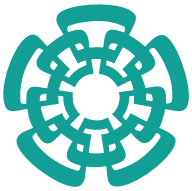


Centro de Investigación y de Estudios
Avanzados del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Marco de desarrollo para la edición colaborativa de objetos en 3D mediante realidad aumentada

Tesis que presenta

Andrés Cortés Dávalos

para obtener el Grado de

Doctor en Ciencias

en Computación

Directora de la Tesis

Dra. Sonia Guadalupe Mendoza Chapa

Ciudad de México **Febrero, 2017**

Resumen

En la actualidad, el uso de los modelos 3D es muy amplio y sigue aumentando en la vida cotidiana. Podemos encontrarlos prácticamente en cualquier aplicación concebible, desde entretenimiento y publicidad hasta visualización científica. En la mayoría de los casos, el modelado tridimensional se realiza por manipulación directa de los componentes del modelo virtual, lo cual puede llegar a ser un proceso lento y tedioso, debido principalmente a la naturaleza bidimensional de la interacción entre el usuario y el sistema, i. e., mediante teclado y ratón. Además, algunos modelos 3D suelen ser demasiado complejos para ser creados por una sola persona, de manera que resulta necesario que varios colaboradores participen en la tarea. Sin embargo, no abundan las herramientas de modelado tridimensional que soporten una colaboración ágil, aún incluso entre personas que se encuentran presentes en el mismo lugar. La propuesta descrita en esta tesis consiste en desarrollar un marco de desarrollo que facilite la construcción de aplicaciones de software para dispositivos móviles orientadas a la autoría y bosquejo de modelos virtuales tridimensionales compartidos, dentro de un entorno colaborativo de tipo cara a cara. Hemos llamado a nuestro marco de desarrollo ShAREd (*Shared Augmented Reality Editing*). Por otra parte, se requiere mucho entrenamiento para que una persona llegue a dominar cualquier herramienta de modelado tridimensional existente en el mercado. A este respecto, proponemos un nuevo enfoque en el área de modelado 3D, que consiste en utilizar Realidad Aumentada para dar a los usuarios una mejor percepción de la forma y estructura tridimensional del modelo, así como reducir el tiempo de aprendizaje necesario para poder manipularlo. Finalmente, se desarrolló un esquema de colaboración para compartir modelos 3D, en un entorno cara a cara, con el fin de ofrecer una colaboración en tiempo real. La funcionalidad proporcionada por ShAREd se ha validado implementando tres aplicaciones colaborativas móviles: un editor de exhibiciones académicas, un editor de terrenos y un editor de mallas poligonales, cuya usabilidad se ha evaluado con grupos de usuarios novatos en el proceso de modelado 3D.

Palabras clave: Modelado cooperativo, colaboración cara a cara en tiempo real, modelos virtuales compartidos en tres dimensiones.

Abstract

Currently, the use of virtual 3D models is widespread, and keeps growing everyday. We can found them in any conceivable application, from advertising and entertainment to engineering, education and scientific visualization. In most cases, the creation of 3D models (3D modeling) is carried out by direct manipulation of the virtual model components, which can be a tedious and slow process, mainly due to the bi-dimentional nature of the interaction between the user and the system, i.e., it is performed by means of keyboard and mouse. Moreover, some 3D models are usually too complex to be created by a single person, so it is needed a group of collaborators working together on the modeling task. However, there are few 3D modeling tools that support an agile collaboration, even among collocated people. The proposal described in this dissertation consists in developing of a framework to facilite the implementation of software applications for authoring and sketching of 3D shared virtual models, using mobile devices in face-to-face collaborative settings. Our framework is called ShAREd (*Shared Augmented Reality Editing*). Also, most 3D modeling applications usually require much training to be acquainted and mastered. In this regard, we propose to use Augmented Reality (AR) to give novice users a better perception of the 3D model shape and structure as well as to reduce the learning curve. Finally, we develop a collaboration scheme to share the 3D model, and to achieve real time collaboration. The functionality provided by ShAREd has been validated by implementing three collaborative mobile applications: a furniture layout editor, a terrain editor and a polygonal mesh editor, whose usability has been evaluated with groups of novel users to the field of 3D modeling.

Keywords: collaborative modeling, face-to-face real time settings, 3D shared virtual models.

Agradecimientos

Al Centro de Investigación y Estudios Avanzados del I.P.N.
por la oportunidad de llevar a cabo este proyecto.

Al Consejo Nacional de Ciencia y Tecnología
por los recursos otorgados para el desarrollo de esta tesis.

A los profesores del Departamento de Computación
por ser de una variedad ejemplar,
una verdadera gama cromática de luces y sombras.

A mi directora de tesis, la Dra. Sonia Mendoza Chapa
por creer en mí y tomar el riesgo
de emprender este viaje a una *terra ignota* virtual.

To her ladyship Sandy
por el apoyo incondicional en los altibajos.

A mi familia
por haber estado siempre ahí.

A mis gatos
por la inmejorable compañia.

A mis amigos, los cercanos y los lejanos, los viejos y los nuevos
por los buenos momentos.

En especial a mis amigos de Kuruchu Soft
sin ustedes no habría salido avante con los dichosos papers.

A Rocksteady Studios Ltd.
por toda la saga de *Arkham*, por las horas de diversión e inspiración.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En años recientes, el uso de dispositivos móviles, tales como teléfonos inteligentes (*smartphones*) y tabletas, se ha incrementado enormemente. De hecho, existe una gran cantidad de aplicaciones que facilita la vida diaria de los usuarios, ayudándolos a mejorar su productividad así como a explorar sus capacidades creativas y artísticas.

El presente trabajo consiste en diseñar e implementar un marco de desarrollo (*framework*) que facilite al desarrollador la construcción de aplicaciones móviles para la creación de bosquejos de contenidos digitales tridimensionales (3D). Aunque existen diversas aplicaciones de autoría y diseño de contenidos 3D, muy pocas son móviles o hacen uso de tecnologías emergentes, tales como la Realidad Virtual (*Virtual Reality*, VR) o la Realidad Aumentada (*Augmented Reality*, AR) y prácticamente ninguna de ellas es colaborativa. Además, en el caso de usuarios que carecen de entrenamiento previo en diseño 3D, la mayoría de estas aplicaciones de modelado son difíciles de aprender y manejar. De hecho, se requieren años de experiencia para utilizarlas de manera eficiente.

Asimismo, nos proponemos que los elementos interactivos del marco de desarrollo propuesto sean amigables con el usuario de acuerdo a los principios de Experiencia del Usuario (*User eXperience*) aplicados a dispositivos móviles. Para desarrollar un trabajo como este, es necesario echar mano de diversas disciplinas, tecnologías y áreas de estudio, como se detalla a continuación.

1.1. Contexto de investigación

En este trabajo utilizamos herramientas y conceptos de diversas áreas, a saber: Trabajo Colaborativo Asistido por Computadora (*Computer Supported Collaborative Work*), Gráficos por Computadora (*Computer Graphics*) y Visión por Computadora (*Computer Vision*), junto con algunas consideraciones adicionales del área de Interacción Hombre-Máquina (*Human-Computer Interaction*) destinadas a mejorar la experiencia del usuario final de las aplicaciones que se desea crear con el marco de desarrollo propuesto.

Como se puede apreciar en la Figura 1.1, el tipo de aplicaciones creadas con

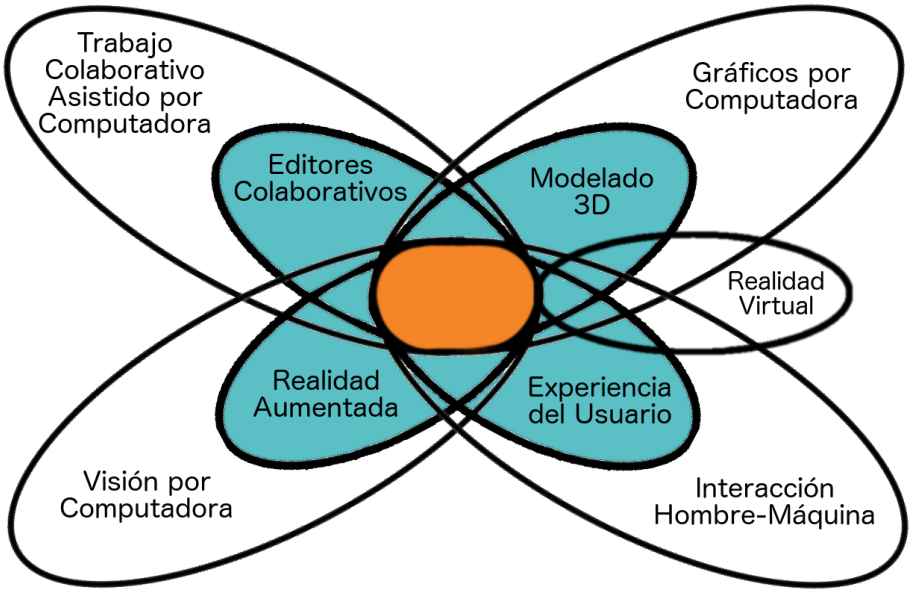


Figura 1.1: Áreas de investigación relevantes para el presente trabajo.

este marco de desarrollo corresponde a la categoría de Editores Colaborativos (*Collaborative Editors*) que se estudian en el área de Trabajo Colaborativo Asistido por Computadora. Por el tipo de contenido que generan, se trata de aplicaciones de Modelado 3D que se encuentran dentro del área de Gráficos por Computadora. Durante el proceso de edición de los contenidos 3D, aprovechamos las ventajas de la Realidad Aumentada, que es una tecnología emergente, de creciente aceptación entre el público, la cual se construye a partir de algoritmos eficientes de reconocimiento de patrones, dentro del área de Visión por Computadora.

Como parte del marco de desarrollo hemos diseñado diversos elementos de interfaz de usuario de acuerdo a los principios que dicta el área de Interacción Hombre-Máquina, junto con diversos patrones de diseño que enfatizan la Experiencia del Usuario.

1.2. Motivación

Considere el siguiente escenario: un grupo de usuarios casuales se encuentra intercambiando ideas sobre la forma de algún objeto 3D, el cual podrá ser utilizado en una animación, un comercial, una presentación, un salón de clases, etc. Estos usuarios no tienen una computadora a la mano, pero cada uno trae consigo un dispositivo móvil que ejecuta iOS o Android. Así que, en lugar de tratar de expresar sus ideas solamente hablando, gesticulando o dibujando en un pizarrón o una servilleta, dando lugar a posibles malentendidos, estos usuarios podrían utilizar una aplicación móvil para bosquejar de manera colaborativa un objeto 3D, haciendo uso de tecnologías emergentes, como la realidad aumentada, pero sin necesidad de adquirir piezas de equipo adicional, tales como lentes, controles o sensores especiales. Más aún, es importante

considerar que los integrantes del grupo suelen tener dispositivos diversos, así que es necesario proporcionarles herramientas colaborativas que se comuniquen entre sí, de manera transparente e independientemente de la plataforma del dispositivo.

En la mayoría de los dispositivos de cómputo actuales, el despliegue se realiza a través de una pantalla bidimensional (2D). En consecuencia, la retroalimentación visual dada al usuario se basa en proyecciones 2D del objeto 3D, las cuales son tomadas desde el punto de vista de una o varias cámaras virtuales. Este paradigma genera dos fuentes de confusión en los nuevos usuarios: es difícil interpretar la forma y estructura del objeto en 3D, a partir únicamente de las vistas; y es difícil mover y apuntar, de manera precisa, las cámaras virtuales hacia la ubicación deseada en el espacio virtual.

Al usuario se le presenta la escena a través de diversas cámaras virtuales. Usualmente, hay una cámara de perspectiva que el usuario puede mover y orientar para obtener el punto de vista deseado. También hay otras cámaras implícitas utilizadas para ver la escena desde direcciones ortogonales fijas, tales como la vista desde arriba (dirección Z), desde el frente (dirección Y) y de lado (dirección X). En algunos casos, el usuario no está completamente consciente de las diferentes cámaras virtuales. La experiencia de primera mano con principiantes nos muestra que ellos tienen dificultad para cambiar de cámara virtual y para orientar la de perspectiva hacia la ubicación deseada. Ellos pueden perderse con facilidad en el espacio virtual tridimensional.

Aunque existen varias aplicaciones que dan soporte a la autoría de contenidos 3D, muy pocas son móviles o colaborativas y en raras ocasiones se hace uso de la realidad aumentada. De manera que, en el caso de usuarios que carecen de entrenamiento previo en diseño 3D, la mayoría de estas aplicaciones de modelado tienen las siguientes desventajas inherentes:

- Cuando se utiliza por primera vez una aplicación de modelado 3D, resulta difícil interactuar con su interfaz de usuario, pues las herramientas y opciones disponibles en ellas son numerosas y poco intuitivas.

- Ocurre con frecuencia que los usuarios novatos pierden la ubicación de su cámara dentro del espacio virtual y también les resulta difícil colocarla en la posición y orientación deseada.

- El proceso de autoría generalmente es monousuario, lo cual no fomenta un flujo de trabajo grupal, volviéndose un obstáculo para la colaboración.

Dichas desventajas no están relacionadas con la velocidad del hardware o el desempeño algorítmico del software, sino que son consecuencia del diseño de interacción de los sistemas actuales de modelado. Es necesario cambiar el paradigma de interacción tradicional para adaptarlo a las necesidades de un soporte de interacción tridimensional más natural.

Cabe mencionar que algunos expertos en Gráficos por Computadora y Modelado 3D, quienes a través de los años ya han llegado a dominar este paradigma de interpretación de vistas proyectivas y el uso de múltiples cámaras virtuales a la vez, no consideran necesario un cambio en el paradigma de interacción. Sin embargo, en

este proyecto nos interesan los usuarios que son nuevos en modelado, quienes aún no han llegado a dominar este arte y necesitarían invertir mucho tiempo y esfuerzo para llegar a utilizar las herramientas existentes. Pretendemos que el enfoque utilizado en el marco de desarrollo, permita construir aplicaciones de modelado que le faciliten el proceso a dichos usuarios finales novatos y a la vez apoyen el trabajo concurrente cuando se encuentran colocalizados.

Actualmente, la mayoría de los usuarios están acostumbrados a apuntar una cámara digital hacia toda clase de objetos (y sujetos) de interés en el mundo real. Por lo tanto, cuando se usa realidad aumentada para visualizar objetos 3D, el usuario ya no necesita manipular múltiples cámaras virtuales para tener diferentes perspectivas. Simplemente necesita apuntar la cámara real del dispositivo de la manera usual dentro del espacio físico. Como el objeto virtual es percibido dentro del entorno real desde la perspectiva deseada, el usuario obtiene un mayor entendimiento de su forma y estructura geométrica en 3D. De esta manera, la tecnología de realidad aumentada ayuda a resolver el segundo problema, mejorando la percepción de la ubicación, forma y escala del objeto virtual, sin necesidad de contar con cámaras virtuales adicionales, ya que solo se requiere una cámara real.

En la práctica, hemos observado que durante una sesión de edición colaborativa de un único objeto 3D, es frecuente que solo un usuario trabaje en la tarea de modelado, mientras el resto del grupo espera u observa. En algún punto, los otros colaboradores pueden sugerir cambios al colaborador que tiene el control de la aplicación. Además, dichos colaboradores ocasionalmente toman turnos en el control de la aplicación para efectuar sus respectivas porciones del trabajo común. Este comportamiento es causado por la propia aplicación, ya que no permite que los participantes trabajen al mismo tiempo. En algunas etapas del modelado colaborativo, es más productivo permitir que los participantes trabajen simultáneamente, mientras que en otras es preferible tomar turnos. El grupo de usuarios es el responsable de decidir si toman turnos o trabajan simultáneamente, así que consideramos que este tipo de restricciones no deben ser parte del diseño de ningún sistema.

1.3. Planteamiento del problema

En los sistemas de modelado 3D actuales, la manipulación del modelo en el espacio tridimensional se efectúa mediante esquemas de interacción bidimensional en contextos monousuario. Por lo tanto, es necesario diseñar esquemas de interacción tridimensional más adecuados y que sean extensibles a un flujo de trabajo colaborativo.

Para resolver este problema, en este trabajo se propone el uso de realidad aumentada para mejorar la percepción tridimensional de la escena virtual, así como el diseño y la implementación de un marco de desarrollo que facilite la construcción de editores colaborativos para objetos 3D. En la actualidad, no existen editores 3D que sean colaborativos y que gocen de las ventajas de la realidad aumentada. Tampoco se tiene noticia de algún marco de desarrollo especializado en crear este tipo de editores.

Una de las posibles causas de la ausencia de sistemas de este tipo, puede ser el desconocimiento y falta de colaboración entre las áreas de Trabajo Colaborativo Asistido por Computadora, Gráficos por Computadora y Visión por Computadora. Este hecho puede dar la impresión de que las soluciones de una u otra son difíciles de aplicar o poco relevantes. Sin embargo, como muestra el presente trabajo, este no es el caso sino que existe una amplia oportunidad de desarrollo, a la vez que se ofrece a los usuarios herramientas más usables y adecuadas para realizar tareas de autoría y bosquejado, que históricamente han recibido poca atención.

La creación de aplicaciones desde cero puede llegar a ser dificil [27]. Para facilitar la tarea a los desarrolladores, existen herramientas tales como *toolkits* y marcos de desarrollo, así como patrones de diseño. Por medio de dichas herramientas, se pueden hacer aplicaciones más fáciles de mantener, pues se limita su dependencia a la plataforma y se permite diseñarlas mediante una serie de capas con funcionalidades específicas. Se reduce también el acoplamiento entre clases ayudando a su extensibilidad. Los sistemas orientados a objetos alcanzan el mayor nivel de reutilización de código mediante el uso de los marcos de desarrollo. Las aplicaciones grandes suelen estar compuestas de varias capas de marcos de desarrollo, *toolkits* y otros componentes, que cooperan entre sí, de manera que la mayoría de las decisiones de diseño e implementación se ven influenciadas por los componentes elegidos.

Un marco de desarrollo predefine los parámetros de diseño de las aplicaciones, de manera que el desarrollador no pierde tiempo creando la funcionalidad subyacente, sino que puede concentrarse en la funcionalidad específica de la aplicación. Notemos aquí que, en el diseño propuesto para editores 3D colaborativos, la funcionalidad relacionada con el proceso de modelado 3D depende en gran medida del tipo de contenido que se desea editar. De modo que usando este marco, el desarrollador puede olvidarse de lo relacionado con la realidad aumentada y la propagación de mensajes a los colaboradores y puede enfocarse en diseñar el tipo de modelo 3D y las operaciones que se le van a aplicar.

Mediante el marco de desarrollo propuesto en este trabajo, se pretende capturar las decisiones de diseño que son comunes al dominio de editores 3D colaborativos. Consideramos también que la modalidad de visualización del contenido virtual, mientras se le está manipulando, es una parte integral del diseño del marco de desarrollo. Las opciones que proporcionamos son: visualización estándar o mediante realidad aumentada. No hemos considerado en este trabajo el uso de realidad virtual, debido a que se requiere de algún dispositivo adicional, como se menciona en la Sección 2.3.3. Recordemos que nos interesa la creación de aplicaciones para usuarios casuales, que cuentan únicamente con sus dispositivos móviles o un navegador, sin la necesidad de adquirir equipo adicional. A este marco de desarrollo se le ha denominado ShAREd (*Shared Augmented Reality Editing*) y con él se pretende facilitar la creación de herramientas de modelado móviles, colaborativas y fácilmente usables por grupos de colaboradores novatos.

La hipótesis de este trabajo de investigación es la siguiente:

> "El proceso de creación y edición colaborativa de mallas poligonales se puede realizar de manera más rápida y eficiente, a través de una interacción usuario-modelo más realista y diseñada para el espacio tridimensional."

Mediante pruebas con usuarios finales se verifica que al aplicar el marco de desarrollo ShAREd en la construcción de prototipos de editores colaborativos, se tiene un impacto positivo en su eficiencia, aceptación y facilidad de uso.

1.4. Objetivos

El objetivo general del presente trabajo es crear un marco de desarrollo que facilite la construcción de aplicaciones colaborativas que permitan, a grupos de colaboradores casuales y novatos, la autoría síncrona y colocalizada de contenido digital en 3D, mediante realidad aumentada, usando únicamente sus dispositivos móviles. Con ello, logramos los siguientes objetivos específicos:

(i) Definir un modelo para dar soporte a la autoría colaborativa del contenido digital tridimensional en sesiones síncronas y colocalizadas.

(ii) Dado que el proceso de edición es altamente dependiente del tipo de contenido que se desea modificar, es necesario analizar y diseñar las estrategias de comunicación, estructuras de datos y operaciones de edición para diversos tipos de contenido 3D, tales como primitivas geométricas, objetos sólidos, mapas de elevación digital, mallas poligonales, entre otros.

(iii) Identificar y definir los componentes de base, no funcionales y de interfaz de usuario del marco de desarrollo, sus relaciones de dependencia y la arquitectura correspondiente.

(iv) Diseñar los componentes fundamentales del marco de desarrollo, haciendo uso de patrones de diseño y de interfaz de usuario, con el fin de facilitar al desarrollador la tarea de creación de editores 3D colaborativos, con un diseño reutilizable, que sea útil para herramientas de modelado para diversos tipos de contenido digital.

(vi) Diseñar e implementar prototipos que permitan mostrar la reutilización de los diferentes módulos ofrecidos por el marco de desarrollo

(vii) Realizar pruebas de usabilidad de los prototipos con usuarios finales para medir la carga de trabajo percibida por dichos usuarios, así como las cualidades hedónica y pragmática de los prototipos.

1.5. Propuesta de solución

La principal contribución del presente trabajo es un nuevo enfoque basado en realidad aumentada para la tarea de crear y editar contenido digital 3D, de manera colaborativa, usando dispositivos móviles. Particularmente, en este trabajo de tesis, nos centramos en aplicaciones de autoría de objetos 3D, construidos a partir de mallas poligonales. En este proyecto nos enfocamos en crear un marco de desarrollo que facilite la creación de editores de modelos 3D compartidos, creados a partir de mallas poligonales, mediante manipulación directa por parte de un grupo de usuarios que se encuentran cara a cara.

El primer problema (de la sección 1.2) se aborda a través de un diseño minimalista de la interfaz de usuario (UI) y aplicando criterios de comodidad y funcionalidad, para decidir cuando un cierto elemento de interfaz debe mostrarse en el HUD (Heads-Up Display) o dentro de la escena 3D, junto con la representación gráfica del objeto 3D que se desea manipular, utilizando realidad aumentada. En algunos casos es conveniente tener los elementos de la interfaz, en el HUD o en la escena 3D, según resulte pertinente. Una de las ventajas de colocar la interfaz de usuario dentro de la escena 3D es que se tiene un mayor espacio virtual para colocar los diversos elementos, sin abarrotar el espacio disponible en pantalla ni obstruir el área de trabajo, independientemente del tamaño del dispositivo. Además, la interacción con la interfaz de usuario se realiza utilizando gestos táctiles intuitivos.

1.6. Organización del documento

En el Capítulo 2, se introducen algunos conceptos fundamentales que se requieren para entender este documento de tesis, incluye conceptos del área de Sistemas Colaborativos, Modelado 3D y Realidad Aumentada.

El trabajo relacionado se presenta en el Capítulo 3, se da una panorámica del estado del arte en el desarrollo de aplicaciones móviles que utilizan realidad aumentada y las líneas de investigación en realidad aumentada colaborativa, así como algunos editores de contenidos 3D.

En el Capítulo 4, se detalla la principal contribución del presente trabajo, el diseño y arquitectura del marco de desarrollo ShAREd, a partir del cual se construyen e implementan los demás elementos y los prototipos.

En el Capítulo 5, se proporcionan los detalles de implementación necesarios para utilizar el motor de juegos Unity, con el cual se construyeron dos de los prototipos. Cabe señalar que el desarrollador debe estar familiarizado con el manejo de Unity.

En el Capítulo 6, se presenta el diseño e implementación de los bloques de contrucción de los prototipos, los diversos tipos de *widgets* que creamos y que forman parte integral del marco de desarrollo ShAREd, pues proporcionan el punto de partida para el desarrollador interesado en comenzar una aplicación, usándolos como se presentan o posiblemente modificándolos según sus necesidades.

El marco de desarrollo ShAREd se valida mediante la implementación de tres prototipos, los cuales se describen en sus respectivos capítulos, donde se explica el diseño e implementación del modelo 3D y sus operaciones de edición correspondientes:

- el editor de exhibiciones académicas se describe en el Capítulo 7,
- el editor de terrenos en el Capítulo 8 y
- el editor de mallas poligonales en el Capítulo 9.

Finalmente, en el Capítulo 10, se presentan las conclusiones y el trabajo a futuro.

Capítulo 2

Conceptos fundamentales

A continuación se definen los conceptos fundamentales que constituyen la base para la creación del marco de desarrollo ShAREd.

2.1. Editores colaborativos

Un editor colaborativo es una aplicación orientada a la creación y manipulación de algún tipo de contenido digital, pero que además permite efectuarlo de manera colaborativa por un grupo de participantes, mediante la interconexión de sus dispositivos de cómputo, i.e., se trata de un sistema colaborativo (*groupware*).

Siguiendo la descripción de González y Robles [33], un sistema colaborativo soporta la habilidad de un grupo para manipular sus documentos a través de un espacio de trabajo compartido, el cual suele ser replicado en los diferentes dispositivos de trabajo de los participantes y cuyas instancias (o copias) se mantienen sincronizadas con sus contrapartes, mediante el intercambio apropiado de mensajes de control.

Si no se tiene cuidado, un sistema colaborativo puede sufrir problemas de concurrencia debido a los mensajes de control que pueden llegar en el orden equivocado, dando lugar a inconsistencias en las diferentes instancias del documento y en el modelo mental del grupo acerca de lo que está ocurriendo durante el proceso de edición.

En la literatura de sistemas distribuidos existen numerosos esquemas para el manejo de concurrencia, pero el caso de los sistemas colaborativos debe ser tratado de manera diferente, debido a que no incluye solamente la interacción entre computadoras sino también entre personas. En el tipo de sistemas colaborativos que nos concierne se deben compartir los datos de manera altamente interactiva en tiempo de ejecución.

Otras características que tiene un sistema colaborativo son: los participantes de una sesión colaborativa deben contar con canales de comunicación en tiempo real, aunque en el caso de la colaboración cara a cara entre participantes colocalizados, la comunicación visual y verbal viene dada implícitamente; el contenido o documento creado debe mostrarse a todos de manera consistente, e.g., el paradigma WYSIWIS (*what you see is what I see*) [58]; los participantes deben estar al tanto de las acciones

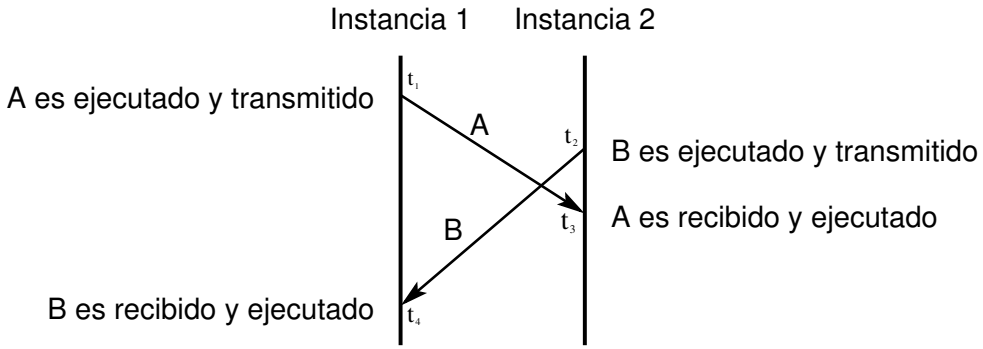


Figura 2.1: Ejemplo del cambio en el orden de ejecución de los mensajes A y B, al transmitirse entre dos instancias de la aplicación, localizadas en dispositivos distintos.

de los demás (consciencia de grupo); pueden interactuar en el espacio de trabajo de manera simultánea; pueden utilizar el espacio compartido como una herramienta cognitiva para la exploración de ideas (como cuando se discute con la ayuda de un pizarrón) o para revisión, discusión y puesta a punto de los documentos o productos creados.

2.1.1. El problema de sincronización

Un mensaje de control pasa por una serie de etapas durante su existencia: creación, ejecución local, transmisión, recepción y ejecución remota [24]. Sin un mecanismo de control de concurrencia, los mensajes son ejecutados en el dispositivo local justo después de su creación y luego son transmitidos. Esto conlleva a que puedan ser ejecutados fuera de orden en las otras instancias, generando inconsistencias [11].

En la Figura 2.1, se muestra un ejemplo en el que los mensajes de control A y B son ejecutados en orden distinto, debido al tiempo que tardan en transmitirse de un dispositivo a otro. El mensaje A se origina en la instancia 1 y el mensaje B en la instancia 2. Los mensajes no viajan de manera instantánea de un dispositivo al otro, de manera que puede darse el caso de que los mensajes se ejecuten en orden distinto en cada instancia. Por ejemplo, en la instancia 1 se ejecuta el mensaje A seguido de B, mientras que en la instancia 2 se ejecuta el mensaje B seguido de A.

Existen algoritmos para garantizar que los mensajes sean ejecutados en el mismo orden en las diferentes instancias de la aplicación (e.g., serialización) [21] o para reparar los efectos causados por la ejecución en desorden (e.g., transformación operacional) [24, 59, 61]. Una solución es crear un orden total para los mensajes [41], luego un planificador decide el orden de los eventos o la manera de detectar y reparar las inconsistencias.

Serialización

En el contexto de la serialización, a un planificador se le llama *pesimista* cuando garantiza que los eventos se reciban en el orden establecido, aunque esto signifique que las instancias deban esperar a que lleguen los eventos atrasados por la latencia

de la red. Esto puede causar la impresión de que el sistema es lento o se bloquea, i.e. no es *responsivo*. Para agilizar la respuesta de la aplicación, un planificador *optimista* permite que los eventos se reciban fuera de orden, en cuyo caso se puede proceder a detectar y corregir las inconsistencias. En el enfoque optimista, se hace la suposición de que los mensajes conflictivos rara vez llegan en el orden equivocado y que es más eficiente continuar con la ejecución y luego efectuar reparaciones poco frecuentes, que garantizar el orden en todo momento. Sin embargo, también puede ocurrir que un solo mensaje fuera de orden cause que toda una cadena de mensajes subsecuentes, que de otra manera serían válidos, queden fuera de sincronía y deban también ser reparados.

Una manera de efectuar la reparación, es regresar (*rollback*) al estado anterior al del evento conflictivo y volver a aplicar las operaciones en el orden correcto. Sin embargo, en sistemas colaborativos este comportamiento sería confuso para los colaboradores y podría interrumpir el flujo de trabajo del grupo. Otro enfoque, utilizado con éxito en simulación en paralelo, es revertir el proceso al último estado consistente y continuar desde ahí. Por supuesto, este comportamiento tampoco sería apropiado en el contexto de sistemas colaborativos, pues los usuarios perderían una parte de sus operaciones de edición. Para volver a un estado anterior se pueden efectuar operaciones de deshacer (*undo*), aplicando las operaciones inversas a las aplicadas y en el orden inverso. Desafortunadamente, no siempre es posible deshacer una operación, e.g., cuando ésta no es invertible.

Bloqueo con candados

Otro esquema de control de concurrencia es el uso de candados para bloquear regiones o elementos del documento cuando uno de los usuarios lo está modificando, previniendo así que otro tenga acceso, para evitar posibles conflictos. En una interacción típica, una instancia solicita acceso a un objeto y, si no ha sido bloqueado por alguien más, la solicitud se aprueba y el objeto queda bloqueado. Si el objeto se encuentra bloqueado, la solicitud es denegada. Cuando la instancia termina de utilizar el objeto, lo libera para que quede disponible para el resto de los colaboradores.

Esta política de candados también muestra niveles de optimismo. En una política pesimista, la ejecución de la instancia solicitante debe entrar en pausa hasta que termine el proceso de verificación para decidir si se otorga el acceso o no. Alternativamente, es posible hacer alguna otra tarea mientras se espera, pero no es posible trabajar en el objeto solicitado. Ahora bien, es posible agilizar la ejecución siendo optimista, al permitir que la instancia comience a trabajar en el objeto mientras espera por la autorización, suponiendo que en muy pocas ocasiones será denegada. Si se diera un rechazo de la solicitud, el objeto debe regresarse a su estado previo, lo cual perturba el proceso de edición del usuario en turno.

Si la aprobación tarda lo suficiente en llegar, es posible que la instancia termine la edición del objeto, lo libere y pase a trabajar en otros. Esta situación puede generar toda una cadena de inconsistencias que deberán detectarse y deshacerse, incrementando la incomodidad del usuario. Para evitarlo, se introduce una política semi

optimista, que se distingue de la completamente optimista en que permite a la instancia trabajar en el objeto mientras espera la autorización, pero no se le permite liberar el objeto hasta saber si se le concede o se le rechaza la solicitud de acceso. De esta manera, se limita la cadena de posibles inconsistencias a un solo eslabón, pero al precio de bloquear momentáneamente a la instancia en cuestión. Los enfoques optimista y semi optimista resultan adecuados para sistemas colaborativos en los que el proceso de autorización no se retrasa frecuentemente, o bien cuando se sabe que el acceso se va a denegar en pocas ocasiones.

2.1.2. Sistemas colaborativos sin control de concurrencia

Para cada aplicación en particular, se debe encontrar un compromiso entre el nivel de respuesta percibido por los usuarios y el nivel de optimismo en el control de concurrencia, incluso algunas veces puede no ser necesario dicho control. La sola idea de permitir que un sistema colaborativo carezca de un control de concurrencia puede parecer inaceptable. Sin embargo, de acuerdo a Greenberg y Marwood [33], puede resultar una estrategía razonable. Dependiendo del tipo de aplicación, las inconsistencias pueden no ser relevantes, o bien los participantes pueden ser capaces de mediar sus propias acciones y de resolver sus conflictos.

Una pequeña cantidad de inconsistencias acumuladas puede ser aceptable para los usuarios o puede no ser percibida. Por ejemplo, en aplicaciones de bosquejo (*sketching*) en las que la importancia radica en discutir ideas y crear un concepto de diseño, más que en generar un producto final. Este es el caso de las discusiones sobre un pizarrón o los bosquejos y anotaciones en un cuaderno. Una diferencia de unos pocos pixeles dentro de una imagen puede pasar desapercibida o simplemente ser aceptable para el grupo de trabajo. Un ejemplo de aplicación de bosquejo es *GroupSketch* [31], que no utiliza control de concurrencia. Por supuesto, tratándose de herramientas de autoría, en las que el documento final es importante, se debe elegir algún mecanismo de control de concurrencia adecuado para el tipo de contenido.

Protocolos sociales espontáneos

Las personas siguen protocolos sociales de manera natural para mediar y regular sus interacciones, como en el caso de la toma de turnos para hablar durante una conversación. En una herramienta de edición colaborativa, podemos ver en lo que los otros participantes están trabajando y usualmente evitamos realizar acciones que interfieran con ellos. A manera de ilustración, en un pizarrón compartido resultaría inapropiado pintar por encima de lo que otros están dibujando o entrar en una pelea o competencia por tomar el mismo plumón que otro desea tomar. Este comportamiento está reportado en la literatura por Stefik et al. [58], quienes hacen referencia a *CoLab*, un prototipo de sala de juntas que soporta el tipo de colaboración cara a cara.

Claro que hay ocasiones en las que los conflictos pueden ocurrir, tales como interferencias accidentales debidas a un descuido o a una falta de atención en las acciones de los otros, por efectos colaterales de una acción hacia las de los demás, o bien in-

terferencias intencionales, posiblemente causadas por diferencias de opinión o luchas de poder. Sin embargo, los conflictos de concurrencia pueden minimizarse cuando las personas se encuentran en la disposición de cooperar y regulan sus acciones por sí mismas. Además, en los pocos casos en que ocurren algunas inconsistencias, pueden no ser problemáticas en la práctica, o bien cuando los conflictos son notorios, los participantes pueden ser capaces de reparar sus efectos negativos e inclusive considerar estas acciones como parte del diálogo natural en la colaboración. En estos casos, el papel del sistema es proporcionar la suficiente retroalimentación y usabilidad de los objetos compartidos, para propiciar en vez de obstaculizar las habilidades sociales de las personas que los utilizan.

2.2. Modelado 3D

En muchas aplicaciones prácticas, es frecuente la necesidad de contar con modelos digitales tridimensionales (modelos 3D) de los objetos reales de interés, los cuales pueden ser construídos a partir de figuras geométricas. En este contexto, por modelo digital nos referimos a alguna estructura de datos que represente al objeto en cuestión y que pueda ser visualizada mediante algún tipo de representación gráfica. Dicha estructura contiene internamente algunas de las múltiples características y comportamientos del objeto real, sólo aquellas que sean relevantes para la aplicación. En particular, nos interesan las aplicaciones de creación de modelos 3D que son representados únicamente por su forma exterior visible y en los que no es necesario conocer su estructura o composición interna. La autoría de dichos modelos corresponde al área de Modelado 3D, en la que se estudian los diversos enfoques y técnicas que existen para crearlos.

Un modelo virtual se puede crear dentro de alguna de las siguientes dos situaciones: se desea crear una réplica virtual de algún objeto real ya existente, o bien se quiere definir la forma de un objeto que no existe en el mundo real, ya sea porque es producto de la imaginación, porque es una entidad abstracta o porque se pretende fabricarlo posteriormente.

Cabe mencionar que, en el área de Visión por Computadora, existen líneas de investigación acerca del modelado 3D automático, el cual se realiza algorítmicamente a partir de mediciones, fotografías o videos de objetos reales [5, 19]. Desafortunadamente, este modelado automático no es aplicable cuando se trata de objetos que no existen. Tal es el caso de ciertas aplicaciones de entretenimiento, en las que se modelan mundos enteros que son producto de la imaginación de los artistas que los diseñan. Sería prácticamente imposible encontrar algún escenario similar en el mundo real, para escanearlo con sensores o siquiera fotografiarlo. También, en algunas aplicaciones educativas es útil mostrar un concepto abstracto, por ejemplo un fractal, mediante una representación virtual tridimensional, aunque el objeto no exista en la realidad. De especial importancia es el diseño de prototipos para la evaluación de productos antes de su fabricación, tales como zapatos, automóviles y aeronaves, entre otros [48, 2].

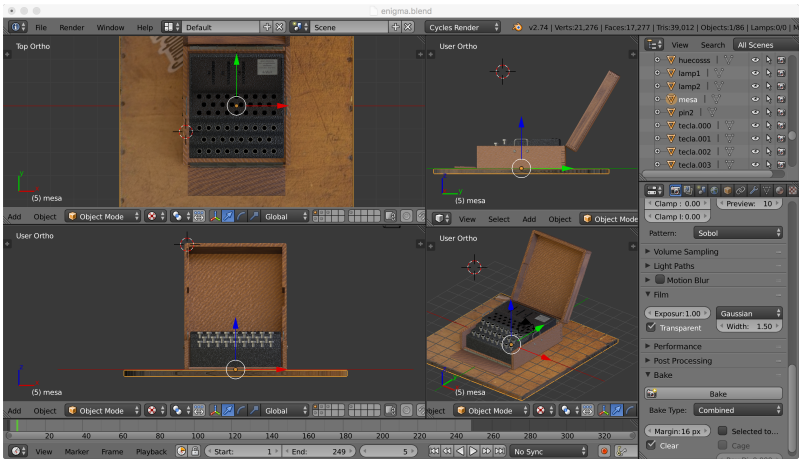


Figura 2.2: Interfaz de usuario de Blender 2.7, en la que se muestra un modelo creado por el autor.

En el proceso tradicional de modelado 3D, la edición se realiza por manipulación directa de los componentes del modelo de la siguiente manera: el usuario proporciona instrucciones al sistema utilizando interfaces de hardware, tales como teclado y ratón, mientras recibe retroalimentación visual a través de vistas proyectivas en la pantalla. Sin embargo, dichas interfaces son de naturaleza bidimensional[1], mientras que el modelo debe ser percibido en tres dimensiones. En la Figura 2.2, se puede observar la interfaz de usuario de Blender, compuesta por elementos WIMP (*Windows*, *Icons*, *Menus*, *Pointer*): ventanas, páneles, menús, botones e íconos, activados mediante el puntero del ratón y accesos directos desde el teclado. El modelo 3D se muestra desde 4 vistas distintas: desde arriba, desde la derecha, desde el frente y desde una cámara con proyección en perspectiva. En el área de Gráficos por Computadora, se estudian diversas técnicas para la creación de modelos, tanto en 2D como en 3D. Algunos tipos de objetos 3D son: mallas poligonales, curvas y superficies de Bézier y NURBS (*Non-uniform Rational B-Splines*), entre otros.

Recordemos que en la geometría tridimensional euclidiana existen entidades tales como puntos, rectas, curvas, superficies, polígonos, cuerpos sólidos, etc. Como mencionan Foley et. al. [25], los objetos de la naturaleza poseen una forma que puede ser aproximada mediante algunas de estas figuras geométricas. Por ejemplo, podemos representar un planeta, como la Tierra, utilizando una forma esférica, aunque si nos acercamos lo suficiente notaremos que no es exactamente una esfera, sino que está achatada en los polos. Si nos acercamos aún más, notaremos que su superficie no es suave en absoluto, tiene montañas, valles, grietas, etc. Esto ocurre también con los objetos fabricados por el hombre, ya que es posible representar una pelota como una esfera y una llanta de automóvil como un cilindro, aunque a pequeña escala tengan irregularidades.

[1]De hecho, la entrada del teclado es secuencial y, por ende, unidimensional.

Frecuentemente, los objetos reales se modelan mediante un conjunto de figuras geométricas. Algunas veces es conveniente utilizar secciones de superficies suaves; en otras ocasiones la forma del objeto se aproxima mejor con una combinación de figuras sólidas, como cubos, esferas, cilindros, conos, prismas y toroides. En este trabajo estamos interesados en la representación de objetos mediante conjuntos de polígonos. A esta representación se le denomina *malla poligonal* o *malla de polígonos* (ver Figura 2.3). Esta representación contiene únicamente la forma del objeto, la cual queda definida por la superficie que constituye la frontera entre el espacio que está dentro del objeto y el que queda fuera. Esta superficie se aproxima mediante un conjunto de polígonos adyacentes, a los que se les suele llamar *caras poligonales*.

Figura 2.3: Modelos 3D construídos a partir de mallas poligonales. La casa y el árbol están formados por figuras geométricas (arriba). La estrella consta de un conjunto de caras triangulares, de las cuales una está seleccionada (abajo izquierda). Se muestran seleccionados los componentes de la cara: una arista (abajo centro) y un vértice (abajo izquierda).

Un modelo construído con *mallas poligonales* se define de la siguiente manera:

- El modelo está formado por un conjunto finito de *mallas poligonales*, posiblemente disconexas.
- Cada *malla* consiste en un conjunto finito de *caras poligonales* adyacentes.
- Cada *cara* está formada por un conjunto finito de *aristas* adyacentes.
- Cada *arista* es el segmento de recta definido por sus puntos extremos, los cuales son llamados *vértices*.

- Dos *caras* son adyacentes si están conectadas entre sí, i.e., si comparten al menos una *arista*.
- Dos *aristas* son adyacentes si comparten exactamente un *vértice*.

2.3. Realidad aumentada

Antes de entrar de lleno en el tema de la realidad aumentada, es importante recordar brevemente la manera en que los seres humanos percibimos la realidad. Desde un punto de vista filosófico, "los seres humanos somos prisioneros de lo que se ha dado en llamar nuestro *predicamento egocéntrico*..."[29], que consiste en el hecho de que todo lo que nos es posible saber sobre el mundo real se basa, en última instancia, en la información que percibimos a través de nuestros cinco sentidos: vista, oído, tacto, olfato y gusto. Aristóteles sostenía una opinión *realista* al afirmar que, detrás de nuestras percepciones, efectivamente existe un mundo que las produce, independientemente de si dicho mundo es percibido por alguna inteligencia. Empero, no es posible demostrar formal y absolutamente esta opinión, debido precisamente a que nuestro único contacto con la realidad son las percepciones de nuestros sentidos.

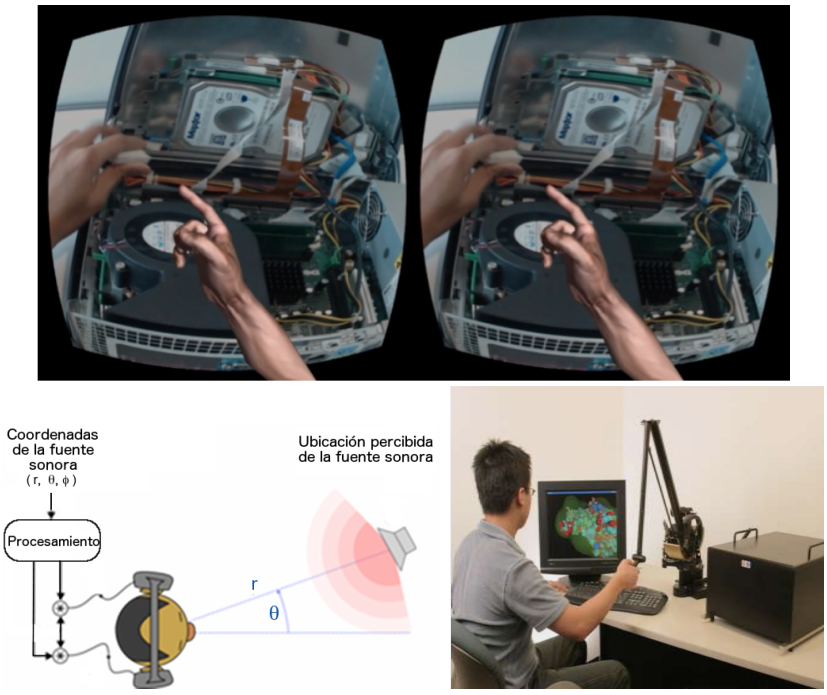


Figura 2.4: Teconogías de presentación de ambientes virtuales a los tres sentidos más importantes: visor estereoscópico Oculus Rift (arriba), sonido estereofónico (abajo izquierda) y dispositivo háptico Phantom (abajo derecha).

Esta circunstancia tiene un aspecto positivo para nosotros como computólogos. A partir de ella se ha originado la idea de que, si fuera posible engañar a nuestros

sentidos con falsas percepciones, entonces sería posible percibir un mundo *virtual*, en el sentido de ser sintético, de la misma manera que si fuese real. Sin embargo, el avance tecnológico actual no es suficiente para engañar todos los sentidos a tal grado. En cambio, se han hecho grandes avances preliminares en la presentación de modelos y ambientes virtuales a los tres sentidos más importantes: la vista, el oído y el tacto [26, 20, 51] (ver Figura 2.4). En menor medida, existen también algunos avances dirigidos a presentar al usuario aromas de manera controlada [37] y a provocar sensaciones de sabor dulce, mediante la estimulación artificial de la lengua [50]. En principio, sería posible experimentar un mundo virtual mediante la estimulación artificial de los cinco sentidos, como se ha mostrado en relatos de ciencia ficción y en producciones cinematográficas, tales como *The Matrix*.

Esta idea de presentar a los sentidos sensaciones de un mundo que no es real, constituye el fundamento de dos importantes tecnologías emergentes: la realidad virtual (*Virtual Reality*, VR) y la realidad aumentada (*Augmented Reality*, AR), las cuales se describen brevemente a continuación. La *realidad virtual* consiste en presentar al usuario la ilusión sensorial de estar, hasta algún cierto nivel de inmersión, en un mundo virtual creado artificialmente, separándolo del mundo real. Por otra parte se tiene la *realidad aumentada*, en la cual no se sustrae al usuario de su mundo real para sumergirlo dentro de uno virtual [17], sino que se le transmiten las percepciones del mundo real, pero enriquecidas con diversos tipos de contenidos virtuales. De hecho, dejar al usuario en su propia realidad es una parte importante del concepto de realidad aumentada. Se intenta crear la sensación de que los modelos virtuales coexisten con el mundo real percibido por el usuario [18]. Para intensificar esta ilusión, los modelos 3D deben presentarse dentro la escena en tiempo de ejecución.

2.3.1. Un poco de historia

De acuerdo a Bimber y Raskar [17] así como a Cawood y Fiala [18], uno de los primeros sistemas de inmersión para Realidad Virtual fue un simulador para una sola persona llamado *Sensorama*, que se muestra en la Figura 2.5. Este simulador fue inventado a finales de los años 50's por el joven cinematógrafo Mort Heilig, que combinaba la proyección de películas 3D, sonido estéreo, un sistema de vibraciones mecánicas de la unidad, el flujo de aire provocado por ventiladores y la aplicación de aromas. Es interesante notar cómo, en esta temprana aproximación al ideal virtual, se intenta involucrar a cuatro de los cinco sentidos. Para provocar el efecto tridimensional en el sentido de la vista se utiliza la *estereoscopía*, que data de 1832, cuando Charles Wheatstone inventó su *visor estereoscópico*.

Sin embargo, un mecanismo como éste solamente proporcionan al usuario la percepción del entorno virtual, sin darle la posibilidad de interactuar con él. El primero en sugerir que el usuario debería ser capaz también de interactuar con el mundo virtual fue Ivan E. Sutherland, en su artículo de 1965, *The Ultimate Display* [60]. Además, desarrolló el primer dispositivo de despliegue visual montado sobre la cabeza, que se muestra en la Figura 2.6. Por ello se dice que fue él quien estableció los fundamentos

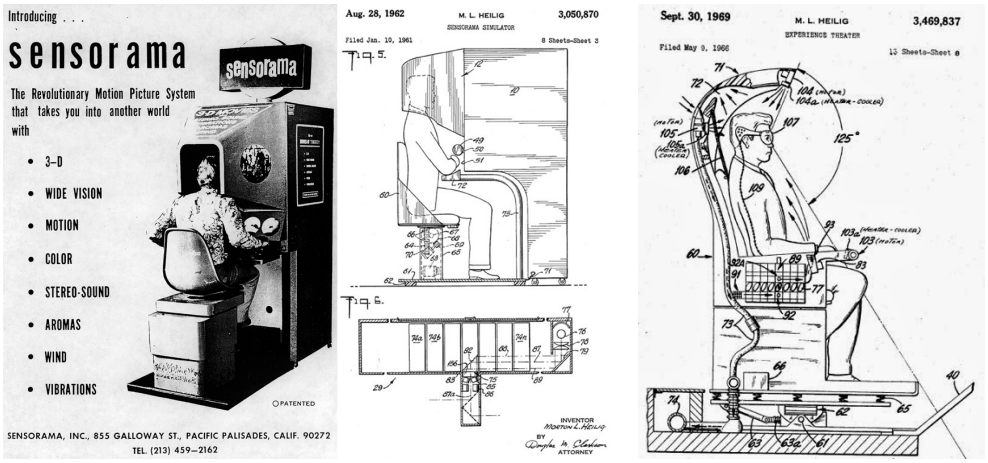
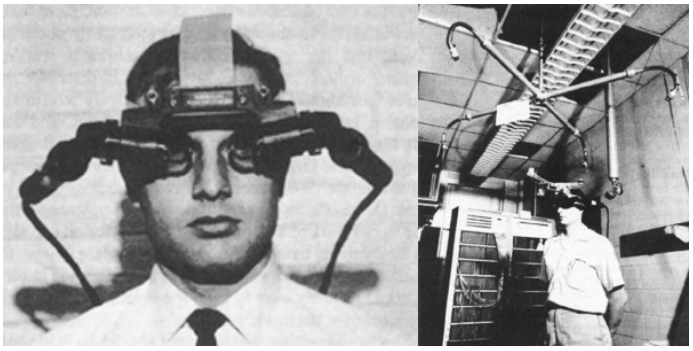


Figura 2.5: Cartel publicitario y diagramas del Sensorama de Mort Heilig.

del área de Realidad Virtual. Además, este dispositivo fue un precursor de los usados para realidad aumentada, pues utilizó espejos semi transparentes para combinar las imágenes de los objetos reales junto con las de los modelos generados por computadora. Este dispositivo de despliegue era tan pesado que debía estar suspendido del techo, motivo por el cual se le puso por apodo la *espada de Damocles*.



(Fuente: https://technosavvyletters.wordpress.com/2013/09/03/virtual-reality/)

Figura 2.6: La espada de Damocles de Ivan Sutherland.

2.3.2. Paradigmas de la realidad aumentada

Los sistemas de realidad aumentada comúnmente se construyen con base en dos paradigmas: el de *espejo mágico* (*magic mirror*) y el de *ventana mágica* (*magic window*). A este último paradigma también se le conoce, en la literatura, como *lente mágico* (*magic lens*). En la parte izquierda de la Figura 2.7 se muestra el paradigma del espejo mágico. Tanto la cámara que captura la escena real, como la pantalla donde se despliega ya aumentada, se encuentran al frente del usuario. La cámara captura lo

Figura 2.7: Paradigmas del *espejo mágico* [18] (izquierda) y de la *ventana mágica* [56] (derecha).

que se encuentra frente a la pantalla, la cual despliega la escena con algunos modelos virtuales adicionales. En conjunto, es posible dar la impresión de que el usuario se está mirando en un espejo, que lo muestra junto con los elementos digitales agregados. En dicha figura, se aprecian también los marcadores que indican al sistema dónde colocar los elementos agregados en tiempo real.

En la parte derecha de la Figura 2.7, se ilustra el paradigma de la ventana mágica. En este caso, se utiliza un dispositivo móvil que cuenta con una pantalla al frente y una cámara ubicada en la parte posterior. Mediante la cámara, se captura la escena que se encuentra al frente del usuario, que en este caso contiene su mano derecha, y en la pantalla se muestra la escena aumentada con un modelo virtual, el cual en este caso está colocado en la palma de su mano. El dispositivo actúa como una ventana a través de la cual se observa una escena aumentada. Este paradigma puede implementarse en cualquier dispositivo, tal como un teléfono móvil, una computadora portátil, una tableta o un dispositivo montado sobre la cabeza, siempre que se cuente con (o se le pueda agregar) al menos una cámara posterior.

