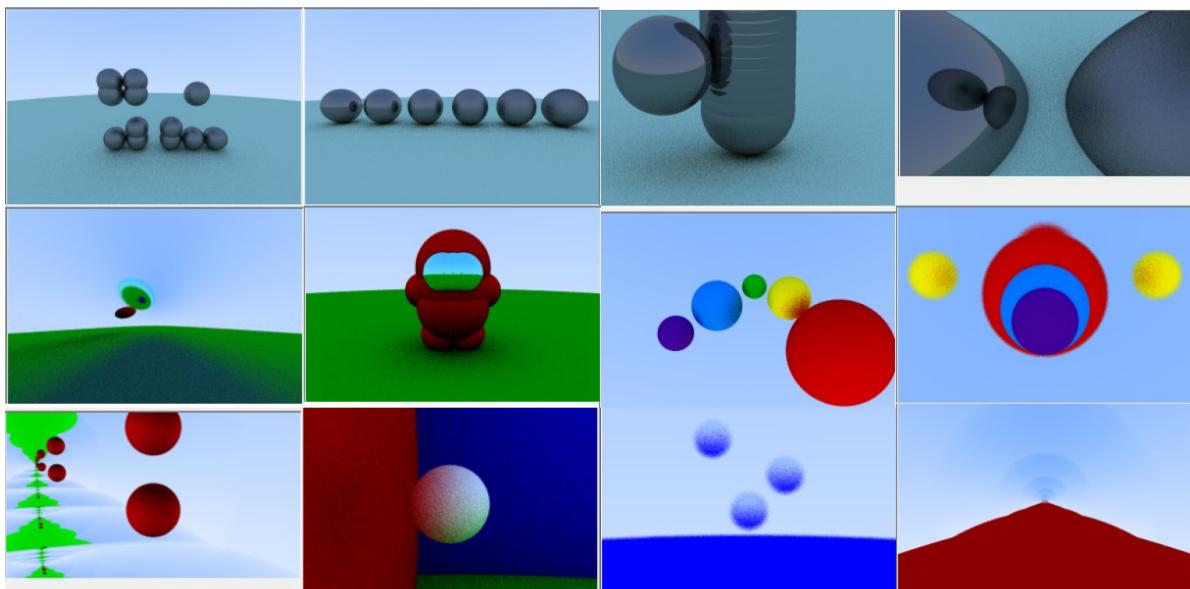


Universidad ORT Uruguay Facultad de Ingeniería

Obligatorio 2 - 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

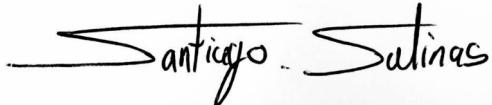
Gastón Mousques

Junio 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 contribuído por otros y qué fue contribuído por nosotros
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes



Santiago Salinas 13/6/2023



Federico Rodriguez 12/6/2023

Introducción al trabajo	4
Descripción general del sistema	4
Fallos conocidos (Se han corregido fallos de la entrega anterior):	4
Guia de instalación	5
Modelo de tablas	7
Mecanismos generales y decisiones	8
Estructura general de la aplicación	8
Controladores por Caso de Uso	10
Manejo de errores	12
Desenfoque:	14
Material Metálico:	16
Soporte de información metálica para materiales	17
Rediseño de Modelo y renderizar su Preview	18
Anexo	22

Introducción al trabajo

En esta entrega trabajamos con Entity Framework 6 para poder persistir los objetos creados por un usuario incluso después de cerrarse la aplicación.

Como también introducir nuevas funcionalidades, que gracias al trabajo de la entrega anterior, partimos desde un punto donde todo el código es fácil de leer y permite facilidad al expandir.

Todo lo anterior se realizó buscando mantener las reglas definidas de código limpio, y teniendo en cuenta los lineamientos de los principios GRASP, SOLID y LISP aprendidos durante el transcurso del curso.

Descripción general del sistema

En esta nueva versión del programa se incluyen tres funciones adicionales: guardar resultados de renderizar como archivos en la máquina del usuario, el material metálico aparte del lambertiano para los modelos, y que la cámara pueda tener desenfoque. Sumándole a su vez la persistencia en una base de datos a través de Entity Framework 6.

Aparte de nuevas funciones, la arquitectura de toda la solución fué completamente reestructurada. Pasando de que solo existan dos paquetes (UI y BusinessLogic) a una distribución de múltiples proyectos que aíslan completamente la interfaz gráfica del dominio y la base de datos. Los repositorios que administran los distintos objetos creados por el usuario pueden cambiar de implementación libremente sin afectar al resto de paquetes, y manteniendo el total funcionamiento del sistema. Hubo también un cambio en la UI: ahora el usuario debe elegir el nombre de la escena cuando la crea y no puede cambiarla más tarde. Pensamos que este cambio ayudaría a la usabilidad del programa.

Fallos conocidos (Se han corregido fallos de la entrega anterior):

- Al crear o editar los objetos del usuario, este tiene la libertad de ingresarles un nombre con el largo que deseé. 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.
- 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*.

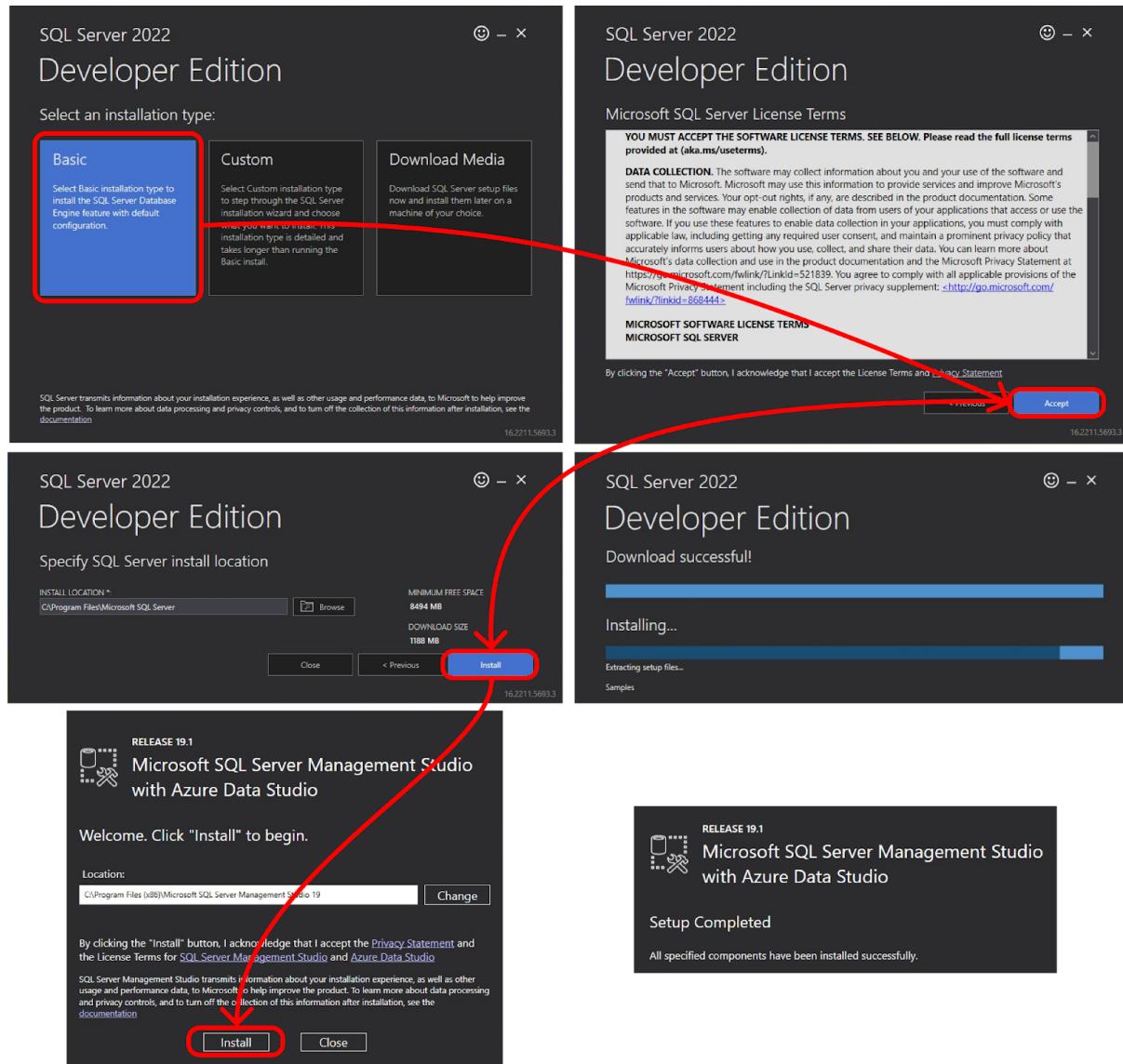
Elegimos estos dos de forma arbitraria, pero ocurriría lo mismo aún si escogiéramos números más grandes, no existe el infinito.

- 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).

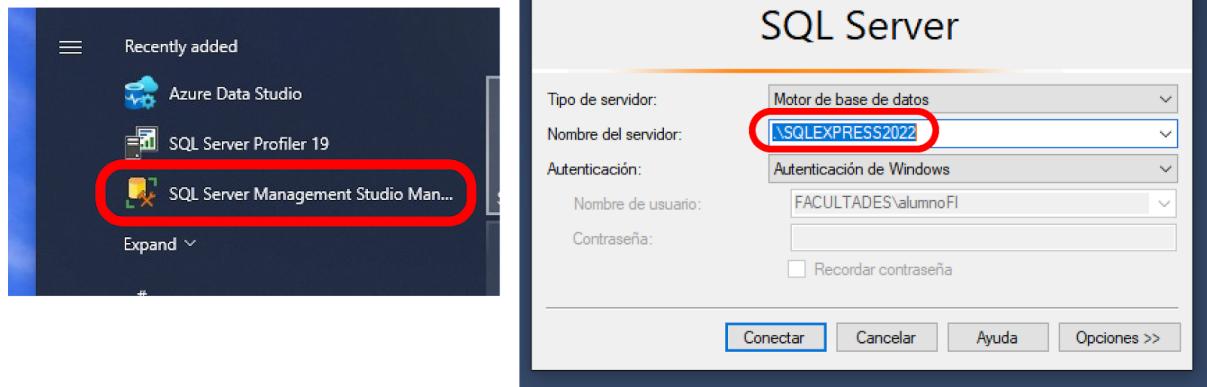
Guia de instalación

Instalar SQL Server Developer

<https://www.microsoft.com/en-us/sql-server/sql-server-downloads>
[ClickToDirectDownload](#)



Correr SQL Server Management Studio
Para conocer el nombre del servidor



Para Usuarios Finales:

Localizar UI.exe.config

Abirlo en un editor de texto

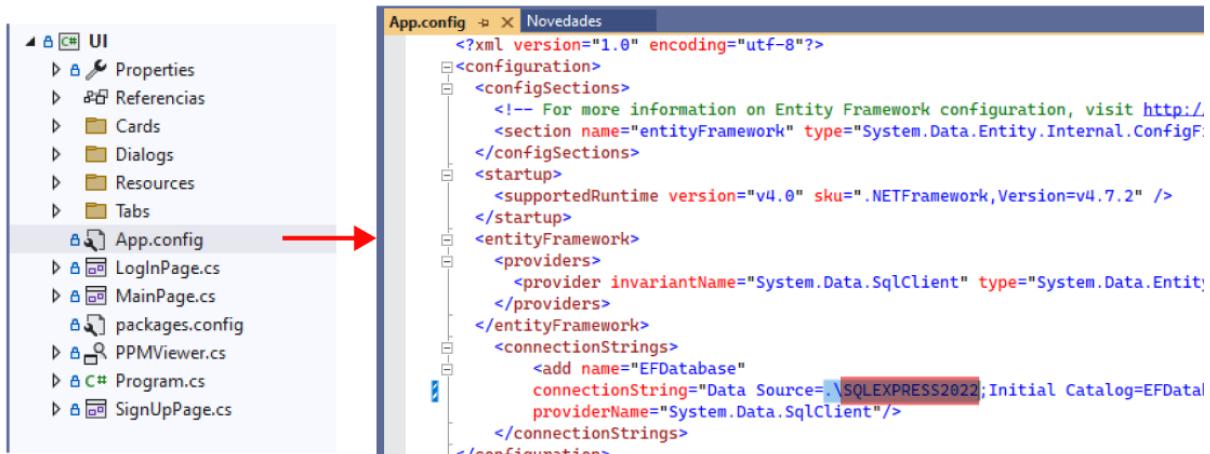
Cambiar el nombre del servidor
al que copiamos anteriormente



