UNIVERSIDAD
DE GRANADA

TESIS DOCTORAL

Programa de Doctorado en
Tecnologías de la Información y la Comunicación

Sistemas interpretados en GPUs: Aplicaciones en patrimonio histórico y visualización expresiva

DOCTORANDO

Luis López Escudero

DIRECTORES

Germán Arroyo Moreno

Domingo Martín Perandrés

Departamento de Lenguajes
y Sistemas Informáticos

Año 2023

Editor: Universidad de Granada. Tesis Doctorales
Autor: Luis López escudero
ISBN: 978-84-1117-024-4
URI: https://hdl.handle.net/10481/84669

El doctorando / The *doctoral candidate* **[Luis López Escudero]** y los directores de la tesis / and the thesis supervisor/s: **[Germán Arroyo Moreno y Domingo Martín Perandrés]**

Garantizamos, al firmar esta tesis doctoral, que el trabajo ha sido realizado por el doctorando bajo la dirección de los directores de la tesis y hasta donde nuestro conocimiento alcanza, en la realización del trabajo, se han respetado los derechos de otros autores a ser citados, cuando se han utilizado sus resultados o publicaciones.

/

Guarantee, by signing this doctoral thesis, that the work has been done by the doctoral candidate under the direction of the thesis supervisor/s and, as far as our knowledge reaches, in the performance of the work, the rights of other authors to be cited (when their results or publications have been used) have been respected.

Lugar y fecha / Place and date:

Granada, 14 de junio de 2023

Director/es de la Tesis / *Thesis supervisor/s;* Doctorando / *Doctoral candidate*:

Firma / Signed

Firma / Signed

Firma (3): **DOMINGO MARTÍN PERANDRÉS**
En calidad de: **Firmante**

Firma (2): **GERMÁN ARROYO MORENO**
En calidad de: **Firmante**

Firma (1): **LUIS LÓPEZ ESCUDERO**
En calidad de: **Firmante**

A mi familia, por su apoyo incondicional. A mis directores, Germán y Domingo, por soportarme todos estos años y guiarme cuando estaba perdido. A mi tutor, Juan Carlos, por ofrecer soluciones a todos mis problemas y recordarme que ser pragmático es ser inteligente. A Roberto, por su indispensable ayuda al final del trayecto.

Sin vosotros no estaría aquí.

Gracias.

Agradecimientos

Quiero agradecer al Patronato de la Alhambra y Generalife, Museo de la Puebla de Don Fadrique, Museo Histórico Municipal de Écija, Centro Andalujz de Arqueología Ibérica, Stanford Computer Graphics Laboratory y Istituto di Scienza e Tecnologie dell'Informazione «Alessandro Faedo» por los modelos 3D que se han empleado en las pruebas realizadas a lo largo de esta tesis.

Resumen

Durante el siglo pasado la preservación del patrimonio histórico ha adquirido un papel esencial en el devenir de nuestra sociedad. Muchas han sido las organizaciones que se han creado para preservar y difundir estos bienes culturales, tanto materiales como inmateriales, que nos ayudan a comprender nuestro pasado.

Los trabajos relacionados con la preservación y difusión de estos bienes son bastante complejos y, a menudo, necesitan de la colaboración de varios especialistas organizados en equipos multidisciplinares. Entre las tareas que estos deben realizar, el proceso de documentación ocupa un lugar fundamental. Si la documentación resulta adecuada, se garantiza un análisis preciso de la información recabada y, en consecuencia, se podrán tomar decisiones bien fundadas en futuros trabajos de conservación.

Este proceso de documentación, sin embargo, resulta bastante laborioso. Requiere gestionar grandes conjuntos de datos heterogéneos: fotografías, bases de datos, informes, grabaciones de audio, ilustraciones, modelos 3D escaneados, etc. En función del tipo y número de aplicaciones que utilicen de manera habitual los integrantes de estos equipos de especialistas, manejar de manera eficiente estos conjuntos de datos puede convertirse en una tarea complicada. Las aplicaciones especializadas pueden proporcionar soluciones más efectivas a determinados problemas, pero un numero excesivo de ellas conduce a la fragmentación de resultados y obliga a establecer protocolos de compatibilidad para la información que se transfiere entre ellas.

Por suerte, existen soluciones software de carácter general que son capaces de manejar estos conjuntos de datos heterogéneos con solvencia. Entre ellas destacan los sistemas de información. En sus inicios estas herramientas trabajaban únicamente con información bidimensional (2D), pero hoy día son capaces de manejar contenidos en tres dimensiones como modelos 3D escaneados. Aunque son muchas las ventajas que ofrecen, al ser soluciones generales no siempre utilizan los modelos de representación de datos más apropiados o emplean eficientemente las capacidades del hardware existente.

En esta tesis doctoral se presentan un conjunto de propuestas que aprovechan las virtudes de las arquitecturas paralelas de las unidades de procesamiento gráfico (GPU)

para solventar este tipo de ineficiencias en la creación, gestión y actualización de datos en sistemas de información 3D dedicados a la documentación de patrimonio histórico.

En concreto, se detalla una nueva arquitectura de sistemas de información 3D optimizada para el procesamiento de datos en GPU. Dicha arquitectura usa el concepto de capas temáticas visto en los sistemas de información geográfica para asignar datos a la superficie del modelo, utiliza texturas 2D para almacenar la información de las capas que indexa a través de coordenadas de textura y realiza el proceso de edición de información íntegramente en la GPU. En las pruebas realizadas se han obtenido resultados que pueden llegar a ser hasta dos órdenes de magnitud más rápidos que los ofrecidos por soluciones anteriores durante la edición de información.

Para expandir la funcionalidad de esta arquitectura también se propone un algoritmo que permite almacenar la información topológica de modelos 3D en texturas 2D. El cálculo de este algoritmo se resuelve completamente en el espacio de GPU y ofrece una solución eficiente al cómputo de operaciones que requieran este tipo de información, como pueden ser los campos de distancias. Comparado con soluciones anteriores, esta propuesta es, como mínimo, dos ordenes de magnitud más rápida a la hora de calcular la topología en las pruebas que se han realizado.

Asimismo, se ha diseñado un lenguaje para poder operar entre capas de información, y se ha incluido un nuevo módulo en la arquitectura que permite traducir sentencias escritas en este lenguaje a código de shader que se ejecuta en la GPU. De esta manera se evitan transferencias innecesarias entre el espacio de CPU y el espacio de GPU que degradarían el rendimiento del sistema propuesto y todas las operaciones se resuelven íntegramente en GPU, siendo consistente con el resto de la arquitectura.

Se incluye también un estudio de usabilidad del sistema de información desarrollado que ha permitido verificar la calidad de la solución. Gracias a la colaboración de los participantes, se han identificado ineficiencias en la interfaz de usuario y se han recopilado valoraciones críticas de usuarios potenciales.

Finalmente, se describe un lenguaje de programación visual que permite documentar de forma gráfica el patrimonio histórico mediante la combinación de múltiples técnicas de visualización expresiva. Gracias a un sistema de cachés de representación y a la definición de una estructura de datos eficiente, el lenguaje permite iterar con facilidad multitud de estilos visuales que se computan en el espacio de GPU.

Índice general

Índice de figuras

Índice de tablas

1

Introducción

La preservación del patrimonio histórico que hemos heredado de civilizaciones y acontecimientos pasados ha ido adquiriendo un papel cada vez más importante en el devenir de nuestra sociedad. No en vano, este conjunto de bienes culturales, tanto materiales como inmateriales, nos ayudan a comprender nuestro pasado e informan el presente y acontecimientos futuros. Fruto de este creciente interés, se crearon a lo largo del siglo XX múltiples organizaciones a nivel nacional e internacional que, entre otros cometidos, intentan ayudar a preservar este patrimonio. Uno de sus máximos exponentes es la Organización de las Naciones Unidas para la Educación, la Ciencia y la Cultura o UNESCO. Esta organización confiere multitud de premios y reconocimientos en numerosos campos como pueden ser educación, ciencia o cultura. El título de Patrimonio de la Humanidad es uno de los más importantes y tiene como objetivo difundir y preservar sitios de gran relevancia cultural o natural a través de medidas como ayudas financieras.

Los trabajos de preservación y difusión de bienes culturales son bastante complejos y requieren la colaboración de multitud de especialistas organizados en grandes equipos multidisciplinares. Entre las tareas que estos especialistas han de completar, el proceso de documentación ocupa un lugar esencial. Una documentación adecuada nos ayuda a realizar análisis precisos de la información obtenida y, en consecuencia, nos permite tomar decisiones informadas durante trabajos futuros, entre los que se pueden encontrar aquellos relacionados con la preservación.

Sin embargo, este proceso de documentación resulta bastante laborioso pues, a menudo, implica trabajar con grandes conjuntos de datos heterogéneos. En el caso de un yacimiento, los arqueólogos podrán tomar cientos de fotografías con el fin de catalogar de forma visual las distintas piezas encontradas. Químicos podrán analizar la aleación o restos de pintura de las piezas encontradas para datarlas o determinar su composición **[NFB15]**. Aquellos con dotes artísticas crearán ilustraciones científicas que acentúen determinadas características de un número indeterminado de piezas **[RD03, INC*06]**. Asimismo, es probable que se genere un informe detallado por cada pieza encontrada y que cierta información relevante sea introducida en una base de datos general o varias con contenidos más específicos. Algunos trabajadores podrán optar por realizar diarios escritos o narrados describiendo el día a día de trabajos concretos. Si se deseara producir contenido multimedia en un futuro o realizar trabajos de medición precisos se realizarán escaneados 3D del yacimiento completo o de las piezas más relevantes **[For14]**.

Dependiendo de las herramientas software con las que estén familiarizados la mayoría del equipo, el uso eficiente de estos conjuntos de datos puede suponer un gran problema. La utilización de múltiples herramientas especializadas permite completar tareas concretas, pero introduce la necesidad de establecer una compatibilidad adecuada entre las salidas producidas por ellas. Este requisito no es trivial y, en casos de software propietario, las compañías van a priorizar sus beneficios sobre facilidades tan necesarias como éstas. Además, cuanto mayor sea el número de herramientas, mayor será la exigencia impuesta sobre los miembros del equipo para gestionar y conocer todas ellas.

Afortunadamente, existen soluciones software que ofrecen la funcionalidad necesaria para almacenar, administrar, procesar o generar nuevos datos en una única herramienta.

Tal es el caso de los sistemas de información, que son capaces de realizar estas tareas utilizando grandes volúmenes de datos. Sin embargo, dado que son soluciones generales, no siempre utilizan los modelos de representación de datos más apropiados o utilizan de forma eficiente las capacidades del hardware existente.

A lo largo de este trabajo de tesis doctoral se presentarán un conjunto de propuestas que harán un uso extensivo de las capacidades de paralelización ofrecidas por las Unidades de Procesamiento Gráfico (GPU) para solventar gran parte de los problemas mencionados en párrafos anteriores. Las propuestas aquí descritas ofrecerán soluciones tanto de carácter general como otras más especificas de determinadas tareas.

2

Objetivos

Los objetivos propuestos en esta tesis son los siguientes:

- Se detallará una arquitectura eficiente para sistemas de información que utilizan modelos 3D para representar su información. Dicho sistema usará el concepto de capas temáticas de información visto en los Sistemas de Información Geográfica (GIS) para asignar información sobre la superficie del modelo. Dichas capas almacenarán su información en texturas bidimensionales (2D), información que será indexada utilizando las coordenadas de textura apropiadas. El proceso de edición será ejecutado íntegramente en la GPU.

- Para poder realizar determinadas operaciones entre capas es necesario disponer de la información topológica del modelo 3D. Con el fin de obtener una solución consistente con la arquitectura que se pretende desarrollar, se propondrá un algoritmo capaz de almacenar dicha información topológica en texturas 2D y este se calculará íntegramente en la GPU.

- Se presentará un lenguaje para operar con capas de información que será traducido por la Unidad de Procesamiento Central (CPU) a código de shaders y será ejecutado íntegramente en la GPU. Este lenguaje está inspirado en el módulo r.mapcalc **[SW92]** de GRASS **[SWG*89]** y permitirá realizar operaciones aritméticas, lógicas y otras que podrán extraer información almacenada en bases de datos asociadas.

- Se realizará un estudio de usabilidad del prototipo de sistema de información desarrollado con el fin de identificar posibles ineficiencias en la interfaz de usuario y obtener valoraciones críticas de usuarios potenciales.

- Se describirá un lenguaje de programación visual que permitirá combinar múltiples técnicas de visualización expresiva para realizar ilustraciones científicas. Este tipo de ilustraciones ayudan a acentuar determinada información presente en los objetos mediante técnicas como el rayado o la representación de siluetas. Sin embargo, los ilustradores invierten bastante tiempo para completarlas, problema que nuestra propuesta trata de resolver en unos pocos minutos.

A continuación se detalla brevemente la estructura que seguirá este documento: el Capítulo 3 enumera trabajos destacados de sistemas de información de patrimonio histórico; el Capítulo 4 describe la nueva arquitectura para sistemas de información 3D en el área de patrimonio histórico; el Capítulo 5 desarrolla el algoritmo para almacenar información topológica de modelos 3D en texturas 2D; el Capítulo 6 describe el lenguaje de operaciones entre capas; el Capítulo 7 detalla el estudio de usabilidad realizado sobre el prototipo funcional desarrollado; el Capítulo 8 explica el lenguaje de programación visual para la documentación gráfica de objetos.

3

Trabajos relacionados

En este capítulo se clasifican trabajos previos de investigación relacionados con sistemas de información en cuatro categorías de acuerdo a la dimensión del espacio empleado en el análisis y visualización de los datos. Capítulos posteriores podrán incluir nuevas referencias a otros trabajos de interés para contextualizar mejor la labor investigadora descrita en los mismos.

3.1 Espacio 2D

Estas propuestas representan la información a través de estructuras 2D y se sirven de sistemas GIS preexistentes, a los que realizan pequeñas modificaciones, para documentar patrimonio histórico. Esta elección les permite utilizar un amplio rango de herramientas robustas con demostradas capacidades para analizar y recuperar información. Sin embargo, en estos trabajos no existe una relación bidireccional entre la información almacenada en el plano 2D y el modelo 3D original digitalizado. Por tanto, es necesario procesar dicho modelo 3D antes de usarlo como una entrada válida en estos sistemas.

Naglic **[Nag03]** y Ioannidis et al. **[IPS03]** son buenos ejemplos de este tipo de propuestas. Ambos usan sistemas GIS para sus trabajos en yacimientos arqueológicos a gran escala y, gracias a estos sistemas, pueden indexar grandes áreas de superficie utilizando sus coordenadas geográficas. Asimismo, Parkinson et al. **[PPB14]** emplea sistemas GIS para estudiar los patrones creados por las marcas de dientes de caninos contemporáneos de gran tamaño sobre los huesos de sus presas y los compara con aquellas encontradas en fósiles de épocas pasadas. Concretamente, fotografían los huesos y, manualmente, crean capas vectoriales de las marcas. En cada uno de estos casos se tiene que comprometer la naturaleza 3D del material original para utilizar sistemas GIS y analizar los datos, limitando por completo su flujo de trabajo a un único punto de vista.

3.2 Espacio 2.5D

Estas propuestas también trabajan con imágenes y sistemas GIS para analizar y procesar la información. A diferencia de las propuestas 2D, las imágenes son modelos de elevación rasterizados que contienen información de altura. Por tanto, trabajan dentro del espacio 3D restringido que este tipo de imágenes proporcionan. Aunque se trata de métodos más flexibles e interesantes que las propuestas estrictamente bidimensionales, comparten los problemas que presentan éstas últimas. Necesitan convertir los modelos 3D digitalizados a un formato de imagen apropiado y se limitan a utilizar un único punto de vista por trabajo.

Benito et al. **[BCCA*15]** utiliza esta metodología para clasificar las herramientas de piedra utilizadas por chimpancés salvajes. Dicha clasificación se realiza en múltiples etapas. En primer lugar, escanean las herramientas. A continuación, convierten los modelos 3D resultantes a modelos de elevación y, finalmente, usan funciones de clasificación GIS morfométricas para discriminar entre golpeadores activos y pasivos en ensamblajes líticos.

3.3 Espacio híbrido

Estas propuestas trabajan en un espacio 2D durante las fases de análisis y procesamiento de datos pero, al contrario que las propuestas anteriores, visualizan sus resultados en un espacio 3D. Son necesarias, por tanto, dos etapas en las cuales se transforma la información dentro del flujo de trabajo de estos métodos. En la primera necesitan proyectar el modelo 3D original, utilizado como referencia geométrica, sobre diferentes planos 2D. A continuación, las imágenes resultantes o modelos de elevación digitales son procesados por sistemas GIS. Una vez que se concluye la etapa de procesamiento, la salida necesita proyectarse sobre el modelo 3D como una textura. Aunque estos métodos carecen de las restricciones de puntos de vista descritas en las propuestas anteriores, su flujo de trabajo es más complejo y sólo apropiado para casos en los cuales el patrimonio histórico se puede dividir fácilmente en planos 2D.

Campanaro et al. [CLDT16] usa esta metodología para ocuparse de la preservación de estructuras arquitectónicas. Proyectan las fachadas de los edificios en múltiples imágenes, las procesan utilizando una variedad de sistemas GIS y, finalmente, vuelven a proyectar los resultados sobre versiones simplificadas de los modelos 3D originales de los edificios para su visualización.

3.4 Espacio 3D

A diferencia de las propuestas anteriores, estos métodos trabajan directamente con los modelos 3D digitalizados. No existen, por tanto, restricciones de puntos de vista ni transformaciones entre los distintos espacios de trabajo.

3.4.1 Sistemas de anotación

El principal objetivo de estos sistemas es asociar información sobre secciones específicas de la superficie del modelo 3D y ofrecer robustas herramientas de recuperación de información. Existe un amplio abanico de mecanismos de indexación en este paradigma de trabajo, desde segmentos de la malla del modelo 3D a líneas o puntos colocados sobre su superficie.

Durand et al. [DDM*06], Meyer et al. [MGP*07] y Mateos et al. [MRVMRÁ*16] proponen múltiples sistemas de información online para documentar lugares de alto interés arqueológico. Estos sistemas requieren que el modelo 3D original sea segmentado en entidades más pequeñas y específicas. El análisis, procesamiento y diseminación de la información se realizan sobre cada una de estas subentidades en lugar del modelo original.

Giunta et al. [GDPCM05] usa modelos AutoCAD para estructurar la información arquitectónica y los resultados de investigación de la fachada de la Catedral de Milán.

Además, el sistema permite al usuario insertar imágenes, textos y documentos de forma georeferenciada.

Serna et al. [SSD*11] describen un sistema de información distribuido para añadir información a objectos multimedia (modelos 3D, imágenes, texto) usando el concepto de áreas. Apollonio et al. [ABC*18] desarrollan un sistema de información web para documentar el proyecto de restauración de la Fuente de Neptuno en Bolonia, Italia. Dicho sistema permite al usuario añadir información al modelo 3D usando tres diferentes primitivas (punto, polilínea y área) y recuperar la información almacenada haciendo uso de un conjunto amplio de herramientas robustas.

3.4.2 Sistemas basados en capas

Estos sistemas estructuran la información asociada a modelos 3D mediante un conjunto de capas, donde cada una de ellas almacena el valor de un atributo. Las capas de datos puede considerarse como funciones que asocian un valor de propiedad a los puntos de la superficie del objeto. Este tipo de sistemas incluye la misma funcionalidad que la ofrecida por los sistemas de anotación pero, además, pueden realizar operaciones complejas entre las capas de datos. Sin embargo, el principal problema que estos sistemas han de solventar es el diseño de estructuras y métodos eficientes para representar dichas capas de datos.

Torres et al. [TCM*10] dividen la superficie del modelo 3D original en celdas haciendo uso de un octree. Concretamente, la superficie se divide de forma recursiva utilizando ocho celdas cúbicas o vóxeles del mismo tamaño hasta llegar a un nivel de resolución predefinido. Por tanto, cada celda almacena los triángulos de la malla que interseca. El nivel de detalle depende del tamaño de estos vóxeles y, por consiguiente, del número de subdivisiones aplicadas. La estructura octree permite al usuario trabajar independientemente de la resolución e irregularidades geométricas de la malla. De esta forma, mallas simples pueden almacenar información mas precisa que su geometría es capaz de ofrecer. Sin embargo, esta indexación espacial requiere bastante recursos de memoria y un alto coste de rendimiento cuando trabaja con niveles de resolución elevados.

Las capas de información se almacenan como secuencias de propiedades asignadas a los nodos hoja que intersecan con la superficie [TLR*13, STLL13]. Este sistema no sólo permite añadir o recuperar información, también es posible realizar cálculos complejos entre capas heterogéneas. Algunos de estos cálculos incluyen consultas a bases de datos, operaciones topológicas, geométricas, aritméticas o lógicas. Dichas operaciones se pueden utilizar para analizar resultados existentes o producir nuevos contenidos. Soler et al. [SML17] intentan mejorar este sistema solventando ciertos problemas topológicos presentes pero, para conseguirlo, requieren el uso de estructuras de datos más complejas.

4

Arquitectura de sistemas de información 3D optimizada para GPUs

A lo largo de este capítulo se detalla una nueva arquitectura para sistemas de información 3D basados en capas que busca solventar carencias presentes en los trabajos desarrollados dentro de esta categoría. Al contrario que la propuesta de Torres et al. [TCM*10] que utiliza octrees y resuelve sus operaciones mediante algoritmos de CPU, esta solución se ha diseñado para aprovechar el paralelismo y eficiencia de las arquitecturas modernas de GPUs. La información se almacena en texturas 2D y se indexa utilizando las coordenadas de textura de los modelos 3D. Las capas residen en memoria de GPU siempre y cuando haya memoria disponible y todas las operaciones se resuelven en la GPU. En consecuencia, las transferencias entre CPU y GPU son inexistentes y la edición de capas es realmente eficiente. Aspectos como el tamaño del área editada no influyen en el rendimiento. Asimismo, el almacenamiento de esta información en dispositivos de almacenamiento secundario se realiza en formatos eficientes que minimizan la necesidad de realizar procesamiento alguno para transferirla a memoria de GPU.

Con el fin de poder evaluar la calidad de la propuesta se ha desarrollado un prototipo funcional que implementa esta arquitectura y que, entre sus múltiples funcionalidades, dispone de una interfaz gráfica para añadir información a las capas de forma intuitiva. La Figura 4.1 muestra una captura de ejecución del prototipo donde se trabaja sobre una vasija escaneada que ha sido reconstruida a partir de los fragmentos que se conservan.

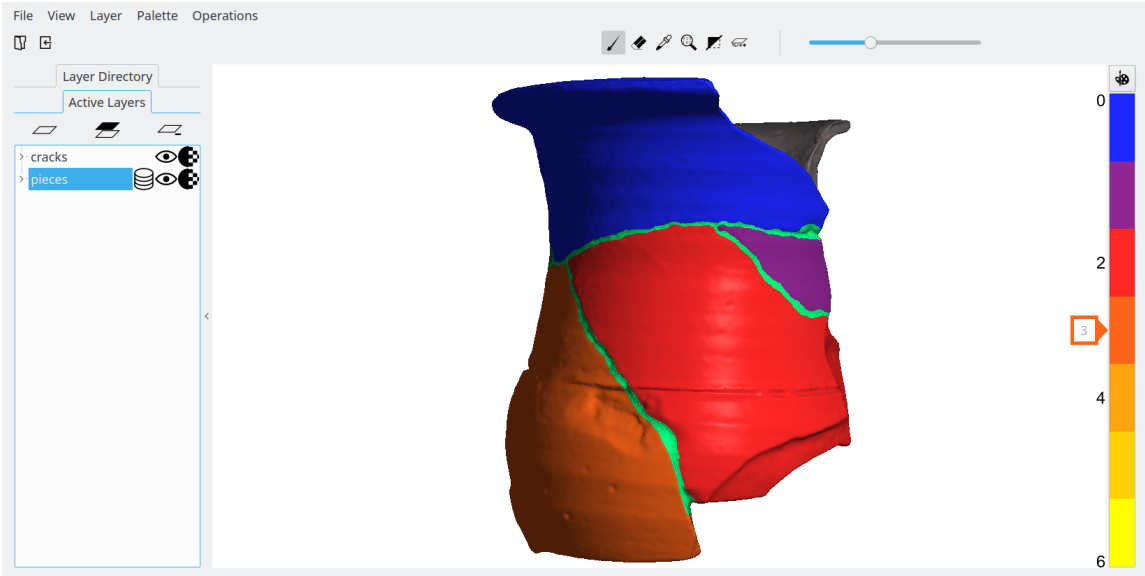


Figura 4.1 Prototipo funcional que implementa la arquitectura propuesta. El usuario ha asociado dos capas de información al modelo 3D de una vasija. La primera capa *cracks* identifica el área cubierta por la grietas. La segunda capa *pieces* identifica las distintas piezas de la vasija utilizando la paleta seleccionada

4.1 Visión general de la arquitectura

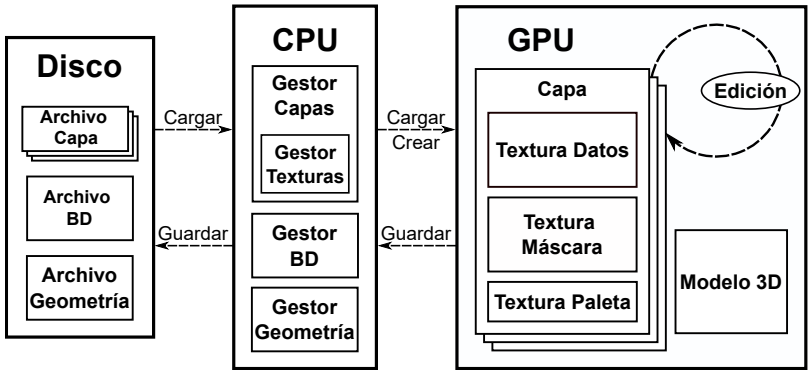


Figura 4.2 Visión general de la arquitectura propuesta

Idealmente, siempre y cuando las necesidades de la solución lo permitan, las estructuras y los algoritmos deberían diseñarse teniendo en cuenta las facilidades ofrecidas por la arquitectura hardware de las GPUs. Este principio ha sido el eje central empleado a la hora de construir la arquitectura software para sistemas de información que aparece descrita en la Figura 4.2 de forma esquemática. Como se puede observar, está organizada en tres espacios: almacenamiento secundario o disco, CPU y GPU. A continuación se ofrece una explicación de las funciones cubiertas por las dos primeras partes y a lo largo del resto del capítulo se explica en detalle qué resuelve la parte de GPU, donde se encuentran las innovaciones más importantes de este trabajo.

4.1.1 Almacenamiento secundario

Los dispositivos de almacenamiento secundario se utilizan principalmente para almacenar de forma permanente los cambios realizados sobre un proyecto del sistema. Análogamente, también se emplean para recuperar esa información del sistema de archivos. Las cargas se pueden realizar de un proyecto completo o bien de otros elementos más específicos. En este último caso se incluyen capas que no se encuentran activas en el proyecto de trabajo actual y con las que se desea trabajar.

Como se puede apreciar en la sección izquierda de la Figura 4.2, los proyectos están estructurados en tres tipos de archivos:

- **Archivo de Geometría**: Contiene un volcado secuencial de los buffers que almacenan los atributos de la malla 3D: posición de los vértices, coordenadas de textura, normales por vértice y índices de primitivas o triángulos. Esta disposición permite una carga directa en memoria a expensas de un pequeño costo adicional en espacio.

- **Archivo de Base de Datos**: Contiene la descripción de la base de datos relacional y las tablas que alberga están asociadas a capas concretas de un determinado tipo. En la implementación actual, el sistema de gestión de base de datos utilizado es SQLite y, por tanto, la gestión y mantenimiento de este archivo se realiza a través de su API.

- **Archivos de Capas**: Contienen un volcado secuencial de la información incluida en las capas. Como se detalla en secciones más avanzadas del capítulo, esta información procede de texturas que residen en la memoria de GPU.

4.1.2 Espacio de CPU

Incluye el conjunto de entidades gestoras que se describen a continuación:

- **Gestor de capas**: se ocupa de las operaciones de creación, destrucción, almacenamiento y carga de capas. Dispone de un módulo especializado, el **Gestor de Texturas**, que emite órdenes a la GPU para crear y destruir texturas o transferir la información que contienen. Se ha de notar que no almacena ningún tipo de información contenida en las capas ya que esta se encuentra en las texturas, que residen en la memoria de GPU. Las operaciones de transferencia tienen lugar cuando se salvan capas a disco o se cargan en el sistema. En este último caso se produce la creación implícita de las texturas que albergarán su información.

- **Gestor de base de datos**: se encarga de resolver todas las operaciones realizadas con las tablas que están asociadas a un tipo específico de capa soportado por el sistema. Entre ellas se encuentran la creación de las propias tablas, la inserción y actualización de registros de la tabla así como cualquier tipo de consulta SQL que se realice para recuperar información de interés.

- **Gestor de geometría**: crea el Archivo de Geometría cuando se importa un modelo 3D en el proyecto actual de trabajo y transfiere la información geométrica a memoria de GPU para que pueda ser representada. En caso de abrir un proyecto que cuenta ya con un Archivo de Geometría, este gestor lee su contenido y, a continuación, lo transfiere a memoria de GPU. En ambos casos, dado que el sistema no modifica la geometría de los modelos, sólo es necesario realizar una única transferencia por importación de modelo o sesión de trabajo con el proyecto.

4.2 Espacio de GPU

4.2.1 Texturas como estructuras de datos heterogéneos

En Informática Gráfica las texturas se suelen definir como contenedores de una o múltiples imágenes. Dichas imágenes están compuestas por un conjunto de téxeles organizados en forma de arrays de una determinada dimensionalidad (1D, 2D o 3D). Un gradiente de blanco a negro podría constituir un buen ejemplo de imagen 1D; una captura del escritorio de nuestro sistema operativo, de una imagen 2D y el conjunto de secciones 2D generado por una tomografía de rayos X, de una imagen 3D.

Asimismo, la información que almacenan las texturas en estos téxeles sigue formatos bastante específicos. En concreto, cada téxel puede contener de uno a cuatro canales y, a su vez, cada canal puede almacenar hasta 32 bits. Por ejemplo, el formato *RGB32F* dispone de tres canales (rojo, verde y azul) y cada uno de ellos almacena un número real de 32 bits.

Esta información es indexada mediante coordenadas de textura. Dichas coordenadas se obtienen al proyectar todas las primitivas de una malla 3D sobre el espacio de textura, que normalmente suele ser bidimensional. Se corresponden, por tanto, con la posición de los vértices de estas primitivas, habitualmente triángulos, en el espacio de textura.

Históricamente se han usado para añadir información de color a modelos 3D, permitiendo niveles de detalle elevados en casos donde la geometría de la malla no es demasiado compleja. Sin embargo, la introducción del cauce programable y posterior aparición del cauce de computación en GPUs han expandido dramáticamente sus aplicaciones durante las dos últimas décadas.

Por tanto, las texturas son estructuras multipropósito bastante versátiles hoy en día. En campos como la Programación de Propósito General para GPUs (GPGPU) se consideran simples arrays o matrices de computación. En cambio, en Informática Gráfica representan el estándar de facto para trabajar con las propiedades de color de modelos 3D y se utilizan en conjunción con las coordenadas de textura para acceder a sus datos y operar sobre ellas.

Dado que los sistemas de información para los que se está definiendo esta nueva arquitectura utilizan modelos 3D, tiene sentido que se utilicen coordenadas de textura para indexar y almacenar la información de cada capa en texturas 2D. Esta información se representa utilizando números enteros o reales de un determinado tamaño dependiendo del tipo de capa empleada.

4.2.2 Capas de información

La arquitectura es capaz de manejar dos tipos diferenciados de capas para el sistema de información:

- **Capas numéricas** almacenan propiedades numéricas del modelo 3D como puede ser la curvatura, rugosidad, un campo de distancias o cualquier otro tipo de valor que se considere relevante. Pueden albergar números enteros y números reales. Normalmente se suelen utilizar para asociar y operar con información cuantitativa.

- **Capas de bases de datos** son capas numéricas que tienen asociada una tabla de la base de datos relacional del proyecto. El usuario puede definir esta tabla empleando una combinación de cualquiera de los tipos de campos disponibles. Entre las opciones posibles se encuentran una selección de tipos de datos estándar de bases de datos relacionales como pueden ser texto, enteros, reales o fecha. Además, existen otros creados expresamente para este sistema de información como recursos externos. Independientemente de como el usuario haya definido la tabla, la llave primaria es siempre un entero positivo y este es el valor que en realidad se almacena en la capa. Gracias a la llave primaria se puede establecer una relación bidireccional entre una tabla de la base de datos y un modelo 3D. A través de esta relación obtenemos una solución elegante que permite asociar datos heterogéneos complejos (imágenes, documentos, vídeos, fechas, texto, etc.) a modelos 3D y, por consiguiente, enriquecer la información semántica asociada. Normalmente este tipo de capas se utilizan para asociar y operar con información cualitativa.

En ambos casos se almacenan valores numéricos que, por si solos, carecen de una representación visual. Por tanto, se necesita una función que nos permita asociar un color a cada valor almacenado y nos ayude a comprenderlo desde un punto de vista visual. Una paleta de colores asociada a cada capa es precisamente lo que necesitamos y, en el sistema propuesto, se describen como una sucesión ordenada de puntos de control $[pc_1, pc_2, pc_3, \ldots, pc_n]$. Cada punto de control es un par $\{n, c\}$ que relaciona un valor numérico n a un color c descrito por cuatro canales (rojo, verde, azul y opacidad). Los puntos de control se ordenan en función de su valor numérico n. Asimismo, los valores incluidos en el rango descrito por dos puntos de control consecutivos, por ejemplo $[pc_1, pc_2]$, pueden tener asociados colores distintos dependiendo de si el proceso de interpolación se encuentra activo en dicha paleta. En caso de que no lo esté, el color asociado al punto de control de menor valor, pc_1, se aplica a todos los valores del rango $[pc_1, pc_2)$. Por contra, si está activo, se aplicará una interpolación lineal entre los colores de pc_1 y pc_2 para determinar el color asociado a un valor incluido en el rango. La Figura 4.3 ofrece un ejemplo de ambos casos para una misma paleta. En la sección izquierda se puede observar cómo los colores son constantes entre puntos de control mientras que la sección derecha muestra los efectos de la interpolación entre los mismos.

Figura 4.3 La parte izquierda de la figura muestra una paleta que no tiene activa la interpolación entre puntos de control. En la parte derecha, se muestra la paleta con la interpolación activada

4.2.2.1 Implementación de capas de información

La arquitectura propuesta implementa los dos tipos de capa siguiendo el esquema mostrado en la Figura 4.2. Concretamente, las capas de información están compuestas por dos texturas 2D y una textura 1D:

- **Textura de Datos** (2D): almacena un número por téxel. Los números pueden ser enteros de 8, 16 y 32 bits o reales de 16 y 32 bits. Esta textura es la que almacena realmente la información asociada a la capa. En el caso de las capas numéricas, los valores de la propiedad descrita; mientras que en el caso de las capas de bases de datos, las llaves primarias de los registros de la tabla asociada.

- **Textura de Máscara** (2D): almacena un valor booleano por téxel que determina si el valor contenido en Textura de Datos tiene validez.

- **Textura de Paleta** (1D): almacena la información de color por téxel necesaria para visualizar los contenidos de Textura de Datos. Cada téxel contiene un vector de cuatro componentes que describe el siguiente formato de color: rojo, verde, azul y opacidad.

La Textura de Paleta implementa la información de color que describe la paleta de la capa a través de su conjunto de puntos de control. Utilizando los limites inferior y superior de la paleta y una transformación lineal, el sistema puede indexar esta textura 1D y recuperar el color apropiado para visualizar la **Textura de Datos**.

4.2.2.2 Configuración y gestión de texturas de capas

Esta variedad de opciones en los tipos de texturas que pueden emplear las capas puede constituir un problema cuando se necesitan numerosas instancias de cada opción de forma simultánea. El Gestor de Texturas logra resolver este posible contratiempo mediante el uso de *texture arrays* **[The08]** y *sparse bindless textures* **[The13b, EFMS14, The13a]**.

Tradicionalmente las GPUs sólo podían usar de forma simultánea tantas texturas como unidades de imágenes de texturas estuvieran disponibles en cada etapa del cauce programable. Por ejemplo, en tarjetas gráficas modernas de NVIDIA, los fragment shaders sólo tienen acceso a 32 texturas distintas. Para evitar este límite, el Gestor de Texturas implementa texturas *bindless*. Esta funcionalidad permite al sistema utilizar texturas en shaders a través de un manejador sin necesidad de asociar cada una de ellas a una unidad de imagen de textura.

Asimismo, cuando existen múltiples instancias de un mismo tipo de textura, resulta ineficiente utilizar manejadores individuales para cada una de ellas puesto que es posible

agruparlas en categorías utilizando un subconjunto de sus propiedades. Para lograr esta categorización, el Gestor de Texturas las clasifica empleando una función hash durante el proceso de creación de las mismas. La clave hash, que se almacena en un entero de 64 bits, se construye mediante la unión de los valores de anchura, altura y tipo de dato. De esta manera, el gestor es capaz de agrupar multitud de texturas que comparten características similares y las almacena en *texture arrays*. Estas estructuras están organizadas en múltiples niveles y en cada uno de ellos pueden almacenar una textura, resultando especialmente útiles en situaciones en las que es posible realizar agrupaciones. Al mismo tiempo, sólo requieren de un manejador para identificarlas, por lo que cuanto mayor sea el número de texturas que pueden albergar, menor será el número de manejadores necesarios.

Cuando el sistema necesita una nueva textura, habitualmente por la creación de una capa, el Gestor de Texturas calcula su clave hash. En caso de que la clave no exista, el gestor solicita a la GPU la creación de un nuevo texture array. A continuación, almacena el manejador asociado a dicho *texture array* y asigna la textura requerida a su primer nivel. Por contra, si la clave hash ya existe, la nueva textura puede asociarse al nivel más bajo que se encuentre desocupado de un *texture array* preexistente.

Crear multiples *texture arrays* con una gran cantidad de niveles puede saturar rápidamente la memoria de GPU disponible. Incluso si sólo necesitamos una pequeña cantidad de texturas para cada texture array, el driver gráfico necesita reservar todo el espacio necesario durante el proceso de creación. Decrementar el número de niveles por *texture array* no soluciona realmente el problema porque continúa existiendo una sección de memoria reservada que no se está usando. Aquí es donde las texturas sparse o parcialmente residentes en memoria ofrecen una solución excelente a esta problemática. Mediante su uso, el Gestor de Texturas puede reservar espacio de memoria virtual para los *texture arrays* sin consumir espacio físico hasta que es estrictamente necesario. Sólo cuando el gestor necesita crear nuevas texturas, este espacio de memoria virtual se adjudica y se asigna en memoria física.

Por consiguiente, mediante el uso de *texture arrays* y *sparse bindless textures* se consigue una gestión de memoria más simple y flexible, consiguiendo que no haya memoria física de GPU desaprovechada. Se pueden crear una mayor cantidad de *texture arrays* con más niveles para cada uno de ellos y, en consecuencia, pueden existir más instancias de diferentes tipos de capas de información de forma simultánea. Estas texturas parcialmente residentes en memoria también mejoran el rendimiento porque no existen transferencias de datos entre CPU y GPU hasta que la memoria física de la GPU está casi completamente ocupada. Dicha funcionalidad también ofrece otra ventaja: las texturas casi siempre residen en la memoria de la GPU con la excepción de dos casos concretos. En el primero el usuario desea guardar la capa a disco y, por tanto, las tres texturas de la capa (Datos, Mascara y Paleta) son copiadas de GPU a CPU. En el segundo no hay suficiente espacio en memoria de GPU para almacenar una nueva capa y el driver gráfico transfiere aquellas texturas menos frecuentemente usadas a memoria de CPU, liberando de esta manera el espacio necesario para aquellas que pretende crear.

Esta solución para gestionar las texturas de las capas da lugar a una propuesta eficiente gracias a que el rendimiento de las GPUs es excepcional cuando trabajan con texturas y coordenadas de texturas. Las arquitecturas de las GPUs actuales disponen de hardware especializado como cachés de texturas y unidades de mapeado de texturas que optimizan las operaciones realizadas con estas estructuras **[HG97, WPSAM10, Dog12]**. La caché explota la localidad espacial y temporal en los accesos para alcanzar ratios de éxito altos. Asimismo, el procesador de shader puede acceder a los datos de textura de forma simultanea en grandes cantidades y empleando operaciones de lectura con latencias bajas. Otra ventaja de esta propuesta es el establecimiento de una independencia entre la cantidad de información asignada a la superficie de una malla 3D a través de texturas y la densidad de la geometría que la conforma. Con esta solución, texturas de alta resolución que son capaces de almacenar una cantidad de información muy elevada pueden asignarse a mallas 3D muy simples.

4.2.3 Edición de capas sobre modelos 3D

Tras activar el modo de edición en el sistema, una herramienta con forma de círculo aparece sobre el cursor del ratón. El usuario puede añadir información a la capa seleccionada presionando el botón izquierdo y arrastrando la herramienta sobre el área deseada. Este es un proceso complejo que involucra a dos algoritmos:

- Algoritmo de Edición de Texturas (AET), descrito en la Subsubsección 4.2.3.1, traduce las posiciones de pantalla donde el usuario pulsa el botón del raton a coordenadas de texturas válidas y usa dichas coordenadas para almacenar el dato seleccionado en los téxeles correspondientes de las texturas de la capa.

