Universidad ORT Uruguay
Facultad de Ingeniería

Diseño de Aplicaciones 1
Obligatorio 1

Entregado como requisito para el Obligatorio 1 de Diseño de Aplicaciones 1

Diego Acuña - 222675
Felipe Briosso - 269851
Nicole Uhalde - 270303

Tutores:
Gastón Mousqués
Facundo Arancet
Franco Galeano

2023

Repositorio: https://github.com/ORT-DA1-2023/222675-269851-270303

Índice

Bugs y defectos 3
Bug 01 3
Defecto 01 3
Defecto 02 3
Defecto 03 3
Arquitectura de la aplicación 4
Diagrama general de paquetes (namespaces) 5
Diagramas de clase por paquete y principales decisiones de diseño: 5
BackEnd 5
Figures 6
Utilidades 7
GraphicMotor Utilities 8
Materials 9
Logs 10
IODrivers 10
Controladores 11
Dtos 12
Interface repositories 13
Servicios 14
Entities 15
Context 16
Diagramas de secuencia 16
Cobertura de tests 18
Instalación 19

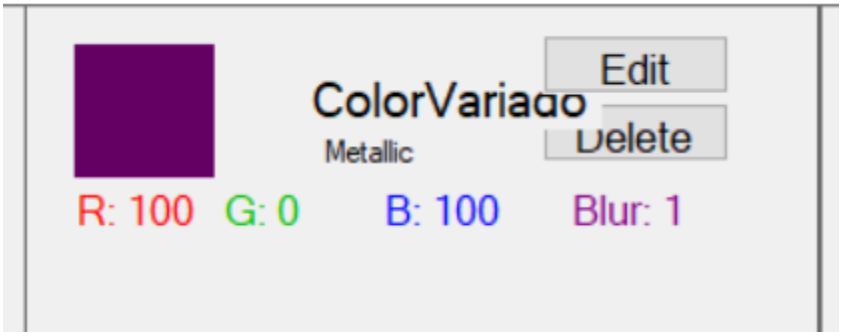
Bugs y defectos

Bug 01

Mantenemos el bug de la entrega anterior que en caso de que matemáticamente no sea posible realizar el render este se rompe. Dado que no afecta el funcionamiento del sistema entendemos que queda a cargo del usuario el no hacer operaciones matemáticamente incorrectas.

Defecto 01

Vemos que en caso de que el nombre de las entidades sea muy largo este se superpone con los botones. A continuación, se muestra un ejemplo para material:



Defecto 02

Si bien se mejoró la UX respecto a la entrega anterior, reconocemos que no es del todo intuitivo.

El cambiar nombre de la escena se podría mejorar, aprovechando la ventana creada para el cambio de nombre que usan las demás entidades.

Defecto 03

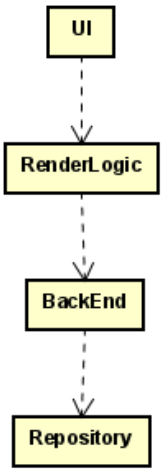
Otro defecto es que algunas labels y otros componentes tienen el nombre por defecto (ej: label24), esto claramente contradice las reglas de nomenclatura de *Clean Code*.

Por lo que para futuras ediciones sería recomendable corregirlo.

Arquitectura de la aplicación

Se decidió para esta versión continuar con una arquitectura basada en capas bien definidas y con propósitos específicos.

Se utilizaron las siguientes capas de modo tal de que cada capa dependa únicamente de la capa inferior a ella. Esta decisión está basada en que de esta forma se mejora la mantenibilidad el código y se facilita el cambio.

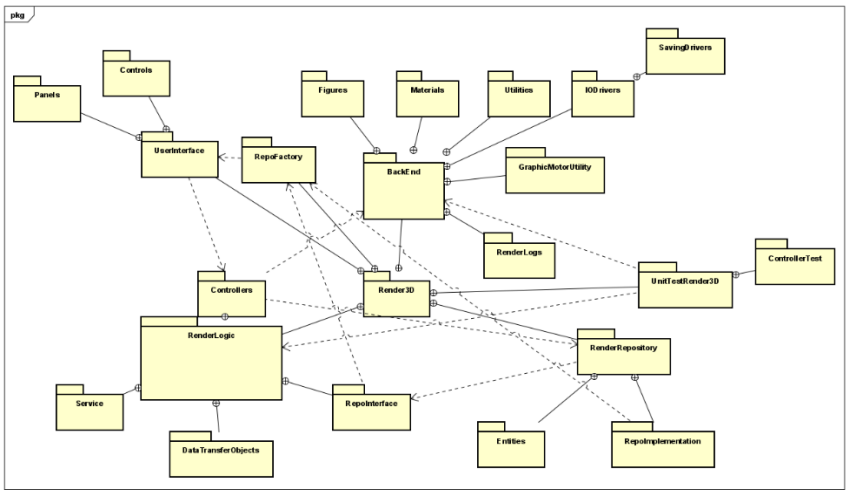


A su vez, decidimos fusionar la “Capa de servicios” con la lógica por motivos de simplicidad, aunque fueron situados en diferentes namespaces, de modo tal de que, si en el futuro se desean agregar nuevos servicios, sea sencillo extraerla a una capa nueva.

La aplicación consta de seis proyectos, cuatro corresponden a las capas del diagrama anterior, uno a los tests unitarios y el otro al RepositoryFactory que será desarrollado más adelante.

Diagrama general de paquetes (namespaces)

A continuación, se indica el diagrama de paquetes:

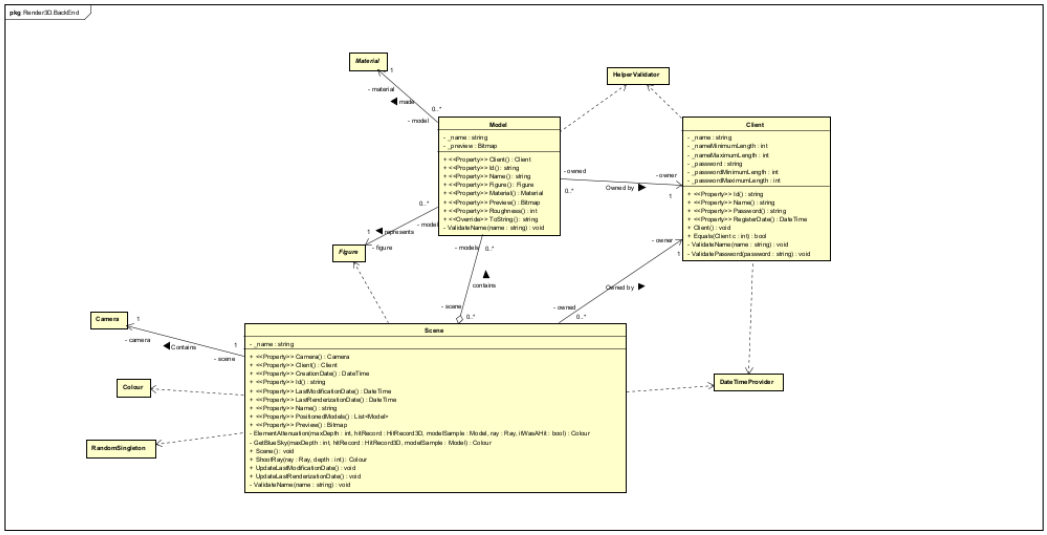


La justificación de porqué se dividió de esta forma será explicado más adelante.

