Desarrollo de un visor modelador 3D colaborativo

Proyecto fin de carrera

Ingeniería Informática

Moisés J. Bonilla Caraballo

Tutor: Agustín Trujillo Pino

Las Palmas de Gran Canaria, Diciembre de 2014

Índice de contenido

1.Introducción....1
1.1.Motivación y descripción del proyecto....1
2.Estado del arte....3
2.1.Verse....3
2.2.Clara.io....4
2.3.Real-time DWG file collaboration - AutoCAD 360....5
2.4.RealXtend Tundra SDK....6
3.Objetivos....7
3.1.Objetivos directos....7
3.2.Objetivos indirectos....7
4.Requisitos hardware y software....9
4.1.Requisitos Hardware....9
4.1.1.OpenGL....9
4.2.Requisitos Software....10
4.2.1.Sistema operativo....10
4.2.2.Herramientas de desarrollo....10
4.2.3.Librerías....14
5.Análisis....17
5.1.Metodología....17
5.2.Glosario de conceptos....17
5.3.Funcionalidades del programa....18
5.3.1.Software cliente....18
5.3.2.Software servidor....19
5.4.Desafíos de la programación distribuida....20
5.4.1.Bloqueo de recursos....20
5.4.2.Diferentes órdenes de bytes (Endianness)....20
5.4.3.Diferentes tamaños de los tipos de datos....21
5.4.4.Serialización....21
5.4.5.Salvado y recuperado de escenas....21
5.5.Análisis de funcionalidades proporcionadas por terceros....21
5.5.1.OpenGL y su pipeline de renderizado....21
5.5.2.Qt y su mecanismo de señales y slots....24
5.5.3.Boost Asio y la E/S asíncrona....25
6.Diseño....27
6.1.Aspectos generales....27
6.1.1.Arquitectura de red....27
6.1.2.Privacidad....27
6.1.3.Sincronización....27
6.1.4.Usuarios y recursos....27
6.1.5.Bloqueo de Recursos....28
6.2.Comunicación cliente – servidor....30
6.2.1.Packables, comandos, paquetes....30
6.2.2.Sincronización....34
6.3.Diseño del cliente....36
6.3.1.Usuarios....36
6.3.2.Recursos....36
6.3.3.Entidades....37

6.3.4.La clase auxiliar OpenGL y los shaders de COMO....37
6.3.5.Mallas....41
6.3.6.Luces y cámaras....43
6.3.7.Sincronizando la escena. Clases Scene, ServerInterface y ServerWriter....44
6.3.8.Gestores de recursos....45
6.3.9.Primitivas....53
6.3.10.Factorías de mallas del sistema....58
6.3.11.Interfaz de usuario....59
6.4.Diseño del servidor....59
6.4.1.Visión general....59
6.4.2.La clase Scene....61
6.4.3.La clase ResourcesSynchronizationLibrary....61
6.5.Jerarquía de directorios....65
6.5.1.Visión general....65
6.5.2.La carpeta de datos (bin/X/data/)....65
6.5.3.La carpeta de código (src/)....66
7.Desarrollo....69
7.1.El origen: un plugin para Blender....69
7.2.El primer incremento: un manipulador de cubos offline....69
7.3.El segundo incremento: un manipulador de cubos online....71
7.4.El tercer incremento: incluyendo iluminación y primitivas....72
7.5.El cuarto incremento: importando primitivas desde el cliente + texturas....74
7.6.El quinto incremento: múltiples luces + mallas del sistema....75
7.7.El sexto incremento: cámara de la escena y carga y salvado de escenas....76
8.Conclusión....79
9.Trabajo futuro....81
9.1.Optimizar la sincronización del histórico de comandos....81
9.2.Reciclaje de IDs....81
9.3.Feedback al cliente sobre el estado de la sincronización....81
9.4.Fragmentación de paquetes y comandos....81
10.Anexo A: Manual de usuario....83
10.1.COMO (Cooperative Modeller)....83
10.1.1.About COMO....83
10.1.2.Features....83
10.1.3.Initializing COMO....83
10.1.4.Creating a scene....84
10.1.5.Connecting to a created scene....85
10.2.GUI overview....86
10.3.Working with entities....87
10.3.1.Defining an entity....87
10.3.2.Adding entities to the scene....87
10.3.3.Selecting and unselecting entities....87
10.3.4.Transforming entities....88
10.3.5.The properties panel....90
10.4.The materials editor....91
10.5.The texture walls editor....91
10.5.1.Introduction....91
10.5.2.Applying a texture to a geometry's wall (face)....92
10.6.Closing the server and saving the scene....95

1. Introducción

1.1. Motivación y descripción del proyecto

Los gráficos 3D inundan nuestra vida diaria, ya sea de manera directa o indirecta. Ayudar al diseño de los productos que usamos a diario, simular experimentos, o dar forma a increíbles mundos de fantasía, son sólo algunas de las muchas necesidades que nos ayudan a satisfacer.

Tal como sucede en casi todos los campos del desarrollo humano, el trabajo de una sola persona no tarda en resultar insuficiente. El aumento en la demanda exige que equipos cada vez mayores de profesionales se coordinen para desarrollar los productos y servicios requeridos por los usuarios. Allí donde el campo de aplicación y la tecnología lo permiten, las redes de computadores se han convertido en aliadas inestimables para coordinar y optimizar la producción.

Con la intención de aprovechar las redes de computadores en la producción de gráficos tridimensionales, se presenta en este documento el desarrollo de un visor y modelador 3D colaborativo. Se trata de un software que permite que varios usuarios se conecten a través de la red y trabajen al unísono sobre la misma escena 3D. **Esta aplicación recibe el nombre de COMO (Cooperative Modeller)**.

2. Estado del arte

Estudiando el mercado, se ha encontrado algunos proyectos que comparten bastantes similitudes con COMO. En los siguientes subapartados se describe someramente algunos de estos proyectos.

2.1. Verse

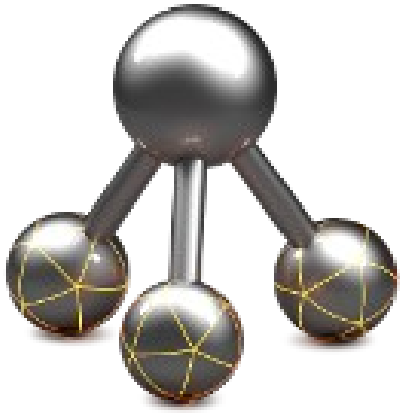


Ilustración 2.1: Logo de Verse 2.0

***Verse*[1] es un protocolo de red que permite compartir datos 3D en tiempo real.** Al ser un protocolo, permite que los usuarios que trabajan sobre una misma escena puedan usar cada uno su software de preferencia, siempre que éste implemente el protocolo *Verse*.

Actualmente se está desarrollando un "fork" con la versión 2.0 del protocolo *Verse*[2], cuya librería, aún en un estado muy temprano del desarrollo, se encuentra bajo una licencia BSD. El protocolo sigue una arquitectura cliente – servidor, se apoya en UDP y cuenta con un sistema de autenticación de usuarios.

Aunque bastante prometedor, el protocolo Verse no parece estar desarrollándose activamente ahora mismo. La sección de noticias del desarrollador de *Verse* 1.0[3] no se actualiza desde Octubre del 2013, y sus últimas actualizaciones no se refieren al propio proyecto *Verse*. El desarrollo de un plugin de *Verse* para *Blender* parece no haberse continuado más allá de la versión 2.4 del modelador[4]. Por su parte, el repositorio de Verse 2.0[5] muestra un estado temprano y no muy activo de su desarrollo.

1 Página web de Verse (versión 1): http://www.quelsolaar.com/verse/

2 Sitio web de Verse 2.0: http://verse.github.io/

3 Sección de noticias del sitio web de Verse 1.0: http://news.quelsolaar.com/#home

4 User documentation for Verse integration to Blender: http://wiki.blender.org/index.php/Dev:2.4/Source/Extensions/Verse/Integration_in_Blender_Documentation

5 Repositorio de Verse 2.0 en Github: https://github.com/verse/verse

2.2. Clara.io

Ilustración 2.2: Logo de Clara.io

Clara.io[6] es una aplicación de modelado, animación y renderizado en la nube. Una de sus principales bazas es que se trata de una aplicación web, por lo que permite crear escenas 3D directamente desde el navegador y sin necesidad de instalar software adicional.

Clara.io ofrece también un SDK que permite que se le añadan plugins[7]. El núcleo de Clara.io hace uso de HTML5, Javascript, WebGL y Three.js[8].

Con Clara.io también es posible la edición simultánea de una escena por parte de múltiples usuarios. A pesar de encontrarse aún en fase Beta, provee de grandes funcionalidades como el soporte para alrededor de 30 formatos de archivos 3D o un completo sistema de materiales e iluminación fotorrealistas.

A diferencia de Verse, **Clara.io se encuentra en un estado bastante activo de desarrollo y cuenta con una gran base de usuarios registrados**.

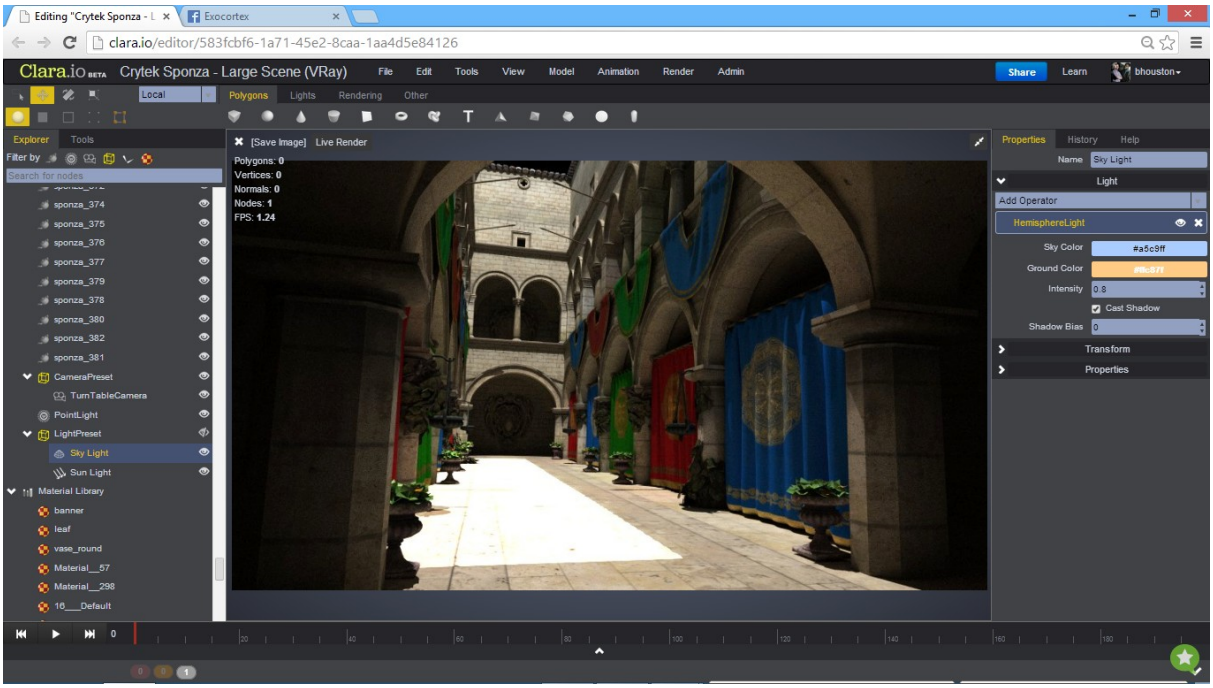


Ilustración 2.3: Clara.io - Ejemplo de escena 3D (imagen obtenida desde la Wikipedia)

6 Sitio web de Clara.io: https://clara.io/

7 Documentación sobre el SDK de Clara.io: https://clara.io/learn/sdk

8 Artículo de *Wikipedia* sobre *Clara.io*: http://en.wikipedia.org/wiki/Clara.io

2.3. Real-time DWG file collaboration - AutoCAD 360

Ilustración 2.4: Logo de AutoCAD 360

El canal de AutoCAD 360 en Youtube[9] publicó, en Marzo del 2011, un vídeo titulado "Realtime DWG file collaboration - AutoCAD 360"[10]. En él se mostraba cómo era posible usar el programa AutoCAD 360, orientado a la creación de planos 2D, de manera colaborativa por parte de múltiples usuarios usando diferentes dispositivos.

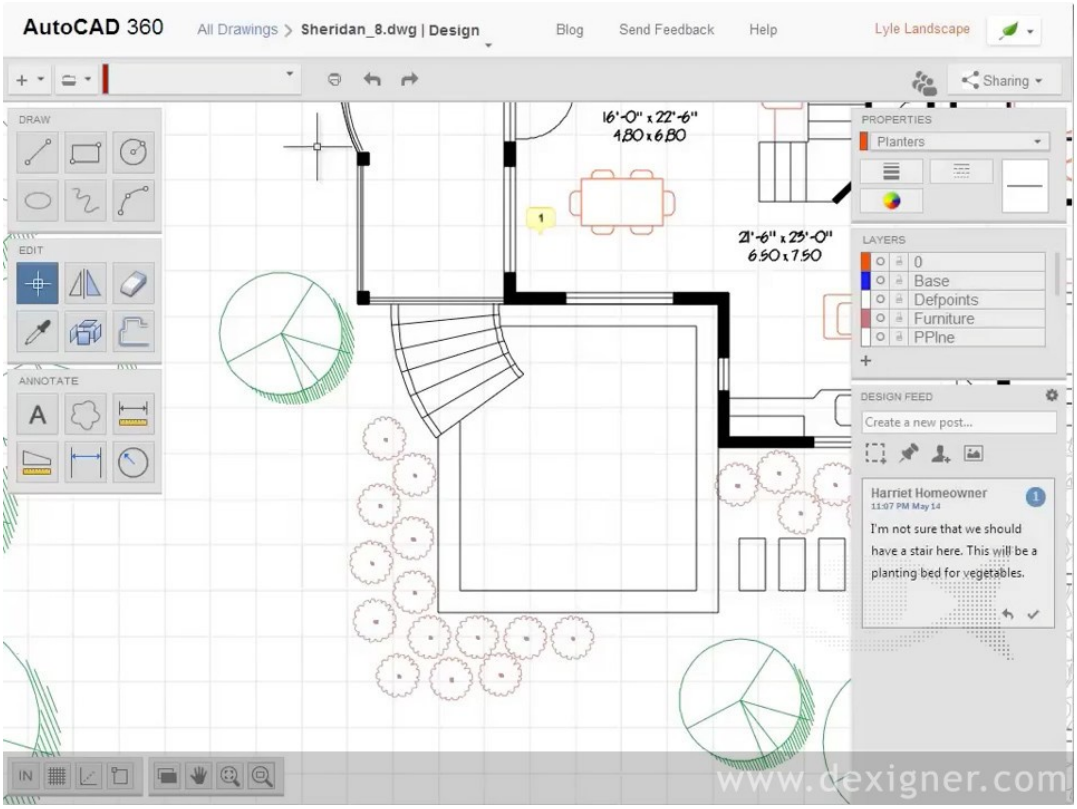


Ilustración 2.5: AutoCAD 360

AutoCAD 360 cuenta con dos versiones: una versión que al igual que Clara.io, se ejecuta en el navegador web, y otra versión para dispositivos móviles (iOS y Android). Además, se distingue entre una versión gratuita y dos versiones de pago, PRO y PRO Plus. Las versiones PRO añaden la funcionalidad de crear nuevos planos desde cero (la versión gratuita sólo permite cargarlos), así como mayor capacidad en disco y características de edición avanzadas[11]. AutoCAD 360 no es software libre, sino que cuenta con una licencia propietaria.

9 Canal de AutoCAD 360 en Youtube: https://www.youtube.com/channel/UC9UWxTY-FLnNvuvcGRapsmg

10 "Real-time DWG file collaboration - AutoCAD 360" (Vídeo en Youtube): http://www.youtube.com/watch?v=OTFOfMeRPRM

11 AutoCAD 360 Pro Mobile Plans: https://www.autocad360.com/mobileplans/

2.4. RealXtend Tundra SDK

Ilustración 2.6: Cabecera de la página http://realxtend.org/

***RealXtend Tundra SDK* es una plataforma de desarrollo de aplicaciones 3D multiusuario en red**[12]. Es software libre (licencia Apache 2.0) y como tal, se apoya en dependencias libres como Ogre3D para la parte gráfica o Qt para la interfaz.

Tundra sigue una arquitectura cliente – servidor, donde cliente y servidor parten del mismo código base pero usan diferentes configuraciones. **Además, es extensible, permitiendo añadir plugins** como indica la siguiente imagen:

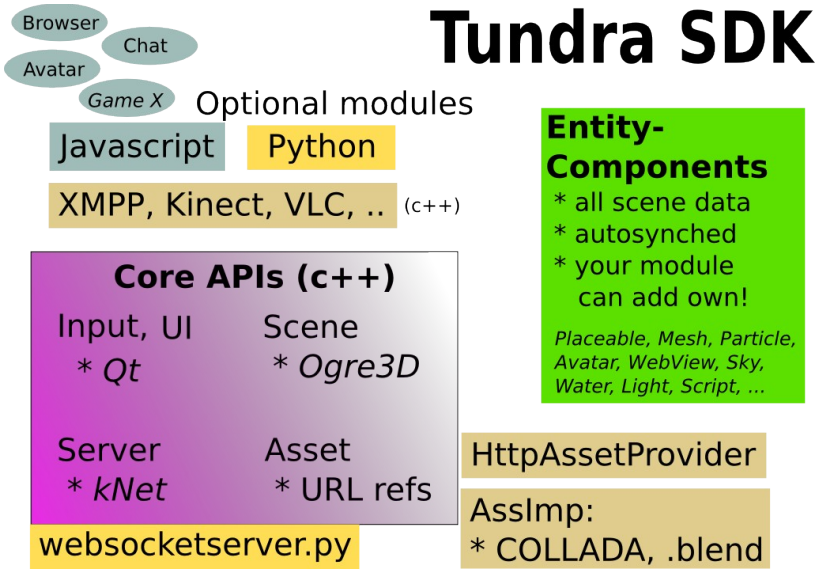


Ilustración 2.7: Estructura del SDK Tundra

Al igual que en el caso de *Verse*, el desarrollo del SDK *Tundra* no parece gozar de buena salud de cara al público. La rama principal del repositorio en Github[13] no se actualiza desde Abril de este año, y el último vídeo de su canal en Youtube[14] se subió hace tres años.

12 About | Sitio web de realXtend: http://realxtend.org/about/

13 Repositorio de Tundra en Github: https://github.com/realXtend/tundra

14 Canal de RealXTend en Youtube: https://www.youtube.com/user/realxtendvideo

3. Objetivos

3.1. Objetivos directos

El objetivo principal del proyecto consiste en desarrollar un visor / modelador 3D colaborativo. Dicho software deberá permitir que **múltiples usuarios se conecten a través de la red y trabajen en tiempo real sobre una misma escena 3D**.

Conviene recalcar desde este momento, que **este software, de nombre COMO, está orientado a la composición de escenas 3D más que a la creación de modelos individuales**. La idea es que, por ejemplo, los usuarios importen objetos “silla” o “mesa” y los sitúen para crear una oficina 3D. **Dos usuarios no podrán trabajar sobre un mismo objeto**, por ejemplo, una misma mesa, ni editar sus vértices.

Este visor / modelador 3D colaborativo, de nombre COMO, se desarrolla directamente para Ubuntu, pero con vistas a facilitar su portado a otras plataformas en el futuro (Windows, GNU/Linux y OS X). Para ello se hace uso de librerías y herramientas multiplataforma, las cuales se listarán en el apartado “requisitos hardware y software” siguiente.

Por último, y no por ello menos importante, **el software debe ser licenciado como software libre**.

3.2. Objetivos indirectos

El proyecto COMO se planteó como la excusa perfecta para **practicar** dos ramas de la programación que son de mi interés: **la programación gráfica y la programación distribuida**.

Aprovechando la naturaleza académica del proyecto, decidí optar por **usar *OpenGL* directamente**, sin apoyarme en motores gráficos de terceros. De esta manera pude **reforzar y aplicar directamente conceptos gráficos** como iluminación, shaders, comunicación con la GPU, etc. Además, **debía usarse una versión moderna de OpenGL**, evitando el uso en todo momento de funcionalidades obsoletas o desaconsejadas.

En cuanto a la programación distribuida, decidí sacrificar nuevamente facilidad de uso por didáctica. Por esta razón, en lugar de usar un protocolo ya existente, decidí **crear un modesto protocolo de sincronización de escenas 3D**.

Además de los objetivos anteriores, este proyecto también me permitió **reforzar mis conocimientos sobre el control de versiones**, al hacer un uso extensivo de Git y Github.

4. Requisitos hardware y software

En esta sección se detallan los requisitos de hardware y software que se han necesitado para desarrollar y ejecutar el modelador COMO.

4.1. Requisitos Hardware

Se requiere **un PC con una GPU que soporte la especificación OpenGL en su versión 4.2**. Además, el equipo debe contar con una **conexión a la red** donde pretenda ejecutarse el software.

4.1.1. OpenGL

Ilustración 4.1: Logo de OpenGL

OpenGL[15] es un API para escribir aplicaciones gráficas (2D y 3D), interactivas y multiplataforma. Se trata de un estándar abierto, estable y extensible, desarrollado activamente desde 1992[16].

OpenGL define un pipeline de renderizado con una secuencia de etapas que transforman las primitivas 2D o 3D especificadas por el usuario en una imagen 2D final. Salvo por unas cuantas etapas prefijadas, es responsabilidad del programador definir las etapas más relevantes del pipeline de renderizado, como lo son la etapa de procesamiento de vértices o la de procesamiento de fragmentos (píxeles candidatos a formar parte de la imagen final). Para especificar aquellas etapas que se desarrollan directamente en la GPU, OpenGL ofrece al programador la especificación del "OpenGL Shading Language" (GLSL)[17], un lenguaje de programación basado en C.

15 Sitio web de OpenGL: https://www.opengl.org/

16 OpenGL overview – Sitio web de OpenGL: https://www.opengl.org/about/

17 OpenGL Shading Language – Sitio web de OpenGL: https://www.opengl.org/documentation/glsl/

4.2. Requisitos Software

4.2.1. Sistema operativo

Ilustración 4.2: Logo de Ubuntu

COMO se desarrolla para y desde el sistema operativo Ubuntu[18], una de las distribuciones de GNU/Linux más exitosas entre los usuarios por su facilidad de uso.

4.2.2. Herramientas de desarrollo

4.2.2.1. Lenguaje de programación C++

C++ es un lenguaje de programación compilado y de propósito general. Incluye características de la programación imperativa, la programación orientada a objetos y la programación genérica.

El modelador COMO está desarrollado prácticamente en su totalidad usando C++ en su versión C++11. Esta versión incluye soporte para hilos, funciones lambda o tipos enumerados fuertemente tipados, entre otros[19].

4.2.2.2. Compilador g++ (v4.8.2)

Ilustración 4.3: Logo de GCC

Debido a su veteranía y a su filosofía libre, se optó por usar g++, un compilador de C++ basado en Unix y desarrollado como parte del proyecto GCC, una colección de compiladores libres[20].

18 Sitio web de Ubuntu: http://www.ubuntu.com/

19 Página de la Wikipedia dedicada a C++11: http://en.wikipedia.org/wiki/C%2B%2B11

20 Sitio web del proyecto *GCC*: https://gcc.gnu.org/

4.2.2.3. Depurador GDB (v7.7.1)

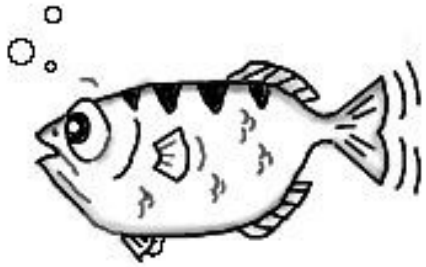


Ilustración 4.4: Mascota del depurador GDB

Para depurar el software se recurrió al depurador GDB, perteneciente al proyecto GNU[21].

4.2.2.4. Valgrind (v3.10.0)

Valgrind[22] **es un framework para construir herramientas de análisis dinámico. También incluye herramientas para depurar el uso de la memoria y herramientas de profiling**[23].

Ilustración 4.5: Logo de Valgrind

Para asistir en el depurado del proyecto COMO se ha empleado la herramienta de análisis de memoria de Valgrind.

21 Sitio web del depurador *GDB*: http://www.gnu.org/software/gdb/

22 Sitio web de *Valgrind*: http://valgrind.org/

23 Artículo sobre *Valgrind* en la *Wikipedia*: http://en.wikipedia.org/wiki/Valgrind

4.2.2.5. Entorno de desarrollo Qt Creator (v3.3.1)

Ilustración 4.6: Logo de Qt Creator

Debido a que el proyecto hace uso de Qt[24], se ha optado por **usar el entorno de desarrollo ideado específicamente para trabajar con Qt: Qt Creator[25]**. Este software cuenta con facilidades para desarrollar software multiplataforma con interfaz gráfica. Además, incluye un soporte muy útil para depurar el programa usando GDB y Valgrind desde la misma interfaz de usuario.

4.2.2.6. Control de versiones: Git (v1.9.1) + Github

Para el control de versiones del proyecto se ha empleado Git[26]. Git es un software de control de versiones distribuido, gratuito y de código abierto.

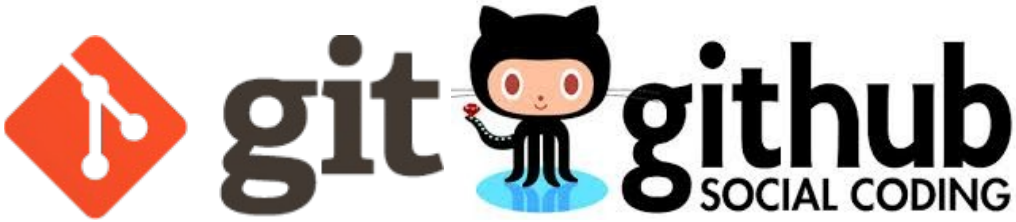


Ilustración 4.7: Logo de Git *Ilustración 4.8: Logo de Github*

El repositorio con todo el código y los archivos propios necesarios para construir y ejecutar COMO se ha alojado en Github[27]. Github es un servicio web que permite alojar repositorios Git de manera gratuita siempre que éstos sean públicos. Su eslogan "social coding" hace referencia a las facilidades que ofrece para desarrollar software de manera colaborativa: sistema integrado de seguimiento de errores, posibilidad de proponer cambios a otros desarrolladores, gestión de colaboradores, etc.[28]

---

24 Ver apartado "Librerías" a continuación.

25 Wiki sobre el entorno de desarrollo QtCreator: https://qt-project.org/wiki/Category:Tools::QtCreator

26 Sitio web de *Git*: http://git-scm.com/

27 Sitio web de *Github*: https://github.com/

28 Features – Sitio web de *Github*: https://github.com/features/

4.2.2.7. LibreOffice (v4.2.7.2)

Por último, la documentación del proyecto, incluyendo esta memoria, se ha redactado empleando la suite ofimática *LibreOffice*[29].

Ilustración 4.9: Logo de LibreOffice

29 Sitio web de *LibreOffice*: https://es.libreoffice.org

4.2.3. Librerías

4.2.3.1. Mesa 3D (v10.1.3)

Ilustración 4.10: Logo de Mesa 3D

Mesa 3D[30] es una implementación de código abierto de la especificación OpenGL, entre otros protocolos de renderizado 3D[31]. Brian Paul inició su desarrollo en Agosto de 1993 y actualmente continúa manteniendo el proyecto, al cuál contribuyen múltiples desarrolladores y organizaciones.

4.2.3.2. Qt (v5.2.1)

Ilustración 4.11: Logo de Qt

Qt[32] es un framework para desarrollar aplicaciones multiplataforma. Aunque permite crear aplicaciones de consola, **se suele usar mayormente para crear software con una interfaz gráfica de usuario (GUI) nativa al sistema operativo destino**. Para ello Qt ofrece una extensa colección de widgets: botones, campos de texto, etc.

Las interacciones de los widgets de Qt entre si mismos y con otras clases se pueden modelar mediante un potente sistema de señales y slots proporcionado por la propia Qt. Este tema se tratará más adelante, en el apartado de análisis.

Qt lleva siendo desarrollado desde 1991 y actualmente se encuentra en la versión 5.3. Está siendo desarrollado por Qt Company y "The Qt Project", y permite su uso bajo una licencia dual (propietaria y libre).

30 Sitio web de *Mesa 3D*: http://www.mesa3d.org/

31 Artículo sobre *Mesa 3D* en la *Wikipedia*: http://en.wikipedia.org/wiki/Mesa_%28computer_graphics%29

32 Artículo en la Wikipedia sobre Qt: http://en.wikipedia.org/wiki/Qt_%28software%29

Se optó por usar Qt para este proyecto porque, además de ser una librería ampliamente usada y testeada, también ofrece un módulo con múltiples facilidades para integrar OpenGL fácilmente en la aplicación a desarrollar.

4.2.3.3. Boost (v1.55.0)

Ilustración 4.12: Logo de Boost

Boost[33] es un conjunto de librerías escritas en C++ que ocupa múltiples dominios de aplicación: hilos, expresiones regulares, sistema de ficheros, etc. Debido a la calidad de las librerías Boost y a su respeto hacia los estándares, es común que algunas de sus librerías acaben formando parte del estándar de C++.

Las librerías Boost llevan en desarrollo activo desde 1998[34] y actualmente se encuentran en su versión 1.56. Su licencia (Boost license) permite el uso de Boost en proyectos privativos y libres.

El proyecto COMO hace uso de un pequeño subconjunto de las librerías de Boost en su versión 1.55.0:

- **Boost Thread**[35] para el manejo de hilos.
- **Boost Filesystem**[36] para el manejo de ficheros.
- **Boost Asio**[37] para la comunicación a través de la red y E/S siguiendo un modelo asíncrono.
- **Boost System**[38] como capa virtual para lidiar con los errores originados en el sistema operativo.

33 Sitio web de Boost: http://www.boost.org/

34 Faq del sitio web de Boost: http://www.boost.org/users/faq.html

35 Documentación de la librería *Boost Thread*: http://www.boost.org/doc/libs/1_55_0/doc/html/thread.html

36 Documentación de la librería *Boost Filesystem*: http://www.boost.org/doc/libs/1_55_0/libs/filesystem/doc/index.htm

37 Documentación la librería *Boost Asio*: http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio.html

38 Documentación de la librería *Boost System*: http://www.boost.org/doc/libs/1_55_0/libs/system/doc/index.html

4.2.3.4. GLM (v0.9.5.1)

Ilustración 4.13: Logo de GLM

GLM[39] es una librería matemática para software gráfico basado en GLSL. Lleva desarrollándose desde el año 2005 y está licenciada bajo una licencia MIT.

El proyecto COMO hace uso de GLM para, entre otras cosas, cálculo vectorial y matricial y para el cómputo de las matrices de vista y de proyección.

4.2.3.5. SDL2_image (v2.0)

Ilustración 4.14: Logo de las librerías SDL

SDL2_image[40] es una librería auxiliar basada en SDL para cargar imágenes en memoria. Acepta múltiples formatos, entre los que se incluyen BMP, GIF, JPEG, PNG, etc. Se trata de una librería desarrollada por Sam Lantinga y Mattias Engdegård y licenciada bajo la licencia GPL.

39 Sitio web de la librería *GLM*: http://glm.g-truc.net

40 Sitio web de la librería *SDL2_image*: https://www.libsdl.org/projects/SDL_image/

5. Análisis

5.1. Metodología

Debido a que se trataba de un proyecto experimental, cuyo alcance no podía definirse por completo desde el principio, **se optó por seguir un proceso de desarrollo evolutivo apoyado en prototipos**. Cada prototipo desarrollado se presentaba al tutor y se debatían los siguientes objetivos a cumplir.