- Algoritmo de Relleno de Texturas (ART), descrito en la Subsubsección 4.2.3.2, elimina mediante un proceso de expansión los defectos visuales que podrían aparecer cuando la aplicación representa la capa sobre la superficie de la malla 3D.

4.2.3.1 Algoritmo de Edición de Texturas

AET es un algoritmo compuesto de múltiples pasos que proyecta el área de la herramienta de edición, expresada en coordenadas de ventana, sobre el espacio de textura y almacena el valor seleccionado en las texturas de capa correspondientes de forma persistente. La Figura 4.4 representa un diagrama de todo este proceso en el cual se van descartando de forma progresiva regiones de la textura hasta que el área editada coincide con la forma naranja de la herramienta de edición. Las entradas de cada paso se muestran en la parte izquierda, mientras que los resultados parciales en 3D lo hacen en la parte derecha. La zona central describe como la textura de la capa se va modificando en cada uno de los diferentes pasos.

Este algoritmo necesita las siguientes entradas para poder ejecutarse:

- **Atributos de la malla 3D**: concretamente hace uso de la posición de los vértices, las normales asociadas a dichos vértices y sus coordenadas de textura.

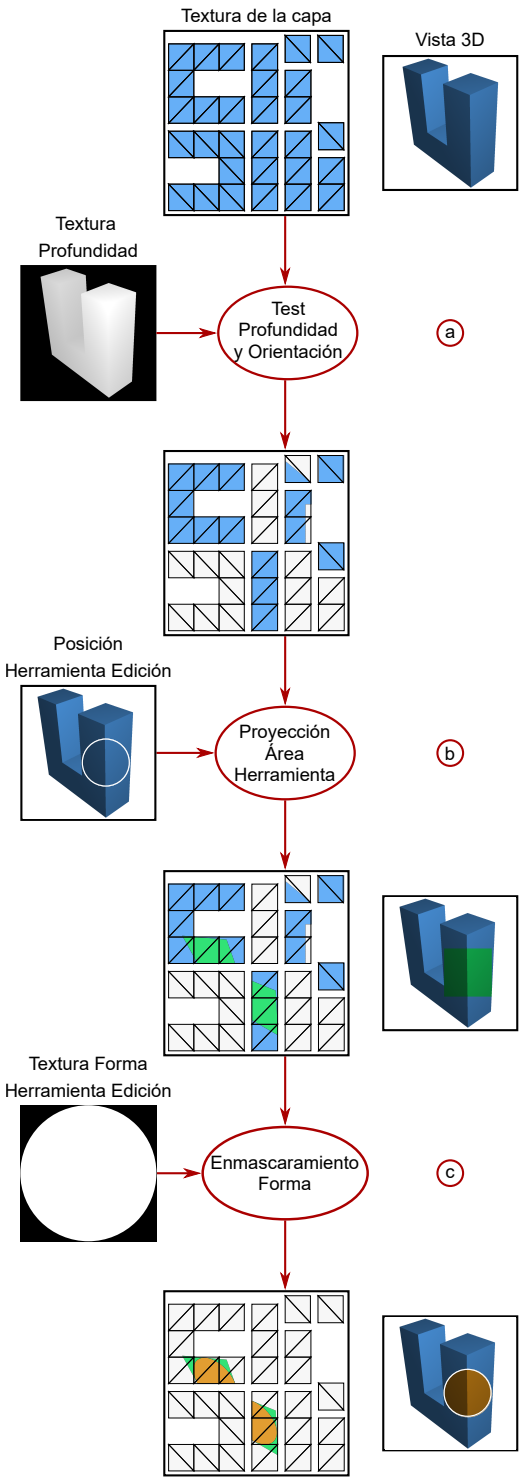


Figura 4.4 AET filtra las áreas de la textura de forma progresiva. (a) El Test de Orientación y Profundidad descarta las secciones ocluidas y las caras traseras del modelo 3D. (b) Proyección del Área de la Herramienta proyecta un cuadrado del tamaño adecuado sobre el área restante de la textura. Finalmente, (c) Enmascaramiento de Forma descarta los téxeles que se encuentran fuera de la forma de la herramienta de edición

- **Información de la cámara**; contenida en las matrices de modelo, vista y proyección.
- **Textura de Profundidad**: se crea como parte del cauce de representación del sistema. Esta textura almacena los valores de profundidad de la escena representada desde el punto de vista actual. Se actualiza cada vez que el usuario modifica el punto de vista de la cámara actual o cuando se carga un nuevo modelo 3D en el sistema.
- **Textura con la Forma de la Herramienta de Edición**: se crea como parte del cauce de representación del sistema. Esta textura almacena la forma de la herramienta de edición actual, que normalmente se corresponde con un círculo. Se actualiza cada vez que el usuario selecciona una forma distinta para la herramienta.

Dado que el objetivo que persigue el algoritmo es la edición de texturas de capa, no debe modificar el framebuffer asociado al widget para evitar interferencias con la visualización. Por tanto, requiere la creación de otro framebuffer especializado al cual se asignan tres texturas: las texturas 2D de la capa, Textura de Datos y Máscara, y una máscara temporal, Máscara del Área Editada.

Cada operación de edición actualiza estas texturas de la siguiente manera:

- Textura de Datos se actualiza con el valor seleccionado durante el proceso de edición en el área marcada por el usuario.
- Textura de Máscara se actualiza con el valor *true* en el área marcada por el usuario.
- Máscara del Área Editada también se actualiza con el valor *true* pero el sistema limpia la textura después de cada operación. Por tanto, se corresponde únicamente con el área marcada por la operación de edición actual.

Haciendo uso de una cámara ortogonal, AET representa una escena en la cual se emplean las coordenadas de textura del modelo 3D como posiciones de los vértices. De esta forma, la parametrización de la malla se representa en un espacio bidimensional o espacio de textura. La Figura 4.4 muestra la parametrización de un modelo 3D con forma de *u* en la parte superior de la columna central. Las líneas negras representan los triángulos 2D creados por las coordenadas de textura y, el color azul, el área en la cual se han realizado estas representaciones dentro de las texturas.

AET usa de forma extensiva el mapeado de texturas proyectivo **[SKVW*92, Eve01]**. Esta técnica proyecta texturas sobre una escena de forma similar a como lo haría un proyector de diapositivas situado en una posición distinta a la cámara. En términos de implementación, se trata de un procedimiento similar al de proyectar la escena en pantalla, pero aquí la proyección se realiza en el proyector de diapositivas. Concretamente, los vértices del modelo 3D se transforman usando las matrices de proyección, modelo y vista del proyector de diapositivas. Esta transformación mapea las coordenadas homogéneas

de dichos vértices a coordenadas de corte:

$$\begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{bmatrix} = M_{projection} \cdot M_{viewing} \cdot \begin{bmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ w_{obj} \end{bmatrix}$$

donde $(x_{obj}, y_{obj}, z_{obj}, w_{obj})$ y $(x_{clip}, y_{clip}, z_{clip}, w_{clip})$ son los vértices y coordenadas de corte, $M_{projection}$ y $M_{viewing}$ son las matrices de proyección y modelo-vista. Se aplica una transformación de escala y un sesgo para trasladar el dominio de las coordenadas de corte $[-1, 1]$ al dominio del mapeado de texturas $[0, 1]$:

$$\begin{bmatrix} x_{prj} \\ y_{prj} \\ z_{prj} \\ w_{prj} \end{bmatrix} = M_{sb} \cdot \begin{bmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{bmatrix}$$

donde $(x_{clip}, y_{clip}, z_{clip}, w_{clip})$ y $(x_{prj}, y_{prj}, z_{prj}, w_{prj})$ son las coordenadas de corte y proyectivas, M_{sb} es la matriz de escala-sesgo.

Al contrario que el mapeado de texturas estándar que usa coordenadas de textura reales (s, t), el mapeado de texturas proyectivo utiliza coordenadas homogéneas o coordenadas en el espacio proyectivo (s, t, w) donde $s = x_{prj}$, $t = y_{prj}$ y $w = w_{prj}$. Estas coordenadas se interpolan sobre la primitiva y, a continuación, en cada fragmento. Finalmente, las coordenadas homogéneas interpoladas se proyectan al espacio de textura real $(s/w, t/w)$ para acceder a la imagen de textura. El valor muestreado es la salida del mapeado de textura proyectivo para cada fragmento.

Aunque AET se completa en una única pasada, se explica su funcionamiento en tres pasos conceptuales distintos para facilitar su comprensión:

(a) **Test de Orientación y Profundidad:** Dado que la escena se representa en un plano 2D usando las coordenadas de textura, AET no puede usar el buffer de profundidad empleado por la API para aplicar el test de profundidad. Esta circunstancia podría producir situaciones indeseadas en las que se editen áreas ocluidas de los modelos 3D. Para solventar este problema, el algoritmo hace uso de la Textura de Profundidad facilitada por el cauce del sistema, la cual almacena los valores de la escena 3D representada desde el punto de vista de la cámara actual.
El proyector comparte la misma información de proyección y vista de la cámara de la escena 3D. Las posiciones de los vértices del modelo 3D son transformadas e interpoladas sobre las primitivas que, en este caso, son los triángulos representados por las coordenadas de textura. Por tanto, cada fragmento dispone del valor de profundidad de la escena 3D, z_{prj}/w_{prj}, y esta profundidad se compara con el valor de la Textura de Profundidad muestreada por las coordenadas $(s/w, t/w)$ utilizando una interpolación del vecino más próximo. Una pequeña corrección se aplica a este cálculo para combatir los posibles errores de precisión. Si la profundidad del

fragmento se queda por detrás del valor de profundidad de la textura, se descarta el fragmento. En cualquier otro caso, el resto de algoritmo continua con normalidad.
Además de caras ocluidas, tampoco se deben poder editar caras traseras. Para testar su orientación, el algoritmo calcula el producto escalar entre el vector de la cámara y las normales de la malla. Si el resultado es inferior a cero, nos encontramos ante una cara trasera y el fragmento es descartado. En cualquier otro caso, el algoritmo continúa con el siguiente paso.
La Figura 4.4.a muestra qué secciones de la parametrización de la malla, representadas en azul, son descartadas después de aplicar el Test de Orientación y Profundidad. La regiones descartadas se representan de blanco.

(b) **Proyección del Área de la Herramienta:** En este paso el proyector comparte la misma posición de mundo que la cámara de la escena y, por tanto, su transformación de vista es también idéntica. Sin embargo, su transformación de proyección necesita ajustarse al frustum de la textura. Si la transformación de proyección fuera idéntica, la textura se proyectaría sobre toda la pantalla. Por consiguiente, la proyección tiene que limitarse al área ocupada por la herramienta de edición. La siguiente ecuación describe esta transformación 2D:

$$T_c = T_f + S_f \cdot S_c$$

donde T_c es la coordenada destino, T_f es el factor de traslación, S_f es el factor de escala y S_c es la coordenada origen. Los factores de escala y traslación se calculan con las siguientes ecuaciones:

$$S_f = \left(\frac{W_w}{2T_w}, \frac{W_h}{2T_h} \right)$$
$$T_f = \left(\frac{T_{px} - 0{,}5 \cdot W_w}{T_w}, \frac{T_{py} - 0{,}5 \cdot W_h}{T_h} \right)$$

donde S_f and T_f son los factores de escala y traslación respectivamente, W_w y W_h son las dimensiones de la ventana (anchura y altura), T_w y T_h son las dimensiones de la textura (anchura y altura), T_{px} y T_{py} son las coordenadas horizontales y verticales de la herramienta de edición.
La Figura 4.4.b muestra la proyección del área cuadrada que contiene la forma de la herramienta en el espacio de textura haciendo uso de su posición y tamaño como datos de entrada. La proyección sólo tiene lugar sobre las coordenadas que todavía no han sido descartadas por los tests anteriores. Dichas coordenadas están representadas por el área azul. Asimismo, la salida de este paso se corresponde con el área verde.

(c) **Enmascaramiento de Forma:** Llegados a este punto, se ha de comprobar que las coordenadas no descartadas pertenecen al área definida por la forma de la herramienta. Para esta comprobación se utiliza la Textura con la Forma de la Herramienta de

Edición y se muestrea usando la interpolación del vecino más próximo. Esta simple operación de enmascaramiento descarta aquellas coordenadas que caen fuera del área de interés. Las coordenadas finales proyectan la forma correcta sobre el modelo 3D y determinan qué áreas de las texturas de la capa necesitan ser actualizadas.
La Figura 4.4.c muestra la operación de enmascaramiento usando un círculo como entrada. Este paso sólo se realiza sobre aquellas coordenadas que ya han sido validadas por la proyección anterior y que están representadas por la zona verde de la textura. Asimismo, la salida de este algoritmo se corresponde con el área redondeada de color naranja.

4.2.3.2 *Algoritmo de relleno de texturas*

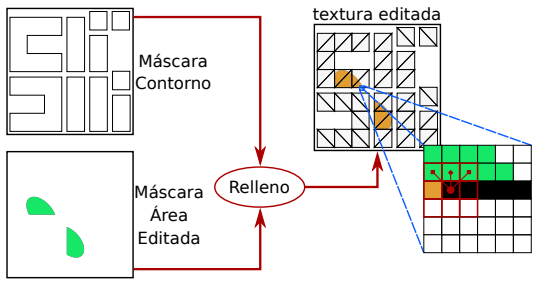


Figura 4.5 En el área ampliada, el algoritmo ha verificado que el téxel actual forma parte de la Máscara de Contorno (línea negra) y el kernel (matriz roja) comprueba si los téxeles cercanos pertenecen a la Máscara del Área Editada (forma verde). En caso afirmativo, almacena el valor de edición seleccionado en las texturas (color naranja)

Mientras que EAT solventa la edición de datos en texturas, el uso de este tipo de estructuras puede producir pequeños problemas de visualización si no se realiza ningún tipo de procesamiento adicional. Cuando las GPUs muestran modelos 3D texturados, las coordenadas de textura definen cómo se muestrean los téxeles y estas normalmente están organizadas en conjuntos de islas de diferentes tamaños y formas a lo largo de un plano 2D. Este espacio de coordenadas de textura es continuo, mientras que las texturas o imágenes utilizadas son estructuras discretas. Por consiguiente, esta disparidad hace que la conversión entre ambos espacios sea propensa a producir pequeñas inconsistencias en torno a las fronteras de las islas. Si no hay ningún tipo de información redundante alrededor de las fronteras, se producirán pequeños defectos visuales cuando estas regiones son muestreadas. Para evitar este problema en la representación de las capas, ART expande los datos presentes en las fronteras a sus téxeles vecinos. La Figura 4.5 muestra una visión general de este proceso, donde las entradas se encuentran en la sección izquierda y las salidas, en la sección derecha. El área ampliada muestra en detalle el proceso de rellenado.

El algoritmo utiliza dos texturas como entrada que son muestreadas utilizando el método de interpolación del vecino más próximo:

- **Máscara de Contorno**: esta textura contiene los contornos de las islas de textura. Se crea como parte del cauce de representación del sistema haciendo uso de un algoritmo de dos pasadas. La primera pasada utiliza las coordenadas de textura para representar la parametrización de la malla 3D en una textura 2D. La segunda pasada aplica un filtro 2D a dicha textura que acepta aquellos téxeles que se encuentran a una distancia menor o igual a un téxel de las fronteras de las islas. Se actualiza cuando se carga un nuevo modelo 3D en el sistema.

- **Máscara del Área Editada**: esta textura contiene la forma del área que ha sido modificada por EAT. Se actualiza cada vez que el usuario edita una capa.

El algoritmo modifica las texturas 2D de la capa que han sido editadas por EAT: Textura de Datos y Textura de Máscara. Para ello, se crea un framebuffer especializado al cual se asocian dichas texturas y cada operación de rellenado las actualiza de la siguiente manera:

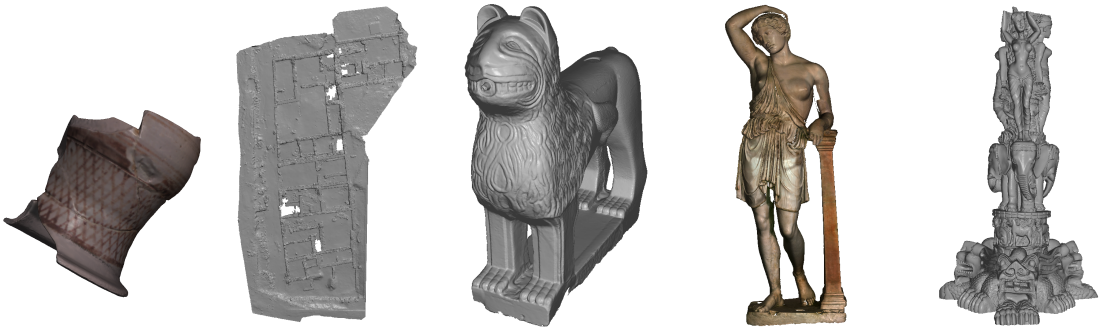
1. Textura de Datos se actualiza con el valor seleccionado para la operación de edición en las áreas de relleno.

2. Textura de Máscara se actualiza con el valor *true* en las áreas de relleno.

Haciendo uso de una cámara ortogonal, ART representa una simple escena con dos triángulos que forman un cuadrilátero del mismo tamaño que las texturas. El algoritmo es un ejemplo estándar de procesamiento de imágenes. Utiliza un kernel y el radio de dicho kernel depende de la anchura, en téxeles, del relleno que se desea aplicar. En el caso que nos ocupa un téxel es más que suficiente porque el sistema utiliza una interpolación del vecino más próximo para muestrear y representar las capas. Cada téxel es evaluado de la siguiente manera: el algoritmo muestrea Máscara de Contorno para comprobar si el téxel forma parte del contorno de una isla de textura. En caso afirmativo, ART muestrea a continuación Máscara del Área Editada para comprobar que su distancia al área editada es menor o igual que el radio del kernel. Los téxeles que satisfacen ambas condiciones forman parte de la región de relleno y, por consiguiente, son actualizados por el algoritmo a su valor correspondiente.

Este rellenado añade cierta información redundante a las texturas y garantiza que se utilice el color adecuado cuando la GPU muestrea y representa las fronteras de las islas de textura. Mientras que este procedimiento reduce el espacio útil en la textura, esta reducción no es suficientemente significativa para tener un impacto negativo en términos de espacio ya que el relleno tiene una anchura de un sólo téxel.

4.3 Análisis de resultados



(a) Vasija
312,094 triángulos
$105.18 \times 98.53 \times 91.41\ mm$

(b) Yacimiento
999,942 triángulos
$36.91 \times 67.37 \times 4.52\ m$

(c) León
3,000,223 triángulos
$10.24 \times 10.98 \times 3.09\ dm$

(d) Amazona
5,711,204 triángulos
$76.38 \times 200.36 \times 58.06\ mm$

(e) Thai
10,000,000 triángulos
$23.52 \times 39.60 \times 20.32\ cm$

Figura 4.6 Modelos 3D empleados en las pruebas. Modelos (a) y (d) disponen de color por vértice mientras que (b), (c) y (d) utilizan un gris estándar. Están organizados de izquierda a derecha en función de su número de triángulos. También se aportan las dimensiones de cada modelo

En esta sección se compara el rendimiento del prototipo que implementa la arquitectura propuesta con el ofrecido por otro sistema que emplea octrees. Concretamente, se ha seleccionado el sistema diseñado por Torres et al. **[TCM*10]**, denotado como OCT-TR, porque trabaja con modelos 3D, implementa estructuras para asociar información con independencia del número de triángulos de dichos modelos, usa capas de información para organizar los datos y, por consiguiente, es muy similar en términos de funcionalidad.

Las especificaciones del ordenador empleado para realizar estos tests se enumeran a continuación: Intel i7 4790k a 4.00 GHz, 16 GB DDR3 de memoria RAM a 1866 MHz y una tarjeta gráfica NVIDIA GTX 970 a 1.114 GHz con 4 GB GDDR5 de memoria RAM a 7 GHz.

Todos los tests realizados miden el rendimiento del proceso de edición de capas en escenarios equivalentes. Para la arquitectura aquí propuesta, esto implica la ejecución de los dos algoritmos explicados en la sección anterior: Algoritmo de Edición de Texturas y de Relleno de Texturas. En el caso de OCT-TR, esto implica el lanzamiento de múltiples rayos por parte de la CPU para encontrar qué vóxeles del octree intersecan, la actualización de los valores correspondientes en la capa y la transferencia de la capa actualizada a memoria de GPU para que pueda ser visualizada.

Se han seleccionado tres niveles de detalle para las capas de información con el fin de analizar el rendimiento cuando el tamaño de la celda decrece: tres resoluciones de textura para la solución propuesta y tres profundidades de octree para OCT-TR. Además, para cada de uno de estos tres casos, se han seleccionado cinco tamaños de herramienta distintos para evaluar el rendimiento cuando el área editada aumenta de tamaño. Todas estas pruebas utilizan los mismos modelos 3D que aparecen en la Figura 4.6.

Método	2048 prof 11	4096 prof 12	8192 prof 13
Propuesto	0.7702	0.1926	0.0481
OCT-TR	0.8752	0.2188	0.0547

Tabla 4.1 Esta tabla muestra el tamaño de celda medio obtenido por ambos algoritmos para el modelo León. Las medidas se han realizado en milímetros cuadrados. La cabecera de las columnas informa de la resolución de texturas (coloreado) y profundidad de octrees (blanco) que se han empleado en la comparación

Puesto que los octrees son estructuras espaciales volumétricas y las texturas son estructuras espaciales bidimensionales, no existe una posible comparación directa entre el área de la superficie contenida en un vóxel y el área contenida en un téxel. Por consiguiente, se han realizado múltiples test para establecer una correspondencia entre el tamaño de celda proporcionado por las distintas resoluciones de textura y los niveles de profundidad de octree y, empíricamente, se ha llegado a los siguientes resultados: la resolución de textura 2048×2048 es similar en términos de tamaño de celda a una profundidad de octree de nivel 11; la resolución de textura 4096×4096, a una profundidad de octree de nivel 12 y la resolución de textura 8192×8192, a una profundidad de octree de nivel 13. La Tabla 4.1 muestra el tamaño de celda medio, en milímetros cuadrados, conseguido por ambos sistemas para el modelo León. Los resultados para los otros cuatros modelos se incluyen en el Apéndice A y estos muestran un patrón similar donde la solución propuesta ofrece habitualmente un tamaño de celda menor que OCT-TR.

Mientras que la arquitectura propuesta no tiene problemas para manejar la estatua Thai, se ha de notar que no se pudieron completar los tests de OCT-TR para las profundidades de octree 12 y 13 con este modelo. OCT-TR requiere gran cantidad de memoria del sistema para generar el octree y el resto de estructuras necesarias. El gestor de memoria del sistema operativo informó que el sistema estaba usando más de 30 GB de memoria virtual antes de que la aplicación dejara de funcionar correctamente.

La Figura 4.7 muestra en escala logarítmica cómo evoluciona el rendimiento de ambos sistemas cuando el se modifica el tamaño del área editada. Tras un análisis detallado se puede concluir que la solución propuesta ofrece un comportamiento mucho más eficiente: los resultados crecen linealmente al contrario que aquellos ofrecidos por OCT-TR. Aunque el comportamiento teórico del algoritmo parece ser lineal, es prácticamente constante en términos prácticos ya que la pendiente de la línea es muy próxima a cero, con independencia del tamaño de la herramienta de edición. Las razones detrás de este excelente rendimiento se deben al buen uso de los recursos de la GPU. El algoritmo asocia las cargas de trabajo entre los núcleos de GPU de forma equitativa y minimiza las paradas en el cauce de ejecución de las GPU gracias a que no hay interdependencias en los cálculos. Todas las operaciones son independientes y no demasiado caras en términos de coste; asimismo, también hace uso de estructuras (texturas) que se encuentran completamente optimizadas por las arquitecturas de las GPUs. En contrapartida, los pobres resultados

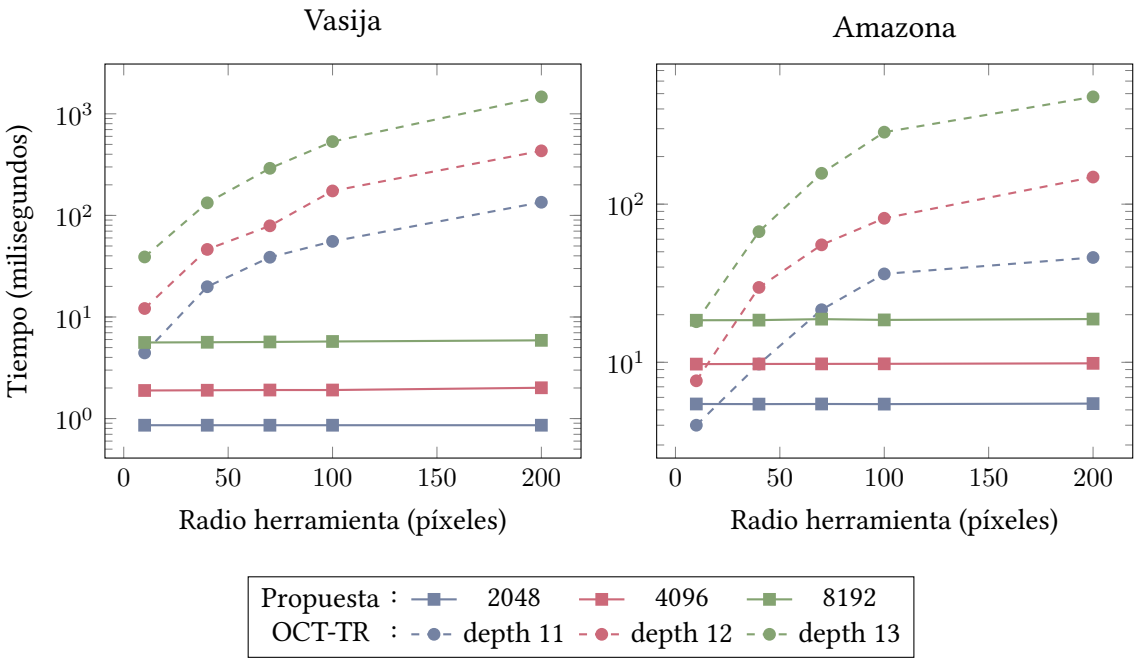


Figura 4.7 Estas gráficas, bajo escala logarítmica, muestran los resultados de las pruebas de edición de capas sobre dos modelos: a) Vasija, b) Amazona. Radio herramienta se corresponde con el radio de la herramienta de edición en píxeles. Las lineas sólidas representan los resultados de la solución propuesta mientras que las líneas discontinuas los propios de OCT-TR

obtenidos por OCT-TR se deben a la propia naturaleza de los octrees. Al contrario que el algoritmo aquí propuesto, estas estructuras residen en la memoria primaria y la CPU es la responsable de realizar todos los cálculos. En concreto, el proceso de edición implica el trazado de múltiples rayos sobre los vóxeles que componen la estructura. Cuanto mayor sea el área editada, mayor será el número de rayos y colisiones que tendrá que comprobar. El nivel de profundidad del octree también es un factor crítico en términos de complejidad de ejecución ya que los vóxeles se dividen por la mitad en cada dimensión entre niveles de profundidad consecutivos y, por consiguiente, el número de colisiones a realizar crece de forma exponencial.

La Figura 4.8 muestra en escala logarítmica cómo evoluciona el rendimiento cuando se editan modelos 3D de distinta complejidad utilizando el mismo tamaño de la herramienta de edición. Después de analizar los resultados, se puede concluir que la arquitectura aquí propuesta ofrece mejor rendimiento y un comportamiento completamente consistente entre los cinco modelos testados. El tiempo requerido para completar la operación de edición se incrementa cuando el número de triángulos es mayor o cuando el tamaño de la textura crece. Estos resultados son razonables porque aunque se está aprovechando el paralelismo de las GPUs, los recursos siempre tienen un límite. Al mismo tiempo, el algoritmo rinde mejor de lo esperado: la diferencia entre los resultados de la vasija y la estatua Thai es siempre inferior a una orden de magnitud a pesar de que la diferencia en

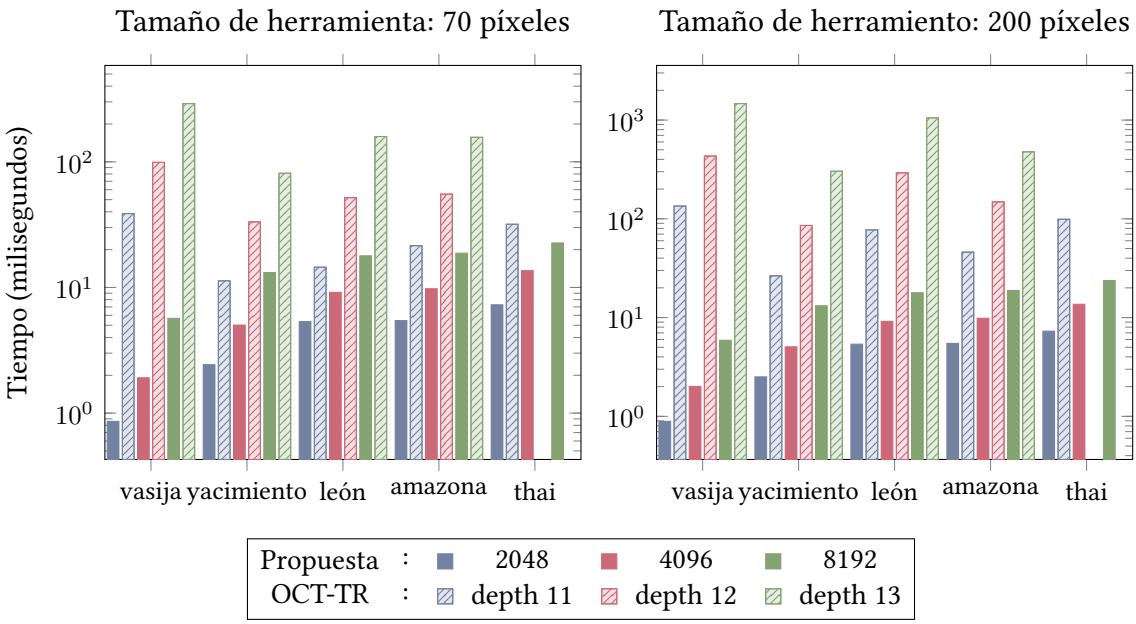


Figura 4.8 Estas gráficas muestran, bajo escala logarítmica, los resultados de las pruebas para la edición de capas empleando el mismo tamaño de la herramienta de edición sobre cada modelo. Se emplearon radios de 70 píxeles en la gráfica izquierda, mientras que en la gráfica derecha se utilizaron radios de 200 píxeles. Los modelos 3D están organizados en función de su número de triángulos. Las barras de colores sólidos se corresponden con los resultados de la solución propuesta y las barras rayadas con los propios de OCT-TR

términos de geometrías es muy cercana a las dos órdenes de magnitud. En contrapartida, OCT-TR muestra inconsistencia entre los modelos porque su rendimiento es altamente dependiente de lo equilibrado que sea el octree y la región que se esté editando. Cuando se proyectan objetos en octrees, una de sus principales características es su capacidad para descartar octantes completos con el fin de reducir el número de colisiones que se han de comprobar. La forma de los modelos 3D y los cambios en su orientación puede hacer que la región central del octree se encuentre bastante poblada. Al proyectar la herramienta de edición en esa región no se puede descartar octante alguno a niveles de profundidad bajos debido a que todos ellos contienen secciones del modelo, afectando negativamente al rendimiento. Por consiguiente, la forma y orientación de los modelos 3D son críticos para estructuras espaciales como los octrees. La vasija y su orientación inclinada es un ejemplo perfecto de esta desventaja. Es uno de los modelos más exigentes a pesar de que está compuesto por el menor número de triángulos. Además, el rendimiento de OCT-TR es dos órdenes de magnitud peor que la solución propuesta en el peor de los casos testados, empleando un tiempo superior a un segundo para completar una única operación de edición.

La Figura 4.9 muestra en escala logarítmica la cantidad de datos que la arquitectura propuesta y OCT-TR transfieren a la GPU durante cada operación de edición. Después

de analizar los resultados, es evidente que el rendimiento de OCT-TR es bastante peor. Dado que las capas casi siempre residen en GPU, la solución propuesta sólo transfiere los 64 bytes de la matriz que representa la posición de la herramienta de edición. En contrapartida, OCT-TR utiliza dos representaciones distintas para sus capas: la memoria del sistema almacena los datos y la memoria de GPU, su representación en colores. Por consiguiente, OCT-TR ha de enviar a la GPU una versión actualizada de los colores de la capa para visualizar estos cambios. Estas transferencias requieren casi 200 Megabytes (MB) por operación de edición para las capas más detalladas de la vasija (profundidad 13).

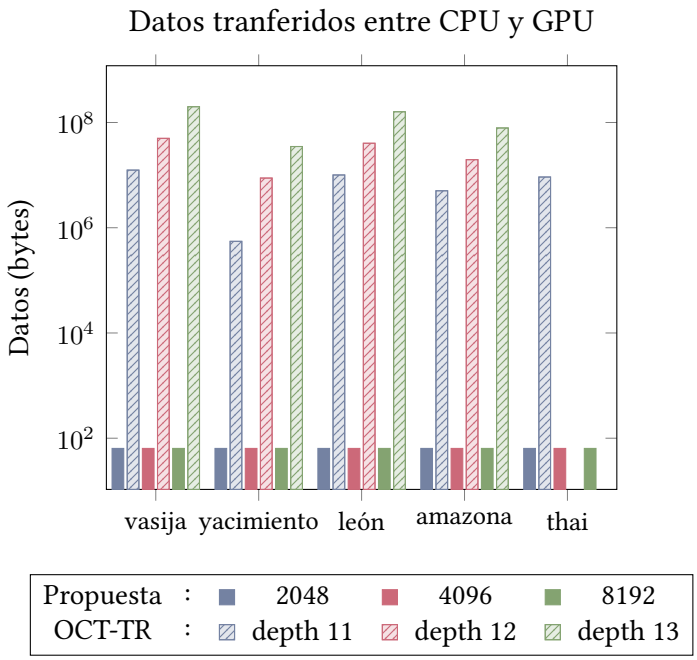


Figura 4.9 Esta gráfica muestra, bajo escala logarítmica, la cantidad de datos que se necesita transferir entre CPU y GPU durante cada operación de edición. Las barras de colores sólidos se corresponden con los resultados de la solución propuesta, mientras que la barras rayadas representan los propios de OCT-TR

Finalmente, la solución aquí propuesta también es, por norma general, más eficiente a nivel de espacio. Mientras que las capas son de mayor tamaño, la representación del modelo 3D es más compacta porque es constante en términos de espacio, con independencia de la resolución de las capas. OCT-TR, por contra, necesita subdividir sus mallas 3D cuando la profundidad del octree se incrementa. Por consiguiente, la solución propuesta es mejor cuando el número de capas usadas simultáneamente es inferior a un determinado umbral.

Utilizando la vasija como ejemplo ilustrativo, el sistema propuesto es más eficiente en términos de espacio cuando se trabaja con menos de once capas. Mientras que OCT-TR necesita 1,27 GB para almacenar el modelo 3D a nivel de profundidad 13, la solución propuesta requiere únicamente de 12 MB. Sin embargo, el tamaño de las capas a máxima resolución es de 327 MB frente a los 199 MB utilizados por OCT-TR.

4.4 Conclusiones

Se ha propuesto un arquitectura eficiente para sistemas de información que trabajan con patrimonio histórico. Para comprobar la eficiencia de la propuesta se han realizado test empíricos comparando esta solución con OCT-TR, demostrando claramente que la representación de datos es más eficiente y puede manejar modelos más complejos. Las principales ventajas de la arquitectura propuesta frente a OCT-TR se resumen a continuación:

- El tamaño de las mallas es constante mientras que OCT-TR necesita subdividirlas cuando la resolución de las capas se incrementa. Este formato de datos más compacto es importante cuando investigadores necesitan compartir información durante trabajos de campo.
- El tiempo requerido para la creación de las estructuras es insignificante en comparación con el tiempo de creación del octree.
- Las estructuras siempre residen en GPU.
- Todas las operaciones se realizan en GPU.
- La transferencia de datos entre CPU y GPU es prácticamente nula.
- Durante la edición de las capas, el tamaño de la herramienta de edición no afecta al rendimiento del algoritmo.

En definitiva, la arquitectura propuesta estructura la información en capas temáticas, utiliza texturas 2D para almacenarlas y coordenadas de textura para indexar sus datos. Además aprovecha el paralelismo inherente de las arquitecturas de GPU para gestionar y operar esta capas con eficiencia.

El trabajo descrito en este capítulo se ha publicado en la revista *Journal of Cultural Heritage* **[LTA*20]**.

5

Cómputo de topologías de modelos 3D en texturas 2D

La funcionalidad de los sistemas de información 3D basados en capas no se limita a asociar datos sobre la superficie de los objetos. Las operaciones entre capas son bastante frecuentes y representan una forma eficiente de obtener nuevos resultados a partir de información preexistente. Tras haber descrito una novedosa arquitectura capaz de utilizar texturas 2D como estructura de almacenamiento de capas, se va a detallar un método que permite a esta arquitectura almacenar la topología de modelos 3D en dichas estructuras y, en consecuencia, extiende su funcionalidad a toda aquella operación que requiere este tipo de información.

Las estructuras de datos topológicos son herramientas versátiles que permiten solventar una gran variedad de problemas en Informática Gráfica. Conocer cómo interactúan los diferentes elementos de una entidad y cómo se relacionan dichos elementos con sus vecinos es una información muy valiosa. Almacenar este tipo de información en estructuras de datos eficientes es un requisito clave a la hora de crear buenos algoritmos. De hecho, algoritmos centrados en este tipo de estructuras son bastante comunes cuando se trabaja con mallas 3D poligonales: Pierre et al. **[BHK14]** calculan contornos expresivos de mallas poligonales que son topológicamente equivalentes a los contornos de la superficie real aproximada; Cazals et al. **[CP05]** proponen un método para extraer líneas características a partir de una malla; Tierny et al. **[TVD07]** utilizan un esqueleto topológico mejorado para aplicar operaciones de segmentación dirigidas por semántica sobre mallas jerárquicas; Cignoni et al. **[CMS98]** analizan múltiples algoritmos de simplificación de mallas que intentan preservar su topología. Estas son algunos ejemplos de los resultados que las estructuras de datos topológicos permiten conseguir, pero no son los únicos. También son útiles a la hora de calcular distancias entre dos elementos de una superficie, estadísticas de vecindad como el valor máximo de un área o la curvatura de una malla.

La parametrización de mallas poligonales también es otra herramienta ampliamente usada en Informática Gráfica y el mapeado de texturas **[Cat74]** su ejemplo de uso más conocido. Esta técnica establece una función biyectiva entre dos superficies de topología similar y, tradicionalmente, se ha utilizado para mapear texturas sobre mallas 3D poligonales. Sin embargo, la creación del cauce programable de representación y la popularización del cálculo de propósito general en GPU han expandido su funcionalidad durante las últimas décadas.

Un gran ejemplo de la relevancia de ambas herramientas lo encontramos en las aplicaciones de patrimonio histórico, donde es frecuente añadir, editar y procesar información asociada a la superficie de objetos y las texturas se utilizan como medio para almacenar dicha información. Tal es el caso de la arquitectura propuesta en el capítulo anterior de esta tesis.

A lo largo de este capítulo se describe un nuevo método para almacenar la información topológica de mallas 3D. Esta propuesta aprovecha la funcionalidad de las GPUs para calcular de forma eficiente la topología en texturas 2D, ofrece soluciones razonables a los problemas que surgen durante la discretización inherente del proceso y es lo suficientemente robusta para poder manejar mallas compuestas de millones de triángulos.

5.1 Estructura de datos topológicos

Todo espacio topológico puede definirse por medio de tres conjuntos distintos: un conjunto de puntos, un conjunto de vecinos para cada punto y el conjunto de axiomas que conectan dichos puntos con sus vecindades. Una topología es la estructura o colección que contiene estos conjuntos y que permite la definición de continuidades en base a ellos. Formalmente, $[S, \mathcal{G}]$ recibe el nombre de espacio topológico si $\mathcal{G}$ es una familia de subconjuntos de S que cumple las siguientes tres propiedades:

P1: $\{\emptyset, S\} \subseteq \mathcal{G}$.

P2: Sean $M_1, M_2, \ldots, M_n$ una familia de conjuntos finitos o infinitos en $\mathcal{G}$; entonces la unión de estos conjuntos también pertenece a $\mathcal{G}$.

P3: Sean $M_1, M_2, \ldots, M_n$ una familia de conjuntos finitos en $\mathcal{G}$; entonces la intersección de estos conjuntos también pertenece a $\mathcal{G}$.

