

Universidad ORT Uruguay
Facultad de Ingeniería

Diseño de aplicaciones 1
Primer obligatorio

https://github.com/ORT-DA1-2023/271568_275898

Alfredo Fernández - 275898
Manuel Morandi - 271568

Tutores:
Gastón Mousqués
Facundo Arancet
Franco Galeano

2023

ÍNDICE

ÍNDICE	1
Descripción general	2
Aclaraciones del diseño	2
Diseño de dominio	3
Diseño de tests	4
Diseño de interfaz de usuario	5
Manejo de errores	6
Herencia y polimorfismo	7
Clean Code	8
Interacción entre clases	9
Funcionamiento	12
Cobertura de pruebas	13
Reflexiones	14
Anexo	15

Descripción general

El objetivo de la tarea es desarrollar e implementar un renderizador 3D, utilizando el algoritmo de Ray Tracing.

Se busca incorporar en el trabajo los conceptos vistos en clase. Debido a esto, el producto entregado debe ser un sistema prolijo, legible y mantenible, aplicando reglas de Clean Code. La metodología de trabajo debe ser la llamada TDD (test driven development), lo que involucra escribir las pruebas antes de implementar el código de producción en cuestión. Posteriormente, se debe refactorizar el código, asegurándonos de que sea sencillo de entender y modificar.

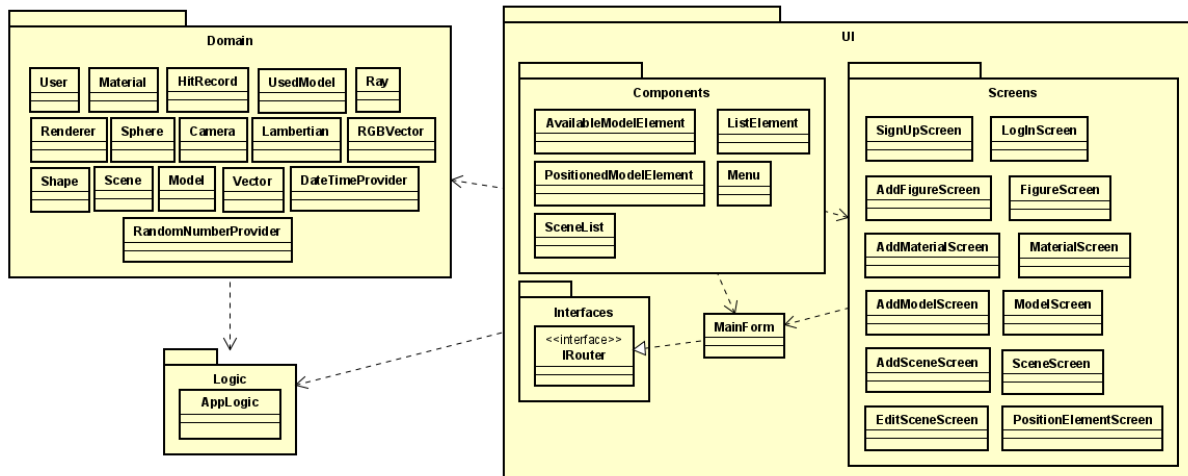
El sistema, que es implementado en C#, debe permitir crear cuentas para que sean usadas por los clientes. Estos usuarios posteriormente pueden crear figuras (esferas en primera instancia) y materiales (lambertianos). Al asignarle un material a una figura, el usuario obtiene un modelo, que luego puede ser colocado en escenas. Estas escenas son creadas y modificadas por el usuario, para luego renderizarlas y visualizar el resultado en la propia aplicación.

Aclaraciones del diseño

A la hora de trabajar se tomó la decisión de comenzar definiendo el dominio, implementando todas las clases necesarias para el desarrollo de la aplicación. Una vez programadas, se siguió con todo lo relacionado al renderizador y el algoritmo de Ray Tracing. En una última fase, se desarrolló de manera concurrente la interfaz de usuario y la lógica de la aplicación, o sea, toda la funcionalidad.

Las clases se dividen en 4 paquetes distintos. Uno (UI) está destinado a todo aquello relacionado con la interfaz de usuario. Otro (Domain) busca hacer posible el proceso de renderizado, teniendo todas las clases y algoritmos pertinentes. El de Logic se ocupa del enlace entre la capa de dominio y la interfaz, además de almacenar todos los objetos guardados por el usuario. Finalmente, el paquete de pruebas se ocupa de comprobar el correcto funcionamiento de dominio y lógica.

A continuación se encuentra un diagrama de paquetes para comprender mejor la división (se obvia el paquete de tests):



El paquete de UI está también dividido en distintas carpetas, teniendo una para pantallas, otra para componentes y una para la interfaz de ruteo (IRouter).

AppLogic es el nexo cohesivo que une frontend y backend. Toda comunicación entre dominio e interfaz debería hacerse a través de él, por lo que tanto UI como Domain utilizan Logic. UI también se comunica directamente con Domain, incumpliendo así la ley de Demeter, cosa que se buscará solventar para el segundo obligatorio.

Centrándonos en UI, vemos que MainForm implementa IRouter para permitirnos movernos entre las pantallas. De ahí, notamos como tanto Components como Screens utilizan MainForm, para así tener acceso a las funciones de ruteo y al objeto de la clase AppLogic que es usado para administrar todos los elementos de la aplicación y como intermediario para comunicarse con el backend.

Diseño de dominio

En el proyecto de dominio se encuentran las clases de todos los elementos que forman la aplicación, tales como las figuras, los materiales, los modelos, etc. Adicionalmente, también encontramos todas las clases relacionadas directamente con el renderizador: la clase Renderer, HitRecord, Camera, entre otras. Finalmente, en un paquete aparte pero que, por simplicidad de la explicación, consideraremos parte del dominio, encontramos la clase destinada a la lógica de aplicación.

Con respecto a la lógica de la aplicación, pensamos que sería conveniente ponerla en un namespace distinto, pese a ubicarse en el proyecto de dominio, por lo que se encuentra sola en un namespace llamado Logic. Esto tiene como objetivo que la UI utilice solamente lo que nosotros queremos que use (la lógica) y no pueda llegar a tocar nada que no tenga que tocar del dominio, salvo que nosotros se lo indiquemos. Esto nos permite tener más control sobre qué clases se comunican con cuales. De esta manera, podemos evitar de manera más sencilla incumplir la Ley de Demeter, que establece que una clase debe ocultar su estructura interna. Al poner la lógica en un namespace aparte e importar únicamente ese namespace en los elementos de la UI, nos aseguramos que esta no acceda directamente a las clases

del dominio y que pueda interactuar con ellas solamente a través de la clase AppLogic. Cabe destacar, sin embargo, que en ciertos casos fue necesario, o conveniente, usar el resto de las clases del dominio desde la UI, como ya se mencionó. Sabemos que no es la mejor solución, pero es la mejor manera de crear objetos sin recurrir a métodos con múltiples parámetros en la clase AppLogic. Esto es algo que, a futuro, pensamos intentar mejorar, buscando una nueva manera de crear los objetos desde la UI sin necesidad de tener clases tan acopladas.

Otra decisión interesante de la lógica fue tener múltiples listas para los objetos. En una instancia de la clase AppLogic hay una lista que guarda todos los usuarios, otra para materiales, otra que guarda todas las figuras, otra para los modelos y una última para las escenas. Esto implica que, si se quiere mostrar las figuras del usuario activo, tengo que recorrer la lista de todas las figuras de todos los usuarios para mostrar solo las de uno, lo que es extremadamente poco eficiente. La decisión que se tomó fue tener una lista con todas las figuras creadas por todos los usuarios y otra que guarda las figuras creadas únicamente por el usuario activo, haciendo lo propio para el resto de objetos. Esta última lista se actualiza cada vez que un usuario hace login, por lo que solo deberemos recorrer todas las figuras solo al entrar el usuario, en el resto de casos se utiliza solamente la lista del usuario.

