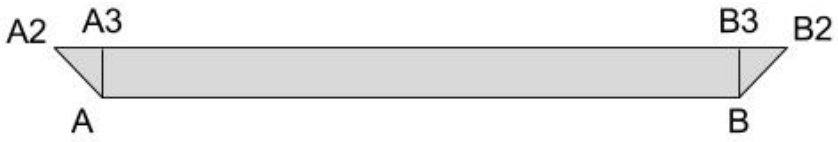


Fig. 4.5: Nomenclatura final de los vértices.

Se trata de una segunda iteración de lo visto en el apartado 4.2.3. Con la nueva geometría las paredes se convierten en dos prismas triangulares unidos por dos planos superior e inferior de la pared. Con esto se consigue que los planos anterior y posterior que ahora le faltan a la pared sean totalmente idénticos aunque con las normales invertidas.

Todas las ventanas o puertas han de ser necesariamente rectangulares. Esta es una precondición que se ha impuesto desde desarrollo para simplificar el cálculo de los agujeros, dado que es más sencillo incorporar geometrías más complejas incluyendo paredes falsas dentro del modelo de la ventana o puerta. Por ejemplo, si en algún momento se deseara incorporar una ventana redonda, sería más sencillo incluir 4 esquinas de pared a la ventana permitiendo que el hueco sea rectangular igualmente.

Para generar los huecos se separará la pared en un conjunto de planos, dejando vacío el espacio que ocupa la ventana. Por lo tanto existirá una función atómica cuya entrada y salida sean una lista de planos. El la figura 4.6 puede verse un ejemplo de cómo debe comportarse la función en 4 llamadas distintas.

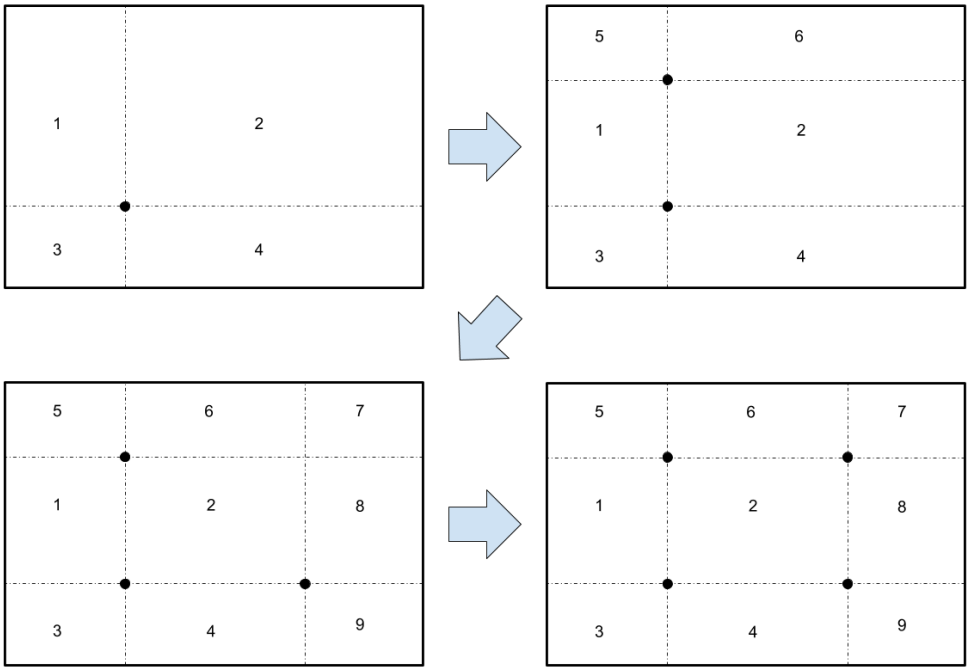


Fig. 4.6: Separación de la pared en planos

4.3 Sistema de estados

La aplicación contará con diversos estados principales: vista estándar, vista en planta ortogonal y vista en planta lateral. Cada uno de estos estados tiene propiedades que le diferencian del resto, tienen cámaras distintas y reaccionan de un modo distinto a los inputs del usuario.

La solución más evidente a esto es utilizar un conjunto de condicionales para controlar lo que se hace en cada uno de los estados; pero cuando estos son largos y complejos, el resultado es un código demasiado complicado y difícil de mantener. Además de este modo es fácil cometer errores y que se realicen acciones del estado incorrecto. Para separar mejor cada estado se ha hecho uso del Patrón de Estado[1].

Con el Patrón de Estado se define una clase princpial llamada `Estado`, que define una serie de métodos abstractos: las acciones a realizar para un mismo evento según el estado. En el caso del planificador sus métodos principales son: `Update`, `In`, `Out`, `OnMouseMove`, `OnMouseClick`, `OnMouseDown`, `OnMouseUp` y `OnScroll`.

Para los inputs de teclado se ha creado un array de comandos (véase 4.4 sobre el Patrón Comando), pudiendo asociar el input con un Comando. Desde la clase `Estado` se recibe el input de teclado y ejecuta el Comando correspondiente.

La clase `Estado` debe entonces extenderse mediante herencia, creando otra clase por cada estado que pueda tener la aplicación. Al cambiar entre estados se ejecutará el método `Out` del estado que cerramos y el método `In` del que entra. Estos estados son instanciados por "World" y según el estado actual los inputs y el método `Update` se redirigen a la instancia correspondiente.

Para que el usuario final pueda cambiar de un estado a otro existirá un input controlado también por "World".

[1] Robert Nystrom. *Game Programming Patterns*. Lightning Source Inc, 2014.

4.4 Gestor de comandos

El Patrón de Comandos[2] permite definir una serie de acciones atómicas. De modo similar a lo que se hace con el Patrón Estado (4.3) se crea una clase padre llamada `Comando`. Desde esta clase pueden extenderse comandos que realicen acciones determinadas. Para ejecutar el comando debe crearse una instancia y llamar al método abstracto `Do` del comando, que se implementa en cada clase derivada.

Este patrón permite implementar la característica de "deshacer" acciones con el método abstracto `Undo`. Del mismo modo que definimos una acción por comando podemos definir una contra-acción que deshaga lo hecho. Esto implica que cada vez que se ejecuta la acción debe guardarse el estado previo para poder recuperarlo, si queremos que el comando pueda deshacerse.

Cuando en un comando se ejecuta la acción, su instancia se añade a una lista de acciones realizadas, llamada `History`. `History` cuenta con un cursor (un número entero que indica una posición en la lista) que indica la acción en la que se encuentra actualmente, y al deshacer una acción el cursor desciende una posición. Si se desea rehacer una acción, se debe buscar la instancia siguiente al cursor y ejecutar de nuevo su acción.

En el caso de que se realice una acción diferente después de haber deshecho un comando, todos los comandos posteriores al cursor se borran, impidiendo que se puedan rehacer. Se trata sin embargo de un comportamiento habitual de la opción "deshacer" en otras aplicaciones, por lo que esto no empeora la experiencia de usuario. Si se alcanza el límite de acciones, el historial de acciones empieza a borrar por las primeras posiciones.

Implementar un comando por cada acción posible en la aplicación es sensiblemente más complicado que realizar las acciones sin más, pero este patrón es una de las pocas formas efectivas de implementar la posibilidad de deshacer. Si se decide que es una característica importante para el programa, es importante empezar a utilizarlo desde el principio, puesto que tratar de incluir este patrón en un programa grande y complejo suele requerir una gran inversión de tiempo.

Los comandos se deben instanciar como un puntero, aunque desde fuera no es necesario gestionar su memoria. La implementación utilizada del Patrón Comando registra automáticamente todas las intancias (aunque no estén en el historial) y libera su memoria cuando es necesario.

Otra característica de la implementación utilizada es que puede ejecutarse un comando de forma "silenciosa" sin que este se registre en el historial, mediante el método `Do_Silent`. Entre otras cosas, esto puede utilizarse para deshacer acciones aprovechando otros comandos existentes: si por ejemplo tenemos la acción "añadir" y "borrar", el método `Undo` de cada uno de ellos puede hacerse con una llamada silenciosa al otro. Esto ahorra complejidad y, si un comando cambia, no es necesario buscar sus equivalentes por el código para reflejar los cambios.

La implementación de cada comando queda como responsabilidad del usuario, en el apartado 5.2.2 se dan detalles del uso que se le ha dado al patrón.

4.5 Clases contra espacios de nombre

Una decisión importante de diseño ha sido la de no utilizar en ningún caso el patrón Singleton. El patrón Singleton limita a una la cantidad de instancias que se puede tener de la clase que lo aplica, lo cual impide que se haga un mal uso de la clase si su propósito era que fuese una clase de una única instancia. Para conseguirlo se hace que el constructor de la clase sea privado y que la clase tenga una referencia estática y privada de sí misma dentro; el Singleton se encarga de manejar su propia instancia.

Aunque es un patrón muy útil y sencillo, en la mayoría de casos puede resolverse el problema de un modo más simple, como utilizando una clase que solo tenga métodos estáticos o, en C++, con funciones dentro de espacios de nombre. Un espacio de nombre es un ámbito (más conocido como scope) que permite limitar el acceso a su contenido, no se puede llamar a ninguna de sus funciones ni acceder a sus variables

[2] Robert Nystrom. *Game Programming Patterns*. Lightning Source Inc, 2014.

si no se especifica antes su espacio de nombre. Desde fuera, puede utilizarse del mismo modo que se haría con una clase puramente estática, pero tiene algunas diferencias importantes:

- Puede extenderse y definirse en ficheros distintos. Es algo que ayuda a la mantenibilidad si el código es extenso y complejo, puesto que es posible separar las diferentes partes. Por otro lado, una desventaja es que el programador usuario del espacio de nombre también podría definir lo que quisiera dentro de este, abriendo la puerta a "hacks", aunque en estos casos se asume que la responsabilidad es de quien utiliza mal la herramienta.
- No tiene métodos ni atributos privados. Sin embargo, es posible emular el comportamiento de los elementos privados extendiendo la clase dentro de la especificación del código, ocultándolo por tanto de los ficheros de cabecera, que son los que permiten que un código sea accesible desde otros puntos del programa.
- No tiene herencia. No es posible utilizar herencia en espacios de nombre, es una característica única de las clases. Si la herencia es importante los espacios de nombre quedan descartados.

Otro de los beneficios de un espacio de nombre es que permite evitar colisiones de nombre entre funciones y variables. Por ejemplo, si una de las librerías utilizadas tuviera una función llamada "max", sería posible redefinirla dentro de un espacio de nombre.

Al final se ha escogido esta opción como sustituto del patrón Singleton en todos los casos donde no se requiera herencia ni instancias.

5 Desarrollo

En este capítulo se describirá el proceso de desarrollo de la aplicación, independientemente de su plataforma.

5.1 Geometría dinámica

Uno de los principales requerimientos para un planificador es la visualización de paredes y configuración de estas para adaptarlas a las medidas de una estancia. Además, deben poderse ubicar ventanas y puertas en las paredes, lo cual afecta a la geometría de la pared dado que se debe poder ver a través de estos elementos. Para ello no es factible utilizar mallas pregeneradas, se necesita generar las paredes de forma dinámica. También se han implementado techos y suelos con mallas dinámicas, pero el algoritmo para generar sus mallas proviene de un código del que ya se disponía antes de comenzar el proyecto, por lo que no se explicará.

