

Universidad ORT Uruguay

Facultad de Ingeniería

Obligatorio 1 - Diseño de Aplicaciones 1

Entregado como primer entrega
obligatoria de Diseño de Aplicaciones 1

 **GitHub** https://github.com/ORT-DA1-2023/266628_255981.git

Federico Rodriguez - 255981
Santiago Salinas - 266628

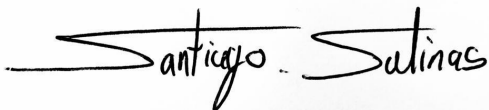
Tutores: Facundo Arancet Iza
Franco Bruno Galeano Roca
Gastón Mousques

Mayo 2023

Declaración de autoría

Federico Rodriguez y Santiago Salinas declaramos que el trabajo que se presenta en esta obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras construimos el primer obligatorio de Diseño de Aplicaciones 1
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra
- En la obra, hemos acusado recibo de las ayudas recibidas
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros y qué fue contribuido por nosotros
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes



Santiago Salinas 6/5/2023



Federico Rodriguez 6/5/2023

Introducción al trabajo	4
Descripción general del sistema	4
Fallos conocidos:	5
Decisiones de diseños	6
API Gateway Class, punto de entrada	6
Uso de polimorfismo en el diseño	7
Responsabilidades	8
Funcionamiento	8
Creación de usuario	8
Inicio y cierre de sesión	10
Creación y manejo de esferas	10
Creación y manejo de materiales	12
Creación y manejo de modelos	12
Creación, edición y manejo de escenas	14
El Render	15
Pruebas unitarias y cobertura	16
Anexo	17

Introducción al trabajo

El sistema entregado es una aplicación hecha con la finalidad de que un usuario pueda generar imágenes, renderizadas mediante la implementación de una versión simplificada de **Ray Tracing**. Cada usuario puede generar sus propias imágenes y recursos, y luego acceder a ellos con exclusividad.

Todo a lo largo del desarrollo del sistema nos mantuvimos al modelo de trabajo de **Git Flow (Anexo 1)**, en el que siempre creamos una nueva rama para cada **feature** partiendo desde la rama **develop** o desde una rama feature ya creada. Estas features, sólo después de haber sido bien implementadas y testeadas, volverían a juntarse con la rama desde donde emergieron.

El proyecto debió mergearse a main solo al estar seguros de que estuviera listo para entregarse, pero encontramos fallos a último momento que debían arreglarse. Esto resultó en más de un commit a main.

El **paradigma general** del proyecto sigue la **POO**, Programación Orientada a Objetos, es decir que se centra en la creación de objetos, que son instancias de clases, y en la interacción entre ellos.

Para la **escritura del código** en sí, se siguió el modelo de **TDD**, Test Driven Development, en el que se comienza escribiendo las pruebas que queremos que nuestra implementación pueda pasar correctamente y luego se escribe el código de manera acorde. En primera instancia, el código escrito es lo mínimo necesario para poder pasar las pruebas, pero luego se puede refactorizar para optimizar su implementación.

En adición, el código debía de ser revisado continuamente para asegurarse de que cumpliera con las normativas de **Clean Code**, resultando en que sea lo **más legible posible**.

Descripción general del sistema

Al iniciar la aplicación, se presenta una ventana donde el usuario puede elegir entre iniciar sesión o crear un nuevo usuario.

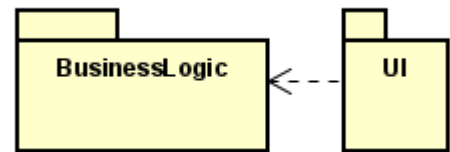
Después de crearse un usuario y de iniciar sesión, se muestran botones para desplegar distintas secciones dedicadas a la creación de figuras, materiales y modelos (estos últimos siendo compuestos por una combinación de una figura y un material previamente creado). Estas herramientas son la base para poder luego crear y editar **escenas**, donde posicionando los modelos que queremos mostrar y el punto de vista desde donde se verán, se define la imagen que será renderizada.

Gracias a la distinción entre iniciar sesión con un perfil u otro, cada usuario tiene sus propios **objetos** (formas, materiales, modelos y escenas) a los que accede

sin que nadie más pueda interactuar con ellos y que persistirán incluso después de cerrar la sesión para cuando decida volver.

El sistema se divide en dos paquetes que llamamos UI y Business Logic. Business Logic contiene las definiciones de todos los objetos que puede crear el usuario, los métodos con los que puede gestionarlos y la implementación del raytracing para renderizar las imágenes.

Ya después de haber completado el proyecto analizamos que de haber implementado otra capa intermedia entre las dos hubiera sido mejor. De esta manera es posible abstraer la implementación de cómo se crean y verifican la integridad de los objetos desde el punto de vista de UI. Ahora mismo, cambiar la implementación de cualquiera de los dos paquetes puede resultar en tener que modificar el otro, por lo que no es muy mantenible.



Fallos conocidos:

- Al crear o editar los objetos del usuario, este tiene la libertad de ingresarles un nombre con el largo que desee. Esto sin embargo, puede resultar en fallos visuales en los que los nombres mostrados se despliegan por encima de otros componentes si son muy extensos.
- Si no se le asigna un nombre a una escena, y se aprieta el botón change name, esta no se guardará, incluso si se colocaron elementos. Puede describirse como comportamiento normal, pero se podría agregar un mensaje de alerta de que se perderán los datos no guardados.
- Bajo ciertas condiciones, la pantalla de editar escena no avisa de que la imagen guardada está “Outdated”, ocurre al abrir una escena modificada. Sí aparece el aviso mientras se están editando parámetros y/o agregando elementos. Entendemos que es fácil de corregir, pero se halló poco antes de la entrega, por lo que se solucionara.
- Los valores para los vectores de posición al agregar un modelo a la escena tienen un rango de $[-9999, 9999]$. También se aplica para los valores de los vectores de *LookAt* y *LookFrom*. Esto ocurre debido a que se deben seleccionar topes en los objetos de Windows Forms *numericUpDowns*. Elegimos estos dos de forma arbitraria, pero ocurriría lo mismo aún si escogiéramos números más grandes, no existe el infinito.
- Si al renderizar una imagen, la cámara mira directamente desde arriba a la posición del centro de una esfera, pinta del mismo color toda la imagen en lugar de renderizar correctamente. (Puede haber alguna otra combinación que rompa, pero en general no hay problemas, como se puede ver en el usuario Test). *Anexo 2*.