5.2. Glosario de conceptos

El análisis y estudio del dominio del problema saca a la superficie algunos conceptos capitales como los siguientes:

- **Recurso**: un recurso es todo elemento de una escena 3D creado por un usuario: mallas, materiales, luces, etc.
- **Entidad**: recurso que puede manipularse directamente desde la vista 3D de la aplicación. Las mallas y las luces son recursos que a su vez son entidades, mientras que los materiales no cumplen con esta definición.
- **Malla**: entidad formada por uno o más grupos de triángulos 3D y que opcionalmente puede llevar asociados uno o más materiales.
- **Material**: estructura de datos que define las propiedades ópticas del todo o parte de una malla[41]. Opcionalmente, un material puede llevar aparejado una textura.
- **Textura**: imagen 2D que puede mapearse directamente sobre el todo o parte de una malla.
- **Primitiva**: especificación de una malla susceptible de **instanciarse**. En el contexto de COMO, conviene diferenciar entre “importar o crear una primitiva” e “instanciar una primitiva”.
    - **Importar / crear una primitiva** significa que, partiendo de un fichero con la especificación de una malla 3D (pe. Un fichero OBJ), se obtiene una especificación de primitiva. Esta especificación es un fichero con extensión .prim que define a la primitiva en un formato propio más directo de leer para la aplicación.
    - **Instanciar una primitiva** implica crear una malla a partir de de una primitiva. La especificación de la malla es una copia exacta (con sus propios materiales) de la especificación de la primitiva.
- **Luz**: malla en modo malla de alambres (wireframe)[42] que emite luz.

41 Materials and Textures – Wiki Blender 3D: Noob to Pro: http://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro/Materials_and_Textures

42 Artículo de la *Wikipedia* sobre el método de renderizado “wireframe”: http://es.wikipedia.org/wiki/Wireframe

    - **Luz direccional**: luz situada en el infinito cuya luz se emite con todos sus rayos siguiendo la misma dirección.

- **Cámara**: malla en modo malla de alambres (wireframe) cuya posición y orientación define una vista de la escena 3D.
- **Comando**: orden enviada entre un cliente y el servidor que altera el estado de una escena. Un ejemplo de comando sería “Crear cubo”.
- **Shader**: programa de dibujado que normalmente se ejecuta directamente en la GPU.

5.3. Funcionalidades del programa

Habiendo definido los conceptos más importantes del dominio de aplicación de COMO, estamos en disposición de listar las funcionalidades que ofrece el software. Estas funcionalidades se clasifican según sean responsabilidad del cliente o del servidor.

5.3.1. Software cliente

El software cliente proporciona al usuario las siguientes capacidades:

- Crear una escena 3D compartida o conectarse a una ya creada.
- Visualizar la escena a través de cuatro ventanas. Para cada ventana puede seleccionarse su vista (frontal, trasera, superior, inferior, izquierda, derecha y cámara) y su proyección (ortogonal o proyección).
- Cargar mallas y sus materiales asociados desde fichero (formato OBJ) y añadirlos a la escena.
    - Mostrar las normales a los vértices de las mallas.
- Editar los materiales asociados a las mallas, alterando los siguientes atributos:
    - Color
    - Reflectividad ambiente.
    - Reflectividad difusa.
    - Reflectividad especular.
    - Exponente especular.
- Añadir luces direccionales hasta un máximo definido por el servidor.
    - Editar el color y el coeficiente de luz ambiente de cada luz direccional creada.

- Cargar texturas desde fichero.
- Listar a todos los usuarios conectados a una escena junto con su color asociado. Las entidades seleccionadas por cada usuario se muestran resaltadas con su color.
- Generar formas geométricas simples: cubos, conos, cilindros y esferas.
- Asociar una textura diferente a cada pared de las formas geométricas simples mencionadas en el punto anterior.
- Seleccionar entidades y añadirlas a la selección del usuario.
- Aplicar y concatenar transformaciones simples sobre la selección de entidades local: translaciones, rotaciones y escalados.
    - Las transformaciones pueden realizarse de manera libre (con respecto a la vista actual) o con respecto a un eje (X, Y o Z).
    - Las rotaciones y escalados pueden realizarse con respecto a un punto de pivotaje a seleccionar entre el origen del mundo, el centroide de la selección o los centroides individuales. En este último caso, cada entidad de la selección se transforma con respecto a su propio centroide.

5.3.2. Software servidor

El software servidor proporciona al usuario las siguientes capacidades:

- Mantener un histórico con todos los comandos ejecutados sobre la escena.
- Sincronizar los comandos del histórico anterior con todos los clientes conectados a la escena.
- Gestionar la propiedad de los recursos, asociando un dueño a cada uno, y respondiendo a las peticiones de bloqueo y desbloqueo recibidas desde los distintos clientes.
- Salvar y cargar la escena.

5.4. Desafíos de la programación distribuida

La naturaleza distribuida del software COMO plantea una serie de desafíos inherentes a la interconexión de múltiples máquinas a través de la red:

5.4.1. Bloqueo de recursos

Como se comentó en el apartado de "Objetivos directos", **COMO se diseñó con una clara orientación hacia el modelado de escenas 3D más que hacia el modelado de objetos individuales**.

Con esto en mente, y con vistas a simplificar la sincronización entre los usuarios, **se requiere que COMO disponga de un mecanismo de bloqueo de recursos, de manera que dos usuarios no puedan trabajar sobre el mismo recurso al mismo tiempo**.

5.4.2. Diferentes órdenes de bytes (*Endianness*)

El "endianness"[43] define la convención seguida por un computador para organizar e interpretar los bytes de una palabra[44] en memoria. Las dos convenciones mayoritarias son Big-endian y Little-endian. En el primer caso (big-endian), el sistema ordena las palabras en memoria desde el byte más significativo al byte menos significativo, mientras que en el segundo caso (little-endian) es justo al revés. Sirva como aclaración la siguiente imagen extraída desde la *Wikipedia*[45]:

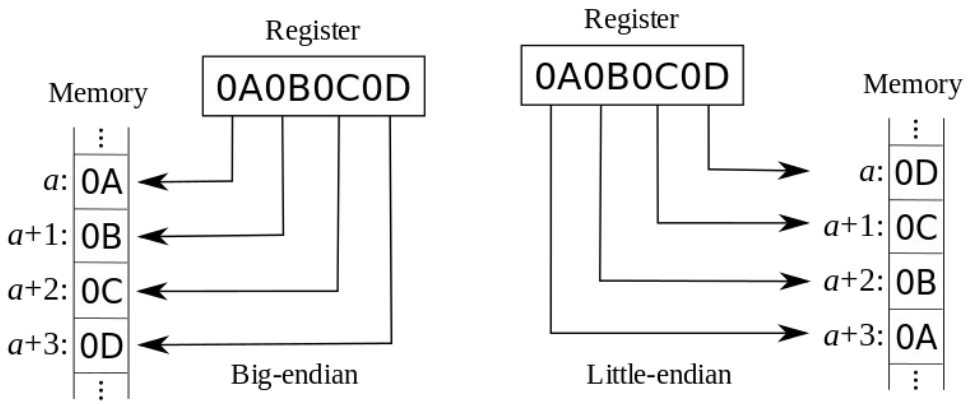


Ilustración 5.1: Comparación de las convenciones "Big-endian" y "Little-endian" para organizar palabras en memoria

Con el nacimiento de las redes de ordenadores se hizo necesario llegar a un acuerdo sobre qué convención se usaría para enviar los datos a través de la red. De esta manera, si dos o más computadores intercambiaban datos, éstos sabrían de antemano cómo interpretar los datos recibidos. **En el caso del protocolo IP se optó por usar Big-endian** como "network order" **a la**

43 Artículo de la *Wikipedia* sobre el "endianness": http://en.wikipedia.org/wiki/Endianness

44 Palabra (informática) – Wikipedia, la enciclopedia libre: http://es.wikipedia.org/wiki/Palabra_%28inform%C3%A1tica%29

45 Wikipedia, la enciclopedia libre: http://www.wikipedia.org/

hora de transmitir datos por la red[46].

Al tratarse de un proyecto puramente distribuido, **el modelador COMO debe permitir la comunicación entre computadores Big-endian y Little-endian**[47]. Esto implica que **si COMO se está ejecutando en un sistema Little-endian, los datos deben traducirse entre Big-endian y Little-endian antes de transmitirse o después de recibirse**.

5.4.3. Diferentes tamaños de los tipos de datos

Los tipos int, float, etc, pueden tener diferentes tamaños en cada máquina. Si esto no se controla, puede darse el caso de que un cliente trabaje con valores que queden fuera del rango permitido para otros clientes y que por tanto, falle la sincronización de la escena.

De lo anterior se desprende que **el software debe asegurar que todos los clientes conectados a una escena, así como el propio servidor, manejen los mismos tipos de datos**.

5.4.4. Serialización

La serialización[48] **implica traducir una estructura de datos a un formato en el que pueda** guardarse en memoria o **enviarse a través de la red**. **En el caso del proyecto COMO, se requiere que la aplicación sea capaz de serializar / deserializar las estructuras de datos susceptibles de ser sincronizadas a través de la red** (mallas, usuarios, comandos, etc) para su transmisión entre clientes.

5.4.5. Salvado y recuperado de escenas

El trabajo colaborativo sirve de poco si los resultados logrados se pierden completamente al apagarse el servidor. Por esta razón, **resulta esencial que el software tenga capacidad para salvar y recuperar escenas**.

5.5. Análisis de funcionalidades proporcionadas por terceros

En este apartado se resume una serie de mecanismos o sistemas proporcionados por librerías de terceros, cuya complejidad y utilidad merece una aclaración previa.

5.5.1. OpenGL y su pipeline de renderizado

OpenGL define un pipeline de renderizado, el cuál define la secuencia de etapas que se sigue para transformar las primitivas gráficas definidas por el usuario (puntos, líneas, triángulos, etc) **en la imagen 2D que se muestra finalmente por pantalla**[49].

46 Sección del artículo sobre “Endianness” donde se menciona el “network order”: http://en.wikipedia.org/wiki/Endianness#Endianness_in_networking

47 Por simplicidad se ha decidido omitir otras convenciones menos empleadas como el Mixed-endian.

48 Artículo en la *Wikipedia* sobre la serialización: http://en.wikipedia.org/wiki/Serialization

49 Rendering Pipeline Overview – OpenGL.org : https://www.opengl.org/wiki/Rendering_Pipeline_Overview

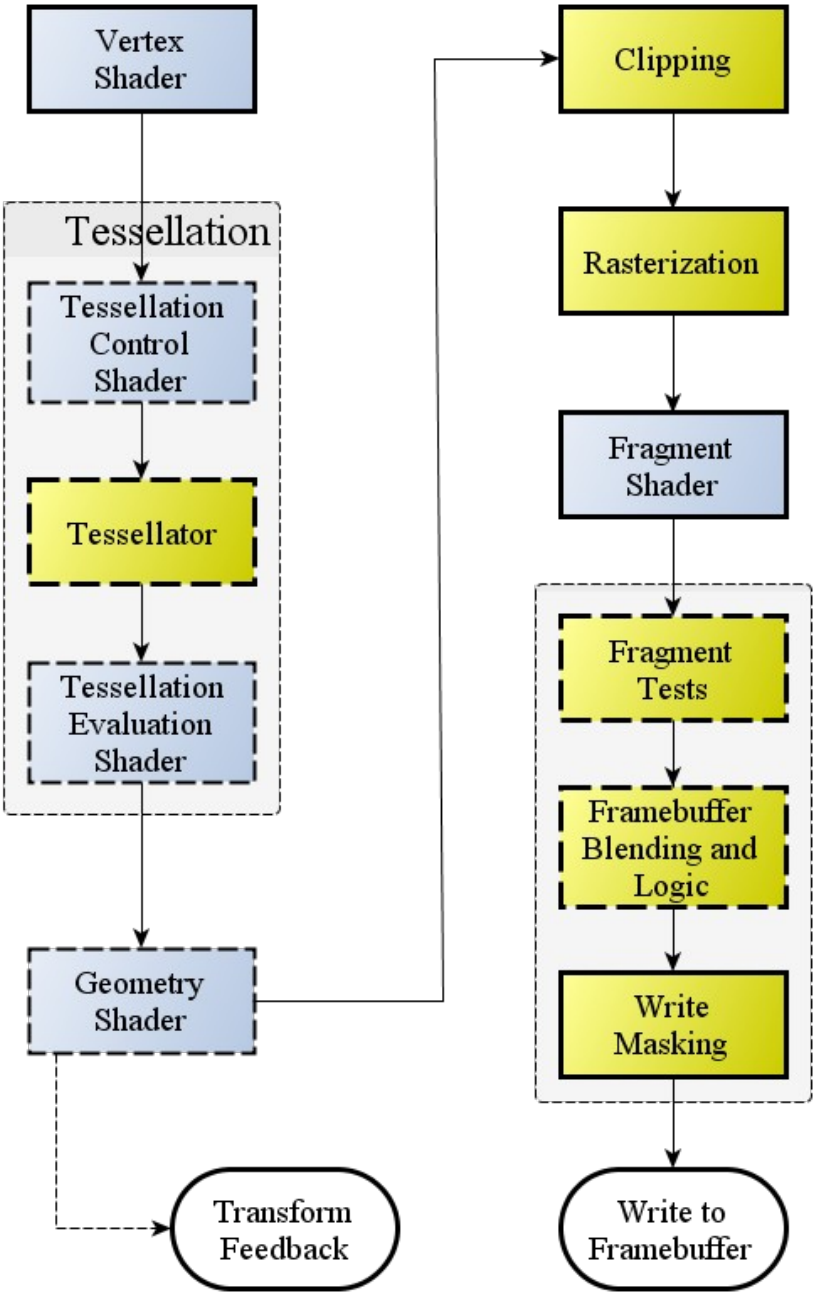


Ilustración 5.2: Pipeline de renderizado de OpenGL. Los bordes continuos indican etapas obligatorias, mientras que los discontinuos denotan etapas opcionales

En el diagrama anterior, los recuadros azules representan etapas shader programables: es responsabilidad del programador codificadas (mediante un shader) e integradas en el pipeline. Por su parte, **los recuadros amarillos indican etapas fijas del pipeline**: el programador no puede codificarlas directamente, pero puede influir en su funcionamiento alterando ciertos parámetros.

Las etapas del pipeline de OpenGL se resumen a continuación:

1. **Shader de vértices:** procesa cada uno de los vértices presentes en las primitivas a dibujar. Normalmente, el shader de vértices incluye la multiplicación de los mismos por una matriz de transformación.

2. **Teselado**: conjunto de etapas opcionales empleado para generar geometría adicional a partir de una lista de vértices de control.

3. **Shader de geometría**: etapa opcional que procesa cada primitiva gráfica (punto, línea, triángulo, etc) y genera 0 o más primitivas a partir de la misma.

4. **Clipping**[50]: etapa donde las primitivas se filtran mediante un test de visibilidad. Aquellas primitivas que no se verán en la imagen final o que se verán parcialmente son desechadas o recortadas, respectivamente.

5. **Rasterización**: traduce cada primitiva en un conjunto de píxeles para su despliegue en una pantalla.

6. **Shader de fragmentos**: procesa cada fragmento (píxel candidato a aparecer en la imagen final[51]). Comúnmente la iluminación de una escena se computa en este shader, modulando la intensidad de cada fragmento según la influencia de las lámparas en el mundo 3D.

7. **Operaciones sobre fragmentos**: conjunto de etapas obligatorias y opcionales que operan a nivel de fragmentos: test de profundidad, filtrado de fragmentos, etc.

Tal como se observa en el diagrama, el programador que hace uso de OpenGL debe definir como mínimo, un shader de vértices y otro de fragmentos.

El pipeline de renderizado no permanece inmutable durante la ejecución del programa. El programador puede usar la API de OpenGL para especificar múltiples programas shader[52], cada uno con una forma diferente de generar la imagen final a partir de las primitivas, y cambiar el programa activo en cualquier momento.

50 Clipping (computer graphics) – Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Clipping_%28computer_graphics%29

51 Los píxeles candidatos o “fragmentos” pueden desecharse de la imagen final, por ejemplo, mediante tests de profundidad: si dos fragmentos, A y B, tienen la misma posición, aquél más lejano al observador se desechará, y el otro pasará a formar parte de la imagen final.

52 Un programa shader es una combinación de etapas shader. Todo programa shader debe estar completo para poder usarse; es decir, debe incluir como mínimo un shader de vértices y otro de fragmentos.

5.5.2. Qt y su mecanismo de señales y slots

Cómo se comentó en un apartado anterior, Qt es un framework orientado a la creación de aplicaciones con interfaz gráfica de usuario (GUI), para lo cuál ofrece una extensa colección de widgets: botones, listas, campos de texto, etc.

Las interacciones de los widgets de Qt entre si mismos y con otras clases se pueden modelar mediante un potente sistema de señales y slots proporcionado por la propia Qt. Toda clase que derive de *QObject* (definida en Qt) puede beneficiarse de este sistema, definiendo sus propias señales y slots[53].

A nivel de código, una señal es la declaración de un método marcada con la palabra reserva de Qt “signal”. Una señal no incluye cuerpo debido a que sólo nos interesa su nombre y los argumentos que la acompañan. **Una señal puede emitirse a voluntad desde el código de la propia clase que la aloja**. Por ejemplo, la clase *QPushButton* de Qt, que implementa un botón simple, está implementada de forma que cuando un usuario pulsa el botón, se emite la señal QPushButton::pressed().

Por otra parte, **un slot** es un método o función marcado con la palabra reservada de Qt “slot” . A modo de ejemplo, la clase *QApplication* de Qt incluye el método QApplication::quit() para cerrar la aplicación actual.

Cualquier slot puede conectarse a una señal siempre que la declaración de ambos (señal y slot) coincidan. Esta conexión provocará que cada vez que se emita la señal se ejecute el slot. Por ejemplo, podemos conectar la señal *QPushButton::pressed*() y el slot *QApplication::quit*() para hacer que cada vez que un usuario pulse el botón anterior, la aplicación se cierre.

El proyecto COMO usa extensivamente el mecanismo de señales y slots anterior para manejar las comunicaciones entre los diferentes elementos de su interfaz gráfica, así como en algunas clases más internas.

53 Signals & slots | Documentation | Qt Project: http://qt-project.org/doc/qt-4.8/signalsandslots.html

5.5.3. Boost Asio y la E/S asíncrona

Boost Asio es una librería de red y de E/S a bajo nivel que sigue un modelo asíncrono. Esto significa que podemos lanzar operaciones de E/S en segundo plano y asociarles a cada una un manejador que se ejecutará cuando la operación termine.

Todas las operaciones de E/S asíncrona incluyendo la comunicación a través de la red, **pasan a través de un objeto de la clase io_service, la cuál actúa de interfaz con los servicios de E/S del sistema operativo. A esta clase le asociamos objetos de E/S**, como puede ser un socket TCP o un temporizador, **desde dónde lanzaremos las operaciones asíncronas**. Por ejemplo, la conexión asíncrona con un servidor remoto se iniciaría llamando al siguiente método:

socket.async_connect(<endpoint>, <manejador>);

donde <manejador> es un procedimiento o función que se ejecutará cuando la operación asíncrona se haya completado. El método *async_connect()* anterior lanzará la petición de conexión asíncrona al sistema operativo a través de io_service y retornará al llamador. La operación se ejecutará en segundo plano, y una vez finalizada, se guardarán los resultados en una cola.

Por otra parte, una llamada a io_service::run() bloqueará al hilo llamador mientras hayan operaciones asíncronas en curso. Cada vez que una operación asíncrona termine, su resultado se tomará de la cola mencionada en el párrafo anterior. Este resultado pasará al manejador asociado a la operación asíncrona, **y éste será ejecutado por el hilo que ha llamado a io_service::run()**.

En un programa multihilo, y tal como ocurren en COMO, lo común es crear un grupo de hilos o "thread pool"[54]. Cada uno de estos hilos llama a io_service::run() y va ejecutando de manera concurrente los manejadores asociados a las diferentes operaciones de E/S a medida que éstas terminan. Cada vez que un hilo termina de ejecutar un manejador, comienza a ejecutar el siguiente, o se mantiene a la espera, en caso de que no haya trabajo pendiente en ese momento.

54 Thread Pool Pattern – Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Thread_pool_pattern

6. Diseño

6.1. Aspectos generales

6.1.1. Arquitectura de red

La aplicación se apoya en una arquitectura cliente – servidor, incluso para las escenas locales.

Para la comunicación entre los clientes y el servidor se ha optado por usar TCP[55] **como protocolo de transporte**, ya que es importante que todos los datos intercambiados entre los nodos se transmitan **en orden y sin errores**.

6.1.2. Privacidad

Sólo existen escenas públicas, lo que significa que cualquier usuario puede conectarse a la escena si conoce la IP del servidor y hay espacio.

6.1.3. Sincronización

La sincronización de la escena entre los clientes y el servidor se realiza mediante el envío de paquetes con uno o más comandos. Ejemplos de comandos serían “Conectar nuevo usuario”, “Crear cubo”, “Añadir entidad a selección”, “Trasladar selección”, etc.

Se ha preferido el uso de este sistema frente al envío de datos “en peso” (mallas completas, matrices de transformación, etc) **para optimizar el uso de la red mediante el uso de paquetes pequeños**.

6.1.4. Usuarios y recursos

Una escena está formada, a grandes rasgos, **por el conjunto de usuarios o clientes conectados a la misma, así como por todos los recursos creados por los usuarios**: mallas, materiales, luces, etc. Para poder manipular y sincronizar los usuarios y recursos entre los clientes y el servidor, se requiere del uso de identificadores. En los siguientes subapartados se resume la estrategia seguida para identificar a ambos elementos de la escena.

6.1.4.1. Identificación de usuarios

Cada cliente cuenta con un entero como ID de usuario único y autoincrementado (tipo *UserID*). El ID es generado y asignado por el servidor en el momento de aceptar al nuevo cliente.

6.1.4.2. Identificación de recursos

La creación de cada recurso de la escena (mallas, luces, materiales, etc), **se inicia en el lado del cliente**. Para evitar la sobrecarga de tener que esperar a que el servidor asigne un ID único a cada recurso, se ha optado por asegurar que **cada cliente asigne uno ID único (tipo *ResourceID*) a**

55 Transmission Control Protocol – Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Transmission_Control_Protocol

los recursos creados por si mismo.

El tipo *ResourceID* consiste en un par de enteros (creatorID, resourceIndex), cuyos elementos se describen a continuación:

- **creatorID**: ID del usuario que ha creado el recurso. Este ID permite distinguir entre los recursos creados por cada usuario conectado a la escena.
- **resourceIndex**: Índice autoincrementado para distinguir los recursos creados por un mismo usuario.

La generación de IDs de recursos (*ResourceIDs*) en cliente y servidor **corre a cargo de una clase auxiliar llamada *ResourceIDsGenerator***.

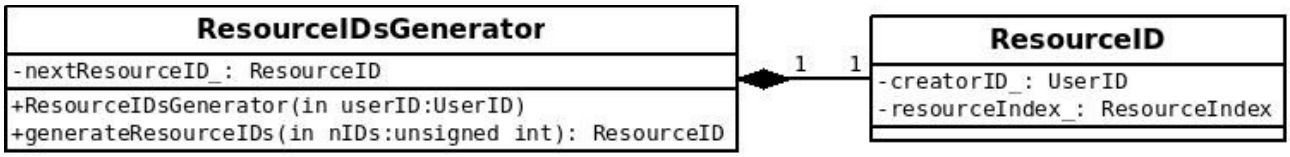


Ilustración 6.1: Clases implicadas en la generación de IDs de recurso (ResourceID)

La clase *ResourceIDsGenerator* requiere un ID de usuario (*UserID*) **para la inicialización de su atributo *nextResourceID_*, el cual se usará para generar los IDs de recurso (*ResourceID*) asociados a dicho usuario. A partir de ese momento, el usuario de la clase puede generar cualquier número de IDs de recurso** (*ResourceID*) **únicos cada vez que invoca a *ResourceIDsGenerator::generateResourceIDs*()**. Debido a que los IDs se generan de manera consecutiva, el método anterior sólo necesita devolver el primer ID del rango.

6.1.5. Bloqueo de Recursos

Cada vez que un usuario U selecciona un recurso R, este último queda bloqueado (locked) a nombre de U. A partir de ese momento, sólo el usuario U podrá trabajar sobre el recurso R hasta que decida deseleccionarlo (desbloquearlo).

6.1.5.1. Motivación

Las principales razones que motivan el bloqueo de recursos son las siguientes:

- **Simplifica la sincronización entre los diferentes clientes y el servidor, al minimizar el número de condiciones de carrera** a únicamente una: el momento de seleccionar un recurso.
- **El permitir que dos usuarios modifiquen a la vez un mismo recurso no resulta, a mi juicio, práctico**, e incluso puede llegar a resultar molesto.

6.1.5.2. Implementación

La selección y bloqueo de recursos entre el cliente y el servidor sigue el proceso indicado en el siguiente diagrama de actividad:

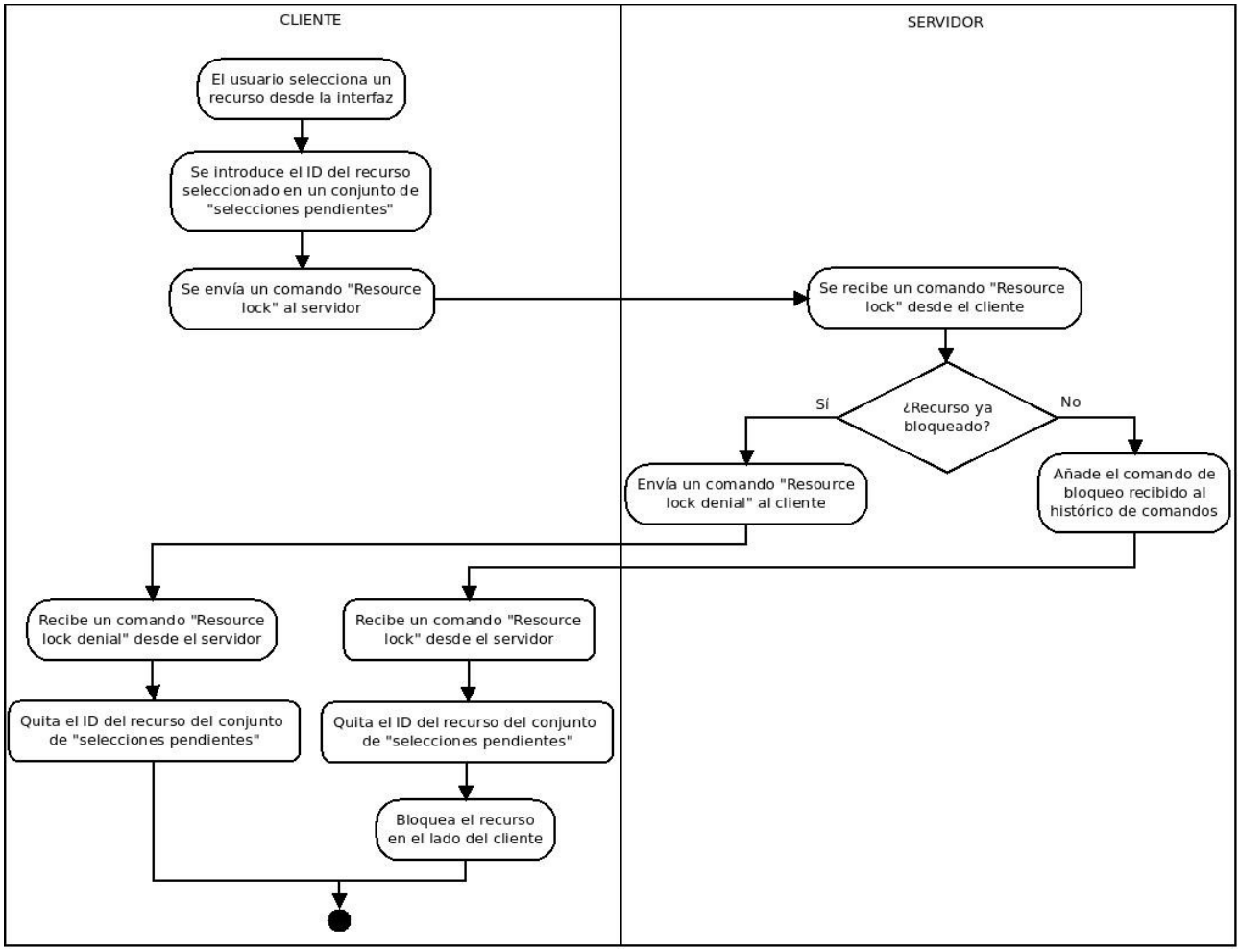


Ilustración 6.2: Procedimiento para bloquear un recurso

6.2. Comunicación cliente – servidor

6.2.1. Packables, comandos, paquetes

Como se ha comentado anteriormente, toda la comunicación entre cliente y servidor se produce a través de comandos. **Los comandos se agrupan en paquetes antes de ser empaquetados y enviados a través de la red**.

6.2.1.1. Packables

Todo objeto de datos susceptible de enviarse a través de la red debe implementar la interfaz *Packable*. Dicha interfaz implementa los métodos necesarios para empaquetar (pack) los datos desde un buffer o desempaquetar (unpack) los datos desde un buffer, siendo éste último el que se transmitirá a través de la red[56].

El método constante y sobrecargado *Packable::unpack()* se emplea en aquellos casos en los que conocemos de antemano el valor que estamos a punto de desempaquetar. Este método desempaqueta el valor desde el buffer y lo compara con el que ya tenemos guardado en nuestro objeto *Packable*. Si los valores no coinciden, se lanza una excepción.

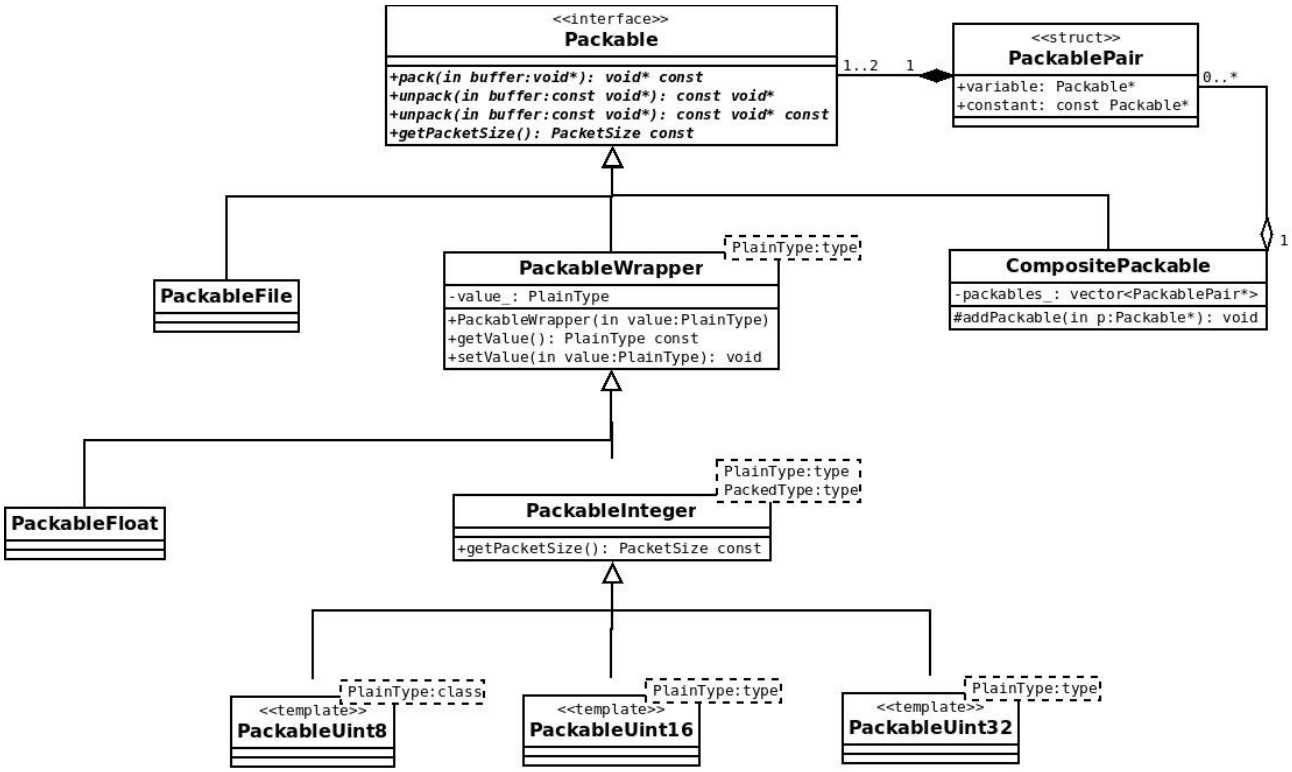


Ilustración 6.3: Jerarquía de objetos Packable

56 El empaquetamiento de tipos empaquetables (packables) equivale a la serialización de dichos tipos de datos para su transmisión a través de la red, uno de los desafíos de la programación distribuida planteados en el apartado de Análisis.