En este apartado se describe el proceso de desarrollo de las paredes en 2 iteraciones, una primera en la que se crea una estructura básica para las paredes, y otra en la que se tienen en cuenta las ventanas y puertas.

5.1.1 Generación de la estructura básica de la pared

Por cada pared hay tres primeros puntos 2D relevantes: las esquinas izquierda de las paredes anterior, actual, y siguiente. A partir de estos tres puntos puede deducirse la información de la figura 5.1, donde los vectores "N1" y "N2" son las normales de cada pared, siempre hacia el exterior de la estancia.

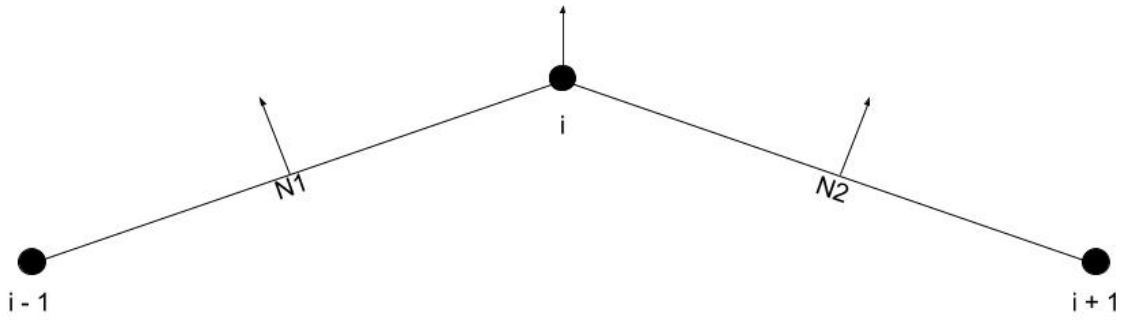


Fig. 5.1: Vectores extraídos a partir de 3 puntos consecutivos.

"N1" y "N2" pueden obtenerse normalizando los vectores de un punto a otro, y girándolos. La dirección del vector central es la suma normalizada de estos dos o, en el caso de las paredes en los extremos cuando el circuito no está cerrado, la normal de la propia pared:

```
1 OBTENER INDICES DE PARED actual, anterior Y siguiente
2
3 v_pc = puntos[actual] - puntos[anterior];
4 v_cn = puntos[siguiente] - puntos[actual];
```

```
5
6  SI cerrado Y actual ES LA PRIMERA O ULTIMA PARED ENTONCES:
7      normal_actual = GIRAR 90 GRADOS ANTI-HORARIO v_cn Y NORMALIZAR
8      direccion = puntos[actual] + normal_actual
9  SINO
10     normal_actual = GIRAR 90 GRADOS ANTI-HORARIO v_cn
11     normal_anterior = GIRAR 90 GRADOS ANTI-HORARIO v_pc
12     direccion = normal_actual + normal_anterior
13     NORMALIZAR direccion
14 FINSI
```

Por último se debe tener en cuenta el grosor que se espera que tenga la pared, dado que si se avanza siempre la misma distancia en el vector director el grosor de estas dependería del ángulo que formen con sus paredes adyacentes. Esto se resuelve con trigonometría:

```
1  SI cerrado ENTONCES:
2      direccion = profundidad_pared / absoluto(producto_punto(normal_actual, direccion));
3  SINO
4      direccion = profundidad_pared * normal_actual
5  FINSI
6
7  punto_esquina = puntos[actual] + direccion
```

El producto punto de dos vectores normales es el coseno del ángulo que forman entre sí. En este momento "direccion" incluye la dirección y distancia entre el punto actual y el punto de la esquina exterior, permitiendo generar dicho punto a partir del actual. En el caso de que la pared que estamos generando no haga esquina con otra, simplemente se utiliza la normal de la pared actual.

Una limitación de este sistema es que los puntos deben introducirse en sentido anti-horario respecto al interior de la habitación. De lo contrario la normal de cada pared queda invertida y estas se extienden hacia el lado opuesto al que deberían, provocando algunos artefactos no deseados.

5.1.2 Índices para la estructura básica

La pared tiene 23 vértices y no 8 como cabría esperar. Esto se debe a que al renderizar, los shaders interpolan la normal de cada vértice con la de sus vecinos; si el mismo vértice se encuentra en dos caras distintas, la normal del vértice no coincide con la de la superficie en la cara que se está pintando. El resultado de esto sería que el color varía en los bordes de cada cara. Esta propiedad es muy útil para objetos que no tienen ángulos tan marcados, pero en este caso se busca que las caras sean muy marcadas y totalmente planas.

Para solucionarlo se repite cada vértice tantas veces como el número de caras en el que se encuentre, de modo que aunque todos se encuentren en la misma posición, cada cara está utilizando un vértice distinto.

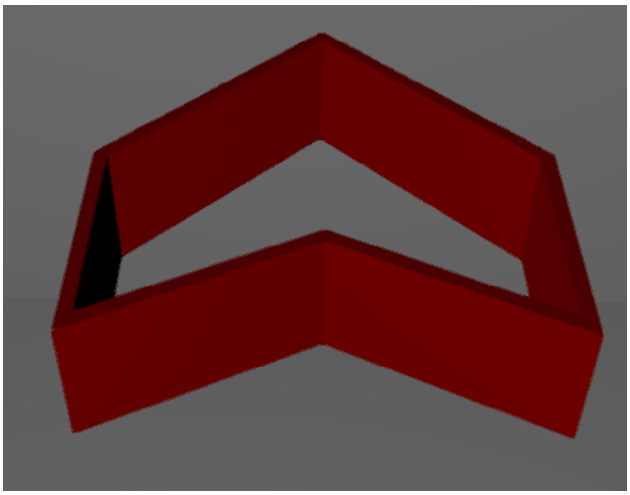


Fig. 5.2: Ejemplo de generación de paredes.

5.1.3 Modificando la estructura para permitir la inclusión de ventanas

Como se puede ver en el apartado de diseño (4.2.4) se requiere modificar la geometría para que el lado anterior y posterior de las paredes sean idénticos. Esto, sin embargo, complica las conexiones entre los vértices, que se han tenido que generar manualmente. Con los nuevos vértices añadidos en total hay 35 vértices.

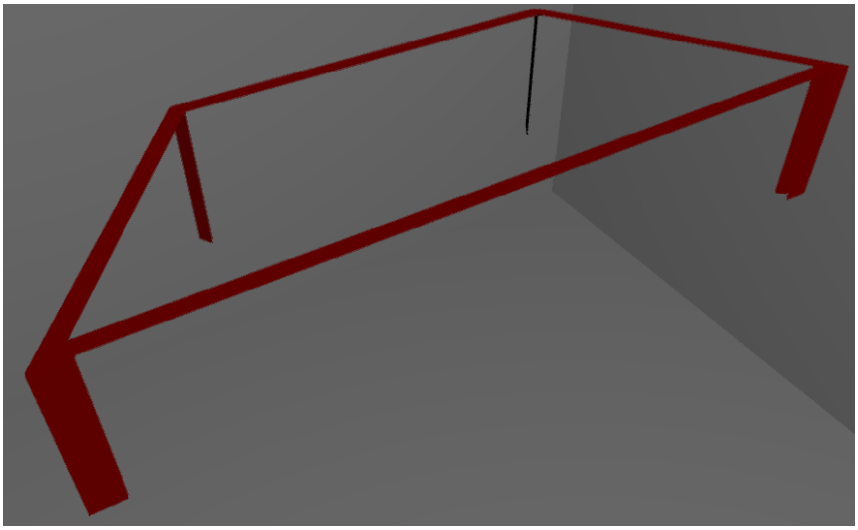


Fig. 5.3: Aspecto de las paredes sin los planos anterior y posterior.

En la figura 5.3 puede verse el resultado. Nótese que los planos inferiores no se ven porque sus normales apuntan hacia abajo y el motor gráfico los oculta como optimización. Al crear los índices se ha tenido en cuenta el orden de estos, pues afecta al cálculo de las normales de los vértices.

5.1.4 Generación de ventanas I: proyección sobre pared

Como ya se ha mencionado en el apartado 4.2.4, las ventanas están definidas por un punto, una altura y un ancho. No se incluye ninguna información respecto a que pared es la que va a contener dicha ventana; por lo que se cogerá la pared más cercana al punto dado.

Para ello se proyecta el punto sobre la línea AB de cada una de las paredes, calculando la distacia hasta cada una de las proyecciones para ver cuál es la más cercana (véase el apéndice A.2 sobre la proyección punto-línea).

Como preparación para el próximo apartado (5.1.5) se calcula sobre la pared los 4 puntos que delimitarán la ventana. Para ello se proyecta el punto de la pared sobre el plano que forma esta, y se calcula el resto de puntos desplazándonos por dicho plano (véase el apéndice A.3 sobre la proyección punto-rectángulo):

```
1 projectVertices(pared, referencia a hueco):
2     origen = PROYECTAR ORIGEN DEL HUECO SOBRE PLANO DE LA PARED
3
4     hueco.A1 = origen
5     hueco.B1 = origen + DIRECCION DERECHA * hueco.ancho
6     hueco.A1H = origen + DIRECCION ARRIBA * hueco.alto
7     hueco.B1H = hueco.A1H + DIRECCION DERECHA * hueco.ancho
8 FIN DE FUNCION
```

5.1.5 Generación de ventanas II: modificación de la geometría de la pared

Al final del apartado 5.1.4, ya se conoce el punto en que están los extremos de la ventana sobre la pared. En este apartado se explica cómo dividir la pared en diferentes planos y descartar aquellos que correspondan a

un agujero.

Una vez más las proyecciones cobran mucho protagonismo (véase el apéndice A.2 sobre la proyección punto-línea). Lo primero que se busca es dividir el plano de la pared por los vértices del hueco, obteniendo una colección de planos.

El algoritmo para conseguir esto tiene como entrada y salida una lista de planos, que empieza siendo uno solo que cubre toda la pared. Mientras iteramos los puntos, proyectamos estos sobre cada lado para obtener los puntos por los que hay que cortar los planos, y posteriormente se añaden a la lista de planos desechando el original. En caso de que una de las proyecciones no contribuya a crear un nuevo plano (como por ejemplo, los puntos inferiores de la puerta en la figura 5.4) la ignoramos.

Posteriormente se comprueba cuales de estos planos forman parte de una ventana y se eliminan de la lista, dejando un hueco en dicha posición. Este algoritmo permite además añadir múltiples ventanas, aumentando la posible complejidad de la pared.

Por último se reprocesan los planos generados comprobando si sus lados coinciden en alguna dirección, en cuyo caso se reúnen para reducir la complejidad. En la figura 5.4, se puede ver un ejemplo de como se separan los planos con múltiples ventanas, y cómo se reúnen los planos adyacentes.