Diagramas de clase por paquete y principales decisiones de diseño:

Con motivo de mejorar la claridad de los diagramas, se optó por mostrar completamente las clases del namespace, y mostrar únicamente el nombre de las clases colaboradoras que no pertenecen al namespace.

BackEnd



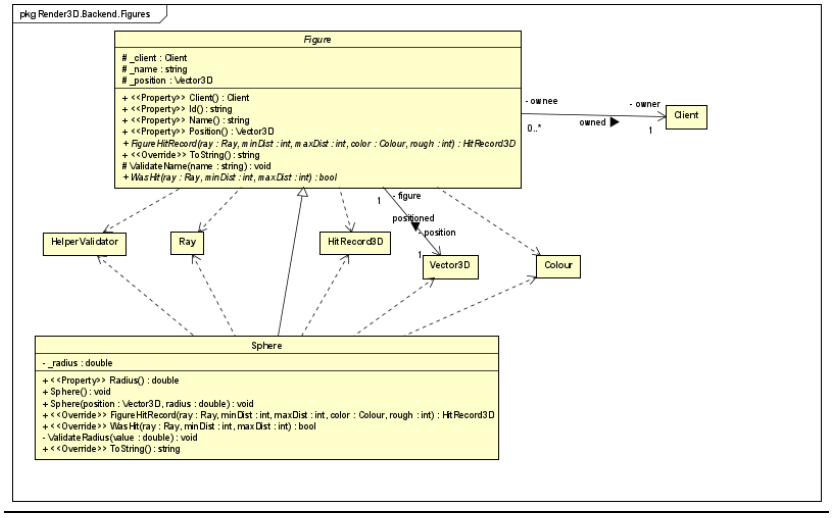
El principal cambio que tiene este namespace respecto a la entrega anterior es la dependencia entre Scene y RandomSingleton. Aunque será detallado en la sección correspondiente, se le delegó la responsabilidad del manejo de números aleatorios a una clase creada con tal fin.

Debido a que es necesario que se utilice la misma instancia de Random para todas las funciones (sino la imagen se ve mal), en la entrega anterior se decidió añadir a todas las funciones involucradas un parámetro adicional, el Random. Esto no estaba bueno porque había que agregar ese parámetro en muchas funciones cuando solo unas pocas en realidad lo necesitaban.

Extrayendo el Random se puede quitar un parámetro en varias funciones, y llamarlo solo cuando es realmente necesario, lo cual mejora la legibilidad y simplicidad del código.

El otro cambio importante que se hizo fue el de agregar identificadores únicos para cada objeto, aunque eso de desarrollará más adelante.

Figures

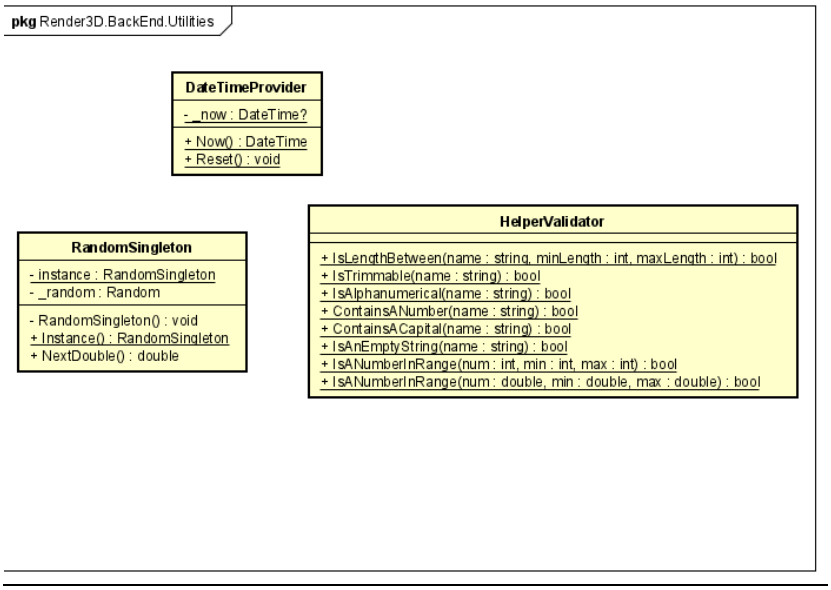


La mayoría de las clases que integran este namespace son aportadas por el script del matemático aunque fueron levemente modificados para mejorar la mantenibilidad del codigo.

Si bien los cambios fueron detallados en la entrega anterior, a modo de resumen se listan las decisiones de diseño más importante que decidimos mantener para esta entrega:

- La función que el matemático llama “isSphereHit”, decidimos que vaya en Sphere en vez de Ray debido a que la esfera tiene todos los datos para determinar si un rayo pasa por ella. Perfectamente este método podría haber ido en Ray, pero si en el futuro hay múltiples figuras, la clase Ray quedaría extremadamente larga.
- Aprovechar la herencia que ya habíamos implementado en Figure, de manera tal que la función que se pueda llamar a “isFigureHit” sin importarnos qué tipo de figura es.