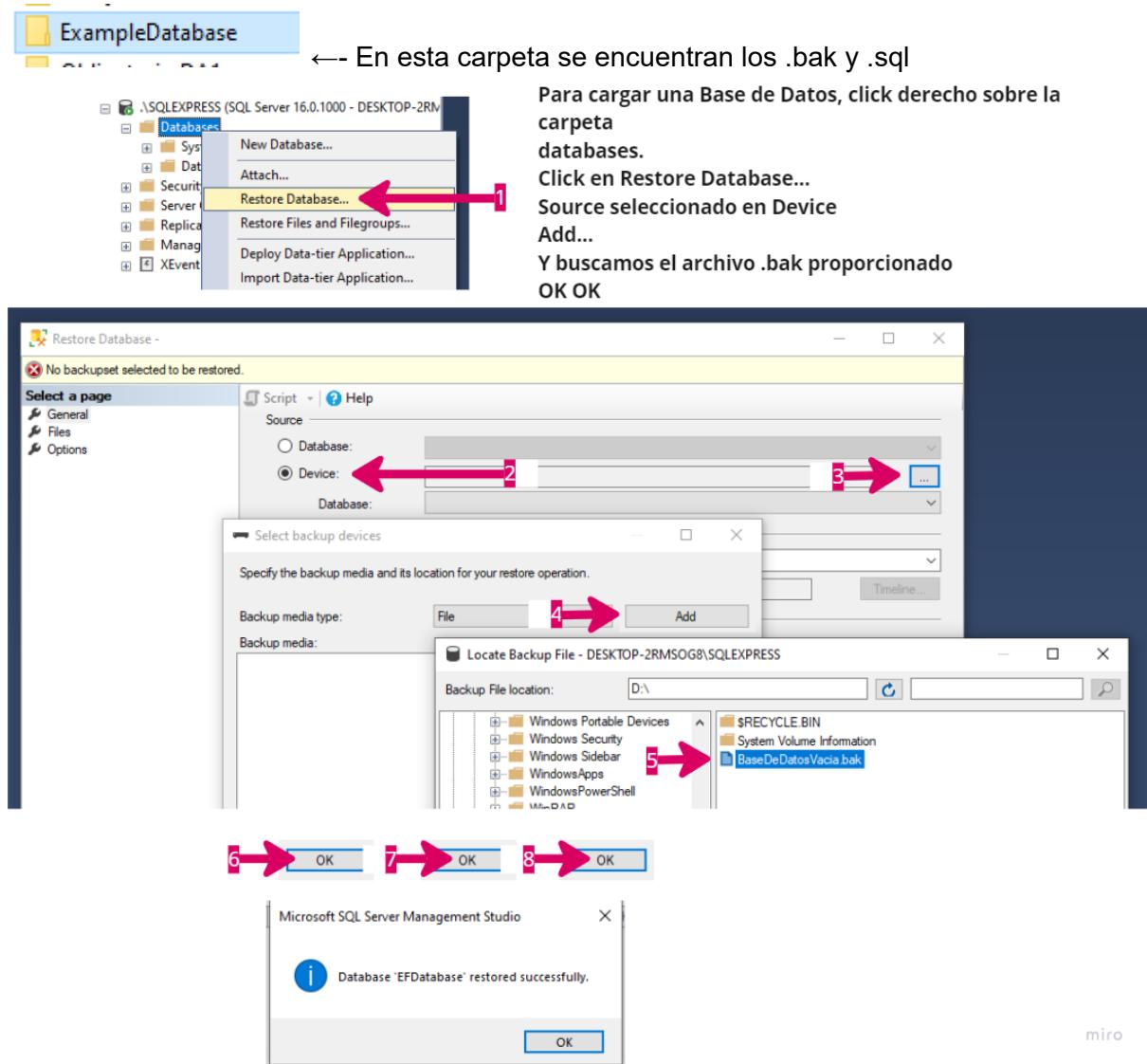
UI.exe.config: Bloc de notas

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigF
  </configSections>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7.2" />
  </startup>
  <entityFramework>
    <providers>
      <provider invariantName="System.Data.SqlClient" type="System.Data.Entit
    </providers>
  </entityFramework>
  <connectionStrings>
    <add name="EFDatabase"
      connectionString="Data Source=.\SQLEXPRESS2022;Initial Catalog=EFData
      providerName="System.Data.SqlClient"/>
  </connectionStrings>
  </configuration>
```

Para Desarrollador (Repetir lo anterior, pero buscando el App.config de UI dentro de VS. Y en App.config de EntityFrameworkTests):



Para todos: Cargar una base de datos .bak



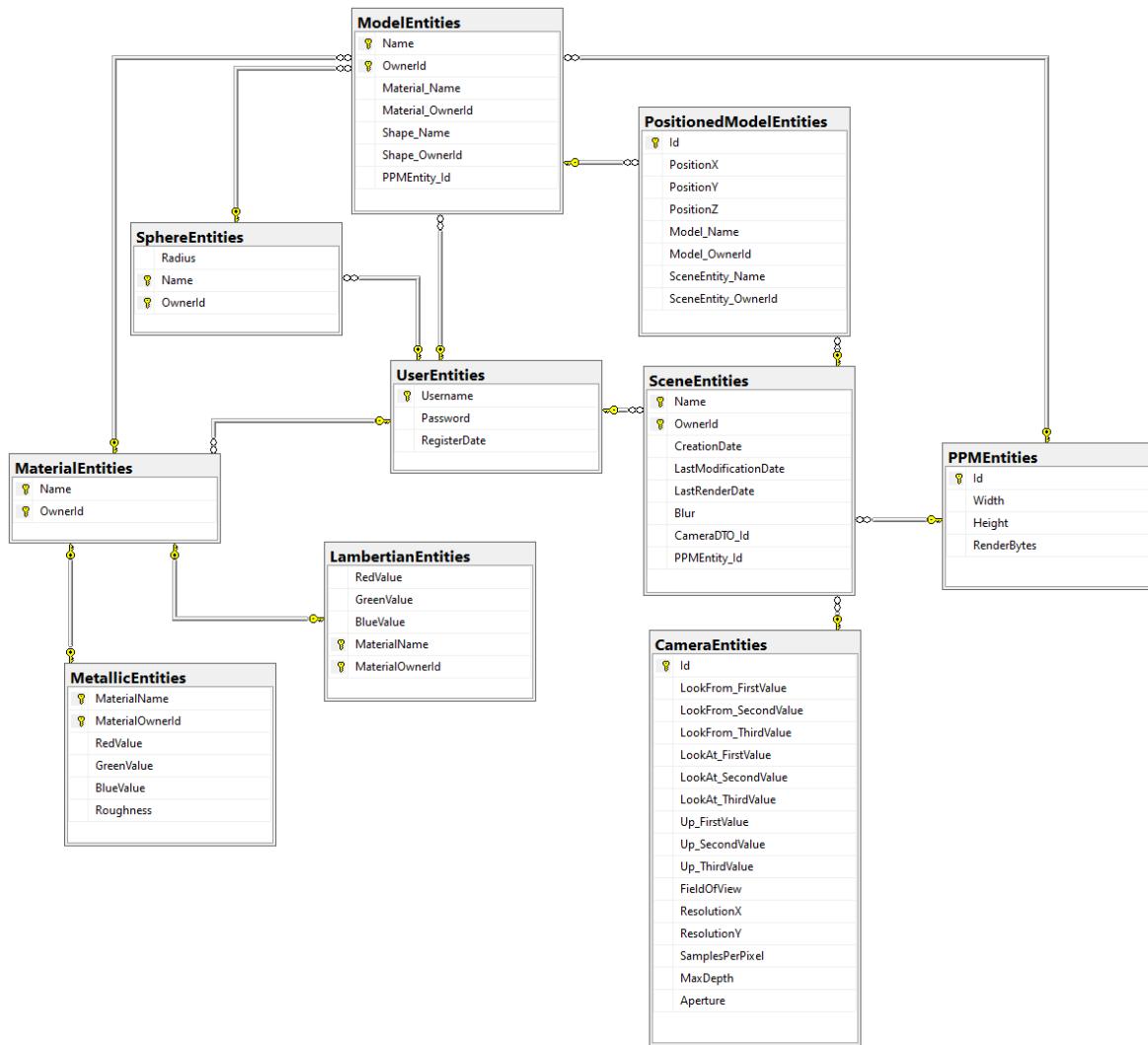
*En caso de no poder encontrar el archivo desde el explorador de SQL Server Management, ubicar carpeta directamente en el disco local C:\ ya que hemos visto que no suele dejar explorar más de 3 subcarpetas en algunas instalaciones.

Modelo de tablas

En general la estructura de las tablas siguen los modelos del dominio, guardando los mismos datos de tipo primitivo como números int,double o strings. Pero en particular vemos que no guardamos los objetos Color para los materiales ni Vector para la posición de los modelos Posicionados.

Ambos, Color y Vector, son clases para que las instancias guardan simplemente tres valores numéricos y puedan realizar operaciones con ellos. En la base de datos, es mejor simplemente guardar esos tres valores como tres campos del objeto al que corresponden, no necesitan su propia tabla.

En el caso de PPMEntities, es una tabla para guardar el último renderizado de cada escena y la preview de los modelos a los cuales se les creó una. En el dominio un PPM se implementa mediante una matriz de objetos Color, cada uno representando un píxel en la pantalla. Pero para la base de datos encontramos que la mejor manera sería convertir esta matriz a un array de bytes, y luego convirtiéndolo en una matriz cuando se extraiga de la base de datos.

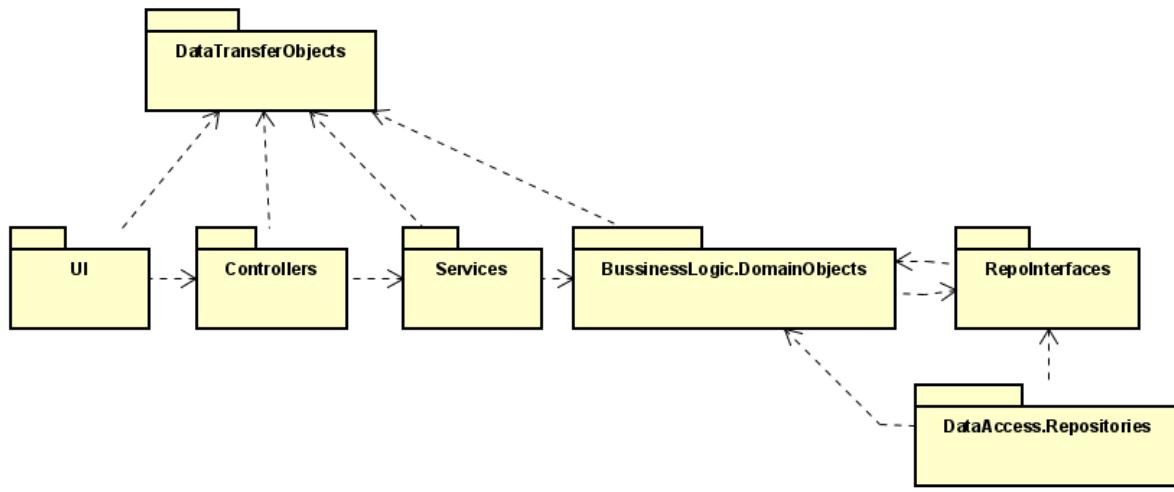


Mecanismos generales y decisiones

Estructura general de la aplicación

Para poder dividir las responsabilidades correctamente, decidimos que nuestra solución debía dividirse en muchos proyectos especializados. Estos proyectos solo conocen a aquellos que necesitan conocer, de manera que por ejemplo la UI no va a poder saltarse capas e interactuar directamente con el dominio. En la siguiente imagen se muestra

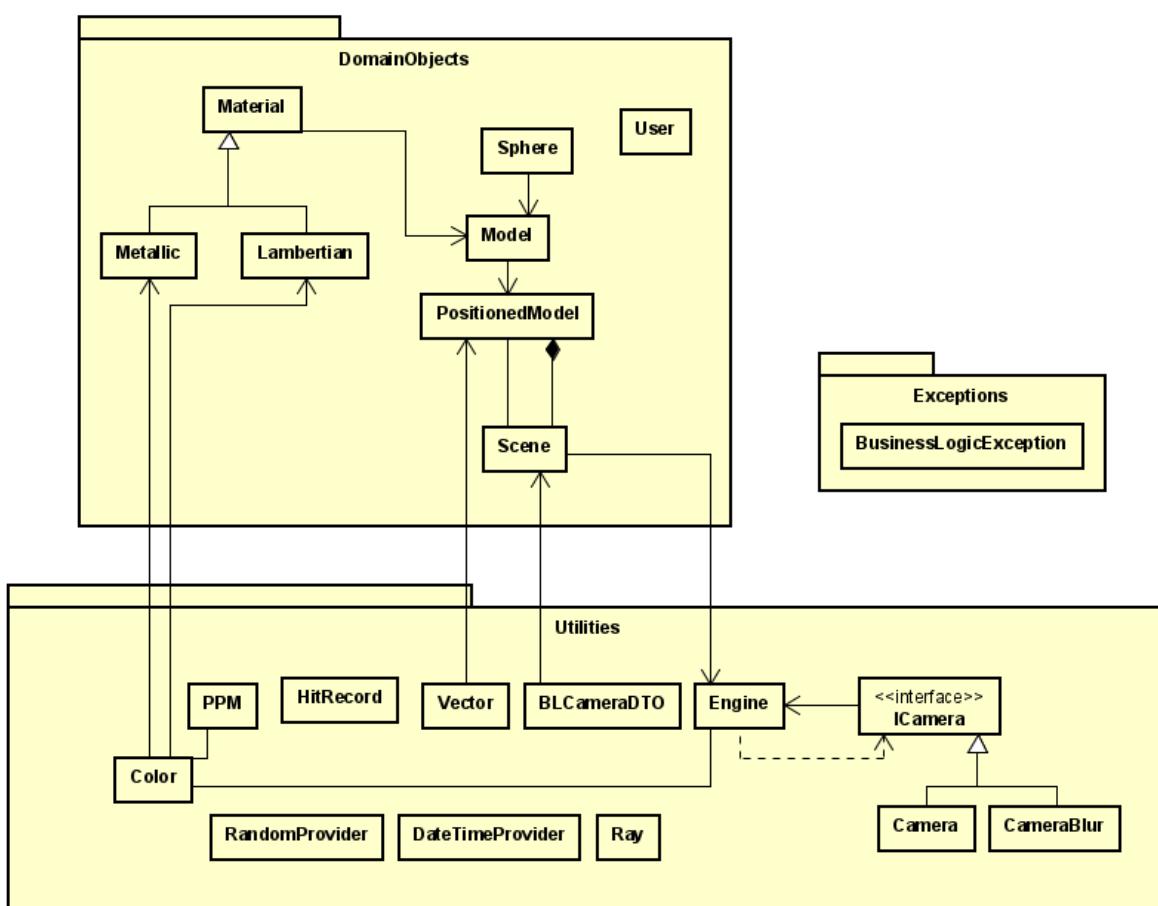
representada la estructura general del diseño, seguida de una explicación punto por punto con el propósito de cada proyecto.



- **UI:** Este proyecto guarda todas las clases que forman las distintas ventanas de la interfaz gráficas hechas con Winforms. También incluye el Program.cs que es lo que se ejecuta primero para iniciar la aplicación. Es en este archivo que se crean e inyectan las instancias de los controladores, servicios y repositorios que se usan en la aplicación. UI, funcionalmente, solo depende de dos proyectos: Controllers y *DataTransferObjects*. En este paquete están definidos los DTO que utiliza para comunicarse con el resto del sistema y en *Controllers* los controladores de uso que simplifican el subsistema de creación y manejo de objetos para la UI que no sabe cómo está implementado. **Ver anexo 1** para la estructura de clases.
- **Controllers:** Este proyecto guarda controladores de uso que manejan cada varios servicios para completar la tarea pedida por el usuario a través de la UI. Controllers solo necesita conocer *DataTransferObjects* para entender la información que le pasa la *UI* y *Services* que es donde están implementados los servicios que necesita usar para completar un caso de uso. Este proyecto tampoco conoce cómo está implementado el dominio o la base de datos. **Ver anexo 2** para la estructura de clases.
- **Services:** Cada servicio en el paquete de Services recibe órdenes desde *Controllers* para realizar operaciones como el renderizado, crear objetos del dominio, convertirlos a DTOs, o manejar la base de datos. Por lo que conoce *DataTransferObjects* que es donde están definidos los DTO y las clases estáticas que convierten entre DTO y objeto de dominio, a *BusinessLogic* que guarda la implementación de los objetos del dominio, y a *RepoInterfaces* que es donde están definidas las interfaces que deben cumplir los repositorios con los que se pretende manejar la base de datos. **Ver anexo 3**.
- **DataAccess:** Este paquete está hecho específicamente para la implementación del backend a través de Entity Framework y podría ser reemplazado por otra implementación manteniendo la integridad de la aplicación. Solo conoce

RepoInterfaces para saber qué especificación seguir para los repositorios y *BusinessLogic* para conocer las clases del Dominio que administran. **Ver anexo 4.**

- **DataTransferObjects:** Es un proyecto en donde se definen los DTO para cada clase del dominio y clases estáticas llamadas *Mappers* que son usadas por services para convertir instancias del dominio en DTOs y viceversa. **Ver anexo 5.**
- **BusinessLogic:** *BusinessLogic* mantiene la implementación del dominio, las reglas que deben seguir las propiedades de cada instancia de un objeto, y muchas clases a las que llamamos Utilities que creamos por conveniencia para el funcionamiento del render y otros objetos del dominio. **Ver anexo 6.**



Controladores por Caso de Uso

Lo que definimos en la primera entrega como “API Gateway”, que prometimos implementar, se define realmente como Controladores por Caso de Uso. Que mediante el uso de DTOs, la UI sin conocer la implementación de BusinessLogic es capaz de interactuar con ella.

Decisiones de diseños

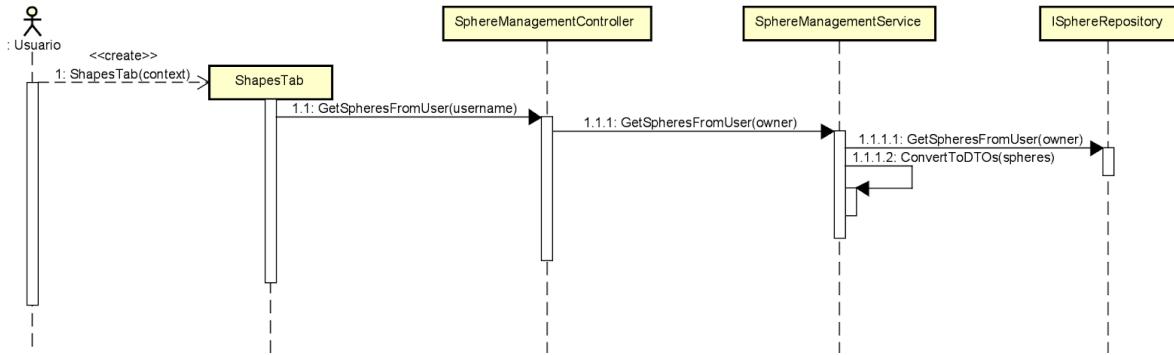
API Gateway Class, punto de entrada

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.



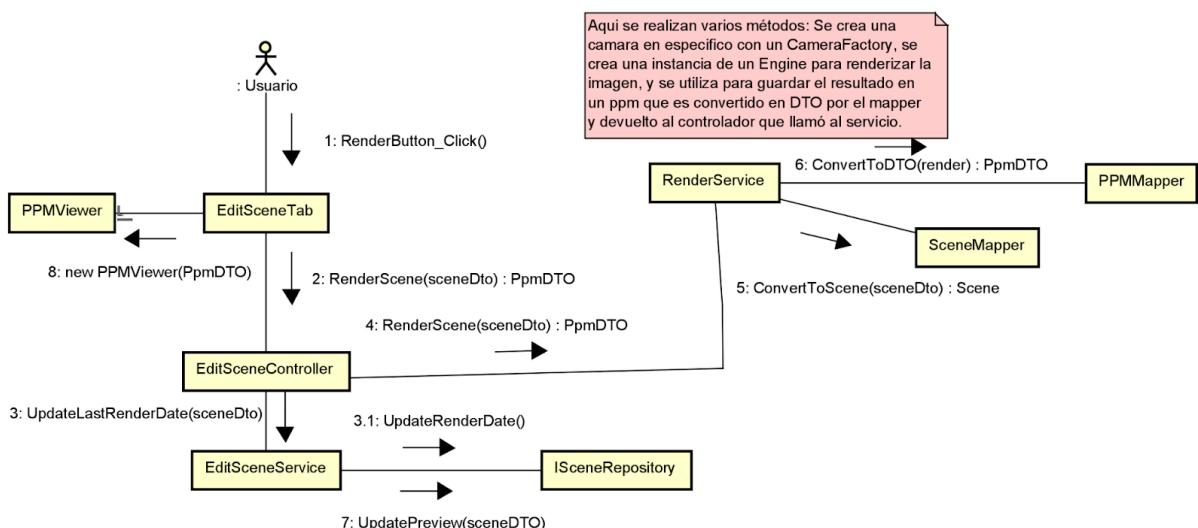
Como se explicó, la UI solo conoce los controladores para hacer uso del resto del sistema, por lo que creamos una clase Context que guarda la instancia de cada controlador y el usuario que inició sesión. Esta clase la podemos inyectar en todas las ventanas para que cada una pueda tener acceso al controlador que necesita y al usuario actual.

Cada tab tiene un controlador asociado con el que pueden hacer las operaciones que les son pertinentes. Mostramos por ejemplo en el siguiente diagrama como se despliegan en la tab de Shapes las esferas del usuario.



Véase cómo la UI lo único que tiene que hacer es pedirle al controlador que traiga las esferas del usuario y este se encarga de pedírselo al servicio para que las extraiga del repositorio y las convierta de Sphere a SphereDTO. Esta estructura de secuencia la siguen todas las tabs al momento de desplegar los materiales, modelos, modelos posicionados y escenas.

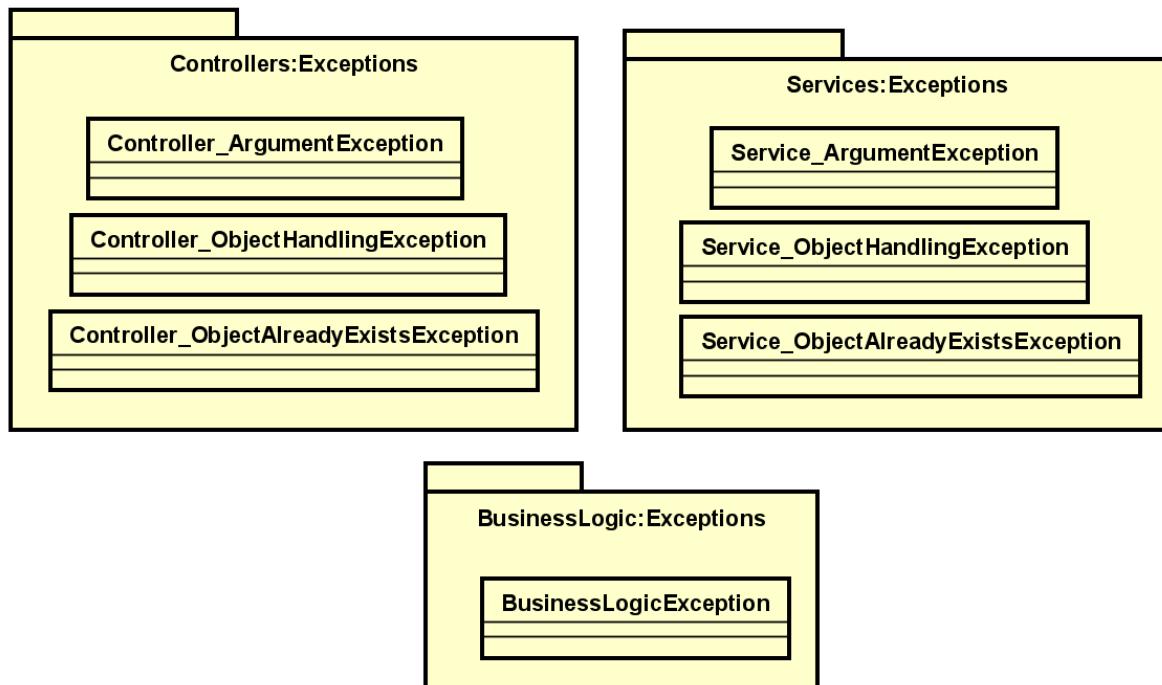
Para poner de ejemplo un caso de uso que involucra un mayor número de clases, a continuación se muestra un diagrama de comunicación representando todo el proceso de llamadas y métodos que se corren al momento de que un usuario hace click en el botón para renderizar una escena. En orden, la UI llama al método para renderizar una escena del controlador asignado, el cual utiliza a su vez dos servicios: el RenderService para crear el PPM de la escena y el EditSceneService que se comunica con el repositorio para guardar la nueva última fecha de renderizado y el nuevo PPM que corresponde a la escena.