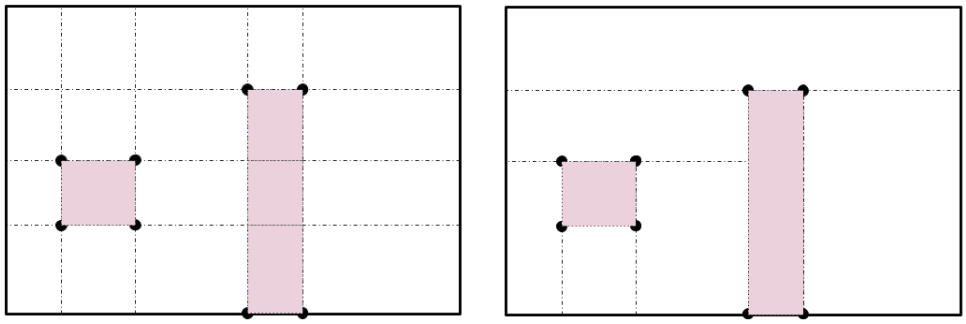


Fig. 5.4: Ejemplo de pared con una ventana y una puerta, y muestra de una posible reducción de los planos.

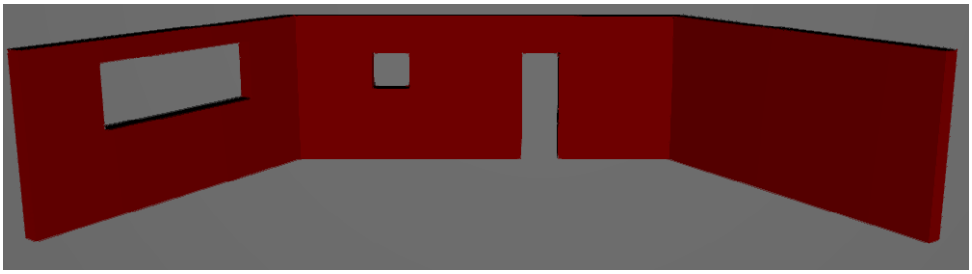


Fig. 5.5: Ejemplo de generación de paredes con huecos y estancia sin cerrar.

5.1.6 Generación de uvs

Para que el motor aplique correctamente las texturas sobre las paredes, se requiere que las uvs estén a escala de mundo; es decir, se espera que dada una distancia entre dos vértices de la mima malla, sus uvs tengan la misma distancia. La mayor dificultad en este caso está en que los vértices se encuentran en espacio tridimensional, mientras que las uv son coordenadas bidimensionales de la textura que estamos mapeando. Por lo tanto, de algún modo hay que desconsiderar la orientación de la pared y centrarnos en sus superficies.

Como la geometría está formada por planos perfectos, la solución encontrada ha consistido en partir siempre de la esquina inferior izquierda de cada uno de ellos. Como precondición, la esquina "A" tiene la uv (0,0).

Por lo tanto por cada plano, se ha definido la uv de la esquina inferior izquierda, y los vectores hacia los cuales "avanzan" las componentes `x` e `y` de las uv (Fig. 5.6).

Fig. 5.6: Datos necesarios para la generación de las uv “2” y “3”.

La razón de usar proyecciones y no la magnitud de los vectores directamente, es que se desconoce en cual de las dos direcciones se encuentra el vértice que observamos al iterar. Proyectando cada punto tenemos la distancia en cada una de las dos direcciones y se la sumamos a la uv del vértice “1”:

```
1  genUV(origen, xDir, yDir, punto, uv_origen, longitud_x, longitud_y):
2      uv.x = PROYECTAR punto SOBRE RAYO EN DIRECCION xDir
3      uv.y = PROYECTAR punto SOBRE RAYO EN DIRECCION yDir
4
5      SI uv.x ES SUPERIOR A LA LONGITUD HORIZONTAL DE LA PARED ENTONCES:
6          uv.x = longitud_x - uv.x
7      FINSI
8      SI uv.y ES SUPERIOR A LA LONGITUD VERTICAL DE LA PARED ENTONCES:
9          uv.y = longitud_y - uv.y
10     FINSI
11
12     uv = uv + uv_origen
13     DEVOLVER uv
14 FIN DE FUNCION
```

Dado que las uv no pueden ser negativas, se hace un paso en las líneas 6-11 para que, en caso de serlo, se les sume la longitud total del lado en el que se encuentran.

5.1.7 Generación de normales, tangentes y bitangentes

El cálculo de cada normal es el resultado de la suma de la normal de cada polígono en el que se encuentra dicho vértice, y para calcular esta se hace el producto vectorial (normalizado) de los vectores que van del primer vértice hacia el segundo y el tercero:

```
1 DADOS LOS PUNTOS v1, v2 Y v3
2
3 normal = NORMALIZAR ( PRODUCTO CRUZADO DE (v2 - v1) Y (v3 - v1))
```

Las tangentes y bitangentes son un poco más complicadas: todo vector tiene infinitos vectores tangentes, pero en este caso no sirve cualquiera. El vector tangente a la normal tiene que estar siempre alineado con las uv.

Como se explica en el tutorial 13 de opengl-tutorial.org[1], para ello debemos resolver el siguiente sistema de ecuaciones:

$$deltaPos1 = deltaUV1.x * T + deltaUV1.y * B$$
$$deltaPos2 = deltaUV2.x * T + deltaUV2.y * B$$

Esto se computa del siguiente modo:

```
DADOS TRES PUNTOS v1, v2 y v3
vec1 = v2 - v1
vec2 = v3 - v1

deltaUV1 = uv2 - uv1
deltaUV2 = uv3 - uv1

r = 1.0 / (deltaUV1.x * deltaUV2.y - deltaUV1.y * deltaUV2.x)

tangente = (vec1 * deltaUV2 - vec2 * deltaUV1.y) * r
bitangente = (vec2 * deltaUV1.x - vec1 * deltaUV2.x) * r
```

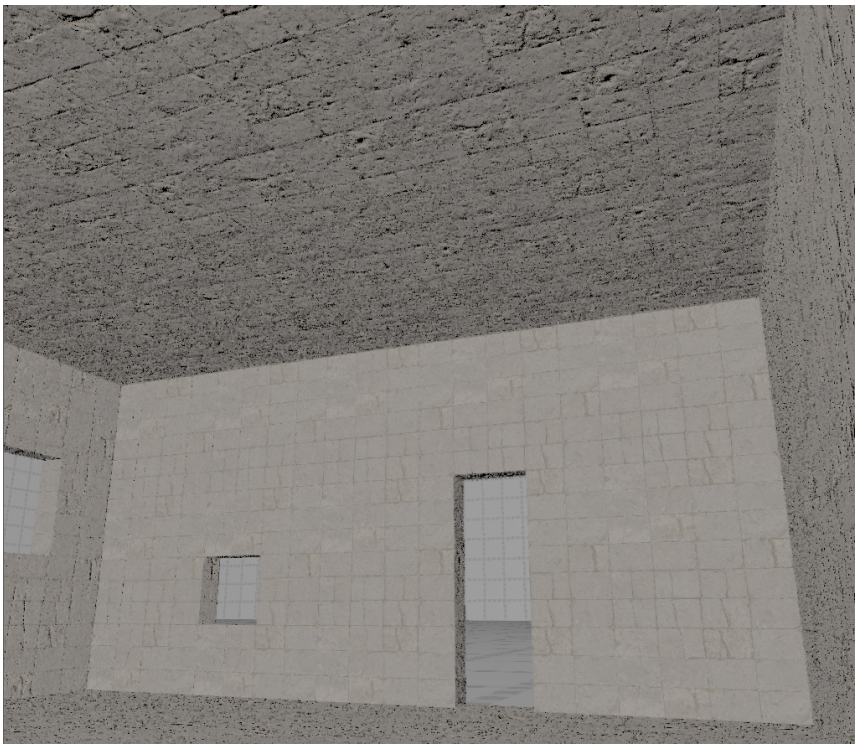


Fig. 5.7: Ejemplo de una habitación con texturas de prueba.

5.2 Interacción

En este apartado se darán algunos detalles sobre cómo se ha implementado la interacción del usuario con la aplicación.

5.2.1 Gestión de los inputs y uso del Patrón Estado

El motor gráfico Manta tiene una interfaz para configurar los inputs. Se puede definir la función que debe llamarse cuando se da un evento de input determinado.

[1] Sam Hocevar. *OpenGL Tutorial*. `http://www.opengl-tutorial.org/`.

Como se ha explicado en el capítulo 4, los inputs se redirigen al estado de la aplicación que esté activo en ese momento a través de `World`. Durante el desarrollo los estados utilizados han sido: `StandardView`, `OverView` y `LateralView`. Podrían definirse más estados en futuros desarrollos.

Cada estado configura la cámara en su método `In`. `OverView` y `LateralView` configuran la cámara a modo ortogonal y la sitúan en frente del elemento que se quiere mostrar: una vista hacia abajo de toda la habitación en el caso de `OverView` y una vista frontal de cada una de las paredes en el caso de `LateralView`. Por su parte, `StandardView` tiene una cámara libre que puede moverse por el usuario final.

El estado más complejo es `OverView`, desde el cual puede modificarse la estancia. Las paredes pueden moverse arrastrando una pared en sí o su esquina, en ambos casos afectando también a las paredes que estén conectadas.

Al hacer click, el método `OnMouseClick` se llama con la posición del click como parámetro. Para saber en que elemento se ha hecho click, los vértices de cada una de las paredes se transforman a coordenadas de pantalla usando las matrices de vista y perspectiva de la cámara. Los vértices superiores de la pared forman un rectángulo en dos dimensiones que puede compararse con la posición del ratón. A su vez, calculando la distancia entre la posición del ratón y la de los vértices en las esquinas puede saberse si se ha clickado en una esquina. Una vez detectado el elemento con el que se está interactuando se guarda su identificador en el estado, y un booleano que indica que el ratón se está manteniendo pulsado.

Mientras el booleano siga activado, el método `OnMouseMove` del estado mandará la posición del ratón y el elemento seleccionado al framework de la aplicación (véase el apartado 3.1), que procesará el modo en que reaccionan los elementos de la escena a la interacción. La respuesta del framework a ese proceso se utilizará para regenerar las paredes.

Para permitir que esto ocurra en tiempo real se utilizarán comandos, como se explica en el apartado 5.2.2.

5.2.2 Uso del Patrón Comando

El Patrón Comando, explicado en el apartado 4.4, permite definir distintas acciones y contra-acciones para la aplicación.

Toda la interacción con el gestor de paredes y huecos se ha realizado a través de comandos. Por lo tanto, la aplicación permite hacer y deshacer acciones como mover, añadir o eliminar paredes y huecos.

Un defecto del Patrón Comando es que requiere crear una instancia cada vez que se realiza un comando. Esto puede ser un problema cuando estamos interactuando con el programa de forma constante, como por ejemplo moviendo una pared con el ratón. No es viable crear una instancia de un comando cada vez que el ratón se mueve un píxel en la pantalla, y además no tiene sentido poder deshacer una acción si esta se ha realizado un gran número de veces para mover un poco algún elemento.

Para solucionarlo, se ha añadido el métodoi `Update` sólo a los comandos que requerían una actualización constante durante un tiempo determinado. La instancia del comando se crea con la información del elemento a modificar, y guardando el estado previo, pero la acción en sí se realiza con el método `Update` en vez de `Do`. Al llamar a `Update` no se guarda la acción en el historial, por lo que podemos realizarla todas las veces que sea necesario; cuando la acción termine (normalmente, cuando el usuario suelte el ratón), se llama al método `Do` para que la acción quede registrada.

Con este sistema se registra una sola acción para algo que realmente se ha hecho muchas veces en un período de tiempo. Si se deshace la acción se volverá al estado previo a empezar la acción.

Otros casos como añadir y eliminar elementos no han necesitado esta complejidad añadida.

