Universidad de las Ciencias Informáticas

FACULTAD 6

# Trabajo de Diploma para optar por el título de Ingeniero en Ciencias Informáticas

**Título:** Visualización de capas de agua, usando la técnica mapeo de textura de proyección, para un visor de modelos digitales de elevación en tres dimensiones.

**Autor:** Nelson González Peñate

**Tutores:** MSc. Gilberto Arias Naranjo

Dr. Liesner Acevedo Martínez

La Habana, 5 de junio del 2015

*"Nosotros, los mortales, logramos la inmortalidad en las cosas que creamos en común y que quedan después de nosotros."*

*Albert Einstein*

# Declaración de Autoría

Declaro ser autor del presente trabajo de diploma y reconozco a la Universidad de las Ciencias Informáticas los derechos patrimoniales de la misma, con carácter exclusivo.

Para que así conste, firmo la presente a los ___ días del mes de ___________ del año ______.

Nelson González Peñate

____________

Firma del autor

| MSc. Gilberto Arias Naranjo | Dr. Liesner Acevedo Martínez |
| --- | --- |
| ____________ | ____________ |
| Firma del tutor | Firma del tutor |

## Dedicatoria

Por su apoyo incondicional y sus sabios consejos esta tesis la dedico a mis padres Odalis y Nelson y a mis abuelos Gloria, Teodoro y Cecilia. También por su apoyo y compañía quiero dedicar este esfuerzo a mis hermanos Noslen y Ernesto y a mi novia Ismary.

## Agradecimientos

Gracias a mis padres y abuelos por el apoyo infinito que me han dado, por siempre estar al tanto de mí. A mis hermanos. A mi novia por su compañía, amor y paciencia con mi carácter. Quiero agradecer a mis amigos y compañeros que me han ayudado a seguir adelante. A los que han sido y son compañeros de apartamento por compartir su vida conmigo.

También quiero agradecer a mis tutores por el apoyo y el tiempo que han dedicado para la tutoría de este trabajo. A mis profesores de la carrera. A los del CEMC, del tribunal y compañeros que colaboraron con sugerencias y revisiones del documento. A los que dedicaron su tiempo y fueron partícipe del Juicio de Expertos. A todo el que fue y es del Movimiento de Programación Competitiva.

## Resumen

Los Sistemas de Visualización de Información Geográfica en tres dimensiones se aplican en muchos campos de la vida del hombre. En el Centro de Estudios de Matemática Computacional se desarrolla un software de este tipo nombrado Componente para la Visualización 3D de Terrenos. Uno de los usos actuales de este componente es la visualización de simulaciones de fenómenos naturales como las crecidas de un río o el embate de un tsunami. Esto hace necesario que exista realismo en la representación de las capas de agua. Sin embargo, actualmente las mismas son representadas con colores sólidos, lo que puede crear confusiones en la interpretación de los fenómenos que se intenten representar. En este trabajo se propone una solución a este problema basada en la técnica Mapeo de Textura de Proyección. Esta técnica que brinda un balance entre eficiencia y calidad de la imagen, se implementó haciendo uso de *shaders*, lo que permitió reducir la carga del CPU al desplazar la mayoría de los cálculos hacia el GPU. Las pruebas de software y el juicio de expertos realizados demostraron un incremento en el realismo de la visualización del agua, manteniendo la eficiencia de la visualización.



**Palabras clave:** OpenGL, realismo, reflexión, refracción, *shaders*, superficie del agua, tres dimensiones.

# Abstract

*The Geographic Information Visualization Systems are widely used in the life of people. In the Computational Mathematics Study Center a software called Component for 3D Visualization of Terrain is being developed. One of the current uses of this component is the visualization of simulations of natural phenomena such as floods from a river or a tsunami surge. That´s why it is necessary that the water layers are rendered as real as possible. However, they are currently represented with solid colors, which can cause confusion when interpreting the phenomena that is being represented. In this work a solution to this problem based on Texture Mapping Projection technique is proposed. This technique that provides a tradeoff between efficiency and image quality was implemented using shaders, which releases CPU load since most of the computations are done in the GPU. Software tests and the expert's judgment demonstrated an increased realism of the visualization of water, keeping the efficiency of visualization.*



***Keywords:*** *OpenGL, reality, reflexion, refraction, shaders, water surface, three dimensions.*

# ÍNDICE

INTRODUCCIÓN ....................................................... 1
CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA ....................................................... 5
1.1 Ópticas del agua ....................................................... 5
*1.1.1 Dispersión de la luz* ....................................................... 5
*1.1.2 Cáusticas del agua* ....................................................... 5
*1.1.3 Absorción, reflejo y transmisión de la luz* ....................................................... 6
1.2 Teoría de la Reflexión y Refracción de la luz ....................................................... 6
*1.2.1 Leyes de Reflexión y Refracción de la luz* ....................................................... 6
*1.2.2 Coeficientes de reflexividad y transmisión de Fresnel* ....................................................... 7
1.3 Técnicas de visualización de superficies de agua ....................................................... 8
1.4 Técnicas de optimización ....................................................... 10
*1.4.1 Técnicas de recorte o desecho* ....................................................... 10
1.5 Tecnologías, Herramientas y Metodología a utilizar ....................................................... 11
*1.5.1 Tecnologías para la representación 3D* ....................................................... 11
*1.5.2 Metodología de desarrollo* ....................................................... 12
*1.5.3 Lenguajes de programación* ....................................................... 13
*1.5.4 Shaders* ....................................................... 14
*1.5.5 Lenguaje de modelado* ....................................................... 15
*1.5.6 Entorno de Desarrollo Integrado* ....................................................... 15
*1.5.7 Herramienta CASE* ....................................................... 15
Conclusiones del capítulo ....................................................... 16
CAPÍTULO 2: ANÁLISIS Y DISEÑO ....................................................... 17
2.1 Modelo del Dominio ....................................................... 17
2.2 Especificación de los requisitos del sistema ....................................................... 18
*2.2.1 Requisitos funcionales* ....................................................... 18
*2.2.2 Requisitos no funcionales* ....................................................... 18
2.3 Modelado del sistema ....................................................... 19
*2.3.1 Actores del sistema* ....................................................... 19
*2.3.2 Casos de Uso del Sistema* ....................................................... 20
*2.3.3 Arquitectura del sistema* ....................................................... 21

*2.3.4 Diagrama de clases* ..... 22
*2.3.5 Patrones de diseño* ..... 23
2.4 Proceso de visualización propuesto ..... 25
Conclusiones del capítulo ..... 27
CAPÍTULO 3: IMPLEMENTACIÓN Y PRUEBAS ..... 29
3.1 Estándares de codificación ..... 29
*3.1.1 Convenciones de nombre* ..... 29
*3.1.2 Estilo de Código* ..... 30
3.2 Implementación del proceso de visualización de una capa de agua ..... 31
*3.2.1 Captura de la textura de reflexión y refracción* ..... 31
*3.2.2 Creación de los shaders* ..... 32
*3.2.3 Transmisión de los datos al GPU* ..... 33
*3.2.4 Visualización* ..... 34
3.3 Diagrama de Componentes ..... 39
3.4 Pruebas de software ..... 40
*3.4.1 Estrategia de pruebas* ..... 41
*3.4.2 Pruebas Unitarias* ..... 41
*3.4.3 Pruebas de Integración* ..... 44
*3.4.4 Pruebas del Sistema* ..... 45
3.5 Validación del componente ..... 46
Conclusiones del Capítulo ..... 48
CONCLUSIONES ..... 49
RECOMENDACIONES ..... 50
Referencias Bibliográficas ..... 51
Anexo A: *Pipeline* de OpenGL ..... 54
Anexo B: Vistas del componente ..... 55
Anexo C: Síntesis biográfica de MSc. Yailen Costa Marrero ..... 56
Anexo D: Síntesis biográfica de Lic. Edisel Navas Conyedo ..... 57
Anexo E: Síntesis biográfica de Ing. Alexis Echemendía González ..... 58
Anexo F: Síntesis biográfica de Dr. Jorge Gulín González ..... 59

# ÍNDICE DE FIGURAS

**Fig. 1** Fenómeno de Reflexión y Refracción de la luz. ....................................................................7
**Fig. 2** (a) Reflexión del ambiente. (b) Reflexión local de un barco y del ambiente. Fuente (Bruneton et al., 2010). ....................................................................9
**Fig. 3** Modelo del Dominio. ....................................................................17
**Fig. 4** Diagrama de Casos de Uso. ....................................................................20
**Fig. 5** Diagrama de clases. ....................................................................22
**Fig. 6** Diagrama del patrón decorador para los shaders. ....................................................................25
**Fig. 7** Imagen del Visor de Terrenos visualizando una escena que contiene agua. ....................................................................26
**Fig. 8** Vista reflejada de la imagen respecto al plano de la superficie del agua. ....................................................................27
**Fig. 9** Diagrama de Componentes. ....................................................................39
**Fig. 10** No conformidades detectadas durante las pruebas unitarias. ....................................................................44
**Fig. 11** Pipeline de OpenGL. Fuente: (Shreiner, 2013) ....................................................................54
**Fig. 12** Vista del juego de datos Monai. ....................................................................55
**Fig. 13** Vista del juego de datos Monai con oleaje. ....................................................................55

# ÍNDICE DE TABLAS

**Tabla 1** Actor del sistema. ....................................................................20
**Tabla 2** Descripción del caso de uso Visualizar agua. ....................................................................20
**Tabla 3** Uso y sintaxis de nombre. ....................................................................29
**Tabla 4** Caso de Prueba #1 para el Fragment Shader. ....................................................................42
**Tabla 5** Caso de Prueba #2 para el Fragment Shader. ....................................................................43
**Tabla 6** Datos y resultados de las pruebas de rendimiento aplicadas al software. ....................................................................46
**Tabla 7** Valoración de los expertos sobre la calidad de la reflexión y refracción de la luz en el agua. .......47

# INTRODUCCIÓN

La informática ha experimentado grandes y numerosos avances en los últimos años. Gracias a ello ha sido posible el desarrollo de software de utilidad e impacto para la sociedad. Un ejemplo que lo evidencia son los Sistemas de Visualización de Información Geográfica (SVIG), pues su activa utilización se ha traducido en una reducción de costes y mejoras continuas en algunos procesos y actividades del hombre. Algunas de estas actividades pueden ser: en el campo de la Ingeniería Civil para el diseño de infraestructura diversa como son minas de cielo, carreteras y presas; en el campo de manejo y planeación de recursos naturales para la planeación urbana y ambiental, teledetección, estudios de impacto ambiental, prevención de desastres y localización de sitios industriales; en el sector militar se usa en el análisis del terreno para el manejo del campo de batalla, análisis de tráfico, animación para simuladores de vuelo, entre otros. Por esta razón su uso ha sido asimilado por universidades, gobiernos, empresas e instituciones como sistema de apoyo a la toma de decisiones.

En el año 2013 en la UCI se inscribe el proyecto de investigación: "Representación multicapas de información geográfica, tomando como base Modelos Digitales de Elevación[1], para el desarrollo y análisis de simulaciones en 3D." suscrito al Centro de Estudios de Matemática Computacional (CEMC) que se realiza en conjunto al Centro Internacional de Métodos Numéricos en la Ingeniería (CIMNE), con sede en Barcelona, España. En este proyecto se realizó un componente para la visualización en 3D de información geográfica (en lo adelante Visor de Terrenos). El objetivo fundamental del proyecto es la representación en 3D, de varias capas de información geográfica, incluyendo aquellas que formen parte de resultados de ejecutar simulaciones sobre determinado terreno como inundaciones o de tsunamis. En estos casos el Visor de Terrenos visualiza sobre la capa base del terreno una animación construida por la superposición de varias capas de agua.

La representación del agua es uno de los efectos que puede incrementar el realismo percibido en escenas virtuales. Entiéndase por realismo la medida en la que la escena virtual se visualiza de forma continua y se asemeja a la realidad. Sin embargo, actualmente las capas de agua son representadas de igual forma que se representa el terreno, utilizando colores sólidos, es decir no se tiene en cuenta sus propiedades ópticas,

[1] Una representación estadística de la superficie continua del terreno, mediante un número elevado de puntos selectos con coordenadas (x, y, z) conocidas en un sistema de coordenadas arbitrario (Felicísimo, 1994).

lo que resta realismo a la visualización y podría generar confusiones al usuario que esté observando el fenómeno que se intente representar.

A partir de lo descrito anteriormente se plantea como **problema de investigación**:

¿Cómo elevar el nivel de realismo en la representación de capas de agua en el Visor de Terrenos del CEMC?

Dado el problema anterior se define como **objeto de estudio:**

Representación virtual del agua.

Estableciendo como **campo de acción**:

Representación virtual en 3D de superficies de agua.

El **objetivo general** que persigue este trabajo de diploma es: Desarrollar funcionalidades en el Visor de Terrenos que permitan la visualización de superficies de agua teniendo en cuenta las propiedades ópticas de reflexión y refracción de la luz, para lograr un mayor realismo en las escenas.

Los **objetivos específicos** que se derivan son:

- Seleccionar técnicas y algoritmos para la visualización de superficies de agua.
- Analizar y diseñar las funcionalidades que se agregarán al Visor de Terrenos.
- Implementar las técnicas y algoritmos seleccionados.
- Validar las funcionalidades desarrolladas mediante la realización de pruebas.

A partir del marco teórico desarrollado se formula la siguiente **hipótesis de investigación**:

Si se tienen en cuenta las propiedades de reflexión y refracción de la luz en la visualización de la superficie de las capas de agua se incrementará el realismo que presentan actualmente las mismas.

**Métodos de investigación:**

- Método analítico-sintético: Para estudiar por separado las diferentes técnicas y procesos utilizados en la representación en 3D de la superficie del agua, y luego sintetizarlos en la elaboración de las funcionalidades.

- Método histórico-lógico: Para constatar teóricamente la evolución de la representación en 3D de la superficie del agua.
- Análisis documental: En la revisión de la literatura sobre la representación en 3D de la superficie del agua para consultar la información necesaria en el proceso de investigación.
- Método de modelado: Para representar los diagramas correspondientes a las etapas de análisis, diseño e implementación de la solución.
- Método experimental: Para la medición del realismo de las capas de agua y su comparación con el realismo precedente.
- Método hipotético-deductivo: Para la formulación de la hipótesis y arribar a conclusiones.

**Resultados Esperados:**

- Que el Visor de Terrenos cuente con nuevas funcionalidades que permitan visualizar las capas de agua con mayor nivel de realismo, dando la impresión de que efectivamente es agua lo que se está representando, manteniendo una visualización continua con una frecuencia igual o mayor a 30 tramas por segundo.
- Que el Visor de Terrenos y los resultados obtenidos contribuyan a futuras investigaciones permitiendo la realización de nuevos productos que sustituyan las dependencias de soluciones privativas, reduciendo costes y apoyando la soberanía tecnológica de Cuba.

El presente trabajo cuenta con la siguiente estructura:

***Capítulo 1: Fundamentación Teórica***

En este capítulo se aborda la base teórica para comprender el problema planteado. Muestra los principales conceptos relacionados con la representación 3D de superficies de agua en ordenador, las tendencias que existen en la actualidad, así como el análisis y selección de las técnicas y herramientas utilizadas en el desarrollo de la solución.