2.3.3. Componentes de un sistema de realidad aumentada

En la Figura 2.8 se muestra un diagrama de los bloques que componen un sistema de realidad aumentada, de acuerdo a Bimber y Raskar [17]. El usuario interactúa directamente con la aplicación, la cual está diseñada como una herramienta para resolver algún problema específico. Por ejemplo, el problema que nos ocupa es la edición de modelos 3D (Sección 2.2). La aplicación generalmente puede realizar tres tipos de tareas: interacción con el usuario, presentación de información y generación de algún tipo de contenido o producto final. Por lo tanto, se requiere de módulos de software que las realicen y sobre ellos se construye la aplicación. En la capa más baja del diagrama se encuentran los componentes de hardware sobre los cuales se construyen los módulos de software que se han mencionado. El desarrollo de estos componentes es considerado, por Bimber y Raskar [17], como el principal reto que el área de Realidad Aumentada tiene en la actualidad.

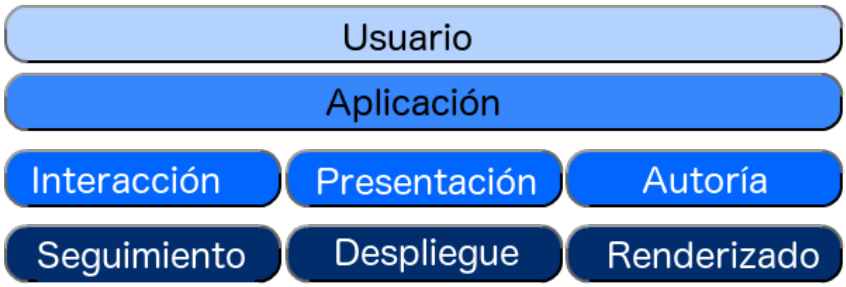


Figura 2.8: Componentes de un sistema de realidad aumentada [17].

Dispositivos de seguimiento

Uno de los problemas fundamentales para desarrollar un sistema de realidad aumentada es llevar el seguimiento de la posición y del movimiento del usuario dentro de su entorno, de una manera precisa, rápida y robusta. A este proceso se le conoce como *registro*. En ocasiones se requiere también llevar el seguimiento de algunos objetos reales que se encuentran en la escena de interés. Existen dispositivos de seguimiento mecánicos, electromagnéticos y ópticos. Los dispositivos de seguimiento suelen clasificarse en dos categorías: con y sin marcadores.

Los dispositivos de seguimiento con marcadores frecuentemente utilizan detectores por medios ópticos (e.g. cámaras) y aprovechan las herramientas del área de Visión por Computadora. Se han utilizado diversos tipos de marcadores que son detectados mediante cámaras: diodos emisores de luz (*Light Emitting Diode*, LED), guantes de colores, etiquetas impresas en tarjetas con patrones similares a códigos QR (del inglés *Quick Response*)[2], entre otros.

El seguimiento sin marcadores constituye un reto mayor, pero a la vez es la línea de investigación más prometedora y se considera lo más deseable en un sistema de realidad aumentada. Aquí también se aprovechan las técnicas de visión computacional para detectar, por ejemplo, las manos o el rostro del usuario mediante cámaras, sin tener que usar algún tipo de marcador.

Según afirman Bimber y Raskar [17], el seguimiento preciso y de alta calidad en entornos muy grandes, es muy difícil de lograr, inclusive con tecnologías como el sistema de posicionamiento global (*Global Positioning System*, GPS) en combinación con dispositivos de medición relativa como giroscopios y acelerómetros.

Tecnologías de despliegue

Las tecnologías de despliegue son otra pieza fundamental de un sistema de realidad aumentada. Existen diversos mecanismos mediante los cuales el sistema despliega la información, destinada a alguno de los sentidos del usuario. Los principales sentidos

[2]Los códigos QR han ganado popularidad por su habilidad de dar acceso a contenidos digitales a partir de medios impresos, usando dispositivos móviles.

a los que se aplican dichos mecanismos son la vista, el oído y, en menor medida, el tacto. Empero, el sentido que no puede faltar en un sistema de realidad aumentada es la vista y frecuentemente se entiende al despliegue como implícitamente referente al despliegue visual. En el presente trabajo seguiremos esta tendencia, por lo que se mencionan únicamente las dos tecnologías de despliegue visual de mayor impacto en la actualidad: las pantallas táctiles y los dispositivos estereoscópicos montados sobre la cabeza. No trataremos los despliegues de audio ni los hápticos, pues quedan fuera del alcance de este trabajo.

Pantallas bidimensionales: La manera más sencilla de presentar la información visual al usuario, mediante una computadora, es a través de vistas proyectivas bidimensionales, tales como la imagen en el monitor de una computadora, la de un proyector sobre una superficie o la pantalla táctil de cristal líquido de un dispositivo móvil. En estos casos, las escenas tridimensionales se muestran proyectadas desde distintas perspectivas, para que el usuario se pueda formar una imagen mental del objeto representado. Sin embargo, esta manera de presentar objetos tridimensionales suele ocasionar confusión en los usuarios no acostumbrados a interpretar las vistas. Por ello, se han desarrollado diversas técnicas orientadas a proporcionar al usuario una percepción tridimensional de las escenas observadas. En el ámbito de los dispositivos móviles, las pantallas no se han diseñado aún para proporcionar una visualización tridimensional[3], pero gracias al uso de realidad aumentada es posible ayudar a la percepción espacial del usuario, como se muestra a lo largo del presente trabajo.

Dispositivo de despliegue montado sobre la cabeza: El dispositivo de despliegue montado sobre la cabeza se conoce como HMD, por sus siglas en inglés *Head Mounted Display*. Se trata de un dispositivo que se lleva en la cabeza, a la manera de un casco, un sombrero o unos anteojos, en el cual se tiene un par de pequeñas pantallas colocadas frente a los ojos del usuario, una para cada ojo. De esta manera, se produce un efecto estereoscópico para brindar la percepción de profundidad en la escena mostrada.

Los seres humanos percibimos las escenas tridimensionales mediante la interpretación de las imágenes obtenidas por nuestros dos ojos. Dado que éstos se encuentran separados por una distancia aproximada de 6.5 cm., dependiendo del individuo, las dos vistas obtenidas son ligeramente diferentes. Interpretando estas diferencias, es como nuestro cerebro es capaz de determinar la profundidad a la que se encuentran los distintos elementos que componen la escena. A este proceso mental se le conoce como *visión estereoscópica* y es la base para el diseño de diversos tipos de *despliegues estereoscópicos*, que son capaces de mostrar un par de imágenes tomadas a partir de una escena tridimensional, real o virtual, desde dos puntos de vista. Para lograr el efecto estereoscópico, cada imagen debe ser mostrada únicamente al ojo que le corresponde, i.e., se debe efectuar la *separación del par estereoscópico*.

[3]Existen dispositivos móviles estereoscópicos, como el *Nintendo 3DS* y el *LG Optimus*, pero queda un largo camino para que se este enfoque se generalice.

En la Figura 2.9, se muestra un diagrama de este mecanismo. Como señalan Bimber y Raskar [17], el movimiento relativo aparente de un objeto distante respecto a uno cercano, conforme el punto de vista se desplaza, se denomina *paralaje* (*paralax*). El punto p_{o1}, de la Figura 2.9, se percibe frente al dispositivo de despliegue estereoscópico, en esta situación se tiene un paralaje negativo. Por otra parte, el punto p_{o2} se percibe detrás del dispositvo, lo cual se conoce como paralaje positivo. Con un paralaje cero o nulo (*zero paralax*) el punto se percibe exactamente sobre la superficie del dispositivo.

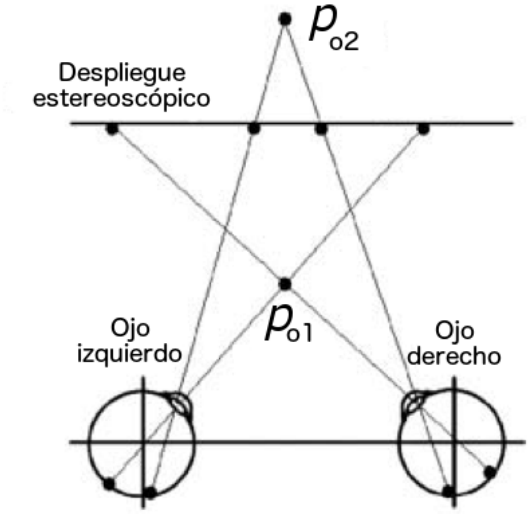


Figura 2.9: Visión estereoscópica [17].

Los primeros HMDs utilizaban pantallas de rayos catódicos (*Cathode Ray Tube*, CRT), pero en la actualidad cuentan con pantallas de cristal líquido (*Liquid Crystal Display*, LCD) que son mucho más ligeras. Estos dispositivos son ampliamente utilizados en aplicaciones de realidad virtual y realidad aumentada, debido al alto nivel de inmersión que se logra con ellos. Un requisito indispensable para obtener dicho nivel de inmersión es efectuar un seguimiento preciso de la posición y orientación de la cabeza del usuario. De esta manera, es posible calcular correctamente el par de imágenes que se debe mostrar en cada pantalla.

Para visualizar una escena en realidad virtual, se colocan las pantallas directamente frente a cada ojo del usuario. De esta manera, se obstruye la visión del mundo real para observar solamente el mundo virtual mostrado en ellas. Cuando se quiere visualizar una escena en realidad aumentada, el par estereoscópico se puede mostrar de dos maneras: por combinación óptica y por combinación por video. Lo que se busca es combinar el par de imágenes estereoscópicas del mundo real con el par estereoscópico virtual generado por computadora. A esta combinación de realidades se le conoce por el nombre *see-through*, que sugiere el hecho de ver el mundo real a través de una escena virtual.

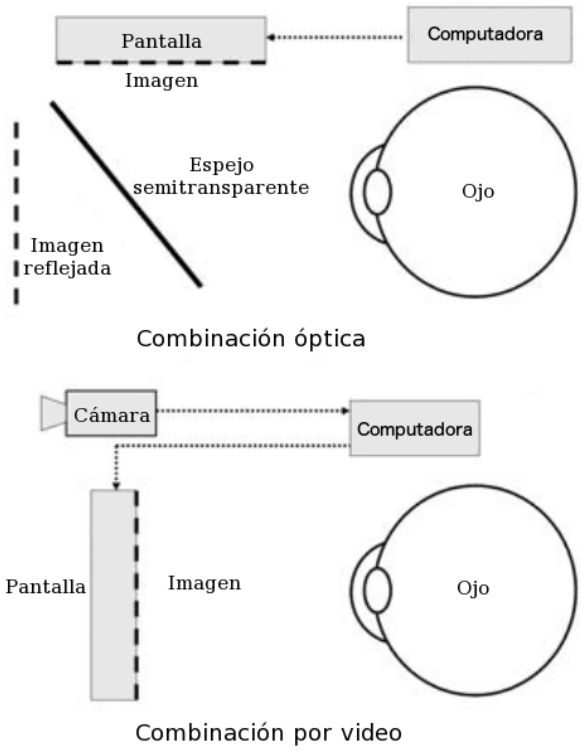


Figura 2.10: Dos maneras de combinar elementos reales y virtuales en un HMD [17].

En la parte superior de la Figura 2.10 se ilustra, de manera simplificada, la idea de combinación óptica para uno de los ojos. Se utiliza un espejo semi transparente para combinar la imagen real que llega al ojo desde el frente y la imagen de la pantalla ubicada a un lado. Otra manera de combinar las imágenes es colocar una pantalla de cristal líquido transparente frente al ojo. De esta manera, literalmente se puede ver la escena real a través de la pantalla.

En la parte inferior de la Figura 2.10, se muestra la manera de realizar la combinación por video. En este caso, la pantalla LCD se encuentra frente al ojo y obstruye la vista del usuario hacia el mundo real. Una cámara ubicada aproximadamente en la línea de visión del ojo captura digitalmente la escena del mundo real. La computadora agrega a esta escena los elementos virtuales y posteriormente se muestra el resultado en la pantalla. El usuario percibe una combinación de las dos realidades.

Los HMD constituyen, en la actualidad, la tecnología dominante en aplicaciones de realidad virtual, debido al nivel de inmersión que porporcionan y a que pueden utilizarse tanto en aplicaciones móviles como en aplicaciones en las que se requiere que interactúen varios usuarios, cada uno usando su propio dispositivo. Este tipo de equipo suele ser muy costoso, si se busca gran calidad de imagen.

El HMD más potente y popular es el *Oculus Rift*, pero es costoso. Oculus ofrece una opción relativamente económica con el dispositivo *Gear*, que consiste en una carcaza similar a la del Rift, a la que se le puede colocar un teléfono inteligente,

el cual funge como unidad de procesamiento y realiza el despliegue en su pantalla. Cabe mencionar también, el proyecto *CardBoard* de Google, en el cual se brinda una manera económica de convertir un teléfono inteligente en un pequeño HMD.

Sin embargo, el uso de HMDs presenta también ciertas desventajas:

- Regularmente el despliegue no tiene una alta resolución, la cual viene limitada por la calidad de las pantallas en miniatura. En el caso de la combinación óptica, la resolución de los elementos virtuales es relativamente menor, en comparación con la escena real, la cual se aprecia a la resolución del sistema visual humano. En la combinación por video, la escena real y los elementos virtuales se muestran con una resolución limitada, tanto por las cámaras como por las pantallas en miniatura.

- El campo de visión está limitado por el tamaño de las pantallas y el sistema óptico del visor.

- Los despliegues de combinación óptica requieren una calibración difícil y dependiente de la sesión y del usuario, además de un seguimiento preciso de la cabeza para realizar la combinación correctamente.

- Los despliegues de combinación óptica convencionales son incapaces de mostrar efectos de oclusión consistentes entre los objetos reales y los modelos virtuales. Se puede simular un objeto real obstruyendo parte del modelo virtual, al dibujar solamente la región visible de éste último. Pero cuando el modelo virtual obstruye al real, no se puede dejar de ver alguna parte de éste.

Proceso de render

El tercer elemento básico en un sistema de realidad aumentada es el proceso de *render* en tiempo real. El término *render* se refiere al proceso de generación de una imagen, calculando el color que le corresponde a cada *pixel* [4] que la compone. En esta etapa, toman un papel muy importante los métodos de *render* rápido y realista. La meta ideal de un sistema de realidad aumentada es integrar los modelos virtuales en un entorno real, de tal manera que el usuario no sea capaz de distinguir lo que es real de lo que no lo es. Sin embargo, como se ha mencionado, la tecnología actual aún no permite obtener tal nivel de realismo en tiempo de ejecución. Las grandes producciones cinematográficas, ahora también en 3D, y los programas televisivos nos tienen acostumbrados a ver escenas digitales con un nivel de realismo sin precedentes. Empero, no es posible lograrlo en tiempo real, sino que se requiere una enorme cantidad de tiempo y recursos computacionales para realizarse.

En las aplicaciones interactivas es imprescindible realizar el proceso de *render* en tiempo real, por lo cual se establece un compromiso práctico entre el nivel de realismo y la velocidad de despliegue, prefiriéndose ésta última a costa de sacrificar en cierta medida la apariencia de los modelos virtuales, haciéndolos notoriamente sintéticos.

[4] Contracción del término *picture element* (*elemento de imagen*).

Hoy en día, los avances en las tarjetas gráficas de escritorio y la aceleración del hardware en los dispositivos móviles han permitido mejorar la apariencia de estos modelos en gran medida. A este respecto, merece especial atención el caso de los videojuegos de última generación que, en la actualidad, ofrecen renderizados interactivos con una apariencia muy cercana a la de las grandes producciones cinematográficas. Esto es posible gracias al enorme avance de las consolas especializadas, que cuentan con una gran cantidad de recursos computacionales. Sin embargo, aún es posible notar las pequeñas diferencias que existen entre los elementos reales y los virtuales, especialmente si se colocan unos junto a otros o se les combina, como se requiere en las aplicaciones de realidad aumentada.

2.3.4. Seguimiento con marcadores

Como se ha mencionado, uno de los problemas fundamentales de la realidad aumentada es alinear correctamente, en tiempo real, los objetos virtuales dentro de la escena real capturada por la cámara del dispositivo. A este proceso de alineación se le conoce, en el área de Gráficos por Computadora, como *registro*. Para ello, se necesita tomar algún elemento del entorno real como marcador para que el software lo pueda reconocer y seguir de manera precisa, con el fin de colocar los elementos virtuales registrados con su entorno.

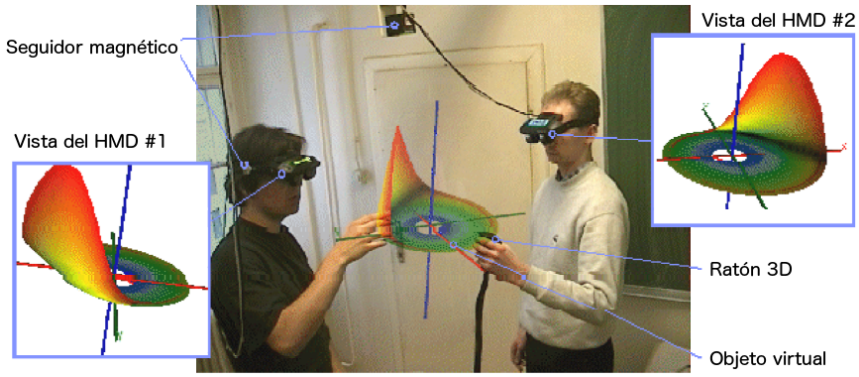


Figura 2.11: Dos colaboradores colocalizados examinan un objeto 3D compartido, usando *Studierstube*. El seguimiento de los HMDs se realiza mediante un dispositivo magnético y la interacción se logra mediante un ratón tridimensional.

Algunos de los primeros intentos de crear experiencias de realidad aumentada colaborativa son *Shared Space* [16] y *Studierstube* [55, 12]. Ambos sistemas utilizan el *kit* de desarrollo para seguimiento de marcadores visuales, llamado *ARToolkit* [5].

El proyecto *Studierstube* [55, 12] pretende crear un administrador de interfaces de usuario en 3D, de propósito general, que proporcione una base sobre la cual construir aplicaciones colaborativas que utilicen realidad aumentada. El objetivo de este administrador es crear una representación de interfaz de usuario en 3D tan poderosa como

[5]Disponible en: http://www.hitl.washington.edu/artoolkit/download/

la representación de escritorio para espacios de trabajo en 2D. En la Figura 2.11 se puede ver un ejemplo de las aplicaciones que se pueden construir con *Studierstube*. El entorno, creado mediante este administrador, es controlado por el panel de interacción personal (*Personal Interaction Panel*, PIP), el cual es una interfaz bimanual con stylus y tablilla, que provee una interacción versátil con los objetos virtuales. Un ejemplo de interfaz controlada mediante un PIP se muestra en la parte izquierda de la Figura 2.12.

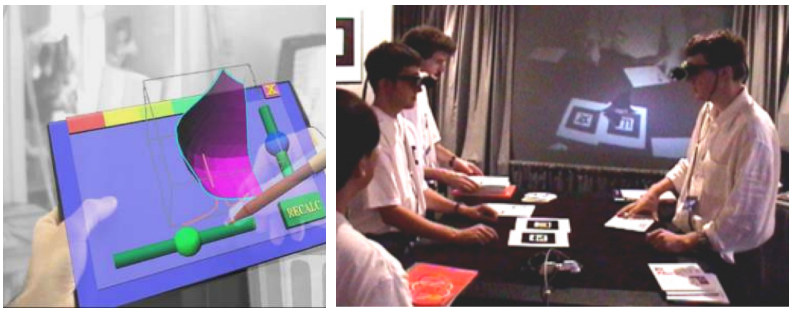


Figura 2.12: Ejemplo de un PIP, al que se han agregado elementos de interfaz de usuario en 2D y 3D (izquierda). En el entorno colaborativo de *Open Shared Space* (derecha) los usuarios pueden ver e interactuar con objetos reales y virtuales, así como con otros participantes.

En el proyecto *Open Shared Space* [16], Billinghurst et al. proponen el uso de realidad aumentada en el espacio de trabajo, mediante el cual un grupo de usuarios colocalizados interactúan entre sí y también con herramientas físicas, tales como lápiz y papel. Su principal objetivo es mantener la continuidad de la interfaz entre las herramientas existentes y los nuevos sistemas computacionales, sin que interfieran entre sí. En la parte derecha de la Figura 2.12, se muestra el entorno colaborativo de este proyecto.

Tomando a *Open Shared Space* como base, varios trabajos posteriores se enfocan en el uso de las llamadas interfaces de usuario tangibles (*Tangible User Interfaces*, TUI), que son objetos físicos con marcadores. Además, Billinghurst et al. [15], proponen el enfoque denominado *realidad aumentada tangible* (*Tangible Augmented Reality*), el cual consiste en combinar las interfaces de usuario tangibles con realidad aumentada.

Seguimiento de imágenes arbitrarias

En los sistemas clásicos de seguimiento, como el de la Figura 2.13, los marcadores suelen tener alguna estructura específica, e.g., en *ARToolKit* los marcadores son un arreglo de cuadrados negros sobre un fondo blanco, enmarcados por un borde negro y grueso. En la actualidad, mediante el uso de algoritmos eficientes de visión por computadora, en dispositivos móviles, es posible realizar el seguimiento de casi cualquier imagen, en tiempo real. Recientemente se han aplicado técnicas de aprendizaje maquinal (*machine learning*) a la tarea de seguimiento visual de imágenes arbitrarias,

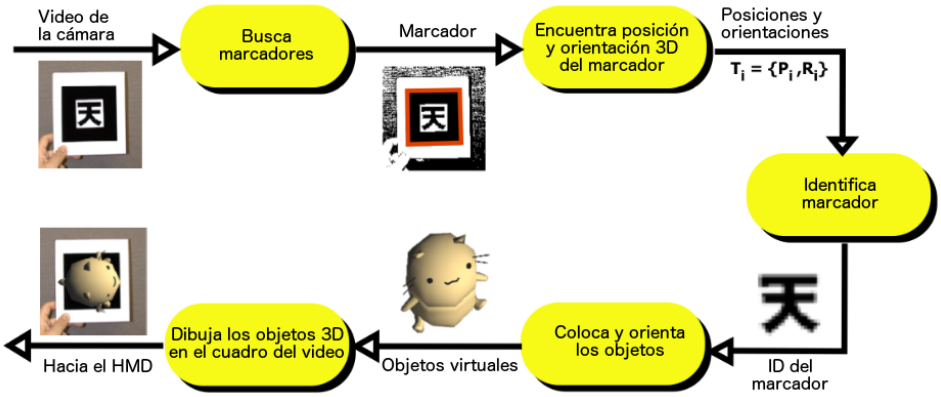


Figura 2.13: Diagrama de flujo para el mecanismo de seguimiento de *ARToolKit*. Se puede observar el tipo de marcador que se utiliza: una forma en color negro sobre un fondo blanco y enmarcada por un borde negro y grueso. Otro tipo de marcadores tiene un arreglo de cuadrados dentro del borde.

i.e., que no tengan restricciones en su contenido. Sin embargo, ciertas imágenes funcionan mejor que otras, dependiendo de los algoritmos utilizados. Una técnica consiste en utilizar el detector de esquinas FAST [36] junto con variantes de los clasificadores por bosques de helechos (*fern forests*) [53] y SIFT (*Scale Invariant Feature Tracking*) [64]. Estos algoritmos se utilizan para hacer el seguimiento de características visuales en el video capturado a través de una cámara. La escena capturada contiene un marcador físico, con una imagen impresa en él, denominada imagen objetivo (*image target*). Como se muestra en la Figura 2.14, en cada cuadro del video, el algoritmo extrae varias características visuales de la imagen objetivo, tales como las esquinas entre regiones con alto contraste local.

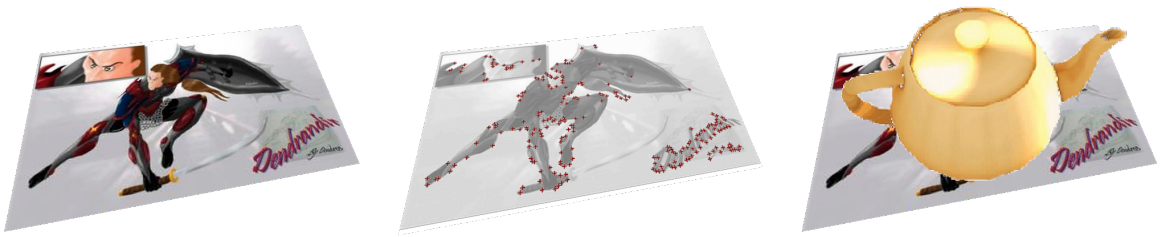


Figura 2.14: La imagen de la escena real con un marcador se captura a color (izquierda), después se transforma a escala de grises y se detectan los puntos característicos (centro), señalados con pequeños signos +. Finalmente, el objeto virtual se dibuja alineado sobre la escena real (derecha).

Usando tales puntos característicos, se calcula la matriz de transformación $M = R|\vec{t}$, que representa la traslación $\vec{t}$ y la rotación R de la cámara respecto al marcador físico. La matriz M transforma los puntos en el marco de referencia del marcador en puntos dentro del marco de referencia de la cámara. En la Figura 2.15, se muestran

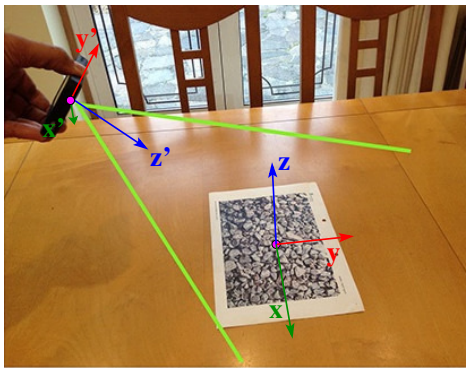


Figura 2.15: La cámara se encuentra trasladada y girada respecto a la imagen objetivo o viceversa. Además, el eje Z se invierte.

ambos marcos de referencia, tanto del marcador físico como de la cámara. Utilizando la matriz de transformación M, se pueden dibujar los objetos virtuales tridimensionales correctamente alineados con el marcador en el video, usando la expresión $\vec{p'} = M\vec{p}$, como sigue:

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \left(\begin{array}{ccc|c} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Un registro correcto genera la ilusión de que el objeto se encuentra dentro de la escena real. En la parte derecha de la Figura 2.14, una tetera virtual se dibuja en la posición y orientación, dada por la matriz M, sobre la imagen capturada de la escena real.

Capítulo 3

Trabajo relacionado

En el presente capítulo presentamos una breve reseña en orden cronológico del trabajo relacionado acerca de: *juegos para dispositivos móviles con realidad aumentada*, *servicios de publicación de contenidos de realidad aumentada*, *editores colaborativos de mallas poligonales* y trabajos de investigación bajo el enfoque de *realidad aumentada colaborativa*.

3.1. Juegos para dispositivos móviles con realidad aumentada

Actualmente, existe una gran variedad de juegos para dispositivos móviles que cuentan con capacidades de realidad aumentada y algunos de ellos están diseñados para múltiples jugadores. Algunos ejemplos de dichos juegos son: *AR Basquetball*[1], *AR Invaders*[2], *Ingress*[3] y *Pokemon Go*[4].

Aunque muchos de ellos son multiusuario, ninguno es colaborativo en el sentido estricto del área de Trabajo Colaborativo Asistido por Computadora y, en general, no se busca crear ni bosquejar contenidos de algún tipo.

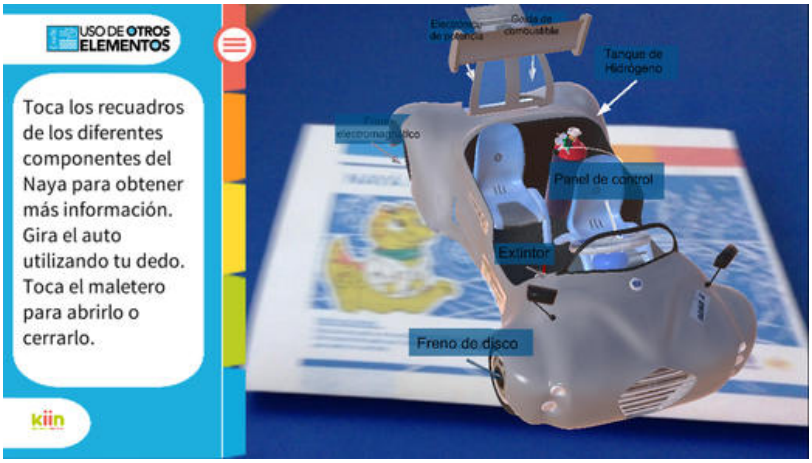
3.2. Servicios de publicación de contenidos en realidad aumentada

También existen diversos servicios de publicación de contenidos digitales para dispositivos móviles, que utilizan realidad aumentada, tales como *Layar*[5] y *Metaio*[6], las

[1]https://itunes.apple.com/mx/app/arbasketball-augmented-reality/id393333529?mt=8
[2]https://play.google.com/store/apps/details?id=com.soulbit7.game.arinvaders
[3]https://www.ingress.com/
[4]https://play.google.com/store/apps/details?id=com.nianticlabs.pokemongo
[5]https://www.layar.com/
[6]http://www.metaio.com/

cuales ofrecen servicios en la nube para publicar, en medios impresos, contenido digital presentado en realidad aumentada. Estos servicios están destinados principalmente a la publicidad, de manera que las aplicaciones, que hacen uso de ellos, generalmente son monousuario y no están orientadas a tareas de autoría o bosquejo de contenidos.

En esta modalidad, usualmente se tiene un medio impreso, una revista, un volante o un poster, que contiene algún tipo de marcador especial reconocible por la aplicación de realidad aumentada, la cual debe ser instalada previamente en el dispositivo móvil. Al ejecutarse la aplicación, esta reconoce el marcador impreso y despliega sobre éste el contenido digital, que puede ser un modelo 3D de un producto, alguna información o un video, como se muestra en la Figura 3.1, con la aplicación Kiin.



Fuente: https://itunes.apple.com/mx/app/kiin/id1168800475

Figura 3.1: Ejemplo de contenido digital aumentado sobre un tríptico impreso: se muestra un modelo virtual del auto de Hidrógeno Naya, construído en CINVESTAV.

Otra modalidad de realidad aumentada, que se utiliza en aplicaciones comerciales y de publicación de contenidos, está basada en geolocalización mediante GPS. Un ejemplo de este tipo de aplicación es propuesto, para su uso en aulas, por Christen Bouffard[7], en el cual utiliza *Wikitude* junto con Google Maps para compartir ubicaciones en mapas y visualizarlas *in situ*, utilizando realidad aumentada.

Aunque se afirma que estos mapas compartidos proveen realidad aumentada colaborativa, no es posible que los usuarios modifiquen el contenido, ni que interactúen de manera alguna entre ellos. El autor publica las ubicaciones y los demás usuarios solo pueden visualizarlas, de modo que los contenidos se comparten, pero no se pueden editar ni se proporciona algún tipo de colaboración.

[7]http://www.wikitude.com/build-wikitude-world-google-collaborative-maps/

3.3. Editores colaborativos de mallas poligonales

Algunos de los editores 3D más populares en la actualidad son *Blender*[8], *Maya*[9] y *3D Max*[10]. Con ellos es posible crear diversos tipos de objetos 3D, incluídos los basados en mallas poligonales. Estas aplicaciones se ejecutan en computadoras de escritorio, son monousuario y utilizan el paradigma de interacción *WIMP* (*Windows, Icons, Menus, Pointer*), i.e., están basadas en ventanas, íconos y menús, activados a través del puntero del ratón.

En proyectos demasiado grandes o complejos, se requiere de la participación de una gran cantidad de personas. Dado que las aplicaciones de modelado son monousuario, es frecuente que los colaboradores se dividan el trabajo. Cada participante modela algunos personajes u objetos de la escena 3D y al terminar los comparte con el resto del equipo, a través de un repositorio centralizado. Ellos cooperan en la creación de los contenidos, pero deben tomar turnos cuando sea necesario que varios trabajen en la creación o modificación del mismo objeto.

Un ejemplo de lo anterior es *Clara.io*[11], una herramienta de modelado basada en la nube. La edición de modelos 3D, basados en mallas poligonales, se realiza en un navegador. Las escenas 3D, compuestas de varias mallas poligonales, se pueden compartir entre varios usuarios registrados, a los cuales se les proporcionan ciertos privilegios de edición. Su enfoque de colaboración consiste en que cada usuario edite en paralelo una malla cada uno, la cual queda almacenada en la nube, luego los colaboradores reciben actualizaciones de los modelos modificados, dando la apariencia de edición en tiempo real. También se permite a varios usuarios trabajar de manera concurrente en una misma malla, sin embargo, Salvati et al. [54] mencionan que *Clara.io* aún presenta problemas en dicha situación.

En *GitMesh* [23], se presenta un algoritmo práctico para calcular diferencias (*diffing*) y uniones (*merging*) de versiones de mallas poligonales, inspirado en el proceso de control de versiones en la edición distribuida de documentos de texto, e.g., la herramienta *GitHub* [12]. En este trabajo, Denning y Pellacini proponen una medida de no-similaridad entre dos versiones de una malla poligonal, llamada distancia de edición entre mallas (*mesh edit distance*), que consiste en estimar el conjunto de operaciones de edición que permiten obtener una versión a partir de la otra. Sin embargo, este algoritmo no es muy robusto y, en ocasiones, descarta operaciones de manera innecesaria, dada la naturaleza aproximativa de dicha distancia, y resulta lento en mallas grandes o complejas.

Posterior a *GitMesh*, se crea *MeshHisto* [54], en el cual también se trata el problema de edición colaborativa de mallas poligonales, como si fuera un caso especial de control distribuido de versiones. El algoritmo de *MeshHisto* permite realizar la edición concurrente al compartir y unir, de manera robusta y en tiempo real, el historial de

[8] https://www.blender.org/
[9] http://www.autodesk.mx/products/maya/overview
[10] http://www.autodesk.mx/products/3ds-max/overview
[11] https://clara.io/
[12] https://github.com/

edición de las distintas versiones de mallas. Además, *MeshHisto* mejora a *GitMesh* en robustez, velocidad y reducción de operaciones descartadas erróneamente.

En la Figura 3.2, se muestra un ejemplo de la unión de dos ramas en la edición de una malla común. Al efectuar uniones de versiones (*merging*) de manera frecuente, los usuarios comparten sus cambios en tiempo real. Durante la unión, cuando el sistema detecta conflictos entre los historiales de edición, el usuario puede elegir una resolución automática, que consiste en descartar las operaciones conflictivas. O bien, se pueden elegir de manera manual las operaciones que se desea conservar. Otra posibilidad consiste en efectuar la edición o en su caso reparación, utilizando una herramienta externa, e.g., un editor 3D monousuario. El prototipo de editor colaborativo que se implementa en *MeshHisto* está basado en Blender.

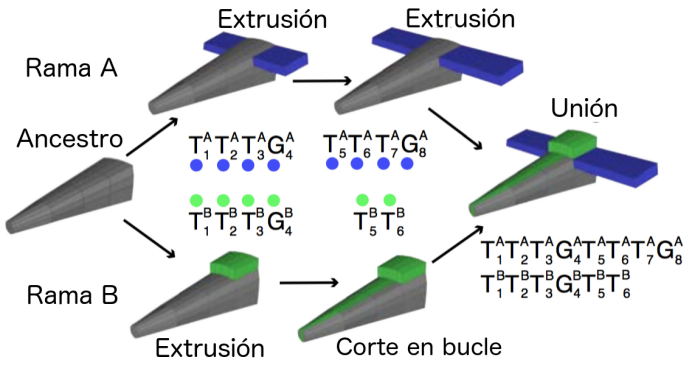


Figura 3.2: Ejemplo de dos ramas de versión en MeshHisto. Partiendo de un ancestro común, en la rama A se aplican dos operaciones de extrusión sobre una selección de caras, mientras que en la rama B se aplica una extrusión a otra cara, seguida de una operación de corte en bucle. Ambas ramas se unen (*merge*) dando lugar a una nueva malla que incorpora todas las operaciones.

Este enfoque de considerar la edición de mallas como un control distribuido de versiones constituye una solución práctica muy útil para abordar el problema de edición colaborativa y es una de las primeras ideas que viene a la mente. Sin embargo, como se mencionó en la Sección 2.1, este enfoque no es del todo adecuado para una sistema colaborativo. Basta mencionar que el control de inconsistencias, al descartar operaciones conflictivas, puede producir desconcierto en los usuarios y causar la impresión de que el sistema se comporta mal o no responde. Obsérvese que, en los editores 3D de la presente sección, no se hace uso de realidad aumentada, sino que se toma el esquema de visualización e interacción tradicional. En la Sección 3.4, se describe el enfoque orientado a la colaboración que toma la comunidad de Trabajo Colaborativo Asistido por Computadora, en el tema de realidad aumentada colaborativa.

3.4. Realidad aumentada colaborativa

En esta sección, se mencionan las características de diversos trabajos de investigación en realidad aumentada colaborativa, en orden cronológico y desde el punto de vista de los sistemas colaborativos. Al final, se muestra una tabla comparativa de estas propuestas con respecto a los prototipos implementados con ShAREd.

Se han desarrollado diversos trabajos utilizando *Studierstube*. Schmalstieg et al. [8] presentan un sistema de videoconferencia con realidad aumentada, que permite colaborar a dos usuarios remotos, sin más equipo que una PC con una cámara Web y un par de marcadores de *ARToolKit*, uno para manipular el modelo virtual y otro para el panel de interacción personal (PIP). En la Figura 3.3 se muestra el espacio de trabajo necesario para la videoconferencia y a dos usuarios utilizando dos tipos de aplicación colaborativa con realidad aumentada: para la exploración de modelos 3D y para la visualización de modelos geométricos.

Figura 3.3: Videoconferencia colaborativa mediante realidad aumentada, utilizando *Studierstube*. Se muestra el espacio ergonómico de uno de los usuarios (izquierda). Dos usuarios durante una videoconferencia: se muestra una aplicación de exploración de modelos 3D (centro) y una aplicación de visualización de modelos geométricos (derecha).

Billinghurst y Kato [40] presentan un prototipo de un sistema de videoconferencia entre un usuario local y dos remotos, basado en *Open Shared Space*. El usuario local usa un HMD óptico y los otros usan una PC convencional con una cámara Web. Ellos comparten un pizarrón virtual en el que todos pueden dibujar y escribir notas de texto de manera concurrente. En la Figura 3.4 se muestra el esquema de comunicación de este sistema y un par de capturas de lo que ve el usuario local.

Open Shared Space también fue utilizado para construir *Magic Book* [14], un sistema que consiste en aumentar un libro físico con escenas en 3D compartidas, en las cuales un grupo de usuarios colocalizados puede ver el mismo contenido y navegar virtualmente dentro de las escenas, cambiando de un modo de despliegue basado en realidad aumentada a uno basado en realidad virtual. Cuando los usuarios entran al modo de realidad virtual, el sistema los muestra como pequeños avatares a los demás usuarios, con el fin de hacerlos conscientes de su ubicación en la escena. Los usuarios inmersos también pueden ver a aquellos que se encuentran fuera de la escena como grandes cabezas que miran desde el cielo. La Figura 3.5 muestra las distintas modalidades del sistema.

Figura 3.4: Sistema de videoconferencia con realidad aumentada, utilizando *Open Shared Space*. Esquema de comunicación entre un usuario local y dos remotos (izquierda). La vista de usuario local a través de un HMD óptico (centro), donde puede ver en video a los colaboradores remotos, aumentados sobre los marcadores. Un pizarrón compartido en realidad aumentada (derecha).

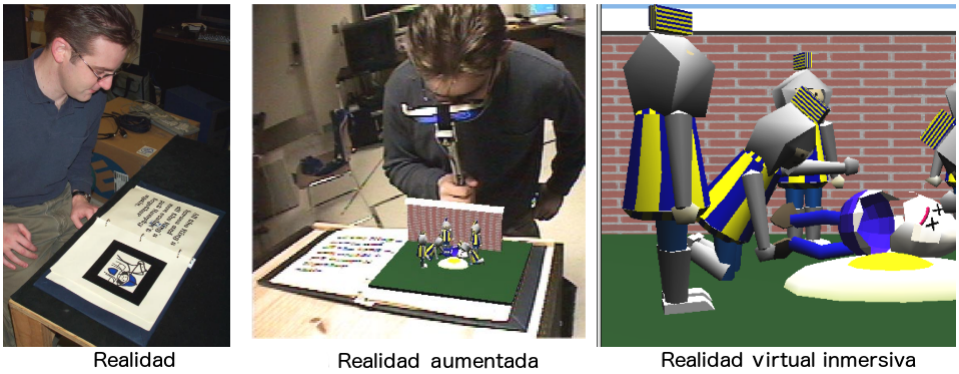


Figura 3.5: El libro real utilizado por *Magic Book* (izquierda). El usuario utiliza un HMD compacto para observar la escena animada con realidad aumentada (centro). La vista inmersiva en realidad virtual de la misma escena del libro (derecha).

Mogilev et al. [46] presentan *AR Pad*, un dispositivo de manejo manual, provisto de una pequeña pantalla, un *trackball* y varios botones (ver Figura 3.6). Se intenta proporcionar una interfaz intuitiva para visualizar contenido digital utilizando realidad aumentada. Se trata de un invento de una época previa al auge de los teléfonos inteligentes, c. 2002, que necesita estar conectado a una computadora, para que realice el procesamiento. En retrospectiva, lo podemos describir como un prototipo de un dispositivo móvil moderno, en el que se delega todo el procesamiento. Utiliza la implementación del mecanismo de seguimiento de ARToolKit y permite la colaboración entre dos usuarios colocalizados que comparten y manipulan un conjunto de cubos virtuales.

Basándose también en Studierstube, Wagner et al. proponen el juego llamado *Invisible Train* [63], que es considerado como la primera aplicación multiusuario para dispositivos móviles, que utiliza realidad aumentada. Los autores optaron por utilizar PDAs como plataforma de ejecución gracias a su buen compromiso entre poder de procesamiento, tamaño y peso, con respecto a los teléfonos móviles y tabletas disponibles en 2004. Wagner et al. diseñaron un escenario, que se muestra en la Figura 3.7,

Figura 3.6: El dispositivo *AR Pad* consta de un control compacto con *trackball*, botones, una pantalla LCD y una cámara Web (izquierda). AR Pad apuntando hacia un marcador de *ARToolKit* (centro). Un colaborador observa la escena real junto con un par de modelos virtuales aumentados y el colaborador con su *AR Pad* (derecha), se muestra también el frustum y el objeto seleccionado por él, en líneas rojas.

para soportar cuatro jugadores, quienes controlan trenes virtuales que se desplazan sobre los rieles físicos. Los usuarios pueden participar en modo competitivo, tratando de estrellar al tren contrario mediante el manejo de los cambios de vía y acelerando o frenando su propio tren, o en modo colaborativo tratando de evitar la colisión.

Figura 3.7: En *Invisible Train* se utiliza una PDA con cámara (arriba izquierda). Se apunta la cámara hacia las vías de madera con marcadores de *ARToolKit* (arriba derecha). Dos usuarios (abajo izquierda) colaboran o compiten al hacer cambios de vía, usando sus stylus. La vista de dos PDAs (abajo derecha) es sincronizada via WiFi.

El sistema *ARprism* [13] ofrece soporte a la colaboración cara a cara en tareas de visualización de datos geográficos, entre dos colaboradores que utilizan un HMD ligero. Este sistema aumenta un mapa desplegado en una mesa, como se observa en la Figura 3.8, con diferentes capas de datos topográficos, tales como coordenadas del

sitio, elevación, tipo de suelo, hidrología y estructura subterránea. Las representaciones en 3D del terreno se modelan previamente y no pueden ser modificadas por los usuarios.

Figura 3.8: Modelo geológico visualizado en realidad aumentada (primera, de izquierda a derecha), dos colaboradores utilizando *ARprism* (segunda), visor *Sony Glasstron* colocado en una montura (tercera) y modelo de un terreno con estructura subterránea (cuarta).

Regresando a las primeras propuestas de desarrollo sobre realidad aumentada, de acuerdo a Wagner et al. [7], la mayoría de las aplicaciones, en el periodo que va de 1968 a 2001 aproximadamente, requería el uso de pesados y engorrosos HMDs, que debían estar conectados a computadoras de escritorio o portátiles transportadas en mochilas. Resulta interesante que, en 2006, Billinghurst et al. [13] afirmaron que sería más amigable para el usuario reemplazar los incómodos HMDs por despliegues LCD portátiles y equipados con pequeñas cámaras Web, posiblemente haciendo referencia a *AR Pad* [46] de 2002. Este tipo de dispositivos evitarían el "mal del simulador" (*simulator sickness*) y reducirían la interferencia en la comunicación entre usuarios, así como el aislamiento del campo de visión. Hoy en día, gracias al advenimiento de dispositivos móviles cada vez más poderosos, esta es la configuración preferida para aplicaciones asequibles de realidad aumentada móvil.

CoMaya [1] es un *plug-in* colaborativo para el popular paquete de modelado 3D, llamado *Maya*, el cual permite que varios usuarios conectados a un server central, trabajen en la misma escena 3D de manera concurrente. En las Figuras 3.9 y 3.10 se muestra el diagrama de funcionamiento y la propagación de operaciones de *CoMaya*. Utiliza la técnica de Transformación Operacional [61] para mantener la consistencia de los datos compartidos entre las copias distribuidas. En caso de que no sea posible resolver las operaciones conflictivas, se le pide a un colaborador que las resuelva. Agustina et al. lograron crear reglas de transformación únicamente para un subconjunto de las numerosas operaciones proporcionadas por *Maya*. Sin embargo, a pesar de que el proyecto parece haberse estancado, fueron capaces de realizar la edición colaborativa de escenas 3D, utilizando la interfaz estándar de *Maya* y el paradigma de teclado y ratón en computadoras de escritorio.

En el ámbito del diseño de productos, Nam et al. [48] presentan un espacio colaborativo basado en realidad aumentada para la revisión síncrona y distribuida de productos. Este espacio colaborativo emplea plataformas giratorias, tangibles y sincronizadas, así como sombras virtuales para soportar consciencia de grupo con colaboradores remotos. Como se muestra en la Figura 3.11, este sistema de revisión

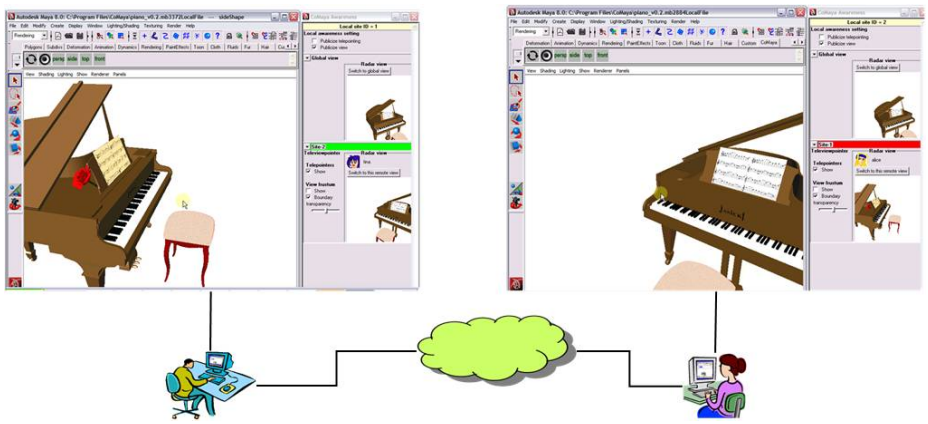


Figura 3.9: Diagrama de funcionamiento de *CoMaya*, las operaciones efectuadas en la interfaz gráfica de usuario de *Maya*, por cada usuario, se transmiten por Internet a los demás colaboradores, mediante un servidor centralizado.

Figura 3.10: Las operaciones ejecutadas por Mel, en la interfaz de Maya, se interceptan y propagan desde su espacio de trabajo hacia el de Tina, donde se ven reflejadas en tiempo real.

de productos permite a dos participantes, alejados entre sí, examinar un modelo 3D compartido, hablar entre ellos vía voz y transmitir la rotación del objeto, con el fin de ayudar a enfocar la discusión en alguna parte del mismo. De acuerdo a los resultados reportados, la plataforma giratoria fue un éxito, pero el uso de sombras virtuales no lo fue, debido principalmente a que fue difícil para los usuarios señalar en un espacio en 3D usando únicamente sombras en 2D.

St-Aubin et al. [57], presentan un sistema colaborativo de diseño y planificación urbana, que combina la tecnología de despliegue geoespacial basado en realidad aumentada con algunas herramientas de modelado 3D y análisis espacial. Crean un marco de desarrollo basado en la herramienta de programación de juegos de Microsoft, *XNA Game Studio 3.0*, añadiéndole funcionalidades básicas de un GIS y utilizan la estructura de datos *Open Graph Scene* (OSG), para la representación de escenas 3D. En la Figura 3.12 se muestra el sistema en ejecución y la herramienta tangible

Figura 3.11: Un colaborador examina el modelo virtual de un producto en la mesa giratoria (superior izquierda). El producto que observa es una cámara (superior derecha). Cuando dos colaboradores examinan el producto pueden girar las mesas de manera sincronizada para enfocar la discusión. Dos colaboradores pueden señalar en algún punto de la mesa mediante la sombra virtual de su brazo, proyectado en la mesa del otro colaborador (inferiores).

utilizada para manipular objetos sobre el mapa.

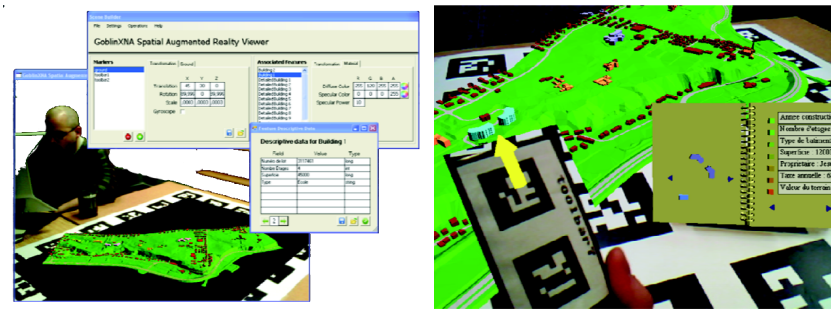


Figura 3.12: Un usuario crea una escena geoespacial mediante una interfaz llamada *Scene Builder* (izquierda) e interactúa con ella mediante una herramienta tangible (derecha).

También han sido propuestos sistemas para modelado virtual. Lau et. al. [42] presentan un sistema de estampado de formas para modelar muebles en 3D a partir de primitivas tangibles. Posteriormente, estos modelos puede ser construidos físicamente, utilizando piezas reales de madera. Lau et al. denominan a su enfoque *Situated Modeling* y utilizan un HMD inmersivo con dos cámaras embebidas, pero este sistema no es móvil. En las Figuras 3.13 y 3.14 se muestran varios ejemplos de muebles creados a partir de las primitivas tangibles, previsualizados en el espacio real donde serían colocados, así como el mueble resultante, fabricado con piezas reales.

Figura 3.13: En *Situated Modeling* el usuario lleva puesto un HMD con una cámara Web y puede observar el entorno y las primitivas reales (izquierda). Los objetos virtuales agregados forman muebles que se pueden visualizar alineados en el entorno real y con el tamaño que tendrá el mueble en la realidad (derecha).

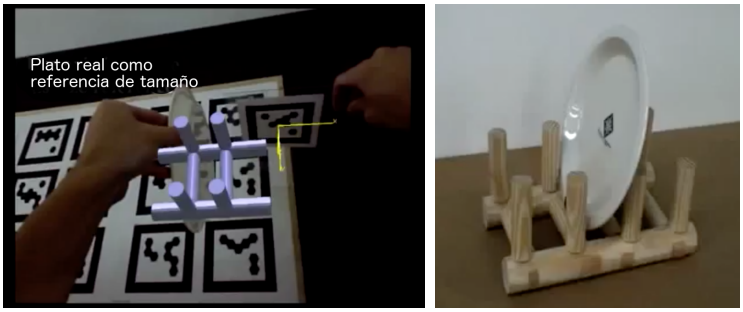


Figura 3.14: El usuario construye un escurridor para platos que se ajusta al tamaño de un plato real (izquierda). El escurridor real construído con piezas de madera reales (derecha).

Wibowo et al. [66] presentan *DressUp*, un sistema para el diseño de vestidos que aplica realidad aumentada a un maniquí y usa dos herramientas tangibles para crear las superficies en 3D que darán forma al vestido virtual: una para trazar las superficies en 3D y la otra para cortarlas. Una vez creado el diseño virtual, el sistema puede generar los patrones de costura que se pueden imprimir para fabricar los vestidos reales en tela. Las herramientas y el maniquí son seguidos por seis cámaras colocadas en la habitación y el modelo 3D del vestido, colocado sobre el maniquí, se despliega en una pantalla. En la Figura 3.15 se muestra el proceso de creación de un vestido.

Un trabajo más reciente es *Second Surface* [39], una aplicación móvil que permite al usuario crear y situar virtualmente contenido digital en el espacio en 3D circundante a ciertos objetos cotidianos, tales como árboles y paredes, tomándolos como marcadores de realidad aumentada. La aplicación también permite al usuario compartir el contenido creado con otros usuarios colocalizados, en tiempo real. En las Figuras 3.16 y 3.17, se muestra un diagrama y fotografías del proceso de creación de un dibujo. Nótese que el contenido creado y compartido en *Second Surface* no puede ser modificado por otros usuarios, por lo tanto no se logra el mismo nivel de edición colaborativa que ofrecen los prototipos creados con el marco de desarrollo ShAREd.

Figura 3.15: *DressUp*: el usuario utiliza dos herramientas tangibles con marcadores para crear la forma de un vestido sobre un maniquí real (izquierda), el modelo virtual del vestido aumentado sobre el maniquí (centro) y el vestido real fabricado a partir del diseño virtual (derecha).