También vale la pena mencionar que dentro del dominio hay dos clases que fueron pensadas a futuro y con las que quizás no respetamos del todo las normas planteadas por la metodología del TDD, ya que no hicimos lo mínimo indispensable para que el código pase las pruebas y cumpla los requerimientos planteados, como se indica. Nos estamos referimos a las clases Shape y Material. Si hubiéramos querido solamente que el código pasara, estas dos clases no hubieran sido necesarias ya que podríamos haber hecho todo solamente con Sphere y Lambertian. Pero, teniendo en cuenta que es de nuestro interés para el segundo obligatorio tener que cambiar lo menos posible y que además es una buena adición a la estructura del código, pensamos que sería correcto agregar estas dos clases abstractas para estructurar el sistema de manera más completa.

Diseño de tests

Pasando al proyecto de tests, cada clase tiene su respectiva clase de pruebas unitarias, con métodos de prueba que buscan cubrir todos los requerimientos del sistema que son cubiertos por esa clase. En todos los casos, las pruebas fueron escritas con anterioridad a la implementación del caso de uso en particular, siguiendo así el concepto de TDD.

Con respecto al renderer, pensamos que sería más fácil transcribir todo el código desde JavaScript (brindado por el profesor) a C# primero y luego encargarnos del testing. Esto lo hicimos de esa manera ya que el código en sí mismo estaba ya hecho y nuestra mayor prioridad era entenderlo, cosa que creímos que sería más sencillo de lograr implementando. Pensar, diseñar y escribir casos de prueba de algo cuyo funcionamiento y resultado esperado es desconocido puede

resultar complejo, confuso y hasta contraproducente. En otras palabras, no se aplicó TDD en renderer. En su lugar, se codificó primero, basado en los scripts del profesor, y luego se probó su funcionamiento a partir de pruebas, tanto empíricas, viendo los resultados obtenidos y viendo si resultaba coherente, como unitarias. No obstante, el render es el único caso donde la consigna del TDD no fue aplicada. En el resto de clases del dominio se siguió al pie de la letra.

Al implementar las pruebas de la clase Vector nos topamos con un problema. Un vector lo definimos como una terna de doubles. El problema radica en que en C#, los doubles son representados con punto flotante. Esto implica que los valores que toman puedan no ser exactos, dando incongruencias como que un número sea diferente a si mismo. Esto es exactamente lo que pasó con sus pruebas, ya que algunos de sus tests fallaban porque, por ejemplo, “se esperaba <13,57> pero se obtuvo <13,57> ”.

Para sortear esta situación se contemplaron diversas soluciones. Primero se pensó en cambiar los doubles por decimals, que no son de punto flotante y solucionarían el problema. Sin embargo, esto implicaba realizar muchos cambios en la implementación, cosa que es poco práctico, sobre todo teniendo en cuenta que los cambios únicamente buscaban hacer pasar pruebas que ya de por sí éramos conscientes de que eran correctas. Posteriormente se pensó en asumir que existiría un margen de error, y que la diferencia entre el resultado esperado y el obtenido debía ser menor a él. Finalmente, se decidió por usar redondeos en las pruebas, reduciendo los resultados esperados y obtenidos a 2 o 4 cifras tras la coma, dependiendo del caso.

Diseño de interfaz de usuario

Para finalizar, encontramos el proyecto de interfaz de usuario. Sus elementos se pueden dividir en cuatro categorías. Los recursos son todas las fotos que se usan en el sistema. Se incorporó también una interfaz IRouter, que nos permite pasar de una pantalla a otra en una sola página, cumpliendo así el objetivo de crear una aplicación SPA (single page app).

Un detalle a mejorar en este aspecto es el uso de las pantallas. En muchos casos, se debe llamar a una pantalla desde un componente. El problema que trae esto es que las ventanas mostradas se guardan en un stack, por lo que al llamar varias pestañas, estas se van guardando, ralentizando el programa. Una solución sencilla sería, en los casos donde generamos una nueva pantalla primero llamar al método GoBack() del router, que nos devuelve a la pestaña anterior y luego pasar a la siguiente pestaña, haciendo que no existan tantas pantallas en el stack. Esto, sin embargo, no se ve bien, ya que se vuelve a la pestaña anterior durante un pequeño tiempo antes de pasar a la siguiente. Se intentará mejorar esto para la próxima entrega.

La siguiente categoría son los componentes, controladores personalizados que creamos para poder reutilizar ciertos elementos de la página. Por ejemplo, al

estar trabajando en la creación y listado de las figuras, notamos que las entradas de la lista de figuras eran en su estructura y funcionamiento muy parecidos a los elementos de las listas de materiales y modelos, por lo que tomamos la decisión de hacerlo todo en un mismo componente llamado ListElement. El ListElement recibe, además de todos los elementos a mostrar, un string que indica el tipo de objeto que se va a representar. El tipo tiene como objetivo determinar qué objeto le estamos pasando para agregarlo a la lista correspondiente, de la manera que se espera, haciendo que un mismo componente se comporte de maneras distintas según el tipo que recibe y así lograr unificar los elementos de todas las listas en un único componente reutilizable.

Como último ítem de UI, se crearon diversas pantallas. Cada una de estas pantallas contiene lo necesario para cumplir un requerimiento de sistema específico. Así, tenemos una pantalla para crear figuras, otra para listarlas y eliminarlas, otra pantalla para el login y otra para el sign out, entre otras funcionalidades.

Hay algo que es relevante aclarar con respecto a la manera en la que trabajamos el proyecto. En el dominio aplicamos TDD como es debido, primero hicimos los test (etapa Red) y luego el código para que los mismos pasarán (etapa Green), pero la etapa de refactor fue pospuesta. Como se mencionó, el dominio fue el inicio del proyecto, lo primero que hicimos. Debido a esto, muchos de los conceptos de clean code aún no habían sido dados en clase. Por exactamente esta razón, quisimos dejar el refactorio para cuando terminamos ya todas las clases del dominio, ya que la falta de conocimientos con respecto al clean code y como hacer bien el refactor, hacía que pareciera mejor encargarnos de ello más tarde.

Una vez fue terminado el dominio y se dieron los temas pertinentes en clase, se abrió una rama fix del proyecto para ocuparnos del refactorio de todo, en paralelo con la implementación de la UI. Es verdad que esto es una mala práctica y no es conveniente, ya que conviene hacerlo cíclico, pero nos pareció mejor alternativa que intentar emprolijar el código sin saber qué es lo que debíamos evitar ni como se debía hacer realmente.