Decisiones de diseños

API Gateway Class, punto de entrada

Reconocemos que nuestra aplicación no posee un paquete de clases puente que vincule los paquetes de interfaz y dominio, sino que optamos inicialmente por un diseño en el cual la UI conoce de las clases del BusinessLogic. Entendemos que haberlo hecho nos hubiera brindado beneficios como facilitar la escalabilidad, bajar el acoplamiento y mejorar el desarrollo. Todos factores altamente buscados dentro de clean code y al momento de hacer diagramas UML, ya que referencia a un solo punto de entrada y salida.

Actual

```
Sphere NewSphere = new Sphere();
try
{
    NewSphere.Name = sphereName;
} catch (ArgumentNullException ex)
{
    nameStatusLabel.Text = "* Name cannot be empty";
    nameIsCorrect = false;
}

if (SphereCollection.ContainsSphere(sphereName, loggedUser)) {
    nameIsCorrect = false;
    nameStatusLabel.Text = "* Sphere with that name already exists";
}

try
{
    NewSphere.Radius = radius;
} catch (BusinessLogicException ex)
{
    radiusStatusLabel.Visible = true;
    radiusIsCorrect = false;
}

if(nameIsCorrect && radiusIsCorrect) {
    NewSphere.Owner = loggedUser;
    DialogResult = DialogResult.OK;
}
```

API Style

```
SphereDTO NewSphereData = new SphereDTO()
{
    Name = sphereName,
    Radius = radius,
    Owner = loggedUser,
};

try
{
    BackendAPI.AddSphere(NewSphereData);
} catch (Exception ex)
{
    exceptionStatusLabel.Text = "* " + ex.Message;
    dataIsValid = false;
}

if(dataIsValid) {
    DialogResult = DialogResult.OK;
}
```

(El código actual es más largo porque indica a qué campo corresponde el error, en cambio tener un diseño con estilo API, tendríamos un texto general para informar sobre las excepciones)

En la práctica, consideramos que se utilizan desde el lado del paquete UI un número de Data Transfer Objects, que se enviaron a la través de la API para que se creen las instancias de las clases. De esta manera, abstraemos la implementación de las clases del dominio de la UI, y si cambiamos completamente de interfaz gráfica es aún posible seguir usando el mismo BusinessLogic.

No fue algo que tuviéramos en mente, no hasta hace no mucho antes de finalizar la entrega. Pero para la próxima entrega se realizará el cambio.

Uso de polimorfismo en el diseño

En términos de diseño, inicialmente consideramos crear una clase "UserObject" que contuviera información básica como el nombre y el propietario. Otras clases como Lambertian, Sphere, Model y Scene heredarán de esta clase base. Esta idea surgió porque todos los objetos compartían estos dos datos clave, así como las validaciones realizadas para garantizar que cumplieran con ciertos formatos.

Sin embargo, pronto abandonamos esta idea porque no cumplía con las reglas del polimorfismo. Cada objeto (clase) tenía propósitos completamente diferentes, incluso si compartían o no estos dos datos simples.

En otro caso similar, encontramos el problema de la estructura de agrupamiento de datos que se carga en memoria durante la ejecución mientras el usuario interactúa con la aplicación. Su función es almacenar los objetos que el usuario crea, como esferas y escenas. Inicialmente, utilizamos simples listas de objetos (Lists<object>), pero esto no nos permitió expresar todos los comportamientos deseados. Por ejemplo, al eliminar un objeto, no debería estar vinculado a otro de nivel superior. Por lo tanto, nos vimos en la necesidad de crear una clase de gestión de datos.

El primer problema surge de tener más de un tipo de objeto (clase), por ello pensamos en tener una clase gestora de los datos por cada categoría de objeto a guardar, y debido a las similitudes de las funciones que necesitan, (Add, Remove, Contains), pensamos en hacer una interfaz con generics <T>, de modo que se tengan que incluir ciertos métodos para mantener una cohesión.

A su vez, queríamos que estos datos se pudieran acceder desde cualquier parte de la aplicación, sin tener que pasarse como parámetro, por lo que queríamos que fueran clases estáticas, con los List como propiedades estáticas. Y aquí es donde nos topamos con un problema,

"By definition, interfaces create a contract for instances to fulfill. Since you cannot instantiate a static class, static classes cannot implement interfaces."

[Why static classes cant implement interfaces?](#) - StackOverflow

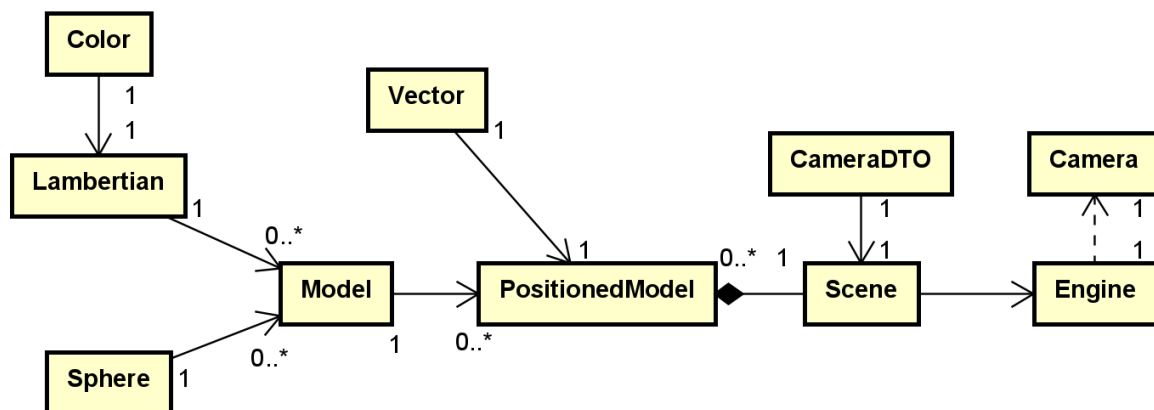
Conocemos que en la segunda entrega veremos Repositorios, que nos permitirán mitigar este problema. Por lo que provisionalmente dejamos una clase static por tipo de objeto.

Finalmente, luego de otros planteamientos sin fundamentos sobre la aplicación, concluimos que para nuestra implementación no habrían casos donde se pudiera aplicar polimorfismo o donde nos fuera beneficioso hacerlo.

