UNIVERSIDAD CENTRAL DE VENEZUELA
FACULTAD DE CIENCIAS
ESCUELA DE COMPUTACIÓN

Despliegue de Volúmenes utilizando tecnología XNA

Trabajo Especial de Grado
presentado ante la Ilustre
Universidad Central de Venezuela
por el Bachiller

Br. Edgar Alejandro Bernal Oropeza

para optar al título de Licenciado en Computación

Tutor
Prof. Ernesto Coto

Caracas, Abril 2013

Universidad Central de Venezuela

Facultad de Ciencias

Escuela de Computación

ACTA DE VEREDICTO

Quienes suscriben, miembros del jurado designado por el Consejo de la Escuela de Computación para examinar el Trabajo Especial de Grado, presentado por el Bachiller Edgar Alejandro Bernal Oropeza, C.I. 17.119.223, con el título "Despliegue de Volúmenes utilizando tecnología XNA", a los fines de cumplir con el requisito legal para optar al título de Licenciado en Computación, dejan constancia de lo siguiente:

Leído el trabajo por cada uno de los miembros del jurado, se fijó el día 10 de Abril de 2013, a las 02:30 p.m., para que su autor lo defendiera en forma pública, en el Centro de Computación Gráfica, lo cual éste realizó mediante una exposición oral de su contenido, y luego respondió satisfactoriamente a las preguntas que le fueron formuladas por el jurado, todo ello conforme a lo dispuesto en la Ley de Universidades y demás normativas vigentes de la Universidad Central de Venezuela. Finalizada la defensa pública del Trabajo Especial de Grado, el jurado decidió APROBARLO.

En fe de lo cual se levanta la presente acta, en Caracas el 10 de Abril de 2013, dejándose también constancia de que actuó como coordinador del jurado el Profesor, Ernesto Coto, Tutor del trabajo.

_____

Prof. Ernesto Coto

(Tutor)

_____ _____

Prof. Héctor Navarro

(Jurado Principal)

Prof. Carlos Acosta

(Jurado Principal)

Resumen

Título:

Despliegue de Volúmenes utilizando tecnología XNA.

Autor:

Edgar Alejandro Bernal Oropeza.

Tutor:

Prof. Ernesto Coto.

Una de las ramas de la visualización que ha tenido el mayor crecimiento en los últimos años es la visualización de volúmenes, que consiste en el despliegue de conjuntos de datos tridimensionales en la pantalla de tal manera que el científico pueda entenderlos e interpretarlos satisfactoriamente. Una de las técnicas más utilizadas para la visualización de volúmenes es el Despliegue Directo de Volúmenes, que consiste en generar una imagen semitransparente de alta calidad a partir del volumen, lo cual requiere gran poder de procesamiento para un despliegue en tiempo real.

Con el pasar de los años los computadores tienen mejores capacidades de hardware y de cómputo de alto desempeño. Es habitual ver en el mercado tarjetas gráficas que cuentan con capacidades aceleradores avanzadas y una elevada capacidad de almacenamiento. Las consolas de videojuegos también han avanzado enormemente, convirtiéndose en dispositivos de hardware muy poderosos, a veces con capacidades que van más allá de los computadores estándares. Aunque típicamente, estas consolas de videojuegos han sido utilizadas para el entretenimiento, existen APIs que hacen posible desarrollar aplicaciones de propósito general que permitan aprovechar las capacidades de cómputo de dichas consolas. Este es el caso del Framework Microsoft XNA, que consiste en un conjunto de herramientas de desarrollo que permite la creación de videojuegos y aplicaciones para las distintas plataformas de Microsoft.

En este trabajo se describe el desarrollo de una aplicación que permite realizar despliegue de volúmenes en una consola de videojuegos Microsoft Xbox 360, utilizando la tecnología XNA.

Palabras claves: Volume Rendering, Despliegue Directo de Volúmenes, XNA, Xbox 360.

Agradecimientos

Agradezco a Dios por estar presente durante toda mi carrera, dándome ánimos para seguir y culminar todas mis metas.

Agradezco a mi familia por educarme cada día y por todo el apoyo incondicional que me han brindado. A mi padre que desde pequeño me enseñó a usar una computadora y enseñarme sus conocimientos. A mi madre por despertarme cada mañana y madrugar conmigo para que pudiera ir a la universidad. A mis hermanos, primos, primas, tíos, tías, abuelas, que en algún momento me han aportado con su granito de arena.

Agradezco a mi estimado tutor por guiarme en cada paso de este trabajo y darme sus mejores consejos para el éxito.

Agradezco a mi jurado por ser parte de este trabajo. Asimismo quiero darles las gracias a todos los integrantes del Centro de Computación Gráfica por darme su apoyo y motivación. A todos los profesores, que han sido herramientas para el aprendizaje y han dedicado su vida a la enseñanza de las ciencias.

Agradezco a Adriana por ser mi complemento, mi guía y mi mejor amiga. Gracias por brindarme el cariño y amor desde que nos conocimos, por estar pendiente de mí desde el inicio de la carrera y por no dejarme caer en los momentos difíciles. Sin ti no hubiese tenido ni la mitad de la felicidad que tengo, y estoy seguro que cada día seré más feliz contigo.

Agradezco a mis mejores amigos (Adriana, Kijam, Pedro, Álex, Emmanuel, Karina, Katiuska y demás personas) quienes figuran recuerdos inolvidables en mi vida.

Agradezco a la numerosa cantidad de amigos por ayudarme y darme sus diferentes puntos de vista de la carrera. A mis alumnos de Algoritmos y Estructuras de datos por dejar enseñarles un poco del conocimiento adquirido en el transcurso de mi carrera. Al equipo de Valor Absoluto de Futsal por dejarme jugar y ser el capitán durante muchos años.

Agradezco a todos, mil gracias, se les quiere.

Tabla de Contenidos

Tabla de figuras

Introducción

Actualmente, los científicos utilizan las herramientas computacionales para visualizar los datos con los que trabajan, para así poder realizar el análisis correspondiente. Una de las ramas de la visualización que ha tenido el mayor crecimiento en los últimos años es la visualización de volúmenes, que consiste en el despliegue de uno o varios conjuntos de datos tridimensionales en la pantalla de forma que el científico pueda entenderlos e interpretarlos satisfactoriamente. La posibilidad de poder desplegar volúmenes en computadores personales es una de las razones principales de este crecimiento.

El despliegue de volúmenes se suele realizar de dos maneras: Despliegue Directo de Volúmenes y Extracción de Isosuperficies. La primera técnica genera una imagen semitransparente de alta calidad, lo cual requiere gran poder de procesamiento para un despliegue en tiempo real. La segunda técnica consiste en transformar los datos escalares del volumen en superficies conformadas por primitivas geométricas que luego son visualizadas utilizando las técnicas comunes de despliegue 3D.

Una consola de videojuegos no es más que un sistema computarizado especializado para el entretenimiento interactivo, capaz de procesar uno o más videojuegos y generar una señal de video que pueda proyectarse, bien sea en una pantalla del mismo dispositivo, en un televisor o en un monitor externo a la consola en sí misma.

Típicamente las consolas de videojuegos han sido utilizadas para el entretenimiento. Sin embargo, es posible desarrollar aplicaciones de propósito general que permitan aprovechar las capacidades de cómputo de dichas consolas.

Aunado a esto, es importante destacar que con el pasar de los años, las computadoras y consolas de videojuegos tienen mejores capacidades de hardware y de cómputo de alto desempeño. Es habitual ver en el mercado tarjetas gráficas que cuentan con capacidades aceleradoras avanzadas y una elevada capacidad de almacenamiento.

La evolución de las consolas de videojuegos a nivel de hardware ha crecido notablemente, sus capacidades aumentan en paralelo a los computadores recientes, e incluso han llegado a superarlos.

Este Trabajo Especial de Grado plantea la posibilidad de desarrollar aplicaciones de propósito general, distintas a un videojuego, en una consola de Xbox 360 con el fin de ampliar su funcionalidad, y aprovechar su potencialidad no solo para el entretenimiento sino también en otras áreas como la enseñanza, y el ámbito científico – investigativo. Esto supone además el estudio comparativo entre la capacidad de cómputo de una computadora y una consola de videojuegos, específicamente el Xbox 360, a través de la implementación de algoritmos complejos, como es el caso del Despliegue de Volúmenes.

Planteamiento del problema y motivación

En ocasiones el poder de cómputo de los computadores de consumo masivo es insuficiente para hacer, en un tiempo razonable, despliegue de volúmenes. La sobrecarga del despliegue de volúmenes en un sistema computacional depende directamente del tamaño del volumen de datos. Si se quiere visualizar volúmenes de gran tamaño (volúmenes médicos, simulaciones de fluidos, fenómenos naturales) se requiere de un sistema que posea un poder de cómputo capaz de manipular e interactuar con dichos datos en tiempo real para su visualización.

Las consolas de videojuegos, requieren de un gran poder de cómputo para el despliegue de gráficos cada vez más costosos, grandes geometrías, modelos 3D complejos, técnicas de iluminación y sombreado foto-realista, despliegue de fenómenos físicos, etc. Todo esto de manera interactiva, en donde el jugador pueda percibir en tiempo real lo que ocurre en el juego y responder con la misma rapidez.

Por ende, las consolas de videojuegos cuentan con un alto poder de procesamiento para cubrir todos estos requerimientos que los juegos modernos exigen de ellas. Esto nos hizo pensar que podríamos aprovechar estas capacidades para propósito general, en este caso para el despliegue de volúmenes. El problema que se plantea en este trabajo, es el de desarrollar y probar una aplicación de despliegue de volúmenes que se ejecute en una consola de videojuegos.

Propuesta de solución

La tecnología XNA de Microsoft, brinda facilidades que permiten crear aplicaciones para la consola Microsoft Xbox 360, que es una de las consolas más populares del mercado y con hardware más poderoso. Desplegar volúmenes a través de esta consola supone una ventaja, no sólo en la visualización de volúmenes, aprovechando las potencialidades de cómputo, sino también en la interactividad que brinda su interfaz para la manipulación del volumen.

Objetivo general

- Desarrollar una aplicación para el despliegue de volúmenes utilizando XNA.

Objetivos específicos

- Implementar la técnica de *Ray Casting* y *Texture Mapping* de *Volume Rendering* bajo el *framework* XNA 3.1 sobre las plataformas Microsoft Windows y Xbox 360.
- Implementar una interfaz capaz de controlar la interacción del usuario en ambas plataformas.

- Comparar el rendimiento de las técnicas de *Volume Rendering* implementadas tanto en la plataforma de Xbox 360 como en un computador tradicional.
- Realizar pruebas de rendimiento de las técnicas implementadas con volúmenes de diferentes tamaños.
- Encontrar el tamaño máximo de volumen que se puede desplegar en la consola Xbox 360.
- Integrar la herramienta desarrollada con la Mesa de Realidad Virtual del Centro de Computación Gráfica de la UCV.

Metodología

Se desarrollará una interfaz gráfica en C# para controlar la carga de volúmenes y manipular la visualización. Así mismo, dicha interfaz debe permitir editar los valores de la función de transferencia.

Se usará Programación Orientada a Objetos durante el desarrollo de la aplicación. Se utilizará *Visual Studio* 10.0 como entorno de desarrollo bajo el lenguaje C# y el *framework* XNA 3.1.

Para el desarrollo de *shaders* se utilizará el lenguaje HLSL (*High Level Shading Language*) por razones de compatibilidad con DirectX 9.0.

Plataforma de Hardware

- Tarjeta de video compatible con DirectX 9.0 o superior para Windows.
- Tarjeta de video que soporte *Shader Model* 2.0.
- Consola de videojuego: Xbox 360 con una suscripción al XNA Creators Club para la utilización de la aplicación en la consola.

Estructura del documento de TEG

Este trabajo especial de grado está estructurado de la siguiente manera: En el primer capítulo se expone brevemente los conceptos teóricos del Despliegue de Volúmenes. En el segundo capítulo se estudia la tecnología XNA como plataforma de desarrollo, así como se presenta su arquitectura y los componentes que lo conforman. En el tercer capítulo se expone brevemente las capacidades técnicas de la consola Xbox 360 y algunas aplicaciones desarrolladas en la consola. Luego, en el cuarto capítulo se muestran los detalles de implementación del Despliegue Directo de Volúmenes utilizando el *framework* de XNA. Los resultados son analizados y expuestos en el quinto capítulo. Finalmente en el sexto capítulo se exponen las conclusiones y los trabajos a futuro.

CAPÍTULO I. Despliegue de Volúmenes

Este capítulo describe toda la teoría relacionada al despliegue de volúmenes y algunas de las diferentes técnicas utilizadas en dicho despliegue.

El Despliegue de Volúmenes (*Volume Rendering*) es una técnica utilizada para desplegar una proyección bidimensional de datos tridimensionales discretos. Entre las aplicaciones principales de esta técnica están la visualización de datos médicos, datos geológicos, o la representación de fenómenos naturales.

Es una de las técnicas más importantes para la visualización de volúmenes. Permite a los científicos obtener rápidamente la comprensión de los datos biomédicos, industriales y de simulación, para que pueda hacer el análisis correspondiente.

Comúnmente los datos tridimensionales suelen provenir de un conjunto de imágenes bidimensionales, los cuales constituyen sus cortes, separados a una distancia constante a lo largo del volumen. Estos datos pueden ser adquiridos mediante Tomografía Computarizada, Resonancia Magnética o Micro-tomografía Computarizada. De igual forma, los datos tridimensionales pueden ser generados por un modelo matemático.

Para desplegar una proyección bidimensional de un conjunto de datos tridimensionales, lo primero que se necesita es definir un punto de vista en un espacio relativo al volumen. También es necesario definir la opacidad y el color de todos los elementos que conforman el volumen. Esto es usualmente definido usando tuplas RGBA (canales Rojo, Verde, Azul y Alpha).

Un volumen puede ser desplegado extrayendo su superficie y desplegándola como un mallado poligonal o desplegando el volumen directamente como un bloque de datos. En el primer caso, el algoritmo de *Marching Cubes* [1] [2] es una técnica común para extraer la superficie del volumen. El segundo caso se explica con mayor detalle a continuación.

1.1. Despliegue Directo de Volúmenes

El Despliegue Directo de Volúmenes es una tarea computacionalmente intensa que puede ser realizada de varias maneras, evitando etapas de reconstrucción previa del volumen para su visualización.

El Despliegue Directo de Volúmenes es una aproximación de la propagación de la luz a través de un medio representado por el volumen. Requiere del muestreo de los elementos que conforman el volumen, los cuales han de tener una opacidad y/o un color, a lo largo de un rayo de luz. El muestreo resulta en un valor RGBA, el cual es proyectado en el píxel correspondiente de la imagen resultante. La manera de hacer esto depende de la técnica de renderización que se utilice. Para propósitos prácticos, el volumen es representado como un arreglo tridimensional de valores de muestras. Cada muestra del volumen se denomina vóxel

(acrónimo de *volume element*). La Figura 1 representa el despliegue de un volumen utilizando este método.

Para simular la propagación de la luz, se utiliza un modelo óptico que simula la acumulación de las propiedades ópticas a lo largo de cada rayo de visualización para formar la imagen final. En esencia, el objetivo del modelo óptico es describir cómo las partículas del volumen interactúan con la luz. Para ello, existen diferentes modelos ópticos que simulan fenómenos como la absorción, emisión y dispersión de la luz sobre los elementos del volumen. Para efectos de esta investigación, se estudiará el fenómeno de emisión y absorción de la luz de los elementos del volumen.

Las propiedades ópticas de los elementos del volumen son especificadas por su valor asociado directamente o aplicando una función de transferencia al volumen. Esta función de transferencia enfatiza o clasifica datos interesantes del volumen.

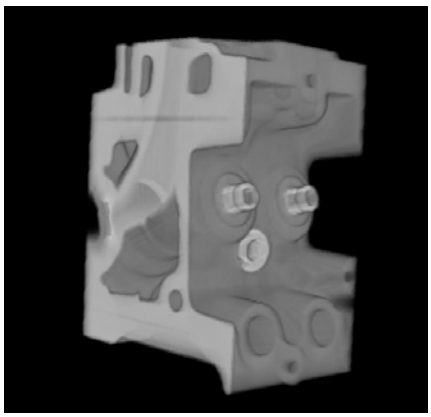


Figura 1: Representación de un volumen utilizando Despliegue Directo de Volúmenes.

La imagen final es creada muestreando el volumen a través de cada rayo de visualización y acumulando las propiedades ópticas resultantes. El modelo óptico utilizado, fue introducido en los trabajos [3] [4], y es representado matemáticamente por la siguiente ecuación:

$$C = \int_0^D c(\lambda)\tau(\lambda)e^{-\int_0^\lambda \tau(\lambda')d\lambda'}d\lambda, \qquad \text{(Ec. 1)}$$

en donde:

- C es el color resultante
- $\lambda \in [0, D]$, donde D representa la distancia que recorre el rayo dentro del volumen
- $c(\lambda)$ corresponde a la emisión de color a una distancia λ de la entrada del rayo en el volumen
- $\tau(\lambda)$ corresponde a la absorción a una distancia λ de la entrada del rayo en el volumen
- $e^{-\int_0^\lambda \tau(\lambda')d\lambda'}$ corresponde a la atenuación de la luz debido a la absorción y dispersión

La integral representa la suma de emisión de luz al entrar el rayo (en un punto del volumen) hasta que sale del mismo. El factor de atenuación puede ser interpretado como la transparencia $T(\lambda)$ del volumen con una distancia λ. Basado en esto, se puede calcular la opacidad acumulada en la travesía del rayo a una distancia λ. Así, la opacidad es representada por α, denotando lo siguiente:

$$\alpha(\lambda) = 1 - T(\lambda) = 1 - e^{-\int_0^{\lambda} \tau(\lambda')d\lambda'} \qquad \text{(Ec. 2)}$$

1.1.1. Clasificación de los datos

La clasificación de datos consiste en elegir en qué forma serán desplegados los datos en base a sus valores. Para poder clasificar los datos del volumen, es necesario definir una función de transferencia. En la Figura 2 se muestra un volumen y un corte del mismo, utilizando una función de transferencia unidimensional. La Figura 2.b muestra la función de transferencia, donde el eje vertical representa el α normalizado.

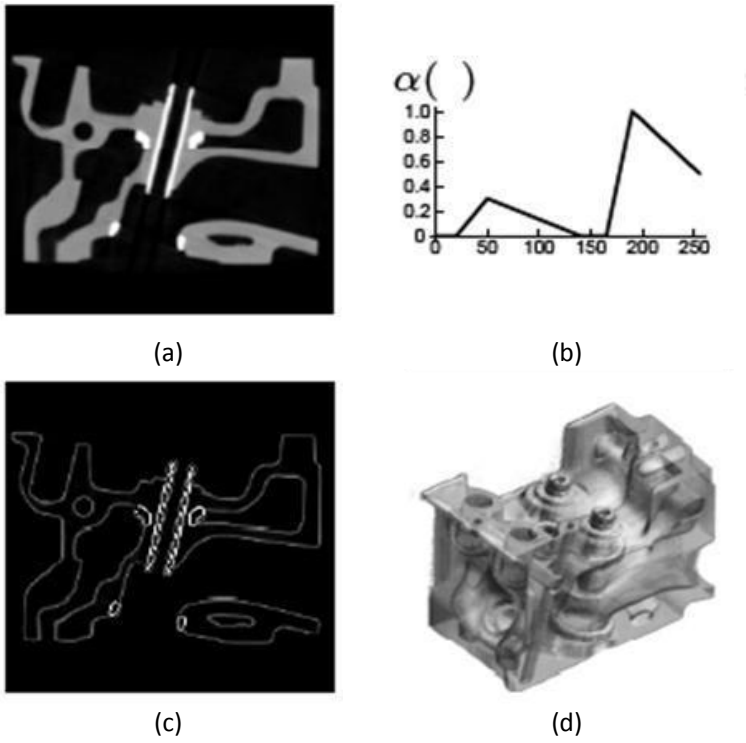


Figura 2: La figura (a) muestra un corte del volumen (d). La imagen (c) muestra el corte (a) con la función de transferencia (b).

Esta permite asignar las propiedades ópticas de cada vóxel de un volumen, el cual permite al usuario enfatizar o resaltar algunas estructuras del mismo. La función de transferencia normalmente es unidimensional y puede ser una simple función identidad, una función lineal a trozos o una tabla arbitraria. En la Figura 3 se muestra la aplicación de diferentes funciones de transferencia a un mismo volumen.

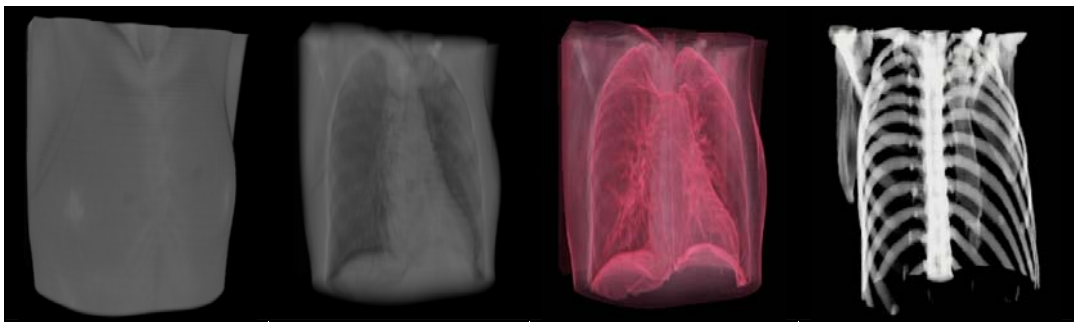


Figura 3: Visualización de un mismo volumen utilizando diferentes funciones de transferencia unidimensionales.

De la misma manera se pueden definir funciones multidimensionales para asignar otras propiedades ópticas a un vóxel (histograma de frecuencias, longitud del gradiente, posición de la muestra, entre otros). En esta investigación sólo se utilizó una función de transferencia unidimensional.

Existen dos formas de clasificar los datos, las cuales básicamente difieren en la forma en que se evalúa la Ecuación 1, a saber:

- **Pre-clasificación:** Cada muestra del volumen es reemplazada por su valor dentro de la función de transferencia. Se almacena una tupla RGBA para cada muestra, en donde se representa la emisión y absorción mediante los canales RGB y A respectivamente. De este modo, el volumen se encuentra preclasificado antes que cada muestra sea interpolada. De esta manera, la Ecuación 1 queda de la siguiente forma:

$$C = \int_0^D s\big(x(\lambda)\big)d\lambda \qquad \text{(Ec. 3)}$$

  en donde $x(\lambda)$ es la parametrización del rayo evaluada en λ, que representa un punto (x, y, z) en el volumen y $s\big(x(\lambda)\big)$ representa el color de la muestra que contiene el valor RGBA interpolado.

- **Post-clasificación:** La función de transferencia es aplicada después de que la muestra haya sido obtenida por la interpolación de muestras, a partir de las muestras escalares. De esta manera, la Ecuación 1 queda de la siguiente forma:

$$C = \int_0^D c\left(s\big(x(\lambda)\big)\right)\tau\left(s\big(x(\lambda)\big)\right)e^{-\int_0^{\lambda}\tau\left(s\left(x(\lambda')\right)\right)d\lambda'}d\lambda \qquad \text{(Ec. 4)}$$

en donde $x(\lambda)$ representa un punto (x, y, z) en el volumen, $s\big(x(\lambda)\big)$ representa la muestra escalar interpolada, $c\left(s\big(x(\lambda)\big)\right)$ y $\tau\left(s\big(x(\lambda)\big)\right)$ representan la aplicación de la función de transferencia a dicha muestra, para así obtener su color y absorción.

Estas ecuaciones evalúan el rayo de forma continua. Para realizar la evaluación de manera discreta, ésta se aproxima mediante sumas de Riemann [5], en donde se divide el rayo en n segmentos, en donde $n = \lceil D/h \rceil$ y cada segmento tiene una longitud h, como se ilustra en la Figura 4.