Manejo de errores

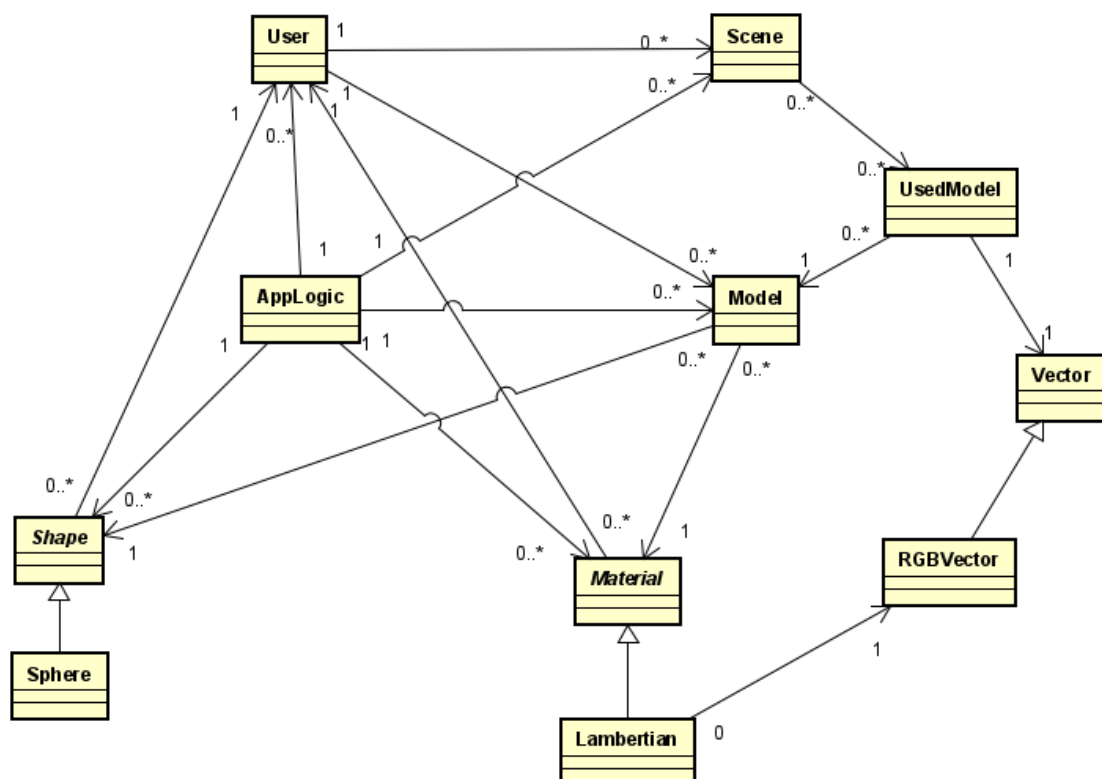
El control de error en nuestro sistema es más bien simple. Se creó una clase de excepción BusinessException que es lanzada cada vez que se incumple una de las reglas del negocio planteadas en la letra. Estos errores pueden ser que un usuario tenga dos modelos con el mismo nombre, que el nombre de usuario sea vacío, o que el radio de una esfera sea negativo, por nombrar algunos.

La validación de estas reglas se puede dar en la capas dominio o lógica, dependiendo del requerimiento. Para ejemplificar, no pueden existir dos usuarios con el mismo nombre, por lo que se requiere revisar todos los usuarios en busca de posibles nombres duplicados. Esto se necesita hacer en AppLogic para poder tener acceso a la lista de usuarios registrados. Sin embargo, los username también deben ser alfanuméricos, pero para validar esto no necesito conocer el resto de usuarios, la validación se lleva a cabo en la propia clase User, en la capa dominio.

Cualquiera sea el caso, si se incumple alguna regla, se lanza una excepción que luego es capturada en las clases de UI. Esta intenta llevar a cabo una acción y, si ocurre una excepción previene el error, capturándola y enseñándole al usuario un mensaje descriptivo para que pueda solventar el problema. Este mensaje se puede mostrar en pantalla dentro de la propia aplicación o en un cuadro de texto emergente.

Herencia y polimorfismo

En el apartado de Aclaraciones de diseño se dijo que se aplicó en ciertos casos herencia para así poder reutilizar código. En concreto, se vió en tres casos. El siguiente diagrama de clases puede ser útil para comprenderlos:



En una primera instancia, se creó la clase RGBVector, que hereda de Vector. Es una clase que representa un vector de colores, donde cada elemento de la terna representa un valor de un color primario. Entonces, RGBVector tiene todos los

métodos de Vector, pero tiene algunos extra relacionados con los colores y tiene la limitante de que sus valores deben ser entre 0 y 1.

Los otros dos casos son los de figura y material. Estas clases fueron implementadas pensando principalmente en el segundo trabajo obligatorio, siguiente instancia de evaluación. Shape es la superclase de Sphere, así como lo será de las posibles futuras figuras a implementar. De manera análoga, Material es la superclase de Lambertian y será lo propio para futuros materiales.

Esta herencia también nos permite tener listas de figuras y materiales. De esta manera, al renderizar una imagen, se puede recorrer la lista de figuras que contiene, ya que no importa si es una esfera u otra forma, la clase contendrá todo lo necesario para ser renderizada. Así logramos trabajar las figuras de manera genérica, sin importar cuál figura es particularmente.

Algo que nos percatamos es la similitud que hay entre ciertos métodos y atributos de algunas clases. Por ejemplo, Shape, Material, Model y Scene tienen todas un dueño y un nombre. A su vez, en todos los casos se debe verificar que el nombre cumple ciertas reglas. Esto lleva a que, por momentos, estas clases sean muy similares y hasta casi idénticas, con código duplicado, cosa que no es favorable. La mejor manera de solventar esto podría ser modificar estos métodos repetitivos para que funcionen con tipos genéricos, permitiendo mayor reutilizabilidad de código. No obstante, por cuestiones de tiempo, no fue posible implementar esto en esta ocasión, por lo que se contempla la idea para la siguiente entrega.

Clean Code

Creemos conveniente explicar no sólo los conceptos de clean code aplicados, sino que ejemplificar también cómo se aplicaron.

Primero que nada, se buscó tener nombres mnemotécnicos e informativos, que expliquen claramente que guarda cada variable, de que se ocupa cada clase o que función cumple cada método. Se evitó la desinformación (dar información innecesaria) cuando, por ejemplo, llamamos models a la lista de modelos, porque llamarlo modelList sería redundante y complicaría la lectura del código sin aportar datos de interés.

En lo que respecta a funciones, se intentó que estas cumplan un único propósito. Si en cierto caso sentíamos que la función estaba tomando muchas responsabilidades, se dividió en múltiples funciones más simples que cumplen solo una. Un buen indicador para ver si un método está haciendo muchas cosas es la cantidad de parámetros que recibe. Si toma más de dos parámetros, es probable que lleve a cabo muchas acciones. Es por esto que intentamos tener métodos con la menor cantidad posible de parámetros, cosa que, salvo en algunos constructores

y funciones complejas del renderer, cumplimos. Adicionalmente, las condiciones de estructuras iterativas y condicionales se extrajeron en métodos privados o en variables booleanas para hacer el método más legible y entendible. Por ejemplo, en vez de preguntar si el valor de rojo en un vector RGB era menor a 0 o mayor a 1, se pregunta si el valor está fuera de los límites.

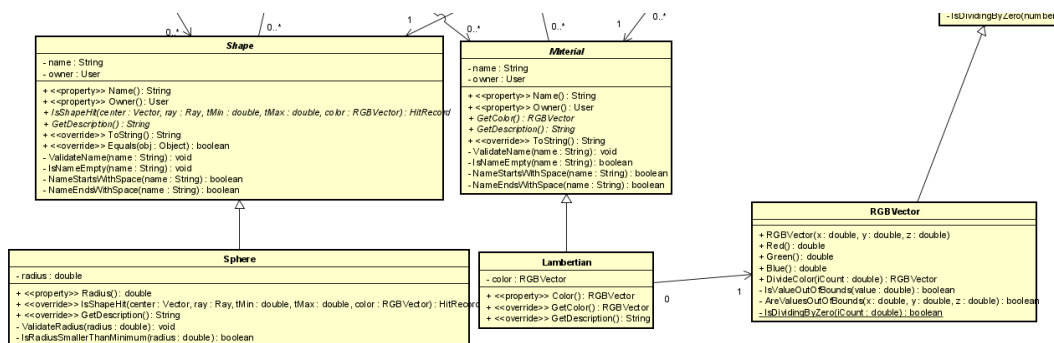
En cuanto al formato, se buscó poner lo más genérico y característico al principio, con todos los métodos más específicos sobre el final. Una manera de ver esto es que los métodos públicos y los atributos se definirán al comienzo de la clase, mientras que los privados se pondrán al final, haciendo la lectura del código más sencilla y natural.