Responsabilidades

El caso en donde mejor se expresa la necesidad de dividir las responsabilidades es en el editor de escenas. En el siguiente diagrama se ve claramente una estructura de interacción entre clases que se asemeja a una cadena o un árbol. Esta empieza desde los elementos más básicos, combinados para formar un modelo. Luego se crean ModelosPosicionados que le asignan una posición a un objeto con las características de un modelo ya creado. Finalmente, un conjunto de modelos posicionados forman una escena, que además agrega su propia información sobre la posición y funcionamiento de la cámara para luego hacerle llegar todo esto junto al Motor que renderiza la imagen.

Todas las clases trabajan juntas, pero cada una solo sabe directamente la información que le concierne.



Un modelo depende de un lambertian,
pero un lambertian puede estar en varios modelos

Funcionamiento

Creación de usuario

Al ejecutar el programa, no se cargará de disco ningún usuario, pero si hemos decidido crear en tiempo de ejecución un usuario predefinido para poder probar una escena compleja sin necesidad de crearla a mano.

Se puede acceder a este usuario con las siguientes credenciales

User: Test Password: Test1

Aun así, se puede empezar un usuario desde cero yendo al formulario de registro de usuario con el botón de Sign Up en la parte inferior.

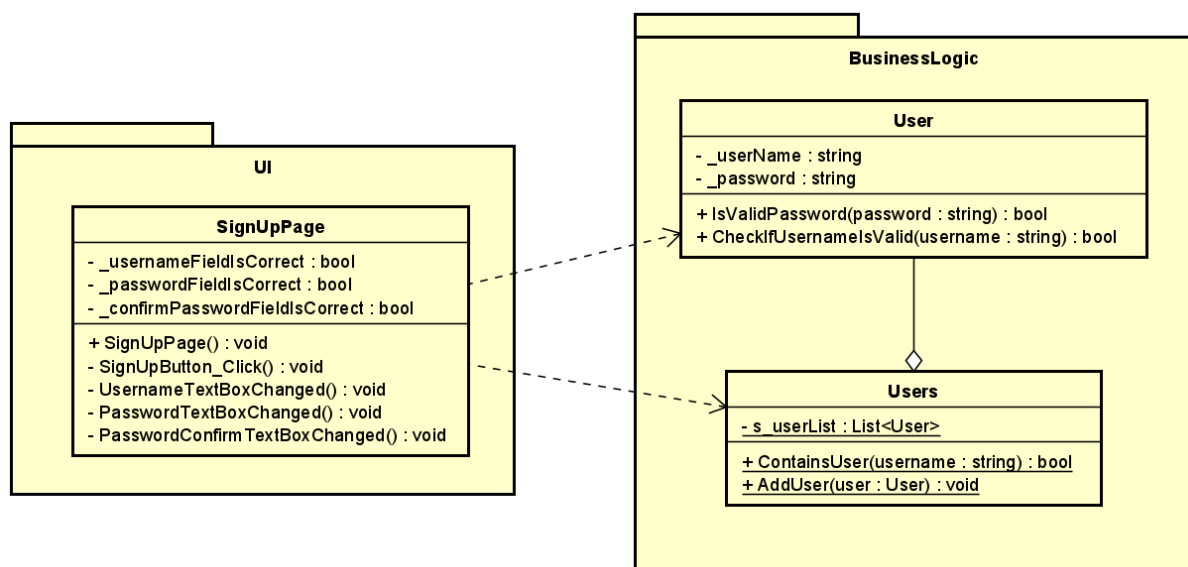
El formulario cuenta con tres campos: **User** que va a ser el nombre con el que se identifica la cuenta, **Password** para la contraseña y **Confirm Password** para asegurarse de que el usuario sabe la contraseña que está ingresando.

Mientras se escriben los datos, ya se comprueba su validez llamando a métodos de la **clase User** en **BusinessLogic** y pasándole como argumento un string con el texto que se pretende sea el usuario o la contraseña. Si no cumple algún requerimiento, saltará una excepción y la UI mostrará el mensaje de la **excepción** en una **label** para que el cliente pueda corregirse.

Al presionar el botón para enviar el formulario, se llamará a un método de la clase **UserCollection** pasándole el username como argumento para que compruebe que no exista ya un usuario con ese nombre. Si la consulta resulta en una excepción, se avisa al cliente de que el nombre ya está en uso para que elija otro; y si termina correctamente, se agrega el nuevo usuario a la lista y se vuelve a la pantalla de inicio de sesión.

Apretar el botón cierra la pestaña de inicio de sesión para ir a la de creación de usuario. Aquí se deben llenar los campos de Username, Password y Confirm Password correctamente para poder crear el usuario. Si lo escrito en alguno de los campos no cumple con los requerimientos, se mostrará un mensaje en rojo con la información necesaria para corregirlo. Incluso si se aprieta el botón para crear el usuario, mientras haya mensajes de aviso, no funcionará.

Al crearse correctamente, se guardarán los datos de nombre y contraseña en memoria y se volverá a la pantalla de inicio para poder ingresar con este u otro usuario creado previamente.

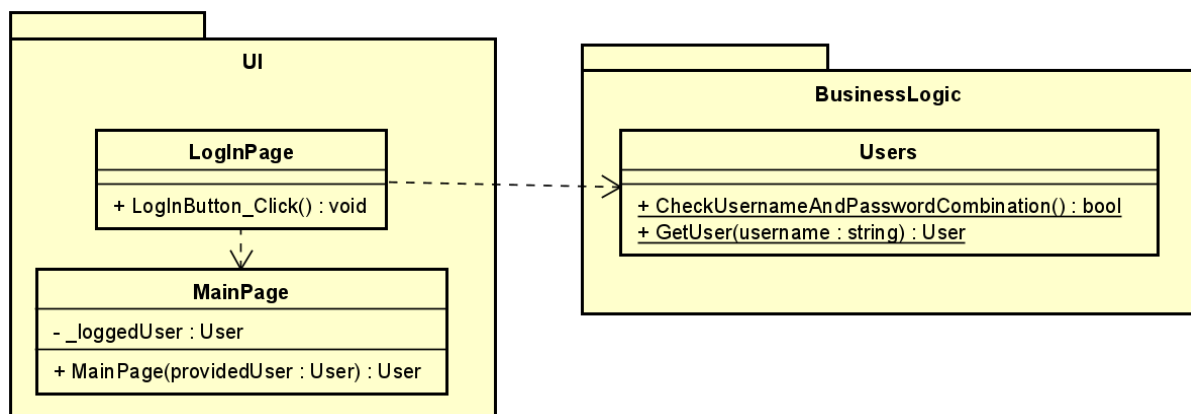


Inicio y cierre de sesión

El otro camino a seguir desde la pantalla inicial es el del Sign In. Llenando los campos de texto que se muestran con las credenciales de un usuario que se haya creado correctamente antes resultará en poder entrar a la pantalla principal de la aplicación desde donde se pueden gestionar todos los objetos asociados al perfil con el que se entró.