Nótese de nuevo como la UI solo tiene que mandar la orden de renderizar la escena para que todo el resto del sistema actúe y el controlador le dé el resultado que espera.

Manejo de errores

Para el manejo de errores nos mantuvimos en la arquitectura de capas que tenemos: UI, Controladores, Servicios, BusinessLogic. Cada capa (exceptuando UI) cuenta con sus propias excepciones definidas.



Así, cuando desde la UI se envía un input que no cumple las condiciones necesarias o es incorrecto en algún otro sentido, la UI no atrapa directamente una excepción originada desde el BusinessLogic. En cambio, la capa de servicios es la que lo atrapa y envía una nueva excepción para ser atrapada por la capa de controladores. Finalmente, el controlador atrapa la excepción originada del servicio y le tira una suya a la UI, para la cual la UI está preparada.

La razón de este sistema es que la interfaz no tiene porque estar preparada para todas las diferentes excepciones que pueden originarse desde los múltiples proyectos involucrados en el sistema. De esta manera nos aseguramos que la UI puede entender un conjunto de excepciones limitado para ella, y desplegar mensajes que tengan sentido para el usuario en lugar de los generados automáticamente por los otros proyectos.

Breve análisis de los criterios seguidos para asignar las responsabilidades

Para asignar las responsabilidades a cada clase, seguimos un número de principios:

1. Principio de responsabilidad única (SRP): Este principio establece que cada clase debe tener una única responsabilidad. Fue el principio que estuvo más presente al crear clases. Desde las del dominio, los servicios, los controladores, y los repositorios y entidades. Siguiendo esto y “fabricación pura” creamos muchas clases auxiliares como Color, Vector, Modelo Posicionado, Engine y todas las otras del paquete Business Logic.Utilities.

También se ve aplicado en la existencia del paquete Data Transfer Objects en donde creamos clases específicas para convertir entre DTO y Objeto de Dominio, en lugar de relegar el mapeado a los servicios o controladores. Esto centraliza cómo se manejan los DTO y como están definidos para todos los paquetes que los usan, de modo que si hay que cambiar la implementación del dominio, el mapeo se cambia en un solo lado.

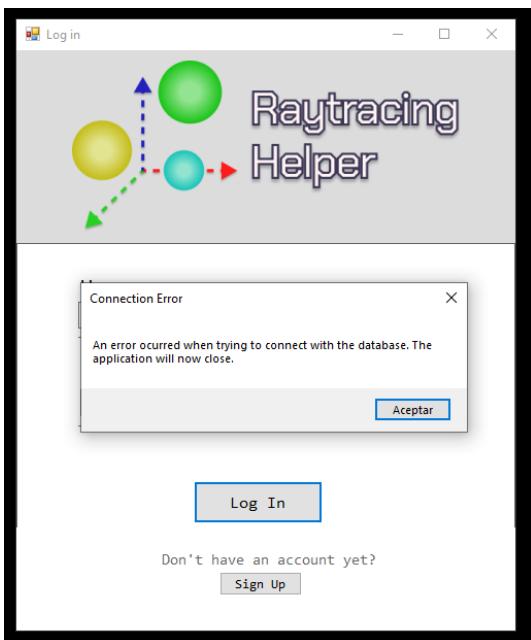
2. Principio de abierto/cerrado (OCP): Según este principio, las clases deben ser abiertas para la extensión pero cerradas para la modificación. Esto se puede lograr mediante el uso de interfaces, herencia y composición. Como ya se explicó, los materiales se manejan de manera tal que está abierto a agregar nuevos materiales como el vidrio que no usan las mismas propiedades que el lambertiano y el metálico.

3. Principio de sustitución de Liskov (LSP): Este principio establece que las clases derivadas deben poder sustituir a las clases base sin afectar la funcionalidad del programa. En nuestro proyecto podemos ver este principio en todos lados que reciba como parámetro un Material, que es la clase de donde heredan Lambertian y Metallic. Esos métodos funcionan independientemente de la subclase de material que reciban.

4. Principio de segregación de interfaces (ISP): Este principio sugiere que los clientes no deben depender de interfaces que no utilizan. Evitar interfaces demasiado grandes o genéricas que obliguen a los clientes a depender de funcionalidades que no necesitan. En nuestro caso, lo podemos ver en como decidimos implementar muchos controladores de uso en lugar de un único controlador de fachada. Las ventanas solo tienen acceso a las funciones que necesitan, y los controladores a servicios y los servicios a repositorios que necesitan.

5. Principio de inversión de dependencia (DIP): Este principio dice que los módulos de alto nivel no deben depender de los módulos de bajo nivel, ambos deben depender de abstracciones. En nuestro caso lo aplicamos al crear interfaces las cuales debían implementar los repositorios para poder ser usados por los servicios. También en los dos tipos de cámaras, que deben implementar una interfaz para poder ser usadas por el Engine. Esto facilita la flexibilidad y el intercambio de componentes en el futuro.

Fallo en la conexión a la DB



La consulta estándar a una base de datos tiene como timeout de 1 minuto.

En caso de demora por causas externas al programa, ya sea por mala instalación, el servicio no está corriendo en el administrador de tareas, y/o el connection string configurado no es el apropiado; este mostrará un mensaje de error.

Si se intenta interactuar con la ventana de forma prematura, puede que windows interprete (erróneamente) que este está bloqueado, pero podemos asegurar que la espera es exacta de 1 minuto.

La excepción se “catchea” de forma exitosa.

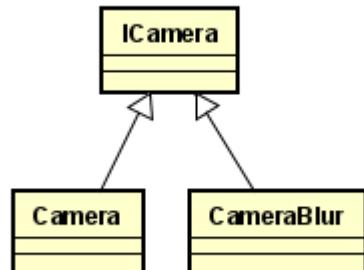
Render

Se introducen tres funcionalidades que afectan el funcionamiento del render.

- Desenfoque
- Material Metálico
- Exportar Imagen

Desenfoque:

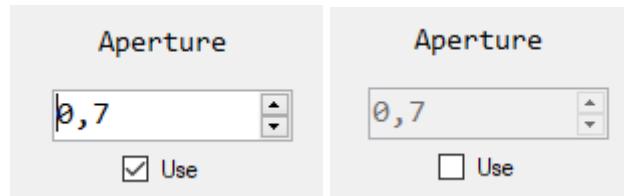
La introducción del desenfoque cambió fundamentalmente el funcionamiento de la cámara, más específicamente en la función Getray, pero no el comportamiento. Es decir, cambia la manera en la que se hacen los cálculos, pero no las firmas de los métodos. En el anterior obligatorio no encontramos casos que ameriten el uso de polimorfismo, pero en esta entrega es aquí donde aplicaremos estos conocimientos. Aplicado mediante el uso de una interfaz



Volviendo al uso de estos, buscamos cumplir con los principios SOLID, específicamente con Liskov y Open Close Patterns, ya que buscamos que el código refleje una estructura tal que como indican los patrones, seamos abiertos a la extensión, para poder agregar más cámaras, y cerrado al cambio, para que sin cambiar más al código relacionado permitir más cámaras.

Liskov nos dice que deberíamos poder cambiar entre las dos cámaras, y el render no se debería enterar de la existencia de dos distintas, este usa únicamente los métodos definidos en la interfaz y con eso es capaz de hacer uso de ambas según la ocasión lo amerite.

En lo que a la usabilidad respecta se agrega a la UI de editar escena, un toggle que habilita o deshabilita el uso de desenfoque. Por heurísticas de nielsen, le damos al usuario un feedback visual y un bloqueo lógico, el cual al estar desactivado el check, el numerical Input se vuelve gris y no se puede editar, simbolizando que no se utiliza.



Hasta ahora hablamos sobre cómo el render, no distingue entre una cámara u otra, pero poco hemos hablado sobre cómo y quién se responsabiliza de la creación de estas. Quien se **responsabiliza** de crear el objeto se puede resolver siguiendo General Responsibility Assignment Software Patterns (GRASP, Patrones Generales de Software para la Asignación de Responsabilidad). Tenemos varios puntos a considerar:

- En principio el information expert de toda información relacionada a los parámetros de inicialización de la cámara es la UI, pero realizar el new del objeto en esta etapa nos genera alto acoplamiento con BusinessLogic
- En respuesta a lo anterior consideramos hacer uso de controllers y servicios (UI - Controller - Service - BusinessLogic)
- Una vez el controller tenga la información, mediante el traspaso por DTOs, este, ahra uso de una fábrica embebida, que sabiendo si la escena usa difuminado (Blur) lo hace