Al igual que las funciones, cada clase busca cumplir un único propósito. Por ejemplo, la clase Sphere almacena y brinda toda la información necesaria para la renderización de esa figura, la clase Vector ofrece una estructura y métodos para operar con vectores y la clase AppLogic se ocupa de manejar y controlar todas las clases, sirviendo de intermediario entre UI y dominio. Esto genera una alta cohesión de clase, las clases tienen un sentido, cumplen una única misión y sus métodos están relacionados entre ellos.

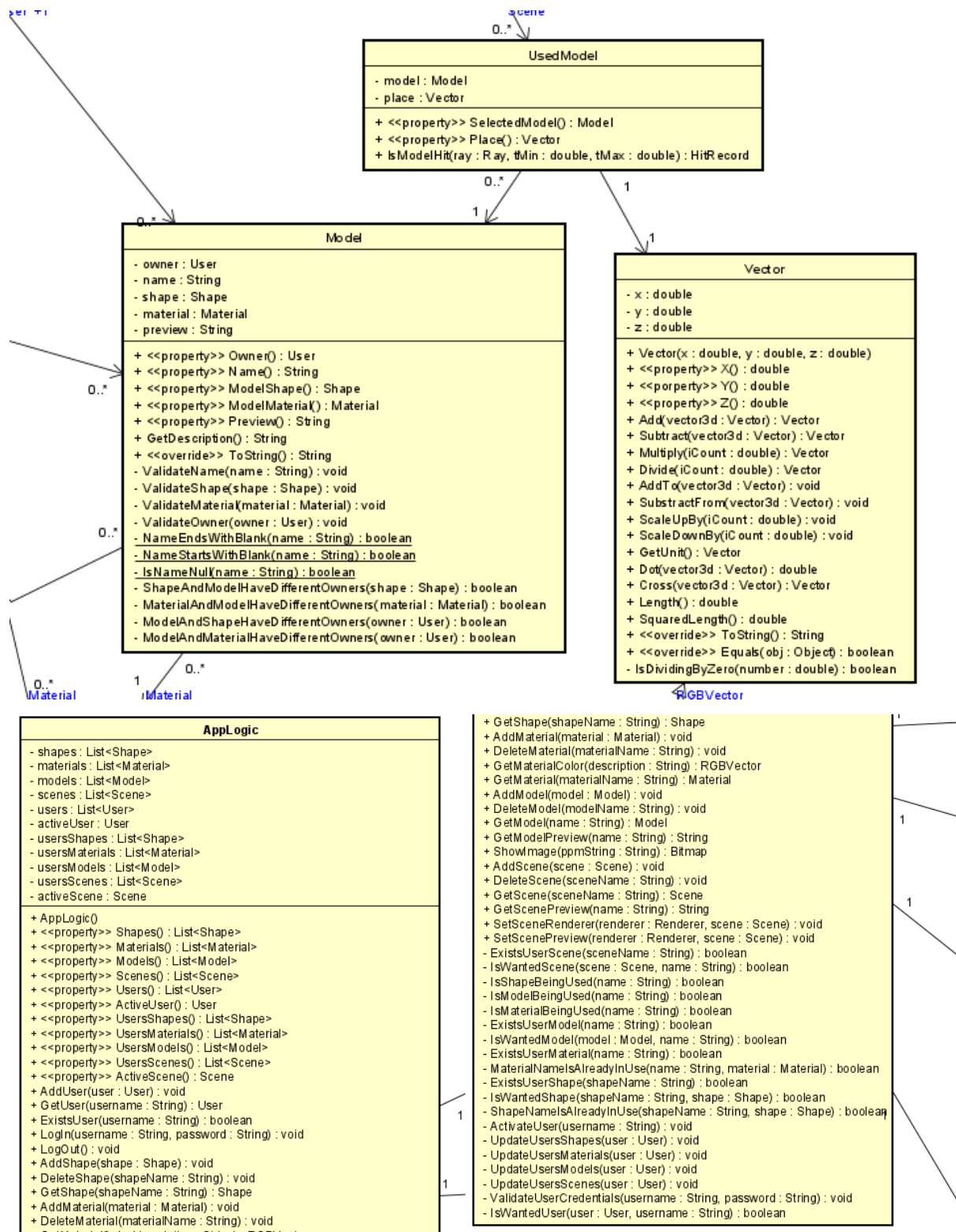
Interacción entre clases

Por simplicidad y para mayor facilidad a la hora de ver los diagramas de clase, se separó el dominio en dos diagramas distintos. A pesar de que la función de todas las clases del dominio es contribuir a la posibilidad de renderizar escenas, hay ciertas clases como Renderer o Camera que están mucho más arraigadas a la construcción del render, por lo que se colocaron en un diagrama separado para así evitar los diagramas muy cargados y poco comprensibles.

Se adjuntan primero capturas con mayor detalle y con todos los atributos y métodos del diagrama de clases visto más arriba:.

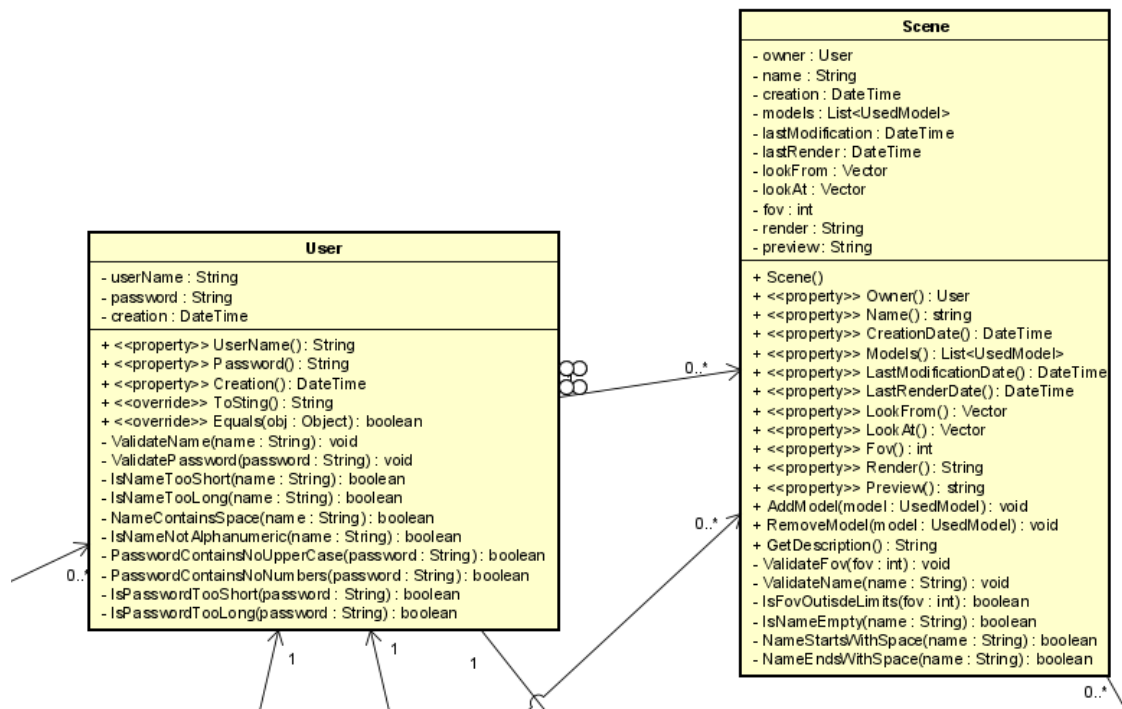


Podemos apreciar en la anterior captura los casos de herencia de los que se habló en párrafos anteriores.