Haciendo énfasis en la Ecuación 4, se aproxima el factor de atenuación de la misma manera:

$$e^{-\int_0^{\lambda}\tau\left(s\left(x(\lambda')\right)\right)d\lambda'} \approx e^{-\sum_{i=0}^{\lceil\lambda/h\rceil}\tau\left(s\left(x(ih)\right)\right)}$$

$$= \prod_{i=0}^{m-1} e^{-\tau\left(s\left(x(ih)\right)\right)} = \prod_{i=0}^{m-1}(1-\alpha_i) \qquad \text{(Ec. 5)}$$

en donde $m = \lceil \lambda/h \rceil$, es la cantidad de segmentos del factor de atenuación con una longitud h para cada segmento, y α_i representa la opacidad del i-ésimo segmento del rayo. Luego se aproxima la emisión de luz del i-ésimo segmento del rayo por $c_i \approx c\left(s\big(x(ih)\big)\right)\tau\left(s\big(x(\lambda)\big)\right)$. De este modo se rescribe la Ecuación 4 como:

$$C \approx \sum_{i=0}^{n-1} c_i\alpha_i \prod_{j=0}^{i-1}\big(1-\alpha_j\big)$$

$$\Rightarrow C \approx c_0\alpha_0 + c_1\alpha_1(1-\alpha_0) + c_2\alpha_2(1-\alpha_0)(1-\alpha_1) + \cdots + c_{n-1}\alpha_{n-1}(1-\alpha_0)$$
$$(1-\alpha_1)\ldots(1-\alpha_{n-3})(1-\alpha_{n-2}) \qquad \text{(Ec. 6)}$$

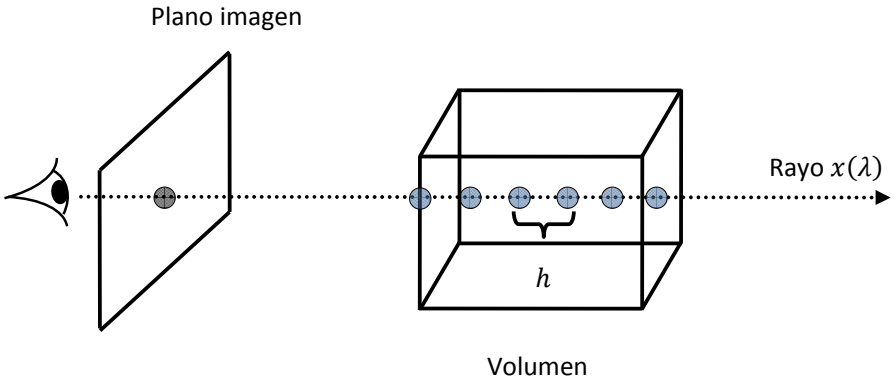


Figura 4: Determinación de un píxel de la imagen con la travesía del rayo desde la cámara hasta atravesar el volumen, con una distancia h entre cada muestra.

La Ecuación 6 puede ser evaluada de dos maneras diferentes:

- **Composición *back-to-front*:** se evalúan y acumulan las muestras desde la más lejana hasta la más cercana, usando las siguientes ecuaciones:

$$C'_n = 0, C'_i = c_i\alpha_i + (1 - \alpha_i)C'_{i+1} \quad \text{(Ec. 7)}$$

$$\alpha'_n = 0, \alpha'_i = \alpha_i + (1 - \alpha_i)\alpha'_{i+1} \quad \text{(Ec. 8)}$$

  en donde C'_i y α'_i son el color y el factor acumulado cuando quedan i muestras por evaluar. El color resultante es aquel encontrado cuando no quedan muestras por evaluar, es decir $C \approx C'_0$.

- **Composición *front-to-back*:** se evalúan y acumulan las muestras desde la más cercana hasta la más lejana, usando las siguientes ecuaciones:

$$C'_0 = 0, C'_{i+1} = (1 - \alpha'_i)c_i\alpha_i + C'_i \quad \text{(Ec. 9)}$$

$$\alpha'_0 = 0, \alpha'_{i+1} = (1 - \alpha'_i)\alpha_i + \alpha'_i \quad \text{(Ec. 10)}$$

  en donde C'_i y α'_i son el color y el factor de atenuación acumulado después de evaluar i muestras respectivamente. El color resultante es aquel encontrado después de evaluar n muestras, es decir $C \approx C'_n$.

1.1.2. Técnicas para el despliegue de volúmenes

Existen varias técnicas para el despliegue directo de volúmenes, entre las cuales se encuentra: *Ray Casting* [6], Basado en Texturas [7], *Shear-Warp* [8] y *Splatting* [9].

Algunos de estos algoritmos son de orden de imagen (*image order*), puesto que determinan el color en cada píxel de la imagen buscando los elementos del volumen que contribuyen a cada uno de estos. En cambio, otros algoritmos son de orden de objeto (*object order*), que consiste en calcular la contribución de cada vóxel del volumen a los píxeles de la imagen [10].

En este trabajo sólo se desarrollan dos técnicas: *Ray Casting* y la técnica Basada en Texturas: Planos Alineados al *Viewport*. El Ray Casting se ha implementado únicamente para GPU, utilizando una composición *front-to-back*. En el caso de Planos Alineados al Viewport, se utilizó composición *back-to-front*. En ambas técnicas se utilizó post-clasificación para la representación de los datos.

1.1.1.1. Ray Casting

La técnica de despliegue de volúmenes a través de *Ray Casting* puede derivar directamente de las ecuaciones de *rendering*. Esto provee resultados de alta calidad, usualmente considerados para proveer la mejor calidad de imagen. Los algoritmos basados en esta técnica son de orden de imagen.

En esta técnica, un rayo es generado por cada píxel de imagen deseado. Usando una simple cámara modelo, el rayo comienza desde el centro de proyección de la cámara (usualmente la posición del ojo) y pasa a través del píxel de la imagen hasta llegar al volumen a ser desplegado, como se indica en la Figura 5 [6]. El rayo es cortado por los límites del volumen con el fin de optimizar tiempo de cómputo. Luego el rayo es muestreado en intervalos de espacio regulares a lo largo del volumen.

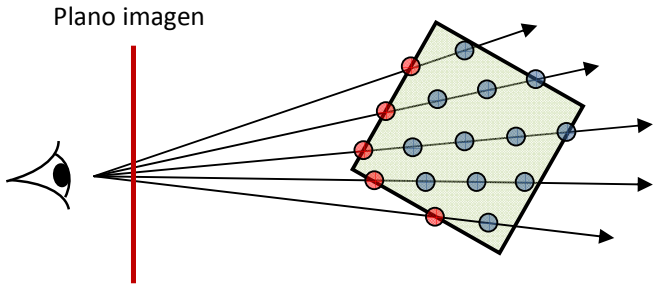


Figura 5: Ray Casting.

En general el rayo atraviesa el volumen en una dirección arbitraria, y las muestras requeridas durante la travesía del rayo no coinciden con las muestras originales del volumen. Por lo tanto, usualmente se utilizan filtros tomando varias muestras cercanas conocidas a la muestra requerida para obtener una aproximación de ésta.

La función de transferencia es aplicada para obtener un color RGBA. Este color se combinará con el color acumulado del rayo, y este proceso es repetido hasta que el rayo sale del volumen. El color RGBA resultante es convertido a formato RGB para ser almacenado como píxel. Este proceso es repetido para cada píxel de la pantalla para formar la imagen completa.

El algoritmo de *Ray Casting* realiza, por su naturaleza, gran cantidad de cómputo. Además no es posible aprovechar la localidad espacial del volumen. Para acelerar el proceso de visualización se pueden aplicar las siguientes optimizaciones:

- **Terminación temprana del rayo**: Consiste en truncar el rayo al conseguir suficiente opacidad en la travesía del mismo, ya que los siguientes elementos no aportan información significativa al color final del píxel [11].
- **Saltos de espacios vacíos:** El volumen puede contener espacios vacíos por lo que, durante la travesía del rayo, se puede optimizar el algoritmo saltando dichos espacios.

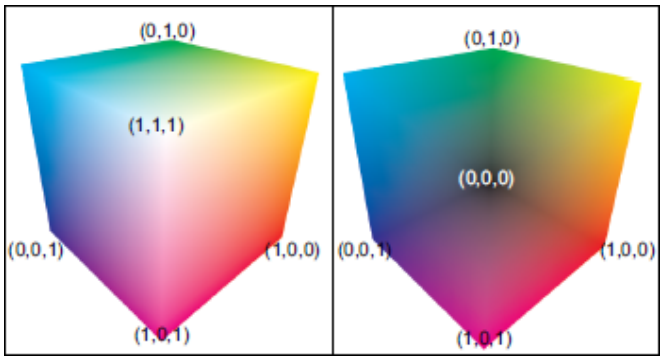
1.1.1.1.1. Ray Casting acelerado por GPU

La técnica de *Ray Casting* fue concebida inicialmente para ser implementada por CPU. Actualmente con la evolución de las tarjetas gráficas, es posible programar esta técnica junto

con las optimizaciones mencionadas anteriormente y ser adaptadas para ser ejecutadas por el GPU.

Krüger y Westermann [12] propusieron un algoritmo para el trazado de los rayos en los procesadores de la GPU, aprovechando el paralelismo, ya que cada rayo es independiente del otro. Almacenando el volumen como una textura 3D, el algoritmo se divide en varias fases que se muestra a continuación:

- **Determinación del punto de entrada:** Se despliegan en una textura 2D, con formato RGB, las caras delanteras de un cubo unitario. Las coordenadas de textura 3D son asignadas a cada vértice de este cubo, y a la textura se le asigna el mismo valor para el color por cada fragmento del cubo. Luego el rasterizador interpola los valores que se encuentran en cada vértice del cubo. Esta textura 2D tiene la misma resolución que la imagen de la pantalla de visualización o *viewport*. La Figura 6.a muestra las caras delanteras que conforman el cubo unitario.
- **Determinación de la dirección de cada rayo:** En esta fase se despliegan las caras traseras del cubo unitario en otra textura 2D, de igual forma que en el paso anterior. Aprovechando la flexibilidad de las tarjetas gráficas, se ejecuta un *shader* que toma el valor que tiene la textura generada en el paso anterior por cada píxel y se calcula la dirección normalizada del rayo correspondiente. Adicionalmente, se almacena la longitud de la dirección del rayo en el canal alfa. Esta nueva textura contiene la dirección del rayo normalizado y la longitud del mismo (ver Figura 6.b).
- **Recorrido del rayo y terminación temprana del rayo:** En esta fase se realiza el recorrido del rayo a través del volumen, utilizando las dos texturas generadas en las fases anteriores. El rayo recorre n cantidades de pasos, muestreando el volumen para así obtener el valor final del píxel. Se define un umbral constante T hasta donde será tomado en cuenta el aporte de opacidad del rayo.



(a) (b)

Figura 6: Despliegue de las caras delanteras (a). Despliegue de las caras traseras (b).

1.1.1.2. Basado en texturas

Las tarjetas gráficas son capaces de aplicar texturas a cualquier primitiva y desplegarla en pantalla. Es posible renderizar un volumen aplicando texturas a cada corte del volumen, aprovechando así las capacidades de las tarjetas gráficas [7].

La opacidad de los vóxeles equivale a la opacidad de los téxeles de la textura, lo cual se puede mapear con una operación *Blending* que provee el hardware gráfico, necesario para construir el volumen dado un punto de vista.

Es posible cargar texturas 2D o 3D en la tarjeta gráfica. Esto da lugar a dos tipos de técnicas de Despliegue de Volúmenes basado en texturas; planos alineados al *viewport*, cuando se trabaja con texturas 3D y planos alineados al objeto si se trabaja con texturas 2D.

1.1.1.2.1. Planos alineados al *viewport*

Esta técnica consiste en cargar una textura 3D a la tarjeta gráfica y crear una geometría a la cual ha de mapearse dicha textura. La geometría está dividida en varios cortes, alineados al plano imagen, que se superponen los unos a los otros, comenzando desde la textura más alejada a la cámara y se componen con una operación de *Blending* realizada por la tarjeta gráfica [13]. Al rotar el volumen para su visualización, se rotan las coordenadas de textura y se mapean nuevamente a los cortes con las nuevas coordenadas como se aprecia en la Figura 7.

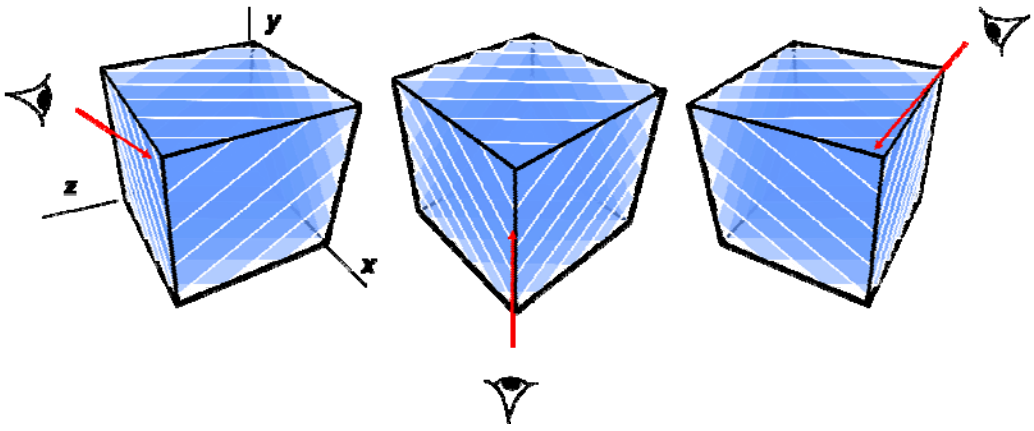


Figura 7: Planos alineados al *viewport*.

CAPÍTULO II. XNA

En este capítulo se describen todos los componentes y plataformas necesarios para el desarrollo de una aplicación en XNA. Así mismo, se describe el modelo de capas de XNA y algunas estructuras básicas del *framework*, esenciales para el desarrollo de la aplicación.

XNA (***X****NA is* ***N****ot an* ***A****cronym*, por sus siglas en inglés) es un conjunto de herramientas de desarrollo que permite la creación de videojuegos en distintas plataformas, de manera ágil y rápida. Fue creada por Microsoft, anunciada al público en el año 2004 y su primera versión fue publicada el 14 de marzo del 2006.

XNA es un *framework* que está basado en el *framework* de .NET y en el .NET *Compact framework* para Windows y algunos dispositivos móviles respectivamente [14]. Ambos *frameworks* se ejecutan bajo el Lenguaje Común de Tiempo de Ejecución o CLR (*Common Language Runtime*) de Microsoft. El CLR actúa como una máquina virtual que permite la ejecución de cualquier código realizado para la plataforma .NET, lo cual trae consigo ciertos beneficios.

Un *framework* es un marco de trabajo con el cual se pretende facilitar al programador el desarrollo de software en general, apartándolo de preocupaciones de implementaciones de bajo nivel y enfocándose más bien en identificar los requerimientos del software.

En el año 2006 fue liberada la versión 1.0 de XNA. Esta versión del *framework* permitía desarrollar videojuegos únicamente para plataformas Windows y Xbox 360. La versión 2.0 liberada en el año 2007, mejoró la arquitectura dando soporte para todas las versiones de *Visual Studio* 2005 (en la versión anterior sólo *Visual C# Express Edition* 2005 estaba soportado). La versión 3.0 permitía ejecutar videojuegos en la plataforma *Zune* (reproductor de audio digital) con soporte para C# 3.0. La versión más reciente fue liberada en Marzo de 2010 (XNA 4.0) que provee soporte para los celulares *Windows Phone 7*, integración con *Visual Studio 2010*, captura de eventos con tecnología de pantalla táctil, entre otras mejoras.

Este Trabajo Especial de Grado se desarrolló utilizando la versión de XNA 3.1 bajo la plataforma de *Visual Studio 2008.*

2.1 La Plataforma de XNA

La Plataforma de XNA consiste en los siguientes componentes principales: El XNA Framework, el XNA Game Studio, DirectX y el .NET Framework. Esta plataforma permite desarrollar aplicaciones para PC, las consolas de videojuego de Microsoft, el reproductor de audio digital Zune y recientemente, los celulares inteligentes con Windows Phone 7. La Figura 8 muestra un esquema de la Plataforma de XNA. A continuación se describen con mayor detalle los componentes de la plataforma.

- **XNA Game Studio:** Es una extensión de *Visual Studio C#* (o en su defecto *Visual Studio C# Express*) que incluye proyectos y plantillas para el uso del *framework* XNA, de manera gratuita. *Visual Studio* es un entorno de programación de última generación

diseñado para construir aplicaciones y software para PC. XNA está diseñado para trabajar en conjunto con las versiones de *Visual Studio* (a partir de la versión 2.0 de XNA). Para generar un videojuego para la consola Xbox 360, es necesario pagar por una subscripción anual en *XNA Creators Club* para ejecutarlo en dicha plataforma.

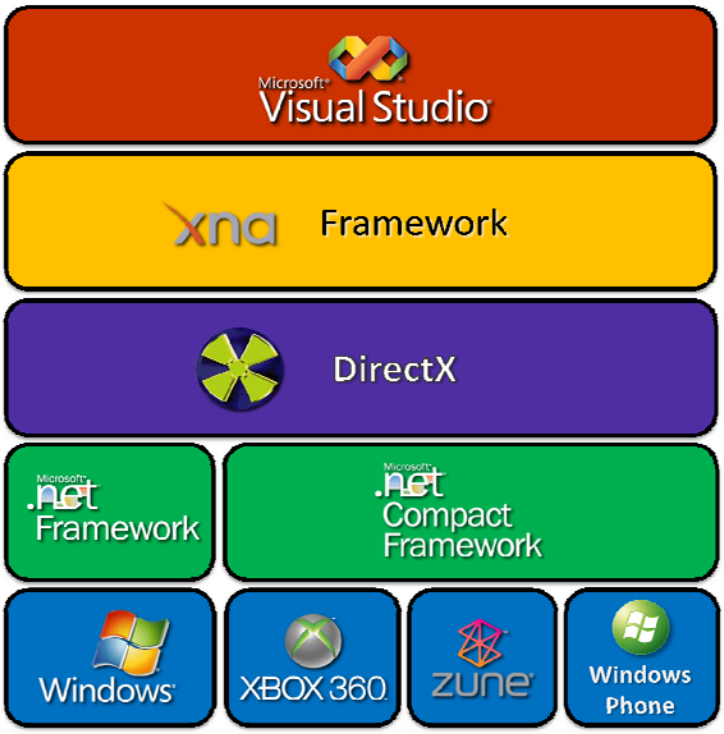


Figura 8: Plataforma de XNA.

- **XNA**: Es un conjunto de librerías que permiten el despliegue de elementos gráficos, reproducción de sonidos, entre otras tareas de un videojuego. Como se observa en la Figura 8, el *framework* de XNA se ejecuta dependiendo de la plataforma destino. El *framework* .NET de una PC no es el mismo que *framework* .NET del Xbox 360, por ejemplo. El *framework* de XNA se explica con mayor detalle en la sección 4.
- **DirectX**: Es una colección avanzada de APIs destinadas al manejo multimedia en un computador, especialmente para la programación de juegos y de vídeo. Funciona sobre las plataformas de Microsoft Windows.

  DirectX permite a los programadores acceder al hardware en donde se ejecuta, por ello no se necesita escribir código para un hardware específico. Proporciona a los programadores una forma estandarizada de acceso a los recursos de hardware de alto rendimiento como las tarjetas gráficas aceleradoras, así como las tarjetas de sonido.

  El programador debe conocer las diferentes APIs que componen DirectX:

- **Direct3D**: Es un conjunto completo de servicios gráficos 3D en tiempo real, que se encarga de todo el renderizado basado en software-hardware de todo el procesamiento gráfico.

- **DirectSound**: Proporciona utilidades de mezcla de sonido a baja-latencia, aceleración por hardware, y acceso directo al dispositivo de sonido. También permite la aceleración por hardware de sonidos 3D.

- **DirectPlay**: Representa una capa de software que simplifica el acceso a los servicios de comunicación entre computadoras. Provee a los juegos una manera de comunicarse entre sí que es independiente del medio de transporte subyacente, protocolo, o el servicio en línea.

- **DirectInput**: Proporciona acceso rápido y consistente a palancas de mando analógicas y digitales (*Joysticks*).

- **DirectMusic**: Reproduce pistas musicales compuestas con *DirectMusic Producer*.

Direct3D en particular permite la ejecución de código HLSL [15]. A pesar de que HLSL es un lenguaje de alto nivel, un código desarrollado en HLSL viene a ser para la tarjeta gráfica lo equivalente a un código escrito en lenguaje ensamblador, puesto que al compilar el código fuente se genera un código de lenguaje máquina. Un código desarrollado en HLSL es conocido como *shader*. Los *shaders* permiten acelerar el despliegue de los gráficos en la plataforma XNA y son un componente importante en el desarrollo de este Trabajo Especial de Grado. A continuación se describe brevemente como se procesa un *shader*.

Proceso del Shader

La tarjeta gráfica puede ejecutar instrucciones que han de ser aplicadas a todos los vértices de la escena (*Vertex Shaders*), así como a todas las partes visibles de las geometrías presentes en la escena una vez que se han proyectado en el plano imagen (*Pixel Shaders*).

La plataforma XNA proporciona a la tarjeta gráfica los datos y parámetros referentes a los vértices de las primitivas para su despliegue. La tarjeta gráfica procesará estos datos y posteriormente les aplicará las transformaciones hechas a dichos vértices a través del *Vertex Shader*. Una vez aplicadas estas transformaciones, el resultado es enviado al rasterizador, el cual se encarga de transformar toda la escena en píxeles. Una vez obtenidos estos píxeles, la tarjeta gráfica le aplica las modificaciones indicadas en el *Pixel Shader,* bien sea interpolación de colores, normales, las coordenadas de texturas, etc. utilizadas en efectos de iluminación, relieves, *multitexturing*, etc.; antes

de asignarle el color correspondiente a cada píxel. Finalmente, los píxeles son enviados al *frame buffer* para ser desplegados por pantalla.

En la Figura 9, se muestra un diagrama del proceso que sigue XNA para ejecutar un *shader*, bien sea de vértices o de píxeles.

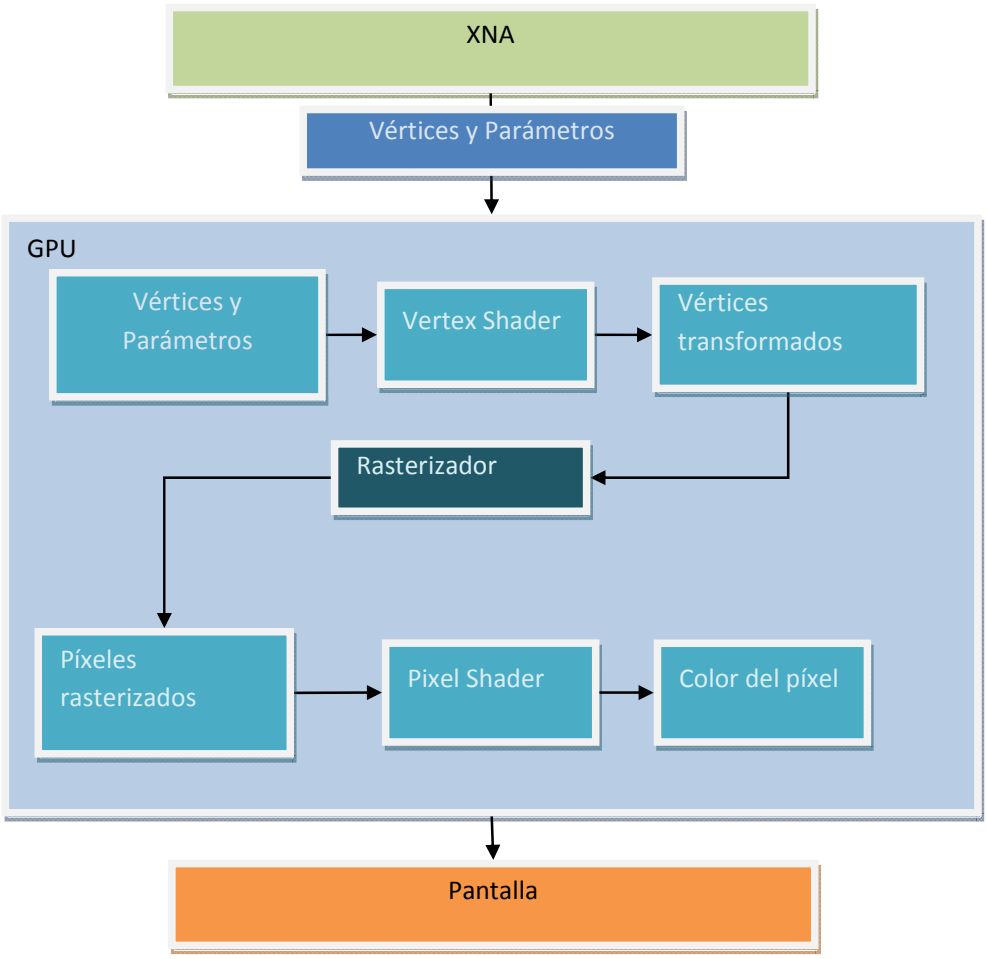


Figura 9: Proceso del Shader.

- **.NET *framework*:** Cuando se compila un programa hecho para la plataforma .NET, el resultado no es un código ejecutable, sino un archivo que contiene un tipo especial de pseudocódigo llamado Lenguaje Intermedio Microsoft o MSIL (*Microsoft Intermediate Language*). MSIL define un conjunto de instrucciones portables que son independientes de cualquier CPU, es decir, define un lenguaje ensamblador portable.

  El trabajo del CLR es traducir el código intermedio en código ejecutable cuando se ejecuta un programa. De esta manera, cualquier programa compilado en MSIL puede ser ejecutado en cualquier ambiente donde esté implementado CLR.

La Figura 10 ilustra el proceso de compilación para cualquiera de los lenguajes de programación que formen parte del entorno de .NET.