```
public PpmDTO RenderScene(SceneDTO providedScene)
{
    Scene renderScene = SceneMapper.ConvertToScene(providedScene);

    ICamera camera = CameraFactory(renderScene);

    Engine renderEngine = new Engine(renderScene, camera);
    PPM render = renderEngine.Render();

    return PPMMapper.ConvertToDTO(render);
}

public ICamera CameraFactory(Scene scene)
{
    if (scene.Blur)
    {
        return new CameraBlur(scene.CameraDTO);
    }
    else
    {
        return new Camera(scene.CameraDTO);
    }
}
```

Material Metálico:

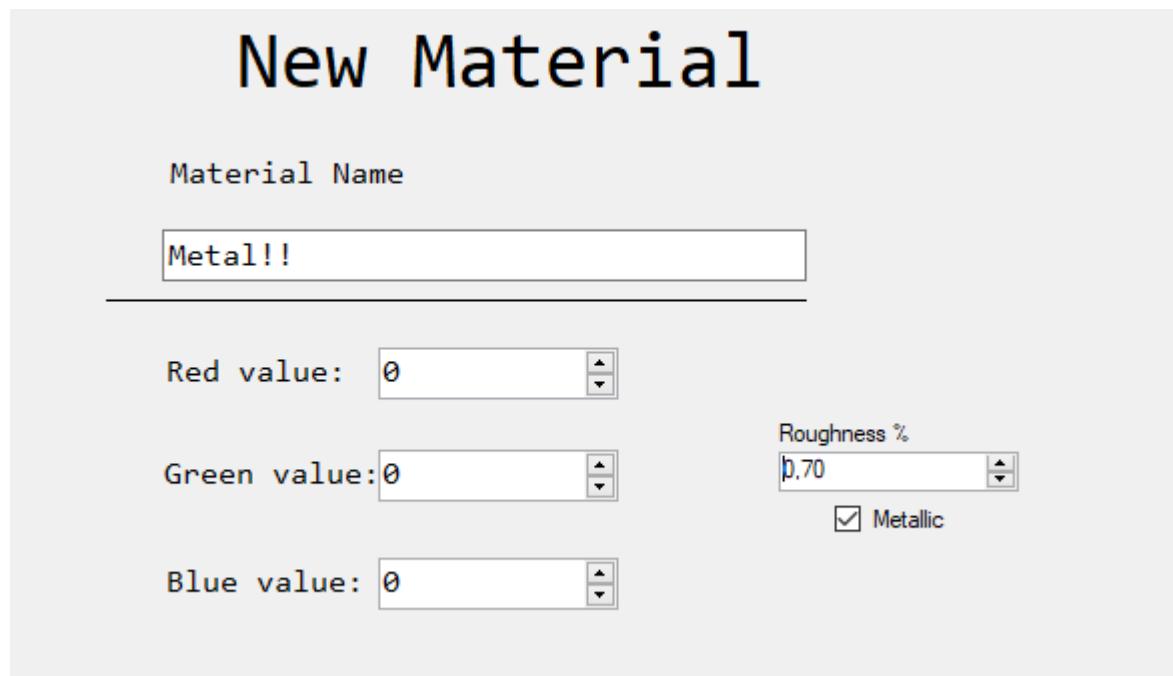
En la primera entrega solo éramos capaces de renderizar materiales opacos llamados lambertianos, pero bajo esta entrega se introduce un nuevo material con una superficie similar a la de un espejo. Aunque esta se puede ajustar bajo roughness para determinar qué tan nítida es la reflexión.

El material por sí solo no tiene color, ya que a futuro, un material podría ser incoloro por su naturaleza. Pero lo que si todo material en general debe conocer, además de conocer su nombre y su creador, a la hora de interactuar con el render cumplen todos el mismo rol, indicarle al rayo como debe rebotar y cómo debe interactuar con el hitRecord (Elemento utilizado para informar sobre el estado de golpe de cada rayo).

```
public abstract Ray GetBouncedRay(HitRecord hitRecord);  
+  
public abstract HitRecord IsHitByRay(HitRecord hit);
```

Son el basamento de todo material para poder ser renderizado por el render. La implementación de cada una de ellas queda a cargo del mismo material a diseñar.

Se puede hacer uso de este nuevo material, aplicando el toggle “metallic” en la fase de creación de un nuevo material. eligiendo un roughness entre 0 y 1.. En caso de no hacer uso de este parámetro por default utiliza el material lambertiano.



Y aquí es donde nuevamente nos encontramos con un problema similar al de las cámaras, donde debemos crear una u la otra. Esto se resuelve nuevamente en la capa de servicio, donde aprovechamos los Material Mappers utilizados para transformar de DTO a Objeto, para injectar una fábrica.

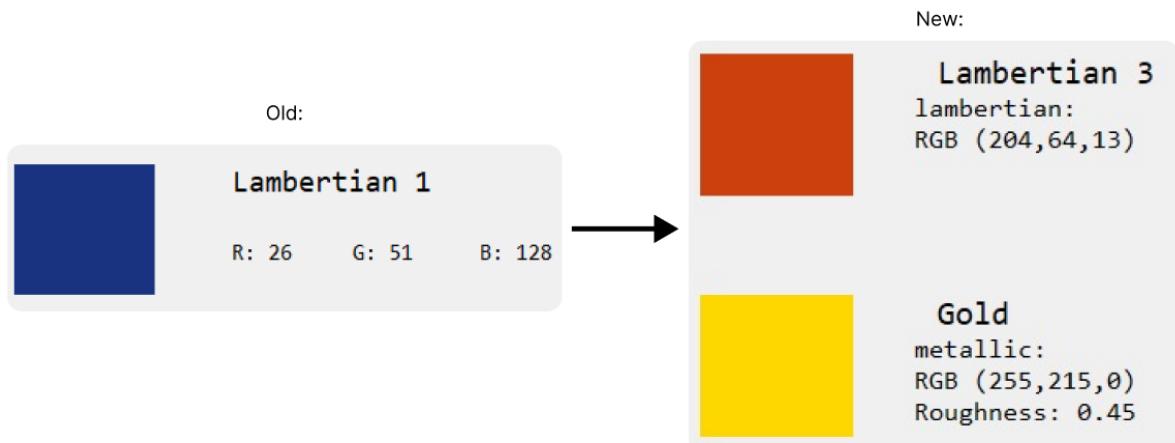
```
3 referencias | Santiago, Hace 9 días | 1 autor, 5 cambios
public static Material ConvertToMaterial(MaterialDTO dto)
{
    Material ret = null;

    if(dto.Type == "lambertian")
    {
        ret = new Lambertian()
        {
            Name = dto.Name,
            Owner = dto.Owner,
            Color = ColorMapper.ConvertToColor(dto.Color),
        };
    }else if(dto.Type == "metallic")
    {
        ret = new Metallic()
        {
            Name = dto.Name,
            Owner = dto.Owner,
            Color = ColorMapper.ConvertToColor(dto.Color),
            Roughness = dto.Roughness,
        };
    }

    return ret;
}
```

Soporte de información metálica para materiales

Previamente, las cards tenían a su disposición un único material, el lambertian, por lo que directamente le pedían los colores y lo mostraban. Pero ahora cada material tiene información distinta a mostrar.



Por lo que ideamos que cada material tenga su propia información bajo `ToString()`, de tal forma que si la card desea mostrar esta información, se le es otorgada a través del DTO que llama esta función.

```
2 referencias | Santiago, Hace 9 días | 1 autor, 6 cambios
public static MaterialDTO ConvertToDTO(Material material)
{
    MaterialDTO materialDTO = new MaterialDTO()
    {
        Name = material.Name,
        Owner = material.Owner,
        Color = ColorMapper.ConvertToDTO(material.Preview),
        Type = material.Type,
        Info = material.ToString(),
    };

    return materialDTO;
}

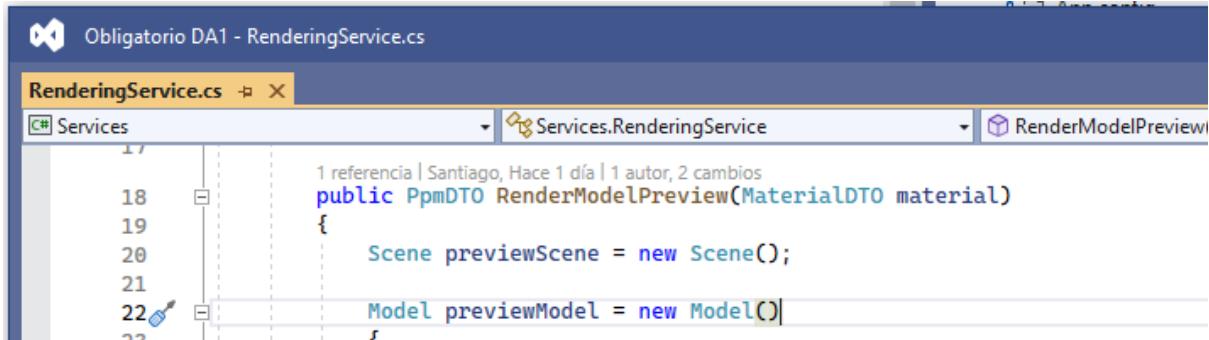
0 referencias | Santiago, Hace 17 horas | 1 autor, 1 cambio
public override string ToString()
{
    return Type + ":\nRGB (" + Color.ToString() + ")\\nRoughness: "+Roughness;
}

0 referencias | - cambios | -autores, - cambios
public override string ToString()
{
    return Type+":\nRGB (" + Color.ToString() + ")";
}
```

Rediseño de Modelo y renderizar su Preview

En la entrega anterior, la clase Model ofrecía una función `RenderPreview()`, que generaba una escena, colocaba el modelo posicionado, generaba un engine y

renderizaba una preview. Claramente todo este proceso no es responsabilidad del Modelo en sí mismo, por lo que estaríamos violando el Principio de Responsabilidad Única (SRP) de SOLID. La responsabilidad relacionada a la preview que tiene el modelo, únicamente es la de property. Por lo que se movió todo este proceso a la capa de servicio.