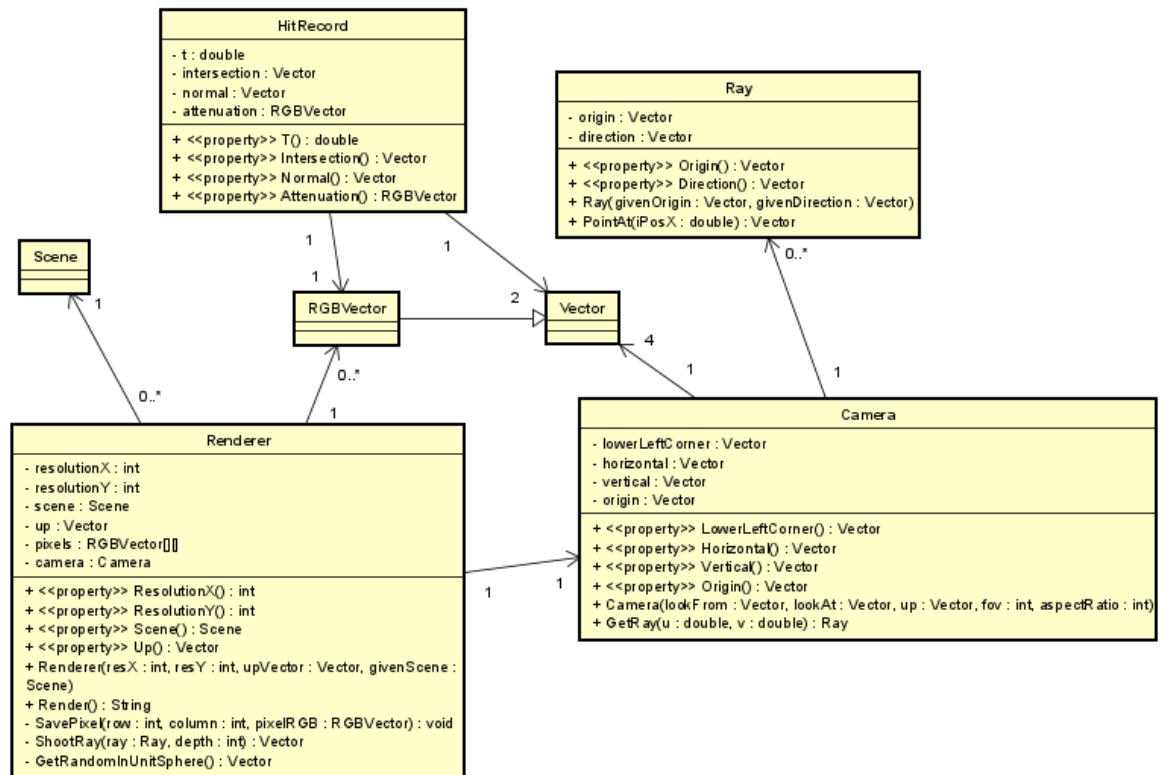


Como se puede notar, la clase AppLogic está muy acoplada, lo que significa que se relaciona con muchas otras clases. Esto, usualmente, no es conveniente, ya que un cambio en AppLogic afectará al resto. No obstante, en este caso sí es deseable, ya que AppLogic es la clase que controla todo. Se intenta que las clases se comuniquen a través de ella y que sea el vínculo entre frontend y backend.

AppLogic también es una clase muy larga y cargada de métodos, cosa que tiende a no ser muy favorable, ya que se podría tender a cumplir múltiples finalidades en ella. En nuestro caso, tiene muchos métodos “repetidos” que podrían intentar unificarse utilizando generics, por ejemplo; pero es necesario que sea una clase amplia. Recordemos que es el nexo de conexión entre todas las clases, entre front y backend, es una clase importante que debe englobar todas las acciones del usuario.



También existe mucho acoplamiento en la clase User. Esto se debe a que clases del backend tales como Shape, Material, Model y Scene tienen un dueño. Evitar esto sería bueno, pero no es posible, ya que si se quiere cumplir los requerimientos del sistema, estas clases deberán tener un atributo de la clase User. Otra alternativa sería que cada usuario guarde una lista de todas sus creaciones (figuras, materiales, etc.). No obstante, pese a favorecer el rendimiento ya que no vamos a tener que crear una nueva lista para cada atributo cada vez que se dé un log in, teniendo ya las listas de objetos creados por cada usuario; esta decisión no trae realmente beneficios en materia de diseño, el acoplamiento seguirá siendo el mismo, simplemente cambia donde se guarda la conexión (en el User o en el Objeto a guardar).



Pasando a las clases del renderizado, vemos que estas clases son más independientes. No requieren de AppLogic ya que es Scene la que crea los objetos de la clase Render, que a su vez instancia, directa o indirectamente, el resto de objetos del diagrama. Sin embargo, no se puede decir que están completamente aisladas del resto de clases, ya que necesitan interactuar con los modelos que renderizan y conocer las figuras y materiales, asociándose de manera indirecta con el diagrama anterior.

De hecho, esta asociación indirecta con el resto se hace más clara al ver que cada figura se encarga de “saber” cómo renderizarse. Más específicamente, el renderer se fija para cada pixel si el rayo le pegó a alguna figura. Para esto, cada figura debe ser capaz de identificar si el rayo impacta, ya que la matemática es diferente para cada figura.

Funcionamiento

Para mostrar el funcionamiento de la aplicación, se grabó un video que muestra el flujo de las pantallas, la creación de los elementos de la escena y el uso en general. Se adjunta enlace:

<https://youtu.be/YNPSHI64S9g>

Se adjuntan en el anexo capturas de pantalla de alertas que muestra el sistema frente a algunas excepciones que no se incluyeron en el video para no hacerlo muy largo.

Cobertura de pruebas

El testing busca probar el correcto funcionamiento del sistema. Adicionalmente, las pruebas sirven como guía a la hora de diseñar el código. El método de trabajo en relación al testeo fue primero escribir las pruebas unitarias y luego implementar el código de producción, lo que se buscaba probar. En algunos casos, posteriormente a la implementación del código y de que las pruebas asociadas pasaran correctamente, se agregaron nuevos tests con el objetivo de ampliar el rango de casos probados.

Se considera también el caso del renderizador que, como ya se explicó, para su implementación no se siguió la metodología del TDD. Luego de codificado, se escribió una prueba unitaria que abarca todo el proceso de renderización, para así demostrar su correctitud.

La implementación de pruebas que requerían números aleatorios y fechas presentaron un dilema. En un inicio, se asignó un atributo de tipo DateTime o Random para cubrir estas necesidades. No obstante, al testear el resultado de ese DateTime no se probaba si era correcto (ya que se consideraba sólo la fecha, obviando la hora, que varía cada segundo) y no encontrábamos manera de probar el aleatorio, ya que no tendríamos manera de saber si es el mismo o no. Para solventar este problema se implementó una clase DateTimeProvider, que toma la fecha y hora actual y la almacena para poder devolverla después y así tener un control de que la hora es correcta y el test salió bien. Adicionalmente, la clase RandomNumberProvider nos permite generar un número aleatorio y poder conocerlo posteriormente, brindándonos la posibilidad de probar el renderizador.

El proceso de construcción del renderizador fue correr el código brindado por el profesor, cambiando los valores aleatorios por uno fijo. Posteriormente, corremos nuestra implementación con el mismo valor fijo, verificando si los resultados coinciden.