A $\mathcal{G}$ se le denomina **topología** en S y sus elementos, conjuntos abiertos. Todo espacio métrico $[S, d]$ genera una topología en S si cualquier punto $p \in S$ y cualquier umbral $\varepsilon > 0$ cumplen:

$$U_\varepsilon(p) = \{q : d(p, q) < \varepsilon\}$$

A $U_\varepsilon(p)$ se la conoce como la $U_\varepsilon(p)$-vecindad (abierta) de p en S. La familia de todas $U_\varepsilon(p)$-vecindades define una base de una topología y permite generar conjuntos abiertos por medio de uniones (finitas o infinitas) de $U_\varepsilon(p)$-vecindades **[Kle00]**.

Por ejemplo, en representaciones como mallas de triángulos o texturas, nos referimos a $U_\varepsilon(p)$ como $\mathcal{N}_n(v)$, donde v es un vértice de una malla y n denota la vecindad n-estrella; y $\mathcal{N}_n(t)$, donde t es un téxel y n denota la vecindad n-disco. Nos referimos a la vecindad n-estrella como aquella formada por v más el conjunto de vértices, aristas y caras conectadas por relación de adyacencia a v a una distancia geodésica inferior a n. Nos referimos a la vecindad n-disco como aquella formada por t más el conjunto de téxeles conectados por relación de adyacencia a t que se encuentran a un distancia inferior a n.

Las estructuras de datos que organizan eficientemente estas relaciones de incidencia y adyacencia entre múltiples elementos de mallas 3D se denominan estructuras de datos topológicos. Aunque existe un amplio rango de estructuras que almacenan información topológica, estas se han clasificado en dos grupos diferenciados: estructuras topológicas basadas en mallas y estructuras topológicas basadas en celdas.

5.1.1 Estructuras topológicas basadas en mallas

Estas estructuras utilizan elementos de las mallas poligonales como vértices, aristas y caras para construir una topología. Normalmente son fáciles de implementar y no requieren realizar modificación alguna de la malla original. Sin embargo, la densidad de dichas mallas determina su precisión. Las posibles disparidades existentes en el tamaño de la primitivas pueden impactar de forma negativa la utilidad de este tipo de estructuras. Áreas compuestas de triángulos de gran tamaño no proporcionan la precisión deseada, mientras que áreas formadas por triángulos muy pequeños añaden información redundante y disminuyen la eficiencia de los algoritmos cuando estos las atraviesan.

Desde que Baumgart describiera la estructura de aristas aladas **[Bau72]** para representar superficies en $\mathbb{R}^3$, se han propuesto múltiples variantes y estructuras relacionadas con ella: Guibas y Stolfi presentan la *quad-arista* **[GS83]**, capaz de encapsular *manifolds* duales y primarias y proporcionar un mecanismo para navegar entre ellas; Mantyla propone la estructura de semiaristas **[Man88]**, donde cada arista se divide en un par de semiaristas con direcciones opuestas; Rossignac diseña la estructura *corner-table* **[Ros01]**, que utiliza dos arrays de enteros y un conjunto de reglas para representar superficies triangulares. Lage et al. extienden esta estructura con la compacta *semi-cara* **[LLLV05]**, enfatizando su escalabilidad y bajos requisitos de memoria.

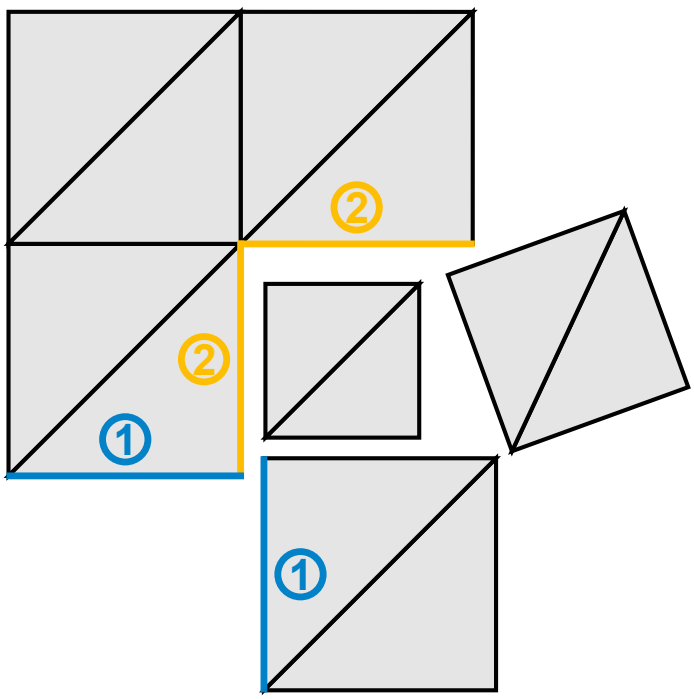


Figura 5.1 Dada la parametrización de un cubo representada en color grisáceo, se describen los dos tipos de aristas que son contiguas en 3D pero no están conectadas en el espacio 2D. Estas aristas pertenecen a islas distintas en el caso 1, mientras que son parte de la misma isla en el caso 2

5.1.2 Estructuras topológicas basadas en celdas

Estas estructuras subdividen las mallas poligonales en celdas de un tamaño similar que, normalmente, descomponen las caras en elementos más pequeños. Las celdas pueden ser bidimensionales o tridimensionales y las relaciones topológicas que establecen son intrínsecas al método de subdivisión empleado.

Los octrees son un buen ejemplo de estructuras de celdas 3D que tratan de subdividir la superficie de la malla en celdas cúbicas o vóxeles del mismo tamaño **[Mea82]**. Por consiguiente, su precisión no depende de las irregularidades que puedan presentar las primitivas. Sin embargo, pueden existir triángulos que formen parte de múltiples celdas. Estos casos especiales necesitan considerarse durante la creación de esta indexación espacial. También pueden producirse situaciones en las que elementos geométricos físicamente separados se encuentren lo suficientemente próximos como para formar parte de un mismo vóxel. Estos elementos podrían tratarse erróneamente como una entidad y, por tanto, representan otro posible problema que se ha de solventar **[SML17]**. En términos de topología, cada vóxel está relacionado con otros seis o doce vóxeles dependiendo del tipo de vecindad elegida. Torres et al. **[TLR*13]** y Soler et al. **[SML17]** usan octrees y los implementan como una lista de vóxeles que son atravesados por la superficie de la malla.

De igual forma, las texturas representan un ejemplo claro de estructura basada en celdas 2D. En este caso la parametrización de la malla se subdivide en cuadrados del mismo tamaño. Su precisión también es independiente de la irregularidades que puedan presentar los triángulos. Al contrario que los octress, la cercanía en el espacio 3D de elementos geométricos independientes no presenta problema alguno. Sin embargo, triángulos independientes pueden formar parte de un mismo téxel. A diferencia del problema en 3D, las islas de textura se pueden reorganizar en el espacio de textura o se pueden descomponer en caso de que fuera necesario, sin ningún coste adicional para la estructura en términos de complejidad. Con respecto a la topología, los téxeles que se encuentran en el interior de las islas se relacionan con otros cuatro u ocho téxeles dependiendo del tipo de vecindad seleccionada. En contrapartida, no existe ninguna relación de conectividad entre téxeles de islas distintas (caso 1 de la Figura 5.1), ni entre téxeles de la frontera de una misma isla que están conectados en el espacio 3D pero separados en el espacio 2D (caso 2 de la Figura 5.1). Otro problema único, presente en este tipo de estructuras, es el ocasionado por la diferencia de escala entre islas de textura vecinas.

5.2 Método propuesto

En esta sección se detalla el método **Topology Texture Algorithm** (TTA), que es capaz de almacenar la información topológica de un modelo 3D en texturas 2D. A continuación se describe la estructura que se seguirá: la Subsección 5.2.1 presenta los retos de usar texturas como contenedores de información topológica de una malla de triángulos; la Subsección 5.2.2 describe la estructura de datos empleada por el algoritmo; la Subsección 5.2.3 detalla el preprocesamiento que la malla necesita para ser usada por el algoritmo; y, finalmente, la Subsección 5.2.4 explica los diferentes pasos del algoritmo propuesto.

5.2.1 Topología y texturas

La parametrización de una malla es un función de mapeo biyectiva entre dos superficies que presentan una topología similar y donde al menos una de ellas es una malla triangular. Asimismo, el dominio de la parametrización es la superficie sobre la cual se mapea dicha malla. Habitualmente se utiliza como dominio un espacio 2D independiente que va a albergar cada triángulo que forma parte de la malla. Además, en el ámbito de la Informática Gráfica, este espacio suele discretizarse en forma de texturas 2D, que actúan como contenedores multipropósito de uno o múltiples arrays de téxeles.

Por consiguiente, la discretización es la primera consecuencia de utilizar texturas para almacenar la información topológica de mallas 3D. La resolución de dichas texturas determinará, por tanto, cuán precisa será la información almacenada. Sin embargo, este no es el único reto que se ha de resolver. La parametrización suele descomponer las mallas en secciones desconectadas que generan una distribución en islas a lo largo del espacio de textura. La adyacencia entre los téxeles contenidos en dichas islas ocurre de forma natural pero, por contra, no existe ninguna conectividad directa entre triángulos que comparten las mismas aristas en la malla 3D y pertenecen a distintas islas en el espacio de textura (Figura 5.1). Además, puede no existir una única correspondencia entre los téxeles que representan estas aristas ya que las islas no han de compartir necesariamente la misma escala u orientación.

La propuesta que se detalla a continuación aprovecha la adyacencia inherente de los téxeles que conforman las islas para añadir un contorno de un téxel de anchura. En él se almacena la información necesaria para enlazar téxeles contiguos que pertenecen a regiones inconexas en el espacio de textura.

5.2.2 Estructura de datos

TTA es un algoritmo compuesto de múltiples etapas que almacena la información topológica de una malla de triángulos en una textura 2D que contendrá dos números enteros por cada téxel. El algoritmo discrimina entre téxeles que pertenecen a islas de textura y aquellos que se encuentran en sus contornos de la siguiente manera:

- **Téxeles interiores** son todos aquellos téxeles que forman parte de las islas. Nos informan de si sus vecinos inmediatos son también téxeles interiores y, por consiguiente, accesibles mediante una traslación de un téxel en uno o dos ejes. La textura utiliza los ocho primeros bits de un número entero para codificar esta información dentro de la primera componente entera de cada téxel. Además, este número se convierte a un entero negativo para poder diferenciarlo de coordenadas de textura válidas.

- **Téxeles de contorno** son todos aquellos téxeles que pertenecen al área de un téxel de anchura que rodea a las islas de textura. Conectan los téxeles situados en la frontera de una isla con sus vecinos remotos. Dichos vecinos se pueden encontrar en otras islas o secciones no conectadas de una misma isla. Para poder establecer esta relación, la textura almacena las coordenadas de los vecinos en las dos componentes enteras disponibles para cada téxel de contorno.

La Figura 5.2 muestra la parametrización de una sencilla malla 3D en la cual los téxeles de contorno se representan en rojo, mientras que los téxeles interiores se representan en gris. Las flechas verdes describen las coordenadas de textura de los téxeles adyacentes en otras islas. Las flechas en azul describen los ocho vecinos inmediatos de un téxel interior.

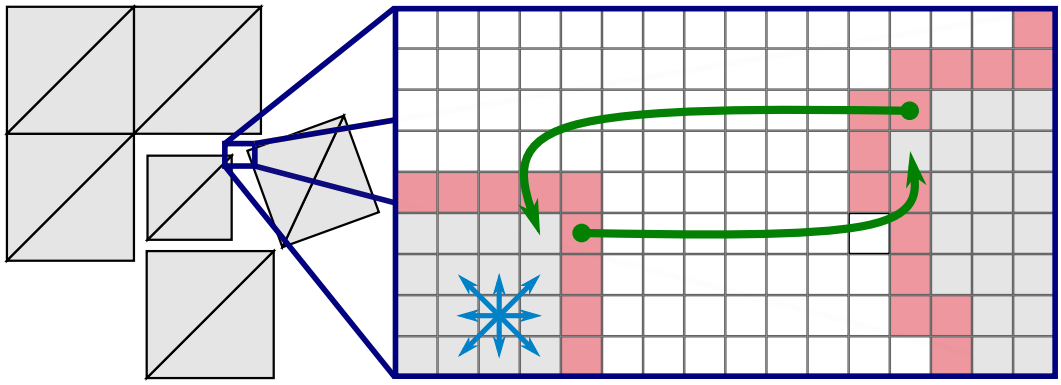


Figura 5.2 Dada la parametrización del cubo mostrada en la figura, las flechas verdes describen las coordenadas de los téxeles vecinos que se encuentran en otra isla de textura. Dichas coordenadas se almacenan en los téxeles del contorno representados en rojo y marcados por círculos verdes. Las flechas azules describen los ocho vecinos de un téxel de una isla. El método codifica y almacena en cada téxel inspeccionado cuáles de estos vecinos son también téxeles de una isla

5.2.3 Preprocesamiento de la malla

Aunque TTA se resuelve completamente en el espacio de la GPU, el algoritmo requiere un preprocesamiento inicial de la malla en CPU. Dado que TTA está intentando solventar un problema relacionado con la topología, este cómputo inicial crea una estructura temporal de semiaristas o *half-edges* **[BSBK02]** que proporciona la información de adyacencia necesaria entre triángulos, aristas y vértices. Mientras que la estructura es capaz de proporcionar esta información topológica para toda la malla 3D, TTA sólo la necesita en aquellas semiaristas cuyas coordenadas de textura pertenecen a la frontera de las islas y que se denominan *semiaristas fronterizas*. Afortunadamente, esta propiedad es realmente sencilla de comprobar con esta estructura. Si una semiarista y su pareja no representan a la misma arista en el espacio de textura, significa que ambas forman parte de una frontera y pueden pertenecer a islas distintas.

Una vez se ha calculado el conjunto de *semiaristas fronterizas*, la etapa de preprocesamiento itera sobre cada una de ellas para crear cuatro buffers de datos con la siguiente estructura:

- **Coordenadas**: almacena las coordenadas de textura de cada semiarista.
- **Coordenadas de pareja**: almacena las coordenadas de textura de las parejas de las semiaristas procesadas.
- **Normales**: almacena las normales en el espacio de textura de las semiaristas procesadas.
- **Nomales de pareja**: almacena las normales en el espacio de textura de las parejas de las semiaristas procesadas.

La Figura 5.3 ofrece una representación esquemática de cómo se organizan las coordenadas de textura y la normal de las *semiaristas fronterizas* en los buffers de datos.

Este preprocesamiento finaliza cuando los buffers se transfieren desde la memoria principal del sistema a la memoria de la GPU y, por consiguiente, están listos para ser utilizados como entradas del algoritmo que se propone.

5.2.4 Topology Texture Algorithm

TTA se organiza en cuatro pasadas de shaders para construir la textura Topology, cuya estructura se ha descrito en la Subsección 5.2.2. El algoritmo calcula la información de ayacencia en los contornos de las islas de textura y, posteriormente, la conectividad dentro de las mismas. Las pasadas de shaders se detallan de la siguiente manera:

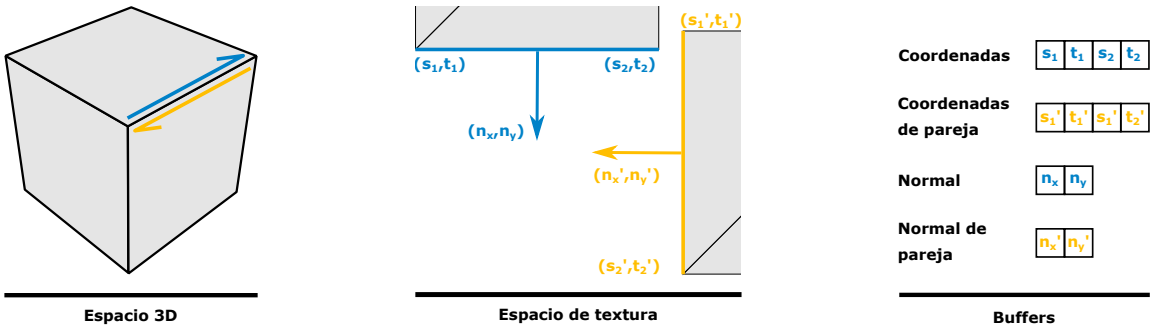


Figura 5.3 Durante la etapa de preprocesamiento se crean cuatro buffers de datos con la normal y las coordenadas de textura de aquellas semiaristas adyacentes que forman parte de la frontera de una isla

5.2.4.1 *Border Pass*

Debido a la organización en islas, los triángulos que son contiguos en el espacio 3D puede que no tengan una relación de adyacencia en el espacio de textura. La Figura 5.1 muestra los dos casos que se han de solventar. En el primero de ellos los téxeles de la frontera son adyacentes, pero pertenecen a islas diferentes (pareja de aristas 1). En el segundo caso, los téxeles de la frontera pertenecen a la misma isla pero se encuentran en regiones diferentes e inconexas (pareja de aristas 2).

Esta primera etapa necesita tres tipos de datos de entrada: los buffers de datos definidos en la Subsección 5.2.3, la información de la cámara empleada y la textura Shape Mask, que contiene la forma de las islas de textura. Concretamente, Shape Mask emplea un bit por téxel para almacenar si estos pertenecen a la parametrización de la malla 3D en el espacio de textura, es decir, si forman parte de una isla.

Asimismo, emplea un *offscreen framebuffer* que tiene asociada la textura Border Adyacency. En esta textura se representan los resultados y sus téxeles están compuestos de cuatro enteros que pueden llegar a almacenar coordenadas de textura en sus primeras dos componentes y vectores normales 2D en las dos últimas.

La escena que genera los resultados se representa desde el punto de vista de una cámara ortogonal usando las coordenadas de las *semiaristas fronterizas* como posiciones de los vértices. En consecuencia, la línea es la primitiva de representación utilizada y las coordenadas de textura de las semiaristas adyacentes se interpolan a lo largo de dichas líneas.

Cada téxel representado se actualiza de la siguiente manera: el algoritmo transforma las coordenadas interpoladas de una semiarista adyacente en índices enteros de textura. A continuación muestrea la textura Shape Mask para comprobar que los índices se encuentran dentro de una isla. Esta comprobación es necesaria debido a que la rasterización de líneas y triángulos se resuelve de diferentes maneras y la representación de las aristas no siempre se encuentra en el área cubierta por la frontera de los triángulos. En caso de que los índices

no se encuentren en el interior de una isla, el algoritmo los traslada un téxel en la dirección de la normal de la semiarista adyacente. Finalmente, los índices resultantes se almacenan en las dos primeras componentes, mientras que la normal de la línea representada se almacena en los restantes.

Algoritmo 1 Border Pass

$indices \leftarrow coords * textureSize$
$inside \leftarrow$ SAMPLE$(ShapeMask, coords)$
if $inside = true$ **then**
  $BorderAdjacency \leftarrow \{indices, normal\}$
else
  $newIndices \leftarrow indices +$ ROUND$(normal_{adjacentHE})$
  $BorderAdjacency \leftarrow newIndices, normal$
end if

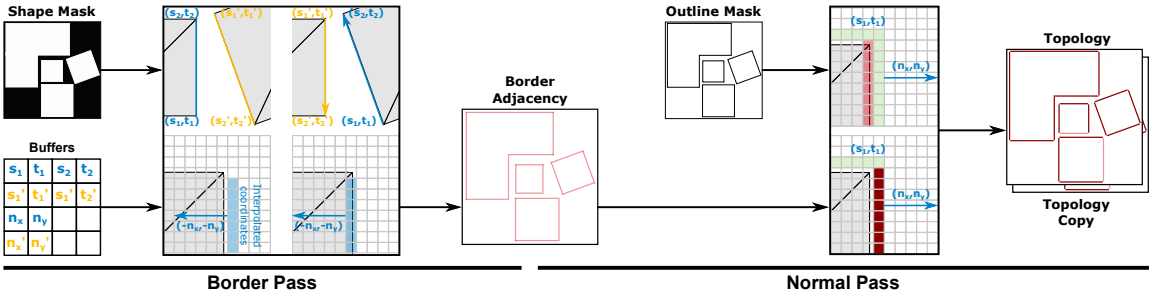


Figura 5.4 Border Pass necesita como entradas cuatro buffers de geometría y la textura Shape Mask. Interpola las coordenadas de textura de aristas adyacentes a lo largo de la frontera de las islas y rellena la textura Border Adjacency de forma adecuada. Normal Pass utiliza la salida generada por Border Pass y la textura Outline Mask como entrada. Traslada la información de la frontera al contorno siguiendo la dirección de la normal y rellena con la información adecuada dos texturas idénticas: Topology y Topology Copy

La sección de la Figura 5.4 que describe Border Pass contiene cuatro paneles. El panel situado en la parte superior izquierda muestra dos semiaristas adyacentes en color rojo. Las coordenadas de textura de la arista situada a la izquierda están representadas en azul, mientras que las coordenadas de la arista situada a la derecha aparecen en dorado. El panel situado en la parte superior derecha describe como estas coordenadas se interpolan de forma lineal a lo largo de las aristas. Las coordenadas doradas lo hacen sobre la arista situada a la izquierda, mientras que las coordenadas azules lo hacen sobre la arista de la derecha.

El panel situado en la parte inferior izquierda describe como las coordenadas de textura interpoladas, que ya han sido transformadas en índices enteros de textura y están representadas en color azul, no se encuentran en téxeles interiores representados en color

gris. El panel situado en la parte inferior derecha muestra como se trasladan a la posición adecuada con la ayuda de la normal.

5.2.4.2 *Normal Pass*

Los índices generados por la pasada anterior no deben quedarse en la frontera de la islas sino que han de ser desplazados al contorno, es decir, los téxeles contiguos a la frontera. Debido a que la rasterización de líneas y triángulos se solventa utilizando distintos algoritmos, algunos de los téxeles actualizados por Border Pass puede que ya estén formando parte del contorno, pero no todos ellos se encontrarán en esta situación. Por tanto, es necesario desplazar al contorno todos aquellos téxeles que se encuentren dentro de las islas.

Esta etapa utiliza dos texturas como entrada: la salida del paso anterior, la textura Border Adjacency, y Outline Mask, que contiene los contornos de las islas. Esta máscara se crea usando un algoritmo de dos pasadas. La primera de ellas utiliza las coordenadas de textura para representar la parametrización de la malla 3D en una textura 2D. La segunda pasada aplica un filtro de imagen que utiliza la textura generada en el paso anterior para comprobar si los téxeles procesados se encuentran a una distancia de un téxel de las islas. Tanto Border Adjacency como Outline Mask se muestrean usando la interpolación del vécino más cercano (*nearest-neighbour*).

Actualiza dos texturas de salida: **Topology**, la textura final que el algoritmo está intentando construir y **Topology Copy**, una copia temporal de la primera. Cada téxel de estas texturas dispone de dos componentes y puede almacenar valores válidos de índices de textura. Ambas se actualizan con los mismos valores.

El algoritmo procesa cada téxel de la siguiente manera: en primer lugar muestrea Outline Mask para comprobar si el téxel que está inspeccionando forma parte del contorno de una isla. En caso afirmativo, el algoritmo pasa a muestrear Border Adjacency para comprobar si el téxel inspeccionado, a su vez, contiene ya índices enteros válidos en sus dos primeras componentes. Cuando los téxeles cumplen ambas condiciones, se actualiza la textura Topology con dichos índices. En caso contrario, si el téxel forma parte del contorno pero no contiene índices válidos, el algoritmo comprueba si los vecinos inmediatos que se encuentran situados en la dirección del vector normal a la línea representada en la pasada anterior sí los tienen. Cuando se cumplen estas tres condiciones, el algoritmo actualiza la textura Topology con los índices válidos encontrados en dichos vecinos.

La sección que describe Normal Pass en la Figura 5.4 representa las islas de textura en color gris, el contorno en verde y los téxeles previamente actualizados por Border Pass en rojo. En el panel superior, los téxeles actualizados se encuentran en el interior de la isla de textura, mientras que en el panel inferior estos téxeles son trasladados al contorno siguiendo la dirección de la normal.

Algoritmo 2 Normal Pass

```
outlineTexel ← SAMPLE(OutlineMask, coords)
if outlineTexel = true then
    {indices, normal} ← SAMPLE(BorderAdj, coords)
    if indices = valid then
        Topology ← indices
        TopologyCopy ← indices
    else
        for n ← 1 to 8 do
            coords_n ← coords + offset_n
            indices_n, normal_n ← SAMPLE(BorderAdj, coords_n)
            outlineTexel_n ← SAMPLE(OutlineMask, coords_n)
            if outlineTexel_n = true & VALID(indices_n)
                & ROUND(normal) = offset_n then
                Topology ← indices_n
                TopologyCopy ← indices_n
            end if
        end for
    end if
end if
```

5.2.4.3 *Gap Pass*

El algoritmo puede no ser capaz de rellenar inicialmente todos los téxeles del contorno debido a la traslación de un téxel realizada por Normal Pass, la divergencia entre las rasterizaciones de líneas y triángulos, así como la diferencia en escala entre islas. Las esquinas y algunos téxeles contiguos a los extremos de las líneas no se rellenan de forma apropiada y necesitan ser actualizados con los índices enteros de textura de un vecino inmediato.

Esta etapa necesita dos entradas: las texturas Outline Mask y Topology Copy, una de las salidas generadas por Normal Pass. Ambas texturas se muestrean utilizando la interpolación del vecino más próximo. Asimismo, sólo actualiza la textura Topology, la otra salida de Normal Pass, con los índices de textura pertinentes.

El proceso en cuestión es un algoritmo de procesamiento de imagen estándar. Utiliza un kernel 3×3 y cada téxel es testado de la siguiente manera: el algoritmo muestrea Outline Mask para comprobar si el téxel inspeccionado forma pare del contorno de una isla. En caso afirmativo, el algoritmo pasa a muestrear la textura Topology Copy para comprobar si dicho téxel no contiene índices de textura válidos. Aquellos téxeles que satisfagan ambas condiciones forman parte del contorno y necesitan ser actualizados con los índices apropiados. Por consiguiente, el algoritmo muestrea la textura Topology Copy

una vez más para comprobar si alguno de sus ocho vecinos contiene índices válidos. En caso afirmativo, copia dichos índices en el téxel inspeccionado.

Algoritmo 3 Gap Pass

```
outlineTexel ← SAMPLE(OutlineMask, coords)
indices ← SAMPLE(TopologyCopy, coords)
if outlineTexel = true & indices ≠ valid then
    written ← false
    for n ← 1 to 8 do
        coords_n ← coords + offset_n
        indices_n ← SAMPLE(TopologyCopy, coords_n)
        outlineTexel_n ← SAMPLE(OutlineMask, coords_n)
        if outlineTexel_n = true & VALID(indices_n) then
            Topology ← indices_n
            EXIT
        end if
    end for
end if
```

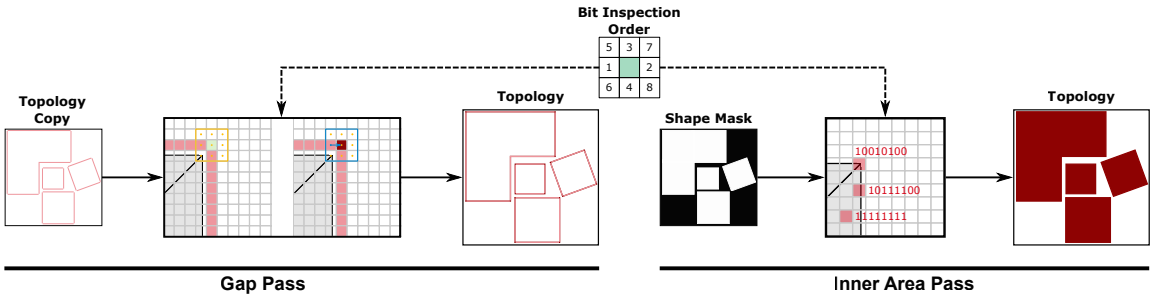


Figura 5.5 Gap Pass sólo necesita la textura Topology Copy como entrada. Se encarga de rellenar los téxeles del contorno aún por asignar con los valores de sus vecinos y actualiza la textura Topology. Inner Area Pass sólo utiliza la textura Shape Mask como entrada. Calcula información topológica de los téxeles interiores y también actualiza la textura Topology. Cada téxel muestrea sus ocho vecinos inmediatos y codifica con el número 1 aquellos vecinos que interiores y con 0 los restantes, siguiendo el orden descrito en la parte central

La sección de la Figura 5.4 que describe Gap Pass muestra la isla de textura en color gris, el contorno en verde y los téxeles actualizados por Normal Pass en rojo. El cuadrado dorado en el panel de la izquierda representa los téxeles vecinos que van a ser muestreados para rellenar la esquina. El panel de la derecha muestra como el vecino del centro-izquierda es seleccionado debido a que cumple todas las condiciones. El cuadrado en el centro de la Figura 5.5 describe el orden en el cual los vecinos son muestreados.

5.2.4.4 *Inner Area Pass*

Mientras las tres etapas anteriores se ocupan de calcular la información topológica entre islas de textura o fronteras no conectadas de una misma isla, el algoritmo todavía necesita rellenar los téxeles interiores con la conectividad apropiada.

Este último paso solo necesita una entrada, la textura Shape Mask que contiene la forma de las islas de textura. Esta entrada se muestrea utilizando la interpolación del vecino más cercano.

De forma análoga, sólo actualiza la textura Topology, la salida del Gap Pass, con la información de adyacencia pertinente. Concretamente, todos aquellos téxeles que se encuentran dentro de la islas.

El algoritmo almacena la información de adyacencia en la primera componente de la textura Topology y la codifica utilizando un único byte. Cada bit de dicho byte representa de forma unívoca a uno de los ocho vecinos inmediatos del téxel inspeccionado. La Figura 5.5 muestra el orden de esta relación, comenzando por el vecino de centro-izquierda y terminado con aquel situado en las esquina inferior izquierda.

Cada téxel interior se actualiza de la siguiente manera: para cada uno de los ocho vecinos inmediatos se comprueba si estos forman parte de la misma isla y en caso afirmativo el algoritmo asigna el número 1 al bit correspondiente. En otro caso, el bit se asigna a 0.

Como último cómputo, se multiplica el valor codificado por -1 para convertirlo en número negativo y, de esta forma, evitar colisiones con coordenadas almacenadas por los tres pasos anteriores. El resultado final se almacena en la primera componente del téxel.

Algoritmo 4 Inner Area Pass

if $outlineTexel = true \ \ \& \ \ indices \neq valid$ **then**
  $written \leftarrow false$
  for $n \leftarrow 1$ to 8 **do**
    $coords_n \leftarrow coords + offset_n$
    $inside \leftarrow$ SAMPLE$(ShapeMask, coords_n)$
    if $inside = true$ **then**
      $adjacency \leftarrow adjacency \mid 1 \ll n$
    end if
  end for
  $Topology \leftarrow -adjacency$
end if

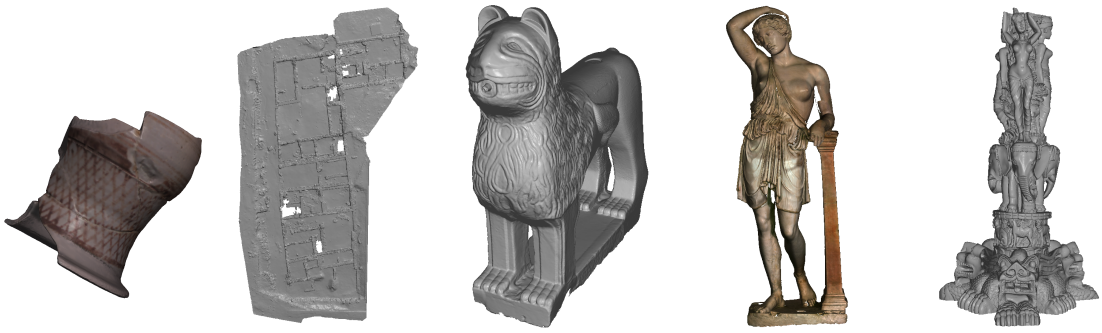
La Figura 5.5 muestra unos pocos ejemplos de información topológica codificada por Inner Area Pass en un byte. La codificación sigue el orden establecido por el cuadrado que se encuentra en el centro.

5.3 Resultados y discusión

En esta sección se compara el rendimiento y precisión del método propuesto empleando múltiples modelos 3D con diferentes parametrizaciones y un número variable de triángulos. Las especificaciones hardware son las siguientes: CPU Intel i7 4790k a 4.00 GHz, 16 GB memoria RAM DDR3 a 1866 MHz y NVIDIA RTX 2080 a 1.8 GHz con 8 GB de memoria RAM GDDR6 a 7 GHz.

5.3.1 Test de rendimiento

Para testar el rendimiento del algoritmo propuesto se han seleccionado los mismos cinco modelos 3D utilizados en las pruebas del Capítulo 4. Con el fin de facilitar el seguimiento de esta sección, se vuelven a mostrar en la Figura 5.6. Cabe recordar que se trata de modelos con grandes diferencias en cuanto al número de triángulos que los componen y que se han obtenido mediante un proceso de escaneado, por lo que ofrecen escenarios reales en términos de complejidad topológica. Además, la forma cilíndrica de Vasija, la superficie plana de Yacimiento y la figura humanoide de Amazona dan lugar a variados escenarios de parametrización de mallas.



(a) Vasija
312,094 triángulos
$105.18 \times 98.53 \times 91.41\ mm$

(b) Yacimiento
999,942 triángulos
$36.91 \times 67.37 \times 4.52\ m$

(c) León
3,000,223 triángulos
$10.24 \times 10.98 \times 3.09\ dm$

(d) Amazona
5,711,204 triángulos
$76.38 \times 200.36 \times 58.06\ mm$

(e) Thai
10,000,000 triángulos
$23.52 \times 39.60 \times 20.32\ cm$

Figura 5.6 Modelos 3D utilizados en las pruebas. Modelos (a) y (d) disponen de colores por vértice mientras que (b), (c) y (d) utilizan un color gris estándar. Están ordenados de izquierda a derecha en función del número de triángulos que los componen. Las dimensiones también se detallan en cada modelo

El algoritmo se ha ejecutado múltiples veces sobre todos los modelos 3D empleando tres tamaños distintos para las texturas: 2048×2048, 4096×4096, 8192×8192. La Figura 5.7 muestra los resultados de estos tests en función del modelo y el tamaño de textura para cada pasada del algoritmo.

Las gráficas permiten extraer varias conclusiones interesantes. Border Pass es la etapa más lenta del algoritmo por una diferencia de un orden de magnitud debido a que el

algoritmo necesita representar la geometría sobre la textura. Sin embargo, el tiempo de cómputo no se incrementa linealmente con el número de triángulos de los distintos modelos. Este comportamiento se justifica fácilmente si se tiene en cuenta que esta primera etapa sólo representa las fronteras de las islas de textura y, por consiguiente, su rendimiento depende del número de aristas necesarias para dibujarlas. Modelos con una cantidad baja de triángulos pueden tener parametrizaciones de malla mas complejas que otros compuestos por una mayor cantidad de los mismos. Precisamente, ese es el caso de Vasija y Yacimiento con respecto a León.

Las tres pasadas restantes (Normal, Gap e Inner Area) son filtros de imagen rápidos. Su rendimiento depende del número y forma de las islas de textura en el caso de Normal y Gap y el área ocupada por las islas en el caso de Inner Area. Como ya ocurriera en la primera etapa, modelos más simples pueden exhibir un rendimiento ligeramente inferior debido a parametrizaciones de malla más complejas. Yacimiento y León, por ejemplo, ofrecen mejor rendimiento que Vasija.

Al mismo tiempo, el rendimiento de las cuatro etapas decrece linealmente con el tamaño de la textura. Comportamiento esperable puesto que se amplía el tamaño de la superficie de representación.

Para obtener una mejor comprensión del rendimiento general, se ha comparado la solución propuesta con OCT-TR, el sistema diseñado por Torres et al. **[TCM*10, STLL13]** que ya fue utilizado en las pruebas realizadas en el Capítulo 4. Este sistema también utiliza estructuras topológicas de datos basadas en celdas y, por consiguiente, pertenece a la mimas categoría definida en la Subsección 5.1.2. En concreto, trabaja con octrees para indexar geometría y asociar información a modelos 3D con independencia de su número de triángulos.

Método	2048-11	4096-12	8192-13
Propuesto	0.7702	0.1926	0.0481
OCT-TR	0.8752	0.2188	0.0547

Tabla 5.1 Tamaño de celda medio logrado por ambos métodos para el modelo León. Las medidas se ofrecen en milímetros cuadrados. La cabecera de cada columna muestra la resolución de la textura a la izquierda y la profundidad del octree a la derecha

Cabe recordar que también se realizaron pruebas empíricas en los tests del Capítulo 4 para establecer una equivalencia entre los tamaños de celda proporcionados por tres resoluciones de textura y tres profundidades de octree. Se vuelve a incluir la Tabla 5.1 que muestra el tamaño de celda medio, en milímetros cuadrados, conseguido por ambos sistemas para el modelo León.

Las pruebas realizadas miden el rendimiento de ambos métodos durante el proceso de creación de las estructuras que necesitan para calcular la información topológica y el cómputo de la topología para los tamaños de textura 2048×2048, 4096×4096, 8192×8192 y las profundidades de octree 11, 12 y 13. La Tabla 5.2 muestra los tiempos obtenidos

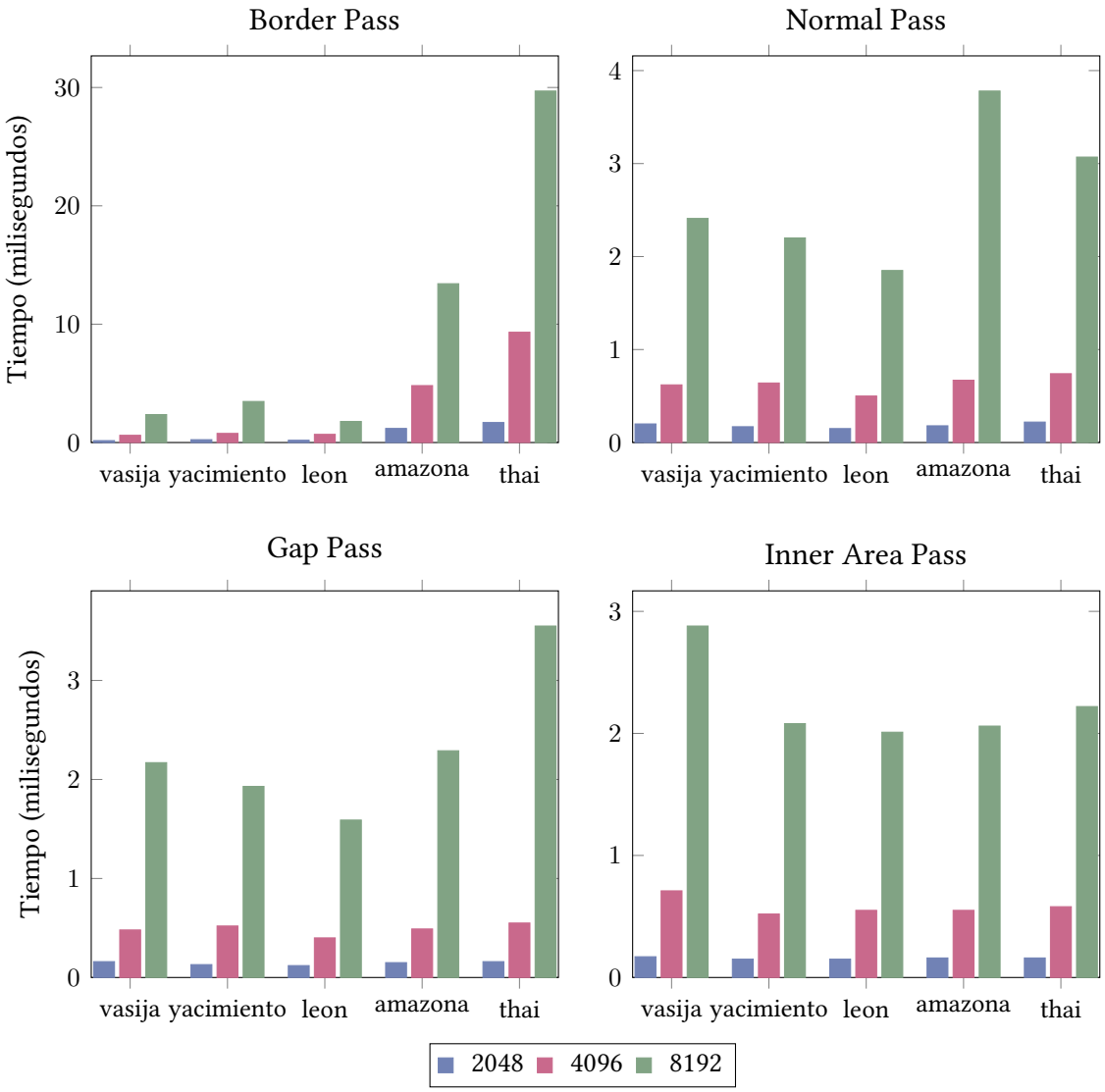


Figura 5.7 Resultados de las pruebas de rendimiento para las cuatro pasadas del algoritmo. Border Pass (superior izquierda), Normal Pass (superior derecha), Gap Pass (inferior izquierda) e Inner Area Pass (inferior derecha). Cada prueba emplea los cinco modelos representados en la Figura 5.6 y mide el rendimiento para tres tamaños de textura: 2048×2048 (azul), 4096×4096 (rojo) y 8192×8192 (verde).

con el modelo León. El resto de resultados se puede encontrar en el Apéndice B y estos muestran un patrón similar al que aquí se presenta.