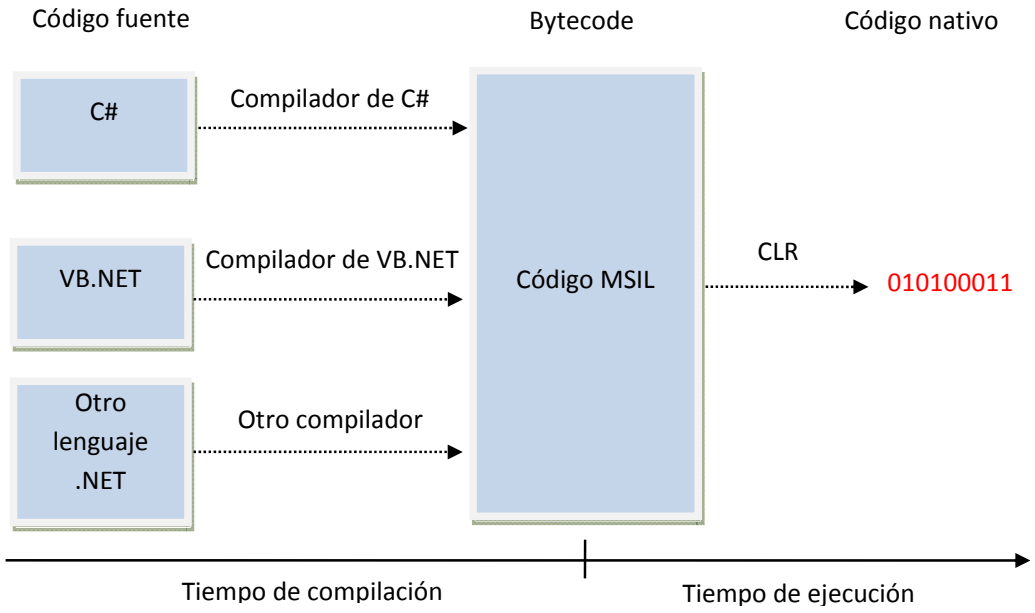


Figura 10: Proceso de compilación con la plataforma .NET.

En pocas palabras, .NET *framework* define un ambiente que soporta el desarrollo y la ejecución de aplicaciones altamente distribuibles basadas en componentes. Permite que diferentes lenguajes de cómputo trabajen juntos y proporciona seguridad, portabilidad y un modelo común para la plataforma de Windows.

2.2 Arquitectura del *framework* de XNA

De acuerdo a la Figura 11, el modelo de capas de XNA está constituido de la siguiente forma:

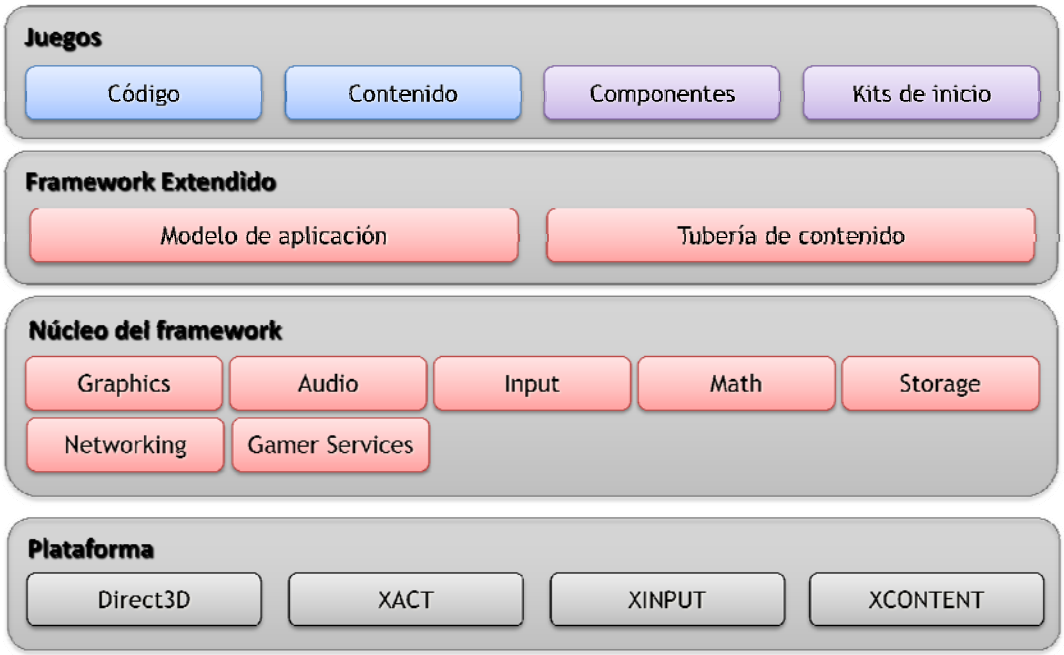


Figura 11: Modelo de capas de XNA.

2.2.1 Capa Plataforma

Es la capa más baja del *framework* de XNA. Contiene las funciones de las API nativas que son utilizadas en las capas superiores. Dentro de esta capa se encuentran las siguientes APIs: ***Direct3D***, ***XACT***, ***XINPUT*** y ***XCONTENT***.

- **Direct3D** es un servicio completo para el procesamiento y programación de gráficos en 3D. Es una de las características más usadas en DirectX. El objetivo principal de esta API es proveer el manejo y despliegue de primitivas en un espacio tridimensional, como son las líneas, polígonos y las texturas, así como aplicar transformaciones geométricas a dichas primitivas. La mayor ventaja de esta interfaz es que es transparente de la tarjeta gráfica aceleradora.
- **XACT** es una biblioteca para la programación de audio que viene como parte del SDK (*Software Development Kit*) de DirectX. Esta API opera sobre XAudio de Xbox, DirectSound en Windows XP, Windows Vista y Windows 7. XAudio es un API diseñado sólo para Xbox para el óptimo procesamiento de señales digitales. Al principio de su lanzamiento sólo era compatible con Xbox, posteriormente fue modificada para su funcionamiento en Windows.

- **XINPUT** es un API que funciona por encima de *DirectInput*. Es compatible con DirectX 9 en adelante y sólo da soporte al mando de Xbox 360 para Windows y la consola de Xbox 360. *DirectInput* es un API de Microsoft para almacenar la entrada de un usuario de la computadora, a través de dispositivos de entrada como el ratón, teclado, joystick y otros controles del juego.
- **XCONTENT** es un API para la administración de contenido multimedia que es utilizado más adelante por la tubería de contenido.

2.2.2 Capa Núcleo del *framework*

Es la capa siguiente a la capa Plataforma y provee un alto nivel de abstracción de los servicios proporcionados por su capa predecesora (ver Figura 11). Proporciona las funcionalidades básicas sobre las que las otras capas trabajan. Esta capa contiene APIs para las siguientes áreas: ***Graphics***, ***Audio***, ***Input***, ***Math***, ***Storage, Game Services y Networking***.

- ***Graphics*** está basado en el API de DirectX versión 9.0 y es una versión mejorada de lo que al principio era *Managed DirectX*. Contiene clases relacionadas con el acceso de modelos, texturas, efectos (*shaders*), entre otros. Además de manipular gráficos en tres dimensiones permite trabajar con gráficos en dos dimensiones.

  Cabe destacar que en el *framework* de XNA no existe soporte al pipeline de función fija al igual que la consola de Xbox 360. Esto es debido a que el pipeline programable permite crear cualquier efecto visual deseado en tiempo real, es capaz de permitir flexibilidad en las funciones predefinidas ya que se pueden reemplazar por cálculos más específicos, utilizando *shaders*.

  Sin embargo, la programación de *shaders* puede ser una tarea difícil. Como refuerzo a ello, existen clases que encapsulan *shaders* como por ejemplo ***BasicEffect***, el cual contiene ciertos atributos controlables como la iluminación o el texturizado para ser aplicado a un objeto dentro de una aplicación de gráficos en tres dimensiones. Usar esta clase permite, de manera muy rápida, desplegar algo en pantalla sin la necesidad de escribir realmente un *shader*.

- El API ***Audio*** está desarrollado sobre el API de *XACT* dentro de la arquitectura del *framework* XNA el cual está destinado a múltiples plataformas de hardware. Con este API se pueden crear paquetes de efectos de audio y configurar el volumen, las repeticiones mediante bucles, la mezcla de canales, entre otras cosas. Luego el desarrollador puede utilizar un paquete, cargarlo y reproducirlo fácilmente sin preocuparse de implementaciones de bajo nivel.

- El API ***Input*** está construido sobre el API de ***XINPUT*** dentro de la arquitectura del *framework* de XNA. Este API permite manipular un mando de control de Xbox 360 multiplataforma de manera muy simple. El programador no necesita inicializar ninguna

clase ni liberar recursos de los dispositivos de entrada, sólo necesita invocar al método *GetState* sobre el controlador adecuado. Entre los tipos de controladores se tiene el *GamePad* (control de mando de Xbox 360 para Windows y Xbox 360), el teclado *(*para ambas plataformas*)* y el ratón (solamente para Windows).

- El API ***Math*** provee una colección de clases que son frecuentemente usadas en la programación para el cálculo matemático. Por ejemplo, incluye la clase *Matrix* que representa una matriz de 4 filas y 4 columnas, con funciones esenciales tales como las transformaciones afines, proyecciones en perspectiva, proyecciones ortogonales, operaciones de suma, resta o multiplicación de matrices, entre otras funciones.

  Entre otras clases existentes en ***Math*** se encuentran ***Vector2, Vector3, Vector4*** (vectores de 2, 3 y 4 coordenadas cartesianas respectivamente), ***Plane*** (representación de un plano), ***Ray*** (representación de un rayo); ***BoundingBox, BoundingSphere,*** y ***BoundingFrustum*** (entes geométricos delimitadores para la detección de colisiones que además realizan pruebas de intersección y contención). Todas estas estructuras de datos son fundamentales en los algoritmos durante el desarrollo de un videojuego.

- El API ***Storage*** ofrece maneras de leer y guardar datos de un videojuego, como por ejemplo el progreso de un jugador en una partida, los trofeos obtenidos, puntuaciones, entre otros. En la consola Xbox 360 se debe asociar el estado del juego con un perfil y el dispositivo de almacenamiento, tales como el disco duro o una unidad de memoria externa.

- ***Gamer Services*** es un API que permite registrar y acceder a los módulos creados en XNA por el modelo de componentes. Es una interfaz que permite el acceso entre componentes modulares que dependen entre sí para su funcionamiento.

- ***Networking*** es un API que permite comunicar por red a varios equipos tanto PC como Xbox 360. Es posible almacenar una lista de amigos en estas dos plataformas y crear sesiones de red. Permite conectar dos equipos o más que estén dentro de una subred o a través de internet. Para establecer estas conexiones, es necesario tener una membresía en *XNA Creators Club*.

  XNA usa las plataformas de Xbox *LIVE*[1] y *Games for Windows LIVE* para realizar conexiones multi-jugador sobre una consola Xbox o computador. Es necesario tener una cuenta de usuario válida e iniciar sesión para tener acceso a ambos servicios.

  Este API resuelve una serie de problemas que el programador debe afrontar, como por ejemplo el envío y recepción de paquetes, manejo de errores, pérdida de datos, entre otros, relacionados con detalles de bajo nivel.

[1] *Xbox Live:* Es el servicio para jugar en línea a través de Internet con una cuenta de usuario.

Hay que considerar el tipo de red a utilizar (punto a punto, cliente/servidor), ya que tendrá un impacto en el manejo del tráfico de paquetes y en el rendimiento de la aplicación.

2.2.3 Capa Framework Extendido

El objetivo principal de la capa Framework Extendido es hacer el desarrollo de un videojuego lo más sencillo posible. Actualmente, esta capa contiene dos componentes básicos: El Modelo de Aplicación (*Application Model)* y la Tubería de Contenido (*Content Pipeline)*.

- **Modelo de aplicación:** El propósito del modelo de aplicación es apartar al programador de los problemas que pueda tener un videojuego al ser ejecutado en cualquier plataforma ya que se debe concentrar en codificar la lógica del juego. No tiene que preocuparse por crear una ventana o manejar los eventos del sistema operativo. Tampoco preocuparse de crear un temporizador que cuente el tiempo transcurrido. Todo esto está ofrecido dentro del modelo de aplicación.

  Cada aplicación hecha en XNA contiene una clase que deriva de la clase **Game,** que contiene todos los componentes esenciales, el dispositivo gráfico, la configuración de la pantalla, y el cargador de contenido multimedia. Además es posible manipular los dispositivos de entrada y la reproducción de sonidos dentro de esta misma clase. Básicamente se puede incluir parte del código del juego aquí o crear componentes basado en el modelo de componentes.

  Para ayudar al programador a organizar su código del juego en módulos manejables, XNA también proporciona un pequeño *framework* de servicio que puede utilizar para construir componentes reutilizables que proporcionan servicios al resto del juego.

  El modelo de componentes de XNA permite crear e incorporar ***GameComponents*** relacionados con el videojuego en desarrollo. Puede ser escrito incluso por otros y ser compartidos dentro de una comunidad de XNA.

- **Tubería de contenido (*Content Pipeline*)**: Provee importadores y procesadores de contenido multimedia para ser utilizados en los proyectos de XNA [14], [16]. Permite incorporar contenido gráfico al juego, tales como imágenes, audio, modelos en 3D, efectos, entre otros. La Figura 12 muestra un contenido gráfico creado por diseñadores y agregado al juego a través de la tubería de contenido de XNA.

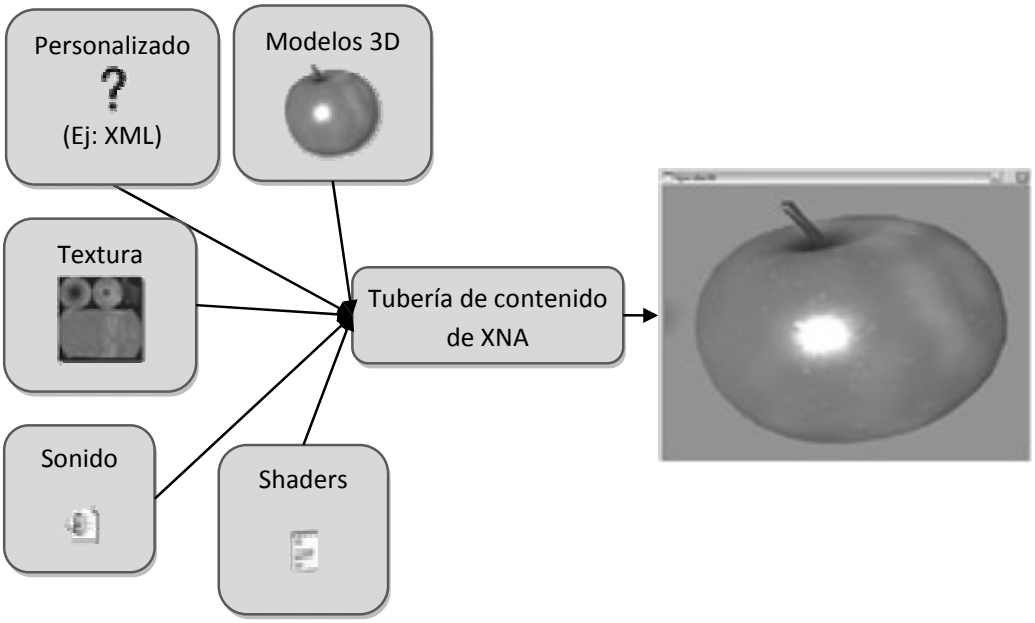


Figura 12: Un ejemplo de diferentes *assets*[2] cargados por el *Content Pipeline* utilizados para desplegar una geometría.

Soporta una gran cantidad de formatos diferentes, que son los más usados en la creación de un videojuego. Sin embargo, no soporta todos los formatos actuales y si es el caso, es posible crear importadores de contenido para nuevos formatos. Es útil comprender las etapas del *Content Pipeline* que transforman un archivo de contenido multimedia digital en archivos binarios para el juego.

El proceso de construcción de *assets* es controlado por los importadores y procesadores del *Content Pipeline*. Cuando se compila un proyecto, el importador y procesador del *Content Pipeline* apropiado para cada *asset* es invocado para su transformación (o construcción). Este proceso comienza con un *asset* como archivo en su forma original (formato DCC[3]), y continúa con su transformación a datos que puedan ser almacenados y usados dentro de la aplicación mediante la biblioteca de clases del *framework* de XNA.

La tubería de contenido es flexible para los diseñadores cuando crean contenido multimedia digital, incorporándolo mediante una interfaz unificada. Además, es extensible si se requiere incorporar un formato no soportado por el *framework* de XNA, ya que puede que el desarrollador utilice archivos con formato independiente de acuerdo a sus necesidades, e incluso extender un formato ya soportado.

[2] Asset: Elementos de arte tales como texturas, modelos 3D, fuentes, efectos y sonidos. También incluye información de interés o datos importantes para el despliegue de los elementos de arte.
[3] DCC: Siglas de *Digital Content Creation*; contenido multimedia creado por diferentes programas, tales como editores de modelos 3D, así como programas para el procesamiento de digital de imágenes.

Para este Trabajo Especial de Grado se utiliza la tubería de contenido para la carga de texturas 2D, fuentes de texto, *shaders*, curvas de animación, sonidos y volúmenes. Para los volúmenes fue necesario crear importadores y procesadores de contenidos para formatos no soportados por XNA. XNA acepta únicamente el formato de volumen *.dds*, sin embargo este trabajo fue diseñado para soportar solo *.pvm* y *.raw* por ser los más comunes.

Componentes de la tubería de contenido

Para que un *asset* esté disponible en un videojuego de XNA, es necesario agregarlo al *Content Project*. Después de ser parte del proyecto, éste es incluido al *Content Pipeline*.

La Figura 13 muestra los procesos involucrados en el *Content Pipeline*. Los procesos se pueden clasificar en dos tipos: Componentes en tiempo de diseño y componentes en tiempo de ejecución.

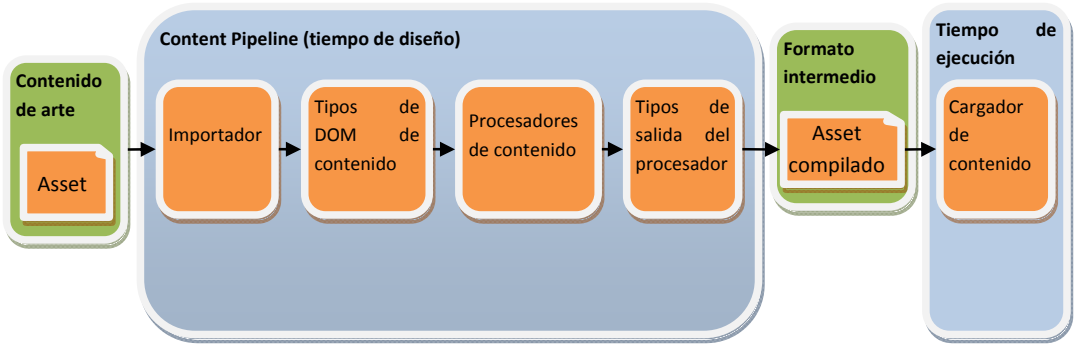


Figura 13: Proceso de transformación del contenido de arte.

Componentes en tiempo de diseño

Cuando se compila un proyecto de XNA en *Visual Studio* en un archivo ejecutable, los componentes en tiempo de diseño procesan el contenido multimedia. Estos procesos realizan la transformación inicial de un *asset* con su formato original a un objeto que la aplicación pueda usar después en la ejecución.

El DOM (Modelo de Objeto de Documento – *Document Object Model*) de contenido representa el conjunto de clases integradas que los procesadores de contenido pueden soportar.

Un importador (*importer*) convierte un asset desde su formato original a objetos DOM de contenido (*Content **D**ocument **O**bject **M**odel*) que los procesadores de contenido (*Content Processor*) puedan soportar, o bien los convierte en otra forma personalizada que un procesador de contenido personalizado pueda procesar.

Un importador típicamente convierte contenido en objetos manejables basados en el DOM de contenido, que incluye una estructura conocida para los *assets*, tales como los *assets* de modelos que contienen mallas, vértices y materiales. Actualmente el DOM provee

compatibilidad para las mallas, materiales, texturas, fuentes y animaciones. No obstante, un importador personalizado puede producir objetos personalizados para un procesador de contenido en particular. La Tabla 1 muestra una descripción de los importadores de contenido estándares de XNA.

Descripción	Extensión	Nombre	Tipo de salida
Autodesk FBX	.fbx	FbxImporter	NodeContent
Fichero .X de DirectX	.x	XImporter	NodeContent
Descripción de tipografía de sprites	.spritefont	FontDescriptionImporter	FontDescription
Efecto	.fx	EffectImporter	EffectContent
Textura	.bmp / .dds .dib /.hdr .jpeg / .jpg .pfm / .png .ppm / .tga	TextureImporter	TextureContent
Proyecto XACT	.xap	NA	NA

Tabla 1: Importadores estándares soportados por XNA Game Studio.

Un procesador de contenido recibe un *asset* de un importador y lo compila en un objeto de código administrado (manejable) que puede ser cargado y utilizado en un juego multiplataforma de XNA. Cada procesador de contenido acepta un tipo de objeto en específico.

Los objetos de código administrado creados por el procesador de contenido son serializados por el compilador del *Content Pipeline* en un archivo de formato intermedio y tiene extensión .XNB. El Escritor de Contenido (*ContentTypeWriter)* provee métodos y propiedades que permiten escribir este objeto en su respectivo archivo. XNA provee por defecto ciertos Escritores de Contenido para los formatos de assets más utilizados, y para formatos no soportados es necesario crear nuevos Escritores de Contenido para cada nuevo formato.

El formato del archivo intermedio es único y sólo puede ser utilizado por las librerías del *framework* de XNA.

Componentes en tiempo de ejecución

Los componentes en tiempo de ejecución del *Content Pipeline* reciben el archivo .XNB de cada *asset* creado por los componentes en tiempo de diseño.

El cargador de contenido se encarga de cargar el archivo .XNB al espacio de memoria del juego para que pueda ser utilizado. Deserializa el archivo de formato intermedio, convirtiéndolo en un objeto y cargándolo en memoria.

Cuando un juego necesita un objeto de código controlado de un *asset*, el programador utiliza el ***ContentManager*** para invocar al cargador de contenido. Esta etapa del proceso del *Content Pipeline* ocurre en tiempo de ejecución, teniendo en cuenta que todos los *assets* han sido compilados y almacenados en disco. El Lector de Contenido (*ContentTypeReader*) realiza la lectura de un tipo de objeto desde un archivo .XNB. La Tabla 2 muestra una descripción de los procesadores de contenido estándares de XNA.

Descripción	Nombre	Tipo de entrada	Tipo de salida
Descripción de tipografía de sprites	FontDescriptionProcessor	FontDescription	SpriteFontContent
Efecto	EffectProcessor	EffectContent	CompiledEffect
Modelo	ModelProcessor	NodeContent	ModelContent
Proyecto XACT	NA	NA	NA
Sin procesamiento requerido	PassThroughProcessor	Object	Object
Textura	TextureProcessor	TextureContent	TextureContent
Textura de tipografía de sprites	FontTextureProcessor	TextureContent	SpriteFontContent

Tabla 2: Procesadores estándares soportados por XNA Game Studio.

2.2.4 Capa Juegos

La capa Juegos es la primera capa dentro del modelo de capas. Aquí es donde los programadores comienzan a construir un videojuego, codificando tanto las estructuras básicas como la lógica del mismo.

Esta capa consta de los siguientes elementos:

- **Código:** Es el código del videojuego a desarrollar.
- **Kits de inicio:** Son un conjunto de juegos completos o casi completos que incluyen tanto código fuente como elementos de arte. Con esto el programador puede fácilmente modificar el código e incluso agregar nuevos componentes, para ser compilado y ejecutado. Cada kit viene con su propia documentación, incluyendo las técnicas utilizadas y algunas sugerencias para crear modificaciones. Éstos sirven como base para la creación de juegos, incluso para saber qué es posible lograr con XNA.
- **Contenido:** Es el contenido de arte que utiliza el juego a desarrollar.
- **Componentes:** Son los componentes creados por el programador o por una comunidad de programadores, basados en el modelo de componentes de XNA.

Estructura básica de la clase *Game*

La clase ***Game*** de XNA es una base sólida para construir un juego. Esta clase implementa el ciclo principal del juego, el cual provee no sólo la ventana en donde se desplegará el videojuego, sino también provee métodos sobrecargables que facilitan la comunicación entre el juego y el sistema operativo, como se ilustra en la Figura 14.

En principio el ciclo principal de un juego consiste en una serie de iteraciones que son invocadas constantemente hasta que el juego finalice su ejecución. En XNA, el ciclo de juego consiste en dos métodos: **Update** y **Draw**.