Volviendo a las pruebas en general, se usó el framework MSTest y cada prueba representa un caso de uso diferente. Se busca tener un 90% de cobertura, o sea, que las pruebas cubran al menos un 90% del código, ya que esto indica que se probó la mayoría de los casos posibles. Se adjunta nuestro análisis de cobertura:

Hierarchy	Covered (Blocks)	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered (Lines)	Covered (%Lines)
alfre_DESKTOP-08A01CN 2023-05-10 20_37_05.coverage	2366	111	2047	7	98	95,12%
tests.dll	1154	64	1173	0	63	94,90%
obligatoriofernandezmorandi.exe	1212	47	874	7	35	95,41%
() Logic	434	33	285	5	23	91,05%
() Domain	778	14	589	2	12	97,68%

Se puede ver que quedaron líneas de código sin testear. Esto se puede deber a diversos factores. En el caso de la lógica hay varios casos que no se pueden dar,

por lo que no tiene sentido testearlos, pero que es necesario programar. Por ejemplo, la única manera de que se llame al método GetShape (dado un string devuelve la figura con ese nombre) de AppLogic es al interactuar con el modelo en la interfaz, por lo que esa figura sí o sí existirá y el método la devolverá. Sin embargo, no se puede no retornar nada el caso en el que la figura no exista, pese a que sea imposible, por lo que hay que poner un retorno extra que nunca se usará. Ese camino no tiene sentido testearlo ya que no sucederá, pero está ahí y es código no cubierto.

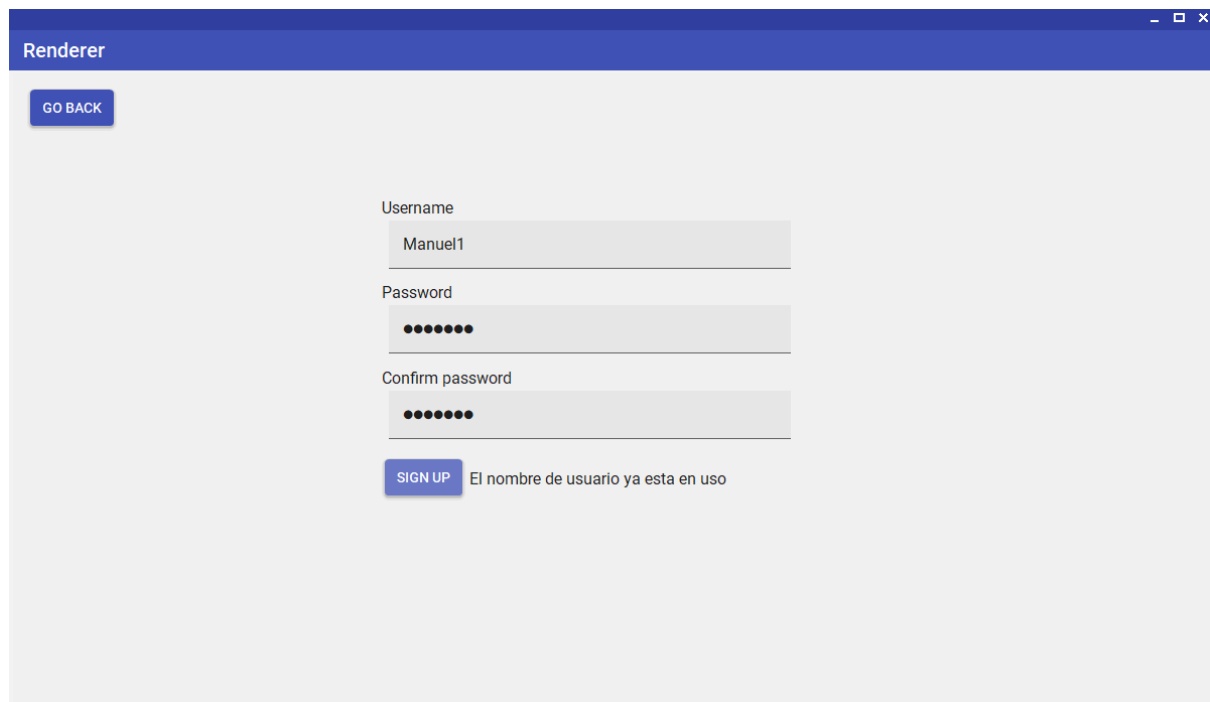
En adición, existen casos que pasamos por alto o que no contemplamos, y se escapan del código cubierto por las pruebas. No obstante, 95% es un muy buen porcentaje y la gran mayoría de la implementación fue testeada.

Reflexiones

Se consiguió el objetivo, logrando construir la aplicación solicitada, de manera prolija y aplicando la metodología requerida. Si bien el producto final tiene imperfecciones para pulir, lo consideramos un trabajo aceptable y acorde a lo que se pidió, con mejoras y arreglos ya en mente para la siguiente instancia de evaluación.

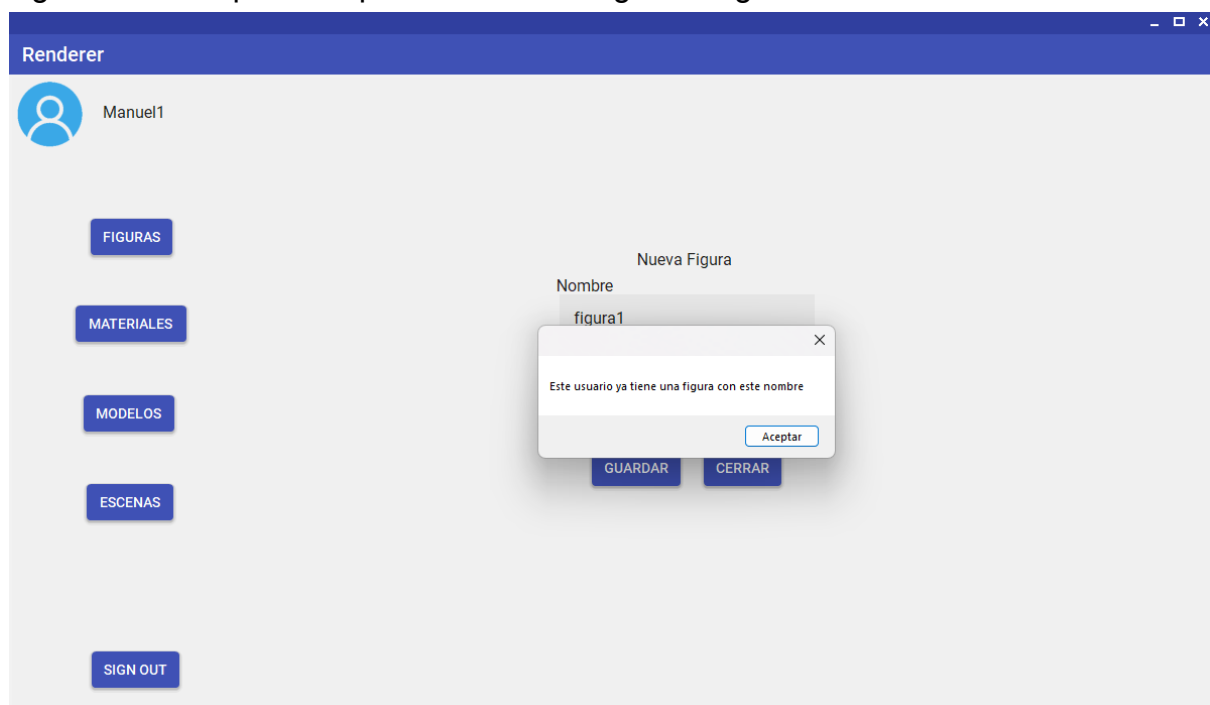
Anexo

Figura 1: No se permite que existan dos usuarios con el mismo username.



The screenshot shows a web application window titled "Renderer". In the top-left corner, there is a "GO BACK" button. The main form contains three input fields: "Username" with the text "Manuel1", "Password" with masked characters, and "Confirm password" also with masked characters. Below these fields is a "SIGN UP" button. To the right of the button, a message states: "El nombre de usuario ya esta en uso".