Tamaño	Estructura		Topología	
	Semiarista	Octree	Textura	Octree
2048-11	8987	265952	0.61	11407
4096-12	8987	455539	2.14	43921
8192-13	8987	618486	7.23	182886

Tabla 5.2 La tabla muestra cuántos milisegundos necesita TTA y OCT-TR en calcular las estructuras y la información topológica para el modelo León. Se utilizan tres niveles de detalle agrupados en resoluciones de textura y profundidades de octree con tamaños de celda similares

Tras inspeccionar los resultados de la Tabla 5.2 resulta evidente la gran diferencia que existe en términos de rendimiento entre ambos métodos. La solución propuesta es capaz de construir la estructura de semiaristas dos órdenes de magnitud más rápido que OCT-TR su octree. La diferencia es incluso más grande si se observa el cálculo de la información topológica, alcanzando cinco órdenes de magnitud en el caso más complejo, 8192-13.

Dado que la estructura de semiaristas se construye utilizando los componentes de la malla, el rendimiento del algoritmo propuesto es constante para los tres niveles de detalle. En contraste, OCT-TR utiliza octrees que subdividen la geometría de forma recursiva utilizando ocho celdas cúbicas o vóxeles. Por consiguiente, cuanto mayor sea el nivel de detalle, más pequeño será el vóxel, incrementando la cantidad de tiempo necesaria para crear la estructura.

Por otro lado, el algoritmo propuesto usa estructuras eficientes de GPU, texturas 2D, para almacenar la información topológica y hace un uso adecuado de su paralelismo inherente. En términos simples, la primera pasada del algoritmo realiza la rasterización de las fronteras de las islas de textura y las tres restantes son filtros de imagen sin ninguna interdependencia en los cálculos que necesitan realizar. Por consiguiente, los retrasos en la ejecución del cauce de la GPU se mantienen al mínimo y sus recursos se utilizan de forma equilibrada. Mientras tanto, OCT-TR emplea algoritmos que se ejecutan en CPU para construir la capa de topología mediante un octree, obteniendo un descenso del rendimiento superior al lineal con el incremento de la complejidad.

5.3.2 Test de precisión

Para evaluar la precisión del algoritmo propuesto se ha elegido la operación de campos de distancias. Este cómputo requiere de la información topológica para ofrecer resultados coherentes y estos pueden compararse con soluciones reales, calculadas con métodos diferentes.

Además de la textura Topology calculada por TTA, esta operación necesita otra textura que almacena el área de la superficie contenida en cada téxel. La textura en cuestión es calculada por el sistema como parte del proceso de carga de un modelo 3D.

A continuación se ofrece un resumen del procesamiento necesario para poder calcular campos de distancia. El área inicial, proporcionada como entrada, se encuentra a una distancia cero y el resto de téxeles se encuentran a una distancia equivalente a infinito. El cálculo procede a expandir el área visitada de forma progresiva hasta que toda la superficie del malla se haya inspeccionado. Cada téxel situado en la frontera de dicha área comprueba la distancia actual a cada vecino inmediato que se encuentra en la parte exterior y la actualiza si la suma de su distancia al área inicial y su distancia al vecino es inferior al valor actual.

Dentro de las islas de textura la vecindad inmediata a cualquier téxel está formada por los ocho téxeles que lo rodean. Esta distribución se ve alterada en los téxeles de frontera debido a la orientación, forma y escala de las otras islas o regiones inconexas de la misma isla que contienen a sus vecinos. El efecto es más pronunciado cuanto mayor sean las diferencias con respecto a estos tres factores.

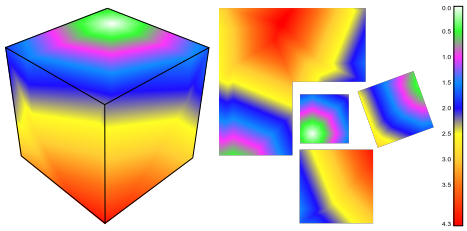


Figura 5.8 Campo de distancias calculado sobre la superficie de un cubo a partir de un téxel inicial seleccionado como entrada del algoritmo. Las dimensiones del cubo son $2\ m \times 2\ m \times 2\ m$. La paleta proporciona los colores empleados en la representación visual de los resultados

Utilizando la textura Topology generada por la solución propuesta, se ha calculado el campo de distancias en múltiples escenarios que mantienen el mismo téxel de entrada y modifican tres variables: el tamaño de la textura, la parametrización de la malla y el número de triángulos. El modelo 3D empleado en los cálculos es un cubo simple, de dimensiones $2\ m \times 2\ m \times 2\ m$, mostrado en la Figura 5.8, que permite obtener los resultados matemáticamente correctos de forma sencilla y compararlos con aquellos obtenidos por TTA. Concretamente, dado el mismo téxel como entrada, se calcula para cada téxel la diferencia entre la distancia obtenida por la solución propuesta y el valor matemáticamente correcto. El resultado de esa diferencia se compara con la mayor distancia obtenida por

el campo de distancias de la solución matemáticamente correcta con el fin de obtener el error relativo.

La Tabla 5.3 muestra el error relativo medio para cuatro casos. Todos ellos mantienen la forma de las cuatro islas de textura incluidas en la Figura 5.8 pero subdividen las caras utilizando distintos ratios. Caso C1 es la versión original con sólo 12 triángulos, dos por cara. Caso C2 subdivide los triángulos originales e incrementa su número hasta $12 \times 441 = 5292$. Caso C3, lo incrementa a $12 \times 26244 = 314928$ triángulos y Caso C4 a $12 \times 164025 = 1968300$. Estos cuatro casos también se evalúan utilizando tres tamaños de textura distintos: 2048×2048, 4096×4096, 8192×8192.

Tamaño	C1 12 t	C2 5292 t	C3 $3x10^5$ t	C4 $2x10^6$ t
2048	2.83 %	2.83 %	2.83 %	2.83 %
4096	2.82 %	2.82 %	2.82 %	2.81 %
8192	2.74 %	2.74 %	2.74 %	2.74 %

Tabla 5.3 La tabla muestra el error relativo medio entre los campos de distancias del método propuesto y la solución matemáticamente correcta para cuatro escenarios distintos y tres resoluciones de textura. En la cabecera se incluye el número de triángulos utilizado por cada caso

La Tabla 5.4 muestra el error relativo medio para cuatro casos que modifican la forma y el número de islas de textura mientras mantienen la misma triangulación de la superficie de la malla. Caso C5 contiene las cuatro islas mostradas en la Figura 5.8 y 5292 triángulos. Caso C6 divide las cuatro islas originales en 64 de menor tamaño con un número variable de triángulos. Igualmente, Caso C7 las divide en 292 y Caso C8 en 5291. Estos cuatro casos también son evalúan con tres tamaños de textura distintos: 2048×2048, 4096×4096, 8192×8192.

Tamaño	C5 4 i	C6 64 i	C7 292 i	C8 5291 i
2048	2.83 %	2.78 %	2.52 %	0.82 %
4096	2.82 %	2.80 %	2.63 %	1.36 %
8192	2.74 %	2.71 %	2.67 %	0.96 %

Tabla 5.4 La tabla muestra el error relativo medio entre los campos de distancia del método propuesto y la solución matemáticamente correcta para cuatro escenarios distintos y tres resoluciones de textura. En la cabecera se incluye el número de islas de textura (*i*) utilizado por cada caso

La Figura 5.9 compara el error relativo máximo y medio entre los casos que mantienen la forma de sus islas de textura pero difieren en el número de triángulos (C1-C4). La Figura 5.10 compara el error relativo máximo y medio entre los casos que mantienen su triangulación pero difieren en el número y la forma de sus islas de textura (C5-C8).

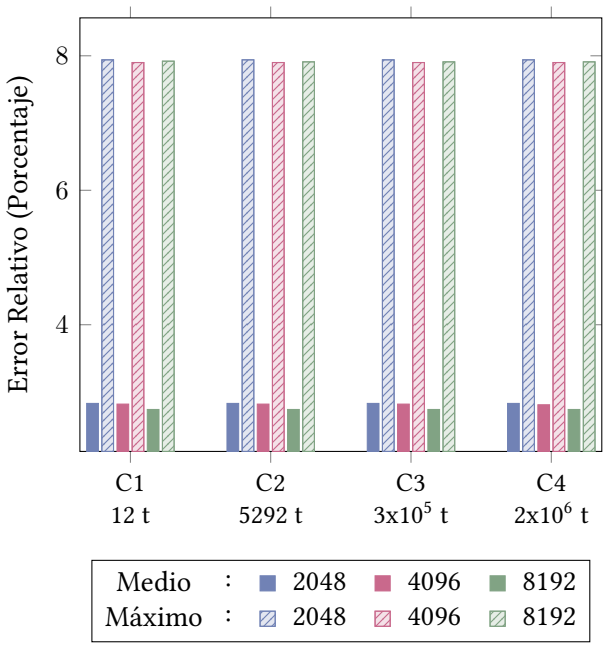


Figura 5.9 La gráfica muestra el error relativo máximo (barras sólidas) y medio (barras ralladas) de los casos C1, C2, C3 y C4 que comparten la forma de sus islas de textura pero difieren en el número de triángulos. Cada caso se evalúa con tres tamaños de textura distintos: 2048×2048, 4096×4096 y 8192×8192.

Después de un análisis detallado de estos resultados, se pueden extraer varias conclusiones interesantes. En primer lugar, los datos ofrecidos por la Tabla 5.3 demuestran que la solución propuesta es independiente de la densidad de la malla, dada una misma configuración de islas de textura. Una mayor densidad produce interpolaciones discretas entre aristas más pequeñas que podrían introducir imprecisiones. Sin embargo, ese no es el caso. La malla compuesta de 12 triángulos (C1) proporciona la misma precisión que la ofrecida por la malla compuesta de 2 millones (C4). Al mismo tiempo, resoluciones de textura más altas producen resultados ligeramente mejores. Como contrapartida, la resolución más baja, 2048×2048, es lo suficiente precisa a una fracción del coste en memoria.

En contrate con los resultados anteriores, los datos de la Tabla 5.4 muestran que la forma y el número de islas de textura tienen, como era de esperarse, el mayor impacto en los resultados finales. Dadas las trabas que el método propuesto tiene que superar debido a la discretización de la malla en el espacio de textura y las diferentes escalas y orientaciones entre las fronteras de islas vecinas, algunos podrían pensar que la solución aquí detallada ofrecería resultados peores cuando los tamaños de las islas de textura disminuyen y el número de estas se incrementa. En estos escenarios es necesario conectar islas distintas y computar interpolaciones discretas entre aristas más pequeñas con mayor frecuencia. De ahí que pudieran producirse imprecisiones mayores. Sin embargo, los casos C6, C7 y C8

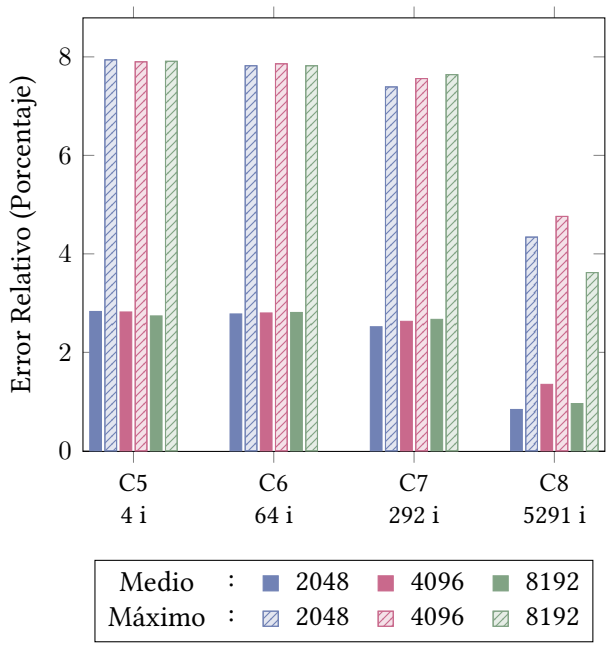


Figura 5.10 La gráfica muestra el error relativo máximo (barras sólidas) y medio (barras ralladas) de los casos C5, C6, C7 y C8 que comparten la misma triangulación de la superficie pero difieren en el número y la forma de sus islas de textura. Cada caso se evalúa con tres tamaños de textura distintos: 2048×2048, 4096×4096 y 8192×8192.

evaluados muestran lo contrario. El algoritmo propuesto es lo suficientemente robusto para proporcionar la información topológica necesaria y ayudar en el cálculo de resultados precisos, incluso cuando la malla se descompone en miles de islas de textura formadas por un único triángulo.

Los motivos son fáciles de entender. Dentro de las islas, la vecindad inmediata de cada téxel está formada por los ocho que lo rodean. Esta discretización produce bandas octogonales en lugar de las circulares ideales. Por tanto, las imprecisiones se acumulan en el espacio entre estas ocho direcciones. Descomponiendo las islas de textura en otras de menor tamaño con orientaciones distintas ayuda a disminuir el error porque la forma de las vecindades cambia y se introducen irregularidades en las bandas octogonales, acercándolas en mayor medida al caso matemáticamente correcto.

La Figura 5.11 muestra un ejemplo gráfico de esta ocurrencia. La primera columna muestra el campo de distancias producido por la solución matemáticamente correcta. La segunda columna muestra el campo de distancias obtenido por C1 en la parte superior y el error por téxel en la parte inferior. De igual forma, la tercera columna muestra el campo de distancias obtenido por C8 en la parte superior y el error por téxel en la inferior. Se puede apreciar con claridad que C8 ofece unos resultados muchos más próximos al caso matemáticamente correcto y que el algoritmo del cálculo de topología propuesto ayuda a

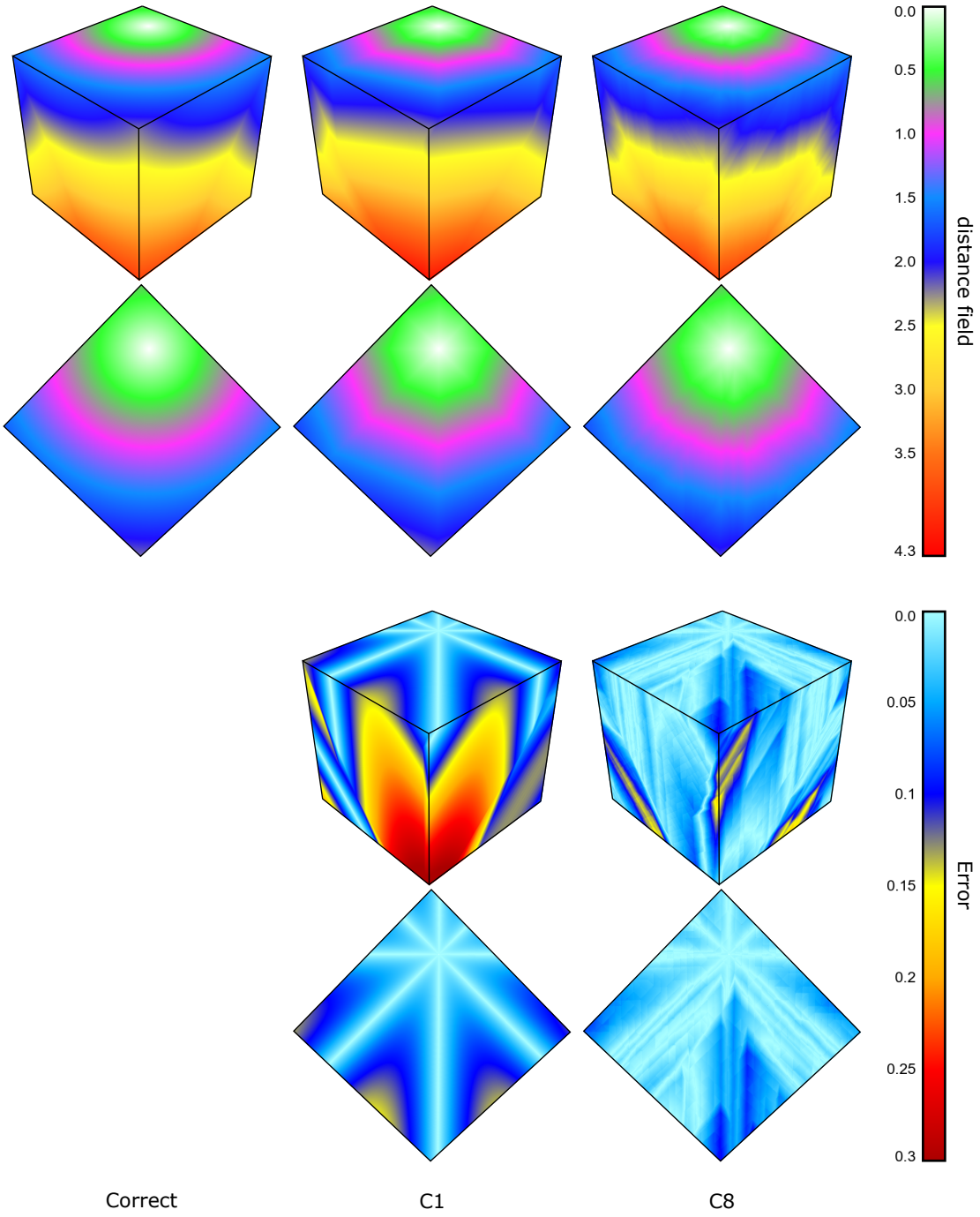


Figura 5.11 Dado un cubo de dimensiones $2\,m \times 2\,m \times 2\,m$ y el mismo téxel de entrada, la parte superior muestra los campos de distancia obtenidos por la solución matemáticamente correcta y los casos C1 y C8. También se ofrece una visión detallada de los resultados obtenidos en la cara inicial del cubo para cada uno de los casos. La paleta utilizada para visualizar los datos se muestra en la parte superior derecha. De igual forma, la parte inferior de la figura muestra el error generado por los casos C1 y C8 con respecto a la solución matemáticamente correcta. También se proporciona una visión detallada de la cara inicial y la paleta empleada.

mejorar las imprecisiones inherentes al uso de una vecindad discreta de ocho téxeles en los cálculos.

En términos de precision, la Figura 5.9 y la Figura 5.10 muestra que el error es inferior al ocho por ciento en el peor de los casos. En media se encuentra casi siempre por debajo del tres por ciento y, en el mejor de los casos, por debajo del uno por ciento. Dado que hemos establecido que estas imprecisiones se deben en parte al proceso de calcular los campos de distancia en sí mismo, se puede afirmar con cierta seguridad que el algoritmo propuesto es lo suficientemente preciso para usarse en escenarios reales.

5.4 Conclusiones

En este capítulo se ha detallado un nuevo método para computar la información topológica de mallas 3D en base a su parametrización. Este método utiliza la conectividad inherente de los téxeles que se encuentran en el interior de las islas e implementa un mecanismo para conectar las fronteras de islas vecinas y secciones inconexas de una misma isla. En términos de rendimiento, el algoritmo se resuelve completamente en el espacio de GPU, aprovechando al máximo su arquitectura paralela. Al contrario que otras estructuras de datos como octrees, este método es independiente de la configuración de triángulos de la malla 3D. Comparado con OCT-TR, puede llegar a calcular la información topológica hasta cinco órdenes de magnitud más rápido en los casos más complejos.

Además, es capaz de trabajar a un nivel de precisión bastante alto para mallas con una alta densidad de triángulos. De hecho, es lo suficientemente robusto para proporcionar la información topológica necesaria en casos donde la malla se descompone en miles de islas de textura compuestas de un único triángulo. Finalmente, su aplicación en escenarios reales, como puede ser el cálculo de campo de distancias, proporciona errores relativos medios por debajo del tres por ciento en media, demostrando lo fiable que es.

6

Lenguaje de operación entre capas de información

Tras describir un método capaz de almacenar información topológica de un modelo 3D en texturas 2D, se continúa expandiendo la funcionalidad de la arquitectura propuesta con un lenguaje de operación de capas. Al igual que las soluciones anteriores, el código generado por el reconocedor del lenguaje se ejecuta íntegramente en la GPU con el fin de calcular las operaciones de la forma más eficiente posible.

Una función esencial de todo sistema de información es poder realizar operaciones sobre los datos que contiene. El tipo de operaciones soportadas y cómo se pueden realizar depende de los objetivos fijados por el sistema en cuestión y la flexibilidad con la cual se ha diseñado. El tipo más común y directo de implementar son las operaciones de consulta. A través de una selección de herramientas de recuperación de información se pueden obtener datos específicos de interés en base al criterio de búsqueda seleccionado. Consultas como encontrar qué piezas se descubrieron en un yacimiento durante un determinado periodo de tiempo o qué secciones de una determinada obra pictórica fueron restauradas son algunos ejemplos de este tipo de operaciones.

Sin embargo, existen sistemas de información que pueden realizar más tipos de operaciones. Tal es el caso de los sistemas basados en capas cuyas estructuras y organización hacen posible el calculo de operaciones aritméticas o lógicas sobre los datos de sus capas. La presencia de este tipo de operaciones permite no sólo recuperar información, sino también analizar la existente para generar nuevos resultados en función de las operaciones aplicadas.

A menudo, este último tipo de sistemas cuentan con lenguajes de operación diseñados específicamente para trabajar con la estructura básica de almacenamiento del sistema. Estos lenguajes suelen tener una sintaxis sencilla, cuentan también con un conjunto variado de operaciones matemáticas y disponen de ciertas funciones de control de flujo. Los usuarios pueden usar expresiones complejas para escribir de forma concisa qué operaciones desean realizar sobre una o múltiples capas. Se evita así la necesidad de estar seleccionando individualmente operaciones sobre resultados parciales mediante interfaces gráficas de usuario que pueden ser ineficientes.

Un ejemplo de sistema con esta funcionalidad lo encontramos en *Geographic Resources Analysis Support System* (GRASS) **[SWG*89, NBLM12]**, un sistema de información geográfica utilizado habitualmente en la gestión y análisis de información geoespacial, modelado espacial, producción y visualización de mapas. Se trata de un complejo sistema estructurado en múltiples módulos que implementan todas esas funcionalidades.

Entre los módulos disponibles se encuentra **r.mapcalc [LST91, SW94]**, que permite realizar operaciones aritmeticológicas sobre la información de capas *raster* y, además, es capaz de generar otras nuevas mediante expresiones que utilizan capas existentes, funciones y constantes enteras o reales.

A lo largo de este capítulo se detalla un lenguaje de operación de capas inspirado en r.mapcalc para la arquitectura presentada en capítulos anteriores. Esta propuesta aprovecha la tecnología proporcionada por el generador de analizadores de lenguajes, *ANother Tool for Language Recognition* (ANTLR) **[PQ95, Par13]**, para poder crear un nuevo lenguaje que expanda la funcionalidad de r.mapcalc y permita trabajar directamente con las bases de datos asociadas a las capas. El reconocedor del lenguaje es capaz de traducir las sentencias escritas por el usuario a código eficiente de shaders que aprovecha la arquitectura paralela de las GPUs para acelerar el cálculo de todas las operaciones disponibles.

6.1 Cambios en la arquitectura

Para poder implementar la funcionalidad descrita en este capítulo es necesario incorporar una nueva entidad a la arquitectura inicial propuesta en el Capítulo 4. La entidad recibe el nombre de **Traductor del Lenguaje** y se encuentra en el espacio de CPU. En la Figura 6.1 se puede ver un esquema actualizado de esta nueva configuración del sistema.

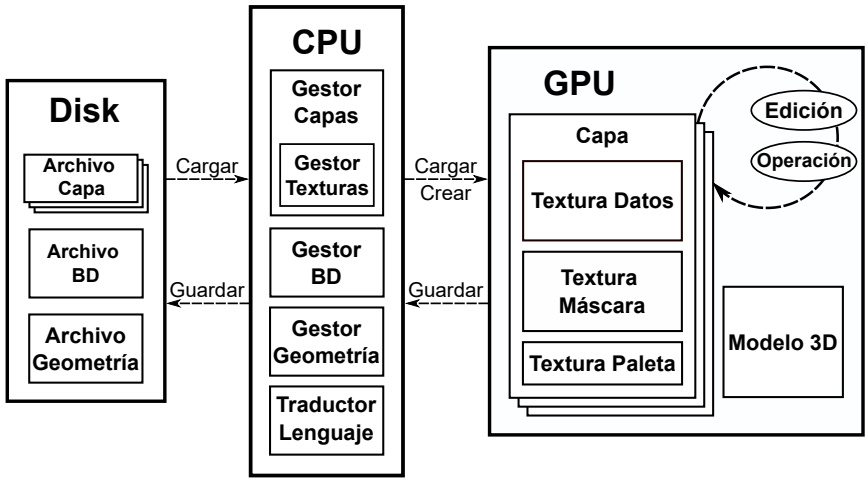


Figura 6.1 Visión general de la arquitectura revisada

Al contrario que el resto de entidades presentes en este espacio, el Traductor del Lenguaje no se limita a ejercer funciones de gestión y sus responsabilidades se extienden a las siguientes tareas:

- Analizar el conjunto de sentencias escritas por el usuario en el lenguaje de operación de capas.
- A partir de los resultados del análisis, generar el conjunto de sentencias equivalente en el lenguaje de Shading de OpenGL (GLSL). En otras palabras, realizar una traducción eficaz del lenguaje de operación propuesto a código de shader.
- Configurar la ejecución del shader de computación que resuelve íntegramente en el espacio de GPU las operaciones expresadas por el usuario.

Por otro lado, debido a su importancia en las siguientes secciones, se vuelve a recordar la implementación de las capas en el sistema, compuestas por dos texturas 2D y una textura 1D:

- Textura de Datos (2D): almacena un número por téxel. Los tipos de valores soportados son dos, enteros (8, 16 o 32 bits) y reales (16 o 32 bits). Esta textura almacena la

información real de la capa. En el caso de las capas numéricas son valores de propiedades, mientras que, en el caso de la capas de base de datos, son llaves primarias.

- Textura de Máscara (2D): almacena un valor booleano por téxel para determinar si el téxel de la Textura de Datos contiene o no información válida.
- Textura de Paleta (1D): almacena la información necesaria de color por téxel para visualizar el contenido de la Textura de Datos. Cada téxel contiene un vector de cuatro componentes con el siguiente formato de color: rojo, verde, azul y opacidad.

6.2 Traductor del lenguaje

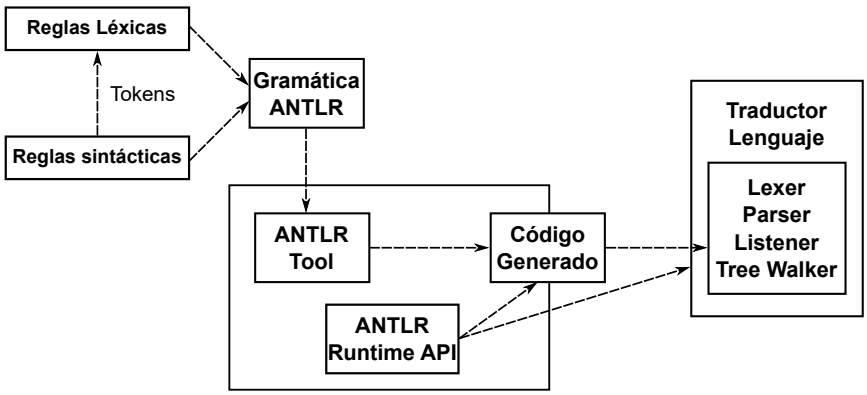


Figura 6.2 ANTLR 4 recibe como entrada una gramática libre de contexto que especifica un lenguaje mediante reglas léxicas y sintácticas. Genera como salida la implementación del reconocedor del lenguaje que cuenta con un analizador léxico (*Lexer*), un analizador sintáctico (*Parser*), un explorador de árboles sintácticos (*Tree Walker*) y escuchadores de eventos (*Listener*) o visitantes (*Visitor*)

ANTLR 4 es un generador de analizadores de lenguajes, es decir, es una herramienta que recibe como entrada una gramática y genera el código fuente del reconocedor del lenguaje especificado por dicha gramática. En la Figura 6.2 se describe el flujo de trabajo habitual empleado por esta herramienta.

El primer paso del proceso es crear la gramática de entrada que describe el lenguaje que se quiere diseñar. A su vez, la gramática se especifica a través de un conjunto de reglas léxicas o léxico y un conjunto de reglas sintácticas o sintaxis. Puede escribirse de forma separada en dos archivos o juntas en un único archivo.

El conjunto de reglas léxicas describe aquellas cadenas de caracteres con significado propio dentro del lenguaje conocidas como componentes léxicos (*tokens*). Tradicionalmente se especifican utilizando patrones o expresiones regulares que definen palabras

reservadas, identificadores, operadores, signos de puntuación y cualquier otro elemento común de lenguajes de programación.

El conjunto de reglas sintácticas define la estructura jerárquica de un lenguaje de programación. Se especifican utilizando un símbolo inicial o axioma, símbolos terminales o componentes léxicos, símbolos no terminales y producciones. Una producción consta de un símbolo no terminal en su parte izquierda, una flecha y un conjunto de símbolos terminales y no terminales en su parte derecha. Cada producción describe una regla sintáctica y el conjunto de ellas define la sintaxis del lenguaje.

ANTLR 4 admite como entrada cualquier tipo de gramática libre de contexto que no presente ninguna recursión por la izquierda indirecta o escondida. Puesto que la solución propuesta en este capítulo intenta replicar la funcionalidad de r.mapcalc, tanto el léxico como la sintaxis del lenguaje de operación de capas son una recreación fiel de las descritas por este módulo, salvo algunas diferencias de notación. Sin embargo, la sintaxis extiende la funcionalidad con los siguientes añadidos:

- Función sql: proporciona un simple protocolo para poder establecer una comunicación con la tabla asociada a una capa de base de datos. Crea una Textura de Datos que rellena con la información de un campo numérico de la tabla. Necesita dos parámetros. El primero se corresponde con el nombre de una capa y el segundo, con el nombre del campo númerico de la tabla asociada a la capa especificada por el primer parámetro.

- Operador #: operador binario que sólo puede aparecer en la parte izquierda de una operación de asignación. Asocia una paleta del sistema a la capa que ha sido creada o sobrescrita por la sentencia de asignación. El operando izquierdo se corresponde con el nombre de la capa, mientras que el operando derecho se corresponde con el nombre de la paleta.

- Operador [entero]: como el operador anterior sólo puede aparecer en la parte izquierda de una operación de asignación. Establece la resolución de las texturas de la capa resultante. Común en casos donde se quiere crear capas con un único valor constante y el analizador sintáctico no puede inferir la resolución.

En el Apéndice C se puede encontrar una descripción detallada del léxico y la sintaxis que definen el lenguaje de operación de capas.

A partir de la gramática que recibe como entrada, ANTLR 4 genera el código correspondiente al reconocedor del lenguaje especificado por ella y puede hacerlo en multitud de lenguajes: Java, Javascript, Python, C++, C#, etc. A su vez, el reconocedor está compuesto por un analizador léxico (*Lexer*) y un analizador sintáctico (*Parser*). En concreto, ANTLR 4 genera analizadores sintácticos descendentes recursivos que utilizan una función de predicción de producciones LL(*) Adaptativa o ALL(*) **[PHF14]**.

El analizador léxico descompone en *tokens* la sentencia recibida como entrada. Mientras tanto, el analizador sintáctico construye un árbol sintáctico (*Parse Tree*) que almacena la estructura de la sentencia y las expresiones componentes. Estás últimas están definidas

por símbolos no terminales y sus nombres coinciden con aquellos que aparecen en la regla sintáctica de la que forman parte. En el nodo raíz se encuentra la expresión más abstracta y los nodos hoja contienen únicamente *tokens*.

La implementación del reconocedor del lenguaje cuenta con con otros tres componentes adicionales:

- **Explorador de árboles sintácticos** (*Parse Tree Walker*): entidad que sabe como recorrer el árbol generado por el analizador sintáctico tras analizar una secuencia de entrada escrita en el lenguaje. Cada vez que explora un nodo dispara una serie de eventos que provocan la ejecución de métodos específicos de los escuchadores de eventos. Los eventos más comunes están asociados a la entrada y salida del nodo explorado y permiten realizar acciones antes de analizar el nodo y después de haberlo hecho respectivamente.

- **Escuchador de eventos** (*Listener*): entidad que se encarga de implementar las acciones asociadas al disparo de un determinado evento generado por el explorador de árboles sintácticos. Describe, por tanto, la respuesta que se va a generar ante una determinada cadena de entrada.

- **Visitante** (*Visitor*): entidad que cumple un propósito similar a los escuchadores de eventos pero que, al contrario que estos últimos, recorre el árbol utilizando llamadas explicitas de visita sobre los nodos hijos. En caso de no hacerlo, el subárbol asociado a ese nodo hijo quedaría inexplorado.

En en sistema propuesto, el Traductor del Lenguaje se ha creado a partir del esqueleto que ANTLR 4 ha generado de la clase del escuchador de eventos y es capaz de traducir sentencias del lenguaje de operación de capas a GLSL. Este proceso de traducción garantiza que se utilizan únicamente los recursos necesarios:

- En caso de ser necesario, carga en memoria de GPU las texturas de la capas involucradas en las operaciones de forma automática.

- Gestiona la creación de las texturas temporales empleadas en ciertas operaciones de capas de base de datos.

- Encola la destrucción de todas las texturas temporales y esta se inicia una vez que el shader ha terminado su ejecución.

Asimismo, el Traductor del Lenguaje trabaja principalmente con un conjunto de plantillas de bloques de código GLSL que implementan una expresión, función u operación escrita en el lenguaje de operación de capas. Los parámetros de estas plantillas se actualizan utilizando estructuras de contexto que los escuchadores de eventos de ANTLR facilitan para compartir información relevante entre los nodos del árbol sintáctico.

6.3 Generación del código de shader

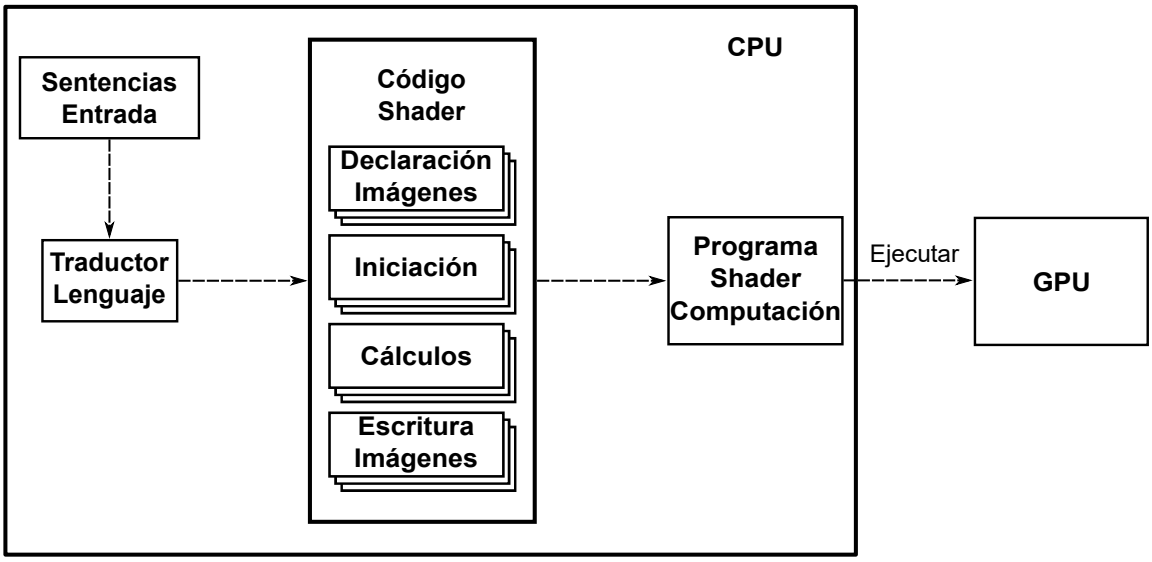


Figura 6.3 El Traductor del Lenguaje utiliza un conjunto de plantillas para generar el código de shader asociado a sentencias expresadas en el lenguaje de operación de capas. La traducción se organiza en torno a cuatro secciones: Declaración de Imágenes, Iniciación, Cálculos y Escritura de Imágenes

Dada una entrada válida escrita por el usuario en el lenguaje de operación de capas que se ha diseñado, el Traductor del Lenguaje genera código de shader siguiendo la estructura mostrada en la Figura 6.3, crea un shader de computación y lo envía a la GPU para su ejecución.

Hay que tener en cuenta que los shaders de computación no disponen de variable de salida, al contrario que los shaders del cauce de representación. Por consiguiente, no pueden escribir directamente en la textura y necesitan realizar una operación de lectura/escritura de imágenes en su lugar. Si consideramos las texturas como contenedores que pueden albergar una o más imágenes con el mismo formato, esto significa que los shaders de computación pueden trabajar con cada una de ellas individualmente pero no al mismo tiempo. En lo que al método propuesto respecta, esto no supone ningún problema porque las texturas que utiliza sólo contienen una única imagen, es decir, no usan mip-mapping.

El código de shader GLSL generado está organizado en torno a cuatro secciones distintas: Declaración de Imágenes, Iniciación, Cálculos y Escritura de imágenes.

Con el fin de facilitar la comprensión del proceso de traducción, se va a utilizar la siguiente sentencia como ejemplo sencillo de análisis:

```
Target = SourceA + 4 * SourceB - sql(SourceC, Num);
```

donde $SourceA$, $SourceB$ y $SourceC$ son nombres de capas válidos; $SourceC$ es, además, una capa de base de datos; Num es un campo de la base de datos asociada a $SourceC$ y $Target$ es la nueva capa creada por esta expresión.

6.3.1 Declaración de Imágenes

Cada vez que el explorador de árboles analiza un identificador, el Traductor del Lenguaje contacta con el Gestor de Capas para comprobar si se trata de un nombre de capa válido. El Gestor de Capas verifica entonces que la capa existe dentro del proyecto actual y, en caso de que sea necesario, la lee de disco y la transfiere a memoria de GPU. Al final de este proceso el traductor recibe el manejador de la capa en cuestión.

Con el manejador disponible, el Traductor del Lenguaje dispone de todo lo que necesita para completar la plantilla de declaración de imágenes con la información adecuada:

```
1 layout(formato, binding=unidad) uniform gimage2DArray nombre;
```

Dado que las capas están compuestas por la Textura de Datos y la Textura de Máscara, esta plantilla se aplica dos veces por cada manejador.

Cuando se pretenden realizar lecturas, las declaraciones de variables de imágenes en GLSL requieren de un calificador *formato*. Dicho calificador define cómo el shader interpreta los bits que lee de la imagen declarada. Por consiguiente, el Traductor del Lenguaje comprueba el tamaño del formato interno de las texturas y lo transforma a la cadena adecuada del calificador de formato.

Antes de que las imágenes puedan ser utilizadas por los shaders de computación, éstas necesitan asignarse a distintas unidades de imágenes en la GPU. Este trabajo es gestionado por el Gestor de Capas de parte del Traductor del Lenguaje y se solicitan dos posibles unidades de imagen para las texturas de las capas, siempre que éstas no hayan sido ya asignadas previamente a otras unidades. El Traductor del Lenguaje transforma la *unidad* solicitada a la cadena correspondiente.

Finalmente, dado que las variables de tipo imagen están organizadas en tres formatos generales, se necesita utilizar el prefijo *g* para especificarlos. El prefijo *i* representa a enteros con signo, el prefijo *u* representa a enteros sin signo y la ausencia de prefijo representa a los números reales, que son el tipo de datos utilizado por defecto. En consecuencia, el Traductor del Lenguaje comprueba el tipo de las texturas y lo transforma a la cadena de texto apropiada.

Una vez que se ha rellenado la plantilla de declaración, el código resultante es añadido a un buffer de declaración de imágenes. Volviendo a la sentencia del ejemplo utilizado, la declaración de $SourceA$ podría ser la siguiente:

```
1 layout(r32f, binding=1) uniform image2DArray SourceAData;
2 layout(r8, binding=2) uniform image2DArray SourceAMask;
```

Las capas de bases de datos constituyen una excepción a este procedimiento. Normalmente sólo aparecen como parámetro de la función **sql** y están acompañadas por un campo numérico de la tabla de la base de datos que tienen asociada. En estos casos, las llaves primarias almacenadas en la Textura de Datos no son relevantes para la solución porque dicha información siempre se corresponde con el campo seleccionado. En la sentencia de ejemplo se trata del campo Num de la tabla asociada a $SourceC$.

Por consiguiente, cuando el explorador del árbol analiza la función **sql**, el Traductor del Lenguaje contacta con el Gestor de Base de Datos para recuperar la información del campo solicitado mediante la sentencia SQL apropiada. A continuación, el Traductor del Lenguaje envía esta información al Gestor de Capas y solicita la creación de una capa temporal que utiliza la misma máscara de la capa de base datos analizada, pero reemplaza la Textura de Datos con la información recuperada. La declaración de la variable de imagen asociada a esta nueva textura se realiza de la misma manera que se ha descrito en párrafos anteriores.