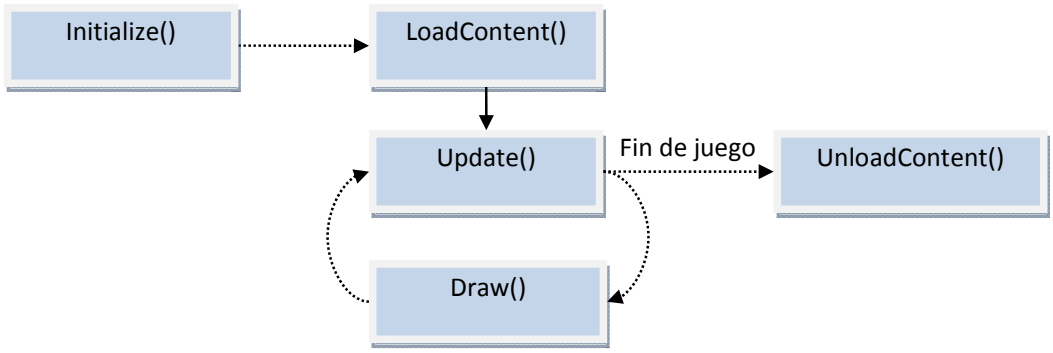


Figura 14: Ciclo de vida de un juego en XNA.

Cuando se crea un proyecto de XNA se crea una clase que deriva de **Game**. Esta nueva clase necesita sobrecargar los métodos **Update**, **Draw**, **Initialize, LoadContent,** y **UnloadContent**.

- El método **Update** es responsable del manejo de la lógica del juego, cualquier lógica que afecte su jugabilidad.
- El método **Draw** es responsable de dibujar en cada cuadro las entidades gráficas pertenecientes a la escena.
- El método **Initialize** es responsable de inicializar el juego antes de ejecutarse el primer cuadro. Permite realizar la inicialización que necesite antes de comenzar a ejecutarse. Aquí es donde se puede consultar cualquier servicio y cargar cualquier contenido no gráfico. El dispositivo gráfico estará creado en este punto y puede ser accedido desde aquí para inicializar otros objetos que dependan de él.
- El método **LoadContent** es invocado después del método **Initialize** e invocado una vez por cada juego, así como cada vez que el contenido gráfico tiene que volver a cargar (por ejemplo, si el dispositivo gráfico se reinicia después que el jugador cambia la configuración de la pantalla). Es el momento propicio para cargar contenidos multimedia y contenidos requeridos, incluyendo imágenes, modelos, sonidos. Al finalizar su invocación, la clase que deriva de **Game** iniciará el ciclo de juego.
- El método **UnloadContent** es invocado una vez que finaliza el ciclo de juego y es el momento oportuno para liberar todo el contenido cargado en el método **LoadContent**

que requiere de ser manejado especialmente. Por lo general, XNA se encargará de toda la colección de basura, pero si se ha modificado la memoria en un objeto que requiere un manejo especial, el método **UnloadContent** le permite manejar dicho objeto.

CAPÍTULO III. Consola Xbox 360

Este capítulo describe todo lo referente a la consola de videojuegos Xbox 360, antecedentes de creación, su arquitectura y especificaciones técnicas.

La Xbox 360 (ver Figura 15) de Microsoft es la primera consola de videojuegos de última generación. Históricamente, las implementaciones de arquitectura y diseño han dado grandes saltos en el rendimiento del sistema, aproximadamente a intervalos de cinco años.

La Xbox 360 es la sucesora directa de la Xbox y compite actualmente contra la PlayStation 3 de Sony y la Wii de Nintendo como parte de esta generación. Fue desarrollada en colaboración con IBM (*International Business Machines*) y ATI (*ATI Technologies Inc.*). Sus principales características son su CPU basado en una *PowerPC*[4] y su GPU[5] que soporta la tecnología de Shaders Unificados.

Figura 15: La consola Xbox 360 con un control de mando inalámbrico.

La arquitectura de hardware de esta consola está destinada a las cargas de trabajo de consolas de videojuegos. El núcleo implementa el objetivo del diseñador de productos para proporcionar a los desarrolladores de juegos una plataforma de hardware para poner en práctica sus ambiciones de juego de la nueva generación.

[4] PowerPC: Es el nombre original de la arquitectura de computadoras de tipo RISC desarrollada por IBM.
[5] GPU (*Graphics Processing Unit*): Unidad de Procesamiento Gráfico.

Cuenta con un CPU personalizado de IBM basado en una *PowerPC* y un GPU diseñado por ATI, basado en la familia ATI Radeon R500, con una memoria integrada desarrollada por NEC Corporation. Dispone de una interfaz SATA para conectar un disco duro, una tarjeta de red, memoria RAM y un sistema de entrada/salida creado por SIS (*Silicon Integrated Systems*) con soporte para controles cableados e inalámbricos compatibles con la nueva versión de Windows.

Un requerimiento clave de la nueva generación de consolas está implementada en el Xbox 360: soporte de video de alta definición, con resoluciones de salida bajo los estándares 720p, 1080i y 1080p, con formato panorámico 16:9[6]. La Xbox 360 está especialmente pensada para ser usada con televisores HDTV de alta resolución.

En lo referente al software, se basa en los mismos APIs en que se fundamenta la consola Xbox (DirectX, PIX[7], XACT). Además incorpora la tecnología de Microsoft *XNA Game Studio* mencionada anteriormente. El Xbox 360 utiliza DirectX 9.0 con soporte parcial a las funcionalidades del Shader Model 3.0.

3.1 Aplicaciones desarrolladas sobre la plataforma Xbox 360

A continuación se describen brevemente algunos trabajos realizados en el desarrollo de aplicaciones de propósito general (aplicaciones distintas a videojuegos) que han de ser ejecutadas en la plataforma Xbox 360.

3.1.1 Herramienta de revisión en entornos utilizando XNA y la consola Xbox 360

En la planificación y diseño para la construcción de entornos, las técnicas actuales de comunicación de diseño prescinden de información importante que es crucial para la comprensión de la audiencia. Investigaciones anteriores [17][18] en Entornos Virtuales muestran cómo las capacidades de visualización pueden proporcionar un lenguaje común para entender el diseño de ciudades y pueblos en general. Los urbanistas y los entornos construidos en general, tienen la oportunidad de beneficiarse de las técnicas de diseño visual que ofrece el equipo y la industria de los videojuegos. Entre las técnicas implementadas se encuentra el manejo y mapeo de texturas, técnicas de iluminación, el uso del mando de control y visualización 3D interactiva en tiempo real.

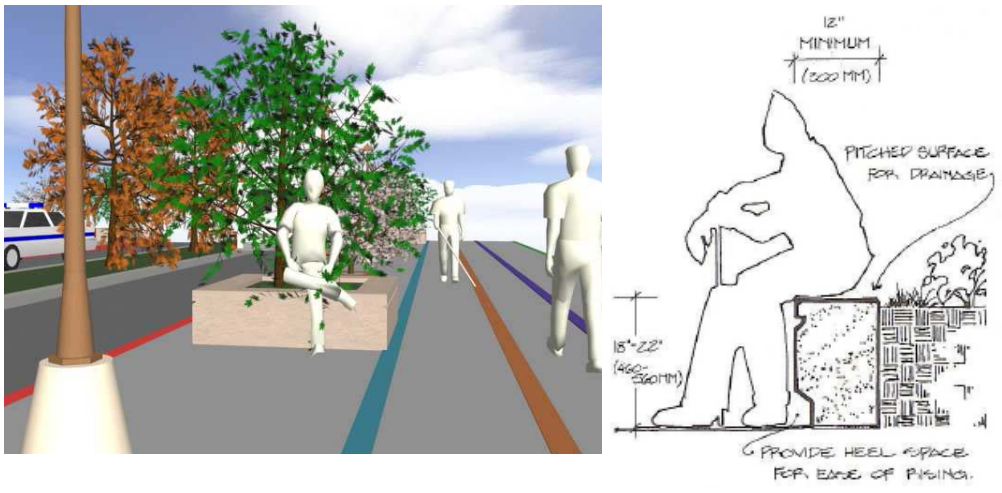
O'Keeffe y Shiratuddin [19] proponen un prototipo de herramienta de Entorno Virtual desarrollado para ayudar a comunicar y aclarar el diseño y su información, usando un motor de juego 3D para desplegar el entorno virtual en la consola Xbox 360. Este prototipo puede ser utilizado para mostrar el resultado de conceptos de diseño y los principios de diseño que

[6] Formato panorámico 16:9: Se refiere al formato de imagen estandarizado, normalmente usado hoy en día en monitores y/o televisores de pantalla ancha.
[7] PIX: Herramienta para la depuración de s*haders*.

afectan a la construcción, materiales de diseño y componentes de construcción estructurales que se aplicarán en el nuevo diseño y la construcción de un futuro campus, específicamente en la Universidad del Sur de Mississippi (USM).

El prototipo propone un estándar basado en las directrices de LEED (*Leadership in Energy and Environmental Design*) de los Estados Unidos de América. LEED es un sistema internacionalmente reconocido para la certificación de construcciones, principalmente diseñado para que una construcción o comunidad utilice ciertas estrategias destinadas a mejorar el rendimiento de vida bajo ciertas métricas.

Esta herramienta de Entornos Virtuales se utilizó principalmente para modelar algunas de las calles pertenecientes a esta universidad basándose en las directrices de LEED como estándar para la construcción de entornos. Para ello, se utiliza la consola Xbox como dispositivo de despliegue e interacción con los diseñadores y urbanistas. En la Figura 16 se ilustran los resultados obtenidos con este prototipo.



(a) (b)

Figura 16: (a) Una captura usando el prototipo de herramienta para diseñar una calle. (b) Guía para el diseño de vías peatonales [19].

3.1.2 Propagación de la luz usando un volumen en CryEngine3

Kaplanyan [20] introduce una nueva técnica para la aproximación de la iluminación global difusa en tiempo real. Como la iluminación global difusa requiere de mucho cómputo, es usualmente implementado como soluciones pre-calculadas.

Este trabajo muestra una solución dinámica usando volúmenes de radiación de SH (*Spherical Harmonic*s) para la aproximación finita de elementos de los campos de luz, rendering inyectivo volumétrico basado en puntos y un nuevo enfoque de la propagación de radiación iterativa.

Esta técnica no requiere de ninguna etapa de pre-procesamiento y es totalmente compatible con iluminación dinámica y objetos con sus materiales, la cual permite ser integrada dentro de un motor tan complejo como el motor *CryEngine* versión 3 desarrollado por *Crytek.*

Esta implementación demuestra que es posible usar esta solución de manera eficiente, incluso con la generación actual de consolas de videojuegos (PlayStation 3 y Xbox 360). Es suficiente con calcular un solo rebote de luz indirecta para presentar veracidad visual, incluso para películas de alta calidad. De tal modo, esta investigación no toma en cuenta múltiples rebotes de luz debido a su alta complejidad computacional.

La Figura 17 ilustra la diferencia visual que se percibe al utilizar la técnica de propagación de la luz mediante la utilización de un volumen implementada en el motor de *CryEngine*3.

(a) (b)

Figura 17: Propagación de la luz a través de un volumen. En la imagen a se despliega la escena sin utilizar la técnica, mientras que en la imagen b se utiliza la propagación de la luz a través de un volumen [20].

La Figura 18 muestra el efecto de radiosidad logrado con el método de propagación de la luz utilizando un volumen.

Figura 18: Aproximación de la radiosidad en un ambiente al aire libre [20].

3.1.3 Visibilidad dinámica para escenas 3D

Con la creciente complejidad e interactividad en el mundo de los videojuegos, la necesidad de una visibilidad dinámica eficiente se hace cada vez más importante.

El trabajo de Stephen Hill y Daniel Collin [21], cubre dos enfoques complementarios para determinar la visibilidad, que han sido usados en juegos de alta categoría tanto para plataforma Xbox, como PS3 y PC.

Así desarrollan dos soluciones para la visibilidad de objetos en una escena, basados en las técnicas utilizadas en los juegos *Splinter Cell Conviction* y *Battlefield: Bad Company* 1 y 2. Estas soluciones deberían ser de propósito general, ya que ellas son capaces de manejar completamente entornos dinámicos con gran cantidad de objetos en escena, con bajo costo, implementaciones sencillas y con pocas modificaciones a los assets originales.

El primer enfoque fue llamado *Conviction Solution*, en el que en principio, solo se despliegan los objetos definidos por el artista en etapa de diseño como oclusores potenciales. Estos oclusores son a menudo un mallado estructural que constituye una versión simplificada de la unión de varios objetos cercanos en la escena, aunque es posible que el artista pueda etiquetar también cualquier objeto aislado como un oclusor potencial.

A continuación se crea una pirámide jerárquica de profundidad o HZB por sus siglas en inglés (*Hierarchical Z-Buffer*) a partir del buffer de profundidad que contiene los oclusores potenciales, para evaluar qué objetos de la escena han de ser desplegados.

La pirámide ha de tener una base de $2^n x 2^{n-1}$ y cada vez que se sube un nivel se decrementa en uno el valor del exponente para cada una de las potencias de dos. En la práctica se usó una base de 512x256 para la pirámide, lo cual representa aproximadamente un cuarto de la resolución de la cámara principal bajo la modalidad de único jugador.

Para crear cada uno de los niveles de la pirámide se toma el valor mayor de profundidad por cada cuatro nodos téxels, este valor será el asignado al nodo correspondiente para el nivel superior como se observa en la Figura 19.

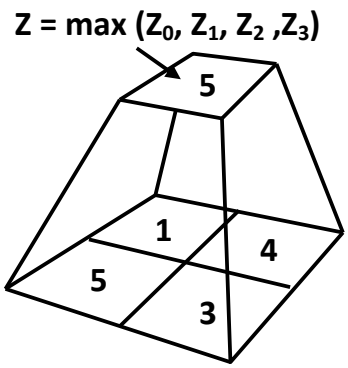


Figura 19: Generación sucesiva de niveles en una HZB.

Finalmente se verifica cuáles serán los objetos a ser desplegados a partir de las colisiones entre el *bounding box* de cada objeto y la pirámide de profundidad. El nivel de detalle se evalúa dependiendo de la cercanía y el tamaño del objeto.

El segundo enfoque es el *Battlefield Solution*, que ya no se basa en desplegar únicamente los objetos de la escena que no serán ocluidos, sino en omitir detalles de estos objetos ocluidos luego de haber sido desplegados. Las características principales de este enfoque son las siguientes:

- Debe ser completamente dinámico, debido a que el entorno puede moverse, rotarse, e incluso deformarse.
- Debe tener poco "overhead" en el GPU.
- Debe ser accesible desde el CPU, para omitir ciertos aspectos de los objetos ocluidos, como la animación de personajes.

Proceso:

Configuración de los oclusores: Esta etapa itera sobre el conjunto de oclusores pertenecientes a la escena, siguiendo los siguientes pasos:

1. El conjunto de mallados está ubicado en una memoria compartida, para que cada hilo de ejecución pueda tomar un mallado utilizando semáforos.
2. Cada hilo verifica si el mallado está ubicado dentro de la pirámide truncada (*frustum*). Si no lo está, se regresa al paso 1.
3. Si el mallado está completamente dentro de la pirámide truncada de visualización, sus triángulos son agregados en un arreglo ubicado dentro de una sección crítica.
4. Si el mallado no está completamente dentro de la pirámide truncada, sus triángulos son cortados antes de ser agregados.

Cada hilo de ejecución tiene su propio z-búfer (de dimensión 256x114) y despliega sus mallados generados anteriormente. El primer hilo en ejecución copiará el contenido de su z-

búfer en la memoria principal, y luego los próximos hilos harán una mezcla de sus z-búferes con la copia en la memoria principal.

A continuación se verifican cuáles mallados están dentro de la pirámide truncada, realizando una comparación entre el volumen delimitador de cada mallado en la escena, y se construye un arreglo resultante.

Finalmente se realiza la prueba de visibilidad mediante el z-búfer. Primero el *bounding box* del mallado es proyectado en el plano imagen y se calcula su área en 2D, si ésta es más pequeña que un cierto valor, el mallado es descartado. Luego sobre esta área se determina la distancia mínima existente entre cada punto de esta área y la cámara, para posteriormente realizar la prueba de oclusión. La prueba de oclusión se realiza computando la distancia mínima con el búfer de profundidad.

CAPÍTULO IV. Implementación

4.1 Recursos de Hardware/Software para plataforma Windows

Se utilizó la versión de XNA 3.1 debido a que no restringe el tamaño máximo del volumen que se pueda cargar sino por el hardware subyacente.

Para el despliegue de volúmenes e interfaz gráfica se utilizó una tarjeta de video compatible con DirectX 9.0 o superior para sistemas Windows, y capaz de soportar Shader Model 2.0 o superior.

Para el desarrollo de *shaders* se utilizó el lenguaje HLSL (*High Level Shading Language*) por razones de compatibilidad con DirectX 9.0. Para la versión Windows se tuvo que utilizar una versión más reciente del compilador de *shaders* debido a que el compilador no soportaba la cantidad de instrucciones necesarias para el despliegue de las técnicas de *RayCasting* y Planos Alineados al *Viewport* en el procesador de píxeles.

4.2 Recursos de Hardware/Software para plataforma Xbox

La aplicación que ha de ejecutarse en el Xbox 360 es la misma desarrollada para PC bajo la versión de XNA 3.1 y los shaders desarrollados en lenguaje HLSL. Sin embargo, para poder ejecutar una aplicación en el Xbox es necesario tener una membrecía de XNA Creators Club y conexión al servicio de Xbox Live.

Durante las pruebas de conexión con Xbox Live se presentó un fallo en la misma, generado por el siguiente error: “El tipo de NAT es estricto (o moderado). Es posible que las personas conectadas a redes con este tipo de NAT no puedan unirse a determinados juegos o escuchar a otros jugadores mientras están jugando en línea.” La visualización en pantalla mostraba lo que podemos apreciar en la Figura 20.

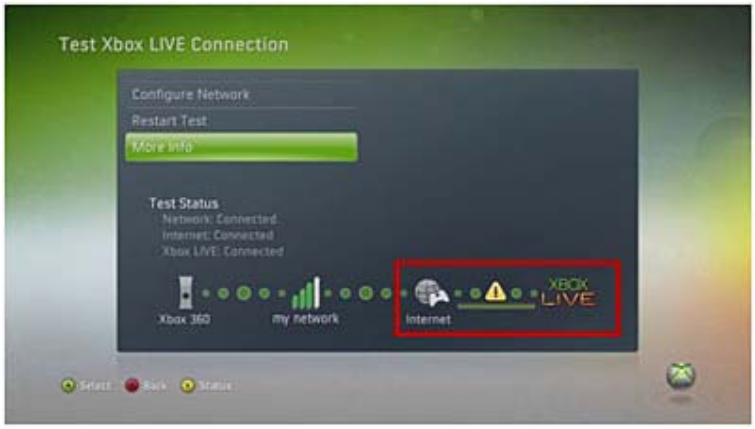


Figura 20: Fallo de NAT en la prueba de conexión a Xbox Live.

NAT (*Network Address Translation*) es el encargado de transformar una dirección de red privada a una pública para poder navegar en la web. El funcionamiento puede ser de varias formas; se tiene una sola dirección pública y múltiples privadas. Generalmente un enrutador tiene el protocolo NAT embebido, sin embargo el enrutador puede aprovechar los protocolos (por ejemplo DHCP) y solicitar una dirección pública y configurarlo a una red, todo lo que está dentro de ella tendrá direcciones privadas.

El problema era que el hardware de red o el firewall de la universidad bloqueaba la comunicación con los servidores Xbox LIVE, al mantener cerrados determinados puertos necesarios para establecer la conexión.

Para poder conectar la consola de Xbox 360 al Xbox Live se necesitan abiertos los siguientes puertos:

- Puerto 88 (UDP)
- Puerto 3074 (UDP y TCP)
- Puerto 53 (UDP y TCP)
- Puerto 80 (TCP)

A nivel de Hardware, la Tabla 3 muestra las especificaciones técnicas de la arquitectura de la consola Xbox 360 [22].

Procesador	• Tiene 3 núcleos de procesamiento simétricos que corren a 3,2 GHz cada uno • 2 hilos de ejecución por núcleo • Contiene 1 MB de caché Nivel 2 • Realiza 9 millardos de operaciones de producto punto
GPU	• Diseño personalizado desarrollado por ATI • Funciona a 500 MHz • Capaz de procesar 500 millones de triángulos por segundo • Contiene 10 MB de tipo EDRAM • Soporta la arquitectura de *Shaders* Unificados • 48 millardos de operaciones de *Shaders* por segundo
Memoria RAM	• 512 MB de tipo GDDR3 • Funciona a 700MHz
Ancho de banda de la Memoria RAM	• Tiene un ancho de banda de 22.4 GB/seg en el bus de memoria • Tiene un ancho de banda de 256 GB/seg hacia la memoria EDRAM • Tiene un ancho de banda de 21.6 GB/seg en el bus frontal
Rendimiento de operaciones punto-flotante	• 1 Tera[8] de operaciones punto flotante por segundo
Almacenamiento	• Soporta un disco duro externo de 20 GB, 60 GB o 120 GB • Contiene una unidad de DVD-ROM que soporta discos DVD de 12X doble capa

[8] Tera: Unidad métrica que denota un factor de 10^{12}

Entrada/Salida	• Soporta hasta 4 controles de mando inalámbricos • Contiene 3 puertos USB versión 2.0 • Contiene 2 ranuras para memorias extraíbles
Internet	• Listo para usar las características de *Xbox Live* • Se conecta a través de un puerto Ethernet • Tiene soporte para Wi-Fi bajo los estándares 802.11 A, B y G (es opcional) • Tiene soporte para cámara de video
Soporte de contenido digital	• Soporta los siguientes formatos: DVD-Video, DVD-ROM, DVD-R/RW, DVD+R/RW, CD-DA, CD-ROM, CD-R, CD-RW, WMA CD y CDs de MP3 • Permite copiar música al disco duro • Permite crear listas de reproducción personalizadas para cada juego
Alta definición (HD)	• Soporta resoluciones de 720p, 1080i y 1080p[9] con anti-aliasing • Tiene tanto salida digital de video como salida RCA
Audio	• Todos los juegos soportan Dolby Digital 5.1 • Tiene una frecuencia máxima de 48 KHz con una resolución de hasta 32 bits

Tabla 3: Especificaciones técnicas del Xbox 360.

En la Figura 21, se muestra la arquitectura interna de una consola de Xbox 360.

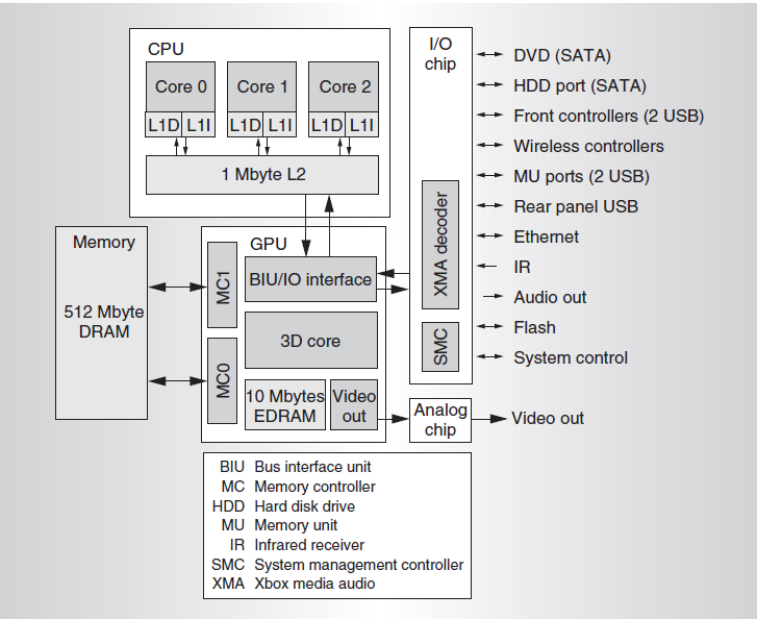


Figura 21: Diagrama de bloques del sistema de la consola Xbox 360 [22].

[9] 1080p: Es soportado por las consolas creadas a partir de la fecha 2008.

4.3 Implementación de la interfaz gráfica

Debido a que el *framework* de XNA no posee clases que faciliten la construcción de interfaces gráficas, se desarrollaron diferentes elementos gráficos para representar la información y acciones disponibles en la aplicación, permitiendo la interacción entre el usuario y el dispositivo de despliegue.

Para una mejor comprensión de la interfaz se implementaron algunas animaciones que hicieran más amigables las transiciones entre menús. Para ello fue necesario utilizar una estructura de datos jerárquica que permitiera el movimiento conjunto de todos los elementos de cada menú. La Figura 22 nos permite ver en detalle dicha estructura.