Tal como se observa en el diagrama anterior, **a partir de la interfaz *Packable* se deriva toda una jerarquía de tipos auxiliares** para poder empaquetar / desempaquetar enteros[57], número en coma flotante, ficheros, etc. En el caso de los tipos enteros, las operaciones de empaquetamiento y desempaquetamiento tienen en cuenta el "endianness" de la máquina local y convierten los datos entre éste y el "endianness" de la red en caso necesario[58].

Especial mención merece el tipo derivado *CompositePackable*, introducido para implementar el patrón de diseño *Composite*[59]. A grandes rasgos, el patrón *Composite* permite crear un tipo compuesto por tipos elementales donde ambos elementos, el todo y las partes, implementan una misma interfaz. En este caso en particular, el tipo compuesto *CompositePackable* nos permite crear tipos de datos compuestos que pueden enviarse a través de la red. Debido a que los objetos introducidos en *CompositePackable* implementan también la interfaz *Packable*, métodos como *CompositePackable::pack()* se implementan de una manera tan sencilla como la siguiente:

```
MÉTODO CompositePackable::pack( buffer : in void* )
        PARA CADA packable EN packables_ HACER
                buffer = packable.constant.pack( buffer )
        FIN PARA
        DEVOLVER buffer
FIN MÉTODO
```

Ilustración 6.4: Pseudocódigo del método CompositePackable::pack()

Como apunte final, la inclusión del tipo auxiliar *PackablePair* (ver pseudocódigo anterior) permite combinar en un mismo *CompositePackable* objetos *Packable* constantes y variables.

6.2.1.2. Comandos

Los comandos representan las órdenes enviadas entre los clientes y el servidor. Algunos ejemplos de órdenes son "conectar usuario", "cargar textura", "trasladar selección", etc.

La clase *Command* es la base de todos los comandos soportados por COMO. Al derivar de *CompositePackable*, todos los comandos se transforman en objetos de datos compuestos capaces de empaquetarse y desempaquetarse para su envío a través de la red (ver apartado anterior).

Cada objeto Command cuenta con el ID del usuario que lo ejecutó, así como con un atributo "target" de tipo *PackableCommandTarget* que define el "objetivo" del comando. El

57 En el apartado de Análisis se mencionó la problemática de conectar máquinas cuyos tipos de datos tuvieran un tamaño diferente entre las diferentes máquinas conectadas. Este problema se atajó, en el caso de los números enteros, mediante el uso de los tipos multiplataforma uint8_t, uint16_t y uint32_t de la librería estándar <cstdint> de C++.

58 Véase "Diferentes órdenes de bytes (endianness)", en la sección de Análisis de este mismo documento.

59 Composite pattern – Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Composite_pattern

objetivo de un comando es el tipo de elemento al que afecta directamente (usuario, recurso, malla, etc).

Derivando directamente de la clase *Command* se encuentra la plantilla *TypeCommand*, que se instancia para cada "target" de comando distinto. Esta clase incluye un atributo type_ que sirve para diferenciar entre todos los comandos con un mismo "target". **De esta manera, un comando dado queda identificado por su objetivo ("target") y su tipo**. Por ejemplo, un comando *UserConnectionCommand* queda identificado por los siguientes valores:

- Command::commandTarget_ = CommandTarget::USER.
- TypeCommand<CommandType>::type_ = UserCommandType::USER_CONNECTION.

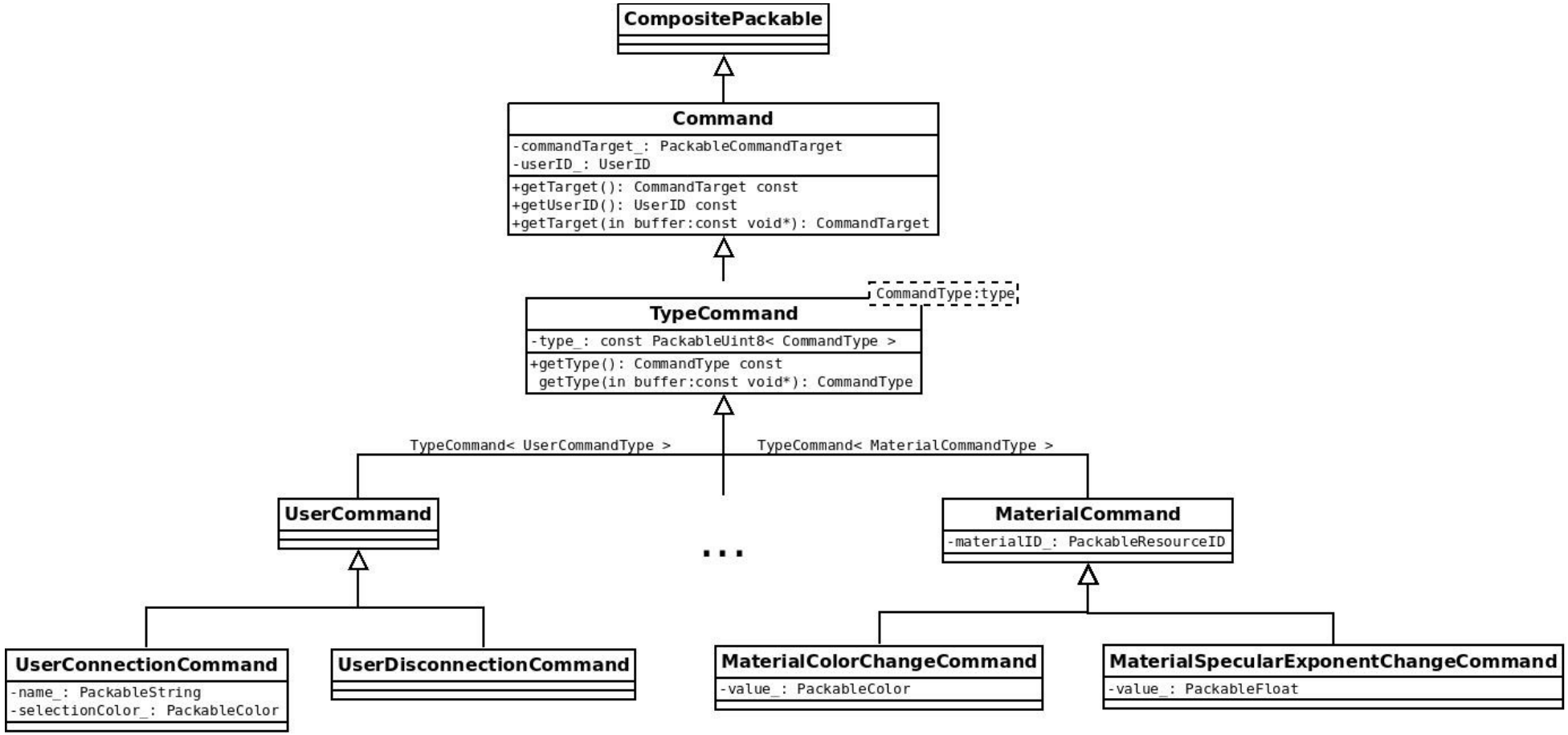


Ilustración 6.5: Jerarquía de clases Command

6.2.1.3. Paquetes

Los paquetes son los objetos de datos que se transmiten directamente a través de la red. **La clase base *Packet* cuenta con métodos para transmitir los paquetes** a través de la red **de manera síncrona** (métodos send() y recv()) **o asíncrona** (métodos asyncSend() y asyncRecv()). En este último caso se requiere que el usuario del método le suministre un *PacketHandler*, un puntero a una función encargada de procesar el paquete una vez ha sido enviado o recibido.

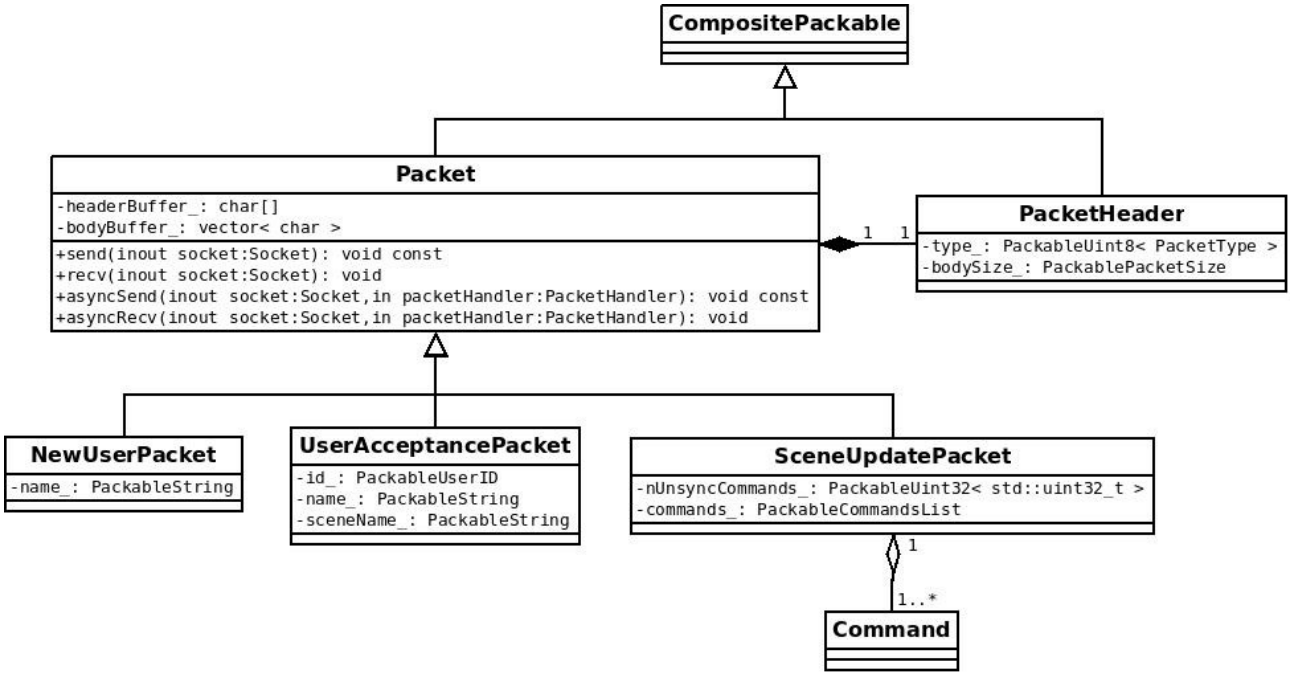


Ilustración 6.6: Jerarquía de clases Packet

En COMO existe un total de tres paquetes distintos: *NewUserPacket, UserAcceptancePacket* y *SceneUpdate*. Los dos primeros se transmiten únicamente al principio de una conexión entre un cliente y el servidor. **Una vez que la conexión queda establecida, cliente y servidor comienzan a intercambiar únicamente paquetes *SceneUpdatePacket*.**

6.2.2. Sincronización

6.2.2.1. Estableciendo la conexión

Cuando un cliente trata de conectarse a una escena, se produce la siguiente interacción cliente – servidor:

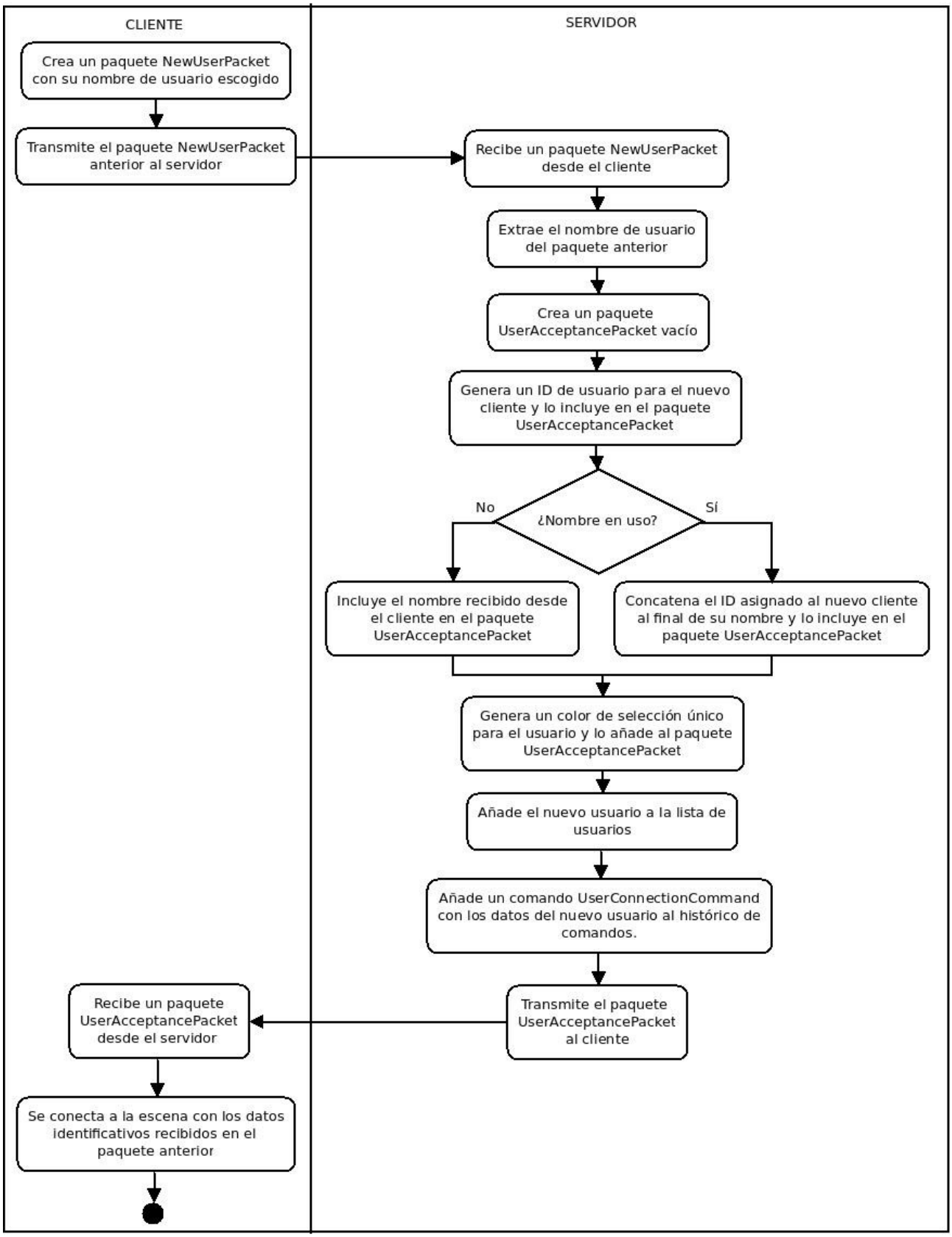


Ilustración 6.7: Procedimiento de establecimiento de la conexión entre el cliente y el servidor

6.2.2.2. Bucle de sincronización

El servidor mantiene un histórico con todos los comandos ejecutados sobre la escena compartida. **Cada vez que un cliente se conecta al servidor, éste último empieza a enviarle todos los comandos del histórico**[60] **para que el cliente actualice su escena local.** Esta sincronización se aplica también a los comandos que se vayan añadiendo al histórico mientras el cliente permanezca conectado a la escena.

Al mismo tiempo, las acciones realizadas por el cliente sobre la escena se encolan y se van transmitiendo periódicamente al servidor para su sincronización en este último.

Este bucle de sincronización bidireccional se ejecuta mientras la conexión cliente – servidor permanezca activa.

6.3. Diseño del cliente

6.3.1. Usuarios

En el cliente, los usuarios de la escena se codifican, a grandes rasgos, **como tuplas formadas por un identificador de usuario** (UserID), **un nombre y un color de selección**. **El gestor de usuarios** (clase *UsersManager*) **actúa como un contenedor para los usuarios, y también se encarga de ejecutar los comandos que recibe desde el servidor y afectan directamente a los usuarios** (pe. *UserConnectionCommand* o *UserDisconnectionCommand*).

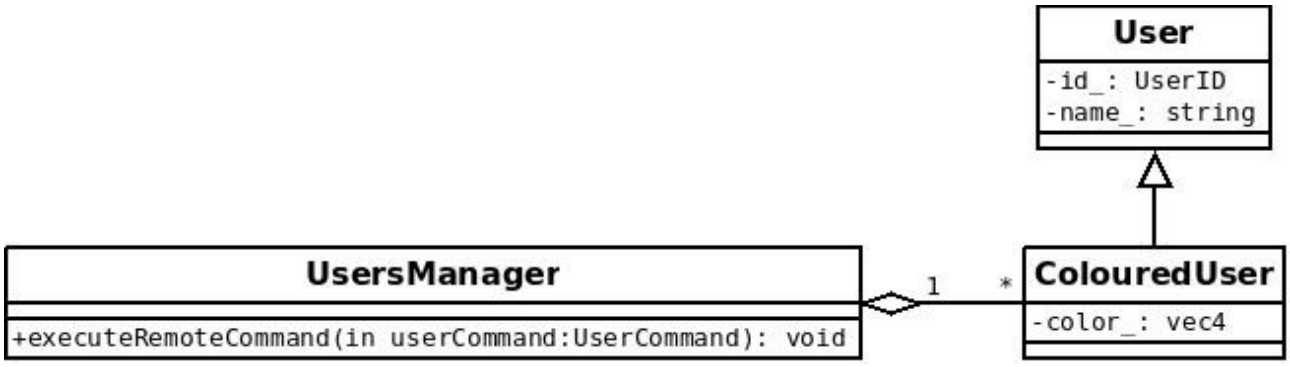


Ilustración 6.8: El gestor de usuarios (UsersManager)

6.3.2. Recursos

Como ya se comentó en el glosario de conceptos (apartado de Análisis), **los recursos son todos aquellos objetos que forman parte de la escena y que son creados por los usuarios**: mallas, materiales, luces, etc. A cada recurso se le asocia un identificador de recurso (*ResourceID*) y un nombre.

60 Los comandos del histórico que han sido enviados por un cliente son ignorados por el servidor a la hora de transmitirlos al mismo cliente.

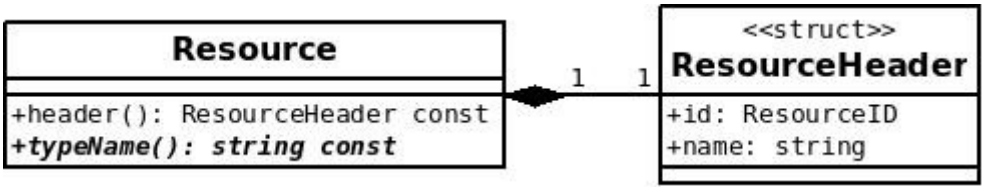


Ilustración 6.9: La clase Resource

El método abstracto *Resource::typeName()* se implementa en cada clase derivada para devolver el tipo de recurso que representa. Por ejemplo, para la clase Mesh, el método Mesh::typeName() devolvería “Mesh”. Esta string se emplea en la interfaz gráfica de usuario (GUI).

6.3.3. Entidades

Las entidades son recursos (clase *Resource*) **que pueden visualizarse y transformarse directamente desde la vista 3D de la aplicació**n.

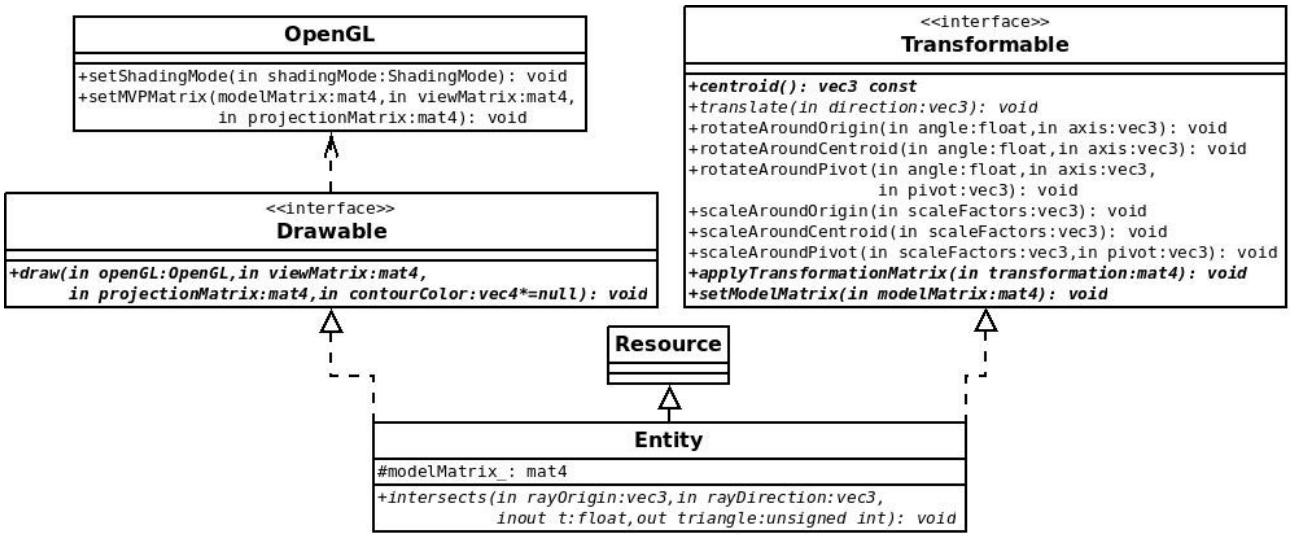


Ilustración 6.10: Diagrama de clases de las entidades en COMO

6.3.4. La clase auxiliar OpenGL y los shaders de COMO

Al implementar la interfaz *Drawable*, toda entidad (clase *Entity*) adquiere acceso a la clase *OpenGL*. Esta última es una clase que asiste en la comunicación con los shaders ofreciendo, por ejemplo, métodos para enviar la matriz MVP (modelado, vista y proyección) al shader, o para establecer el **modo de sombreado actual**.

El modo de sombreado define el flujo de control seguido por el programa shader de COMO para el renderizado de la escena. Actualmente en COMO se usan hasta cuatro modos de sombreado diferentes:

1. **SOLID_LIGHTING_AND_TEXTURING**: renderiza las primitivas incluyendo iluminación y texturizado.

2. **SOLID_LIGHTING**: dibuja las primitivas computando la iluminación en cada píxel y sin

incluir texturas.

3. **SOLID_PLAIN**: renderiza las primitivas sin iluminación ni texturizado.

4. **NORMALS**: modo especial empleado para renderizar normales a partir de los vértices proporcionados y sus normales.

6.3.4.1. El programa shader "basic"

Los tres primeros modos de sombreado presentados (SOLID_*) **usan el mismo programa shader "basic", formado por un shader de vértices y otro de fragmentos**. Cambiar de un modo SOLID* a otro únicamente cambia los valores de las variables de control *enableLighting* y *enableTexturing* presentes en el shader de fragmentos.

A continuación se muestra el flujo de control del programa shader "basic"[61]:

61 La línea discontinua que une los shaders de vértices y de fragmentos recuerda el hecho de que ambos shaders no están conectados directamente en el pipeline de renderizado de OpenGL. Véase el apartado "OpenGL y su pipeline de renderizado" en la sección de Análisis.

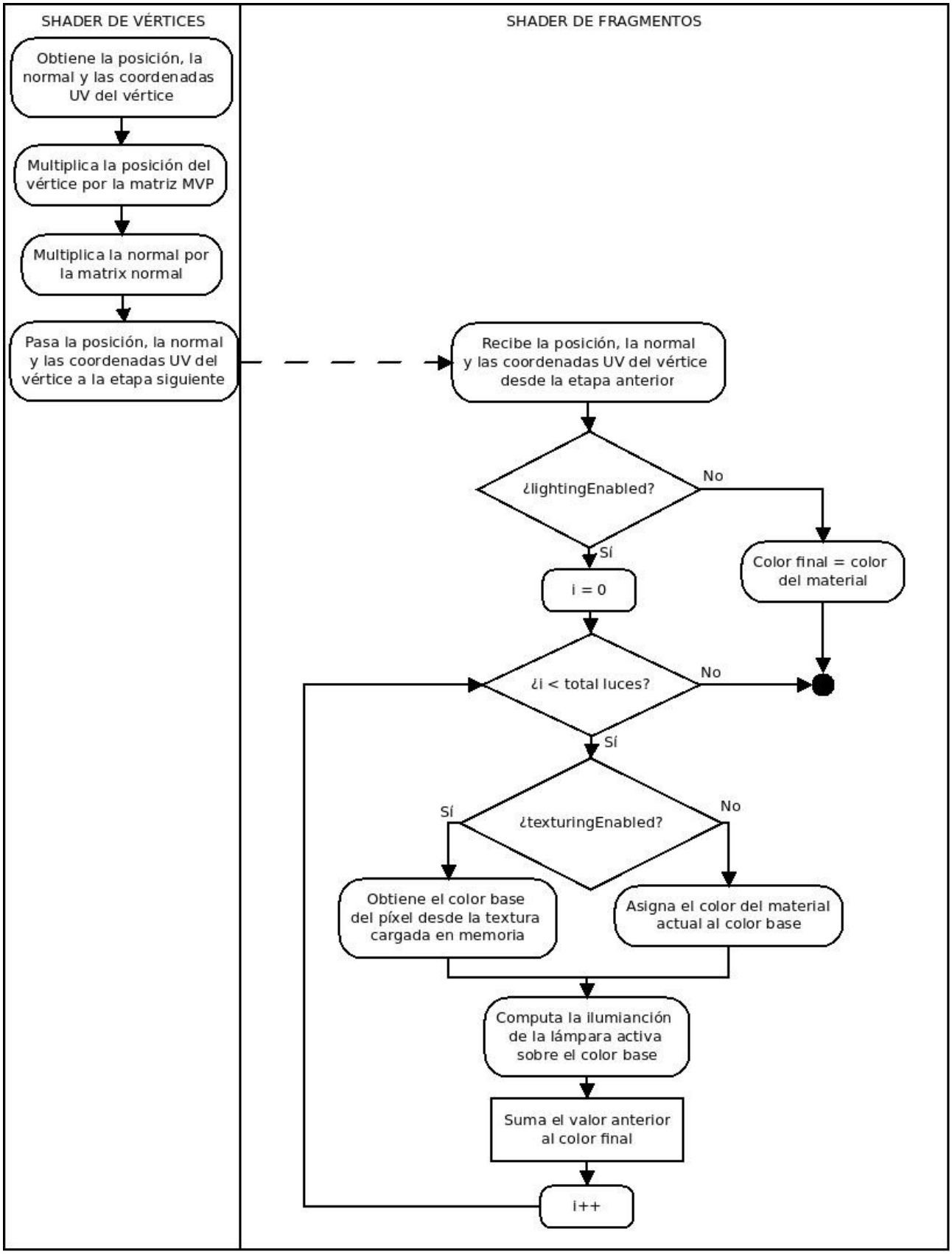


Ilustración 6.11: Flujo de control del programa shader "basic"

En el diagrama anterior, la actividad “Computa la iluminación de la lámpara activa sobre el color base” se realiza mediante la aplicación del modelo de reflexión Blinn – Phong[62].

6.3.4.2. El programa shader “normals”

Si cambiamos el modo de sombreado a NORMALS, el programa shader “basic” anterior se sustituye por un programa shader llamado “normals”. Este programa se dedica a renderizar normales a partir de pares (vértice, normal), y se compone de un shader de vértices, **otro de geometrías**[63] y un tercero de fragmentos.

El flujo de control del programa shader “normals” se presenta a continuación:

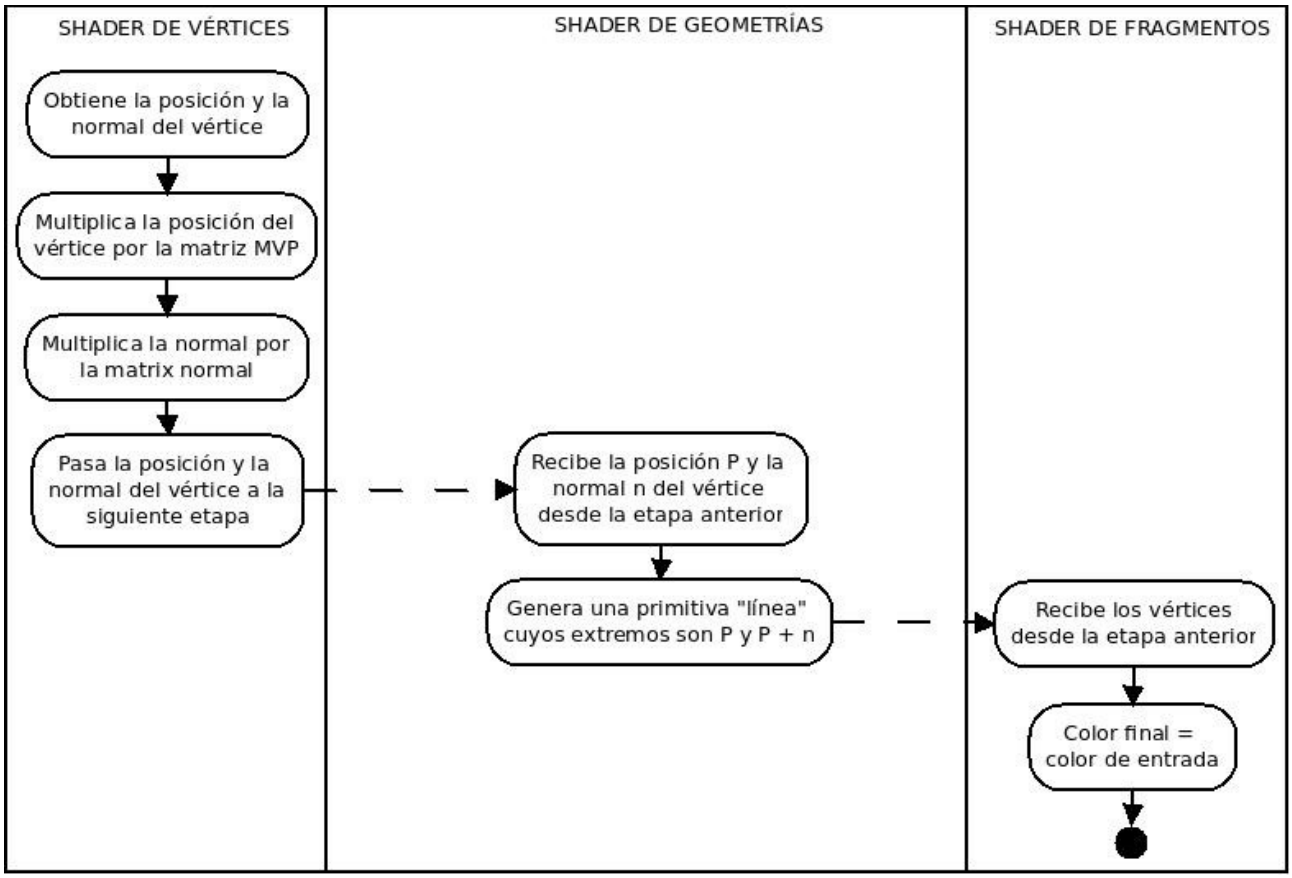


Ilustración 6.12: Flujo de control del programa shader “normals”

62 Blinn – Phong shading model – Wikipedia, the free encyclopedia : http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model

63 El shader de geometría es una etapa opcional y programable del pipeline de renderizado de OpenGL. Esta etapa recibe como entrada una primitiva gráfica, y puede generar 0 o más primitivas a partir de ésta. En el caso del shader geometrías del programa “normals”, se toma una primitiva “punto” y se genera una primitiva “línea”.

6.3.5. Mallas

Las mallas son entidades formadas por uno o más conjunto de triángulos (vértices + aristas). La clase base *Mesh* contiene los identificadores de los objetos de OpenGL necesarios para renderizar la malla:

- **VBO (Vertex Buffer Object)**: Buffer de memoria donde se guardan los atributos de cada vértice (posición, normal y coordenadas UV asociadas).
- **VAO (Vertex Attribute Object)**: Objeto que define la manera en la que se organizan los atributos de cada vértice dentro del VBO.
- **EBO (Elements Buffer Object)**: Buffer de memoria que contiene los índices de los vértices a renderizar. Resulta especialmente útil cuando los vértices de un objeto suelen ser compartidos por múltiples triángulos.