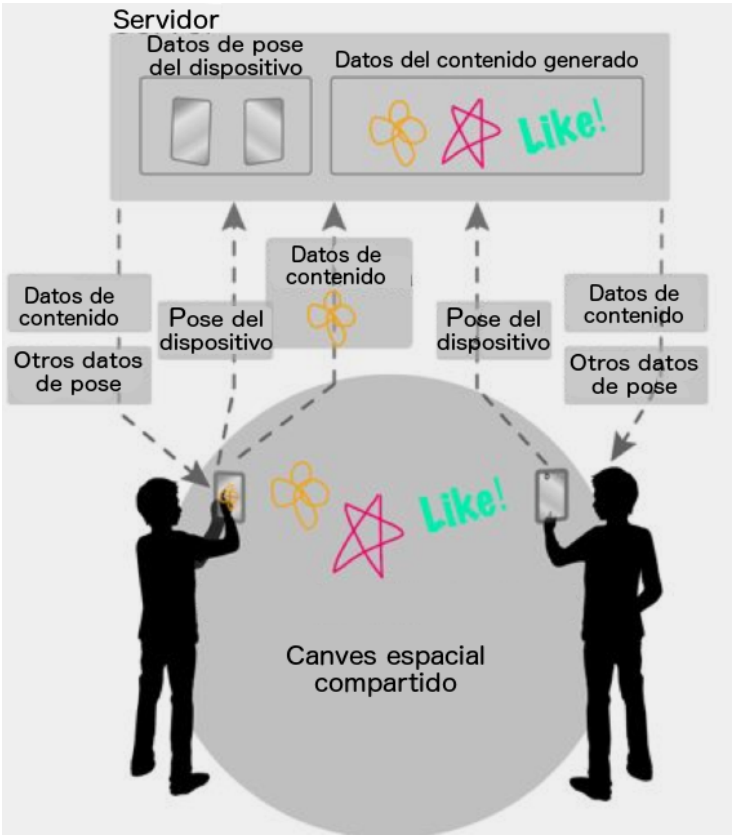


Figura 3.16: Proceso de creación de contenidos digitales en *Second Surface*: dos dispositivos se conectan al servidor, reciben datos de contenido y su respectiva ubicación en el espacio compartido, de acuerdo a la pose de cada uno. Uno de los usuarios sube un dibujo al servidor, mientras el otro usuario recibe y observa la actualización.

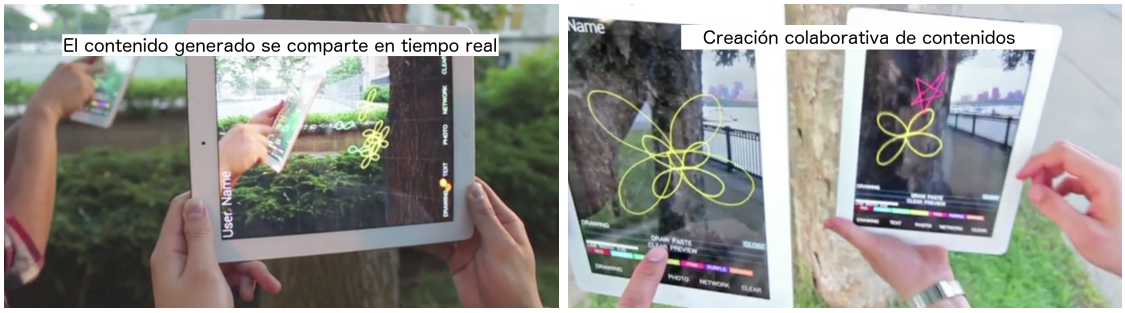


Figura 3.17: Fotografías de los dispositivos durante la ejecución de *Second Surface*. Uno de los usuarios observa el espacio compartido, mientras el otro crea y comparte un dibujo (izquierda). Los dos dispositivos agregan contenidos distintos ubicados en el mismo espacio compartido, lo cual puede intrepretarse como una composición colectiva (derecha).

LandscapAR[13] es una aplicación móvil que utiliza algoritmos de visión por computadora para capturar y hacer el seguimiento de un mapa de alturas, dibujado en una hoja de papel. Primero, el usuario dibuja un conjunto de curvas de nivel, utilizando un marcador negro grueso. Luego, la aplicación captura la imagen del dibujo y, a partir de las curvas, genera la correspondiente representación 3D y la muestra alineada con la hoja de papel, mediante realidad aumentada. El usuario puede tomar instantáneas del terreno desde el ángulo deseado y compartirlas en redes sociales. De esta manera, el usuario puede generar terrenos a partir de sus dibujos, pero los modelos generados no se pueden modificar directamente. Si el usuario desea hacer algún cambio, debe editar físicamente el dibujo, o bien hacer uno nuevo, y la aplicación generará el terreno correspondiente. Así que no se trata de un editor del terreno. En la Figura 3.18 se muestra el proceso de creación de una superficie usando *LanscapAR*.

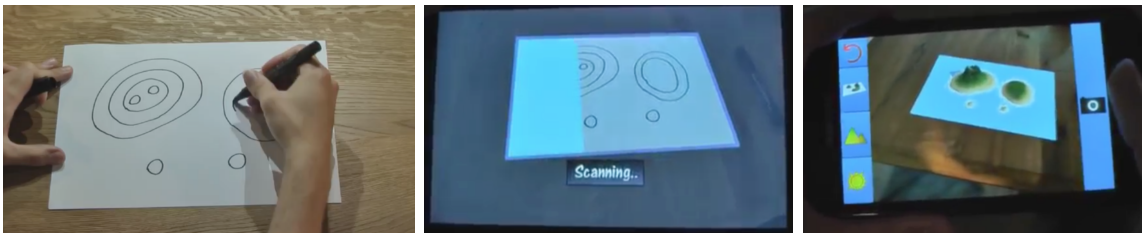


Figura 3.18: Uso de la aplicación *LandscapAR*: el usuario dibuja varias curvas de nivel (izquierda), luego captura el dibujo a través de la aplicación (centro), la cual genera el mapa de elevación y visualiza la superficie, aumentada sobre el dibujo (derecha).

En *Smart Avatars* [4] se propone un sistema de interacción entre el usuario y los objetos reales y virtuales, con base en una narrativa que incremente el interés y la respuesta emocional del usuario al efectuar la interacción. Básicamente, el escenario de este trabajo consiste en insertar un avatar virtual que realice las acciones que el usuario está solicitando del objeto real. De manera que este trabajo tiene la finalidad

[13]https://play.google.com/store/apps/details?id=de.berlin.reality.augmented.landscapar

de investigar las virtudes y obstáculos de dicho escenario.

Para ello, se presenta el prototipo de una lámpara inteligente, llamada *Flexo*, la cual se puede encender y apagar a control remoto desde un iPad via WiFi. Pero en lugar de utilizar el enfoque tradicional de pulsar un botón (u otro control) en el dispositivo e inmediatamente se encienda y apague la lámpara, se trata de involucrar emocionalmente al usuario mediante una narrativa, implementada como una animación digital en la que el avatar actúa y realiza la acción solicitada. Dependiendo de la actuación del avatar virtual, se puede generar dicha respuesta emocional en el usuario y posiblemente le añada interés a la acción.

En la Figura 3.19, se muestra la aplicación en funcionamiento, utilizando la lámpara *Flexo* que, aunque es un dispositivo muy sencillo, sirve como una prueba del concepto de *Smart Avatars*. Amores et al. argumentan que, para el caso de un aparato más complicado, tendrá mayor utilidad agregar una narrativa basada en la intervención de un avatar inteligente, e.g., para proporcionar instrucciones o soporte al usuario, o bien que le ayude a memorizar una serie de pasos complejos, de manera más significativa que si se usara un tutorial tradicional y poco interesante.

Figura 3.19: *Smart Avatars*: lámpara apagada con el avatar aproximándose a ella (izquierda), lámpara encendida por el avatar (centro), vista del dispositivo (derecha).

iAR [38] es un sistema de manipulación de objetos 3D, que utiliza realidad aumentada sobre dispositivos móviles. Implementa un algoritmo de detección y seguimiento visual de puntos de interés sobre un flujo de video de cierta maquinaria compleja con el fin de agregar modelos virtuales, información y animaciones al video, que ayuden al usuario a operar dicha maquinaria. Como se puede observar en la Figura 3.20, es posible insertar a un experto, que se encuentra conectado remotamente, para que guíe al usuario local (el aprendiz) por medio de instrucciones vía voz y la capacidad de apuntar en la escena real, colocando flechas aumentadas que están registradas correctamente sobre la máquina real.

Para el año 2013, surgió un gran interés en el estudio de las interfaces de usuario tangibles (*Tangible User Interfaces*) y el Internet de las cosas (*Internet of Things*), dando origen a proyectos como *Smarter Objects* [34], que consiste en la creación de objetos de tipo *hágalo usted mismo*, con funcionalidades electrónicas y sensores que se conectan a Internet o a otros dispositivos, como un iPad, via WiFi.

De esta manera, es posible comunicarse desde el objeto tangible, conocido como *smart object*, hacia el dispositivo móvil. Incluso es posible actuar mecánicamente sobre dicho *smart object* a través de servos y otros actuadores controlados por el dispositivo

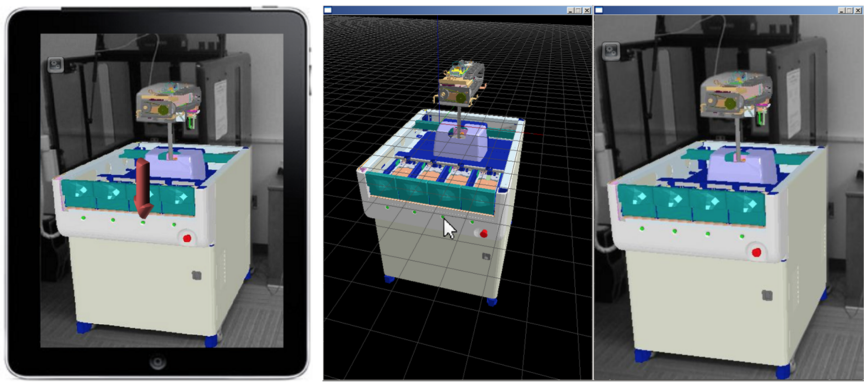


Figura 3.20: Una sesión colaborativa con *iAR*: la vista del aprendiz hacia la máquina a través de un iPad (izquierda) y las vistas del experto remoto en una PC, la de exploración del modelo 3D (centro) y la que tiene el aprendiz desde su iPad (derecha). El experto es consciente de la ubicación y de lo que observa el aprendiz y es capaz de señalar cualquier parte de la máquina colocando una flecha en el espacio 3D.

móvil. En este trabajo, Heun et al. presentan un radio tangible provisto de perillas giratorias para sintonizar la frecuencia y cambiar el volumen, que el usuario puede manipular de manera manual, o bien a través de un iPad (ver Figura 3.21). Si el usuario observa el radio a través del iPad, usando la aplicación de realidad aumentada, se agrega funcionalidad al radio, e.g., se despliega, flotando frente al radio, una lista de reproducción interactiva, en la cual el usuario puede elegir una canción, mediante gestos táctiles en el iPad, y la selección se escucha en el radio tangible. Adicionalmente, el programa le permite al usuario programar el comportamiento del radio, al conectar visualmente, mediante un gesto, una de las perillas del radio con una bocina tangible. Al conectarlos, la canción suena en la bocina de manera inalámbrica.

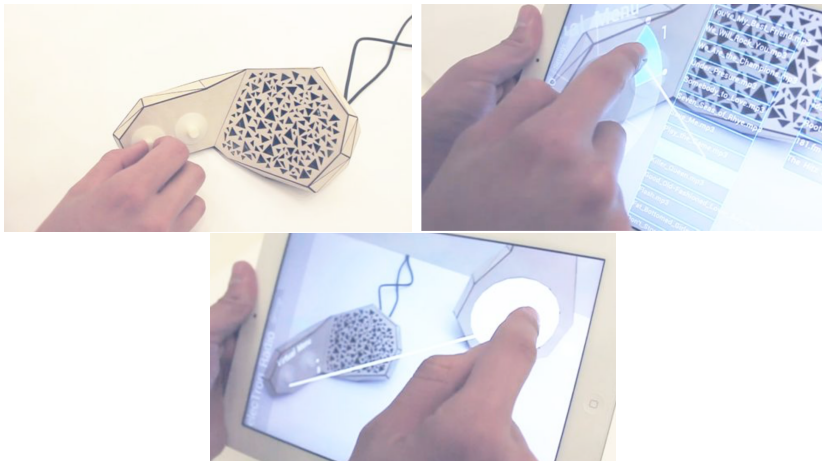


Figura 3.21: Proyecto *Smarter Objects*: radio tangible (arriba izquierda), radio con funcionalidad aumentada a través de un iPad (arriba derecha), el usuario programa un comportamiento virtual entre dos dispositivos conectados al iPad via WiFi (abajo).

El arenero de realidad aumentada desarrollado en *Keck Center for Active Visualization in the Earth Sciences* de la Universidad de California (UC), en Davis [52], es un sistema integrado para crear físicamente modelos topográficos a escala, los cuales son escaneados en tiempo real y utilizados como base para visualizaciones geográficas y simulaciones de fluidos didácticas. El sistema puede ser utilizado en exhibiciones interactivas para museos de ciencias con poca supervisión, como se muestra en la Figura 3.22. Consiste en un arenero tangible que puede ser manipulado físicamente, al cual se le agrega un mapa de elevación a color, curvas de nivel y agua simulada, mediante un sensor de profundidad *Kinect* de Microsoft, un proyector y un poderoso software de simulación.

El prototipo del arenero de realidad aumentada fue diseñado y construído por el especialista Peter Gold, del Departamento de Geología de la UC Davis. El software está basado en el *kit* de desarrollo de realidad virtual *Vrui*[14] y el marco de desarrollo para procesamiento de video *Kinect 3D*[15]. Está disponible para descarga bajo la licencia pública general GNU[16].

Fuente: http://idav.ucdavis.edu/~okreylos/ResDev/SARndbox/

Figura 3.22: Disposición del arenero de realidad aumentada (izquierda), el sensor *Kinect* y el proyector se encuentran suspendidos sobre el cajón de arena. El arenero en funcionamiento (derecha), mostrando una montaña con un lago en su cráter y rodeada de varios lagos.

ShowMe [3] es un sistema colaborativo móvil inmersivo que permite la interacción entre dos usuarios, un aprendiz local y un experto remoto, comunicados a través de audio, video y gestos manuales. En la Figura 3.23, podemos ver al usuario local usando un HMD, provisto de dos cámaras para capturar su vista en estéreo, la cual se le envía al colaborador remoto, quien también utiliza un HMD. En su dispositivo, el experto remoto cuenta con un sensor de profundidad, con el que captura sus manos para ser mostradas en la vista 3D del aprendiz local, en realidad aumentada, lo que le permite al experto apuntar de manera precisa dentro de la escena. Además, es posible mostrarle al aprendiz los movimientos que debe realizar con sus manos para operar el equipo.

Posteriormente, Benavides et al. dan un paso adelante respecto al esquema de *ShowMe*, al utilizar un objeto inteligente (*smart object*), i.e., el radio tangible del

[14]http://idav.ucdavis.edu/~okreylos/ResDev/Vrui
[15]http://idav.ucdavis.edu/~okreylos/ResDev/Kinect
[16]http://idav.ucdavis.edu/~okreylos/ResDev/SARndbox/Download.html

Figura 3.23: Colaboración usando *ShowMe*: el aprendiz local (izquierda),el experto remoto (centro) y las manos del experto aumentadas en la vista estereoscópica del aprendiz (derecha).

proyecto *Smarter Objects*. Así, proponen el sistema *RemotIO* [9], con el cual el experto remoto es capaz de manipular el radio a través de gestos manuales. De esta manera, el experto puede mostrar al aprendiz, no solamente los movimientos y gestos de sus manos, sino también su efecto en el radio, pues éste responde en tiempo real, como se muestra en la Figura 3.24.

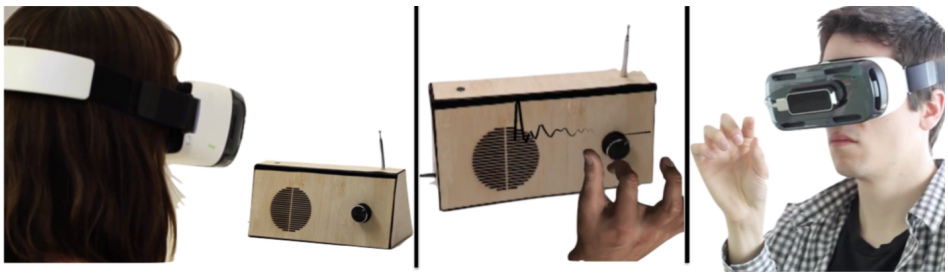


Figura 3.24: Mediante *RemotIO*, un colaborador local observa el radio tangible (izquierda), el experto remoto ejecuta gestos con las manos (derecha), los cuales se envían a través de la red y se muestran en realidad aumentada (centro). Además, los gestos disparan acciones sobre el radio, como girar la perilla del volumen. El colaborador local puede visualizar una representación gráfica de la onda del sonido que se está reproduciendo.

En la Tabla 3.1, se comparan las propuestas previamente descritas. Encontramos que la idea de fusionar las áreas de Realidad Aumentada con Trabajo Colaborativo Asistido por Computadora no es nueva. Sin embargo, se puede ver que los esfuerzos de investigación se han enfocado principalmente en ofrecer soluciones, ya sea para compartir e interactuar con objetos 3D previamente fabricados o, bien para crear y compartir contenido digital en 3D con otros, pero sin permitir que los colaboradores efectúen modificaciones. Prácticamente, no hay otros trabajos reportados que tengan el enfoque propuesto en esta tesis: facilitar el desarrollo de herramientas que den soporte a la creación y edición concurrente de recursos digitales, usando realidad aumentada.

Tabla 3.1: Tabla comparativa del trabajo relacionado

Applicación	**Plataforma**	**Es multi usuario**	**Es colaborativo**	**Es editor**	**Utiliza realidad aumentada**	**Requiere equipo adicional**
Editores 3D	Escritorio	No	No	Sí	No	No
Clara.io	Web	Sí	Sí	Sí	No	No
GitMesh	Escritorio	Sí	Sí	Sí	No	No
MeshHisto	Escritorio	Sí	Sí	Sí	No	No
AR Basquetball	Móvil	No	No	No	Sí	No
AR Invaders	Móvil	No	No	No	Sí	No
Ingress	Móvil	Sí	Sí	No	Sí	No
Pokemon Go	Móvil	Sí	Sí	No	Sí	No
Layar	Móvil	No	No	No	Sí	No
Metaio	Móvil	No	No	No	Sí	No
Wikitude	Móvil	No	No	No	Sí	No
Videoconferencia	Escritorio	Sí	Sí	No	Sí	Sí
Magic Book	Escritorio	Sí	Sí	No	Sí	Sí
AR Pad	Escritorio	Sí	Sí	No	Sí	Sí
Invisible Train	Móvil	Sí	Sí	No	Sí	No
ARprism	Escritorio	Sí	Sí	No	Sí	Sí
CoMaya	Escritorio	Sí	Sí	Sí	No	No
Nam et. al	Escritorio	Sí	Sí	No	Sí	Sí
Planif. urbana	Escritorio	No	No	Sí	Sí	Sí
Shape Stamping	Escritorio	No	No	Sí	Sí	Sí
DressUp	Escritorio	No	No	Sí	Sí	Sí
Second Surface	Móvil	Sí	No	No	Sí	No
LandscapAR	Móvil	No	No	No	Sí	No
Smart Avatars	Escritorio	No	No	No	Sí	Sí
iAR	Escritorio	Sí	Sí	No	Sí	No
Smarter Objects	Escritorio	No	No	No	Sí	Sí
Arenero AR	Escritorio	No	No	Sí	Sí	Sí
Show Me	Móvil	Sí	Sí	No	Sí	Sí
RemotIO	Móvil	Sí	Sí	No	Sí	Sí
Prototipos ShAREd	Móvil	Sí	Sí	Sí	Sí	No

3.5. Discusión

De acuerdo a lo encontrado en la literatura científica, no se han realizado proyectos de edición colaborativa de mallas poligonales, que también utilicen técnicas de realidad aumentada.

Existen proyectos de edición de modelos 3D colaborativos, pero que no utilizan realidad aumentada, sino que conservan el paradigma de interacción con teclado y ratón. Asimismo, casi desde sus orígenes, se ha visto el potencial de la realidad aumentada para ayudar en los procesos colaborativos cara a cara y, en varias ocasiones, se ha intentado crear aplicaciones de autoría y bosquejo de modelos 3D. Sin embargo, aún no ha sido posible obtener productos terminados que cumplan los requerimientos de los editores colaborativos y que también aprovechen la tecnología de realidad aumentada. Al menos no se informa en la literatura acerca de la existencia de un marco de desarrollo especializado en la creación de este tipo de aplicaciones. En este proyecto, nos proponemos avanzar en esta dirección.

Capítulo 4

Marco de desarrollo ShAREd

De acuerdo a Gamma et al. [28]: "Un marco de desarrollo (*framework*) es un conjunto de clases que cooperan entre sí y conforman un diseño reutilizable para construir una clase específica de software". Un marco de desarrollo dicta la arquitectura de las aplicaciones que se pueden construir con él, a diferencia de un *toolkit*, que es un conjunto de clases reutilizables, diseñadas para proporcionar funcionalidad de propósito general. Un *toolkit* debe ser útil para resolver problemas específicos, independientemente de la arquitectura que pudieran tener las aplicaciones que lo utilicen.

En este proyecto nos interesa construir un marco de desarrollo que permita a los desarrolladores crear herramientas colaborativas para bosquejar (*sketching*) modelos tridimensionales, que utilicen realidad aumentada. Al abstraer la arquitectura general para este tipo de aplicaciones, el desarrollador sólo tendrá que enfocarse en la funcionalidad específica para manipular el tipo de modelos que se desea crear con el editor.

A continuación se mencionan algunas consideraciones de diseño que son comunes a los editores que nos ocupan. En cuanto a la funcionalidad de modelado, nos enfocamos en aplicaciones que manipulan modelos 3D, que están construidos a partir de mallas poligonales. Algunos ejemplos de aplicaciones de modelado que se pueden construir utilizando dicha arquitectura son:

- Editores basados en la colocación de piezas sólidas, e.g., construcción basada en apilar bloques, organización de edificios en un entorno urbano, o bien para amueblado de espacios virtuales.
- Editores de superficies, como en el caso de los mapas de elevación digital.
- Editores de mallas poligonales, del tipo de Blender y Maya.

En cuanto al soporte de colaboración, se trata de aplicaciones que permiten a un grupo de usuarios colocalizados, de no más de 15 personas, operar de manera concurrente sobre el modelo, sin restricciones de zona (i.e., de la región en la que puede trabajar cada uno) y sin un esquema de permisos ni de control de concurrencia. Estas restricciones se las dejamos al grupo para que ellos mismos decidan y se organicen para conseguir la meta común, como se menciona en la Sección 2.1.

Respecto a la realidad aumentada, nos inclinamos por la metáfora de la ventana mágica (mencionada en la Sección 2.3.2) haciendo uso del dispositivo móvil, como una ventana a través de la cual se puede observar el entorno real junto con el contenido digital agregado. Asimismo, utilizamos módulos que ejecutan algoritmos de realidad aumentada eficientes en dispositivos móviles y que requieren de un marcador para realizar el seguimiento y registro en tiempo de ejecución.

El marco de desarrollo propuesto consta de varias capas (como muestra la Figura 4.1) para ofrecer la infraestructura necesaria para construir dichos editores, aprovechando las tecnologías emergentes: poderosos dispositivos móviles, dotados de cámaras empotradas, para capturar la escena real; el componente fundamental es un motor de juegos que administra los objetos de la escena 3D y administra las interacciones entre ellos; sobre este, se agregan funcionalidades para el manejo de redes y de gestos táctiles, así como una biblioteca de realidad aumentada, que cuenta con algoritmos de seguimiento y registro eficientes para dispositivos móviles, con ayuda del cual, se efectúa el *render* de todos los elementos tridimensionales del espacio virtual, sobre la imagen tomada del mundo real; luego, con ayuda de las piezas anteriores, implementamos los componentes dedicados a las tareas de modelado 3D, por medio de una interfaz de usuario amigable y a proporcionar consciencia de grupo.

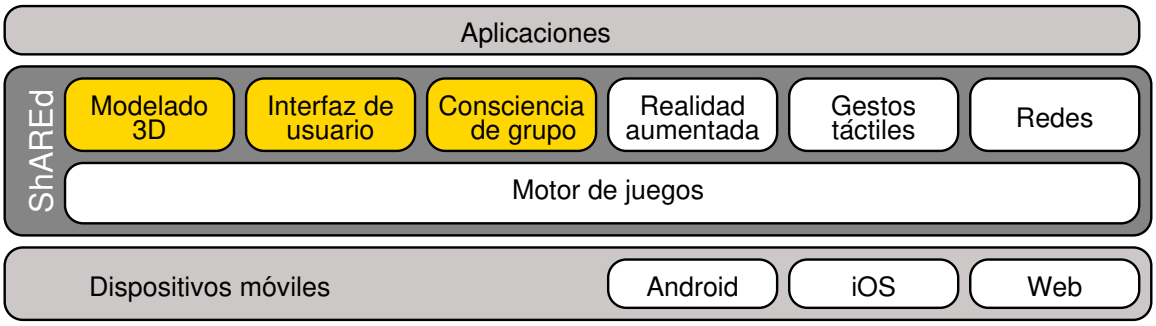


Figura 4.1: Esquema de las capas de ShAREd, los componentes resaltados fueron creados desde cero.

Para lograr esto, proponemos la arquitectura que se muestra en la Figura 4.2, misma que será utilizada para la construcción de editores de los distintos tipos de contenido virtual. En el diagrama se muestran los componentes funcionales y de datos, representados como rectángulos y óvalos, respectivamente. Las flechas continuas representan interacciones entre componentes funcionales: las líneas punteadas significan que el componente funcional maneja o administra al de datos y la flecha punteada indica que el componente funcional utiliza al de datos, i.e., tiene acceso de lectura a los datos, pero no los puede modificar.

En las siguientes secciones, se describen los componentes funcionales de nuestra arquitectura: manejo de la escena 3D (Sección 4.1), *widgets* de interfaz de usuario (Sección 4.2), intercambio de datos (Sección 4.3), modelado 3D (Sección 4.4), consciencia de grupo (Sección 4.5) y modalidades de despliegue (Sección 4.6).

Es posible utilizar esta arquitectura tanto para aplicaciones móviles, en iOS y

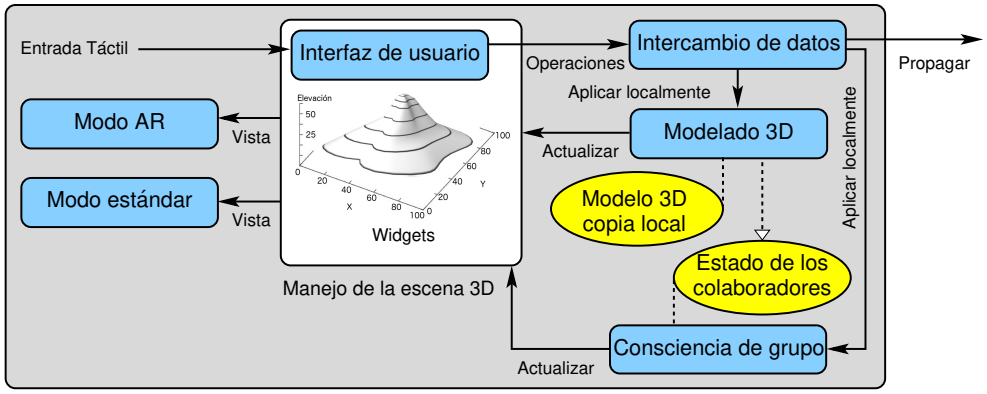


Figura 4.2: Arquitectura horizontal del marco de desarrollo ShAREd, donde se muestran los componentes funcionales y de datos, así como sus interacciones.

Android, como para aplicaciones de escritorio y en navegadores Web. Dado que en los dispositivos móviles la entrada se realiza a través de gestos táctiles (*multi-touch*), para interactuar mediante el ratón es necesario efectuar un simple mapeo de dichos gestos a operaciones de pulsar (*click*) y arrastrar (*drag*).

La arquitectura propuesta fue validada construyendo tres prototipos de editores colaborativos: una aplicación móvil para ayudar en el montaje de exposiciones académicas (Capítulo 7), otra para editar un mapa de elevación digital de manera colaborativa (Capítulo 8) y una herramienta de manipulación de mallas poligonales (Capítulo 9).

4.1. Manejo de la escena tridimensional

El componente encargado de manejar la escena tridimensional constituye la base sobre la cual se erige la aplicación, de manera que los demás componentes colaboran en torno a este. La escena 3D es el escenario en el cual se ubican los elementos que componen la aplicación. Dichos elementos pueden interactuar entre ellos y algunos pueden ser controlados por el usuario. En el presente trabajo, estos elementos se denominan *widgets*[1], tienen una representación gráfica y se les dibuja en pantalla, durante el proceso de *render*, desde el punto de vista de una cámara virtual.

En el marco de desarrollo ShAREd, consideramos cuatro tipos de widgets, de acuerdo a su función en la aplicación: 1) de interfaz de usuario, 2) de datos (o de contenido), 3) de consciencia de grupo y 4) no interactivo.

Como sus nombres lo indican, algunos *widgets* constituyen la interfaz de usuario, otros representan el modelo 3D que se desea manipular y otros muestran el estado de los participantes en la sesión colaborativa. Además, dependiendo del tipo de aplicación, se pueden requerir *widgets* no interactivos, los cuales pueden o no tener alguna

[1]El término *widget* es la contracción de *window gadget*, que se interpreta como un dispositivo o artilugio de ventana. En el contexto de interfaces de usuario se refiere a cada control o elemento que las compone.

función dentro de la aplicación. Estos pueden ser utilizados para proporcionar algún tipo de información pertinente o pueden servir como un marco de referencia dentro de la escena 3D. Incluso pueden utilizarse con fines ornamentales pues, aunque no sean indispensables para la funcionalidad de la aplicación, cumplen un papel importante al mejorar la apariencia y el diseño, incrementando el nivel de aceptación e interés de los usuarios.

El componente que maneja la escena 3D tiene las siguientes responsabilidades: dibujar el contenido de la escena desde el punto de vista de una o varias cámaras virtuales, recibir los eventos de toques (*multi-touch*) dentro de la escena para efectuar el control de los *widgets* pertenecientes al usuario, ejecutar las interacciones entre todos los *widgets*, aunque no estén bajo el control del usuario. De este modo, el comportamiento de una aplicación 3D consiste básicamente en la evolución temporal de la interacción entre los distintos *widgets* en escena. En varios proyectos de realidad vitual y realidad aumentada de la literatura, se suele utilizar un motor de juegos que proporciona dicha funcionalidad y el resto de la aplicación se construye sobre éste.

4.2. Elementos de interfaz de usuario

El paradigma tradicional para diseñar interfaces de usuario es conocido como WIMP (*Windows, Icons, Menus, Pointer*) por estar compuestas de elementos tales como ventanas, íconos y menús, que interactúan con el puntero del ratón. Sin embargo, es frecuente que las aplicaciones 3D no se ajusten completamente a dicho paradigma. Además, en el caso de las aplicaciones móviles, el desarrollador no puede darse el lujo de desperdiciar el espacio disponible en la pantalla con múltiples ventanas o gran cantidad de botones y menús, como en la interfaces convencionales. Este problema se puede mitigar usando menús contextuales, que el usuario puede mostrar y ocultar a voluntad, o cuando el contexto de la aplicación permite determinar cuando el usuario los puede necesitar.

Sin embargo, el uso de realidad aumentada ofrece una manera natural de lidiar con dicho problema, pues se aprovecha al máximo el espacio limitado de la pantalla del dispositivo. Normalmente, las aplicaciones de realidad aumentada se ejecutan en una sola ventana, en modo de pantalla completa sin bordes ni barras de desplazamiento. La pantalla se vuelve el área de trabajo y el contenido de la aplicación se muestra como parte de la escena del mundo real, capturada por la cámara.

En el diseño del marco de desarrollo ShAREd, la interfaz de usuario se ubica dentro de la escena junto con los otros elementos 3D, de manera que el motor de juegos, al efectuar el *render* de la escena, muestra juntos todos los *widgets*, independientemente de su función dentro de la aplicación. Una ventaja de adoptar esta estrategia es que el espacio disponible para colocar los elementos se vuelve potencialmente mayor que el tamaño de la pantalla. Otra ventaja es que dichos elementos siempre están virtualmente disponibles, aún cuando no se dibujan en pantalla en todo momento, sino cuando el usuario apunta la cámara a la región del espacio real en la que se espera que se encuentren. De esta manera, los *widgets* parecen estar siempre ahí en la escena

real, aún cuando no estén en pantalla. Esta estrategia es más intuitiva para el usuario que tener que invocar algún menú oculto, cada vez que deseen realizar interacciones frecuentes.

La interfaz de usuario se diseña dependiendo del tipo de contenido, que se desea manipular, y está compuesta por varios *widgets* de interfaz de usuario, con diversos comportamientos específicos, que operan sobre un *widget* de datos. El usuario puede interactuar con dicho *widgets* mediante toques. En la arquitectura propuesta, los *widgets* de interfaz de usuario se pueden colocar de dos maneras: en el HUD[2] (*Heads-Up Display*) o en la escena 3D junto con los otros *widgets*. Un HUD es un panel transparente que se ubica, en todo momento, justo al frente de la cámara y se mueve junto con ella, de manera que los elementos que se le coloquen son siempre visibles. Frecuentemente, en el HUD se colocan *widgets* bidimensionales, mientras que en la escena se pueden colocar de cualquier tipo, de dos y de tres dimensiones. El marco de desarrollo ShAREd, ofrece a los desarrolladores un conjunto de *widgets* prefabricados, con comportamientos básicos, para construir una interfaz de usuario a la medida a la medida. En el capítulo 6 se describe el diseño de los *widgets* de interfaz de usuario, a detalle.

4.3. Intercambio de datos

Nos estamos enfocando en el entorno colaborativo de tipo cara a cara, de acuerdo a la clasificación de Ellis et al. [24], en la que un grupo de usuarios localizados en el mismo lugar y al mismo tiempo, trabajan juntos en la realización de una tarea colaborativa. En este proyecto, la tarea colaborativa consiste en manipular o bosquejar mallas poligonales compartidas utilizando realidad aumentada. El esquema utilizado para compartir la estructura de datos de la malla es el de replicación, sin mantener la consistencia de datos, como se justifica en la Sección 2.1, pues nos estamos enfocando en la creación y modificación de bosquejos. Cada instancia del editor mantiene una copia local de los datos de la malla y es capaz de aplicar operaciones de modelado sobre ella, a través del *Componente de Modelado*. Las operaciones que desea aplicar al modelo se transmiten a los demás colaboradores mediante mensajes de operación, como se puede observar en el diagrama de la Figura 4.3. El esquema de comunicación utilizado se conoce como cliente-servidor con autoridad (*authoritative client-server*), en el cual cualquiera de las instancias de la aplicación puede adquirir el rol de servidor, durante la sesión colaborativa y los otros participantes se conectan a él como clientes. A dicho servidor lo llamamos coordinador de sesión y es quien administra las conexiones e interacciones entre los clientes, siendo él mismo también un cliente.

En la Figura 4.3, se muestran tres instancias del editor, ejecutándose sobre distintas plataformas. Una de ellas es coordinador de la sesión colaborativa y las demás se conectan a ella mediante canales de comunicación unidireccionales, representados por

[2]El nombre HUD proviene de los cascos y las cabinas de los pilotos militares, en los que se despliega información sin obstruir su visión.

flechas. Si se desea interconectar n instancias del editor en una sesión, se puede usar un esquema como el *peer-to-peer*, en el que se requiere un par de canales unidireccionales entre cada par de instancias, lo que requiere un total de $n(n-1)$ canales de comunicación. Sin embargo, con el esquema cliente-servidor con autoridad, solamente son necesarios $2(n-1)$ canales, reduciendo el tráfico en la red WiFi local y simplificando el proceso de intercambio de datos. De hecho, este esquema es ampliamente utilizado en juegos de rol multijugador masivos en línea (*Massively Multi-Player Online Role-playing Game*), que no dependen de un servidor centralizado.

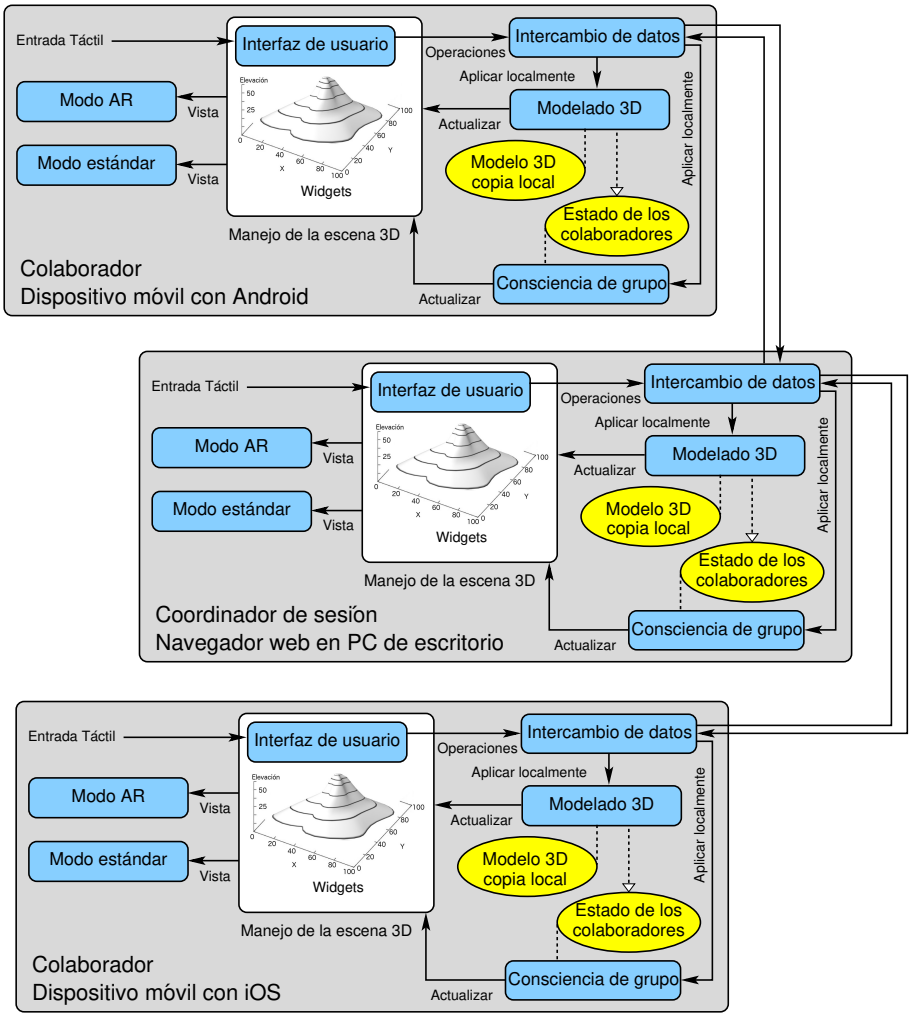


Figura 4.3: Esquema de distribución y comunicación para aplicaciones de modelado colaborativo construidas mediante el marco de desarrollo ShAREd.

Cuando una instancia cliente origina un mensaje de operación, se lo envía al coordinador, el cual lo distribuye a todas las instancias conectadas, incluyendo la que lo generó, y solicita la operación correspondiente para que la aplique, de manera local, el *Componente de Modelado* o el de *Consciencia de Grupo*, según sea el caso. Las otras instancias, al recibir el mensaje desde el coordinador, también ejecutan la

solicitud de operación en sus respectivas copias. De este modo, la instancia que originó la operación no la aplica dos veces. Si el coordinador es quien origina el mensaje, ejecuta la operación localmente y distribuye el mensaje a las demás instancias, que a su vez la ejecutarán.

Dado que los mensajes de operación se envían codificados en una cadena de texto, es posible la comunicación entre instancias que se ejecutan en plataformas diferentes, por ejemplo en la Figura 4.3 se muestra la comunicación entre una instancia móvil con iOS, otra con Android y una más en una PC a través de un navegador Web. Otra ventaja de enviar los mensajes de esta manera, es que resulta más eficiente que si se enviase la estructura de datos completa del modelo 3D para actualizar las copias locales, pues dicha estructura puede llegar a ser de gran tamaño.

4.4. Modelado

Los detalles de implementación del *Componente de Modelado* están fuertemente ligados al tipo de modelo 3D que se desea manipular. Hemos mencionado que el *Componente de Intercambio de Datos* recibe mensajes de operación desde el servidor y luego envía las solicitudes de operación correspondientes al *Componente de Modelado*, para que éste manipule la estructura de datos del modelo 3D que contiene internamente, representado con un óvalo en el diagrama de la arquitectura (Figura 4.2). Además, el *Componente de Modelado* también se encarga de actualizar la representación gráfica del modelo dentro de la escena 3D, i.e., el *widget* del terreno, en tiempo real para proporcionar retroalimentación visual al usuario acerca de lo que está ocurriendo en la sesión. Es importante notar aquí que, en el presente diseño, los prototipos trabajan con una sola malla poligonal a la vez, lo cual se refleja en la arquitectura. Es tarea del programador implementar dicha estructura de datos y las operaciones de modelado mediante las cuales se le puede manipular, de manera que se capture la esencia y el comportamiento del tipo de modelo en cuestión.

En algunas aplicaciones, cada participante puede tener asociado un estado, como se describe en la Sección 4.5. Dependiendo del tipo de modelo, es posible que se necesite conocer el estado del participante particular, que está aplicando una operación, para poderla efectuar en todas las instancias. Para ello, el *Componente de Modelado* tiene acceso para consultar los estados de los participantes conectados, pero no los puede modificar, ni eliminar. Por supuesto, también pueden existir casos en los que no sea necesario almacenar el estado de cada participante.

En capítulos posteriores se explican los detalles referentes al tipo de modelo 3D que se implementa para cada prototipo y las operaciones de edición que les corresponden. Pero en general, las operaciones de edición para modelos 3D van a realizar alguna de las siguientes acciones:

- Crear el modelo base a partir del cual se va a generar un modelo nuevo, así como restaurarlo a su estado original (operación de *reset*), en el caso de que ya exista en escena.

- Seleccionar un conjunto de elementos que componen al modelo, con el fin de manipularlo de alguna manera. Para algunos tipos de modelo, no es necesario efectuar una selección previa a la ejecución de una modificación.

- Modificar un conjunto de elementos, que puede ser el conjunto previamente seleccionado, o bien una parte determinada del modelo 3D.

- Eliminar el conjunto de elementos seleccionados o crear nuevos. Estas operaciones pueden no ser necesarias para algunos tipos de aplicación.

- Guardar el modelo. Este tipo de operación también es opcional y consiste en almacenar los datos del modelo en algún formato útil, para ser usado posteriormente. Dicho formato depende de los propósitos de la aplicación e incluso esta acción puede ser opcional.

- Modificar el estado de un participante concreto, en el caso de que los colaboradores necesiten tener un estado para efectuar operaciones.

Un tipo de operación que es común en los editores monousuaurio y que no aparece en la lista anterior, es la operación deshacer (*undo*) [30]. Como su nombre lo indica, consiste en deshacer el efecto producido por la aplicación de una o más operaciones anteriores. La acción de deshacer el efecto de aplicar la misma operación deshacer, es la operación rehacer (*redo*), en la que se vuelve a aplicar la operación que se había deshecho.

Las operaciones deshacer y rehacer son relativamente fáciles de implementar en un editor monousuario, aunque dependiendo del tipo de documento que se edite, pueden llegar a ser ineficientes o requerir una gran cantidad de recursos, en memoria o en procesamiento. Una manera de implementarlas es llevar una lista de las operaciones de edición, en el orden que se ejecutan y para deshacer una de ellas, se aplica su inverso y se remueve de la lista o se mueve a otra para poder rehacerla. Si ocurre que cierta operación no es invertible, se pueden guardar copias del documento que se está editando, antes y después de la aplicación de dicha operación. Así, cuando se quiera deshacer, sólo se necesita cargar la copia de respaldo. Este enfoque puede llegar a ser muy costoso en memoria, en especial cuando el documento es grande y se permite que sea muy larga la lista de operaciones para deshacer y rehacer.

Estas soluciones pueden aplicarse al caso de las arquitecturas cliente-servidor, siendo este último el que lleva la lista completa de las operaciones aplicadas y es el responsable de deshacer las operaciones en el documento centralizado, para luego propagar los cambios. Sin embargo, en un entorno colaborativo, ya no se trata de una tarea trivial. Si la lista de operaciones se encuentra replicada entre los colaboradores, puede ocurrir que su contenido no sea consistente con las demás instancias de la aplicación, con las que se está colaborando. Entonces es necesario resolver el problema de consistencia de datos, tanto para el documento compartido, como para la lista de operaciones para deshacer. Además, es importante garantizar la consistencia de dicha lista. En el presente marco de desarrollo, se ha decidido no implementar esta

funcionalidad por el momento y concentrar esfuerzos en construir primero los tipos de operación mencionados anteriormente.

4.5. Consciencia de grupo

El *Componente de Consciencia de Grupo* de nuestra arquitectura, proporciona a los participantes tres de los once elementos de consciencia de grupo, definidos por Greenberg y Gutwin [32], a saber: presencia, ubicación y acción realizada.

En algunas aplicaciones, para poder efectuar las operaciones de edición correctamente, es indispensable distinguir cuál participante está realizando cada acción y se debe llevar un registro, en una estructura de datos, que mantenga el respectivo estado para cada participante conectado en la sesión colaborativa. Aunque también habrá casos en los que los colaboradores puedan manipular el modelo sin necesidad de llevar estado alguno. Así pues, en general, no es posible definir *a priori* el contenido del estado de los participantes, pues depende completamente del tipo de operaciones que se requieren.

El comportamiento del *Componente de Consciencia de Grupo* es similar al de *Modelado* y también depende en gran medida del tipo de contenido que se quiere manipular. Es importante notar que, así como el *Componente de Intercambio de Datos* solicita al de *Modelado* que aplique cada operación sobre la estructura del modelo, así también solicita al *Componente de Consciencia de Grupo* que aplique los cambios de estado a las estructuras de los colaboradores. Asimismo, una vez realizados los cambios de estado de los colaboradores, este componente se encarga de reflejar dichos cambios en los *widgets* de consciencia de grupo, en caso de haberlos, los cuales fungen como avatares de los participantes dentro de la escena 3D, en tiempo de ejecución.

En algunas aplicaciones es necesario que los avatares estén ubicados en la misma posición de los colaboradores, para representarlos en el espacio tridimensional, como en el caso de Magic Book y AR Pad, vistos en el Capítulo 3. En otras aplicaciones, puede no ser necesario que haya avatares en la escena, o bien puede ser más útil mostrar alguna información relevante, aumentada sobre la vista real de los otros o ubicada en las coordenadas donde cada uno está efectuando una acción.

4.6. Modos de despliegue en pantalla

En versiones tempranas del marco de desarrollo, el despliegue de la escena 3D se realizaba únicamente mediante realidad aumentada, sobre un marcador impreso, visto a través de la cámara del dispositivo móvil. El uso de realidad aumentada en el proceso de edición y bosquejo de modelos 3D presenta una serie de ventajas, que ya han sido mencionadas en la motivación del proyecto (en el Capítulo 1.2). Sin embargo, durante las pruebas con usuarios de los primeros prototipos de editores colaborativos, se observaron algunas desventajas de usar solo realidad aumentada para visualizar la escena.

Si el dispositivo es muy grande o pesado, puede ser cansado sostenerlo firmemente con una sola mano, mientras se utiliza la otra para efectuar la tarea de edición mediante toques. En el caso en que la mano del usuario comienza a temblar, se dificulta enormemente la tarea de manipular de manera precisa el modelo 3D. Además, al utilizar realidad aumentada, se debe mantener encendida la cámara todo el tiempo, lo que reduce significativamente la duración de la batería. Otro problema consiste en que el usuario queda ligado físicamente al marcador impreso, afectando su movilidad. Para utilizar el marcador de realidad aumentada, los usuarios deben contar con un espacio físico en el que se puedan mover, pero no pueden abandonar el lugar sin llevarse el marcador, o bien si éste se le pierde al usuario, la aplicación resulta inservible.

Considerando lo anterior, es deseable en ocasiones poder utilizar la aplicación sin realidad aumentada, por lo cual en el marco de desarrollo ShAREd se ofrecen dos modalidades de despliegue: una utilizando realidad aumentada y la otra mostrando la escena 3D de la manera estándar. Al arranque de la aplicación, cada usuario puede elegir la modalidad de despliegue que desee.

Capítulo 5

Implementación en el motor de juegos

El marco de desarrollo ShAREd toma como base Unity[1], un motor de juegos ampliamente usado para crear distintos tipos de aplicaciones 2D y 3D multiplataforma, que pueden ejecutarse en computadoras de escritorio y dispositivos móviles. Unity ha sido usado ampliamente en la literatura para construir diversos tipos de aplicaciones estándar, con realidad virtual y aumentada. Ofrece un editor visual que permite realizar prototipos rápidamente, contiene una gran variedad de utilidades y herramientas de desarrollo para implementar prácticamente cualquier tipo de aplicación, además de videojuegos. Otra ventaja de Unity es que es gratuito si se utiliza para desarrollo sin fines de lucro.

Otra opción para el desarrollo de realidad aumentada en dispositivos móviles es el motor Unreal[2], junto con Vuforia o ARToolkit[3]. Desafortunadamente la licencia de Unreal era muy costosa al inicio del presente trabajo, por lo cual se descartó. Recientemente, a partir de la versión 4, se ofrece una licencia gratuita de Unreal, por lo que constituye una buena opción en el futuro.

5.1. Implementación de aplicaciones en Unity

El flujo de trabajo para el proceso de creación de aplicaciones en Unity comienza con un enfoque de creación visual orientado a objetos, seguido de la definición de comportamientos para cada objeto, via *scripting*. Finalmente, se realiza la etapa de generación de la aplicación (*deployment*) en diversas plataformas. Una aplicación 3D creada en Unity se compone de una serie de escenas ligadas entre sí. Una de ellas se indica como inicial y es el punto de entrada de la aplicación. Desde ella el usuario puede acceder a las otras escenas, según se haya diseñado el flujo de la aplicación.

[1]Disponible en `https://unity3d.com/`
[2]Disponible en `https://www.unrealengine.com/`
[3]Disponible at `http://artoolkit.org/`

Para algunas aplicaciones, bastará con una sola escena para cumplir la funcionalidad requerida, como es el caso de los prototipos presentados en el presente trabajo.

El primer paso en la construcción de cada escena 3D consiste en agregar un conjunto de objetos tridimensionales, colocados en coordenadas específicas, con la orientación y tamaño deseados, i.e., tienen una transformación geométrica. Cada escena es un espacio tridimensional dentro del cual los objetos van a interactuar, según sus respectivos comportamientos programados. Así, al ejecutarse la aplicación, se efectúa una simulación, con el motor de juegos, en la que se permite que interactúen los objetos entre sí. Además, algunos de ellos pueden recibir entrada del usuario por teclado, ratón, *joystick* o mediante gestos táctiles, según la plataforma utilizada. De este modo, el comportamiento de la aplicación es producto de la interacción entre dichos objetos con el usuario y entre sí, via sus comportamientos programados.

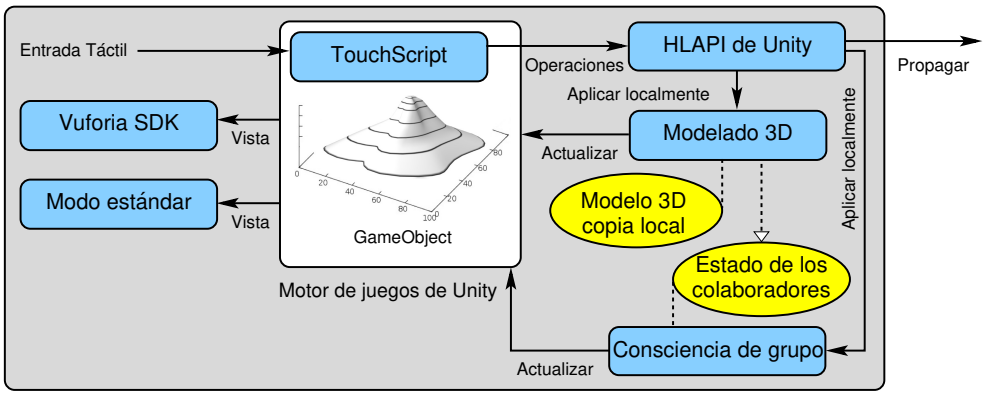


Figura 5.1: Diagrama de la arquitectura propuesta en el marco de desarrollo ShAREd, en el que se muestran las soluciones de software con las que se implementan los componentes funcionales en Unity.

En la Figura 5.1 se muestra el diagrama de la arquitectura propuesta en el marco de desarrollo ShAREd, indicando las herramientas con las que se implementaron los componentes funcionales:

- el manejo de la escena 3D está a cargo de Unity (Sección 5.2),
- los gestos táctiles se procesan por medio de la biblioteca *TouchScript* (Sección 5.3),
- los *widgets* de interfaz de usuario se implementan como instancias de la clase `GameObject` y también como elementos del `Canvas` de Unity (Sección 5.4),
- el intercambio de datos se implementa utilizando la API de manejo de redes *HLAPI* de Unity (Sección 5.5),
- la visualización del contenido en modo estándar es efectuada por Unity, mientras que el modo de realidad aumentada se implementa mediante Vuforia (Sección 5.6).

El modelado 3D y la consciencia de grupo dependen de cada prototipo y se mencionan en sus respectivos capítulos.

5.2. Manejo de la escena 3D

Los objetos 3D son los *actores* de la aplicación y se implementan en Unity como instancias de la clase `GameObject`, la cual contiene las propiedades necesarias para ubicar el objeto dentro de la escena: posición, orientación y escala. En la Figura 5.2 se muestra el diagrama de clases de la configuración de un objeto de este tipo en Unity.