Este formulario toma los input de los dos textBox como strings para el nombre de usuario y la contraseña.

Si se intenta iniciar sesión con credenciales incorrectas, se mostrará un mensaje informando al usuario del error sin cambiar de pantallas. Luego, si logra conectarse, puede cerrar la sesión con el botón de Sign Out para volver a la pantalla inicial.



Cuando se inicia sesión correctamente, se trae desde Users el objeto de clase User que corresponde al nombre de usuario que ingresó. Este User se pasa como argumento en la creación de la MainPage y desde ahí llegará (también como argumento) a todos los forms que se crean como hijos del form principal y que necesiten saber que usuario inicio sesión.

Viendo en perspectiva, en realidad no era necesario traer todo el objeto User, debido a que la propiedad username es única para cada instancia de esta clase, solo un string con este atributo es suficiente para identificar correctamente quién inicio sesión.

Creación y manejo de esferas

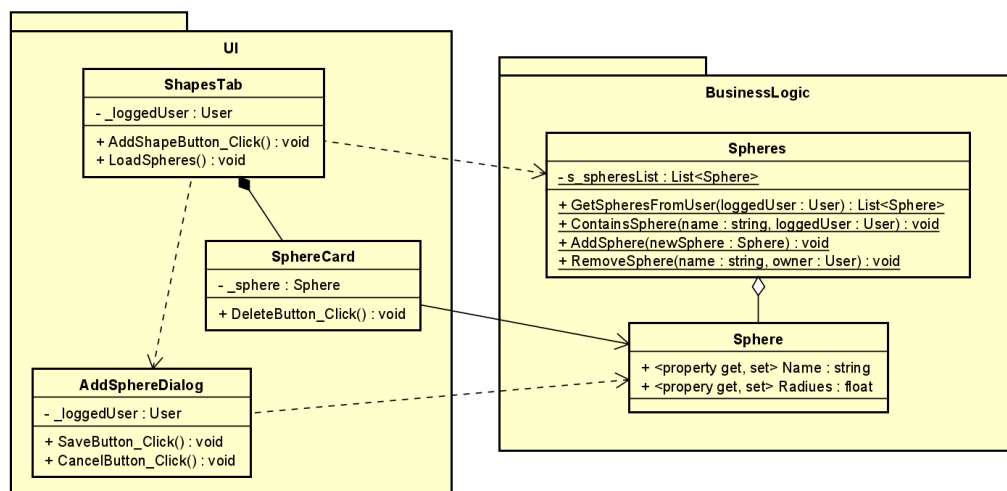
Desde la barra lateral izquierda se debe seleccionar la opción Shapes, que le presentará al usuario un listado de las esferas previamente creadas. En este listado de esferas creadas previamente se incluye un icono de esfera, el nombre de la esfera, y su radio establecido. También se dispone de un botón X que permite al usuario borrar la esfera de su repertorio.

Al momento de crearse esta sección de la aplicación, y al agregar nuevas esferas, la clase de ShapesTab llama a un método de la clase Spheres en BusinessLogic para que le envíe una lista con todas las esferas que sean propiedad del usuario que inició sesión. Recordemos que el usuario lo sabe el MainPage y este se lo comparte a todas las pestañas que nacen de él.

Tomando esta lista de instancias de la clase Sphere, ShapesTab crea para cada una instancia del UserControl llamado SphereCard que muestra los datos relevantes de la esfera.

Además de listar las esferas disponibles, presionando un botón se despliega un diálogo con el que es posible crear una nueva esfera que quedará automáticamente asociado al usuario de la sesión.

El diálogo para crear la esfera recibe en un textBox el string para el nombre de la esfera y en un numericUpDown el valor del radio. Luego intenta asignarlos a un objeto Sphere creado al iniciar el diálogo. La clase Sphere tiene implementada medidas para asegurar la integridad de los datos, por lo que al intentar asignarle el nombre y radio, si no son válidos, atraparará una excepción y le avisará al usuario para que pueda ingresar datos correctos.



Finalmente, si esta esfera está siendo utilizada por un modelo, no podrá ser eliminada del repertorio hasta que se elimine el modelo. Esto es gracias a que la función de RemoveSphere en la clase Spheres implementa un método que investiga si la esfera está siendo utilizada por un modelo existente.

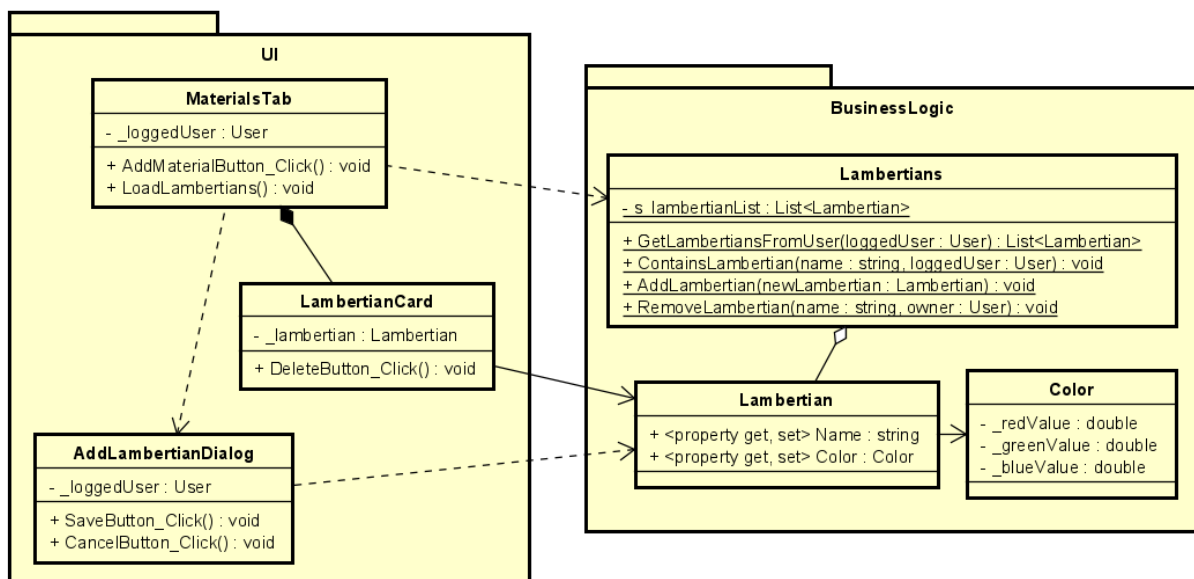
Esto es algo ocurente en las partes del proyecto donde se deben evitar acciones prohibidas para el usuario. Cuando se intenta hacer algo que resulta en una excepción de código, se toma el mensaje de la excepción y se muestra en la interfaz para que el usuario sepa porque su petición no pudo ser realizada.