Los objetos anteriores se almacenan directamente en la GPU para un renderizado más eficiente. **Sin embargo, también se requiere que la CPU pueda acceder a dichos datos** para computar cuándo el usuario hace clic sobre una malla dada. Para evitar la sobrecarga de transferir constantemente los triángulos de una malla desde la GPU a la CPU, **los triángulos de una malla se duplican en la estructura de datos *MeshVertexData***. Este sacrificio de memoria permite un acceso a los vértices más rápido para la CPU y la GPU.

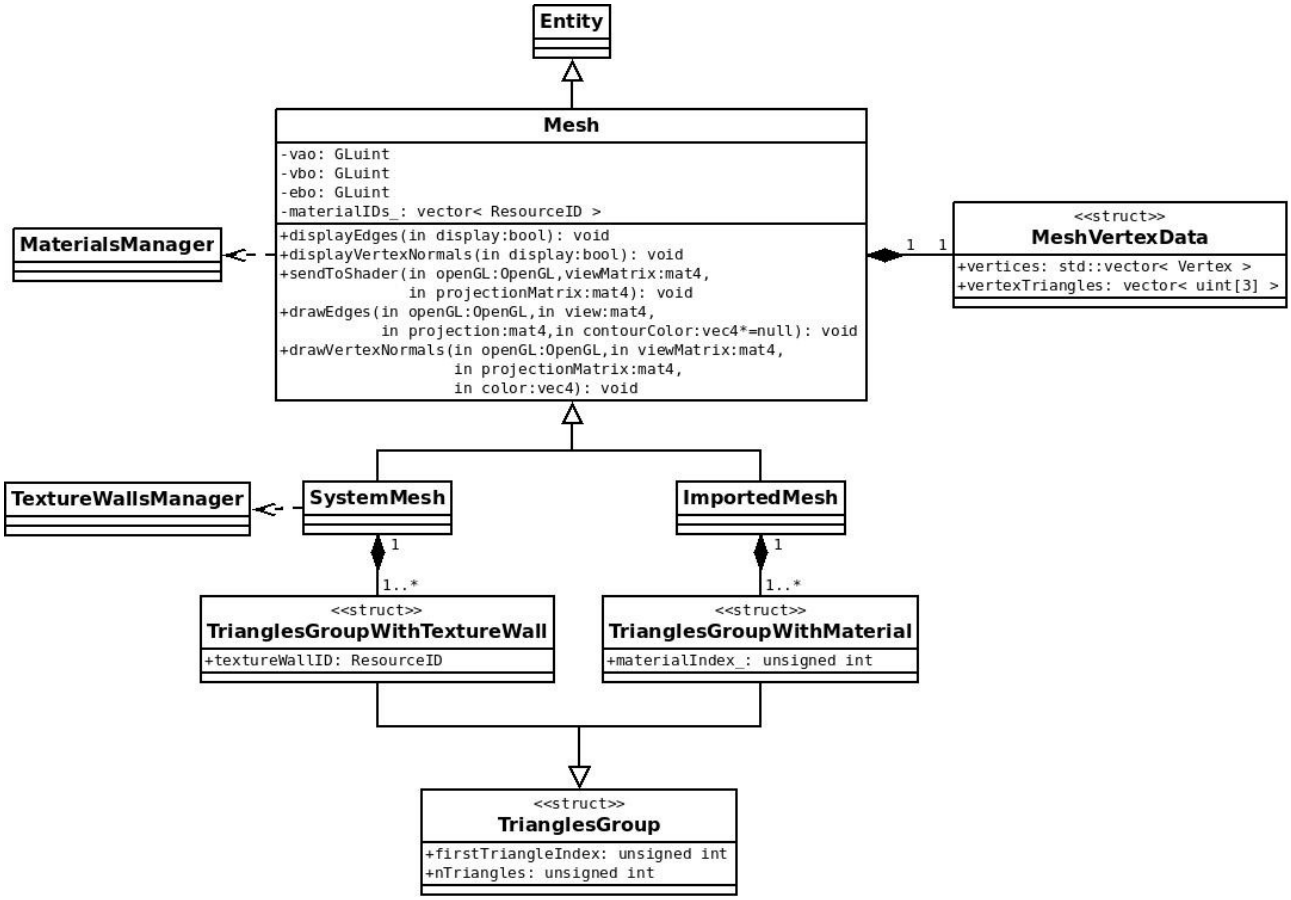


Ilustración 6.13: Jerarquía de clases Mesh

Las mallas se dividen en mallas del sistema (*SystemMesh*) **y mallas importadas** (ImportedMesh).

6.3.5.1. Mallas del sistema. Texture walls.

Las mallas del sistema son aquellas generadas por el propio software a partir de parámetros definidos por el usuario. Dos ejemplos de mallas del sistema son los cubos y las esferas. En el primer caso, el sistema genera un cubo a partir de un tamaño de lado definido por el usuario. En el segundo caso, una esfera es generada a partir del número de divisiones que especifica el usuario.

Las mallas del sistema constan de una serie de "paredes" o caras: superficies continuas y suaves. Por ejemplo, un cubo contiene 6 caras (cada uno de los planos que lo encierran), una esfera contiene una única pared envolviendo toda su superficie, etc. **COMO permite a los usuarios aplicar una textura diferente a cada pared de una primitiva del sistema (texture wall)**, así como definir el desplazamiento (offset) y la escala de la textura sobre la pared.

6.3.5.2. Mallas importadas

Por su parte, las mallas importadas son aquellas cuya especificación se leyó (importó) desde un fichero de especificación de primitiva. **COMO usa su propio formato de fichero *.prim**, en el cuál se incluye todos los vértices, normales, coordenadas UV y materiales de una malla.

6.3.6. Luces y cámaras

Las luces y cámaras en COMO son mallas con propiedades especiales: las luces son mallas que emiten luz, mientras que las cámaras son mallas capaces de definir una vista de la escena. Ambas entidades derivan de *ImportedMesh* debido a que la especificación de sus respectivas mallas se importan desde ficheros de primitiva (.prim) que se distribuyen junto con COMO.

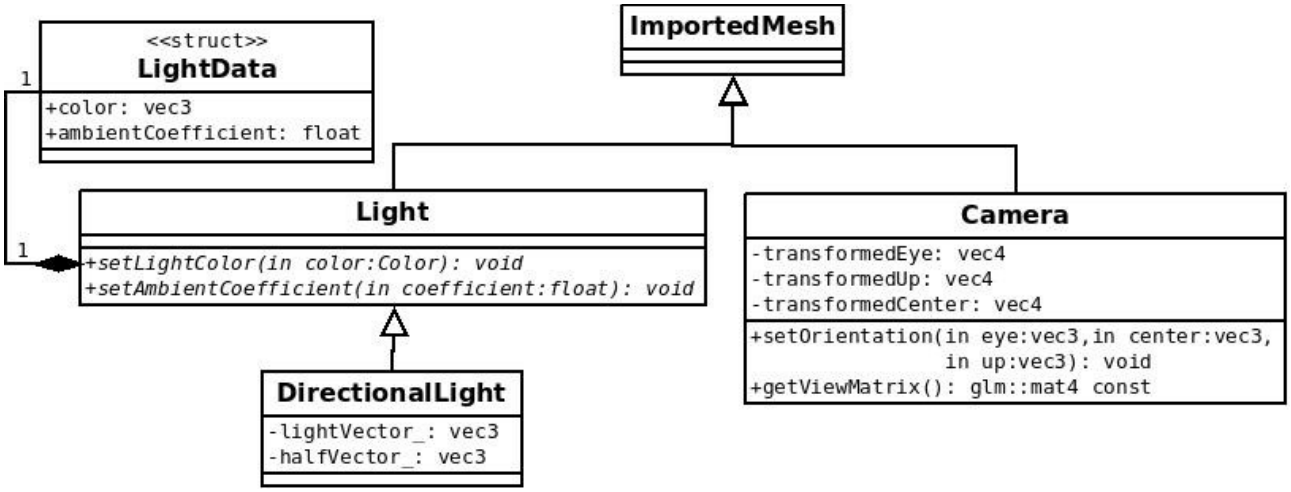


Ilustración 6.14: Las clases Light y Camera en contexto.

6.3.7. Sincronizando la escena. Clases *Scene*, *ServerInterface* y *ServerWriter*

La sincronización de una escena remota en el cliente se produce mayormente en las clases *Scene, ServerInterface* y *ServerWriter*.

La clase *Scene* mantiene un gestor de recursos (manager) por cada tipo de recurso disponible en la escena. Existe un gestor de entidades, un gestor de materiales, un gestor de texturas, etc.

Por su parte, **la clase *ServerInterface* es la interfaz a través de la cuál se realizan todas las comunicaciones con el servidor**. **Esta clase se apoya en un conjunto de hilos que son los que mantienen la conexión con el servidor, realizando los envíos y escuchando las recepciones**[64].

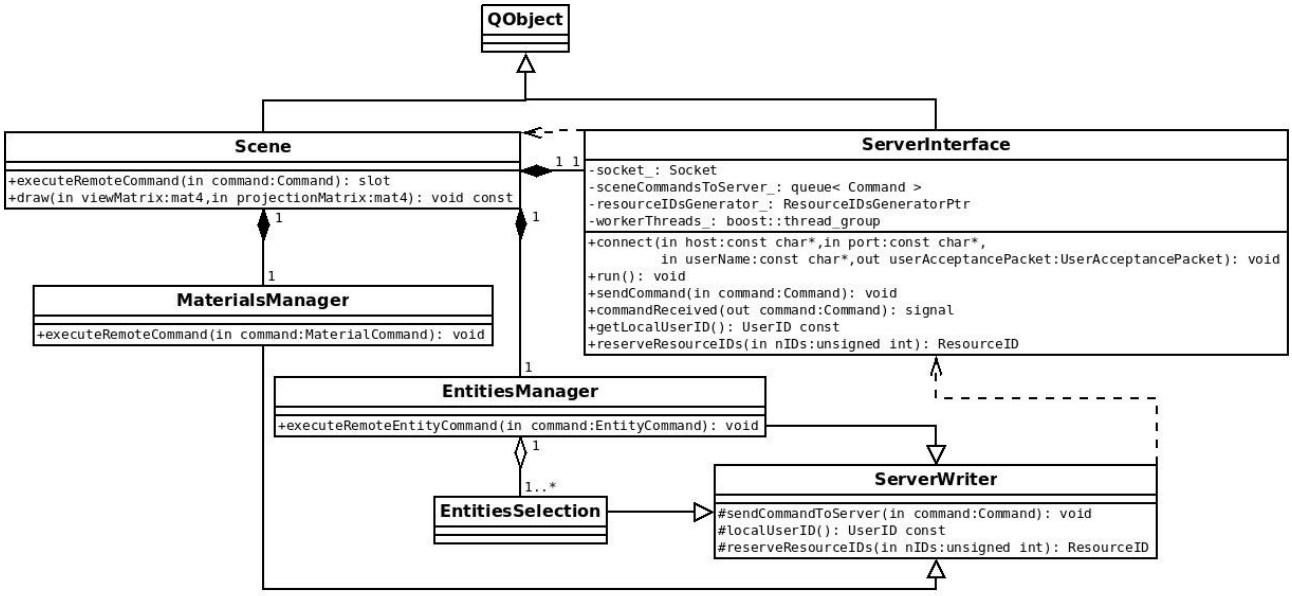


Ilustración 6.15: Subconjunto con las clases más relevantes para la sincronización de una escena con el servidor

La recepción de comandos desde el servidor y su posterior ejecución o sincronización se apoya en el sistema de señales y slots de Qt, de ahí que *Scene* y *ServerInterface* deriven de *QObject*. Cuando se recibe un comando, la clase *ServerInterface* emite una señal ServerInterface::commandReceived(Command) con el mismo comando. La señal anterior está conectada al slot *Scene::executeRemoteCommand(Command)*, el cual identifica el comando recibido y lo redije al manejador del manager asociado. Por ejemplo, si se recibe un comando que afecta a una textura, se invoca a TexturesManager::executeRemoteCommand().

El envío de comandos se basa en la clase *ServerWriter*, de la que derivan todos los objetos capaces de transmitir comandos al servidor. Todo objeto *ServerWriter* mantiene una referencia a *ServerInterface*, y el método *ServerWriter::sendCommand*() invoca a *ServerInterface::sendCommnad*() sobre dicha referencia.

64 La clase ServerInterface hace uso de las operaciones de E/S síncronas y asíncronas proporcionadas por la librería Boost Asio. Véase el subapartado “Boos Asio y la E/S asíncrona” en el apartado de Análisis.

Conviene aclarar que el método *ServerInterface::sendCommand*() no envía directamente los comandos al servidor. **En su lugar, introduce los comandos suministrados en una cola que actúa como bandeja de salida.** De manera concurrente, el servidor usa un temporizador (timer) para, de manera periódica, comprobar si hay comandos pendientes de enviar y enviarlos en caso afirmativo.

6.3.8. Gestores de recursos

Una escena 3D en COMO contiene un gestor diferente para cada tipo de recurso disponible: un gestor de materiales, un gestor de mallas, etc. **Cada gestor actúa como un contenedor para todos los recursos del mismo tipo que existen en la escena, creándolos o eliminándolos bajo demanda**. **Además, los gestores son los encargados de asociar a cada recurso su dueño actual.** Las órdenes de creación, bloqueo, desbloqueo y eliminación de recursos se obtienen por dos vías: mediante una llamada directa a un método de la clase para el caso del usuario local, o mediante la ejecución del comando recibido desde el servidor para el caso de usuarios remotos.

6.3.8.1. Gestores de edición múltiple. La clase ResourcesManager

Los gestores de edición múltiple son aquellos que permiten editar más de un recurso al mismo tiempo. Un ejemplo sería el gestor de mallas (*MeshesManager*), ya que un usuario puede seleccionar múltiples mallas y transformarlas conjuntamente.

Todos los gestores de edición múltiple derivan, directa o indirectamente, de la clase *ResourcesManager*. En la siguiente figura se muestra esta clase junto con la jerarquía de clases e interfaces que la sustenta:

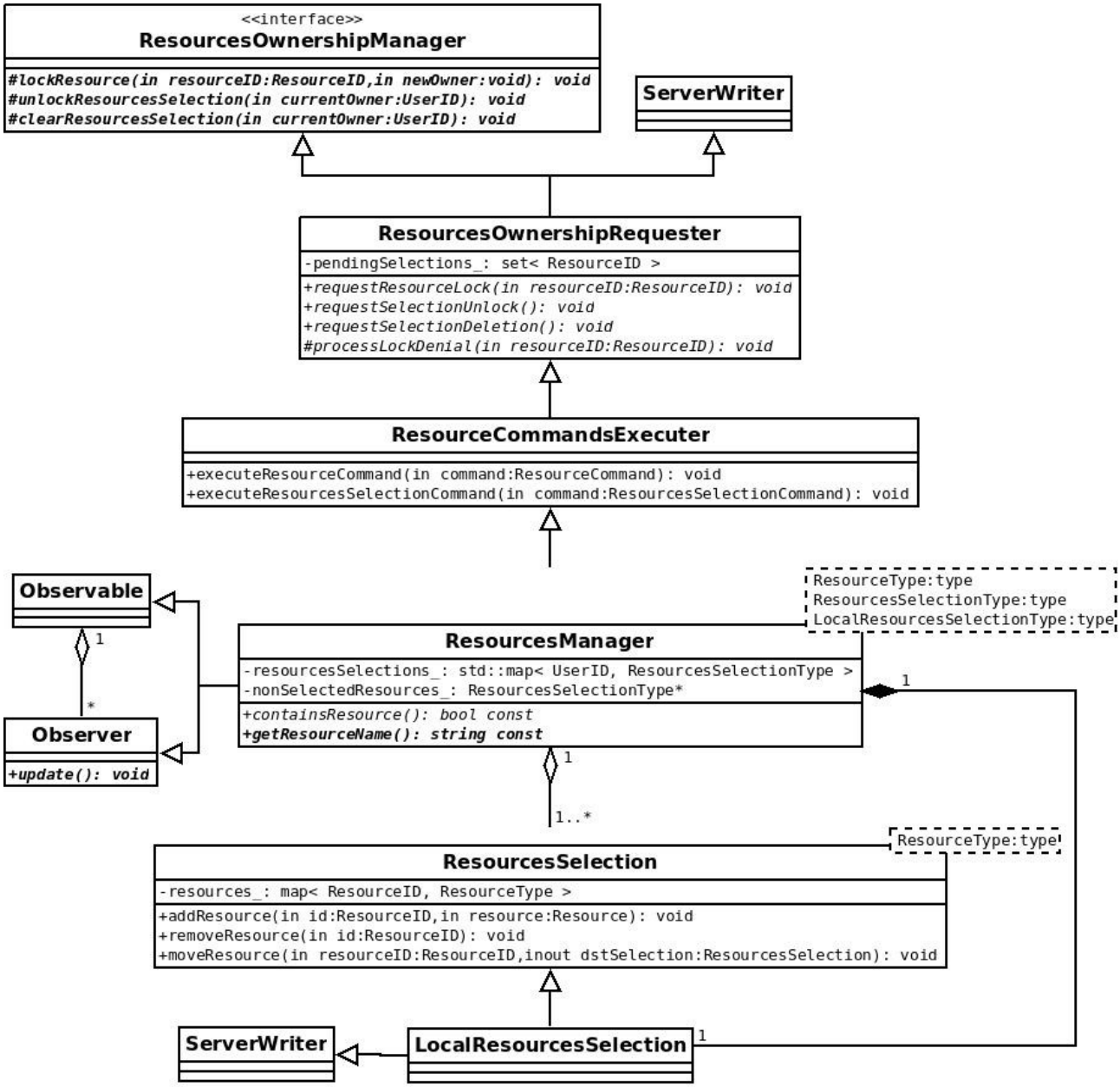


Ilustración 6.16: Jerarquía de clases que sustentan a ResourcesManager

Como se observa en la figura anterior, *ResourcesManager* deriva de una jerarquía de clases e interfaces que le confieren todas sus funcionalidades. A continuación se presenta un resumen de cada clase e interfaz de la mencionada jerarquía[65]:

- **La interfaz *ResourcesOwnershipManager*** especifica los métodos necesarios para bloquear recursos individualmente, así como para desbloquear y eliminar la selección de recursos de un usuario dado. **Esta clase es común al cliente y al servidor**, por cuanto ambos necesitan manejar recursos y sus dueños.

65 Salvo la clase ServerWriter, la cuál ya se ha descrito en un apartado anterior.

- **La clase *ResourcesOwnershipRequester*** permite enviar peticiones de bloqueo, desbloqueo y eliminación de recursos al servidor. También se encarga de procesar las denegaciones de bloqueo recibidas desde el servidor.

- **La clase *ResourcesCommandsExecuter*** se encarga de procesar los comandos recibidos que se aplican a todo recurso, sin importar el tipo de recurso que sea. Aquí entrarían los comandos de bloqueo, desbloqueo y eliminación de recursos.

La clase *ResourcesManager* contiene una selección de recursos (clase *ResourcesSelection*) **asociada a cada usuario, así como una selección de recursos sin seleccionar**. Selecciones de recursos especializados derivarán de *ResourcesSelection* e implementarán métodos para transformar la selección completa (pe. *MeshesSelection* y su método *MeshesSelection::translate*()).

Como puede observarse en el diagrama anterior, ***ResourcesManager* se beneficia del patrón de diseño *Observer***[66]. En este patrón de diseño, un objeto de tipo “Observable” mantiene una lista de “observadores” (Observer). Cada vez que se produce un cambio en el objeto de tipo “Observable”, se notifica a sus “observadores” llamando al método Observer::update() sobre cada uno. En este caso en concreto, **un gestor de recursos (*ResourcesManager*) es a la vez observable y observador**, permitiendo así que los diferentes gestores puedan observarse entre ellos.

6.3.8.2. El gestor de entidades

El gestor de entidades (*EntitiesManager*) **es un gestor de edición múltiple que gestiona las entidades disponibles en COMO**: recursos que pueden visualizarse y manipularse directamente desde la vista 3D. Como existen múltiples tipos de entidades (mallas, luces y cámaras), el gestor de entidades contiene a su vez un gestor diferente por cada tipo de entidad distinta: *MeshesManager*, *LightsManager* y *CamerasManager*[67].

66 Observer pattern – Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Observer_pattern

67 Cada gestor de edición múltiple contiene una serie de selecciones especializadas derivadas de *ResourcesManager*: *LightsManager* contiene un *LightSelection* por cada usuario, *CamerasManager* contiene un *CamerasSelection* por cada usuario, etc. Véase el apartado “Gestores de edición múltiple. La clase ResourcesManager” anterior.

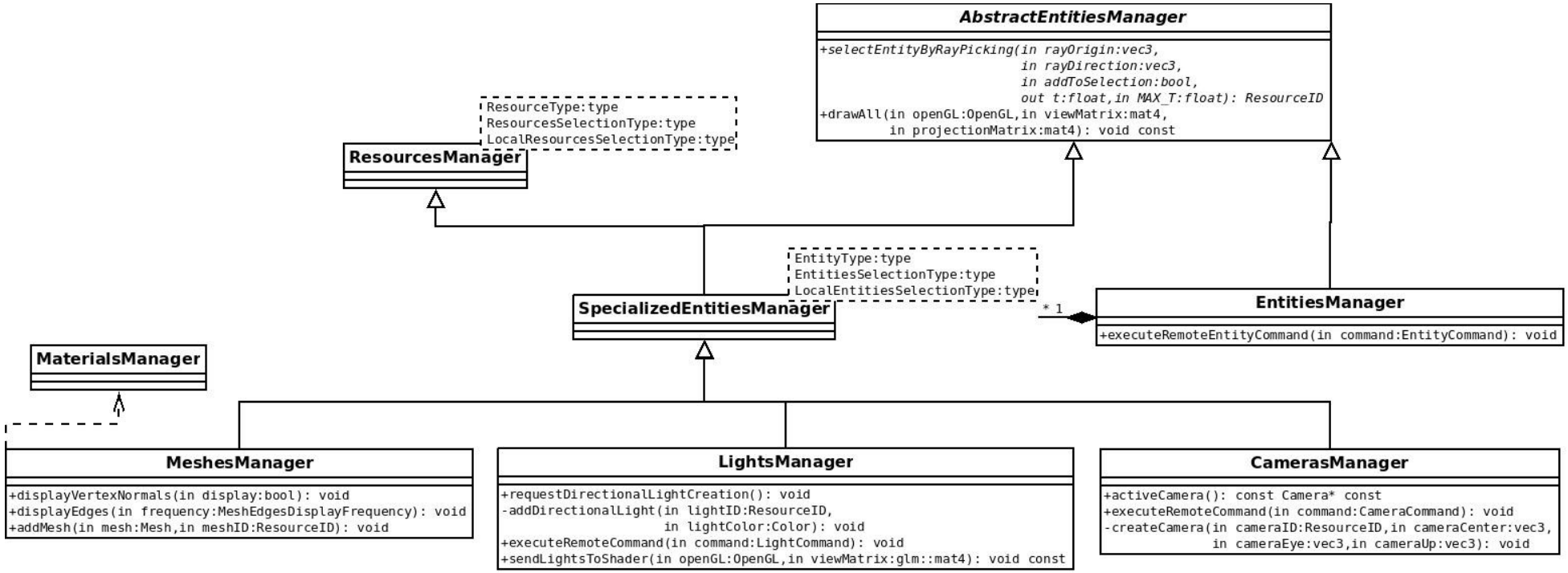


Ilustración 6.17: El gestor de entidades en contexto

En los siguientes apartados se describe someramente cada gestor de entidades especializado.

6.3.8.2.1. El gestor de mallas (*MeshesManager*)

El gestor de mallas contiene todas las mallas 3D de una escena. **Por lo general, las mallas se crean fuera de esta clase**, mediante importadores o factorías externas (*PrimitivesImporter*, *CubesFactory*, etc) y se insertan aquí mediante los métodos oportunos.

El gestor de mallas incluye métodos para establecer si se desea mostrar o no las normales de los vértices (todas las mallas), **o las aristas de las mallas** (sólo mallas no seleccionadas).

6.3.8.2.2. El gestor de luces (*LightsManager*)

COMO sólo permite la creación de luces direccionales hasta alcanzar un máximo definido por el servidor. Por esta razón, el gestor de luces, a diferencia de otros gestores, no crea sus recursos directamente y luego manda la orden al servidor. Por el contrario, el usuario local debe pedir permiso al servidor para crear la luz mediante una llamada a *LightsManager::requestDirectionalLightCreation*(). Cuando se reciba una respuesta del servidor, el método *LightsManager::executeRemoteCommand*() la procesará y creará la luz en caso afirmativo.

El método *LightsManager::sendLightsToShader*() se invoca al principio de cada iteración de renderizado de la escena. Este método **envía los datos de todas las luces creadas al shader de fragmentos**, donde se computa la iluminación de la escena siguiendo el modelo clásico o modelo de reflexión Blinn–Phong[68].

6.3.8.2.3. El gestor de cámaras (*CamerasManager*)

En la versión actual de COMO, sólo se permite la creación de una única cámara asociada a la escena, cuya creación es iniciada desde el servidor. Por esta razón, el único método de la clase para crear una cámara es privado, y se invoca desde *CamerasManager::executeRemoteCommand*().

6.3.8.3. Gestores de edición individual

Los gestores de edición individual son aquellos en los que cada usuario sólo puede editar un recurso al mismo tiempo. Salvo excepciones, cuando el usuario selecciona un recurso en un gestor de este tipo, se le devuelve un manejador (handler) derivado de *ServerWriter*, que le permitirá modificar el recurso en cuestión y sincronizar los cambios automáticamente en el servidor (pe. MaterialHandler).

COMO incluye los siguientes gestores de selección individual: *MaterialsManager*, *TexturesManager* y *TextureWallsManager*.

68 Blinn – Phong shading model – Wikipedia, the free encyclopedia : http://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_shading_model

6.3.8.3.1. El gestor de materiales (MaterialsManager)

Todo material en COMO se asocia desde su creación a una única malla y se elimina automáticamente cuando la malla es eliminada. **Debido a esto, cuando una malla es seleccionada, todos sus materiales quedan bloqueados a nombre del usuario que seleccionó la malla**. A partir de ese momento, el usuario puede seleccionar para su edición cualquier material asociado a cualquiera de las mallas que tiene seleccionadas actualmente, sin esperar confirmación por parte del servidor.

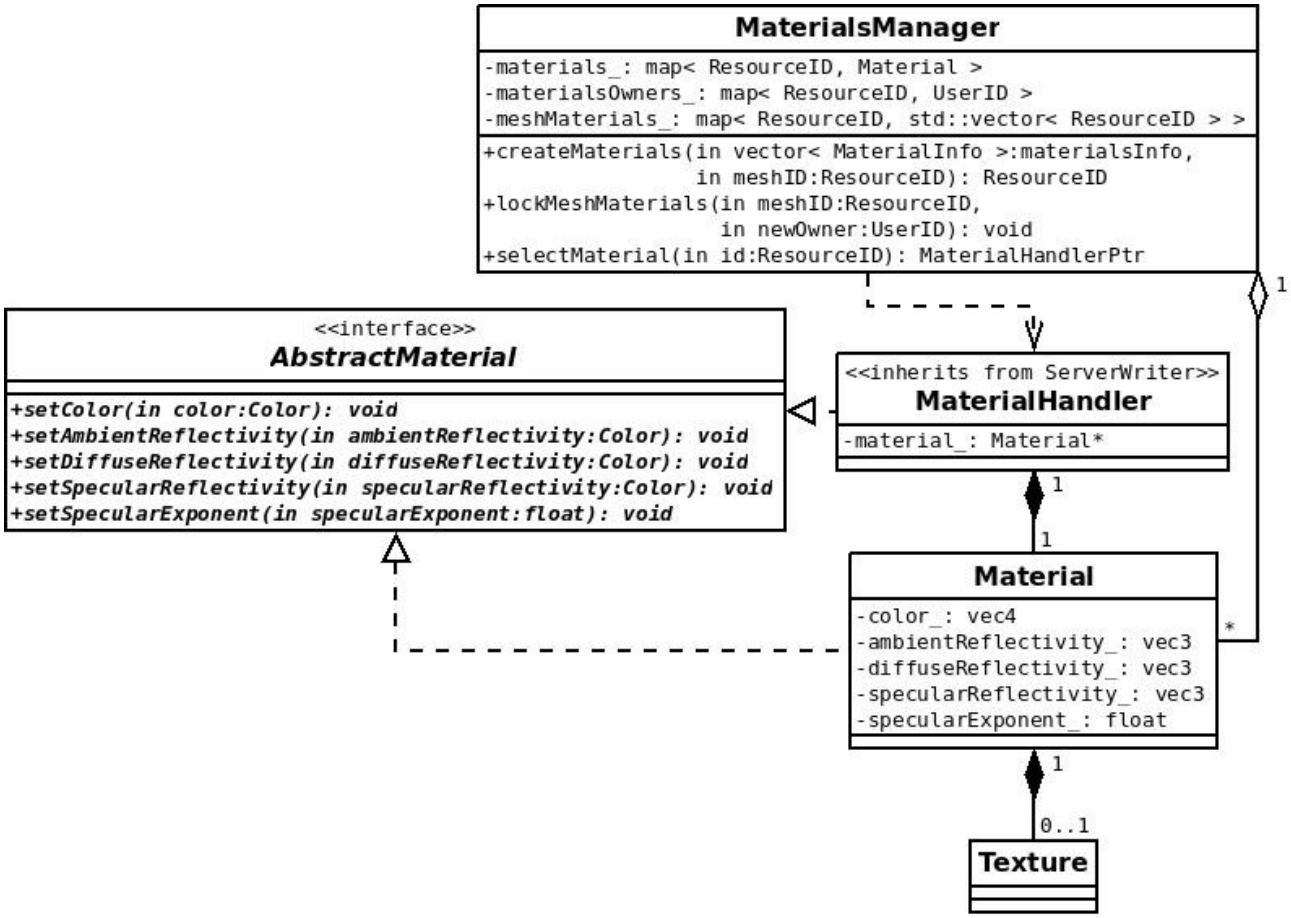


Ilustración 6.18: El gestor de materiales, en contexto

Cuando el usuario local selecciona un material en la clase *MaterialsManager*, se le devuelve un manejador *MaterialHandler*, a través del cuál modificará el material. *MaterialHandler* deriva de *ServerWriter*, y envía automáticamente al servidor todas las acciones del usuario sobre el material en forma de comandos.

6.3.8.3.2. El gestor de texturas (*TexturesManager*)

El gestor de texturas actúa únicamente como contenedor para las texturas cargadas desde fichero para su uso en mallas del sistema (cubos, conos, cilindros y esferas). Las texturas cargadas como parte de una malla importada desde fichero se guardan en el gestor de materiales, por lo que se eliminan junto con su malla asociada.

Debido a que COMO no permite editar texturas, *TexturesManager* no cuenta con métodos de bloqueo y desbloqueo.