Los identificadores de capas situados a la izquierda de una operación de asignación constituyen la otra excepción. Dichos identificadores representan capas nuevas que se han de crear y que almacenarán los resultados de las operaciones descritas a la derecha del operador de asignación. Por tanto, el Traductor del Lenguaje notifica al Gestor de Capas que es necesario crear las capas correspondientes antes de rellenar la plantilla de declaración de imágenes con su información.

6.3.2 Iniciación

Después de añadir el código de declaración de imágenes al buffer de declaraciones, el Traductor del Lenguaje rellena la plantilla de carga de imágenes de forma ordenada. De esta manera se inician las variables locales con los valores de los píxeles de la imagen indexados por las coordenadas especificadas:

```
1  tipoGL value = imageLoad(nombre, ivec3(coords, índice)).x;
```

Una vez más, la plantilla se rellena para la Textura de Datos y la Textura de Máscara de cada capa o identificador.

En primer lugar, el Traductor del Lenguaje comprueba el formato interno de las texturas y lo traduce a tipos escalares básicos en *tipoGL*. Continuando con la sentencia del ejemplo, el formato de $SourceAData$ es *r32f* o un único real de 32 bits y, en consecuencia, el tipo escalar de su variable local es *float*.

El siguiente paso consistiría en escribir *nombre* de la imagen declarada en el paso anterior. Siguiendo el caso del ejemplo propuesto, $SourceAData$ sería la respuesta.

Finalmente, el Traductor del Lenguaje comprueba el índice de la textura dentro del texture array que la contiene y actualiza *índice* en consecuencia.

Una vez se ha rellenado la plantilla, el código resultante se incorpora al buffer de iniciación. En la sentencia de ejemplo, la iniciación de $SourceA$ podría ser la siguiente:

```
float data = imageLoad(sourceAData, ivec3(coords, 0)).x;
float mask = imageLoad(sourceAMask, ivec3(coords, 3)).x;
```

6.3.3 Cálculos

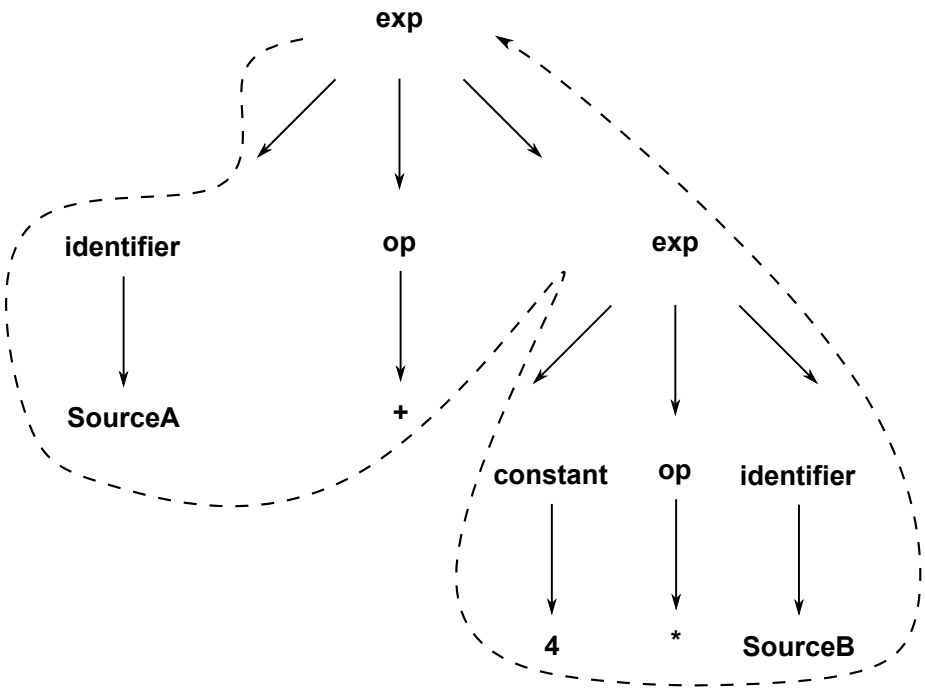


Figura 6.4 Orden de análisis de esta subsección del ejemplo

Cada vez que el explorador del árbol analiza un nodo que involucra operadores, una función o un literal, el Traductor del Lenguaje añade nuevas sentencias al buffer de cálculos. La Figura 6.4 describe el orden de análisis de esta subsección del árbol en el ejemplo propuesto.

Aunque cada operación o función del lenguaje dispone de una plantilla única, su estructura sigue un patrón similar. Estos nodos siempre producen nuevos valores y, como resultado, es necesario declarar e iniciar dos variables locales para almacenarlos. Una de ellas almacena el valor de los datos y la otra, el valor de la máscara. El tipo de la variable que almacena este último valor es siempre un booleano, mientras que el tipo de la variable que almacena los datos necesita ser deducido. Para poder realizar esta deducción, es necesario que la información del operando o el parámetro esté disponible.

Afortunadamente, ANTLR proporciona estructuras conocidas como contexto que facilitan este tipo de información entre los nodos conectados. En la solución propuesta se han configurado para que proporcionen los nombres de las variables de máscara y datos locales, así como el tipo de la variable de datos y la resolución de las texturas.

Con la información proporcionada por los contextos, el Traductor del Lenguaje puede deducir el tipo de la nueva variable que almacena los datos. Una conversión implícita es necesaria en aquellos casos donde el tipo de los operandos difiera. Si se utiliza como ejemplo el nodo $4 * SourceB$ de la Figura 6.4 y se supone que el tipo de $SourceB$ es real, la nueva variable necesita ser un real también para almacenar los datos con garantías.

Dado que estas variables usan nombres genéricos, es posible que se produzcan colisiones de nombres entre las mismas. Con el fin de evitar estos casos indeseados, el Traductor del Lenguaje gestiona dos contadores globales cuyo valor es utilizado como sufijo de estas declaraciones.

Sirva de ejemplo la plantilla de resolución de operaciones binarias que se muestra a continuación:

```
1  bool expMaskn;
2  tipoOpBin expDatam;
3
4  if (contextA.Mask && contextB.Mask)
5  {
6      expDatam = contextA.Data op contextB.Data;
7      expMaskn = true;
8  }
9  else
10 {
11     expMaskn = false;
12 }
```

El Traductor del Lenguaje utiliza el contador de máscaras para sustituir *n* y generar el nombre de la variable que almacena la máscara resultado, expMask*n*. Análogamente, utiliza el contador de datos para sustituir *m* y generar el nombre de la variable que almacena el dato resultante, expData*m*. Con *contextA.Mask* y *contextB.Mask* accede a los contextos de los operandos y recupera el nombre de las variables de máscara asociadas a cada uno de ellos. De esta manera se verifica que contienen datos válidos y, en caso afirmativo, se realiza la operación binaria especificada por el operador *op*, cuyo resultado se almacena en la recién creada expData*m*. Para completarla necesita recuperar los nombres de las variables de datos con ayuda de los contextos, *contextA.Data* y *contextA.Data*. Finalmente, se modifica la variable recién creada expMask*n* en concordancia.

Una vez que la plantilla apropiada se ha rellenado, el Traductor del Lenguaje guarda la información del nodo actual analizado en su correspondiente contexto con el fin de que pueda ser consultada en el futuro.

En el árbol representado en la Figura 6.4, las variables de máscara y datos de la expresión $4 * SourceB$ y el tipo de su correspondiente variable de datos se escriben en el contexto que $SourceA + exp$ utilizará para solventar esta subsección del árbol.

6.3.4 Escritura de Imágenes

Los resultados de procesar todos los cálculos realizados por sentencias complejas del lenguaje se almacenan en variables locales hasta esta última etapa. Sin embargo, para que sean persistentes, necesitan escribirse en las texturas de las capas y, para lograrlo, el Traductor del Lenguaje rellena la siguiente plantilla de escritura de imágenes:

```
1 imageStore(nombreImagen, ivec3(coords, índice), gvec4(datos));
```

Una vez más, la plantilla se completa con la información de la Textura de Datos y la Textura de Máscara de cada identificador.

El Traductor del Lenguaje reemplaza *nombreImagen* con la variable de imagen apropiada que se declaró en la primera etapa; *índice*, con el índice de la textura en el texture array; *g*, con el prefijo que ha resultado de deducir el tipo de la variable local que almacena el resultado final; y *datos*, con el nombre de la última variable.

Si la capa de destino no existía con anterioridad, esta etapa crea una nueva capa que va a contener los resultados de los cálculos descritos en la parte de derecha del operador de asignación. En cualquier otro caso, sobrescribe la información anterior de un capa existente.

Una vez que la plantilla se ha rellenado por completo, el código resultante se añade al buffer de escritura. En la sentencia de ejemplo, las escrituras correspondientes a $SourceA$ podrían realizarse de la siguiente manera:

```
1 imageStore(SourceAData, ivec3(coords, 3), vec4(binaryData2));
2 imageStore(SourceAMask, ivec3(coords, 2), vec4(binaryMask2));
```

6.3.5 Ejecución del código generado

El código fuente del shader de computación es el resultado de la unión de los buffers actualizados por cada etapa: Declaración de Imágenes, Iniciación, Cálculos y Escritura.

El último paso de la solución propuesta compila el shader de computación, lo enlaza a un programa objeto y procede a enviárselo a la GPU para su ejecución. Para poder realizar esta última acción, la orden de envío necesita especificar cómo ha de procesar el trabajo la GPU.

El principio detrás de los shaders de computación es divide y vencerás. Concretamente, el dominio de trabajo se divide en múltiples grupos de trabajo del mismo tamaño de grupo para procesar una tarea. De esta manera cada grupo de trabajo puede ejecutarse independientemente de los otros y, por tanto, muchos de ellos pueden hacerlo también de forma concurrente, aprovechando el paralelismo de la arquitectura de las GPUs para acelerar el proceso de cálculo. La Figura 6.5 muestra una representación gráfica de este concepto.

En el caso particular de la solución propuesta, el dominio de trabajo es una textura bidimensional; en concreto, cada Textura de Datos y Textura de Máscara de las capas

involucradas en los cálculos. Por consiguiente, el tamaño del grupo es también bidimensional y una potencia de dos para alinearse correctamente con el tamaño de las texturas. Definido el tamaño del grupo, se puede dividir el dominio de trabajo por dicho valor, obteniendo el número de grupos. Con estos valores se puede realizar la orden de envío del shader de computación a la GPU y completar su ejecución.

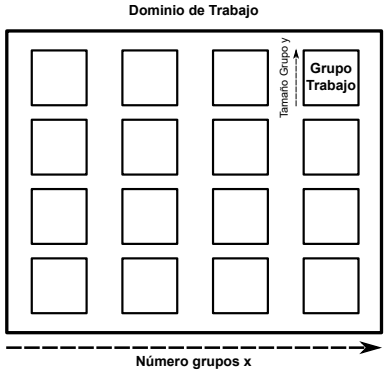


Figura 6.5 Dominio de trabajo de un shader de computación

6.4 Resultados

A continuación se describe un ejemplo que muestra de forma gráfica la aplicación de una operación capaz de discriminar los valores almacenados en una capa de información.

Supóngase que un proyecto dispone de una capa que contiene el campo de distancias calculado sobre toda la superficie del modelo 3D. La Figura 6.6 muestra una captura de este escenario. El usuario sólo está interesado en conocer la superficie del modelo que se encuentra a una determinada distancia del área utilizada como entrada para el cálculo del campo de distancias. Esta operación se puede aplicar con un sentencia similar a la que se muestra a continuación:

```
Band = if (DistanceField > 400, DistanceField);
```

La funcion **if** comprueba si los valores de la capa DistanceField se encuentran a una distancia superior a $400\ mm$. En caso afirmativo, conservan su valor y se asignan a la capa resultado. En caso contrario, se descartan.

Una vez que el usuario solicita su ejecución, el Traductor del Lenguaje analiza la sentencia y comprueba que debe generar la capa Band para almacenar los resultados, por lo que solicita su creación al Gestor de Capas. Al completar el análisis genera el siguiente código de shader:

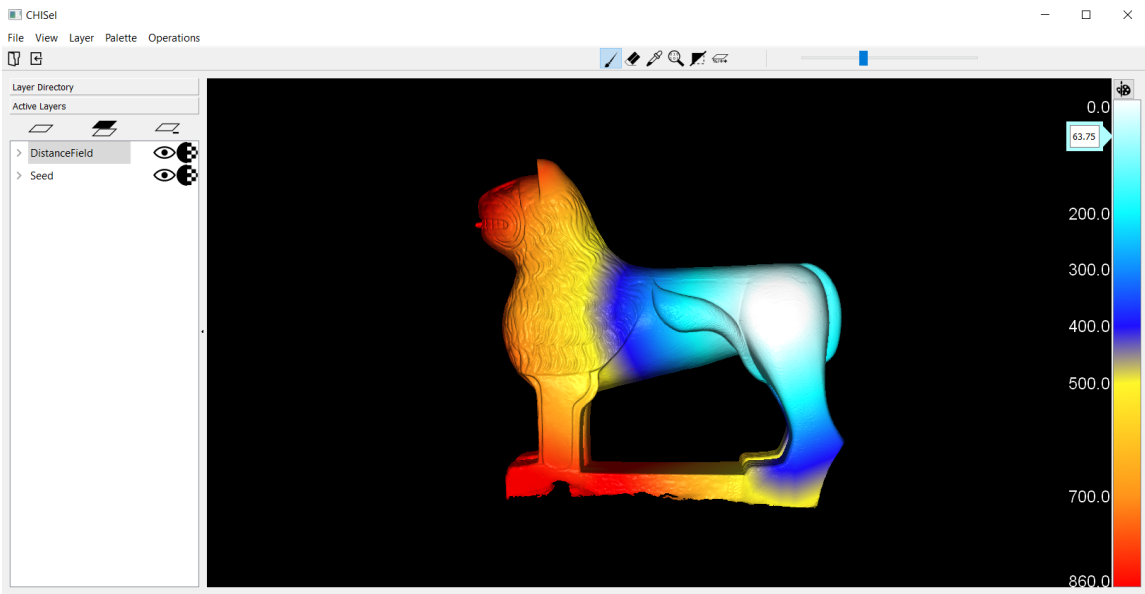


Figura 6.6 Captura del prototipo donde se trabaja en un proyecto que contiene una capa con un campo de distancias calculado sobre todo el modelo

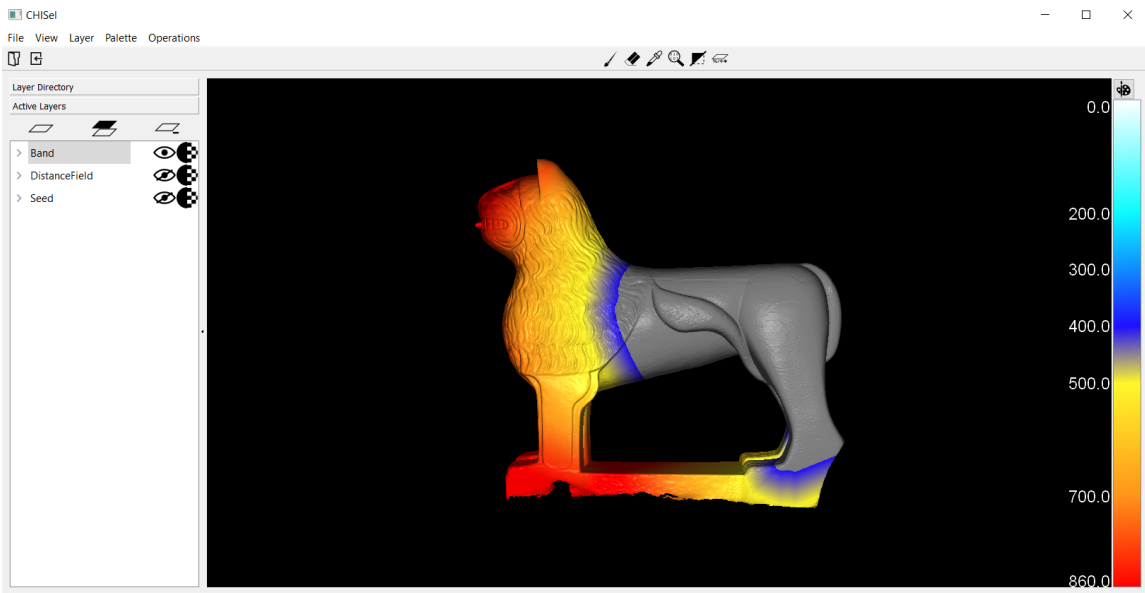


Figura 6.7 La capa Band contiene el resultado de aplicar una operación sobre la capa DistanceField, vista en la Figura 6.6, que elimina todas aquellas secciones del modelo que se encuentren a una distancia igual o inferior a 400 mm

```
#extension GL_ARB_compute_variable_group_size : enable

layout(local_size_variable) in;
layout(r32f, binding=1) uniform image2DArray Data1;
layout(r8, binding=2) uniform image2DArray Mask2;

void main()
{
    ivec2 pixelCoords = ivec2(gl_GlobalInvocationID.xy);
    float DistanceFieldData = imageLoad(Data1,
                                        ivec3(pixelCoords, 0)).x;
    float DistanceFieldMask = imageLoad(Mask2,
                                        ivec3(pixelCoords, 3)).x;
    bool expMask0;
    bool expData0;
    if (DistanceFieldMask > 0 && true)
    {
        expData0 = DistanceFieldData > 400;
        expMask0 = DistanceFieldData > 400;
    }
    else
    {
        expMask0 = false;
    }

    bool ifMask1;
    float ifData1;
    if (expMask0)
    {
        if (expData0)
        {
            ifData1 = DistanceFieldData;
            ifMask1 = DistanceFieldMask > 0;
        }
        else
        {
            ifData1 = 0;
            ifMask1 = true;
        }
    }
    else
    {
        ifMask1 = false;
    }
    if (ifMask1)
    {
        imageStore(Data1, ivec3(pixelCoords, 1), vec4(ifData1));
        imageStore(Mask2, ivec3(pixelCoords, 4), vec4(ifMask1));
    }
}
```

A continuación, el Traductor del Lenguaje envía el programa con el shader de computación a la GPU para su ejecución. Una vez concluida la operación, el sistema hace visible la capa que contiene los resultados. En la Figura 6.7 se puede observar de forma gráfica el contenido de la nueva capa Band.

Aunque se trata un ejemplo simple que describe los distintos pasos del proceso, permite vislumbrar la versatilidad y economía de acciones ofrecida por los lenguajes de operaciones de capas. Por otro lado, el prototipo soporta programas que contienen múltiples sentencias y definen conjuntos de operaciones complejas.

6.5 Conclusiones

En este capítulo se ha diseñado un versátil lenguaje de operación de capas. Está inspirado en el lenguaje ofrecido por el módulo r.mapcalc, pero expande su funcionalidad para poder trabajar con tablas de capas de base de datos y controlar la visualización de los resultados con las paletas. La gramática que define el lenguaje se ha utilizado como entrada de ANTLR 4, una herramienta capaz de generar analizadores de lenguajes. A partir de la salida generada por esta herramienta se ha construido un traductor con las siguientes características:

- Dispone de un sistema de plantillas que le permite traducir las sentencias escritas en el lenguaje original a código eficiente de shader que se ejecuta íntegramente en GPU.
- Carga en memoria de GPU las texturas de aquellas capas que intervienen en los cálculos pero no están en uso en el momento de la operación.
- Gestiona automáticamente la creación de los recursos temporales que se necesitan para operar con los campos de las tablas asociadas a las capas de base de datos.
- Destruye los recursos temporales una vez dejan de ser necesarios.

7

Estudio de usabilidad

Todo el trabajo desarrollado hasta el momento se ha ido implementando en un prototipo funcional que cuenta con una interfaz de usuario avanzada. A lo largo de este capítulo se describe el estudio de usabilidad realizado para evaluar la calidad de la interfaz de este sistema especializado.

Los sistemas de información creados para la documentación de patrimonio histórico son herramientas complejas que manejan una gran variedad de datos heterogéneos. Aunque carecen de la especificidad proporcionada por herramientas especializadas, ofrecen soluciones generales que centralizan en un solo lugar las funcionalidades necesarias para procesar y trabajar con toda esa variedad de datos.

Diseñar una interfaz de usuario simple y accesible no es una tarea sencilla. Cuando la interfaz necesita permitir la inserción de información sobre la superficie de modelos 3D, soportar multitud de contenidos multimedia (fotos, vídeos, documentos, etc.), presentar de forma intuitiva grandes cantidades de información que ayuden a los usuarios a tomar decisiones correctas, editar contenidos en tablas de bases de datos o realizar operaciones aritmeticológicas entre capas de información, el trabajo resulta especialmente complejo. Además, se ha de tener en cuenta que, dentro del área de patrimonio histórico, el público al que va dirigido este tipo de sistemas puede no estar especialmente versado en el manejo de cierto tipo de tecnologías y, por tanto, es importante adecuar el diseño a sus necesidades.

Durante las últimas décadas se han realizado multitud de estudios de usabilidad de sistemas de información relacionados con el área de patrimonio histórico. Phiri et al. **[PWR*12]** propone un estudio de una biblioteca digital que utiliza un sofisticado sistema de almacenamiento jerárquico de objetos digitales con el fin de simplificar la arquitectura software del repositorio. Ruthven et al. **[RC15]** explora los retos de crear y gestionar sistemas de información de patrimonio histórico, ofreciendo estrategias generales para mejorar su usabilidad y el compromiso de los usuarios. Muqtadiroh et al. **[MADA17]** evalúa WikiBudaya, una enciclopedia web que proporciona información y conocimiento acerca de la cultura indonesia. Hu **[Hu18]** estudia la usabilidad de la biblioteca digital e-Dunhuang que pretende preservar el patrimonio histórico de esta región y dar a conocer su importancia en la Ruta de la Seda. Diulio et al. **[DGGG21]** realiza un estudio sistemático de la usabilidad y las estrategias de experiencia de usuario empleadas en sistemas orientados a datos dentro del marco de patrimonio histórico.

7.1 Usabilidad de CHISel 3.0

A lo largo de este trabajo se ha ido desarrollando un prototipo funcional que, entre otras cosas, implementa toda la funcionalidad descrita en capítulos anteriores. Este sistema de información se conoce como *Cultural Heritage Information System based on Extended Layers 3.0* o, simplemente, CHISel 3.0. Su código fuente es público y se puede encontrar en el siguiente repositorio de Github **[CHI23]**.

En este capítulo se describe el estudio de usabilidad que se ha realizado de esta herramienta con ayuda de un experto internacional. Entre los objetivos propuestos por el estudio se encuentran:

- Encontrar un grupo significativo de participantes cuyo perfil sea lo más cercano posible al usuario potencial de la herramienta.
- Determinar aquellas áreas de la interfaz de usuario con inconsistencias de diseño y problemas de usabilidad. Fuentes potenciales de errores pueden incluir:
  - Problemas de navegación: fallo al localizar funciones, número excesivo de pulsaciones de teclas o botones de ratón, fallos a la hora de seguir el flujo de trabajo recomendado, etc.
  - Problemas de presentación: fallo para actuar sobre la información deseada que se presenta por pantalla, errores de selección debido a etiquetados ambiguos, etc.
  - Problemas de uso: utilización inapropiada de campos de entrada, barra de herramientas, etc.
- Establecer unos niveles básicos de satisfacción y rendimiento de usuarios para futuras prueba de usabilidad.

7.1.1 Interfaz de usuario

La interfaz de usuario de CHISel 3.0 está dividida en las cuatro secciones que se detallan a continuación y que se pueden ver en la Figura 7.1:

- **Widget de Capas**: ocupa la parte izquierda de la interfaz e incluye las acciones más frecuentes de las capas: crear, cargar, borrar, eliminar y duplicar. También dispone de herramientas para modificar su visualización: nivel de opacidad, estado de visibilidad y orden de visualización. Asimismo, las capas de bases de datos disponen de una opción para mostrar la tabla asociada e interactuar con sus contenidos.
- **Barra de herramientas**: ocupa la parte superior de la interfaz y dispone de una selección de herramientas para la manipulación y consulta de datos de las capas: añadir, borrar, copiar y consultar valores. Cuenta también con un deslizador para modificar el tamaño de varias de estas herramientas.
- **Visualizador**: ocupa la parte central de la interfaz y muestra el modelo 3D sobre el cual se trabaja en el proyecto actual. Es aquí donde se aplican las acciones asociadas con la herramienta que se encuentra seleccionada, como añadir información a la capa actual o borrarla. También es donde se ven reflejados los cambios de visualización de estas últimas que se han seleccionado en el Widget de Capas.
- **Widget de Paletas**: ocupa la parte derecha de la interfaz y gestiona las operaciones relacionadas con las paletas. Por defecto, representa en vertical una visualización de la paleta actual con los puntos de control que la definen. En caso de que no puedan visualizarse todos los puntos por falta de espacio, se muestra sólo una selección de

los mismos. Contiene también un editor de paletas que permite modificar el color y el valor numérico de los puntos de control definidos, así como incorporar otros nuevos o eliminar los innecesarios. Cuenta además con una colección de paletas del sistema que permite asociar paletas preexistentes a la capa actual.

- **Widget del Valor actual**: se encuentra a la izquierda del Widget de Paletas y muestra el valor numérico que se quiere asociar a la capa seleccionada cuando el modo de edición se encuentra activo. El usuario puede insertar manualmente el valor o arrastrar el widget a lo largo del rango de valores contenidos en la paleta actual.

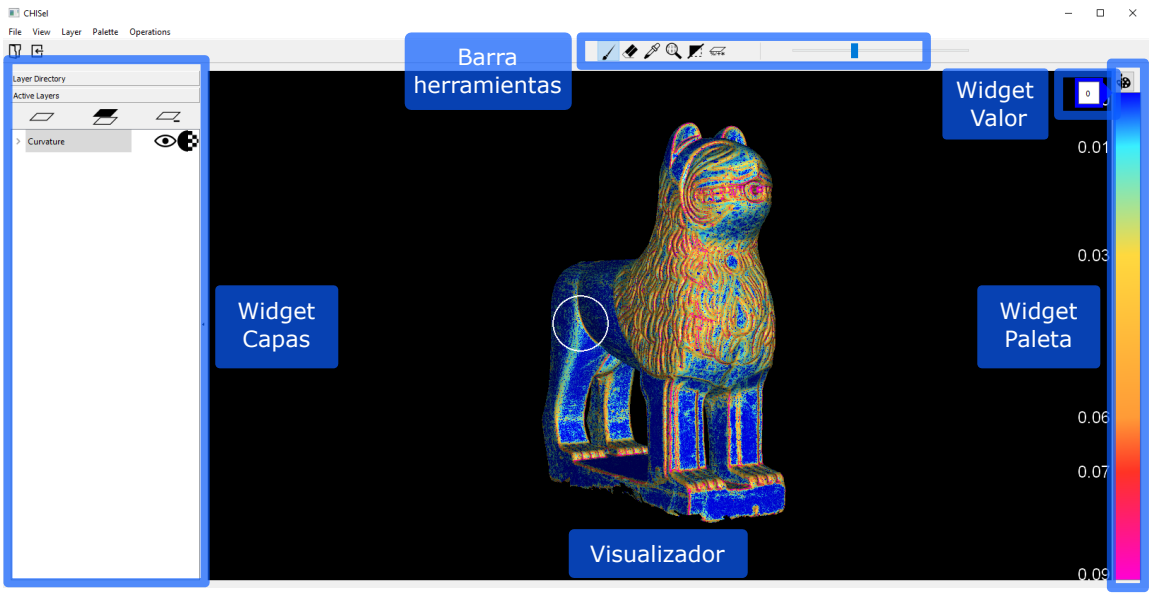


Figura 7.1 La interfaz de CHISel 3.0 está dividida en cuatro secciones: Widget de Capas, Visualizador, Widget del Valor actual y Widget de Paletas

7.2 Reclutamiento y plan general de pruebas

Con el fin de evaluar apropiadamente la usabilidad del sistema de información CHISel 3.0 se decidió buscar posibles participantes entre los estudiantes del Máster Universitario en Arqueología impartido por la Facultad de Filosofía y Letras de la Universidad de Granada. Estos cuentan con experiencia demostrada en trabajos de campo y, por tanto, conocen de primera mano las actividades que allí tienen lugar, así como las necesidades que requieren este tipo de tareas. En consecuencia, su perfil es idóneo para el propósito de este estudio, pudiendo ofrecer perspectivas informadas de las posibles ventajas que ofrece una herramienta como CHISel 3.0.

El método principal de reclutamiento empleado fue el envío de correos electrónicos desde el Departamento de Lenguajes y Sistemas Informáticos de la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación a todos los alumnos del Máster anteriormente mencionado. En el cuerpo del correo se les instaba a participar y a rellenar un formulario de reclutamiento online. Gracias a la colaboración de los primeros participantes inscritos se pudo extender el reclutamiento a grupos específicos de redes sociales que usaban con frecuencia los alumnos del Máster y Grado en Arqueología.

El formulario de reclutamiento incluía un conjunto de preguntas que permitía determinar el rango de edad, género, nivel educativo y de conocimientos informáticos, así como el grado de experiencia laboral de las personas inscritas. No fue necesario desechar ninguna de las solicitudes por no cubrir los requisitos exigidos.

Dado el perfil altamente especializado de los candidatos, inicialmente se intentó reclutar a diez estudiantes. Aunque ese fue precisamente el número de participantes inscritos, finalmente sólo acudieron a las sesiones 9 candidatos. Esta selección terminó incluyendo a siete estudiantes de Máster y dos de Grado.

Los participantes se encontraban en un rango de edad comprendido entre los 18 y 59 años, si bien sólo uno de ellos era mayor de 40. Dos participantes tenían más de cuatro años de experiencia trabajando en arqueología; otros seis candidatos tenían entre 1 y 3 años de experiencia y sólo dos de ellos no tenían experiencia alguna. Respecto a su habilidad con ordenadores y aplicaciones informáticas, únicamente un participante declaró ser un experto.

Las pruebas fueron supervisadas por dos facilitadores, uno lo hizo de forma presencial y otro asistió por videoconferencia a través de Google Meet. Las pruebas presentaban dos escenarios con tres tareas a completar en cada uno de ellos. Las sesiones de pruebas contenían cuestionarios pre-test, después de la realización de cada tarea y tras la finalización de la misma. En lo que a remuneración respecta, cada participante recibió un disco duro de 2TB una vez completaba su correspondiente sesión.

Entre los meses de marzo y abril del 2022 tuvo lugar la fase de reclutamiento de estudiantes. Las pruebas se realizaron de forma concurrente durante el mes de abril y terminaron extendiéndose a las primeras semanas de mayo del 2022.

7.3 Sesiones de pruebas

Los participantes realizaron el test de usabilidad de forma presencial en un despacho del Centro de Investigación en Tecnologías de la Información y las Comunicaciones de la Universidad de Granada (CITIC-UGR). Se emplearon dos ordenadores durante su realización: un ordenador de sobremesa que ejecutaba el sistema CHISel 3.0 y disponía de una pantalla de 24 pulgadas y un portátil que mostraba las tareas a completar en una pantalla de 15 pulgadas. Las interacciones de los participantes fueron supervisadas por un facilitador sentado a su derecha y un segundo que asistía de forma remota a través de una videoconferencia de Google Meet. La mayoría de las interacciones se realizaron íntegramente en inglés dado que uno de los facilitadores no sabía hablar español. La duración media de cada sesión fue de aproximadamente una hora. Con el fin de analizarlas en el futuro, las sesiones se grabaron con OBS Studio **[OBS23]**, un software de de grabación de audio y vídeo.

Al inicio de cada sesión el facilitador se presentaba, resumía el propósito de la prueba de usabilidad, permitía al participante leer el documento de consentimiento y pedía verbalmente su permiso para grabar su voz y todas las acciones realizadas en el sistema que ejecutaba CHISel 3.0. Una vez recibía el consentimiento del participante, el facilitador mostraba un vídeo de aproximadamente cinco minutos que explica los principales elementos de la interfaz de CHISel 3.0: organización, menús, uso de funcionalidad esencial, etc.

Tras esta introducción se solicitaba al participante que leyera cada tarea en voz alta, describiera también en voz alta sus pensamientos y preguntase cualquier tipo de duda de forma que existiera un registro de sus interacciones con el sistema. Dado que el inglés no es primera lengua de los participantes, en ocasiones estos se dirijieron al facilitador español en dicha lengua para aclarar conceptos sobre las tareas y este último traducía las interacciones al inglés para que el facilitador extranjero pudiera entenderlas.

Durante el desarrollo de la prueba los facilitadores observaron a los participantes, escribieron notas sobre su comportamiento y respondieron a cualquier tipo de dudas que pudieran tener. Después de finalizar cada tarea de un escenario concreto, los participantes respondían a un breve cuestionario sobre el trabajo realizado. Asimismo, una vez completaban los dos escenarios propuestos, rellenaban una encuesta de satisfacción **SUS** (System Usability Scale) **[JTMW96, BKM09]**.

7.3.1 Escenarios de las pruebas de usabilidad

Con el fin de cubrir los objetivos propuestos en el estudio, se diseñaron dos escenarios, con tres tareas a realizar en cada uno de ellos. Durante su diseño se hizo hincapié en la necesidad de abarcar la mayor cantidad posible de funcionalidad ofrecida por CHISel 3.0, aunque fuera de forma superficial. De esta manera, los participantes tendrían que importar modelos 3D, crear capas nuevas, insertar nuevos datos, realizar operaciones entre capas, modificar la visualización de los datos a través de las paletas, etc. En la Figura 7.2 y la Figura 7.3 se pueden ver algunas capturas de los objetivos que se persiguen en cada uno de los escenarios propuestos.

7.3.1.1 Escenario 1

Eres un miembro del *Proyecto de restauración de los leones* que se realiza en la Alhambra (Granada). El trabajo requiere crear documentación digital de la información recabada durante el proceso de restauración. Para lograrlo has de utilizar *CHISel 3.0*, un sistema de información 3D para patrimonio histórico desarrollado por la Universidad de Granada. El modelo 3D escaneado se encuentra disponible en la carpeta *Models*.

- **Tarea 1**: Necesitamos resaltar el área del modelo 3D de la que se recabó la información. Empezando en la ventana principal, importe el modelo 3D del león. El archivo se llama *lion.fbx*. Una vez que el proceso se complete y el modelo aparezca en el visualizador, cree una nueva *capa escalar*. Llámela *leftEye* y asígnele el tipo *INT 8*. Asigne el valor *0* a la superficie del ojo izquierdo.

- **Tarea 2**: Calcule un campo de distancias usando la capa previa como entrada. Compruebe que *leftEye* está seleccionada como *Base Layer* y cambie *Max Distance* a *1000*.

- **Tarea 3**: La paleta de colores asignada por defecto dista de ser la mejor opción para visualizar los datos de la capa resultante. Desde *Palette Collection*, aplique la paleta *Distance Field* a la capa recién creada.

7.3.1.2 Escenario 2

Nuestros colegas han estado trabajando en el *Proyecto de restauración de la vasija* durante los últimos días. Para poder documentar digitalmente el proyecto han estado utilizando *CHISel 3.0*. Te dispones a verificar si han terminado su trabajo y añadir nueva información en caso de que sea necesario. Los proyectos activos se encuentran en la carpeta *Projects*.

- **Tarea 1**: Empezando en la ventana principal, abre el proyecto *Vessel*. El archivo se llama *vessel.chl*. Una vez cargado, seleccione la capa *restoration*. Compruebe los valores que almacena el área coloreada de verde.

- **Tarea 2**: El valor del área coloreada de verde debería tener una fecha de comienzo de *10/12/21*. Si no es correcta, modifíquela.
- **Tarea 3**: En la superficie del modelo 3D, el área coloreada de verde no cubre de forma exacta la pieza que representa. Elimine la secciones que se solapan con el resto de piezas.

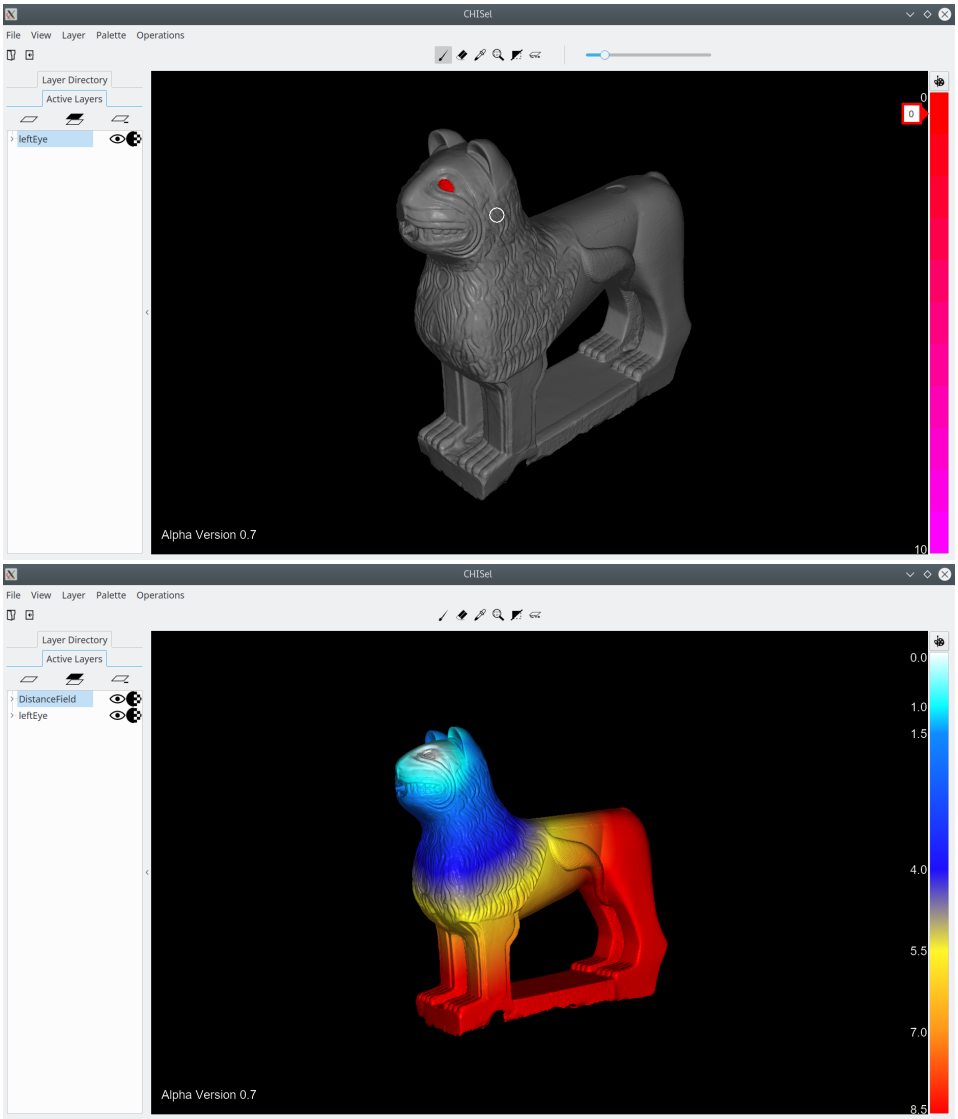


Figura 7.2 Capturas del Escenario 1. Arriba se muestra la capa *leftEye* que cubre el ojo izquierdo del modelo del león (Tarea 1). Abajo se muestra el campo de distancias obtenido al utilizar *leftEye* como entrada de la operación y cambiar la paleta de visualización (Tareas 2 y 3)

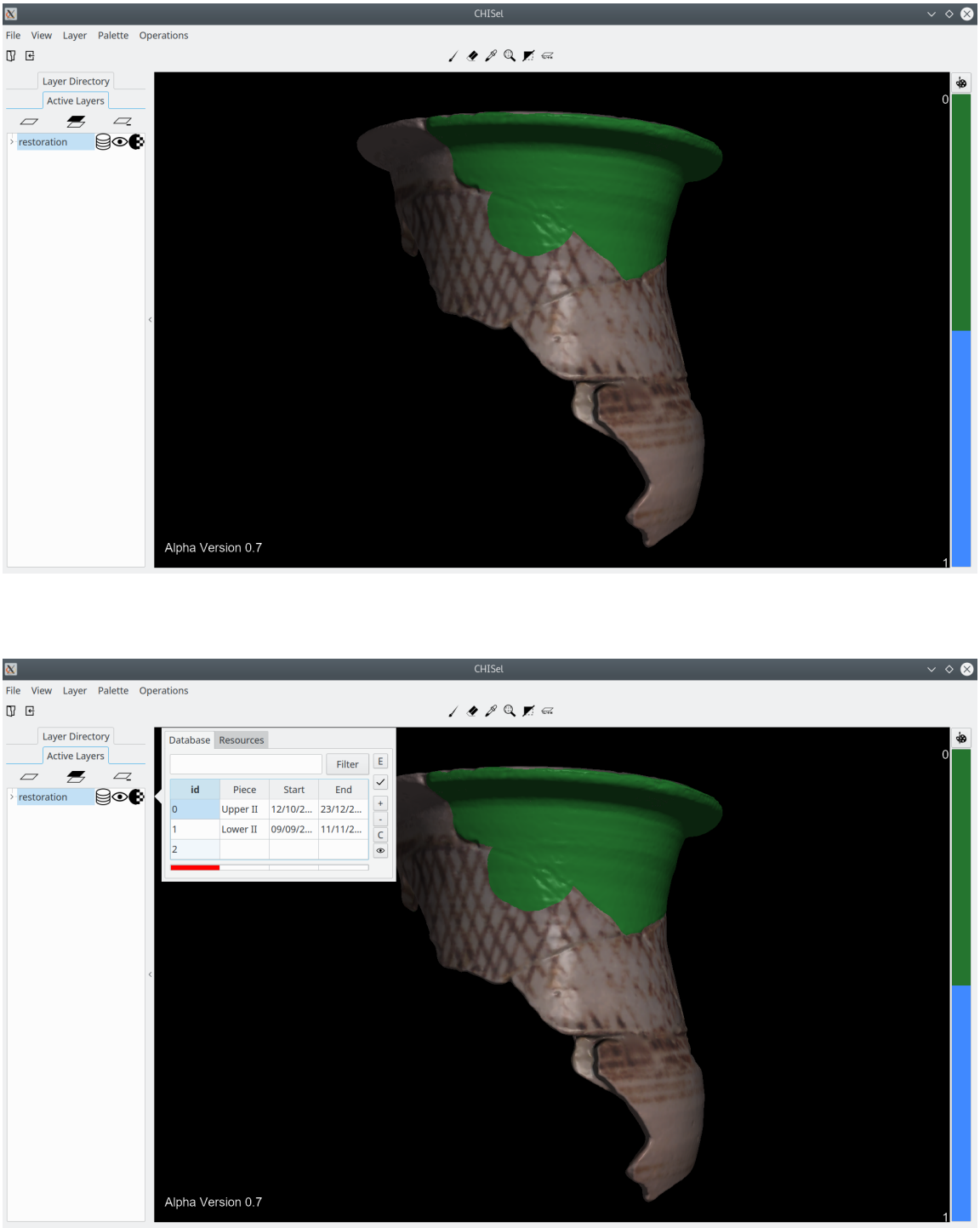


Figura 7.3 Capturas del Escenario 2. Arriba se muestra el estado inicial del proyecto. Abajo se muestra la tabla asociada a la capa *restoration* (Tarea 2)

7.3.2 Cuestionario de realización de tareas

Al completar cada tarea de los distintos escenarios propuestos, los participantes debían responder a tres cuestiones con una valoración de 1 a 5, donde el valor 1 representa *No estoy para nada satisfecho* y el valor 5, *Muy satisfecho*. Las preguntas realizadas fueron las siguientes:

- ¿Cómo valorarías tu rendimiento a la hora de completar la tarea?
- ¿Cómo valorarías lo fácil que es navegar por la interfaz para poder encontrar los recursos que te permitieron completar la tarea?
- ¿Cómo valorarías lo fácil que es encontrar la información necesaria?