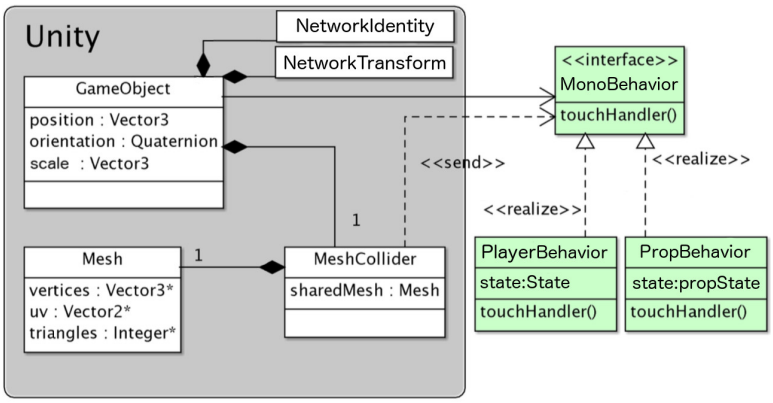


Figura 5.2: Diagrama de clases de una instancia de `GameObject`. Una vez configurado se puede convertir en un objeto prefabricado.

Cada uno de dichos objetos tiene una representación geométrica, definida como una malla poligonal en la clase `Mesh`, que almacena los datos necesarios, tales como vértices, aristas, caras y texturas, para dibujarlo en la pantalla del dispositivo. En caso de que sea necesario calcular colisiones entre este y otros objetos, o que deba responder a eventos táctiles (*multi-touch*), se le puede agregar la clase `MeshCollider`, encargada de detectar tales colisiones y disparar los eventos correspondientes.

A cada objeto se le pueden agregar los comportamientos deseados, programándolos en *scripts* que heredan de la clase `MonoBehavior`, mediante los lenguajes C#, Javascript o Boo[4]. En la Figura 5.2 se muestran dos comportamientos que responden de manera diferente a los eventos táctiles, cada uno con su propio estado, implementado también como una clase. En caso de que el objeto vaya a ser compartido, se le agregan componentes de Unity[5] para el manejo de redes, como se explica en la Sección 5.5.

[4]El lenguaje Boo es una variante de Python.

[5]Es importante distinguir el uso que se le da al término *componente* en esta sección. En el contexto de Unity, se le llama componente a una entidad de software que se asigna a cada objeto para definir su comportamiento. No se trata de los componentes funcionales de la arquitectura propuesta.

En ocasiones se necesita generar varias instancias de algún objeto, previamente configurado con todos sus comportamientos, en tiempo de ejecución. Por ejemplo, cuando en un juego multiusuario se conectan nuevos participantes y se generan sus respectivos avatares dentro de la escena 3D, o cuando se agregan nuevas piezas o bloques de construcción. Para lograr esto, Unity proporciona un mecanismo para definir plantillas de objetos prefabricados, que puedan agregarse de esta manera. A estos objetos prefabricados se le llama *prefabs*.

5.3. Manejo de gestos táctiles

Para efectuar el manejo de eventos y gestos táctiles (*multi-touch*), utilizamos *TouchScript*[6], una biblioteca gratuita para Unity y Flash. Ha sido desarrollada por Valentin Simonov y está inspirada en el sistema de gestos de iOS.

Cuenta con las siguientes características:

- Tiene implementados los gestos más comunes, pero es fácil implementar nuevos gestos personalizados.
- Su API es intuitiva y fácil de usar.
- Soporta diversos métodos de entrada con ratón y superficies táctiles, desde teléfonos inteligentes hasta pantallas y mesas táctiles gigantes.
- Abstrae la lógica de los gestos y toques, independientemente de la plataforma.
- Soporta el reconocimiento de gestos simultáneos dentro de la jerarquía de la escena, inspirado en el manejo de gestos de iOS.
- Soporta la mayoría de las plataformas disponibles.
- Eficiente y optimizado para dispositivos con recursos restringidos.
- Toma en cuenta las diferencias en puntos por pulgada (DPI) entre una superficie táctil muy grande y una pequeña, como un iPad.
- Es gratuito y de código abierto, bajo la licencia MIT.

A continuación se describen los pasos para agregar *TouchScript* a un proyecto y configurar una instancia de `GameObject` para que responda al gesto de pulsar (*tap*).

- Si no se ha hecho ya, se instala *TouchScript* desde la tienda en línea, dentro del IDE de Unity.
- Al crear el nuevo proyecto de Unity se eligen los paquetes que se van a utilizar, en este punto se elige el de *TouchScript*.

[6]Disponible en la tienda en línea de Unity o en: http://touchscript.github.io/

- Se agrega a la escena un objeto vacío (conocido como *Empty*), al cual se le agrega el *script* `TouchManager` de *TouchScript*.
- Se agrega a la escena el objeto que ha de responder al toque y se le agregan los siguientes componentes:
  * un `MeshCollider` de Unity, necesario para calcular las intersecciones de la malla del objeto con el toque,
  * un `TapGesture` de *TouchScript*, el cual tiene varias opciones que se pueden configurar, como el número de *taps* requerido para que el gesto se dispare, i.e., puede reaccionar a cada pulsación individual, o a una doble o triple pulsación y
  * un comportamiento `MonoBehavior`, el cual se modifica como se muestra en el listado siguiente:

```
using UnityEngine;
using System.Collections;
using TouchScript.Gestures;
using System;

public class TapManager : MonoBehaviour {
        // Registra el componente TapGesture
        private void OnEnable(){
                GetComponent<TapGesture> ().Tapped += tappedHandler;
        }
        // Es buena práctica desregistrarlo al terminar
        private void OnDisable(){
                GetComponent<TapGesture> ().Tapped -= tappedHandler;
        }
        // Método asociado para responder al gesto de pulsar
        private void tappedHandler(object sender, EventArgs e){
                // Hacer algo...
                print ('Pulsaste sobre el objeto');
        }
}
```

5.4. Implementación de los *widgets* de interfaz

Cada uno de los *widgets* de interfaz de usuario, presentes en nuestro marco de desarrollo, está implementado en Unity como una instancia de `GameObject`, a la que se le agregan los componentes necesarios para responder a ciertos gestos táctiles, usando *TouchScript*. Al recibir los gestos, se dispara el envío de mensajes de operación hacia los objetos encargados de diseminarlos por la red, los cuales se describen en la Sección 5.5. La forma de los *widgets* se ha creado previamente, mediante la herramienta de modelado Blender. Una vez configurado el *widget* de interfaz, se crea su respectivo objeto *prefab* con el que se genera un paquete de Unity (*unitypackage*). Este paquete

permite al desarrollador reutilizar el *widget* en otros proyectos de Unity. De este modo se construyen los *widgets* de interfaz prefabricados, cuyo diseño y comportamientos se describen en la Sección 6.2.

Sin embargo, aparte de estos *widgets*, también es posible utilizar los elementos de interfaz de usuario nativos de Unity, los cuales usualmente van colocados en el objeto *Canvas*, que es el que proporciona, en Unity, la funcionalidad del HUD, que se describe en la Sección 6.1.

5.5. Intercambio de datos

Es posible crear aplicaciones multiusuario en Unity usando la API de alto nivel de Unity para el manejo de redes, llamada HLAPI (*High Level API*), la cual simplifica la creación de aplicaciones con el esquema de comunicación cliente-servidor con autoridad, mencionado en la Sección 4.3.

> "La API de alto nivel es un sistema para construir capacidades multijugador para juegos de Unity. Está construído sobre el nivel de transporte más bajo de la capa de comunicación en tiempo real (*lower level transport real-time communication layer*) y maneja muchas de las tareas comunes que se requieren en los juegos multijugador. Mientras que la capa de transporte soporta cualquier tipo de topología de red, HLAPI es un sistema cliente-servidor con autoridad; sin embargo, se le permite a uno de los participantes ser un cliente y el servidor al mismo tiempo, de manera que no se requiere de un proceso de servidor dedicado."[7]

HLAPI cuenta con un nuevo conjunto de comandos en red y proporciona servicios útiles para construir juegos multijugador, como:

- manejo de mensajes,
- serialización de propósito general y alto desempeño,
- manejo de objetos distribuidos,
- sincronización de estados,
- diversas clases para el manejo de redes (`Server`, `Client`, `Connection`), entre otros.

Para poder realizar la conexión entre los participantes, se requiere que los clientes puedan encontrar al servidor, para lo cual se tienen dos soluciones: que todos los participantes se encuentren en la misma red local, o bien que el servidor tenga una dirección IP pública, a fin de que las instancias cliente lo puedan encontrar usando el protocolo NAT (*Network Address Translation*) para realizar el proceso *punch-through*,

[7]Fuente: https://docs.unity3d.com/Manual/UNetUsingHLAP.html

que consiste en saltar el servicio que oculta las direcciones IP privadas de los dispositivos detrás de un cortafuegos (*firewall*).

El componente `NetworkManager`, debe estar asociado a alguna instancia de la clase `GameObject` dentro de la escena, la cual suele ser una instancia vacía. Este componente encapsula toda la funcionalidad necesaria para poner un servidor en marcha y abrir una sesión multijugador, a la cual se pueden conectar otras instancias de la aplicación. O en su caso, se puede buscar una sesión abierta por otra instancia de la aplicación y conectarse como cliente.

En una aplicación multiusuario se recomienda no compartir todos los objetos en escena, sino solamente aquellos que sean indispensables para garantizar el correcto funcionamiento de la aplicación. En el marco de desarrollo ShAREd, se comparten el *widget* de datos junto con el modelo 3D y los *widgets* de consciencia de grupo junto con los estados de los participantes, pero no se comparten los *widgets* de interfaz. Esta decisión se debe a que la interfaz de usuario es propiedad del colaborador local, i.e., sólo este la puede manipular y ningún otro necesita conocerla.

Dentro de HLAPI, cada instancia de `GameObject`, que se necesite compartir, se encarga de transmitir su propia posición, orientación y escala a las otras instancias conectadas, usando los componentes `NetworkIdentity` y `NetworkTransform`.

En el caso de aquellos widgets que no se mueven en el espacio 3D o que poseen otro tipo de información que se desea compartir, las modificaciones se realizan mediante llamadas remotas a través de la red, conocidas como RPC (*Remote Procedure Call*). En el sistema HLAPI se tienen dos tipos de RPCs: *Commands*, que son llamados desde un cliente y ejecutados en el servidor; y *ClientRpc*, que son llamados en el servidor y ejecutados en los clientes.

Cuando el usuario interactúa con los *widgets* de interfaz de usuario, las operaciones de edición y de cambio de estado son disparadas mediante llamadas *RPC*, distribuyéndolas a los demás participantes, para luego ser aplicadas localmente en la estructura de datos local del modelo 3D. Además, se actualiza la malla del *widget* de datos para que se visualice el cambio de su forma geométrica en tiempo real.

5.6. Modos de visualización

Hemos mencionado en el capítulo anterior que el marco de desarrollo ShAREd ofrece dos modalidades para la visualización de la escena: estándar y usando realidad aumentada. La visualización estándar está a cargo de Unity mediante las bibliotecas de *render* que comúnmente se encuentran en los dispositivos, e.g. OpenGL en sus distintas versiones. Para visualizar la escena de este modo, se define al menos una cámara virtual dentro de la escena 3D y se proyectan las mallas de las instancias de `GameObject` sobre ésta. Cada malla contiene una serie de arreglos que OpenGL necesita para el proceso de *render*, se trata principalmente de coordenadas e índices para definir los vértices, aristas y caras de la malla. El proceso de *render* es transparente para el desarrollador, ya que este simplemente coloca los objetos dentro de la escena y el motor de juegos se encarga de mostrarlos en pantalla.

Para agregar las capacidades de despliegue usando realidad aumentada, utilizamos la biblioteca de desarrollo Vuforia, diseñada para utilizarse en diversas plataformas, tales como iOS, Android y motores de juegos, como Unity y Unreal. Vuforia ofrece una licencia gratuita para cualquiera que desee experimentar y desarrollar sin fines de lucro. Otro SDK para el desarrollo de aplicaciones de realidad aumentada es String[8]. Sin embargo, las imágenes que utiliza como marcadores deben estar enmarcadas con un borde negro, mientras que Vuforia no tiene este tipo de restricciones para el marcador. Además, su costo era elevado y actualmente se encuentra descontinuado.

Vuforia utiliza algoritmos de Visión por Computadora y Aprendizaje Maquinal eficientes para efectuar el reconocimiento y seguimiento de marcadores (imágenes arbitrarias previamente aprendidas) sobre los cuales puede agregar diversos contenidos digitales. En la Sección 2.3.4, se mencionaron los algoritmos que utiliza Vuforia y se describió el proceso de seguimiento y registro para el uso de realidad aumentada. Vuforia también se encarga de obtener las imágenes del video capturado por la cámara del dispositivo y de mostrarlas como textura en un plano detrás de la escena 3D. De esta manera, se logra el efecto de realidad aumentada.

Para agregar la funcionalidad de Vuforia a un proyecto de Unity se hace lo siguiente:

- Si no se ha hecho ya, se descarga el paquete para Unity desde el sitio de Vuforia y se copia en la carpeta *Standard Assets* de la instalación de Unity.

- Al crear un nuevo proyecto de Unity, se eligen los paquetes que se van a utilizar; en este punto se elige el de Vuforia.

- En el sitio de Vuforia, previo registro gratuito, es necesario generar la base de datos para las imágenes que se desean utilizar como marcadores (llamados *targets* en Vuforia). Se suben los archivos en JPG o PNG para que el sitio genere el paquete correspondiente (en forma de *unitypackage*), que se instala en el proyecto en curso, al dar doble click. Puede haber varios marcadores en la misma base de datos.

- Se agrega a la escena una instancia del *prefab* de la cámara configurada para realidad aumentada, llamada *ARCamera*, localizada en la carpeta *Assets/Vuforia/Prefabs*. En el panel de propiedades de *ARCamera*, se activan las opciones para cargar y activar la base de datos de los marcadores (creada en el punto anterior) en el componente `DatabaseLoadBehavior`.

- Se agrega a la escena una instancia del *prefab* del marcador, llamado *ImageTarget*, ubicado en la misma carpeta del punto anterior. En el panel de propiedades de *ImageTarget*, en el componente `ImageTargetBehavior`, se selecciona la base datos y el marcador elegido para esta instancia de *ImageTarget*.

[8] `http://string.co/`

- Las instancias de `GameObjects`, que se desea visualizar en realidad aumentada, se colocan en la escena, sobre la instancia de *ImageTarget* y emparentados dentro de ésta.

Hemos implementado dos maneras de ofrecer al usuario la opción de elegir la modalidad de visualización. La primera es utilizando varias escenas en la aplicación. En una de las escenas se tiene el contenido de la aplicación, dispuesto para visualizarse en la modalidad de realidad aumentada, como se indicó anteriormente, mediante los objetos prefabricados (*prefabs*) de Vuforia. En una escena diferente, se coloca una copia del mismo contenido, pero sin las instancias de *ARCamera* ni de *ImageTarget*. En su lugar, se agrega al menos una cámara estándar de Unity, colocada de manera conveniente para abarcar el contenido de la esena. Finalmente, en una tercera escena, se crea un elemento *Canvas*, al que se agregan dos botones: uno para cargar la escena estándar y el otro para cargar la escena de realidad aumentada.

La segunda manera de cambiar la modalidad de la aplicación cosiste en utilizar una sola escena, pero colocando dos cámaras: una estándar y una de Vuforia para realidad aumentada; de modo que se pueda elegir cuál de las dos cámaras activar en tiempo de ejecución.

Se puede agregar un *widget* de interfaz de usuario para activar o desactivar el modo de realidad aumentada, como un botón o una casilla de verificación (*checkbox*), con un *script* que cambie la propiedad *enabled* de ambas cámaras, activando una y desactivando la otra, y viceversa. También puede ser útil para indicar cuál cámara está activa, a los objetos que necesiten usarla. Tal es el caso de los elementos *Canvas*, que se encuentran anclados a una cámara. Hacerlo de esta manera tiene las siguientes ventajas respecto al enfoque anterior:

- Se diseña la escena de la aplicación una sola vez, en lugar de tener los elementos repetidos en dos escenas.

- En caso de que sea necesario hacer modificaciones a la escena, es preferible tener una sola en lugar de dos, pues así no se tienen que repetir las modificaciones en ambas copias, lo cual es propenso a errores.

- El usuario puede cambiar la modalidad en tiempo de ejecución cuantas veces quiera, lo cual no es posible en el enfoque anterior, pues si el usuario se cambiara de escena, perdería el trabajo realizado.

La desventaja que identificamos para este enfoque es que la cámara física del dispositivo permanece encendida, independientemente de cuál cámara virtual se encuentra activa en la escena. En la solución con escenas distintas, la cámara física no se enciende en la modalidad estándar, ahorrando así batería.

Capítulo 6

Patrones de diseño en la interfaz de usuario

Como mencionan Hoober y Berkman [35], "los patrones de diseño son simplemente buenas prácticas bien definidas e investigadas, pero en las cuales se deben respetar los principios del buen diseño y siempre se debe considerar al usuario en primer lugar". En este capítulo se describe el diseño de los diversos elementos que proponemos para la creación de editores colaborativos de modelos 3D con realidad aumentada. Nos proponemos que dichos elementos, sean fáciles de usar para los usuarios finales novatos, y que sean útiles y adaptables por un desarrollador que posea los conocimientos técnicos básicos de la plataforma de desarrollo utilizada en esta tesis, i.e., Unity.

En el arte del diseño de interfaces de usuario, actualmente ocurre una gran proliferación de dialectos, i.e., tipos o estilos de interfaz reconocibles, cada uno con su propio vocabulario de objetos, acciones y apariencia visual [62]. Por ejemplo, en el caso de los editores de texto, se sigue un esquema con la apariencia y los elementos de interfaz de usuario a los que ya estamos familiarizados (ver parte izquierda de la Figura 6.1), mientras que en un editor 3D (ver parte derecha de la Figura 6.1) el esquema es distinto. En este capítulo, hacemos uso de los elementos de diseño familiares para los usuarios y los adaptamos a la tarea de modelado 3D dentro de un entorno tridimensional, con la finalidad de obtener un catálogo de componentes de software, que se puedan reutilizar en contextos diferentes.

En el marco de desarrollo ShAREd, estamos experimentando con la idea de tratar todos los objetos del editor, tanto los elementos de la interfaz de usuario como los de contenido, como *widgets* con comportamientos diversos, que interactúan entre sí. Los *widgets* se colocan, ya sea en la escena 3D o en un panel especial para elementos de la interfaz de usuario, llamado HUD por sus siglas en inglés *Heads-Up Display* (Sección 6.1). Tenemos tres tipos de *widgets* interactivos en nuestro marco de desarrollo, los cuales se describen a continuación: *widgets* de interfaz de usuario (Sección 6.2), *widgets* de datos (Sección 6.3) y *widgets* de consciencia de grupo (Sección 6.4).

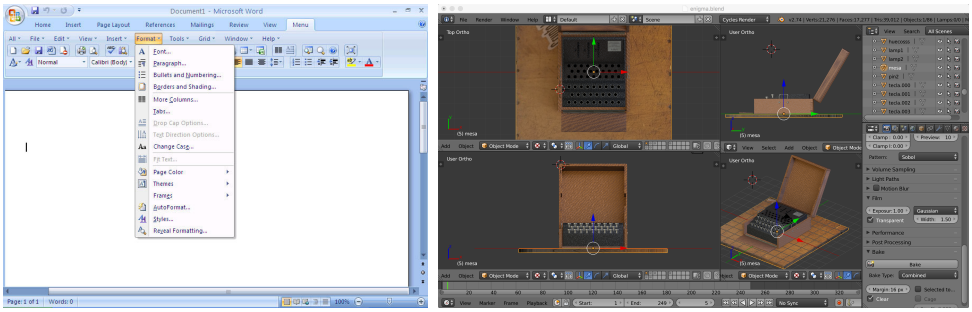


Figura 6.1: Ejemplos de interfaces de usuario para un editor de texto (Word) y para un editor 3D (Blender).

6.1. Diseño del *Heads-Up Display*

Para la construcción de aplicaciones de edición con realidad aumentada, proponemos colocar los *widgets* de la interfaz de usuario dentro de la escena 3D, junto con los *widgets* del contenido y de los avatares de los colaboradores. Con base en las experiencias previas con usuarios, en algunas ocasiones resulta cómodo y conveniente colocar algunos elementos de la interfaz sobre el plano de la pantalla, en un HUD[1]. Sin embargo, existen casos en los que tiene más sentido colocar los elementos aumentados sobre la escena 3D. En el HUD se muestran los elementos de interfaz que son visualizados en el plano de la cámara, ubicados sobre la pantalla del dispositivo, a la manera del HUD de la cabina de un avión.

Dado que nos enfocamos en aplicaciones de realidad aumentada en dispositivos móviles, nos interesa aprovechar al máximo el área disponible de la pantalla. Para ello, se ha diseñado la disposición de elementos como se muestra en la Figura 6.2. Tenemos dos tipos de menús: el panel lateral desplegable y la paleta circular desplegable. Dichos menús se encuentran normalmente ocultos, pero se da la pista visual al usuario de que puede hacer un toque en el elemento para su despliegue, o bien que se le puede arrastrar hacia el interior de la pantalla para revelar su contenido. Una estrategia común, en algunas aplicaciones móviles, es mostrar los menús desplegables abiertos la primera vez que se ejecuta la aplicación, con el fin de informar al usuario de su existencia, y posteriormente éste lo puede ocultar.

La ubicación en pantalla y la dirección de ocultamiento se pueden controlar via código. Ambos menús es posible colocarlos al lado derecho o al lado izquierdo. Por ejemplo, cuando se tiene en cuenta a los usuarios zurdos o diestros, la ubicación se puede modificar de acuerdo a las preferencias del usuario. El diseño de la ventana HUD se puede adaptar a diferentes orientaciones del dispositivo, como se muestra en la Figura 6.3.

[1]El nombre *Heads-Up Display* se puede traducir como *despliegue de avisos o notificaciones* y se refiere al hecho de mostrar información frente a la cara del usuario.

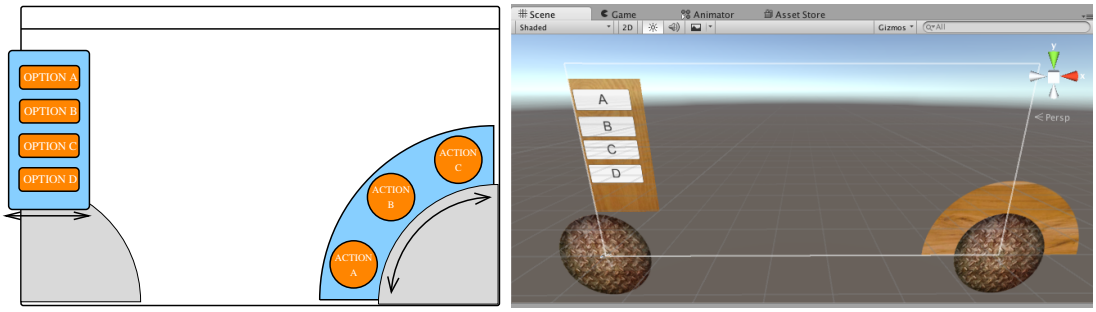


Figura 6.2: Diagrama de la ventana HUD propuesta (izquierda) e implementación en Unity (derecha).

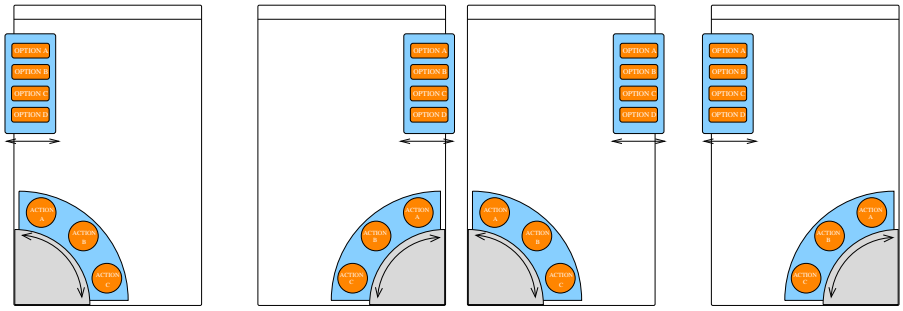


Figura 6.3: Diversas disposiciones de los elementos de la ventana HUD en orientación vertical.

En el recuadro superior de la Figura 6.2, se muestra el área donde aparece un mensaje de texto que informa al usuario acerca de los diversos aspectos de la ejecución de la aplicación, o que simplemente despliegue datos de interés. En la Figura 6.2 se muestra el menú lateral a la izquierda y la paleta circular a la derecha. Ambos menús se pueden poblar con botones, barras de desplazamiento, etiquetas de texto, entre otros. En el menú lateral, se pueden colocar opciones de configuración variadas, de manera que es posible abrirlo, manipular varios de los elementos que contiene y finalmente cerrarlo. La paleta circular se ha diseñado como un selector de opciones, con un comportamiento diferente al del menú lateral. Su tamaño es menor y puede contener botones de los cuales se puede elegir uno. Entonces, el usuario lo abre y selecciona una de las opciones, e.g., un color de una paleta de colores. Tras efectuar la selección, la paleta se cierra automáticamente. Con esto se agiliza el proceso de selección en este tipo de menú y se reduce el número de toques necesario para cambiar una opción de manera frecuente. Se puede implementar también un menú contextual, colocando un pánel que ocupe la parte central de la pantalla y sea llenado con los elementos que dependan del contexto o el objeto que lo ha llamado. Una vez utilizado el menú, se puede ocultar nuevamente.

La ubicación y apariencia de estos menús es distinta con el fin de mostrar al usuario que tienen comportamientos y contenidos distintos, y así podrá saber qué esperar de un cierto menú cuando lo invoque.

En la Figura 6.2 se muestran pequeñas regiones circulares en las esquinas inferiores de la pantalla. Estas indican regiones para sujetar el dispositivo, sin que se produzca toque alguno. Funcionan interceptando los toques efectuados sobre ellos, para que no caigan accidentalmente en el contenido de la escena 3D que se puede encontrar debajo. La idea de colocar estos manipuladores (*handlers*) surge al observar a los usuarios de nuestro primer prototipo. Frecuentemente, los usuarios se cansan de sujetar el dispositivo, cuidando de no tocar la pantalla táctil, y suelen sujetarlo con el pulgar de una u otra mano, cerca de las orillas, generando toques accidentales sobre el contenido virtual. Mediante los manipuladores, se invita al usuario a sujetar el dispositivo con una o ambas manos, en estos puntos, haciendo la aplicación más cómoda de usar, pues el usuario se cansa menos. Además, sujetando el dispositivo de esta manera, la paleta circular es más fácil de manipular, a la vez que se reducen los toques accidentales en las regiones activas de la pantalla.

6.2. *Widgets* de interfaz de usuario

Los *widgets* de interfaz de usuario componen la interfaz de usuario, pero no tienen interacción directa con la estructura de datos del modelo 3D. Además, estos *widgets* no se comparten con los colaboradores, sino que pertenecen únicamente al usuario local. El usuario interactúa con cada *widget* de interfaz mediante gestos táctiles en la pantalla de su dispositivo, disparando eventos de operación que se envían directamente al *Componente de Intercambio de Datos*, para su propagación a través de la red. Para ayudar a que los desarrolladores utilicen el marco de desarrollo ShAREd para crear la interfaz de usuario de sus aplicaciones, hemos diseñado un conjunto de *widgets* reutilizables, los cuales se describen a continuación.

Figura 6.4: Estados básicos de un botón: inactivo, activo y presionado.

6.2.1. Botón

En una interfaz de usuario, se puede utilizar una imagen o una malla 3D como botón, el cual responde a gestos tales como: pulsar (*tap*), presionar (*press*) y liberar (*release*). Al pulsar un botón se dispara alguna acción o un llamado a función. También se espera que el botón muestre alguna retroalimentación visual que indique al usuario que la acción se ha disparado con éxito, para ello se agregan estados al botón. Los estados básicos del botón (inactivo, activo y presionado) se ilustran en la Figura 6.4 y se expresan mediante un cambio de color y no de forma.

En la Figura 6.5, se presenta el diagrama de cambios de estado del botón. Si no se implementan cambios de estado en el botón, el usuario tendrá la impresión de un mal funcionamiento de la aplicación, sentirá que no responde, pues simplemente no tendrá manera de darse cuenta de que la acción ya está en curso. Esto suele ocasionar que los usuarios impacientes lo pulsen repetidas veces, causando un comportamiento inesperado de la aplicación.

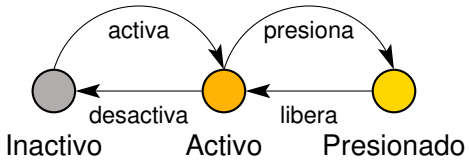


Figura 6.5: Diagrama de cambios de estado de un botón.

También se implementó un prototipo de botón tridimensional, que se muestra en la Figura 6.6, el cual solamente tiene los dos estados necesarios para experimentar con cuatro tipos de comportamiento: *disparador*, *interruptor*, *pulsador* y *disparador en dos fases*.

Figura 6.6: Estados del botón tridimensional: activo (izquierda) y presionado (derecha).

Disparador (trigger): al pulsar sobre un botón en modo disparador, se ejecuta una acción una sola vez por toque. En este contexto, la acción de pulsar el botón consiste en presionarlo y luego liberarlo, ya sea lentamente (*press-release*) o rápidamente (*tap*). Si se pulsa repetidas veces, dicha acción se dispara esa misma cantidad de veces. Es útil en casos como un contador de pulsaciones o un teclado en pantalla, en el cual si se presiona n veces la tecla de un caracter, éste se escribe n veces en la región de texto. Este tipo de botón no puede responder a pulsaciones múltiples (*multitap*), sino que los reconoce como varias pulsaciones individuales consecutivas.

Interruptor (switch): al pulsar el botón en modo interruptor, su estado cambia de activo a presionado o viceversa, y permanece en dicho estado hasta que se pulsa otra vez. Como en el caso anterior, nos referimos a eventos de pulsar y liberar (*press-release*) y pulso rápido (*tap*). El objeto controlado por este botón ejecuta la acción que le corresponde, en tanto éste se encuentre en estado presionado. Este comportamiento es útil, por ejemplo, para simular una luz que se enciende y se apaga o para reproducir la funcionalidad de una casilla de verificación (*checkbox*), la cual se muestra en la Figura 6.7.

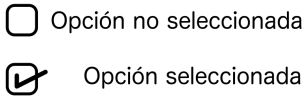


Figura 6.7: Una casilla de verificación permite activar o desactivar la selección de un elemento.

Pulsador (push): en este modo, la acción asociada al botón se ejecuta mientras el usuario lo presiona y se interrumpe en cuanto lo suelta. En esta modalidad se puede simular un pedal de acelerador de un auto, el cual acelera mientras se pulsa el botón y deja de hacerlo al soltarlo.

Disparador en dos fases: en esta modalidad, al presionar el botón se ejecuta una acción, mientras que al liberarlo, se ejecuta una acción diferente. Esta modalidad es útil en casos como las teclas de la *máquina Enigma* que, al ser pulsadas, efectúan el giro de un rotor y, al ser liberadas, realizan la operación de cifrado. Conviene verificar si las dos tareas son dependientes, para asegurar que se ejecuten en el orden correcto; primero se ejecuta la que corresponde a presionar y luego la de soltar o liberar el botón.

Los cuatro comportamientos anteriores se pueden implementar en un botón con los estados activo y presionado. Además, es posible implementar un mecanismo para activar y desactivar el botón, así como para resaltarlo, e.g., haciéndolo parpadear. Para utilizar el botón desde los *scripts* de Unity, definimos la siguiente API:

```
void setBehavior(int i)
void onPress(Event e)
void onRelease(Event e)
void setGlow (bool b)
void setBlink (bool b)
```

Con el método `setBehavior()` se establece el comportamiento de un botón, de entre los cuatro mencionados anteriormente, `onPress()` y `onRelease()` disparan las acciones, dependiendo del comportamiento elegido, `setGlow()` sirve para resaltar el botón, incrementando el brillo de su color y `setBlink()` lo pone a parpadear.

6.2.2. Panel de botones

Existen casos en los que tiene sentido tener un conjunto de botones relacionados entre sí. En nuestra propuesta, establecemos una asociación visual entre un conjunto de botones relacionados, al colocarlos dentro de un contenedor con representación gráfica. En la Figura 6.8 se muestra un diagrama de un panel de botones.

Hemos orientado el diseño de este panel de botones a situaciones en las que se presenta al usuario un conjunto de opciones, de las cuales puede elegir una cantidad limitada. Para este fin, el panel contiene una cantidad de n botones del tipo interruptor para representar cada opción elegible. El panel tiene dos comportamientos de selección: elegir una de las n opciones disponibles y elegir m de las n opciones.

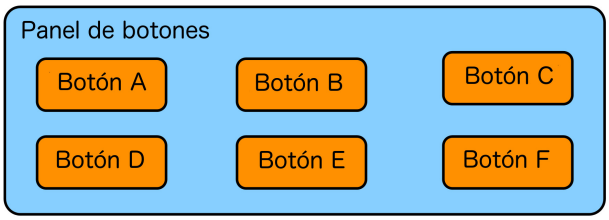


Figura 6.8: El panel de botones es útil para ofrecer al usuario una serie de opciones relacionadas.

Elegir una opción: En esta modalidad, todos los botones se encuentran en estado activo y ninguna opción está seleccionada. Cuando el usuario pulsa un botón, éste pasa al estado presionado y se selecciona esa opción. Si el usuario presiona algún otro botón, se selecciona este y se deselecciona el anterior. Si vuelve a pulsar el botón que ya se encuentra presionado, no se deselecciona. La única manera de deseleccionar una cierta opción es seleccionar otra distinta. También es posible inicializar el panel para que alguna de las opciones se encuentre seleccionada por defecto.

Elegir m de n opciones: En esta modalidad, es posible seleccionar m botones, con $m \leq n$. El usuario pulsa en los botones mientras un contador registra el número de opciones seleccionadas. Si se alcanza el valor m, ya no es posible seleccionar otra opción, hasta que una de las anteriores sea deseleccionada. Para deseleccionar la opción correspondiente a un botón, que previamente ha sido pulsado y se encuentra en estado presionado, se pulsa nuevamente y éste vuelve al estado activo, decrementando en uno el contador de opciones seleccionadas. En caso de que $m = n$, es posible seleccionar todas las opciones, mientras que el caso $m = 1$ es similar al comportamiento para una sola opción, excepto que para cambiar la selección, es necesario deshacer la selección previa antes de pulsar en la nueva opción.

Dado que el panel de botones utiliza instancias del *widget* de botón, se utiliza su misma API para disparar alguna acción al ser pulsado. Además, el panel cuenta con un método `setBehavior()`, para seleccionar una de las dos modalidades anteriores.

6.2.3. *Sprite* de imagen

De acuerdo al diccionario Oxford, un *sprite* es un gráfico de computadora, que se puede mover a través de la pantalla o ser manipulado de alguna otra manera, como una entidad individual, sin modificar sus componentes, i.e., sus pixeles. En este caso, el *sprite* de imagen (*ImageSprite*) consta de una imagen a la que agregamos algún tipo de comportamiento. Se puede ver como un botón 2D, pero que responde a los gestos de arrastrar (*drag*), pellizcar (*pinch*), girar (*rotate*) y pulsación múltiple, además de la pulsación simple.

Mediante estos gestos, se tiene la oportunidad de implementar comportamientos diversos para el *sprite*, como puede ser mostrar información adicional, consejos (*tool-*

tips), un menú de opciones o un panel de propiedades, al pulsarlo múltiples veces o hacer un gesto deslizante, conocido como *swipe*. Se puede también cambiar su textura o comportamiento, en respuesta a gestos distintos. Con este tipo de *sprite* es posible crear aplicaciones para visualizar y ordenar fotografías o escenas de un guión gráfico (*storyboard*), así como implementar juegos de cartas o memoramas. Un *sprite* se puede desplazar, girar y redimensionar, mediante gestos táctiles que ya son familiares para la mayoría de los usuarios.

La API para controlar un *sprite* de imagen es la siguiente:

```
void onPress(Event e)
void onRelease(Event e)
void onDrag(Event e)
void onTransform(Event e)
void setTexture(Image im)
```

Por medio de los métodos `onPress()` y `onRelease()`, un *sprite* responde como un botón. Los métodos `onDrag()` y `onTransform()` permiten que el usuario aplique los gestos para manipular el *sprite* y `setTexture()` sirve para cambiar su textura.

6.2.4. Carrusel de *sprites*

Consiste en una especie de menú desplegable cuyos elementos son *sprites*, pero con la particularidad de que puede contener muchos más elementos que lo que caben en pantalla, debido a que el contenido se puede deslizar con un gesto de arrastre. Como se ilustra en el diagrama de la Figura 6.9, la tira de *sprites* se puede arrastrar con un gesto táctil, haciendo aparecer los elementos ocultos fuera de la pantalla. Es útil, por ejemplo, para seleccionar fácilmente de entre un conjunto grande de fotografías. El carrusel se ubica en la parte inferior de la ventana HUD y los *sprites* aparecen alineados de manera horizontal. Para una mejor apariencia, los *sprites* deben ser del mismo tamaño, y tener una separación uniforme entre ellos. El conjunto de imágenes a mostrar en los *sprites* se proporciona en un arreglo. Al arrastrar la zona del carrusel, los *sprites* se desplazan de manera horizontal y al terminar el gesto, se alinea al centro el *sprite* más cercano.

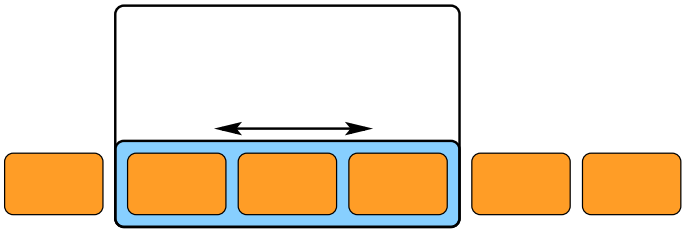


Figura 6.9: Diagrama del carrusel de *sprites*: el panel del carrusel es desplegable y la tira de *sprites* se puede arrastrar para acceder a los que quedan fuera de la pantalla. Cada uno dispara un evento distinto al ser pulsado.

El carrusel utiliza instancias de *Sprite* y cada cual responde al gesto de pulsar, de modo que al elegir uno de ellos se dispara la acción correspondiente.

La API para utilizar un carrusel desde un *script* es:

```
void show()
void hide()
void toggle()
void tappedOnIndex(int i)
```

Los métodos `show()`, `hide()` y `toggle()` se usan para mostrar y ocultar el carrusel, de la misma manera que un menú desplegable. El método `tappedOnIndex()` se utiliza para disparar una acción dependiendo de en cuál *sprite* ha pulsado el usuario.

Un ejemplo del uso del carrusel es el visualizador de volcanes mostrado en la Figura 6.10. Sobre el marcador se muestra en realidad aumentada el modelo de un volcán, de entre ocho disponibles en la aplicación. Cuando el usuario toca el volcán, aparece el menú de carrusel, mostrando ocho fotografías correspondientes a los volcanes disponibles para visualización. No todas las fotografías son visibles a la vez, sino que se extienden a la derecha. El usuario tiende a arrastrar las fotografías para revelar el resto, lo cual se debe a su experiencia previa con sistemas de navegación de imágenes, ya sea con barras de desplazamiento (*scrollbars*) o mediante un carrusel de fotos, como en la aplicación *Finder* de *MacOS*. Al pulsar en cada foto, el *widget* del volcán se actualiza y el carrusel se oculta. También se actualiza el nombre del volcán, en el área de texto de la ventana HUD.

Figura 6.10: Ejemplo de un carrusel para seleccionar uno de los volcanes.

6.2.5. Perilla

Este *widget* de interfaz de usuario permite seleccionar un valor numérico dentro de un cierto rango, i.e., tiene la misma funcionalidad del control deslizante (*slider*). Se puede configurar para tomar valores numéricos enteros o flotantes y simplemente se le deben proporcionar los valores máximo y mínimo. La idea de desarrollar este *widget*, está inspirada en las perillas de un radio antiguo, usadas para sintonizar la frecuencia o para seleccionar el nivel de volumen. Además, la perilla (*knob*) cuenta con

una etiqueta de texto que muestra el valor actual. A la izquierda de la Figura 6.11, se muestra el prototipo de una perilla 3D.

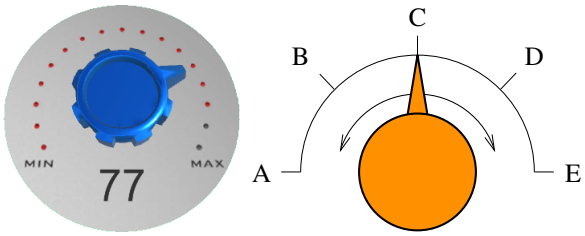


Figura 6.11: Perilla tridimensional (izquierda) y diagrama de configuración para selección discreta (derecha).

Es posible obtener un comportamiento similar al de un panel de botones o de un carrusel, i.e., seleccionar una opción de entre n disponibles, restringiendo la perilla a valores enteros y definiendo un rango de 1 a n. Cuando se selecciona el valor deseado, se puede disparar la acción correspondiente sobre otro objeto. Con la perilla se puede seleccionar sólo una de las opciones disponibles. A la derecha de la Figura 6.11, se muestra el diagrama de configuración de una perilla como un selector de cinco opciones.

Las funciones para configurar una perilla desde un *script* son:

```
void setMax (float max)
void setMin (float min)
void useIntegers (bool b)
void onValueChanged()
```

Con los métodos `setMax()` y `setMin()` se indican los valores máximo y mínimo de una perilla. Mediante `useIntegers()` se indica si los valores serán considerados como enteros o como flotantes. El evento `onValueChanged()` se dispara cuando se mueve la perilla y se puede conectar con otro objeto para controlarlo.

6.3. *Widgets* de datos

Un *widget* de datos constituye la representación gráfica de la estructura de datos del modelo 3D que se desea editar. De la misma manera que el modelo 3D es compartido por los colaboradores y se encuentra replicado en cada instancia de la aplicación, su *widget* de datos se puede considerar en cierta forma compartido y replicado, pues cada instancia tiene una copia y su forma se actualiza en tiempo real, en la medida que lo hace el modelo al que representa. El marco de desarrollo ShAREd está diseñado para manipular un solo modelo 3D, de manera que en las aplicaciones resultantes se tiene un solo *widget* de datos. En principio, el *widget* de datos solamente refleja la forma geométrica del modelo 3D y no es necesario que sea interactivo. Sin embargo, si se requiere, es posible diseñar *widgets* de datos que respondan a la interacción del

usuario, de la misma manera que un *widget* de interfaz, en cuyo caso se puede considerar al *widget* de datos como parte de la interfaz de usuario. Durante la creación de los prototipos de editores colaborativos, diseñamos los *widgets* de datos que se describen a continuación.

6.3.1. Pieza sólida

Para el prototipo del Capítulo 7, se diseñó un *widget* que puede representar piezas sólidas, tales como las piezas del juego de ajedrez u otros juegos de tablero. Cada pieza sólida es un *widget* manipulable al que se pueden asignar diversos comportamientos, via código. La pieza tiene una malla asociada, pero que no es modificada por el usuario. En lugar de ello, el usuario la manipula mediante operaciones de transformación geométrica, tales como traslación, rotación y escala. Dicha interacción se realiza mediante gestos táctiles aplicados directamente sobre la pieza. En el prototipo del editor de exhibiciones, el *widget* de pieza sólida es utilizado para representar los muebles que se acomodan dentro de una sede virtual.

También se utilizó el *widget* de pieza sólida en un experimento de apilamiento de bloques. Se crean nuevos bloques cúbicos con un gesto de pulsar y el usuario los puede trasladar y girar. Además los bloques están sujetos a varias modalidades del simulador de física: *Sin física* los bloques no caen por la gravedad y no colisionan entre ellos, i.e., se pueden traslapar. *Con física* los bloques caen al piso y colisionan entre ellos, realizando una simulación mecánica completa en la que pueden girar y rebotar. *Con pseudo-física* la simulación es incompleta, i.e., los bloques se desplazan por efecto de la gravedad y pueden empujarse entre ellos, pero no se simula la interacción mecánica. Los bloques se trasladan, pero no giran por el efecto de las fuerzas de torque. Esta modalidad es más ligera, pero no es realista.

La API para controlar los bloques desde un *script* es:

```
void onDrag(Event e)
void onTransform(Event e)
void performCollissions(bool active)
void useGravity(bool active)
void setGravity(float value)
```

De manera similar al *sprite*, con los métodos `onDrag()` y `onTransform()` la pieza responde a los gestos para trasladar, rotar y cambiar el tamaño. El método `performCollissions()` sirve para activar la detección de colisiones entre objetos, mientras que los métodos `useGravity()` y `setGravity()` controlan la gravedad: el primero sirve para activarla o desactivarla y el segundo establece su valor, e.g., para simular las condiciones en otro planeta o bajo el agua.

6.3.2. Visualizador de volcanes

Se construyó un *widget* de datos capaz de visualizar un conjunto de volcanes 3D con realidad aumentada, modelados a partir de los datos de elevación y su correspondiente textura satelital. Como se expone en el Capítulo 5.1, este *widget* es un

`GameObject` que tiene asociado un componente `BoxCollider` que le permite recibir gestos táctiles (via el componente *TouchScript*) para poder interactuar con el contenido. Mediante un componente `MonoBehavior`, se controla el comportamiento del *widget* y, a través de él, se eligen los datos de elevación y las texturas[2] que el *widget* necesita para crear la malla y darle la forma del volcán deseado, en tiempo real. Por supuesto, este *widget* no está intrínsecamente limitado a visualizar volcanes, sino que se puede visualizar cualquier terreno para el que se hayan preparado un mapa de elevación y una textura.

El usuario no edita ni modifica la malla del *widget* en modo alguno, sino que el propio comportamiento se encarga de ajustar la altura de los vértices de la malla, según los datos proporcionados, así como cambiar su textura, usando la imagen satelital correspondiente. La interacción del usuario con este *widget* se limita a elegir cuál terreno se va a mostrar, e.g., al seleccionar de entre el conjunto de volcanes disponibles mediante un *widget* de interfaz, tal como el carrusel. En este caso, ambos *widgets* interactúan entre sí; al tocar el terreno se muestra el carrusel de *sprites* y al seleccionar un *sprite*, el terreno se actualiza. La API para controlar este *widget* es:

```
void onTap (Event e)
void onDrag (Event e)
void setTexture (Texture tex)
void setDEMfromImage (Image im)
void setDEM (int index)
```

El método `onTap()` se usó en este caso para mostrar el carrusel y `onDrag()` para trazar puntos sobre el volcán. Por medio del método `setTexture()` se puede elegir la textura del volcán. Mediante el método `setDEMfromImage()` se modifican los datos del mapa de elevación y se actualiza la forma del volcán, de acuerdo a una imagen en escala de grises que contiene los datos. Mediante el método `setDEM()` se elige uno de los volcanes almacenados en el arreglo.

6.4. *Widgets* de consciencia de grupo

Los *widgets* de consciencia de grupo fungen como avatares de los colaboradores o de sus acciones dentro de la escena 3D. Pueden representar a los colaboradores o solamente mostrar la información necesaria, para que el usuario pueda formarse una idea mental de las acciones que los demás están realizando, en el transcurso de la sesión colaborativa. Este tipo de *widget* puede ser interactivo, dependiendo de las necesidades de la aplicación, pero no es controlable por el usuario, puesto que debe reflejar el estado de otro. Dicha interacción pudiera limitarse, por ejemplo, a que el usuario local elija qué aspecto del estado de los colaboradores mostrar. Este *widget* se comparte de la misma manera que uno de datos, pero no tiene interacción directa con el modelo 3D. De los prototipos desarrollados, solamente el editor de terrenos cuenta con un *widget* de consciencia de grupo, el cual se describe en la Sección 8.3.2.

[2]Para cada volcán se requieren dos imágenes, un mapa de elevación y una textura, previamente preparadas como se muestra en el ejemplo de los videos en https://www.youtube.com/playlist?list=PLdRMZeqyUGGhEkkFfIQyaHkBPWVJmlpMr.

Capítulo 7

Editor de exhibiciones académicas

Un aspecto fundamental en la academia es la divulgación de la ciencia y la tecnología, dirigida al público en general, pero especialmente a los jóvenes. Sin embargo, la mayoría de los científicos y tecnólogos, carecen del tiempo necesario para organizar eventos científicos, tales como exhibiciones en museos, ferias de ciencia, entre otros. No obstante, algunos investigadores logran darse un tiempo para llevar a cabo la difusión del conocimiento que la sociedad demanda y hacer llegar sus temas de investigación a los jóvenes estudiantes, tratando con ello de revivir su interés por la ciencia y la tecnología. El prototipo descrito en este capítulo ha sido desarrollado con la finalidad de apoyar a tales investigadores.

Una primera aproximación para construir una escena 3D es colocar una serie de objetos individuales, mediante transformaciones afines de traslación, rotación y escala. Dichos objetos pueden ser figuras geométricas, tales como cajas, cubos, esferas, pirámides, conos, entre otros. También es posible utilizar modelos 3D creados previamente, como vehículos, edificios y otros bloques de construcción. Esta aproximación es similar a construir una maqueta física a partir de un conjunto de miniaturas prefabricadas, como casas, autos o árboles. En esta situación, la tarea consiste en colocar las piezas en una cierta disposición para obtener la forma de la maqueta deseada, pero no es necesario modificar la forma de las piezas individuales, sino solamente trasladarlas, girarlas e inclusive cambiar su tamaño.

Inspirados en este tipo de interacción, creamos un prototipo de editor colaborativo para dispositivos móviles, que permite a un grupo de usuarios manipular un conjunto de modelos 3D con la forma de diversos muebles, a fin de diseñar y previsualizar la disposición del mobiliario, utilizando realidad aumentada. Mediante este editor, los participantes pueden definir el tamaño de una sala de exposiciones virtual y acomodar en ella el mobiliario necesario para montar una exhibición de tipo académico. Pueden colocar los muebles de maneras distintas para elegir la disposición más adecuada para sus necesidades, dentro del espacio disponible. Para desarrollar el editor, aplicamos el marco de desarrollo ShAREd, el cual brinda al grupo de colaboradores la funcionalidad de compartir y manipular los objetos de la escena 3D, utilizando realidad aumentada.

La previsualización basada en realidad aumentada es interactiva, pues cada colaborador puede visualizar la disposición de los muebles desde distintas perspectivas, seleccionar piezas individuales y manipularlas con el fin de evaluar la disposición general de la exhibición, dentro del espacio disponible en la sede del evento. La aplicación se puede ejecutar en varias plataformas: en iOS, en Android y en navegador Web en una computadora de escritorio.

Durante la etapa de diseño y planeación de una exhibición, una herramienta de previsualización ayuda al grupo de colaboradores a analizar distintas distribuciones, sin costo adicional y sin el esfuerzo físico de trasladar, en repetidas ocasiones, una gran cantidad de muebles reales, para así poder elegir la mejor disposición.

Además, en algunas ocasiones, los responsables de montar la exhibición no tienen acceso al lugar del evento, sino hasta unas pocas horas antes de que ésta comience. Al utilizar una herramienta de previsualización, es posible decidir, por adelantado, la forma más conveniente de acomodar los contenidos de la exhibición, estimar la cantidad de asistentes y diseñar el recorrido entre las distintas secciones. Con ayuda de una previsualización virtual, los colaboradores pueden evaluar y llegar a un consenso acerca de la distribución final. Una vez que se ha establecido la disposición del mobiliario, el prototipo puede utilizarse para guiar al equipo responsable de colocar los muebles reales en el espacio físico. Así también, puede ser útil para indicar a los ponentes cuál es su lugar asignado dentro de la sede.

A continuación, se describe el modelo de la exhibición en la Sección 7.1. Luego, en la Sección 7.2, se discute el diseño de los *widgets* para representar el mobiliario y, finalmente, se dan algunos detalles de implementación del prototipo, en la Sección 7.3.

7.1. Modelo de la exhibición

El problema que abordamos es el de construir una aplicación que ayude a un grupo de colaboradores a diseñar, previsualizar y modificar la disposición del mobiliario en una exposición académica, compuesta por elementos diversos. La clase de exhibición en la que nos enfocamos, en este trabajo, consta de los siguientes elementos:

- Un **espacio físico** en el que se van a colocar los contenidos de la exhibición, e.g., la sala de un museo, un auditorio o un salón de clase. Dicho espacio está definido por un conjunto de muros que encierran un área dada, la cual puede también ser dividida en secciones. En la Sección 7.2.1 se describe la implementación de este elemento.

- Diversos tipos de **mobiliario** en los que se colocarán los contenidos a exhibirse. Algunos muebles utilizados con frecuencia en exhibiciones académicas son: mamparas para pósters, mesas, sillas, bancas, tapetes, páneles divisores y pantallas. Los modelos 3D, creados en Blender, que representan estos muebles se muestran en la Figura 7.1 y los *widgets* con los que interactúa el usuario en Unity se describen en la Sección 7.2.2.

- El **contenido** que será colocado en cada mueble. Es necesario representar dentro de la aplicación, de la manera más sencilla posible, cada tipo concebible de contenido que pudiera presentarse en cualquier exhibición y sin la necesidad de crear réplicas virtuales cada vez que se requiera un nuevo objeto, producto de investigación o concepto abstracto, pues intentarlo sería abrumador e infructuoso. Así pues, como se describe en la Sección 7.2.3, simplemente se coloca en cada mueble una etiqueta de texto, que identifica y describe el contenido que tendrá asociado.

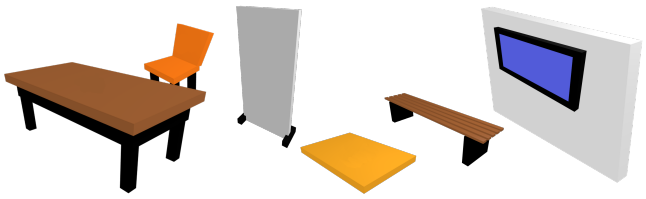


Figura 7.1: Modelos del mobiliario disponible en la aplicación: mesas, sillas, mamparas para póster, tapetes, bancas, pantallas planas y páneles divisores.

En el contexto del presente trabajo, el plan para la disposición del mobiliario (*layout plan*) se define como la distribución espacial del mobiliario y sus contenidos asociados, dentro de un espacio físico dado. De manera que, cuando los colaboradores finalmente han decidido la ubicación y orientación de cada pieza de mobiliario, así como el contenido que se colocará en cada una de ellas, se tiene una guía completa para montar la exhibición.