Creación y manejo de materiales

La pestaña de gestión de Materials sigue la misma estructura ya establecida por la pestaña de Shapes.

Se listan los materiales pertenecientes al usuario que inició sesión, que para esta entrega del proyecto solo incluirá materiales de tipo Lambertiano. Cada material en la lista muestra su nombre, sus valores RGB, un recuadro que muestra el color del material y un botón para eliminarlo de la lista. De la misma manera que con las esferas, no es posible eliminar un material que esté siendo utilizado en un modelo. La interacción entre los paquetes UI y BusinessLogic sigue las mismas interacciones que con Spheres, solo cambiando las clases y forms por sus equivalentes para Lambertians.

En la parte superior se encuentra el botón para crear un nuevo material. Al presionarlo, se muestra un diálogo que actúa de formulario para agregar un nuevo material. Este diálogo también cuenta con medidas de integridad para evitar crear materiales con datos incorrectos.



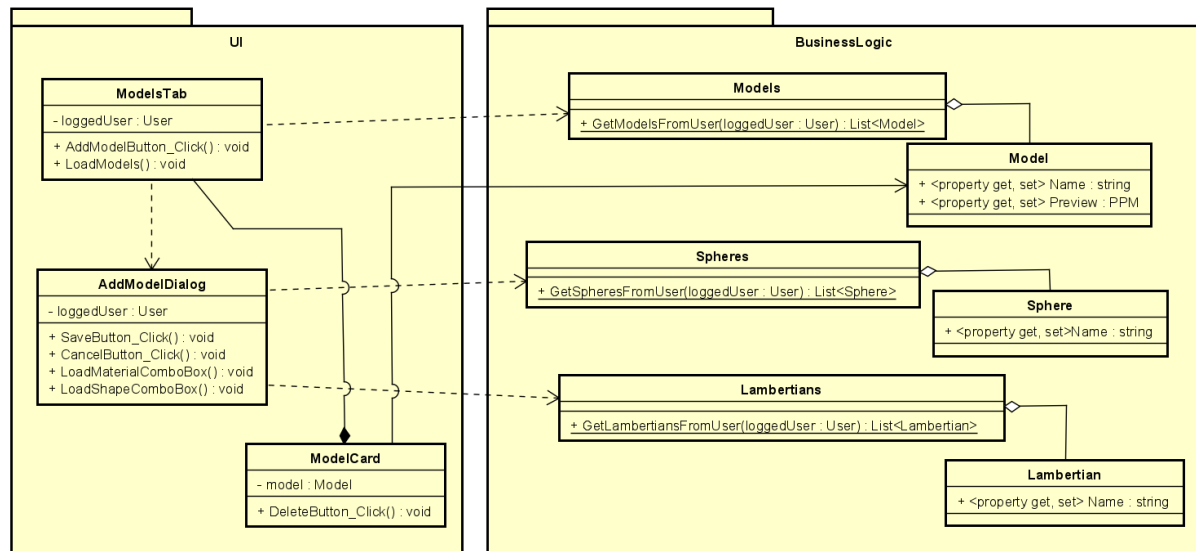
Creación y manejo de modelos

La gestión de modelos continúa con la estructura general establecida por las pestañas previas. Un listado de los modelos disponibles con información relevante de cada uno, cada uno con su botón para borrarlos de la lista, y un botón que despliega un diálogo que permite agregar nuevos modelos.

Los datos para crear un modelo incluyen un string para el nombre como con el resto de objetos, pero además debe guardar una Sphere y un Lambertian de los existentes en la colección perteneciente al usuario. Por eso, en el diálogo de creación de modelo implementamos dos comboBox, uno para las Sphere del

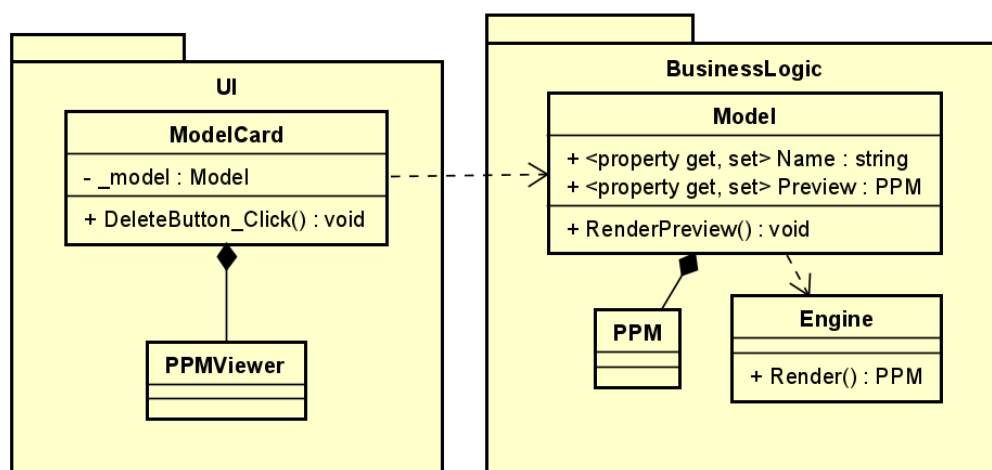
usuario y otro para los Lambertian. Esto es mejor que un textBox en donde se ingresa el nombre porque evita que el usuario deba recordar cuales tiene para elegir y el nombre exacto de cada uno para elegir correctamente.

Estos combos se cargan pidiendo a las clases de agrupamiento respectivas la lista de objetos cuyo owner es el cliente mediante las funciones Get<objeto>FromUser.