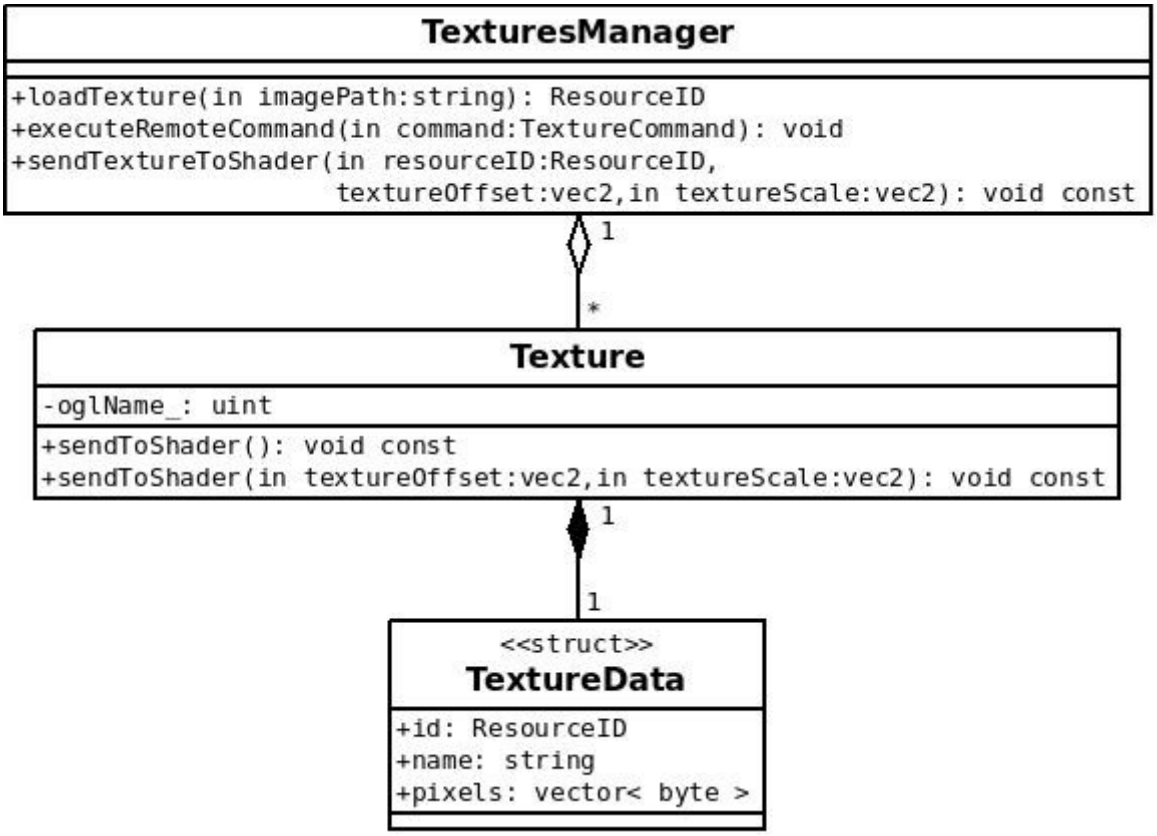


Ilustración 6.19: El gestor de texturas en contexto

6.3.8.3.3. El gestor de “texture walls” (TextureWallsManager)

El gestor de “texture walls” permite al usuario local bloquear para su edición una única “texture wall” de entre aquellas presentes en su selección actual[69]. Cuando se bloquea una “texture wall”, el usuario local recibe un manejador (clase *TextureWallHandler*) para editar el recurso en cuestión mientras los cambios se sincronizan automáticamente en el servidor (gracias a que el manejador deriva de *ServerWriter*).

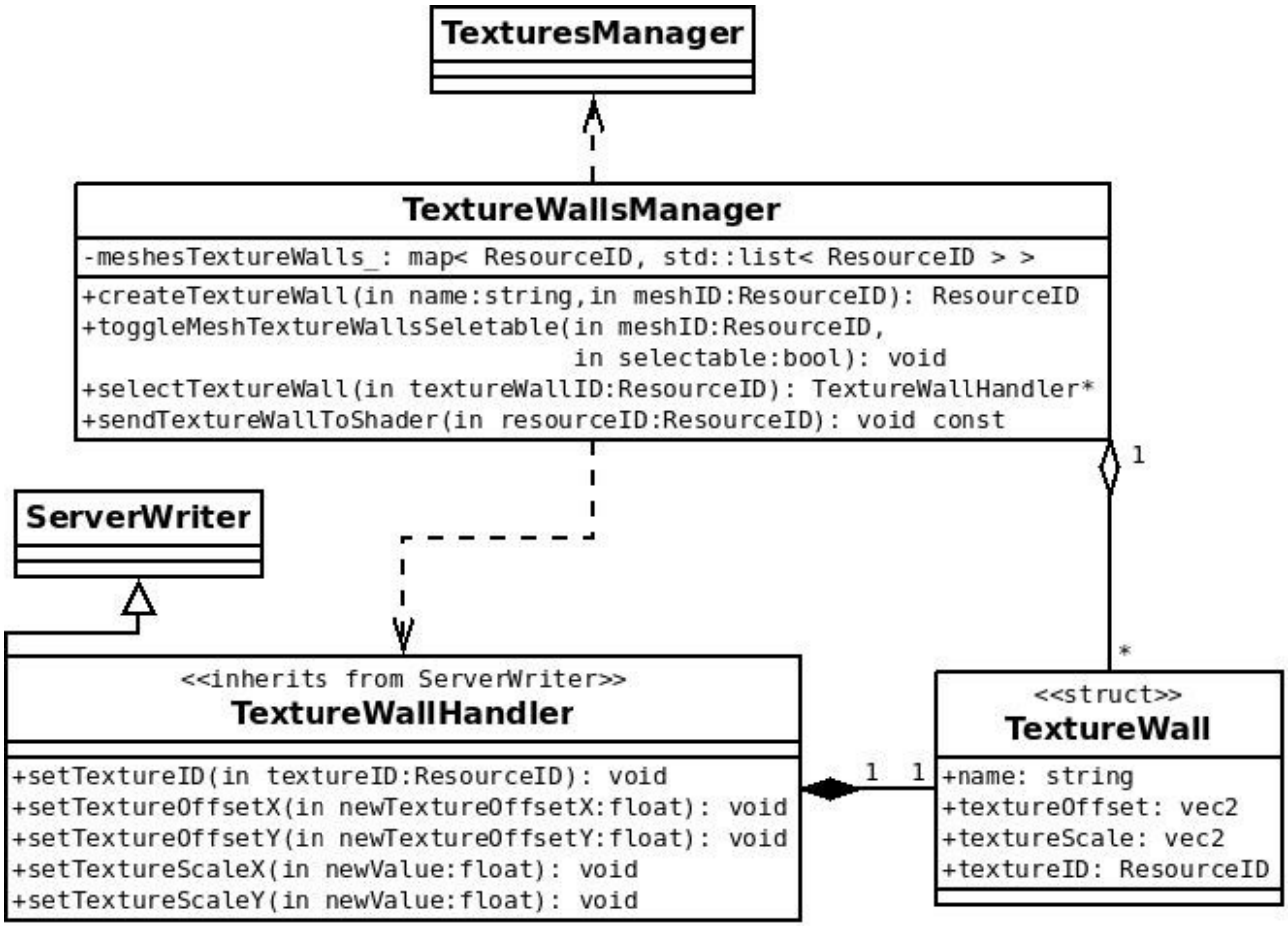


Ilustración 6.20: El gestor de “texture walls”, en contexto.

69 El gestor de “texture walls” asocia a cada una una malla mediante un mapa que relaciona el ID de una “texture wall” con el ID de malla a la que pertenece. Esto permite restringir que el usuario sólo pueda bloquear las “texture walls” pertenecientes a mallas presentes en su selección actual.

6.3.9. Primitivas

En el contexto de COMO, **una primitiva es una plantilla a partir de la cual pueden generarse múltiples mallas idénticas.** Usando un símil con la programación orientada a objetos, una primitiva sería una clase y una malla sería la instanciación de dicha clase.

COMO permite importar primitivas a partir de ficheros OBJ[70]. A partir del fichero OBJ se genera un fichero monolítico con formato propio y extensión *.prim. Este fichero contiene la especificación completa, en modo texto, de una primitiva incluyendo sus vértices, sus materiales y sus texturas. **Los ficheros de primitiva (*.prim) generados se sincronizan entre los clientes**. A partir de ese momento, cualquier cliente puede generar mallas a partir de la primitiva.

Como se observa en la figura siguiente, **el gestor de primitivas del cliente (*ClientPrimitivesManager*) deriva de *AbstractPrimitivesManager,* la cual se usa tanto en el cliente como en el servidor** para forzar que ambos sean capaces de crear primitivas.

Sin movernos de la misma figura, se observan dos estructuras de datos auxiliares con nombres similares: *PrimitiveInfo* y *PrimitiveData*. **La primera se usa de manera persistente en el gestor de primitivas**, donde existe una estructura de tipo PrimitiveInfo por cada primitiva registrada en el sistema. **La estructura *PrimitiveInfo* contiene el nombre y la ruta al fichero de especificación (*.prim) de una primitiva, así como el identificador de la categoría asociada a la primitiva.**

Por otra parte, la estructura *PrimitiveData* se usa de manera temporal, al crear o instanciar una primitiva. **Dicha estructura contiene la especificación completa de una primitiva**, leída directamente desde su correspondiente fichero de especificación. Durante la instanciación de una primitiva, el método *ClientPrimitivesManager::instantiatePrimitive*() obtiene la estructura PrimitiveData requerida y se la suministra a MeshesManager::createMesh() para crear la malla a partir de la primitiva.

70 Wavefront .obj file – Wikipedia, the free encyclopedia: http://en.wikipedia.org/wiki/Wavefront_.obj_file

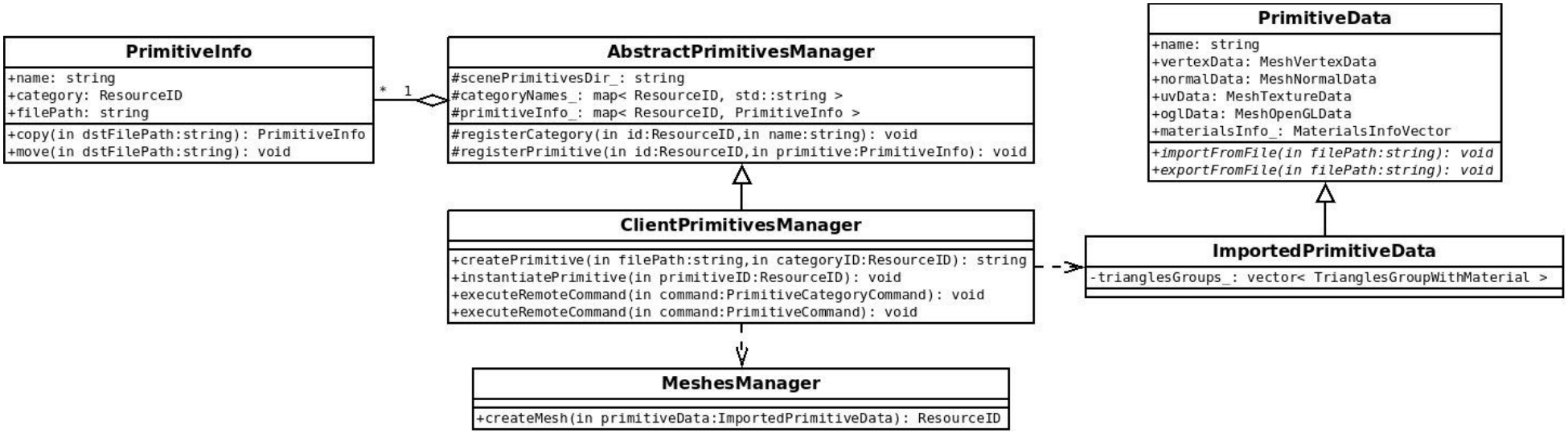


Ilustración 6.21: Jerarquía de clases responsables de la creación e instanciación de primitivas

6.3.9.1.1. Importando primitivas. El importador de OBJ

El importador de OBJ (clase *OBJPrimitivesImporter*), **lee un fichero en formato OBJ y genera una especificación de primitiva en una estructura ImportedPrimitiveData**. Acto seguido, guarda la especificación de primitiva en un fichero con formato propio y extensión .prim.

Como puede observarse en el diagrama, la clase OBJPrimitivesImporter (y cualquier otro importador que se añada debería hacerlo también), implementa la interfaz *PrimitivesImporter*. Esta última define un único método *importPrimitive*() en el que se especifica el fichero de origen (a partir del cual se generará la primitiva) y el fichero destino (aquel en el que se guardará la primitiva generada).

En el caso de *OBJPrimitivesImporter*, su implementación del método *importPrimitive*() anterior llamará a *processMeshFile*() para procesar, línea a línea, el fichero .obj con la especificación de la malla. Cuando encuentre una referencia a un fichero de material (.mtl), invocará a *processMaterialFile*() sobre el mismo. Si el fichero de material contiene referencias a ficheros de texturas, el importador las cargará llamando a *processTextureFile*().

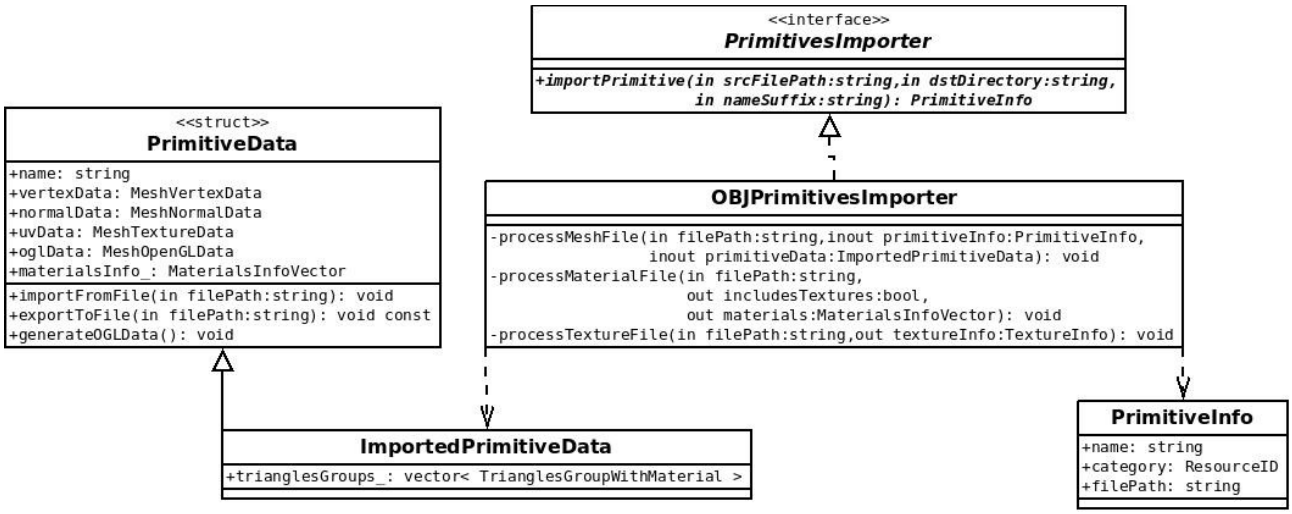


Ilustración 6.22: Las clases OBJPrimitivesImported, PrimitivData y PrimitiveInfo en contexto

Como se ha comentado anteriormente, el importador de primitivas irá rellenando una estructura *ImportedPrimitiveData*, la cual deriva de *PrimitiveData*. Esta última incluye el nombre de la primitiva, así como la información de sus materiales y sus triángulos de posiciones, coordenadas UV y normales[71]. Además, la clase *PrimitiveData* incluye una estructura *MeshOpenGLData* con la forma siguiente:

71 Las estructuras *MeshVertexData*, *MeshNormalData* y *MeshTextureData* contienen cada una un vector de posiciones, normales o coordenadas UV respectivamente, además de un vector de índices al primero. Este último define los triángulos de la malla.

Ilustración 6.23: Estructura MeshOpenGLData

La clase *MeshOpenGLData* incluye dos buffers (vectores) con los contenidos que poblarán, respectivamente, el VBO y el EBO asociados a las mallas que se generen a partir de la primitiva[72].

Esta estructura no se lee desde fichero, sino que se genera, por medio del método PrimitiveData::generateOGLData(), a partir de los triángulos de posiciones, normales y UV que sí se obtuvieron desde el fichero. **Esto se debe al hecho de que el formato OBJ trabaja con múltiples índices a la vez a la hora de definir las caras de la malla, mientras que OpenGL sólo permite trabajar con un índice a la vez durante el renderizado**.

Por ejemplo, en el formato OBJ, para indexar un vértice perteneciente a una cara, se usaría tres índices:

- Un índice i, que apuntaría a la posición dentro del vector de posiciones de la malla donde se define la posición del vértice actual.
- Un índice j, que apuntaría a la posición dentro del vector de coordenadas UV de la malla donde se define las coordenadas UV del vértice actual.
- Un índice k, que apuntaría a la posición dentro del vector de normales de la malla donde se define la normal del vértice actual.

Por el contrario, OpenGL sólo permite trabajar con un índice a la vez para indexar vértices para su renderizado. En este caso se tendría un índice i a la posición del VBO donde se define la posición, la coordenada UV y la normal del vértice.

Lo anterior requiere una transformación de un sistema de múltiples índices (OBJ) **a un sistema de un único índice** (OpenGL). Esta transformación se desarrolla en el método *PrimitiveData::generateOGLData*() y se apoya en un mapa que relaciona cada multi-índice (i, j, k) con un índice simple I. El modo de proceder del método de transformación se resume en el siguiente diagrama de flujo:

72 El VBO y el EBO son dos objetos de OpenGL que contienen, respectivamente, los datos e índices de todos los vértices que conforman la malla.

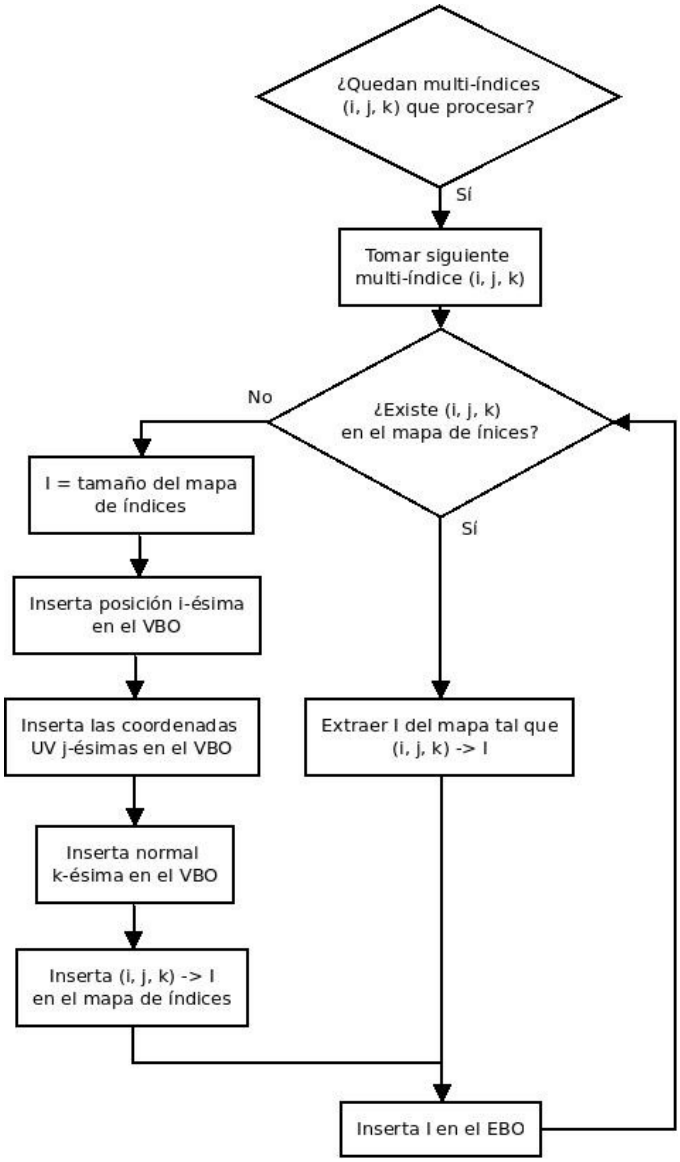


Ilustración 6.24: Procedimiento de transformación de multi-índices (OBJ) a mono-índices (OpenGL)

6.3.10. Factorías de mallas del sistema

COMO permite generar hasta cuatro tipos distintos de mallas a partir de parámetros definidos por el usuario (mallas del sistema): cubos, conos, cilindros y esferas. Cada tipo de malla del sistema cuenta con su propia clase factoría, como se muestra en el siguiente diagrama:

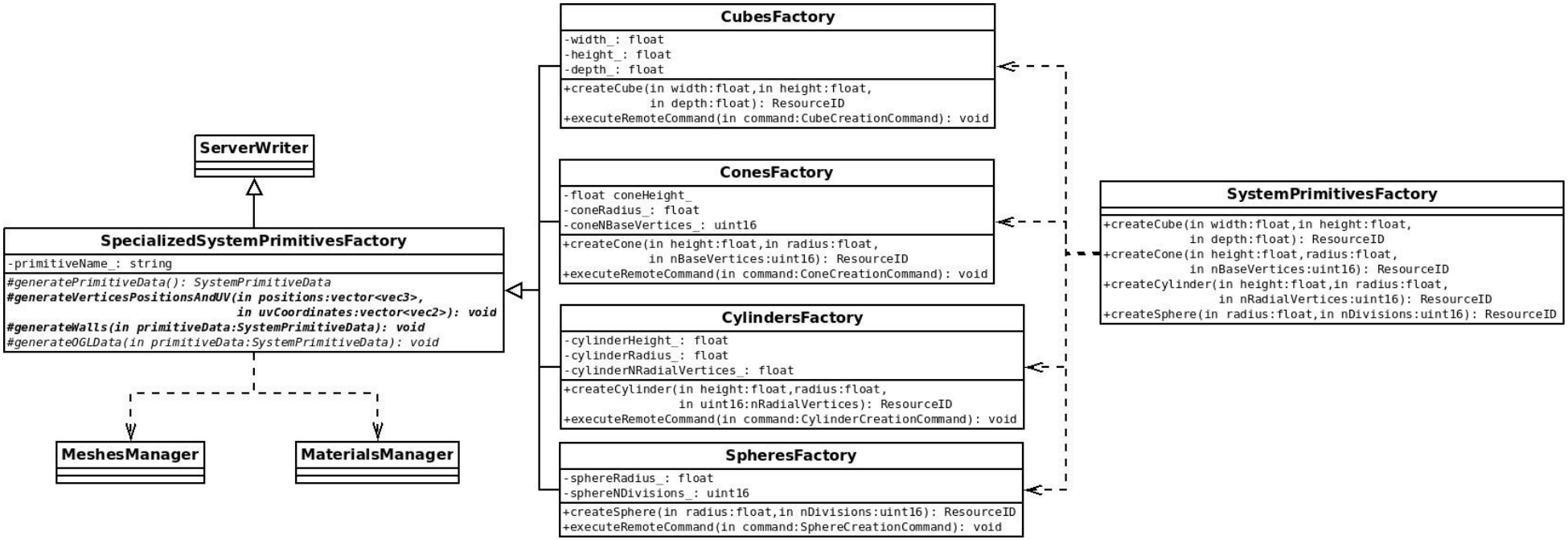


Ilustración 6.25: Factorías de primitivas del sistema

Cada método de creación en las clases derivadas (createCube(), createCone(), etc) **copia los parámetros de construcción** (pe. altura del cono) **a atributos de la clase y, acto seguido, invoca al método *SpecializedSystemPrimitivesFactory::generatePrimitiveData()***. Este método va invocando una serie de métodos auxiliares que irán completando la estructura *SystemPrimitiveData*: generateVerticesPositionsAndUVs(), generateWalls(), generateOGLData(), etc. Cómo puede observarse, algunos de estos métodos son abstractos en la definición de la clase *SpecializedSystemPrimitivesFactory*. **Cada factoría concreta implementará estos métodos de forma que creen la primitiva requerida**.

Por último, **la clase *SystemPrimitiveFactory* actúa de interfaz entre el usuario de las factorías y cada una de las factorías concretas**.

6.3.11. Interfaz de usuario

La interfaz de usuario se describe en el manual de uso de COMO, el cuál se encuentra anexo al final de esta memoria.

6.4. Diseño del servidor

El servidor de COMO es un programa carente de modo gráfico (modo consola), **para el que una escena es una combinación de usuarios, recursos con sus respectivos dueños, y comandos**.

6.4.1. Visión general

A continuación se muestra un diagrama con las principales clases del servidor y sus interrelaciones:

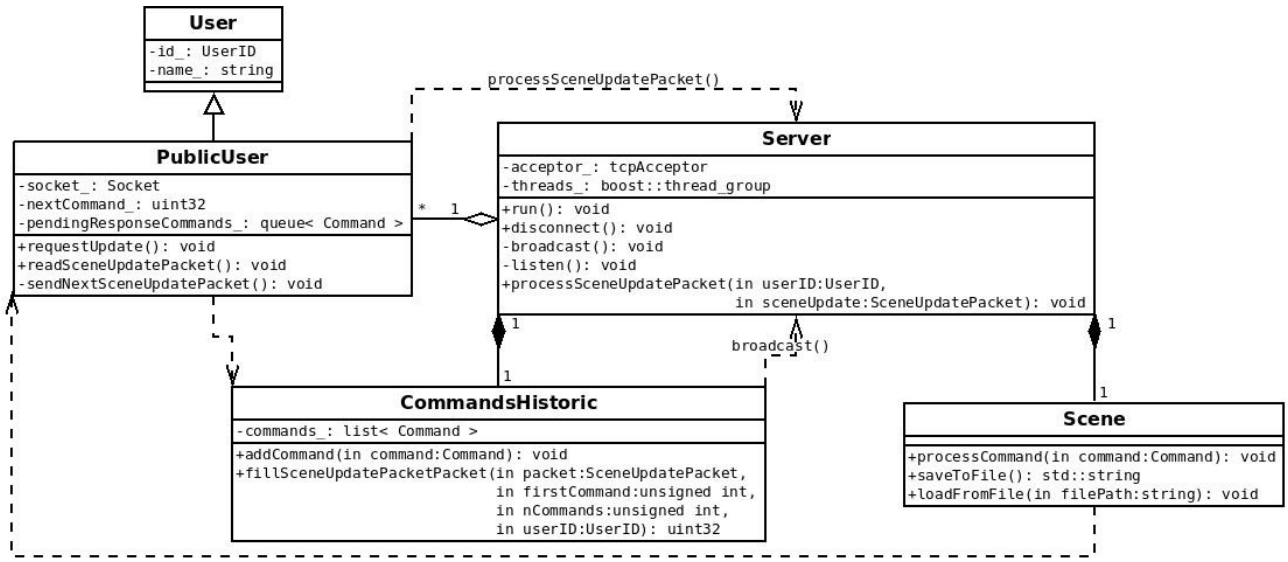


Ilustración 6.26: Visión general de las clases principales del servidor

En los siguientes subapartados se describe someramente cada una de estas clases salvo la clase *Scene*. Esta última, debido a su mayor complejidad, se describe en detalle el siguiente apartado.

6.4.1.1. La clase Server

La clase *Server* representa al servidor de COMO. Su responsabilidad principal es la de establecer una conexión con cada cliente que lo solicite, o denegar la petición si se ha alcanzado el límite de usuarios conectados.

La clase *Server* cuenta con un grupo de hilos (boost::thread_group) **que realizan las labores de mantenimiento del servidor**[73]. Las tareas que realizan se listan a continuación:

- **Aceptar las peticiones de conexión recibidas por parte de los usuarios y establecer la conexión definitiva con los mismos.**
- **Denegar las peticiones de conexión recibidas por parte de los usuarios cuando no haya más espacio en el servidor.**
- **Recibir los paquetes de tipo *SceneUpdatePacket* desde cada cliente** y extraer sus comandos para pasárselos a la clase *Scene* para su ejecución.
- **Acceder al histórico de comandos** (clase *CommandsHistoric*) **y sincronizar sus comandos con cada cliente conectado a la escena**.

6.4.1.2. El histórico de comandos (clase CommandsHistoric)

El histórico de comandos (CommandsHistoric) **actúa como el contenedor de todos los comandos recibidos que hayan sido aceptados por el servidor**. Cada vez que un nuevo comando se añade al histórico, se invoca a *Server::broadcast()*. Este último método se encarga de invocar a *PublicUser::requestUpdate()* sobre cada usuario conectado a la escena para informarle de que existe un nuevo comando por sincronizar.

Además de permitir la inserción de comandos, **la clase *CommandsHistoric* cuenta con un método para rellenar los paquetes de actualización** (*SceneUpdatePacket*) **que se enviarán a cada cliente**. El método *CommandsHistoric::fillSceneUpdatePacket*() toma un paquete de tipo *SceneUpdatePacket* y un índice “nextCommand” y rellena el primero con comandos del histórico a partir del índice “nextCommand”.

6.4.1.3. La clase PublicUser

A cada cliente conectado al servidor se le asocia un objeto de tipo *PublicUser*, el cual incluye el identificador (*UserID*) y el nombre del usuario. Este objeto contiene además el socket con la conexión con el cliente, además de métodos para “escuchar” los paquetes que provienen desde el cliente o para enviarle paquetes desde el servidor.

73 Algunas de estas tareas de mantenimiento incluyen operaciones de E/S como la transmisión y recepción de datos a través de sockets. Estas operaciones hacen uso de io_service, la interfaz con los servicios de E/S proporcionada por la librería Boost Asio. Véase el subapartado “Boos Asio y la E/S asíncrona” en el apartado de Análisis.

6.4.2. La clase Scene

La clase *Scene* es la encargada de actualizar el estado de la escena en el servidor procesando los comandos que recibe desde los clientes. Además, también se encarga del salvado y carga de la escena actual.

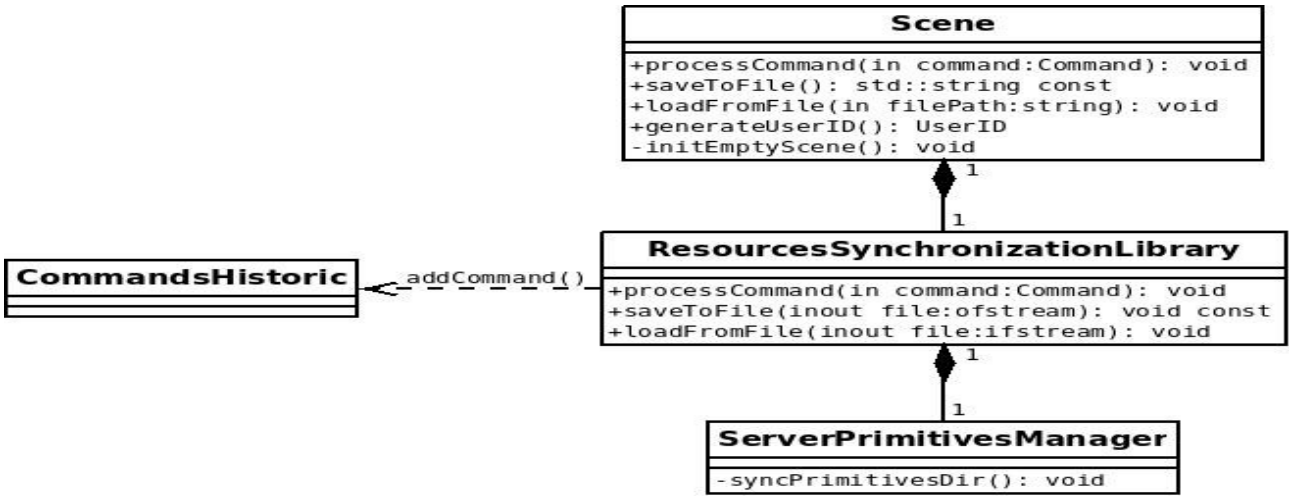


Ilustración 6.27: Clase Scene y sus colaboraciones

La actualización de la escena (*Scene*) **se apoya en clase *ResourcesSynchronizationLibrary*, la cual se encarga de sincronizar todos los recursos de la escena**. Esta clase, así como el procedimiento de salvado y carga de escenas, se describirá en detalle en el siguiente apartado.

Contenido en la clase *ResourcesSynchronizationLibrary* se encuentra la clase *ServerPrimitivesManager*. **Durante la inicialización del servidor** (para escenas vacías), **el gestor de primitivas recorre el directorio *data/local/primitives* y emplea la clase *OBJPrimitivesImporter*[74] para importar todos los ficheros *.obj que se encuentren organizados por categorías** (una categoría = un directorio) **en el mismo**. Cada fichero *.obj es procesado y se genera un fichero de especificación de primitiva (*.prim) que se sincroniza en el histórico de comandos.

6.4.3. La clase *ResourcesSynchronizationLibrary*

La clase *ResourcesSynchronizationLibrary* se encarga de sincronizar los recursos de la escena en el servidor, asociándole a cada recurso un objeto de tipo *ResourceSyncData*.

La administración de la propiedad de cada recurso por parte de los usuarios, surge del contrato que adquiere *ResourcesSynchronizationLibrary* al implementar la interfaz *ResourcesOwnershipManager*. Esta responsabilidad se satisface al incluir en cada objeto *ResourceSyncData* un campo de tipo *UserID* indicando el dueño actual del recurso. El servidor usa este valor para procesar las peticiones de bloqueo y desbloqueo de recursos recibidas desde los clientes.

74 Véase el subapartado “Importando primitivas. El importador de OBJ” en el apartado de “Diseño del cliente”.

Por otra parte, ***ResourcesSyncData*** **mantiene la información mínima necesaria para reproducir cada recurso de la escena desde cero, facilitando el salvado y carga de la escena desde fichero**.