Utilidades

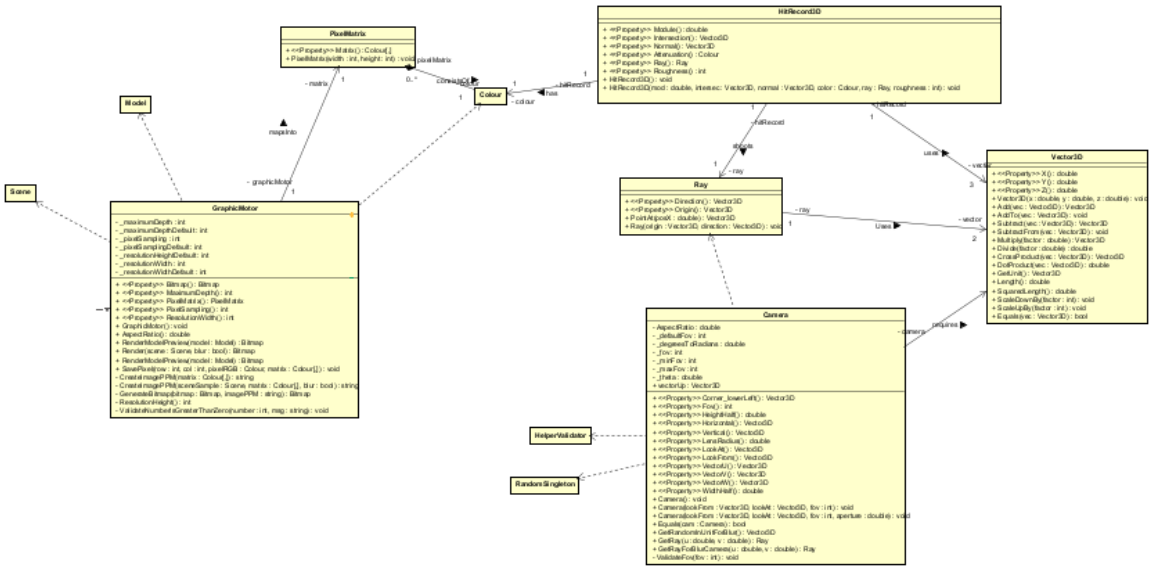


En este namespace se agregó la clase RandomSingleton.

Dado que las funciones de renderizar requieren que siempre se use la misma instancia del random, utilizamos el patrón singleton con el fin de mejorar la calidad del código. De esta forma logramos reducir tener que pasar el random a múltiples funciones, aspecto que impactaba en varias clases.

Una ventaja de esta decisión es que, al asignar la responsabilidad de los números aleatorios a una clase experta en ello, si en el futuro se desea cambiar la forma de calcular los números aleatorios, solo habría que modificar la clase RandomSingleton.

GraphicMotor Utilities

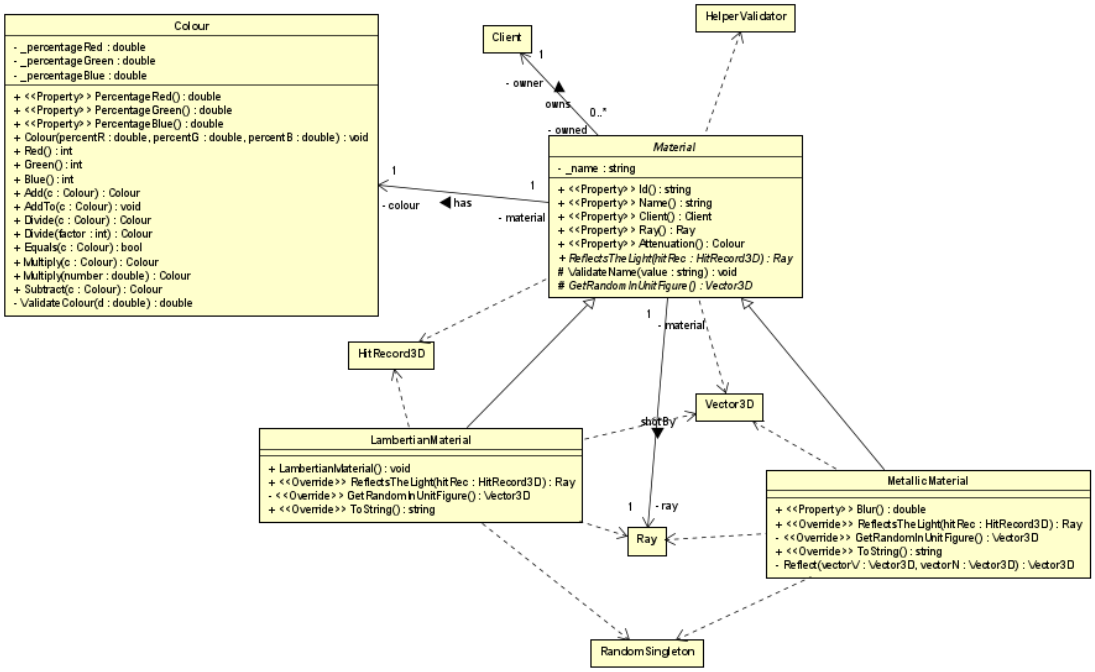


Para poder implementar la nueva funcionalidad de la cámara se agregó un nuevo atributo a guardar, la apertura. Esto llevó a realizar cambios en la cámara donde se creó un constructor extra el cual tiene un comportamiento distinto en algunas partes de la cámara.

La apertura se guarda en la cámara. Si bien hubiera sido más conveniente y mantenible guardar el Lensradius de la cámara, esto implicaba realizar cambios profundos en el dominio, la lógica del renderizado y la base de datos.

A su vez, nos gustaría recalcar que para renderizar una imagen se arrastra un booleano por varios métodos. Pasar un booleano contradice las prácticas de *Clean Code* porque la función no realiza una única cosa, pero por motivos de fuerza mayor no se pudo corregir.

Materials



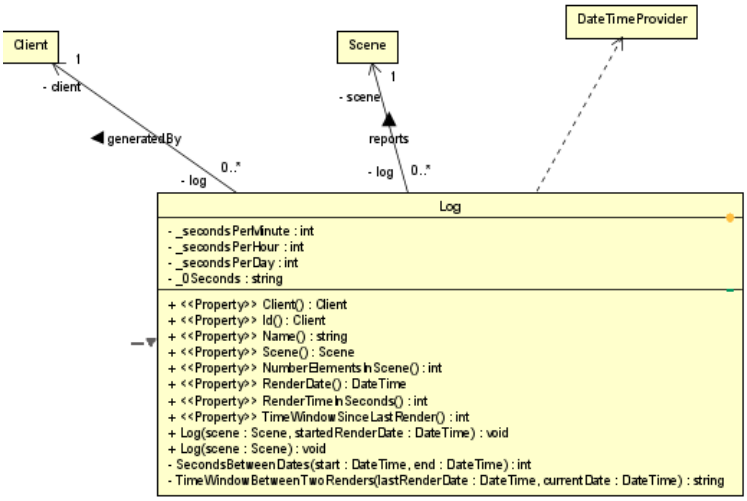
En la entrega anterior, se mencionó que se optó por utilizar herencia para simplificar al agregar nuevos materiales en el sistema. Esta decisión resultó ser acertada, ya que, al implementar un nuevo material, como el material metálico, solo fue necesario crear una clase que heredara de la clase base y que implementara los métodos abstractos.