6 Compilación a web con Emscripten

Aunque la exportación a Emscripten puede considerarse parte del proceso de implementación, su extensión e importancia en este documento hace que valga la pena dedicarle su propio capítulo.

Uno de los puntos fuertes de nuestro planner es el poder ejecutarlo en web. El hecho de estar programado en C++, sin embargo, dificulta esta tarea. Teniendo cuenta una serie de consideraciones, es posible adaptar código C++ a web a través de Emscripten.

6.1 Consideraciones previas

En esta sección se describen diversas consideraciones que se han tenido en cuenta a la hora de adaptar el código con Emscripten.

6.1.1 Sobre Emscripten

Emscripten es capaz de compilar código IL (Intermediate Language) a Javascript. IL es un lenguaje de código máquina que se puede generar compilando desde otros lenguajes, entre ellos C++ usando el compilador LLVM. Dado que Emscripten compila desde IL (y no C++ directamente) realmente esto significa que podemos ejecutar código escrito en diversos lenguajes a través de Emscripten.

Emscripten[1] nació como respuesta a la necesidad de escribir programas para web en lenguajes tradicionales de escritorio, con el objetivo de ganar velocidad, trabajar con un lenguaje familiar, y aprovechar las librerías existentes en dichos lenguajes. Hasta su aparición la única alternativa para poder hacer esto era el uso de plugins embebidos en el navegador, lo cual fracasó debido a la falta de estandarización y a la evolución de los propios navegadores, que cada vez más rechazan los plugins desde la popularización de las plataformas móviles.

Dada la extensión de funcionalidades de Javascript, para realizar el compilado se ha utilizado un subset de este, "asm.js". Asm.js es un lenguaje intermedio que podemos ejecutar en navegadores; tiene una serie de restricciones con respecto a Javascript, como por ejemplo el hecho de ser un lenguaje tipado (aunque Javascript no lo es, en asm.js se comprueba el tipo de cada variable antes de trabajar con ella) o el requerimiento de trabajar sobre una única pila de memoria (un simple array de Javascript dentro del cual deben estar todos los datos que utilice el programa).

Si se aplica esta filosofía al código que programamos en C++, se puede entender fácilmente que dicha pila de memoria equivale a la memoria tal y como la ve un programa tradicional en C++. Es posible reservar memoria dentro de esta pila, y tener punteros a posiciones de la pila (que en realidad no son más que simples índices). Tiene un funcionamiento similar al Memory Pool descrito en el apartado 3.1.6.

Aunque en cuanto a eficiencia el código en asm.js en web está en torno al 50-67% de su equivalente en escritorio, los resultados pueden ser sorprendentes pues asm.js es capaz de superar en velocidad a su equivalente escrito en Javascript tradicional. Al trabajar sobre datos estáticos y memoria pre-inicializada, el código

[1] Alon Zakai (Emscripten). *EMSCRIPTEN & ASM.JS: C++'S ROLE IN THE MODERN WEB*. `https://kripken.github.io/mloc_emscripten_talk/cppcon.html`.

resultante cuenta con una serie de optimizaciones que un programador no suele realizar manualmente.

Una de las actuales tendencias del web es la ejecución de código máquina, para lo cual está en fase de desarrollo la tecnología "WebAssembly". Sin embargo, aún a día de hoy no está del todo claro cuando estará lista lo suficientemente avanzada[2], por lo que el papel de asm.js es importante como paso intermedio, para que los desarrolladores puedan ir adoptando la tecnología antes de ser funcional del todo. Emscripten también tiene un rol muy importante en este proceso, porque es el encargado tanto de compilar a asm.js hoy como lo será de compilar a WebAssembly en el futuro[3].

Emscripten ya se está utilizando hoy en día en diversos proyectos de cierta categoría, especialmente en aplicaciones gráficas.

Es importante también tener en cuenta algunos de los defectos de utilizar Emscripten respecto a programar directamente en Javascript:

- Es necesario saber desde el primer momento cuanta memoria se va a necesitar. Aunque superar dicha memoria no hará que el programa falle (Emscripten es capaz de reasignar memoria cuando se sobrepasa el límite), permitir que esto ocurra no es en absoluto recomendable, puesto que ralentizaría el programa.
- Se requiere un proceso de adaptación considerable: muchas de las cosas que el programador da por sentadas en C++ no se cumplen en Javascript. Como se puede ver a lo largo del capítulo 6, no se puede inicializar el programa, gestionar los ficheros o ejecutar el bucle del programa del mismo modo en que se haría en C++.
- Se pierde poder de decisión sobre algunas de las tecnologías que se pueden querer utilizar porque simplemente no están disponibles, como versiones modernas de OpenGL o las operaciones SIMD (un tipo de operaciones que permite realizar cálculos con múltiples datos en una sola instrucción[4]). Esto cambiará en el futuro según avance la tecnología, pero a día de hoy es un limitante.
- Si se desea que el mismo código se pueda ejecutar en diversas plataformas, habrá otro proceso de adaptación considerable para hacer que el programa tenga en cuenta las consideraciones mencionadas según la plataforma. En realidad esto es más bien un defecto de la programación multi-plataforma que de Emscripten, pero se debe tener en cuenta que existirá un esfuerzo extra de desarrollo y un aumento de la complejidad del código.
- El debugging resulta sensiblemente más incómodo que con Javascript o C++ en escritorio[5]. Normalmente para seguir los errores se hace uso de impresiones a través de la consola y el stack-trace que devuelve Emscripten cuando se produce un error (el cual no siempre es suficientemente descriptivo).

6.1.2 API gráfica

Como se ha podido ver en el apartado 3.2, el motor está pensado para funcionar utilizando la API gráfica Vulkan. El primer paso para hacer funcionar el programa con Emscripten ha sido utilizar una API gráfica que sea compatible con las tecnologías web.

WebGL está basado en OpenGL ES 2, y Emscripten puede reconocer las llamadas a esta segunda API y transformarlas en llamadas a WebGL. De modo que para hacer funcionar el motor será necesario con buscar los puntos donde se interactúe con la API de Vulkan y cambiarlos para que funcionen con OpenGL ES 2.

En el momento de realizar la exportación se deberá tener en cuenta que Vulkan y WebGL son completamente incompatibles. En el fondo no se sustituyen directamente las llamadas a la API de Vulkan, sino que se detectan los puntos en que un conjunto de código se puede considerar equivalente a una llamada de

[2]Lin Clark (Mozilla). *Where is WebAssembly now and what's next?* `https://hacks.mozilla.org/2017/02/where-is-webassembly-now-and-whats-next/`.

[3]WebAssembly. *WebAssembly Roadmap*. `http://webassembly.org/roadmap/`.

[4]Huon Wilson. *What is SIMD?*. `http://huonw.github.io/blog/2015/07/what-is-simd`.

[5]Emscripten Contributors. *Debugging*. `https://kripken.github.io/emscripten-site/docs/porting/Debugging.html`.

OpenGL. El código de Vulkan es mucho más complejo y largo de lo que requiere OpenGL, y lo que en Vulkan requiere muchas líneas de código puede hacerse en OpenGL con una sola llamada.

También será necesario adaptar los shaders y el proceso de carga de estos, dado que OpenGL ES 2 tiene limitaciones como la necesidad de definir el número de variables que recibirá el shader antes de compilarlo, dificultando la carga dinámica de luces. Aunque ambas APIs utilizan el mismo lenguaje de shading, GLSL, no es posible reutilizar sin más el código porque se utilizan diferentes versiones de este. En WebGL se utiliza la versión 100 de GLSL mientras que SPIR-V (véase 3.3) parte de la versión 450; se trata de versiones muy diferentes y aunque se han reutilizado algunos fragmentos, no es posible reaprovechar el código por completo.

6.1.3 Comunicación C++/Javascript

Como es de esperar, es posible (necesario, de hecho) comunicar el código de Javascript con el código compilado desde C++[6]. En el desarrollo de la aplicación en C++ pueden definirse funciones pensadas para ser llamadas desde Javascript, y en el momento de compilar deben especificarse cuáles son esas funciones. Posteriormente, el Javascript resultante al compilar pone a disposición del desarrollador las funciones `cwrap` i `ccall`.

La función `cwrap` provee una función intermedia que se puede llamar como si de una función normal de Javascript se tratara. Esta función se encargará de llamar a su vez a la función de C++ correspondiente. En cambio, `ccall` permite llamar directamente a la función de C++, pero como se explica a continuación es más verboso, por lo que si la función va a ser llamada varias veces puede ser más recomendable utilizar la primera alternativa.

Ambas opciones requieren que se especifique la firma de las funciones que van a ser llamadas. La firma de una función define el tipo de los parámetros que recibe y el tipo del valor de devuelve. De ahí que se diga que `ccall` es más verboso: cada vez que se llame es necesario especificar la firma mientras que con `cwrap` solo se debe hacer una vez.

Otro detalle importante a tener en cuenta es que no cualquier función puede ser llamada desde Javascript. Emscripten sólo permite que llamemos a funciones propias del lenguaje C, dado que C++ altera el nombre de las funciones al compilarlas (C++ permite a dos funciones llamarse igual si tienen firmas distintas, pero lo resuelve cambiando los nombres en el compilado resultante). Para garantizar que las funciones sean propias de C cuando programamos en C++ se utiliza el prefijo `extern "C"` antes de la definición.

Como se verá en el apartado 6.1.4 la comunicación entre C++ y Javascript es importante para controlar el flujo de la aplicación, pero también lo será para que desde el lado web se puedan crear interacciones que tengan efecto en el planificador. Con HTML y Javascript se pueden crear toda clase de eventos que pueden transmitirse a C++, como elementos de interfaz. Además, el planificador no controla los datos relacionados con el dominio de la aplicación, como por ejemplo qué muebles están disponibles en qué países; esta información se transmitirá a la aplicación a través de la web.

6.1.4 Inicialización y bucle de ejecución

Uno de los inconvenientes de Emscripten es que Javascript ejecuta todos los procesos relacionados con la página en un único hilo de ejecución, incluyendo el renderizado, la gestión de eventos y la actualización de la propia página.

Javascript tiene una naturaleza orientada a eventos[7]: dispone de una cola de eventos que se van añadiendo y ejecutando en un orden difícil de predecir. Entre esos eventos estará la propia ejecución del código Javascript y en consecuencia el código generado con Emscripten, pero también el resto de eventos mencionados que gestionan el comportamiento de la página. Si uno de los eventos tarda demasiado en terminar, bloqueará la cola de eventos y provocará que toda la página se bloquee.

[6] Emscripten Contributors. *Interacting with code*. `http://kripken.github.io/emscripten-site/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html`.

[7] Mozilla Developer Network. *Conceptos de un programa en ejecución*. `https://developer.mozilla.org/es/docs/Web/JavaScript/EventLoop`.

Como el desarrollador no sabe en que orden se ejecutan los eventos, se dice que el lenguaje es asíncrono, y eso implica que se debe programar con cuidado para controlar que todo ocurra en el orden que se espera. La naturaleza asíncrona de Javascript puede confundir al programador haciéndole pensar que puede utilizar más de un hilo de ejecución (dado que en otros lenguajes la programación multi-hilo y la asincronía están estrechamente relacionadas); es importante tener presente que no es posible realizar dos tareas al mismo tiempo en este lenguaje. Sí se espera, en cambio, que WebAssembly pueda en el futuro ejecutar programas multi-hilo.