7.2. Diseño de los *widgets*

Para implementar este prototipo, usando el marco de desarrollo ShAREd, únicamente fue necesario diseñar e implementar los *widgets* de datos que componen el modelo que se desea crear y un *widget* de interfaz de usuario, colocado en el HUD. En este caso, no fue necesario implementar la funcionalidad para el modelado de mallas, ni los *widgets* para consciencia de grupo.

El modelo que queremos crear es una maqueta de una exhibición compuesta por *widgets* de mobiliario, que están basados en el *widget* de datos para representar piezas sólidas, descrito en la Sección 6.3.1. El usuario puede interactuar, de manera intuitiva, con estos *widgets*, mediante gestos táctiles. Se tiene también un *widget* de un menú desplegable, que es utilizado para agregar nuevas instancias de los *widgets* de mobiliario. A continuación, se describen las consideraciones de diseño para la creación de cada uno de dichos *widgets*.

7.2.1. Delimitación la sede

Comúnmente, las exhibiciones tienen lugar en sedes con forma rectangular de diversos tamaños. Aunque, en algunos casos, las sedes pueden tener otras formas, como pueden ser triangulares, poligonales, circulares o elípticas, la forma más frecuente y versátil sigue siendo la rectangular. Así, en el presente prototipo, se define el espacio de la sede con forma rectangular de tamaño variable.

Al inicio de la aplicación, se elige el tamaño del área rectangular y se muestra el modelo de la sede a escala sobre el marcador, como se puede ver en la Figura 7.2. Con un gesto de arrastre (*drag*) se puede desplazar el modelo completo de la sede, junto con su contenido. Asimismo, mediante el gesto de pellizco (*pinch*) se puede acercar o alejar la vista de todo el conjunto.

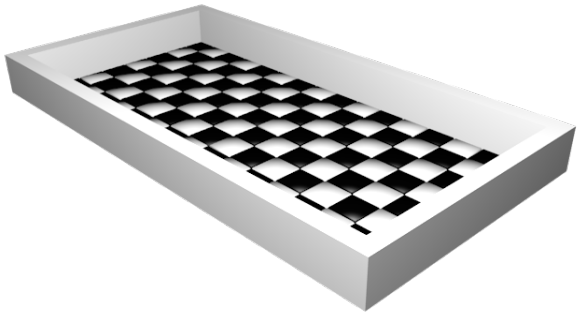


Figura 7.2: Área rectangular que representa la sede, escalada al tamaño del marcador.

En esta etapa de delimitación de la sede, también podemos indicar los puntos de acceso, i.e., las entradas y salidas a la sede. Estas están representadas mediante un *widget* con forma de puerta, como el que se muestra en la Figura 7.3. El funcionamiento de este *widget* es prácticamente el mismo que el de los otros *widgets* de mobiliario, los cuales se describen en la Sección 7.2.2.

Como se puede observar en las figuras 7.2 y 7.3, el área de la sede está delimitada por cuatro muros bajos, de manera que no obstruyen la visión hacia el interior desde cualquier perspectiva. Mientras que las puertas, al igual que el resto de los muebles, se han modelado más altas para hacerlas más visibles.

7.2.2. *Widgets* de mobiliario

Para representar los muebles, creamos varios *widgets* del tipo de pieza sólida, descritos en la Sección 6.3.1. Como se mencionó en la Sección 5.1, en Unity, a cada instancia de `GameObject` en la escena pueden agregársele comportamientos diversos en la forma de *scripts*. Cada instancia de `GameObject` se puede empaquetar, junto con todos sus componentes y comportamientos asociados, en una estructura llamada *prefab*. En este editor, los *widgets* de mobiliario vienen definidos en *prefabs* y pueden ser agregados fácilmente a la escena en tiempo de ejecución. Es posible agregar la cantidad deseada de muebles, con la ayuda del menú lateral desplegable, que contiene

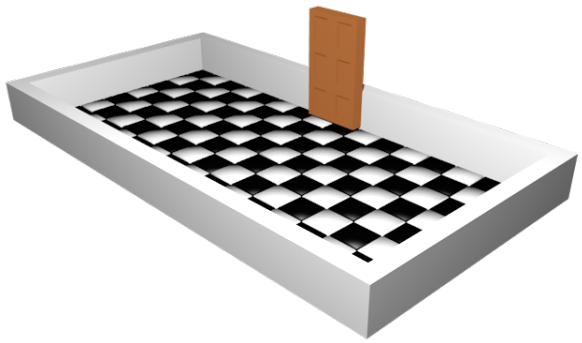


Figura 7.3: Un *widget* de puerta representa un acceso a la sede virtual.

una serie de botones, uno para cada tipo de mueble disponible. En este prototipo, los *widgets* de mobiliario tienen un tamaño fijo, con respecto al espacio de la sede, y no pueden ser redimensionados. Pero cuando la sede se desplaza o se redimensiona, también el mobiliario lo hace junto con esta.

Al igual que el *widget* original de pieza sólida, los *widgets* de mobiliario tienen asociada una malla poligonal y pueden ser trasladados sobre la superficie del marcador, mediante el gesto de arrastre (*drag*). Partiendo del *widget* de pieza sólida original, diseñamos uno que se puede girar sobre su eje vertical, por medio del gesto estándar de rotar con dos dedos. Sin embargo, no todos se pueden rotar fácilmente con este gesto, por ejemplo las pantallas, las mamparas y los paneles son más delgados que los otros muebles y muestran muy poca superficie en la parte superior, lo que dificulta la colocación precisa de dos dedos para realizar el gesto. En estos casos, se agregó una pequeña barra en la parte superior que responde al gesto de arrastre con un solo dedo, pero que hace girar el *widget*.

Los *widgets* de mobiliario también pueden ser eliminados de la escena, mediante un gesto. Proponemos que el gesto utilizado para eliminar los muebles no sea fácil de ejecutar por accidente, como lo es la pulsación simple. Incluso notamos que, en ocasiones, también se puede hacer una doble pulsación accidental. Con base en nuestra experiencia en el uso de la aplicación, se propone que el gesto para eliminar un *widget* de mobiliario sea una triple pulsación. Dado que los colaboradores se encuentran cara a cara, es de suponer que discutirán primero, antes de eliminar un mueble que haya sido colocado por alguien más.

En la Figura 7.4, se muestra un ejemplo de una exhibición ubicada en una sala de tamaño reducido que tiene una sola puerta. Se colocaron tres mamparas para poster, una mesa con su silla, a manera de punto de registro o de información, una pantalla para observar por ejemplo un video o una animación, junto con dos bancas para que los asistentes observen cómodamente.

Uno de los *widgets* que se implementaron es el panel separador. Se trata de una especie de muro que se puede mover, girar y cambiar de longitud. Es útil para separar en secciones el interior de la sede. Este panel se puede trasladar y girar de la misma manera que una mampara o una mesa, pero no tiene un contenido asociado. Su

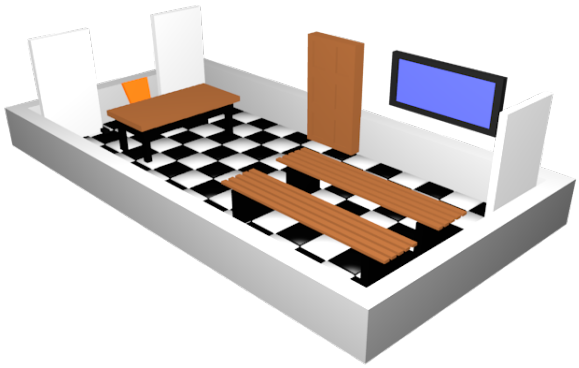


Figura 7.4: Un ejemplo de disposición del mobiliario para una sala de tamaño pequeño.

longitud se modifica mediante el gesto de pellizco, pero su altura y grosor son fijos y distintos a los que tienen los muros de la sede, para indicar al usuario que se trata de un panel móvil y así evitar que los confunda.

7.2.3. Contenido de la exhibición

Una vez colocados los *widgets* de mobiliario dentro del espacio de la sede, los colaboradores pueden darse una idea clara acerca de la apariencia global que tendrá la exhibición en la sede real (como se muestra en la Figura 7.4), pero además es necesario indicar cuál contenido se debe colocar en cada lugar. Para ello, se agrega a cada mueble un *widget* de notas, mediante el cual los colaboradores pueden escribir una nota de texto sobre cada mueble. En la Figura 7.5 se muestra el proceso para agregar un texto a la nota, se pulsa sobre ella (en la parte superior izquierda) para invocar un cuadro de entrada de texto (en la parte superior derecha) y se escribe en este la nota deseada (al centro izquierda). El texto escrito se despliega sobre la nota (al centro derecha). En las capturas inferiores de la Figura 7.5, se muestran los muebles con anotaciones desde perspectivas distintas, usando realidad aumentada. En la nota se puede describir el contenido o cualquier información útil para el montaje de la pieza. Sugerimos que se asigne un nombre clave o un código a cada contenido real que se llevará a la exhibición, posiblemente con un *post-it* real, para identificarlo dentro y fuera de la aplicación.

7.3. Implementación

El presente prototipo del editor de exhibiciones fue probado con éxito en varios dispositivos iOS y Android, de gama media y alta, así como en algunos de gama baja, con excepción de ciertos dispositivos antiguos con arquitectura *x86* (en lugar de la reciente *Armv7*) en los cuales se denegaba el acceso a la cámara del dispositivo. Sin embargo, en dichos dispositivos y en PCs con navegador Web, fue posible ejecutarlo en la modalidad estándar, sin realidad aumentada.

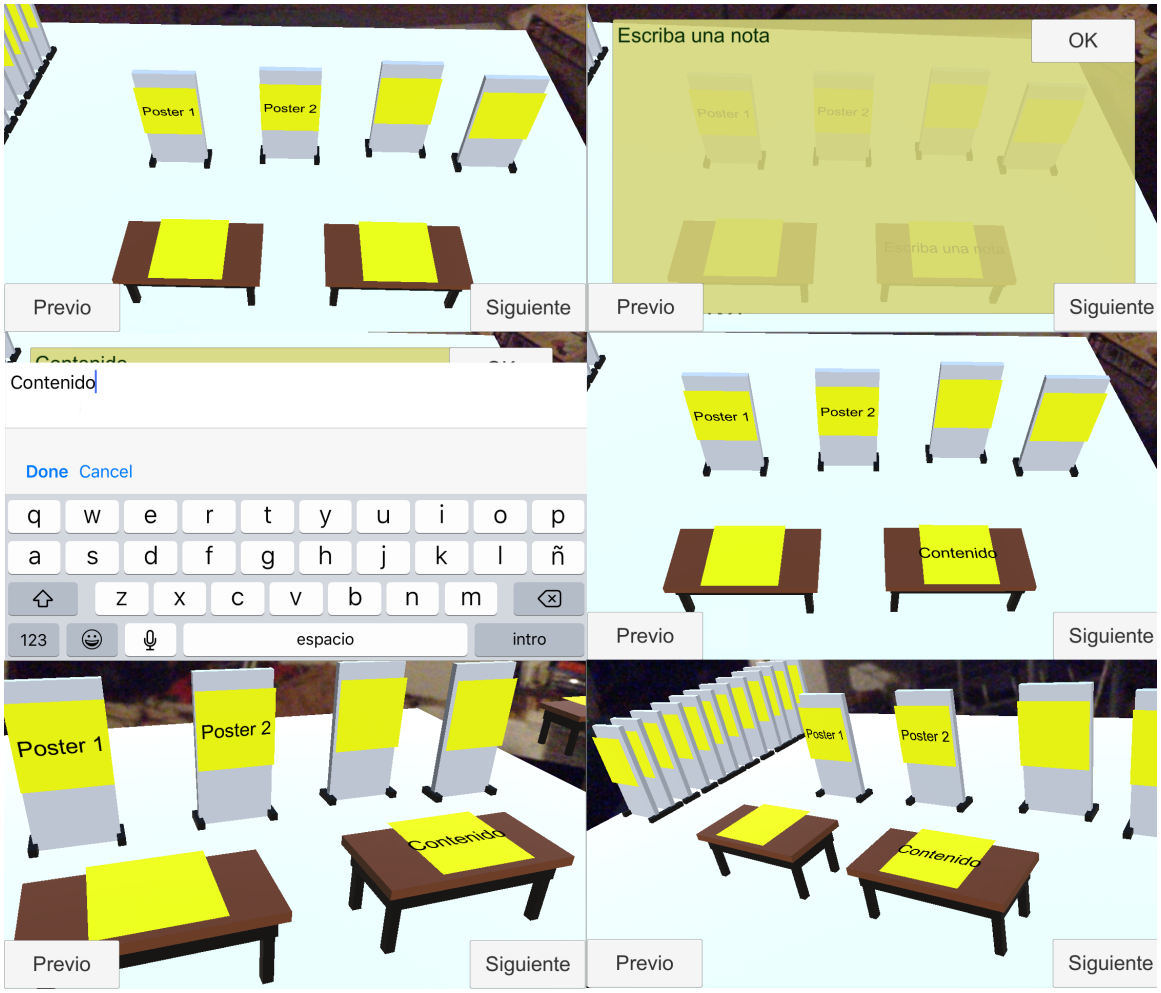


Figura 7.5: Proceso para agregar una nota sobre uno de los muebles. El panel de texto mostrado corresponde al sistema operativo iOS.

Los *widgets* de mobiliario se crean dinámicamente en tiempo de ejecución y fueron modelados con una cantidad reducida de polígonos. Durante las pruebas en dispositivos de gama media y baja fue posible crear hasta aproximadamente unas 50 piezas de mobiliario en la escena, sin que se percibiera una pérdida de respuesta en el modo de realidad aumentada.

7.3.1. Distribución de datos

Como se mencionó en la Sección 5.5, mediante las clases `NetworkIdentity` y `NetworkTransform` de la API de alto nivel (*HLAPI*) de Unity, cada instancia de `GameObject` es responsable de transmitir su transformación a todos los participantes conectados a la sesión colaborativa. Aprovechamos esta funcionalidad en el presente prototipo, para manipular las piezas sólidas, mientras que para compartir los textos de las notas, utilizamos el mecanismo de comandos con llamadas remotas *RPC*.

7.3.2. Modos de visualización

En este prototipo se proporciona a cada usuario la posibilidad de cambiar el modo de visualización en tiempo de ejecución. Utilizamos el diseño propuesto en la Sección 5.6, en donde se describe una escena con dos cámaras: una estándar y otra de Vuforia para realidad aumentada. Usando la modalidad de realidad aumentada, se requiere el uso de un marcador impreso[1]. En la modalidad estándar, el usuario puede visualizar la sede vista desde arriba, como si se tratara de un mapa de ubicación. Esta modalidad puede ser útil para proporcionar instrucciones a los colaboradores para el montaje de la exhibición o para indicar la ubicación de ciertos puntos de interés, dentro de la sede. Es posible tomar una captura de pantalla (pulsando de manera simultánea los botones físicos de encendido y de menú del dispositivo) la cual puede ser compartida con los encargados de montar la exhibición o con los ponentes. Además, en ambas modalidades, se puede desplazar y hacer acercamiento a la vista de manera local, i.e., cada participante se puede enfocar en una parte distinta de la exhibición.

7.3.3. Ciclo de vida de la aplicación

Los participantes necesitan contar con un dispositivo móvil para cada uno, con la aplicación instalada, y estar colocalizados en un lugar que cuente con servicio de WiFi local.

- Se requiere el marcador impreso, de un tamaño adecuado al lugar de reunión del equipo.
- Los colaboradores se conectan a la misma red WiFi y arrancan la aplicación.
- El coordinador del equipo crea una nueva sesión colaborativa con un nombre reconocible por los colaboradores.
- Los demás colaboradores se conectan a dicha sesión.
- Al inicar la sesión se establece el tamaño de la sede, arrastrando las esquinas de la región. El diseño de la sede se encuentra en la Sección 7.2.1.
- Cada participante puede agregar, manipular y eliminar muebles, así como poner en ellos las anotaciones pertinentes, como se describió en las Secciones 7.2.2 y 7.2.3.
- La disposición final del mobiliario se puede observar en la vista estándar, como un mapa de la exhibición, o en realidad aumentada desde diversos ángulos.

En la presente versión de la aplicación, solamente se puede tener una exhibición a la vez, pues aún no se implementa un mecanismo de persistencia.

[1] La imagen que se usó como marcador en los prototipos se llama *chips*, y se encuentra en la página 2 de los ejemplos de Vuforia, disponibles en `https://developer.vuforia.com/sites/default/files/sample-apps/targets/imagetargets_targets.pdf`

Capítulo 8

Editor de terrenos

En el Capítulo 7 se construye una escena 3D manipulando objetos sólidos, pero sin modificar su forma internamente. El siguiente paso hacia la edición colaborativa de mallas poligonales es modificar concurrentemente su forma, trasladando sus vértices, pero sin afectar su topología, i.e., sin eliminar ni agregar elementos: vértices, aristas o caras. Esta aproximación es útil, por ejemplo, para definir un mapa de elevación, una cierta superficie o un terreno virtual. En el área de Gráficos por Computadora, este tipo de superficies es útil para construir escenarios virtuales para animaciones y videojuegos, pero también en aplicaciones científicas y educativas, e.g., en simulaciones de procesos geológicos y pluviales. Se pueden clasificar los programas de creación de terrenos en dos categorías: generadores y editores.

Los generadores de terrenos suelen estar basados en el paradigma de síntesis algorítmica, en el cual el terreno se sintetiza de manera global, usando varios algoritmos y procedimientos computacionales. El término global se refiere a que, en este paradigma, el terreno se procesa como un todo, en lugar de hacerlo por regiones, de modo que el control sobre las características locales presentes en el resultado final es muy limitado. Se pueden mencionar tres tipos de métodos para la generación de terrenos: los basados en fractales [45, 49], los basados en simulación física [10, 47] e híbridos de éstos [22]. Estos generadores son útiles cuando no es importante la forma final del terreno generado. Sin embargo, pueden ser útiles para proporcionar una base para crear el nuevo terreno y modificarlo posteriormente para darle la forma deseada.

Por otra parte, los editores de terrenos suelen utilizar el paradigma de pintar con brocha (*brush painting*), llamado también escultura interactiva con brocha (*interactive brush-based sculpting*). En este paradigma, el sistema proporciona una herramienta que funciona bajo la metáfora de una brocha, i.e., el usuario *pinta* sobre un objeto y localmente aplica cambios en alguna de sus propiedades, como puede ser color, textura o forma. Al *pintar* una operación, ésta se aplica únicamente sobre la región cubierta por el trazo de la brocha, en vez de afectar a todo el terreno, como ocurre con la síntesis algorítmica. Por medio de este paradigma, los usuarios de este tipo de editores tienen un control total en la posición y tamaño de las formas que están esculpiendo.

En este proyecto, se diseñó una aplicación colaborativa de realidad aumentada para la autoría de bosquejos de terrenos para dispositivos móviles, utilizando la arquitectura del marco de desarrollo ShAREd y el paradigma de pintar con brocha. A partir de dicho diseño, se implementaron dos versiones del editor de terrenos. La primera versión se presentó en el evento *AppyHour* llevado a cabo en el SIGGRAPH 2014[1], donde recibimos una valiosa retroalimentación, tanto de expertos en el área como de asistentes en general. La aplicación recibió buenos comentarios y el concepto de edición colaborativa utilizando realidad aumentada fue bien recibido. De este modo, fue posible identificar puntos de oportunidad para mejorar el diseño, dando lugar a la segunda versión.

Como se mencionó en el Capítulo 4, ShAREd establece la arquitectura básica del editor y proporciona la funcionalidad necesaria para mostrar el contenido virtual en dos modalidades: usando realidad aumentada o mediante despliegue estándar. También se encarga de transmitir los mensajes de operación, originados al interactuar con la interfaz de usuario, hacia los demás colaboradores utilizando una arquitectura cliente-servidor con autoridad. En la Sección 2.1, se explica que, en algunos tipos de editores colaborativos, no es necesario preocuparse por mantener la consistencia de datos en las copias replicadas del contenido compartido. A nuestro saber, no se ha reportado en la literatura, si se requiere implementar o no algún tipo de control de consistencia de datos en un editor como el desarrollado en este capítulo. Por ello, en el presente trabajo, se evalúa la necesidad de incorporar dicho control a este prototipo, midiendo las inconsistencias de datos ocurridas durante varias sesiones colaborativas con usuarios novatos reales.

El presente editor de terrenos se diseñó para el siguiente escenario:

- Un grupo pequeño de colaboradores[2] se encuentra reunido, en un mismo lugar, con el fin de bosquejar la forma de un terreno con algún cierto propósito.

- Cada uno de ellos cuenta con un dispositivo móvil. Para la primera versión del editor se requieren iPads con conectividad WiFi, mientras que para la segunda versión se pueden utilizar también dispositivos Android y PCs con un navegador Web.

- El lugar cuenta con una red WiFi local, la cual no se encuentra muy ocupada. Durante el intercambio de mensajes, pueden ocurrir pérdidas de paquetes, sin embargo, la resolución de este problema no forma parte de los objetivos del presente trabajo.

[1] 41st International Conference and Exhibition on Computer Graphics and Interactive Techniques (http://s2014.siggraph.org/attendees/appy-hour.html)

[2] En un grupo de a lo más 15 personas, es posible que cada una pueda desplazarse cómodamente dentro de un salón.

8.1. Modelo del terreno

En el contexto de este proyecto, un terreno del mundo real es una parcela de tierra o un territorio, que contiene ciertas características geométricas que definen su forma. Tales características pueden ser montañas, cráteres, cañones, valles, grietas, cauces de ríos, lechos lacustres, plataformas, planicies, entre otros.

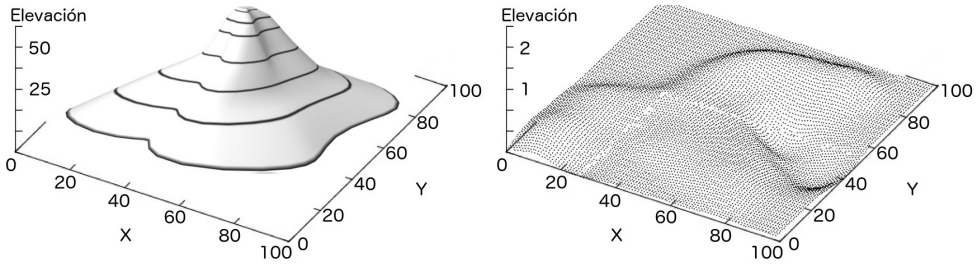


Figura 8.1: Gráfica de un mapa de elevación contínuo con curvas de nivel (izquierda). Gráfica de un DEM (derecha), en la que se tiene un valor de elevación para cada punto del dominio discreto.

La manera más sencilla de abstraer la forma de un terreno, en un modelo matemático, es asignar un valor numérico para la elevación en cada punto de su superficie, medido con respecto a alguna referencia previamente establecida, e.g., el nivel del mar. Podemos describir al terreno a través de una función contínua:

$$h' : \mathcal{R}' \to \mathbb{R},$$

donde $\mathcal{R}' \subset \mathbb{R}^2$ es la región del espacio en la que se ubica el terreno. En el lado izquierdo de la Figura 8.1 se muestra la gráfica contínua de un mapa de elevación con la forma de una montaña. Se muestran las curvas de nivel sobre la superficie.

Sin embargo, en aplicaciones computacionales, no podemos darnos el lujo de utilizar una función contínua. En su lugar, hacemos uso de una representación discreta, conocida como mapa de elevación digital (*Digital Elevation Map*, DEM), la cual es una función discreta:

$$h : \mathcal{R} \to \mathcal{F},$$

donde $\mathcal{R}$ es la región rectangular $[0, m-1] \times [0, n-1] \subset \mathbb{N}^2$, para los enteros dados $m, n \in \mathbb{N}$ y $\mathcal{F}$ representa el dominio de números de punto flotante, que se pueden almacenar en una computadora. De este modo, un DEM puede ser almacenado como un arreglo rectangular de números de punto flotante, de tamaño $m \times n$. En el lado derecho de la Figura 8.1, se muestra la gráfica de los puntos discretos de un DEM.

Los mapas de elevación digitales se utilizan comúnmente, en el área de Gráficos por Computadora, para representar objetos como la superficie de cuerpos de agua o la forma de terrenos virtuales, aunque también es posible utilizarlos en otros contextos. Por esta razón, es frecuente encontrar en la literatura los términos "superficie", "terreno" y "DEM", empleados de manera indistinta, dependiendo del contexto.

En el presente prototipo, se representa gráficamente al arreglo de datos del DEM como una superficie 3D, como se muestra en la Figura 8.2, mediante una malla poligonal.

Figura 8.2: Representación gráfica de un DEM, usando una malla poligonal. A la malla se aplicó una textura de piedras.

Dado que h es función, no puede representar todos los detalles presentes en un terreno real, tales como cuevas, salientes o la composición interna del subsuelo. No obstante, esta representación es suficiente en la mayoría de las aplicaciones prácticas.

8.2. Diseño de las operaciones de edición

Se ha mencionado que el editor de terrenos utiliza el paradigma de *pintar con brocha*, inspirado en el proceso de aplicar pintura sobre una superficie utilizando una brocha como herramienta. El lector seguramente se encuentra familiarizado con esta forma de edición, utilizada en aplicaciones de dibujo, tanto en 2D como en 3D. De hecho, este paradigma es el estándar *de facto* en los editores de terrenos actuales, como es el caso del editor de terrenos de Rob Chadwick para navegador Web, denominado *WebGL Terrain Editor* y el trabajo sobre brochas procedurales (*procedural brushes*) de DeCarpentier et al. [22].

De manera similar a una brocha real, que puede aplicar colores distintos sobre una superficie, con una brocha digital se pueden aplicar diferentes operaciones de edición sobre el terreno digital. En nuestra propuesta, se ofrecen cuatro operaciones elementales de edición para manipular la forma del terreno. El efecto de aplicar cada una de las operaciones se describe a continuación:

- *Raise:* eleva el terreno, al incrementar los valores de elevación de aquellos puntos del DEM, que se encuentran en el interior de la región barrida por el trazo de la brocha. El efecto de esta operación es acumulativo, pues los incrementos se añaden a los valores previos.

- *Lower:* desciende el terreno dentro de la región del DEM que es barrida por el trazo, decrementando sus valores de elevación. Su efecto también es acumulativo, pues sustrae el valor sobre los valores previos.

- *Flatten:* aplana el terreno dentro de la región barrida por el trazo, al desplazar los puntos hacia cierto valor de elevación, dado por el usuario. El efecto de esta operación es acumulativamente destructivo, pues eventualmente se sobreescriben los valores previos tras repetidas aplicaciones de la operación sobre la misma región. Si el valor de elevación dado es cero, se comporta como un borrador ponderado por la fuerza del trazo.

- *Smooth:* reduce los cambios abruptos en la forma del terreno, dentro de la región barrida por el trazo, al desplazar sus puntos hacia el promedio de los valores que se encuentran dentro del radio de la brocha. Como en el caso de la operación anterior, esta también sobrescribe los valores previos, tras sucesivas aplicaciones sobre la misma región.

Cuando se aplica una cierta operación de edición sobre una región del terreno, la magnitud de su efecto queda en función de las propiedades de la brocha (su forma y radio) y de la fuerza del trazo. Existen varias formas que una brocha puede tomar: círculo, cuadrado, paralelepípedo, etc. En el prototipo, ofrecemos únicamente las dos formas circulares básicas, que se muestran en la Figura 8.3: de bordes duros y de bordes suaves. La forma de la brocha es utilizada para enmascarar el efecto de la operación sobre el terreno.

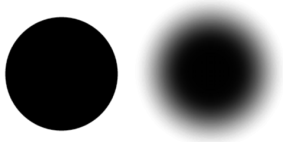


Figura 8.3: La brocha circular de bordes duros (izquierda) y de bordes suaves (derecha).

La operación seleccionada se aplica dentro de un círculo centrado en las coordenadas del terreno, dadas por el usuario mediante un toque en la pantalla táctil del dispositivo. El radio del círculo puede modificarse en tiempo de ejecución. La magnitud del cambio aplicado a cada punto, dentro del círculo, depende de la forma de la brocha y de un factor de fuerza del trazo, que el usuario también puede elegir. Cuando el usuario efectúa un trazo en la pantalla, los eventos táctiles (*touch events*), efectuados en la serie de coordenadas correspondientes, disparan operaciones de edición sucesivas sobre el DEM.

8.3. Diseño de la interfaz de usuario

Para que el usuario interactúe con el editor de terrenos, se diseñó la interfaz con un mínimo de elementos en el área de trabajo, acomodados de tal manera que no obstruyan el proceso de edición En la Figura 8.4, se muestra un diagrama con el diseño de la interfaz del editor de terrenos.

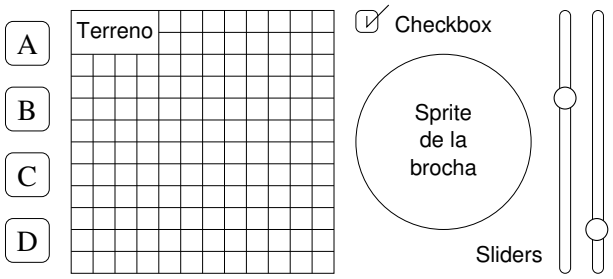


Figura 8.4: Diagrama del diseño básico de la interfaz de usuario para el editor de terrenos. La barra de herramientas se encuentra al lado izquierdo del terreno y los controles de la brocha al lado derecho.

Al centro del diagrama, se muestra la representación gráfica del terreno, una malla poligonal compuesta por $n \times n$ vértices. Esta malla se crea al iniciar la aplicación utilizando el Algoritmo 1 de la Sección 8.4.3. En ambas implementaciones elegimos $n = 100$, obteniendo una rejilla de 99×99 cuadrados, cada uno formado por dos triángulos. Con esta cantidad de triángulos, se obtiene una apariencia suave en los terrenos representados y es posible acercar la cámara sin que sean visualmente notorios los polígonos. Se puede elegir un valor menor para n y con ello reducir el costo de procesar una menor cantidad de vértices, pero se observó empíricamente que, en los dispositivos utilizados, $n = 100$ no representa un problema en la fluidez y respuesta de los editores.

Al lado izquierdo del terreno, se encuentra una barra de herramientas, compuesta por cuatro botones (etiquetados A, B, C y D en el diagrama), utilizados para seleccionar una de las cuatro operaciones descritas en la Sección 8.2. Para seleccionar una de las operaciones se pulsa sobre el botón correspondiente, el cual se resalta, para indicar que la operación correspondiente se encuentra activa. Entonces, el usuario puede aplicar la operación pintando sobre la representación del terreno. Sólo una operación puede estar activa a la vez.

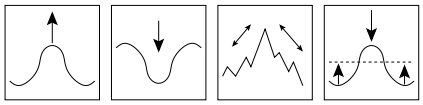


Figura 8.5: Íconos diseñados para expresar visualmente las cuatro operaciones, de izquierda a derecha: *Raise*, *Lower*, *Smooth* y *Flatten*.

Con el fin de expresar claramente el tipo de operación seleccionado, al pulsar alguno de los botones, se diseñaron los cuatro íconos mostrados en la Figura 8.5. Por defecto, al arranque de la aplicación, se encuentra seleccionada la primera operación (*Raise*) y el botón correspondiente está resaltado desde el principio, hasta que se elige otra operación.

Hemos mencionado que se utiliza una brocha digital para aplicar la operación seleccionada, al hacer un trazo sobre la región que se desea modificar. Los controles de las propiedades de la brocha se encuentran al lado derecho del terreno. Estas

propiedades se representan de manera visual con la ayuda de un *sprite*, con el cual se muestra la forma y el tamaño que se utilizará para aplicar la operación seleccionada.

El tamaño de la brocha se puede cambiar con uno de los controles deslizantes (*sliders*) o mediante un gesto de pellizco sobre el *sprite*, causando que éste cambie de tamaño de acuerdo al valor seleccionado. El otro control deslizante se utiliza para cambiar la fuerza del trazo, la cual también se mencionó en la Sección 8.2. Para elegir entre usar la brocha con borde suave o con borde duro, se utiliza la casilla de verificación (*checkbox*) o se aplica un gesto de doble pulsación sobre el *sprite* de la brocha. Con esta representación de *sprite* para la brocha digital, es posible dar al usuario una idea precisa de la porción del terreno que será afectada cuando se pinte sobre éste.

En el caso de la operación *Flatten*, se requiere una propiedad adicional, el valor de altura que se desea aplicar al terreno. Para seleccionar este valor, se agrega un tercer control deslizante que solamente es visible cuando la operación *Flatten* se encuentra activa.

8.3.1. Primera versión

Para la primera versión de editor, se implementó el diseño de interfaz de usuario descrito en la Sección 8.3, para la plataforma iOS, utilizando el lenguaje *Objective C* y la biblioteca gráfica *OpenGL*. Se muestra, en la Figura 8.6, una captura de pantalla la interfaz de usuario de esta primera versión.

Figura 8.6: Interfaz de usuario del editor de terrenos. La malla del terreno, la barra de herramientas y el *sprite* de la brocha están indicados.

Barra de herramientas

En esta versión del editor, implementamos cada botón como una caja con el ícono representativo sobre su cara superior (ver los íconos de la Figura 8.5). Por sugerencia de uno de los usuarios durante las pruebas, se agregó una etiqueta de texto con el nombre de la operación al lado del botón, además del ícono. Para seleccionar una de

las operaciones se pulsa sobre el botón correspondiente. El botón pulsado comienza a girar sobre su eje vertical, para indicar visualmente al usuario que ha sido seleccionado con éxito y se encuentra activa la operación correspondiente. Si el usuario elige otra operación, la seleccción previa se desactiva automáticamente, pues solo una puede estar activa a la vez. El botón que ha sido pulsado comienza a girar y el previo se detiene.

Botón de menú

Para esta versión del editor, agregamos un botón de menú, ubicado en la esquina superior izquierda del terreno. Al pulsar en este botón, aparece un menú flotante que contiene funciones que no son utilizadas con frecuencia en la aplicación, tales como crear una sesión colaborativa y unirse a una sesión creada por otro usuario. En este menú también hemos agregado toda la funcionalidad necesaria para la realización de las pruebas con los grupos de usuarios (Sección 8.6.4) tales como el envío de los datos de la copia local del DEM a un repositorio.

Sprite de la brocha

El *sprite* de la brocha se implementó con un cuadrado que puede mostrar dos texturas de imagen: una para la brocha con bordes duros y la otra con bordes suaves. El *sprite* responde al gesto de pulsar (`TapGesture`) de iOS; al hacer una doble pulsación se cambia la propiedad de forma de la brocha y se muestra la textura correspondiente. Además, el *sprite* responde al gesto de pellizco (`PinchGesture`) de iOS, con el cual se modifica su tamaño y se actualiza el valor del radio de la brocha. El radio y la fuerza de la brocha se pueden cambiar también con dos controles deslizantes, los cuales se implementaron como pequeñas cajas que responden al gesto de arrastrar (`DragGesture`) en una sola dirección.

8.3.2. Segunda versión

En la segunda versión del editor se efectuaron diversos cambios en el diseño de la aplicación y de la interfaz de usuario, a partir de la retroalimentación recibida de los usuarios de la primera versión del editor. Las mejoras que hicimos se mencionan a continuación, en la Tabla 8.1.

Tabla 8.1: Tabla comparativa de las dos versiones del editor de terrenos.

Versión	Plataformas	Consciencia de grupo	Modalidades
Primera	iOS	No	Realidad aumentada
Segunda	iOS, Android y navegador Web	Sí	Realidad aumentada y estándar

Plataformas

Mientras que la primera versión se implementó exclusivamente para iOS, la segunda se desarrolló en Unity, por lo cual se puede instalar en varias plataformas, en particular utilizamos las plataformas iOS, Android y PC a través de un navegador Web. Respecto a esta última plataforma, el editor se generó utilizando *WebGL*, que es la versión para navegador de *OpenGL*. Dicho formato está quedando en desuso en Unity y la alternativa actual es generar las aplicaciones en *HTML 5*. Cabe mencionar que la aplicación generada en *WebGL* sólo funciona en navegadores de PC, no así en los navegadores de los dispositivos móviles utilizados, i.e., iOS y Android.

Widgets de consciencia de grupo

Una mejora importante que se agregó a la segunda versión del editor fue la implementación de un *widget* de consciencia de grupo, que muestra los trazos de cada participante mediante una pequeña caja que funge como marcador de la coordenada sobre el terreno, como puede verse en la Figura 8.7. Este *widget* no señala la posición en el espacio 3D del participante correspondiente, sino de sus acciones sobre el terreno. Muestra además, de manera visual, ciertos aspectos relevantes del estado de cada colaborador representado: el nombre del participante, la operación que tiene seleccionada y el tamaño actual de su brocha.

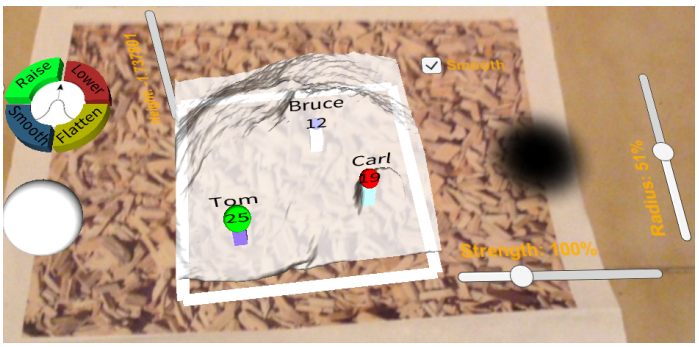


Figura 8.7: Captura de pantalla que muestra los *widgets* de consciencia de grupo para tres colaboradores.

Obsérvese, en la Figura 8.7, que el nombre de cada participante se proporciona en una etiqueta de texto, la operación seleccionada se muestra codificada en el color de la esfera ubicada sobre la caja del *widget* y el tamaño de la brocha se expresa, a escala, en el tamaño de dicha esfera. Además, se muestra el tamaño en pixeles en otra etiqueta colocada en la esfera. El color asignado a cada operación se corresponde con el del botón en el *widget* de selección de operación, el cual se describe en una subsección posterior. En este ejemplo, el colaborador Tom tiene seleccionada la operación *Raise* y una brocha de 25 pixeles de radio, mientras que Bruce tiene seleccionada la operación *Smooth* y una brocha con radio de 12 pixeles.

Las otras propiedades no son mostradas en esta versión, por ser menos importantes, pero es posible deducirlas al observar la respuesta del terreno a sus acciones. Por ejemplo, si la brocha está configurada con bordes suaves, el terreno cambiará de manera distinta que si estuviera con bordes duros. También, la velocidad de respuesta del terreno depende directamente de la fuerza del trazo. No hemos querido saturar el *widget* de consciencia con más información, debido al poco espacio disponible, sobre todo si se conectan varios usuarios y la vista se abarrota.

Modalidades de visualización

Nótese que, en la primera versión, la escena se visualiza solamente con realidad aumentada. Al observar a los usuarios interactuar con el editor de esta manera, pudimos darnos cuenta de los inconvenientes que se mencionaron en la Sección 4.6. En la segunda versión, se ofrecen las dos modalidades de visualización del editor: estándar y con realidad aumentada. Se implementó un mecanismo de selección, que se basa en utilizar escenas distintas para cada modalidad. En una escena inicial, se colocan dos botones, al pulsar en uno de ellos se abre la escena con realidad aumentada y al pulsar en el otro se abre la escena en modalidad estándar.

En la Figura 8.8, se muestra la interfaz de usuario de la segunda versión del editor. A continuación se describe la implementación en Unity de los *widgets* mostrados en esta figura.

Figura 8.8: Captura de pantalla de la interfaz de usuario de la segunda versión del editor de terrenos. Se indican los *widgets* del terreno, de selección de operación y de rotación del terreno, así como el *sprite* de la brocha.

Widget de datos del terreno

El *widget* de datos, que representa al terreno, cuenta con un componente de Unity, llamado `BoxCollider`, responsable de capturar los toques usando *TouchScript*, como se describe en la Sección 5.3. Cuenta también con un componente `Mesh`, que contiene toda la información de la malla. Al arranque, se sustituye el contenido de este componente `Mesh`, que comienza teniendo un solo cuadrado, por una rejilla de 100×100

vértices, utilizando el mismo algoritmo de la primera versión (el Algoritmo 1 de la Sección 8.4.3), pero ahora implementado en C#.

Widget **de selección**

El concepto de la barra de herramientas, que se utiliza en el primer diseño, evolucionó en la segunda versión al del panel de botones de la Sección 6.2.2 aunque, de hecho, la funcionalidad es la misma para ambos conceptos. Al lado izquierdo de la Figura 8.8, se encuentra el *widget* de selección de operación. Cabe notar que se cambió la disposición y forma de los botones por un diseño más agradable, inspirado en una paleta radial para la selección de colores. A cada botón se agregó una etiqueta de texto con el nombre de la operación que le corresponde y tiene asignado un color primario distintivo: verde para la operación *Raise*, rojo para *Lower*, azul para *Smooth* y amarillo para *Flatten*. Cuando se selecciona la operación deseada, pulsando sobre el botón correspondiente, se resalta el color incrementado su brillo y se muestra, en la parte central del *widget*, el ícono correspondiente a la operación, como se muestra en la Figura 8.9. Sólo una operación puede estar seleccionada a la vez.

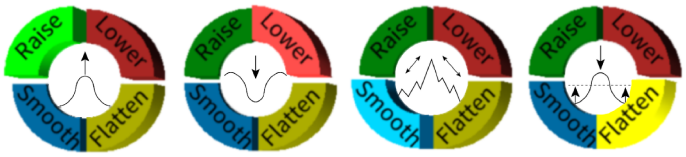


Figura 8.9: *Widget* de selección de operación.

Widget **de rotación**

En la versión previa del editor, observamos que los usuarios buscaban una manera de girar el terreno, sin tener que moverse alrededor del marcador. Así que, en la segunda versión, agregamos un *widget* de rotación. Se trata de la esfera blanca que se encuentra en la esquina inferior izquierda de la Figura 8.8, la cual funciona como un control de *trackball* [3], que permite girar el terreno sobre sus ejes X y Z. Este *widget* es útil también cuando el usuario ha elegido el modo de visualización estándar, sin realidad aumentada. Para este modo, también se agregó un control deslizante para acercar y alejar la cámara virtual (haciendo un *zoom*), de modo que el usuario pueda observar la forma del terreno desde cualquier perspectiva.

Sprite **de la brocha**

Este *widget* se comporta igual que en la primera versión del editor, pero se implementó en Unity a partir del *sprite* de imagen de la Sección 6.2.3. Está ubicado al

[3]Un control de *trackball* consiste en una pequeña bola, colocada dentro de una cavidad, de manera que, al girarse manualmente, permite mover el cursor en la pantalla de una computadora, de la misma manera que se hace utilizando un ratón.

lado derecho del terreno y la imagen mostrada indica la forma de la brocha seleccionada. Se le agregaron dos controles deslizantes y una casilla de verificación de Unity para controlar sus propiedades, aunque también se pueden modificar mediante gestos táctiles.

El tamaño de la brocha se puede cambiar con el respectivo control deslizante, o por medio de un gesto de pellizco sobre el *sprite*, causando que éste cambie de tamaño de acuerdo al valor seleccionado. El otro control deslizante controla la fuerza del trazo y, además, se da una indicación visual de este valor a través de la transparencia del *sprite*: mientras más transparente se vea, menor es la fuerza del trazo. Con la casilla de verificación se controla un valor booleano, que indica si la brocha es de bordes suaves o duros.

Indicador del valor de *Flatten*

Un tercer control deslizante controla el valor para la operación *Flatten* y se encuentra alineado con el eje vertical del terreno. Al mover este control, también se desplaza un plano indicador, hacia arriba o abajo, hasta la altura seleccionada, para indicarla visualmente al usuario en el mismo contexto geométrico del terreno.

8.4. Implementación de la primera versión

La primera versión del editor se implementó en *Objective C*, para ejecutarse sobre la plataforma iOS. En la Figura 8.10, se muestra la estructura de clases de dicha implementación. Hemos separado las clases en tres grupos: las clases responsables de la funcionalidad de realidad aumentada, tomadas del *SDK* de Vuforia, las que realizan las funciones de red junto con la publicación del servicio Web y finalmente, las dedicadas a la manipulación del DEM. También se indica con colores diferentes cuáles clases fueron implementadas desde cero, cuáles fueron modificadas para satisfacer nuestros requerimientos y aquellas que se dejaron sin modificar.

Las clases encargadas de mostrar y administrar las ventanas de una aplicación iOS son llamadas *vistas*, las cuales no deben ser confundidas, en este contexto, con las vistas o perspectivas de una cámara virtual en la escena 3D. La clase que maneja la ventana principal de la aplicación se conoce como vista principal (*main view*).

En el *SDK* de Vuforia, hay muchas clases dedicadas al control del ciclo de vida de la aplicación, pero solamente se mencionan aquí las que que son necesarias para implementar este prototipo. A la clase `AppDelegate` se delega el comportamiento de la aplicación, i.e., los procesos de arranque, pausa y reinicio, así como la rotación las vistas de la aplicación, de acuerdo a la orientación del dispositivo. `AppDelegate` se comunica con la clase `ParentViewController`, que controla la vista de la aplicación en la que se muestra el contenido. También utiliza las clases `BonjourBrowserController` y `BonjourRoomController`, descritas en la Sección 8.4.2, las cuales administran el manejo de las conexiones y la comunicación en red.

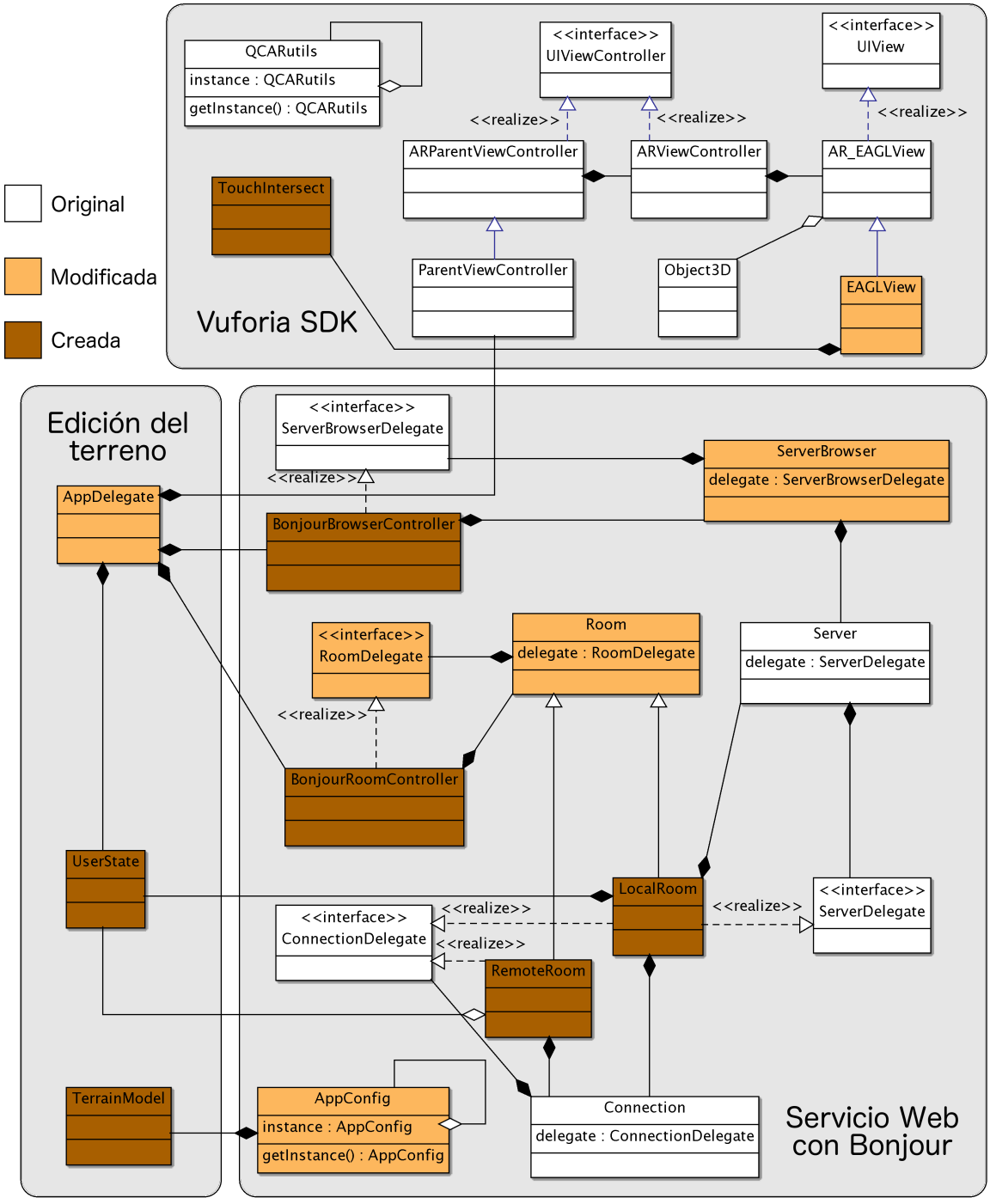


Figura 8.10: Diagrama de clases del editor de terrenos. Las clases de blanco se usaron sin modificar, las de color café claro se modificaron en cierta medida y las de café oscuro fueron implementadas completamente desde cero.

8.4.1. Clases de Vuforia

La clase `QCARutils`, del *SDK* de Vuforia, contiene el código de las utilidades para inicializar y administrar el ciclo de vida de la aplicación, junto con algunas funciones útiles relacionadas con el manejo de la realidad aumentada, tales como el acceso a los datos de los marcadores y a la matriz de transformación (ver Seccion 2.3.4). Esta clase sigue el patrón de diseño *Singleton* [27] que da acceso a las funciones de realidad aumentada desde cualquier lugar dentro de la aplicación.

La vista principal se define en la clase `EAGLView`, la cual hereda de `AR_EAGLView` toda la funcionalidad necesaria para configurar *OpenGL* y realizar el proceso de *render* 3D en pantalla, así como obtener el flujo de video de la cámara y mostrarlo como fondo en la vista principal. La clase `EAGLView` se debe modificar para mostrar en pantalla el terreno y los *widgets* de interfaz de usuario, así como manejar las interacciones táctiles necesarias. A esta vista la controla una instancia de `ParentViewController`, la cual se encuentra ligada a `AppDelegate` por agregación.

Cada objeto 3D, que va a dibujarse en la escena, está definido por una instancia de la clase `Object3D`, que contiene los arreglos necesarios para que se pueda dibujar en pantalla una malla poligonal, usando *OpenGL*: varios arreglos de flotantes para almacenar los vértices, las normales y las coordenadas de texturas; un arreglo de enteros para almacenar los índices de las caras triangulares y un apuntador a la clase `Texture`, encargada del acceso a la imagen que será usada como textura.

A continuación, se muestra la estructura de `Object3D` en *Objective C*:

```
@interface Object3D : NSObject {
    unsigned int numVertices;
    const float *vertices;
    const float *normals;
    const float *texCoords;
    unsigned int numIndices;
    const unsigned short *indices;
    Texture *texture;
}
```

8.4.2. Clases de Bonjour

Para implementar la funcionalidad de manejo de red, adaptamos una aplicación de conversaciones (*chat*) basada en *Bonjour*[4] y creada por Peter Bakhyryev (*peter@byteclub.com*) en el año 2009. Adaptamos algunas clases de dicha aplicación, dedicadas al manejo de conexiones en la red WiFi local, al envío de paquetes de datos, así como la publicación y descubrimiento de servicios Web, usando el protocolo *Bonjour*. Así pues, nuestra aplicación es capaz de arrancar un servidor en la red local y publicar la sesión de edición colaborativa, como un servicio Web. Además, puede buscar otros servicios publicados y conectarse a ellos como cliente.

[4]Bonjour es un protocolo de publicación y descubrimiento de servicios Web para iOS. (Fuente: https://developer.apple.com/bonjour/)

La clase `AppConfig`, de la Figura 8.10, es también un *Singleton*, que define los valores de configuración de la aplicación y provee acceso a los datos del DEM, a través de la clase `TerrainModel`, desde cualquier lugar de la aplicación.

La lógica de la sala de conversación se define en la clase `Room`, que permite diseminar mensajes de texto desde cualquier instancia de la aplicación, ya sea servidor o cliente, hacia todas las demás instancias conectadas. De esta clase `Room`, se extienden las clases `LocalRoom` y `RemoteRoom`, utilizadas respectivamente cuando la instancia de la aplicación actúa como servidor o como cliente, lo cual sólo puede decidirse en tiempo de ejecución. `LocalRoom` es responsable de crear y arrancar un servidor, usando la clase `Server`. Cada instancia cliente utiliza a `RemoteRoom` para administrar una conexión al servidor, usando la clase `Connection` para comunicarse con la demás instancias conectadas al servidor. Ambos tipos de `Room` tienen una referencia al arreglo de instancias de `UserState`, en el que se almacenan los estados de todos los participantes conectados.

Partiendo de dichas clases de la Figura 8.10, diseñadas para el manejo de salas de conversación sin un servidor centralizado, adaptamos las clases `AppConfig`, `ServerBrowser`, `Room` y `RoomDelegate` para generar y transmitir, a los colaboradores, los mensajes de texto que contienen las operaciones de edición y de cambio de estado. Para encapsular esta funcionalidad, se crearon dos clases de control con una API simplificada, para usarse desde `AppDelegate`: `BonjourBrowserController` y `BonjourRoomController`.

La clase `BonjourBrowserController` se encarga de la tarea de buscar servicios de *Bonjour* publicados en la red, que sean del mismo tipo que el del editor. Cuenta con los siguientes métodos:

```
- (void)activate;
- (void)initServerBrowser;
- (void)stopServerBrowser;
- (NSMutableArray*)getServers;
- (void)showServers;
- (void)joinRoom:(NSNetService*)server;
- (void)createNewRoom;
```

Básicamente, estos métodos proporcionan para funcionalidad de arrancar el buscador del servicio Web, detenerlo, obtener un arreglo de los servicios publicados, mostrarlos al usuario, unirse a uno de ellos o crear uno nuevo.