Al utilizar la herencia, se pudo reutilizar el código existente en la clase padre y se evitó tener que realizar cambios en otras partes del sistema que ya utilizaban la superclase. Esto permite una mayor escalabilidad, ya que un nuevo material puede ser implementado sin afectar el funcionamiento de los materiales existentes.

Sin embargo, luego de haber terminado nos percatamos de que no todos los materiales tienen un color asociado, un ejemplo de esto es el vidrio. Por lo que no sería del todo correcto según el principio de sustitución de Liskov que la property attenuation esté en la clase padre.

Logs

Se decidió crear un nuevo namespace para agrupar las clases cuya responsabilidad sea vinculada a los logs. Si bien solo contiene una clase, en el futuro puede que se sume alguna otra. A continuación se muestra el diagrama:



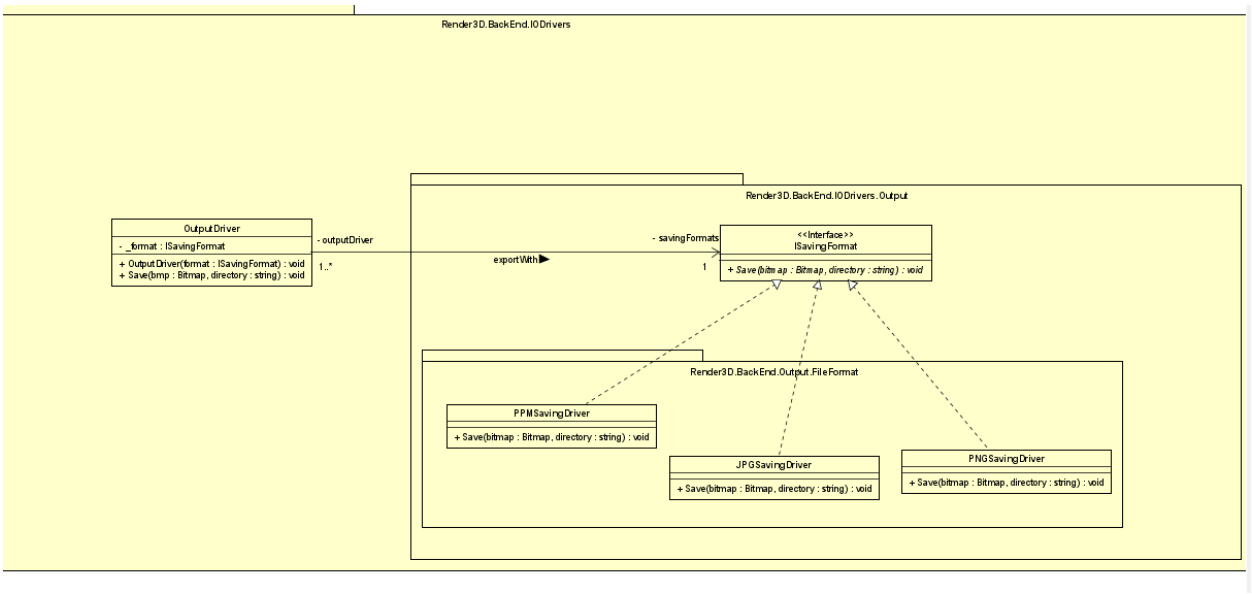
Una decisión de diseño tomada fue utilizar dos constructores distintos, uno para escenas y otro para previews. Como setean atributos de distinta manera, el código repetido es muy poco. No obstante, si en el futuro se agregan más tipos de logs, lo más correcto sería utilizar polimorfismo y que haya una clase por cada tipo de log.

Otra decisión tomada es la de evitar pasar por parámetro atributos que la misma función puede calcular. Por ejemplo, en el constructor de logs para escena (el que recibe dos parámetros) solo se indica la fecha de inicio del renderizado. Esto se debe a que la fecha de fin de renderizado es prácticamente igual a la fecha actual (la diferencia es de milisegundos). De esta forma sacrificamos levemente la precisión de los datos, con el fin de que las funciones sean más simples y legibles.

IODrivers

En este namespace se colocaron todas las clases que hacen posible manejar archivos. Si bien en esta instancia solo se exporta, si en el futuro se desea importar archivos el cambio sería mínimo ya que se sigue OCP.

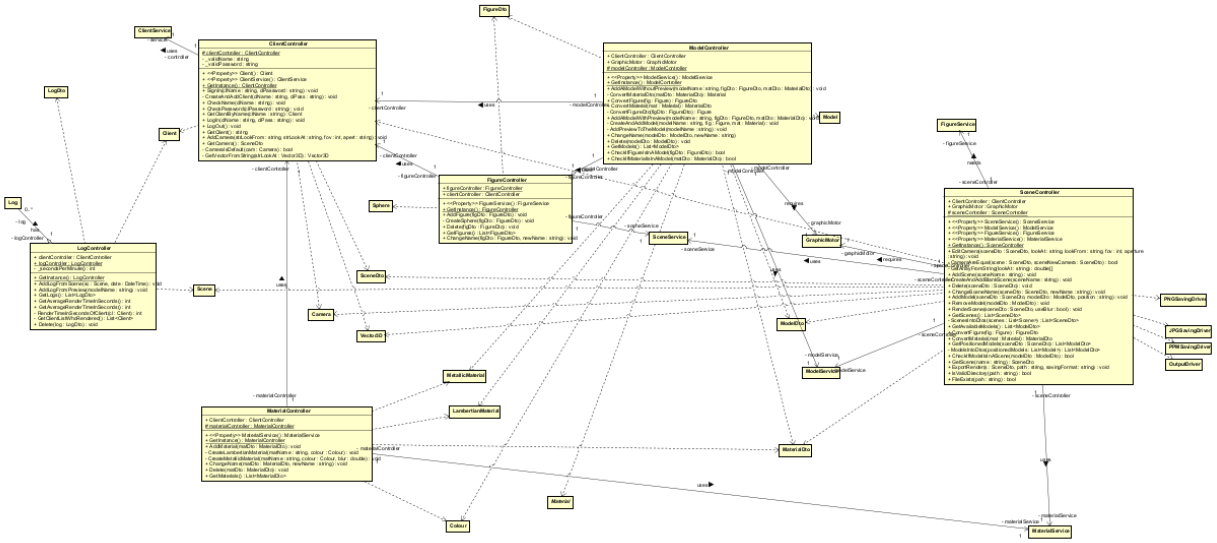
A continuación, se muestra el diagrama de las clases con los respectivos namespaces:



OutputDriver es la clase manejadora del guardado de archivos. Como no queremos que OutputDriver dependa de clases de tan bajo nivel como son los formatos de guardado utilizamos el patrón de diseño strategy, de forma tal que a OutputDriver en su constructor se le inyecta una dependencia al ISavingFormat. De esta forma evitamos que OutputDriver dependa de los formatos específicos.