Como se menciona en el apartado 6.1.3, el flujo de la aplicación debe controlarse desde Javascript. El bucle de ejecución (3.1.1) es un bucle infinito, por lo que si lo ejecutamos estando dentro de C++ bloquearemos por completo la aplicación impidiendo que la cola avance. Por lo tanto el primer paso para hacer funcionar la aplicación pasa por hacer que las funciones `Update` y `Draw` del motor se llamen desde Javascript.

Para conseguirlo se debe utilizar la función `setInterval`[8], que dado un intervalo de tiempo añade un evento para llamar a una función repetidamente. Especificando un intervalo de $1/6$ segundos, podemos hacer que la función de `Update` se llame 60 veces por segundo. Es importante tener en cuenta que setInterval no garantiza que la función se llame realmente en el intervalo dado, sino que ese el mínimo de tiempo que se tardará en llamar; si la aplicación se ralentiza Javascript esperará a que termine la repetición anterior para llamar de nuevo. También el cálculo del tiempo entre frames (explicado en el apartado 3.1.1) se hará desde Javascript aunque esto no es estrictamente necesario.

Antes de iniciar el bucle de ejecución se llamará a una función `Init`, con el objetivo de incializar el motor y el contexto de WebGL.

6.1.5 Gestión de ficheros

A lo largo de la ejecución del planificador será necesario proveer al software con una serie de assets (modelos 3D o texturas) y datos que provienen de ficheros. En C y C++ típicamente se accede al sistema de ficheros a través de la función `fopen`, que solicita al sistema operativo acceso a un fichero determinado; y después se extrae la información de este.

En C++, los datos del fichero que se quiere leer pasan a estar disponibles justo en el momento en que se piden, de forma síncrona; pero el proceso es mucho más complicado en Javascript. En Javascript no tenemos un sistema de ficheros, y si queremos tener acceso a alguno, debemos antes solicitarlo a un servidor remoto.

Es un problema porque las peticiones a servidores remotos se hacen de forma asíncrona (técnicamente se puede hacer de forma síncrona, pero se trata de una función obsoleta, que debe ser evitada y que tiene efectos secundarios como el bloqueo de la página); el hecho de que el funcionamiento sea diferente según la plataforma dificulta considerablemente el desarrollo.

Emscripten pone a disposición de los desarrolladores varias soluciones para facilitar el acceso a ficheros externos[9]. Para que los cambios a realizar según la plataforma sean mínimos, Emscripten emula un sistema de ficheros de modo que podamos utilizarlo con `fopen` igual que lo haríamos en C++. El modo en que esos ficheros pasen a estar disponibles en el sistema de ficheros, en cambio, puede variar:

- Precarga sistema de ficheros: Los ficheros que se pre-carguen con este sistema, en el momento de compilar, se añadirán a un único archivo binario que se descarga al navegador junto al código generado. El sistema de archivos de Emscripten tiene una lista de estos ficheros con sus rutas y su posición en el archivo de precarga. Teniendo en cuenta que tanto los modelos 3D como las texturas son bastante pesados, y que en un planificador puede haber una gran cantidad, esto aumentaría considerablemente el peso de la descarga inicial, y el tiempo de carga; pero la ventaja es que los archivos pasan a estar disponibles de manera síncrona y sin cambiar en absoluto el código que los lee.

[8] Mozilla Developer Network. *WindowOrWorkerGlobalScope.setInterval() (Javascript Web APIs).* `https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setInterval`.

[9] Emscripten Contributors. *Files and File Systems.* `http://kripken.github.io/emscripten-site/docs/porting/files/index.html`.

- Descarga síncrona: Emscripten provee la función `emscripten_wget`, que dada la url de un archivo la descarga de forma síncrona a través del navegador. También se necesita dar un nombre al archivo cargado puesto que Emscripten lo guarda en el sistema de archivos virtual para que podamos acceder a él con `fopen`. Aunque con este método no sería necesario cambiar el flujo de la ejecución, como ya se ha mencionado este sistema es obsoleto.
- Descarga asíncrona: La alternativa asíncrona al anterior método es `emscripten_async_wget`. Del mismo modo que antes, esta función carga el archivo solicitado en el sistema de ficheros virtual, pero si intentamos leerlo justo después de llamar a la función la carga fallará, puesto que la carga aún no se ha realizado realmente. Para acceder al archivo debemos utilizar un puntero de función (o callback) que pasamos como parámetro a `emscripten_wget_async`, y que se llamará si el archivo se carga correctamente. También debemos pasarle otro callback que se llamará en caso de que la descarga fracase, y que nos permite gestionar tal evento. El callback de carga también recibe como parámetro los datos binarios del fichero solicitado, de modo que aunque seguimos pudiendo acceder a él desde el sistema de ficheros virtual con `fopen`, no es realmente necesario. Este sistema nos obliga a cambiar el flujo del programa, puesto que al no disponer aún del fichero solicitado, no podemos hacer nada que haga uso de este hasta que el callback de éxito se haya llamado.

La segunda opción ni siquiera será tenida en cuenta por sus características. Por un lado la primera alternativa obliga a descargar todos los assets desde el principio, y por otro la última no da acceso inmediato a los archivos como se suele presuponer cuando se programa en C++.

La solución pasa por utilizar una mezcla de ambas. Cuando se carga un modelo 3D o una textura, por la fuerza es necesario tener una respuesta inmediata, pero una vez se ha cargado un asset y se está utilizando, nada impide cambiar en un momento dado cambiar los datos de ese asset con un contenido diferente. Para tener acceso inmediato a los assets se pre-cargargan unos assets "dummies", muy pequeños, que no supone ningún coste añadir junto con la carga de la app.

Cuando se quiera tener acceso a un asset, este se solicitará de forma asíncrona, pero inmediatamente después se proveerá al motor con un dummy, y se guardará el identificador para tener acceso al asset guardado. En el momento en que la carga asíncrona finalice, el asset dummy será sustituido por el correcto. Durante el tiempo intermedio se verá en pantalla un asset que no es el que debería verse, pero se trata de un tiempo generalmente muy corto y da un efecto de "carga" al que los usuarios ya están bastante habituados.

6.2 Pasos para la exportación

Una vez descritas las consideraciones a tener en cuenta para poder realizar la exportación, se va a proceder a exportar el código. En esta sección se describirá el proceso paso a paso.

6.2.1 Gestión del código discordante

En C++, las discordancias entre las distintas plataformas se resuelve mediante macros. Una macro es un fragmento de código que se ejecuta antes de la compilación, y es capaza de modificar el código en función de diversos factores.

Normalmente, el código que es específico para una plataforma provoca un fallo en el resto de plataformas, por lo que se requiere "intercambiar" estos fragmentos de código incompatibles. También las librerías a utilizar pueden cambiar y la firma de algunas funciones.

Cuando compilamos con Emscripten, se define la macro `__EMSCRIPTEN__`, permitiéndonos controlar fácilmente el código que se compilará en cada plataforma.

Un modo de entender lo que hacen las macros en C++ es imaginar que literalmente modifican el código. Desde el punto de vista del código generado, el uso de macros es equivalente a borrar y añadir la parte correspondiente a una u otra plataforma. Los distintos tipos de macros son simplemente formas diferentes de realizar modificaciones al código en tiempo de compilación.

```
#ifdef __EMSCRIPTEN__
    // Codigo especifico para Emscripten
#endif

#ifndef __EMSCRIPTEN__
    // Codigo que debe ignorarse al compilar con Emscripten
#endif
```

También es necesario en determinadas ocasiones utilizar funciones distintas para realizar la misma operación en diferentes plataformas. Sería posible solucionar este caso mediante la macro `#ifdef`, pero tendría que hacerse cada vez que se utiliza dicha función.

Las macros de definición de C++ permiten incluir parámetros; por ejemplo la macro `#define f(A) A * 2` cogería el parámetro "A" y lo multiplicaría por dos. Lo interesante es que el usar las macros de este modo no requiere que "A" tenga ningún tipo en especial, ni siquiera que sea una variable; siempre y cuando el código resultante al procesar la macro sea válido.

Esta característica se utiliza, por ejemplo, con la función `fopen`. En Windows se requiere utilizar la función `fopen_s`, que tiene una firma distinta que `fopen`; mientras que en Emscripten solo se reconoce esta última, provocando un fallo de compilación si se utiliza `fopen_s`. Se puede solucionar esto con la macro:

```
#ifdef __EMSCRIPTEN__
#define fopen_s(pFile,filename,mode) ((*(pFile))=fopen((filename),(mode)))==NULL
#endif
```

Definiendo esto podemos utilizar `fopen_s` con normalidad sin tener que preocuparnos por la plataforma dado que, sólo en Emscripten, cuando el compilador se encuentre una llamada a `fopen_s` la sustituirá por su equivalente.

Aunque las macros pueden resultar tentadoras, es buena costumbre no abusar de ellas, dado que pueden terminar generando un código extremadamente confuso y difícil de mantener. Es por ello que durante todo el desarrollo siempre se ha exigido tener una buena razón antes de utilizarlas, especialmente en niveles más altos de complejidad como en este último ejemplo.

6.2.2 Compilación

Normalmente se compila a Emscripten mediante la terminal de comandos. Se debe conocer en cierta profundidad las opciones de compilado, por lo que en este apartado se introduce el proceso de compilación y algunas de las opciones más genéricas. En los próximos apartados se comentarán otras opciones más específicas.

Tras la instalación del SDK de Emscripten, se dispone el comando `emcc`[10]. En su forma más básica, este comando puede utilizarse especificando los ficheros C++ de input y el fichero Javascript de output con la opción `-o`: `emcc codigo.cpp -o resultado.js`. En la práctica, será necesario aplicar diversas opciones para hacer que un código complejo funcione.

A continuación se describen los que se han utilizado para hacer funcionar el planificador:

- `-s FULL_ES2=1`: Activa la emulación de OpenGL ES 2. Sin esta opción no se traducen las llamadas de OpenGL a llamadas de WebGL.
- `-std=c++14`: Especifica la versión de C++ a compilar, en este caso C++14. Si se utilizan las características más nuevas del lenguaje, compilar con una versión más antigua puede provocar errores.
- `-msse` y `msse2`: Permite incorporar instrucciones SIMD SSE1 y SSE2. Las instrucciones SIMD permiten realizar cálculos con múltiples datos con una sola instrucción[11] y no se profundizará sobre ellas en este documento. Son utilizadas por algunos componentes del motor para ganar eficiencia.

[10] Emscripten Contributors. *Building Projects*. `http://kripken.github.io/emscripten-site/docs/compiling/Building-Projects.html`.

[11] Huon Wilson. *What is SIMD?*. `http://huonw.github.io/blog/2015/07/what-is-simd`.

- `-s TOTAL_MEMORY=N`, donde N es un número de bytes: Permite especificar la memoria que se le asigna al programa (véase la sección 6.1.1). Dicha memoria será inicializada una sola vez al ejecutar el programa. Mediante la opción `-s ALLOW_MEMORY_GROWTH=1` se permite que en caso de superar el límite de memoria, esta se readapte, lo cual es extremadamente ineficiente pero puede ser útil para hacer debugging. Por defecto Emscripten asigna 16MB de memoria.