Una vez conectado a uno de los servicios, o luego de haber creado uno propio, se utiliza la clase `BonjourRoomController` para administrarlo y enviar mensajes a todos los participantes conectados al mismo. Sus funciones son:

```
- (void)activate;
- (void)exit;
- (UserState*)findUserWithName:(NSString*)usrName
        inSet:(NSMutableSet*)set;
- (void)sendMessage:(NSString*)msg;
- (void)sendCoordsX:(float)x Y:(float)y;
- (void)sendReset;
```

Tiene la funcionalidad de arrancar el servicio, detenerlo o desconectarse (dependiendo de si es propio o ajeno), encontrar el estado de un cierto participante, enviar un mensaje de operación, enviar las coordenadas de un toque en el terreno o un mensaje de reinicio del editor a los valores por defecto. De esta manera, es posible publicar el arreglo del DEM como un servicio Web de *Bonjour*, el cual los demás participantes pueden descubrir y conectarse para colaborar.

8.4.3. Modelado de la malla del terreno

Las clases que realizan la edición del terreno fueron creadas desde cero y se describen en los párrafos siguientes. La clase `TerrainModel` contiene un arreglo de flotantes para almacenar los valores de elevación del DEM del terreno compartido. En ella se implementan los métodos que aplican las cuatro operaciones de edición, descritas en la Sección 8.2, para modificar dicho arreglo, usando la brocha digital de cada participante. Esta clase tiene los métodos:

```
- (void)initMesh;
- (void)applyOperationAtLocation:(CGPoint)location
        UsingState:(UserState*)state;
- (void)raiseTerrainAtLocation:(CGPoint)location
        UsingState:(UserState*)state;
- (void)lowerTerrainAtLocation:(CGPoint)location
        UsingState:(UserState*)state;
- (void)smoothTerrainAtLocation:(CGPoint)location
        UsingState:(UserState*)state;
- (void)flattenTerrainAtLocation:(CGPoint)location
        UsingState:(UserState*)state;
- (void)resetTerrain;
```

Con estos métodos, se puede crear y manipular la malla del terreno, junto con el arreglo del DEM, así como restablecerlo a su estado por defecto. La malla que representa gráficamente al terreno es una rejilla de 100×100 vértices y es generada al arranque por el método `initMesh()` del Algoritmo 1.

La brocha digital se implementa mediante un arreglo dinámico de flotantes, conocido como máscara, el cual se inicializa a partir de los parámetros establecidos por el usuario: el radio y la forma de la brocha, utilizando el método `initMask()` del Algoritmo 2. Estos parámetros de la brocha, junto con otras propiedades de cada colaborador, se almacenan en la clase `UserState` que consta de: nombre del participante, máscara de la brocha, radio de la brocha, forma de la brocha, fuerza del trazo y valor de aplanado para la operación *Flatten*.

`UserState` cuenta con los métodos para crear y modificar dinámicamente su propia máscara, en función de los valores de la brocha, elegidos en tiempo de ejecución, para poder usarla en las operaciones de edición. Además, cuando el terreno se comparte entre varios colaboradores, se almacena una copia del estado de cada uno de ellos, en un arreglo de instancias de `UserState`.

Método `initMesh`

Entrada: terrainResX, terrainResY: el tamaño de la rejilla en valores enteros, texCoordFactorS, texCoordFactorT: el factor de escala en las coordenadas de textura (s,t), texCoordOffsetS, texCoordOffsetT: el offset en las coordenadas de textura (s,t)

Salida: Los arreglos contenidos en una instancia de Object3D, inicializados de manera dinámica: vertices, texCoords e indices (aquí se omite el arreglo de normales)

```
// Aparta memoria para los arreglos;
numVertices ← terrainResX * terrainResY;
vertices ← (GLfloat *)malloc(3*numVertices*sizeof(GLfloat));
texCoords ← (GLfloat *)malloc(2*numVertices *sizeof(GLfloat));
numIndices ← (terrainResX-1)*(terrainResY-1);
indices ← (GLushort *)malloc(numIndices *sizeof(GLushort));
for i ← 0 to terrainResY do
    for j ← 0 to terrainResX do
        // Calcula arreglo de vértices;
        vertices[(i*terrainResX+j)*3+0] ← -1 + j*(2/(terrainResX-1));
        vertices[(i*terrainResX+j)*3+1] ← 1 - i*(2/(terrainResY-1));
        vertices[(i*terrainResX+j)*3+2] ← 0;
        // Calcula arreglo de coordenadas de textura;
        texCoords[(i*terrainResX+j)*2+0] ← (float)i/(terrainResY-1) *
        texCoordFactorS + texCoordOffsetS;
        texCoords[(i*terrainResX+j)*2+1] ← (1 - (float)j/(terrainResX-1)) *
        texCoordFactorT + texCoordOffsetT;
    end
end
k ← 0;
for j ← 0 to terrainResX do
    for i ← 0 to terrainResY do
        // Calcula arreglo de índices, dos triángulos por cada cuadrado;
        aux ← n*j+i;
        // triangulo 1;
        indices[k++] ← aux;
        indices[k++] ← aux + 1 + n;
        indices[k++] ← aux + 1;
        // triangulo 2;
        indices[k++] ← aux;
        indices[k++] ← aux + n;
        indices[k++] ← aux + n + 1;
    end
end
```

Algoritmo 1: Método `initMesh()` para inicializar la malla del terreno.

La clase `UserState` define la estructura de datos para almacenar el estado de cada participante, así como los métodos para modificarla, los cuales se muestran en el listado siguiente:

```
@interface UserState : NSObject {
        NSString* name;
        NSString* conn;
        unsigned int currentOP;
        float *brushMask;
        int brushRadius;
        BOOL brushIsSmooth;
        float brushStrength;
        float flattenValue;
}

- (id)initWithName:(NSString*)newName
         andConnection:(Connection*)newConn;
- (void)updateState:(NSDictionary*)packet;
- (void)resetState;
- (void)initMask;
- (void)setRadius:(int)val;
        . . .
- (void)setSmooth:(BOOL)val;
@end
```

Una operación sobre el terreno en la coordenada (x, y) se aplica mediante el método `applyOperation()` del Algoritmo 3, la cual utiliza los métodos `raiseTerrain()`, `lowerTerrain()`, `smoothTerrain()` y `flattenTerrain()`, descritas respectivamente en los algoritmos 4, 5, 6 y 7.

8.4.4. Proyección de gestos táctiles

La vista principal recibe los gestos táctiles (*multi-touch*) que se generan sobre la pantalla bidimensional del dispositivo. Para poder usarlos dentro de la escena 3D, es necesario realizar una proyección de las coordenadas de la pantalla a las coordenadas del marcador. Con este propósito, creamos la clase `TouchIntersect` que contiene las utilidades necesarias para calcular dicha proyección. Dado que colocamos los *widgets* de interfaz y la malla del terreno sobre el plano del marcador, es posible determinar sobre qué objeto está tocando el usuario y obtener las coordenadas del toque.

En el Algoritmo 9, se muestra el método `projectScreenPointToPlane()` que proyecta un evento de toque desde la pantalla 2D del dispositivo, hacia un punto 3D sobre el plano del marcador, con ayuda de la matriz de transformación, que calcula Vuforia, y utilizando el método `linePlaneIntersection()` del Algoritmo 8.

Método `initMask`
Entrada: radius: el radio en pixeles, C: una constante que depende de la escala vertical del terreno, softBrush: booleano que indica si la brocha será de contorno duro o suavizado
Salida: brushMask: el arreglo de coeficientes de la máscara de la brocha

```
// Libera el espacio asignado previamente a la máscara;
free (brushMask);
// Asigna nuevo tamaño;
brushMask ← (float *)malloc((2*radius+1)*(2*radius+1)*sizeof(float));
for y ← 0 to 2*radius do
    for x ← 0 to 2*radius do
        distance ← √((x − radius)² + (y − radius)²);
        if distance ≤ radius then
            if softBrush then
                factor ← distance/radius;
                brushMask[y*(2*radius+1)+x] ← (cos(factor*PI)+1.f) * C;
            else
                brushMask[y*(2*radius+1)+x] ← C;
            end
        else
            brushMask[y*(2*radius+1)+x] ← 0;
        end
    end
end
```

Algoritmo 2: Método `initMask()` para inicializar la máscara de un participante.

Método `applyOperation`
Entrada: state: el estado del participante que aplica la operación, location: las coordenadas normalizadas dentro del terreno, terrainResX, terrainResY: el tamaño de la rejilla en valores enteros
Salida: heightMap: el arreglo de valores del mapa, vertices: el arreglo de vértices de la malla del terreno

```
switch state.currentOperation do
    Case RAISE: heightMap ← raiseTerrain (state, location, terrainResX,
    terrainResY, vertices); break;
    Case LOWER: heightMap ← lowerTerrain (state, location, terrainResX,
    terrainResY, vertices); break;
    Case SMOOTH: heightMap ← smoothTerrain (state, location, terrainResX,
    terrainResY, vertices); break;
    Case FLATTEN: heightMap ← flattenTerrain (state, location, terrainResX,
    terrainResY, vertices); break;
endsw
```

Algoritmo 3: Método `applyOperation()` para aplicar la operación seleccionada en el estado de un participante, en las coordenadas dadas.

Método `raiseTerrain`
Entrada: state: el estado de un participante que aplica la operación, location: las coordenadas normalizadas dentro del terreno, terrainResX, terrainResY: el tamaño de la rejilla en valores enteros
Salida: heightMap: el arreglo de valores del mapa, vertices: el arreglo de vértices de la malla del terreno

```
// Encuentra el índice a partir de las coordenadas del toque;
xIndex ← (int) (location.x*terrainResX);
yIndex ← (int) (terrainResY-location.y*terrainResY);
// Aplica la operación;
for y ← yIndex - state.brushRadius to yIndex + state.brushRadius do
    for x ← xIndex - state.brushRadius to xIndex + state.brushRadius do
        if x ≥ 0 and x < terrainResX and y ≥ 0 and y < terrainResY then
            heightMap[y*terrainResX+x] ← heightMap[y*terrainResX + x] +
            state.brushMask[(y - (yIndex - state.brushRadius))*(2*state.brushRadius
            + 1) + x - (xIndex - state.brushRadius)]*state.brushStrenght;
            // Actualiza la malla del terreno;
            vertices[(y*terrainResX+x)*3+2] ← heightMap[y*terrainResX+x];
        end
    end
end
```

Algoritmo 4: Método `raiseTerrain()`.

Método `lowerTerrain`
Entrada: state: el estado un participante que aplica la operación, location: las coordenadas normalizadas dentro del terreno, terrainResX, terrainResY: el tamaño de la rejilla en valores enteros
Salida: heightMap: el arreglo de valores del mapa, vertices: el arreglo de vértices de la malla del terreno

```
// Encuentra el índice a partir de las coordenadas del toque;
xIndex ← (int) (location.x*terrainResX);
yIndex ← (int) (terrainResY-location.y*terrainResY);
// Aplica la operación;
for y ← yIndex - state.brushRadius to yIndex + state.brushRadius do
    for x ← xIndex - state.brushRadius to xIndex + state.brushRadius do
        if x ≥ 0 and x < terrainResX and y ≥ 0 and y < terrainResY then
            heightMap[y*terrainResX+x] ← heightMap[y*terrainResX + x] -
            state.brushMask[(y - (yIndex - state.brushRadius))*(2*state.brushRadius
            + 1) + x - (xIndex - state.brushRadius)]*state.brushStrenght;
            // Actualiza la malla del terreno;
            vertices[(y*terrainResX+x)*3+2] ← heightMap[y*terrainResX+x];
        end
    end
end
```

Algoritmo 5: Método `lowerTerrain()`.

Método `smoothTerrain`
Entrada: state: el estado un participante que aplica la operación, location: las coordenadas normalizadas dentro del terreno, terrainResX, terrainResY: el tamaño de la rejilla en valores enteros
Salida: heightMap: el arreglo de valores del mapa, vertices: el arreglo de vértices de la malla del terreno

```
// Encuentra el índice a partir de las coordenadas del toque;
xIndex ← (int) (location.x*terrainResX);
yIndex ← (int) (terrainResY-location.y*terrainResY);
// Calcula el promedio dentro del rango de la brocha;
acum ← 0;
cont ← 1;
for y ← yIndex - state.brushRadius to yIndex + state.brushRadius do
    for x ← xIndex - state.brushRadius to xIndex + state.brushRadius do
        if x ≥ 0 and x < terrainResX and y ≥ 0 and y < terrainResY then
            acum ← acum + heightMap[y*terrainResX+x];
            cont ← cont + 1;
        end
    end
end
acum ← acum / cont;
// Aplica la operación;
for y ← yIndex - state.brushRadius to yIndex + state.brushRadius do
    for x ← xIndex - state.brushRadius to xIndex + state.brushRadius do
        if x ≥ 0 and x < terrainResX and y ≥ 0 and y < terrainResY then
            magnitude ← acum - heightMap[y*terrainResX+x];
            heightMap[y*terrainResX+x] ← heightMap[y*terrainResX + x] +
            magnitude*state.brushMask[(y - (yIndex -
            state.brushRadius))*(2*state.brushRadius + 1) + x - (xIndex -
            state.brushRadius)]*state.brushStrenght;
            // Actualiza la malla del terreno;
            vertices[(y*terrainResX+x)*3+2] ← heightMap[y*terrainResX+x];
        end
    end
end
```

Algoritmo 6: Método `smoothTerrain()`.

Método `flattenTerrain`
Entrada: state: el estado un participante que aplica la operación, location: las coordenadas normalizadas dentro del terreno, terrainResX, terrainResY: el tamaño de la rejilla en valores enteros
Salida: heightMap: el arreglo de valores del mapa, vertices: el arreglo de vértices de la malla del terreno

```
// Encuentra el índice a partir de las coordenadas del toque;
xIndex ← (int) (location.x*terrainResX);
yIndex ← (int) (terrainResY-location.y*terrainResY);
// Aplica la operación;
for y ← yIndex - state.brushRadius to yIndex + state.brushRadius do
    for x ← xIndex - state.brushRadius to xIndex + state.brushRadius do
        if x ≥ 0 and x < terrainResX and y ≥ 0 and y < terrainResY then
            magnitude ← state.flattenValue - heightMap[y*terrainResX+x];
            heightMap[y*terrainResX+x] ← heightMap[y*terrainResX + x] +
            magnitude*state.brushMask[(y - (yIndex -
            state.brushRadius))*(2*state.brushRadius + 1) + x - (xIndex -
            state.brushRadius)]*state.brushStrenght;
            // Actualiza la malla del terreno;
            vertices[(y*terrainResX+x)*3+2] ← heightMap[y*terrainResX+x];
        end
    end
end
```

Algoritmo 7: Método `flattenTerrain()`.

Método `linePlaneIntersection`
Entrada: pointA, pointB: puntos inicial y final de la línea, pointP: cualquier punto sobre el plano, normal: vector normal al plano, ϵ: un valor de umbral pequeño (i.e. 0.0001)
Salida: intersection: el punto de intersección, result: valor booleano para indicar si hubo éxito

```
linedir ← pointB - pointA;
linedir ← linedir / ||linedir||;
planedir ← pointP - pointA;
n ← normal · planedir;
d ← normal · linedir;
if |d| < ϵ then
    // La línea es prácticamente paralela al plano;
    Devuelve falso;
end
dist ← n / d;
offset ← dist * linedir;
intersection ← pointA + offset;
Devuelve verdadero;
```

Algoritmo 8: Método `linePlaneIntersection()` para calcular el punto de intersección entre la línea dada por dos puntos y un plano.

Método `projectScreenPointToPlane`
Entrada: point: punto en la pantalla 2D
Salida: intersection: el punto de intersección 3D
// Obtener de Vuforia el tamaño de la pantalla y de la vista (*viewport*);
halfScreenWidth ← QCARutils.viewSize.width / 2;
halfScreenHeight ← QCARutils.viewSize.height / 2;
halfViewportWidth ← QCAR.getVideoBackgroundConfig().mSize.x / 2;
halfViewportHeight ← QCAR.getVideoBackgroundConfig().mSize.y / 2;
// Normalizar las coordenadas de la pantalla;
x ← (QCARutils.contentScalingFactor * point.x - halfScreenWidth) /
halfViewportWidth;
y ← -(QCARutils.contentScalingFactor * point.y - halfScreenHeight) /
halfViewportHeight;
// Obtener de Vuforia la matriz de proyección;
$M_{projection}$ ← [QCARutils getInstance].projectionMatrix;
// Calcular la inversa de $M_{projection}$;
$M_{inv} \leftarrow M_{projection}^{-1}$;
// Elegir un punto cercano a la cámara y uno lejano en coordenadas homogéneas, y transformarlo a las coordenadas de la escena;
near ← M_{inv} * (x, y, -1, 1);
far ← M_{inv} * (x, y, 1, 1);
// Normalizar los puntos homogéneos;
near ← near / near.data[3];
far ← far / far.data[3];
// Obtener de Vuforia la matriz de transformación;
cameraPose ← QCAR::Renderer::getTrackableResult(0) → getPose();
$M_{modelview}$ ← QCAR::Tool::convertPose2GLMatrix(cameraPose);
// Calcular la inversa de $M_{modelview}$;
$M_{inv} \leftarrow M_{modelview}^{-1}$;
// Transformar de las coordenadas de la cámara a las de la escena ;
nearWorld ← M_{inv} * near;
farWorld ← M_{inv} * far;
// Elegimos un punto sobre el plano del marcador;
center ← (0,0,0);
// Tomamos el vector normal del plano del marcador;
normal ← (0,0,1);
// Calcular la intersección entre la línea dada por dos puntos y un plano;
result ← **linePlaneIntersection** (nearWorld, farWorld, center, normal, intersection);
if *result es verdadero* **then**
  Devuelve intersection;
else
  Devuelve nulo;
end

Algoritmo 9: Método `projectScreenPointToPlane()` para proyectar un punto en la pantalla 2D hacia un punto sobre el plano 3D del marcador.

Sesión de edición colaborativa típica

Una sesión colaborativa típicamente comienza cuando los participantes se encuentran reunidos en un mismo lugar, que cuenta con servicio de red WiFi local. Los participantes conectan sus iPads a la red local y designan a uno de ellos como coordinador para esa sesión, el cual es responsable de crear la sesión y publicarla como un servicio Web, usando el protocolo *Bonjour*.

El coordinador de sesión realiza los siguientes pasos:

- Inicia la aplicación y proporciona su nombre de usuario único o confirma el que se encuentra almacenado.
- Pulsa el botón de menú y selecciona la opción para compartir el terreno (*Share Terrain*).
- Espera a que el resto de los participantes se una a la sesión.

Entonces, los demás participantes se unen a la sesión publicada como servicio de *Bonjour*, quedando conectados al coordinador. Ellos deben seguir los siguientes pasos:

- Iniciar la aplicación y proporcionar un nombre de usuario único o confirmar el que se encuentra almacenado.
- Pulsar el botón de menú y elegir la opción que tiene el nombre del coordinador.
- Esperar a que los demás participantes se unan a la sesión.

En las presentes versiones del editor, los usuarios deben elegir nombres únicos, pues son usados por la aplicación como identificadores para encontrar el estado de cada participante y aplicar con él las operaciones correctamente. Una vez que se encuentran conectados los participantes, la sesión de edición puede comenzar.

Al comenzar la sesión de edición, todos los participantes deben tener el terreno y sus estados exactamente con los mismos valores, i.e., los *widgets* del editor deben comenzar con un estado consistente. Esto se debe a que, en el prototipo actual, aún no se han implementado mecanismos para sincronizar los estados o para acomodar a usuarios que se conectan tardíamente, llamados también recién llegados (*latecomers*) [43, 44].

Para garantizar que todos comiencen con el estado inicial por defecto del editor, el coordinador puede enviar un mensaje de restaurar los valores por defecto (*reset*). Durante la sesión, suponemos que los paquetes de datos no se pierden en la red, que los participantes no abandonarán la sesión y luego volverán a conectarse (en especial el coordinador, pues se cerraría la sesión completamente) y que la red admite la publicación de servicios Web de *Bonjour*.

Al finalizar la sesión, los participantes pueden abandonar la sesión eligiendo la opción *Leave Terrain* del menú. También, cuando el coordinador cierra la sesión, todas las instancias conectadas se desconectan automáticamente.

8.5. Implementación de la segunda versión

La segunda versión del editor de terrenos tiene la misma arquitectura mostrada en el Capítulo 4, implementada sobre el motor de juegos de Unity como se detalla en el Capítulo 5 y utilizando ciertos *widgets* cuyo diseño se describe en el Capítulo 6. En la Figura 8.11 se muestra el diagrama de clases de dicha implementación.

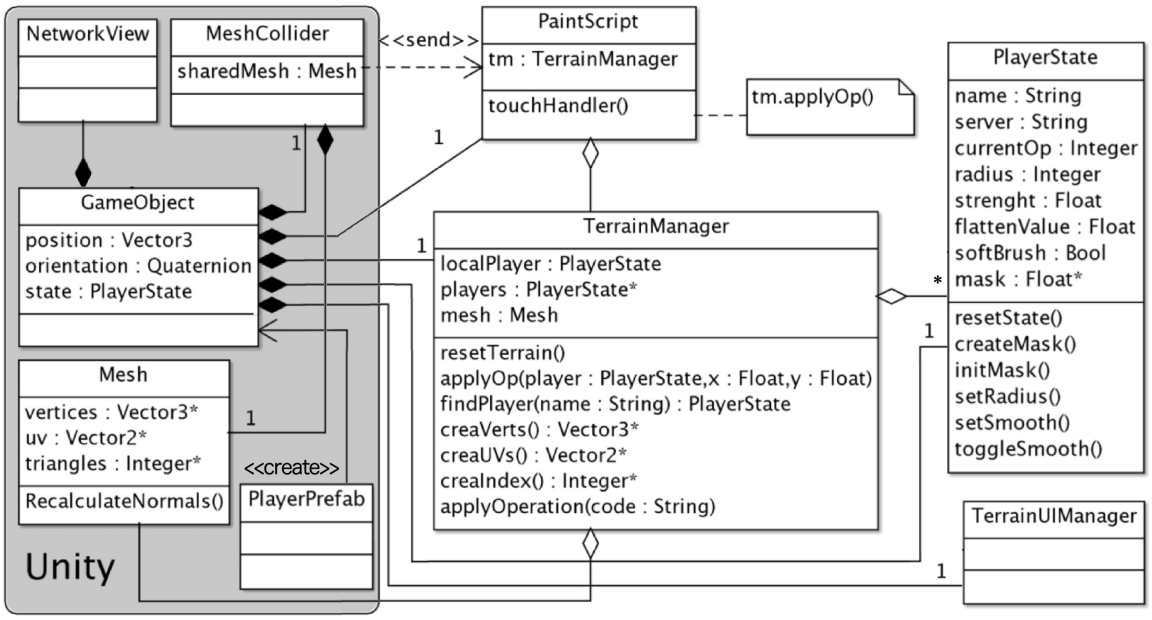


Figura 8.11: Diagrama de clases para la funcionalidad de edición del terreno, en la segunda versión del editor.

Implementación del *widget* del terreno

El *widget* de datos que representa al terreno en la escena 3D, es una instancia de `GameObject`, que tiene asociado un componente `MeshCollider`, el cual sirve para detectar y manejar la colisión de eventos táctiles con la malla del objeto. Dado que el usuario puede interactuar pintando directamente sobre el *widget* del terreno, puede ser considerado como parte de la interfaz de usuario.

El comportamiento del *widget* del terreno viene dado en la clase `TerrainManager`. Es responsable de modificar la forma del terreno en tiempo real, cambiando los valores de la coordenada Z de sus vértices, de acuerdo a los valores almacenados en el DEM. La clase `PaintScript`, procesa los toques recibidos por `MeshCollider` y es responsable de mapear las coordenadas del toque hacia el índice correcto dentro del arreglo del DEM, y así poder aplicar la operación correspondiente en ese punto, mediante un llamado a `TerrainManager`. En esta última clase se almacena el arreglo del DEM y contiene los métodos para aplicar sobre éste las cuatro operaciones, utilizan-

do el estado de cualquier participante del arreglo *players* que contiene instancias de `PlayerState`.

La API de `TerrainManager` es prácticamente la misma que la de la primera versión y se utilizan los mismos algoritmos para el manejo de la malla, de las máscaras de los participantes y la aplicación de operaciones, pero implementados para Unity en C#. A continuación se listan los métodos de `TerrainManager`:

```
void applyOperation (PlayerState s, float x, float y)
void createTerrainMesh()
void resetTerrain()
PlayerState findPlayer (string name)
```

Implementación de los widgets de interfaz de usuario

En la Sección 8.3.2 hemos descrito el diseño de la interfaz de usuario del editor de terrenos, compuesta por un panel de cuatro botones, tres controles deslizantes, una casilla de verificación, un *widget* de rotación y un *sprite*. Todos ellos son instancias de `GameObject`, pero en lugar de asignarle una clase de comportamiento a cada uno, en este prototipo centralizamos el control de la interfaz de usuario en una sola clase, llamada `TerrainUIManager`, que controla todos los *widgets* de interfaz de usuario. Cuando el colaborador local interactúa con dichos *widgets*, se llaman los métodos correspondientes de `TerrainUIManager` destinados a realizar los cambios de estado, en la estructura `PlayerState` del colaborador local.

El *sprite* de la brocha se implementó con un *sprite* de imagen, que puede mostrar dos texturas: una para la brocha con bordes suaves y otra con bordes duros. Al *sprite* se agregaron los comportamientos de *TouchScript* para reconocimiento de los gestos `DoubleTapGesture` y `PinchGesture`, con los cuales se disparan los métodos de `TerrainUIManager` que cambian la forma y radio de la brocha.

8.6. Pruebas y resultados

En esta sección, se muestran las pruebas aplicadas a varios grupos de usuarios que editaron terrenos compartidos, usando ambas versiones del editor de terrenos. En los primeros experimentos, se tuvieron hasta cuatro iPads conectados via WiFi, debido a que el número de dispositivos iOS disponibles para las pruebas era limitado. En tanto que con la segunda versión es posible usar dispositivos tanto Android como iOS, con lo cual fue posible usar más dispositivos y efectuar algunas observaciones empíricas acerca del rendimiento del editor.

En principio, la arquitectura propuesta puede soportar una cantidad arbitraria de participantes, sin embargo durante las sesiones colaborativas utilizando la red local WiFi, se observó que la principal limitante al número de participantes es el ancho de banda disponible. Como ejemplo, durante las pruebas realizadas en una red WiFi pública, la red soportó hasta 13 usuarios conectados, antes de que se comenzaran a perder mensajes de operación o se saturara la red. Los problemas de saturación

ocurrieron en función de la hora del día, lo que sugiere que la saturación fue causada por el uso de datos de la red, por ser pública, no por la implementación del editor.

Por otra parte, es posible utilizar una *zona móvil* [5] en un teléfono inteligente con plataforma Android, en la cual permite la conexión de hasta 10 usuarios. Nótese que dicha limitación viene dada por el hardware, y no por la arquitectura del editor. De hecho, al utilizar esta solución de red WiFi *ad-hoc*, se obtiene una muy buena respuesta del editor, pues la red es privada.

En las pruebas descritas en la Sección 8.6.1, se mide la carga de trabajo percibida por los usuarios del editor, con el fin de verificar cuánto se les facilita o dificulta, en su caso, el uso del editor, durante la realización de una tarea de edición colaborativa. También se realizó un segundo conjunto de pruebas utilizando la herramienta *Attrakdiff* [6] con la cual es posible evaluar la usabilidad y el diseño de un producto interactivo, midiendo las cualidades hedónica y pragmática. Los resultados de dicha evaluación se presentan en la Sección 8.6.2. Luego, en la tercera ronda de pruebas, el propósito es investigar la necesidad de implementar un mecanismo de consistencia de datos en el editor de terrenos propuesto. Posteriomente, en la Sección 8.6.3, se describe la manera en la que se realizaron los experimentos y el cálculo de inconsistencias entre todas las instancias de la aplicación. En la Sección 8.6.4, se describen las opciones que se agregaron para facilitar la realización de las pruebas. Finalmente, se presentan los resultados obtenidos de diez experimentos con grupos de colaboradores realizando tres tareas diferentes, descritas en las Secciones 8.6.5, 8.6.6 y 8.6.7, respectivamente.

8.6.1. Índice de carga de trabajo percibida

Se midió la carga de trabajo percibida por ocho usuarios del editor de terrenos, utilizando el cuestionario *NASA Task Load indeX* (NASA-TLX). Se trata de una herramienta de evaluación que utiliza seis escalas, también llamadas dimensiones, para medir la carga de trabajo percibida por un individuo o un grupo, durante la ejecución de alguna tarea.

La prueba consta de dos partes. En la primera, los colaboradores evalúan la tarea en las escalas: *demanda mental*, *demanda física*, *demanda temporal*, *nivel de desempeño*, *nivel de frustración* y *nivel de esfuerzo*. Para cada escala, el usuario selecciona un valor dentro del intervalo $[0, 100] \subset \mathbb{N}$ con una separación de cinco unidades entre cada valor, dando un total de veintiún posibles calificaciones. Los resultados se muestran en la Figura 8.12. Es importante notar que lo que se mide es la carga de trabajo, así que una calificación grande significa una mayor carga de trabajo, lo cual consideramos como malo. Por otra parte, una calificación pequeña significa una carga de trabajo menor, lo cual es bueno.

[5] Mediante esta funcionalidad, el teléfono inteligente toma el papel de un punto de acceso WiFi, al cual se pueden conectar otros dispositivos, formando una red *ad-hoc*, la cual se puede declarar privada o pública.

[6] Disponible en: http://www.attrakdiff.de/index-en.html

En la parte final de la prueba, cada colaborador asigna un conjunto personalizado de pesos para las seis escalas, con el cual se calcula la calificación final ponderada. Los resultados ajustados con el peso de cada escala se muestran en la Figura 8.13 y la calificación final ponderada en la Figura 8.14.

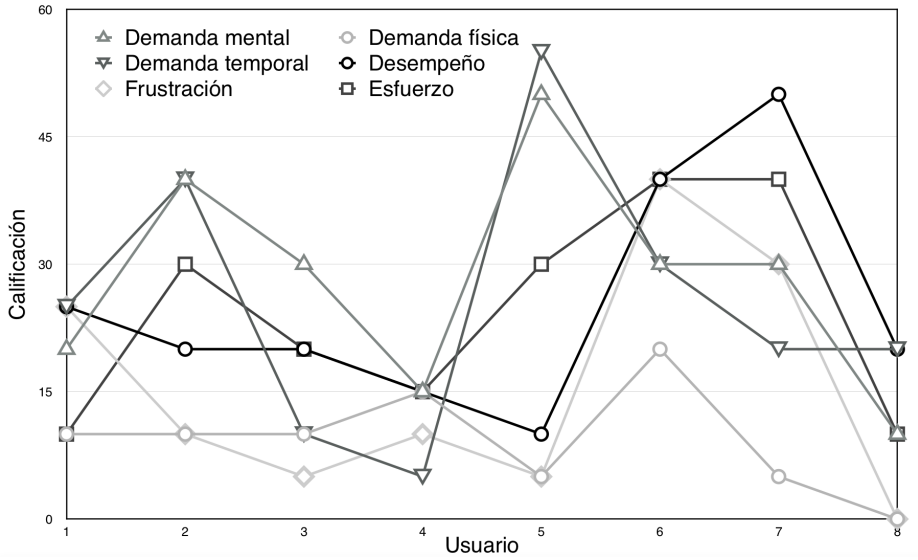


Figura 8.12: Resultados de las evaluaciones en las seis escalas.

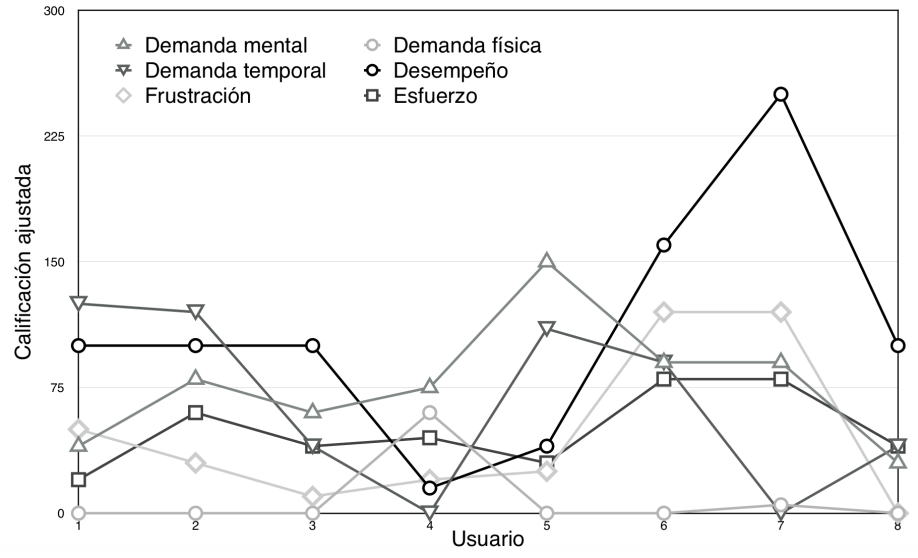


Figura 8.13: Resultados de las evaluaciones ajustadas en las seis escalas.

La calificación promedio para la carga de trabajo percibida por los ocho colaboradores, en la escala de 0 a 100, fue de $\mathbf{23.67 \pm 8.29}$. Con estos valores de la media y la desviación estándar, el intervalo de confianza del 68.3 % es $[\mathbf{15.38}, \mathbf{31.96}]$. En la Figura 8.15, se muestra gráficamente la calificación promedio y el correspondiente

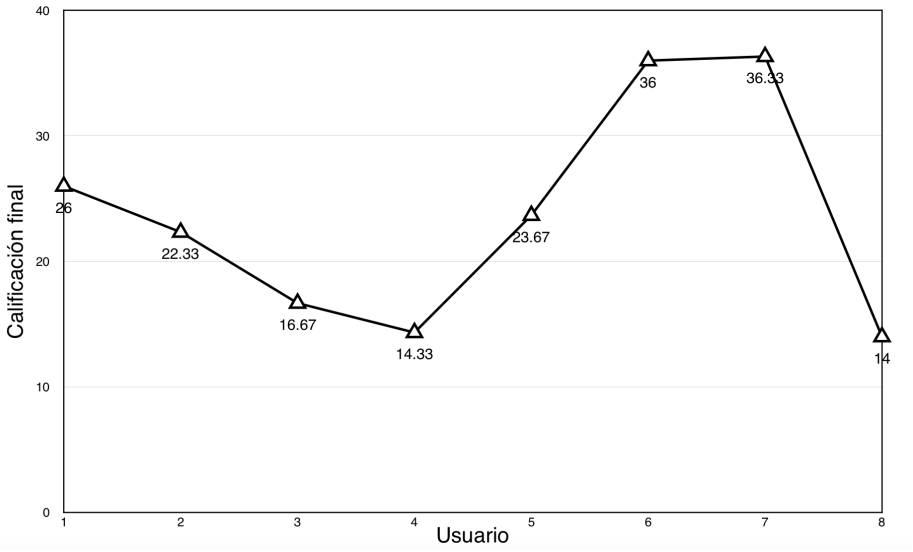


Figura 8.14: Resultados finales de las evaluaciones por usuario.

intervalo, lo que da una idea aproximada de cuán buena fue la percepción de los colaboradores acerca del editor.

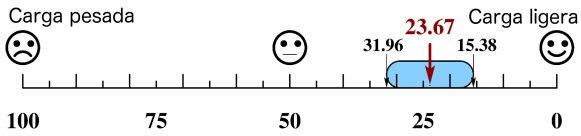


Figura 8.15: Resultado del cuestionario NASA-TLX para ocho usuarios del editor de terrenos: la calificación fue de $\mathbf{23.67 \pm 8.29}$, con un intervalo de confianza del 68.3 %.

8.6.2. Evaluación de las cualidades hedónica y pragmática

Las cualidades hedónica y pragmática son las medidas que utiliza la herramienta *AttrakDiff* para evaluar respectivamente el diseño y la usabilidad de una aplicación interactiva. Se evaluó con grupos de trece usuarios, usando diversos dispositivos. Se les pidió que utilizaran el editor de terrenos creado con el marco de desarrollo ShAREd, junto con otro editor similar, con el fin de comparar sus cualidades. La evaluación consiste en asignarle un peso a una serie de pares de palabras opuestas, indicando qué tanto se puede describir la aplicación con una u otra palabra.

En una primera etapa, los usuarios utilizaron el editor de terrenos para navegador Web de Rob Chadwick (*WebGL Terrain Editor*)[7] durante unos minutos, para familiarizarse con su uso. Luego se les pidió que modelaran una forma cónica con un cráter en la cúspide, similar al volcán Paricutín, la cual se eligió por ser relativamente

[7]Disponible en https://www.chromeexperiments.com/experiment/webgl-terrain-editor o en (copia local) http://computacion.cs.cinvestav.mx/~acortes/TerrainEditor-master/

sencilla. Este editor es gratuito y se eligió por ser lo más similar al prototipo pues, a pesar de que no es móvil, también tiene operaciones basadas en brochas que son análogas a las nuestras.

En la segunda etapa, se le pidió al grupo que hicieran la misma tarea con el editor propuesto, utilizarlo unos pocos minutos y luego modelar la forma del Paricutín, pero primero en el modo estándar sin realidad aumentada, con lo cual es muy similar al editor Web. Finalmente se repitió el experimento, pero esta vez utilizando el modo de realidad aumentada y un marcador impreso. Al final de cada uno de estos tres experimentos, cada usuario evaluó el editor en turno con *AttrakDiff*. El propósito de esto es comparar por pares los tres experimentos de la siguiente manera:

- Comparar el editor Web con el propuesto en modalidad estándar.
- Comparar el editor Web con el propuesto en modalidad de realidad aumentada.
- Comparar las dos modalidades del editor desarrollado: el modo estándar y el de realidad aumentada.

Los resultados de dichas comparaciones se muestran en las siguientes gráficas. En estas pruebas no se evalúan las habilidades de modelado de los participantes, sino su percepción en las dos cualidades medidas en la evaluación. En la Figura 8.16 se muestra la gráfica generada por *AttrakDiff* comparando el editor Web con el prototipo en modalidad estándar.

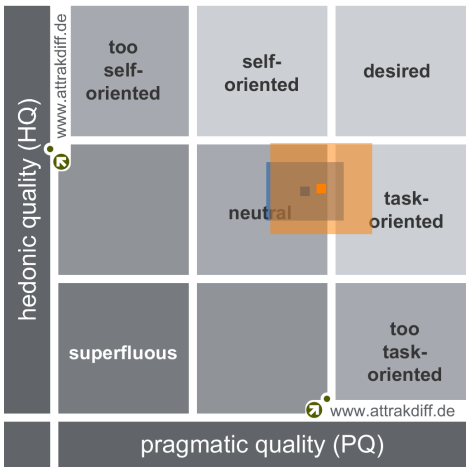


Figura 8.16: Comparación de los resultados del editor Web (recuadro naranja) contra el editor propuesto en modo estándar (recuadro azul).

Los dos rectángulos de color indican los resultados para ambos editores, en una escala de -3 a 3 en los dos ejes: la cualidad pragmática (*pragmatic quality*, PQ) en la horizontal y la hedónica (*hedonic quality*, HQ) en la vertical. Las coordenadas del centro de cada rectángulo indican el promedio obtenido en ambas cualidades y su tamaño indica los intervalos de confianza. Un rectángulo mayor indica que las calificaciones fueron más variadas que en un rectángulo pequeño.

El recuadro naranja indica los resultados del editor Web, que obtuvo una calificación de 0.87 ± 0.75 en la cualidad pragmática y 0.32 ± 0.66 en la cualidad hedónica. El recuadro azul corresponde al prototipo en modalidad estándar, el cual obtuvo una calificación de 0.64 ± 0.57 en la cualidad pragmática y 0.29 ± 0.43 en la cualidad hedónica. El dominio de la gráfica se divide en nueve regiones, etiquetadas algunas por categorías como *neutral*, *deseada*, *supérflua*, con las cuales se pretende calificar la aplicación interactiva. Como puede apreciarse, ambos editores se percibieron como neutrales y con sus centros muy cercanos, aunque tienden ligeramente hacia la región deseada, con un mayor avance en la usabilidad que en el diseño. Además, con el editor Web los usuarios muestran un mayor desacuerdo en ambas dimensiones.

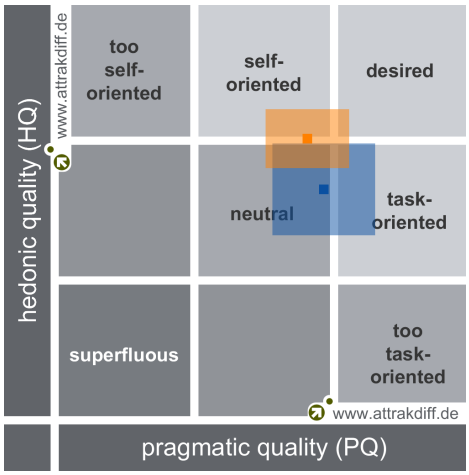


Figura 8.17: Comparación de los resultados del editor Web (recuadro azul) contra el presente editor en modo de realidad aumentada (recuadro naranja).

En la Figura 8.17 se muestran los resultados de comparar el editor Web con el prototipo usando realidad aumentada. El recuadro azul indica los resultados del editor Web, que obtuvo una calificación de 0.87 ± 0.75 en la cualidad pragmática y 0.32 ± 0.66 en la cualidad hedónica. El recuadro naranja corresponde al prototipo usando realidad aumentada, el cual obtuvo una calificación de 0.64 ± 0.60 en la cualidad pragmática y 1.07 ± 0.42 en la cualidad hedónica. En este caso, ambos editores se perciben neutrales, con una cualidad pragmática similar, pero con el uso de la realidad aumentada se tiene un impacto positivo en la cualidad hedónica, colocando al prototipo prácticamente en la frontera de la zona neutral. De nuevo, los usuarios muestran mayor desacuerdo con el editor Web que con el prototipo.

Finalmente, en la Figura 8.18, se muestra la comparación entre las dos modalidades del editor propuesto: estándar y con realidad aumentada. En esta gráfica se presentan los resultados de los trece colaboradores anteriores, junto con otros siete más, que realizaron este experimento solamente, dando un total de veinte usuarios, que es el máximo que permite la herramienta *Attrakdiff*. El recuadro naranja indica los resultados del prototipo en modo estándar, que obtuvo una calificación de 0.56 ± 0.43 en la cualidad pragmática y 0.26 ± 0.39 en la cualidad hedónica. El recuadro azul

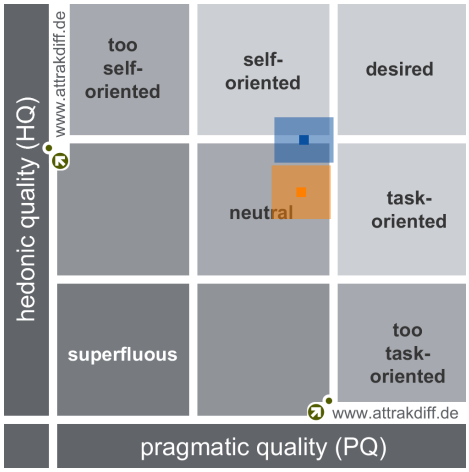


Figura 8.18: Comparación de los resultados de las dos modalidades del presente editor: modo estándar (recuadro naranja) y modo de realidad aumentada (recuadro azul).

corresponde al prototipo usando realidad aumentada, el cual obtuvo una calificación de 0.59 ± 0.43 en la cualidad pragmática y 1.03 ± 0.33 en la cualidad hedónica.

Nótese como la percepción en la cualidad pragmática es muy similar en las dos modalidades, mientras que en la cualidad hedónica se observa una mejora significativa. Con la participación de nuevos usuarios, se observa una reducción en la incertidumbre de la medición en ambas cualidades, en tanto que las calificaciones promedio prácticamente no cambian, indicando un mayor acuerdo entre los usuarios.

8.6.3. Medición de inconsistencias

Utilizando la primera versión del editor de terrenos, se evaluó la inconsistencia de datos acumulada durante diez sesiones colaborativas, en las que participaron diferentes grupos de usuarios. Se formaron grupos de dos, tres y cuatro participantes, con un iPad cada uno y se les pidió que explorasen el editor durante cinco minutos, con el fin de familiarizarse con su interfaz de usuario, prácticamente sin instrucción previa. Entonces, se les asignó una de las tres tareas de edición colaborativa, que se describen a continuación:

- Editar el terreno usando las cuatro operaciones disponibles, con el fin de medir las inconsistencias acumuladas (Sección 8.6.5).

- Editar el terreno usando solamente las operaciones *Raise* y *Lower*, con el fin de verificar la conmutatividad entre ellas (Sección 8.6.6).

- Editar el terreno usando las cuatro operaciones, pero con la meta grupal de crear la forma de una letra previamente acordada. La finalidad de esta tarea es observar los comportamientos auto-regulatorios que emergen de manera espontánea en un grupo (Sección 8.6.7).

Proponemos utilizar la varianza en cada punto de la superficie del terreno para definir una medida de las inconsistencias acumuladas entre las réplicas de los colaboradores, a la cual denominamos *error relativo*. Para calcularlo, supongamos que se tienen n participantes en una sesión colaborativa, editando los datos de un DEM en forma de un arreglo de tamaño $m = 100 \times 100$ flotantes. Los participantes realizan la tarea asignada durante aproximadamente diez minutos o hasta que quedan satisfechos, entonces cada uno de ellos envía el contenido de su arreglo local $h_k[i]$, donde $k \in [1, n]$ e $i \in [1, m]$, a un repositorio, con el fin de compararlos y analizar las diferencias entre ellos. Primero, se calcula el promedio $\bar{h}[i]$, y luego la varianza $\mathrm{Var}[i]$, para cada $i \in [1, m]$, como sigue:

$$\bar{h}[i] = \frac{1}{n} \sum_{k=1}^{n} h_k[i],$$

$$\mathrm{Var}[i] = \frac{1}{n} \sum_{k=1}^{n} (h_k[i] - \bar{h}[i])^2.$$

Finalmente, se estima el error relativo, denotado por E_{rel}, dividiendo la raíz cuadrada de la varianza máxima entre el tamaño del promedio del terreno. Definimos el tamaño de un terreno como la diferencia entre los valores de elevación máximo y mínimo:

$$E_{\mathrm{rel}} = \frac{\sqrt{\max_{\mathrm{i}}(\mathrm{Var}[i])}}{\max_{\mathrm{i}}(\bar{h}[i]) - \min_{\mathrm{i}}(\bar{h}[i])}.$$

Para visualizar correctamente los arreglos de la varianza, en las figuras de las secciones subsecuentes, se grafican los 100×100 valores en 3D, sobre una región cuadrada del plano XY, aunque internamente dichos arreglos sean unidimensionales.

8.6.4. Opciones para la realización de los experimentos

Para reducir el riesgo de introducir inconsistencias causadas por errores humanos, procuramos mantener simple el diseño de la interfaz de usuario y mostrar en el menú la menor cantidad posible de opciones, eligiendo de manera automática cuáles opciones activar, a partir del estado de conectividad y los objetivos experimentales. El estado de conectividad se determina verificando si la instancia de la aplicación es el coordinador de sesión o si se encuentra conectada a otra instancia como cliente. Durante los experimentos, se proporcionó a los usuarios una manera sencilla para conectarse a la misma sesión, activar únicamente las operaciones de edición que se van a utilizar y enviar los datos del DEM al mismo repositorio para su análisis. En particular, los datos se envían a la cuenta de correo electrónico del desarrollador.

Con este fin, se aplicaron las siguientes modificaciones:

- Se agregó un botón para mostrar el menu de opciones. En la aplicación, existe un único menú contextual con las opciones pertinentes, las cuales dependen del estado de conectividad del usuario.

- Se diseñó una manera sencilla de pedir el nombre del usuario actual y almacenarlo en un archivo de configuración. Aquí se debe tomar en cuenta que los dispositivos utilizados durante el experimento no son propiedad de los sujetos de prueba, de manera que el nombre del usuario cambia con frecuencia. Por esta razón, la aplicación debe confirmar el nombre de usuario al arranque.

- Dado que aún no se ha resuelto el problema de restauración (*reset*) concurrente del terreno, se evita mostrar dicha operación a los usuarios, con el fin de prevenir que alguno la ejecute accidentalmente. Se diseñó un mecanismo oculto, que se activa haciendo triple pulsación en un botón invisible.

- Es posible elegir cuáles herramientas de operación estarán activas durante una sesión, para restringir su uso por parte de los colaboradores durante el experimento. Durante la ejecución normal de la aplicación, no hay razones para ocultar una operación.

- Se agregó la opción de enviar por correo electrónico los datos del DEM, como un arreglo de flotantes. Cada correo se identifica mediante una tupla *(expID, coordName, userName)*, donde *expID* es el identificador del experimento, que es el número de minutos transcurridos desde alguna fecha elegida, *coordName* es el nombre del coordinador de sesión y *userName* es el nombre del propietario del arreglo enviado. Suponemos que cada experimento durará más de un minuto, de modo que el siguiente arreglo que se envíe deberá ocurrir en un minuto distinto.

8.6.5. Utilizando las cuatro operaciones

Se hicieron cuatro experimentos con la tarea de editar libremente el terreno compartido, con las cuatro operaciones activadas: *Raise*, *Lower*, *Flatten* y *Smooth*. Cuando varios participantes aplican las operaciones de edición, de manera concurrente en regiones traslapadas del terreno, puede ocurrir que se apliquen en orden diferente en cada instancia del editor. Esta situación puede provocar que, al finalizar la sesión colaborativa, los participantes tengan los datos de su DEM local inconsistentes entre sí.

En la Tabla 8.2, se muestra un resumen de los resultados para esta tarea. Por cada experimento, aparece el número de participantes y el error relativo entre los DEMs finales.

Tabla 8.2: Experimentos usando las cuatro operaciones.

Experimento	**I**	**III**	**V**	**VII**
Número de participantes	3	2	3	4
Error relativo (%)	15.4919	0.3873	9.7979	0.0030

Nótese que, para esta tarea, las inconsistencias encontradas son muy pequeñas (menores al 5 %) en dos experimentos, mientras que en los otros dos son mayores

(cerca del 15 %) pero aún son visualmente aceptables para aplicaciones de bosquejo, i.e., la diferencia en la forma de las representaciones gráficas de los terrenos es difícil de notar a simple vista. Las gráficas de la varianza para los experimentos I, III, V y VII se muestran en la Figura 8.19, en las cuales los puntos están coloreados en escala de grises, en función de su elevación, y las curvas de nivel se muestran sobre la superficie.

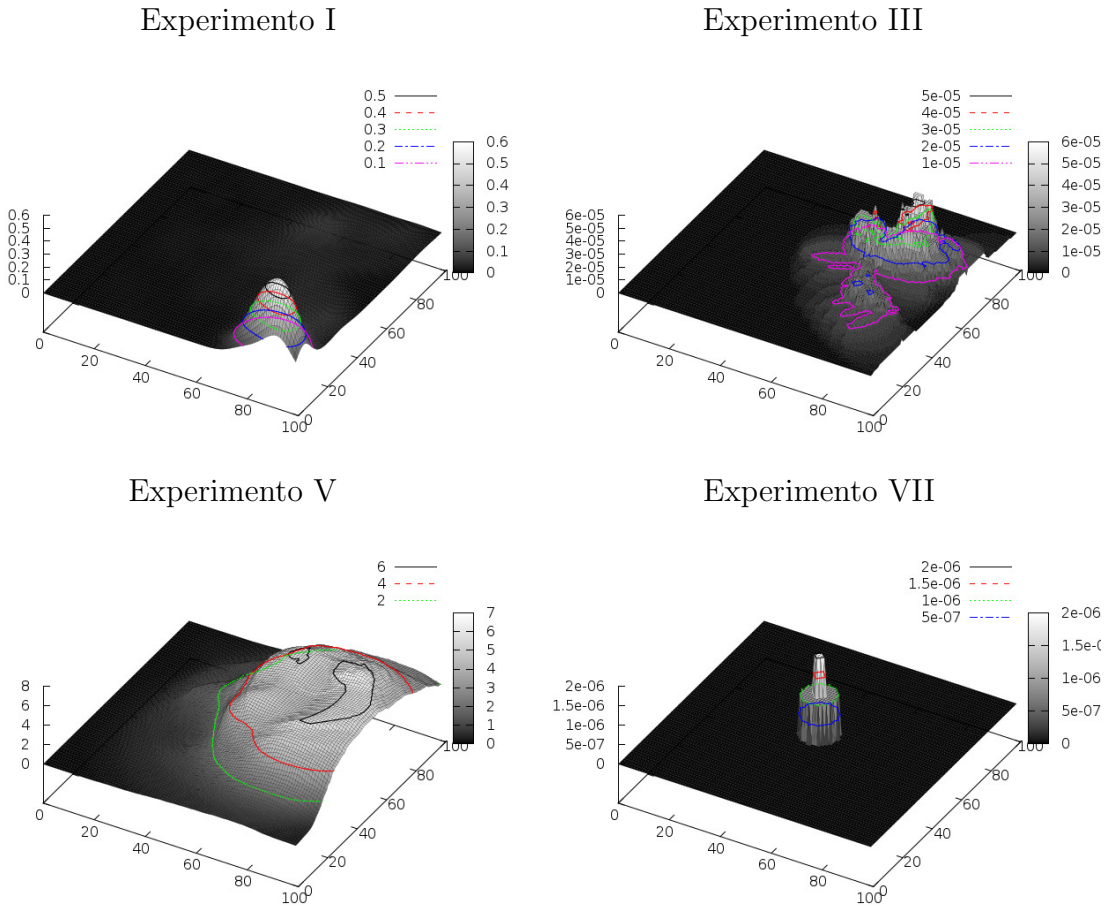


Figura 8.19: Gráficas de la varianza en los experimentos I, III, V y VII, en los que se utilizaron las cuatro operaciones.

8.6.6. Utilizando únicamente *Raise* y *Lower*

La siguiente tarea asignada a los participantes es editar el terreno compartido, utilizando solamente las operaciones *Raise* y *Lower*. Dado que dichas operaciones consisten en adiciones y sustracciones de valores sobre el DEM, se espera que sean conmutativas, i.e., afirmamos que el orden en el cual se aplican no afecta el resultado final.