7.3.3 Encuesta de satisfacción

A continuación se listan los diez enunciados predefinidos de la encuesta de satisfacción SUS que tuvieron que rellenar los participantes una vez completaron las tareas propuestas. Cada entrada disponía de un campo de valoración de 1 a 5, donde el valor 1 representa *Muy en desacuerdo* y el valor 5, *Totalmente de acuerdo*.

- Creo que me gustaría usar este sistema con frecuencia.
- He encontrado el sistema innecesariamente complejo.
- Creo que el sistema era fácil de usar.
- Creo que necesitaría el apoyo de una persona con conocimientos técnicos para poder usar este sistema.
- Pienso que toda la funcionalidad está muy bien integrada en el sistema.
- Pensé que había demasiada inconsistencia en este sistema.
- Imagino que la mayoría de los usuarios aprendería a utilizar este sistema muy rápidamente.
- Encontré el sistema muy complicado de usar.
- Me sentí muy seguro de ser capaz de utilizar el sistema.
- Necesitaba aprender muchas cosas antes de poder empezar a manejar el sistema.

7.4 Resultados

Con el fin de poder realizar un análisis pormenorizado de los resultados, se realizó una revisión de las grabaciones realizadas durante las sesiones con los participantes. Esto permitió registrar tanto sus comentarios positivos sobre un funcionalidad concreta de la herramienta, como los distintos problemas que tuvieron a la hora de completar las tareas. A estos problemas se les asignó una valoración de: 0 (—), 1 (*Menor*), 2 (*Grave*), 3 (*Catastrófico*).

La figura Tabla 7.1 organiza los resultados obtenidos en función del usuario, escenario y tarea realizada. Para cada tarea incluye el tiempo empleado en completarla, el número de clicks de ratón realizados, las respuestas al cuestionario de tres preguntas y la valoración del error explicada en el párrafo anterior. Además de estos datos, incluye la puntuación obtenida en la encuesta de satisfacción de cada participante.

El análisis de los errores refleja los siguientes resultados:

- Catastróficos **(5)**: aglutinan casos en los cuales la aplicación dejó de funcionar correctamente debido a un fallo de programación. En uno de ellos esta situación se produjo debido a que el usuario borró el contenido de la tabla asociada a la capa y fue incapaz de recrearla para completar la Tarea 2 del Escenario 2.

- Graves **(12)**: describen casos en los cuales el usuario fue incapaz de completar la tarea propuesta, invirtió demasiado tiempo en encontrar la función necesaria para finalizarla o completó la tarea de manera totalmente errónea. La realización de las Tareas 2 y 3 del Escenario 1 representaron la mayoría de estos casos. Los usuarios tuvieron problemas para encontrar el menú de operaciones donde se incluye el cálculo del Campo de Distancias (*Distance Field*) y, en ocasiones, ni siquiera comprendieron que debían realizar una operación entre capas. En el caso de las paletas, los usuarios no entendieron que la Colección de Paletas (*Palette Collection*) era una pestaña que podían desplegar para poder elegir entre una selección de paletas preexistentes. A menudo modificaron los puntos de control de la paleta seleccionada.

- Menores **(16)**: representan casos en los que el usuario mostró cierta confusión a la hora de abordar una tarea pero pudo completarla satisfactoriamente o con pequeños errores. No comprender inicialmente la diferencia entre importar un modelo 3D y abrir un proyecto o eliminar más información de la debida en la Tarea 3 del Escenario 2 son algunos ejemplos de estas situaciones.

- Sin problemas **(21)**: describen casos en los que no se produjo ninguna incidencia destacable durante la tarea.

Los resultados de las encuestas de satisfacción SUS arrojaron resultados bastante favorables. Con más de tres décadas de uso y miles de encuestas completadas, se ha comprobado que la valoración media de esta escala se encuentra en 68 **[Sau11]**. Entre los

participantes del estudio, solo un usuario se mostró insatisfecho con la aplicación. Tres de las encuestas produjeron valoraciones cercanas a la media con 2 puntuaciones de 65 y una de 67,5. Más satisfechos se mostraron los cinco participantes cuyas encuestas ofrecieron puntuaciones superiores a 75, llegando incluso dos casos a superar la barrera de los 92 puntos.

Entre los comentarios positivos realizados, los usuarios destacaron la posibilidad de asociar información directamente sobre la superficie de modelos 3D a través de las capas. Esta funcionalidad les pareció una forma intuitiva de trabajar en proyectos de documentación pero también se destacó su posible utilidad a la hora de dar a conocer dicha información en proyectos de divulgación.

Asimismo, los participantes señalaron las ventajas que ofrece una solución global como CHISel 3.0. Acostumbrados a utilizar múltiples herramientas en sus quehaceres diarios y a transferir con frecuencia datos entre las mismas, la posibilidad de utilizar una única herramienta para solventar gran parte de sus tareas les resultó una forma más simple y eficiente de trabajar.

Usr.	Escen.	Tarea	Tiempo (m:s)	Media # clicks	Valorac. Rendim.	Facilid. Navega.	Facilid. Inform.	Errores	SUS
1	1	1	2:05	23	5	5	5	—	67,5
		2	2:00	15	5	5	4	Menor	
		3	2:38	13	4	3	3	Grave	
	2	1	3:17	30	2	2	2	Grave	
		2	0:54	18	5	5	5	—	
		3	1:37	33	5	5	5	—	
2	1	1	2:32	22	3	4	4	Catast	65
		2	5:44	11	1	1	2	Grave	
		3	2:15	20	4	4	4	Grave	
	2	1	3:05	28	3	4	2	Grave	
		2	9:02	8	5	5	5	—	
		3	1:20	15	4	5	4	—	
3	1	1	3:57	58	5	4	5	Menor	75
		2	2:10	12	2	3	2	Grave	
		3	1:54	22	4	4	3	Grave	
	2	1	2:53	30	4	5	4	Menor	
		2	1:00	21	5	5	5	—	
		3	2:25	3	5	5	5	—	
4	1	1	3:16	27	2	5	4	Catast	85
		2	1:53	18	5	3	5	—	
		3	2:35	16	4	3	3	Menor	
	2	1	1:08	9	5	5	5	—	
		2	0:43	10	5	4	4	Grave	
		3	1:00	18	5	5	5	—	

Usr.	Escen.	Tarea	Tiempo (m:s)	Media # clicks	Valorac. Rendim.	Facilid. Navega.	Facilid. Inform.	Errores	SUS
5	1	1	10:55	116	2	2	3	Menor	47,5
		2	4:25	47	2	2	2	Grave	
		3	0:27	5	3	3	2	Catast	
	2	1	4:18	52	2	3	3	Catast	
		2	2:02	34	2	2	2	Catast	
		3	0:56	7	3	4	4	—	
6	1	1	3:18	13	3	3	3	—	65
		2	4:55	15	3	4	4	Grave	
		3	1:41	13	2	3	3	Menor	
	2	1	4:06	43	3	3	3	Menor	
		2	0:45	5	4	4	4	—	
		3	1:04	13	5	5	5	—	
7	1	1	6:38	66	4	3	4	Menor	92,5
		2	2:42	46	3	4	4	Menor	
		3	0:52	17	4	5	5	—	
	2	1	2:26	39	1	4	4	Menor	
		2	1:35	24	4	4	4	Menor	
		3	0:23	6	5	5	5	—	
8	1	1	3:22	39	4	4	5	Menor	80
		2	1:45	11	2	5	5	Menor	
		3	6:08	50	2	3	3	Grave	
	2	1	1:33	30	4	5	5	Menor	
		2	0:36	10	5	5	5	—	
		3	1:07	18	4	5	5	Menor	
9	1	1	3:20	40	4	4	5	—	92,5
		2	2:42	30	2	5	5	Menor	
		3	1:53	13	2	3	3	Grave	
	2	1	0:25	11	4	5	5	—	
		2	2:00	40	5	5	5	—	
		3	0:16	5	4	5	5	—	

Tabla 7.1 Resumen de los resultados obtenidos en el estudio, ordenados en función del escenario y tarea realizada por cada usuario. Se incluye el tiempo empleado en resolver una tarea, el número de clicks de ratón, las respuestas a las tres preguntas realizadas después de cada tarea y un valoración de errores. Además, se muestra la puntuación recibida en la encuesta de satisfacción

7.5 Conclusiones

En este capítulo se ha realizado un estudio de usabilidad de CHISel 3.0, el sistema de información que implementa toda la funcionalidad descrita hasta el momento. El estudio se estructura en torno a dos escenarios con tres tareas a completar en cada uno de ellos. Estas tareas abarcan gran parte de la funcionalidad principal ofrecida por el sistema.

En el estudio participaron estudiantes de Grado y Máster en Arqueología de la Universidad de Granada. En general, se mostraron bastante satisfechos con la aplicación, destacando su versatilidad a la hora de completar tareas que, habitualmente, implican el uso de múltiples herramientas en sus trabajos.

Al mismo tiempo, los resultados del estudio han permitido encontrar varios problemas de navegabilidad y presentación en la herramienta. El acceso al menú de operaciones entre capas y la colección de paletas fueron las acciones que plantearon más dificultades a los participantes.

8

Lenguaje de programación visual para la documentación gráfica

Tras detallar cómo podemos utilizar GPUs eficientemente en arquitecturas de sistemas de información 3D, almacenamiento de topologías en texturas 2D y resolución de operaciones entre capas, este capítulo se centra en documentar gráficamente el patrimonio histórico mediante un intuitivo lenguaje de programación visual

El proceso de documentación gráfica de piezas arqueológicas requiere de la participación de un artista capaz de recrear ilustraciones empleando distintas técnicas expresivas. A menudo, la labor del artista se ve limitada por la inconveniencia de trabajar únicamente con fotografías de las piezas a ilustrar. Las razones pueden ser variadas: el acceso directo a la pieza no es posible debido a cuestiones de preservación, existe algún tipo de incompatibilidad con el programa de visitas, la distancia entre lugar de trabajo del artista y el yacimiento arqueológico, etc. Además, si la calidad de las fotografías no es lo suficientemente buena, los resultados podrían ser inaceptables.

La posibilidad de trabajar directamente con modelos 3D escaneados solventa todos estos inconvenientes y ofrece una representación exacta de la pieza desde cualquier ángulo o posición. No existen problemas de acceso a la pieza en cuestión, ni de precisión de la representación y, al mismo tiempo, abre la puerta a una mayor variedad en la presentación de las ilustraciones y técnicas que se pueden aplicar a ellas.

A lo largo de las últimas décadas se han presentado una gran variedad de trabajos que tratan de generar ilustraciones de forma automática a partir de fotografías y modelos 3D. El área de la informática gráfica que se ocupa de este tipo de investigaciones se conoce como visualización expresiva o *Non-Photorealistic Rendering* y su propósito es mimetizar distintos estilos visuales creados por artistas.

Estos últimos expresan sus inquietudes a través de sus creaciones y las plasman utilizando estilos visuales que beben de su propia experiencia, estado de ánimo o habilidades. La visualización expresiva habitualmente describe estilos artísticos mediante primitivas gráficas complejas que aplica sobre la superficie de modelos 3D o fotografías haciendo uso de sofisticados algoritmos. Dichas primitivas suelen describir elementos básicos que utilizan los artistas en sus obras como son puntos, trazos o siluetas.

Aunque se pueden utilizar de forma exclusiva para lograr estilos específicos como ilustraciones de punteado, también existen sistemas que permiten combinar primitivas gráficas para generar efectos visuales más sofisticados. Sin embargo, estos sistemas a menudo presentan restricciones sobre qué tipo de primitivas se pueden emplear y, especialmente, cómo se puede operar entre ellas.

A lo largo de este capítulo se describe un sistema unificado capaz combinar multitud de técnicas de visualización expresiva mediante un intuitivo lenguaje de programación visual. El usuario puede crear en tiempo real una gran variedad de estilos artísticos sin necesidad de conocer el orden de procesamiento interno ni sufrir penalización alguna en el rendimiento. Además, las técnicas implementadas y las estructuras diseñadas utilizan los recursos de la GPU para garantizar una exploración de estilos interactiva con acceso inmediato a todos los resultados parciales y capacidad para alterarlos.

8.1 Trabajos relacionados

La idea de combinar algoritmos simples para crear estilizaciones más complejas no es especialmente nueva **[Sch94, LS95]**. Inspirado por el trabajo desarrollado por las primeras técnicas de visualización expresiva, Richens **[Ric99]** diseña e implementa el sistema comercial Piranesi que utiliza formas 3D como modelos de entrada. Entre sus principales características destaca el uso de técnicas apoyadas en buffers de geometría o G-buffers **[ST90]** para combinar distintas técnicas expresivas; elección que, al mismo tiempo, también limita el espectro de combinaciones posibles. Halper et al. **[HIR*03]** desarrollan OpenNPAR, un sistema capaz de usar técnicas basadas en superficies, imágenes o líneas. Su propuesta consiste en facilitar la combinación de varios módulos que pueden afectar de forma potencial a distintos tipos de datos dentro de un mismo cauce. A su vez, cauces creados con anterioridad pueden reutilizarse para ser encapsulados en otros más complejos y construir estilos más sofisticados. El sistema RenderBots desarrollado por Schlechtweg et al. **[SGS05]** se centra en el uso de una pila de G-buffers que contiene información para un sistema multi-agente que, a su vez, crea los efectos deseados. Los agentes pueden describir contornos, líneas características, rayados, punteados, mosaicos o pinturas y componerlas de forma interactiva en una única imagen.

Otros sistemas, sin embargo, buscan crear distintos tipos de efectos de visualización expresiva mediante el uso de un tipo específico de datos. Kalnins et al. **[KMM*02]** propone WYSIWYG-NPR, implementado en el sistema JOT^2, que se centra fundamentalmente en la extracción de líneas y el procesamiento, estilización y combinación interactiva de trazos y estilos hechos a mano en una escena. Mientras que WYSIWYG-NPR ofrece resultados de alta calidad y soporta una amplia variedad de efectos y técnicas de líneas, también utiliza un cauce estricto que hace imposible reordenación alguna de sus componentes. G-strokes, desarrollado por Isenberg et al. **[IB06]**, unifica el procesamiento de trazos siguiendo la filosofía de OpenNPAR, mientras que ViewMap, desarrollado por Grabli et al. **[GTDS10]** e implementado en el sistema Freestyle3, se centra en la extracción y representación de propiedades de trazos en una escena y la aplicación de módulos similares a los shaders a la misma. En contraste con WYSIWYG-NPR y GStrokes, los shaders programables de ViewMap permiten al programador combinar módulos libremente y facilitar una integración de diferentes estilos en una única representación.

8.2 Visión general del sistema

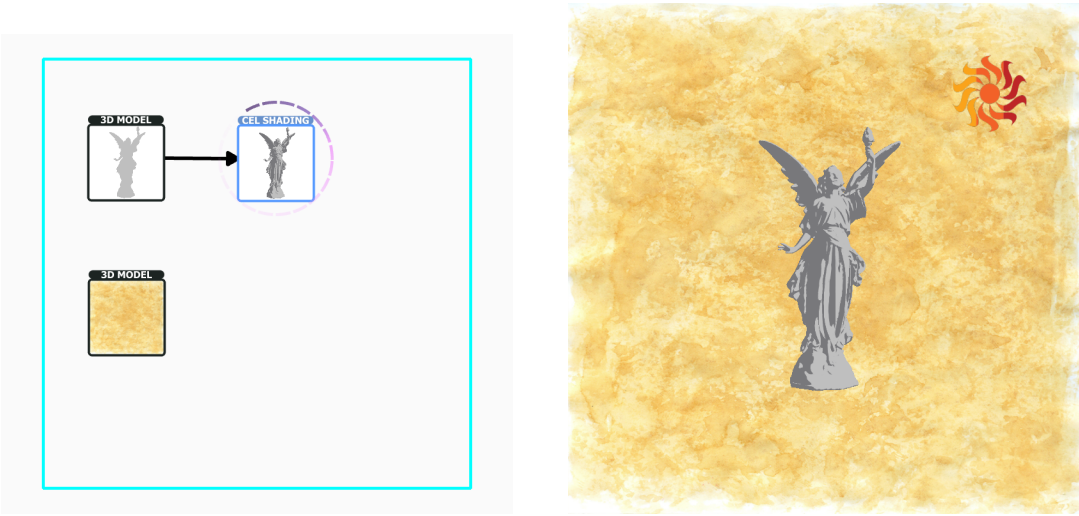
El sistema propuesto soporta imágenes y modelos 3D como datos de entrada. Tras un procesamiento inicial, el usuario puede utilizar el lenguaje de programación visual para componer estilos visuales distintivos a partir de las técnicas expresivas disponibles. Las ilustraciones generadas pueden emplearse para multitud de fines, siendo la documentación gráfica de patrimonio histórico uno de ellos.

8.2.1 Componentes de la interfaz de usuario

El prototipo desarrollado como prueba de concepto del sistema está organizado en torno a los siguientes módulos:

- **Editor**: sección de carácter esquemático donde el usuario describe estilos artísticos mediante la composición de funciones expresivas. El establecimiento de una relación entre nodos de la hoja de edición se realiza a través de flechas direccionales.

- **Visualizador**: sección donde se visualizan en tiempo real los resultados generados por el grafo de operaciones descrito en el editor. Además, permite interactuar con manejadores o herramientas gráficas que modifican atributos espaciales, como la posición de la luz, de la función seleccionada en el Editor.



(a) Editor del prototipo **(b)** Visualizador del prototipo

Figura 8.1 Principales módulos del prototipo

La Figura 8.1 muestra una captura de cada módulo. A la izquierda se muestra el Editor y a la derecha, el Visualizador.

8.2.2 Algoritmos desarrollados

Los algoritmos utilizados por el sistema se pueden organizar en las siguientes categorías:

- **Procesamiento de datos de entrada**: procesa imágenes y modelos 3D descritos en los formatos soportados para crear la estructura de datos que requieren el resto de algoritmos del sistema.

- **Resolución de grafos**: usa la estructura de datos creada y los grafos de operaciones diseñados por el usuario para resolver las dependencias y propagar la información necesaria desde la raíz hasta los nodos hoja que se han de representar. Dado que se trata de un proceso interactivo, los resultados se actualizan en tiempo real en base a cambios realizados al modelo o imagen de entrada, la estructura del grafo o los atributos de configuración de operaciones específicas.

- **Algoritmos de representación**: empleando las estructuras de datos actualizadas y el conjunto de valores asignados a los atributos, estos algoritmos modifican la apariencia inicial de la imagen o modelo 3D para conseguir el estilo visual descrito.

8.3 Estructura de datos

El modelo subyacente ha de definirse de forma que describa un resultado predecible ante la combinación de una técnica de visualización expresiva con cualquier otra. Esta condición requiere que se diseñe una estructura de datos adecuada, capaz de soportar la integración de las diferentes técnicas en las cuales se basan los algoritmos implementados. Esta estructura de datos necesita capturar los datos gráficos que sean necesarios, incluyendo geometría, imágenes o trazos. Mientras que algunas técnicas de visualización expresiva operan enteramente sobre imágenes, se ha decidido utilizar el modelo 3D como representación fundamental debido a su gran versatilidad. Un modelo 3D permite representar geometrías 3D, trazos e imágenes usando vértices, aristas, caras, texturas, trasformaciones, etc.

La estructura que cumple estos requisitos y es utilizada por el sistema propuesto recibe el nombre de NPR Data Object (NDO). A alto nivel, está compuesta por tres tipos de datos:

- **Geometría**: almacena las mallas 3D, lienzos 2D, localizaciones de los trazos así como coordenadas de textura y matrices de transformación. A excepción de las matrices, esta información geométrica se estructura en forma de arrays de vértices que contienen posiciones 3D, coordenadas de texturas e índices de caras o triángulos. Gracias a esta configuración, se realiza una transferencia directa y eficiente a la GPU para su representación.

Con el fin de agilizar consultas de carácter topológico, que suelen ser comunes en multitud de técnicas de visualización expresiva, se construye una estructura de half-edges o semiaristas a partir de la información geométrica original. Gracias a ella, el acceso a caras, aristas y vértices se realiza de forma eficiente en el espacio de la CPU.

- **Propiedades de shader**: almacenan los atributos necesarios para definir un algoritmo de visualización expresiva concreto, incluido el código del propio shader. Representan un conjunto heterogéneo de datos que incluye escalares que pueden definir el grosor de un trazo, vectores que pueden describir su color o texturas 2D en escala de grises que pueden representar la forma de dicho trazo. El número y tipo de las propiedades es definido por los requisitos de la implementación del algoritmo a representar.

- **Caché de representación**: se trata de una representación intermedia en forma de textura 2D que permite mejorar el rendimiento cuando se combinan múltiples técnicas que modifican las propiedades de color de la superficie de un modelo. Gracias a ello sólo es necesario calcular los cambios sobre el nodo actual y, en caso de que sea necesario, propagarlos a sus descendientes.

La estructura de datos NDO vive parcialmente tanto en el espacio de CPU como en el de GPU. La estructura topológica de semiaristas y las propiedades del shader residen en memoria principal, mientras que el array de vértices, las texturas de trazos y representaciones intermedias se almacenan en memoria de GPU. De esta manera se intenta minimizar las transferencias entre CPU y GPU, relegándolas en la mayoría de los casos a actualizaciones rápidas de propiedades de shaders.

La carga de un modelo 3D o imagen en el sistema crea una instancia inicial de esta estructura. La información del modelo 3D se almacena en los arrays de vértices. La imagen, a su vez, se almacena como textura 2D en la caché de representación pero, además, genera una lienzo 2D cuya geometría se incluye en los arrays de vértices. Por consiguiente, la imagen pasa también a ser un modelo 3D.

8.4 Función Expresiva

Sea el atributo a_i un elemento del conjunto $A_t \subseteq A_e \cup A_v$ donde A_e es el conjunto de valores escalares y A_v es el subconjunto de valores vectoriales. $A_e \subseteq I \cup R$ donde I es el conjunto de imágenes 2D, mientras que $A_v \subseteq \mathbb{R}^3_1 \times ... \times \mathbb{R}^3_k \cup R^4_1 \times ... \times \mathbb{R}^4_k$, con $k > 0$.

Sea A el conjunto de de todos los atributos tal que $A \subseteq A^m_t \cup \{\varepsilon\}$, con ε siendo el elemento vacío y $m > 0$.

Sea el NDO n_i un elemento del conjunto $N \subseteq A_e \cup A_v$ que contiene todos los posibles NDOs.

Una función expresiva es cualquier cualquier función que cumple:

$$f : N^m \times A \rightarrow N \tag{8.1}$$

donde $m > 0$.

De forma menos formal, una función expresiva es aquella que recibe uno o más NDOs como entrada y genera un nuevo NDO como salida. Mediante los atributos de entrada se especifican las propiedades que determinan cómo ha de operar la función sobre los NDO de entrada para producir la salida.

Cada función expresiva se corresponde con una única técnica de visualización y el sistema propuesto dispone de un amplio abanico de funciones que cubre multitud de necesidades: desde aquellas que modifican el aspecto de la superficie de modelos 3D, como puede ser el cel shading; hasta otras que los segmentan, como el selector de geometría.

La función expresiva identidad es aquella que cumple:

$$I : N \times A \rightarrow N \tag{8.2}$$

tal que $I(x, \varepsilon) = x$ con $x \in N$. En el sistema propuesto se corresponde con la función Modelo 3D. Carece de atributos y no realiza modificación alguna sobre el NDO de entrada.

Las funciones expresivas soportadas por el lenguaje de programación visual se estructuran en las siguientes categorías lógicas según los resultados que producen.

8.4.1 Funciones expresivas Estilizadas

Las funciones expresivas **Estilizadas** modifican las propiedades de color de la superficie del modelo. La Figura 8.2 ofrece ejemplos gráficos de cada tipo.

- **Silueta**: genera una silueta sobre el modelo utilizando el algoritmo propuesto por Hermosilla et al. **[HV09]**. Su aspecto se puede modificar en función del tipo, color y grosor del trazo seleccionado. Además dispone de un multiplicador, que afecta a la longitud de los trazos, y un factor de ajuste, que afecta a su posicionamiento. Formalmente:

  $$f(x, \vec{c}, g, t, m, fp) = y \tag{8.3}$$

  donde $\vec{c}$ es un vector que describe el color rgba seleccionado, g es un escalar que describe el grosor, t es el trazo seleccionado, m es un escalar que modifica la longitud de los trazos y fp, un factor de ajuste de su posición sobre el modelo.

- **Cel shading**: aplica un aplanamiento de color al modelo. La dirección del la luz, el número de franjas, su color y amplitud constituyen las variables que puede modificar el usuario. Formalmente:

  $$f(x, \vec{l_d}, v_1, ...v_n) = y \tag{8.4}$$

  donde $\vec{l_d}$ es el vector de dirección de la luz, v_i es un vector que define la información de una franja y n es el numero de franjas aplicadas. En concreto, v_i contiene un

escalar a_i para definir la amplitud de la franja y un vector $\vec{c_i}$ para el color rgba empleado.

- **Rayado**: produce un efecto de rayado sobre el modelo. Su aspecto varía en función de la frecuencia, grosor, color y dirección de las líneas así como la luminancia y dirección de la luz. Formalmente:

$$f(x, \vec{l_d}, fr, g, lm, \vec{c}, \vec{r_d}) = y \tag{8.5}$$

donde $\vec{l_d}$ es el vector de dirección de la luz, fr es un escalar que describe la frecuencia, g es un escalar que describe el grosor, lm es un escalar que describe la luminancia, $\vec{c}$ es un vector que describe el color rgba seleccionado y $\vec{r_d}$ es un vector que indica la dirección del rayado.

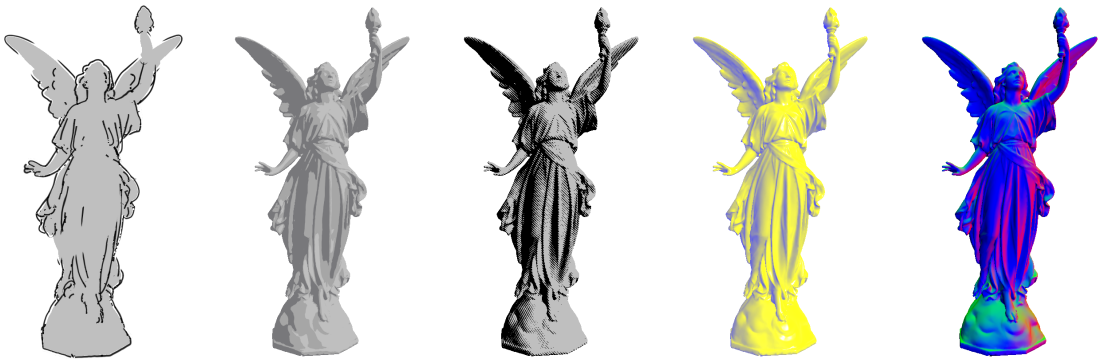
- **Gooch shading**: representa la superficie del modelo contrastando un color frío con otro cálido. Propuesto por Gooch et al. [GGSC98]. Su aspecto varía en función de la propiedad difusa de la superficie, el color cálido, el color frío y los factores aplicados a estos dos últimos. Formalmente:

$$f(x, \vec{c}, fc, \vec{fr}, f_{fr}, \vec{d}) = y \tag{8.6}$$

donde $\vec{c}$ y f_c son el color cálido y su factor, $\vec{fr}$ y f_{fr} son el color frío y su factor y $\vec{d}$ el color difuso de la superficie.

- **Normal**: representa las normales de los vértices empleando sus coordenadas normalizadas x, y y z como valores de las componentes rojo, verde y azul de un color respectivamente. Carece de atributos. Formalmente:

$$f(x) = y \tag{8.7}$$



(a) Silueta **(b)** Cel shading **(c)** Rayado **(d)** Gooch shading **(e)** Normales

Figura 8.2 Ejemplos de aplicación de funciones expresivas Estilizadas

8.4.2 Funciones expresivas de Transformación

Las **Funciones expresivas de Transformación** modifican la matriz de transformación del modelo. La Figura 8.3 ofrece ejemplos gráficos de cada tipo.

- **Traslación**: aplica una traslación al modelo según las coordenadas especificadas. Formalmente:
$$f(x, \vec{t}) = y \tag{8.8}$$
donde $\vec{t}$ es un vector que describe el factor de traslación en el espacio 3D.

- **Rotación**: aplica una rotación al modelo según el ángulo y dirección especificados. Formalmente:
$$f(x, a, \vec{e}) = y \tag{8.9}$$
donde a es un escalar que representa el ángulo de rotación y $\vec{e}$, el vector que define el eje de rotación.

- **Escalado**: aplica un escalado al modelo según los factores de escala especificados. Formalmente:
$$f(x, \vec{e}) = y \tag{8.10}$$
donde $\vec{e}$ es un vector que define los factores de escalado en el espacio 3D.

(a) Traslación **(b)** Rotación **(c)** Escalado

Figura 8.3 Ejemplos de aplicación de funciones expresivas de Transformación

8.4.3 Funciones expresivas de Filtros

Las **Funciones expresivas de Filtros** aplican un procesamiento de imagen sobre el resultado proporcionado por el NDO de entrada. Sustituyen la geometría original por un cuadrado que representa un lienzo. La Figura 8.4 ofrece ejemplos gráficos de cada tipo.

- **Gauss**: genera un lienzo 2D que contiene una imagen emborronada del resultado anterior. Carece de atributos. Formalmente:

$$f(x) = y \tag{8.11}$$

- **Sobel**: genera un lienzo 2D que contiene los contornos de la geometría representada por el resultado anterior. Formalmente:

$$f(x, \vec{c}) = y \tag{8.12}$$

  donde $\vec{c}$ es el vector que define el color de los contornos detectados.

- **Profundidad**: genera un lienzo 2D que contiene la información de profundidad, en escala de grises, de la geometría representada por el resultado anterior. Formalmente:

$$f(x, p_d, p_t, m) = y \tag{8.13}$$

  donde p_d define la posición del plano delantero, p_t define la posición del plano trasero y m es el multiplicador aplicado sobre la escala de grises.

8.4.4 Funciones expresivas de Blending

Las **Funciones expresivas de Blending** componen los resultados de dos entradas en base a su información de color o de profundidad. Sustituyen la geometría original por un cuadrado que representa un lienzo. La Figura 8.5 ofrece ejemplos gráficos de cada tipo.

- **Blend**: genera un lienzo 2D que mezcla la información de color de dos NDO pasados como entrada según la operación seleccionada (suma, resta, multiplicación y división). Formalmente:

$$f(x, y, o) = z \tag{8.14}$$

  donde donde x e y son dos NDO recibidos como entrada, o es la operación aritmética de mezcla de colores y z, el NDO resultante.

- **Z-Blend**: genera un lienzo 2D utilizando la información de profundidad de los NDO pasados como entrada. Formalmente:

$$f(x, y) = z \tag{8.15}$$

  donde x e y son dos NDO recibidos como entrada y z, el resultante.

(a) Gauss **(b)** Sobel **(c)** Profundidad

Figura 8.4 Ejemplos de aplicación de funciones expresivas de Filtros

(a) Blending **(b)** Z-Blending

Figura 8.5 A la izquierda se muestra el resultado de mezclar la salida del cel shading y un lienzo 2D con una imagen de lino. A la derecha se utiliza la información de profundidad para crear un lienzo 2D que representa correctamente las siluetas del modelo 3D y la salida generada por la función de Rayado

8.4.5 Otras funciones expresivas

Además de las funciones anteriormente descritas, el lenguaje de programación visual incluye las siguientes funciones expresivas:

- **Selección**: Selecciona un subconjunto de triángulos de la geometría del NDO recibido como entrada.

$$f(x, s) = z \tag{8.16}$$

  donde s es un conjunto de índices de triángulos de la geometría definida por el NDO x y z es el NDO resultante que contiene únicamente los triángulos que han sido seleccionados.

- **Agrupación**: Permite agrupar un subgrafo de funciones expresivas en una sola función que tiene como entrada el conjunto de entradas de las funciones seleccionadas y como salida, el resultado de aplicarlas sobre dichas entradas en el orden descrito por el grafo.

- **Modelo 3D**: función expresiva identidad. Representa el nodo inicial de una secuencia de funciones expresivas.

8.5 Operando con funciones expresivas

Una operación con funciones expresivas consiste en enlazar dos funciones mediante una flecha u operador de enlace. A la función expresiva que se encuentra en el extremo inicial del operador de enlace se le denomina función emisora, mientras que a aquella que se encuentra en el extremo final se le denomina función receptora. Como resultado de esta asociación, la función receptora opera sobre el NDO facilitado por la emisora. Los atributos de la función receptora personalizan el comportamiento del algoritmo que implementa dicha función y, tras ejecutarse, genera un nuevo NDO de salida.

En términos matemáticos, el establecimiento de un enlace entre funciones expresivas equivale a operar entre múltiples funciones mediante composición, hecho bastante usual en lenguajes funcionales. La siguiente expresión denota un ejemplo de esta operación:

$$g(f(x, a), a') = y \tag{8.17}$$

donde f y g son dos funciones expresivas, x es el NDO que recibe f como entrada, a el conjunto de atributos de f y a' el conjunto de atributos de g e y el NDO resultante de componer la salida generada por f con g.

En ciertos casos la función expresiva receptora necesita varias entradas y, por tanto, requiere el establecimiento de múltiples operaciones de enlace para que pueda llegar aplicarse.

La aplicación encadenada de operaciones entre funciones expresivas da lugar a un grafo ordenado que describe una técnica de visualización personalizada. Esta construcción conceptual permite conectar la salida de funciones emisoras con las entradas de una o varias funciones receptoras. Sin embargo, también establece una serie de restricciones. No se puede establecer una relación entre dos salidas, ni entre dos entradas, ni tampoco entre la salida y entrada de una misma función expresiva.

Únicamente las salidas de los nodos hoja del grafo se representan en el Visualizador. Cada NDO cuenta con la información necesaria para producir una visualización válida y esta será el resultado de la composición de un número variable de funciones expresivas.

A continuación se detallan los cuatro casos que pueden darse a la hora de operar con funciones expresivas en el Editor. Un ejemplo gráfico de cada uno de estos casos se muestra en la Figura 8.6 y Figura 8.7.

- **Caso base**: la función receptora opera sobre el NDO transferido por la función emisora una vez se ha establecido la relación mediante una operador de enlace. Común al iniciar la cadena de operaciones y en aquellos casos en los cuales sólo se desee trabajar con un único cauce. Formalmente se corresponde con:

$$g(f(x,a),a') = y \tag{8.18}$$

  donde f es la función emisora, a y x, los atributos y el NDO de entrada de f, respectivamente; g, la función receptora; a', los atributos que utiliza g e y, el NDO que genera.

- **Múltiples salidas**: la función emisora establece una relación con múltiples funciones receptoras. Estas últimas operan sobre el NDO transferido por la función emisora. Común en aquellos casos en los cuales se desea generar efectos visuales distintos a partir un resultado parcial común. Formalmente se corresponde con:

$$g(f(x,a),a') = y, h(f(x,a),a'') = w, j(f(x,a),a''') = z, \ldots \tag{8.19}$$

  donde f es la función emisora y g, h y j, las funciones receptoras.

- **Múltiples entradas**: la función receptora necesita varias entradas para poder aplicarse. Opera con todos los NDOs que recibe por parte de cada una de las funciones emisoras. Común en la aplicación de filtros. Formalmente se corresponde con:

$$h(f(x,a),g(y,a'),a'') = z \tag{8.20}$$

  donde f y g representan las funciones emisoras, h es la función receptora, a'' el conjunto de atributos que utiliza y z el NDO que produce como salida.

- **Propagación de actualizaciones**: cada una de las funciones expresivas que sean descendientes de la función actualizada operan sobre el nuevo NDO transferido por esta última. Este mismo proceso se aplicará de forma recursiva sobre todas y

cada una de las descendientes hasta que no quede ninguna función por actualizar. Formalmente, un ejemplo de este conjunto de casos se puede expresar con:

$$h(g(f(x,p),p'),p'') = w \xrightarrow{y} h(g(f(y,p),p'),p'') = z \tag{8.21}$$

donde el NDO de entrada x es sustituido por y, propiciando la actualización en cadena de las funciones f, g y h.

Si bien es cierto que los cuatro casos enumerados cubren las distintas posibilidades que se pueden dar a la hora de componer operaciones con funciones expresivas, también se ha de detallar qué ocurre cuando se elimina alguna de estas relaciones. La consecuencia directa de eliminar un enlace entre dos funciones es que los resultados generados por el subgrafo que queda desconectado dejan de representarse en el Visualizador. Asimismo, la función emisora ahora pasa a ser un nodo terminal del grafo y, por tanto, se representa su salida.

Esto no quiere decir que el subgrafo que ha quedado huérfano carezca de utilidad alguna. Se puede volver a establecer la relación que existía entre las funciones expresivas y, gracias a la propagación de actualizaciones, se recuperarían sus resultados. Sin embargo, también se puede establecer una nueva relación empleando un función emisora diferente y, gracias a la propagación de actualizaciones, se generarían, en este caso, resultados completamente distintos. De esta manera la función expresiva definida por el subgrafo se reutiliza para lograr nuevos resultados empleando entradas diferentes. Esta funcionalidad permite experimentar con distintas técnicas de visualización de forma rápida y eficiente. La única restricción que presenta es la incompatibilidad parcial con la función de selección de geometría que, por razones obvias, es dependiente de la malla sobre la que se aplica. En la Figura 8.8 se puede observar un ejemplo gráfico de reutilización de un subgrafo gracias a la eliminación de relaciones.

8.5.1 Resolución de grafos

Las funciones expresivas soportadas por el sistema se pueden agrupar en tres categorías dependiendo del tipo de datos que modifican:

- **Funciones geométricas**: modifican la información geométrica del NDO y se emplean en técnicas de extracción de líneas, trazos y selección de triángulos.
- **Funciones de materiales**: implementan shaders que modifican la información de color de la superficie de modelos 3D en base a las propiedades definidas. Modifican la caché de representación para reflejar dichos cambios.
- **Funciones de filtros**: producen representaciones sobre lienzos 2D a partir de los resultados facilitados por los NDO de entrada. Modifican, por tanto, la geometría para construir el cuadrado del lienzo 2D y la caché de representación para reflejar la operación realizada sobre sus entradas.