- `-s NO_EXIT_RUNTIME=1`: Hace que al finalizar la ejecución de la función `main` las funciones definidas desde C++ sigan estando disponibles para llamarse desde C++, y por extensión el resto de código disponible para ejecutarse.

- `-s EXPORTED_FUNCTIONS="['_f1','_f2',...,'_fn']"`: Define las funciones que están disponibles para ser llamadas desde Javascript. Para ello las funciones deben definirse con el prefijo `extern "C"`. Los nombres de las funciones van siempre con un caracter "_" antes.

- `-I path`, donde "path" es la ruta de una carpeta o fichero: Define una carpeta o fichero de cabecera para que el compilador pueda encontrarlos. Esto incluye también las librerías externas.

- `--preload-file buildpath@runpath` donde buildpath es la ruta de un fichero o carpeta en el momento de compilar y runpath la ruta del mismo en tiempo de ejecución: Permite añadir ficheros estáticos al sistema de ficheros de Emscripten. Estos ficheros se pueden cargar de forma síncrona. Mediante el símbolo "@" podemos definir un alias que será donde se ubicarán estos ficheros en el sistema de ficheros virtual.

6.2.3 Exportación de la API gráfica

Realizar un render en OpenGL resulta considerablemente más simple de lo que sería utilizando Vulkan. Utilizando las herramientas descritas en el apartado 6.2.1 es posible sustituir los fragmentos de código referentes a Vulkan por sus equivalentes en OpenGL ES 2.

Vulkan dispone de una serie de funciones de incialización. En su equivalente de OpenGL se aprovechan estas llamadas para incicializar OpenGL, compilar los shaders y obtener los identificadores de cada uno de los uniforms de este. También aquí se especifican todos los parámetros de OpenGL.

La escena del motor Manta, explicada en el apartado 3.1.3, está gestionada por el espacio de nombre `SceneManager`. `SceneManager` se encarga de extraer los datos de la estructura de árbol en cada iteración, prepararlos, y ejecutar el renderizado.

Al crear los componentes de malla de las distintas entidades que hay en la escena (explicados en el apartado 3.1.4) es necesario cargar en memoria de GPU los datos de la malla (a no ser que esta haya sido cargada anteriormente, en cuyo caso se puede reutilizar). Ocurre igual con las texturas del material de la entidad. Al cargar estos elementos obtenemos un identificador para cada uno que debe guardarse y mantener relacionado con la entidad.

Tanto este proceso como el propio renderizado se realiza dentro del propio `SceneManager`. El renderizado de cada entidad en OpenGL puede resumirse en los siguientes pasos:

- Especificar el shader que va utilizarse para el renderizado, a través de un identificador que hemos obtenido al compilarlo.

- Transmitir a los drivers los datos de cada uniform, cuyos identificadores hemos guardado previamente, incluyendo los de la cámara, las luces, así como la transformación de la entidad.

- Especificar las texturas y datos que utiliza el material de la entidad y los distintos componentes de la malla por separado: vertices, uvs, colores, normales, tangentes, bitangentes e índices. Todos estos elementos ya están en memoria y solo es necesario especificar sus identificadores antes de renderizar.

- Mandar la orden de renderizado a OpenGL.

Todo este proceso se realiza mediante llamadas a la API de OpenGL ES 2 que, al compilar el código, Emscripten reconocerá y traducirá a su equivalente en WebGL.

6.2.4 Implementación de la carga asíncrona

Como se ha descrito en el apartado 6.1.5, las funciones de carga asíncrona de Emscripten requieren definir una función de callback. Primero se resuelve la solicitud de archivo con una carga síncrona de un fichero de menor peso para después sustituir los datos con los datos correctos, una vez se hayan terminado de cargar.

Es importante entender por tanto que van a existir dos callbacks: el callback que llama Emscripten, para informar de que ha finalizado la carga del fichero, y a su vez otro callback que se llamará desde el gestor de ficheros para informar al código que ha solicitado el fichero de que ha finalizado la carga.

Esto dispara considerablemente la complejidad de cargar ficheros en C++. El gestor de ficheros de Manta no sabe realmente qué clase de dato se está cargando, así que para poder realizar la carga "doble" se ha hecho que el propio gestor de fichero también reciba funciones callback como parámetro, que serán definidas desde el gestor de mallas y de texturas.

Gestor de ficheros

Se ha definido la función `getFile_async`, que será llamada cuando se solicite un fichero y la plataforma de ejecución sea Emscripten. Sus parámetros son: la ruta del fichero a cargar; un puntero de argumentos para el callback, en forma de variable con el tipo `size_t`; y el callback para cuando se resuelva la petición, que recibirá por parámetro un puntero sin tipo (es decir, con tipo `void`) que contendrá el puntero de argumentos recibido desde `getFile_async` y los datos del propio archivo recibido.

Seguimos teniendo sin embargo un problema: desde el callback de Emscripten es necesario conocer el callback del gestor de ficheros, y sus argumentos. No es posible guardar esta información en atributos del gestor de ficheros debido a la propia asincronía, que nos impide saber cuando podemos disponer de nuevo de dicha variable. Si se solicitaran dos ficheros seguidos, los datos del segundo sobreescribirían el primero antes de resolverse la petición. La siguiente aproximación más evidente sería utilizar un mapa que relacione los datos de cada petición con el nombre del fichero solicitado, pero eso tampoco es sencillo de implementar porque el mismo fichero podría pedirse dos veces, haciendo que se sobreescriban los datos de todos modos.

Para solucionar este problema se ha utilizado el puntero de argumentos que proporciona emscripten para la función `emscripten_async_wget`. Los parámetros de esta función son: la ruta del fichero solicitado, un puntero de argumentos sin tipo, un callback de éxito de carga y otro de fracaso de carga.

El callback de éxito tiene como parámetros: el puntero de argumentos, un puntero sin tipo que contiene los datos del fichero, y un entero que indica el tamaño de dichos datos. El callback de fracaso simplemente recibe el puntero de argumentos.

Para tener accesibles los callbacks del gestor de ficheros desde los callbacks de Emscripten se han guardado todos los datos en el puntero de argumentos de la función `emscripten_async_wget`. De este modo es posible extraer los datos desde el callback de Emscripten y llamar al callback del gestor de ficheros desde este. Esta operación requiere tener especial cuidado dado que al introducir varios datos en un puntero sin tipo se debe tener en cuenta el orden y tamaño de los datos, y extraerlos del mismo modo en que se han introducido.

Teniendo todo esto en cuenta, el pseudocódigo del gestor de ficheros es el siguiente:

```
1  VARIABLES EXTERNAS:
2      cache: relacion entre ruta_fichero y datos del fichero
3      intentos_por_ruta: relacion entre el numero de intentos de carga y la ruta del
           fichero
4      maximos_intentos: numero maximo de intentos
5
6  onGetFile_error(argumentos):
7      EXTRAER DE argumentos LAS VARIABLES:
8          ruta_fichero,
9          argumentos_callback_externo
10
11     SI SE HA SUPERADO EL NUMERO DE INTENTOS
12         NOTIFICAR ERROR
```

```
13          DETENER_PROGRAMA
14      SINO
15          LLAMAR A emscripten_async_wget_data CON LOS CALLBACKS onGetFile_success,
                onGetFile_error Y argumentos
16      FINSI
17  FIN DE FUNCION
18
19  onGetFile_success(argumentos, puntero_datos, tamaño_fichero):
20      EXTRAER DE argumentos LAS VARIABLES:
21          tamaño(ruta_fichero),
22          ruta_fichero,
23          tamaño(callback_externo),
24          callback_externo,
25          puntero_argumentos
26
27      COPIAR A datos_fichero:
28          puntero_datos,
29          tamaño_fichero,
30          ruta_fichero
31
32      COPIAR datos_fichero A cache
33
34      LLAMAR AL CALLBACK EXTERNO CON datos_fichero
35  FIN DE FUNCION
36
37  getFile_async(ruta_fichero, puntero_argumentos, callback_externo):
38      SI archivo EN cache ENTONCES:
39          LLAMAR AL CALLBACK EXTERNO CON LA INFORMACION
40          FIN DE FUNCION
41      FINSI
42
43      VARIABLE argumentos
44
45      COPIAR A argumentos EN ORDEN:
46          tamaño(ruta_fichero),
47          ruta_fichero,
48          tamaño(callback_externo),
49          callback_externo,
50          puntero_argumentos
51
52          LLAMAR A emscripten_async_wget_data CON LOS CALLBACKS onGetFile_success,
                onGetFile_error Y argumentos
53  FIN DE FUNCION
```

Un detalle que puede observarse es que cuando el archivo ya está en caché, el callback se llama al momento y de forma síncrona. En el siguiente apartado veremos que esto no afecta al comportamiento dado que antes de llamar a `getFile_async`, el código que la llama debe haber cargado la versión síncrona. Como el orden de sucesos es el mismo en el fondo no afecta negativamente el hecho de que el callback se llame síncronamente. En cualquier caso, las funciones de carga de assets también cuentan con su propia caché por lo que, normalmente, esta función ni siquiera se llamará si el fichero ya había sido cargado anteriormente.

De cara a la versión en producción de la aplicación, el callback de error reintentará la carga un número arbitrario de veces y detendrá la aplicación si no se llega a resolver en ninguno de los intentos.

Gestor de mallas y texturas

Una vez resuelto el problema de la carga asíncrona, debemos gestionarla desde las funciones que hacen uso de esta. Como ya se ha comentado anteriormente C++ espera recibir los datos de un fichero de forma síncrona. Para realizar la carga doble el proceso es: cargar los datos de baja calidad, obtener los identificadores de las entidades que hacen uso de estos datos, esperar la llamada asíncrona y finalmente, cargar de nuevo los datos esta vez sobre los ya existentes.

La función de carga se reutiliza tanto para realizar la primera carga como para volver a hacerlo con los nuevos datos. Para diferenciar los dos casos está el parámetro `async_ready`, que por defecto es falso y se le asigna verdadero si la llamada procede del callback de éxito. Por último, si `async_ready` es verdadero,

hay un tercer parámetro con los datos del fichero.

Antes ver el código es importante tener presente el apartado 3.1.7, donde se explica el funcionamiento de los identificadores en el motor Manta.

Como el código es equivalente tanto para mallas como para texturas, para esta explicación se generaliza haciendo referencia a "assets", pero en el código esta funcionalidad se encuentra repetida en ambos gestores. El pseudocódigo del proceso es:

```
1
2  VARIABLES EXTERNAS:
3      relacion_nombre_id: mapa de cadenas de texto a numeros enteros
4
5  callback_asset(puntero_argumentos, datos_fichero)
6      EXTRAER DE datos_fichero LAS VARIABLES:
7          ruta_fichero,
8          puntero_datos
9
10     BORRAR DATOS ANTIGUOS DE API GRAFICA
11
12     LLAMAR A cargarAsset EN MODO async_ready Y CON LOS DATOS DEL FICHERO
13 FIN DE FUNCION
14
15 cargarAsset(ruta_fichero, async_ready = FALSO, data = 0) => numero_entero:
16     SI async_ready ENTONCES:
17         CARGAR DATOS DESDE LA VARIABLE DE DATOS
18         SUSTITUIR DATOS DEL ID CARGADO PREVIAMENTE POR LOS NUEVOS
19     SINO
20         SI ruta_fichero EN relacion_nombre_id ENTONCES:
21             DEVOLVER EL ID GUARDADO
22         SINO
23             CARGAR DATOS DE ARCHIVO DUMMY
24             LLAMAR A getFile_async PARA CARGAR ARCHIVO EN ALTA CALIDAD
25         FINSI
26     FINSI
27
28     DEVOLVER ID DEL ASSET CARGADO
29 FIN DE FUNCION
```

El parámetro de argumentos para el callback no se utiliza en este caso, puesto que es un remanente de anteriores iteraciones de el código, pero se ha decidido conservar la característica como decisión de diseño, porque puede ser útil en otras situaciones.