Para esta tarea, se realizaron cuatro experimentos, cuyos resultados se muestran en la Tabla 8.3. Como antes, se reporta el número de participantes y el error relativo calculado por cada experimento.

Tabla 8.3: Experimentos usando las operaciones Raise y Lower.

Experimento	**II**	**IV**	**VIII**	**IX**
Número de participantes	3	3	4	4
Error relativo (%)	0.0000	0.0000	0.0000	0.0000

Para esta tarea, se observa que, en cada experimento, todos los participantes terminaron exactamente con el mismo DEM, i.e., no se registró ninguna inconsistencia. La varianza calculada fue cero en todos los puntos del terreno, confirmando nuestra afirmación acerca de la conmutatividad de las operaciones *Raise* y *Lower*.

8.6.7. Dibujando una letra del alfabeto

Finalmente, se le pidió a dos grupos, de cuatro participantes cada uno, que editaran el terreno usando las cuatro operaciones para crear la forma de alguna letra de su elección. En la Tabla 8.4, se reporta el número de participantes y el error relativo para estos experimentos.

Tabla 8.4: Experimentos dibujando una letra.

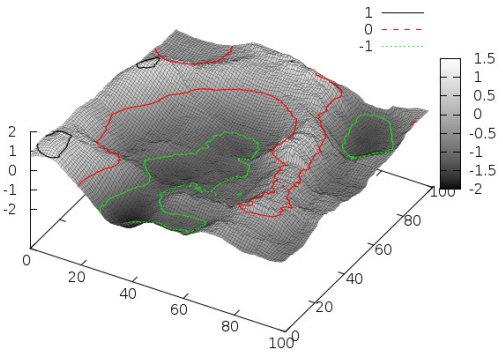
Experimento	**VI**	**X**
Número de participantes	4	4
Error relativo (%)	0.0000	0.0000

A pesar de que se utilizaron todas las operaciones, en ambos experimentos no hubo inconsistencias. Los datos del DEM resultante para los experimentos VI y X, se muestran en la Figura 8.20. La meta grupal en el experimento VI fue dibujar la letra 'C'. Observamos que, durante la actividad, los participantes espontáneamente se mostraron muy cuidadosos, tratando de no obstruir los trazos de otros participantes, mientras trataban de completar la tarea como equipo.

Posteriormente, en el experimento X[8], los usuarios dibujaron una letra 'M'. En este caso, se observó una mayor obstrucción entre los participantes, pues dibujaban en regiones traslapadas. Sin embargo, al notar la situación, decidieron comenzar de nuevo y esta vez optaron por repartirse las regiones de la letra e incluso tomaron turnos espontáneamente, lo cual explica el resultado consistente. Podemos afirmar que el prototipo les proporcionó suficiente retroalimentación visual como para darse cuenta de la situación y decidir un plan de acción alternativo.

[8] http://youtu.be/bMe7Hz4NX7Y

Experiment VI



Experiment X

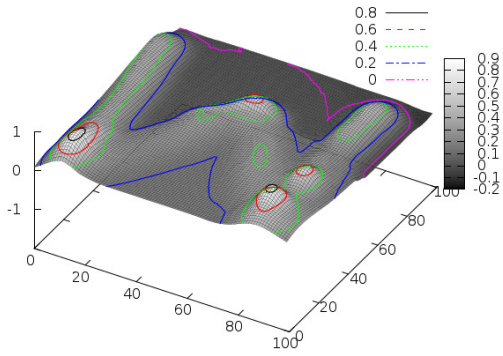


Figura 8.20: DEMs para las letras 'C' y 'M', respectivamente. Fueron creadas durante los experimentos VI y X, en los cuales no se encontraron inconsistencias entre las instancias del editor.

Aquí se observa un caso real de colaboración orientada a una meta grupal, en la que los usuarios se regularon a sí mismos, como un equipo, y desarrollaron un protocolo social, como se menciona en la Sección 2.1. Mientras que otros enfoques colaborativos optan por imponer mayores restricciones a los colaboradores, con el fin de enfocar sus esfuerzos, nosotros preferimos darles total libertad de acción para no obstaculizar sus intenciones. Así pues, se permite que los colaboradores efectúen cambios de roles de manera ágil y espontánea.

Capítulo 9

Editor de mallas poligonales

En el prototipo anterior, es posible modificar una rejilla poligonal para darle la forma de una superficie en 3D, pero sin agregar o eliminar vértices, aristas ni caras. Sin embargo, la forma que se le puede dar a dicha malla está limitada a terrenos y otras gráficas de funciones $h : \mathbb{R}^2 \to \mathbb{R}$. Ahora bien, la verdadera utilidad de las mallas poligonales es que se les puede dar cualquier forma arbitraria, siendo posible obtener una representación aproximada de prácticamente cualquier objeto. En el presente prototipo, avanzamos un paso hacia la construcción de un editor colaborativo completo de mallas poligonales de forma arbitraria. Sin embargo, en el estado actual de desarrollo, el prototipo es monousuario y es capaz de modificar dichas mallas manipulando sus vértices, pero sin llegar a eliminar sus componentes ni agregar otros nuevos.

9.1. Antecedentes

Con el fin de describir la manera en que se modela la malla poligonal en el presente prototipo, es necesario definir primero tres conceptos teóricos del área de la Topología: *espacio topológico*, *homeomorfismo* y *n-variedad*.

Un espacio topológico se puede definir como un conjunto de puntos, junto con un conjunto de vecindades para cada punto, el cual satisface ciertos axiomas que relacionan los puntos con sus vecindades. Se trata de una entidad matemática de la teoría de conjuntos y es la noción más general de un espacio matemático, que permite definir conceptos como *continuidad*, *conectividad* y *convergencia*. La definción formal de *espacio topológico* se debe a Felix Hausdorff [6]:

Sea X un conjunto, que puede estar vacío y cuyos elementos suelen llamarse *puntos*, aunque puede tratarse de cualquier objeto matemático. Sea N una función que asigna a cada punto $x \in X$ una colección $N(x)$ no vacía de subconjuntos de X. Los elementos de $N(x)$ serán llamados *vecindades* de x con respecto a N. La función N es una *topología*, si se cumplen los axiomas siguientes, en cuyo caso el par (X, N) es un espacio topológico.

Los axiomas son:

- Si el conjunto A es una vecindad de x (i.e., $A \in N(x)$), entonces $x \in A$, i.e., todo punto pertenece a cada una de sus vecindades.
- Si $A \subset X$ y $B \in N(x)$ son tales que $A \supset B$, entonces $A \in N(x)$, i.e., cada conjunto que contiene a una vecindad del punto x es también una vecindad de x.
- Sean $A \in N(x)$ y $B \in N(x)$, entonces $A \cap B \in N(x)$, i.e., la intersección de dos vecindades de x es también una vecindad de x.
- Sea $A \in N(x)$, entonces $\exists\ B \in N(x)$ tal que $A \supset B$ y $A \in N(y), \forall y \in B$, i.e., cualquier vecindad A de x contiene a una vecindad B de x, tal que A es una vecindad de cada punto de B.

Un *homeomorfismo* es una función continua entre espacios topológicos que tiene una función inversa continua. Los homeomorfismos son los isomorfismos de los espacios topológicos, i.e., son los mapeos que preservan todas las propiedades topológicas del espacio dado. Dos espacios son homeomorfos, si existe un homeomorfismo entre ellos y se consideran equivalentes, desde el punto de vista topológico.

Se puede decir, de manera informal, que un espacio topológico es un objeto geométrico y un homeomorfismo le aplica una deformación y estiramiento continuos, para darle una forma diferente. Así pues, un cuadrado y un círculo son homeomorfos entre sí, pero una esfera y un toroide no lo son.

Una *variedad* es un espacio topológico que es localmente similar al espacio Euclidiano, en la cercanía de cada punto. Más precisamente, cada punto de una variedad n-dimensional tiene una vecindad que es homeomorfa al espacio Euclidiano n-dimensional (i.e., a $\mathbb{R}^n$).

Una variedad generaliza la noción intuitiva de curva y de superficie a cualquier dimensión y sobre campos diversos, en particular nos interesa el campo de los reales. De esta manera, una curva es una variedad de dimensión 1, denominada 1-variedad, una superficie es una variedad de dimensión 2, denominada 2-variedad. Entonces, un espacio topológico de dimensión n es una n-variedad.

Sin embargo, existen objetos geométricos que no son variedades de una dimensión dada. De Floriani y Hui[1] proponen una estructura de datos para representar mallas poligonales, que no cumplen la condición de ser una n-variedad, a los cuales definen como sigue:

> "De manera informal, un objeto con topología de n-variedad (*n-manifold*) es un subconjunto del espacio Euclidiano, para el cual la vecindad de cada punto interior es localmente equivalente a una bola abierta de dimensión n. Los objetos que no cumplen esta propiedad son llamados objetos *sin-variedad* (*non-manifold*)."

[1]Preprint disponible en: http://mangrovetds.sourceforge.net/related_papers/NMIA_article.pdf

En la Figura 9.1, se muestran ejemplos de variedades de dimensión 1 y 2. Las líneas y los círculos son 1-variedades, pero una forma de ocho (8) no lo es, pues tiene un punto de cruce que no es localmente homeomorfo al espacio Euclidiano de una dimensión, i.e., es sin-variedad. La esfera y las superficies $h : \mathbb{R}^2 \to \mathbb{R}$ son 2-variedades.

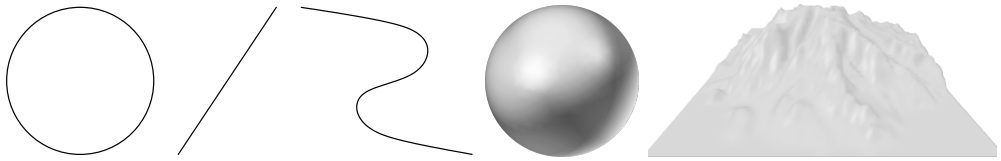


Figura 9.1: Algunos ejemplos de 1-variedades son: el círculo, la recta y la curva. La esfera y la superficie son 2-variedades.

9.2. Modelo de la malla

Con el fin de aterrizar los conceptos teóricos, presentados en la Sección 9.1, al área de Modelado 3D, se tiene que los vértices, aristas y caras de una malla poligonal son entidades topológicas de dimensión 0, 1 y 2, respectivamente. Se trata de n-variedades, con $n = 0, 1, 2$. Una malla poligonal en el espacio tridimensional es un espacio topológico, pues posee un conjunto de puntos y una topología que define las relaciones de vecindad topológica y conectividad entre estos. De manera que podemos afirmar que el prototipo desarrollado permite deformar la malla sin alterar su topología. En general, se tiene que una malla puede ser una n-variedad, con $n \leq 3$ o bien puede ser una entidad sin-variedad.

Como se mencionó en la Sección 2.2, las mallas poligonales están conformadas por vértices, aristas y caras, interrelacionadas de una manera jerárquica. En general, una malla está formada por caras, las caras se componen de aristas y las aristas unen pares de vértices. Existe una gran variedad de representaciones computacionales para las mallas poligonales, que son útiles en aplicaciones diversas. En particular, para aplicaciones de modelado, nos interesan mallas con la propiedad de no estar restringidas a una topología de n-variedad fija, i.e., que son sin-variedad. Así que necesitamos una estructura de datos capaz de representar y editar mallas, que no tengan esta restricción en su topología.

En la Figura 9.2, se muestran ejemplos de mallas poligonales con diversas condiciones de variedad. Se resaltan los elementos de 0, 1 y 2 dimensiones para mejorar su visibilidad. La malla sin-variedad contiene regiones cuyos puntos tienen vecindades de dimensiones diferentes.

Para construir el presente prototipo de editor de mallas poligonales, se utilizó la estructura de datos `BMesh` de Joseph Eagar[2], creada para sustituir, en Blender,

[2]https://www.blendernetwork.org/joe-eagar

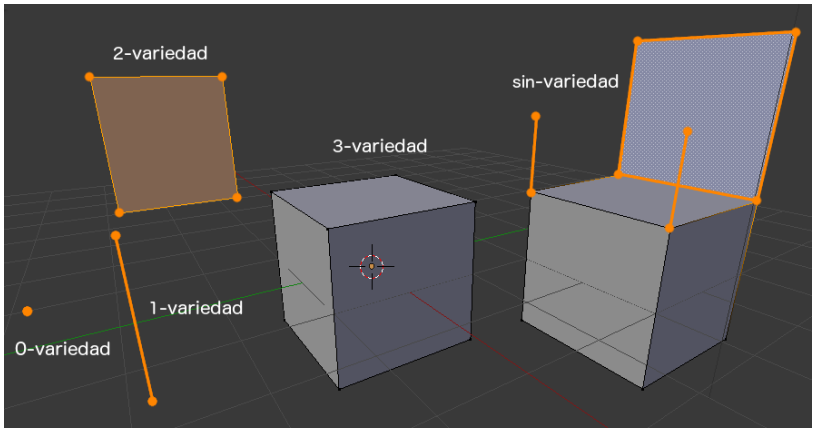


Figura 9.2: Algunos ejemplos de mallas con distintas condiciones de variedad.

a la obsoleta estructura `EditMesh`. Proporciona la funcionalidad necesaria para almacenar y manipular, de manera persistente, toda la información de conectividad entre los elementos de la malla. El proyecto *BMesh* también cuenta con las contribuciones de Geoffry Bantle, Levi Schooly, Wael Oraiby y Fabian Fricke. En adelante, seguiremos la descripción acerca de la estructura de datos `BMesh` dada en: https://wiki.blender.org/index.php/Dev:Source/Modeling/BMesh/Design.

La estructura `BMesh` proporciona una representación de contorno para mallas poligonales, que no necesariamente cumplen una condición de variedad definida. Cabe recordar que a esta propiedad se le conoce como sin-variedad (*non-manifold*). Esta estructura es simillar a la estructura *Radial Edge* [65], pero se le han añadido otras listas circulares útiles para la manipulación de la topología a nivel local, las cuales se explican más adelante. `BMesh` cuenta con las siguientes características:

- proporciona información de adyacencia persistente,
- su topología se puede modificar de manera local,
- se pueden representar fácilmente caras de longitud arbitraria, i.e., polígonos de n lados, conocidos como *N-Gons* y
- permite representar, de manera trivial, cualquier malla con condiciones sin-variedad, incluyendo cadenas de aristas de alambre (*wire edges*).

Las últimas tres características listadas dependen de la primera, de modo que la implementación de esta característica es fundamental para toda la estructura `BMesh`. La información de adyacencia se almacena mediante un sistema de listas circulares doblemente ligadas, que mantiene las relaciones topológicas entre las entidades de la malla, i.e., vértices, aristas y caras. Estas listas son conceptualmente equivalentes a las que se encuentran en otras representaciones de contorno, tales como *Half-Edge* y *Radial Edge*. Cada entidad topológica es un nodo accedido por referencia en las listas cíclicas a las que pertenece, de manera que los requerimientos de memoria se mantienen al mínimo.

Las conexiones entre elementos están definidas por bucles alrededor de cada entidad topológica, conocidos también como ciclos. Cada ciclo está diseñado para responder a ciertas consultas de adyacencia, acerca de una cierta entidad, la cual es la base del ciclo. Por ejemplo, un ciclo diseñado para responder a la consulta "¿cuáles aristas comparten este vértice?" tendrá a dicho vértice como base del ciclo y a sus aristas como nodos del ciclo. Nótese que no es necesario almacenar explícitamente todas las relaciones posibles de adyacencia, ya que toda la información de conectividad se puede derivar rápidamente a partir de consultas sobre dos o más ciclos. En la Sección 9.3.2, se describen algunos ejemplos de consultas que se pueden realizar en una estructura `BMesh`. Los tres ciclos que se almacenan de manera explícita dentro de `BMesh` son el ciclo de aristas (*Disk Cycle*), el ciclo radial (*Radial Cycle*) y el ciclo de eslabones (*Loop Cycle*).

9.2.1. Entidades topológicas

Básicamente, `BMesh` almacena la información de la topología de la malla en cuatro estructuras principales, una para cada elemento o entidad topológica básica:

- `BMVert`: almacena las coordenadas de cada vértice de la malla, su vector normal y un apuntador a alguna de las aristas de su ciclo de aristas.

- `BMEdge`: define la arista que representa una conexión entre dos vértices y también almacena una liga a un eslabón dentro de su ciclo radial.

- `BMLoop`: define un eslabón de la cadena de vértices presentes en una cara. Cuenta con apuntadores a sus elementos asociados: a un vértice, a una arista, a una cara y a las listas del ciclo de eslabones y del ciclo radial.

- `BMFace`: representa una cara de la malla y lleva un apuntador a cualquiera de los eslabones del ciclo que define su contorno. La cara puede tener su propia normal independientemente de la de sus vértices.

En el siguiente listado, se definen las estructuras en lenguaje C para vértices y aristas, junto con las estructuras auxiliares `BMFlagLayer` y `BMDiskLink`. Las estructuras de eslabón y cara se explican en la Sección 9.2.2. En las definiciones de las estructuras y en las figuras se han omitido ciertos elementos por claridad.

```
/* Banderas utilizadas por la pila de operadores */
typedef struct BMFlagLayer {
        short flags; /* banderas */
} BMFlagLayer;

/* Un nodo del ciclo de aristas, usado solamente por BMEdge */
typedef struct BMDiskLink {
        struct BMEdge *next, *prev;
} BMDiskLink;
```

```
/* Estructura del vértice */
typedef struct BMVert {
        BMHeader head;
        struct BMFlagLayer *oflags;
        float coord[3];          /* coordenadas del vértice */
        float normal[3];         /* vector normal del vértice */
        struct BMEdge *edge;     /* apuntador a cualquier arista que
              tenga a este vértice */
} BMVert;

/* Estructura de la arista */
typedef struct BMEdge {
        BMHeader head;
        struct BMFlagLayer *oflags;
        /* vértices de esta arista (no importa el orden) */
        struct BMVert *v1, *v2;
        /* lista de eslabones asociados a esta arista (use loop->
              radial_prev/next, para acceder a los otros ciclos de
              eslabones que la usan) */
        struct BMLoop *loop;
        /* listas de aristas que usan a los vértices v1 y v2 */
        BMDiskLink v1_disk_link, v2_disk_link;
} BMEdge;
```

En la Figura 9.3, se muestra la representación geométrica de la entidad topológica arista, junto con sus dos vértices V1 y V2. La arista contiene dos estructuras `BMDiskLink` para llevar el registro de las aristas que comparten cada uno de sus vértices, D1 para el primer vértice y D2 para el segundo. En el caso de una arista individual, D1 y D2 apuntan a sí mismos, pues sólo existe una arista que contiene a V1 y a V2.

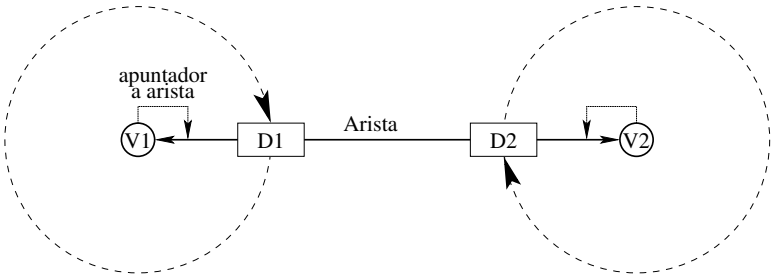


Figura 9.3: Diagrama de una arista individual.

A continuación se describen las propiedades de cada ciclo y una lista de funciones para procesarlos.

9.2.2. Ciclo de aristas

Como se puede apreciar en la Figura 9.4, el mismo vértice V se puede encontrar en varias aristas, cada una con sus dos estructuras `BMDiskLink` y V puede estar asociado a D1 en algunas aristas, y a D2 en otras. El vértice V tiene un apuntador a una de

las aristas que lo contiene y es a través de esta arista que puede utilizar la lista D1, para acceder a las demás aristas que lo contienen. De esta manera, se responde a la consulta "¿cuáles aristas comparten este vértice?".

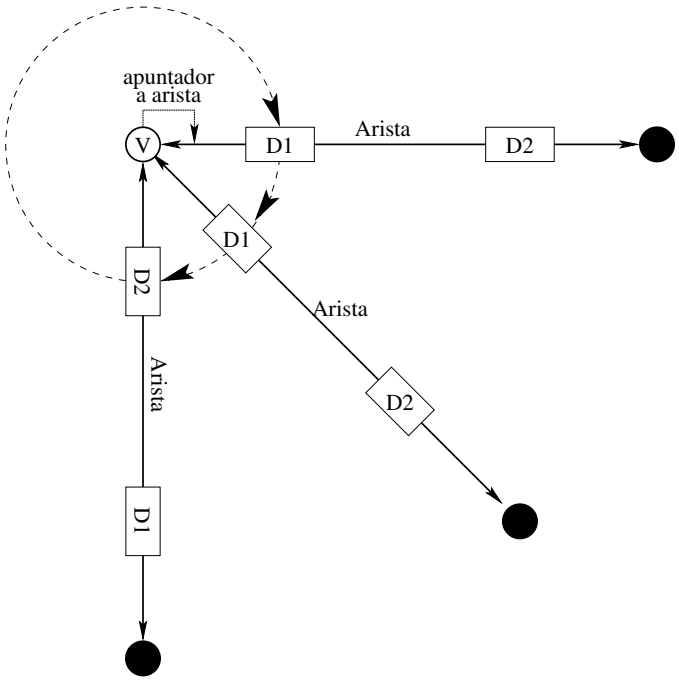


Figura 9.4: Diagrama del ciclo de aristas.

Las funciones relacionadas con este ciclo se listan a continuación:

- `void bmesh_disk_edge_append (BMEdge* e)`
- `void bmesh_disk_edge_remove (BMEdge* e)`
- `BMEdge* bmesh_disk_edge_next (void )`

Se proporciona la funcionalidad para navegar a través del ciclo de aristas, de manera transparente para los operadores de modelado que lo utilizan en un nivel superior. Nótese que, a diferencia de *Half-Edge*, el ciclo de aristas es completamente independiente de los datos de las caras, lo cual proporciona la ventaja de poder representar naturalmente las cadenas de aristas de alambre. De hecho, este ciclo de aristas no presenta problemas con ningún tipo de condiciones de sin-variedad que involucren caras poligonales.

9.2.3. Ciclo de eslabones

Para explicar el ciclo de eslabones, es necesario describir la manera en que se almacenan las caras en la estructura `BMesh`. Cada cara poligonal se representa mediante dos estructuras en C: `BMLoop` y `BMFace`.

```
/* Estructura del eslabón */
typedef struct BMLoop {
        BMHeader head;
        // note la ausencia de banderas
        struct BMVert *vert;     /* vértice asociado a este eslabón */
        struct BMEdge *edge;     /* arista asociada, tiene los vértices
                (v, next->v) */
        struct BMFace *face;     /* la cara formada */
        /* next y prev son los eslabones siguiente y anterior, en la
               cadena que forma la cara */
        struct BMLoop *next, *prev;
        /* lista circular de los eslabones que utilizan la misma arista
               que este eslabón, aunque no necesariamente el mismo
               vértice (puede ser v1 o v2 de edge) */
        struct BMLoop *radial_next, *radial_prev;
        /* per-face-vertex data */
        float color[3], UVtexcoords[2];
} BMLoop;

/* Estructura de la cara */
typedef struct BMFace {
        BMHeader head;
        struct BMFlagLayer *oflags;
        BMLoop *l_first;// primer eslabón de la cadena que forma la cara
        int    len;       /* número de vértices en esta cara */
        float normal[3];/* vector normal de la cara */
        short material; /* índice del material para esta cara */
} BMFace;
```

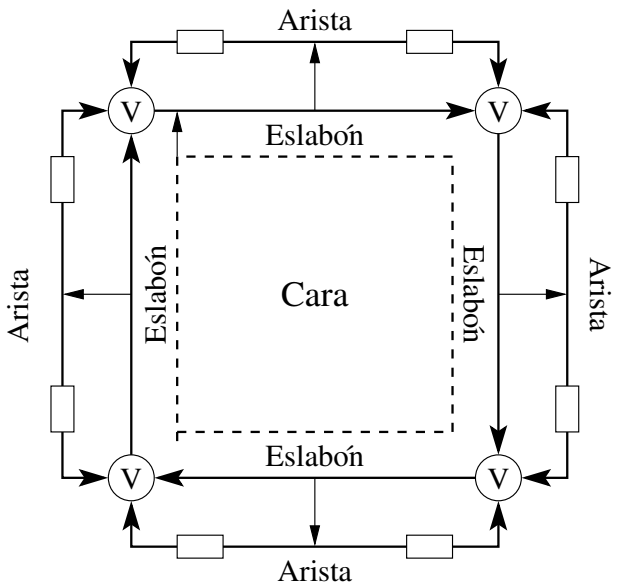


Figura 9.5: Diagrama de una cara poligonal que muestra sus aristas, sus vértices y el ciclo de eslabones que la forman. El cuadrado en línea punteada representa la entidad cara, que apunta a uno de los eslabones.

La estructura `BMesh` contiene una lista dinámica de estructuras `BMFace`, para almacenar las caras poligonales que forman la malla. Cada cara está formada por una cadena de eslabones que enlistan los vértices que forman la cara. Aunque cada eslabón se corresponde lógicamente con una arista, le pertenece a una cierta cara, de manera que es posible tener varios eslabones por cada arista, con excepción de las aristas de contorno de una superficie. Cada eslabón lleva un apuntador a la arista que le corresponde, otro hacia el vértice inicial de dicha arista y otro más hacia la cara asociada con este ciclo. Además, los eslabones son útiles para almacenar los datos asociados con cada vértice de cada cara (*per-face-vertex data*), tales como coordenadas de textura y color. Nótese que cada cara `BMFace` no almacena sus vértices ni sus aristas de manera explícita, sino solamente un apuntador al primer elemento `BMLoop` de su ciclo de eslabones. En el diagrama de la Figura 9.5 se muestra una cadena de eslabones en sentido horario.

Como se puede notar en la Figura 9.5, la estructura `BMLoop` es similar a *Half-Edge* en que se trata de una gráfica dirigida, de acuerdo al orden de los vértices de la cara, y en que solamente almacena referencias al vértice. Dado que cada eslabón está asociado también con un `BMEdge`, se debe satisfacer la propiedad de que los vértices *loop→vert* y *loop→next→vert* deben estar ambos contenidos en la arista *loop→edge*. Independientemente de lo anterior, los apuntadores a los vértices se acceden desde la cara y no desde las aristas, el primer vértice de una cara es *face→l_first→vert*, el segundo es *face→l_first→next→vert* y así sucesivamente.

9.2.4. Ciclo radial

El ciclo radial es similar a su análogo dentro de la estructura *Radial Edge*. Sin embargo, `BMesh` se diferencía de este en que no requiere una gran cantidad de memoria para representar mallas con condiciones de sin-variedad, pues no lleva un registro de sus regiones.

En la Figura 9.6, se ilustra la construcción de un ciclo radial con dos caras que utilizan una misma arista. No se muestran todas las aristas, pues no son necesarias para este ciclo y, de representarlas, se abarrotaría el diagrama. A pesar de que a una arista se le puede asociar cualquier cantidad de caras y, por ende, puede tener cualquier cantidad de eslabones diferentes, cada estructura `BMEdge` almacena solamente un apuntador a uno de sus eslabones `BMLoop`, el cual se considera la base de su ciclo radial. De este modo, es posible visitar cada cara asociada con una arista, siguiendo los apuntadores *radial_prev* y *radial_next*. De esta manera, se responde a la consulta "¿cuáles caras comparten esta arista?".

Las funciones relacionadas con este ciclo se listan a continuación. Aquellas marcadas con el símbolo (♣) se usan internamente por otras funciones de modelado.

- `void bmesh_radial_append (BMLoop* l)` ♣
- `void bmesh_radial_remove (BMLoop* l)` ♣
- `BMLoop* bmesh_radial_next ()`
- `BMLoop* bmesh_radial_find (BMLoop* l)`

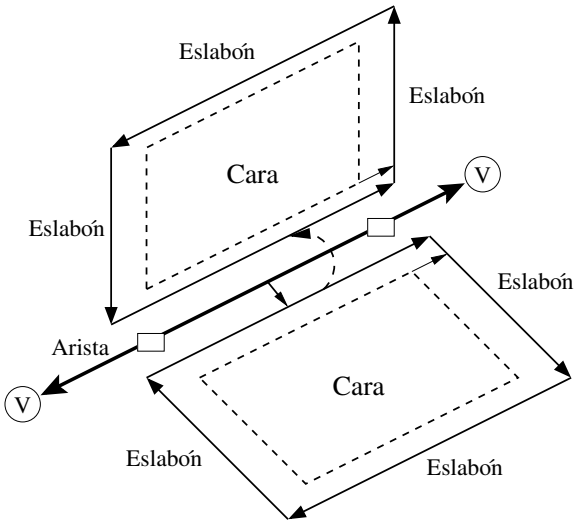


Figura 9.6: Diagrama del ciclo radial de una arista compartida por dos caras.

Nótese que, tanto en el ciclo de aristas como en el ciclo radial, el orden de sus elementos no es importante, mientras que en el ciclo de eslabones sí lo es. En estos ciclos sin orden definido, se tiene un incremento en el tiempo de búsqueda necesario para derivar algunas relaciones de adyacencia. Sin embargo, se tiene la ventaja de que ninguna de las propiedades intrínsecas de la malla quedan en dependencia del orden de dichos ciclos y, con ello, las mallas sin la propiedad de variedad se representan de manera trivial. En la Figura 9.7, se muestra cómo se complementan los tres ciclos para formar una malla poligonal.

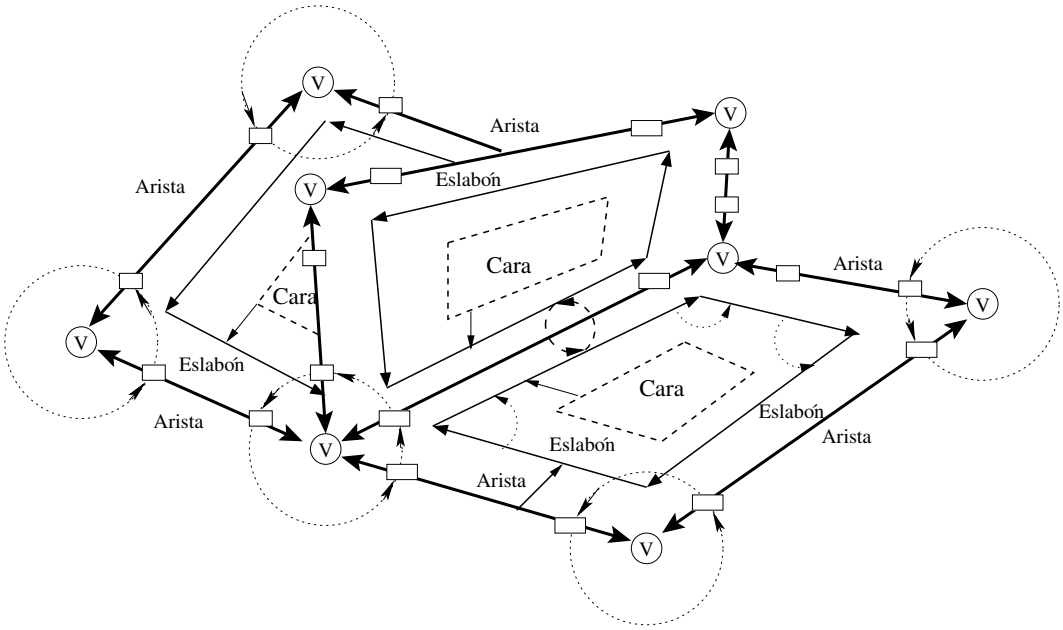


Figura 9.7: Diagrama de los tres ciclos de `BMesh` en una malla poligonal.

9.3. Diseño de las operaciones

La implementación de `BMesh` cuenta con una variedad de operadores y utilidades para la edición de las mallas representadas en la estructura, las cuales se describen en la Sección 9.3.1. Sin embargo, no se proporcionan las operaciones de selección y de transformación afín de las entidades topológicas, pues estas se realizan desde la interfaz de usuario de Blender. Por ello fue necesario implementar estas operaciones, pero diseñadas para realizarse mediante gestos táctiles, en lugar de ratón y teclado, como se hace en Blender. Tambien se crearon funciones para crear las mallas que se van a manipular, durante la ejecución del prototipo, las cuales se describen en la Sección 9.4.1. La implementación de los gestos que realizan estas dos operaciones se describe en las Secciones 9.4.2 y 9.4.3.

9.3.1. API de edición de `BMesh`

De manera informal, el conjunto de operaciones, que vienen implementadas en la API de `BMesh`, se pueden clasificar en tres capas: de bajo nivel, de medio nivel y de alto nivel.

API de bajo nivel

En el nivel más bajo, se proporcionan las funciones que permiten recorrer los ciclos de `BMesh`, además de un conjunto de funciones, conocidas como operadores de Euler, para aplicar cambios locales a la malla. Dichos operadores de Euler realizan las operaciones primitivas (o atómicas) que utilizan las otras funciones de modelado para manipular la topología de la malla. Estos operadores básicos se pueden combinar para crear operaciones más sofisticadas. Cada operador de Euler es nombrado con un acrónimo que describe su función y cada uno tiene su inverso lógico. Se listan a continuación los operadores con sus inversos y su respectiva función:

- `BM_vert_create/kill()`, `BM_edge_create/kill()` y `BM_face_create/kill()`: son los operadores para crear y eliminar un vértice, una arista y una cara, respectivamente. Son inversos por pares.

- `bmesh_semv/jekv()`: el acrónimo `semv` (del inglés *split edge, make vert*) indica la acción de partir una arista en dos y crear un vértice común. Su inverso es `jekv` (del inglés *join edge, kill vert*) que indica la acción de unir dos aristas, eliminando su vértice común. Ambas acciones se muestran en la Figura 9.8.

- `bmesh_sfme/jfke()`: el acrónimo `sfme` (del inglés *split face, make edge*) indica la acción de partir una cara en dos, mediante una nueva arista que pase por dos vértices existentes en ella. Su inverso es `jfke` (del inglés *join face, kill edge*) que indica la acción de unir dos caras contiguas eliminando su arista común. Ambas acciones se muestran en la Figura 9.9. Para que el operador `bmesh_jfke()` funcione correctamente, es necesario que las dos caras sea contiguas y estén

unidas por una sola arista. En la Figura 9.10, se ilustran dos casos en los que esto no ocurre. En el recuadro izquierdo, las caras están separadas por dos aristas, primero se tendrían que unir ambas aristas con el operador `bmesh_jekv()`. En el recuadro derecho de la misma figura, las caras también tienen dos aristas en común, pero además existe un agujero entre estas.

- `bmesh_loop_reverse()`: invierte el orden en el ciclo de eslabones en una cara. Este operador es su propio inverso.

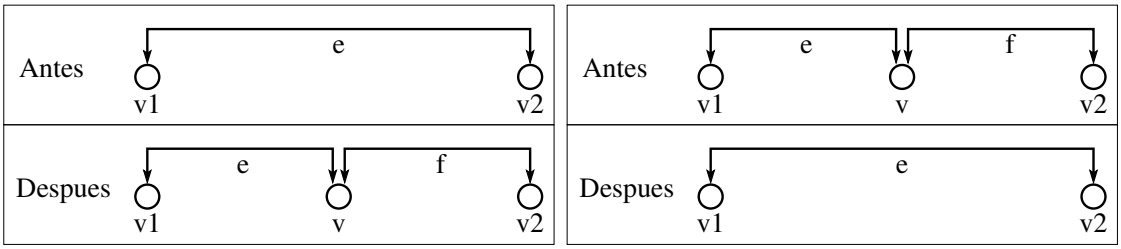


Figura 9.8: El operador `bmesh_semv()` parte una arista en dos (izquierda), y su inverso `bmesh_jekv()` une dos aristas (derecha).

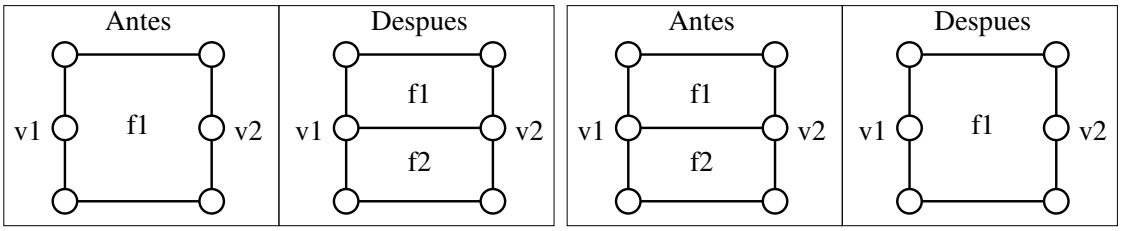


Figura 9.9: El operador `bmesh_sfme()` parte una cara en dos (izquierda), y su inverso `bmesh_jfke()` une dos caras (derecha).

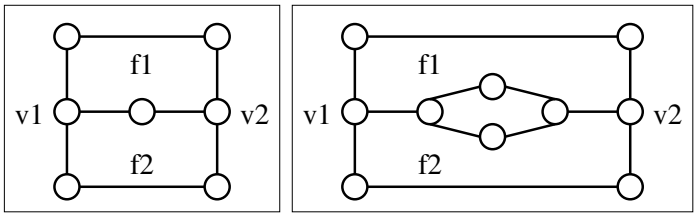


Figura 9.10: Dos situaciones en las que el operador `bmesh_jfke()` falla.

Lo que se busca es construir cualquier operación de modelado como una combinación de operadores de Euler. Estos operadores utilizan la información de adyacencia almacenada en la estructura `BMesh` para realizar modificaciones a la topología local, lo cual implica que el tiempo de ejecución de cada operador dependa únicamente de la complejidad local de la región modificada, mas no de la malla completa.

Adicionalmente, al aplicar los operadores de Euler, se garantiza la validez de la malla obtenida, con lo cual se evita que las herramientas que hacen uso de estos, tengan que verificar su validez o hacer reparaciones en la malla. Un ejemplo de una operación construida a partir de operadores de Euler se ilustra en la Figura 9.11. Partiendo de la cara $f1$, se proporcionan las aristas $e1$ y $e2$, a través de las cuales se desea efectuar un corte de la cara. Primero se crea un vértice a la mitad de $e1$ y luego otro en $e2$. Finalmente, se divide la cara mediante una arista que pasa por los nuevos vértices. Este proceso está codificado en el listado siguiente:

```
void split_face (BMFace *f1, BMEdge *e1, BMEdge *e2) {
        BMVert *v1, *v2;
        BMFace *f2;
        v1 <- bmesh_semv (e1);
        v2 <- bmesh_semv (e2);
        f2 <- bmesh_sfme (f1, v1, v2);
}
```

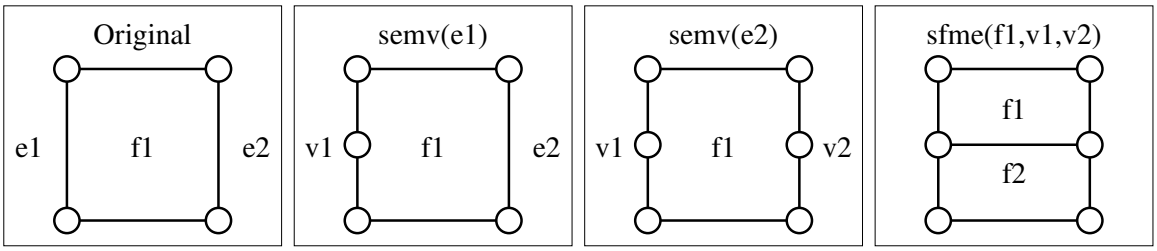


Figura 9.11: Proceso de división de una cara en dos, utilizando solamente operadores de Euler.

API de nivel medio

Las funciones de bajo nivel se utilizan como base para construir los operadores llamados *bmops*[3], que efectúan algunos tipos fundamentales de operaciones de edición de mallas, sobre las regiones especificadas por los parámetros de entrada. Estos *bmops* están diseñados para ser utilizados por las herramientas de edición de alto nivel. Algunos tipos de operadores de medio nivel son los iteradores sobre conjuntos de entidades topológicas (llamados *iterators* y *walkers*) y algunas utilidades de edición, tales como: `bmop_mirror()`, `bmop_bevel()` y `bmop_similarfaces()`. La mayoría de las herramientas de edición de alto nivel están compuestas por varios de estos operadores de medio nivel, los cuales también pueden llamar a otros para realizar sus tareas. Esto no ocurre con los operadores de Euler, pues son atómicos y no dependen de otros operadores.

API de alto nivel

En el nivel más alto se encuentran las herramientas de modelado, que conectan la funcionalidad de edición de mallas con la interfaz de Blender y pueden ser ejecu-

[3]El nombre *bmop* es una contracción de *bmesh operator*, i.e., operador de `BMesh`.

tados directamente por los usuarios o llamados via *scripting*. Las herramientas son las únicas operaciones de la API de `BMesh` que pueden escribir en las banderas de encabezado `BMFlagLayer`, por lo que dichas herramientas pueden afectar las propiedades de selección y visibilidad de las entidades topológicas. Como parte de una buena práctica de diseño, se recomienda que, en las herramientas, no se implemente ninguna lógica compleja para recorrer la malla ni editarla directamente. Para ello, se deben utilizar los iteradores y operadores de modelado de nivel inferior. Algunos ejemplos de herramientas de modelado de alto nivel son: *Disuelve (Dissolve)*, *Cuchillo (Knife)* y *Biselado (Bevel)*.

Disuelve (Dissolve)

Permite fusionar las caras, aristas o vértices seleccionados en una sola cara poligonal. Esta herramienta produce resultados ligeramente distintos, dependiendo del tipo de entidades que conforman la selección. Con ayuda de esta herramienta, es posible agregar detalles en la región deseada o eliminarlos rápidamente.

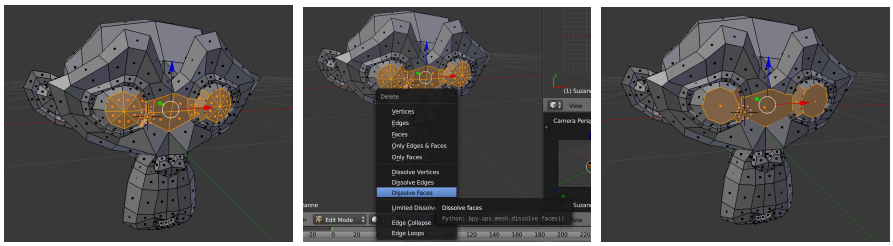


Figura 9.12: Efecto de aplicar la herramienta para disolver una selección de caras contiguas, usando la implementación en Blender de `BMesh`.

En la Figura 9.12, se observa el resultado de aplicar esta herramienta en una malla de Blender. Se seleccionan las caras (izquierda), se aplica la operación para disolver caras (centro) y se obtiene el resultado (derecha): las caras contiguas se fusionan en una sola.

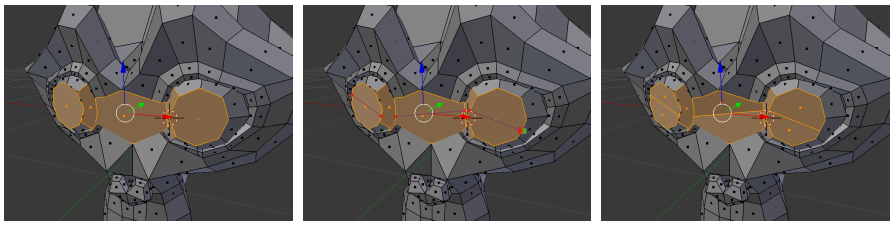


Figura 9.13: Efecto de aplicar la herramienta cuchillo para cortar las caras de una malla.

Cuchillo (Knife)

La herramienta cuchillo permite subdividir la malla, de manera interactiva, para realizar una variedad de cortes, dibujando líneas o bucles cerrados, de manera sencilla

y flexible. Sin embargo, el resultado depende de la calidad de la malla inicial. En la Figura 9.13, se observa el proceso de corte, trazando varias líneas sobre la malla. Se tiene la malla antes del corte (izquierda), se efectúa el trazo con la herramienta cuchillo (centro) y se dividen las caras a través del trazo (derecha).

Biselado (Bevel)

La herramienta de biselado permite agregar un bisel sobre las caras o aristas seleccionadas, agregando nuevas caras espaciadas de manera uniforme. Usualmente se aplica sobre las esquinas de una malla para redondearlas. En la Figura 9.14, se muestra el resultado de biselar una esquina, antes y después de aplicar la herramienta a las aristas seleccionadas.

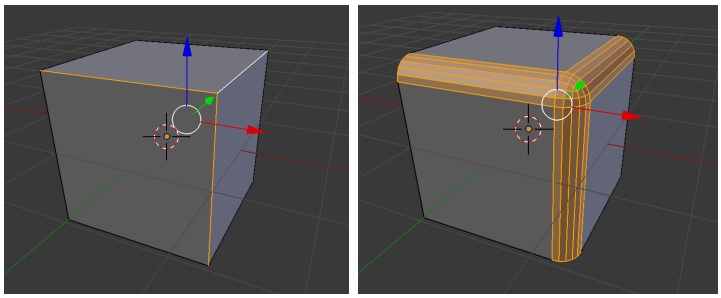


Figura 9.14: Efecto de aplicar la herramienta de biselado sobre una esquina de la malla.

Aunque en el presente prototipo aún no se requieren estas herramientas de alto nivel, se mencionan en esta sección por completitud para la descripción de `BMesh`. En lugar de utilizar dichas herramientas, en la etapa actual de desarrollo, se diseñaron e implementaron otras operaciones de modelado, que se describen en la Sección 9.4, las cuales sirven para crear una nueva malla, seleccionar un conjunto de vértices y aplicar una transformación afín a la selección.

9.3.2. Consultas acerca de la topología de la malla

Se mencionó en la Sección 9.2 que los ciclos ayudan a responder consultas acerca de la topología local de la malla: el ciclo de aristas está formado por las aristas que comparten un vértice, el ciclo de eslabones indica la cadena de vértices y de aristas que forman una cara y el ciclo radial permite el acceso a las cara que comparten una arista, a través de sus eslabones. Para recorrer fácilmente las listas de entidades topológicas, se utilizan los iteradores de la API, de nivel medio y bajo.

También se pueden efectuar diversas consultas como las siguientes:

- "¿cuáles caras comparten este vértice?",
- "¿este vértice se encuentra dentro de esta cara?",
- "¿esta arista se encuentra dentro de esta cara?" y

- "¿existe una arista común a dos caras?", entre otras.

A continuación, se presentan algunas consultas que se pueden implementar a partir de los ciclos e iteradores. La función `BM_edge_in_face()` devuelve verdadero si una arista forma parte de una cara dada:

```
bool BM_edge_in_face(BMEdge *edge, BMFace *face) {
        if (edge->loop) {
                BMLoop *l_iter, *l_first;
                l_iter <- l_first <- edge->loop;
                do {
                        if (l_iter->face == face) {
                                return true;
                        }
                } while ((l_iter <- l_iter->radial_next) != l_first);
        }
        return false;
}
```

La función `BM_vert_in_face()` devuelve verdadero si el vértice se encuentra en la cara dada:

```
bool BM_vert_in_face (BMFace *face, BMVert *vert) {
        BMLoop *l_iter, *l_first;
        l_iter <- l_first <- face->l_first;
        do {
                if (l_iter->vert == vert) {
                        return true;
                }
        } while ((l_iter <- l_iter->next) != l_first);
        return false;
}
```

La función `BM_vert_pair_share_face_check()` llama a `BM_vert_in_face()` para verificar si dos vértices comparten una cara, usando un iterador definido sobre el conjunto de caras asociadas a un vértice:

```
bool BM_vert_pair_share_face_check (BMVert *vertA, BMVert *vertB) {
        // Si ambos vértices pertenecen a alguna arista
        if (vertA->edge && vertB->edge) {
                BMIter iter;
                BMFace *face;
                // Itera sobre el conjunto de caras con el vértice vertA
                BM_ITER_ELEM (face, &iter, vertA, BM_FACES_OF_VERT) {
                        // Verifica si face tiene a vertB
                        if (BM_vert_in_face(face, vertB)) {
                                return true;
                        }
                }
        }
        return false;
}
```

9.4. Implementación del componente de modelado

De manera similar a los prototipos mencionados anteriormente, en los Capítulos 7 y 8, al utilizar el marco de desarrollo ShAREd, se tiene resuelta la funcionalidad de visualizar los modelos tridimensionales, usando realidad aumentada, mediante el *plugin* de Vuforia para iOS. Como este prototipo en particular se encuentra en una etapa muy temprana de desarrollo, no se han diseñado aún los *widgets* de consciencia de grupo ni se ha utilizado el *Componente de Intercambio de Datos*.

En la etapa actual de desarrollo, solamente se ha implementado el *Componente de Modelado* de la malla, utilizando la implementación de `BMesh` de Blender, en lenguaje *C* y portándola a *Objective C* para la plataforma iOS. Como parte del *Componente de Modelado*, se han implementado también tres operaciones básicas de edición, que se describen a continuación: una para la creación de la malla, una para la selección de vértices y una para manipular los vértices previamente seleccionados.

9.4.1. Creación de una malla

Utilizando los operadores de nivel medio de `BMesh` en *C*, se crearon algunas funciones de creación de primitivas geométricas en *Objective C*, las cuales se listan a cotinuación:

- `my_create_cube()`: crea un cubo unitario.
- `my_create_circle()`: crea un círculo unitario, aproximado como un polígono regular de n lados, cuyos vértices se encuentran a la misma distancia del origen.
- `my_create_cone()`: crea un cono con base circular.
- `my_create_uvsphere()`: crea una esfera unitaria con U meridianos y V paralelos.
- `my_create_monkey()`: crea una cabeza de chimpancé, que representa a la mascota de Blender, llamada *Suzanne*.

En esta sección, se describe la implementación de dos de estas funciones. En la primera se agregan las caras directamente, mientras que la segunda utiliza los operadores de medio nivel. El cubo se construye en dos fases: primero se crean los ocho vértices de tipo `BMVert`, mediante la función `BM_vert_create()` y luego se crean las seis caras de tipo `BMFace`, usando la función `BM_face_create_quad_tri()`. Esta función solamente necesita tres o cuatro `BMVert`s para construir correctamente caras triangulares o cuadradas, con sus aristas y eslabones.

En el listado siguiente, se muestra una parte de la función `my_create_cube()`, en la que se crean las caras de un cubo unitario y se agregan directamente a la estructura.

```
void my_create_cube(BMesh *mesh) {
        // crea los 8 vértices
        BMVert *v1, *v2, *v3, *v4, *v5, *v6, *v7, *v8;
```

```
        float vec[3], a = 0.5;
        vec[0] = -a;
        vec[1] = -a;
        vec[2] = -a;
        v1 = BM_vert_create(mesh, vec, BM_CREATE_NOP);
        BMO_elem_flag_enable(mesh, v1, VERT_MARK);
        // ...   y así para v2 a v7 ...
        vec[0] = a;
        vec[1] = -a;
        vec[2] = a;
        v8 = BM_vert_create(mesh, vec, BM_CREATE_NOP);
        BMO_elem_flag_enable(mesh, v8, VERT_MARK);
        // crea los 4 lados
        BM_face_create_quad_tri(mesh, v5, v6, v2, v1, BM_CREATE_NOP)
        BM_face_create_quad_tri(mesh, v6, v7, v3, v2, BM_CREATE_NOP)
        BM_face_create_quad_tri(mesh, v7, v8, v4, v3, BM_CREATE_NOP)
        BM_face_create_quad_tri(mesh, v8, v5, v1, v4, BM_CREATE_NOP)
        // crea la base y la tapa
        BM_face_create_quad_tri(mesh, v1, v2, v3, v4, BM_CREATE_NOP)
        BM_face_create_quad_tri(mesh, v8, v7, v6, v5, BM_CREATE_NOP)
}
```

En la función para crear la esfera unitaria, se utilizan los operadores *bmop* de extrusión de aristas. Primero se crea un círculo de U aristas y luego se aplica la extrusión $V-1$ veces hasta formar la esfera. Los polos de la esfera se fusionan a un punto, mediante el uso de un operador que fusiona vértices cercanos (`rem_doubles_verts()`), pues si se encuentran lo suficientemente cerca entre sí, se consideran repetidos. El listado para esta función se muestra a continuación:

```
void my_create_uvsphere (BMesh *mesh) {
        // construye el primer segmento
        phi = 0;
        for (i = 0; i <= V; i++) {
                // al aplicar la extrusión, las normales se alinean
                        hacia afuera
                x = -diametro * sinf(phi);
                y = 0.0;
                z = diametro * cosf(phi);
                vert = BM_vert_create(mesh, (x, y, z), BM_CREATE_NOP);
                BMO_elem_flag_enable(mesh, vert, VERT_MARK);
                if (i != 0) {
                        edge = BM_edge_create(mesh, prevvert, vert,
                                BM_CREATE_NOP);
                        BMO_elem_flag_enable(mesh, edge, EDGE_ORIG);
                }
                phi += 2 PI / V;
                prevvert = vert;
        }

        // extruye el segmento U-1 veces
        for (i = 0; i < U; i++) {
                if (i != 0) {
                        extrude_edge_only edges (mesh,
                                BM_CREATE_NO_DOUBLE);
```

```
                }else {
                        extrude_edge_only edges (mesh,
                                BM_CREATE_NO_DOUBLE, EDGE_ORIG);
                }
        }

        if (i != 0) {
                BMO_op_finish(mesh);
                // elimina vértices repetidos, usando rem_doubles_verts
                        con un valor de umbral
                remove_doubles_verts (mesh, BM_CREATE_NO_DOUBLE,
                        VERT_MARK, umbral);
        }
}
```

9.4.2. Implementación del gesto de selección

Para programar los comportamientos disparados por los eventos táctiles, *Objective-C* proporciona un *kit* de utilidades para el manejo de eventos de interfaz de usuario, llamado `UIKit`, dentro del cual vienen definidas varias clases para el reconocimiento de gestos. Básicamente, se crean una o varias instancias de la familia de clases extendidas de `UIGestureRecognizer`, especializadas en trabajar con un cierto tipo de gesto y se le agregan a la vista actual. También es posible crear subclases de `UIGestureRecognizer`, personalizadas para reconocer gestos nuevos.