Por primera vez, los objetos desplegados tienen dos posibilidades para el gráfico que los representa: al momento de crear un nuevo modelo, el usuario puede optar por generar una imagen de referencia. Si se decide generar la imagen, se utilizará un **userControl** especial llamado **PPMViewer**. Su único propósito es el de ser añadido como un componente a otro **userForm** o **userControl** que muestra una imagen generada a través de un **PPM**.

En especial para la imagen del **modelCard**, el modelo mismo tiene un método **renderPreview** con el que crea una escena por defecto donde el modelo está en el centro de la cámara y luego la manda a ser renderizada por un objeto de la clase Engine. El PPM generado es guardado en la instancia del modelo y es luego utilizado por su **modelCard** para dárselo al **PPMViewer** que finalmente muestra en pantalla la preview del modelo.



Creación, edición y manejo de escenas

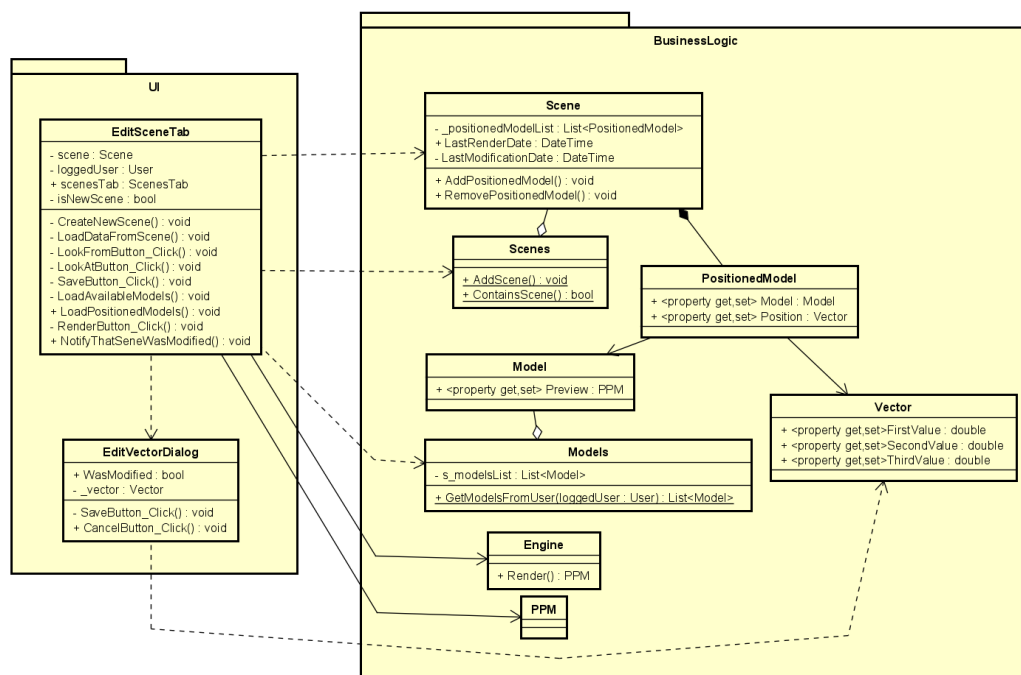
Aunque la pestaña de gestión de escenas es simple como las demás, listando las escenas del usuario, y con su botón para agregar una nueva, vemos también que las escenas listadas tienen un botón para editarlas.

Ya sea que se haga clic en uno de estos o empecemos con una nueva en blanco, la pestaña cambiará a la de edición de escena.

Hasta aquí, venimos construyendo un camino claro. Como si de un árbol se tratase, donde el nodo hoja es nuestro objetivo, renderizar una imagen. Para ello se precisa de cada una de las partes anteriores, que se van juntando para formar una colección de elementos posicionados en el espacio.

La pestaña de edición de escena es la más compleja de todas y la que se comunica con más clases del BusinessLogic.

- Utiliza la clase Models para mostrar los modelos disponibles para añadir a la escena.
- Maneja instancias de la clase Vector para determinar la posición de la cámara, de los modelos en la escena y del punto a donde mira la cámara.
- Combina Models con Vectores para generar PositionedModel que informan al motor gráfico sobre las colisiones de los rayos con superficies.
- Naturalmente usa una instancia de la clase escena para guardar los positioned models y un CamaraDTO con toda la información necesaria para la cámara.
- Utiliza métodos de Scenes para agregar las nuevas escenas al repertorio del usuario.
- Y finalmente renderiza la imagen representativa de la escena pasándole toda la información necesaria a una instancia de la clase Engine. Engine genera el PPM que luego es utilizado para desplegar la imagen en la interfaz.



El Render

El proceso del render es sencillo, se genera un pseudo-archivo PPM mediante su correspondiente clase. Y se genera la cámara mediante el CameraDTO guardado en escena. De aquí comienza un proceso arduo que se repite para cada pixel, y que a su vez, se repite varias veces según la calidad de muestreo que se desee.

Se pide a la cámara que genere un rayo, desde el origen hacia al píxel, este será usado para analizar qué objeto se encuentra más cerca de la cámara. Para cada modelo posicionado de la escena, se le provee de este rayo, para comprobar su distancia.

Decisión de diseño: Para evitar trainwrecks y cumplir con la Ley de Demeter decidimos que el proceso transcurre en una serie de llamadas recursivas, de nombres similares, que van preguntando a sus propiedades internas las mismas llamadas, hasta que el objeto con la responsabilidad de conocer esa información, sea capaz de responder y hacer flotar la información a través de los objetos.