```

Obligatorio DA1 - RenderingService.cs
RenderingService.cs
Services
Services.RenderingService
RenderModelPreview

1 referencia | Santiago, Hace 1 día | 1 autor, 2 cambios
public PpmDTO RenderModelPreview(MaterialDTO material)
{
    Scene previewScene = new Scene();
    Model previewModel = new Model();
}

```

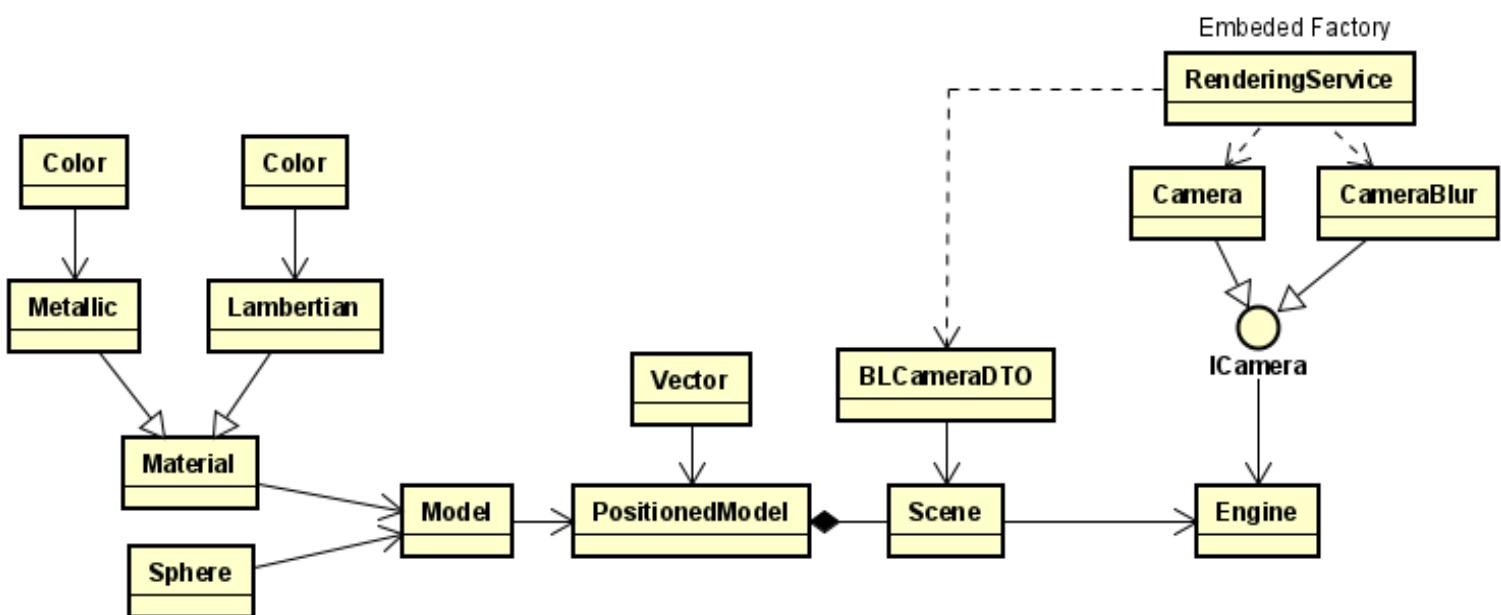
Exportar imagen

Hicimos referencia a la documentación oficial de Microsoft sobre Windows Forms. Modificando la función de exportar como ppm, para que soporte nuestro formato propietario.

<https://learn.microsoft.com/en-us/dotnet/desktop/winforms/controls/how-to-save-files-using-the-savefiledialog-component?view=netframeworkdesktop-4.8>

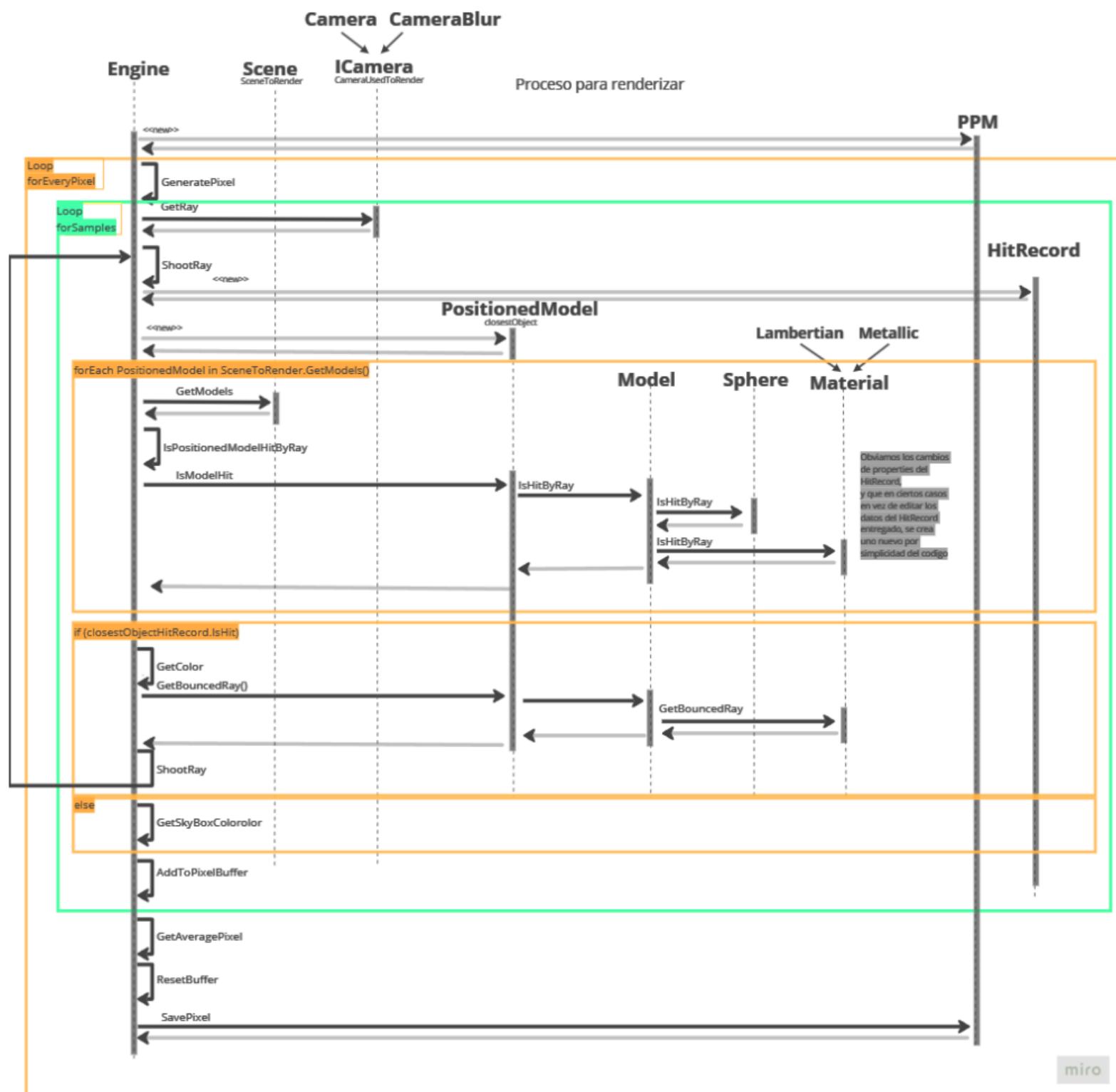
Proceso de un render

En la entrega anterior (Anexo 6), se realizó una explicación del proceso que se recorre hasta generar una imagen (ppm), y he aquí la actualización del diagrama relacionado. En esencia el recorrido es el mismo, únicamente que incluye la abstracción a materiales polimórficos y el uso de la interfaz de cámara.



Y veremos a continuación un diagrama actualizado de comunicación para el método render del Engine, siendo este el tercer y último diagrama de comunicación presentado en esta entrega; y el diagrama con más información de todos. En especial se agrega el uso de estructuras de control, se denota el uso de <>new>> y más aspectos que fuimos aprendiendo sobre diagramas de interacción posterior a la entrega anterior.

Diagrama viejo en Anexo 7



Cobertura de pruebas unitarias

Las pruebas de businessLogic se hacen directamente sobre instancias de objetos de dominio y los repositorios en memoria, sin hacer uso de la base de datos, debido a que se busca probar su funcionalidad de forma excluida al entorno.

DataAccess está bajo en porcentaje debido a la carpeta de Migrations, intesteable.

Repositories.dll (85%) es debido a que marca como si no se testeán las expresiones lambda