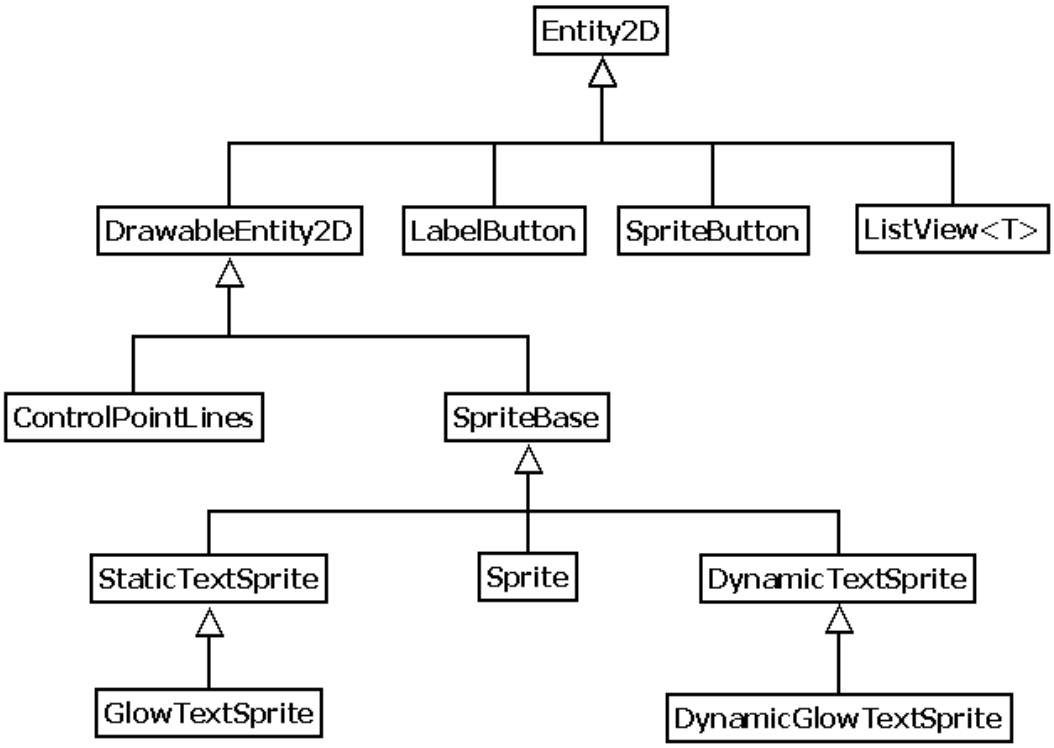


Figura 22: Jerarquía de clases.

El elemento gráfico básico de esta jerarquía es ***Entity2D***, el cual posee atributos que permiten ubicar dicho elemento en un espacio 2D, como por ejemplo la traslación o rotación, tal como se puede apreciar en la Figura 23. Luego se tiene el elemento ***DrawableEntity2D*** que representa una entidad básica en un espacio 2D capaz de desplegar contenido en pantalla.

La clase ***Entity2D*** contiene un atributo *position* que representa su posición en un espacio 2D. El atributo *scale* permite definir un vector de escalamiento. El atributo *rotation* representa el ángulo de rotación de la entidad expresado en radianes. Una lista de objetos de tipo ***Entity2D*** es almacenada en el atributo *children,* que representa una colección de objetos hijos.

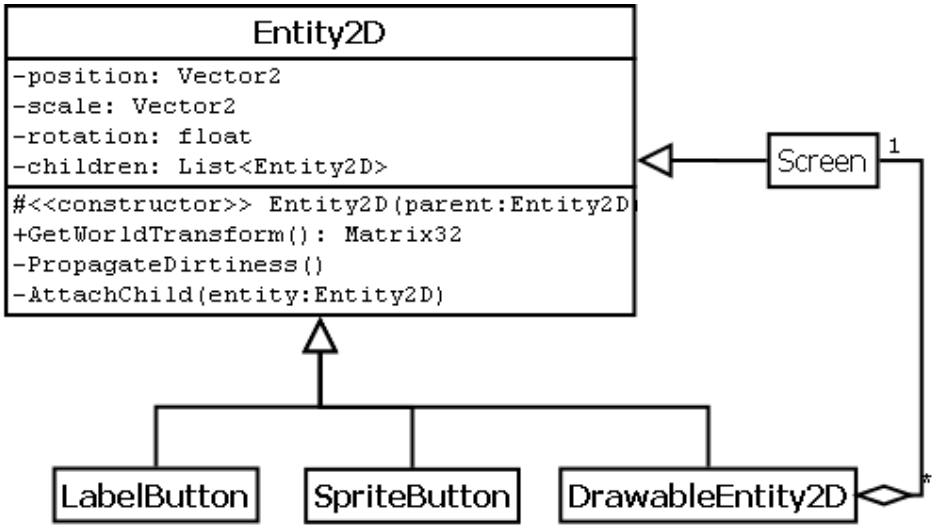


Figura 23: Diagrama de clases de la clase Entity2D.

Los atributos *position*, *scale* y *rotation* están dados en coordenadas locales. Para obtener sus coordenadas absolutas (coordenadas de mundo) es necesario aplicar una serie de transformaciones hasta encontrar la transformada del nodo raíz en la jerarquía de objetos. Esto es logrado en el método GetWorldTransform(). Cabe destacar que esta transformación calculada es almacenada en un atributo y permanecerá con el mismo valor hasta que exista un cambio en su transformada. El método PropagateDirtiness() propaga la existencia de un cambio de transformada hacia los nodos hojas, para que éstos puedan recalcular sus transformadas absolutas.

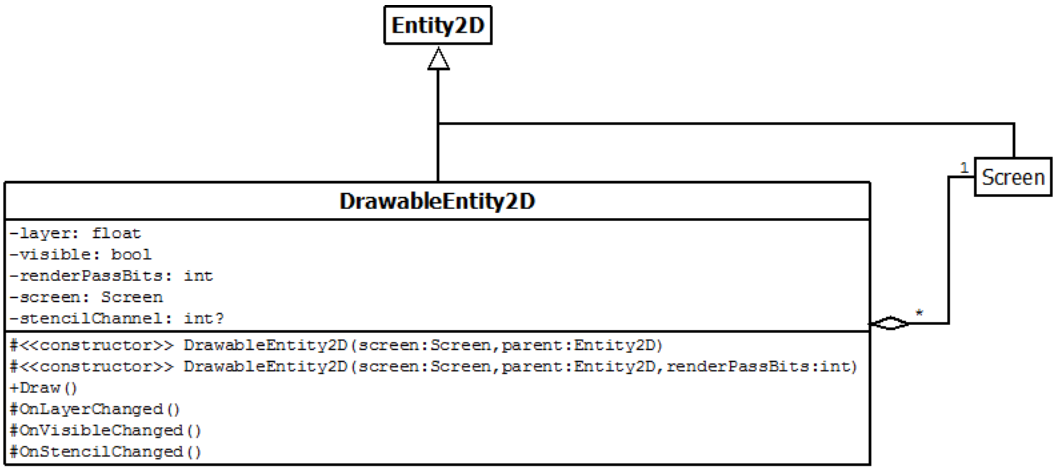


Figura 24: Diagrama de clases de la clase DrawableEntity2D.

La Figura 24 ilustra en detalle los elementos que conforman la clase ***DrawableEntity2D***, la cual contiene un atributo *layer* que es utilizado para posicionar la entidad en "capas", con valores reales entre 0 y 1, donde indican la capa más cercana y lejana respectivamente. Además cada entidad tiene un atributo *renderPassBits* donde almacena una bandera de bits que indica en qué pase de renderizado será desplegado. El atributo *visible* establece si la entidad será o no desplegada. *stencilChannel* indica el canal de esténcil utilizado, puede tener valor nulo si no se desea aplicar una plantilla de esténcil.

Esta clase es una clase abstracta por lo que cada clase concreta debe implementar el método Draw(). Los métodos OnLayerChanged(), OnVisibleChanged() y OnStencilChanged() son métodos virtuales y son invocados cuando existen cambios de valores en los atributos *layer*, *visible* y *stencilChannel* respectivamente. Pueden ser sobrescritos por las clases derivadas.

Esta clase puede desplegar dos tipos de contenidos:

- **ControlPointLines:** Permite dibujar líneas indicando dos puntos en el espacio de imagen. Cada punto tiene asociado una posición y un color. Esta clase es utilizada para dibujar las líneas de control en la función de transferencia en una sola llamada a la tarjeta gráfica (ver Figura 25).

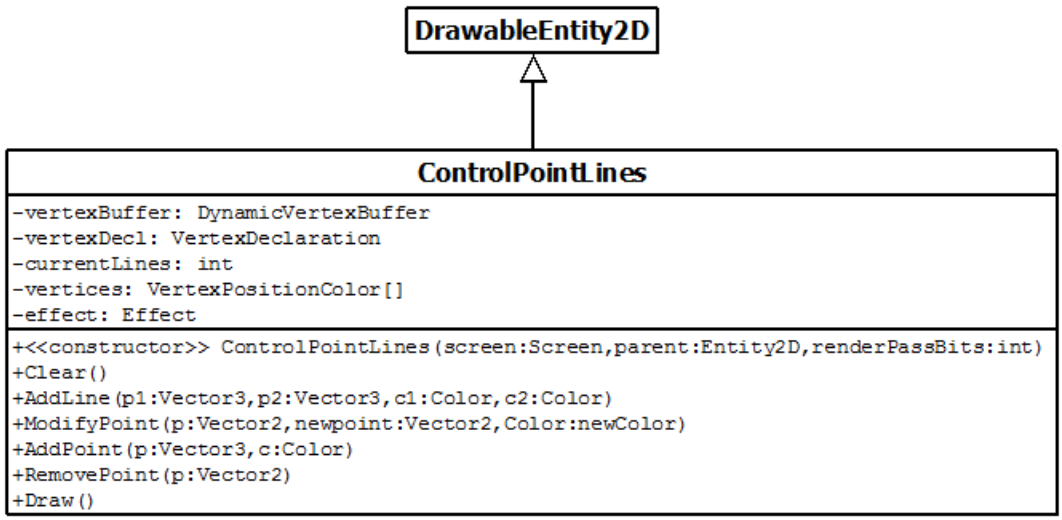


Figura 25: Diagrama de clases de la clase ControlPointLines.

El búfer dinámico *vertexBuffer* contiene la lista de vértices de cada línea, y es llenada o modificada a medida que el usuario agrega o modifica los puntos de control. El atributo *currentLines* indica la cantidad de líneas agregadas, *vertices* representa una copia de la lista de vértices almacenada en la CPU, *effect* es el efecto a ser aplicado a las líneas. El método Clear() elimina todas las líneas agregadas, AddLine() agrega una línea dado dos puntos y sus colores, ModifyPoint() modifica un punto existente indicando su nueva coordenada y su nuevo color, AddPoint() agrega un nuevo punto dado su posición y color como parámetros, RemovePoint() elimina un punto dado su posición y Draw() despliega las líneas existentes usando el efecto y el búfer de vértices.

- **SpriteBase:** Clase abstracta que permite agrupar los distintos tipos de *Sprites* que han de ser desplegados. Su estructura puede ser apreciada en la Figura 26. Utiliza una clase de XNA llamada *SpriteBatch* que permite desplegar varios objetos de tipo *Sprite* de manera simultánea. Contiene dos atributos: *origin* indica el punto de pivote y *color* indica el color de tinte.

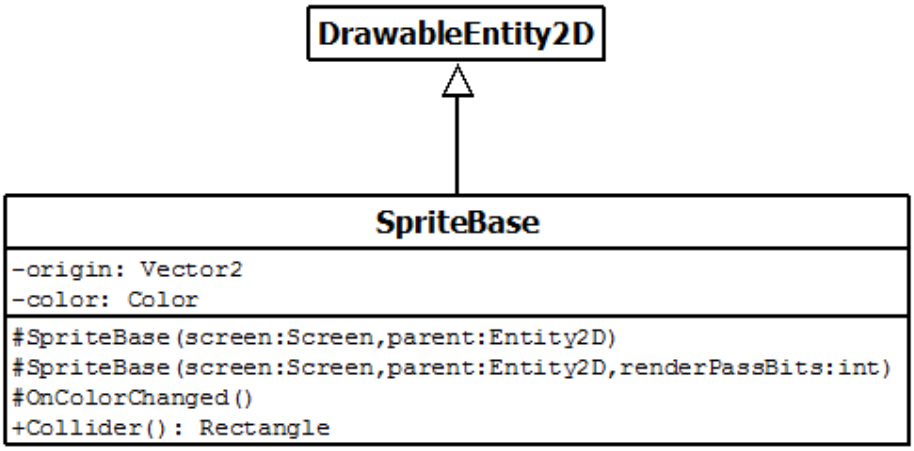


Figura 26: Diagrama de clases de la clase SpriteBase.

Finalmente se definen los elementos concretos que permiten desplegar imágenes y textos en dos dimensiones, los cuales son especificaciones de la clase *SpriteBase*:

- **Sprite:** Representa una textura 2D en el espacio de pantalla (ver Figura 27). El atributo *texture* representa la textura cargada en la GPU. El método Draw() permite desplegar la textura en pantalla especificando su textura, posición, color de tinte, rotación, escalamiento, entre otras opciones. La propiedad Size() retorna el tamaño en píxeles de la textura, en coordenadas locales.

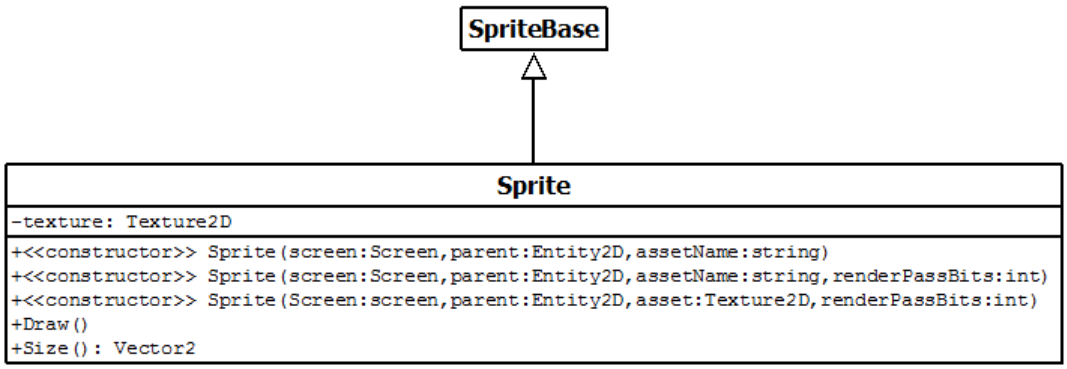


Figura 27: Diagrama de clases de la clase Sprite.

- **StaticTextSprite:** Representa un texto estático que ha de ser desplegado en pantalla. En la Figura 28 se puede apreciar en detalle los métodos y atributos que la conforman. El atributo *text* representa la cadena de caracteres, *font* contiene la descripción de la fuente de letra a utilizar, *alignment* es un enumerado que indica la alineación del texto de manera horizontal (Izquierda, Derecha, Centrada), *size* indica el tamaño en píxeles del texto. El método Draw() permite desplegar el texto en pantalla con la configuración asociada, UpdateAlignment() actualiza la posición del texto de acuerdo a su alineación, los métodos OnTextChanged() y OnHorizontalAlignmentChanged() son métodos virtuales y son invocados cuando existen cambios de valores en los atributos text y *alignment* respectivamente.

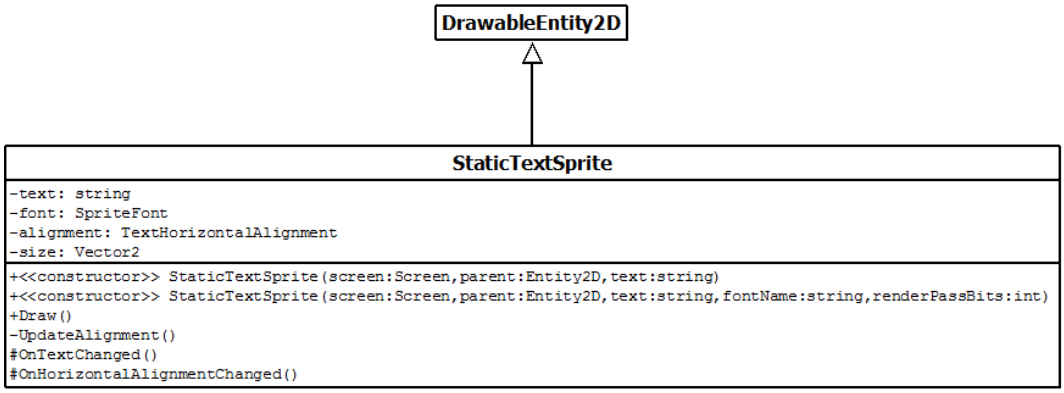


Figura 28: Diagrama de clases de la clase StaticTextSprite.

- **GlowTextSprite:** Hereda de la clase *StaticTextSprite* y le añade un borde de color al texto que ha de ser desplegado para resaltarlo. Esto es utilizado para indicar cuál es la opción seleccionada en un menú.

  Al construir un objeto de este tipo se añaden instancias de texto estático, con el mismo texto, localizados con un desplazamiento relativo al texto original como lo ilustra la Figura 29, todo esto para simular el efecto de brillo o resalte. Estas nuevas instancias son agregadas dentro de la colección de hijos.

  La lista *points* almacena una lista de colores que será utilizada para asignar un color a las instancias de texto de acuerdo a su desplazamiento. La función GetColor() devuelve el color de acuerdo al desplazamiento pasado por parámetro, ShowGlow() y HideGlow() muestran u ocultan el texto de resalte, los métodos OnLayerChanged(), OnVisibleChanged(), OnTextChanged() y OnHorizontalAlignmentChanged() reajustan las propiedades *layer*, *visible*, *text* y *alignment* de cada instancia, respectivamente.

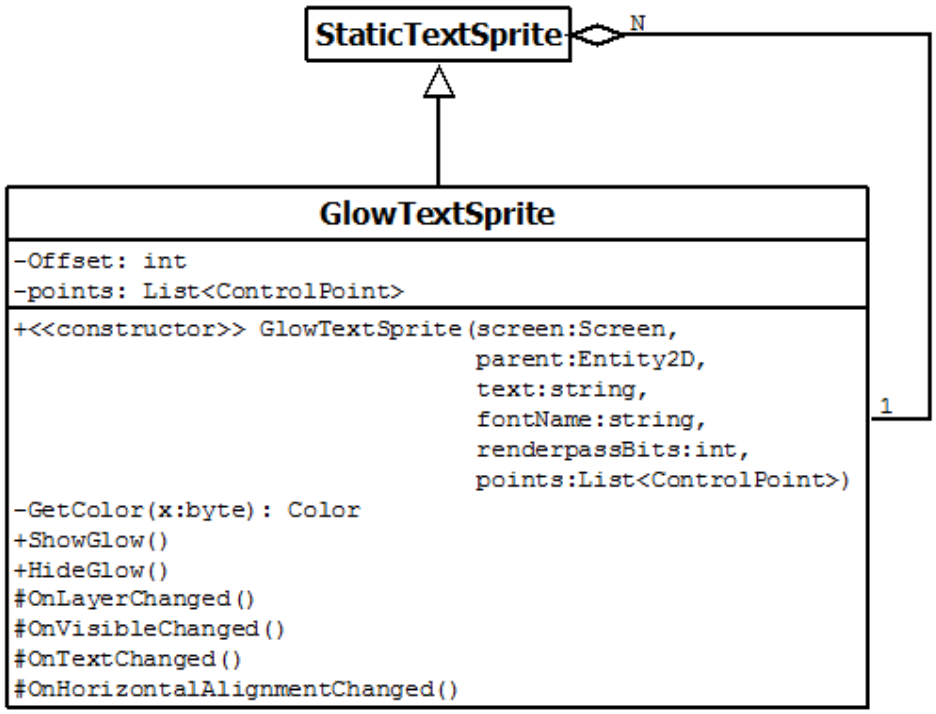


Figura 29: Diagrama de clases de la clase GlowTextSprite.

- **DynamicTextSprite:** Representa un texto que ha de ser desplegado en pantalla y que puede cambiar en el tiempo de manera eficiente. Cada vez que se gestiona una cadena de caracteres, se crea un nuevo objeto de cadena en la memoria, que requiere una nueva asignación de espacio para el objeto. La clase S*tringBuilder* se puede utilizar para modificar una cadena sin crear un nuevo objeto.

  El atributo *builder* almacena la cadena de texto dinámica, *font* contiene la descripción de la fuente de letra a utilizar, *alignment* es un enumerado que indica la alineación del texto de manera horizontal (Izquierda, Derecha, Centrada), *size* indica el tamaño en píxeles del texto. El método Clear() elimina la cadena dinámica, Append() agrega un elemento al final de la cadena, CopyTextFrom() copia el contenido de una cadena dinámica en el objeto actual, Insert() inserta un elemento en una posición arbitraria de la cadena, UpdateSize() actualiza la variable *size*, EraseAt() elimina un carácter dado una posición, Draw() permite desplegar el texto en pantalla con la configuración asociada, los métodos OnTextChanged() y OnHorizontalAlignmentChanged() son métodos virtuales y son invocados cuando existen cambios de valores en la cadena y en la alineación respectivamente (ver Figura 30).

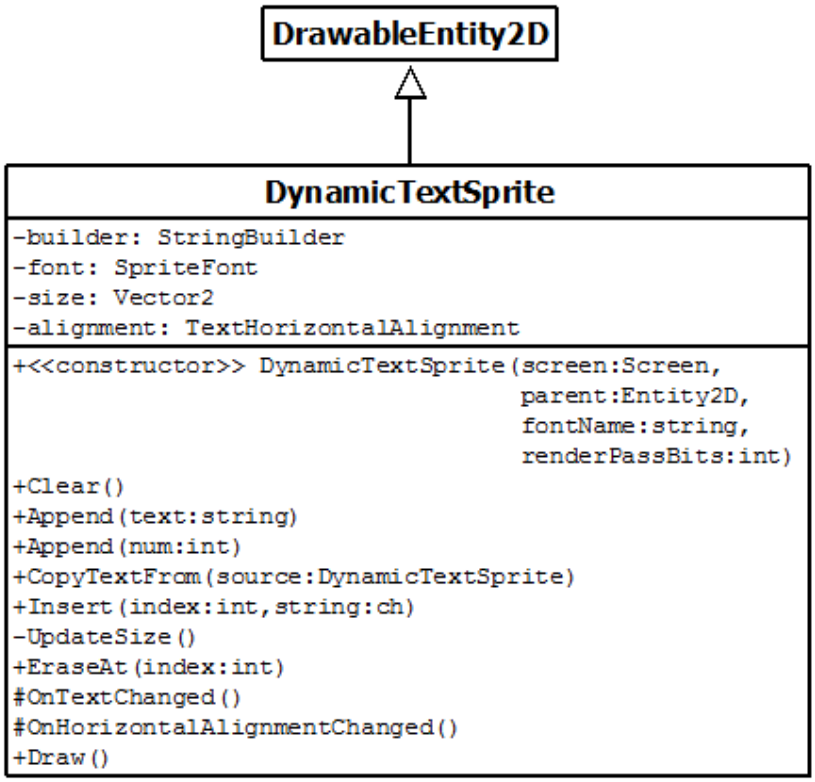


Figura 30: Diagrama de clases de la clase DynamicTextSprite.

- **DynamicGlowTextSprite:** Hereda de la clase *DynamicTextSprite* y le añade un borde de color a un texto dinámico que ha de ser desplegado para resaltarlo. Esto es utilizado para indicar cuál es la opción seleccionada en un menú.

  Al construir un objeto de este tipo se añaden instancias de texto dinámico, con el mismo texto, localizados con un desplazamiento relativo al texto original, todo esto para simular el efecto de brillo o resalte. Estas nuevas instancias son agregadas dentro de la colección de hijos. La Figura 31 ilustra en detalle los elementos que forman parte de esta clase.

  La lista *points* almacena una lista de colores que será utilizada para asignar un color a las instancias de texto de acuerdo a su desplazamiento. La función GetColor() devuelve el color de acuerdo al desplazamiento pasado por parámetro, ShowGlow() y HideGlow() muestran u ocultan el texto de resalte, los métodos OnLayerChanged(), OnVisibleChanged(), OnTextChanged() y OnHorizontalAlignmentChanged() reajustan la capa, la visibilidad, el texto y la alineación de cada instancia, respectivamente.