A su vez, según la distribución de namespaces logramos que se cumpla DIP e ISP, por lo que logramos que los formatos de los archivos (png,jpg,ppm) dependan de ISavingFormat y no al revés.

Controladores

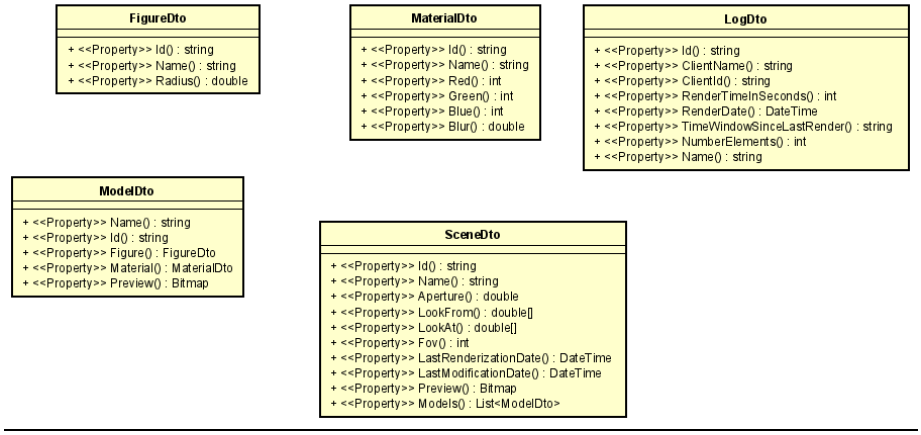


Al principio pensamos en hace un único controlador de fachada. Pero debido a la cantidad de métodos tuvimos que separarlo en casos de usos. A pesar de esto, la comunicación entre los

controladores es alta. Lo ideal hubiera sido delegar a otros para evitar el acoplamiento y reutilizar código, pero por falta de tiempo esto no fue posible.

Por último, se usó el patrón Singleton para la creación de los Controladores dado que estos son llamados desde distintas partes de la interfaz. Lo que nos evitó el uso del parenting para poder acceder a la lógica.

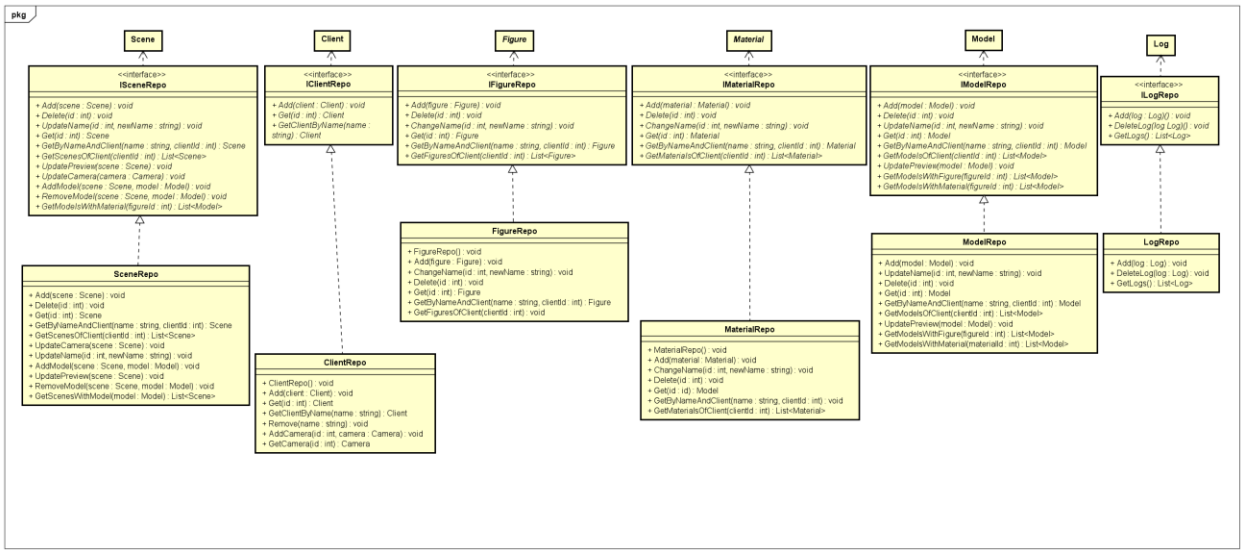
Dtos



Los Data Transfer Object fueron agregados al proyecto de la lógica con el fin de quitar la dependencia de la IU del dominio. De esta forma se reduce el acoplamiento y no le permitimos a la IU saber sobre lo que pasa en el back permitiendo una mayor mantenibilidad y portabilidad del sistema.

Estos se utilizan en el dialogo entre el usuario con los controladores. Aun así, es verdad que en algunos casos tenemos un mal uso. El caso más claro es en el guardado de la cámara por defecto del cliente, donde usamos el dto de escena cuando deberíamos haber creado uno para la cámara, pero por temas de tiempo no se pudo hacer más prolijo.