Figura 2: No se permite que un usuario tenga dos figuras con el mismo nombre.



The screenshot shows a web application window titled "Renderer". At the top left, there is a user profile icon and the name "Manuel1". On the left side, there is a vertical menu with buttons: "FIGURAS", "MATERIALES", "MODELOS", "ESCENAS", and "SIGN OUT". In the center, a modal titled "Nueva Figura" is open. It has a "Nombre" field containing "figura1". Below the field, a message says: "Este usuario ya tiene una figura con este nombre". At the bottom of the modal is an "Aceptar" button. Behind the modal, "GUARDAR" and "CERRAR" buttons are visible.

Figura 3: No se permite que un usuario tenga dos materiales con el mismo nombre.

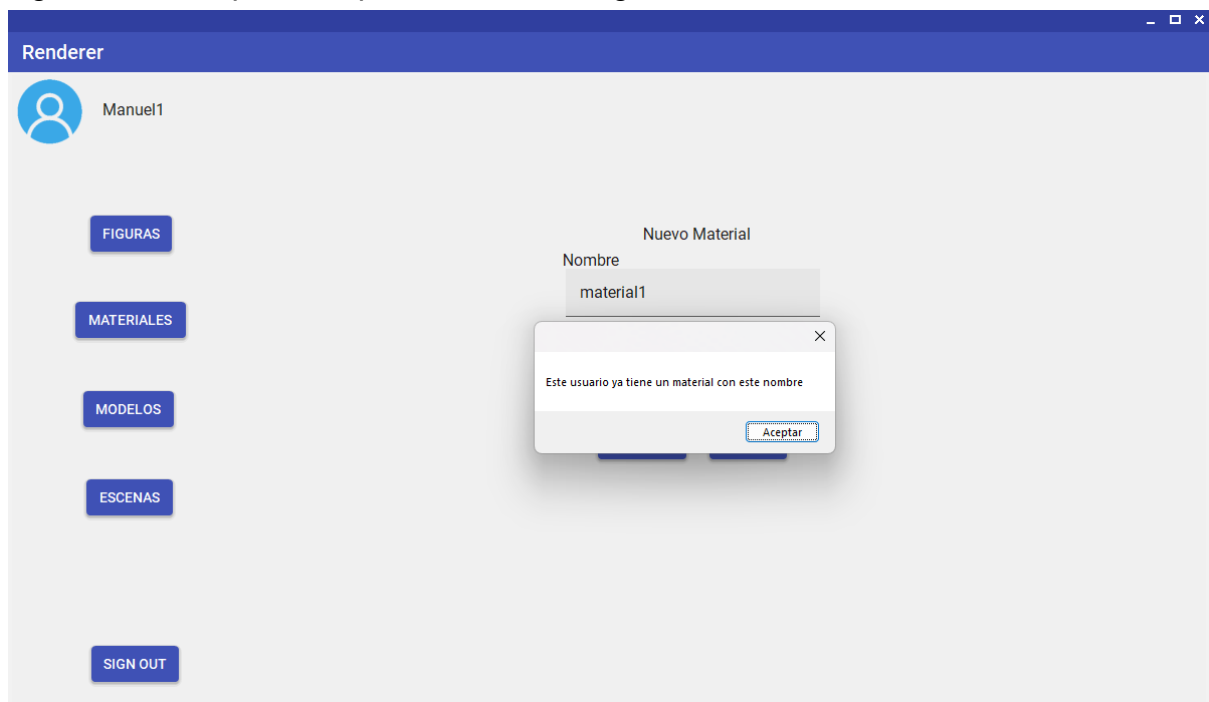


Figura 4: No se permite que un usuario tenga dos modelos con el mismo nombre.

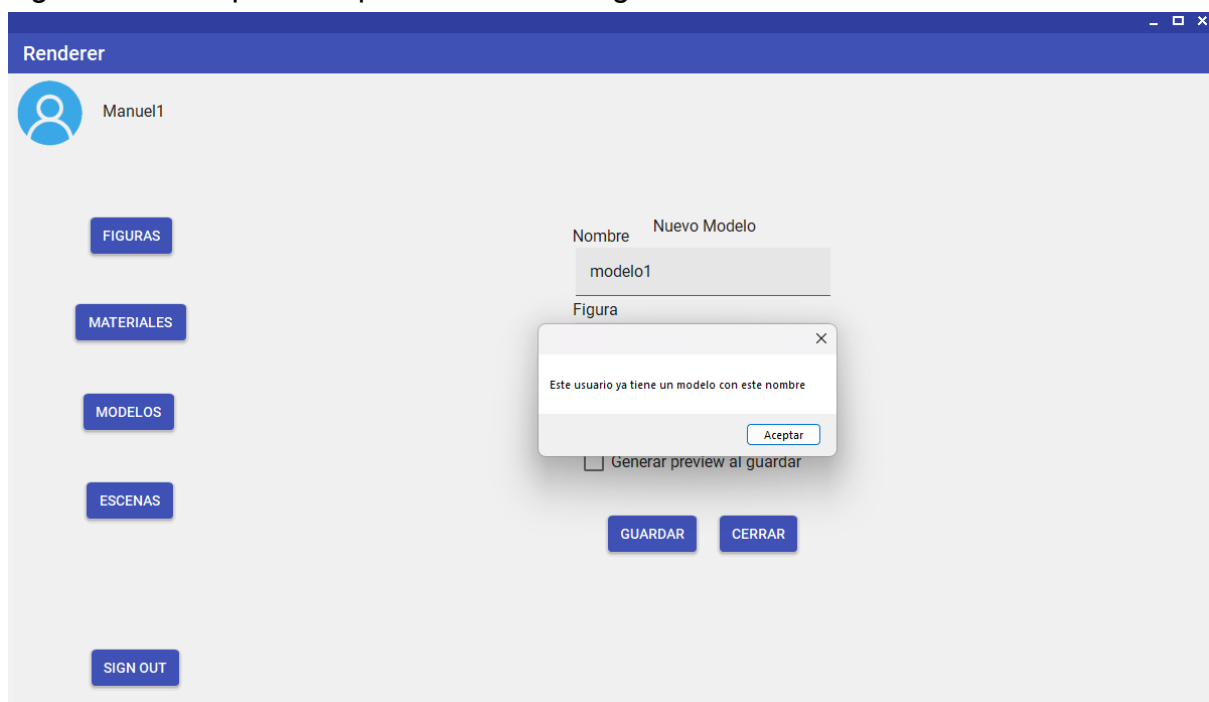


Figura 5: No se permite que un usuario tenga dos escenas con el mismo nombre.