Como `id_asset` no se modifica en ningún momento desde que se genera, externamente no se necesita hacer nada para poder trabajar con los datos finales, el motor ya se encarga automáticamente de que el identificador provisto haga referencia a los datos correctos. En la figura 6.1 puede verse el proceso completo cuando el fichero no ha sido cargado previamente, donde "A" y "B" son los datos a los cuales apunta el identificador, dicho de forma abstracta.

En el tiempo que transcurre desde que se devuelve el identificador del asset hasta que se llama la función de callback, el motor trabajará con los datos que se han obtenido del fichero `.mock`. Normalmente la carga es prácticamente instantánea pero para archivos grandes y/o conexiones lentas puede ser apreciable. En cualquier caso, aunque fuera completamente inapreciable para el usuario final y no afectara a la experiencia, la asincronía obliga a tomar decisiones de diseño como estas, puesto que de lo contrario el motor debería trabajar con datos vacíos hasta disponer de los correctos, lo cual no es posible.

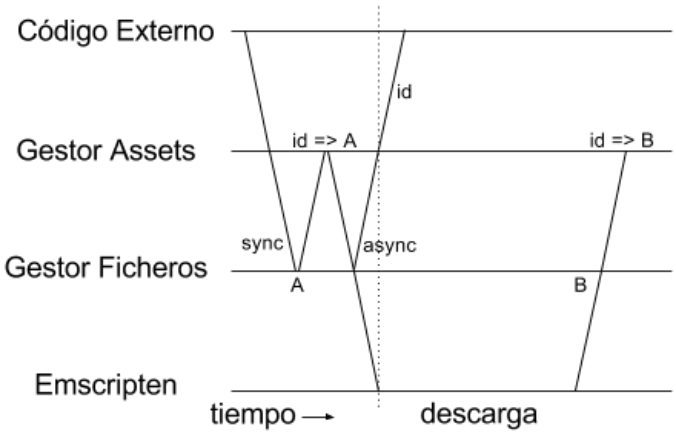


Fig. 6.1: Esquema de la carga de ficheros.

7 Conclusiones

7.1 Estado actual del desarrollo

Para el desarrollo de la interfaz en la versión actual de la aplicación se ha hecho uso de la librería ImGui[1]. Se trata de una API que permite añadir diversos tipos de elementos de interfaz en la aplicación. Aunque es muy fácil de utilizar y versátil, probablemente será insuficiente de cara a la versión en producción de la aplicación: ImGui está altamente enfocado a interfaces de debugging y los requisitos estéticos de los planificadores no podrán cumplirse fácilmente. Será necesario buscar alternativas, por lo que de momento ImGui sólo se utiliza para debugging.

Una posible alternativa, al menos como paso intermedio antes de crear una interfaz más avanzada dentro del canvas 3D, es integrar la interfaz dentro del HTML. El buen funcionamiento de la aplicación en web hace que ya sea posible empezar a desarrollar la interfaz de usuario en HTML, utilizando la comunicación entre C++ y Javascript. Es una buena opción para empezar a trabajar en la interfaz sin interrumpir el desarrollo de la propia aplicación.

Al comenzar el desarrollo existía cierto escepticismo sobre las posibilidades de Emscripten. Llevar código de lenguajes de escritorio a plataformas web hasta hace poco ha sido algo muy poco común, a excepción de algunos plug-ins de navegador con muy poca recepción. Las primeras pruebas con Emscripten resultaron sorprendentes y eso fue un aliciente.

A pesar de ello, durante el desarrollo y la exportación del código con Emscripten han habido muchos problemas. Los más importantes han sido:

- La compilación de código de C++ a Javascript es considerablemente lenta, hasta aproximadamente 5 minutos. En los momentos en los que han surgido dificultades específicas de Javascript o Emscripten esto ha ralentizado el desarrollo, puesto que se han necesitado varias iteraciones hasta dar con la combinación correcta de elementos. Aunque las interrupciones solo sean de 5 minutos la pérdida de tiempo productivo puede ser mucho mayor en muchos casos. Debido a esto la mayor parte del desarrollo se ha realizado debugando primero en escritorio, y no se ha empezado a probar en Emscripten hasta tener muy avanzado el código.
- Las diferencias entre la naturaleza síncrona de C++ y asíncrona de Javascript han generado dificultades. Las primeras aproximaciones para ejecutar el motor en web incluían el bucle de ejecución dentro del código C++, provocando el bloqueo de la aplicación. Esto ha afectado especialmente a la gestión de ficheros.
- La exportación del sistema de ficheros tiene un gran número de consideraciones a tener en cuenta. Aunque Emscripten ofrece herramientas muy efectivas para la carga de ficheros tanto síncrona como asíncrona, implementar dicha carga sin irrumpir demsiado en el comportamiento del motor gráfico ha sido difícil. El apartado 6.1.5 sobre la implementación del sistema de ficheros en Emscripten es el resultado de varias iteraciones según surgían problemas inesperados. Por suerte, la posibilidad de cargar assets "dummys" ha facilitado una solución al problema de la asincronía.
- La implementación del Patrón Comando, explicada en el apartado 5.2.2 también es el resultado de varios intentos. El desconocimiento del patrón hizo que no se tuvieran en cuenta diversos detalles,

[1] Omar Cornut and contributors. *ImGui*. `https://github.com/ocornut/imgui`.

como la necesidad de realizar acciones sin registrarlas en el historial o agrupar en un solo comando un conjunto de acciones cuando estas ocurren en tiempo real. La flexibilidad que ofrece la herencia de C++ hace que muchas soluciones puedan aplicarse directamente en los comandos, pero ello requiere conocer dichas soluciones por lo que la creación de comandos no está carente de cierta complejidad.

- La generación de mallas dinámicas ha tenido un gran número de detalles que no se habían tenido en cuenta al comenzar el desarrollo. El motor no estaba preparado para la actualización de mallas dinámicas y fue necesario añadir dicha funcionalidad, tanto en su versión de escritorio como en WebGL. El desarrollo se empezó sin disponer de dicha funcionalidad pero se pudo avanzar generando ficheros dinámicamente y leyéndolos al instante. Ha habido un gran número de pequeños bugs que han ralentizado el proceso. Se ha necesitado tener una gran cantidad de código antes de poder empezar a probarlo. Puede considerarse un acierto haber realizado el desarrollo de forma iterativa, puesto que no tenía sentido empezar a generar huecos dentro de las paredes sin tener funcionando una versión básica de estas.
- Ha habido pequeñas diferencias entre la API de OpenGL ES 2 y WebGL que han sido difíciles de solucionar. Un ejemplo es que OpenGL ES 2 permite asignar a un mismo buffer dos targets diferentes, lo cual fue un error en el código pero no hacía fallar la aplicación, mientras que en WebGL sí. En este punto se han unido varios factores como la falta de conocimiento sobre la API, la dificultad de debugar Emscripten, la poca cantidad de gente que se ha encontrado con problemas similres en Internet (el uso de Emscripten aún no está muy extendido) y la extensión del código del motor gráfico, que hace difícil encontrar los puntos específicos donde se produce un error.

Tras esta experiencia no se puede afirmar categóricamente que Emscripten sea la mejor alternativa para todas las situaciones, hay que recordar que Javascript también puede trasladarse a otras plataformas; pero Emscripten ha demostrado ser perfectamente capaz de trasladar código C++ a Javascript, y ejecutarlo con la misma estabilidad y una eficiencia muy similar.

Aunque la interfaz de usuario deja mucho que desear, el código es capaz de añadir, mover y eliminar paredes así como añadir, mover y eliminar huecos en estas. Todas las acciones pueden deshacerse y rehacerse, lo cual es una característica que nunca había existido en ninguna aplicación de la empresa.

Ha sido muy satisfactorio ver los buenos resultados de realizar un diseño preliminar, incluyendo la decisión de utilizar patrones. El código ha ganado calidad y mantenibilidad gracias a ello.

Por otro lado, el código tiene margen de mejora en cuanto a los gestores y generadores de entidades, explicados en el apartado 4.2. Muchas de las buenas ideas introducidas en este apartado (como el uso de identificadores para que no se manipulen los datos desde fuera) se han incorporado a mitad del desarrollo y con bastante prisa. El resultado es que en su estado actual, mantener este fragmento de código requiere entender muy bien lo que hace. No debería ser difícil sin embargo, una vez conocidos los problemas y sus soluciones, mejorar este fragmento, pues está muy aislado del resto del código.

El hecho de ser un proyecto en empresa ha aportado ventajas y desventajas al desarrollo. Por un lado es difícil coordinar las prioridades de desarrollo con las prioridades del departamento de ventas. Ha habido mucha presión para tener resultados rápidamente a menudo en detrimento de la calidad del código. Existe una cierta deuda técnica en el código que pasará factura en el futuro. Por contra, trabajar en una empresa ha hecho contar con el apoyo de compañeros con muchos conocimientos sobre la materia. El motor gráfico es propio de la empresa y no se puede buscar información sobre este en Internet pero no solo se ha contado con ayuda para el desarrollo sino que incluso se ha podido adaptar el propio motor a las necesidades que han surgido. El hecho de tratarse de un proyecto real que será utilizado por una gran empresa eventualmente es un gran componente motivador, además del aliciente económico y la presión, aunque a priori no es una experiencia agradable, ayuda a no dormirse. Tener un horario estable de ocho horas diarias ha hecho que los avances fueran constantes y estables.

Habría sido deseable que más personas participaran directamente en el desarrollo de la aplicación. En un principio el equipo iba a estar formado por 3 personas además del equipo que programa el motor. Al final la presión de otros proyectos de la empresa ha hecho que solo hubiera un programador de en la mayor parte del desarrollo y eso ha afectado negativamente las previsiones iniciales.

Estos inconvenientes sin embargo han ido acompañados de comprensión por parte de la empresa, y una readaptación de las expectativas. En términos globales puede decirse que el balance de trabajar en un

proyecto de empresa es más que positivo.

7.2 Futuras iteraciones

La adición de elementos interiores e interacción con estos no está terminada, aunque ya se sabe cómo se van a organizar estos elementos (véase el apartado 4.2). Tampoco está hecha la interfaz de usuario y los inputs deben pulirse. Estas serán las prioridades en el futuro inmediato.

A más largo plazo, se deben implementar las diferentes aplicaciones que van a hacer uso del planificador. Se requerirá hacer aplicaciones con múltiples habitaciones y elementos estructurales como posiblemente zonas exteriores (balcones, terrazas o jardines). El planteamiento de diseñar el planificador para ser genérico hará que esto se pueda hacer con relativamente poco esfuerzo. Teniendo en cuenta que la empresa trabaja con diferentes aplicaciones en ambientes muy diferentes, hacer una aplicación genérica ha sido un acierto de cara al futuro.

Es probable que, una vez conocidos los problemas que se han encontrado a lo largo del desarrollo, se de otra iteración a algunos fragmentos del código, para mejorar la eficiencia y la estructura del propio código. Estos fragmentos se han desarrollado con prisa y asumiendo una cierta deuda técnica. Sin embargo, el diseño modular de la aplicación hace que sea posible realizar estas iteraciones trabajando con fragmentos aislados de código.

En su conjunto, la mantenibilidad del código es bastante buena y deberían poder incorporarse nuevos desarrolladores sin excesivo esfuerzo. Algunos fragmentos de código, en cambio, requieren un conocimiento más profundo del problema y de los elementos que intervienen en la solución. Un desarrollador descuidado podría llegar a tener problemas con dichos fragmentos por lo que existe margen de mejora en este aspecto.

A Utilidades matemáticas

Normalmente un motor gráfico incluye una serie de herramientas para facilitar el desarrollo, pero debido algunas carencias del motor Manta en este aspecto, se han programado como parte de la aplicación. Este apéndice se presenta como referencia para los fragmentos de código que hacen uso de estas herramientas a lo largo de este documento, y como muestra del funcionamiento de estas.

A.1 Comparación de tipos imprecisos

En la mayoría de dispositivos los números de coma flotante tienen un cierto nivel de imprecisión. Esto no suele ser un problema porque suelen tener mucha más precisión de la necesaria, y en su defecto hay otras formas de conseguir aún más precisión. El problema es no se pueden comparar dichos números porque rara vez van a ser *exactamente* idénticos. Para ello se han creado las funciones `compare_float(n1, n2, precision)` y `compare_vec(v1, v2, precision)`.

Estas funciones comparan que el valor absoluto de la diferencia entre los dos números sea inferor a la precisión requerida que por defecto es 0.01 pero puede configurarse con un parámetro. Como los vectores en glm utilizan números de coma flotante, se hace lo mismo para poder compararlos, pero esta vez comparando sus 3 componentes.

A.2 Proyección punto-rayo y punto-línea

Entendiendo un rayo como un elemento formado por un punto y una dirección y una línea como un segmento de un rayo delimitado por dos puntos, se han creado las funciones:

```
1 point_ray_projection(origen_rayo, direccion_rayo, punto)
2 point_line_projection(punto_linea_A, punto_linea_B, referencia resultado) => booleano
```

Su funcionamiento es muy similar, de hecho la segunda hace uso de la primera para obtener el resultado, pero tienen dos diferencias importantes: `point_ray_projection` devuelve la distancia entre el origen del rayo y la proyección de nuestro punto, en la dirección especificada, mientras que la función `point_line_projection` devuelve un booleano que indica si la proyección está dentro de nuestra línea, y asigna a la referencia "result" el punto exacto de la proyección.

Hay diferentes situaciones en las que puede ser más útil una u otra: a veces se querrá saber el punto exacto de la proyección, otras veces la distancia de esta proyección respecto al punto de origen (sin necesidad de calcular el punto), y otras comprobar si esta proyección se encuentra delimitada entre dos puntos.

El cálculo de la proyección rayo-punto se basa en las propiedades del producto punto. A continuación se explica el razonamiento por el cual el producto punto puede usarse para calcular una proyección entre dos vectores.

El producto punto, o producto escalar de dos vectores, cumple la siguiente fórmula:

$$dot(A, B) = |A||B| * cos(\Theta)$$

Donde θ es el ángulo que forman los dos vectores. Si asumimos que A es un vector normal (se garantizará normalizandolo desde la propia función), esto se reduce a:

$$dot(|A|, B) = |B| * cos(\Theta)$$

Si miramos esta ecuación desde el punto de vista trigonométrico, B puede entenderse como la hipotenusa del triángulo que forman A, B, y el vector de B a la proyección de B en A. El coseno, por definición, nos indica el ratio entre el lado contiguo de una esquina y la hipotenusa de un triángulo, por lo que al multiplicarlo por la magnitud de B obtenemos la longitud del lado contiguo, es decir, la distancia entre A y la proyección de B.

$$distancia = dot(direccion, punto - origen)$$

Una vez calculada esta distancia, se puede sacar el punto de proyección con la fórmula:

$$P = origen + |direccion| * distancia$$

A.3 Proyección punto-plano y punto-rectángulo

De un modo similar al visto en el apartado A.2, se calcula la proyección a partir del producto punto.

Para ello debe disponerse de la normal de dicho plano. En el caso de la proyección punto-plano se requiere como parámetro para definir el plano, pero en el caso de la proyección punto-rectángulo la calculará a partir de 3 puntos del plano. Este detalle es importante porque el orden de dichos puntos afectará a la dirección de la normal.

Dados 3 puntos A, B y C, la normal del plano que forman es el producto vectorial de los vectores que van de un punto hacia los otros dos, normalizados.

$$normal = |(B - A) \times (C - A)|$$

Proyectando el punto sobre el rayo que forman la normal del plano y un punto cualquiera de este, podemos saber a qué distancia se encuentra el punto de dicho plano. Como disponemos de la normal, podemos invertirla y multiplicarla por la distancia obtenida para extraer la proyección sobre el plano.

$$distancia = dot(normal, punto - A)$$

$$P = punto - normal * distancia$$

A diferencia de lo que ocurría en el apartado A.2, este método no comprueba que la proyección se encuentre dentro del rectángulo. En el apartado A.5 se explica como realizar esta comprobación.

A.4 Comprobar si cuatro puntos forman parte del mismo plano

En un espacio de D dimensiones, $D+1$ puntos forman parte del mismo espacio de $D-1$ dimensiones si el determinante de la matriz formada por las posiciones de los puntos organizadas verticalmente es 0[1]. En 3D:

$$\begin{vmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \\ 1 & 1 & 1 & 1 \end{vmatrix} = 0$$

Glm ya dispone de una implementación para calcular el determinante de una matriz por lo que no es necesario profundizar más.

A.5 Comprobar si la proyección de un punto sobre un plano está dentro de un rectángulo

Aunque es similar, este se diferencia del apartado A.4 en que no se requiere que el punto que queremos comparar esté en el mismo plano que el resto. En algunos casos se desea comprobar si la proyección está dentro de un rectángulo sin importar cuál sea esa proyección.

Para ello utilizamos la proyección punto-línea explicada en el apartado A.2. La proyección del punto está dentro del rectángulo si se puede proyectar con éxito sobre sus cuatro lados. Poder proyectarlo sobre un lado implica que se podrá proyectar también sobre el lado opuesto, así que en el fondo solo se necesitan dos comprobaciones.

```
1 in_rec(A, B, C, punto) => booleano:
2     VARIABLES p1, p2 COMO VECTOR
3     DEVOLVER
4         point_line_projection(A, B, punto, p1)
5         Y
6         point_line_projection(A, B, punto, p2)
7 FIN DE FUNCION
```

[1] Martin von Gagern. *Check if a point is on a plane*. https://math.stackexchange.com/questions/684141.

Bibliografía

Planner5D. *Planner5D*. https://planner5d.com/.

IKEA. *Historia de IKEA – cómo empezó todo*. http://www.ikea.com/ms/es_ES/about_ikea/the_ikea_way/history.

Roca. *Empresa global - 100 años de historia que nos han convertido en referente a nivel mundial*. http://www.roca.es/nuestra-empresa/sobre-nosotros/una-empresa-global.

GeForce. *Adaptive VSync Technology*. http://www.geforce.com/hardware/technology/adaptive-vsync/technology.

(Nvidia), Phill Miller. *NVIDIA Brings AI to Ray Tracing to Speed Graphics Workloads*. https://blogs.nvidia.com/blog/2017/05/10/ai-for-ray-tracing.

Nystrom, Robert. *Game Programming Patterns*. Lightning Source Inc, 2014.

Julián Pérez Porto, María Merino. *Definición de albedo*. http://definicion.de/albedo/.

PBR, Free. *Free PBR*. http://freepbr.com.

(Marmoset), Jeff Russell. *Basic theory of physically-based rendering*. https://www.marmoset.co/posts/basic-theory-of-physically-based-rendering.

Lenaert, David. *Screen-Space Ambient Occlusion: Battling your Contrast Bias*. http://www.derschmale.com/2013/12/12/screen-space-ambient-occlusion-battling-your-contrast-bias.

Group, Khronos. *Vulkan® 1.0.50 - A Specification*. https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html.

— *History Of Opengl*. https://www.khronos.org/opengl/wiki/History_of_OpenGL.

— *The Standard for Embedded Accelerated 3D Graphics*. https://www.khronos.org/opengles/.

Adobe. *Stage3D*. http://www.adobe.com/devnet/flashplayer/stage3d.html.

Group, Khronos. *OpenGL and multithreading*. https://www.khronos.org/opengl/wiki/OpenGL_and_multithreading.

Apple. *Guides and sample code*. https://developer.apple.com/library/content/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/OpenGLESontheiPhone/OpenGLESontheiPhone.html.

Google. *OpenGL ES*. https://developer.android.com/guide/topics/graphics/opengl.html.

— *Vulkan Graphics API*. https://developer.android.com/ndk/guides/graphics/index.html.

Graham Sellers Richard S. Wright, Nicholas Haemel. *OpenGL SuperBible*. Addison Wesley, 2015.

Group, Khronos. *OpenGL Shading Language*. https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language.

— *The first open standard intermediate language for parallel compute and graphics*. https://www.khronos.org/spir.

— *Isn't glslang an official GLSL to SPIR-V compiler?* https://www.lunarg.com/faqs/glslang-glsl-spir-v-compiler/.

Henning, Neil. *An Introduction to SPIR-V*. http://cdn2.imgtec.com/idc-docs/gdc16/AnIntroductionToSPIR-V.pdf.

Group, Khronos. *ARB_gl_spirv*. https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_gl_spirv.txt.

Hocevar, Sam. *OpenGL Tutorial*. http://www.opengl-tutorial.org/.

(Emscripten), Alon Zakai. *EMSCRIPTEN & ASM.JS: C++'S ROLE IN THE MODERN WEB*. https://kripken.github.io/mloc_emscripten_talk/cppcon.html.

(Mozilla), Lin Clark. *Where is WebAssembly now and what's next?* https://hacks.mozilla.org/2017/02/where-is-webassembly-now-and-whats-next/.
WebAssembly. *WebAssembly Roadmap*. http://webassembly.org/roadmap/.
Wilson, Huon. *What is SIMD?* http://huonw.github.io/blog/2015/07/what-is-simd.
Contributors, Emscripten. *Debugging*. https://kripken.github.io/emscripten-site/docs/porting/Debugging.html.
— *Interacting with code*. http://kripken.github.io/emscripten-site/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html.
Network, Mozilla Developer. *Conceptos de un programa en ejecución*. https://developer.mozilla.org/es/docs/Web/JavaScript/EventLoop.
— *WindowOrWorkerGlobalScope.setInterval() (Javascript Web APIs)*. https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setInterval.
Contributors, Emscripten. *Files and File Systems*. http://kripken.github.io/emscripten-site/docs/porting/files/index.html.
— *Building Projects*. http://kripken.github.io/emscripten-site/docs/compiling/Building-Projects.html.
Cornut, Omar and contributors. *ImGui*. https://github.com/ocornut/imgui.
Gagern, Martin von. *Check if a point is on a plane*. https://math.stackexchange.com/questions/684141.