En casos como el Modelo en sí, debe completar a la información del hitRecord que le devuelve Shape con la atenuación correspondiente del Material. Ya que el Shape por sí solo no es capaz, ni su responsabilidad, conocer del Material.

```
1 reference | Santiago, 2 days ago | 1 author, 2 changes
public HitRecord IsHitByRay(Ray ray, double tMin, double tMax, Vector position)
{
    HitRecord hit = Shape.IsHitByRay(ray, tMin, tMax, position);
    hit.Attenuation = Material.Color;
    return hit;
}
```

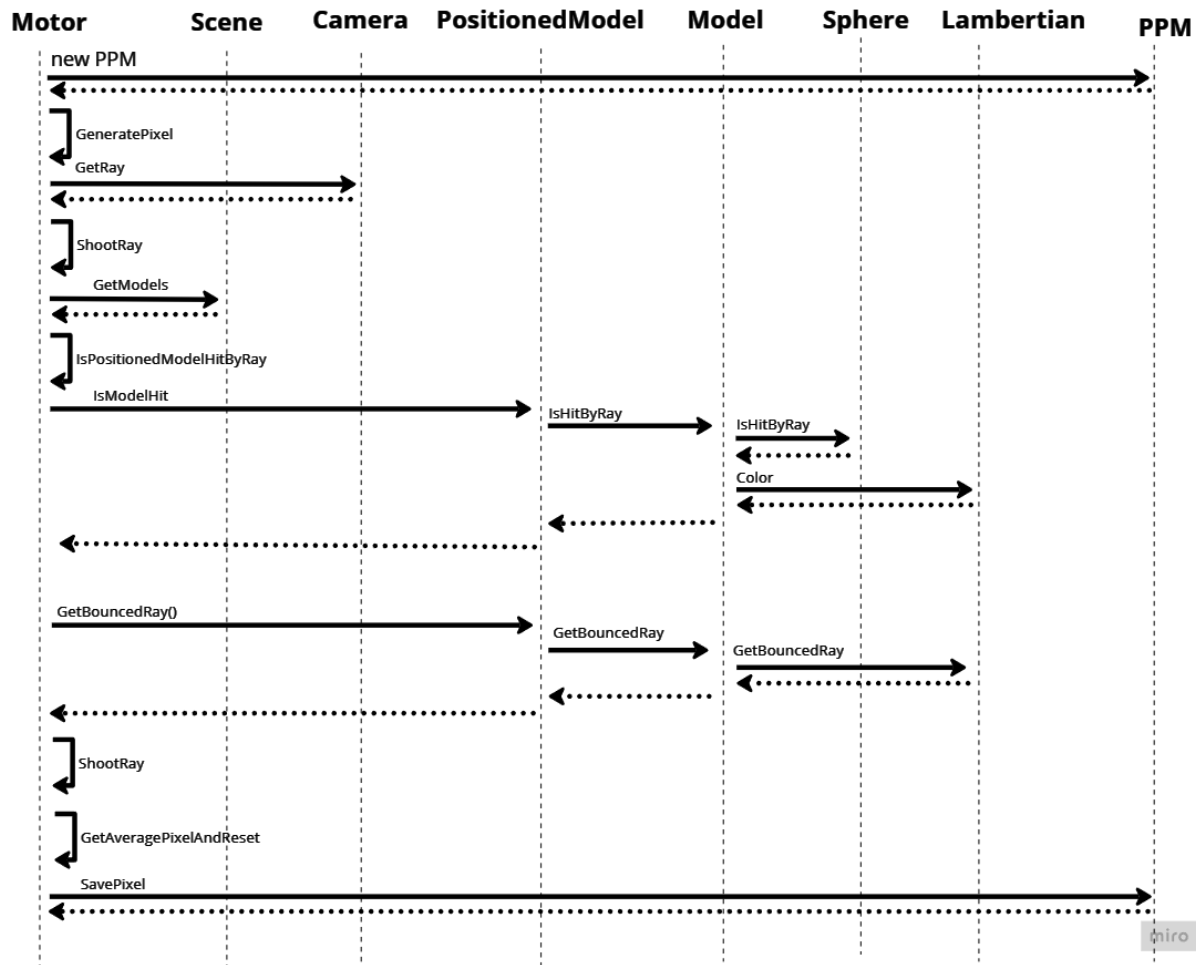
Este diseño surge de la idea de cómo fluyen los paquetes en las capas de redes, donde cada capa agrega su debida información agregando cabeceras al paquete.

Lo mismo hace después para calcular los rebotes de rayos, como rebota un rayo sobre una superficie es capacidad del Material.

Finalmente, como se han lanzado muchos rayos para el mismo píxel, se calcula un promedio de ellos y se guarda.

A continuación presentamos un diagrama de secuencia de este proceso, que deja en evidencia las responsabilidades y una vez más, como buscamos cumplir con la ley de Demeter.

Proceso para renderizar 1 pixel



Pruebas unitarias y cobertura

Para realizar los test unitarios de la lógica de nuestro proyecto, seguimos la metodología de TDD (Test Driven Development). Es decir, el proceso de codificar empezaba por escribir las pruebas que consideramos deben poder pasar nuestras clase, y con solo el código suficiente para que compilen. Ya definidas las pruebas, es cuando comenzamos a implementar el mínimo código necesario para que todas pasen correctamente. Y una vez que las pasen, somos libres de arreglar la implementación para hacerla más eficiente y mantenible.

Creamos test unitarios para cada objeto como Sphere y Lambertian, y para sus clases que los agrupan, pensando en todos los posibles casos en los que podemos utilizar instancias de esas clases.

Para el render se hizo solo una prueba que comprueba si el PPM generado es el esperado. Consideramos que esto era suficiente pues para poder generar el PPM específico todos los métodos involucrados deben funcionar correctamente. El

único ajuste necesario fué el de suplantar el valor aleatorio involucrado en el renderizado por uno fijo, pues sino sería cuestión de suerte que se genere el PPM esperado.