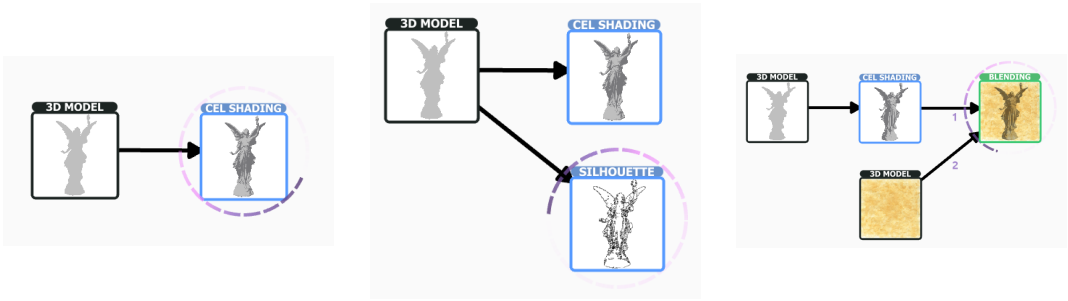


Figura 8.6 Casos de establecimiento de relaciones entre funciones expresivas. A la izquierda se muestra el caso base; en el centro, múltiples salidas; a la derecha, múliples entradas

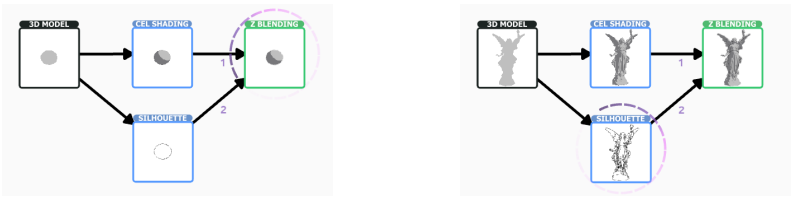


Figura 8.7 Caso de propagación de actualizaciones a través del grafo propiciado por una modificación del modelo 3D

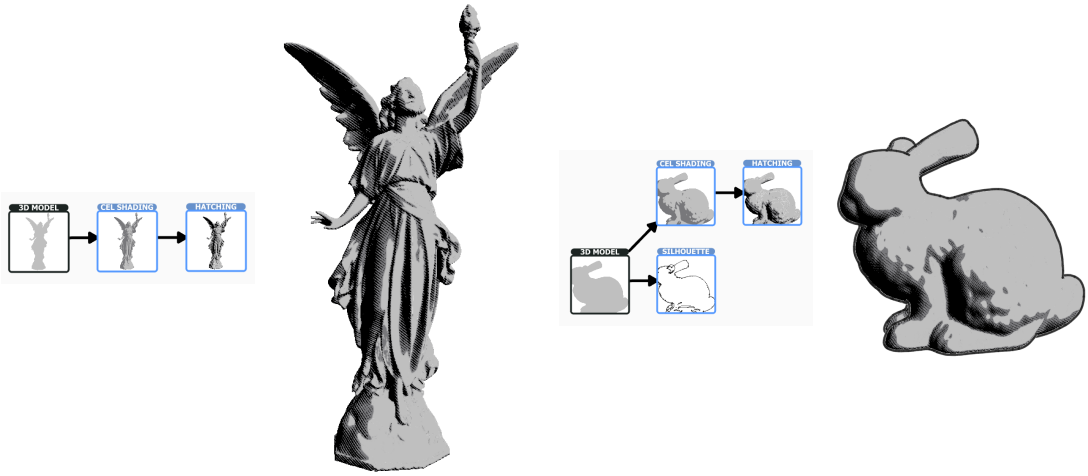


Figura 8.8 La parte izquierda de la figura muestra la composición de una función de cel shading con otra de Rayado. Este pequeño cauce se reutiliza en el modelo del conejo que ya cuenta con un función de siluetas aplicada

En el lenguaje de programación visual esta clasificación aparece reflejada en el color que tienen asignadas las funciones expresivas. Azul, para las funciones de materiales; fucsia, para las geométricas y verde, para los filtros.

Aunque se trata de una consecuencia más o menos obvia, conviene notar que el establecimiento de una relación entre dos funciones expresivas geométricas no produce ninguna modificación en las propiedades de shaders del NDO y se limita copiar la caché de representación de la función emisora a la receptora. De igual forma, el establecimiento de una relación entre dos funciones de materiales no produce ninguna operación que altera la información geométrica y se limita a copiarla de la función emisora a la receptora.

Dado que el sistema maneja un único tipo de dato, el usuario es libre de componer las funciones expresivas en el orden que desee para definir un estilo visual específico. Los resultados de dicha composición de operaciones siempre son consistentes y siguen el funcionamiento descrito en la categorización anterior.

Además, otra consecuencia directa de esta separación de espacios de trabajo sobre un único tipo de dato es que el usuario no debe preocuparse de operar con las funciones expresivas en un orden específico. Por ejemplo, al contrario que otras soluciones, no es necesario agrupar todas las funciones geométricas antes de comenzar a operar con las funciones de materiales.

Cuando existen secuencias de funciones geométricas y de materiales intercaladas en un diagrama, el sistema es capaz de resolverlas en el orden correcto. Conceptualmente, se realiza de forma implícita una reordenación de las funciones tal que las funciones geométricas se ejecutan en primer lugar y, a continuación, las de materiales, siguiendo en ambos casos la secuencia descrita por el usuario para cada categoría. La Figura 8.9 describe de forma gráfica un ejemplo de este comportamiento.

Las funciones expresivas de Filtros representan puntos de inflexión en el procesamiento de los grafos y sobrescriben la información acumulada hasta entonces en los NDOs. Al contrario que las otras dos categorías, estas funciones no trabajan con la información preexistente del NDO, sino con la salida que genera. Dicho de otra manera, simplemente operan en el espacio de imagen y representan sus resultados sobre lienzos 2D. La geometría del NDO se sustituye por un cuadrado que define la forma del lienzo 2D y, análogamente, la caché de representación se sustituye con la textura que contiene la salida de las funciones previas. La Figura 8.10 describe un ejemplo gráfico de este comportamiento.

8.5.2 Persistencia de la información de color

Supóngase que se ha representado un modelo 3D con la función expresiva Cel shading y que se desea combinar el resultado obtenido con la función de Rayado. Esta situación es completamente natural para el ser humano. Los ilustradores combinan a menudo distintas técnicas con total espontaneidad en sus trabajos. Parecería lógico que las APIs escritas para producir gráficos fueran capaces de realizar esta operación aparentemente sencilla. Sin embargo, a pesar de que existen ciertos paralelismos entre el funcionamiento de las

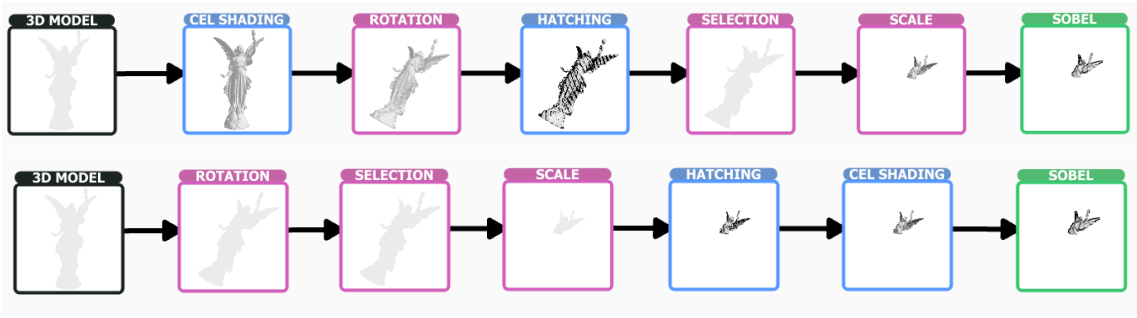


Figura 8.9 El cauce descrito en la parte superior de la figura representa la composición realizada por el usuario; el cauce descrito en la parte inferior, la reordenación implícita que realiza el sistema para resolverlo

Figura 8.10 Las funciones de Filtros crean lienzos 2D que sustituyen a la geometría original y trabajan con los resultados generados por las funciones anteriores. En la figura se puede apreciar como la función de Rayado se aplica sobre la superficie plana del lienzo en lugar de la superficie del conejo

máquinas de estados y las APIs de programación gráfica como OpenGL, estas últimas no son capaces de recordar más allá de la información de color y profundidad del frame que está siendo representado en un instante.

Por defecto, OpenGL escribe en el framebuffer *proporcionado por el sistema de ventanas* con el fin de visualizar las representaciones especificadas en pantalla. Si se desea redirigir su salida a otros destinos, se puede hacer uso de *framebuffer objects* (FBO). Asignando texturas de color o profundidad a estos objetos permite a OpenGL escribir su salida directamente sobre ellas.

Con esta funcionalidad se pueden crear, de forma efectiva, cachés que permitan reutilizar en un futuro la información que se representa en las texturas. Tal es el caso de la estructura NDO cuando se establece una relación entre funciones y las receptoras son funciones de materiales o filtros. En dichas casos, el sistema crea una textura por cada función emisora y la asigna al NDO contenido en cada una ellas. A continuación, haciendo uso de FBOs, ejecuta la secuencia de ordenes para representar los resultados de la función emisora en la textura creada. Una vez concluidas, el NDO de la función emisora es utilizado como entrada de la función expresiva receptora.

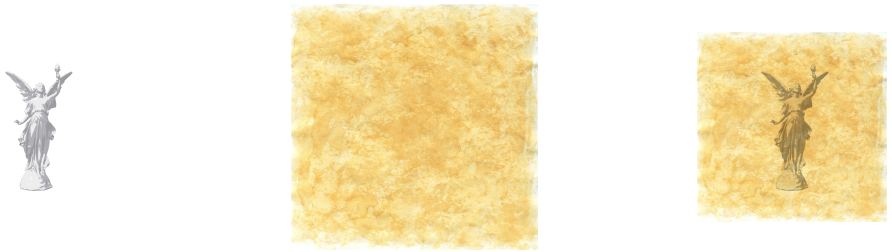


Figura 8.11 La figura muestra cómo la función de Blending opera con la imagen generada por el Cel Shading y el lienzo con el fondo de lino para realizar una mezcla aditiva de colores

A continuación se describe qué información se representa en la textura caché cuando las siguientes funciones son receptoras:

- **Filtros**: estas funciones no operan con la geometría de funciones emisoras y tampoco modifican las propiedades del material de la superficie. Se limitan a trabajar con los resultados de sus entradas en el espacio de imagen. Por tanto, las funciones expresivas emisoras son representadas utilizando el shader estándar. Generan un lienzo 2D y utilizan un G-buffer con información de profundidad si es necesario.
  La Figura 8.11 muestra como la función utiliza las imágenes generadas con el Cel shading y el fondo de lino para combinar ambos resultados en un único lienzo 2D.

- **Funciones expresivas de materiales**: estas funciones operan en el espacio de la superficie del modelo 3D, alterando su información de color. Por tanto, necesitan que la funciones emisoras representen dicha información en caché y, para lograrlo, el sistema emplea un *vertex shader* especial. En lugar de utilizar las posiciones de los vértices como coordenadas de objeto, se usan las coordenadas de textura en su lugar. Esta modificación equivaldría a desenrollar en un plano 2D el modelo 3D, dando lugar a una o más islas definidas por las coordenadas de textura. El resto de cálculos realizados por el vertex shader se mantienen en el espacio 3D para garantizar que producen el efecto deseado sobre la superficie del modelo.
  La Figura 8.12 muestra la malla desenrrollada de un esfera en el espacio de textura y como la función de Cel Shading se representa en ella para que, posteriormente, la función de Rayado pueda utilizar los resultados de forma correcta.

8.5.3 Procesando información geométrica

Se pueden distinguir dos casos bien diferenciados cuando se compone una operación en la cual las funciones expresivas Geométricas actúan como receptoras:

- **Funciones de transformación**: representa el caso más sencillo de esta familia de funciones. Utiliza la información almacenada en la matriz de transformación de la función emisora para componer el resultado con la matriz que refleja la operación que se está realizando (Traslación, Rotación o Escalado).

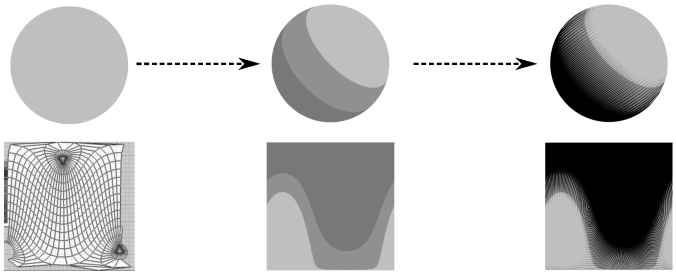


Figura 8.12 La parte superior de la figura muestra la composición de una función de Cel Shading con un Rayado sobre una esfera. En la parte inferior se presenta la parametrización de la esfera en el espacio de textura y cómo se almacenan los resultados de las dos operaciones en dicha textura

Función de selección: representa el único caso en el cual una función expresiva trabaja de forma híbrida tanto en el espacio de GPU como CPU. Utiliza un FBO y un shader especial que escribe en una textura 2D el índice de los triángulos seleccionados por el usuario en el Visualizador 3D. Dicha textura es posteriormente analizada en CPU para crear una caché de triángulos seleccionados en el NDO y modificar su propiedad de color en los arrays de vértices con un color distintivo.
Los cambios en la geometría se retrasan hasta que esta función expresiva se enlaza con cualquier otra. En dicho instante se creará un nuevo array de vértices que contendrá únicamente los triángulos seleccionados y, al mismo tiempo, se actualizará la estructura de semiaristas para reflejar dicho cambio en la topología.
La Figura 8.13 muestra la selección de la parte superior de una escultura y cómo esta geometría es la única con la que se opera una vez se aplica la función de Rayado.

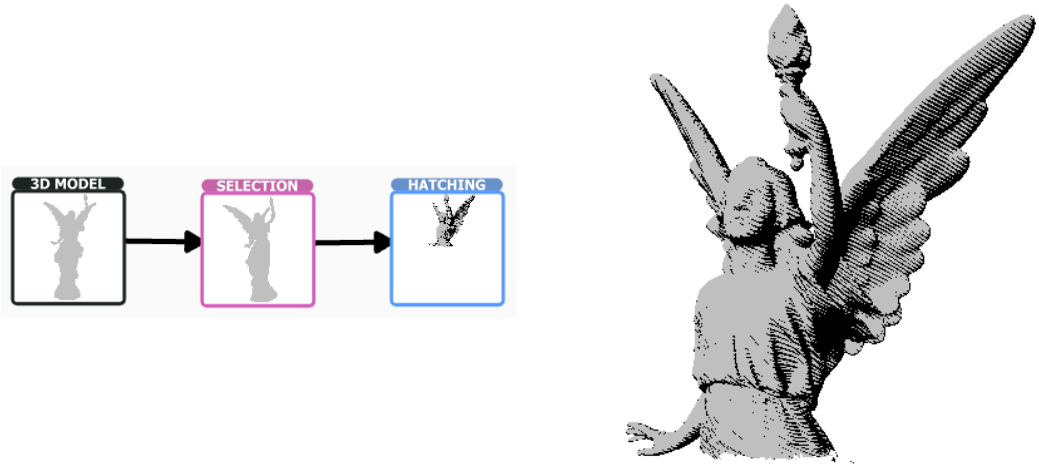


Figura 8.13 La función de selección permite delimitar los resultados posteriores a las áreas de la superficie de un modelo 3D que se han seleccionado

8.6 Prototipo y resultados

Como se puede observar en la Figura 8.14 la interfaz de usuario del prototipo desarrollado se encuentra estructurada en tres secciones:

- **Editor**: permite crear nuevos estilos de visualización mediante la composición de funciones expresivas. Se divide a su vez en dos secciones: la hoja de edición y el selector de funciones expresivas. El usuario puede seleccionar con el botón izquierdo la función deseada y, con una pulsación adicional, colocarla en la hoja de edición. Las propiedades escalares y aquellas relacionadas con los materiales se pueden editar en una ventana desplegable al hacer doble click con el botón izquierdo del ratón sobre el icono de la función. Para establecer una relación entre dos funciones expresivas, basta con hacer un click con el botón central del ratón sobre la función emisora y otro sobre la función receptora.

- **Visualizador**: representa los resultados de procesar los grafos de funciones descritos en la hoja de edición. Si una función expresiva se encuentra seleccionada, se visualizarán los manejadores gráficos de propiedades vectoriales como la dirección de la luz, las líneas de rayado o los ejes de rotación.

- **Conmutadores de visualización**: permiten modificar el estado de visualización del Editor y el Visualizador con un click del botón izquierdo del ratón. Por defecto al menos una de las secciones ha de ser visible. De esta manera el usuario puede, por ejemplo, dedicar todo el espacio de la aplicación a visualizar los resultados.

Figura 8.14 Captura del prototipo desarrollado

La función expresiva identidad que aparece en la interfaz como **3D MODEL** permite cargar una imagen o un modelo y, una vez procesados, el sistema se encarga de crear la estructura NDO que recibe la función para representarlos en el Visualizador. El icono de la función se actualiza con una previsualización de dicha representación.

De igual forma, cada vez que se establece una relación entre funciones expresivas, el icono de la función receptora también se actualiza con una previsualización de los resultados que se pueden observar en el Visualizador. Si el usuario altera cualquiera de las propiedades de la función, la previsualización se actualiza en concordancia. Este comportamiento es extensible a los casos de propagación donde se modifican todas las previsualizaciones de las funciones sucesoras de aquella que se edita en la hoja de edición.

Cabe destacar también la función **GROUP** (Agrupación) que permite compactar complejos estilos de visualización expresiva en una única función. La Figura 8.15 muestra un ejemplo de su aplicación: *(a)* se crea el cauce; *(b)* se utiliza GROUP y se seleccionan aquellas funciones que se desean agrupar definiendo un área cuadrada que las contiene; *(c)* el grupo se crea de forma automática y, opcionalmente, se puede modificar su nombre haciendo click con el botón izquierdo sobre el texto; *(d)* para abrir el grupo, se hace doble clic sobre su icono y, de forma análoga, se puede volver a compactar haciendo click sobre el área que ocupa.

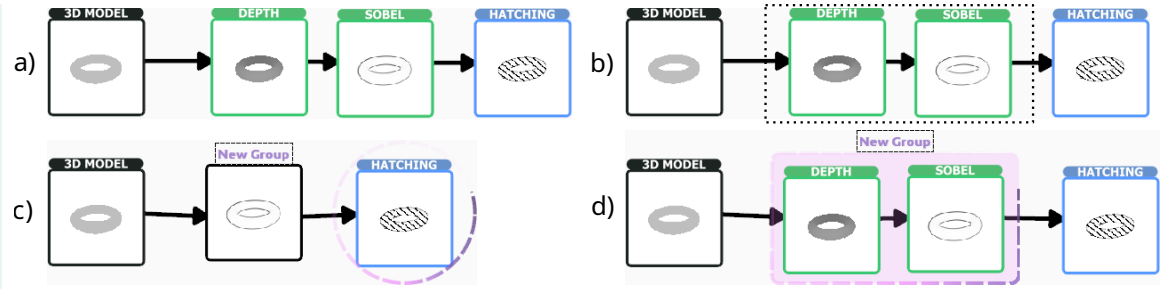


Figura 8.15 Ejemplo de agrupación de múltiples funciones expresivas en una sola función

Cuando el grafo descrito en la hoja de edición contiene más de dos funciones terminales el motor de representación utiliza simplemente el buffer de profundidad para componer la imagen en el orden correcto. Ese es exactamente el mismo comportamiento que se contempla cuando se utiliza la función Z-Blend para componer resultados. Se puede forzar un orden de representación específico utilizando la función expresiva Blend, que ignora el buffer de profundidad y dibuja los elementos en el orden expresado.

A continuación se muestran algunas capturas con distintos estilos de visualización expresiva diseñados con el prototipo en la Figura 8.16 y la Figura 8.17.

Figura 8.16 Distintos estilos creados sobre el ángel Lucy utilizando un lienzo de lino de fondo

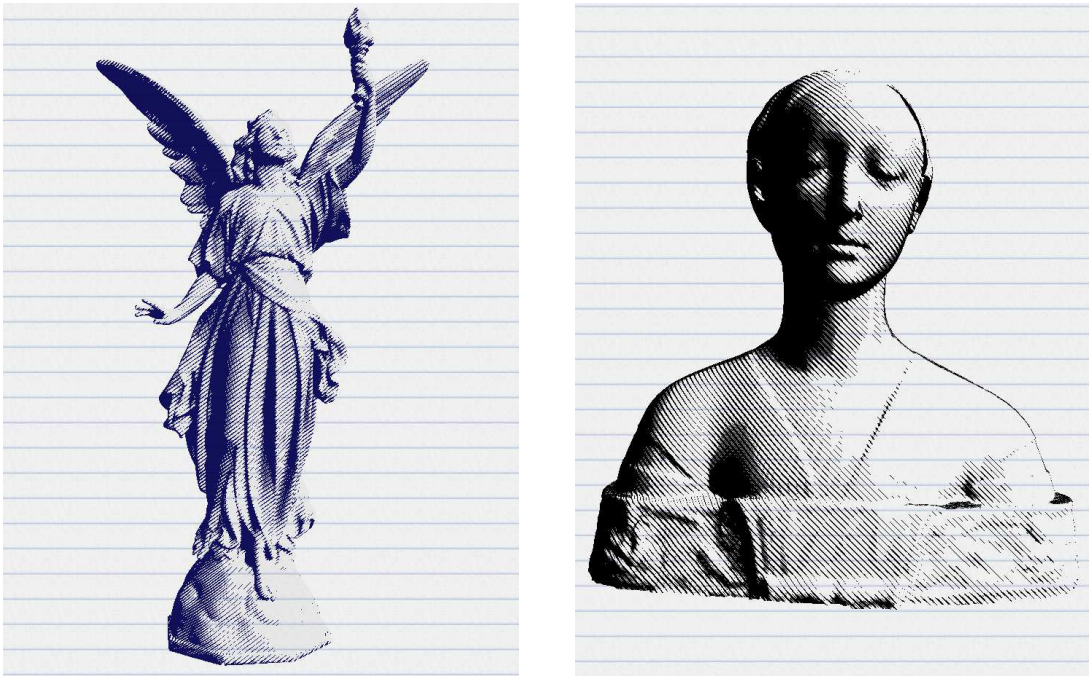


Figura 8.17 A la izquierda se puede ver el ángel Lucy utilizando una combinación de técnicas de rayado y blending. A la derecha se ofrece una combinación de estilos similar con el busto de Ippolita Sforza

8.7 Conclusiones

A lo largo de este capítulo se ha desarrollado un sistema capaz de crear sofisticados estilos visuales mediante la composición de funciones expresivas más simples. Entre su funcionalidad destacan las siguientes características:

- Dispone de un lenguaje de programación visual intuitivo para operar con funciones expresivas.
- Sólo utiliza un único tipo de dato para almacenar la información. El usuario puede centrarse en los resultados sin necesidad de conocer el orden interno de procesamiento de las operaciones.
- Los resultados producidos siempre son predecibles.
- Implementa multitud de algoritmos que operan a nivel geométrico, sobre las propiedades de color de la superficie del modelo o aplican filtros 2D. Aunque la mayoría de los algoritmos se resuelven en el espacio de GPU, también existen aquellos que se ejecutan en CPU. En dichos casos, se intenta minimizar las transferencias entre CPU y GPU.
- Cuando se crean composiciones de funciones expresivas complejas, mantiene un sistema de caché de resultados parciales con el fin de mantener la máxima eficiencia posible. Sólo se computan los cambios realizados y se propagan los resultados únicamente a aquellas funciones que los utilizan.

Las ilustraciones elaboradas con el sistema pueden cubrir multitud de necesidades, desde intereses puramente recreativos hasta la documentación de patrimonio histórico.

Parte del trabajo descrito en este capítulo se ha publicado en la revista *Virtual Archaeology Review* **[LAM12]**.

9

Conclusiones

A lo largo de este trabajo de tesis doctoral se han cumplido los siguientes objetivos:

- Se ha desarrollado una nueva arquitectura para sistemas de información 3D que utiliza texturas 2D para almacenar los datos de capas temáticas de información. Su diseño permite que los datos residan en memoria de GPU y el proceso de edición se solvente íntegramente en GPU. Comparado con soluciones anteriores de sistemas basados en capas, esta solución ha demostrado ser mucho más eficiente a la hora de crear las estructuras y editar la información sobre la superficie de modelos 3D, llegando a existir una diferencia de hasta dos órdenes de magnitud en las pruebas realizadas.

- Se ha diseñado un algoritmo capaz de almacenar la información topológica de modelos 3D en texturas 2D. También se calcula íntegramente en el espacio de GPU y permite ampliar la funcionalidad de la arquitectura propuesta. Tanto el cálculo de las estructuras durante la fase de preprocesamiento como el cómputo de la topología en la texturas es significativamente más rápido que otras soluciones que utilizan octrees. Las pruebas de rendimiento realizadas muestran que la diferencia al calcular la topología puede alcanzar cinco órdenes de magnitud en los casos de máximo nivel de detalle.

- Se ha diseñado un lenguaje de operación de capas para este sistema que, entre otras funcionalidades, permite realizar operaciones aritmeticológicas entre capas y trabajar de forma transparente con los campos numéricos de las tablas asociadas a las capas de bases de datos. El nuevo traductor añadido a la arquitectura permite generar código eficiente de shader a partir de sentencias descritas en dicho lenguaje. Esto permite que todas las operaciones se calculen en el espacio de GPU y se eviten transferencias innecesarias de información entre en el espacio de CPU y espacio de GPU.

- Se ha realizado un estudio de usabilidad del sistema de información 3D desarrollado. El estudio ha arrojado luz a problemas de navegabilidad y presentación en la interfaz de usuario que se han de solventar. Al mismo tiempo ha permitido validar la funcionalidad ofrecida por el sistema con bastantes buenos resultados entre personal especializado.

- Para finalizar se ha desarrollado un prototipo que permite documentar gráficamente el patrimonio tangible empleando un intuitivo lenguaje de programación visual. Mediante la composición de funciones expresivas sencillas, esta herramienta permite crear estilos visuales variados para representar modelos 3D e imágenes de multitud de formas. La gran mayoría de algoritmos utilizados y el sistema de caché implementado se resuelven también en el espacio de GPU.

Como trabajo futuro más inmediato, se pretende integrar el lenguaje de programación visual para la documentación gráfica en el sistema de información 3D desarrollado. Al mismo tiempo, se desea investigar cómo se podría incorporar la funcionalidad aportada por

sistemas de inteligencia artificial como DALL·E [Dal23] a este proceso de documentación gráfica.

También se quiere explorar qué nuevas posibilidades puede ofrecer la inclusión del cauce programable de trazado de rayos en GPUs dentro del contexto de sistemas de información 3D. ¿Qué nuevas optimizaciones se podrían aplicar tanto a la asignación de información como a la visualización de modelos 3D?

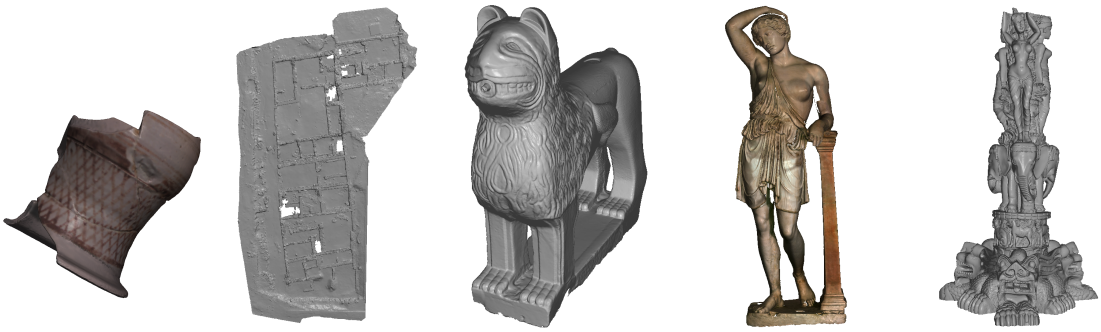
Apéndice del Capítulo 4

A.1 Resultados de las pruebas

Se compara el rendimiento del prototipo basado en la arquitectura propuesta con el ofrecido por otro sistema basado en un octree. Específicamente, se ha seleccionado el sistema diseñado por Torres et al. **[TCM*10]**, notado como OCT-TR.

Todas las pruebas realizadas testan el rendimiento del proceso de edición de capas en escenarios equivalentes. Para la solución propuesta, dicho proceso involucra los dos algoritmos explicados en el Capítulo 4: Edición de Textura y Relleno de Textura. En el caso de OCT-TR, es necesario que la CPU lance múltiples rayos para encontrar con qué vóxeles del octree colisionan, actualizar los valores de la capa de forma apropiada y transferir la versión actualizada de la misma a la GPU.

Se han elegido tres niveles de detalle para las capas de información con el fin de analizar el rendimiento cuando se reduce el tamaño de las celdas: tres resoluciones de textura para la solución propuesta y tres profundidades distintas para OCT-TR. Además, para cada uno de esos casos, se han seleccionado cinco tamaños de la herramienta de edición para poder analizar el rendimiento cuando el área editada se incrementa. Para todas las pruebas se han utilizado los mismos cinco modelos 3D representados en la Figura A.1.



(a) Vasija	(b) Yacimiento	(c) León	(d) Amazona	(e) Thai
312,094 triángulos	999,942 triángulos	3,000,223 triángulos	5,711,204 triángulos	10,000,000 triángulos
$105.18 \times 98.53 \times 91.41\ mm$	$36.91 \times 67.37 \times 4.52\ m$	$10.24 \times 10.98 \times 3.09\ dm$	$76.38 \times 200.36 \times 58.06\ mm$	$23.52 \times 39.60 \times 20.32\ cm$

Figura A.1 Modelos 3D utilizados en las pruebas. Modelos (a) y (d) disponen de colores por vértice mientras que (b), (c) y (d) utilizan un color gris estándar. Están ordenados de izquierda a derecha en función del número de triángulos que los componen. Las dimensiones también se detallan en cada modelo

Dado que los octrees son estructuras espaciales volumétricas y las texturas son estructuras espaciales 2D, no existe una comparación directa posible entre el área de la superficie contenida en un vóxel y el área contenida en un téxel. Por consiguiente, se han realizado múltiples pruebas para determinar una correspondencia entre el tamaño de celda proporcionado por distintas resoluciones de textura y profundidades de octree. Empíricamente se han obtenido los siguientes resultados: la resolución de textura 2048×2048 ofrece

un tamaño de celda similar al ofrecido por una profundidad de octree 11; la resolución de textura 4096×4096, al ofrecido por una profundidad de octree 12 y la resolución de textura 8192×8192, al ofrecido por una profundidad de octree 13.

Mientras que el método propuesto no tuvo problema para manejar el modelo Thai, se ha de notar que los tests de OCT-TR para los profundidades 12 y 13 no pudieron completarse.

La Tabla A.1 y la Tabla A.2 contienen todos los resultados de las pruebas realizadas para la solución propuesta y OCT-TR respectivamente.

La Figura A.2 y Figura A.3 muestran, en escala logarítmica, como evoluciona el rendimiento cuando el área editada cambia para cada uno de los cinco modelos. Asimismo, la Figura A.4 y Figura A.5 muestran, también en escala logarítmica, como evoluciona el rendimiento cuando se edita una capa de cada modelo con el mismo tamaño de la herramienta de edición.

La Tabla A.3 muestra los tamaños de celda medios obtenidos por ambos métodos para cada modelo.

Tam.	Vasija			Yacimiento			León			Amazona			Thai		
Herr.	2048	4096	8192	2048	4096	8192	2048	4096	8192	2048	4096	8192	2048	4096	8192
10	0,86	1,89	5,61	2,42	5,01	13,11	5,35	9,14	17,70	5,45	9,74	18,45	7,27	13,56	22,26
40	0,86	1,90	5,64	2,42	5,01	13,14	5,36	9,14	17,85	5,44	9,76	18,48	7,28	13,59	22,38
70	0,86	1,91	5,68	2,43	5,02	13,14	5,35	9,15	17,86	5,45	9,77	18,75	7,28	13,61	22,61
100	0,86	1,91	5,74	2,44	5,04	13,18	5,38	9,15	17,86	5,44	9,78	18,54	7,28	13,62	22,52
200	0,89	2,01	5,89	2,51	5,06	13,18	5,38	9,16	17,86	5,48	9,85	18,77	7,29	13,64	23,72

Tabla A.1 Resultados de las pruebas para la solución propuesta en milisegundos. Se han utilizado tres resoluciones de capa por modelo: 2048×2048, 4096×4096 y 8192×8192. Para cada una de estas resoluciones, se han editado las capas empleando 5 radios para la herramienta de edición: 10, 40, 70, 100 y 200 píxeles

Tam.	Vasija			Yacimiento			León			Amazona			Thai		
Herr.	11	12	13	11	12	13	11	12	13	11	12	13	11	12	13
10	4,43	12,12	39,00	2,00	4,00	9,80	2,16	8,63	32,75	4,00	7,64	18,60	3,71	–	–
40	19,86	46,25	133,00	6,16	13,67	39,25	7,38	28,50	90,25	9,75	29,75	67,00	16,29	–	–
70	38,75	99,00	290,83	11,29	33,25	81,11	14,50	51,88	159,00	21,50	55,30	156,72	31,86	–	–
100	55,50	174,14	533,00	14,86	43,57	127,37	35,70	92,25	360,33	36,22	81,43	285,28	59,00	–	–
200	134,75	432,22	1,469,00	26,50	85,66	303,42	77,30	292,13	1,052,16	46,00	148,40	477,00	99,00	–	–

Tabla A.2 Resultados de las pruebas de OCT-TR en milisegundos. Se han utilizado tres profundidades de octree por modelo: 11, 12 y 13. Para cada uno de esas profundidades, se han editado las capas empleando 5 radios distintos para la herramienta de edición: 10, 40, 70, 100 y 200 píxeles

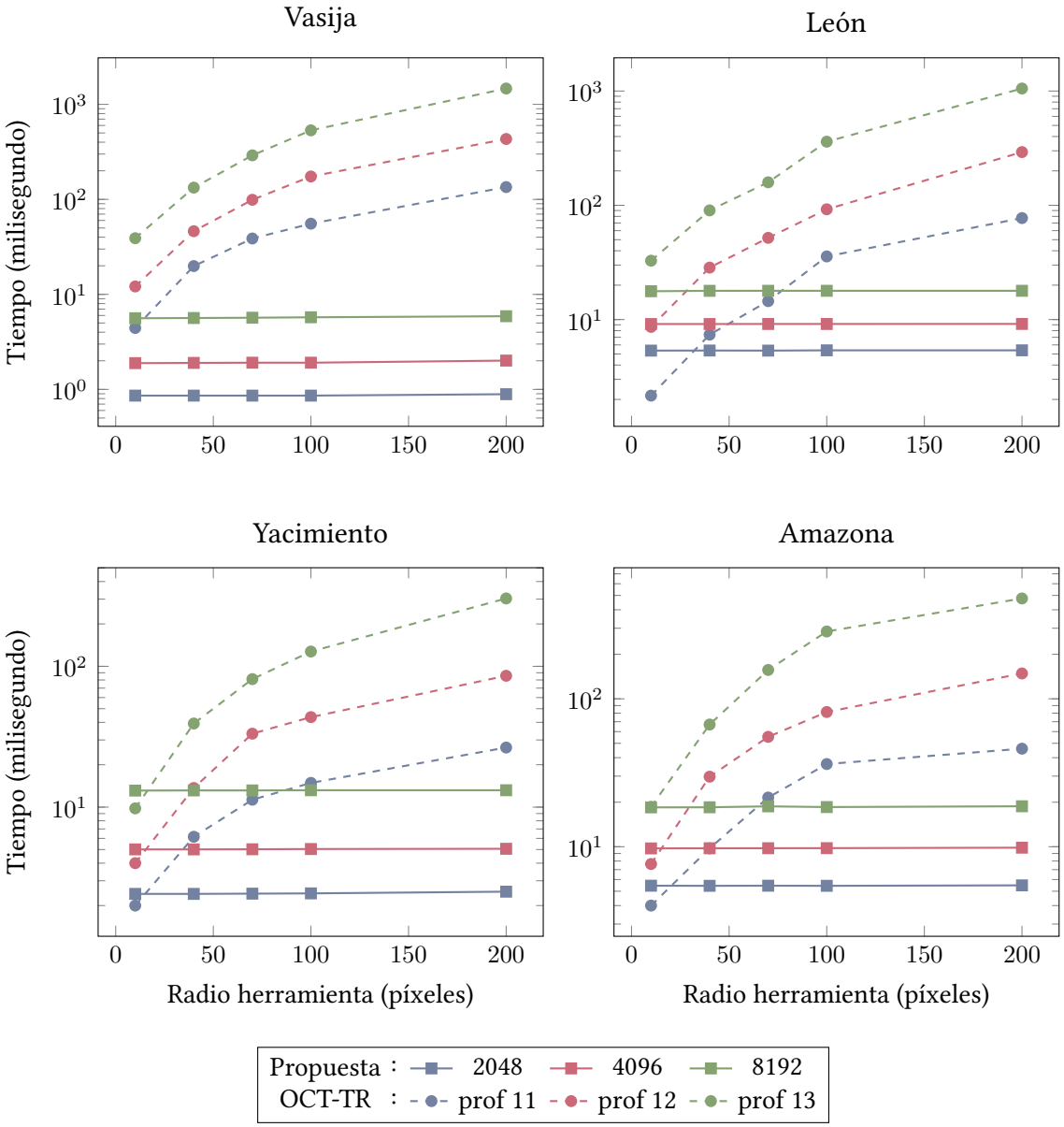


Figura A.2 Estas gráficas muestran los resultados de las pruebas, en escala logarítmica, para la edición de capas de cuatro modelos: a) Vasija, b) Yacimiento, c) Leon, d) Amazona. *Radio herramienta* se corresponde con el radio de la herramienta de edición en píxeles. Las líneas sólidas representan la solución propuesta, mientras que las líneas discontinuas representan OCT-TR

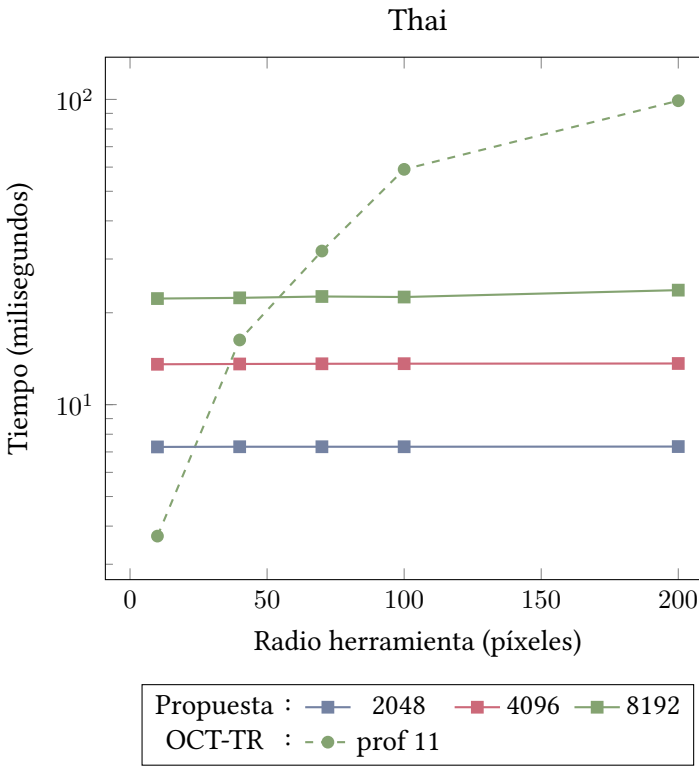


Figura A.3 Estas gráficas muestran los resultados de las pruebas, en escala logarítmica, para la edición de capas del modelo Thai. *Radio herramienta* se corresponde con el radio de la herramienta de edición en píxeles. Las líneas sólidas representan la solución propuesta, mientras que las líneas discontinuas representan OCT-TR