En el prototipo propuesto, la vista encargada de hacer el proceso de *render* con *OpenGL* se asigna como delegada del reconocedor de gestos por defecto, que es capaz de procesar varios gestos comunes, como la pulsación simple o múltiple y de monitorear gestos con varios dedos. Para seleccionar los vértices que el usuario toca sobre la pantalla táctil, es necesario reconocer los toques de un solo dedo. Con el método `handleTouches()` se reciben los toques sobre cualquier objeto en la escena y es posible verificar sus propiedades. Se verifica primero el número de dedos involucrados en el toque y luego se procede a obtener las coordenadas del mismo sobre el marcador en la escena 3D.

```
- (void)handleTouches: (NSSet *) touches {
        // Se ignoran los gestos con más de un dedo
        if ([touches allObjects].count > 1)
                return;

        // Obtiene las coordenadas del toque
        CGPoint screenPoint = [[touches anyObject] locationInView:self];

        // Encuentra el punto de intersección con el plano del marcador
        CGPoint location;
        getPointOnMarker(screenPoint.x, screenPoint.y, location);

        // Aplica la selección en el punto location
        [self selectVertsAtLocation:location];
}
```

La función `getPointOnMarker()` utiliza el Algoritmo 9 de la Sección 8.4.4. Con el método `selectVertsAtLocation()` se seleccionan los vértices que se encuentran dentro del área de influencia del toque:

```
- (void)selectVertsAtLocation: (CGPoint) location {
        // define el tamaño del área de influencia del toque
        float delta = 5.0f/meshsize;

        for (int i=0; i < mesh->totvert; i++) {
                if (fabsf(mesh->v[3*i]-location.x/meshsize) < delta &&
                    fabsf(mesh->v[3*i+1]-location.y/meshsize) < delta) {
                        BMVert* v = BM_vert_at_index(mesh, i);
                        // Activa la bandera del vértice selecionado
                        BM_elem_flag_enable(v, BM_ELEM_SELECT);
                }
        }

        // booleano que indica si algún vértice está seleccionado
        is_vert_select = true;
}
```

Con el gesto de doble pulsación, aplicado en cualquier parte de la pantalla táctil, se efectúa una rutina de selección/deselección de todos los vértices de la malla. Si no hay ningún vértice seleccionado, se seleccionan todos, pero si alguno se encuentra seleccionado, entonces se deseleccionan todos. Con ayuda de la variable booleana *is_vert_select* se determina cuál de los dos métodos siguientes ejecutar: `selectAllVerts()` o `unselectAllVerts()`.

```
- (void)selectAllVerts {
        for (int i=0; i < mesh->totvert; i++) {
                BMVert* v = BM_vert_at_index(mesh, i);
                BM_elem_flag_enable(v, BM_ELEM_SELECT);
        }
        // booleano que indica si algún vértice está seleccionado
        is_vert_select = true;
}

- (void)unselectAllVerts {
        for (int i=0; i < mesh->totvert; i++) {
                BMVert* v = BM_vert_at_index(mesh, i);
                BM_elem_flag_disable(v, BM_ELEM_SELECT);
        }
        // booleano que indica si algún vértice está seleccionado
        is_vert_select = false;
}
```

Con estas funciones es posible seleccionar un conjunto de vértices *pintando* sobre ellos. No se tiene implementada una modalidad de deselección, la cual sería muy similar a la de selección. Pero en caso de que sea necesario enmendar un error en la selección, se utiliza la doble pulsación para deseleccionar todo e intentar de nuevo.

9.4.3. Implementación del gesto de transformación

En algunas aplicaciones de visualización de imágenes, es común utilizar los gestos de arrastrar, pellizcar y girar, usando dos dedos. Al arrastrar la imagen, esta se desplaza por el plano de la pantalla. Al pellizcar hacia adentro y hacia afuera, se le aplica un cambio de tamaño. Al girar con dos dedos, la imagen gira como si fuera una fotografía colocada sobre una mesa. Estas son transformaciones afines.

En el presente prototipo, implementamos un gesto que realiza las tres operaciones de transformación sobre la selección de vértices de la malla. Las transformaciones se aplican respecto al plano del marcador, de manera que el gesto funciona mejor si se observa la malla desde arriba.

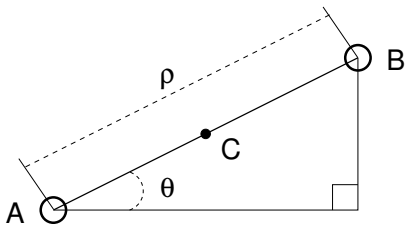


Figura 9.15: Diagrama de los cálculos realizados en el gesto de dos dedos.

Definición del reconocedor de gestos personalizado

Un reconocedor de gestos personalizado extiende la clase `UIGestureRecognizer`. Se definió el reconocedor de gestos `TransformGestureRecognizer` personalizado para reconocer gestos de dos dedos exactamente y tiene las siguientes propiedades:

```
@interface TransformGestureRecognizer : UIGestureRecognizer
        @property (retain) UIView* useThisView;
        @property CGPoint currentPointA;
        @property CGPoint currentPointB;
        @property CGPoint previousPointA;
        @property CGPoint previousPointB;
        @property CGFloat scaleFactor;
        @property CGFloat orientation;
@end
```

Para procesar el gesto de transformación, se necesitan los puntos de toque de los dos dedos para el gesto actual y los del gesto anterior, en caso de haberlo. También se almacena en esta clase el factor de escala y la orientación indicada por las coordenadas de los dedos. Conociendo las coordenadas de los dos puntos A y B, se puede calcular el punto central C del toque, como se ilustra en la Figura 9.15. Además, el segmento de recta AB tiene dos propiedades útiles para implementar este gesto: tiene una longitud ρ y forma un ángulo θ con la horizontal. De este modo, al variar la longitud ρ se puede controlar el factor de escala y, al variar el ángulo de inclinación del segmento, se puede controlar la orientación de la selección.

El nuevo reconocedor de gestos debe implementar algunos de los métodos de su clase superior `UIGestureRecognizer`, los necesarios para realizar su función. En el listado se muestran los métodos de manejo del gesto que se utilizaron: `touchesBegan()` se ejecuta al comenzar el gesto, `touchesMoved()` funciona mientras los dedos involucrados se mueven por la pantalla y finalmente el gesto termina cuando los dedos se separan de la pantalla, ejecutando el método `touchesEnded()`. Sin embargo, en este gesto no se utiliza este último método.

```
- (void)touchesBegan: (NSSet*)touches withEvent: (UIEvent *)event {
        [super touchesBegan:touches withEvent:event];

        // verifica que haya dos dedos en el gesto
        if (touches.count != 2) {
                self.state = UIGestureRecognizerStateFailed;
                return;
        }

        // se definen los dos toques
        UITouch *touchA, *touchB;
        CGFloat x, y;

        // obtiene el arreglo de toques en el gesto
        NSArray *touchesArray = [touches allObjects];
        touchA = [touchesArray objectAtIndex:0];
        self.currentPointA = [touchA locationInView:self.useThisView];
        touchB = [touchesArray objectAtIndex:1];
        self.currentPointB = [touchB locationInView:self.useThisView];

        // obtiene la distancia inicial entre los dos toques
        x = self.currentPointA.x - self.currentPointB.x;
        y = self.currentPointA.y - self.currentPointB.y;
        self.initialDist = sqrtf(x*x + y*y);

        // obtiene la orientación inicial de los dedos
        self.initialOrientation = [self calcAngle:self.currentPointB :
                self.currentPointA];
}
```

Al arrastrar los dedos sobre la pantalla, se utiliza `touchesMoved()` para obtener las nuevas coordenadas para ambos dedos y calcular los nuevos parámetros de la transformación. Se calcula el desplazamiento del centro de los dos dedos, respecto a la posición inicial. Luego se calcula el cambio en la separación entre los dedos, lo cual proporciona el factor de escala. Finalmente, se obtiene la diferencia del ángulo de inclinación de los dedos, respecto al inicial.

```
- (void)touchesMoved:(NSSet*)touches withEvent:(UIEvent *)event {
        [super touchesMoved:touches withEvent:event];
        // obtiene el arreglo de toques en el gesto
        NSArray *touchesArray = [touches allObjects];

        // verifica que siga habiendo dos dedos en el arreglo
        if (touchesArray.count != 2) {
                //self.state = UIGestureRecognizerStateFailed;
                return;
        }
        UITouch *touchA, *touchB;
        CGFloat x, y;

        // obtiene la posición actual y la anterior del primer dedo
        touchA = [touchesArray objectAtIndex:0];
        self.currentPointA = [touchA locationInView:self.useThisView];
        self.previousPointA = [touchA previousLocationInView:self.
                useThisView];

        // obtiene la posición actual y la anterior del segundo dedo
        touchB = [touchesArray objectAtIndex:1];
        self.currentPointB = [touchB locationInView:self.useThisView];
        self.previousPointB = [touchB previousLocationInView:self.
                useThisView];

        // calcula el centro entre los dedos
        x = (self.currentPointA.x + self.currentPointB.x)/2.0f;
        y = (self.currentPointA.y + self.currentPointB.y)/2.0f;
        self.currentCenter = CGPointMake(x, y);

        // calcula el centro anterior
        x = (self.previousPointA.x + self.previousPointB.x)/2.0f;
        y = (self.previousPointA.y + self.previousPointB.y)/2.0f;
        self.previousCenter = CGPointMake(x, y);

        // calcula la separación actual entre los dedos
        x = self.currentPointA.x - self.currentPointB.x;
        y = self.currentPointA.y - self.currentPointB.y;
        CGFloat dist = sqrtf(x*x + y*y);

        // obtiene el factor de escala
        self.scaleFactor = dist / self.initialDist;

        // calcula el incremento en la orientación respecto a la inicial
        self.orientation = [self calcAngle:self.currentPointB :self.
                currentPointA] - self.initialOrientation;
}
```

El método `calcAngle()` calcula el ángulo del segmento de recta que va del punto A al punto B, en radianes entre 0 y 2π. Esta función elimina el problema de ambigüedad en el cuadrante, que tiene la función arco tangente por sí sola. En caso de error, la función devuelve -1.

```
- (float)calcAngle:(CGPoint)pointA :(CGPoint)pointB {
        // se mueve el origen al punto A, nótese la inversión en la
              coordenada Y
        float x = pointB.x - pointA.x;
        float y = pointA.y - pointB.y;

        // evitar una división entre cero
        if (y == 0) {
                if (x > 0) {
                        return M_PI_2;
                } else {
                        return 3 * M_PI_2;
                }
        }

        // calcula el arco tangente
        float arctan = atanf(x/y);

        // corregir el ángulo en función del cuadrante
        // cuadrante I
        if ((x >= 0) && (y > 0)) {
                return arctan;
        }

        // cuadrante II
        else if ((x < 0) && (y > 0)) {
                return arctan + 2 * M_PI;
        }

        // cuadrante III
        else if ((x <= 0) && (y < 0)) {
                return arctan + M_PI;
        }

        // cuadrante IV
        else if ((x > 0) && (y < 0)) {
                return arctan + M_PI;
        }

        // ocurrió un error
        return -1;
}
```

Agregando el reconocedor de gestos al proyecto

Una vez definido el reconocedor de gestos, se agrega al proyecto para ser utilizado en una de las vistas de la aplicación. En este caso, se tiene una sola vista que dibuja la escena 3D mediante *OpenGL*. Esta vista es un delegado del reconocedor del gesto e implementa la interfaz `TransformGestureRecognizerDelegate`. Durante la inicialización de la vista, se crea una instancia de `TransformGestureRecognizer`, encargada de gestionar los gestos de transformación, y se le asigna la propia vista como delegado.

```
TransformGestureRecognizer *gesture =
        [[TransformGestureRecognizer alloc] initWithTarget:self
        action:@selector(doTransform:)];
[gesture setDelegate:self];
gesture.useThisView = self;
[self addGestureRecognizer:gesture];
[gesture release];
```

Cuando la instancia de `TransformGestureRecognizer` reconoce un gesto de dos dedos, la clase delegada ejecuta el método siguiente:

```
- (void)doTransform:(TransformGestureRecognizer *)gesture {
        // Encuentra el punto de intersección del toque
        CGPoint location;
        getPointOnMarker(gesture.currentCenter, location);
        // Obtiene el desplazamiento
        transDelta.x = location.x;
        transDelta.y = location.y;
        // Obtiene el factor de escala
        scaleFactor = gesture.scaleFactor;
        // Obtiene la orientación
        rotateAngle = gesture.orientation;
}
```

Estos valores se aplicarán a los vértices seleccionados, durante el siguiente ciclo de *render* en pantalla, el cual se encuentra implementado en la vista. De este modo, la transformación aplicada a los vértices seleccionados se muestra en tiempo real en pantalla.

9.5. Avance actual del prototipo

Utilizando todo lo anterior, se tiene implementado un prototipo de editor de mallas, que permite visualizar una malla creada utilizando la estructura `BMesh`. También es posible deformarla, de manera interactiva, mediante gestos táctiles intuitivos y familiares, pero sin alterar su topología.

En la Figura 9.16, se muestran dos capturas de pantalla del prototipo en dos dispositivos con resoluciones distintas, un iPad a la izquierda y un iPod a la derecha. La malla corresponde al modelo de Suzanne, la chimpancé mascota de Blender, la cual se muestra en malla de alambre en la imagen izquierda, mientras que en la derecha se muestran también las caras.

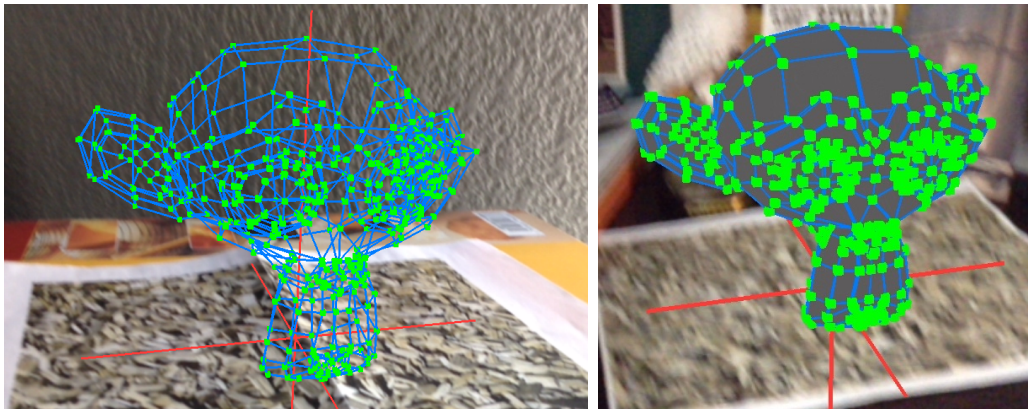


Figura 9.16: Capturas de pantalla del prototipo en dos dispositivos distintos. Se muestran los vértices y aristas de una malla poligonal (izquierda), aunque también se pueden visualizar las caras (derecha).

El usuario puede seleccionar los vértices, utilizando el gesto de arrastre con un solo dedo, y puede desplazar, girar y cambiar de tamaño la selección. Con una doble pulsación, se puede seleccionar o deseleccionar todos los vértices a la vez y con una triple pulsación se activa o desactiva la visualización de las caras, para poder observar la malla de alambre o el modelo poligonal.

Capítulo 10

Conclusiones y trabajo a futuro

10.1. Recapitulación del problema

Tradicionalmente, las aplicaciones de modelado 3D son difíciles de aprender y el proceso de modelado puede llegar a ser lento y tedioso. Las interfaces de usuario suelen ser poco intuitivas, sus herramientas de modelado ofrecen demasiadas opciones que la mayoría de los usuarios, en especial los novatos, nunca utilizan. Adicionalmente, a los usuarios novatos les resulta confuso interpretar las vistas bidimensionales y frecuentemente pierden la cámara virtual en el espacio 3D virtual. Además, la mayoría de las aplicaciones de modelado son monousuario y difíciles de construir.

10.2. Contribuciones

La principal contribución de este trabajo es el marco de desarrollo *ShAREd*, por sus siglas en inglés *Shared Augmented Reality Editing*, que consta de dos partes. Un diseño para una arquitectura modular, especializada en la construcción de aplicaciones de autoría y bosquejo de modelos 3D, en un entorno colaborativo de tipo cara a cara, utilizando realidad aumentada. Y también se proporciona un conjunto de *widgets* personalizables, implementados en Unity y orientados a facilitar la creación de este tipo de aplicaciones, para desarrolladores familiarizados con esta plataforma.

Mediante el uso de los principios de diseño centrado en el usuario y de realidad aumentada, durante el proceso de creación del modelo 3D, se observa un impacto positivo en la facilidad de uso para los usuarios novatos y en su percepción de la carga de trabajo durante la realización de la tarea. Por medio de la visualización con realidad aumentada, se proporciona a los colaboradores una mejor percepción de la forma y estructura tridimensional del modelo 3D, pues se tiene una retroalimentación visual de los cambios efectuados en tiempo real. En opinión de los usuarios durante las pruebas, el uso de esta tecnología hace el editor más interesante y fácil de usar.

Se validó el marco de desarrollo mediante la construcción de tres prototipos de aplicaciones de autoría y bosquejo de tres tipos de modelos 3D distintos y con un nivel de complejidad creciente: una maqueta virtual formada por piezas sólidas, un mapa

de elevación digital y una malla poligonal. En cada uno de estos casos, se explica cómo se puede diseñar e implementar la funcionalidad de modelado, junto con las operaciones necesarias para modificar el modelo 3D.

Hemos verificado que la cantidad de inconsistencias que se pueden acumular en una estructura de datos sencilla, como el mapa de elevación digital, durante sesiones reales de edición colaborativa, con grupos de usuarios inexpertos y sin un mecanismo de control de consistencia, es relativamente pequeña y puede ser tolerable en la práctica. Además, en ciertas tareas, los colaboradores regulan su comportamiento mediante protocolos sociales espontáneos. Los prototipos de editores pueden utilizarse para bocetado de contenidos virtuales y como un medio para dirigir discusiones grupales, tales como pruebas de concepto de un diseño 3D.

En el caso del último prototipo, se hizo un avance importante en la construcción de una herramienta de autoría de mallas poligonales, que consiste en visualizar una malla creada utilizando la estructura `BMesh`, así como deformarla de manera interactiva mediante gestos táctiles intuitivos, pero sin alterar su topología.

10.3. Limitaciones

El marco de desarrollo ShAREd tiene las siguientes limitaciones:

- Si se utiliza solamente la modalidad de realidad aumentada, la movilidad de los usuarios se ve afectada, por estar ligados al marcador impreso, sin el cual las aplicaciones no pueden usarse.

- Los colaboradores necesitan estar reunidos en un espacio, en el que se puedan mover cómodamente alrededor del marcador, y tener acceso a una red WiFi que no se encuentre muy ocupada.

- La funcionalidad de edición colaborativa y opciones de visualización de los prototipos son las elementales, debido principalmente al tiempo limitado de desarrollo. Sin embargo, es posible agregar las características faltantes.

- La interacción táctil con los *widgets* se diseñó tomando en cuenta a los usuarios finales de las aplicaciones. Sin embargo, para personalizarlos y ajustarlos a otras necesidades específicas, así como para crear nuevos diseños desde cero, se requiere que el desarrollador tenga cierto nivel de conocimiento previo en el uso de Unity y C#, Javascript o Boo.

- El único esquema de comunicación que se tiene implementado es del tipo cliente-servidor con autoridad. Para cambiar a esquemas más sofisticados se requiere implementar por completo el módulo, para lo cual el desarrollador necesita estar familiarizado con HLAPI, la API de bajo nivel para redes de Unity.

10.4. Trabajo a futuro

En el estado actual del marco de desarrollo ShAREd, el *Componente de Intercambio de Datos* permite compartir las operaciones entre los colaboradores que se encuentran conectados a la sesión de edición colaborativa, pero sin implementar un esquema para mantener la consistencia de los datos compartidos. Se ha mencionado que, para ciertas aplicaciones, resulta más práctico no tener dicho esquema y, por ello, se recomienda verificar si existe la necesidad de implementarlo para cada tipo de contenido virtual.

Como trabajo a futuro, se implementarán diversos esquemas de consistencia de datos, como la técnica de transformación operacional (*Operational Transformation*, OT), mediante la creación de reglas para transformar las operaciones de modelado de mallas poligonales. Actualmente esta técnica se ha aplicado con éxito en editores de texto colaborativos y algunas aplicaciones de dibujo 2D, como en el caso de *Co-Office* que extiende *Word* y *PowerPoint* a un contexto colaborativo. También se aplicó esta técnica en el proyecto *Co-Maya* para extender una parte de la funcionalidad del popular editor 3D, llamado *Maya*.

En los editores colaborativos existen mecanismos para sincronizar estados completos y, en ocasiones, es posible llevar una bitácora de la evolución de la sesión colaborativa. Estos mecanismos pueden ser usados para implementar operaciones de deshacer/rehacer (*undo/redo*) y protocolos diseñados para soportar la incorporación de colaboradores de manera tardía, i.e., cuando la sesión ya se ha iniciado. A estos usuarios tardíos se les conoce como recién llegados (*latecomers*) [43, 44]. En futuras versiones de los prototipos, deberán implementarse dichos protocolos de soporte a recién llegados y algún mecanismo para deshacer/rehacer operaciones de manera colaborativa y en función del tipo de modelo 3D creado.

Por supuesto, será de vital importancia implementar mecanismos de persistencia de datos, que permitan guardar el modelo 3D creado y exportarlo en formatos que sean útiles, dependiendo del tipo de contenido y el uso que se le pretende dar.

10.4.1. Editor de exhibiciones académicas

En versiones futuras del editor de exhibiciones académicas, será necesario implementar mecanismos para eliminar piezas sólidas de manera concurrente, pues no es deseable que cualquier colaborador elimine el trabajo de otros de manera unilateral, i.e., sin fomentar algún tipo de consenso en la decisión de eliminar objetos. También será necesario conducir estudios con usuarios finales para determinar qué tipo de *widgets* de mobiliario adicionales se necesitan en la práctica. Podemos pensar en *widgets* para representar esculturas, pinturas, maquinaria, vehículos u otros productos que necesiten exhibirse. Dependiendo del uso que se dará a los nuevos elementos, se puede decidir su comportamiento en la aplicación, i.e., si se va a poder girar y sobre cuáles ejes, si se podrá redimensionar en alguna dirección o si se requiere que cambie alguna de sus propiedades, como el color.

Se propone también la posibilidad de agregar imágenes para colocarlas en las notas o sobre las mamparas, como pueden ser imágenes desde el sistema de archivos o fotografías tomadas con la cámara del dispositivo. En este caso, surge el problema de compartir imágenes a través de la red mediante algún tipo de compresión o codificación, aún cuando éstas no sean editables, pues suelen ser de gran tamaño. No es un propósito de este trabajo abordar el problema de edición colaborativa de imágenes.

Adicionalmente, la generación del mapa de ubicación, i.e., la vista superior de la exhibición, sin el uso de realidad aumentada, también plantea un reto peculiar. Para una sede muy grande, las notas de texto pueden no ser visibles en una captura de pantalla. Una posible solución es que las notas miren siempre hacia la cámara, como los *sprites* de tipo cartel publicitario (mejor conocidos como *billboards*) y cuyo tamaño de letra se ajuste de manera dinámica, en caso de que la cámara se aleje o se acerque demasiado.

10.4.2. Editor de terrenos

En el editor de terrenos será de utilidad agregar nuevas opciones de visualización de la superficie, como puede ser mostrar la superficie en malla de alambre, con curvas de nivel, en escala de grises o con una paleta de colores personalizada. También sería posible elegir su color o su textura para que el terreno luzca más real, para ello se pueden agregar ciertos elementos de utilería, como árboles, rocas, cuerpos de agua e incluso construcciones y carreteras.

Una funcionalidad interesante, para un proyecto alternativo inspirado en el presente trabajo, es vincular el editor de terrenos con un sistema de información geográfica (*Geographical Information System*, *GIS*) con el fin de obtener datos de terrenos reales con los que se pueda crear un modelo inicial, para luego modificarlo manualmente. Sería interesante también agregar funciones de procesamiento global para alterar la apariencia del terreno, como los algoritmos de erosión procedural, o para añadir ruido fractal. Es importante implementar mecanismos para guardar el terreno compartido de manera consistente. También es necesario establecer un consenso al momento de cargar nuevos datos desde un archivo, pues al hacerlo se pierden los valores de elevación actuales del terreno. De manera similar, es necesario resolver el problema de restauración del terreno compartido a su estado original (un *reset* colaborativo).

En el algoritmo encargado de calcular la intersección del toque con el marcador, se toma la colisión con el plano XY, lo cual es muy eficiente. Sin embargo, en versiones posteriores, es necesario implementar un algoritmo para calcular la colisión con la malla 3D del terreno, en lugar del plano. La razón se explica en el diagrama de la Figura 10.1, en el que se muestra una versión 2D de un terreno (i.e., una curva) colocado sobre el plano del marcador (la recta gruesa) y tres rayos apuntando desde ángulos diferentes hacia un punto del terreno, ubicado en la montaña más alta. El rayo A se dispara desde una posición cercana al cénit del punto, en el que se desea aplicar el toque, y se calcula su colisión con el marcador, la cual queda por debajo del punto indicado, pero con coordenadas (x, y) muy cercanas a las de dicho punto.

En este caso, el toque se registra cerca del punto deseado y el resultado, al aplicar una operación, es aproximadamente el esperado. Por otro lado, el rayo B tiene un ángulo mayor respecto a la vertical, por lo cual toca al marcador más lejos del punto deseado. De hecho, el toque se registra en la otra montaña del terreno. El ángulo del rayo C es aún mayor y ni siquiera toca la región del marcador en la que se ubica el terreno, por lo cual falla también. En las pruebas con usuarios, notamos que estos siempre esperan que el efecto de la operación ocurra en el punto del terreno al que va dirigido el toque, lo que provoca desconcierto cuando no ocurre así. Es por esto que la implementación de un mejor algoritmo es importante para mejorar la experiencia del usuario.

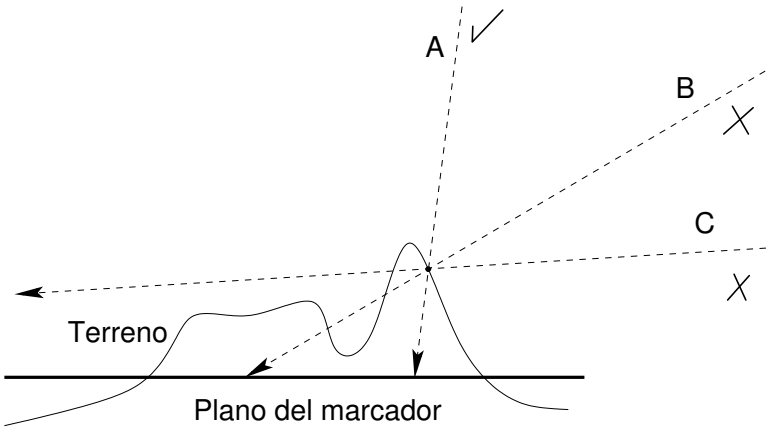


Figura 10.1: Vista lateral de tres rayos incidiendo sobre un punto en el terreno, pero la colisión se registra con el plano del marcador, indicado por la línea gruesa.

10.4.3. Editor de mallas poligonales

El prototipo de editor de mallas poligonales se encuentra en una etapa temprana de desarrollo. Aunado a ello, la estructura `BMesh` es la más complicada de las que han sido implementadas en este proyecto. Además, el proceso de modelado de una malla arbitraria es por demás complejo. Se tiene pues, una gran cantidad de trabajo a futuro para este prototipo.

Es necesario implementar las operaciones de selección y manipulación de aristas y caras, primero en un entorno monousuario y posteriormente extenderlas a un entorno colaborativo. Igualmente es importante verificar las inconsistencias que pueden surgir al compartir y manipular una estructura de datos tan compleja, las cuales se espera sean significativas.

También se introducirán operaciones para agregar y eliminar componentes. Asimismo, será necesario diseñar gestos intuitivos y nuevos *widgets* de interfaz de usuario, para activar y utilizar los operadores de modelado de nivel medio de `BMesh`, como los de extrusión y remoción de vértices repetidos, mencionados anteriormente. Dichos operadores fueron diseñados para ser utilizados por un solo usuario, así que es posible

que deban ser modificados de manera sustancial para utilizarlos en un entorno colaborativo. En particular, es necesario establecer un protocolo para aprobar, en consenso grupal, cuando alguien intente eliminar cualquier elemento. Esta problemática, por sí sola, constituye todo un tema para una tesis completa y el presente trabajo puede servir de base para afrontar este reto en trabajos posteriores.

Otra posibilidad para el futuro es adaptar las herramientas de alto nivel de `BMesh` a un entorno móvil y colaborativo. En principio, el diseño de `BMesh` sugiere que la malla se manipule sólo mediante operadores de Euler, lo cual sería benéfico, pues son estables, fácilmente invertibles y su comportamiento es confiable, dado que ya han sido probados extensivamente. Sin embargo, muchas de las implementaciones actuales de las herramientas y operadores (*bmops*), manipulan directamente la estructura interna de la malla. Los desarrolladores de `BMesh` tienen la meta a futuro de convertir todas las herramientas para que únicamente utilicen operadores de Euler e incluso de agregar algunos operadores nuevos, de ser necesario. Así pues, queda como trabajo a futuro analizar si se pueden aprovechar las implementaciones existentes, aunque estén diseñadas para usarse con Blender en sistemas de escritorio o si será necesario reescribirlas en su totalidad, orientándolas hacia un entorno colaborativo.

Operación de deshacer/rehacer

En la implementación actual de Blender, la operación deshacer (*undo*) se realiza guardando copias de la estructura de la malla, antes de aplicar las operaciones, y sobrescribiendo la malla actual con la copia previa, cuando se quiere deshacer la operación más reciente. La desventaja de esta implementación es que puede ocasionar un alto consumo de memoria y posibles retardos causados por realizar frecuentemente copias de respaldo, para mallas de muchos polígonos.

Sin embargo, en el diseño de la estructura `BMesh`, se contempla la implementación de una operación deshacer más eficiente, en la que se mantiene una lista ordenada con los operadores de Euler que han sido aplicados a la malla. Dado que dichos operadores son invertibles, deshacerlos en el orden inverso cumplirá la tarea de deshacer el efecto de la última herramienta ejecutada. Desafortunadamente, como se mencionó antes, no todas las herramientas de modelado se apegan a la regla de aplicar solamente operadores de Euler, con lo cual se dificulta la implementación de esta lista de operaciones para deshacer y por ello la operación deshacer no se puede resolver de esta manera.

A este respecto, es interesante mencionar el proyecto `BMeshUndo` de Nicholas Bishop[1], que se encuentra actualmente en desarrollo e intenta crear una bitácora dinámica (`BMLog`) para almacenar operadores de Euler junto con operaciones no topológicas, como las transformaciones afines. Desafortunadamente aún quedan varios problemas por resolver en dicho proyecto, pues uno de sus objetivos es no modificar, en lo posible, la implementación actual de `BMesh`. En el presente prototipo se ha dejado esta funcionalidad como trabajo a futuro, pues además de los problemas relacionados con

[1] https://wiki.blender.org/index.php/User:Nicholasbishop/BMeshUndo

la estructura `BMesh`, en este caso también es necesario considerar la complicaciones que surgen en un entorno colaborativo. La estructura `BMesh` es la más general para trabajar con mallas en este proyecto, de manera que una solución al problema de implementar la operación deshacer, en esta estructura, puede ser utilizada en los otros tipos de modelo 3D.

Así pues, es posible observar que para cada tipo de contenido que hemos introducido en este trabajo, se abre una línea propia de investigación para proyectos futuros, con amplias oportunidades de mejorar el diseño e implementación para cada *Componente de Modelado* y posiblemente aplicarlo en la solución de nuevos problemas.

10.4.4. Líneas de investigación alternativas

Durante el desarrollo de este trabajo, se exploraron algunas líneas de investigación alternativas, i.e., que apuntan a direcciones distintas al enfoque que se adoptó finalmente, pero que decidimos no tomar por diversos motivos. En esta sección mencionamos los avances logrados sobre dichas líneas de investigación y los motivos por los cuales no continuamos es esas direcciones, pero que constituyen buenas oportunidades para trabajo a futuro. En este trabajo se exploraron los siguientes temas: uso de la realidad aumentada mediante el paradigma de *lente mágico*, utilizando HMDs para realidad aumentada (*Vuzix HMD*) y manipulación directa de las entidades de la malla, mediante el reconocimiento y seguimiento visual de las puntas de los dedos o utilizando sensores de profundidad, ya sea en móviles (*Structure Sensor*) o en PC (*Leap Motion*).

Realidad aumentada en dispositivos de despliegue montados sobre la cabeza

Al principio del proyecto se propuso un escenario en el que los colaboradores utilizarían un HMD para realidad aumentada, como el de *Vuzix*, para efectuar el despliegue estereoscópico del modelo virtual junto con la escena real, utilizando una computadora de escritorio. Los HMDs para realidad aumentada traen integradas dos cámaras Web, utilizadas para capturar video de la escena real y mostrarlo en el par de pantallas, una para cada ojo. Como las cámaras están separadas un distancia similar a la de los ojos del usuario promedio, se tiene una vista estereoscópica e inmersiva, tanto del entorno real como de los objetos virtuales.

Sin embargo, se tuvo una dificultad técnica, debida al montaje desde fábrica de las cámaras del dispositivo de *Vuzix*. Ambas cámaras vienen conectadas internamente mediante un solo *hub USB* en lugar de utilizar puertos separados. Esto nos impidió obtener las capturas de ambas cámaras al mismo tiempo, mediante las utilidades de captura de imagen que ofrece MacOS, por lo cual no fue posible obtener el efecto estereoscópico deseado. Caber mencionar que los fabricantes ofrecen un controlador del dispositivo (*driver*), que supuestamente permite utilizar ambas cámaras, pero sólo soporta *Windows*.

Figura 10.2: Montaje de hardware para la captura de video estereoscópico en un sistema de escritorio. Se muestra el video aumentado con un objeto virtual, mostrado también en estéreo.

Sin embargo, se hizo una prueba de concepto, sustituyendo las cámaras del dispositivo por dos cámaras Web comunes. El montaje de hardware que se realizó se muestra en la Figura 10.2, donde se aprecian las dos cámaras Web separadas aproximadamente 7 cm. y colocadas sobre el monitor de una computadora de escritorio. Se muestra el par estereoscópico generado a partir de la captura de video con dichas cámaras y al cual se ha agregado un modelo virtual. Al visualizar el par estereoscópico con el HMD de Vuzix, se percibe el efecto estereoscópico y la ilusión inmersiva de que el objeto virtual existe dentro de la escena real. Finalmente, se decidió mantener el proyecto en la línea de realidad aumentada basada en dispositivos móviles, dejando el uso del HMD para un proyecto posterior.

Otra posibilidad, que surge de manera natural al tratar el tema de los HMDs y que es factible explorar a futuro, es realizar la sesión de modelado colaborativo utilizando realidad virtual. Un ejemplo de una aplicación de dibujo y escultura en realidad virtual es el proyecto *Quill* de *Oculus* [2], que consiste en utilizar el famoso HMD llamado *Oculus Rift* y un par de controles manuales, basados en el giroscopio de los dispositivos móviles, para ser seguidos en el espacio 3D, así como dibujar y esculpir formas en el espacio virtual tridimensional, mediante el software llamado *Quill*. Este sistema es monousuario y requiere un poderoso sistema de escritorio.

Seguimiento visual de manos y dedos

Otra idea que se abordó es el desarrollo de un algoritmo de detección visual y seguimiento de las manos y dedos del usuario, a partir del color de piel mediante técnicas de Visión por Computadora. Se implementó un prototipo que realiza las primeras etapas del algoritmo de detección visual del color de piel. El algoritmo se ejecuta en un sistema de escritorio, utilizando el mismo montaje de hardware de la Figura 10.2.

[2] https://www.engadget.com/2016/11/20/oculus-vr-painting-tool-quill-free/

Las etapas del procesamiento de imágenes que se implementaron son:

- se captura el video en estéreo usando las dos cámaras,
- se transforma el color de los pixeles del espacio de color RGB a coordenadas en el espacio cromático YUV,
- se descarta la componente de intensidad Y y se aplica un umbral de segmentación a las zonas cuyas coordenadas de color UV se encuentren en cierta región, determinada de manera experimental usando imágenes de varios tonos de piel,
- finalmente, se obtiene una máscara binaria que indica las zonas de la imagen en las que se detectó el color de piel.

En las etapas posteriores, que están por implementarse:

- se analizará la máscara con el fin de encontrar las puntas de regiones alargadas, que serían reconocibles como dedos y obtener sus coordenadas en la imagen,
- luego se utilizará la matriz de transformación de pose calculada por Vuforia, para estimar los rayos alineados con la cámara del dispositivo y que pasan por la punta de los dedos detectados,
- finalmente, con la información de paralaje de cada dedo en el par estereoscópico, se estimará la distancia del dedo a la cámara, para calcular colisiones con los objetos virtuales.

En caso de que la salida de coordenadas resultara ruidosa, se pueden utilizar filtros de Kalman para estabilizar el seguimiento. En la etapa de transformación del color, se elije el esquema de color YUV, pues es ampliamente utilizado en la literatura por su habilidad de detectar el tinte de los objetos, sin ser afectado substancialmente por la intensidad de la iluminación. En la Figura 10.3 se muestra el resultado obtenido al aplicar segmentación del color de piel al par estereoscópico. El inverso de la máscara se utilizó para obscurecer las zonas que no tienen color piel y así resaltar aquellas zonas que sí lo tienen. El par estereoscópico se puede ver usando la técnica de *hacer bizcos*.

Sin embargo, decidimos dejar este escenario como trabajo a futuro, debido a que el tiempo disponible para desarrollarlo es insuficiente y preferimos invertir este tiempo en hacer un diseño más cuidadoso para el escenario de realidad aumentada móvil.

Seguimiento de dedos con sensores de profundidad

Posteriormente, se exploró la posibilidad de mezclar el seguimiento visual de los marcadores de realidad aumentada de Vuforia junto con el seguimiento de manos y dedos basado en sensores de profundidad. Para ello, es posible utilizar el dispositivo *Leap Motion* en el escenario de escritorio y el dispositivo *Structure Sensor* en el escenario móvil. Sin embargo, notamos que el dispositivo *Structure Sensor* está diseñado

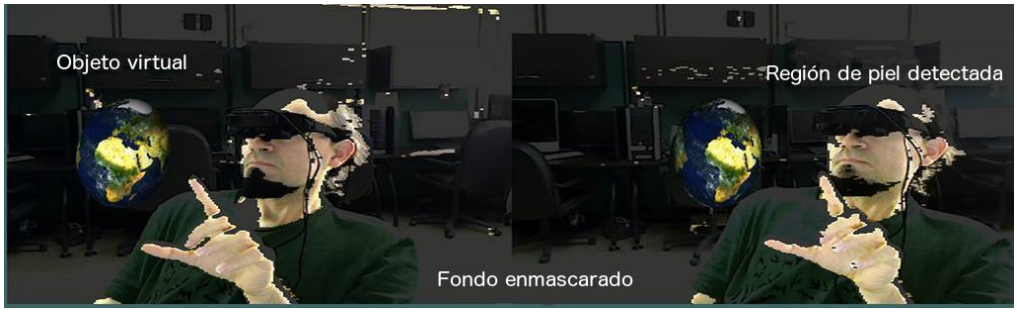


Figura 10.3: Segmentación del color de piel aplicada a la señal de video estereoscópico.

para detectar profundidad sólo a partir de los 40 cm. al frente del iPad. Esta distancia está alejada del alcance de las manos del usuario, de manera que este dispositivo resulta incómodo para el escenario planteado. Además, aún no se libera su SDK para Android, solamente existe para iOS. Al final optamos por seguir la tendencia actual en las aplicaciones de realidad aumentada móvil, que es el uso la pantalla táctil como medio principal de interacción.

Publicaciones

Andrés Cortés-Dávalos y Sonia Mendoza, AR-based Modeling of 3D Objects in Multi-user Mobile Environments, In *Proceedings of the 22nd International Conference on Collaboration and Technology*, Springer International Publishing Switzerland, T. Yuizono et al. (Eds.): CRIWG 2016, LNCS 9848, pp. 21–36, 2016. ISBN: 978-3-319-44799-5, DOI: 10.1007/978-3-319-44799-5 3, url: http://dx.doi.org/10.1007/978-3-319-44799-5_3

Andrés Cortés-Dávalos y Sonia Mendoza, Layout Planning for Academic Exhibits using Augmented Reality, In *Proceedings of the 2016 13th International Conference on Electrical Engineering, Computing Science and Automatic Control*, CCE 2016, IEEE, CFP16827-USB, ISBN: 978-1-5090-3511-3/16.

Andrés Cortés-Dávalos y Sonia Mendoza, Collaborative Web Authoring of 3D Surfaces using Augmented Reality on Mobile Devices, In *Proceedings of the 2016 IEEE/-WIC/ACM International Conference on Web Intelligence*, WI 2016, IEEE Computer Society, ISBN: 978-1-5090-4470-2/16, DOI: 10.1109/WI.2016.113

Andrés Cortés-Dávalos y Sonia Mendoza, Augmented Reality-based Groupware for Editing 3D Surfaces on Mobile Devices, In *Proceedings of the 2016 International Conference on Collaboration Technologies and Systems*, CTS 2016, IEEE Computer Society, ISBN: 978-1-5090-2300-4/16, DOI: 10.1109/CTS.2016.63

Bibliografía

[1] Agustina, F. Liu, S. Xia, H. Shen, and C. Sun. Comaya: incorporating advanced collaboration capabilities into 3d digital media design tools. In *Proceedings of the 2008 ACM conference on Computer supported cooperative work*, CSCW '08, pages 5–8, New York, NY, USA, 2008. ACM.

[2] K. H. Ahlers, A. Kramer, D. E. Breen, P.-Y. Chevalier, C. Crampton, E. Rose, M. Tuceryan, R. T. Whitaker, and D. Greer. Distributed augmented reality for collaborative design applications. *Computer Graphics Forum*, 14(3):3–14, 1995.

[3] J. Amores, X. Benavides, and P. Maes. Showme: A remote collaboration system that supports immersive gestural communication. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '15, pages 1343–1348, New York, NY, USA, 2015. ACM.

[4] J. Amores, L. Salle-URL, X. Benavides, M. Comın, A. Fusté, P. Pla, and D. Miralles. Smart avatars: using avatars to interact with objects. 2012.

[5] M. Andreetto, N. Brusco, and G. M. Cortelazzo. Automatic 3d modeling of textured cultural heritage objects. *IEEE Transactions on Image Processing*, 13(3):354–369, March 2004.

[6] M. A. Armstrong. *Basic topology*. Springer Science & Business Media, 2013.

[7] C. Arth, R. Grasset, L. Gruber, T. Langlotz, A. Mulloni, and D. Wagner. The history of mobile augmented reality. *CoRR*, abs/1505.01319, 2015.

[8] I. Barakonyi, W. Frieb, and D. Schmalstieg. Augmented reality videoconferencing for collaborative work. In *Proc. of the 2nd Hungarian Conference on Computer Graphics and Geomtery,Budapest*, 1996.

[9] X. Benavides, J. Amores, and P. Maes. Remot-io: A system for reaching into the environment of a remote collaborator. In *Adjunct Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15 Adjunct, pages 99–100, New York, NY, USA, 2015. ACM.

[10] B. Benes and R. Forsbach. Layered data representation for visual simulation of terrain erosion. In *Computer Graphics, Spring Conference on, 2001.*, pages 80–86. IEEE, 2001.

[11] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. 1987.

[12] M. Billinghurst and H. Kato. Collaborative augmented reality. *Commun. ACM*, 45(7):64–70, July 2002.

[13] M. Billinghurst, H. Kato, N. R. Hedley, L. Postner, and R. May. Explorations in the use of augmented reality for geographic visualization. *Presence: Teleoperators and Virtual Environments*, 11(2):119 – 133, 2006.

[14] M. Billinghurst, H. Kato, and I. Poupyrev. The magicbook: a transitional augmented reality interface. *Computers & Graphics*, 25(5):745 – 753, 2001. Mixed realities - beyond conventions.

[15] M. Billinghurst, I. Poupyrev, H. Kato, and R. May. Mixing realities in shared space: an augmented reality interface for collaborative computing. In *Multimedia and Expo, 2000. ICME 2000. 2000 IEEE International Conference on*, volume 3, pages 1641–1644 vol.3, 2000.

[16] M. Billinghurst, S. Weghorst, and I. Furness, T. Shared space: An augmented reality approach for computer supported collaborative work. *Virtual Reality*, 3(1):25–36, 1998.

[17] O. Bimber and R. Raskar. *Spatial Augmented Reality: Merging Real and Virtual Worlds*. A K Peters, Ltd, 2006. http://www.spatialar.com.

[18] S. Cawood and M. Fiala. *Augmented reality : a practical guide*. Pragmatic Bookshelf, 2007.

[19] X. Chen, B. Zhou, F. Lu, L. Wang, L. Bi, and P. Tan. Garment modeling with a depth camera. *ACM Trans. Graph.*, 34(6):203:1–203:12, Oct. 2015.

[20] I. Choi and S. Follmer. Wolverine: A wearable haptic interface for grasping in vr. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16 Adjunct, pages 117–119, New York, NY, USA, 2016. ACM.

[21] G. F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[22] G. J. P. de Carpentier and R. Bidarra. Interactive gpu-based procedural heightfield brushes. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, FDG '09, pages 55–62, New York, NY, USA, 2009. ACM.

[23] J. D. Denning and F. Pellacini. Meshgit: Diffing and merging meshes for polygonal modeling. *ACM Trans. Graph.*, 32(4):35:1–35:10, July 2013.

[24] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18(2):399–407, June 1989.

[25] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, and R. L. Phillips. *Introducción a la Graficación por Computador*. Addison-Wesley Iberoamericana, 1996. ISBN: 0-201-62599-7.

[26] K. Fujii, S. S. Russo, P. Maes, and J. Rekimoto. Moveme: 3d haptic support for a musical instrument. 2015.

[27] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable Object-Oriented software*. Addison-Wesley, 1994. ISBN: 0-201-63361-2.

[29] M. Gardner. *Los Porqués de un Escriba Filósofo*. Barcelona, España : Tusquets, 2a. edition, Junio 2001. ISBN: 84-7223-131-3.

[30] T. Göhnert, N. Malzahn, and H. U. Hoppe. A flexible multi-mode undo mechanism for a collaborative modeling environment. In *International Conference on Collaboration and Technology*, pages 142–157. Springer, 2009.

[31] S. Greenberg and R. Bohnet. Group sketch: A multi-user sketchpad for geographically-distributed small groups. 1990.

[32] S. Greenberg and C. Gutwin. Implications of we-awareness to the design of distributed groupware tools. *Computer Supported Cooperative Work (CSCW)*, 25(4):279–293, 2016.

[33] S. Greenberg and D. Marwood. Real time groupware as a distributed system: concurrency control and its effect on the interface. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 207–217. ACM, 1994.

[34] V. Heun, S. Kasahara, and P. Maes. Smarter objects: Using ar technology to program physical objects and their interactions. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '13, pages 961–966, New York, NY, USA, 2013. ACM.

[35] S. Hoober and E. Berkman. *Designing mobile interfaces*. O'Reilly Media, Inc., 2011.

[36] K. Jeong and H. Moon. Object detection using fast corner detector based on smartphone platforms. In *2011 First ACIS/JNU International Conference on Computers, Networks, Systems and Industrial Engineering*, pages 111–115, May 2011.

[37] C. H.-L. Kao, E. Dreshaj, J. Amores, S.-w. Leigh, X. Benavides, P. Maes, K. Perlin, and H. Ishii. Clayodor: Retrieving scents through the manipulation of malleable material. In *Proceedings of the Ninth International Conference on Tangible, Embedded, and Embodied Interaction*, TEI '15, pages 697–702, New York, NY, USA, 2015. ACM.

[38] N. Karlsson, G. Li, Y. Genc, A. Huenerfauth, and E. Bononno. iar: An exploratory augmented reality system for mobile devices. In *Proceedings of the 18th ACM Symposium on Virtual Reality Software and Technology*, VRST '12, pages 33–40, New York, NY, USA, 2012. ACM.

[39] S. Kasahara, V. Heun, A. S. Lee, and H. Ishii. Second surface: Multi-user spatial collaboration system based on augmented reality. In *SIGGRAPH Asia 2012 Emerging Technologies*, SA '12, pages 20:1–20:4, New York, NY, USA, 2012. ACM.

[40] H. Kato and M. Billinghurst. Marker tracking and hmd calibration for a video-based augmented reality conferencing system. In *Augmented Reality, 1999. (IWAR '99) Proceedings. 2nd IEEE and ACM International Workshop on*, pages 85–94, 1999.

[41] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[42] M. Lau, M. Hirose, A. Ohgawara, J. Mitani, and T. Igarashi. Situated modeling: a shape-stamping interface with tangible primitives. In *Proceedings of the Sixth International Conference on Tangible, Embedded and Embodied Interaction*, TEI '12, pages 275–282, New York, NY, USA, 2012. ACM.

[43] S. Lukosch. Adaptive and transparent data distribution support for synchronous groupware. In *International Conference on Collaboration and Technology*, pages 255–274. Springer, 2002.

[44] S. Lukosch. Transparent latecomer support for synchronous groupware. In *International Conference on Collaboration and Technology*, pages 26–41. Springer, 2003.

[45] B. B. Mandelbrot. *The fractal geometry of nature*, volume 173. Macmillan, 1983.

[46] D. Mogilev, K. Kiyokawa, M. Billinghurst, and J. Pair. Ar pad: An interface for face-to-face ar collaboration. In *CHI '02 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '02, pages 654–655, New York, NY, USA, 2002. ACM.

[47] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. In *ACM Siggraph Computer Graphics*, volume 23, pages 41–50. ACM, 1989.

[48] T.-J. Nam and K. Sakong. Collaborative 3d workspace and interaction techniques for synchronous distributed product design reviews. *International Journal of Desing (Online)*, 3(1), 2009.

[49] K. Perlin and E. M. Hoffert. Hypertexture. In *ACM SIGGRAPH Computer Graphics*, volume 23, pages 253–262. ACM, 1989.

[50] N. Ranasinghe and E. Y.-L. Do. Virtual sweet: Simulating sweet sensation using thermal stimulation on the tip of the tongue. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16 Adjunct, pages 127–128, New York, NY, USA, 2016. ACM.

[51] N. Ranasinghe, P. Jain, D. Tolley, S. Karwita, S. Yilei, and E. Y.-L. Do. Ambiotherm: Simulating ambient temperatures and wind conditions in vr environments. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16 Adjunct, pages 85–86, New York, NY, USA, 2016. ACM.

[52] S. E. Reed, O. Kreylos, S. Hsi, L. H. Kellogg, G. Schladow, M. B. Yikilmaz, H. Segale, J. Silverman, S. Yalowitz, and E. Sato. Shaping Watersheds Exhibit: An Interactive, Augmented Reality Sandbox for Advancing Earth Science Education. *AGU Fall Meeting Abstracts*, Dec. 2014.

[53] E. Rosten and T. Drummond. Machine learning for high-speed corner detection. In *European conference on computer vision*, pages 430–443. Springer, 2006.

[54] G. Salvati, C. Santoni, V. Tibaldo, and F. Pellacini. Meshhisto: Collaborative modeling by sharing and retargeting editing histories. *ACM Trans. Graph.*, 34(6):205:1–205:10, Oct. 2015.

[55] D. Schmalstieg, Z. Szalavri, A. Fuhrmann, and M. Gervautz. Studierstube: An environment for collaboration in augmented reality. *Virtual Reality*, 3(1):37–48, 1998.

[56] B.-K. Seo, J. Choi, J.-H. Han, H. Park, and J.-I. Park. One-handed interaction with augmented virtual objects on mobile devices. In *Proceedings of The 7th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry*, VRCAI '08, pages 8:1–8:6, New York, NY, USA, 2008. ACM.

[57] B. St-Aubin, M. Mostafavi, S. Roche, and N. Dedual. A 3d collaborative geospatial augmented reality system for urban design and planning purposes. In *Proceedings of Canadian Geomatics Conference, Calgary, AB, Canada, June*, pages 15–18, 2010.

[58] M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar. Wysiwis revised: Early experiences with multiuser interfaces. *ACM Trans. Inf. Syst.*, 5(2):147–167, Apr. 1987.

[59] D. Sun and C. Sun. Context-based operational transformation in distributed collaborative editing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 20(10):1454–1470, Oct 2009.

[60] I. E. Sutherland. The ultimate display. 1965. http://www.informatik.umu.se/~jwworth/TheUltimateDisplay.pdf.

[61] G. S. Tejada. Biblioteca para mantener la coherencia de datos compartidos en dispositivos móviles. Master's thesis, CINVESTAV, 2011.

[62] J. Tidwell. *Designing interfaces*. O'Reilly Media, Inc., 2010.

[63] D. Wagner, T. Pintaric, F. Ledermann, and D. Schmalstieg. Towards massively multi-user augmented reality on handheld devices. In H.-W. Gellersen, R. Want, and A. Schmidt, editors, *Pervasive Computing*, volume 3468 of *Lecture Notes in Computer Science*, pages 208–219. Springer Berlin Heidelberg, 2005.

[64] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, and D. Schmalstieg. Pose tracking from natural features on mobile phones. In *Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality*, ISMAR '08, pages 125–134, Washington, DC, USA, 2008. IEEE Computer Society.

[65] K. Weiler. The radial edge structure: a topological representation for non-manifold geometric boundary modeling. *Geometric modeling for CAD applications*, pages 3–36, 1988.

[66] A. Wibowo, D. Sakamoto, J. Mitani, and T. Igarashi. Dressup: a 3d interface for clothing design with a physical mannequin. In *Proceedings of the Sixth International Conference on Tangible, Embedded and Embodied Interaction*, TEI '12, pages 99–102, New York, NY, USA, 2012. ACM.