Interface repositories

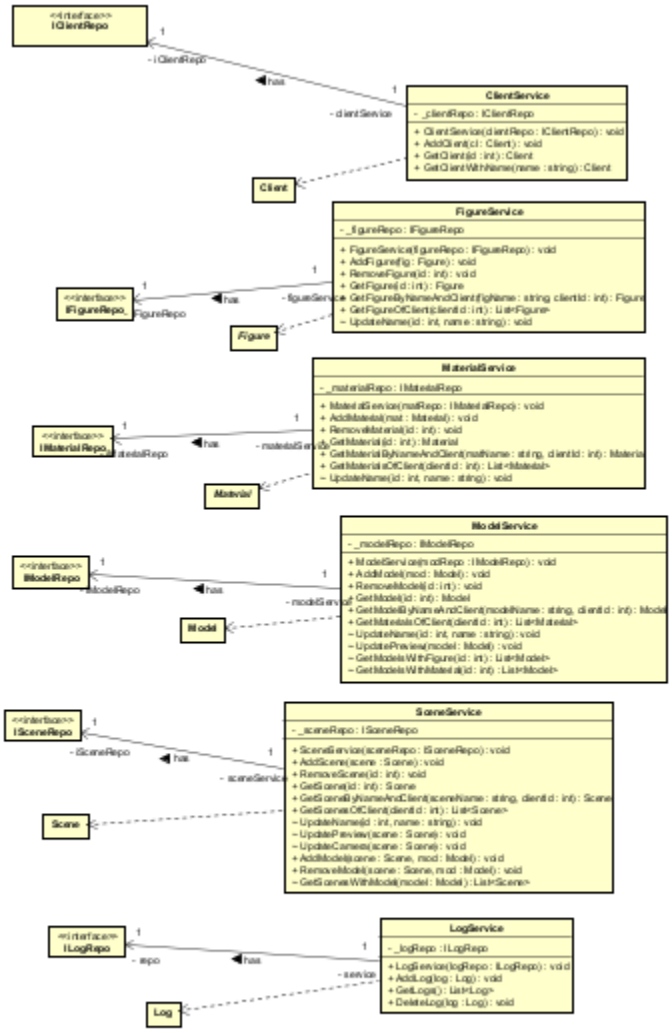


Para los repositorios se decidió usar una interfaz que estaría ubicada en la lógica, y seria implementada en el repositorio de esta forma generamos una inyección de dependencia cumpliendo DIP.

Luego para evitar la dependencia circular y la dependencia de la IU con el repositorio se utilizó el patrón fabrica donde, en la clase repoFactory asignamos los repositorios implementados a los servicios.

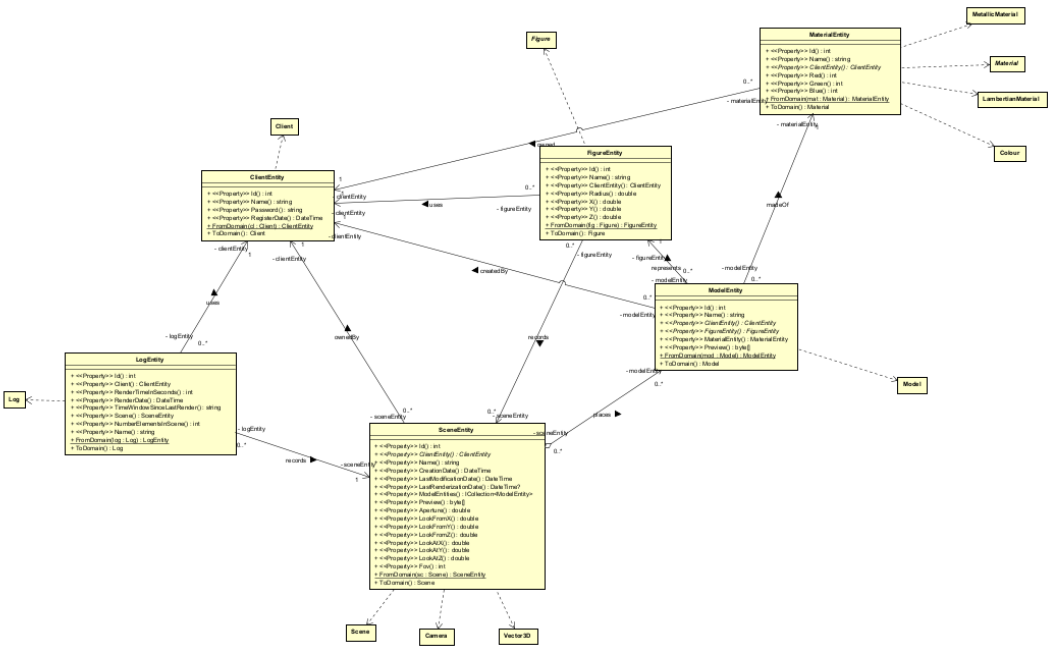
Se decidió usar lazy loading en vez de eager loading debido a que muchos métodos como el GetScenesWithModel entre otros los cuales no necesitan de todos los atributos de la entidad.

Servicios



Como mencionamos antes, los servicios reciben a través del repoFactory la implementación del repositorio y funcionan como un pasamanos entre el controlador (lógica), y el repositorio. De esta forma el controlador no trabaja directamente con el repositorio.

Entities



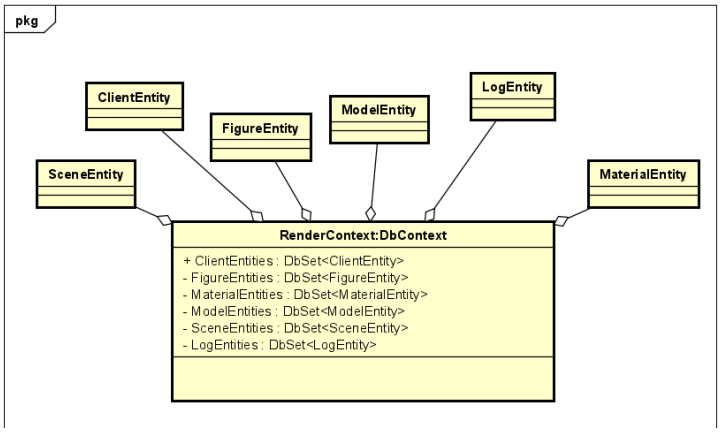
Las entidades fueron modeladas en base al code first y a su vez a TPH (Table per Hierarchy) Esta decisión se tomó, dado que la única entidad que se ve afectada de forma negativa sería el material, dado que el material lambertiano siempre tiene el atributo de blur en nulo. Aun así, esto facilitó bastante el uso del Entity Framework y el manejo del repositorio, por ende, consideramos que fue una buena decisión.

En caso de que en el futuro se deseara agregar nuevos materiales o figuras con atributos muy distintos a los actuales se debería reconsiderar el uso del tph.

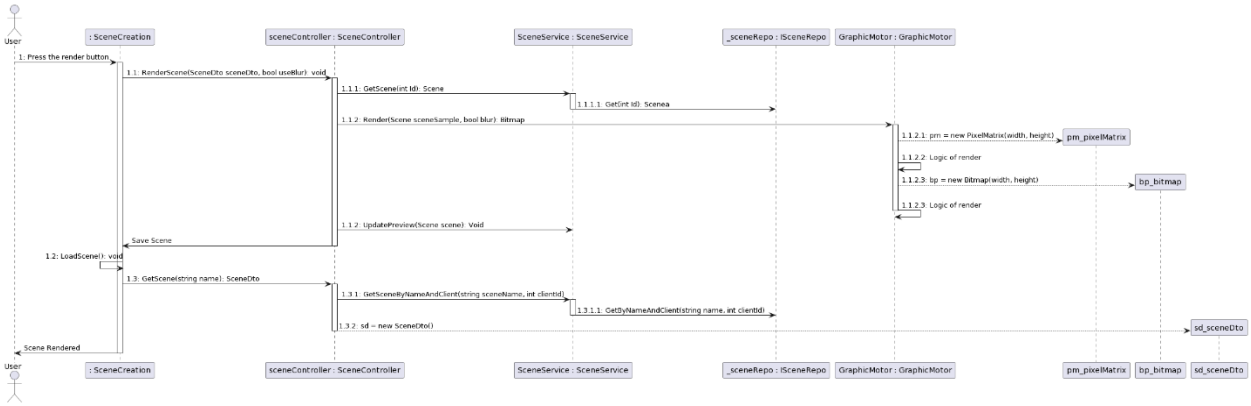
Decidimos que todas las entidades tengan un Id como clave primaria. Por más que hay otras como la clave compuesta por nombre propio y el del cliente, esto traía un problema. Debido a que por ejemplo cuando cambiáramos el nombre de una figura, esto implicaría ir a todos los modelos que la utilizan y cambiar la clave foránea que se tenía sobre esa figura lo cual es poco eficiente.

A su vez si en el futuro se desea que el nombre deje de ser único, las entidades no se verían afectadas.

Context

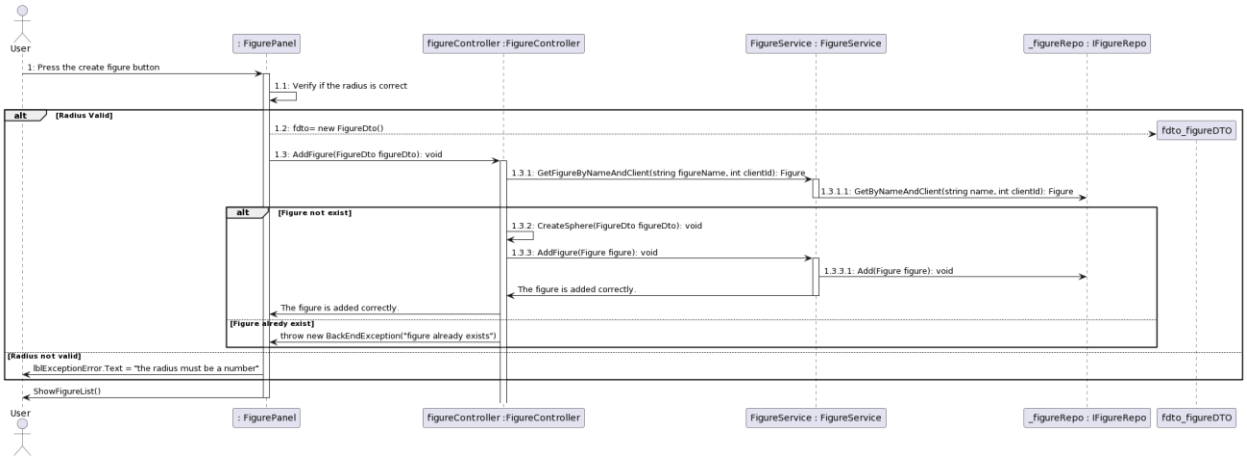


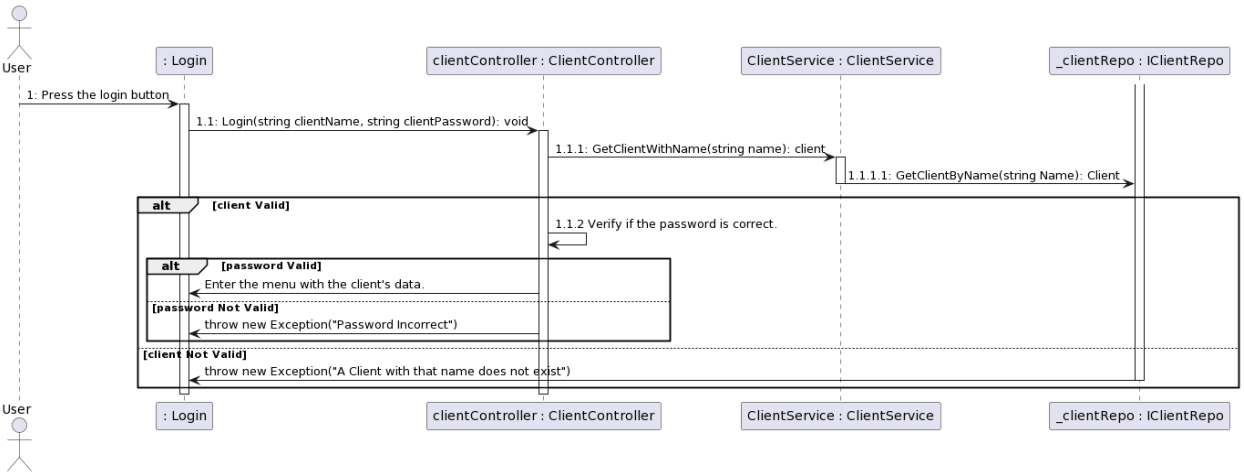
Diagramas de secuencia



Se decidió utilizar la ejecución del render dado que es la que atraviesa por más partes del sistema.

Pasando por todos los repositorios y cumpliendo muy bien la ley de Demeter. Donde el usuario solo se comunica con la interfaz, luego baja a la lógica en el controlador, luego pasa a los servicios, donde como explicamos antes estos no tiene ninguna función por fuera de la comunicación entre el controlador y el repositorio. Vemos que cuando vuelve también es el controlador el encargado de llamar al backEnd para que se renderice.