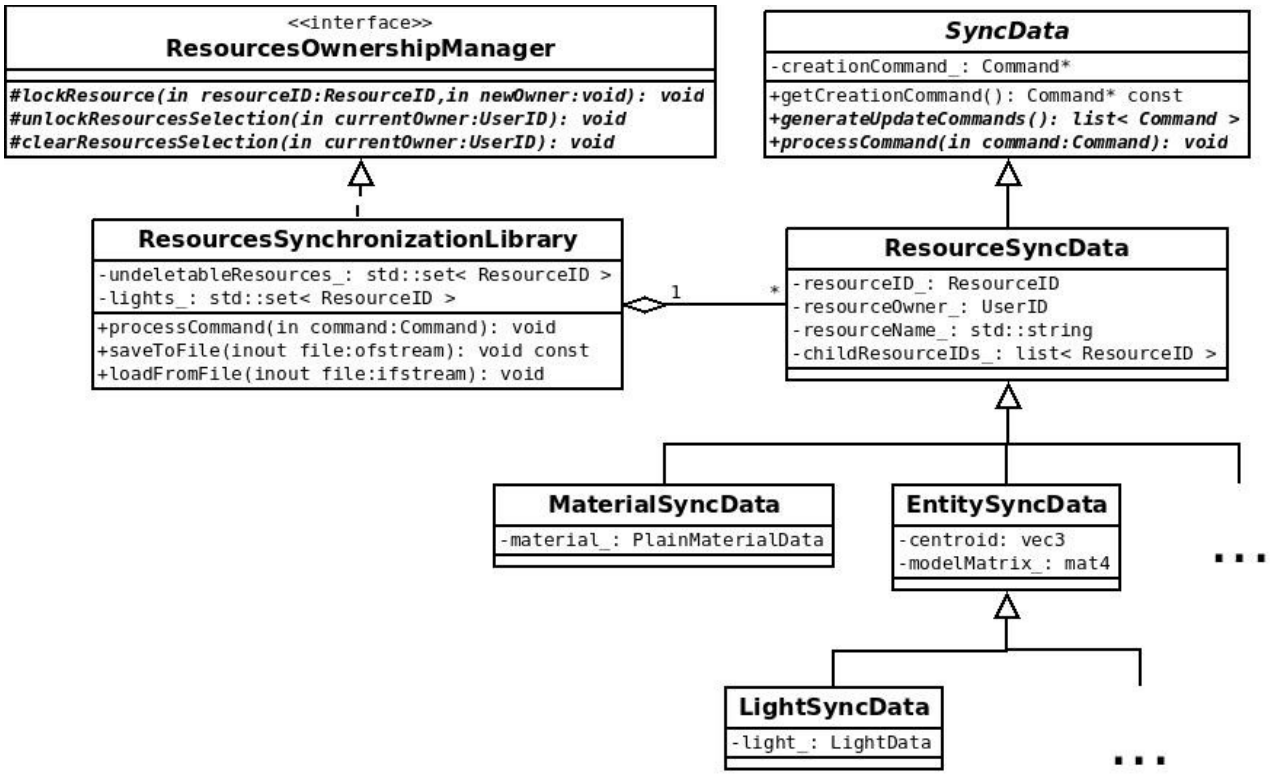


Ilustración 6.28: Clase ResourcesSynchronizationLibrary y la jerarquía de objetos SyncData en la que se apoya

6.4.3.1. La clase *ResourceSyncData*

Como se mencionó en el apartado anterior, cada objeto ***ResourceSyncData*** **mantiene la información mínima necesaria para reproducir un recurso de la escena desde cero, facilitando el salvado y carga de la escena desde fichero**. **Cada objeto *ResourceSyncData* asocia a su recurso un comando opcional de creación y 0 o más comandos de actualización**.

El comando de creación de un recurso es, como su nombre indica, el comando que se empleó para crear el recurso. **La existencia de este comando de creación en la estructura *ResourceSyncData* es opcional puesto que existen recursos que se crean de manera indirecta**. Por ejemplo, **un comando de instanciación de primitiva** (clase *PrimitiveInstantiationCommand*) **provoca que se cree una malla de manera directa e, indirectamente, todos los materiales asociados a dicha malla**. Por esta razón, sólo el objeto *ResourceSyncData* asociado a la malla contiene el comando de creación anterior; incluirlo también en los materiales llevaría a duplicidades y confusión.

Por su parte, **los comandos de actualización asociados a un objeto *ResourceSyncData*** no están contenidos en este último, sino que **se generan de manera temporal cuando son requeridos**

para salvar la escena. Esta generación se realiza mediante el método *SyncData::generateUpdateCommands()*, a partir de los datos guardados sobre el recurso en la misma estructura. Por ejemplo, la clase *LightSyncData* contiene una estructura de datos con el color y el coeficiente ambiente de la luz. Cuando se invoca a LightSync*Data::generateUpdateCommands()*, se genera una lista con un comando *LightColorChangeCommand* y otro comando *LightAmbientCoefficientChangeCommand* con los últimos valores guardados en la estructura anterior.

6.4.3.2. Salvado de escenas

Cuando el usuario cierra el servidor, éste le pregunta si desea guardar la escena en el directorio data/save. En caso de que el servidor haya sufrido un error, intentará guardar la escena en el fichero “data/save/autosave”.

El salvado de la escena comienza con una llamada a *Scene::saveToFile()*. El procedimiento que se sigue internamente se resume en el siguiente diagrama de flujo:

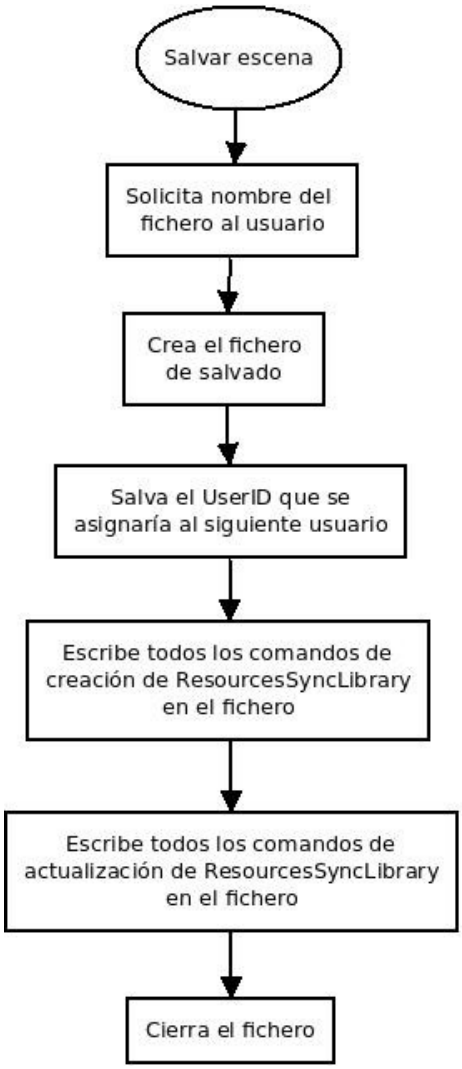


Ilustración 6.29: Procedimiento de salvado de una escena

Como se puede observar en el diagrama anterior, **lo primero que se salva en el fichero de la escena, es el ID que se le asignaría al siguiente usuario que se conectara a la escena**. Esto es así para evitar que se reutilicen IDs de usuarios anteriores al cargar la escena salvada. En caso contrario, a cada usuario que se conectara a la escena se le tendría que vetar las *ResourceIDs* que ya hubieran usado aquellos usuarios que tuvieron el mismo ID de usuario antes que él o ella[75].

A continuación se escriben todos los comandos de creación y de actualización de recursos directamente en el fichero. Cada comando es empaquetado (método Packable::pack()) para su escritura en fichero.

6.4.3.3. Carga de escenas

La carga de una escena desde fichero se produce en el momento de crear el servidor,

75 (Recordatorio) Esto se produce debido a que cada *ResourceID* incluye en su interior el *UserID* del usuario que creó el recurso.

siempre que el usuario haya especifica una ruta al fichero en cuestión. El procedimiento es análogo al de salvado: primero se lee el identificador (*UserID*) que se dará al siguiente usuario conectado. A continuación se leen y procesan uno tras otro todos los comandos de creación y de actualización escritos en el fichero durante el salvado de la escena.

6.5. Jerarquía de directorios

6.5.1. Visión general

El repositorio de COMO[76] cuenta con tres carpetas principales: bin/, project/ y src/. Cada carpeta a su vez divide sus contenidos entre las subcarpetas client/, server/ y common/ según se refieran al programa cliente, al programa servidor o a la parte común a ambos, respectivamente.

- **bin/**: directorio donde se generan los ejecutables (binarios). En esta carpeta también se incluyen todos aquellos ficheros de configuración y de otra índole necesarios para la correcta ejecución del programa: ficheros de especificación de primitivas, shaders (cliente), primitivas a importar (servidor), etc.

- **project/**: aquí se encuentran los ficheros de proyecto (extensiones .pro y .pri) necesarios para construir el cliente y el servidor usando el entorno de desarrollo *Qt creator*. Tales ficheros listan los fuentes del software, los parámetros del compilador y del enlazador (linker), etc.

- **src/**: código fuente del software.

6.5.2. La carpeta de datos (bin/X/data/)

Las carpetas bin/server y bin/client incluyen una subcarpeta data/ con una estructura muy similar en ambos casos. A continuación se dedica un subapartado a la carpeta data/ del cliente y del servidor

6.5.2.1. La carpeta de datos (cliente)

La carpeta bin/client/data contiene la siguiente jerarquía de directorios:

- **scenes/** : contiene una subcarpeta <scene> por cada escena que el cliente esté sincronizando actualmente.
    - **<scene>/**
        - **primitives/** : ficheros de especificación (.prim) de las primitivas sincronizadas por el cliente. Las primitivas se organizan por carpetas según su categoría.
        - **textures/** : texturas cargadas en la escena.
        - **.temp/** : directorio temporal usado durante la transmisión de primitivas entre cliente

76 Repositorio de COMO en Github: https://github.com/moisesjbc/como

y servidor.

- **shaders/**: programas shader usados para el renderizado 3D.
- **system/primitives/**: contienen ficheros de especificación de primitivas (.prim) para aquellas primitivas usadas internamente por COMO, como la primitiva “camera” o la primitiva “directional_light”.

6.5.2.2. La carpeta de datos (servidor)

- **scenes/** : contiene una subcarpeta por cada escena que el servidor esté sincronizando actualmente.
  - **<scene>/**
    - **.temp/** : directorio temporal usado cuando se transmiten primitivas al y desde los clientes.
- **local/primitives/** : ficheros *.obj con las primitivas que se desean importar al comienzo del servidor organizadas por categorías.
- **save/** : ficheros de salvado de escena (extensión .csf).

6.5.3. La carpeta de código (src/)

Las carpetas src/client, src/common/ y src/server/ contienen el código fuente de la aplicación.

6.5.3.1. La carpeta de código del cliente (src/client/)

El código del cliente se divide en tres carpetas:

- **gui/** : contiene todas las clases y estructuras de datos auxiliares usadas por la **interfaz gráfica del cliente**. Incluye el panel de herramientas (clase *ToolsPanel*) y la vista 3D (clase *Viewport*), entre otros.
- **managers**/ : alberga los objetos de datos que se manejan directamente desde la interfaz. Clases como *ResourcesManager*, *EntitiesSelection* o *MaterialHandler* se encuentran aquí.
- **model/** : incluye los bloques de construcción básicos de la escena, aquellos a los que se acceden a través de los managers. Clases como *Mesh*, *Light*, o *Camera* están presentes aquí.

6.5.3.2. La carpeta de código del servidor (src/server/)

El código del servidor se divide en las siguientes subcarpetas:

- **managers/ :** aquí se encuentran los gestores empleados por el servidor (gestor de primitivas,

la clase *Scene*, etc).

- **sync_data/** : incluye la especificación de la clase *ResourceSyncData* y sus clases derivadas.

6.5.3.3. La carpeta de código común al cliente y al servidor (src/common/)

La carpeta de código común incluye las siguientes subcarpetas principales:

- **commands/** : todos los comandos usados por COMO.
- **ids/** : definiciones de las clases directamente con la identificación de usuarios y recursos: *ResourceID*, *UserID*, *ResourceIDsGenerator*.
- **managers/** : incluye el código base de aquellos gestores que se encuentran presentes en el cliente y el servidor, cómo el gestor de primitivas o el de texturas.
- **packables/** : contiene la jerarquía de tipos de datos auxiliares y empaquetables: *Packable*, *PackableUint32*, *PackableFile*, etc.
- **packets/** : definiciones de todas las clases que representan paquetes para su transmisión a través de la red: *Packet*, *NewUserPacket*, *SceneUpdatePacket*, etc.
- **utilities/** : clases auxiliares: *Lockable*[77], *Log*[78], etc.

77 La clase *Lockable* es la clase base de todas las estructuras de datos susceptibles de acceder concurrentemente a sus datos. *Lockable* incluye un cerrojo privado y los mecanismos apropiados para tomar y liberar dicho cerrojo.

78 La clase *Log* proporciona métodos para imprimir mensajes de información, precaución (warning) o error por consola. *Log* deriva de *Lockable* para permitir su uso seguro en un programa multi-hilo.

7. Desarrollo

7.1. El origen: un plugin para Blender

Antes de idear el proyecto COMO, este proyecto de final de carrera se orientó hacia la creación de un plugin para Blender. Este plugin permitiría extender Blender para que múltiples usuarios se conectaran a través de la red y pudieran trabajar al unísono sobre una misma escena.

Debido a la gran complejidad del proyecto Blender y a que supuse más didáctico comenzar un pequeño modelador cooperativo desde cero, **decidí desechar este proyecto e iniciar el proyecto COMO**.

7.2. El primer incremento: un manipulador de cubos offline

Para el primer incremento de COMO opté por ignorar la componente online y **centrarme en crear un pequeño modelador en local, donde el usuario pudiera crear y transformar cubos**.

Lo primero que hice fue tratar de decidir qué librería usaría para crear la interfaz gráfica de usuario (GUI) de la aplicación. Finalmente opté por usar Qt, debido a que se trataba de un proyecto veterano, multiplataforma y libre, que además ofrecía facilidades para su integración con OpenGL. Además, yo ya contaba con algo de experiencia usando esta librería para interfaces gráficas.

A continuación me dispuse a aprender OpenGL en su versión más moderna (4.3), intentando huir de los tutoriales que plagan la red usando características obsoletas de la API. Para ello adquirí el libro "OpenGL Programming Guide"[79] y procedí a estudiarme los temas introductorios.

Habiendo repasado la librería Qt y la API de OpenGL, estudié la integración de ambas librerías. Gracias a los tutoriales del sitio de Qt[80], no tardé en crear una aplicación de prueba consistente en una ventana (Qt) donde se renderizaba un cubo 3D (OpenGL).

COMO se diseñó para incluir las cuatro vistas típicas que suelen usarse en todo programa de modelado 3D: frontal, izquierda, superior y cámara. Por esta razón, **se estimó oportuno que el siguiente paso consistiera en crear una aplicación donde coexistieran cuatro vistas 3D compartiendo el mismo contexto de OpenGL**.

Al principio se dejó que las cuatro vistas 3D mostraran lo mismo: la escena vista desde el frente; no se permitía al usuario cambiar la vista. En su lugar **se procedió a permitir que el usuario pudiera crear cubos desde un botón en la interfaz**. A continuación se implementó la selección y deselección de cubos mediante clics del ratón sobre las vistas 3D. Finalmente se añadieron las capacidades de transformar selecciones de cubos: traslaciones, rotaciones y escalados. **Todas las**

79 **SHREINER, Dave; SELLERS, Graham; KESSENICH, John; LICEA-KANE, Bill**. *OpenGL Programming Guide*. Eight Edition. Addison-Wesley.

80 OpenGL Window Example | QtGUI 5.3 | Documentation | Qt Project: http://qt-project.org/doc/qt-5/qtgui-openglwindow-example.html

transformaciones se realizaban con respecto al origen del mundo.

Después de permitir la creación y manipulación de cubos 3D, se ofreció al usuario la opción cambiar la vista (“front”, “back”, “left”, “right”, “top”, “bottom”) y el tipo de proyección (ortogonal o perspectiva) de cada vista 3D.

Finalmente, se añadieron opciones para poder trasladar, rotar y escalar las selecciones de cubos con respecto a un eje (X, Y o Z). **Además, se añadió la opción de especificar tres puntos de pivotaje para las rotaciones y los escalados**: origen del mundo, centroide y centroides individuales[81].

La siguiente imagen ha sido extraída de un vídeo publicado en Youtube[82] mostrando el estado actual del proyecto en este punto.

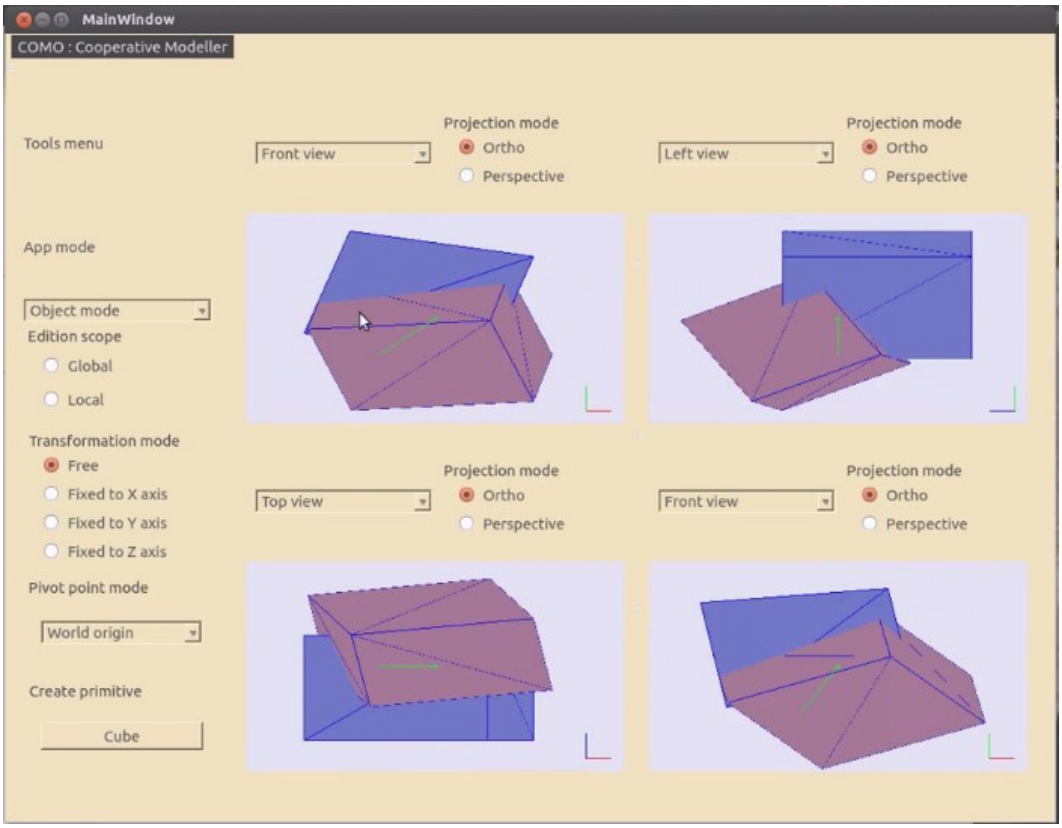


Ilustración 7.1: Primer incremento del proyecto COMO - Un manipulador de cubos offline

81 Al seleccionar “centroides individuales” y rotar una selección de entidades, cada entidad gira en torno a su propio centroide.

82 Cooperative Modeller – First video (vídeo en Youtube): https://www.youtube.com/watch?v=_thvV-icsM0&feature=youtu.be

7.3. El segundo incremento: un manipulador de cubos online

Ya se había conseguido un sencillo manipulador de cubos en local. Ahora **era el momento de transformarlo en un manipulador de cubos online**. Mientras diseño este nuevo incremento, **me decanto por el bloqueo de recursos[83] y establezco que todas las escenas compartidas en COMO sean públicas**: cualquier usuario puede conectarse al servidor si sabe su IP.

Para el salto a la programación distribuida opto por usar la librería Boost Asio. Invierto tiempo en aprenderla siguiendo e implementando los ejemplos de dos fuentes: la propia documentación de Boost Asio[84], y una guía de introducción[85] a la misma.

Tras el estudio, comienzo la implementación del programa servidor con la **creación de una clase *Server***. Para ello parto del ejemplo "Chat server" disponible en la documentación de Boost Asio[86]. **La primera versión de la clase Server crea una "piscina de hilos"** (thread pool) para atender la E/S de red, **y posteriormente inicia una espera asíncrona** (no bloqueante) **por una conexión desde el cliente**.

Cuando consigo la conexión cliente – servidor, **me preocupo por permitir múltiples usuarios conectados a la vez hasta un límite**. **También implemento que el servidor borre a un cliente si se pierde la conexión** con éste.

A continuación **comienzo a diseñar las estructuras de datos necesarias para dar soporte a los diferentes tipos de paquetes** que se enviarán entre los clientes y el servidor. **En el diseño tengo en cuenta la problemática de conectar a máquinas con diferentes "endianness"[87]**.

Después de diseñar los diferentes paquetes a intercambiar entre cliente y servidor, **hago que el cliente se conecte al servidor usando un paquete NewUserPacket. Si el servidor lo acepta, le responde con un paquete UserAcceptancePacket**.

Hasta este punto del desarrollo, la creación del servidor y la conexión a éste desde el cliente se desarrollaba desde consola y con parámetros fijos. **Permito la creación del servidor y la conexión con el mismo a través de la interfaz del cliente mediante un asistente** (connection wizard).

Ya puedo conectar usuarios al servidor mediante intercambio de paquetes, pero ninguno tiene constancia del resto de clientes conectados. **Codifico el paquete *SceneUpdatePacket* y los comandos *UserConnectionCommand* y *UserDisconnectionCommand*. Modifico el servidor para que haga un "broadcast" de ambos comandos cada vez que un cliente se conecta o desconecta, respectivamente**. También añado una lista de usuarios a la interfaz gráfica del sofware cliente.

83 Véase "Bloqueo de recursos" en el apartado de Análisis.

84 Tutorial de Boost Asio: http://www.boost.org/doc/libs/1_54_0/doc/html/boost_asio/tutorial.html

85 *A guide to getting started with boost::asio*: http://www.gamedev.net/blog/950/entry-2249317-a-guide-to-getting-started-with-boostasio/

86 Chat server – Ejemplo de código en la documentación de Boost Asio: http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio/example/cpp11/chat/chat_server.cpp

87 Veáse el apartado "Diferentes órdenes de bytes (Endianess)" en la sección de Análisis.

En este momento cuento con un servidor con capacidad para múltiples clientes, donde cada cliente tiene constancia de los demás. **Ha llegado el momento de sincronizar la creación y manipulación de cubos entre los diferentes clientes y el servidor**. En primer lugar sincronizo la creación de cubos, y a continuación, la selección y deselección de los mismos. **Finalmente completo el manipulador de cubos online al sincronizar también la transformación de selecciones de cubos**: traslaciones, rotaciones y escalados. Permito transformaciones libres y fijas (con respecto a un eje).

El siguiente fotograma pertenece a un vídeo publicado en Youtube[88] con el estado final de este incremento:

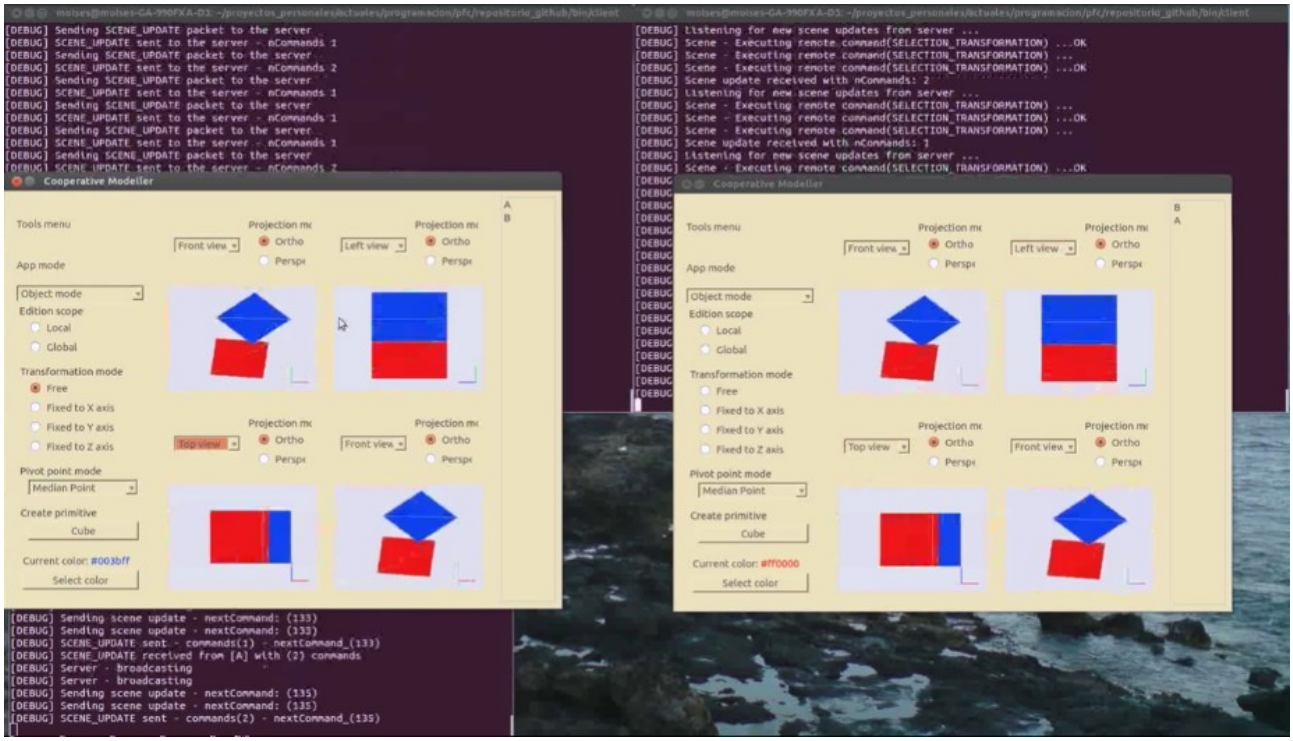


Ilustración 7.2: Segundo incremento del proyecto COMO – Un manipulador de cubos online

7.4. El tercer incremento: incluyendo iluminación y primitivas

Como se ha podido comprobar en las capturas anteriores, el software no contaba con iluminación, mostrando todos los cubos con colores planos. **El siguiente hito del proyecto consistió**, por tanto, **en dotar de una iluminación básica a COMO**. **Además, también se propuso permitir la creación de mallas a partir de primitivas importadas en el servidor.**

En primer lugar se procedió a la creación de un modesto importador que leyera ficheros *.obj y los convirtiera en primitivas para su uso en COMO. **Este importador, en su primera versión, sólo se usaría en el servidor durante su inicialización**: sólo crearía primitivas desde los

88 Modelador en red – prueba de escena compartida: https://www.youtube.com/watch?v=CUBIeVKh8ko&feature=youtu.be

ficheros *.obj alojados en su carpeta local en el momento de crearse el servidor. La creación de primitivas desde el cliente no se incluiría hasta más adelante. Además, como aún no se había implementando los materiales ni las texturas, el importador de OBJ sólo leería los vértices especificados en el fichero fuente.

Tras completar la importación de ficheros OBJ en el servidor y su sincronización en el cliente, **se procedió a trabajar en la iluminación de la escena. El primer paso consistió en asociar una normal a cada vértice de cada malla.** En el caso de la primitiva "Cubo", las normales se generaban en el constructor de la clase *Cube*. En el caso de las primitivas importadas, se amplió el importador de OBJ para que leyera las normales desde fichero.

Con las normales implementadas, **lo siguiente fue añadir una luz direccional a la escena y permitir cambiar su color**. Esta luz era fija y única: no se podía borrar, ni se podía añadir luces adicionales a la escena.

Finalmente se implementaron los materiales, asociando uno por defecto a la primitiva "Cubo" y leyéndolos desde fichero en el caso de las primitivas importadas. Con la inclusión de un "editor de materiales" al panel de herramientas del software cliente, se dio por zanjado este incremento.

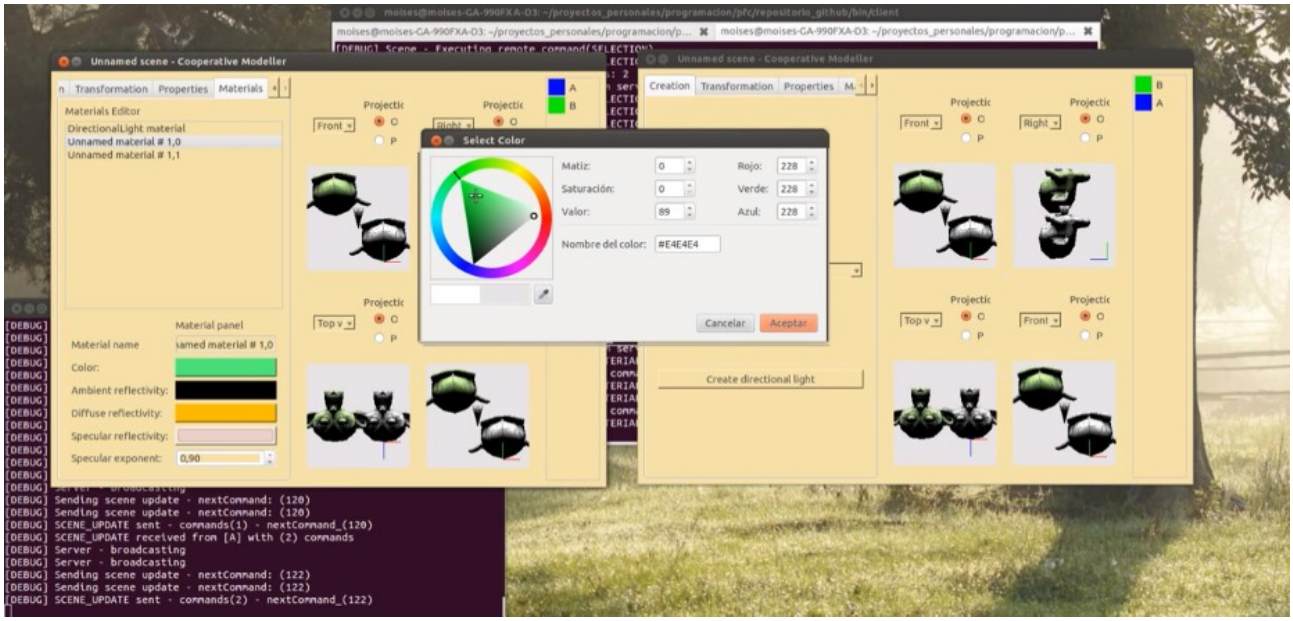


Ilustración 7.3: Tercer incremento del proyecto COMO – Primitivas + iluminación

La imagen anterior es un fotograma de un vídeo publicado en Youtube: "Modelador 3D cooperativo – Prueba de materiales"[89].

89 Modelador 3D cooperativo – Prueba de materiales: https://www.youtube.com/watch?v=NVI6PgY8MrQ

7.5. El cuarto incremento: importando primitivas desde el cliente + texturas

Para el cuarto incremento planteo que cualquier cliente pueda crear primitivas y que éstas sean sincronizadas en el resto de los clientes. Además, trabajo en la inclusión de texturas.

Permitir que los clientes pudieran importar primitivas como lo hacía el servidor fue relativamente sencillo y sin grandes complicaciones. **La inclusión de texturas**, sin embargo, **fue más compleja**. En primer lugar, mi inexperiencia con las texturas en OpenGL hizo necesaria una formación previa. A continuación, tuve que ampliar el importador de OBJ para que leyera coordenadas UV. También tuve que incluir estructuras de control y procesamiento adicionales en el código de la aplicación y del shader para distinguir las mallas susceptibles de incluir texturas (aquellas que incluían coordenadas UV en su especificación) de las que no. Por último, tuve que integrar una nueva librería en el proyecto para poder leer imágenes desde fichero (SDL2_image) y computar su conversión a una textura de OpenGL.