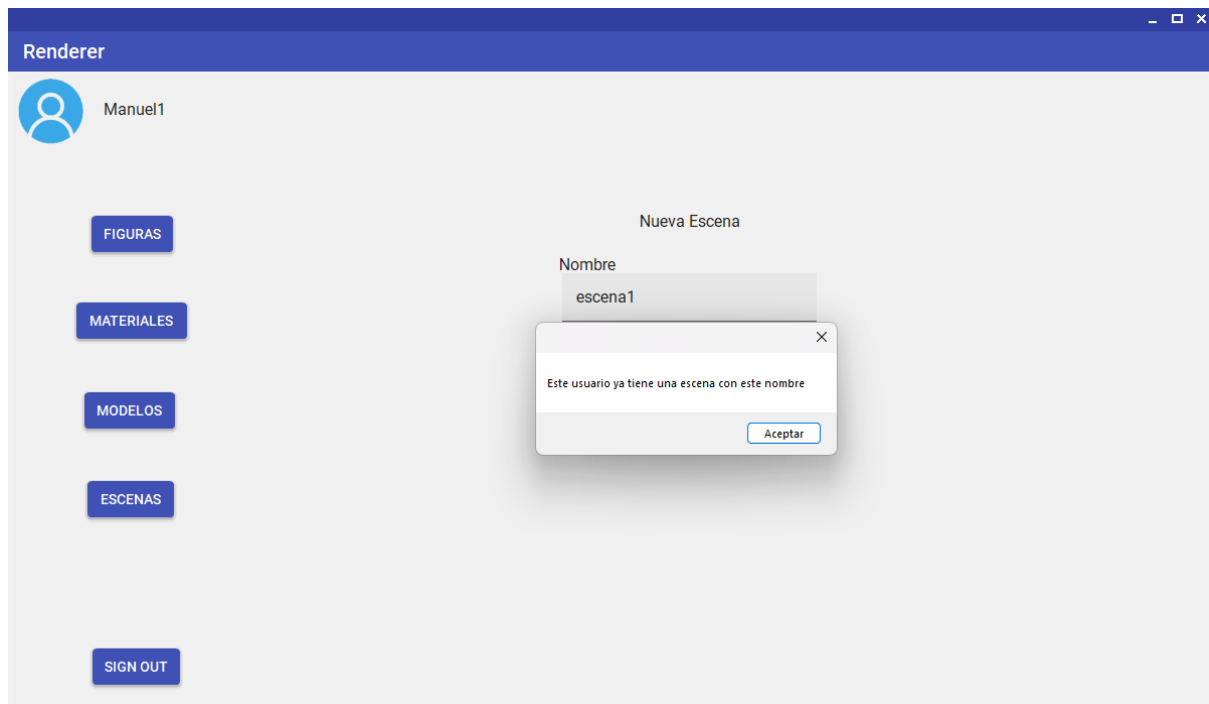


Figura 6: No se permite borrar una figura usada en un modelo.

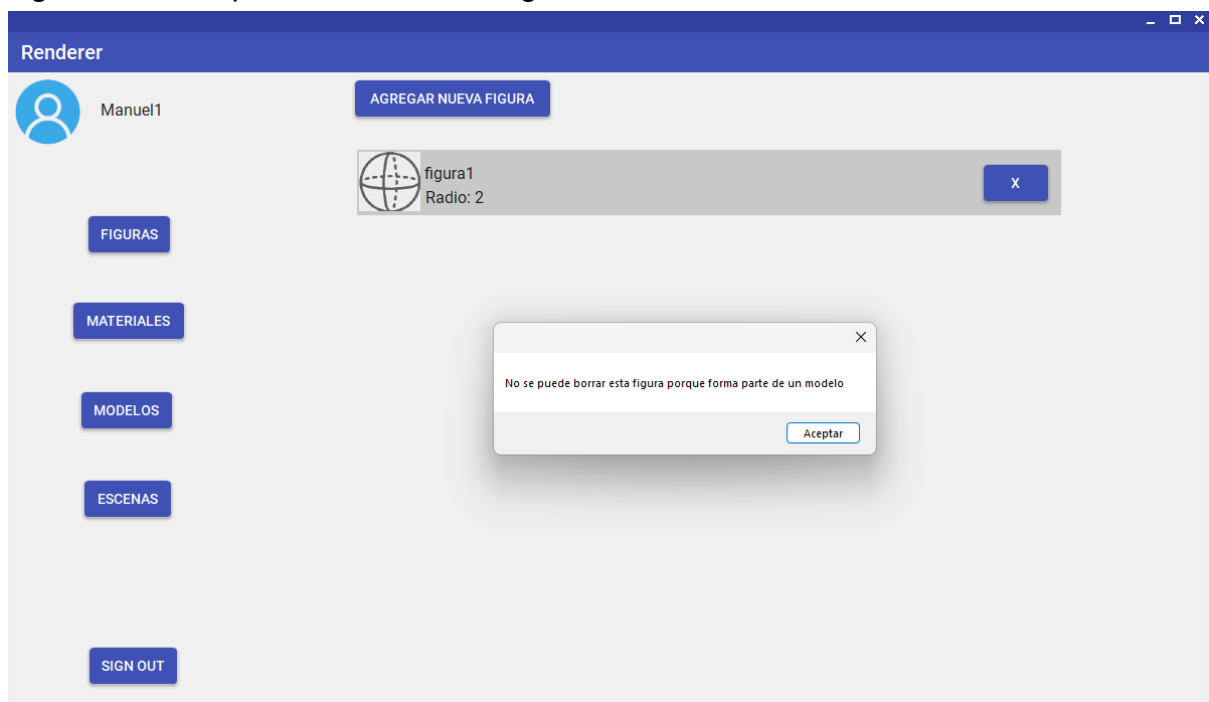


Figura 7: No se permite borrar un material usado en un modelo.

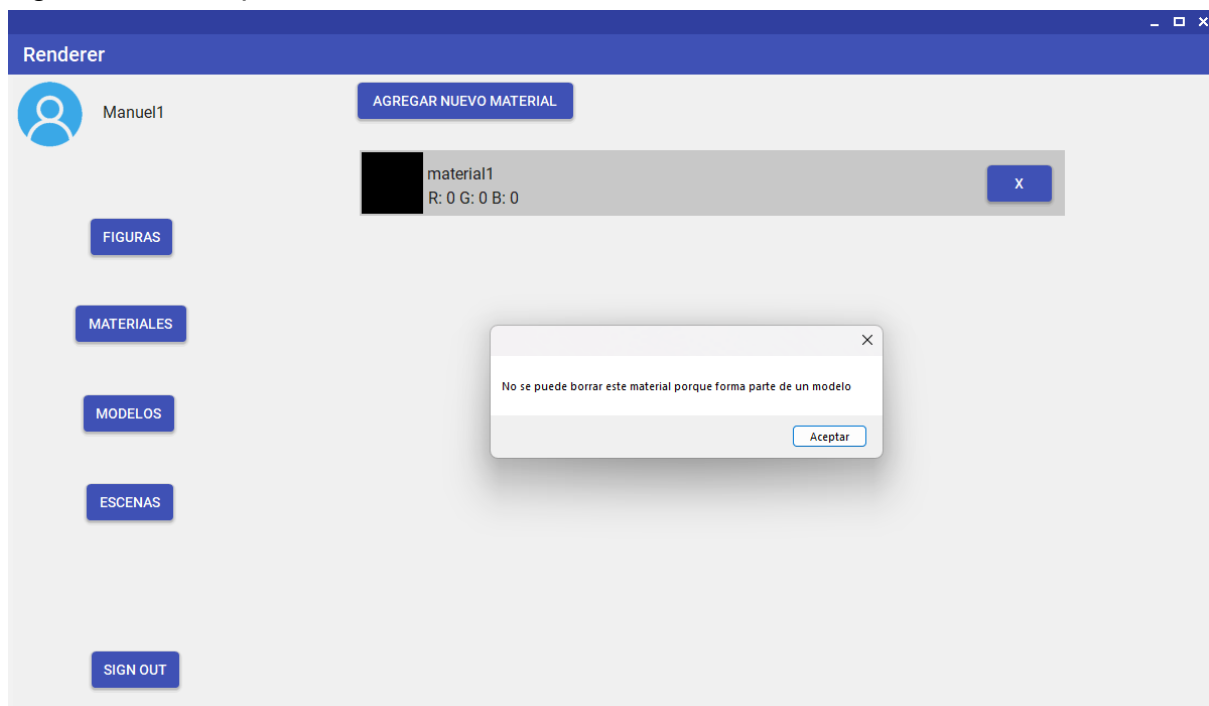


Figura 8: No se permite borrar un modelo usado en una escena.