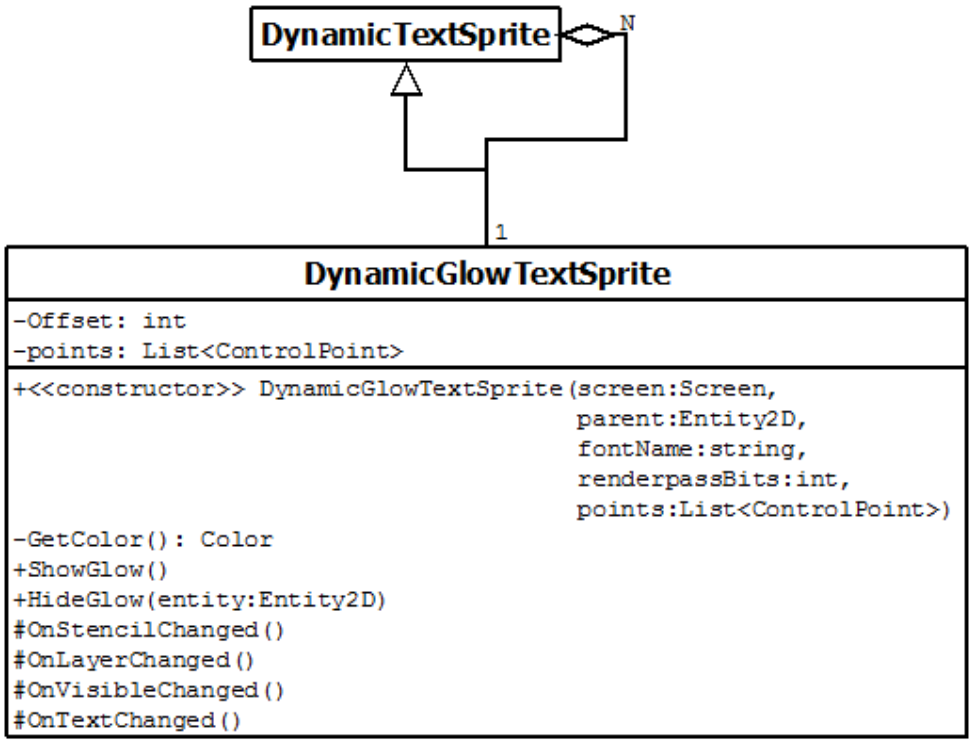


Figura 31: Diagrama de clases de la clase DynamicGlowTextSprite.

- **Listview:** Representa una lista de elementos que permiten visualizar un texto y/o una imagen en forma de ícono asociadas. Cada elemento puede contener información adicional que no es desplegada pero que puede ser utilizada para otras operaciones. Este tipo de dato es parametrizado con el uso de plantillas. La Figura 32 ilustra la estructura de esta clase.

  Las propiedades *Enabled* y *Visible* permiten habilitar/deshabilitar y mostrar/ocultar la lista, respectivamente, las variables *ítems* y *icons* permiten almacenar las listas de texto e imagen respectivamente, *itemsInfo* contiene la información adicional de cada elemento, *stencilmask* representa la máscara de esténcil a utilizar. Este *stencilmask* permite que sólo sean visibles los elementos que se encuentren dentro del área del *ListView*. El atributo *background* representa la textura de fondo, *SelectedItem* indica el elemento seleccionado con el ratón.

  El método AddItem() permite agregar un elemento especificando su nombre, dato adicional e ícono, Clear() permite eliminar los elementos agregados.

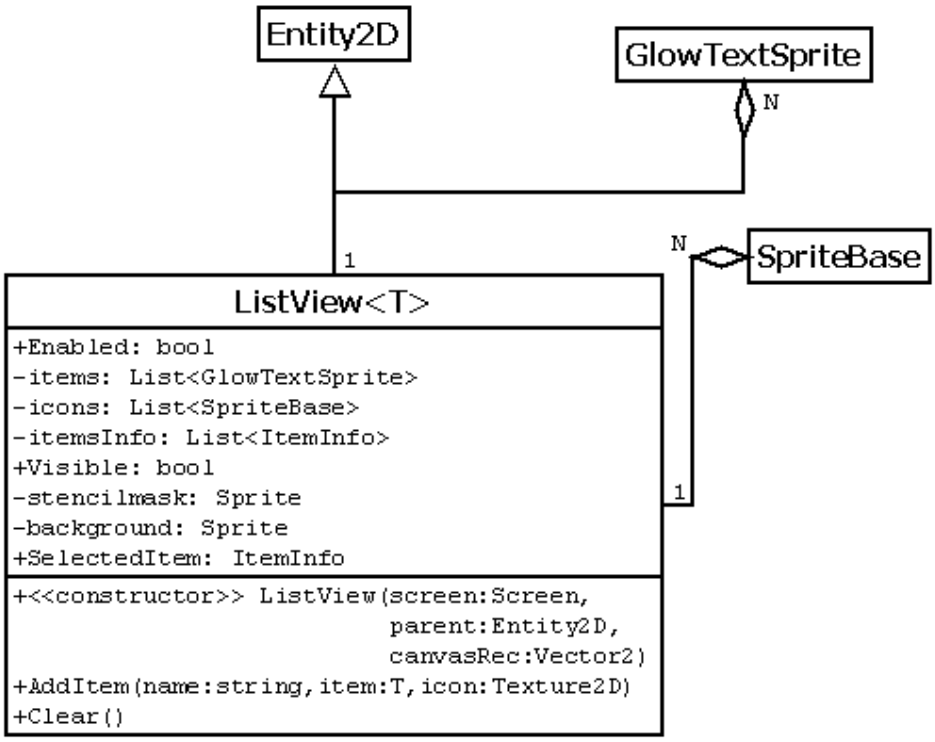


Figura 32: Diagrama de clases de la clase ListView.

4.4 Implementación de las clases de despliegue gráfico

XNA emplea un conjunto de clases que controlan el despliegue gráfico utilizando DirectX y adaptando todos los parámetros necesarios inherentes a la plataforma en la que la aplicación está siendo ejecutada.

Para la gestión de los elementos que deben dibujarse, así como el control del orden en que han de ser desplegados, fue necesario crear una serie de clases que manejaran todo esto como se puede apreciar en la Figura 33.

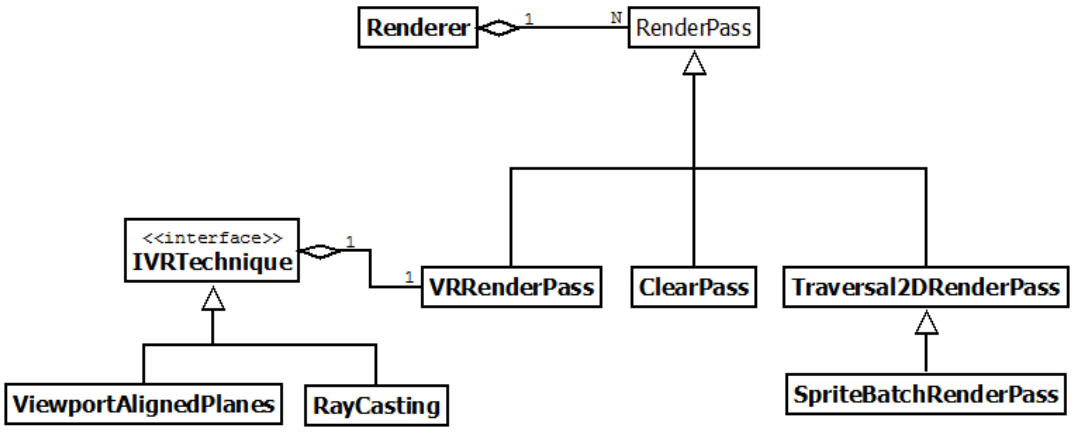


Figura 33: Diagrama de clases de despliegue.

Las clases que realizan el despliegue son las siguientes:

- **Renderer**: Gestiona todo el renderizado mediante pases de renderizado. El método AddRenderPass() agrega un pase a la lista, Render() realiza el proceso de despliegue al iterando sobre la lista de pases, basado en la configuración de los pases de renderizado. La Figura 34 permite ver la relación entre esta clase y la clase RenderPass.

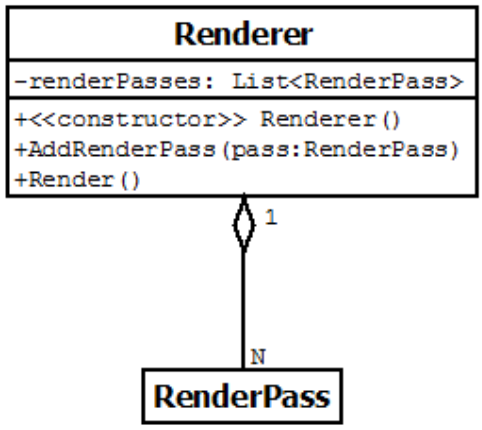


Figura 34: Diagrama de clases de la clase Renderer.

- **RenderPass**: Un pase de renderizado define los estados de renderizado que serán asignados en la tarjeta gráfica antes que una lista arbitraria de elementos gráficos sea renderizada. Cada pase tiene además definido una bandera de bits que representa la bandera de los elementos gráficos a ser renderizados. La clase RenderPass gestiona todos los pases de renderizados que son necesarios para el despliegue final, en la Figura 35 podemos apreciar todos los elementos que intervienen en el proceso.

  El atributo *renderPassBits* indica la máscara de bits, *rasterizer* representa el estado del rasterizador, el cual determina cómo debe ser convertida en píxeles la geometría a ser desplegada, *depthStencil* representa el estado del mapa de profundidad y el esténcil, *blend* representa el estado de mezcla de colores. El método SetRenderState() configura los estados de despliegue en la tarjeta gráfica, Draw() es un método abstracto que debe ser implementado por las clases concretas y es invocado en el renderizador (Renderer).

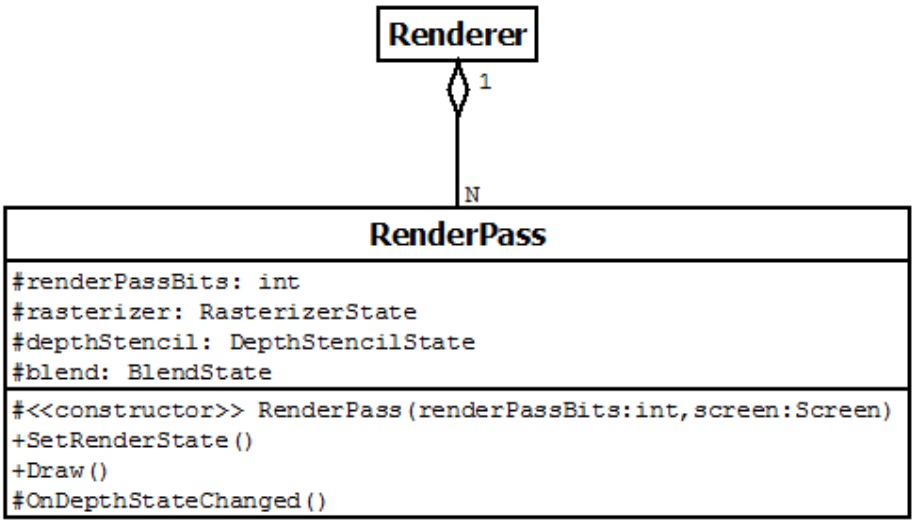


Figura 35: Diagrama de clases de la clase RenderPass.

Las clases concretas son las siguientes:

- **ClearPass**: Permite limpiar los búferes de profundidad, el mapa de colores y el mapa de esténcil. Las variables *Color*, *Depth* y *Stencil* indican los valores iniciales de color, profundidad y esténcil (ver Figura 36).

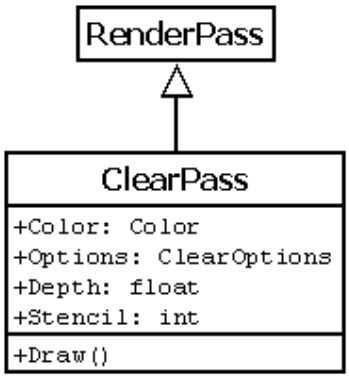


Figura 36: Diagrama de clases de la clase ClearPass.

- **Traversal2DRenderPass**: Permite renderizar elementos gráficos de 2D estableciendo un recorrido en que van a ser desplegados dichos elementos. Utiliza un manejador de interfaz gráfica (*IUIManager*) que se encarga de hacer el recorrido de los elementos, bien sea *front-to-back* o *back-to-front* (ver Figura 37.

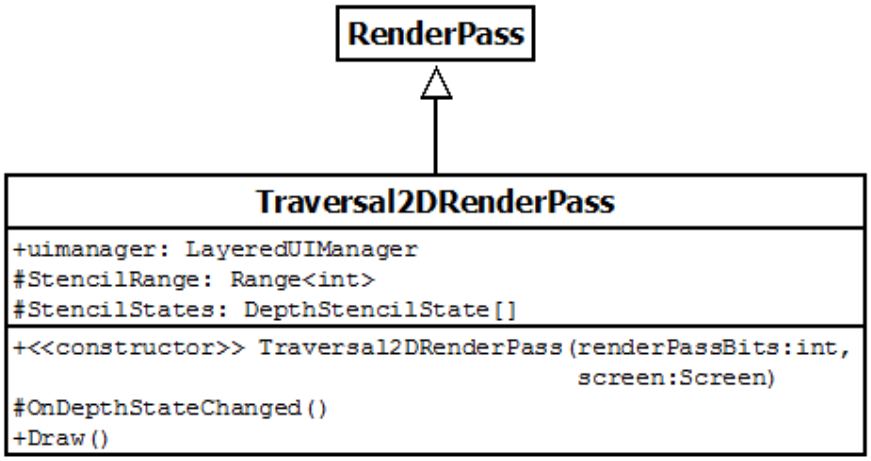


Figura 37: Diagrama de clases de la clase Traversal2DRenderPass.

- **SpriteBatchRenderPass**: Permite renderizar elementos gráficos de tipo *SpriteBase* utilizando la clase *SpriteBatch* proporcionada por XNA. En la Figura 38 se puede apreciar los elementos que conforman esta clase. El atributo *blendMode* indica el modo de mezcla de colores a utilizar por la clase *SpriteBatch, sortMode* indica el orden de despliegue de los elementos gráficos, *lastStencilGroup* permite agrupar elementos que tengan una misma máscara de esténcil. El método Draw() permite desplegar los elementos en la pantalla, DrawSingleEntity() permite configurar el esténcil para un elemento gráfico en particular.

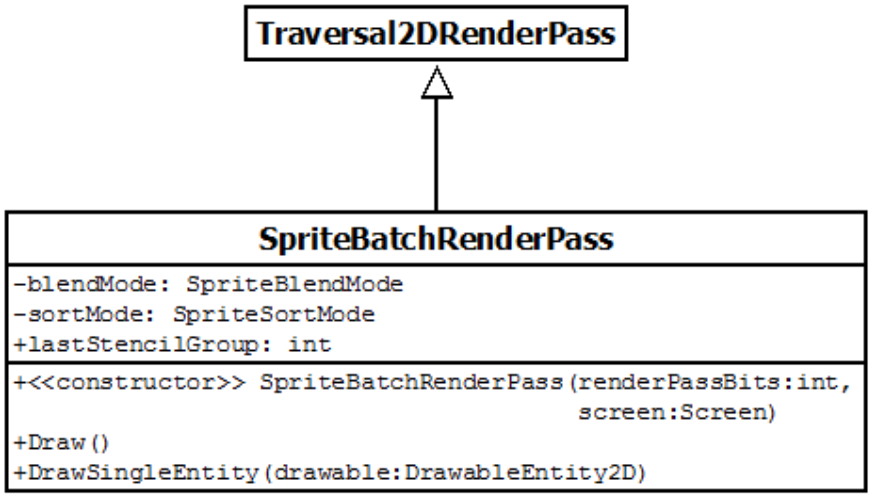


Figura 38: Diagrama de clases de la clase SpriteBatchRenderPass.

- **VRRenderPass**: Permite renderizar utilizando cualquiera de las técnicas de Despliegue de Volúmenes implementadas. El atributo *Technique* almacena la

técnica a desplegar. El método Draw() invoca al método Draw() del objeto *Technique* (ver Figura 39).

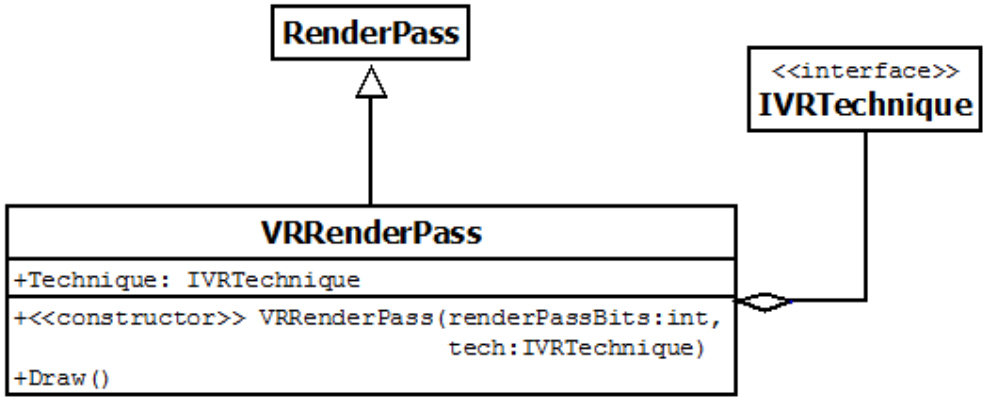


Figura 39: Diagrama de clases de la clase VRRenderPass.

4.5 Implementación del despliegue de volúmenes

Las dos técnicas de Despliegue de Volúmenes implementan la interfaz *IVRTechnique*, en donde cada clase concreta re-define los métodos y acciones para obtener el despliegue deseado. A continuación se explican las clases encargadas de hacer dichas tareas, cuya relación se ilustra en la Figura 40.

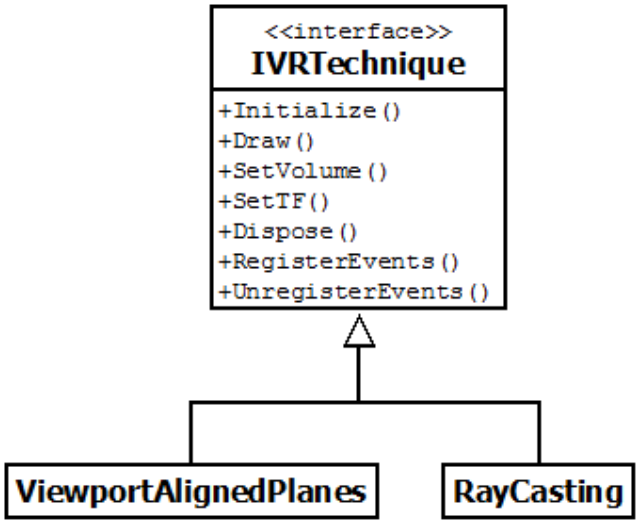


Figura 40: Diagrama de clases sobre las técnicas de despliegue de volúmenes.

Las dos clases concretas son las siguientes:

- **RayCasting:** Esta clase contiene un atributo *indexBuffer* que describe el orden de despliegue de los vértices del cubo unitario, *vertexBuffer* contiene una lista de vértices del cubo unitario, *rayCastingEffect* es el efecto utilizado para desplegar la técnica de *Raycasting*. El método SampleDistance() permite ajustar la distancia de muestreo en la travesía del rayo, Initialize() permite inicializar los búferes del cubo y cargar el efecto en la GPU; Draw() y DrawCube() permiten desplegar la técnica y el cubo unitario respectivamente, SetVolume() y SetFT() configuran el volumen y la función de transferencia en el efecto. Los métodos RegisterEvents() y UnregisterEvents() permiten gestionar los eventos de la cámara al ser rotada y movida. En la Figura 41 pueden apreciarse en detalle todos estos métodos y atributos que forman parte de la clase Ray Casting y de la clase IVRTechnique, así como la relación entre ambas.

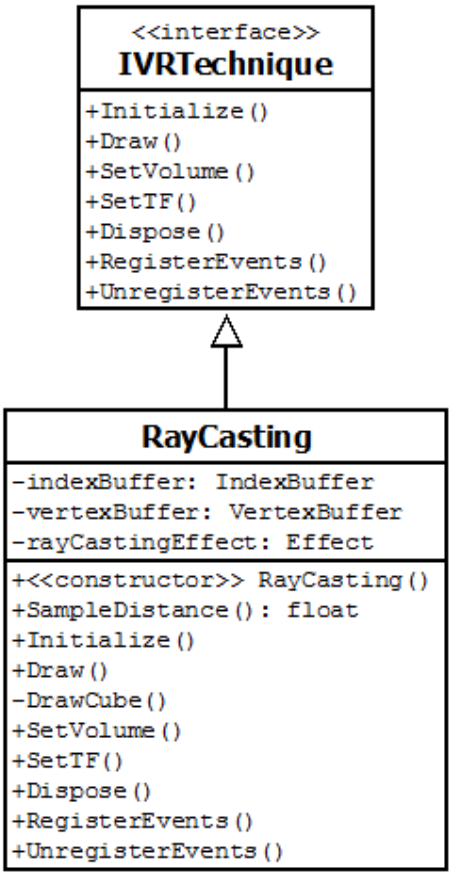


Figura 41: Diagrama de clases de la clase RayCasting.

- **ViewportAlignedPlanes:** Esta clase contiene un atributo *indexBuffer* que describe el orden de despliegue de los vértices de los cortes, *vertexBuffer* contiene una lista de vértices de los costes, *viewportEffect* es el efecto utilizado para desplegar la técnica de *Planos alineados al viewport*. El método Initialize() permite inicializar la matriz de textura de los cortes y cargar el efecto en la GPU; Draw() permite desplegar la técnica, SetVolume() y SetFT() configuran el volumen y la función de transferencia en el efecto.

Los métodos RegisterEvents() y UnregisterEvents() permiten gestionar los eventos de la cámara al ser rotada y movida. La Figura 42 ilustra la relación entre las clases IVRTechnique y ViewportAlignedPlanes, así como los elementos que forman parte de dichas clases.

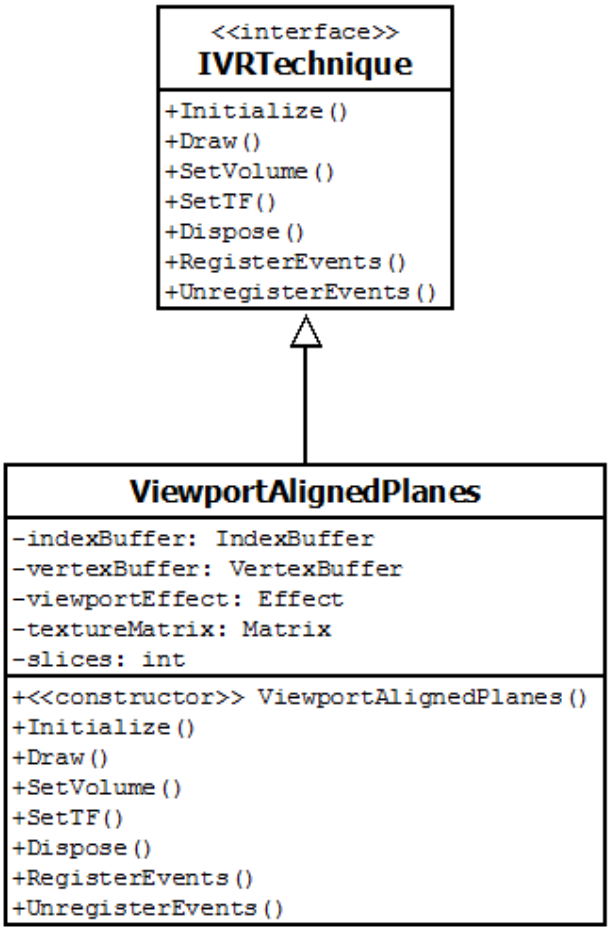


Figura 42: Diagrama de clases de la clase ViewportAlignedPlanes.

4.6 Implementación de la carga de volúmenes

La carga de volúmenes en tiempo de diseño se pudo lograr utilizando la Tubería de Contenido. XNA por defecto sólo soporta el formato DDS para texturas tridimensionales, por lo cual se extendió con dos nuevos formatos, PVM y RAW, que son los comúnmente utilizados en el Despliegue de Volúmenes.

Para cada nuevo formato se desarrolló un importador y un procesador, como se muestra en la Figura 43.

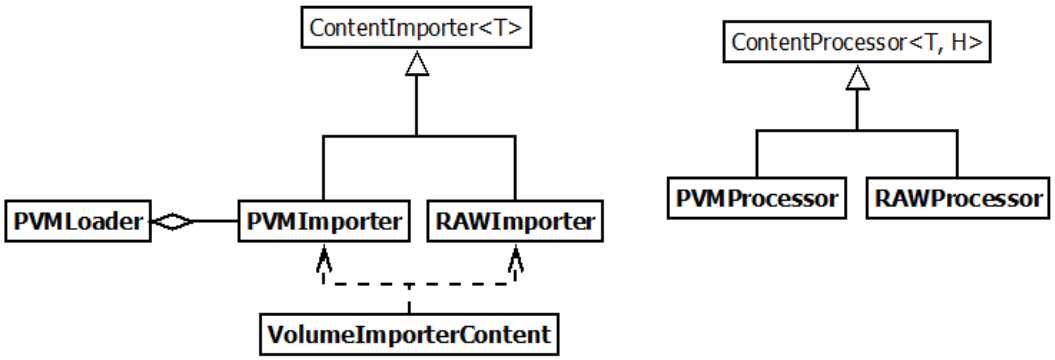


Figura 43: Diagrama de clases de los importadores y procesadores.