El siguiente fotograma se ha extraído de otro vídeo publicado en Youtube, titulado “Modelador 3D cooperativo - Jugando con cubos texturizados”[90]:

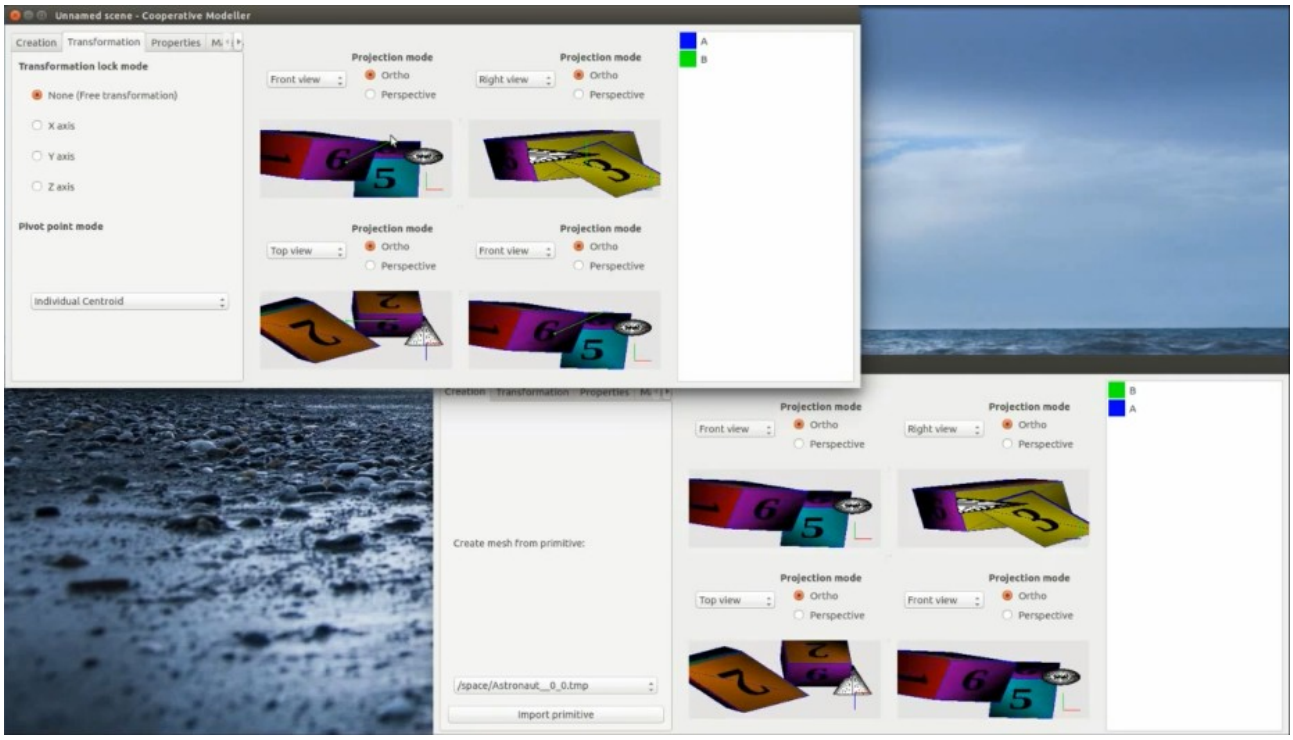


Ilustración 7.4: Cuarto incremento del proyecto COMO – importando primitivas desde el cliente + texturas

90 Modelador 3D cooperativo – jugando con cubos texturizados : https://www.youtube.com/watch?v=K4EI0jkYTXs

7.6. El quinto incremento: múltiples luces + mallas del sistema

Para el siguiente incremento me propuse permitir que los clientes conectados a una escena pudieran crear más de una luz direccional. Además, Agustín me propuso incluir las mallas del sistema: aquellas mallas que se generan a partir de parámetros definidos por el usuario y que permiten aplicar una textura diferente a cada una de sus paredes.

El sistema de múltiples luces (hasta un límite) requirió de cambios en el cliente y en el servidor. En el primero, se debería pedir permiso al servidor para poder crear la luz en cuestión. En el caso del servidor, éste debería comprobar que no se había alcanzado el máximo número de luces permitido, y responder al cliente en consonancia.

Para el caso de las mallas del sistema, se requirió la separación de una nueva clase *SystemMesh* que derivara de *Mesh*. También se creó una nueva estructura *SystemPrimitiveData*, y se definió una clase abstracta *SpecializedSystemPrimitivesFactory* a partir de la cual se derivaría una factoría para cada tipo de malla del sistema: *CubesFactory*, *ConesFactory*, *CylindersFactory* y *SpheresFactory*. Por último, se añadió una clase *SystemPrimitiveFactory* que actuara de interfaz entre las clases anteriores y la interfaz de usuario, además de encargarse de procesar los nuevos comandos incluidos para sincronizar la creación de las nuevas primitivas.

Con la inclusión de las mallas de sistema, ahora quedaba implementar la texturización individual de cada pared de las mismas. El primer paso consistió en crear un gestor de texturas (*TexturesManager*) para permitir a cada cliente importar sus propias texturas (incluyendo sincronización). El siguiente paso fue incluir la estructura de datos *TextureWall*, con atributos para definir su nombre, su desplazamiento y su escala. También se añadió el manejador TextureWallHandler para la manipulación de la “texture wall” por parte del cliente, así como la sincronización de los cambios realizados con el servidor. Por último, se integró las nuevas estructuras de datos con las mallas del sistema.

El siguiente fotograma pertenece a un vídeo en Youtube titulado “Modelador cooperativo COMO - Creando y editando cubos texturizables”[91]. En él se muestra la creación y manipulación de un cubo y las texturas aplicadas a cada cara.

91 Modelador cooperativo COMO - Creando y editando cubos texturizables : https://www.youtube.com/watch?v=41QuDwXVDww&feature=youtu.be

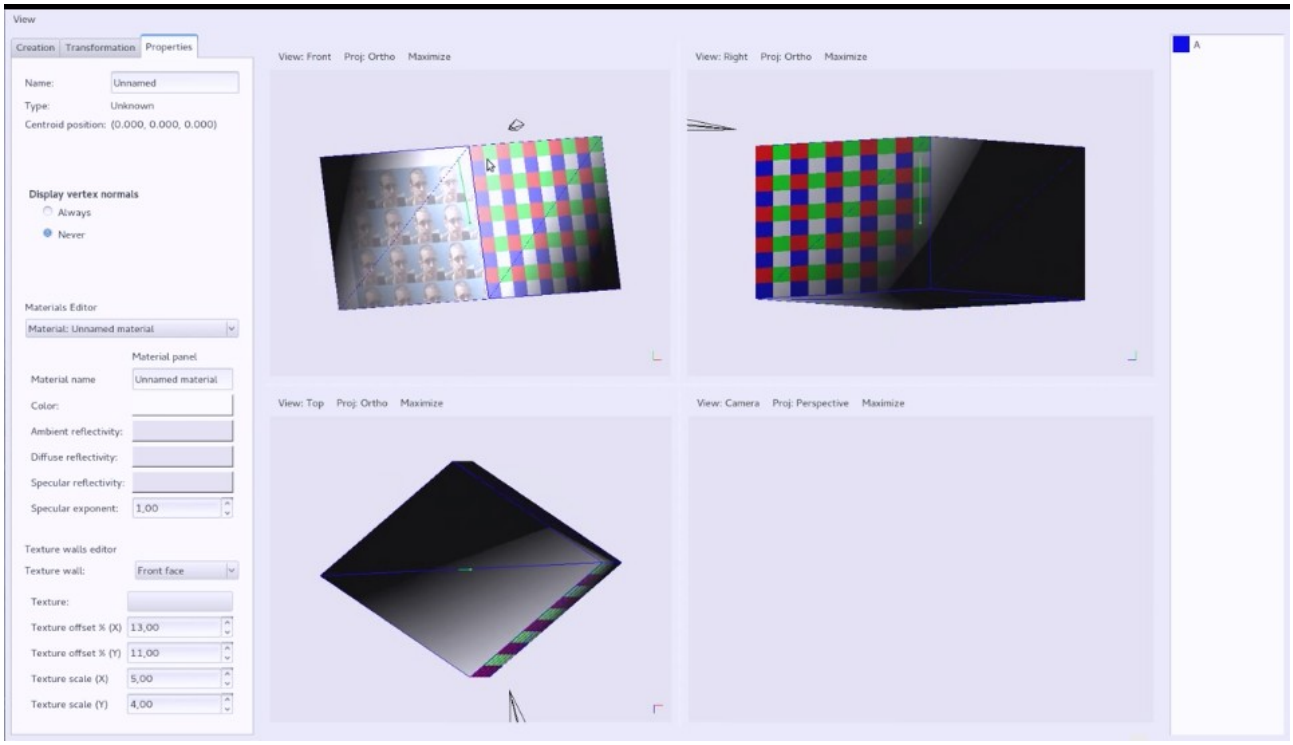


Ilustración 7.5: Quinto incremento del proyecto COMO – Aplicando y editando texturas en un cubo

7.7. El sexto incremento: cámara de la escena y carga y salvado de escenas

Para el sexto y último incremento, se trabajó en añadir una cámara para la escena 3D. También se implementó la carga y salvado de escenas desde fichero.

En un primer momento se pensó en realizar un sistema de cámaras más genérico, donde se permitiera a los clientes crear múltiples cámaras hasta llegar a un límite. Sin embargo, esta idea se desechó temporalmente, optando por **crear un sistema extensible, que en el futuro permitiera manipular múltiples cámaras, pero que actualmente permitiera una única cámara. Dicha cámara se sincronizaría entre todos los clientes y no podría ser borrada**.

El sistema de una cámara anterior se comenzó a implementar mediante la inclusión de un gestor de cámaras (*CamerasManager*) **en el cliente**. Este gestor contaría con dos particulares: en primer lugar, no dispondría de métodos para crear cámaras localmente; sólo procesaría los comandos que le llegaran desde el servidor. En segundo lugar, las llamadas al método de borrado, únicamente desbloquearían la cámara. **El servidor también se modificó para que incluyera una lista de *ResourceIDs* identificando elementos imborrables**. Cada vez que el servidor recibiera la orden de eliminar un recurso, cotejaría su ResourceID con los elementos de la lista anterior, optando por eliminar o desbloquear el recurso en cuestión según el caso.

Había llegado el momento de implementar el salvado y carga de escenas. En primer lugar estuve analizando los pros y los contras de implementar esta funcionalidad en el lado del cliente o del servidor. Si lo hubiera hecho desde el lado del cliente, éste ya contaba con toda la información

sobre cada recurso de la escena, por lo que sólo restaba escribirlos o leerlos en fichero. Sin embargo, ¿qué ocurría si ese cliente se desconectaba? ¿qué sucedía si ese cliente "no era de fiar" y corrompía el fichero de salvado de la escena?. Además, igualmente el cliente debería sincronizar el fichero de salvado en el servidor. **Por estas y otras cuestiones, decidí que lo correcto era implementar la carga y el salvado de escenas en el lado del servidor, aún a costa de añadir más responsabilidades al mismo**, más allá de la sincronización de comandos entre los clientes.

Con la decisión anterior tomada, **me planteé múltiples alternativas para su implementación**. La alternativa más directa era escribir el histórico de comandos directamente en el fichero: demasiado ineficiente. Una mejora al método anterior consistía en eliminar los comandos obsoletos (por ejemplo, eliminando todos los comandos que modificaban el color de un material salvo el último). Ciertamente era más eficiente que escribir todo el histórico, pero aún no me convencía. Entonces pensé en que fuera un cliente el que salvara la escena previa petición del servidor, ¿pero qué sucedía si el usuario cliente cerraba la aplicación antes de terminar el salvado?. **Evalué alternativas hasta llegar a la que implementé finalmente: salvar la información mínima y necesaria para generar el conjunto de comandos mínimo que replicara cada recurso de la escena**.

Con el método de salvado y carga de escenas decidido, comencé su implementación. Empecé por migrar el servidor de un sistema que únicamente asociaba un dueño a cada recurso a un sistema en el que se asociaba un objeto *ResourceSyncData* a cada recurso. **Continué generando nuevas clases derivadas de la anterior según el tipo de recurso, y testeando el salvado y carga de la escena con cada cambio hasta llegar a la versión final**.

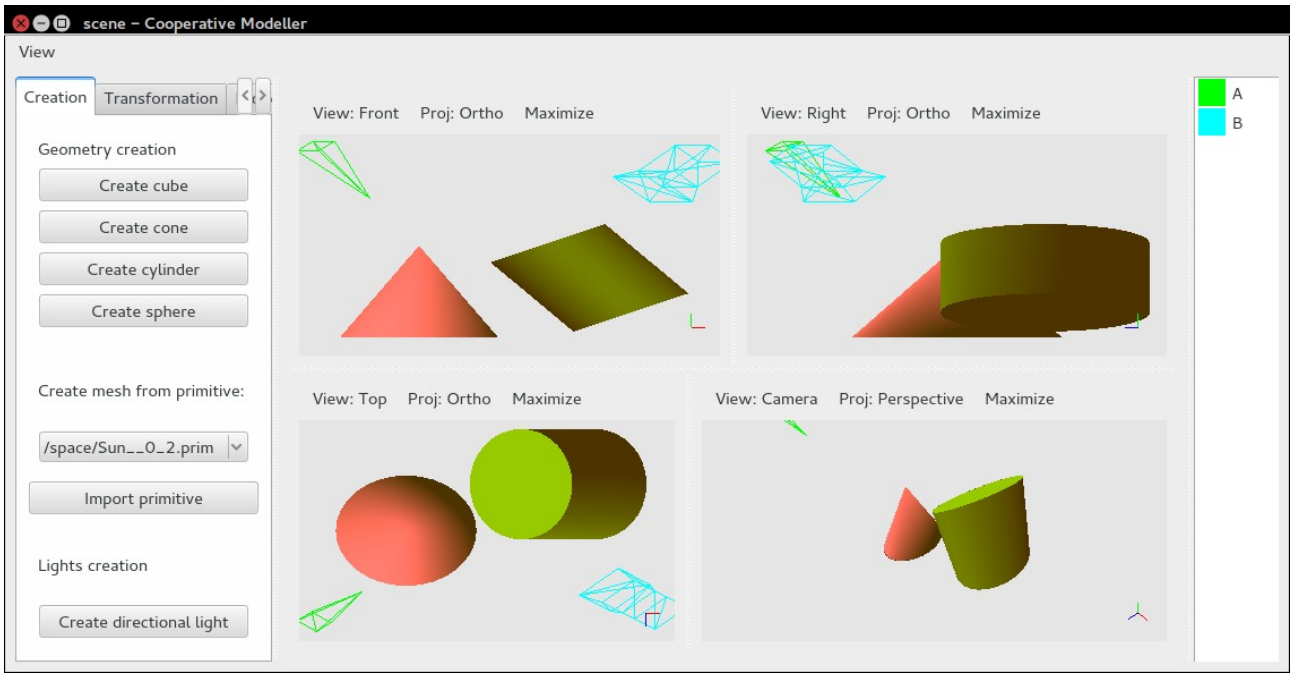


Ilustración 7.6: Sexto incremento de COMO – Cámara de la escena y carga y salvado de escenas

8. Conclusión

La creación de un visor modelador 3D colaborativo ha sido una experiencia que me ha aportado múltiples ventajas en diversos campos del desarrollo software.

En primer lugar, me ha permitido ganar una gran experiencia en la computación gráfica usando una versión moderna de la API de OpenGL. Conceptos como VAO, VBO, shader o sampler me eran desconocidos, o al menos extraños, antes de iniciar este proyecto.

Además, también ha servido para introducirme de lleno en las bondades y los desafíos de la programación distribuida: conexión y sincronización de máquinas, identificación de recursos, transmisión de estructuras de datos complejas, etc.

El desarrollo del proyecto COMO también me ha servido para reforzar mi uso de la ingeniería del software: planificación, uso de modelos UML, refactorizaciones, etc.

Por último, esta odisea también ha servido para seguir ganando experiencia en el control de versiones con Git o la realización de documentos con LibreOffice, entre otros.

9. Trabajo futuro

Como cualquier proyecto software, COMO admite bastantes correcciones y mejoras. A continuación se listan algunas mejoras propuestas.

9.1. Optimizar la sincronización del histórico de comandos

La sincronización de la escena realizada en el servidor gira en torno al histórico de comandos: una lista enlazada en la que se van insertando todos los comandos ejecutados sobre la escena. Esta lista no tarda en crecer y alcanzar los miles de comandos. **Además, cada vez que un usuario se conecta a la escena, éste comienza a sincronizar el histórico de comandos entero, desde el primer comando**. Por esta razón, el histórico de comandos actual es un gran consumidor de memoria y de procesamiento.

La mejora propuesta para atajar este problema pasa por dividir este histórico de comandos monolítico en n colas de comandos. Cada cola se asociaría a un cliente, y contendría los comandos pendientes de sincronizar por el mismo, eliminando cada comando a medida que se transmite al usuario. Cuando un usuario se conectara a la escena, el servidor generaría el conjunto de comandos mínimo para reproducir la escena[92] y lo añadiría a la cola de comandos del usuario.

9.2. Reciclaje de IDs

El uso de IDs autoincrementadas para los usuarios y los recursos podría conducir al improbable caso en el que un servidor se quede sin IDs libres. **Debido a la improbabilidad de que esto sucediera a corto plazo, se omitió su resolución por el momento.**

En futuras versiones se podría implementar algún tipo de reciclaje de IDs: guardar aquellos IDs pertenecientes a usuarios o recursos que ya no existen para asignárselos a los nuevos.

9.3. Feedback al cliente sobre el estado de la sincronización

Actualmente el usuario del software cliente no tiene ningún indicativo visual del estado de sincronización de la escena con respecto al servidor, más allá de las vistas 3D. **La manera más sencilla de corregir esto sería mostrar en la interfaz el campo "*nUnsycCommands*" que viaja con cada paquete *SceneUpdatePacket* enviado desde el servidor**. Dicho campo el número de comandos por sincronizar con el servidor.

9.4. Fragmentación de paquetes y comandos

En esta versión de COMO, los paquetes y comandos se transmiten enteros a través de la red. Esto significa que el receptor no comienza a procesar los paquetes hasta que los recibe por completo. **La división de los paquetes y sus comandos en fragmentos para su transmisión y procesamiento individuales podría aumentar la eficiencia de la aplicación**, especialmente en el caso de la creación de primitivas, donde el receptor espera a recibir el fichero de especificación

92 La generación del "conjunto mínimo de comandos" se realizaría de manera análoga a cómo se realiza cuando se pretende salvar la escena en fichero.

completo antes de procesarlo.

10. Anexo A: Manual de usuario

El manual de usuario de COMO se encuentra alojado Github como una wiki del repositorio de COMO[93,94]. En las siguientes páginas se muestra una copia del manual online.

10.1. COMO (Cooperative Modeller)

10.1.1. About COMO

COMO is a cooperative 3D modeller, meaning that multiple users can connect and work on the same scene at the same time.

10.1.2. Features

COMO allows users to:

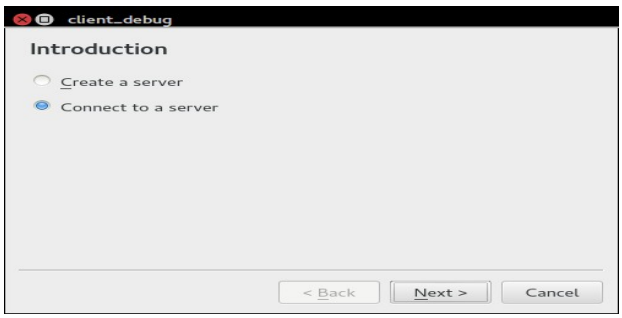
- create a shared scene or connect to a created one.
- load meshes from files and add them to the scene (only OBJ files).
- create cubes, cones, cylinders and spheres.
- create directional lights until the limit defined by the server is reached.
- select and transform multiple entities (meshes, lights and cameras).
    - Allowed transformations are translations, rotations and scales.
    - Incorporates arbitrary and axis related (X, Y or Z) transformations.
    - The pivot point for rotations and scales can be one of three: world origin, selection's centroid or individual centroids.
- remove entities from the scene (excepting the scene's camera).
- edit the parameters of the materials associated to the meshes:
    - Color
    - Ambient reflectivity
    - Diffuse reflectivity
    - Specular reflectivity
    - Specular exponent
- load textures and apply them to walls of cubes, cones, cylinders and spheres. For every wall and its texture, user can set the texture's offset and scale over the wall.
- save and load scenes.

10.1.3. Initializing COMO

When COMO's client program is started, the following connection wizard is presented to the user:

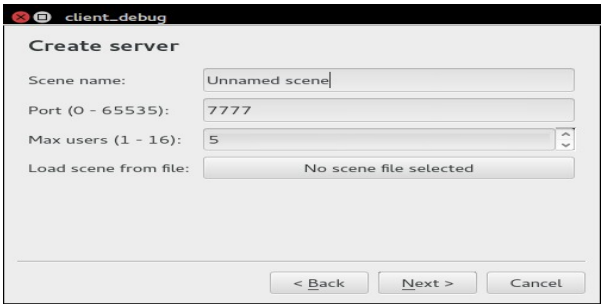
93 Repositorio de COMO en Github: https://github.com/moisesjbc/como

94 Wiki de COMO en GIthub: https://github.com/moisesjbc/como/wiki



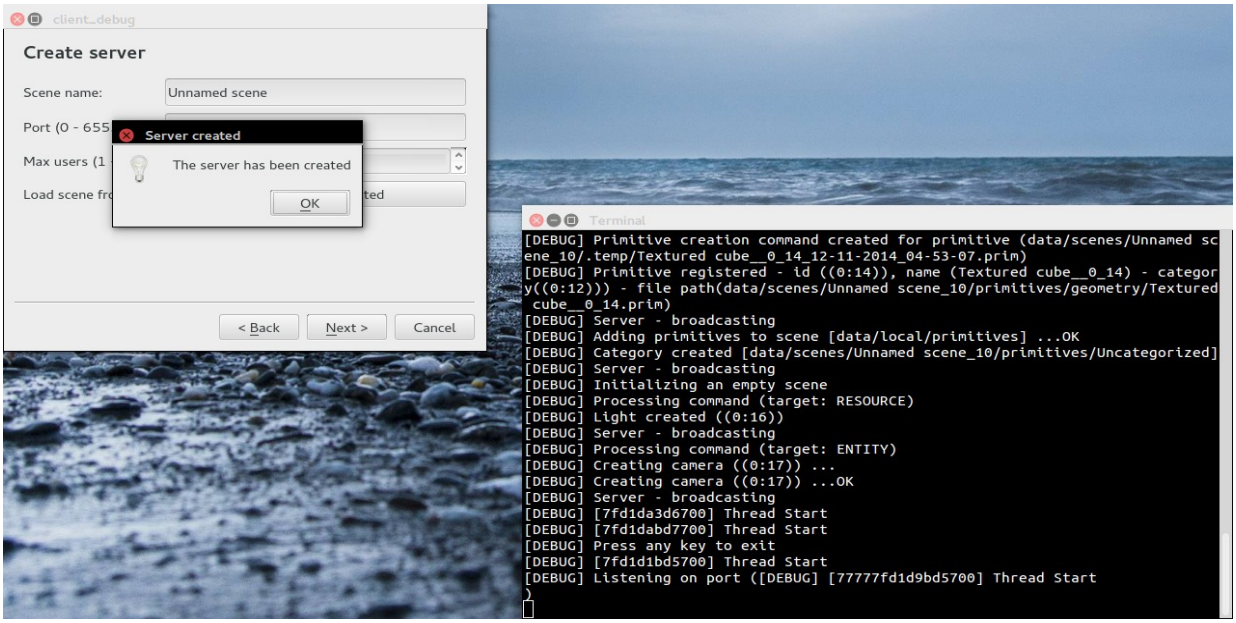
10.1.4. Creating a scene

For creating a scene, select "Create a server" on previous screen and then press the "Next" button. COMO will then display the next form for creating a server:



1. **Scene name:** Name given to the scene.
2. **Port:** The port used that upcoming users will have to connect to in order to share the scene. By default, 7777 port will be used.
3. **Max uses:** Limits the amount of users allowed to connect and concurrently work on the scene.
4. **Load scene from file:** Allows user to load the scene from a previous saved scene file. Leave this field blank for a new empty scene.

Once the previous fields are filled, press the "Next" button to create a server. A new terminal with the server will open:



This new terminal will be displaying a lot of log messages describing what the server is doing. **This window must not be closed until you want to shutdown the server**. **Also, be careful with pressing any button while the focus is on this windows, because it will shutdown the server as well**

So, the server is already open and running! Press "OK" in the dialog with the message "The server has been created" to proceed.

10.1.5. Connecting to a created scene

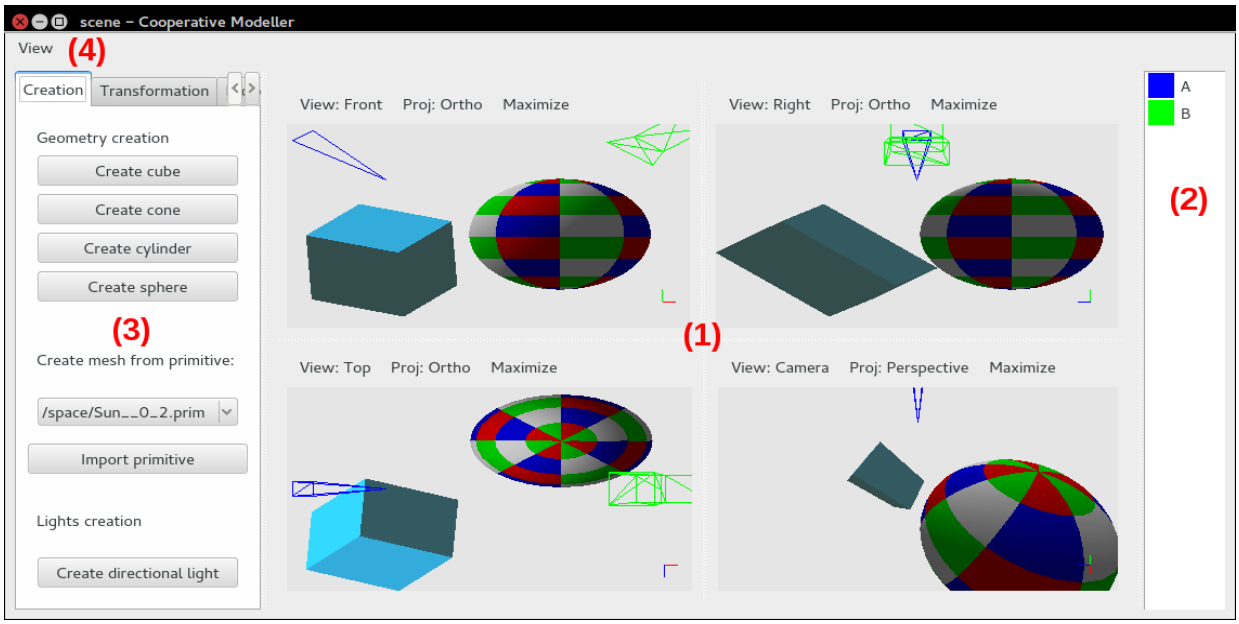
The connection page of COMO is shown below:

1. **IP:** The IP of the server you want to connect. If you created the server in the same machine from where you are trying to connect, leave the default "127.0.0.0" value.
2. **Port:** Server's port.
3. **User name:** Your name / nick, used to uniquely identify you in the scene.

Once you have filled the previous fields, press on the "Finish" button. If there wans't any error, you will be connected to the scene and COMO's main window will be displayed. See the GUI overview page to continue.

10.2. GUI overview

The following screenshot shows the main COMO window:



1. **Render panel:** Panel with 4 viewports displaying the current 3D shared scene. User can change the view and projection of each viewport. Viewport can be also resized, or maximized to use the entire render panel.
2. **Users list:** All the users connected to the scene are displayed here. Next to each user's name is their selection color, used when drawing the edges of all the meshes, lights and cameras selected by that user.
3. **Tools panel:** Panel organized by tabs with controls for creating and editing the resources of the scene: meshes, lights, materials, etc.
4. **View menu:** Here the user can edit some parameters which affect how the scene is displayed locally (it doesn't affect other users sharing the scene). For example, this menu includes an option for setting whether unselected meshes are drawn with or without edges.

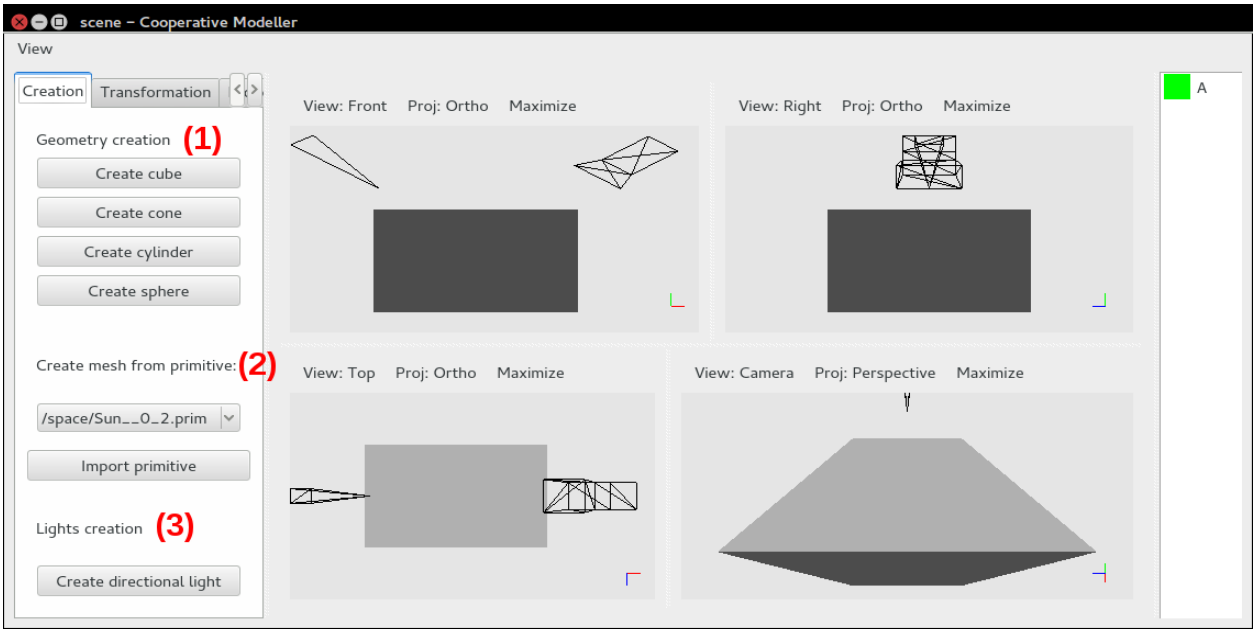
10.3. Working with entities

10.3.1. Defining an entity

An entity in COMO is a resource which can be manipulated directly from any of the viewports in the render panel. All the Meshes and lights, as well as the scene's camera are entities.