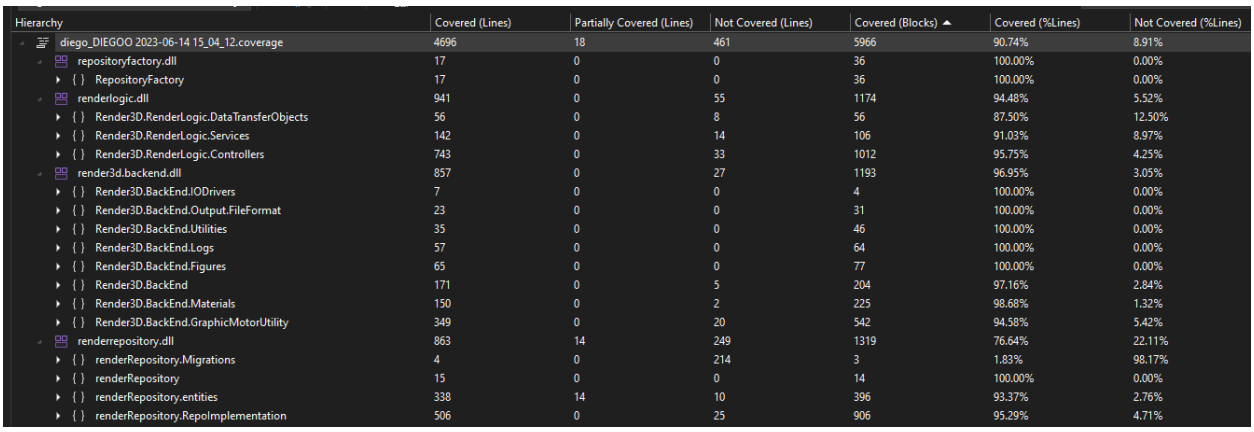




Estos últimos dos diagramas fueron elegidos de forma arbitraria, en donde mostramos casos alternativos, que aún se cumplen Demeter y que mantiene la separación de capas pasando por IU, lógica y luego Repositorio.

Cobertura de tests

El código fue probado ampliamente a través de tests unitarios que intentaron cubrir tanto los happy paths como los sad paths. En la siguiente imagen se indica los resultados obtenidos.

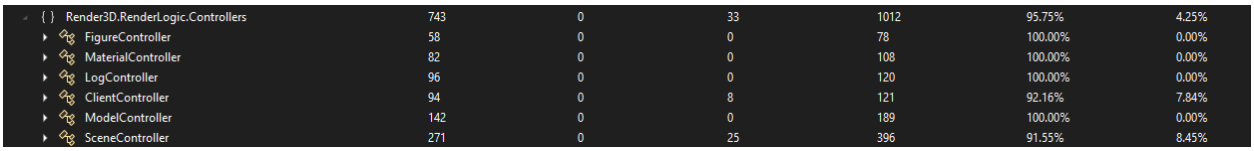


Hierarchy	Covered (Lines)	Partially Covered (Lines)	Not Covered (Lines)	Covered (Blocks)	Covered (%Lines)	Not Covered (%Lines)
diego_DIEGOO 2023-06-14 15_04_12.coverage	4696	18	461	5966	90.74%	8.91%
repositoryfactory.dll	17	0	0	36	100.00%	0.00%
{ } RepositoryFactory	17	0	0	36	100.00%	0.00%
renderlogic.dll	941	0	55	1174	94.48%	5.52%
{ } Render3D.RenderLogic.DataTransferObjects	56	0	8	56	87.50%	12.50%
{ } Render3D.RenderLogic.Services	142	0	14	106	91.03%	8.97%
{ } Render3D.RenderLogic.Controllers	743	0	33	1012	95.75%	4.25%
render3d.backend.dll	857	0	27	1193	96.95%	3.05%
{ } Render3D.BackEnd.IODrivers	7	0	0	4	100.00%	0.00%
{ } Render3D.BackEnd.Output.FileFormat	23	0	0	31	100.00%	0.00%
{ } Render3D.BackEnd.Utilities	35	0	0	46	100.00%	0.00%
{ } Render3D.BackEnd.Logs	57	0	0	64	100.00%	0.00%
{ } Render3D.BackEnd.Figures	65	0	0	77	100.00%	0.00%
{ } Render3D.BackEnd	171	0	5	204	97.16%	2.84%
{ } Render3D.BackEnd.Materials	150	0	2	225	98.68%	1.32%
{ } Render3D.BackEnd.GraphicMotorUtility	349	0	20	542	94.58%	5.42%
renderrepository.dll	863	14	249	1319	76.64%	22.11%
{ } renderRepository.Migrations	4	0	214	3	1.83%	98.17%
{ } renderRepository	15	0	0	14	100.00%	0.00%
{ } renderRepository.entities	338	14	10	396	93.37%	2.76%
{ } renderRepository.RepoImplementation	506	0	25	906	95.29%	4.71%

Observamos que los proyectos están ampliamente testeados teniendo casi todos ellos una cobertura mayor al 90%.

Con respecto a RenderLogic, se observa que los Dtos tienen una cobertura de 87,5%. Esto se debe a que se priorizó testear fuertemente los controladores dado que ellos poseen comportamiento.

Otro aspecto para destacar es que a excepción de SceneController y ClientController, los controladores se encuentran 100% testeados.



{ } Render3D.RenderLogic.Controllers	743	0	33	1012	95.75%	4.25%
FigureController	58	0	0	78	100.00%	0.00%
MaterialController	82	0	0	108	100.00%	0.00%
LogController	96	0	0	120	100.00%	0.00%
ClientController	94	0	8	121	92.16%	7.84%
ModelController	142	0	0	189	100.00%	0.00%
SceneController	271	0	25	396	91.55%	8.45%

Con respecto a scene controller las líneas que no se testearon son las siguientes:

```
1 reference | - changes | -authors, -changes
public void ExportRender(SceneDto s, string directory, string savingFormat)
{

    Bitmap bitmap = s.Preview;
    ISavingFormat format;

    switch (savingFormat)
    {
        case "png":
            format = new PNGSavingDriver();
            break;
        case "jpg":
            format = new JPGSavingDriver();
            break;
        case "ppm":
            format = new PPMSavingDriver();
            break;
        default:
            throw new Exception("invalid format");
    }

    OutputDriver o = new OutputDriver(format);
    o.Save(bitmap, directory);

}

1 reference | - changes | -authors, -changes
public bool IsValidDirectory(string path)
{
    return  Directory.Exists(path);
}

1 reference | - changes | -authors, -changes
public bool FileExists(string path)
{
    return File.Exists(path);
}
}
```

Decidimos reducir al mínimo la cantidad de pruebas que simulan comportamientos como la búsqueda de directorios o la búsqueda y guardado de archivos. Esto se debe a que el resultado de la prueba depende de varios factores, como los permisos, si el archivo está en uso, entre otros. En caso de que estas pruebas fallen, resulta difícil determinar la causa exacta y/o poder replicarlos. Y como estas líneas ya eran testeadas en OutputDriver no vimos prioritario volver a testearlos acá.

Con respecto a cliente, esto no se logró por cuestión de tiempo.

Instalación

Una vez bajado el repositorio, dentro de la carpeta entrega2 se encuentra el ejecutable.

Dentro del directorio DB se encuentran los .bak y scripts SQL solicitados.

Para poder conectar con la base de datos será necesario abrir el appconfig en UserInterface si se desea correr el programa,y UnitTestRender3D para correr las pruebas.