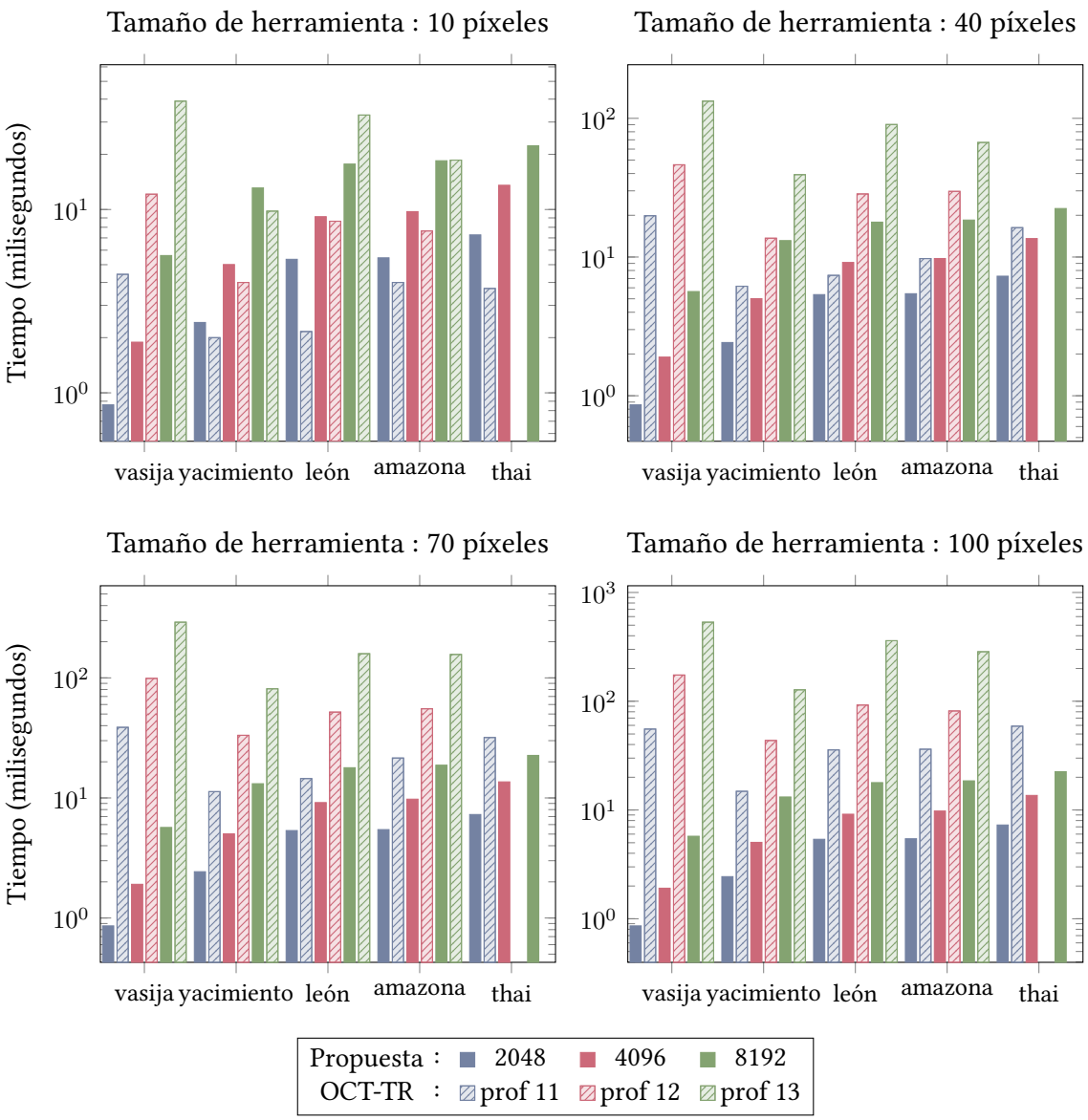


Figura A.4 Estas gráficas muestran los resultados de las pruebas, en escala logarítmica, para la edición de capas empleando el mismo tamaño de la herramienta de edición en cada modelo. Los radios utilizados son 10, 40, 70, 100 píxeles. Los modelos 3D están organizados en función de su número de triángulos. Las líneas sólidas representan la solución propuesta, mientras que las líneas discontinuas representan OCT-TR

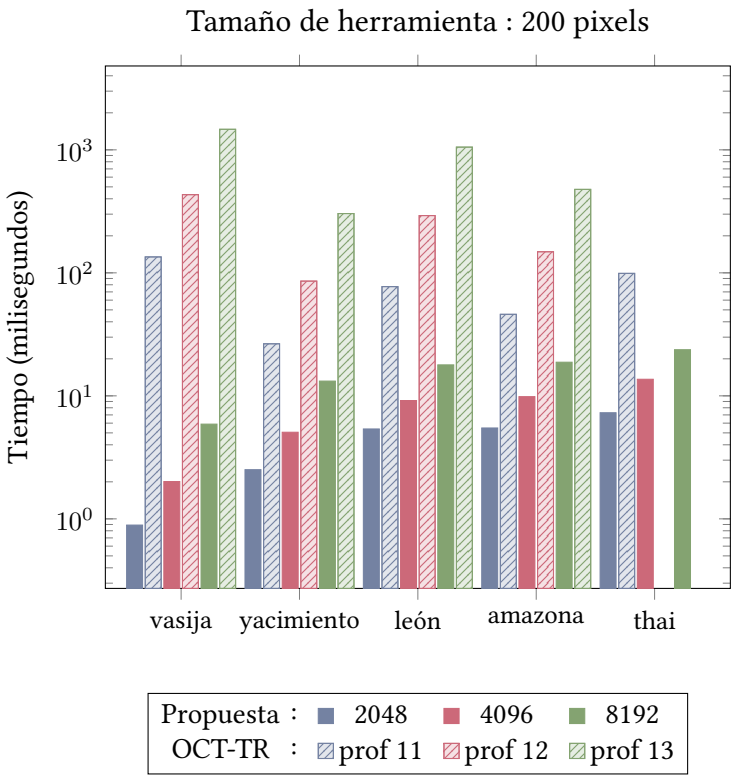


Figura A.5 Estas gráficas muestran los resultados de las pruebas, en escala logarítmica, para la edición de capas empleando el mismo tamaño de la herramienta de edición en cada modelo. El radio utilizado es 200 píxeles. Los modelos 3D están organizados en función de su número de triángulos. Las líneas sólidas representan la solución propuesta, mientras que las líneas discontinuas representan OCT-TR

Método	2048	4096	8192
	prof 11	prof 12	prof 13
Propuesto	0.0076	0.0019	0.0005
OCT-TR	0.0055	0.0014	0.0003

(a) Vasija (mm^2)

Método	2048	4096	8192
	prof 11	prof 12	prof 13
Propuesto	8.71	2.18	0.54
OCT-TR	36.09	9.00	2.25

(b) Yacimiento (cm^2)

Método	2048	4096	8192
	prof 11	prof 12	prof 13
Propuesto	0.7702	0.1926	0.0481
OCT-TR	0.8752	0.2188	0.0547

(c) León (mm^2)

Método	2048	4096	8192
	prof 11	prof 12	prof 13
Propuesto	0.0239	0.0060	0.0015
OCT-TR	0.0275	0.0069	0.0017

(d) Amazona (mm^2)

Método	2048	4096	8192
	prof 11	prof 12	prof 13
Propuesto	0.1187	0.0297	0.0074
OCT-TR	0.1066	-	-

(e) Thai (mm^2)

Tabla A.3 Las tablas muestran el tamaño de celda medio obtenido por ambos algoritmos para cada modelo. Las medidas se han realizado en centímetros cuadrados para b) Yacimiento y milímetros cuadrados para a) Vasija, c) León, d) Amazona y e) Thai. La cabecera de las columnas muestra los tamaños de textura (coloreado) y profundidades de octree (blanco) que se han empleado con cada modelo. Las pruebas de OCT-TR con profundidades 12 y 13 para e) Thai no se pudieron completar

B

Apéndice del Capítulo 5

B.1 Pruebas de rendimiento

Se han seleccionado los cinco modelos 3D representados en Figura B.1 para testar el rendimiento de la solución propuesta. Dichos modelos ofrecen una amplia variedad en el número de triángulos que los conforman y se han creado a partir de modelos escaneados 3D y, por tanto, representan escenarios reales en términos de complejidad topológica. Asimismo, la forma cilíndrica de Vasija, la superficie planar de Yacimiento o la figura humanoide de Amazona proporcionan parametrizaciones de malla distintivas.

Para obtener una visión mas detallada de su rendimiento, se han comparado la solución propuesta con otro método que también utiliza estructuras topológicas de datos basadas en celdas y que ha diseñado por Torres et al. **[TCM*10, STLL13]**. Este sistema, que notamos como OCT-TR, trabaja con octrees para indexar la geometría y asociar información a los modelos 3D con independencia del número de triángulos que los componen.

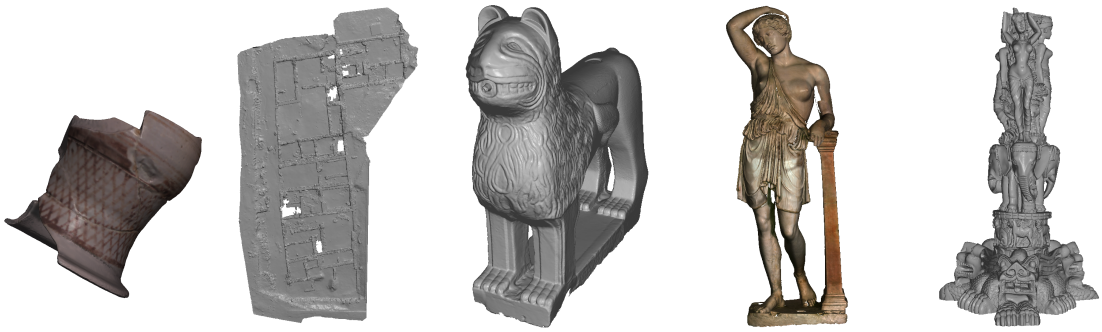
Dado que los octrees son estructuras espaciales volumétricas y las texturas son estructuras espaciales 2D, no existe una comparación directa posible entre el área de la superficie contenida en un vóxel y el área contenida en un téxel. Por consiguiente, se han realizado múltiples pruebas para determinar una correspondencia entre el tamaño de celda proporcionado por distintas resoluciones de textura y profundidades de octree. Empíricamente se han obtenido los siguientes resultados: la resolución de textura 2048×2048 ofrece un tamaño de celda similar al ofrecido por una profundidad de octree 11; la resolución de textura 4096×4096, al ofrecido por una profundidad de octree 12 y la resolución de textura 8192×8192, al ofrecido por una profundidad de octree 13.

Mientras que el método propuesto no tuvo problema para manejar el modelo Thai, se ha de notar que los tests de OCT-TR para los profundidades 12 y 13 no pudieron completarse.

La Tabla B.1 muestra los tamaños de celda medios obtenidos por ambos métodos para cada modelo. La cabecera de cada columna muestra la resolución de textura a la izquierda y la profundidad del octree a la derecha.

B.1.1 Resultados de las pruebas de rendimiento

La Tabla B.2 muestra cuantos milisegundos emplean la solución propuesta y OCT-TR en calcular las estructuras y la información topológica para cada modelo utilizando tres niveles de detalle agrupados por resoluciones de textura y profundidad de octree con tamaños de celda similares.



(a) Vasija
312,094 triángulos
$105.18 \times 98.53 \times 91.41\ mm$

(b) Yacimiento
999,942 triángulos
$36.91 \times 67.37 \times 4.52\ m$

(c) León
3,000,223 triángulos
$10.24 \times 10.98 \times 3.09\ dm$

(d) Amazona
5,711,204 triángulos
$76.38 \times 200.36 \times 58.06\ mm$

(e) Thai
10,000,000 triángulos
$23.52 \times 39.60 \times 20.32\ cm$

Figura B.1 Modelos 3D utilizados en las pruebas. Modelos (a) y (d) disponen de colores por vértice mientras que (b), (c) y (d) utilizan un color gris estándar. Están ordenados de izquierda a derecha en función del número de triángulos que los componen. Las dimensiones también se detallan en cada modelo

Método	2048-11	4096-12	8192-13
Propuesto	0.0076	0.0019	0.0005
OCT-TR	0.0055	0.0014	0.0003

(a) Vasija (mm^2)

Método	2048-11	4096-12	8192-13
Propuesto	8.71	2.18	0.54
OCT-TR	36.09	9.00	2.25

(b) Yacimiento (cm^2)

Método	2048-11	4096-12	8192-13
Propuesto	0.7702	0.1926	0.0481
OCT-TR	0.8752	0.2188	0.0547

(c) León (mm^2)

Método	2048-11	4096-12	8192-13
Propuesto	0.0239	0.0060	0.0015
OCT-TR	0.0275	0.0069	0.0017

(d) Amazona (mm^2)

Método	2048-11	4096-12	8192-13
Propuesto	0.1187	0.0297	0.0074
OCT-TR	0.1066	-	-

(e) Thai (mm^2)

Tabla B.1 Las tablas muestran el tamaño de celda medio obtenido por ambos algoritmos para cada modelo. Las medidas se ofrecen en centímetros cuadrados para b) Yacimiento y milímetros cuadrados para a) Vasija, c) León, d) Amazona y e) Thai. La cabecera de cada columna muestra la resolución de textura a la izquierda y la profundidad del octree a la derecha. Las pruebas de OCT-TR con profundidad 12 y 13 para el modelo e) Thai. No pudieron completarse

Tamaño	Estructura		Topología	
	Semiarista	Octree	Textura	Octree
2048-11	636	32634	0.69	13464
4096-12	636	62488	2.42	54487
8192-13	636	209412	9.82	234379

(a) Vasija

Tamaño	Estructura		Topología	
	Semiarista	Octree	Textura	Octree
2048-11	7539	70901	0.69	2465
4096-12	7539	86059	2.45	9144
8192-13	7539	116186	9.68	36688

(b) Yacimiento

Tamaño	Estructura		Topología	
	Semiarista	Octree	Textura	Octree
2048-11	8987	265952	0.61	11407
4096-12	8987	455539	2.14	43921
8192-13	8987	618486	7.23	182886

(c) León

Tamaño	Estructura		Topología	
	Semiarista	Octree	Textura	Octree
2048-11	17860	435635	1.68	6334
4096-12	17860	559859	6.52	23233
8192-13	17860	769249	21.55	90284

(d) Amazona

Tamaño	Estructura		Topología	
	Semiarista	Octree	Textura	Octree
2048-11	30886	717810	2.23	12253
4096-12	30886	0	11.19	0
8192-13	30886	0	38.54	0

(e) Thai

Tabla B.2 Las tablas muestras cuantos milisegundos emplean la solución propuesta y OCT-TR en calcular las estructuras y la información topológica para cada modelo utilizando tres niveles de detalle agrupados por resoluciones de texturas y profundidades de octree con tamaños de celda similares. Las pruebas de OCT-TR para el modelo e) Thai con profundidades 12 y 13 no pudieron completarse

B.2 Pruebas de precisión

Con el fin de medir la precisión del algoritmo propuesto, se ha seleccionado el cálculo de campos de distancias porque dicho cómputo necesita la información topológica para ofrecer resultados coherentes y, además, estos resultados se pueden comparar con soluciones matemáticamente correctas.

Se han calculado campos de distancias utilizando el método propuesto en tres escenarios distintos que mantienen el mismo téxel de entrada y modifican tres variables: el tamaño de la textura, la parametrización de la malla y el número de triángulos. Asimismo, se ha utilizado un simple cubo de dimensiones $2\ m \times 2\ m \times 2\ m$ con el fin de calcular las distancias matemáticamente correctas de forma sencilla y compararlas con los resultados obtenidos por el método propuesto. Concretamente, dado un mismo téxel de entrada, se ha calculado la diferencia por téxel entre el resultado del método propuesto y el caso matemáticamente correcto y ésta se ha comparado con la mayor distancia proporcionada por el campo de distancia de la solución correcta con el fin de obtener el error relativo.

La Tabla B.3(a) y la Tabla B.3(c) muestran los errores relativos máximos y medios para los cuatro casos iniciales. Asimismo, la Tabla B.3(e) muestra la distancia más grande proporcionada por la solución matemáticamente correcta. Todos estos casos mantienen la forma de las cuatro islas de textura pero subdividen las caras con distintos ratios. Caso C1 es la versión original con sólo 12 triángulos, dos por cara. Caso C2 subdivide los triángulos originales e incrementa su número hasta $12 \times 441 = 5292$. Caso C3 incrementa el número de triángulos hasta $12 \times 26244 = 314928$ y caso C4 hasta $12 \times 164025 = 1968300$.

La Tabla B.3(b) y la Tabla B.3(d) muestran los errores relativos máximos y medios para los cuatro casos que modifican la forma y el número de las islas de textura mientras mantienen la triangulación de la superficie. Asimismo, la Tabla B.3(f) muestra la distancia más grande proporcionada por la solución matemáticamente correcta. Caso C5 contiene las cuatro islas y 5292 triángulos. Caso C6 descompone estas cuatro islas en 64 más pequeñas con un número variable de triángulos. De igual forma, Caso C7 las descompone en 292 y Caso C8 en 5291.

Además, los ocho casos anteriormente mencionados también se han evaluado con tres tamaños de textura distintos: 2048×2048, 4096×4096, 8192×8192.

Tamaño	C1 12 t	C2 5292 t	C3 $3x10^5$ t	C4 $2x10^6$ t
2048	0,3283	0,3283	0,3283	0,3283
4096	0,3267	0,3267	0,3267	0,3267
8192	0,3277	0,3273	0,3273	0,3272

(a) Error máximo manteniendo las islas

Tamaño	C5 4 i	C6 64 i	C7 292 i	C8 5291 i
2048	0,3283	0,3233	0,3056	0,1795
4096	0,3267	0,3251	0,3126	0,1968
8192	0,3273	0,3235	0,3162	0,1497

(b) Error máximo manteniendo la triangulación

Tamaño	C1 12 t	C2 5292 t	C3 $3x10^5$ t	C4 $2x10^6$ t
2048	0,1170	0,1170	0,1170	0,1169
4096	0,1165	0,1165	0,1164	0,1164
8192	0,1135	0,1135	0,1134	0,1134

(c) Error medio manteniendo las islas

Tamaño	C5 4 i	C6 64 i	C7 292 i	C8 5291 i
2048	0,1170	0,1149	0,1041	0,0340
4096	0,1165	0,1156	0,1086	0,0561
8192	0,1135	0,1122	0,1104	0,0399

(d) Error medio manteniendo la triangulación

Tamaño	C1 12 t	C2 5292 t	C3 $3x10^5$ t	C4 $2x10^6$ t
2048	4,1354	4,1354	4,1354	4,1354
4096	4,1362	4,1362	4,1362	4,1362
8192	4,1367	4,1367	4,1367	4,1367

(e) Máxima distancia manteniendo las islas

Tamaño	C5 4 i	C6 64 i	C7 292 i	C8 5291 i
2048	4,1354	4,1353	4,1343	4,1343
4096	4,1362	4,1361	4,1353	4,1360
8192	4,1367	4,1365	4,1365	4,1366

(f) Máxima distancia manteniendo la triangulación

Tabla B.3 Las tablas muestran los errores máximos y medios para cada uno de los ocho escenarios testados y la máxima distancia proporcionada por la solución matemáticamente correcta. Los casos mostrados a la izquierda mantienen la forma de las cuatro islas de textura y subdividen las caras utilizando distintos ratios. Los casos de la derecha modifican la forma de las islas mientras mantienen la triangulación de la superficie

C

Apéndice del Capítulo 6

C.1 Léxico

```
BKNL            : '\\' WS? '\n' -> skip;

SHARP           : '#';

OPENPAR         : '(';
CLOSEPAR        : ')';
OPENBRACK       : '[';
CLOSEBRACK      : ']';
ASSIGN          : '=';
COLON           : ':';
SEMICOLON       : ';';
COMMA           : ',';

NEGSUB          : '-';
COMPLEMENT      : '~';
EXPONENT        : '^';
MODULUS         : '%';
DIV             : '/';
MULT            : '*';
ADD             : '+';
LSHIFT          : '<<';
RSHIFT          : '>>';
RSHIFTU         : '>>>';
GREATER         : '>';
GREATEREQUAL    : '>=';
LESS            : '<';
LESSEQUEAL      : '<=';
EQUAL           : '==';
NEQUAL          : '!=';
BITAND          : '&';
BITOR           : '|';
LAND            : '&&';
LANDONE         : '&&&';
LOR             : '||';
LORONE          : '|||';
LNOT            : '!';
COND            : '?';

ABS             : 'abs';
ACOS            : 'acos';
ASIN            : 'asin';
ATAN            : 'atan';
CEIL            : 'ceil';
COS             : 'cos';
```

```
DOUBLECAST      : 'double';
EVAL            : 'eval';
EXPFUNC         : 'exp';
FLOATCAST       : 'float';
FLOOR           : 'floor';
GRAPH           : 'graph';
GRAPH2          : 'graph2';
IF              : 'if';
INTCAST         : 'int';
ISNULL          : 'isnull';
LOG             : 'log';
MAX             : 'max';
MEDIAN          : 'median';
MIN             : 'min';
MODE            : 'mode';
NMAX            : 'nmax';
NMEDIAN         : 'nmedian';
NMIN            : 'nmin';
NMODE           : 'nmode';
NOT             : 'not';
POW             : 'pow';
RAND            : 'rand';
ROUND           : 'round';
SIN             : 'sin';
SQRT            : 'sqrt';
TAN             : 'tan';
XOR             : 'xor';
NULLV           : 'null';
SQL             : 'sql';

FLOAT           : INT '.' INT? EXP? FL?
                | '.' INT EXP? FL?;
fragment FL     : [fF];
fragment EXP    : [eE][-+]? DIGIT+;
DOUBLE          : INT '.' INT? EXP?
                | '.' INT EXP?;
INT             : DIGIT+ | HEX;
fragment DIGIT  : [0-9];
fragment HEX    : [0][xX][0-9a-fA-F]+;
NAME            : LETTER (LETTER | DIGIT)+;
fragment LETTER : [a-zA-Z\u0080-\u00FF_];
STRING          : '"' .*? '"'
                | '\'' .*? '\'';
WS              : [ \t\r]+ -> skip;
```

C.2 Sintaxis

```
parser grammar OParser;

options
{
    tokenVocab = OLexer;
}

program         : defs;

defs            : def (';' def)* ';'?

def             : STRING '=' expr
                | NAME ('#' NAME)? ('[' INT ']')? '=' expr
                ;

expr            : '(' expr ')'                                  # parExp
                | function                                      # functionExp
                | constant                                      # constantExp
                | unitaryOp expr                                # unitaryExp
                | expr '^' expr                                 # exponentialExp
                | expr op=('*' | '/' | '%') expr                # binaryOpExp
                | expr op=('+' | '-') expr                      # binaryOpExp
                | expr op=('<<' | '>>' | '>>>') expr            # shiftExp
                | expr op=('>' | '>=' | '<' | '<=') expr        # binLogOpExp
                | expr op=('==' | '!=') expr                    # binLogOpExp
                | expr '&' expr                                 # binaryOpExp
                | expr '|' expr                                 # binaryOpExp
                | expr op=('&&' | '&&&') expr                   # logAndExp
                | expr op=('||' | '|||') expr                   # logOrExp
                | expr '?' expr ':' expr                        # condExp
                ;

function        : 'abs' '(' expr ')'                            # abs
                | 'acos' '(' expr ')'                           # acos
                | 'asin' '(' expr ')'                           # asin
                | 'atan' '(' expr ')'                           # atan
                | 'atan' '(' expr ',' expr ')'                  # atan2
                | 'ceil' '(' expr ')'                           # ceil
                | 'cos' '(' expr ')'                            # cos
                | 'double' '(' expr ')'                         # castToDouble
                | 'eval' '(' expr_list expr ')'                 # eval
                | 'exp' '(' expr ')'                            # exponential
                | 'exp' '(' expr ',' expr ')'                   # exponential2
                | 'float' '(' expr ')'                          # castToFloat
                | 'floor' '(' expr ')'                          # floor
                | 'graph' '(' expr (',' expr)+ ')'              # graph
                | 'graph2' '(' expr (',' expr)+ ')'             # graph2
                | 'if' '(' expr ')'                             # if
                | 'if' '(' expr ',' expr ')'                    # if2
                | 'if' '(' expr ',' expr ',' expr ')'           # if3
                | 'if' '(' expr ',' expr ',' expr ',' expr ')'  # if4
                | 'int' '(' expr ')'                            # castToInt
                | 'isnull' '(' expr ')'                         # isNull
                | 'log' '(' expr ')'                            # log
                | 'log' '(' expr ',' expr ')'                   # log2
                | 'max' '(' expr (',' expr)+ ')'                # max
                | 'median' '(' expr (',' expr)+ ')'             # median
                | 'min' '(' expr (',' expr)+ ')'                # min
                | 'mode' '(' expr (',' expr)+ ')'               # mathMode
                | 'nmax' '(' expr (',' expr)+ ')'               # nmax
```

```
                | 'nmedian' '(' expr (',' expr)+ ')'          # nmedian
                | 'nmin' '(' expr (',' expr)+ ')'            # nmin
                | 'nmode' '(' expr (',' expr)+ ')'           # nmode
                | 'not' '(' expr ')'                         # not
                | 'pow' '(' expr ',' expr ')'                # exponential2
                | 'rand' '(' expr ',' expr ')'               # rand
                | 'round' '(' expr ')'                       # round
                | 'round' '(' expr ',' expr ')'              # round2
                | 'round' '(' expr ',' expr ',' expr ')'     # round3
                | 'sin' '(' expr ')'                         # sin
                | 'sqrt' '(' expr ')'                        # sqrt
                | 'tan' '(' expr ')'                         # tan
                | 'xor' '(' expr ',' expr ')'                 # xor
                | 'null' '(' ')'                             # null
                | 'sql' '(' NAME ',' NAME ')'                # sql
                ;

expr_list       : expr (',' expr)*;

constant        : INT                                        # intConst
                | FLOAT                                      # floatConst
                | DOUBLE                                     # doubleConst
                | NAME                                       # identifier
                | STRING                                     # stringConst
                | NAME '[' expr ',' expr ']'                 # neighborMod
                ;

unitaryOp       : '-'                                        # negOp
                | '~'                                        # compOp
                | '!'                                        # notOp
                ;
```

Bibliografía

[ABC*18] Apollonio F. I., Basilissi V., Callieri M., Dellepiane M., Gaiani M., Ponchio F., Rizzo F., Rubino A. R., Scopigno R., Sobra' G.: A 3d-centered information system for the documentation of a complex restoration intervention. Journal of Cultural Heritage 29 (2018), 89 – 99. doi:https://doi.org/10.1016/j.culher.2017.07.010.

[Bau72] Baumgart B. G.: Winged Edge Polyhedron Representation. Tech. rep., Stanford, CA, USA, 1972.

[BCCA*15] Benito-Calvo A., Carvalho S., Arroyo A., Matsuzawa T., de la Torre I.: First gis analysis of modern stone tools used by wild chimpanzees (pan troglodytes verus) in bossou, guinea, west africa. Plos One 10, 3 (2015), e0121613.

[BHK14] Bénard P., Hertzmann A., Kass M.: Computing smooth surface contours with accurate topology. ACM Trans. Graph. 33, 2 (Apr. 2014). URL: https://doi.org/10.1145/2558307, doi:10.1145/2558307.

[BKM09] Bangor A., Kortum P., Miller J.: Determining what individual sus scores mean: Adding an adjective rating scale. Journal of usability studies 4, 3 (2009), 114–123.

[BSBK02] Botsch M., Steinberg S., Bischoff S., Kobbelt L.: Openmesh-a generic and efficient polygon mesh data structure.

[Cat74] Catmull E. E.: A subdivision algorithm for computer display of curved surfaces. PhD thesis, University of Utah, 1974.

[CHI23] Repositorio de chisel 3.0. https://github.com/Granada-Graphics-Group/Chisel3, 2023. Accessed: 2023-06-17.

[CLDT16] Campanaro D. M., Landeschi G., Dell'Unto N., Touati A.-M. L.: 3d gis for cultural heritage restoration: A 'white box' workflow. Journal of Cultural Heritage 18 (2016), 321 – 332. doi:http://dx.doi.org/10.1016/j.culher.2015.09.006.

[CMS98] Cignoni P., Montani C., Scopigno R.: A comparison of mesh simplification algorithms. Computers & Graphics 22, 1 (1998), 37–54. URL: https://www.sciencedirect.com/science/article/pii/S0097849397000824, doi:https://doi.org/10.1016/S0097-8493(97)00082-4.

[CP05] Cazals F., Pouget M.: Topology driven algorithms for ridge extraction on meshes. Research Report RR-5526, INRIA, 2005. URL: https://hal.inria.fr/inria-00070481.

[Dal23] Página oficial de dall·e 2. https://openai.com/dall-e-2, 2023. Accessed: 2023-06-17.

[DDM*06] Durand A., Drap P., Meyer E., Grussenmeyer P., Perrin J.: Intra-site level cultural heritage documentation: Combination of survey, modeling and imagery data in a web information system. CoRR abs/cs/0611036 (2006).

[DGGG21] Diulio M. d. l. P., Gardey J. C., Gomez A. F., Garrido A.: Usability of data-oriented user interfaces for cultural heritage: A systematic mapping study. Journal of Information Science (2021), 01655515211001787. URL: https://doi.org/10.1177/01655515211001787, arXiv:https://doi.org/10.1177/01655515211001787, doi:10.1177/01655515211001787.

[Dog12] Doggett M.: Texture caches. IEEE Micro 32, 3 (May 2012), 136–141. doi:10.1109/MM.2012.44.

[EFMS14] Everitt C., Foley T., McDonald J., Sellers G.: Approaching zero driver overhead in opengl. In 2014 Game Developers Conference (San Francisco, CA, USA, 2014).

[Eve01] Everitt C.: Projective texture mapping. White paper, NVidia Corporation 4 (2001).

[For14] Forte M.: 3d archaeology: new perspectives and challenges—the example of çatalhöyük. Journal of Eastern Mediterranean Archaeology & Heritage Studies 2, 1 (2014), 1–29.

[GDPCM05] Giunta G., Di Paola E., Castiglione B. M. V., Menci L.: Integrated 3d-database for diagnostics and documentation of milan's cathedral façade. In CIPA 2005 XX International Symposium (Torino, Italy) (2005), vol. 3.

[GGSC98] Gooch A., Gooch B., Shirley P., Cohen E.: A non-photorealistic lighting model for automatic technical illustration. In Proceedings of the 25th annual conference on Computer graphics and interactive techniques (1998), pp. 447–452.

[GS83] Guibas L. J., Stolfi J.: Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. In Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (New York, NY, USA, 1983), STOC '83, Association for Computing Machinery, p. 221–234. URL: https://doi.org/10.1145/800061.808751, doi:10.1145/800061.808751.

[GTDS10] Grabli S., Turquin E., Durand F., Sillion F. X.: Programmable Rendering of Line Drawing from 3D Scenes. ACM Transactions on Graphics 29, 2 (Apr. 2010), 18:1–18:20. doi:http://doi.acm.org/10.1145/1731047.1731056.

[HG97] Hakura Z. S., Gupta A.: The design and analysis of a cache architecture for texture mapping. SIGARCH Comput. Archit. News 25, 2 (May 1997), 108–120. doi:10.1145/384286.264152.

[HIR*03] Halper N., Isenberg T., Ritter F., Freudenberg B., Meruvia Pastor O. E., Schlechtweg S., Strothotte T.: OpenNPAR: A System for Developing, Programming, and Designing Non-Photorealistic Animation and Rendering. In Proc. Pacific Graphics (Los Alamitos, 2003), IEEE Computer Society, pp. 424–428. doi:http://dx.doi.org/10.1109/PCCGA.2003.1238288.

[Hu18] Hu X.: Usability evaluation of e-dunhuang cultural heritage digital library. Data and information management 2, 2 (2018), 57–69.

[HV09] Hermosilla P., Vázquez P.: Single pass gpu stylized edges. In IV Iberoamerican Symposium in Computer Graphics - SIACG (2009) (2009), pp. 1–8.

[IB06] Isenberg T., Brennecke A.: G-Strokes: A Concept for Simplifying Line Stylization. Computers & Graphics 30, 5 (Oct. 2006), 754–766. doi:http://dx.doi.org/10.1016/j.cag.2006.07.006.

[INC*06] Isenberg T., Neumann P., Carpendale S., Sousa M. C., Jorge J. A.: Non-photorealistic rendering in context: an observational study. In Proceedings of the 4th international symposium on Non-photorealistic animation and rendering (2006), pp. 115–126.

[IPS03] Ioannidis C., Potsiou C., Soile S.: An integrated spatial information system for the development of the archaeological site of mycenae. International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences 34, 5 (2003).

[JTMW96] Jordan P. W., Thomas B., McClelland I. L., Weerdmeester B.: Usability evaluation in industry - SUS: A "quick and dirty" usability scale. CRC Press, 1996.

[Kle00] Klette R.: Cell complexes through time. In Vision Geometry IX (2000), Latecki L. J., Mount D. M., Wu A. Y., (Eds.), vol. 4117, International Society for Optics and Photonics, SPIE, pp. 134–145. URL: https://doi.org/10.1117/12.404813, doi:10.1117/12.404813.

[KMM*02] Kalnins R. D., Markosian L., Meier B. J., Kowalski M. A., Lee J. C., Davidson P. L., Webb M., Hughes J. F., Finkelstein A.: WYSIWYG NPR: Drawing Strokes Directly on 3D Models. ACM Transactions on Graphics 21, 3 (July 2002), 755–762. doi:http://doi.acm.org/10.1145/566654.566648.

[LAM12] López L., Arroyo G., Martín D.: Computer tool for automatically generated 3d illustration in real time from archaeological scanned pieces. Virtual Archaeology Review 3, 6 (Nov. 2012), 73–77. URL: https://polipapers.upv.es/index.php/var/article/view/4447, doi:10.4995/var.2012.4447.

[LLLV05] Lage M., Lewiner T., Lopes H., Velho L.: Chf: a scalable topological data structure for tetrahedral meshes. In XVIII Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'05) (2005), IEEE, pp. 349–356.

[LS95] Lansdown J., Schofield S.: Expressive Rendering: A Review of Nonphotorealistic Techniques. IEEE Computer Graphics and Applications 15, 3 (May 1995), 29–37. doi:http://doi.ieeecomputersociety.org/10.1109/38.376610.

[LST91] Larson M., Shapiro M., Tweddale S.: Performing map calculations on grass data: r. mapcalc program tutorial.

[LTA*20] López L., Torres J. C., Arroyo G., Cano P., Martín D.: An efficient gpu approach for designing 3d cultural heritage information systems. Journal of Cultural Heritage 41 (2020), 142–151. URL: https://www.sciencedirect.com/science/article/pii/S1296207418307167, doi:https://doi.org/10.1016/j.culher.2019.05.003.

[MADA17] Muqtadiroh F. A., Astuti H. M., Darmaningrat E. W. T., Aprilian F. R.: Usability evaluation to enhance software quality of cultural conservation system based on nielsen model (wikibudaya). Procedia Computer Science 124 (2017), 513–521.

[Man88] Mantyla M.: An Introduction to Solid Modeling. Computer Science Press, 1988.

[Mea82] Meagher D.: Geometric modeling using octree encoding. Computer Graphics and Image Processing 19, 2 (1982), 129 – 147. URL: http://www.sciencedirect.com/science/article/pii/0146664X82901046, doi:https://doi.org/10.1016/0146-664X(82)90104-6.

[MGP*07] Meyer É., Grussenmeyer P., Perrin J.-P., Durand A., Drap P.: A web information system for the management and the dissemination of cultural heritage data. Journal of Cultural Heritage 8, 4 (2007), 396–411.

[MRVMRÁ*16] Mateos Redondo F. J., Valdeón Menéndez L., Rojo Álvarez A., Armisén Fernández A., García Fernández-Jardón B.: Plataforma virtual para el diseño, planificación, control, intervención y mantenimiento en el ámbito de la conservación del patrimonio histórico "petrobim". In Construction Pathology, Rehabilitation Technology and Heritage Management REHABEND (2016).

[Nag03] Naglič K. K.: Cultural heritage information system in the republic of slovenia. In ARIADNE 5 Workshop on Documentation, Interpretation, Presentation and Publication of Cultural Heritage (Prague, 2003).

[NBLM12] Neteler M., Bowman M., Landa M., Metz M.: GRASS GIS: a multipurpose Open Source GIS. Environmental Modelling & Software 31 (2012), 124–130. doi:10.1016/j.envsoft.2011.11.014.

[NFB15] Nigra B. T., Faull K. F., Barnard H.: Analytical chemistry in archaeological research. Analytical chemistry 87, 1 (2015), 3–18.

[OBS23] Página oficial de obs studio. https://obsproject.com/, 2023. Accessed: 2023-06-17.

[Par13] Parr T.: The definitive ANTLR 4 reference. Pragmatic Bookshelf, 2013.

[PHF14] Parr T., Harwell S., Fisher K.: Adaptive ll(*) parsing: The power of dynamic analysis. SIGPLAN Not. 49, 10 (oct 2014), 579–598. URL: https://doi.org/10.1145/2714064.2660202, doi:10.1145/2714064.2660202.

[PPB14] Parkinson J. A., Plummer T. W., Bose R.: A gis-based approach to documenting large canid damage to bones. Palaeogeography, Palaeoclimatology, Palaeoecology 409 (2014), 57 – 71. doi:http://dx.doi.org/10.1016/j.palaeo.2014.04.019.

[PQ95] Parr T. J., Quong R. W.: Antlr: A predicated-ll (k) parser generator. Software: Practice and Experience 25, 7 (1995), 789–810.

[PWR*12] Phiri L., Williams K., Robinson M., Hammar S., Suleman H.: Bonolo: A general digital library system for file-based collections. In The Outreach of Digital Libraries: A Globalized Resource Network (Berlin, Heidelberg, 2012), Chen H.-H., Chowdhury G., (Eds.), Springer Berlin Heidelberg, pp. 49–58.

[RC15] Ruthven I., Chowdhury G. G.: Cultural heritage information: Access and management, vol. 1. Facet Publishing, 2015.

[RD03] Roussou M., Drettakis G.: Photorealism and non-photorealism in virtual heritage representation. In First Eurographics Workshop on Graphics and Cultural Heritage (2003) (2003), Eurographics, p. 10.

[Ric99] Richens P.: The Piranesi System for Interactive Rendering. In Computers in Building: Proceedings of CAAD Futures 1999 (Boston, 1999), Kluwer Academic, pp. 381–398. URL: http://opus.bath.ac.uk/11386/.

[Ros01] Rossignac J.: 3d compression made simple: Edgebreaker with zipandwrap on a corner-table. In Proceedings International Conference on Shape Modeling and Applications (2001), pp. 278–283. doi:10.1109/SMA.2001.923399.

[Sau11] Sauro J.: A practical guide to the system usability scale: Background, benchmarks & best practices. Measuring Usability LLC, 2011.

[Sch94] Schofield S.: Non-photorealistic Rendering: A Critical Examination and Proposed System. PhD thesis, School of Art and Design, Middlesex University, United Kingdom, May 1994. URL: http://eprints.mdx.ac.uk/6723/.

[SGS05] Schlechtweg S., Germer T., Strothotte T.: RenderBots—Multi Agent Systems for Direct Image Generation. Computer Graphics Forum 24, 2 (June 2005), 137–148. doi:http://dx.doi.org/10.1111/j.1467-8659.2005.00838.x.

[SKVW*92] Segal M., Korobkin C., Van Widenfelt R., Foran J., Haeberli P.: Fast shadows and lighting effects using texture mapping. In ACM Siggraph Computer Graphics (1992), vol. 26, ACM, pp. 249–252.

[SML17] Soler F., Melero F. J., Luzón M. V.: A complete 3d information system for cultural heritage documentation. Journal of Cultural Heritage 23 (2017), 49 – 57. doi:http://dx.doi.org/10.1016/j.culher.2016.09.008.

[SSD*11] Serna S. P., Scopigno R., Doerr M., Theodoridou M., Georgis C., Ponchio F., Stork A.: 3d-centered media linking and semantic enrichment through integrated searching, browsing, viewing and annotating. In The 12th International Symposium on Virtual Reality, Archaeology and Cultural Heritage. VAST (2011), pp. 89–96.

[ST90] Saito T., Takahashi T.: Comprehensible Rendering of 3-D Shapes. ACM SIGGRAPH Computer Graphics 24, 3 (Aug. 1990), 197–206. doi:http://doi.acm.org/10.1145/97880.97901.

[STLL13] Soler F., Torres J. C., León A. J., Luzón M. V.: Design of cultural heritage information systems based on information layers. Journal on Computing and Cultural Heritage 6, 4 (Dec. 2013), 15:1–15:17. doi:10.1145/2532630.2532631.

[SW92] Shapiro M., Westervelt J.: R.MAPCALC: An Algebra for GIS and Image Processing. Tech. rep., Army Corps of Engineers, Construction Engineering Research Laboratories, 1992.

[SW94] Shapiro M., Westervelt J.: r. mapcalc: An algebra for GIS and image processing. Tech. rep., Construction Engineering Research Lab (ARMY) Champaign IL, 1994.

[SWG*89] Shapiro M., Westervelt J., Gerdes D., Higgins M., Larson M.: GRASS 3.0 programmer's manual. Tech. rep., CONSTRUCTION ENGINEERING RESEARCH LAB (ARMY) CHAMPAIGN IL, 1989.

[TCM*10] Torres J., Cano P., Melero J., España M., Moreno J.: Aplicaciones de la digitalización 3d del patrimonio. Virtual Archaeology Review 1, 1 (2010), 51–54. doi:10.4995/var.2010.4768.

[The08] The Khronos Group Inc: Texture array, 2008. https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_texture_array.txt.

[The13a] The Khronos Group Inc: Bindless texture, 2013. https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_bindless_texture.txt.

[The13b] The Khronos Group Inc: Sparse texture, 2013. https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_sparse_texture.txt.

[TLR*13] Torres J. C., López L., Romo C., Arroyo G., Cano P., Lamolda F., Villafranca M.: Using a cultural heritage information system for the documentation of the restoration process. In Digital Heritage International Congress (DigitalHeritage), 2013 (2013), vol. 2, IEEE, pp. 249–256.

[TVD07] Tierny J., Vandeborre J., Daoudi M.: Topology driven 3d mesh hierarchical segmentation. In IEEE International Conference on Shape Modeling and Applications 2007 (SMI '07) (2007), pp. 215–220. doi:10.1109/SMI.2007.38.

[WPSAM10] Wong H., Papadopoulou M., Sadooghi-Alvandi M., Moshovos A.: Demystifying gpu microarchitecture through microbenchmarking. In 2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS) (March 2010), pp. 235–246. doi:10.1109/ISPASS.2010.5452013.