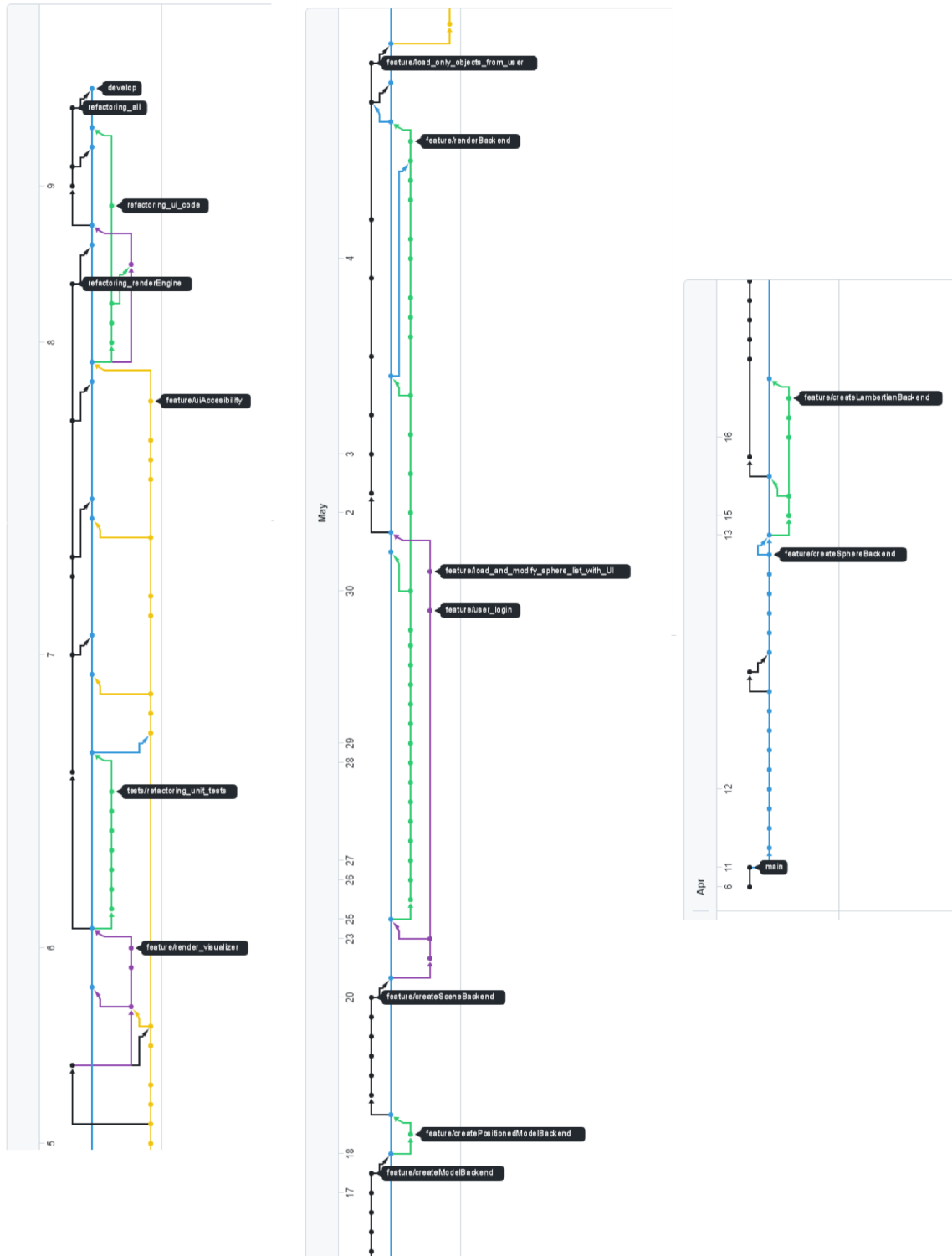
Este paradigma de desarrollo nos llevó también a especificar e implementar varios métodos en las clases que al final no fueron utilizados en el proyecto en general. Métodos como el Equals que fue especialmente implementado para ciertas pruebas o porque pensamos que podría ser necesario más adelante pero que no se utilizaron para el proyecto.

La cobertura de código de las pruebas alcanzó al **91,66%** de las líneas de código, y el porcentaje no cubierto representa métodos de renderizado o no utilizados en el proyecto.

Hierarchy ▲	Covered (%Lines)	Covered (%Blocks)
feder_LAPTOP-22GQOADE 2023-05-10 14_11_04.coverage	94,32%	94,23%
▶ businesslogic.dll	91,66%	93,14%
▶ businesslogic_tests.dll	96,36%	95,52%

Anexo

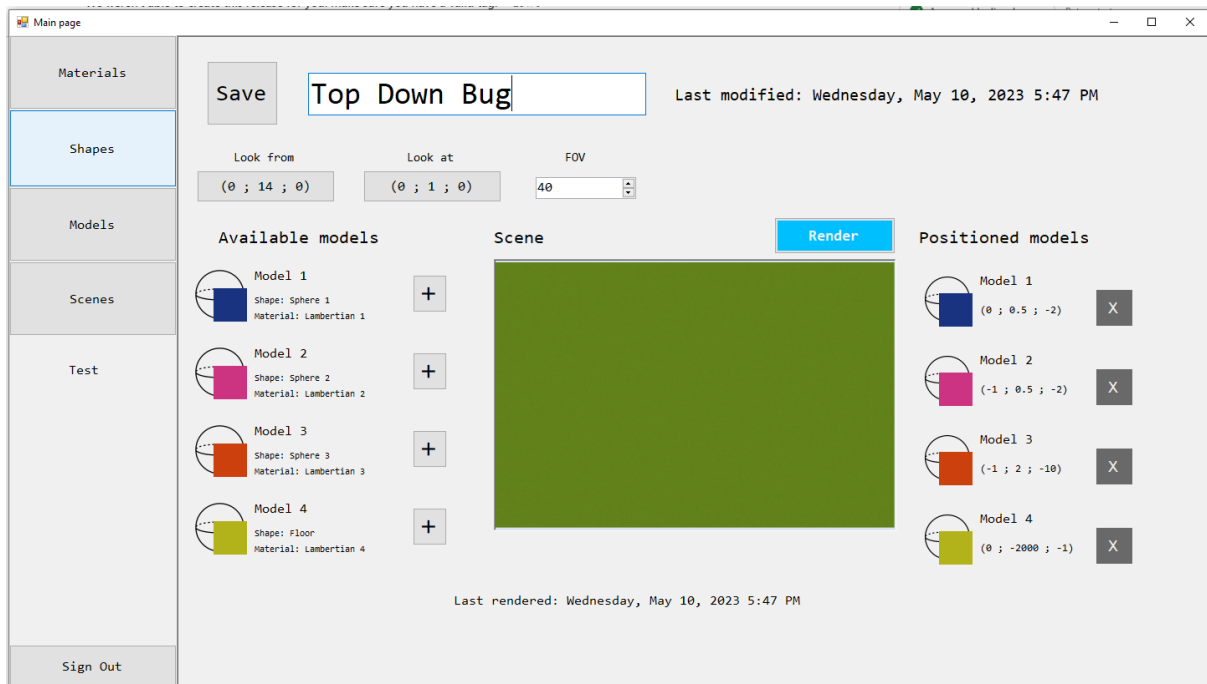
Anexo 1



-Muestra del uso de ramas git flow (Previo al merge main)

https://github.com/ORT-DA1-2023/266628_255981/network

Anexo 2



-Error de renderizado al mirar directamente desde arriba al centro de una esfera, independientemente de la distancia o FOV, toda la imagen es pintada del color de la esfera.

Video Demostrativo de los casos de uso:

<https://www.youtube.com/watch?v=Hgj6rBsb7zI>