```
3 references | Fede PC, 2 days ago | 1 author, 1 change
public bool ExistsModelUsingTheMaterial(string materialName, string owner)
{
    Model ret = _modelList.Find(m => m.Material.Name == materialName && m.Owner == owner);
    return ret != null;
}
```

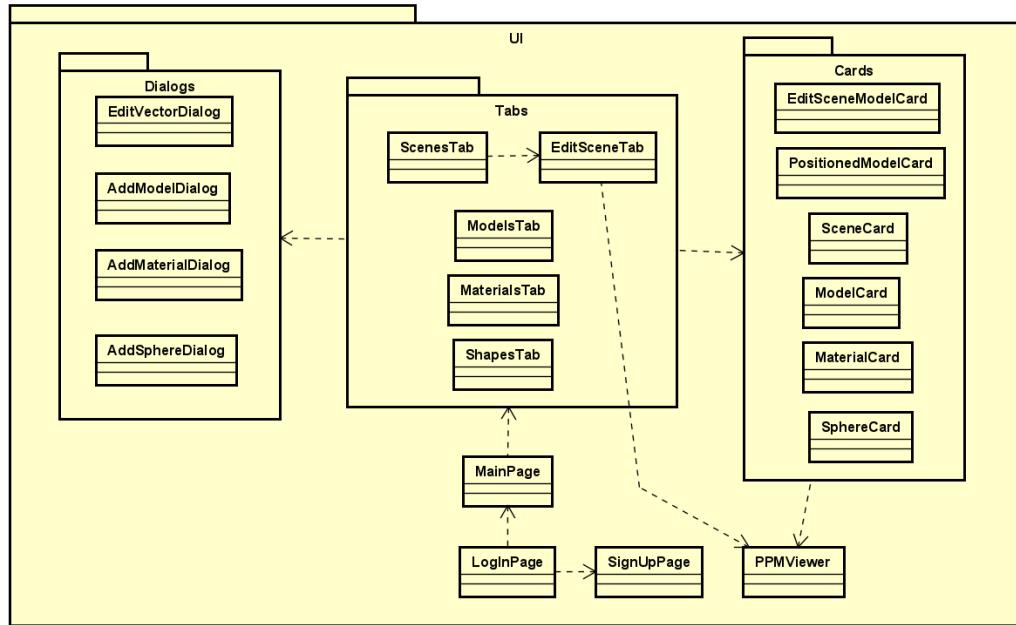
Si se nos escapó algo es relacionado a algún método de property, que de por sí guardan y devuelven correctamente al ser una estructura propia de C#.

Ver Anexo 8 para ver más

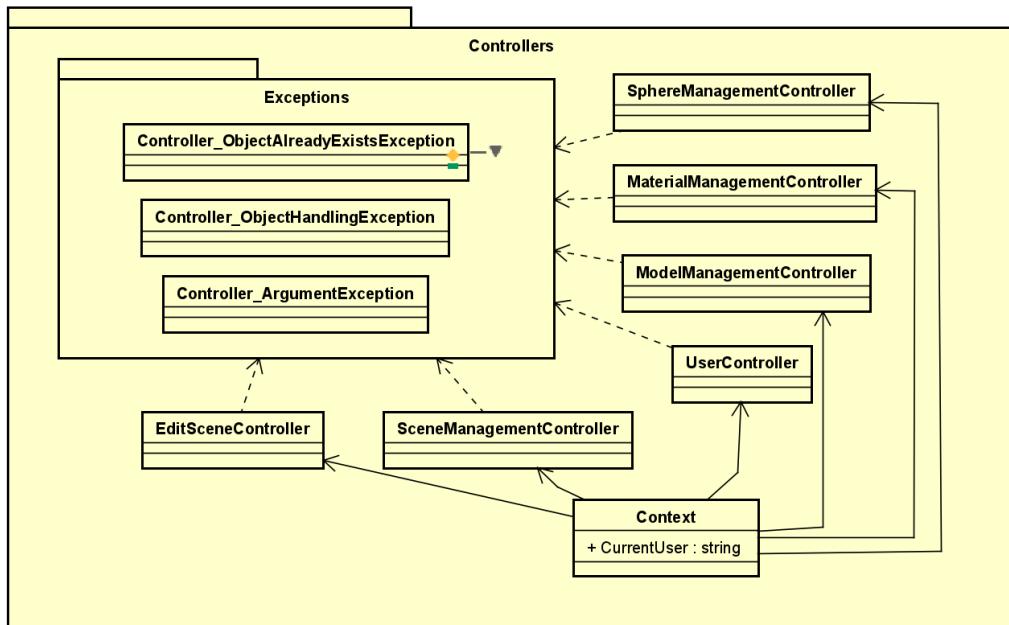
Hierarchy	Covered (%Lines) ▲
└ Santiago_DESKTOP-2RMSOG8 2023-06-14 18_32_08.coverage	90.55%
└ dataaccess.dll	66.87%
└ DataAccess.Migrations	1.05%
└ DataAccess.Entities	93.85%
└ DataAccess.Repositories	95.20%
└ DataAccess	100.00%
└ repositories.dll	82.47%
└ controllers.dll	90.77%
└ Controllers	90.55%
└ Controllers.Exceptions	100.00%
└ datatransferobjects.dll	92.33%
└ DataTransferObjects.DTOs	81.75%
└ DataTransferObjects.Mappers	99.00%
└ services.dll	95.79%
└ Services	95.73%
└ Services.Exceptions	100.00%
└ businesslogic_tests.dll	96.89%
└ BusinessLogic_Tests	96.89%
└ businesslogic.dll	97.29%
└ BusinessLogic.DomainObjects	95.70%
└ BusinessLogic.Utilities	98.38%
└ BusinessLogic.Exceptions	100.00%
└ entityframeworktests.dll	98.77%
└ EntityFrameworkTests	98.77%

Anexo

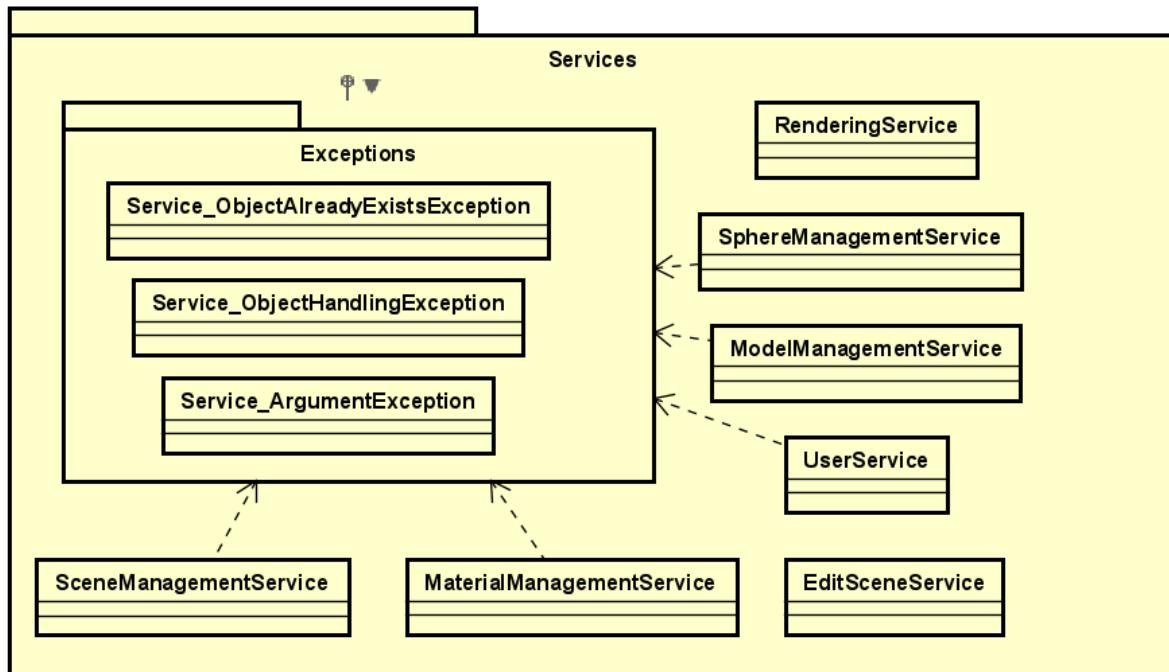
Anexo 1: Estructura del paquete UI. La página del login es el origen de todo el resto de ventanas pues desde ahí se crea la del Sign Up y la MainPage. MainPage crea los 4 tabs principales y los tabs crean los diálogos o cards que les corresponden. Las ventanas o cards que necesitan mostrar el preview de un modelo o el de una escena necesitan crear un PPMViewer para desplegar la imagen resultante.



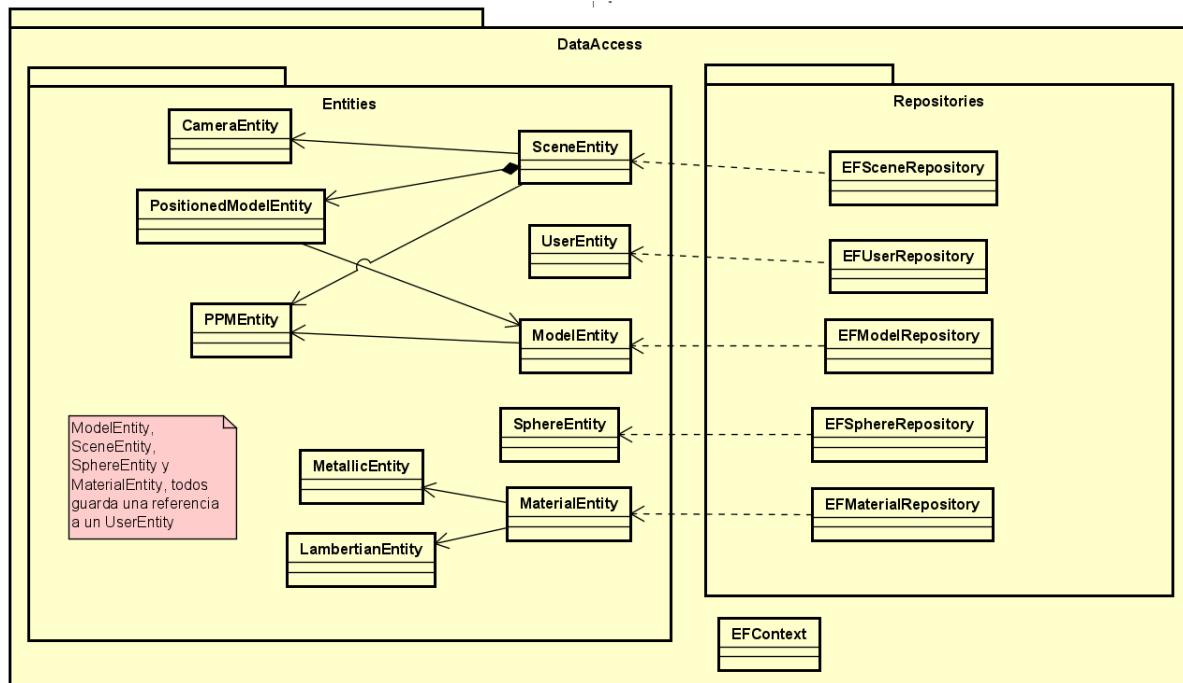
Anexo 2: Estructura del paquete Controllers. Además de los controladores y las excepciones que tiran, encontramos la clase Context que pasamos por todas las ventanas para inyectarle a cada una el controlador que necesita.



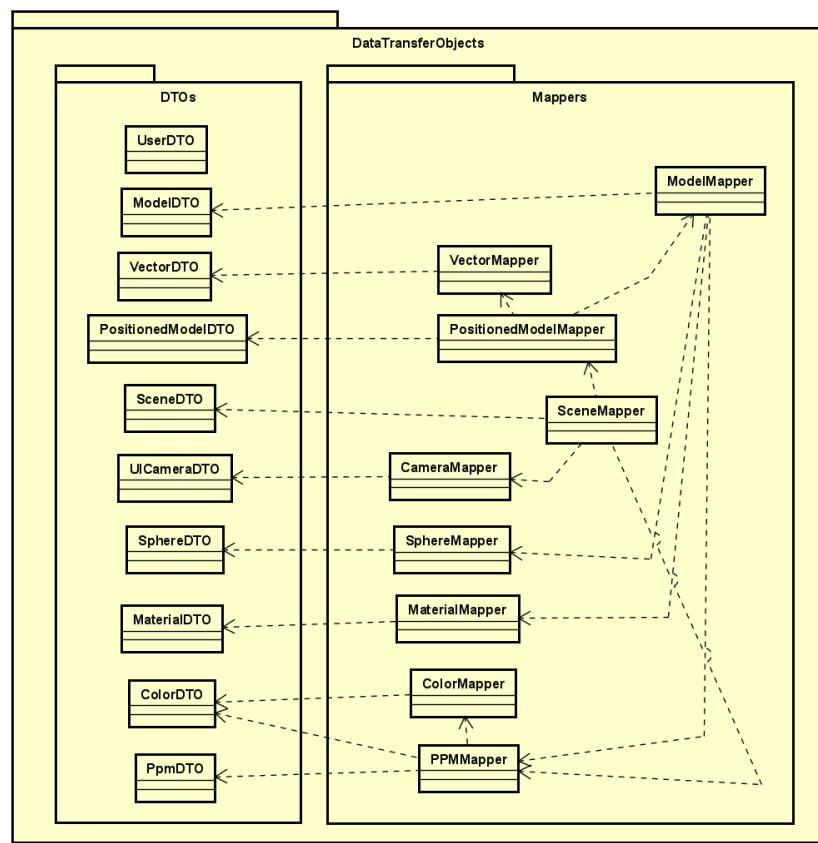
Anexo 3: Estructura del paquete Services. Cada servicio actúa independiente del resto, pero varios de ellos tiran las excepciones del paquete para que sean atrapadas por los controladores.



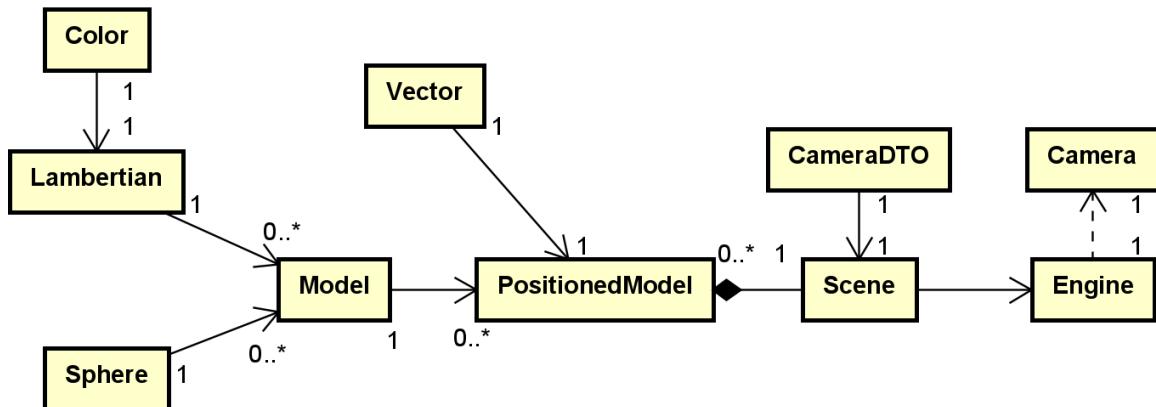
Anexo 4: Estructura del paquete DataAccess. Vemos tanto los repositorios con los que administramos las entidades como las entidades que creamos para guardar toda la info necesaria en la base de datos.



Anexo 5: Estructura del paquete DataTransferObjects. Un conjunto de mappers y los DTOs que utilizan, algunos mappers también hacen uso de otros mappers para completar sus propias operaciones.

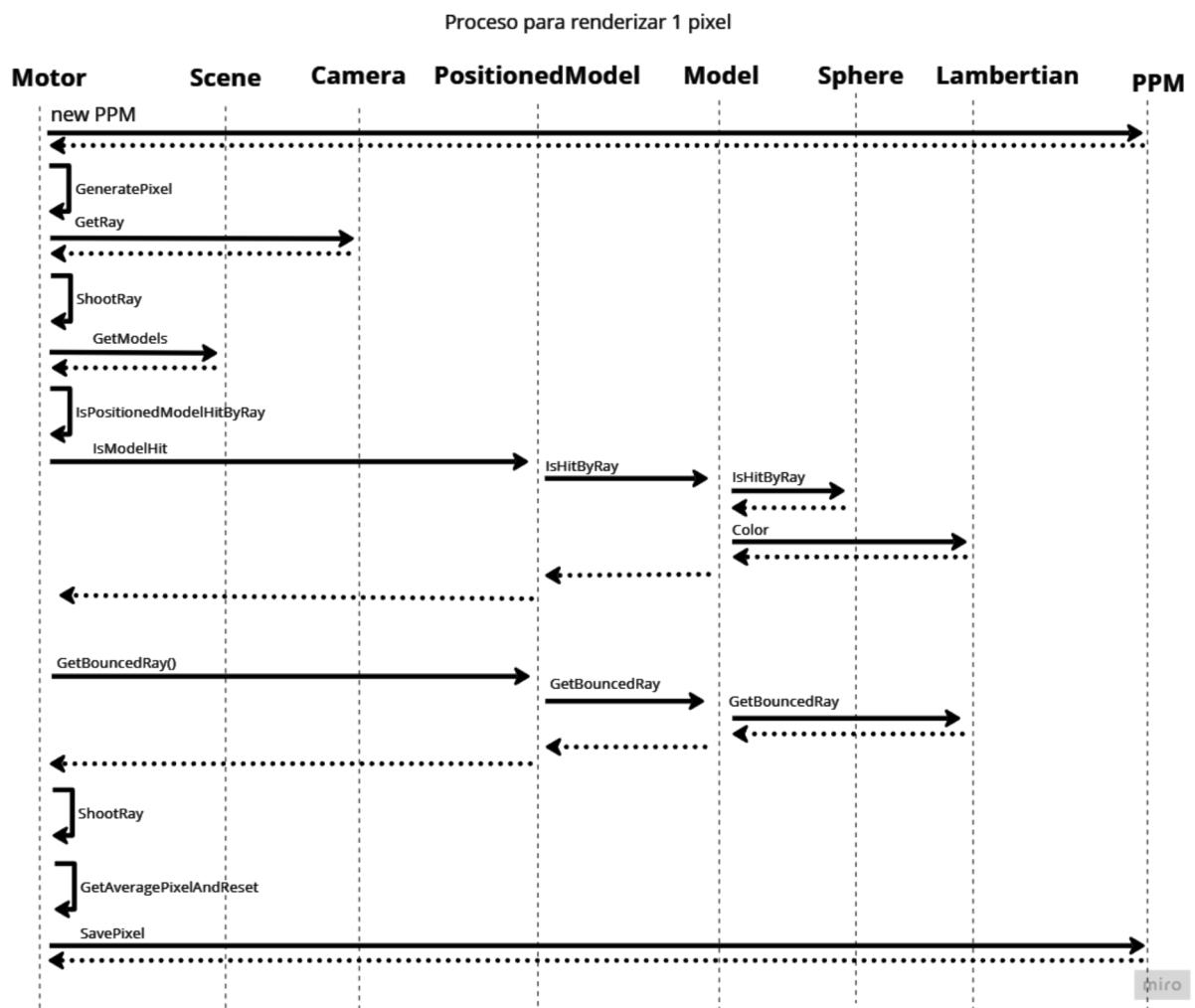


Anexo 6: Estructura vieja del proceso de un render



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

Anexo 7: Estructura vieja del proceso de un render



Anexo 8: Cobertura completa de tests

Hierarchy	Covered (%Lines)
Santiago_DESKTOP-2RMSOG8 2023-06-14 18_32_08.coverage	90.55%
dataaccess.dll	66.87%
{ } DataAccess.Migrations	1.05%
↳ configuration	0.00%
↳ configuration.<>c	0.00%
↳ Configuration	75.00%
{ } DataAccess.Entities	93.85%
↳ LambertianEntity	69.05%
↳ PositionedModelEntity	75.86%