Los importadores *PVMImporter* (Figura 44) *y RAWImporter* retornan objetos de tipo *VolumeImporterContent* que contienen información leída desde el disco duro, como lo son el conjunto de vóxeles, las dimensiones del volumen y el vector de escalado. Es importante resaltar que para el formato RAW se deben especificar las dimensiones del volumen en el *RAWProcessor* en tiempo de diseño, para que éste pueda cargar volúmenes válidos.

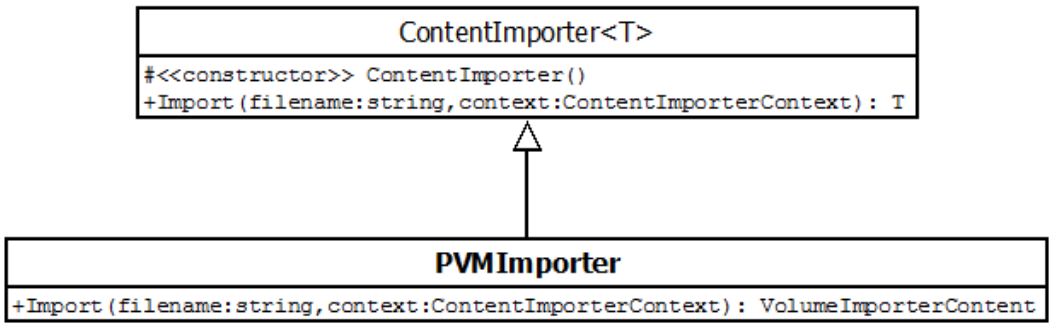


Figura 44: Diagrama de clases de la clase PVMImporter.

Los procesadores *PVMProcessor* (Figura 45) y *RAWProcessor* se encargan de procesar los vóxeles cargados en el paso anterior y generar dos texturas: una textura de tipo ***Texture3DContent*** que representa el volumen, y una textura de tipo ***Texture2DContent*** que representa el histograma de frecuencias del volumen, con un tamaño de 256x256 téxeles. Este proceso de generación de datos se puede observar en la Figura 46.

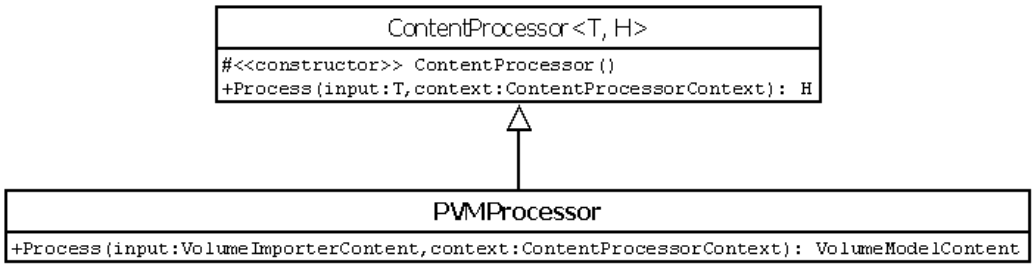


Figura 45: Diagrama de clases de la clase PVMProcessor.

La carga de volúmenes a través de la tubería de contenido está disponible en ambas plataformas. Por otra parte para cargar volúmenes sin la tubería se implementó dos nuevas maneras:

a. **Especificando una ruta de acceso**: Disponible únicamente en la plataforma Windows; es posible buscar y seleccionar un archivo de volumen almacenado en una ubicación arbitraria dentro del computador.
b. **Por medio de una conexión de red**: Es posible conectar un computador con plataforma Windows y la consola Xbox y establecer una conexión de red para transmitir hacia el Xbox volúmenes que estén almacenados en el computador.

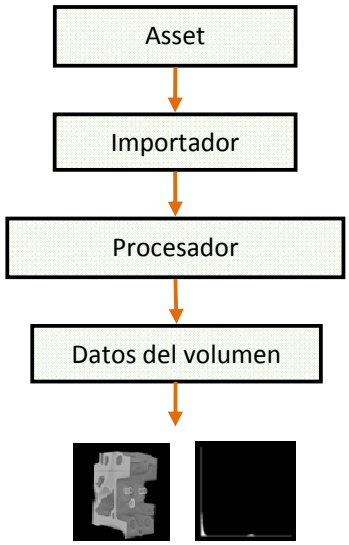


Figura 46: Proceso de transformación del volumen utilizando la tubería de contenido.

4.7 Implementación de las clases de control

A continuación se explican las clases que gestionan todo lo referente a la entrada de datos del usuario, la interfaz, la carga de datos y las configuraciones de pantalla. En la Figura 47 pueden apreciarse las relaciones entre las clases que controlan la entrada. La Figura 69 ilustra la relación existente entre las clases que gestionan todo lo referente a las configuraciones de pantalla.

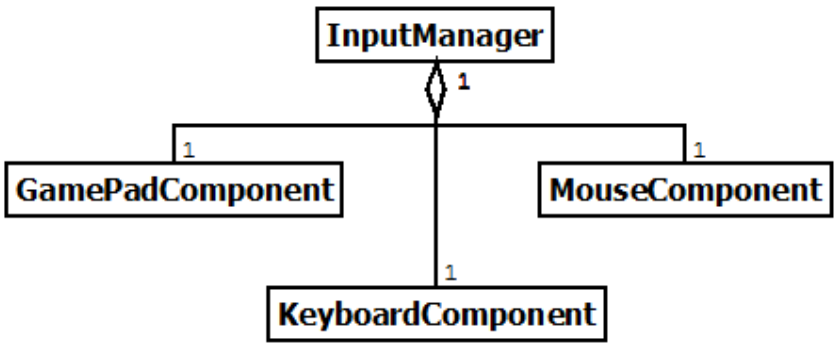


Figura 47: Diagrama de clases para el control de los dispositivos de entrada.

InputManager: Permite obtener el estado de los dispositivos de entrada mediante una interfaz unificada. Esta clase posee un constructor el cual se encarga de inicializar todos los dispositivos a gestionar. Cada dispositivo es accedido por medio de su atributo. El método *Update*() es invocado en cada cuadro de actualización y permite almacenar el estado de cada dispositivo, como lo es si un botón o tecla ha sido presionada.

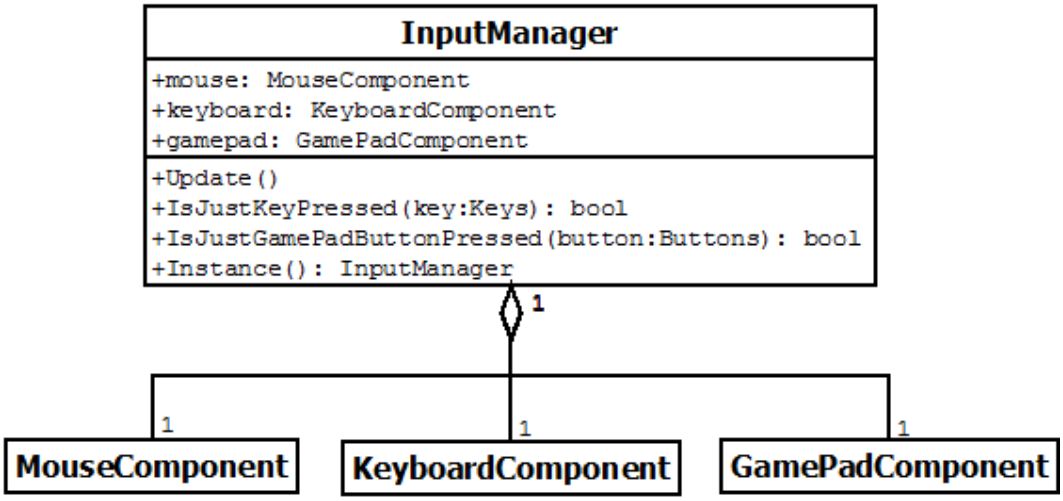


Figura 48: Diagrama de clases de la clase InputManager.

Los dispositivos soportados son el ratón (*MouseComponent*), el teclado (*KeyboardComponent*) y control de mando (*GamePadComponent*) como se ve en la Figura 48.

- **MouseComponent:** Permite conocer la posición del ratón en Windows y el estado de los botones. La implementación en la plataforma Windows es habitual, en Xbox 360 se simuló el movimiento con la palanca izquierda del controlador. Esta clase tiene su constructor por defecto, un método *Update*() se encarga de obtener el estado del ratón en el cuadro actual, las funciones *IsLeftButtonPressed*() y *IsRightButtonPressed*() determinan si el botón izquierdo y derecho han sido presionados respectivamente, las funciones *IsJustLeftButtonPressed*() y *IsRightLeftButtonPressed*() determinan si el botón izquierdo y derecho han sido presionado en el estado anterior pero no en el estado actual, la función *Speed*() devuelve el vector velocidad del ratón. Ver Figura 49.

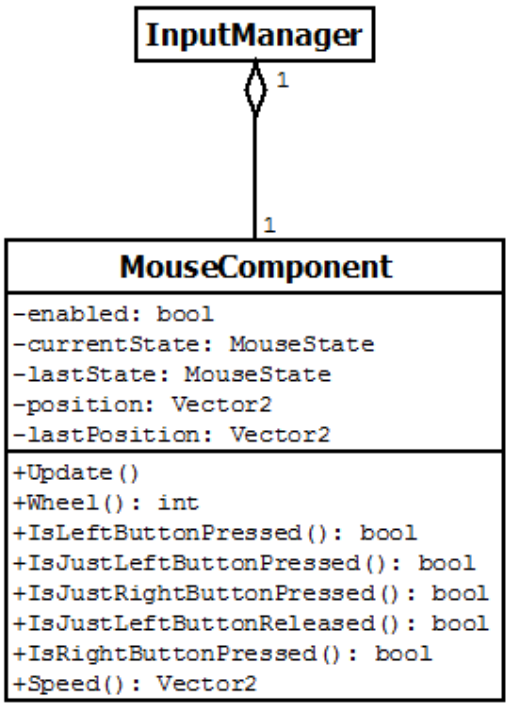


Figura 49: Diagrama de clases de la clase MouseComponent.

- **KeyboardComponent:** Permite conocer el estado del teclado, es decir, determinar si una tecla ha sido presionada o soltada. Es compatible en ambas plataformas. Esta clase contiene un constructor por defecto, la función *IsKeyDown*(key) determina determinar si la tecla 'key' ha sido presionada o no, la función *IsKeyJustPressed*(key) determina si una tecla ha sido presionada en el estado actual y no ha sido presionada en el estado anterior (ver Figura 50).

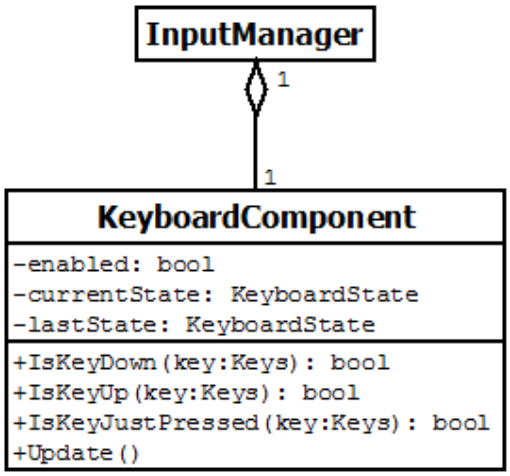


Figura 50: Diagrama de clases de la clase KeyboardComponent.

- **GamePadComponent:** Permite determinar el estado de un controlador de Xbox 360, teniendo acceso a las palancas de mando y los botones, determinar si un botón ha sido presionado o soltado, o presionado por un tiempo determinado. Es soportado para ambas plataformas y sólo permite conectar un controlador a la vez (ver Figura 51).

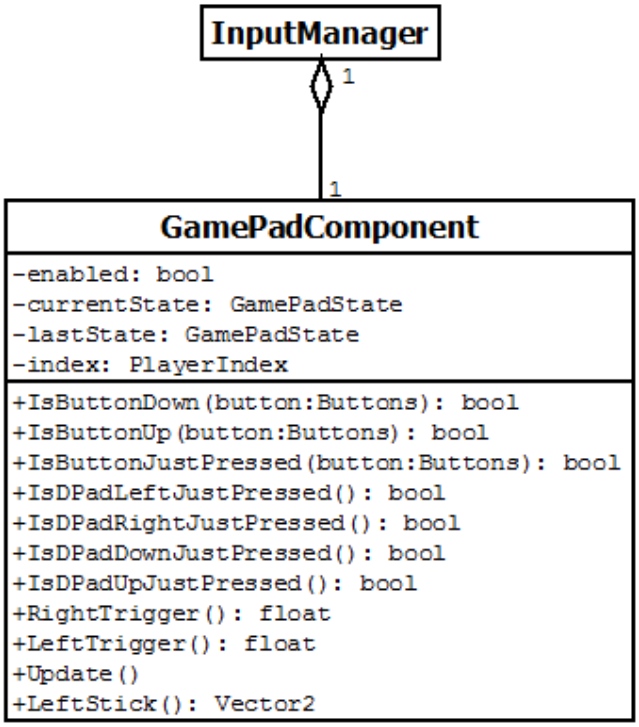


Figura 51: Diagrama de clases de la clase GamePadComponent.

ScreenManager: Permite gestionar las pantallas. Cada pantalla tiene un conjunto de elementos gráficos a ser desplegados. Permite agregar o eliminar pantallas de la pila de pantallas. La pila es simulada con una lista de arreglos. La Figura 52 muestra los elementos que componen esta clase.

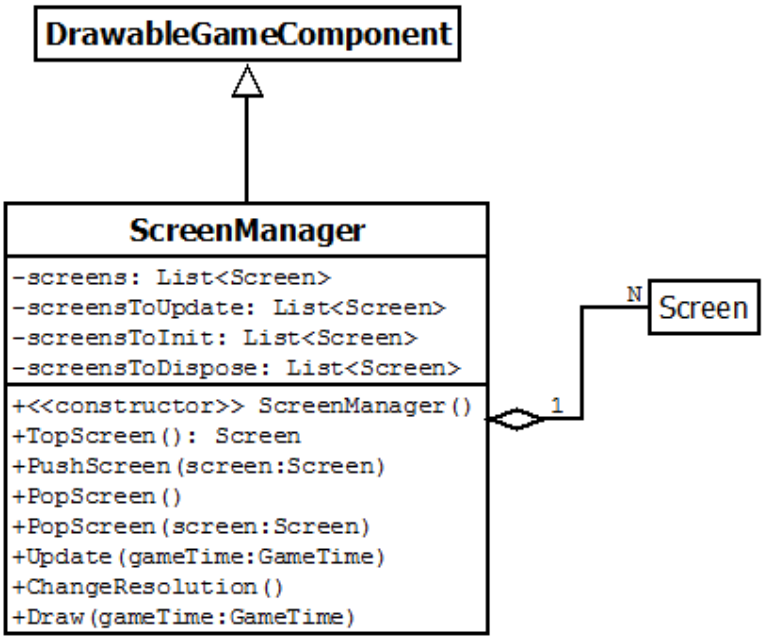


Figura 52: Diagrama de clases de la clase ScreenManager.

El atributo *screens* almacena las pantallas actuales, *screensToUpdate* mantiene una copia de la pila de pantallas y representa las pantallas a ser actualizadas dentro de un cuadro de ejecución, *screensToInit* indican las pantallas a ser inicializadas en el próximo cuadro de ejecución, *screensToDispose* indican las pantallas a ser liberadas en el próximo cuadro de ejecución. La función TopScreen() retorna la pantalla más reciente, PopScreen() elimina la pantalla más reciente, PopScreen(screen) elimina una pantalla en específico, Update() es invocado en cada cuadro de ejecución y se encarga de invocar al método Update de cada entidad agregada de cada pantalla, ChangeResolution() permite cambiar la resolución de pantalla de la aplicación y además permite reajustar todas las pantallas actuales. El método Draw() despliega todas las pantallas agregadas al manejador.

LayeredUIManager: Es un manejador de interfaz gráfica. Mantiene los elementos gráficos en una lista ordenada por capas de mayor a menor. Permite saber cuáles elementos de 2D son visibles dentro de la pantalla. Los atributos y métodos que conforman esta clase pueden apreciarse en la Figura 53.

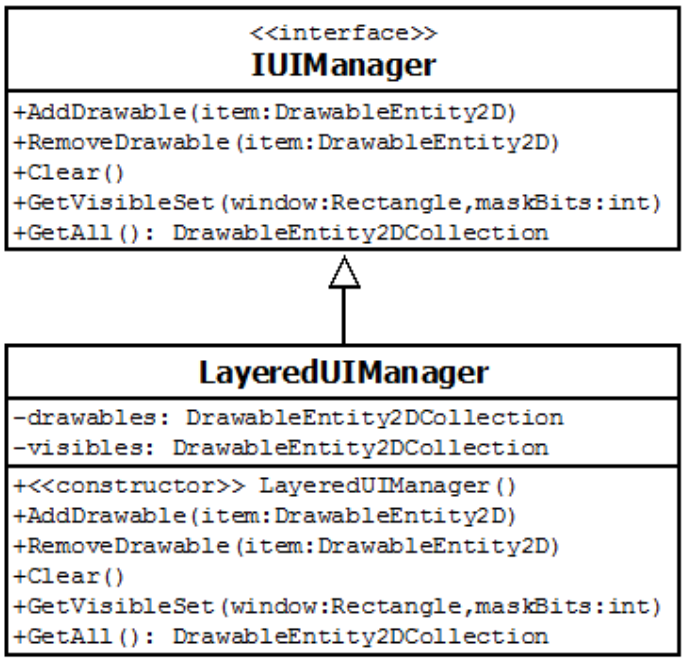


Figura 53: Diagrama de clases de la clase LayeredUIManager.

ContentTrackerManager: Es la clase responsable de mantener un monitoreo de los assets cargados mediante la tubería de contenido. Permite separar la carga de assets en nodos lógicos llamados rastreadores de contenido, que pueden ser liberados de memoria de manera independiente.

Un rastreador de contenido (*ContentTracker*, ver Figura 54) hereda de la clase *ContentManager* proporcionada por XNA, y ésta es la responsable de cargar un asset en memoria.

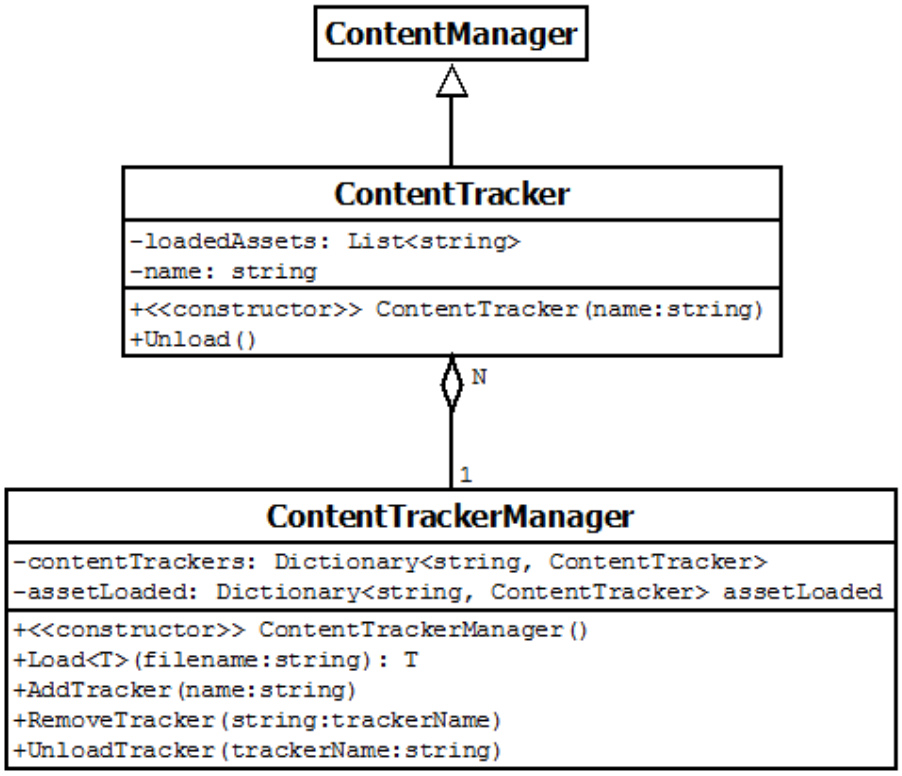


Figura 54: Diagrama de clases de la clase ContentTrackerManager.

Un objeto de tipo *ContentTrackerManager* tendrá una lista de nodos lógicos, y además se puede determinar si un asset ha sido cargado en memoria y a qué rastreador pertenece. Esto se almacena en los atributos *contentTrackers* y *assetLoaded* respectivamente.

El método Load<T>() permite cargar un asset dado su ruta por parámetro. AddTracker() y RemoveTracker() permiten agregar y eliminar un rastreador. UnloadTracker() permite liberar el contenido cargado de un rastreador en específico.

VolumeLoader: Es la clase responsable de cargar un volumen y su histograma en memoria a partir de un archivo generado por un Procesador de Contenido. Si el archivo no ha sido manipulado por un Procesador de Contenido, entonces esta clase genera las texturas correspondientes y las carga en memoria. Puede cargar sólo un volumen a la vez en memoria. Los formatos soportados son: RAW y PVM de 8 o 16 bits por vóxel. Su estructura es ilustrada en la Figura 55. Utiliza la clase *ContentTrackerManager* para gestionar la carga a través de la tubería de contenido.

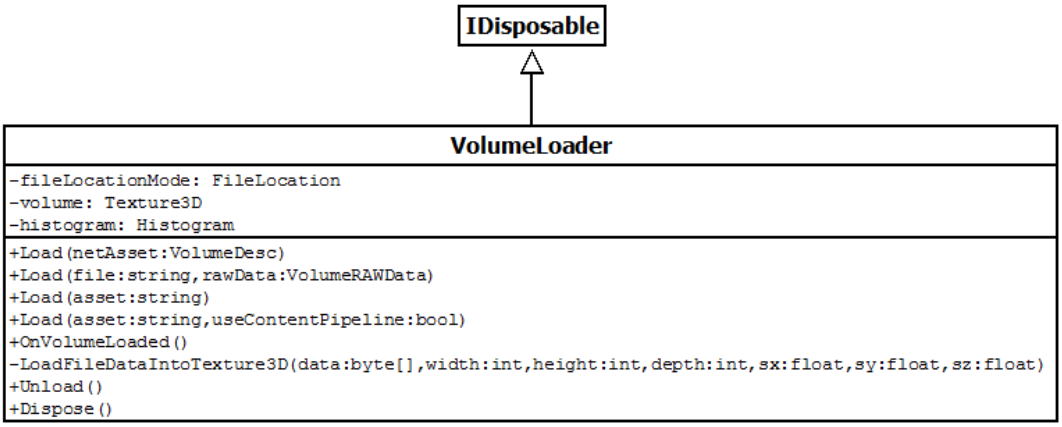


Figura 55: Diagrama de clases de la clase VolumeLoader.

El atributo *fileLocationMode* es un enumerado e indica en qué forma fue cargado el volumen (tubería de contenido, archivo ubicado en disco o mediante una conexión de red), *volumen* representa la textura 3D del volumen, *histogram* contiene el histograma de frecuencias del volumen.

El método Load() es un método polimórfico que se encarga de cargar un volumen en memoria de la tarjeta de video, independientemente de si proviene de la red, desde el disco duro, si fue procesado por la tubería de contenido. OnVolumeLoaded() permite levantar un evento cuando el volumen ya ha sido cargado, LoadFileDataIntoTexture3D() permite cargar un volumen de manera manual, dado el arreglo de bytes y la información asociada. Unload() libera el volumen de la memoria y Dispose() libera los recursos utilizados.

Screen: Clase abstracta que representa la ventana donde se despliegan entidades en 2D y 3D. La Figura 56 muestra los métodos y atributos que componen esta clase. Existen 2 tipos de pantalla:

- **Modal:** La pantalla más reciente es actualizada, mientras que las pantallas que estén por debajo de ésta quedan sin actualizarse. Este modo es utilizado para los diálogos.
- **Popup:** Es utilizado generalmente por los menús contextuales.