***Capítulo 2: Análisis y Diseño***

Se selecciona la metodología de desarrollo a utilizar y se listan los requisitos funcionales y no funcionales detectados, en conjunto con los casos de uso que los agrupan. Se muestra la arquitectura del componente. Se exponen el diagrama de clases y los patrones de diseño empleados. Por último se describe una propuesta para el proceso de visualización del agua.

## *Capítulo 3: Implementación y Pruebas*

En este capítulo se describe la implementación de las partes que conforman la solución así como los estándares de codificación utilizados. Se hace uso de pruebas de software y la técnica Juicio de Expertos para validar el correcto funcionamiento del mismo, con lo que queda evidenciado el desempeño del componente en la representación de superficies de agua en tiempo real.

# CAPÍTULO 1: FUNDAMENTACIÓN TEÓRICA

Este capítulo se describen los principales conceptos relacionados con la representación en 3D de superficies de agua, las tendencias que existen en la actualidad, así como el análisis y selección de las técnicas y herramientas utilizadas en el desarrollo de la solución.

## 1.1 Ópticas del agua

La generación de imágenes 3D físicamente precisas es un problema computacional desafiante. Uno de los aspectos que más incide en la precisión (realismo) de la imagen obtenida es la simulación del fenómeno de propagación de la luz. El problema fundamental es que este fenómeno de interacción no tiene lugar solamente en la superficie sino en cualquier punto del medio. En particular cuando se trata de simular los fluidos, como el agua, existen un conjunto de propiedades que hay que tener en cuenta. Estas son: la dispersión, las causticas y la absorción, reflejo y transmisión. A continuación se describen cada una de ellas.

### *1.1.1 Dispersión de la luz*

La dispersión de la luz es la forma de propagación de la energía que se dispersa. Se puede ver como la desviación de un rayo de su trayectoria recta, por irregularidades en el medio de propagación. A diferencia de la reflexión, donde la radiación es desviada en una dirección, algunas partículas tienen la capacidad para dispersar la radiación solar en todas las direcciones. Estas partículas se denominan dispersores (Henderson, 2014).

La dispersión de la luz posibilita que incluso en un ambiente con agua tranquila y cristalina no se vea el fondo con la mejor nitidez.

### *1.1.2 Cáusticas del agua*

Según (RAE, 2012) el significado físico de cáustica es: superficie tangente a los rayos reflejados o refractados por un sistema óptico.

Las cáusticas aparecen como patrones de belleza en superficies difusas, formadas por caminos de luz que se originan en una fuente y visitan superficies reflectantes o refractantes. Son la concentración de la luz que puede quemar. El nombre cáustica viene del griego *Kaustiko* lo que significa quemar (de la Fuente, 2011).

El volumen de las cáusticas es causado por la misma concentración de luz que crea las iluminaciones en la superficie tangente a la luz reflejada o refractada. La diferencia viene del hecho de la presencia de medio participante (partículas en el agua) que afectan la dirección de propagación de la luz y por lo tanto los ases de luz pueden ser vistos como volúmenes.

### *1.1.3 Absorción, reflejo y transmisión de la luz*

Las ondas de luz visibles consisten en un intervalo continuo de longitudes de onda o frecuencia. Cuando una onda de luz con una sola frecuencia incide sobre un objeto puede suceder que la onda de luz:

- Se absorba por el objeto, en cuyo caso su energía se convierte en calor.
- Se refleje en la superficie del objeto.
- Se transmita por el objeto.

Sin embargo rara vez la luz que incide sobre un objeto se limita a una sola frecuencia. Cuando esto ocurre los objetos tienen una tendencia de absorber, reflejar o transmitir selectivamente ciertas frecuencias de luz. La manera en que la luz visible interactúa con un objeto depende de la frecuencia de la luz y la naturaleza de los átomos del objeto (Henderson, 2014).

El agua (como la mayor parte de los fluidos) es un caso especial en que se ponen de manifiesto las tres propiedades mencionadas. Debido a su influencia en la visualización realista del agua, de las propiedades ópticas estudiadas en este epígrafe solamente se aplicarán la reflexión y la transmisión de la luz, teniendo en cuenta las leyes físicas que las sustentan (reflexión y refracción de la luz).

## 1.2 Teoría de la Reflexión y Refracción de la luz

En este epígrafe se describirán física y matemáticamente las leyes asociadas a las propiedades ópticas que se aplicarán en la solución. Estas propiedades fueron seleccionadas en el epígrafe anterior y son las de reflejar y transmitir la luz.

### *1.2.1 Leyes de Reflexión y Refracción de la luz*

Estos fenómenos pueden observarse siempre que un haz de luz viaja de un medio a otro.

- Reflexión: Es la acción de reflejarse parte del haz de luz incidente de regreso al medio de donde proviene (Resnick, 1992).
- Refracción: Parte del haz de luz incidente debe transmitirse al segundo medio desviándose al entrar al mismo (Resnick, 1992).

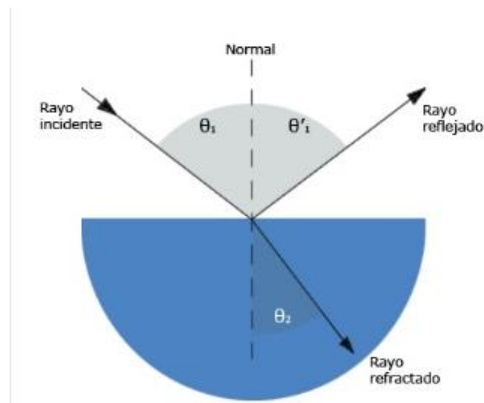


**Fig. 1** Fenómeno de Reflexión y Refracción de la luz.

En la Figura 2 se ilustran los ángulos de incidencia $\theta_1$, de reflexión $\theta'_1$ y refracción $\theta_2$ con respecto al vector normal a la superficie. Usando la notación de la misma:

- Según la **ley de reflexión,** el rayo reflejado se encuentra en el plano de incidencia, y el ángulo de incidencia tiene que ser igual al ángulo de reflexión.

$$\theta_1 = \theta'_1. \quad (1)$$

- Según la **ley de la refracción,** el rayo refractado se encuentra en el plano de incidencia, y se tiene que cumplir la siguiente igualdad:

$$n_1 sen\, \theta_1 = n_2 sen\, \theta_2. \quad (2)$$

La ecuación 2 se conoce como **Ley de Snell**, donde $n_1$ y $n_2$ son constantes llamadas **índice de refracción** del medio 1 y del medio 2. El índice de refracción es la razón entre la velocidad de la luz en el vacío y la velocidad de la luz en el medio (Resnick, 1992).

### *1.2.2 Coeficientes de reflexividad y transmisión de Fresnel*

La potencia inicial de la luz (la potencia inicial es 1) del rayo incidente se divide entre los rayos de reflexión y refracción. La Reflexividad de Fresnel (R) es una fracción de la potencia inicial que es reflejada y la transmisión (T) es otra fracción de la potencia inicial que es refractada (Grindstad y Rasmussen, 2011).

Según (de la Fuente, 2011) la relación entre R y T está sometida a la ley de conservación de la energía por lo que

$$R + T = 1. \tag{3}$$

La derivación de las expresiones para R y T está basada en la teoría electromagnética de los dieléctricos y asumiendo que la luz no es polarizada según (de la Fuente, 2011)

$$R = \frac{R_s + R_p}{2} \tag{4}$$

donde $R_s$ y $R_p$ son los coeficientes de reflexión para ondas paralelas y perpendiculares al plano de incidencia. Estos coeficientes son bastante costosos de calcular incluso en hardware de gráficos, pues se deben calcular en cada trama para cada píxel que se dibujará en pantalla. En aplicaciones en tiempo real, necesitamos una aproximación que sea mucho menos costosa de calcular y también lo suficientemente precisa como para no afectar mucho la calidad de la imagen. (Schilick, 1994) ofrece una aproximación de la Reflexividad de Fresnel que es bastante exacta para la mayoría de las sustancias:

$$Rf(\theta_i) = Rf(0^\circ) + (1 - Rf(0^\circ))\,(1 - \cos\theta_i)^5 \tag{5}$$

Esta ecuación ofrece unos resultados razonablemente precisos a un menor costo computacional. Con esta aproximación $Rf(0^\circ)$ es el único parámetro que controla la reflexividad y se calcula según (de la Fuente, 2011) de la siguiente forma:

$$Rf(0^\circ) = \left(\frac{n-1}{n+1}\right)^2 \tag{6}$$

Asumiendo que el índice de refracción de $n_1$ es 1, como es el índice del aire, y usando n en lugar de $n_2$.

## 1.3 Técnicas de visualización de superficies de agua

El realismo de la superficie del agua se puede lograr aplicando técnicas especiales de visualización. Según (de la Fuente, 2011) en la actualidad coexisten tres familias de estas técnicas:

1. Rayo de selección: esta técnica considera que la escena será observada desde un punto de vista específico, calculando la imagen observada basada solamente en geometría y aspectos básicos de la ley de reflexión. Imágenes generadas con esta técnica carecen de calidad visual y no siempre tienen exactitud los cálculos, dada la consideración antes mencionada (de la Fuente, 2011).
2. Trazado de rayos: Es similar a las técnicas Rayo de selección, pero emplean una más avanzada simulación óptica. El rayo de reflexión se calcula recursivamente en dependencia de la calidad que se quiera lograr (Rodgers, 2014).
3. Rasterización: geométricamente se proyectan objetos de la escena a una imagen plana (en dos dimensiones). Las imágenes en 3D se proyectan a texturas en dos dimensiones (Jensen y Golias, 2011). Se aplica para proyectar las escenas a píxeles en los monitores.

Las familias de técnicas Rayo de selección y Trazado de rayos al reflejar un rayo tienen que buscar su colisión con un objeto de la escena (de la Fuente, 2011). Por lo costoso que es computacionalmente realizar este proceso, se ve afectado el logro de las visualizaciones en tiempo real[2]. La presente investigación se centra únicamente en las técnicas basadas en la familia Rasterización para poder lograr el realismo requerido. Los principales enfoques de rasterización se describen a continuación.

### *Mapeo del Ambiente*

Es una técnica desarrollada en el año 1976 que simula el resultado de la técnica Trazado de rayos. Se mapea el ambiente hacia una textura usando hardware de mapeado de textura, con esto se pueden obtener reflexiones globales e iluminación en tiempo real (Blinn y Newell, 1976).

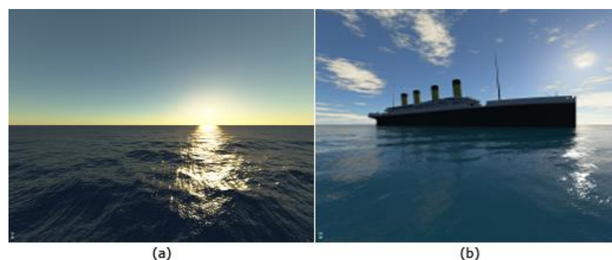


**Fig. 2** (a) Reflexión del ambiente. (b) Reflexión local de un barco y del ambiente. Fuente (Bruneton et al., 2010).

[2] Tiempo real: se refiere a crear imágenes suficientemente rápido (a un mínimo de 30 fotogramas por segundo) de manera que el usuario pueda interactuar con el entorno virtual sin perder la continuidad de la visualización.

Esta técnica es válida para situaciones en las que no se necesiten reflejar objetos cercanos al área de reflexión. Reflexiones locales no pueden tratarse con un mapeado por píxeles del entorno en el océano (Bruneton et al., 2010). Por ejemplo, para la visualización de un océano donde la escena esté compuesta solamente de mar y cielo esta técnica es factible (Figura 1, a).

***Mapeo de Textura de Proyección***

Esta técnica permite reflexiones locales y globales en la escena, como se puede ver en la figura 1 (b). La geometría en 3D puede ser proyectada a un mapa de textura de igual forma que se proyecta la geometría en 3D al monitor en 2D (de la Fuente, 2011). Simula también la técnica de trazado de rayos pero mapeando la geometría en mapas de textura. El algoritmo básico para superficies planas es propuesto por (Kilgard, 1999), pero para una superficie de agua es mejor utilizar la modificación hecha por (Jensen y Golias, 2001).

Para visualizar una escena con agua, los objetos a reflejarse tienen que ser dibujados primero de forma reflejada a lo largo del plano perteneciente a la superficie del agua. Entonces la escena se rasteriza y mapea a la textura de reflexión. Lo mismo ocurre para mapear la textura de refracción, pero esta vez no se reflejan los objetos a refractarse (el fondo del río, lago o mar) y solo se mapearían los que están por debajo del agua (de la Fuente, 2011).

Los dos enfoques antes descritos pueden ser muy útiles en dependencia del escenario. A partir de lo expuesto se decide utilizar la técnica Mapeo de Textura de Proyección teniendo en cuenta las siguientes características:

- Permite tanto reflexiones locales como globales en la escena.
- Logra un balance entre calidad de la imagen y eficiencia del proceso de visualización.

## 1.4 Técnicas de optimización

Debido a la gran cantidad de cálculos que se realizan y la gran cantidad de datos para representar las capas de agua en 3D es necesario aplicar algunas técnicas de optimización para mantener la fluidez con que se navega en el Visor de Terrenos. Esto contribuye en gran medida al realismo de la superficie del agua, puesto que si la visualización de la misma se realiza de manera lenta, entonces no parecerá real.

### *1.4.1 Técnicas de recorte o desecho*

Estas técnicas se utilizan para eliminar de la escena la geometría que no se necesita dibujar en pantalla porque no aparece en la imagen final. Con esto se logra un considerable incremento de velocidad, principalmente en escenas donde solo se visualiza una mínima parte de la modelación. Estas técnicas son usadas ampliamente.

Según (Vega, 2013) las técnicas de recorte más usadas son:

- Recorte de cara: Elimina los polígonos que pertenecen a la parte trasera o delantera de los objetos. Aunque esta técnica puede aplicarse también para eliminar los polígonos delanteros y traseros a la vez.
- Recorte de *Frustum*[3]: Selecciona los objetos que se encuentran dentro del *frustum*.
- Recorte de Oclusión: Elimina los objetos que no serán visualizados por encontrarse detrás de otro(s) objeto(s).
- Recorte de Portal: Deforma el *frustum* identificando los portales o grietas en la visualización y calcula el *frustum* que pasa a través de estos.

De las técnicas anteriores se utilizarán las técnicas *Recorte de cara* y *Recorte de Frustum*. La primera se utilizará con el objetivo de no visualizar la cara de atrás de las capas de agua, pues esas caras estarían entre la superficie y el suelo debajo del agua, logrando que menos cantidad de información se modele. La técnica de *Recorte de Frustum* será empleada durante el proceso de captura de las texturas de reflexión y refracción tratando de evitar visualizar la parte de la escena que está debajo del agua cuando lo que se necesita es lo que está por encima del agua y viceversa.

## 1.5 Tecnologías, Herramientas y Metodología a utilizar

Según el tipo de solución que se aborda en el presente trabajo, se definirán un conjunto de herramientas y tecnologías para el desarrollo de la solución. Debido a que se desarrollarán funcionalidades para el Visor de Terrenos será necesaria la utilización de la misma tecnología usada para la construcción de este último.

### *1.5.1 Tecnologías para la representación 3D*

[3] Frustum: Un volumen de visualización en forma de pirámide que crea una vista en perspectiva (Sellers, 2013).

Una de las tecnologías más usadas para llevar una modelación de objetos en 3D al ordenador y en particular a aplicaciones de escritorio es la API[4] OpenGL[5] que está escrita en el lenguaje de programación C. Para poder hacer uso de la misma desde otro lenguaje como es el caso de C# hay que hacer uso de una biblioteca avanzada que permita el acceso a sus funcionalidades.

OpenTK[6] (Kit de herramientas de Open) es una biblioteca avanzada de C# que envuelve a OpenGL. Es adecuada para juegos, aplicaciones científicas y de cualquier otro proyecto que requiere gráficos 3D, audio o funcionalidad de cálculo. En el Visor de Terrenos se utiliza como envoltura de OpenGL a la biblioteca OpenTK en su versión 1.1.

### *1.5.2 Metodología de desarrollo*

"Una metodología es una colección de procedimientos, técnicas, herramientas y documentos auxiliares que ayudan a los desarrolladores de software en sus esfuerzos por implementar nuevos sistemas de información." (Avison y Fitzgerald, 1995). En los últimos años el uso de las metodologías ha incidido positivamente en la calidad del software.

Las metodologías de desarrollo de software se dividen en dos vertientes:

- Metodologías Tradicionales o Pesadas.
- Metodologías Ágiles.

Las tradicionales llevan una documentación exhaustiva de todo el proyecto. Se utilizan en grandes equipos de desarrollo y necesitan generar una gran cantidad de documentación para mantener controlado el proceso de desarrollo y gestionar la comunicación entre los departamentos en los que puede dividirse el equipo. Las metodologías ágiles son flexibles ante requisitos cambiantes y son utilizadas en equipos de desarrollo pequeños.

Con las características mencionadas de ambas vertientes, las metodologías tradicionales son descartadas, por tanto para la solución se decide el uso de una metodología ágil. Entre las metodologías ágiles más

---

[4] Interfaz de Programación de Aplicaciones en inglés Application Programming Interface (API)

[5] https://www.opengl.org/

[6] www.opentk.com/

populares se encuentran XP y OpenUP, para la solución se selecciona OpenUP debido a la familiaridad que tiene el desarrollador con la misma además de haber sido usada para el desarrollo del Visor de Terrenos.

***Metodología de desarrollo OpenUP***

OpenUP[7] es un proceso unificado ligero que aplica enfoques iterativos e incrementales y define las fases, actividades y artefactos que se generan durante el ciclo de desarrollo del software. La finalidad de esta metodología de desarrollo es garantizar la eficacia mediante el cumplimiento de los requisitos iniciales y minimizar las pérdidas de tiempo en el proceso de generación del software.

Entre las ventajas de OpenUP se encuentran:

- Es extensible pues los procesos se pueden agregar o adaptar según lo vayan requiriendo los sistemas.
- Es ligero y proporciona una comprensión detallada del proyecto, beneficiando a clientes y desarrolladores sobre productos a entregar y su formalidad.
- Se centra en una arquitectura temprana para reducir al mínimo los riesgos y organizar el desarrollo.
- Maneja el ciclo de vida del desarrollo de software de manera apropiada al ofrecer una buena administración de las áreas del proyecto.

### *1.5.3 Lenguajes de programación*

Debido a que se desarrollarán funcionalidades al Visor de Terrenos, los lenguajes de programación serán dependientes de los que se usen en dicho software.

***C Sharp***

C Sharp (conocido comúnmente como C#) es un lenguaje de programación orientado a objetos que forma parte de la plataforma .NET de Microsoft. Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos similar al de Java, aunque incluye mejoras derivadas de otros lenguajes. A pesar de formar parte de la

[7] http://epf.eclipse.org/wikis/openup/

plataforma antes mencionada es un lenguaje de programación independiente diseñado para generar programas.

### *GLSL*

Lenguaje de Shading de OpenGL (GLSL por sus siglas en inglés) (Shreiner, 2013). Está basado en el lenguaje ANSI C y se creó para permitir a los programadores migrar funcionalidades de la aplicación hacia los *shaders* de OpenGL. Este lenguaje ha sido diseñado para permitir a los programadores de aplicaciones expresar el tratamiento que se produce en esos puntos programables de la Pipeline de OpenGL (Shreiner, 2013). Los *shaders* son unidades independientemente compilables que están escritas en GLSL.

Los dos lenguajes descritos anteriormente se usarán para la solución. El primero será el lenguaje principal y se usará en su versión 5.0, ya que el componente está escrito en C#. Con GLSL se escribirán los *shaders* para reemplazar las funcionalidades por defecto de la Pipeline de OpenGL, aunque en versiones recientes de este último algunas funcionalidades que antes eran por defecto en la tubería ya no existen y es obligación del programador proveerlas.

### *1.5.4 Shaders*

Los *shaders* son programas que permiten sustituir el procesamiento estándar o por defecto de los datos de una escena que realiza la unidad de procesamiento gráfico (GPU). Los *shaders* para cualquier API gráfica se escriben, usualmente, en un lenguaje de programación especializado. Para la familia de bibliotecas de visualización en 3D de OpenGL los *shaders* se escriben en el lenguaje GLSL. En el caso de OpenGL existe un proceso por el cual pasan los datos para visualizarse luego en pantalla. Este proceso es denominado *Pipeline de OpenGL* y son los pasos de este proceso los que se pueden sustituir con *shaders* proporcionados por el usuario del API. Para más claridad en cuanto a dicho *Pipeline* puede observar el Anexo A.

Existen *shaders* para sustituir el procesamiento de varios tipos de datos como vértices, píxeles, geometrías, entre otros. Los más usados son:

- *Vertex Shader*
- *Fragment Shader*

- *Geometry Shader*

La presente investigación estará centrada en la utilización de dos de ellos:

- *Vertex Shader*: Procesa datos de entrada, en este caso vértices, aplicando transformaciones, o cálculos para lograr efectos de luz, desplazamiento, valores de color, entre otros. Este *shader* se ejecuta una vez por cada vértice de manera simultánea (Sellers, 2013).
- *Fragment Shader*: Se ejecuta para cada píxel de manera simultánea, proporcionándole a los mismos el color final para ser representados en pantalla (Sellers, 2013).

### *1.5.5 Lenguaje de modelado*

Para la comunicación de ideas entre desarrolladores y para el análisis de algunos procesos es necesaria una forma estandarizada de representar un modelo o diseño. Para esto se necesita un lenguaje en el que se pueda modelar lo antes descrito.

Durante el desarrollo de la solución se utilizará como lenguaje de modelado el Lenguaje Unificado de Modelado (UML por sus siglas en inglés) en su versión 2.0, el cual permite especificar, visualizar y construir los artefactos que exigen la ingeniería del software.

### *1.5.6 Entorno de Desarrollo Integrado*

Un Entorno Integrado de Desarrollo (IDE por sus siglas en inglés) es un software que provee facilidades comprensivas a los programadores de computadora para el desarrollo de software. Para el desarrollo de la solución se seleccionó el IDE Visual Studio en su versión 12.0.21005.1 del 2013 debido a:

- Es desarrollado por la misma empresa que creó el lenguaje C# y la tecnología .Net.

- Es el IDE que se utiliza para el desarrollo del Visor de Terrenos.

### *1.5.7 Herramienta CASE*

Las herramientas para la Ingeniería de Software Asistida por Computadoras (CASE por sus siglas en inglés) son aplicaciones informáticas destinadas a aumentar la productividad en el desarrollo de software, ayudan a economizar el consumo de recursos como puede ser el tiempo. Estas herramientas pueden ayudar en

todos los aspectos del ciclo de vida de desarrollo del software, ya sea en el diseño del proyecto, cálculo de costos, implementación de parte del código automáticamente a partir del diseño, documentación, entre otras. La herramienta CASE seleccionada para realizar los artefactos pertinentes a las etapas de análisis y diseño de la solución es Visual Paradigm para UML en su versión 8.0 pues se caracteriza por:

- Uso de un lenguaje estándar común a todo el equipo de desarrollo que facilita la comunicación.
- Diseño centrado en casos de uso y enfocado al negocio.
- Capacidades de ingeniería directa e inversa.
- Modelo y código que permanece sincronizado en todo el ciclo de desarrollo.
- Diagramas de flujo de datos.
- Disponibilidad en múltiples plataformas (Microsoft Windows y GNU\Linux).

## Conclusiones del capítulo

El estudio de las propiedades ópticas del agua y los fenómenos que ocurren en la superficie de la misma permitió identificar las principales propiedades que permiten lograr un mayor realismo en la visualización del agua. Se analizaron las técnicas de visualización de superficies de agua reportadas en la literatura y se arribó a la conclusión de que la técnica Mapeo de Textura de Proyección, de la familia de técnicas Rasterización, es la adecuada para dar solución al problema planteado en esta investigación debido al balance que logra entre eficiencia y calidad de la visualización. Se propone además que se complete su uso con las técnicas de optimización Recorte de cara y Recorte de *frustum* las cuales descartan una cantidad considerable de puntos innecesarios en la visualización.

# CAPÍTULO 2: ANÁLISIS Y DISEÑO

A partir de la metodología OpenUP se generan algunos artefactos de acuerdo a las necesidades de la solución, los cuales son mostrados en este capítulo. Se muestra el modelo del dominio, un listado de requisitos funcionales y no funcionales detectados y los casos de uso que los agrupan. Se describe la arquitectura que presenta el software y los elementos que la componen.

## 2.1 Modelo del Dominio

El modelo del dominio es una visualización de los conceptos del dominio, es una representación de las clases conceptuales del mundo real (Larman, 2003). La figura 3 muestra el modelo del dominio para la visualización de una capa de agua, pues la visualización del agua es la funcionalidad que se le agregará al Visor de Terrenos.

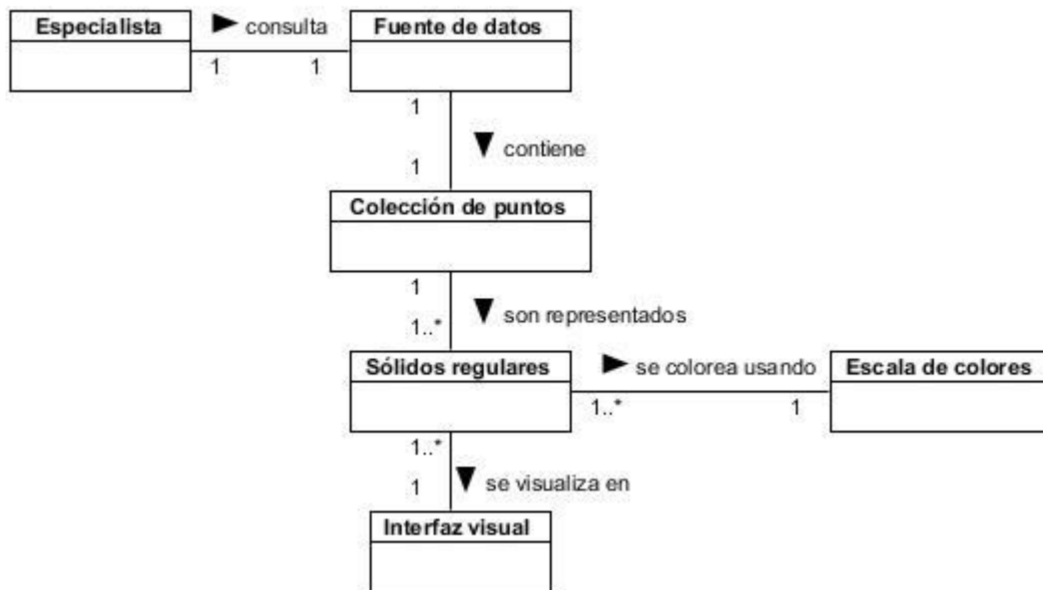


**Fig. 3** Modelo del Dominio.

Descripción de los objetos del Modelo del Dominio:

- **Especialista:** Representa al usuario que selecciona los datos a visualizar e interactúa con los mismos.

- **Fuente de Datos:** Origen de los datos que contiene la información referente al agua que se desea visualizar.
- **Colección de Puntos:** Representa la información contenida en la fuente de datos.
- **Sólidos Regulares:** Figuras geométricas formadas con la colección de puntos seleccionada. Estos sólidos regulares representan la geometría de lo que se desea visualizar, en este caso agua.
- **Escala de colores:** Es un conjunto de pares (altura, color) con los que cada sólido regular se colorea dependiendo de su altura.
- **Interfaz visual:** La interfaz visual es la región donde se visualizarán los componentes representados por el motor gráfico, es el punto de comunicación con el usuario.

## 2.2 Especificación de los requisitos del sistema

La ingeniería de requisitos es el amplio espectro de tareas y técnicas que conducen a la comprensión de los requisitos (Pressman, 2010). Con esta se facilita el entendimiento de lo que desea el cliente, arribando a una solución razonable y sin ambigüedades. Estos requerimientos permiten confirmar la viabilidad de la solución, gestionándolos de forma correcta se pueden transformar en un software funcional ya que determinan qué hará el mismo y definen las restricciones de su operación e implementación.

### *2.2.1 Requisitos funcionales*

Los requisitos funcionales establecen el comportamiento del sistema. Un requisito funcional típico contiene un nombre, un número de serie único y un resumen.

**RF1.** Visualizar capa de agua activa.

Representa en pantalla la capa de agua que está activa en el instante de visualización.

### *2.2.2 Requisitos no funcionales*

Las características requeridas del sistema, del proceso de desarrollo, del servicio prestado o de cualquier otro aspecto del desarrollo que señala una restricción del mismo así como las exigencias de cualidades que se le imponen al proyecto son requisitos no funcionales. Estos son los que aseguran que se desarrolle un software usable, fiable y eficiente.

**Eficiencia**

**RNF1.** La velocidad de visualización debe ser de 30 o más tramas por segundo.

**Interfaz de Software:**

Para que el Visor de Terrenos funcione correctamente se requiere que los siguientes requisitos de software se cumplan en el ordenador donde sea ejecutado.

**RNF2.** Existencia de la plataforma .Net en su versión 4.0 o superior.

**Interfaz de Hardware:**

Los requerimientos mínimos de hardware con los que debe cumplir un ordenador para un correcto funcionamiento. Los requisitos de Interfaz de Hardware fueron establecidos teniendo en cuenta los requisitos del Visor de Terrenos.

**RNF3.**

- Procesador: Dual Core 2.0 GHz.
- Memoria RAM: 2 GB.
- Tarjeta gráfica: que soporte OpenGL 2.0 o superior.

**Requerimientos de Restricción del Diseño y la Implementación**

**RNF4.** La implementación del Visor de Terrenos debe hacerse en el lenguaje C# usando OpenGL.

**RNF5.** La implementación del Visor de Terrenos tiene como restricción que no se debe hacer uso de bibliotecas o componentes de terceros.

## 2.3 Modelado del sistema

### *2.3.1 Actores del sistema*

Un actor es una persona o sistema que se comunica con el sistema o producto y que es externo a al mismo. Son quienes usan el producto dentro del contexto de la función y comportamiento que debe ser descrito (Pressman, 2010). En la tabla 1 se describe el actor del sistema.

**Tabla 1** Actor del sistema.

| Actor | Descripción |
| --- | --- |
| Usuario | Persona que interactúa con el Visor de Terrenos. |

### *2.3.2 Casos de Uso del Sistema*

Un caso de uso es lo que describe una función o característica del sistema desde el punto de vista del usuario. Sirve como una base para la creación de un modelo de requisitos más comprensivo (Pressman, 2010).

#### Diagrama de Casos de Uso del Sistema

El Diagrama de Casos de Uso refleja cómo los actores interactúan con los casos de uso (Figura 4).

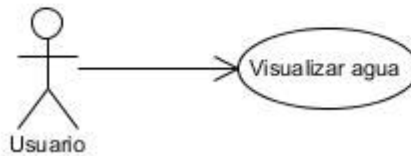


**Fig. 4** Diagrama de Casos de Uso.

#### Descripción de los casos de uso del sistema

Los casos de uso se elaboran adicionalmente para proporcionar considerablemente más detalles sobre la interacción. Los casos de uso se escriben a menudo informalmente (Pressman, 2010). Sin embargo, la descripción formal que se muestra en la Tabla 2 es para asegurar que se aborden todas las cuestiones clave.

**Tabla 2** Descripción del caso de uso Visualizar agua.

| | |
| --- | --- |
| **Objetivo** | Visualizar la capa de agua activa. |
| **Actores** | Usuario (inicia) |
| **Resumen** | El caso de uso se inicia cuando el usuario carga los datos de la capa de agua que desea visualizar. Finaliza cuando se visualiza en pantalla dicha capa de agua. |

| | | | |
|---|---|---|---|
| **Complejidad** | Alta. | | |
| **Prioridad** | Crítica. | | |
| **Referencias** | RF1. | | |
| **Precondiciones** | Deben existir los modelos de la capa de agua a visualizar. | | |
| **Poscondiciones** | Se visualiza la capa de agua en pantalla. | | |
| **Flujo de eventos** | | | |
| **Flujo básico <Representar agua>** | | | |
| | **Actor** | **Sistema** | |
| 1. Carga los datos de la capa de agua a visualizar. | | 2. Procesa los datos a visualizar.<br>3. Aplica técnicas de optimización y genera la geometría.<br>4. Visualiza la capa de agua y finaliza el caso de uso. | |
| **Flujos alternos** | | | |
| No existen flujos alternos. | | | |
| **Relaciones** | **CU Incluidos** | No incluye otros casos de uso. | |
| | **CU Extendidos** | No extiende otros casos de uso. | |
| **Requisitos no funcionales** | RNF1, RNF6, RNF7. | | |
| **Asuntos pendientes** | No se considera la realización de ningún asunto de este tipo. | | |

### *2.3.3 Arquitectura del sistema*

La arquitectura de software debe modelar la estructura de un sistema y la manera en que los datos y los componentes colaboran unos con otros (Pressman, 2010). La arquitectura del Visor de Terrenos fue definida a partir del patrón arquitectónico N-capas, definiendo para el software tres capas:

- **Acceso a Datos:** Esta capa es responsable de extraer los datos de la fuente de datos e interpretarlos para transmitirlos a la capa de Modelación en un formato entendible por la misma.
- **Modelación:** Los componentes que aquí se encuentran son los responsables de modelar la información para que pueda ser utilizada para la visualización. En esta capa están las estructuras de datos responsables de la triangulación del modelo y la gestión de los elementos que se visualizarán.
- **Visualización:** En esta capa se encuentran todos los componentes responsables de realizar la representación gráfica, dígase motor gráfico, gestor de luces, cámaras, *shaders*, entre otros. El desarrollo de la nueva funcionalidad al Visor de Terrenos tendrá lugar en esta capa.

### *2.3.4 Diagrama de clases*

El diagrama de clases describe gráficamente las especificaciones de las clases de software (Larman, 2003). En otras palabras, se evidencian las clases con sus relaciones. En el diagrama de clases correspondiente a la funcionalidad que se desarrollará (ver figura 5) se evidencia la pertenencia de las clases a la capa de Visualización.

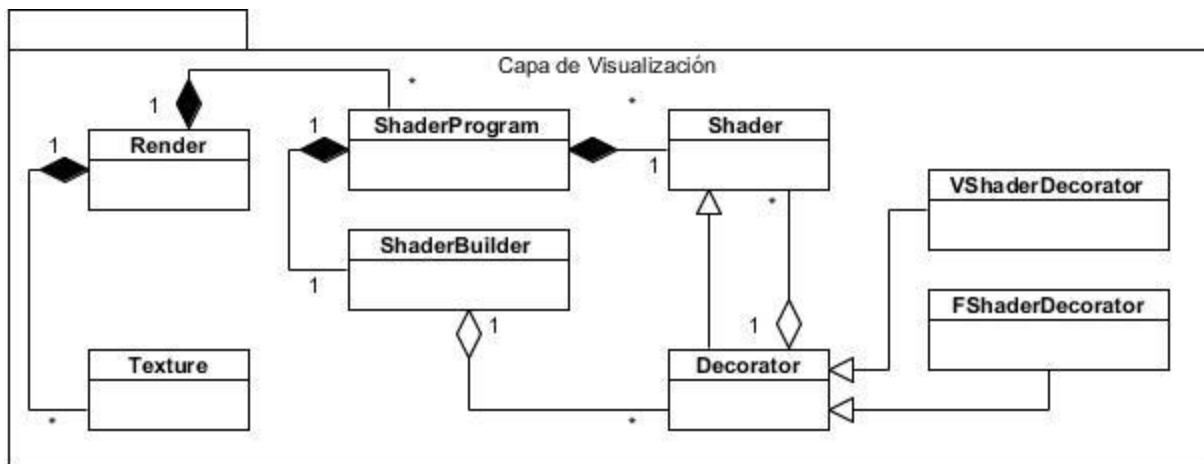


**Fig. 5** Diagrama de clases.

La clase `Texture` es la encargada de reservar memoria para las texturas, generar el identificador de las mismas, así como manejar propiedades referentes a las texturas haciendo uso de las funciones de la API gráfica seleccionada.

Las clases `ShaderBuilder`, `Shader`, `Decorator`, `VShaderDecorator` y `FShaderDecorator` son las clases que manejan toda la información referente a los *shaders*. Estas trabajan de conjunto conformando el patrón *Decorator* para crear un *shader* en tiempo de ejecución que explote las características del hardware gráfico.

La clase `ShaderProgram` es la clase que compila los *shaders* y los envía al hardware gráfico.

La clase `Render` es responsable de realizar la representación gráfica haciendo uso de las funciones de la API gráfica y de algunas clases antes descritas.

### *2.3.5 Patrones de diseño*

Los patrones de diseño de software son los que permiten describir fragmentos de diseño y reutilizar ideas de diseño, ayudando a beneficiarse de la experiencia de otros. "… Los patrones de diseño comunican los estilos y soluciones consideradas como buenas prácticas, que los expertos en el diseño orientado a objetos utilizan para la creación de sistemas." (Larman, 2003).

Entre los patrones de diseño más conocidos están los Patrones Generales de Asignación de Responsabilidades de Software, más conocidos por sus siglas en inglés como GRASP (Larman, 2003). Para lograr un diseño eficaz se utilizaron algunos que se describen a continuación.

**Patrón Experto:** Es asignar una responsabilidad al experto en información. El experto en información es la clase que posee la información necesaria para cumplir con dicha responsabilidad (Larman, 2003). Este patrón es usado en todas las clases ya que cada una posee la información para realizar la tarea que le corresponde. Un ejemplo se evidencia en la clase `Render`, que es responsable de capturar las texturas de reflexión y refracción. Para realizar la captura de las texturas mencionadas es necesaria la cámara, y las instancias de la clase Textura, informaciones que posee la clase `Render`.

**Patrón Creador:** El patrón Creador guía la asignación de responsabilidades relacionadas con la creación de objetos. El propósito fundamental de este patrón es encontrar un creador que debemos conectar con el objeto producido en cualquier evento. (Larman, 2003). En la solución se evidencia la utilización de este patrón en el método `Init` la clase `Render`, donde se crean las instancias de la clase `Textura`.

**Patrón Bajo Acoplamiento:** El acoplamiento es "… una medida de la fuerza con que un elemento está conectado a, tiene conocimiento de, confía en, otros elementos." (Larman, 2003). Un elemento con bajo acoplamiento sufre de pocos cambios o ninguno al realizarse modificaciones en las clases con las que se relaciona por lo que facilita su reutilización ya que no tiene dependencias fuertes con otros elementos. Con el objetivo de lograr un diseño flexible, con facilidad de soporte y que posea elementos reutilizables se utiliza este patrón.

**Patrón Alta Cohesión:** La cohesión es la medida de la fuerza con la que se relacionan las responsabilidades de un elemento. Un elemento con responsabilidades altamente relacionadas, y que no hace una gran cantidad de trabajo, tiene alta cohesión (Larman, 2003).

Según (Larman, 2003) una clase con baja cohesión al hacer muchas cosas no relacionadas presenta los siguientes problemas:

- Difícil de entender.
- Difícil de reutilizar.
- Difícil de mantener.
- Delicada y constantemente afectada por los cambios.

Para el diseño de las clases se tuvo en cuenta este patrón para lograr un software más robusto. Por ejemplo las responsabilidades de la clase `Render` están estrechamente relacionadas entre ellas, pues el objetivo de todas es la visualización de la escena.

Otros patrones de diseño muy utilizados son los de la Banda de los Cuatro (*GoF*[8] por sus siglas en inglés), de los cuales en la solución es utilizado uno de ellos llamado Patrón Decorador.

**Patrón Decorador:** "agrega responsabilidades adicionales a un objeto dinámicamente. Los decoradores proveen una alternativa flexible a la creación de subclases para extender la funcionalidad." (Gamma et al., 1995). La necesidad de agregar funcionalidades y características a un objeto de forma dinámica está presente en la solución cuando se debe construir el código de los *shaders* para que se ejecuten en la tarjeta

[8] En reconocimiento al importante trabajo que realizaron los cuatro autores del libro *Design Patterns: Elements of Reusable Object-Oriented Software*, se les llama afectivamente Gang of Four.

gráfica. En este caso se desea conformar el código del *Vertex Shader* y *Fragment Shader* con el código pertinente a las características con que visualizará el Visor de Terrenos. Una de las características que puede tener variación seguidamente es la luz y otra es la utilización o no del *Geometry Shader*. La figura 6 evidencia este patrón.

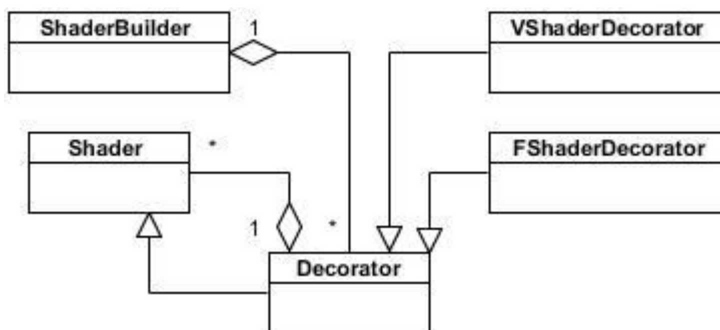


**Fig. 6** Diagrama del patrón decorador para los *shaders*.

## 2.4 Proceso de visualización propuesto

Luego de finalizadas las tres primeras fases para la visualización en el Visor de Terrenos:

1. Carga de los metadatos del modelo.
2. Obtención de la información que se encuentra dentro del campo de visión de la cámara.
3. Aplicación de técnicas de optimización y generación de la geometría.

Comienza la cuarta fase nombrada Representación de la Escena. Es donde se aplica color, iluminación, textura, perspectiva y otros parámetros a la imagen final. Las capas de agua se representan actualmente usando una escala de colores por lo que estas se visualizan con un color sólido al igual que se hace con un terreno (ver figura 7). En esta fase es donde se ejecutará la visualización del agua.

Luego de seleccionar las técnicas y realizar el diseño y análisis de la solución se propone un proceso para la visualización de las capas de agua utilizando Mapeo de Textura de Proyección. Dicho proceso está dividido en cuatro fases:

1. Captura de la textura de reflexión y refracción.
2. Creación de los *shaders*.
3. Recopilación y transmisión de los datos al GPU.

4. Visualización.

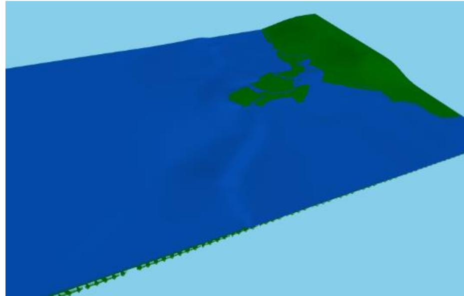

**Fig. 7** Imagen del Visor de Terrenos visualizando una escena que contiene agua.

La **primera fase** del proceso a su vez está dividida en dos pasos muy similares que son la captura de las texturas de reflexión y refracción. Para la captura de la textura de reflexión, antes de mostrarse en pantalla un fotograma, se deben dibujar todas las capas excepto las de agua, haciendo uso antes de la técnica *Recorte de cara*, recortando de esta forma la cara delantera para que, al reflejarse la imagen, desde el punto de visión de la cámara no se obstruya la vista reflejada que se observa por la parte trasera de la geometría. El mecanismo que se usará para dibujar los objetos de forma reflejada será el de multiplicar el componente z de cada vértice por -1 cumpliendo así con la ley de reflexión. Con esto se logra una ubicación exacta de la vista virtual de reflexión. La figura 8 muestra con más claridad lo planteado en la idea anterior, desde el punto de visión de la cámara se observa la vista reflejada (en verde) con respecto a la superficie del agua multiplicando la z por -1.

De forma análoga pero sin mover la cámara se obtiene la imagen con todos los objetos excepto el agua y se mapean en la textura de refracción.

En la **segunda fase** se crean, compilan y unen los *Vertex* y *Fragment Shaders* que realizarán sus funciones en la GPU.

En la **tercera fase** se recopila información que se utilizará en la visualización de las capas de agua, como pueden ser, entre otros:

- Posición de cada vértice.

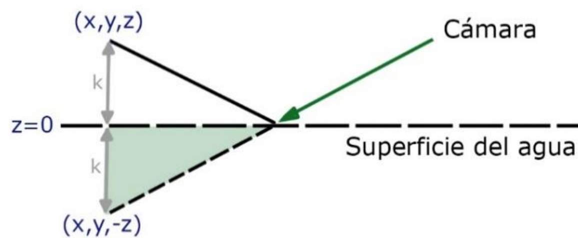


**Fig. 8** Vista reflejada de la imagen respecto al plano de la superficie del agua.

- Matriz de proyección.
- Matriz de vista y modelo.
- Textura de reflexión.
- Textura de refracción.

Luego, los datos recopilados se envían al GPU para utilizarse por los *shaders* en el proceso de visualización del agua.

La **cuarta fase** perteneciente a la visualización del agua se divide en dos etapas ejecutándose cada una en el *Vertex* y *Fragment Shader* respectivamente.

En el *Vertex Shader* se calculará el color inicial del agua antes de tener reflexión y refracción. Este color se calcula teniendo en cuenta la iluminación de la escena. Luego se calculan las coordenadas para acceder más tarde a la textura de reflexión y de refracción en el *Fragment Shader.* Es válido aclarar que como el *Vertex Shader* reemplazará esta funcionalidad en el *Pipeline* de OpenGL este debe realizar todas las operaciones necesarias de transformación de los vértices.

En el *Fragment Shader* las texturas de reflexión y refracción se usan para calcular el color de cada píxel del agua teniendo en cuenta varias leyes físicas como la de Snell. El color de un píxel será igual al color de reflexión interpolado linealmente con el resultado de interpolar linealmente también el color del agua y el color de refracción, teniendo en cuenta el coeficiente de reflexividad calculado con las ecuaciones (5) y (6). Con esta última fase queda representada en pantalla la capa de agua a visualizar.

## Conclusiones del capítulo

Durante el análisis y diseño de la solución se definieron los requisitos funcionales y no funcionales que apoyarán al desarrollo de un software útil y funcional. La utilización de los patrones GoF y GRASP permitió

dar respuestas eficientes a algunos problemas de diseño. También se propuso un proceso para la visualización de las superficies de agua, detallando los pasos para poder realizar su implementación.

Con una implementación que se ajuste al diseño y modelado de la solución se logrará un componente sencillo, robusto, de fácil comprensión, mantenimiento y reutilización, satisfaciendo los requisitos funcionales y no funcionales.

# CAPÍTULO 3: IMPLEMENTACIÓN Y PRUEBAS

En este capítulo se describe cómo se desarrolló la solución, en el mismo se muestra el diagrama de componentes correspondiente a las funcionalidades desarrolladas y las descripciones para su entendimiento. También se presentarán el estándar de codificación usado, la implementación de los algoritmos principales, las pruebas de software realizadas y por último la validación de la solución.

## 3.1 Estándares de codificación

Los estándares de codificación definen directrices para aplicar un estilo y formato coherentes (Hunt, 2007). Permite que el código sea más fácil de entender y ayuda a los desarrolladores a evitar errores comunes de codificación. Los estándares de codificación que se siguieron en la presente solución son los estándares de C# para .NET detallados en (Hunt, 2007). A continuación se describirán los aspectos más relevantes de dichos estándares.

### *3.1.1 Convenciones de nombre*

En este subepígrafe se describen algunas convenciones usadas para los nombres de variables, métodos, clases, atributos, entre otros (Tabla 3). De esta forma se logra que el código tenga una mayor consistencia, siendo la clave para obtener un código mantenible (Hunt, 2007).

**Tabla 3** Uso y sintaxis de nombre.

| Identificador | Convención |
|---|---|
| Archivo fuente | Se escribe en *Pascal Case*[9].<br>El nombre de la clase y el fichero son el mismo. |
| Clase o estructura | Se escribe en *Pascal Case*.<br>Se agrega un sufijo apropiado cuando hereda o es subclase de otra clase.<br>Ejemplo:<br>public class VShaderDecorator : Decorator {…} |
| Método | Se escribe en *Pascal Case*.<br>Se utiliza un verbo o un par verbo-sustantivo. |
| Propiedad | Se escribe en *Pascal Case*. |

[9] *Pascal Case*: Una palabra con la primera letra capitalizada y la primera letra de cada palabra o parte de palabra subsecuente capitalizada.

| | |
|---|---|
| | El nombre siempre representa la entidad a la que pertenece la propiedad, usando la misma palabra.<br>Ejemplo:<br>public string Name<br>{<br>get { return _name; }<br>set { _name = value; }<br>} |
| Atributo de clase | Si es *public*, *protected* o *internal* se escriben en *Pascal Case*.<br>Si es *private* se escribe en *Camel Case*[10] y con un prefijo "_".<br>Ejemplo:<br>protected int Id;<br>private Bitmap _bitmap; |
| Variable | Se escribe en *Camel Case*.<br>No se enumeran variables como text1, text2, etc; |
| Parámetro | Se escribe en *Camel Case*. |

### *3.1.2 Estilo de Código*

Es la manera de dar formato al código con el objetivo de crear una implementación más legible, clara, consistente y de fácil mantenimiento. Un mal estilo de código puede causar polémica entre desarrolladores. A continuación se mencionan algunas de las convenciones de estilo de código más relevantes en la solución.

- Un archivo fuente contiene una única clase.
- Las llaves de comienzo y fin ({ }) siempre se colocan en una nueva línea.
- Siempre se usan las llaves de comienzo y fin en sentencias condicionales.
- Se usa el margen adicional de 4 espacios.
- Las variables se declaran independientes en líneas nuevas, no en las mismas sentencias.
- Las declaraciones de uso (*using*) de espacios de nombre se colocan en la cabeza del archivo.
- La implementación de las clases tendrán el siguiente orden:

[10] Camel Case: Una palabra con la primera letra en minúscula y la primera letra de cada palabra o parte de palabra subsecuente capitalizada.

1. Atributos de la clase.
2. Constructores y Finalizadores.
3. Propiedades.
4. Métodos.

- Las declaraciones de atributos tendrán el siguiente orden según sus modificadores de acceso:
  1. Públicos.
  2. Protegidos.
  3. Internos.
  4. Privados.
- Los atributos relacionados se declaran en una misma línea. El resto se declara en líneas separadas.
- Los comentarios están declarados en el mismo idioma, gramaticalmente correctos y contienen los signos de puntuación apropiados.
- Para los comentarios se usa // o /// pero no /*…*/.

## 3.2 Implementación del proceso de visualización de una capa de agua

En este epígrafe se abordarán los detalles de implementación del proceso de visualización propuesto en el Capítulo 2. Los epígrafes del 3.2.1 al 3.2.4 corresponden respectivamente a las fases de la 1 a la 4 del proceso de visualización propuesto.

### *3.2.1 Captura de la textura de reflexión y refracción*

Para capturar la textura de reflexión primeramente se visualizan las capas que no son de agua y que además serán reflejadas. Inicialmente se activa la técnica de Recorte de Cara en el método `drawLayers` de la clase `Render`, recortando la cara frontal de los polígonos. Esto se logra activando en OpenGL dicha técnica e indicando qué cara será recortada como se muestra a continuación:

```
1  GL.Enable(EnableCap.CullFace);
2  GL.CullFace(CullFaceMode.Front);
```

Para lograr la vista reflejada se crea una matriz de escala que al multiplicarla con la posición transforma la z multiplicándola por -1. Esta matriz de escala se pasa a los *shaders* pertenecientes a las capas de terreno para que se aplique la transformación a sus vértices.

```
1  Matrix4 scaleMatrix = Matrix4.CreateScale(1, 1, -1);
```

En el código anterior se crea la matriz de escala usando la estructura `Matrix4` que ofrece OpenTK. Este es ejecutado en el método `drawLayers` de la clase `Render`.

Al dibujarse en pantalla las capas de forma reflejada, se guarda en la textura de reflexión la información de cada píxel para poder hacer uso de los mismos en la visualización del agua. Para ello se realiza una llamada al método `BindTexture`, de la clase `Render`, pasándole la textura en la que se quiere copiar la información visualizada  y se desactiva la técnica Recorte de Cara de una forma similar a como fue activada usando `GL.Disable(EnableCap.CullFace)`. A continuación la implementación del método `BindTexture`:

```
1  private void BindTexture(Texture tex)
2  {
3      GL.ActiveTexture(tex.TextureUnit);
4      GL.BindTexture(TextureTarget.Texture2D, tex.Id);
5      GL.CopyTexSubImage2D(TextureTarget.Texture2D, 0, 0, 0, 0, 0, Width, Height);
6  }
```

La textura que se pasa por parámetro es donde se guardará una copia de lo que se muestra en pantalla. Primeramente se activa el número de textura que se afectará al llamar posteriormente cualquier función que realice cambios sobre una textura. Luego en la línea 4 se enlaza la textura con el identificador `tex.Id` al tipo `TextureTarget.Texture2D`. Ahora esta textura en 2D pertenece a la unidad de textura asignada anteriormente en la línea 3. Con la línea 5 se copia la información en pantalla para la textura activa y enlazada anteriormente. Los parámetros pasados a la función `GL.CopyTexSubImage2D` son tipo de textura, nivel de detalle (0 es exactamente la propia imagen), corrimiento por el eje x, corrimiento por el eje y, los próximos dos valores son las coordenadas de la esquina inferior izquierda de la región rectangular de píxeles a ser copiados y los dos últimos valores son el ancho y alto del área que se va a copiar.

Para capturar la textura de refracción se debe borrar antes lo que se visualizó en pantalla haciendo una llamada a la función `GL.Clear` de OpenGL pasándole por parámetro lo enumeradores correspondientes a los búferes de color y profundidad. Posterior al borrado se visualizan nuevamente las capas que no son de agua y se realiza la llamada al método `BindTexture` pero pasándole esta vez la textura donde se almacenará lo que está debajo del agua. De esta forma quedan capturadas ambas texturas.

### *3.2.2 Creación de los shaders*

Para instanciar la clase *ShaderProgram* se hace uso del método `FactoryShader` de la clase `Render`, pasándole como parámetro a la capa que se visualizará. Este método implementa el patrón Fábrica (Gamma et al., 1995) que forma parte de las funcionalidades implementadas anteriormente en el componente. En dependencia del tipo de capa se crea un programa *shader* correspondiente a dicha capa. Dicho patrón es necesario pues las capas pueden ser de tipo vectoriales[11] o ráster[12]. El agua se representa usando el tipo de capa *Ráster*.

```
1   currentProgram = FactoryShader(currentLayer);
```

A su vez el código de los *shaders* se crea a partir de la implementación del patrón Decorador. La clase `ShaderBuilder` es la encargada de crear el código final de los *shaders* mediante el método `build` para luego crear el objeto `ShaderProgram`. A continuación se muestra el código de dicho método.

```
1   public Shader build()
2   {
        Shader code = new Shader();
3       code.VERTEX_SHADER   = buildVertexShaderCode();
        code.FRAGMENT_SHADER = buildFragmentShaderCode();
4       code.NameAttributes = this._vertexShader.NameAttributes;
5       code.NameUnifform = this._vertexShader.NameUnifform.Concat(
                            this._fragmentShader.NameUnifform).ToArray();
6       return code;
7   }
```

Los patrones Fábrica y Decorador utilizados para la creación de los *shaders* están detallados en (Gamma et al., 1995).

### ***3.2.3 Transmisión de los datos al GPU***

Los datos necesarios para la visualización se pasarán a los *shaders* utilizando las funcionalidades que ofrece OpenGL. A continuación se describe la forma de enviar una matriz de cuatro filas y cuatro columnas (la matriz de escala que se usó para visualizar los objetos de forma reflejada). Esto se realiza desde el método `drawLayers` de la clase `Render`.

```
1   int location = GL.GetUniformLocation(currentProgram.SProgramHandler, "tMatrix");
```

[11] Vectorial: una imagen digital formada por objetos geométricos independientes definidos por atributos matemáticos de color, posición, entre otros.

[12] Ráster: una estructura de datos en forma de matriz que representa una grilla rectangular de píxeles formando una imagen.

```
2   GL.UniformMatrix4(location, false, ref scaleMatrix);
```

En la primera línea se obtiene la ubicación de la variable `tMatrix` en el *shader* correspondiente al identificador `SProgramHandler`. La cadena de texto indicando el nombre de la variable tiene que coincidir con el nombre con que se declaró en el *shader*. Por último se le pasa a la variable `tMatrix` que se encuentra en la ubicación `location`, una referencia de la matriz `scaleMatrix` que contiene la matriz de escala.

Para enviar la textura de reflexión primeramente se debe activar su número y enlazar su identificador con el número de textura activado para entonces poder pasar los datos de la textura al *shader*. Estos dos primeros pasos se explicaron en la fase de captura de las texturas. Los dos últimos son los mismos que para enviar la matriz de escala con la diferencia de que cambia `GL.UniformMatrix4` por `GL.Uniform1` ya que el método que se invoca depende de las dimensiones de los datos que se van a enviar.

```
1   GL.ActiveTexture(reflection.TextureUnit);
2   GL.BindTexture(TextureTarget.Texture2D, reflection.Id);
3   int location = GL.GetUniformLocation(currentProgram.SProgramHandler,"reflection");
4   GL.Uniform1(location, reflection.TextureUnit - TextureUnit.Texture0);
```

### *3.2.4 Visualización*

**Visualización en OpenGL**

Esta fase recibe puntos, líneas, polígonos y otros datos para convertirlos en píxeles mostrados en el monitor haciendo uso del *Pipeline* de OpenGL. El propósito de este *Pipeline* es convertir objetos en 3D en una imagen en 2D. Para cumplir con esta transformación OpenGL define varios espacios de coordenadas y transformaciones entre esos espacios. Cada espacio de coordenada tiene alguna propiedad que lo hace útil en alguna parte del proceso de visualización. Los atributos de objetos en 3D, tales como vértices y normales a la superficie, son definidas en el Espacio Objeto. Este último es conveniente para describir el objeto que está siendo modelado, por ejemplo el origen de su sistema de coordenadas puede ser diferente para cada objeto. Con el objetivo de componer una escena que contenga una variedad de objetos en 3D, los cuales están definidos en su propio Espacio Objeto se necesita un sistema de coordenadas común. Este sistema de coordenadas común se llama Espacio Mundo. El origen de este sistema de coordenadas es también arbitrario. Luego de ser definido este último espacio, todos los objetos de la escena deben ser transformados

de sus coordenadas de objeto a las coordenadas del mundo utilizando la matriz de transformación del modelo.

Una vez la escena ha sido definida, es necesario especificar los parámetros de visualización. La posición, el punto de visión y el vector de dirección hacia arriba de la cámara deben definirse. Estos parámetros de visualización se combinan en la matriz de visualización. Cuando se multiplica por esta matriz una coordenada, esta se transforma del Espacio Mundo al Espacio Ojo. Por definición, el origen de este sistema de coordenadas es la posición de la cámara.

La API gráfica OpenGL combina las matrices modelo y visualización en una simple matriz llamada modelo vista. Esta transforma coordenadas de Espacio Objeto a Espacio Ojo. El próximo paso es definir un volumen de visualización. Es la región de la escena que va a ser visible en la imagen, este volumen se describe en la matriz de proyección. La matriz de proyección toma los objetos en el volumen de visualización y los transforma al Espacio de Corte, un espacio de coordenadas que es adecuado para recorte. El próximo paso en la transformación de la posición de los vértices es la división de perspectiva. Esta operación divide cada componente de la coordenada en Espacio de Corte por la coordenada homogénea w. Los resultantes *x*, *y* y *z* van a estar en el rango [-1,1]. Este espacio es llamado Espacio de Coordenada Normalizada de Dispositivo. En este espacio las coordenadas que no se encuentren en el rango antes mencionado significa que están fuera del volumen de visualización y quedan descartadas, de ahí el nombre de Espacio de Corte.

Finalmente para obtener la posición de los píxeles en la pantalla se realiza una conversión a Espacio Pantalla que se produce durante el proceso de rasterización. Estas coordenadas van desde cero hasta el ancho o alto del área donde se visualizará en el monitor.

**Visualización del agua**

Como se trató en el capítulo 1, el coeficiente de reflexividad de Fresnel es esencial para lograr una reflexión realista en el agua. Este valor se calcula usando la ecuación 5, que es una aproximación del mismo, implementada en el *Vertex Shader* con el siguiente método.

```
1  void fresnel(in vec3 incom, in vec3 normal, in float index, out float reflectance)
2  {
3      float rf = pow((index - 1.0)/(index + 1.0),2.0);
4      float cosTheta = dot(normal, incom);
```

```
5        reflectance = rf + (1.0 - rf) * pow((1.0 - abs(cosTheta)),5.0);
6    }
```

Donde `incom` es el vector que va desde el vértice hacia la posición de la cámara, `normal` es el vector normal a la superficie perteneciente al vértice, `index` es el índice de refracción del medio al que entra la luz (en este caso es el del agua). En la línea 3 se calcula el coeficiente de reflexividad para el ángulo de incidencia igual a 0º como se plantea en la ecuación 6. En la línea 4 se calcula el coseno del ángulo formado por los vectores `normal` e `incom`. El coseno del ángulo entre dos vectores es igual al producto escalar entre ellos. Por último en la línea 5 se calcula el coeficiente de reflexividad.

A continuación el método `main` del *Vertex Shader*:

```
1    void main(void)
2    {
3        vec4 rColor = vec4(0.023, 0.384, 0.8711, 1);
4        vec4 diffuseMaterial = rColor;
5        vec3 vertex=in_position;
6
7        normal = normalize(in_normal);
8
9        if(lightUse==1)
10       {
11           vec4  uMaterialDiffuse = vec4(0.0,0.0,0.0,1.0);
12           vec4  uMaterialAmbient = vec4(0.0,0.0,0.0,1.0);
13           float uShininess       = 0;
14           vec3  dir = uLightPD;
15           vec4  Ia = uMaterialAmbient * uAmbientLight;
16           vec3  L       = normalize(dir);
17           float lambert = max(dot(normal,-L),0.001);
18           vec4  Id      = diffuseMaterial * uDiffuseLight * lambert;
19           rColor = Ia + Id;
20           rColor.a = diffuseMaterial.a;
21       }
22       rColor.a = alphaValue-(1.0-rColor.a);
23       if(rColor.a < 0.0)
24       {
```

```
25          rColor.a = 0.0;
26      }
27
28      if(in_position.z-noData <= 0.000001)
29      {
30          vertex.z= 0.0;
31          rColor.a= 0.0;
32      }
33
34      vColor= rColor;
35
36      vec4 vert = vec4(vertex, 1);
37      if(vert.z > 0.0)
38      {
39          vert.z = 0.0;
40      }
41
42      mat4 mvp = projection_matrix * modelview_matrix;
43      reflCoord = mvp * vert;
44      refrCoord = mvp * vec4(vertex, 1);
45
46      view = normalize(camPosition - vertex);
47      fresnel(view, normal, 1.33333, fresnelR);
48
49      gl_Position = refrCoord;
    }
```

Las líneas de la 3 a la 34 corresponden a los cálculos del color inicial del agua teniendo en cuenta la intensidad de la luz que incide sobre la misma. Esta porción de código es parte de las funcionalidades que tenía implementado el componente y su explicación no es objetivo de este trabajo.

De la línea 36 a la 40 se hace una copia del vértice que se está procesando y asigna el valor de cero al componente z si dicho vértice tiene un valor de z por encima de cero. Las coordenadas de los vértices que se procesan están en Espacio Objeto. La variable `vert` se utiliza para calcular las coordenadas para acceder a la textura de reflexión. Es importante este paso pues da como resultado que el reflejo se distorsione con respecto a la altura de las olas.

En la línea 42 se calcula la multiplicación de las matrices proyección y modelo vista para almacenarla y no volver a calcularla en las restantes sentencias. Las líneas 43 y 44 calculan las coordenadas en Espacio de Corte para acceder a las texturas de reflexión y refracción en el *Fragment Shader*.

Luego se calcula en la línea 46 el vector que va desde el vértice hasta la posición de la cámara. Este vector es necesario para pasarlo como parámetro al método invocado en la próxima sentencia para calcular el coeficiente de reflexividad.

Por último se devuelve como posición del vértice que se procesa la misma posición que se calculó para acceder a la textura de refracción, pues se calculan de igual forma.

En el *Fragment Shader* se calcula el color de cada píxel perteneciente al agua:

```
1   void main(void)
2   {
3       vec4 projCoord = reflCoord / reflCoord.q;
4       projCoord = (projCoord + 1.0) * 0.5;
5       projCoord = clamp(projCoord, 0.001, 0.999);
6
7       vec4 reflectionColor = texture(reflection, projCoord.st);
8
9       projCoord = refrCoord / refrCoord.q;
10      projCoord = (projCoord + 1.0) * 0.5;
11      projCoord = clamp(projCoord, 0.001, 0.999);
12
13      vec4 refractionColor = texture(refraction, projCoord.st);
14
15      vec4 final = mix(vColor, refractionColor, 0.5);
16      final = mix(final, reflectionColor, fresnelR);
17      out_frag_color = final;
18  }
```

En las líneas 3 y 9 se transforman las coordenadas de reflexión y refracción al Espacio de Coordenada Normalizada de Dispositivo. Como en ese espacio las coordenadas pertenecen al intervalo [-1, 1] y para acceder a las texturas deben estar en el intervalo [0, 1], se ejecutan las sentencias de las líneas 4 y 5 para

la reflexión y las líneas 10 y 11 para la refracción. Luego se extrae el color de dichas texturas usando el método `texture` en las líneas 7 y 13.

En las líneas 15 y 16 se mezclan los colores del agua, refracción y reflexión utilizando el método `mix` que los interpola linealmente atendiendo a un factor que en el caso de la reflexión es el coeficiente de reflexividad calculado en el *Vertex Shader*. Finalmente, en la última línea, se asigna al color del píxel que se procesa el color final, siendo esta la salida del *Fragment Shader*.

## 3.3 Diagrama de Componentes

Según (Pressman, 2010) un componente es una parte modular, desplegable y reemplazable de un sistema que encapsula implementación y expone un conjunto de interfaces. Un diagrama de componentes modela cómo un sistema de software se divide en componentes y representa las dependencias entre ellos (Figura 9).

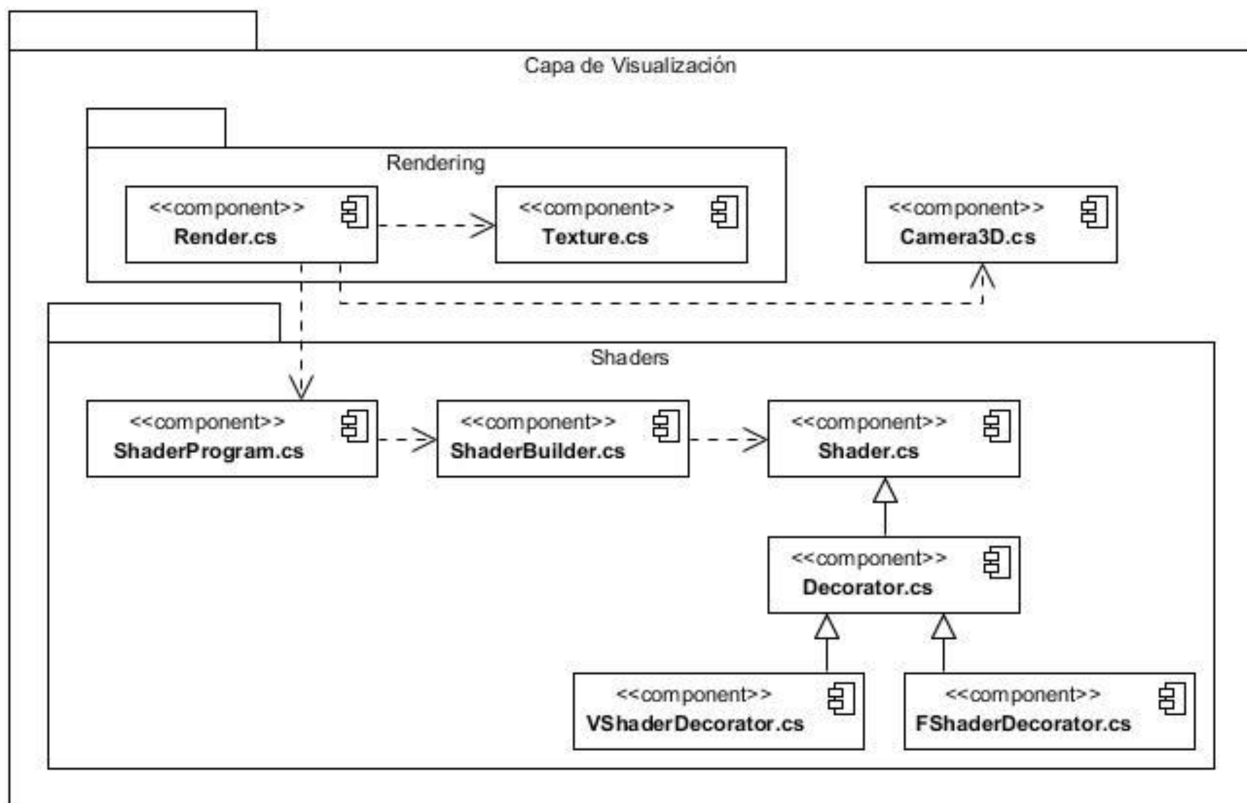


**Fig. 9** Diagrama de Componentes.

En el paquete *Rendering* se encuentran los siguientes componentes:

- `Textura.cs`: contiene los datos asociados a una textura como son el identificador, dimensiones y el color por cada píxel.
- `Render.cs`: Este es el componente principal para la representación gráfica. Contiene toda la información y algoritmos necesarios para la visualización.

El paquete *Shader* contiene los componentes que describen los programas *shaders* que se utilizan en la solución:

- `ShaderProgram.cs`: contiene los programas de *shaders* (*vertex* y *fragment*) necesarios para el procesamiento en el GPU.
- `ShaderBuilder.cs`: este componente es el encargado de construir el código GLSL del *Vertex Shader* y *Fragment Shader* haciendo uso de los decoradores.
- `Shader.cs`: define una estructura que deben implementar el resto de los componentes para ejecutar su estructura GLSL.
- `Decorator.cs`: define una estructura para que los componentes con sufijo *Decorator* puedan enlazar su código GLSL entre ellos y construir los programas de *shaders*.
- `VShaderDecorator` y `FShaderDecorator`: Estos componentes hacen uso de los componentes antes mencionados para conformar el código GLSL necesario de los programas *shaders* en cada caso.

El paquete *Camera* contiene el componente *Camera3D* que es quien maneja todo lo referente a la visualización, punto de visión, ángulo, perspectiva, entre otros. Este componente es externo a la solución pues no se implementó durante la misma.

## 3.4 Pruebas de software

Las pruebas del software son un elemento crítico para la garantía de la calidad del software y representa una revisión final de las especificaciones, del diseño y de la codificación. El objetivo de las mismas es encontrar el máximo número posible de errores con una cantidad manejable de esfuerzo aplicado en un período realista de tiempo (Pressman, 2010). Con las mismas se garantiza que el producto final funcione como fue diseñado e implemente de manera correcta los requerimientos identificados.

### *3.4.1 Estrategia de pruebas*

Según (Pressman, 2010) la estrategia clásica de pruebas de software comienza probando las partes pequeñas del software y se enfocan luego en las partes más grandes, probando finalmente el sistema como un todo. En este caso se seguirá la estrategia clásica cumpliendo con los siguientes niveles de pruebas:

- Pruebas Unitarias.
- Pruebas de Integración.
- Pruebas de Sistema.

### *3.4.2 Pruebas Unitarias*

La prueba de unidad centra el proceso de verificación en la menor unidad del diseño del software. En el caso de un contexto orientado a objetos el concepto de unidad cambia. La menor unidad que se prueba es la clase (Pressman, 2010). La prueba de las clases es dirigida por las operaciones que esta encapsula y el estado de comportamiento de las mismas.

Para estas pruebas se utilizó el método de Caja Blanca y la técnica de Camino Básico. El método de Caja Blanca es una filosofía de diseño de casos de prueba que utiliza la estructura de control descrita para derivar casos de prueba que:

- Garanticen que todos los caminos independientes dentro de un método han sido ejercitados al menos una vez.
- Ejerciten todas las decisiones lógicas en sus lados verdaderos y falsos.
- Ejecuten todos los ciclos en sus límites y dentro de sus límites operacionales.
- Ejerciten las estructuras de datos internas para verificar su validez.

La técnica de Camino Básico permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental y utilizar esta medida como una guía para la definición de un conjunto base de rutas de ejecución. De esta forma se garantiza que los casos de pruebas derivados ejecuten cada sentencia al menos una vez durante la prueba.

A continuación se describe la prueba realizada al código del *Fragment Shader* analizado en el epígrafe de Implementación:

```
1   out vec4 out_frag_color;
2   in vec4 vColor;
3   in vec4 reflCoord;
4   in vec4 refrCoord;
5   in float fresnelR;
6   uniform sampler2D reflection;
7   uniform sampler2D refraction;
8   void main(void)
9   {
10      vec4 projCoord = reflCoord / reflCoord.q;
11      projCoord = (projCoord + 1.0) * 0.5;
12      projCoord = clamp(projCoord, 0.001, 0.999);
13      vec4 reflectionColor = texture(reflection, projCoord.st);
14      projCoord = refrCoord / refrCoord.q;
15      projCoord = (projCoord + 1.0) * 0.5;
16      projCoord = clamp(projCoord, 0.001, 0.999);
17      vec4 refractionColor = texture(refraction, projCoord.st);
18      vec4 final = mix(vColor, refractionColor, 0.4);
19      final = mix(final, reflectionColor, fresnelR);
20      out_frag_color = final;
21  }
```

Este código, al no contener sentencias condicionales posee una complejidad cilcomática[13] de valor 1. Al poseer ese valor de complejidad, existe un único camino independiente por lo que se debe generar un único caso de prueba para cada iteración.

La entrada para este método son las variables `vColor`, `reflCoord`, `refrCoord`, `fresnelR`, `reflection` y `refraction`. La salida es la variable `out_frag_color`. Las entradas y salida deseada (última fila) para el caso de prueba 1 se muestran en la tabla 4.

**Tabla 4** Caso de Prueba #1 para el *Fragment Shader*.

| Variable | Valor |
|---|---|
| vColor | (0.023, 0.384, 0.8711, 1) |

[13] La complejidad ciclomática es una métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa (Pressman, 2010).

| | |
|---|---|
| reflCoord | (0, 0, 0, 1) |
| refrCoord | (0.02, 0.02, 0, 1) |
| fresnelR | 0.5 |
| reflection | Una imagen con el color (1, 0, 0, 1) y dimensiones 800 por 600 píxeles. |
| refraction | Una imagen con el color (0, 1, 0, 1) y dimensiones 800 por 600 píxeles. |
| out_frag_color | (0.50575, 0.346, 0.217775, 1) |

Al ejecutarse el *Fragment Shader* con los juegos de datos de la tabla 4 se tuvo como resultado un valor para out_frag_color diferente al esperado. Se registró la no conformidad y se corrigió el error. En el código presentado anteriormente en la línea 18 el factor de interpolación lineal se cambió por 0.5.

Se realizó una segunda iteración con el caso de prueba de la tabla 5.

**Tabla 5** Caso de Prueba #2 para el *Fragment Shader*.

| Variable | Valor |
|---|---|
| vColor | (0, 0, 0, 1) |
| reflCoord | (0.5, 0.5, 0, 1) |
| refrCoord | (0.48, 0.48, 0, 1) |
| fresnelR | 0.5 |
| reflection | Una imagen con el color (1, 0, 0, 1) y dimensiones 800 por 600 píxeles. |
| refraction | Una imagen con el color (0, 1, 0, 1) y dimensiones 800 por 600 píxeles. |
| out_frag_color | (0.5, 0.25, 0, 1) |

Luego de ejecutarse la segunda iteración no se detectaron no conformidades y finaliza la prueba unitaria.

En las restantes unidades de la solución se realizaron dos iteraciones. Durante la realización de la primera iteración se detectaron tres no conformidades. Al corregirlas se realizó la segunda iteración sin detección de errores. En total se detectaron cuatro no conformidades en las Pruebas Unitarias (Figura 10).

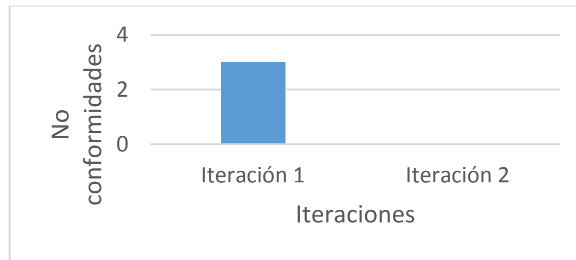


**Fig. 10** No conformidades detectadas durante las pruebas unitarias.

### *3.4.3 Pruebas de Integración*

Las Pruebas de Integración se centran en comprobar que los módulos probados por separado funcionen en conjunto, con el objetivo de verificar que interactúan correctamente a través de sus interfaces y cubren las funcionalidades establecidas en los requisitos. Como el software orientado a objetos no tiene una estructura de control jerárquico, las estrategias de integración *top-down* y *bottom-up* tienen poco significado. Integrar operaciones una a la vez en una clase (este es el enfoque tradicional de integración) es comúnmente imposible por las relaciones directas e indirectas de los componentes que conforman una clase (Pressman, 2010). Para la realización de esta prueba se utilizó la estrategia de Prueba Basada en Uso propuesto en (Binder, 1994) y recomendada por (Pressman, 2010). Esta estrategia comienza la construcción del sistema por probar aquellas clases (independientes) que usan muy pocas o ninguna clase. Después que esas clases son probadas, se prueba la próxima capa llamadas clases dependientes que usan a las independientes. Esta secuencia de capas de pruebas de clases dependientes continúa hasta que el sistema completo es construido.

A continuación se describen las capas de clases que se seleccionaron durante el proceso.

1. *Texture*, *VShaderDecorator*, *FShaderDecorator*.
2. *Decorator*.
3. *Shader*, *ShaderBuilder*.
4. *ShaderProgram*.
5. *Render*.

Cada vez que fue seleccionada una capa de clases se probaron las relaciones entre ellas y las que se habían probado con anterioridad. Verificando de esta forma que las llamadas y mensajes pasados entre clases fueran los correctos. Para la primera iteración de la prueba de Integración se utilizó como juego de

datos Monai con todas las capas de agua. El juego de datos Monai se describe al final de este epígrafe. A continuación todos los mensajes de comunicación entre la clase `Render` y la clase `ShaderProgram`.

```
En Render: instanciando el objeto de tipo ShaderProgeam.
En ShaderProgram: Enlazando locations.
En Render: llamando el método UseProgram de ShaderPorgram.
En Render: llamando el método BindLocations de ShaderPorgram.
En ShaderProgram: Enlazando locations.
En Render: llamando el método SetAlphaValue de ShaderPorgram. Pasando por parámetro 1.
En ShaderProgram: Estableciendo alphaValue a 1.
En Render: actualizando valores uniformes del shader.
En Render: llamando el método EnableFlag de ShaderProgram. Pasando como parámetro el string
lightUse.
En ShaderProgram: Habilitando lightUse.
En Render: llamando el método EnableFlag de ShaderProgram. Pasando como parámetro el string
scaleUse.
En ShaderProgram: Habilitando scaleUse.
En Render: llamando el método SetAmbienteLigth de ShaderPorgram. Pasando por parámetro
Terrain3DViewerControl.AmbientLight.
En Render: llamando el método SetDirectionalLight de ShaderPorgram. Pasando por parámetro
Terrain3DViewerControl.DirectionalLight.
En Render: llamando el método SetScaleColor de ShaderPorgram. Pasando por parámetro
Terrain3DViewerControl.SceneManager.ColorDecorator.
En Render: llamando el método SetNoDataValue de ShaderPorgram. Pasando por parámetro -9999.
En Render: llamando el método SetWidthMap de ShaderPorgram. Pasando por parámetro 67068,5.
En Render: llamando el método SetHeightMap de ShaderPorgram. Pasando por parámetro 76360,5.
En Render: accedientdo a la property get SprogramHandler de ShaderProgram.
En ShaderProgram: Devolviendo ShaderProgramHandler con valor 3.
En Render: accedientdo a la property get SprogramHandler de ShaderProgram.
En ShaderProgram: Devolviendo ShaderProgramHandler con valor 3.
```

En esta iteración se encontró una no conformidad en la integración de las clases `OpenGLTerrain3DViewerWFormControl` y `Render`. Existía una referencia de la primera clase en la segunda. Fue ejecutada una segunda iteración usando como juego de datos Monai con solo 10 capas simples de agua que conforman la animación de la misma. En esta última iteración no se encontraron no conformidades. En total se detectó una no conformidad que fue corregida.

Monai es un juego de datos correspondiente a un segmento playa en Japón, utilizado para simular el oleaje en la orilla del mar. Está conformado por 60 capas de agua y una capa base de terreno. Fue brindado por el centro CIMNE para el desarrollo de la visualización del agua.

### *3.4.4 Pruebas del Sistema*

La Prueba del Sistema está constituida por una serie de pruebas diferentes cuyo propósito primordial es ejercitar profundamente el sistema basado en computadora. Aunque cada prueba tiene un propósito diferente trabajan para verificar que se han integrado adecuadamente todos los elementos del sistema y que realizan las funciones apropiadas. Las pruebas que se realizadas con el fin de probar el sistema son las pruebas de Rendimiento, pues para sistemas en tiempo real es inaceptable el software que proporciona las funciones requeridas pero no se ajusta a los requisitos de Rendimiento. Esta prueba está diseñada para probar el cumplimiento de dichos requisitos en tiempo de ejecución dentro del contexto de un sistema integrado.

Las pruebas de rendimiento se llevaron a cabo en un ordenador con las siguientes prestaciones:

- Procesador: DualCore Inter Core i3, 3100 MHz.
- Memoria RAM: 2GB DDR3.
- Tarjeta Gráfic: Intel HD Graphics.

En la tabla siguiente se muestran los juegos de datos utilizados para la prueba de rendimiento, las características de los mismos y los resultados obtenidos en cuanto a cantidad de tramas visualizadas por segundo en el componente.

**Tabla 6** Datos y resultados de las pruebas de rendimiento aplicadas al software.

| Juegos de datos | Dimensiones en vértices | Tramas por segundo |
|---|---|---|
| Monai (terreno y agua) | 244 x 393 | 48 |
| Monai (terreno y agua) | 2431 x 3921 | 33 |

Los resultados obtenidos muestran un desempeño por encima de 30 tramas por segundo tanto en un juego de datos pequeño como en uno grande. Siendo 30 el mínimo necesario (Microsoft, 2003) para una visualización en tiempo real, cumpliendo así con el requisito no funcional de eficiencia.

## 3.5 Validación del componente

Para validar la aplicación de las propiedades ópticas de reflexión y refracción así como la existencia de una buena calidad visual se hizo uso de la técnica Juicio de Expertos.

La evaluación mediante el Juicio de Expertos consiste en solicitar a una serie de personas la demanda de un juicio o su opinión hacia un aspecto concreto. La cantidad de expertos seleccionada para la evaluación

fue de cuatro personas, teniendo en cuenta la sencillez de la evaluación y la inexistencia de un acuerdo para la determinación de la cantidad necesaria de expertos para dicha técnica (Powell, 2003). Esta técnica se utilizó mediante la Agregación Individual, que consiste en obtener la información de manera individual de cada uno de ellos, sin la exigencia de que se pongan en contacto (Cabrero y Llorente, 2013).

El procedimiento utilizado para la selección de los expertos es el Biograma. Este procedimiento consiste en elaborar una biografía del experto, incorporándose en la misma varios aspectos, en la cual se recoja el mayor número de aportaciones que permita justificar al evaluador o investigador los motivos que le han llevado a seleccionar una persona como experto en su estudio (Cabrero y Llorente, 2013).

En la etapa de selección de los expertos fueron seleccionados cuatro expertos de la UCI, tres de ellos pertenecientes al Grupo de Matemática y Física Computacionales y el cuarto perteneciente al centro de desarrollo *Vertex* (en los anexos C, D, E y F se encuentran las síntesis biográficas que validan su experticia).

Los expertos son:

- MSc. Yailen Costa Marrero
- Lic. Edisel Navas Conyedo
- Dr. Jorge Gulín González
- Ing. Alexis Echemendia González

Mediante una entrevista individual se les presentó un modelo con una explicación breve sobre los objetivos del trabajo y los resultados que se deseaban obtener. Se les presentó los aspectos a valorar a través de una tabla Aspectos / Rangos de Valoración y un video con las imágenes a evaluar. Los aspectos a evaluar son Refracción de la luz, Reflexión de la luz, Calidad total de la reflexión y refracción en conjunto. Los rangos de valoración son Muy Malo, Malo, Regular, Bueno, Muy Bueno. Ver tabla 7.

**Tabla 7** Valoración de los expertos sobre la calidad de la reflexión y refracción de la luz en el agua.

| Experto | Refracción de la luz | Reflexión de la luz | Calidad total de la reflexión y refracción en conjunto |
|---|---|---|---|
| MSc. Yailen Costa Marrero | Muy Bien | Muy Bien | Muy Bien |
| Lic. Edisel Navas Conyedo | Bien | Muy Bien | Muy Bien |
| Dr. Jorge Gulín González | Muy Bien | Muy Bien | Muy Bien |
| Ing. Alexis Echemendia González | Bien | Muy Bien | Bien |

Con las valoraciones obtenidas por parte de los expertos se puede concluir que la reflexión de la luz aplicada a la superficie de agua tiene muy buena calidad, la refracción de la luz posee una buena calidad y de manera general la visualización en conjunto de ambos fenómenos se realiza muy bien.

Por otra parte las pruebas realizadas al código y al sistema validan el cumplimiento de los requisitos del componente. Con lo antes mostrado se puede afirmar que con la aplicación de las propiedades de reflexión y refracción de la luz a la superficie de las capas agua se incrementó el realismo que presentaban las mismas.

## Conclusiones del Capítulo

Con la aplicación de los estándares de C# para .NET se logró una implementación legible y de fácil mantenimiento. El diagrama de componentes sirvió de apoyo para representar cómo la solución está dividida en componentes y cómo se relacionan entre ellos. Durante las pruebas realizadas se detectó un total de tres no conformidades permitiendo corregirlas y dejando un software con mayor calidad.

Finalmente se realizó una validación del componente basándose en los resultados de las pruebas de software y la técnica Juicio de Expertos, demostrando la validez de la hipótesis de investigación planteada en la introducción de este trabajo. La valoración general de los expertos apoya significativamente al desarrollo futuro del componente, pues realizan varias sugerencias que pueden mejorar el realismo de la visualización.

# CONCLUSIONES

Se desarrollaron las funcionalidades de visualización del agua al Visor de Terrenos del CEMC teniendo en cuenta las propiedades de reflexión y refracción de la luz mejorando el realismo de las escenas. Con la utilización de la técnica Mapeo de Textura de Proyección se incrementó el realismo de forma eficiente en la visualización de las capas de agua. Las tecnologías y herramientas seleccionadas para el desarrollo de las nuevas funcionalidades del componente ofrecieron el soporte necesario para dar cumplimiento a los requisitos funcionales y no funcionales del mismo. La aplicación de patrones en el diseño de la solución propició resolver problemas de forma eficiente. El uso de la técnica de optimización Recorte de Cara permitió que se llevara a cabo la visualización de forma reflejada. La utilización de los *shaders* en la implementación hizo posible que se desarrollara gran parte de los cálculos necesarios en el GPU liberando de esta forma el CPU. La aplicación de las leyes físicas a los gráficos por computadora aumenta el realismo de los mismos. Las pruebas de software realizadas en conjunto con el Juicio de Expertos llevado a cabo demostraron la validez de la solución.

## RECOMENDACIONES

Para futuros trabajos se tuvo en cuenta las sugerencias realizadas por los expertos por lo que se recomienda:

- Incorporar la visualización de las imágenes que se proyectan en el suelo debajo del agua como producto de la refracción de la luz.
- Adicionar el efecto de luz especular al agua.

# Referencias Bibliográficas

AVISON, D y FITZGERALD, G. Information Systems Development: Methodologies, Techniques, and Tools. New York, McGraw-Hill, 1995. 505 p.

BINDER, R. *Testing Object-Oriented Systems: A Status Report*. American Programmer, 1994, 7(4): p. 23-28.

BLINN, J. F.; NEWELL, M. E. Texture and reflection in computer generated images. Communications of the ACM 1976, 19 (10): 542-547.

BRUNETON, Eric; NEYRET, Fabrice; HOLZSCHUCH, Nicolas. Real-time Realistic Ocean Lighting using Seamless Transitions from Geometry to BRDF. En: Computer Graphics Forum, 2010, Vol. 29, No. 2, p. 487-496.

CABERO, Almenara J.; LLORENTE, Cejudo M. *The expert´s judgment application as a technic evaluate information and communication technology.* Revista de TIC en Educación. 2013. 7 (2): p. 11-22.

DE LA FUENTE, Antonio C. *Fluids real-time rendering.* Tesis de maestría, Universitat Politécnica de Catalunya, Barcelona, 2011.

FELICÍSIMO, Angel M. *Modelos digitales del terreno. Introducción y aplicaciones en las ciencias ambientales.* [en línea]. 1994. [Consultado el: 1 de enero de 2015]. Disponible en: [http://www.etsimo.uniovi.es/~feli/pdf/libromdt.pdf].

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. New York, Addison-Wesley, 1995. 315 p.

GRINDSTAD, Thorvald C.; RASMUSSEN, Runar J. F. *Deep water ocean surface modelling with ship simulation.* Tesis de maestría, NTNU Marine Technology, Oslo, 2011.

HENDERSON, Tom. *Light Absorption, Reflection, and Transmission.* [en línea]. Physics Classroom. 2014. [Consultado el: 5 de octubre de 2014]. Disponible en: [http://www.physicsclassroom.com/class/light/Lesson-2/Light-Absorption,-Reflection,-and-Transmission].

HENDERSON, Tom. *Blue Skies and Red Sunsets.* [en línea]. Physics Classroom. 2014. [Consultado el: 3 de noviembre de 2014]. Disponible en: [http://www.physicsclassroom.com/class/light/Lesson-2/Blue-Skies-and-Red-Sunsets].

HUNT, Lance. C# Coding Standards for .NET. [en línea]. 2007. [Consultado el: 1 de abril de 2015]. Disponible en: [http://se.inf.ethz.ch/old/teaching/ss2007/251-0290-00/project/CSharpCodingStandards.pdf].

JENSEN, Lasse S.; GOLIAS, Robert. Deep-water animation and rendering. En: Game Developer's Conference. Proceedings of the Game Developer's Conference Europe. Disponible en: [http://www.gamasutra.com/gdce/2001/jensen/jensen_01.htm]. 2001.

KHRONOS Group. *OpenGL 4.5 Reference Pages.* [en línea] OpenGL Software Development Kit, 2014. [Consultado el: 10 de ocutbre de 2014]. Disponible en: [https://www.opengl.org/sdk/docs/man].

KILGARD, Mark J. *Improving shadows and reflections via the stencil buffer.* En: Game Developers Conference. Advanced OpenGL Game Development. https://developer.nvidia.com/sites/default/files/akamai/gamedev/docs/stencil.pdf. 1999.

LARMAN, Craig. *UML y Patrones. Una introducción al análisis y diseño orientado a objetos y al proceso unificado.* México, Prentice Hall, 2003. 520 p.

MICROSOFT. *Understanding Frames Per Second (FPS)*. [en línea] Microsoft Support, 2003 [Consultado el: 3 de noviembre de 2014]. Disponible en: [https://support.microsoft.com/en-us/kb/269068].

POWELL, C. *The Delphi technique: myths and realities.* Journal of Advanced Nursing, 2003, 41 (4): p. 376-382.

PRESSMAN, Roger S. *Software Engineering. A Practitioner's Approach.* Boston, Mc Graw Hill, 2010. 870 p.

RAE. *Diccionario de la Real Academia Española.* [en línea]. Cáustico (ca), 2012. [Consultado el: 1 de mayo de 2015]. Disponible en: [http://lema.rae.es/drae/?val=c%C3%A1ustico].

RESNICK, Robert; HALLIDAY, David; KRANE, Kenneth S. *Physics.* Wiley, 1992. 480 p.

RODGERS, Alexander P. *CUDA Ray Tracer.* Tesis de B.Sc. (Hons), Universidad de Brighton, Brighton, *2014.*

SCHLICK, Christophe. *An Inexpensive BRDF Model for Physically-based Rendering.* Proceedings of Eurographics, 1994, 13 (3): p. 233-246.

SELLERS, Graham; WRIGHT, Richard S.; HAEMEL, Nicholas. *OpenGL Superbible: Comprehensive Tutorial and Reference*. Pearson Education, 2013. 1002 p.

SHALLOWAY, Alan; TROTT, James R. *Design Patterns Explained. A New Perspective on Object Oriented Design.* Addison-Wesley, 2004. 480 p.

SHREINER, Dave; WOO, M; NEIDER, J; DAVIS, T. *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley Professional, 2013. 986 p.

VEGA, Gendrys E. *Algoritmo de recortes y de niveles de detalles para el incremento de la velocidad de visualización de modelos 3D en dispositivos de bajo coste.* 3C TIC, 2013, vol. 2, no 4.

# Anexo A: *Pipeline* de OpenGL

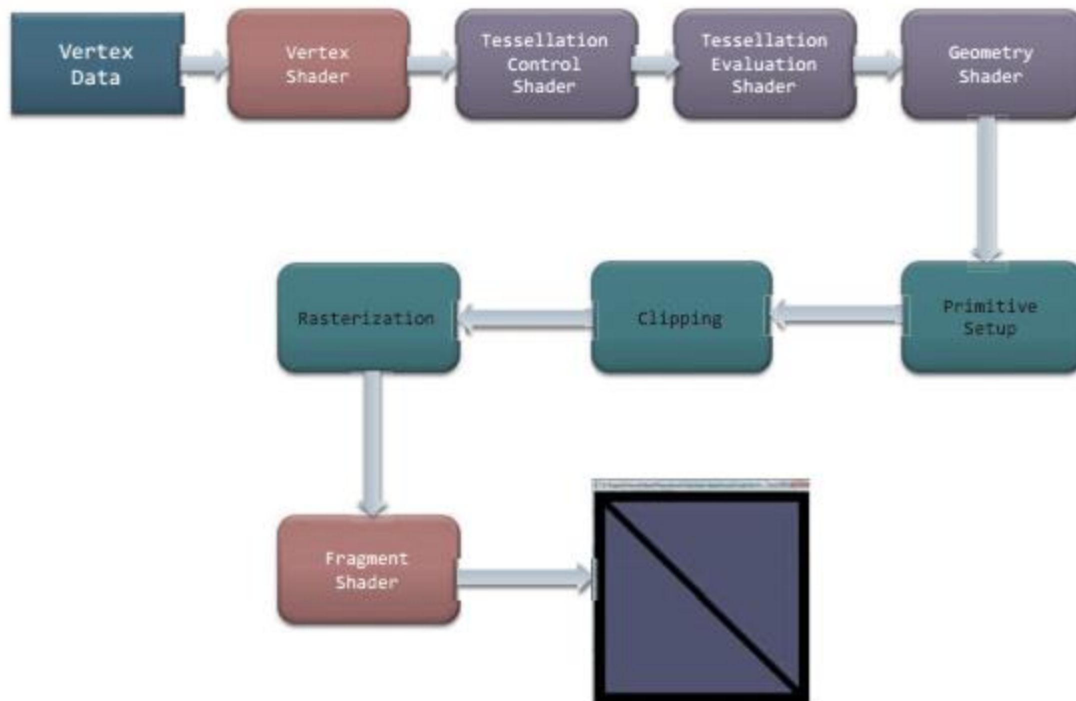


**Fig. 11** Pipeline de OpenGL. Fuente: (Shreiner, 2013)

# Anexo B: Vistas del componente

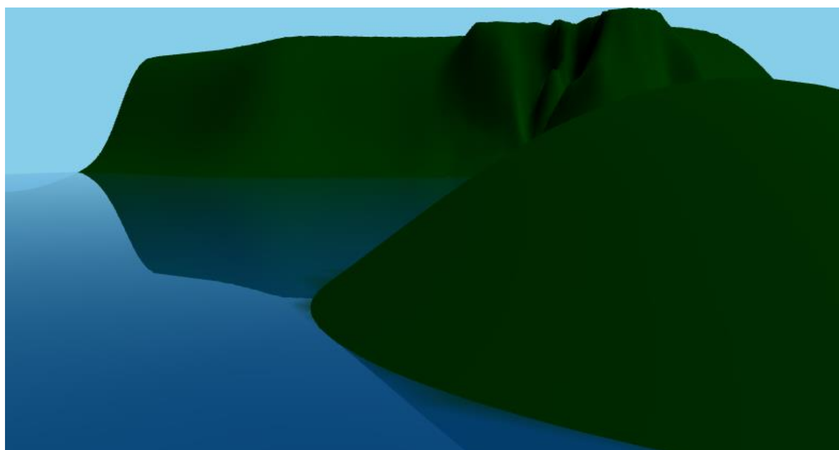

**Fig. 12** Vista del juego de datos Monai.

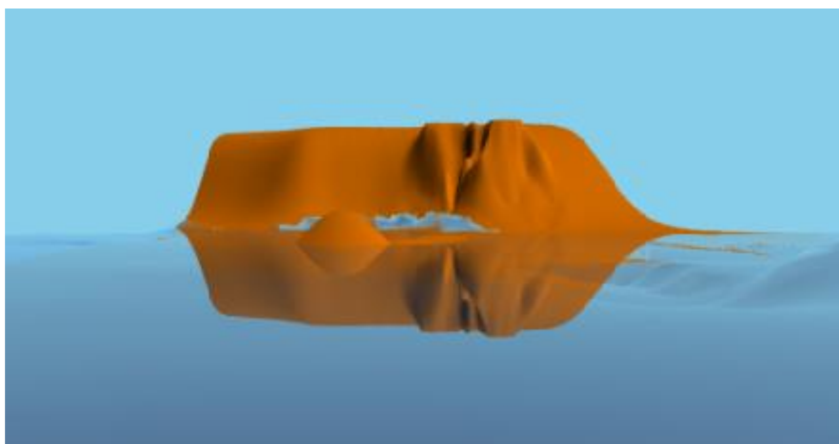

**Fig. 13** Vista del juego de datos Monai con oleaje.

## Anexo C: Síntesis biográfica de MSc. Yailen Costa Marrero

Yailen Costa Marrero se graduó en el año 2003 en la Universidad de La Habana de la carrera Licenciatura en Física. En el año 2014 terminó la maestría en la especialidad Ciencia y Tecnología de Materiales-Física. Del año 2004 al 2005 impartió clases como Profesor Adiestrado de Física General I y II en las facultades de Ingeniería Mecánica y Ciencias Empresariales en la Universidad Central de las Villas. Como Profesor Instructor impartió las asignaturas Física I y II en la Facultad 4 de la Universidad de las Ciencias Informáticas desde 2005 hasta 2007. Desde el 2007 hasta el 2012 impartió clases de Física I y II como Profesor Asistente en la Universidad de las Ciencias Informáticas. Actualmente es Asesora del Departamento Docente Metodológico Central de Ciencias Básicas de la UCI e imparte clases de las asignaturas Física y Fundamentos Físicos. Tiene diversos cursos de postgrado en la especialidad de Física. Ha participado en eventos tales como el XIII Simposio y XI Congreso de la Sociedad Cubana de Física, el VI Taller de Física y Matemática Computacionales de la VI Conferencia Científica de la Universidad de las Ciencias Informáticas UCIENCIA, la XIV Escuela Internacional de Ciencia y Tecnología de Materiales, el XXVII Congreso Latinoamericano de Química y VI Congreso Internacional de Química e Ingeniería Química, el III Simposio de Fotobiología, Fotofísica y Fotoquímica, FOTOCIENCIAS 2005, V Escuela Regional de Cristalografía y Difracción, entre otros. Actualmente cuenta con 6 publicaciones y ha formado parte de los proyectos “Proyecto de Innovación Pedagógica, Física” y “Caracterización Computacional de Sistemas Difusivos”.

## Anexo D: Síntesis biográfica de Lic. Edisel Navas Conyedo

Edisel Navas Conyedo se graduó en el año 2003 con título de oro en la Universidad de La Habana de la carrera Licenciatura en Física. Desde el año 2012 es estudiante de doctorado en la especialidad de Física. Del año 2003 al 2004 se desempeñó como Profesor Adiestrado en el Instituto de Materiales y Reactivos (IMRE) de la Universidad de La Habana. Desde 2004 hasta 2005 como Profesor Adiestrado impartió clases de Mecánica Teórica I y II y Física General I y II en la Facultad de Matemática Física y Computación y la Facultad de Ingeniería Mecánica respectivamente. En la Universidad de las Ciencias Informáticas desde el año 2005 hasta el 2007 impartió las asignaturas Física General I y II como Profesor Instructor. Desde el 2007 hasta el 2013 se desempeñó como Profesor Asistente en la UCI impartiendo clases de Física y Matemática IV. Desde 2013 hasta la actualidad es investigador y Profesor Auxiliar perteneciente al Centro de Estudios de Matemática Computacional y Coordinador del grupo de Matemática y Física Computacionales de la UCI. En el 2012 fue invitado como joven investigador al Instituto de Sistemas Complejos y Simulaciones Avanzadas de Jülich, Alemania. Ha cursado numerosos estudios de postgrado y ha participado en eventos como el XIII Symposio y XI Congreso de la Sociedad Cubana de Física, el Congreso internacional COMPUMAT2013, XVIII Escuela de Verano de Ciencias y Tecnologías de los Materiales, el IV Simposio de Foto-Biología-Física-Química Fotociencias 2008, el XI Simposio y IX Congreso de la Sociedad Cubana de Física, V Taller de Tecnologías Láser, TecnoLáser 2007, XIV Escuela de Verano de Ciencias y Tecnologías de los Materiales, el IV Taller de Tecnologías Láser, TecnoLáser 2005, III Simposio de Fotobiología, Fotofísica y Fotoquímica, FOTOCIENCIAS 2005, a la Escuela y Conferencia sobre Física Estadística y Aplicaciones Interdisciplinarias, V Escuela Regional de Cristalografía y Difracción, entre otros. Actualmente cuenta con 10 publicaciones y es jefe del proyecto Caracterización Computacional de Sistemas Difusivos.

# Anexo E: Síntesis biográfica de Ing. Alexis Echemendía González

Alexis Echemendía González se graduó en el año 2007 de Ingeniero en Ciencias Informáticas en la Universidad de las Ciencias Informáticas. Del 2007 al 2014 como Profesor en Adiestramiento imparte la asignatura Tele-Informática I y II, el curso optativo de Diseño 3D y el curso de Introducción al modelado con Salome-Meca. En el 2003 participó en el desarrollo de la primera versión del paseo virtual de la UCI celebrado en la Cumbre de Ginebra. En el 2007 participó en el modelado del entorno para Simulador de Camión presentado en la Feria de Informática 2008. Del 2007 al 2008 fue Jefe de modelado de autos del juego Rápido y Curioso expuesto en varias Ferias de Informática y en el Parque de Diversiones Lenin. Desde el 2009 hasta 2010 se desempeñó como jefe de la línea Escenarios 3D en la UCI. En el 2011 y hasta el 2012 fue diseñador principal en el proyecto CDSEM, trabajando directamente en la Fábrica de Taladros ICVT perteneciente a la República Bolivariana de Venezuela. En 2012 fue diseñador del laboratorio virtual sobre Vibraciones mecánicas. Desde el 2013 hasta el 2014 fue Jefe, diseñador y programador del Show-Run-Virtual de los productos de la UCI. En 2015 participó en el evento World Game Jam presentando el video-juego Sweet Carrot. En el 2010 recibió el curso de postgrado Animación 3D en Chandigarh, India. Actualmente se desempeña en la línea de desarrollo Modelado y animación 3D.

## Anexo F: Síntesis biográfica de Dr. Jorge Gulín González

Jorge Gulín González se graduó en el año 1995 en la Universidad de La Habana de Licenciatura en Ciencias Físicas. Del 1996 al 2000 realizó su doctorado en Ciencias Físicas. Desde el año 1996 hasta el 1997 se desempeñó como investigador en adiestramiento en el departamento de Física de la Cuidad Universitaria José A. Echeverría (CUJAE). En el 2002 impartió clases en la CUJAE como Profesor Asistente. Desde el 2008 hasta la actualidad es profesor titular en la Universidad de las Ciencias Informáticas (UCI). Del 2008 hasta el 2013 fue Director de Investigaciones en la UCI. Sus líneas de investigación son Física computacional, Física-Química, materia condensada, Bioinformática, materiales nanoporosos, catalizadores sólidos y difusión anómala. En el 2002 recibió el Premio Nacional del CITMA en la modalidad de Investigador Joven. En el 2004 se le otorga la membresía de la fundación A.v.Humboldt's de *University of Leipzig* (Alemania). En los años 2003 y 2009 fue premio de la Academia de Ciencias de Cuba. Desde el 2008 hasta el 2013 se desempeñó como asistente y editor de la Revista Cubana de Ciencias Informáticas. Es actualmente miembro del Consejo Científico de la CUJAE y la UCI, miembro de la Sociedad Cubana de Física y de la Sociedad Cubana de Matemática y Computación. Es miembro del Claustro de la Universidad de Sassari en Italia, también del claustro para doctorados en Bioinformática de la Universidad de La Habana. Fue presidente del comité científico de los congresos: COMPUMAT 2013, UCIENCIA 2012 y UCIENCIA 2013. Actualmente cuenta con más de 40 publicaciones.