↳ MaterialEntity	94.44%
↳ ModelEntity	95.65%
↳ SceneEntity	98.65%
↳ CameraEntity	100.00%
↳ MetallicEntity	100.00%
↳ PPMEntity	100.00%
↳ UserEntity	100.00%
↳ SphereEntity	100.00%
{ } DataAccess.Repositories	95.20%
↳ EFSphereRepository	91.67%
↳ EFMaterialRepository	97.50%
↳ EFModelRepository	98.00%
↳ EFSphereRepository	100.00%
↳ EFUserRepository	100.00%
↳ EFSphereRepository.<>c	100.00%
↳ EFSphereRepository.<>c_DisplayClass6_0	100.00%
↳ EFMaterialRepository.<>c	100.00%
↳ EFModelRepository.<>c	100.00%
↳ EFSphereRepository.<>c	100.00%
{ } DataAccess	100.00%
↳ EFContext	100.00%
repositories.dll	82.47%
controllers.dll	90.77%
{ } Controllers	90.55%
↳ Context	60.00%
↳ ModelManagementController	90.91%
↳ EditSceneController	91.89%
↳ MaterialManagementController	92.31%
↳ SceneManagementController	100.00%
↳ UserController	100.00%
↳ SphereManagementController	100.00%
{ } Controllers.Exceptions	100.00%
↳ Controller_ObjectAlreadyExistsException	100.00%
↳ Controller_ArgumentException	100.00%
↳ Controller_ObjectHandlingException	100.00%

datatransferobjects.dll		92.33%
{ } DataTransferObjects.DTOs		81.75%
>UserDTO		0.00%
>PpmDTO		58.33%
>SphereDTO		58.33%
>MaterialDTO		92.31%
>UICameraDTO		100.00%
>ColorDTO		100.00%
>ModelDTO		100.00%
>PositionedModelDTO		100.00%
>SceneDTO		100.00%
>VectorDTO		100.00%
{ } DataTransferObjects.Mappers		99.00%
>ColorMapper		90.91%
>CameraMapper		100.00%
>MaterialMapper		100.00%
>ModelMapper		100.00%
>PositionedModelMapper		100.00%
>PPMMapper		100.00%
>SceneMapper		100.00%
>SphereMapper		100.00%
>VectorMapper		100.00%
services.dll		95.79%
{ } Services		95.73%
>EditSceneService		92.65%
>MaterialManagementService		95.00%
>SceneManagementService		95.00%
>ModelManagementService		96.00%
>RenderingService		97.30%
>SphereManagementService		100.00%
>UserService		100.00%
{ } Services.Exceptions		100.00%
>Service_ArgumentException		100.00%
>Service_ObjectAlreadyExistsException		100.00%
>Service_ObjectHandlingException		100.00%

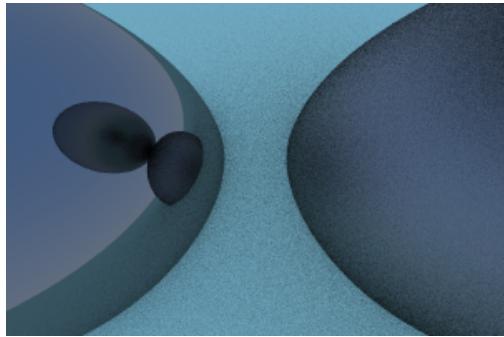
businesslogic.dll	97.29%
{ } BusinessLogic.DomainObjects	95.70%
➤ Scene.<>c_DisplayClass42_0	0.00%
➤ Lambertian	87.50%
➤ Scene	91.36%
➤ Metallic	96.43%
➤ Material	100.00%
➤ Model	100.00%
➤ PositionedModel	100.00%
➤ User	100.00%
➤ Sphere	100.00%
➤ Scene.<>c_DisplayClass47_0	100.00%
{ } BusinessLogic.Utilities	98.38%
➤ PPM	92.59%
➤ Color	94.12%
➤ Engine	97.70%
➤ Camera	100.00%
➤ CameraBlur	100.00%
➤ BLCameraDTO	100.00%
➤ RandomProvider	100.00%
➤ HitRecord	100.00%
➤ DateTimeProvider	100.00%
➤ Ray	100.00%
➤ Vector	100.00%
{ } BusinessLogic.Exceptions	100.00%
➤ BusinessLogicException	100.00%

Uso moderado de recursos

<https://aulas.ort.edu.uy/mod/url/view.php?id=439176>

Para resolución de bugs de compilación, entendimiento de ciertas prácticas y conversión de test en memoria a test que impacten en BD dando un ejemplo. Conversiones de sentencias SQL a LINQ

Curiosidad: Preparación de una escena



●	a : Sphere((-2.5, 4, 0), 2) = $(x + 2.5)^2 + (y - 4)^2 + z^2 = 4$
●	b : Sphere((2.5, 0, 0), 2) = $(x - 2.5)^2 + y^2 + z^2 = 4$
●	c : Sphere((-2.5, 0, 0), 2) = $(x + 2.5)^2 + y^2 + z^2 = 4$
●	up = Vector((0, 0, 0), (0, 1, 0)) = $\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$
+	Input...