10.3.2. Adding entities to the scene

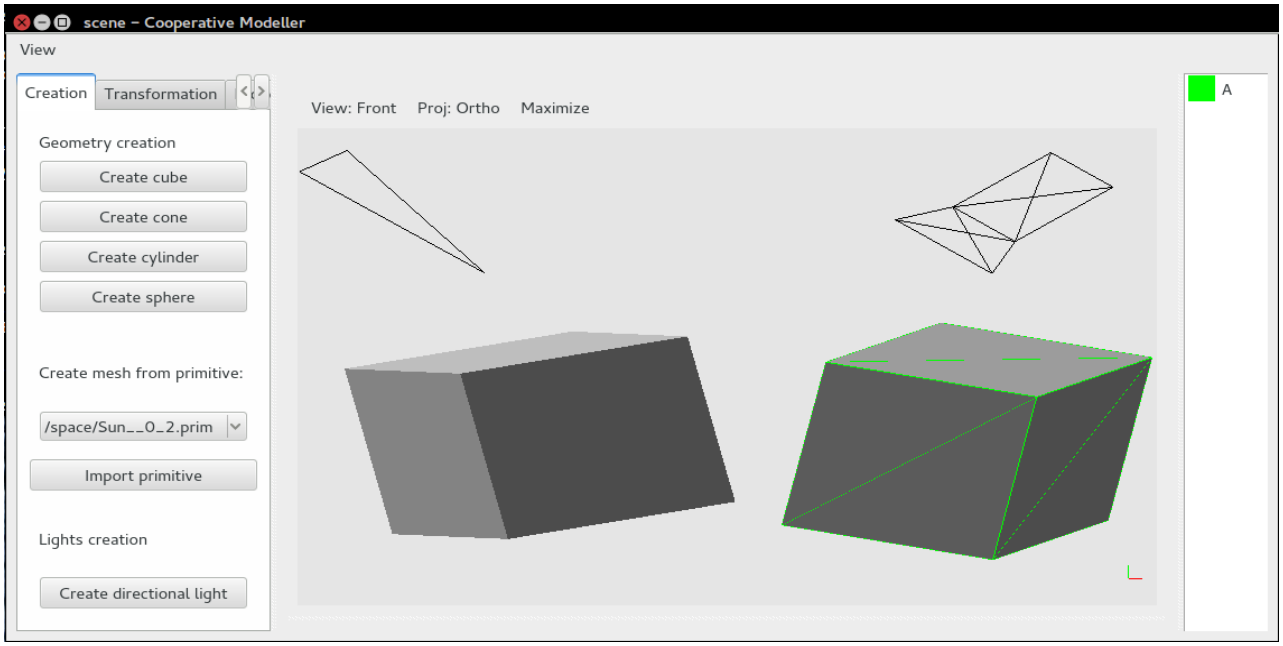
Entities are added to the scene from the "Creation" tab of the tools panel (right)



1. **Geometry creation:** allows user to create geometric shapes such as cubes, cones, cylinders and spheres and add them to the scene.
2. **Create mesh from primitive:** from here, user can load primitives (mesh specification) from OBJ files by pressing the "Import mesh" button. All the imported primitives are added to the dropdown list above the "Import mesh" button. By selecting one of the primitives in this list, a mesh copy will be created and added to the scene.
3. **Lights creation:** you can **request** the creation of a directional light in the scene by pressing the "Create directional light" button. The server will process your request and, if accepted, a directional light will be created in the scene.

10.3.3. Selecting and unselecting entities

Entities can be selected by clicking (mouse's left button) on them in any of the viewports present in COMO. When selecting an entity this way, all your previously selected entities will be unselected. If you want to keep your current selection while adding a new entity to it, hold the "ctrl" button while selecting.



For unselecting your current selection, simply click on the scene's background.

IMPORTANT: When any user in the scene selects an entity, **such entity is locked. This means that no one else can select that entity** until the user who locked it unlocks it or disconnects from server.

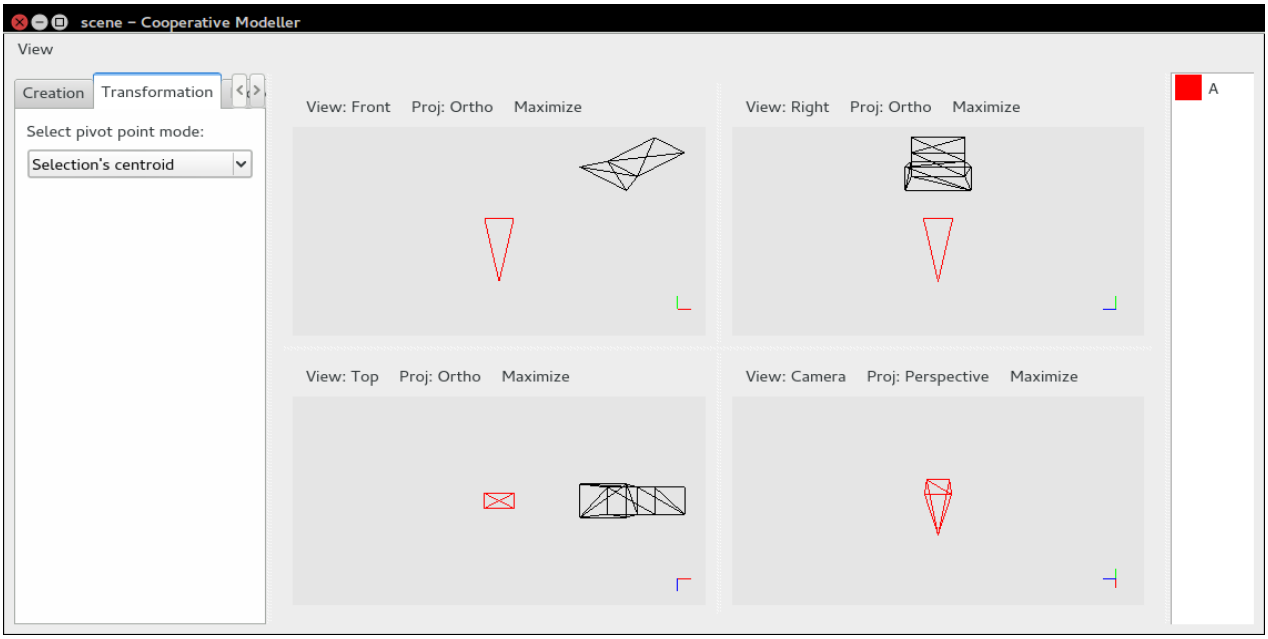
10.3.4. Transforming entities

Right after selecting one or more entities on a viewport, you can perform the following transformations:

- **Translate:** Press "t" on your keyboard and drag your mouse over the viewport for translating your entities selection. If you press "x", "y" or "z" while translating, the translation will be fixed to the X, Y or Z axis, respectively.
- **Rotate:** Press "r" on your keyboard and drag your mouse over the viewport for rotating your entities selection. If you press "x", "y" or "z" while rotating, the rotation will be fixed to the X, Y or Z axis, respectively.
- **Scale:** Press "s" on your keyboard and drag your mouse over the viewport for scaling your entities selection. If you press "x", "y" or "z" while scaling, the scale will be fixed to the X, Y or Z axis, respectively.

10.3.4.1. Changing the pivot point mode

By default, the rotations and scales of selections will be done with the selection's centroid as the pivot point. You can change the current pivot point by selecting the "Transformation" tab on your tools panel and selecting a new pivot point mode from the "Select pivot point mode" dropdown list.

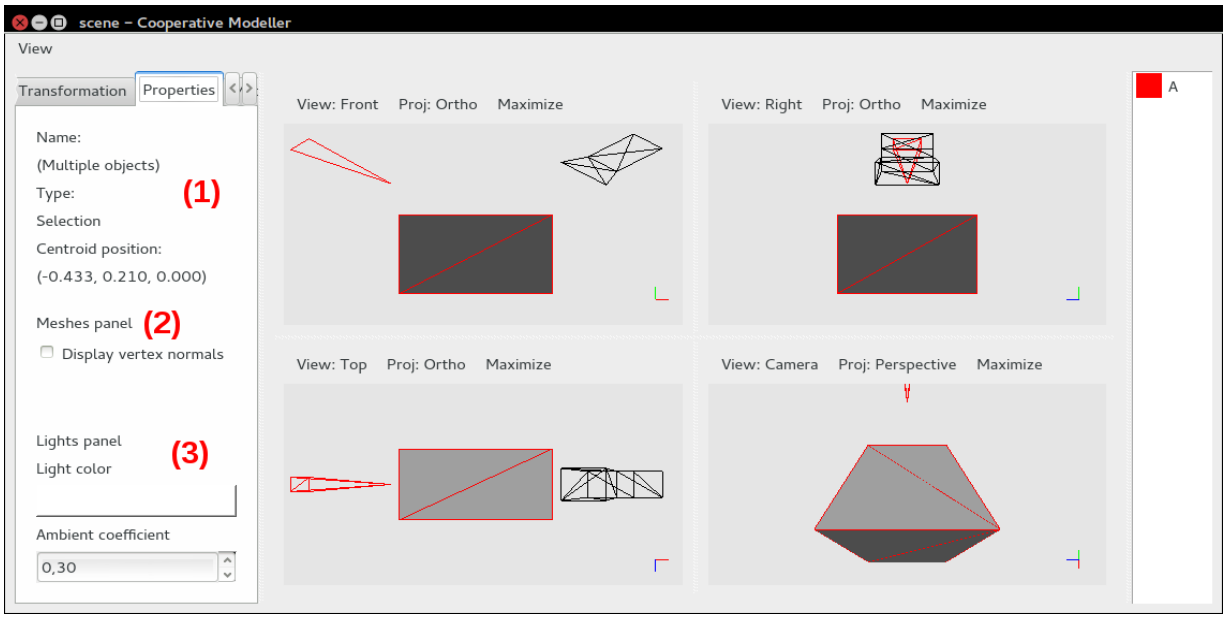


The selectable pivot point modes are listed below:

- **Selection's centroid:** The default mode. All the entities in the selection will be rotated or scaled around the selection's centroid.
- **Individual centroids:** Each entity in the selection will be rotated or scaled around its own centroid.
- **World origin:** Each entity in the selection will be rotated or scaled around the world's origin or (0, 0, 0).

10.3.5. The properties panel

If you go to the "Properties" tab on the tools panel you will see the Properties Panel. Here you will see a variable number of panels depending on what types of entities are in your current selection.

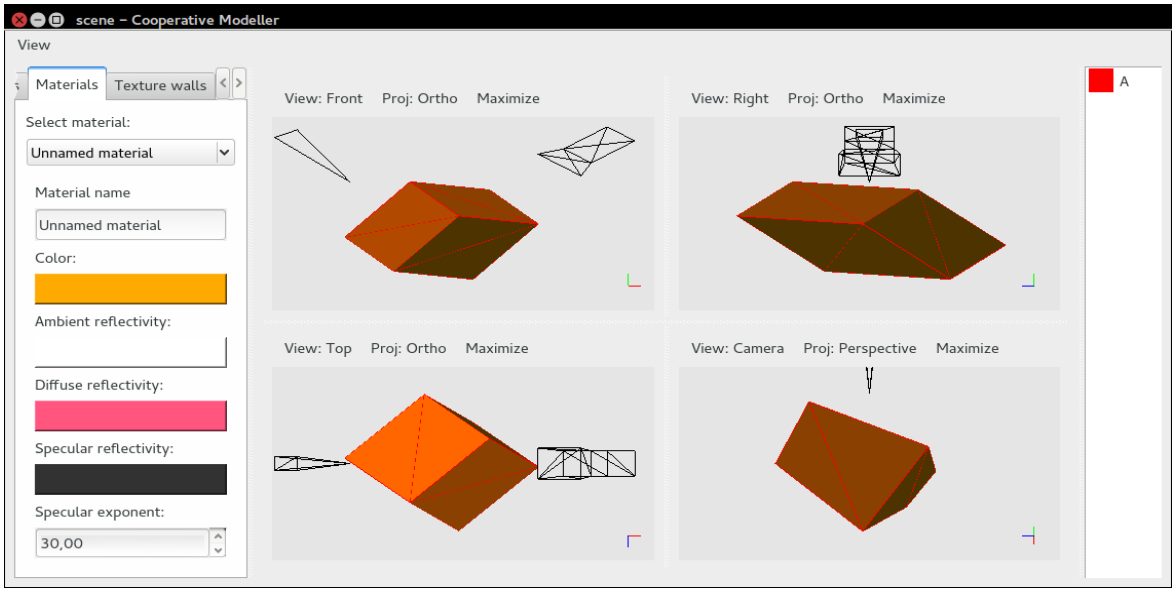


1. **General info:** displays general **read-only** data about the current selection of entities, such as its name, its type and its centroid.
2. **Meshes info (only if there are meshes selected):** allows you to set parameters affecting the meshes in your selection. Currently it only allows you to select whether to display the vertex normals of your selected meshes or not.
3. **Lights info (only if there are lights selected):** allows you to set the color and the ambient coefficient of your selected lights.

10.4. The materials editor

The materials editor allows to edit the materials **associated with the current user mesh selection**. This means that, if there are two meshes, "A" and "B", in the scene and the user selected mesh "A", the materials editor will only edit mesh "A" materials. The user must select mesh "B" in order to edit its materials.

The materials editor can be accessed using the "Materials" tab in the tools panel.

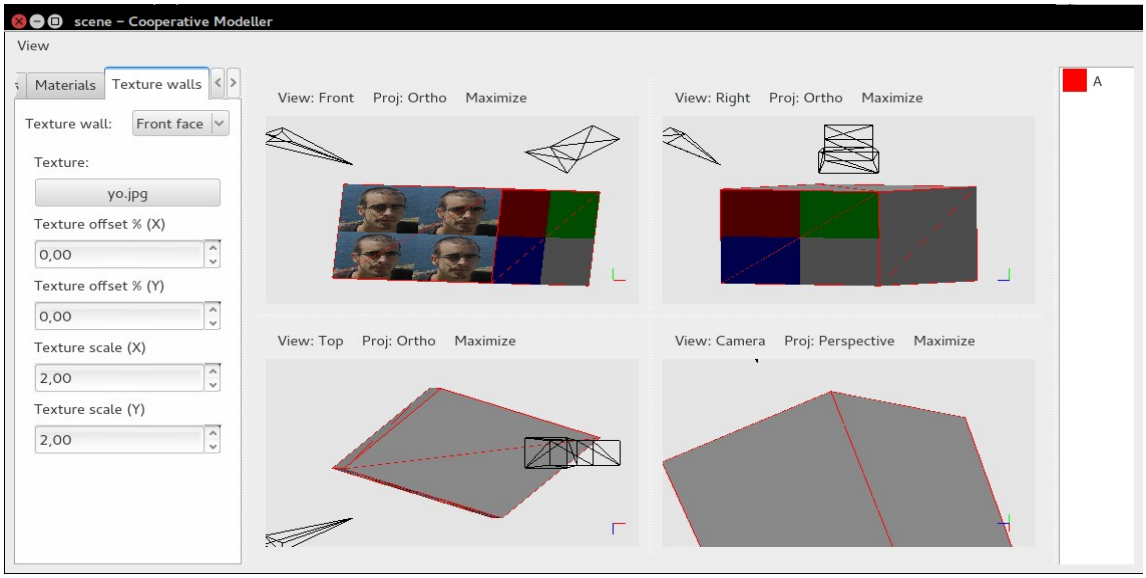


Only one material may be edited at a time. Select the material to edit from the "Select material" dropdown list, at the top of the materials editor. The following buttons and controls can be used to edit the properties of the currently selected material: color, ambient reflectivity, etc.

10.5. The texture walls editor

10.5.1. Introduction

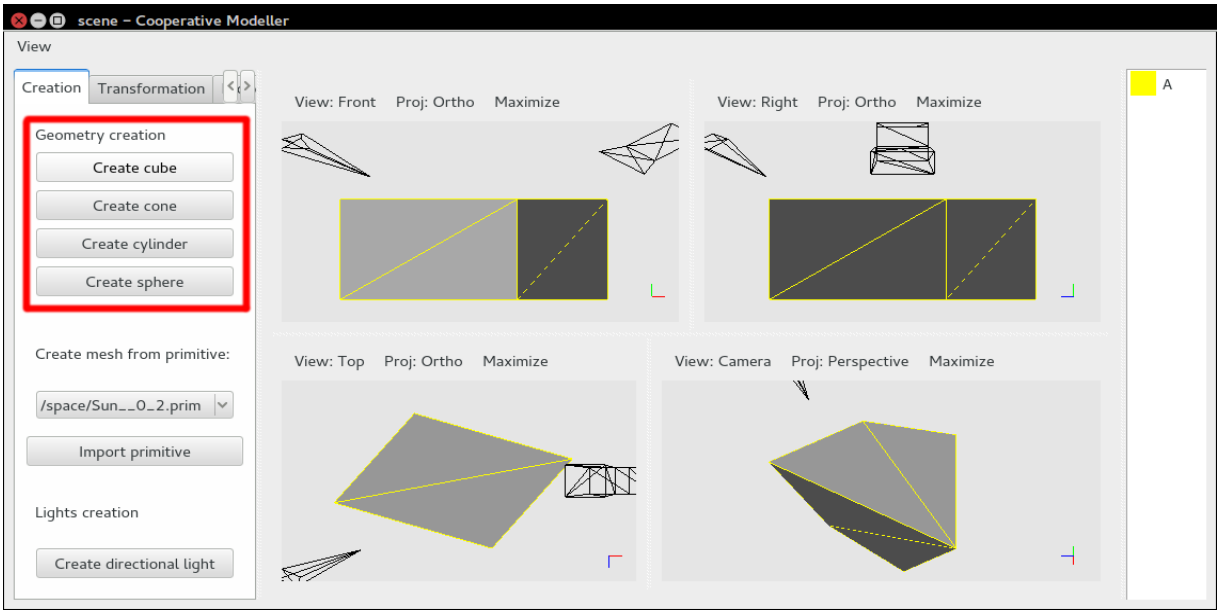
The texture walls editor allows you to load images from disk and apply them as textures to the walls of the geometric meshes (*) present in the scene. When applying a texture to a wall, you can set the offset and the scale of the texture over the wall.



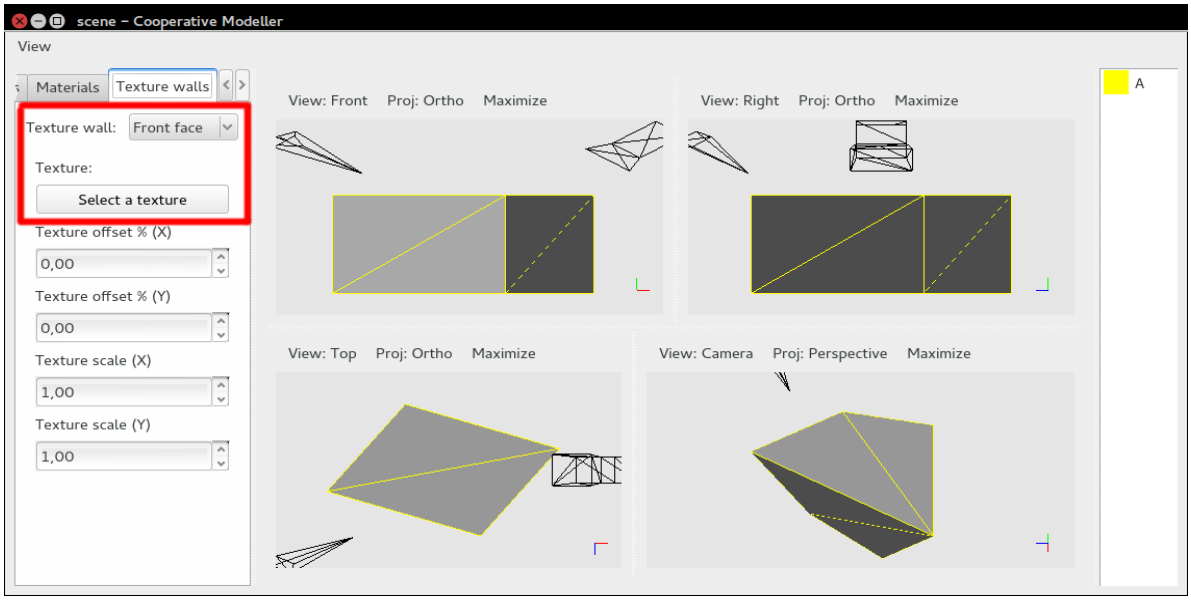
(*) The geometric meshes are the cubes, cones, cylinders and spheres created using their corresponding buttons in the "Geometry creation" panel, under the "Creation" tab in tools panel (see Working with entities).

10.5.2. Applying a texture to a geometry's wall (face)

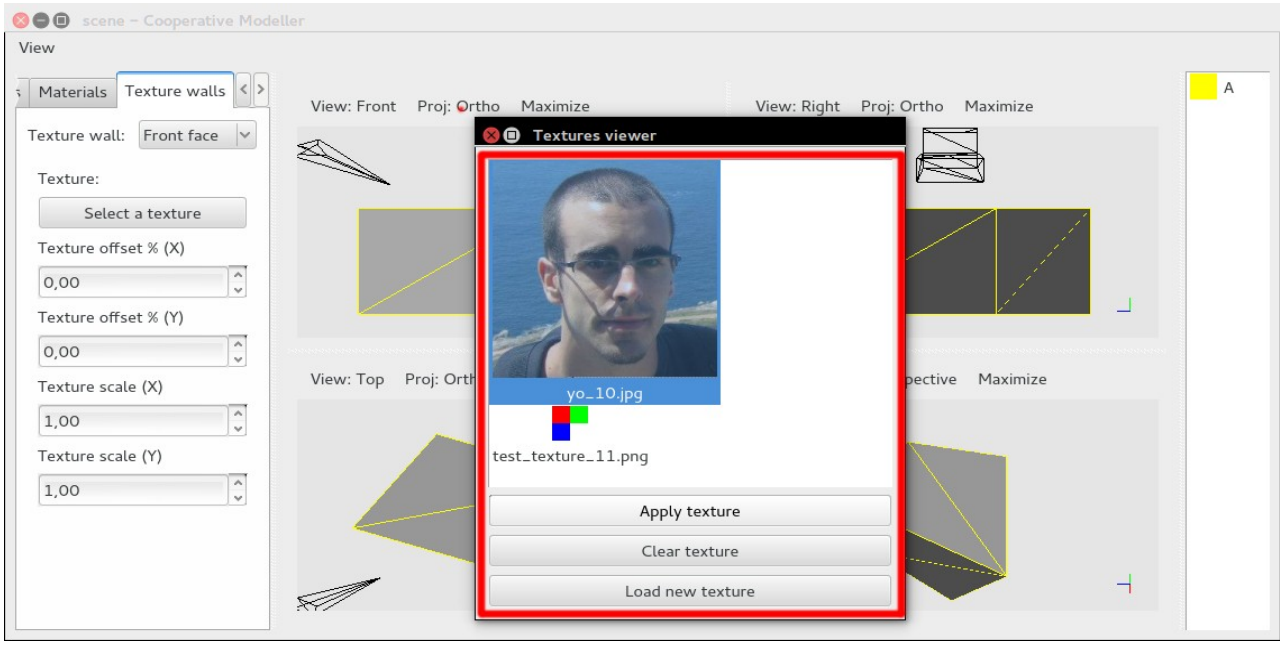
The process of applying a texture to a geometry's wall (fase) starts with the **creation of a geometric primitive** from the "Geometry creation" panel ("Creation" tab - Tools panel). In this example we created a cube:



Next, **with the cube selected**, we go to the tools panel and select the tab "Texture walls" for displaying the texture walls editor. At the top of this editor we can **select the geometry's face we want to texturize**. For example, select the "front face" of the cube:

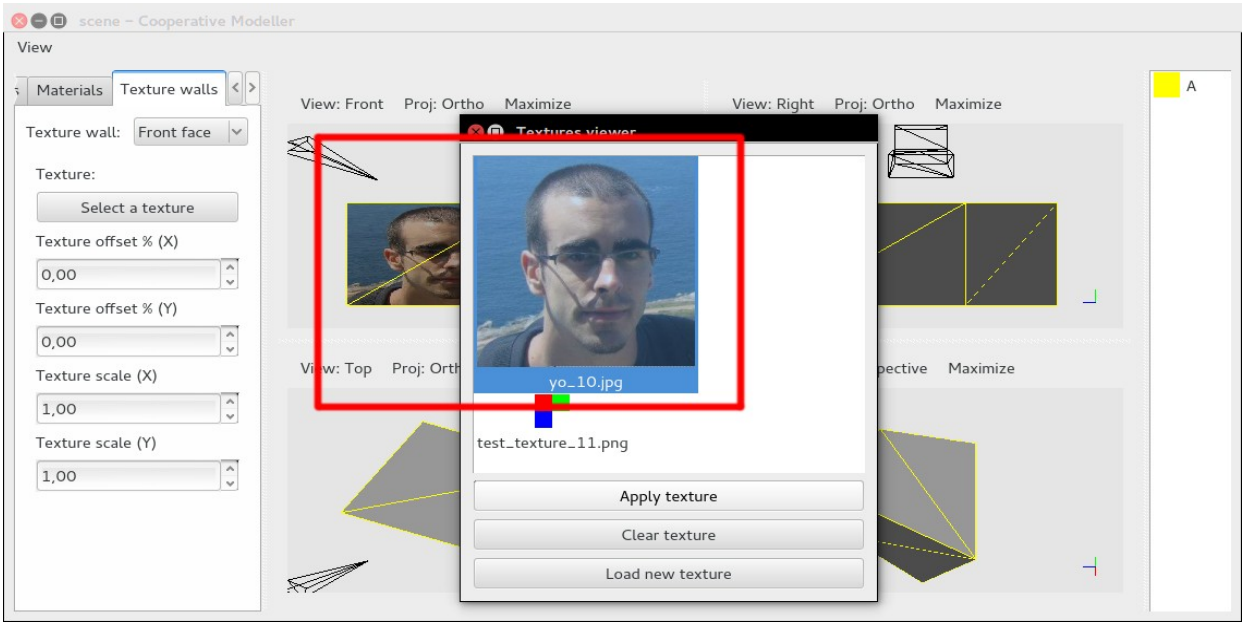


Once the wall you want to texturize is selected, press the button "Select a texture" for opening a new window called Textures Viewer. Here you will see the textures already loaded in COMO, as well as a button for loading new ones.

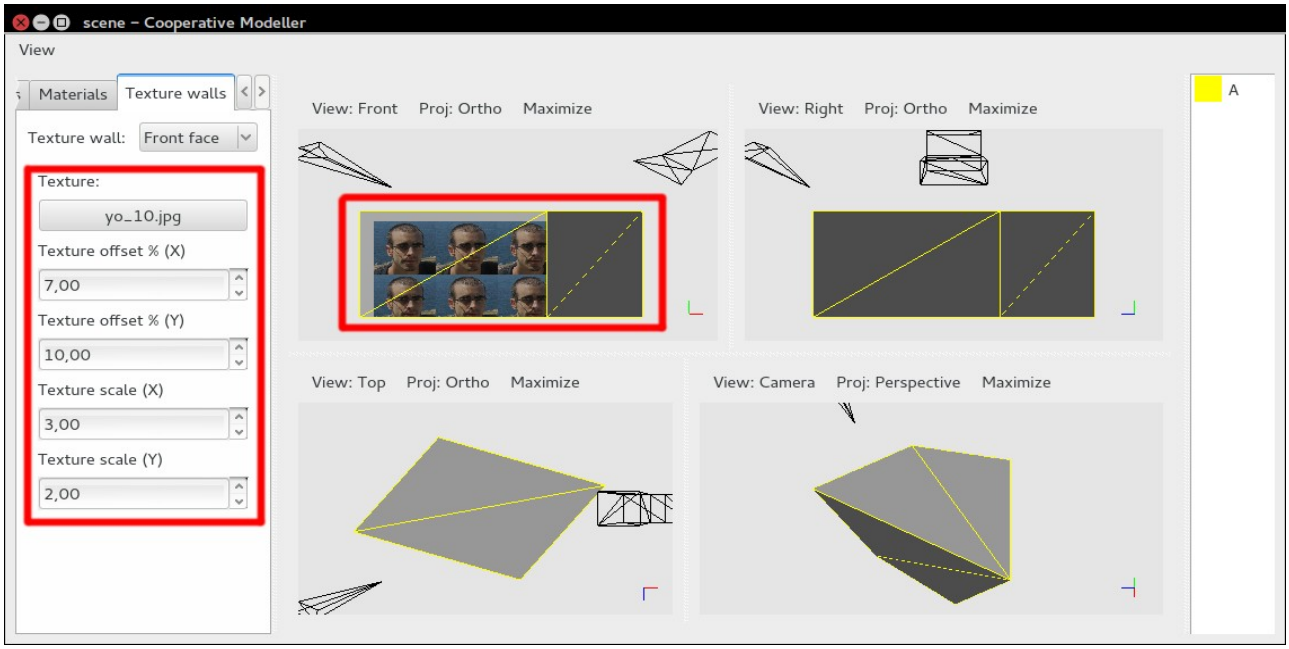


Clic on the texture you want to use from the textures gallery (load it before it you haven't done it yet) and press the "Apply texture" button. The texture will be instantly applied to the geometry's face.

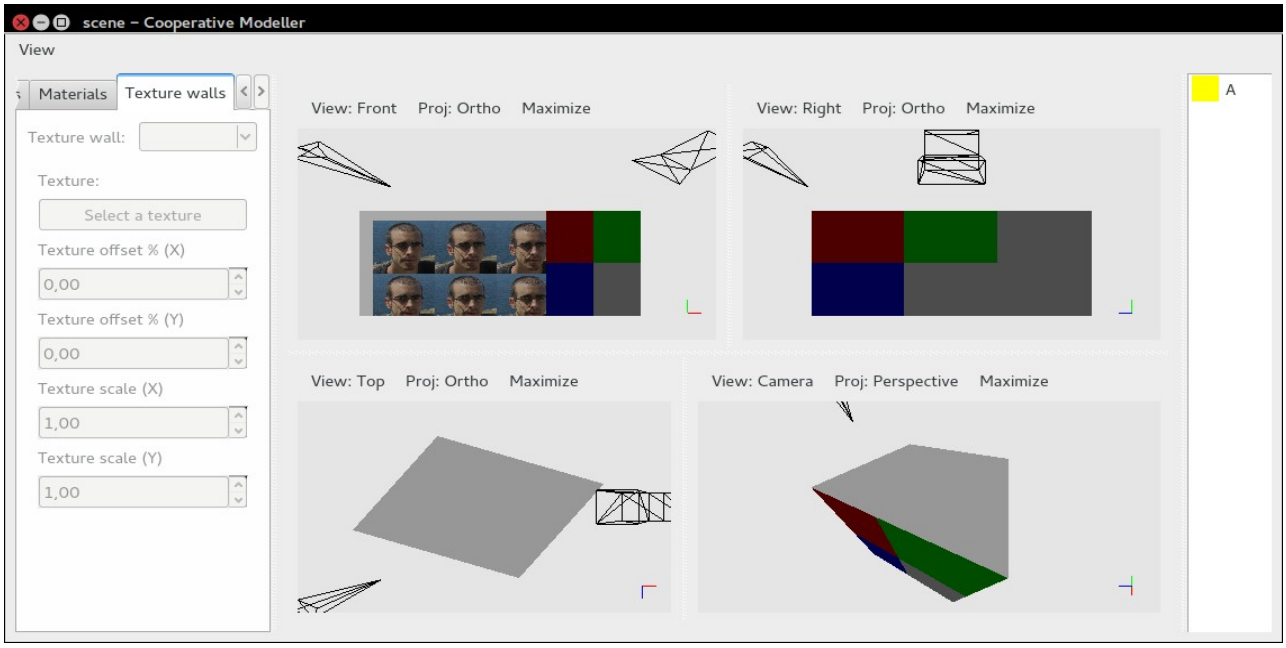
NOTE: At any time you can repeat these steps to replace the texture applied to the geometry's face, or remove it by clicking on the button "Clear texture".



Close the Textures Viewer window to return to the COMO's main window. With a texture applied to a geometry's face, you can use the Texture Walls Editor for setting the texture's offset and scale over its face.

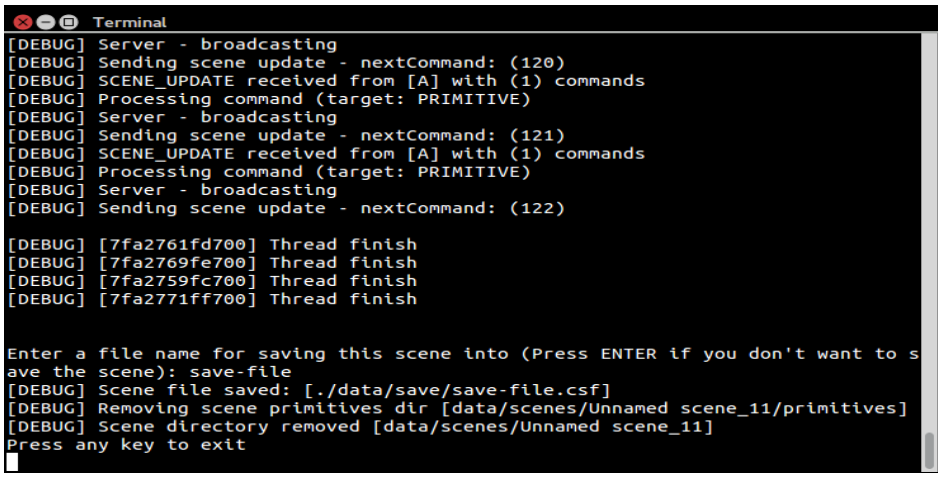


And that's it! You can repeat previous steps for changing the current face's texture, or for applying a texture to another face of the geometry.



10.6. Closing the server and saving the scene

In this version of COMO, the scene isn't saved until the server is closed. In order to close the server, go to its window and press any key. Then follow the instructions displayed on screen to save the scene to a file.



NOTE: In case of abrupt sever closure all unsaved work will be lost! D: