Universidad ORT Uruguay
Facultad de Ingeniería

Diseño de aplicaciones 1
Segundo obligatorio

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

Alfredo Fernández - 275898
Manuel Morandi - 271568

Tutores:
Gastón Mousqués
Facundo Arancet
Franco Galeano

2023

ÍNDICE

Descripción general

A grandes rasgos, el objetivo de la tarea sigue siendo el mismo que en la entrega anterior. Se busca desarrollar un renderizador 3D a partir del algoritmo de Ray Tracing, usando los patrones, principios y buenas prácticas de diseño vistos en clase y utilizando TDD como metodología principal.

En particular, esta nueva entrega genera un par de funcionalidades nuevas a la aplicación, siendo estas la incorporación de un nuevo material metálico que refleja sus alrededores, la posibilidad de activar un efecto de desenfoque en la cámara y la opción de descargar los renders realizados en alguno de los tres formatos disponibles (ppm, jpeg y png).

No obstante, la principal adición a la aplicación es la persistencia de datos. Para desarrollar esta funcionalidad se usó Entity Framework, un ORM (object-relational mapper), que construye una base de datos relacional en SQL Server a partir de los objetos definidos en el dominio de nuestra aplicación.

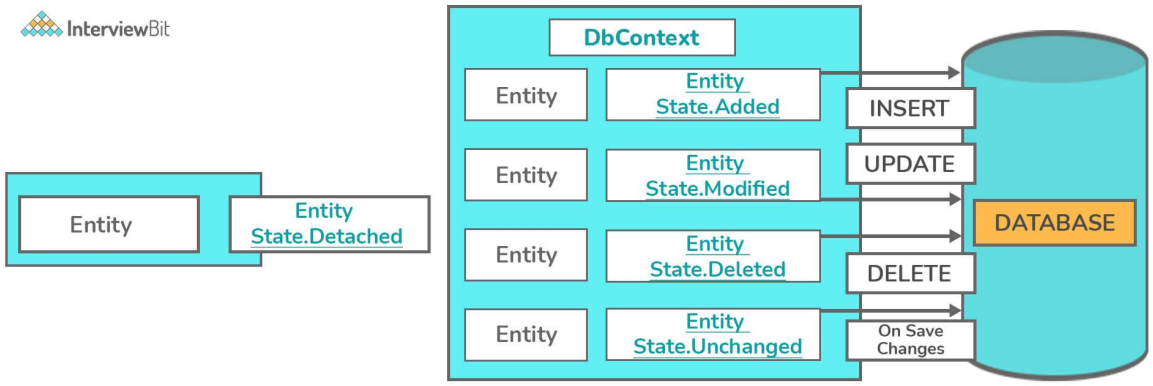
Entity Framework

Entity Framework es un ORM desarrollado por Microsoft para .NET. Con afán de contextualizar, se explicarán los conceptos básicos de esta tecnología.

El elemento más fundamental de EF (Entity Framework) son las entidades. Estas son clases que se mapean a la base de datos. Dentro de estas se definen atributos que luego serán las columnas de la tabla, mientras que cada instancia creada y guardada de una entidad se materializa como una fila.

Los atributos, llamados propiedades, pueden ser de dos tipos. Los escalares son los primitivos, los “originales” del lenguaje (int, string, DateTime, entre otros ejemplos). Adicionalmente encontramos las referenciales que, hablando mal y pronto, son aquellas que almacenan aquellas clases del dominio.

Estas entidades y sus atributos tienen estados, pudiendo encontrarse en estado Added, Modified, Deleted, Unchanged o Detached. El estado se actualiza al llevar a cabo operaciones sobre la base de datos. Sin embargo, la base de datos en si no se actualiza hasta llevar a cabo la operación SaveChanges(). El siguiente esquema ejemplifica este funcionamiento:



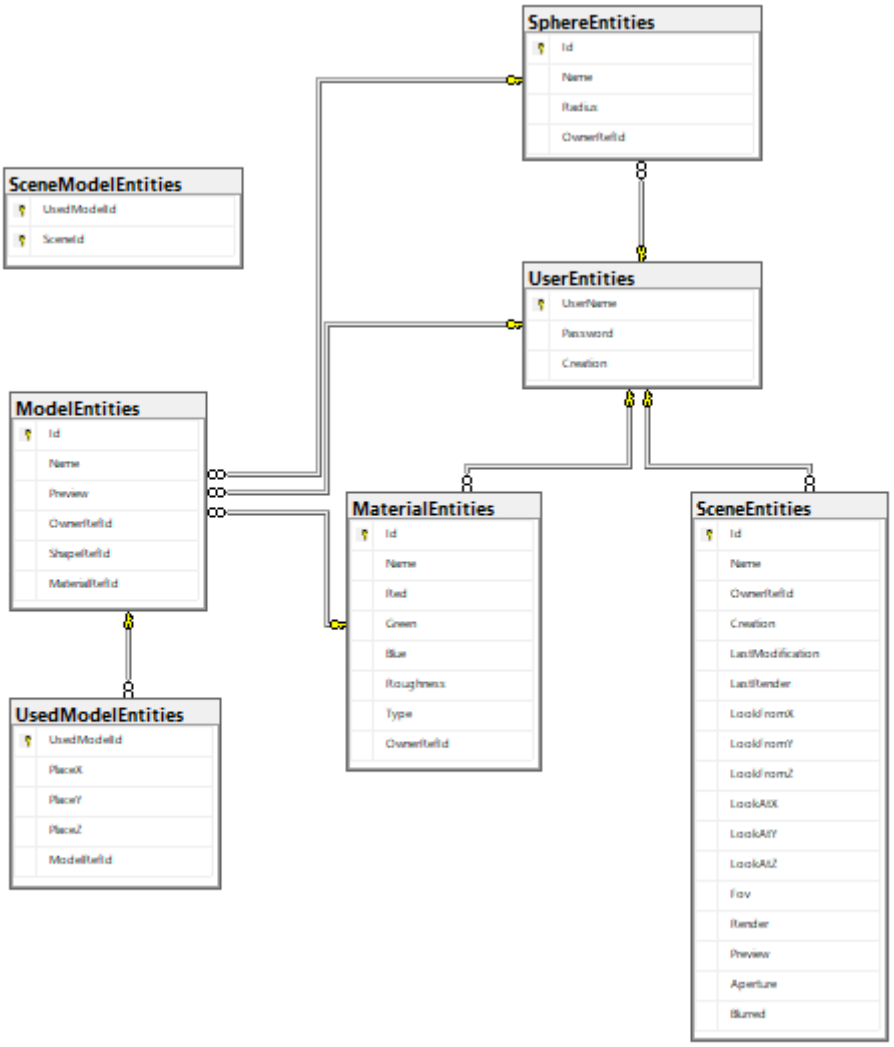
Entity State in Entity Framework

Esta operación es propia de la clase Context. A su vez, esta clase es la utilizada para definir las distintas tablas de nuestra base de datos. Cada una de estas se define como un DbSet de la entidad deseada. Hay distintas maneras de trabajar con la base de datos, pero nosotros trabajamos con un escenario conectado, por lo que tenemos un contexto que siempre está abierto, haciendo uso de él cada vez que es pertinente.

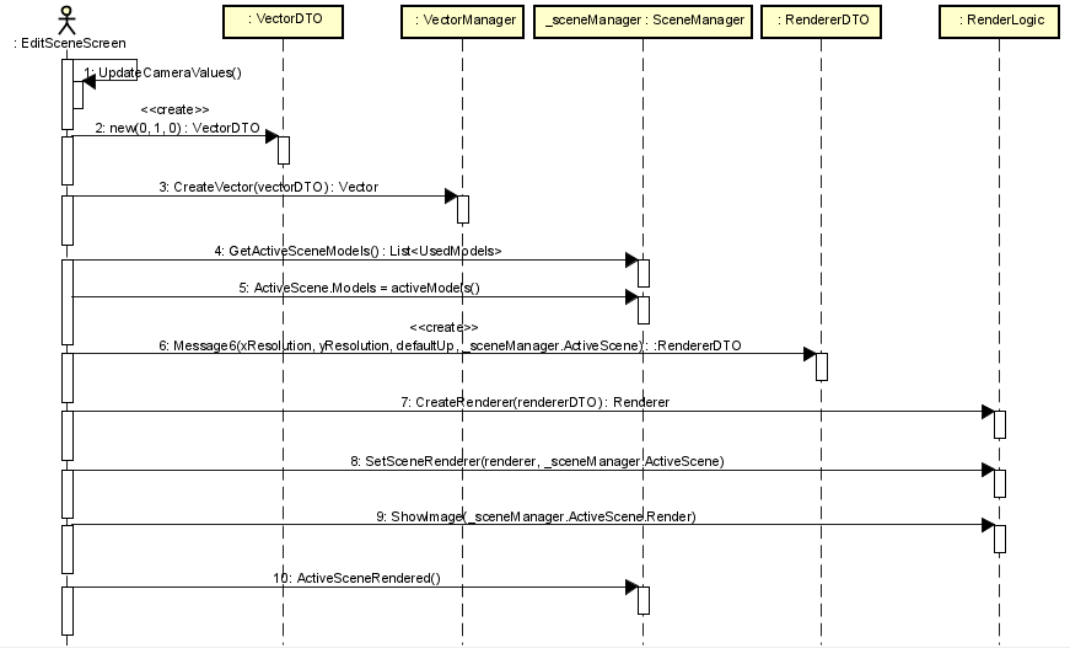
A la hora de definir la base de datos, hay tres enfoques distintos. El primero es el llamado model-first, en el que comenzamos haciendo la diagramación UML del sistema. El siguiente enfoque es el DB-first, en el que codificamos a partir de una base de datos ya establecida. Finalmente encontramos el enfoque que fue utilizado en nuestro caso, el code-first, a partir de un código ya implementado, se genera base de datos.

Un último punto relevante para contextualizar es como se deben encarar las herencias a la hora de representarlas como tablas. de nuevo, hay 3 enfoques: TPH, se establece una gran tabla con todos los atributos posibles; TPT, tabla para el padre con la información común entre los hijos, junto con tabla para cada uno de los hijos; y TPC, cada hijo tiene su tabla y los datos compartidos se repiten, sin tabla para la superclase. En esta instancia, para la clase Material (que tiene subclases Lambertian y Metallic), se utilizó TPH, implementando entonces una gran tabla Materials con todos los atributos posibles. Esto implica que ciertas entradas de la tabla sean nulas o incoherentes, ya que los datos no se utilizan (por ejemplo, si se guarda un material lambertiano, su roughness será -1, que es un valor inválido pero eso no afecta, ya que no se utilizará nunca).

Una vez explicados los aspectos básicos de EF, se puede aclarar que se creó una entidad para cada objeto a persistir, junto con una tabla (DbSet) para poder guardarlos. De esta manera, tenemos entidades y tablas para Material, Model, Scene, Sphere, UsedModel y User. Se adjunta el modelo de tablas implementado:



Cabe aclarar que, como se puede notar en el diagrama, para asignar UsedModels a escenas, se tomó un camino poco ortodoxo. Frente a problemas a la hora de implementarlo de manera convencional, se decidió agregar una nueva tabla auxiliar, que solamente guarda la clave primaria del modelo usado (un int autogenerado por el sistema) y la de la escena a la que pertenece (un string con el nombre del propietario y el nombre de la escena). En otras palabras, la nueva tabla representa que modelo se colocó en que escena. De esta manera, se logra mapear que un modelo pertenece a una escena de manera normalizada y funcional. El UsedModel no sabe a qué Scene pertenece y una Scene no sabe que UsedModel tiene hasta el momento de la renderización, ya que la conexión entre ambos se hace a nivel aplicación de manera lógica, y la tabla SceneModelEntities brinda toda la información necesaria para llevar a cabo la asignación de modelos en escenas de esta manera.

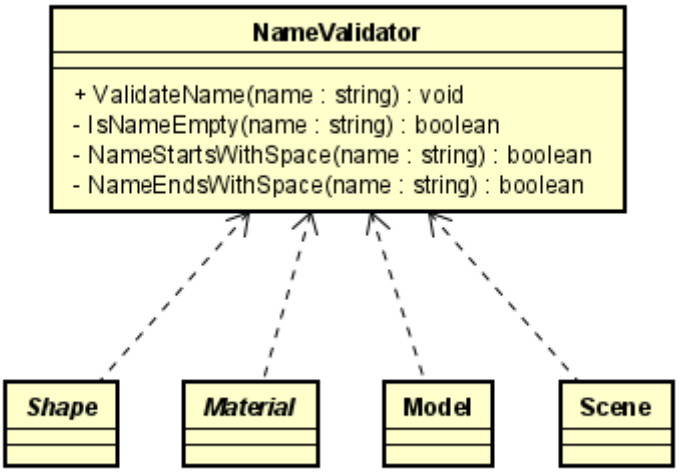


Dicho de otra manera, la lista de UsedModel que tiene cada Scene no se persiste como tal en la base de datos, debido a que en su lugar se tiene una nueva tabla que guarda la asociación entre estas clases. Como se nota en el diagrama anterior, al tocar el botón de renderizar escena, los modelos pertenecientes a esa escena son seleccionados de la base de datos y la lista Models de Scene es sustituida por la lista obtenida con esa consulta. Posteriormente, la escena es renderizada nuevamente con menor resolución, para conseguir así el preview de la escena (esta última parte del método no fue representada en el diagrama, ya que representa solo la renderización de la imagen principal).

Aclaraciones de diseño

Antes de comenzar a ver la implementación a mayor detalle, conviene explicar algunas decisiones de diseño que resultan muy generales para incluir en un apartado específico.

La primera decisión es sobre algo de lo que se habló como posible mejora en la documentación de la entrega anterior. Los nombres de materiales, figuras, modelos y escenas debían cumplir los mismos requisitos, lo que significaba que la implementación de métodos de validación de nombres estaba repetida, ya que se implementó una vez para cada clase distinta. Esta reiteración no es conveniente, ya que un cambio en los requisitos implica cambiar cada una de las implementaciones, modificando entonces varias cosas aunque el cambio sea el mismo para todos. Para solucionar esto, se incorporó una nueva clase NameValidator, cuyo único propósito es verificar que los nombres dados a los objetos de nuestro sistema sigan las reglas establecidas.



Esta clase no representa un objeto real del dominio, por lo que es fabricación pura, fue creada para cumplir su propósito, sin tener que modelar algo real. Ahora, al verse en la necesidad de validar un nombre en cualquiera de las clases mencionadas anteriormente, se instancia un objeto de NameValidator que recibe el string y, en caso de no ser válido, este lanza una excepción.

Otro aspecto importante que cambió fue el uso de un controlador de fachada. En la anterior entrega, se tenía una clase, llamada AppLogic, que servía como intermediario entre las clases de bajo nivel (UI principalmente) y las de alto nivel (dominio, renderizador, etc.). Esta clase era muy extensa y poco cohesiva, ya que, al servir como barrera para que la UI no accesara al dominio, debía proveer a las clases de bajo nivel todo aquello que puedan necesitar de las de alto nivel. En otras palabras, AppLogic brindaba todos los servicios que la UI requería.

Esta clase se fue “eliminando” naturalmente al implementar la base de datos con EF. Al no necesitar las listas de objetos que se utilizaron en la instancia anterior y al tener que implementar nuevos métodos para acceder a la base de datos, ese controlador de fachada se fue dividiendo en varios controladores, cada uno enfocado en un objeto específico. De esta manera, todos los servicios que la UI puede requerir relacionados con los materiales, se pusieron en una clase MaterialManager, que tiene como única responsabilidad proveerlos. Lo mismo aplica para Sphere, Model y el resto de clases. Adicionalmente, las funciones del renderizador también se pusieron en un controlador aparte, separándolo del resto de managers.

Este cambio no fue implementado de un momento a otro, ya que hacerlo de manera drástica significa hacer que el sistema entero se rompa, dado que las funcionalidades programadas en AppLogic sencillamente dejarían de existir. Debido a esto, se fue implementando de manera paulatina. Al crear un nuevo controlador, este era programado e incluido en todas las pantallas donde fuera necesario, manteniendo la dependencia a AppLogic. Una vez todos los controladores fueron terminados y se hallaban funcionales, se eliminó completamente la clase AppLogic.

Esta decisión trae también otro beneficio. Ahora, cada pantalla solo requerirá y dependerá de los controladores que use, ignorando al resto. En la anterior entrega, la pantalla que muestra las figuras dependía de todo AppLogic, conociendo también los métodos de Scene, pese a que no le es necesario. Esa misma pantalla ahora solo depende del controlador de usuarios, esferas y renderizador, ya que es todo lo que requiere para funcionar.

Nuevos requerimientos

Antes de comenzar a tratar las nuevas funcionalidades implementadas, se quiere reportar un error del sistema que no se pudo solucionar. Al intentar renderizar una escena colocando la cámara desde arriba o desde abajo, paralelo al eje vertical, el sistema lanzará una excepción ya que no se puede dividir entre 0. La aplicación no se cae de manera repentina ni funciona de manera inadecuada, simplemente muestra un cartel que indica el problema y no renderiza. Es un caso borde que no fue solucionado.

Exportación de imágenes

La implementación de este requerimiento (RF22) surgió de manera natural debido a como se almacenaban los renders obtenidos desde la entrega pasada.

A la hora de seleccionar el directorio, se usó la clase FolderBrowserDialog de C#, que con el método ShowDialog() abre una ventana que invita al usuario a seleccionar un directorio. El único aspecto a cuidar es que si el usuario cierra esta ventana sin seleccionar un directorio, se guardará un directorio nulo, por lo que se dará un error al intentar escribir.

Al renderizar una imagen, se guarda como string en formato ppm. Debido a esto, si se solicita una descarga en formato ppm, simplemente se genera un archivo con ese string en el directorio seleccionado.

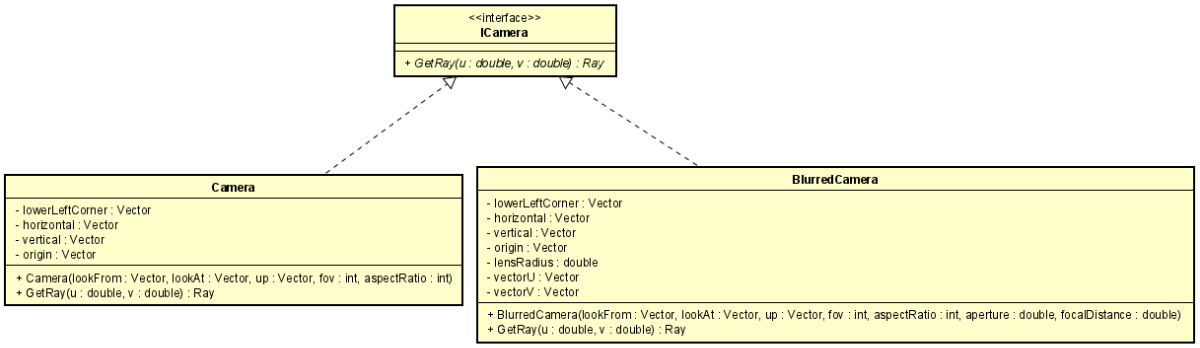
Si, por el contrario, se solicita otro formato, la solución se basa en crear un Bitmap (clase de C# que representa la información de los píxeles de una imagen) a partir de la imagen ppm. La clase Bitmap tiene una función Save, que dado un directorio y un formato, guarda la imagen de la manera solicitada. Debido a esto, la descarga de imágenes png y jpg se encaró a través de métodos de la clase Bitmap.

Es verdad que esta solución puede no ser la mejor desde el aspecto de expansión. Hubiera sido mejor utilizar lo planteado por el patrón Strategy dado en clase, para poder encapsular cada uno de los algoritmos de exportación de imagen para cada tipo. Así, si se agrega un nuevo tipo, simplemente se puede recurrir a su algoritmo, ya que serían intercambiables. No obstante, nuestra solución es aceptable y funciona correctamente.

Incorporación de nuevos elementos

Los otros dos requerimientos de este trabajo solicitan incorporar la posibilidad de utilizar una cámara con desenfoque y la aparición de un nuevo material metálico. Para llevar a cabo estos pedidos, se utilizó polimorfismo.

Veamos primero el caso de las cámaras. Se creó la interfaz ICamera, que tiene el método GetRay. Luego, existen dos clases, Camera y BlurredCamera, que implementan esta interfaz. Cada tipo de cámara implementa su constructor y GetRay, además de tener sus propios atributos (como el valor numérico LensRadius de BlurredCamera, no presente en Camera) de la manera en la que se hace en el código brindado por el profesor.



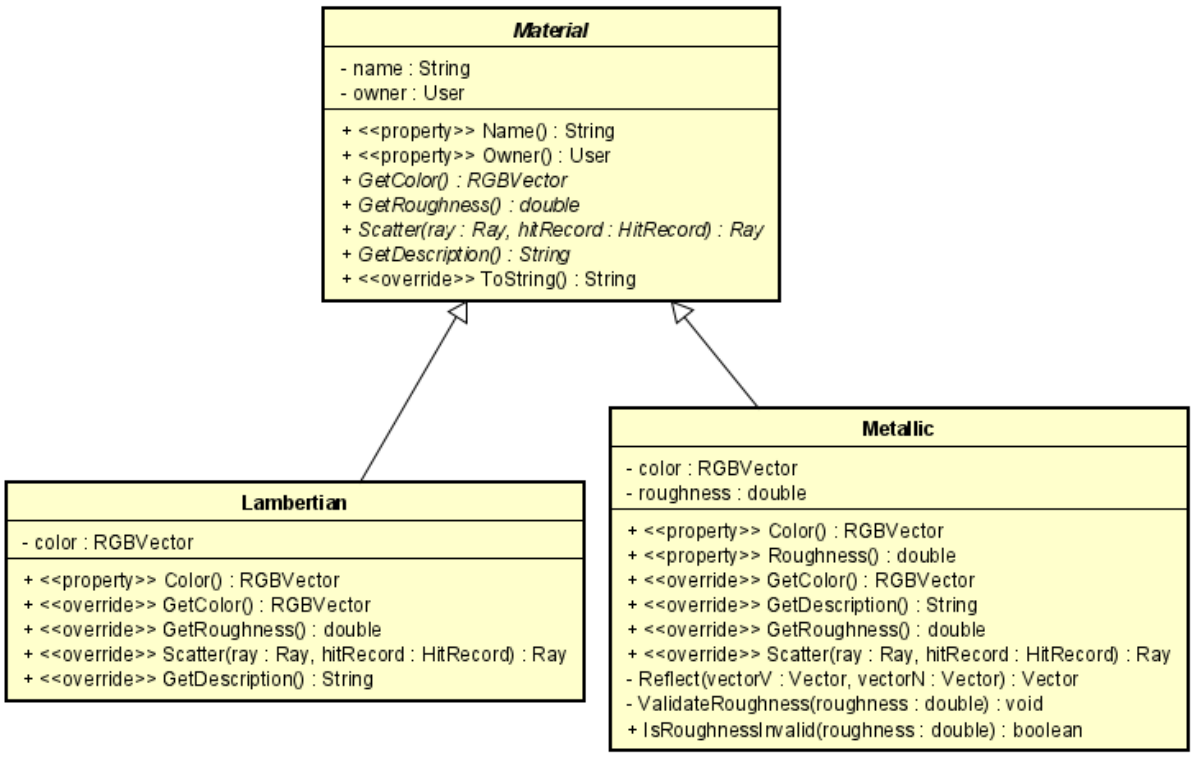
Gracias a esta nueva interfaz, se logra que cada objeto de la clase Renderer tenga un objeto que implemente ICamera. Esto tiene como implicancia que cada Renderer puede tener o bien una cámara normal (como la implementada anteriormente) o una cámara con desenfoque.

La escena tiene un atributo booleano que indica si la cámara debe desenfocar o no. Al crear un Renderer, dentro del constructor, nos fijamos si ese booleano está activo. En caso de estarlo, creamos una BlurredCamera, en otro caso una Camera; y en cualquier caso esta es asignada como el elemento ICamera del Renderer creado. Cabe destacar que toda la información requerida para la creación de cualquier tipo de cámara se encuentra almacenada dentro de la propia escena.

De manera similar, se utilizó polimorfismo para implementar el material metálico. En lugar de usar una interfaz, se dispuso de una clase abstracta Material. Cada objeto de esta clase tiene un nombre y un propietario. Además, sus subclases deben implementar ciertos métodos necesarios para el correcto funcionamiento del renderizador.

Esta clase tiene entonces dos subclases, Lambertian y Metallic. Cada una de estas tiene sus propios atributos, como color y difuminado en el caso de Metallic. Además, implementan los métodos abstractos de la superclase, tales como GetDescription, que devuelve el texto que describe el material y que se muestra en

la pantalla de lista de materiales del usuario; y Scatter, que representa como los rayos se distribuyen y rebotan al impactar con ese material.

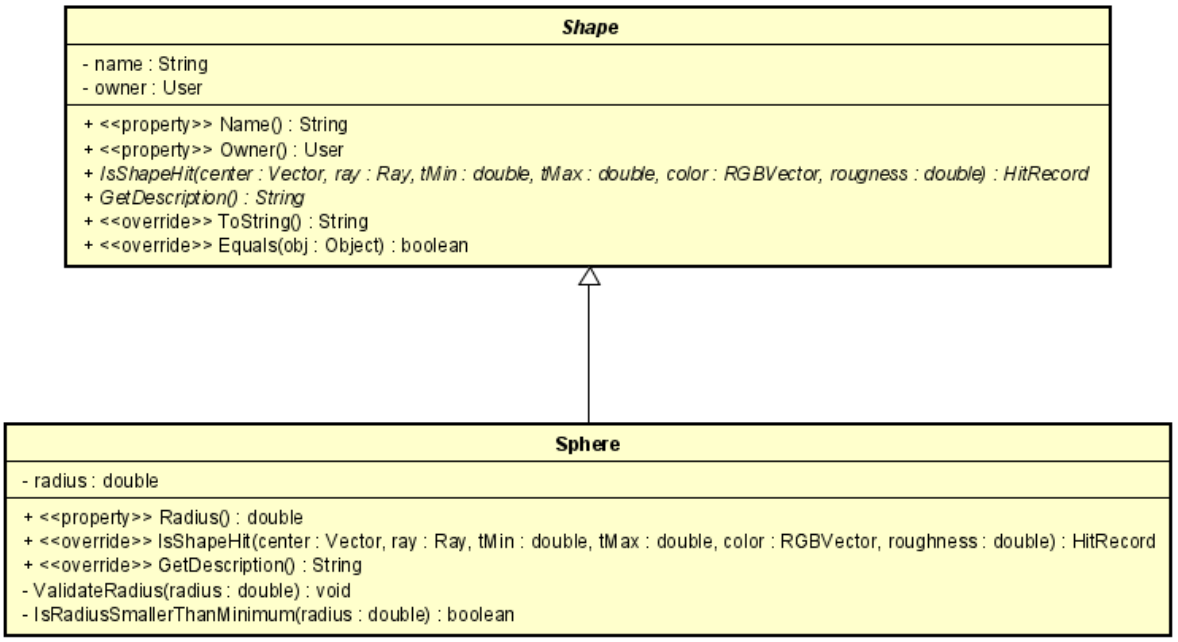


Una crítica a la manera en la que se decidió implementar este polimorfismo y que podría ser una mejora a futuro es que la superclase material obliga a las subclases a implementar métodos que tal vez no deberían. Por ejemplo, Material tiene un método GetRoughness que devuelve el difuminado de un material. Este método debe, obligatoriamente, ser implementado por Lambertian, a pesar de que este no tiene difuminado. Esto rompe el principio de sustitución de Liskov, ya que las subclases no se comportan como la superclase. En nuestro caso, GetRoughness de un material lambertiano devuelve -1, un valor inválido que no se va a usar nunca, por lo que no presenta ningún inconveniente, pero es algo que debería ser modificado.

Esto de poner valores incoherentes o inutilizados se realizó también en HitRecord, ya que se agregó el atributo roughness para referenciar el difuminado. Al estar trabajando con un lambertiano, este atributo tomará un valor incoherente que nunca será utilizado. De esta manera, se ahorra la creación de un HitRecord específico para cada tipo de Material.

Volviendo al punto anterior, Scatter es una función muy importante que todos los materiales deben implementar, ya que enuncia el comportamiento de los rayos al impactar con el material. Este método es utilizado en la clase Renderer para poder renderizar las escenas solicitadas. Algo similar sucede con Shape, ya que su método IsShapeHit es implementado por todos sus subclases (Sphere únicamente en esta entrega, pero queda abierta la puerta a extensión) y permite saber al Renderer cuando un rayo se intersecta con una figura. Los métodos Scatter y IsShapeHit varían según cada material y figura respectivamente.

Para mostrar las similitudes entre la implementación de los materiales y las figuras, veamos también cómo se implementó la figura:



Vemos que en el caso de la figura no se rompe LSP, como si se hace con Material. Toda subclase de Shape se comporta como la superclase, dejando abierta la posibilidad de introducir nuevas figuras sin tener que modificar el código. Esto cumple el principio Open-Closed presentado por SOLID, ya que es abierto a extensión y cerrado a la modificación.

Manejo de errores

En la primera parte del obligatorio manejamos las excepciones en una misma clase, BussinesLogicException. Este tipo de excepción la usábamos tanto para el dominio como para la lógica de la aplicación.

En esta segunda parte del obligatorio para mejorar el manejo de excepciones y representar más acertadamente lo que significa cada una decidimos implementar una clase de excepción distinta por cada paquete, para que representen errores propios de cada capa y poder manejar las excepciones de manera más acertada, solo capturando las que nos interesan. Debido a que al implementar entity framework, la clase AppLogic “desapareció”, BusinessLogicException es el tipo de excepción utilizada en el dominio, o la lógica de negocio solamente. En la UI utilizamos la excepción UserException, con respecto a la base de datos se usó DataBaseException. Por último para los servicios se utilizó ManagerException.

Decidimos que las excepciones capturen las mismas lanzadas por niveles anteriores, lanzando una nueva excepción en ese nivel con el mismo texto de la

anterior. Esto tiene como objetivo lograr independencia entre los niveles en las excepciones.

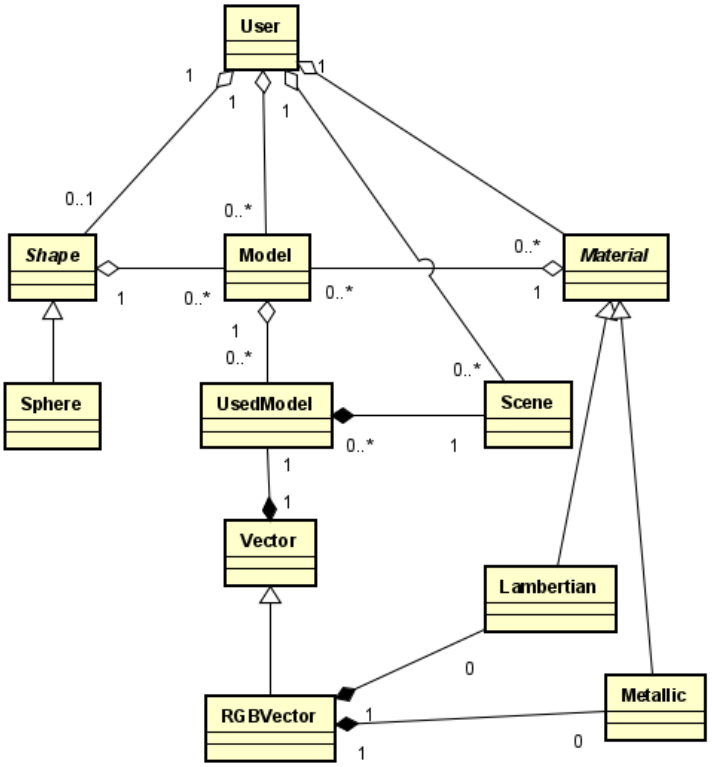
La mayoría de errores que pueden llegar a ocurrir son capturados en algún punto de la ejecución. Sin embargo, existen errores que ocurren en lugares donde no serían capturados, provocando un cierre abrupto de la aplicación. Para evitar errores de la aplicación e informar de una forma no tan brusca al usuario, se utiliza una excepción para la desconexión inesperada de la base de datos. Puede ser tanto porque la misma se haya caído o porque se haya desconectado de cualquier forma que no debería de haber pasado. Por ejemplo, que físicamente la base de datos tenga una falla o que el software dé fallos. En estos casos, se informa al usuario por medio de una ventana emergente, un cartel de aviso, y luego para evitar que se siga utilizando la aplicación con errores, la misma se cierra. Esta excepción debe ser capturada en el main de nuestro proyecto ejecutable, la UI en nuestro caso.

Interacción entre clases

La aplicación se dividió en múltiples 5 proyectos, cada uno con su propósito. El primer proyecto a mencionar es el de Tests, casa de las pruebas unitarias de cada clase de la aplicación. Se verá en el siguiente punto.

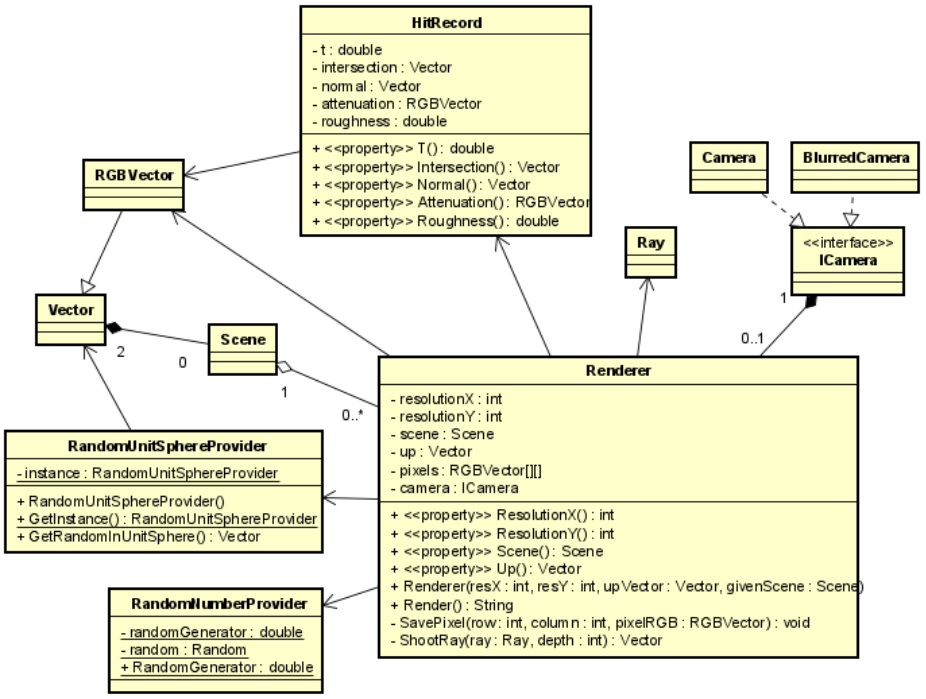
Domain se mantiene muy similar a la entrega anterior, siendo el lugar de almacenamiento de todas las clases básicas del sistema. Brinda todas las clases fundamentales para el correcto funcionamiento del renderizador. Al igual que en la entrega anterior, se puede imaginar una suerte de separación lógica entre las clases como Shape y Model, que atribuyen elementos de la escena; y clases como Renderer o HitRecord, cuya responsabilidad está más estrechamente relacionada a la pura renderización de escena. Debido a esto y a la simplicidad que esto trae a la hora de enseñar diagramas, se dividirá la explicación del funcionamiento del dominio en dos partes.

Ahora bien, el comportamiento de esta capa es bastante similar a como lo fue en el pasado, por lo que no vale la pena detenerse mucho en su funcionamiento. No obstante, veamos los nuevos diagramas y cómo estos fueron alterados desde la entrega anterior. Cabe destacar que para el siguiente par de diagramas, las clases que no sufrieron cambios en sus atributos o métodos en comparación con la entrega anterior, sus atributos y métodos no serán detallados, para simplificar el esquema.



Podemos notar que este diagrama es ahora mucho más simple y existe una cantidad significativamente menor de acoplamiento que el equivalente de la entrega pasada. Esto se debe principalmente a la ausencia de AppLogic, ya que esta clase producía mucho acoplamiento. Como novedad, vemos la incorporación de Metallic, que ya se detalló en puntos anteriores.

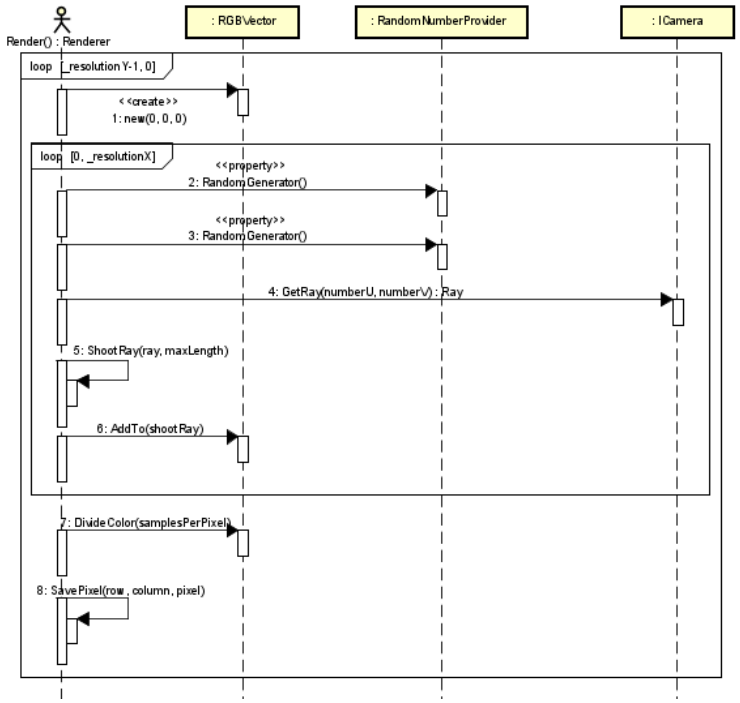
Por su parte, el aspecto de renderizado sufrió bastantes cambios:



Se crearon clases que brindan al renderizador números y esferas unitarias aleatorias, funciones que antes cumplía el propio Renderer y que ahora ya no son

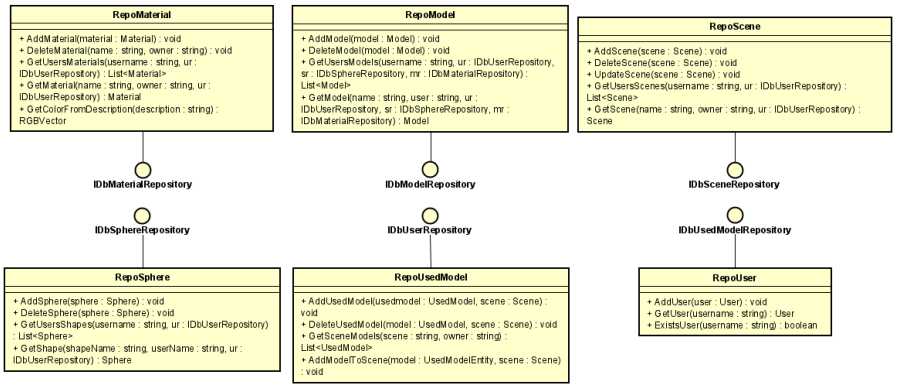
su responsabilidad. También notamos la incorporación de múltiples cámaras a través de la interfaz ICamera, de lo que ya se habló anteriormente. Se observa además la adición de un atributo de roughness al HitRecord que, como se dijo con anterioridad, es usado únicamente si el material es metálico, ya que se obvia si es lambertiano.

Dejando de lado esos cambios, la principal diferencia es que ahora el Renderer tiene menos responsabilidades, ya que la escena y la cámara conocen su información pertinente y es responsabilidad de cada figura y material saber cómo debe renderizarse. También, Renderer ya no requiere de generar números y esferas unitarias aleatorias, ya que clases especialistas en eso lo hacen. Esto libera a Renderer de responsabilidades y lo deja con una única responsabilidad: renderizar imágenes. Gracias a esto, y desde un punto de vista de los principios SOLID, el diseño podría ser considerado más prolijo, ya que ahora cumple el principio de responsabilidad única. Puede ser conveniente ver el siguiente diagrama de interacción para entender cómo se relaciona la clase a la hora de renderizar una escena:



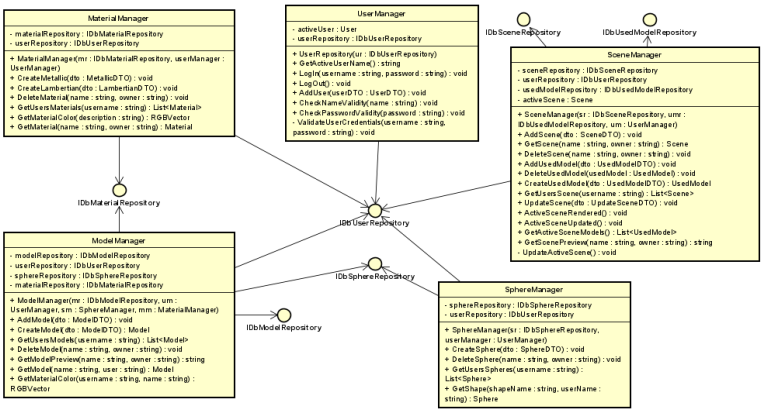
El siguiente paquete es el llamado DataAccess, que es el punto de acceso del sistema a la base de datos a través de EF. En este paquete encontramos todos los entities utilizados para almacenar los objetos. También se encuentra aquí la clase DBContext, que establece todas las tablas de la base de datos. Se incorporan múltiples clases, llamadas repositorios. Estas clases, de las que hay una por clase a persistir, tienen la función de brindar al sistema todos las operaciones que se podrían llegar a llevar a cabo en relación a la base de datos. Para ejemplificar, encontramos RepoUser, que implementa todos los métodos necesarios para trabajar con objetos de la clase User, tales como agregar usuario, recuperar usuario o un método que devuelve true si el usuario ya existe.

Todos estos repos implementan interfaces. De nuevo, existe una interfaz por cada clase a persistir, y esta presenta todos los métodos que luego son implementados por los repositorios. Estas interfaces, ubicadas en la capa Services, nos permiten ocultar el resto de métodos, permitiendo que las clases de más bajo nivel no puedan acceder a los métodos que no requieren y así abstraer la implementación y la estructura interna. Esto es un claro caso del principio de segregación de interfaces (ISP) propuesto en SOLID, ya que conseguimos que la UI y los servicios (clases cliente) solo dependan de aquellos métodos que invocan, y no del resto.



Además de las interfaces de repositorio, este paquete tiene otras dos cosas. Por un lado, guarda todos los DTOs. Estos objetos de transferencia de datos tienen la información mínima e indispensable para llevar a cabo la creación de un objeto específico. Gracias a estos objetos, logramos independizar la UI del dominio, cosa altamente conveniente. Por ejemplo, si se registra un usuario en la pantalla de Sign Up, la UI no creará el objeto de User, ya que generaría dependencia con Domain. En su lugar, crea un UserDTO, que luego es envíado al controlador de User para que este cree el objeto User, sirviendo como intermediario.

Recién se habló de controlador, que es el otro apartado de objetos que guarda Services. Cada posible caso de uso tiene un controlador asignado, que se ocupa de realizar las operaciones y crear los objetos, sirviendo como intermediario entre UI y las capas de mayor nivel, ocultando la implementación al usuario. De esta manera, existe un UserManager que, haciendo uso de un objeto que implemente IDbUserRepository (RepoUser en nuestro caso), brinda a la UI todas las operaciones que puede llegar a necesitar relacionadas con los User (crearlos y almacenarlos en la base de datos, establecer qué usuario está logueado, validar las credenciales, etc.). Lo mismo sucederá con Material, Scene y el resto de las clases.



Existe también otro controlador, llamado RenderLogic, que se ocupa de proporcionarle a la UI todas las operaciones relacionadas con el Renderer que pueda llegar a necesitar, sin que la UI se comunique directamente con la clase. Si la UI necesita establecer la preview de una escena, llamará a un método de RenderLogic que renderizará la imagen y la definirá como preview.

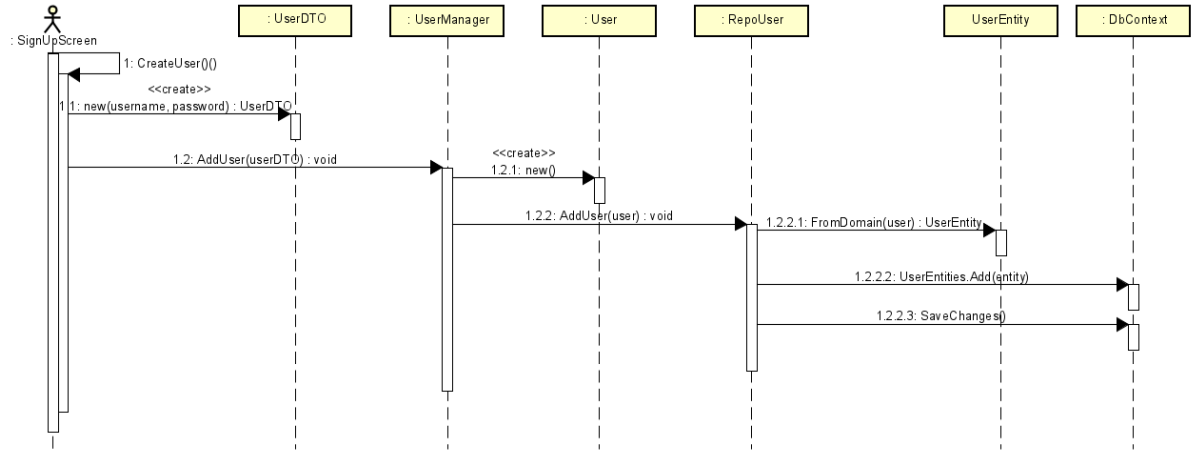
Como ya debe haberse entendido a este punto, Service es un punto de acceso a las clases de mayor nivel para el paquete UI, el de menor nivel y con el que interactúa el usuario. Este paquete está formado por todos los recursos (imágenes) que utiliza, las distintas pantallas y los componentes que las forman, junto con la ventana principal y la interfaz que nos permite pasar de una pantalla a la otra (explicado en la anterior entrega, el funcionamiento se mantuvo sin cambios).

La ventana principal instancia una vez cada tipo de controlador distinto, creando todos los controladores y repositorios necesarios y los guardandolos. Luego, al abrirse una pestaña, se le pasa por parámetro los controladores que necesite para funcionar. De esta manera, una pestaña dependerá únicamente de los controladores que use y no de un controlador general de fachada con todas las operaciones del sistema, como solía hacerlo cuando se usaba AppLogic. Usando múltiples controladores se logra ocultar las operaciones que la pestaña no requiere, brindándole solo las necesarias.

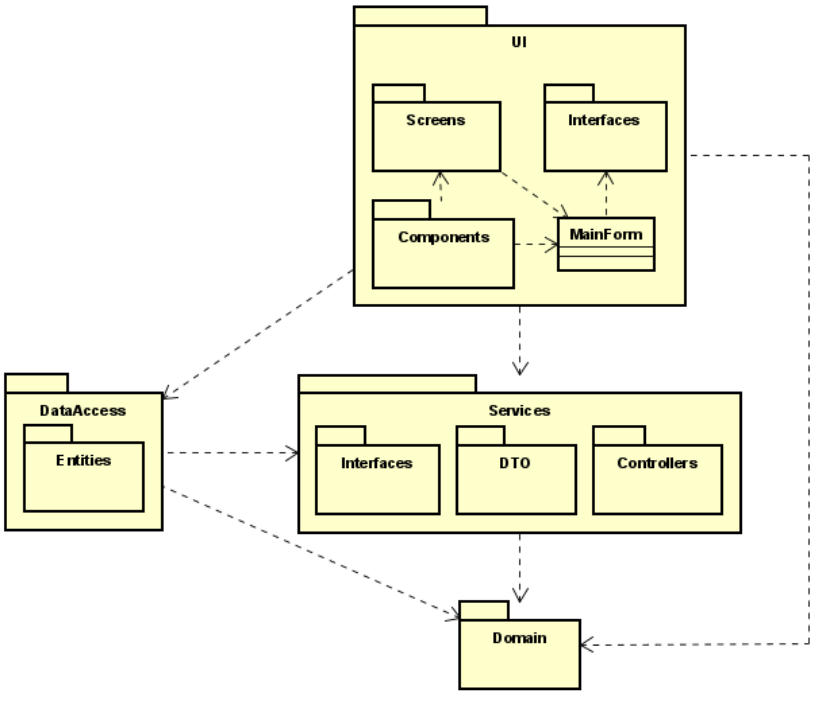
Entonces, si un usuario se encontrará en la pantalla de Sign Up y quisiera registrarse en el sistema, la pantalla de la UI crearía un UserDTO con los datos que proporcionó el usuario y luego se lo daría al UserManager para que este se ocupe de crear el User y lo guarde en la base de datos.

Todo esto puede resultar confuso. Es complejo ver que la UI llame a un método de un controlador para que este utilice operaciones de un repositorio para comunicarse con la base de datos. Debido a esto, es conveniente ver el siguiente

diagrama de interacción, que detalla todo el proceso de creación de un usuario:



Una vez explicado esto, podemos entender que la estructura de paquetes en la que se divide el trabajo es la siguiente:



Al ver este diagrama queda claro que la UI tiene dependencia con paquetes que no debería. Para ejemplificar, en ciertos casos, la UI crea objetos de Domain, generando dependencias no convenientes en el diseño. Estas creaciones deberían realizarse en Services, y que UI solamente se relacione con este paquete, pero la complejidad y la falta de tiempo hicieron que no sea posible hacer un diseño que comunique las capas de manera correcta, sin excepciones.

Debido a esto, la aplicación está rompiendo la ley de Deméter, ya que la UI habla con extraños, no permitiendo ocultar la estructura interna del programa. En concreto, se utiliza Domain ya que AvailableModelElement, componente usado para mostrar los modelos disponibles para colocar en una escena, usa directamente el

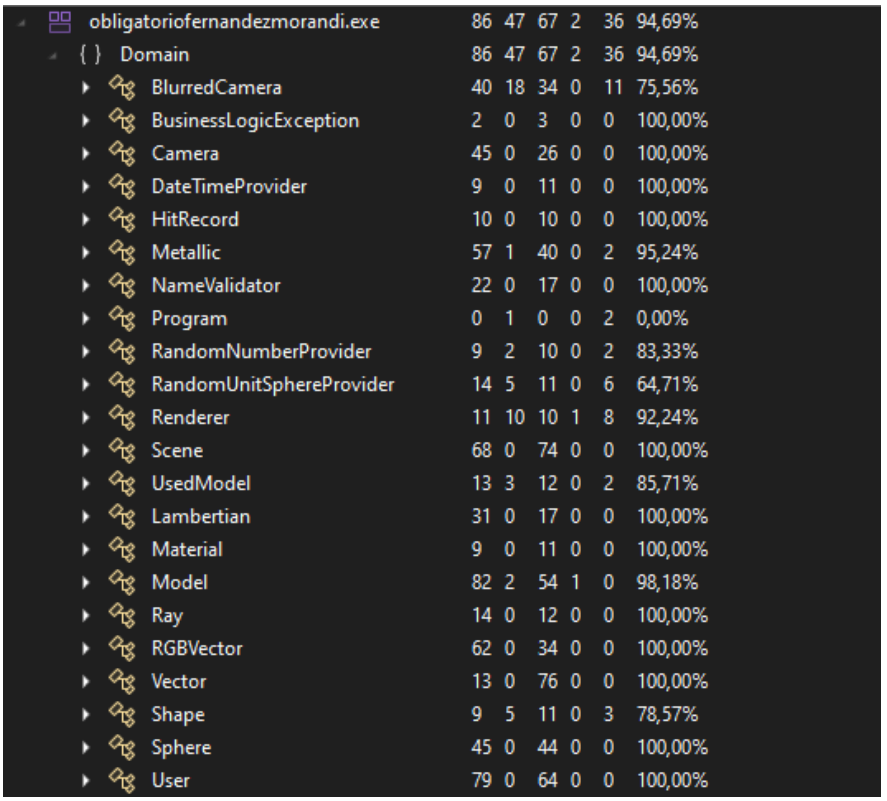
Model. Esta dependencia es muy puntual y podría eliminarse con un poco más de tiempo y planificación. Expandir ModelManager para que realice las acciones que AvailableModelElement está haciendo directamente con Model puede ser una posible solución.

Adicionalmente, UI se relaciona directamente con DataAccess para crear los repositorios. Esto no es recomendable, ya que tanto la UI como la tecnología detrás de DataAccess (EF) podría cambiar en el futuro, impactando en el otro.

De cualquier manera, dejando de lado esos dos casos, las capas del sistema se comportan de manera correcta, siguiendo los principios de SOLID. En párrafos anteriores se habló de la importancia de ISP en esto, pero esta estructura cumple también el principio DIP (), ya que las clases de bajo nivel, que son propensas a cambio (UI, DataAccess) dependen de las de alto nivel (Services y, en nivel aún más alto, Domain) y no viceversa. Esto es claramente visible en el diagrama anterior.

Pruebas

Al igual que en la entrega anterior, y tal como pide la letra, el enfoque de desarrollo fue TDD. Antes de implementar cada una de las nuevas funcionalidades, se escribieron las pruebas pertinentes. Posteriormente, se codificó el requerimiento y se verifica que las pruebas escritas antes pasan. Finalmente, se refactoriza para que quede prolijo y entendible, siguiendo las normas de clean code.



obligatoriofernandezmorandi.exe	86	47	67	2	36	94,69%
{ } Domain	86	47	67	2	36	94,69%
BlurredCamera	40	18	34	0	11	75,56%
BusinessLogicException	2	0	3	0	0	100,00%
Camera	45	0	26	0	0	100,00%
DateTimeProvider	9	0	11	0	0	100,00%
HitRecord	10	0	10	0	0	100,00%
Metallic	57	1	40	0	2	95,24%
NameValidator	22	0	17	0	0	100,00%
Program	0	1	0	0	2	0,00%
RandomNumberProvider	9	2	10	0	2	83,33%
RandomUnitSphereProvider	14	5	11	0	6	64,71%
Renderer	11	10	10	1	8	92,24%
Scene	68	0	74	0	0	100,00%
UsedModel	13	3	12	0	2	85,71%
Lambertian	31	0	17	0	0	100,00%
Material	9	0	11	0	0	100,00%
Model	82	2	54	1	0	98,18%
Ray	14	0	12	0	0	100,00%
RGBVector	62	0	34	0	0	100,00%
Vector	13	0	76	0	0	100,00%
Shape	9	5	11	0	3	78,57%
Sphere	45	0	44	0	0	100,00%
User	79	0	64	0	0	100,00%

Se obtuvo casi un 95% de cobertura de prueba del dominio. Esto significa que el 95% del código escrito fue probado. Este es un muy buen número, ya que se espera que sea mayor a 90%.

Ciertas clases no llegaron al 90% deseado y esto puede ser por diversas razones. Pueden ser clases auxiliares de las cuales no se hacen tests directos, ya que su testeo se hace en base a pruebas en otras clases (tal es el caso de RandomUnitSphereProvider, que se abarca con los tests del Renderer) y por lo tanto es complejo contemplar todos los casos. También puede ser por ser abstracta y tener que probarla a través de un intermediario (como es el caso de Shape).

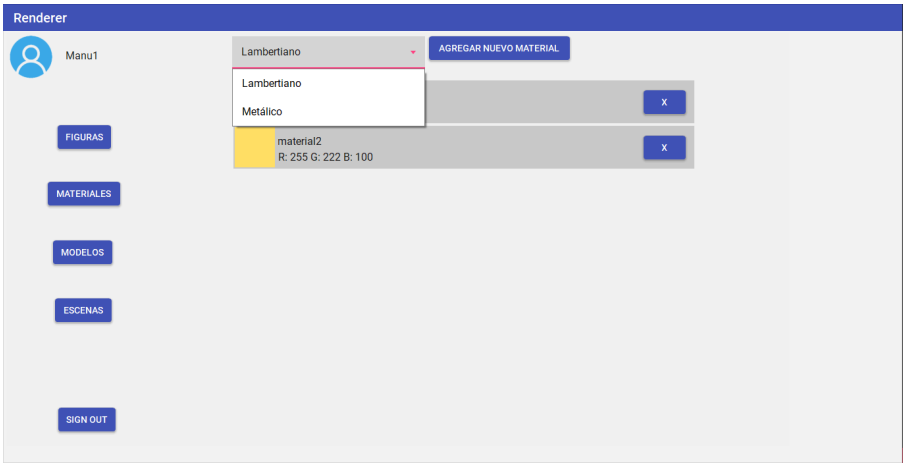
Vale la pena destacar que se llevaron a cabo pruebas únicamente de dominio, ya que las pruebas de UI son más bien empíricas y para testear el resto de paquetes sería necesario utilizar bases de datos de pruebas, cosa que escapa de la consigna.

Cualquiera sea el caso, se logró cubrir una cantidad satisfactoria de código y se siguió la metodología solicitada.

Funcionamiento

De manera general, el funcionamiento de la aplicación sigue siendo igual a como era en la versión anterior, salvo por las funcionalidades nuevas.

Al ingresar a la pantalla de creación de materiales, ahora se puede observar un combobox junto al botón de crear, que brinda la posibilidad de elegir el tipo de material a crear (lambertiano o metálico). Al presionar el botón de crear material, se abrirá una nueva pestaña en la cual deberán introducirse los datos pertinentes para la creación.

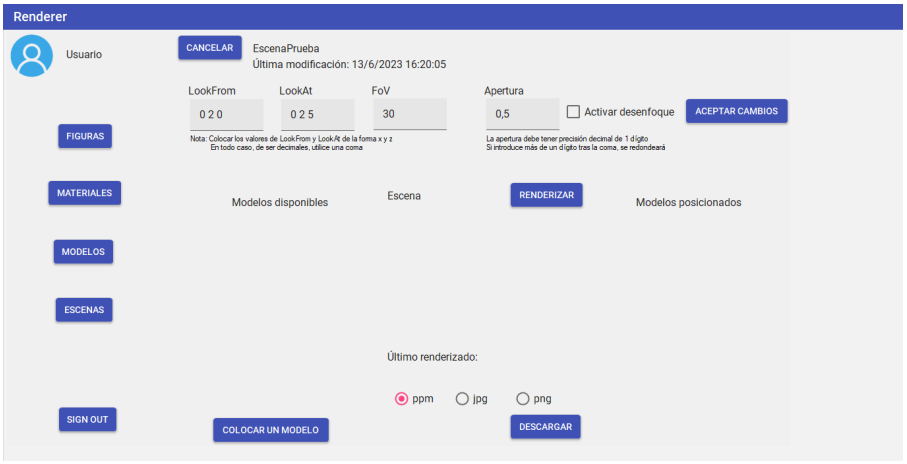


Los otros dos nuevos requerimientos se ven en la edición de escena, ya que en esta pestaña veremos dos nuevas adiciones, una para cada requerimiento.

Arriba a la derecha veremos una sección donde podremos indicar la apertura (valor numérico con precisión decimal de 1 dígito, si se recibe más de un dígito tras la coma el número se redondea) y marcar si queremos que la cámara esté

desenfocada o no. Cabe destacar que la apertura es un valor usado únicamente si el checkbox de cámara con desenfoque está activado.

Adicionalmente, abajo de la imagen renderizada encontramos la opción para elegir un formato y descargar la imagen en dicho formato. Al tocar el botón Descargar, se abre una ventana que permite al usuario seleccionar el directorio deseado.



Instalación

Para instalar la aplicación, se debe tener en cuenta la siguiente información.

El ejecutable se encuentra dentro de la carpeta descargada, bajo el nombre de UI.exe en la carpeta Release, ubicada en Entrega2.

Para hacer uso de la base de datos, deberá actualizar el connection string. Este se ubicará en el archivo App.config ubicado en Obligatorio/UI. Para actualizarlo, en la línea 11, donde lee connectionString, escriba el nombre de su sesión de SQL Server tras la contrabarra (\), modificando únicamente la sesión por defecto (SQLEXPRESS).

La base de datos tiene backups y scripts para asignarlos automáticamente. Se tienen 2 archivos de base de datos, uno vacío y otro con datos de prueba preestablecidos para probar el sistema, junto con sus respectivos scripts. Todo esto se puede encontrar en la carpeta Datos dentro del directorio raíz.

Reflexiones

Alfredo Fernández

Luego de terminar el obligatorio quedé satisfecho con la entrega final y pienso que la experiencia es muy valiosa, se disfrutó del trabajo a pesar de los dolores de cabeza cuando algo no funcionaba como queríamos y resolvimos los problemas que nos encontramos tanto en este como los errores del obligatorio anterior.

Aunque ahora esté más tranquilo porque el trabajo está siendo entregado, debo decir que el tiempo límite es bastante ajustado y no hay suficiente tiempo para corregir todos y cada uno de los detalles y entregar algo perfecto, por lo que llega un momento en el que se debe decir listo y entregarlo como está.

Comparado con la parte anterior esta segunda parte del obligatorio es algo que no podríamos haber hecho en otra materia como p2 o algoritmos ya que incorporamos la persistencia de datos.

Dado que este obligatorio es mucho más extenso y exigente que cualquier otro que hayamos hecho se podría decir que esta experiencia de un “verdadero” desarrollo de software nos sirve para entender cómo se nos va a pedir que trabajemos en un futuro y la exigencia que debemos de tener con nuestro código para entregar algo que respete los estándares.

Manuel Morandi

Los objetivos planteados fueron cumplidos, consiguiendo realizar un software funcional, consistente y, en gran medida, prolijo. La mayoría de los problemas y posibilidades de mejoras que quedaron sin resolverse de la anterior entrega fueron tratadas y solucionadas. Considero que la calidad del trabajo fue en ascenso y, pese a que el producto final no es perfecto, es correcto y cumple los estándares buscados.

El tiempo no fue mucho y la tarea presentó varias dificultades, pero considero que me sirvió como aprendizaje. Me dejó en claro que es siempre importante planificar lo que se va a hacer, porque un error de diseño puede ser complejo de solucionar con el tiempo. Fue necesario ser organizado y trabajar de manera constante para comprender y manejar las tecnologías aplicadas.

Fue especialmente complicado adaptar el trabajo que ya habíamos hecho a todo lo que íbamos dando en clase semana tras semana. Los distintos patrones de diseño y convenciones que veíamos en clase eran rotos en nuestro trabajo ya que, cuando realizamos la implementación, aún no lo estudiamos. Debido a esto, gran parte del tiempo fue invertido en emprolijar cosas, dejando menos tiempo para la implementación, por lo que este debía ser utilizado con cuidado.

En conclusión, estoy contento con el producto final. Es un software a la altura que cumple todos los requerimientos, sin bugs ni errores mayúsculos.