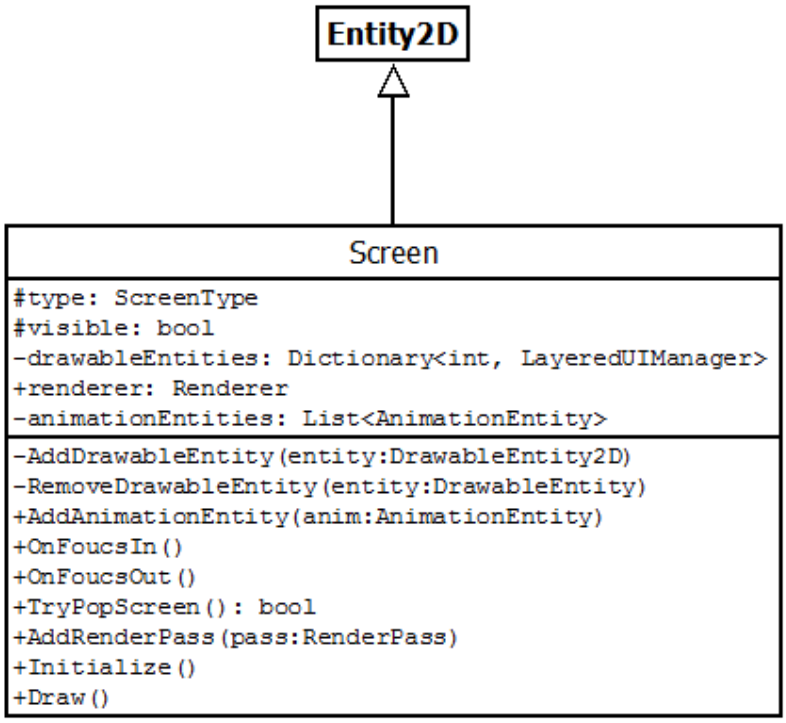


Figura 56: Diagrama de clases de la clase Screen.

Cada pantalla tiene un conjunto *DrawableEntity2D* clasificadas por bandera de bits, una lista de entidades de animación, y un objeto de tipo *Renderer*.

El atributo *type* indica el tipo de pantalla, *visible* indica si la pantalla debe ser desplegada, *drawableEntities* representa una tabla de entidades clasificadas por máscara de bits, *renderer* representa el renderizador, *animationEntities* almacena la lista de entidades para la animación de elementos gráficos.

Los métodos AddDrawableEntity() y RemoveDrawableEntity() agregan o eliminan entidades de la tabla de entidades desplegables, AddAnimationEntity() permite añadir una entidad de animación, OnFocusIn() y OnFocusOut() son métodos virtuales y son invocados cuando una pantalla obtiene o no el foco respectivamente. El método TryPopScreen() retorna si la pantalla debe ser eliminada de inmediato por el manejador de pantalla, AddRenderPass() permite añadir un pase de renderizado en el renderizador, Initialize() es invocado por el manejador de pantallas y cada clase concreta debe inicializar todas las entidades pertenecientes. Draw() renderiza todos los pases que pertenecen a la ventana.

MenuScreen: Es la clase base para las pantallas que contienen un menú de opciones (ver Figura 57). El usuario puede desplazarse entre una lista de opciones hacia arriba o hacia abajo para seleccionar alguna.

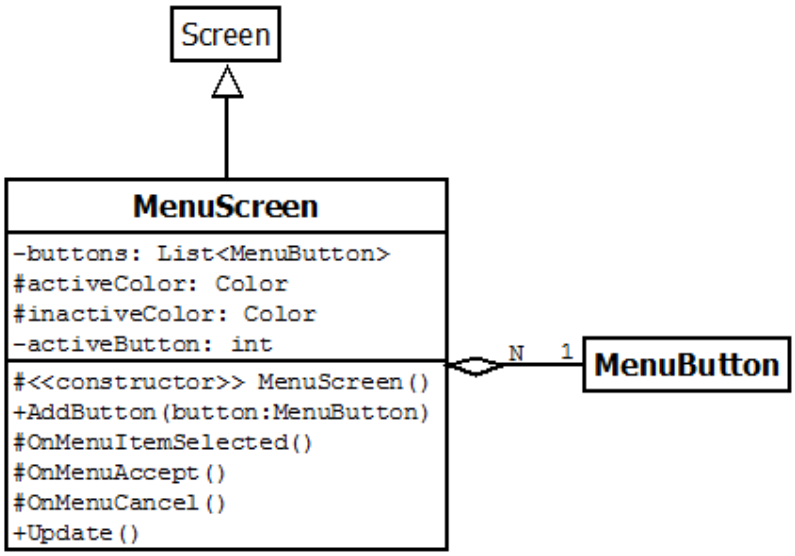


Figura 57: Diagrama de clases de la clase MenuScreen.

Las clases concretas de *Screen* y *MenuScreen* son:

- **MainMenuScreen:** Es la clase que controla el menú principal de la aplicación.
- **ResolutionMenuScreen:** Es una clase que controla la edición de la resolución de pantalla. Se definen varias resoluciones predeterminadas para ser elegidas por el usuario. Además en la versión Windows es posible expandir a tamaño completo la ventana y restaurarla a su tamaño por defecto. La opción en el menú principal se llama *Opciones de Pantalla* como se ve en la Figura 58.

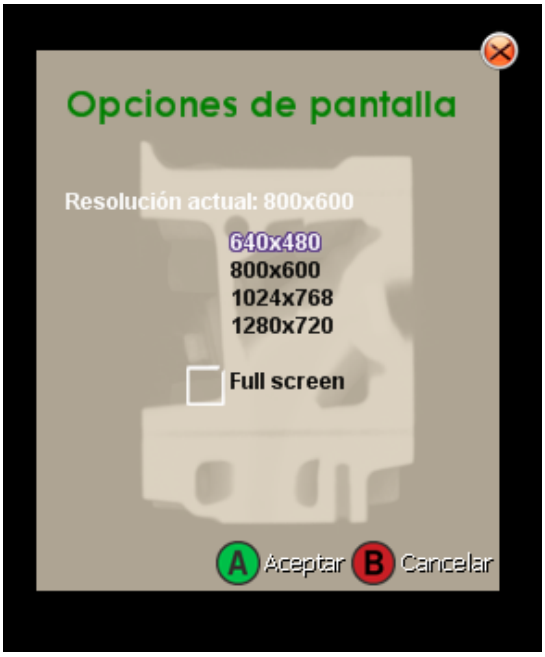


Figura 58: Opciones de pantalla.

- **TransferFunctionScreen:** Clase que controla la función de transferencia y despliega la interfaz correspondiente (Figura 59).

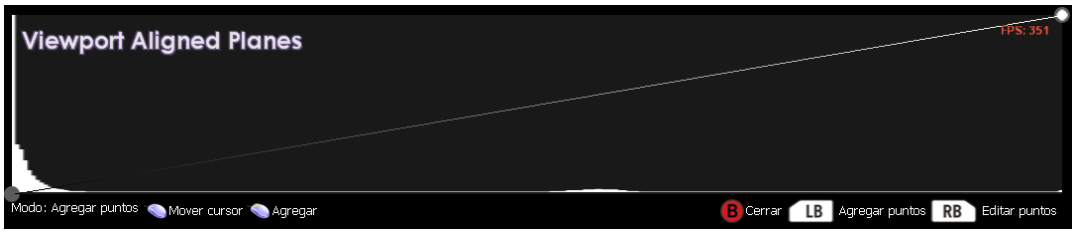


Figura 59: Función de transferencia.

Además permite agregar, editar, visualizar y eliminar puntos de control en la función de transferencia como se ve en la Figura 60.

- **PointEditorMenuScreen:** Es un menú contextual que permite editar un punto de control de la función de transferencia. En ambiente Windows, se despliega un menú con dos opciones: Editar color y Eliminar (Figura 60), mientras que en la plataforma Xbox se agrega la opción mover como se ve en la Figura 61.
- **RGBScreen:** Clase que permite modificar el color de un punto de control en la función de transferencia a partir de los canales RGB.

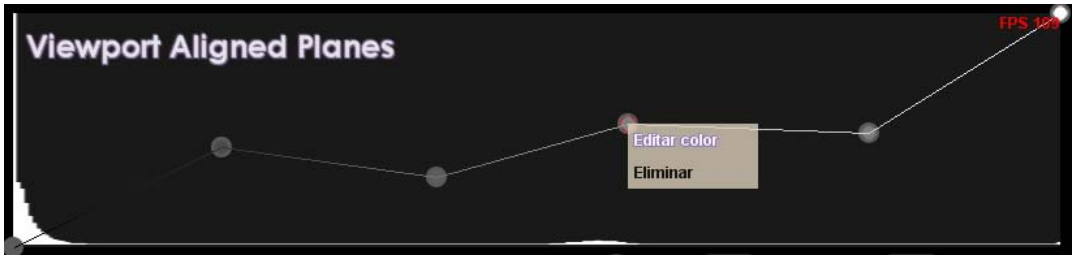


Figura 60: Edición de color de un punto de control en la función de transferencia en plataforma Windows.

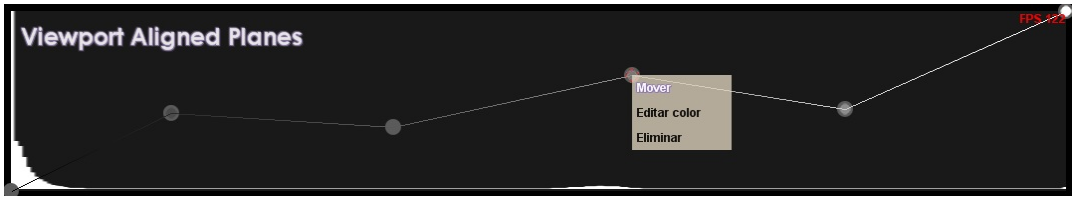


Figura 61: Edición de color de un punto de control en la función de transferencia en plataforma Xbox.

Al entrar en la opción de edición de color para un punto de control, se visualiza un menú que permite la modificación de cada uno de los valores RGB de ese punto de control. La Figura 62 ilustra dicho menú. Nótese que el canal del RGB que está siendo editado es resaltado en un color más claro para indicar que es el seleccionado.

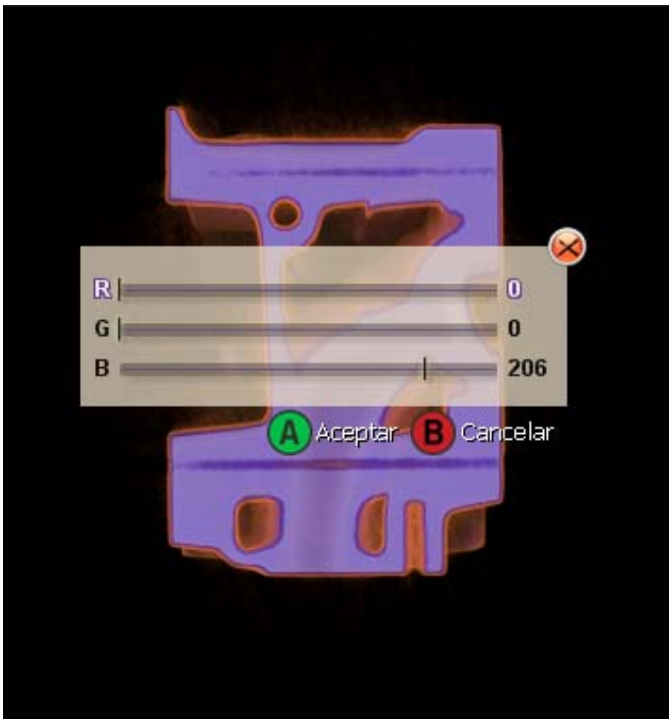


Figura 62: Menú de edición de color en un punto de control de la función de transferencia.

- **VolumeLoaderScreen:** Clase que permite cargar un volumen. En ambas plataformas se definen volúmenes predeterminados pero además esta clase permite buscar un archivo específico no predeterminado en el directorio, como se ve en la Figura 63.

Figura 63: Menú para la carga de volúmenes.

- **VolumeRenderingScreen:** Clase encargada del despliegue del volume rendering. En este trabajo está contemplado poder desplegar el volumen a través de dos tipos de técnica: Ray Casting y Planos Alineados al Viewport. Al hacer el despliegue se indicará en la parte superior izquierda de la pantalla el tipo de técnica que está siendo utilizada, como lo indica la Figura 64.

Figura 64: Despliegue de volúmenes utilizando Planos Alineados al Viewport.

Al ocultar el menú principal se mostrará información del volumen: las dimensiones (alto, ancho y profundidad) y la cantidad de bits por vóxel (8 bits o 16 bits), tal como se muestra en la Figura 65.

Figura 65: Despliegue del volumen utilizando la técnica Ray Casting.

- **VolumeRenderingTechniqueScreen:** Esta clase permite alternar la técnica de despliegue del volumen entre Ray Casting y Planos Alineados al Viewport. Al seleccionar la opción "Técnica de VR" en el menú principal se instancia esta clase desplegando el menú que muestra la Figura 66.

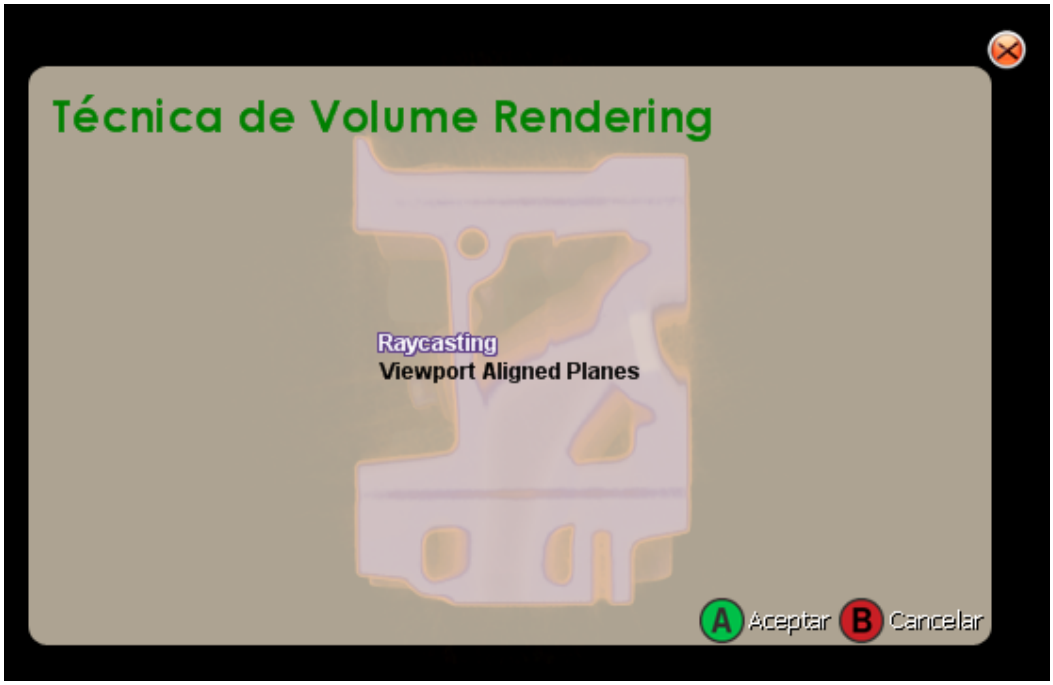


Figura 66: Menú para elegir la técnica de Volume Rendering.

- **RayCastingParametersScreen:** Esta clase permite ajustar los parámetros del Ray Casting como lo muestra la Figura 67.

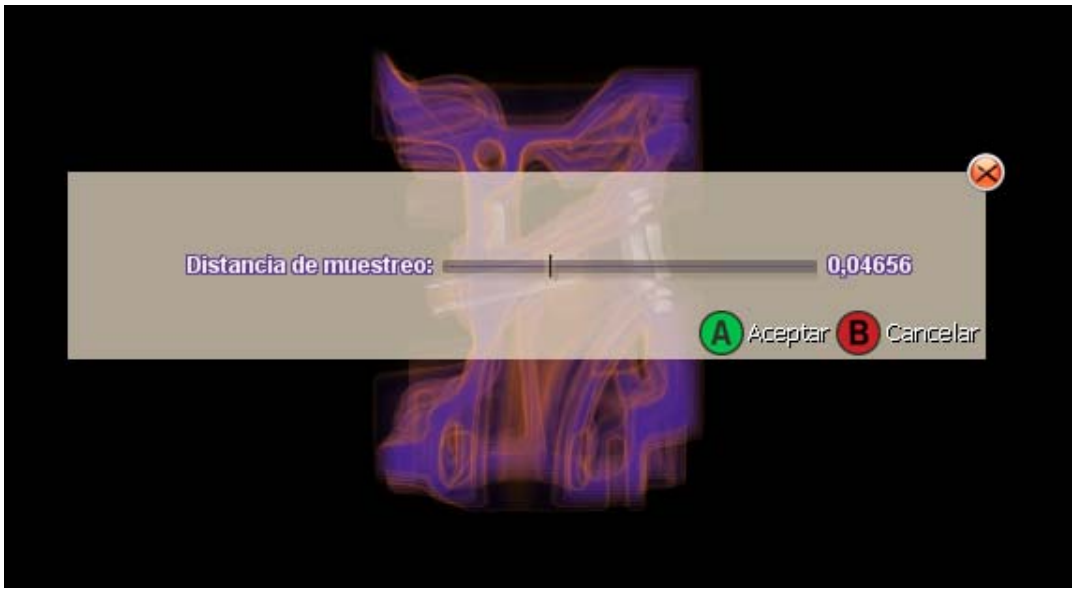


Figura 67: Edición de parámetros del Ray Casting.

- **ChangeBackgroundColorScreen:** Clase que permite modificar el color del fondo de la aplicación a partir de los canales RGB (Figura 68).

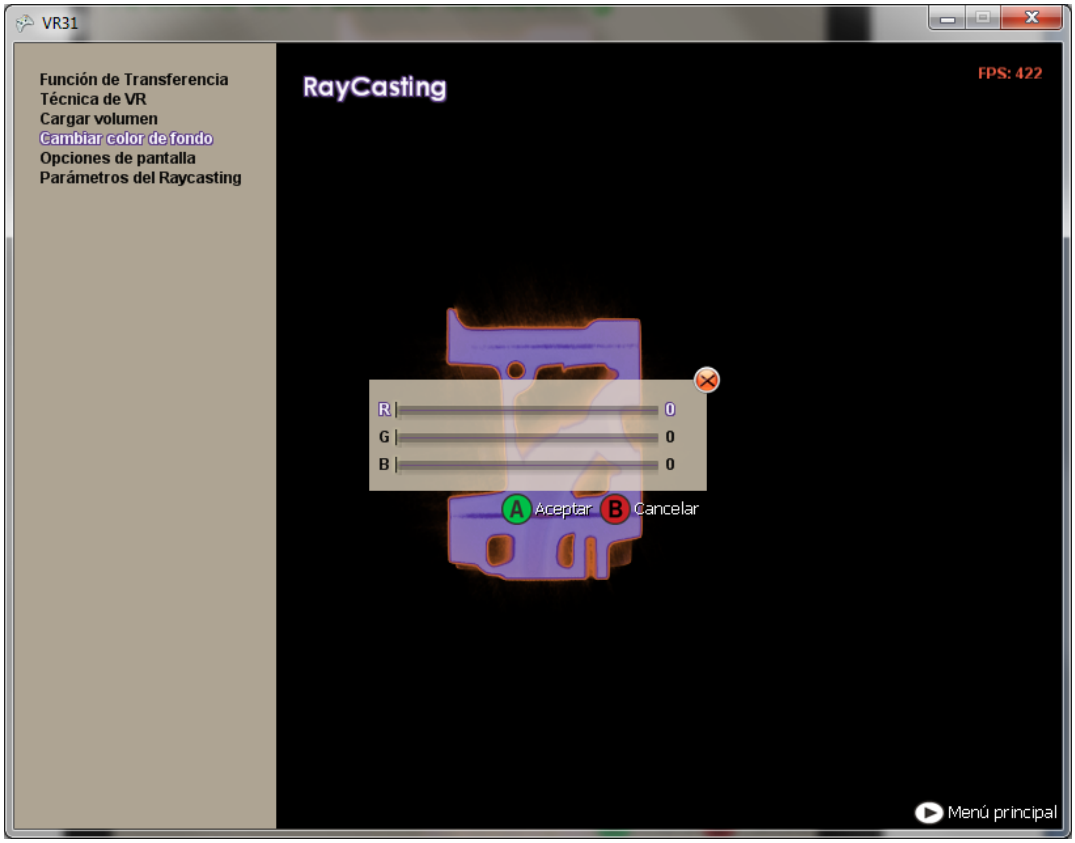


Figura 68: Menú de edición de color de fondo.

La Figura 69 muestra un diagrama de clases de **Screen** y sus clases concretas:

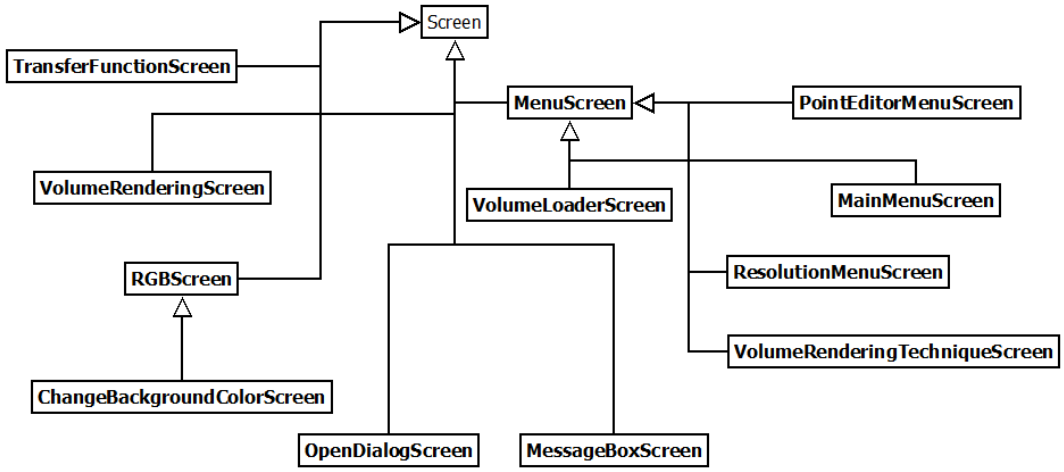


Figura 69: Diagrama de clases de la clase Screen y sus clases concretas.

4.8 Implementación de la transmisión de datos por red

Para la transmisión de datos por red se utilizó el modelo de desarrollo Cliente-Servidor, donde la consola Xbox hará las veces de cliente y el servidor será un computador personal con plataforma Windows.

La conexión de red se realiza a través de objetos de sesión. Cada sesión puede ser creada por un host (servidor), la sesión permanecerá activa hasta que se desconecte el servidor. El cliente podrá conectarse al servidor e instanciar una sesión.

Existen 3 tipos de sesiones:

- Usando un solo dispositivo (Xbox o PC).
- Utilizando una red LAN.
- A través de internet usando Xbox Live.

El código fuente de la aplicación que gestiona la transmisión de datos por red para cualquier tipo de sesión es el mismo. No es necesario escribir ningún código diferente para una sesión local, por Internet o de tipo LAN. Esta abstracción es realizada mediante un objeto de tipo *NetworkSession,* y es la columna vertebral de cualquier aplicación de red realizada con XNA.

Mediante esta clase se pueden acceder los miembros de la sesión, el miembro host y otras propiedades pertinentes a una sesión. Para crear una sesión es necesario tener una cuenta de usuario válida e iniciar sesión. Cada plataforma requiere un tipo de cuenta diferente, como se muestra en la Tabla 4.

Tarea	Xbox 360	Windows	Zune o Windows Phone
Ejecutar una aplicación con XNA	Membresía LIVE Silver + Membresía Premium de XNA Creators Club	No requiere membresía	No requiere cuenta
Crear una conexión LAN	Membresía LIVE Silver + Membresía Premium de XNA Creators Club	No requiere membresía	No requiere cuenta
Iniciar sesión en servidores de Xbox y Windows	Membresía LIVE Silver + Membresía Premium de XNA Creators Club	Membresía LIVE Silver + Membresía Premium de XNA Creators Club	No disponible
Utilizar el servicio de LIVE para conectar dispositivos a través del internet	Membresía LIVE Gold + Membresía Premium de XNA Creators Club	Membresía LIVE Silver + Membresía Premium de XNA Creators Club	No disponible

Tabla 4: Requerimientos de cuentas de usuario.

Cualquier aplicación de XNA debe inicializar explícitamente el sistema los Servicios de Jugador (*Gamer Services*, en inglés) antes de crear una sesión. Los Servicios de Jugador permiten a la aplicación recibir notificaciones, como los mensajes enviados a través de Xbox Live, entre otras notificaciones.

Para habilitar los Servicios de Jugador es necesario agregar el componente **GamerServicesComponent** a la lista de componentes de nuestra clase principal (**Game**).

La clase *Guide* provee una interfaz gráfica que facilita el inicio de sesión. Es necesario invocar al método *ShowSignIn*, el cual permitirá al usuario seleccionar una cuenta existente o crear una nueva.

Una vez creada la sesión, en cada cuadro de ejecución de la aplicación se debe invocar al método **Update** de la sesión. Esta invocación realizará las siguientes acciones:

- Enviar y recibir paquetes de red.
- Cambiar el estado de la sesión, como por ejemplo controlar los usuarios que pertenecen a la sesión.
- Realizar notificaciones de eventos para cualquier cambio relevante.

En este trabajo especial de grado se implementó una sesión de tipo LAN y el modelo cliente-servidor.

Para el caso del servidor se desarrolló una nueva aplicación que consta de una pantalla (*ServerMainMenuScreen*). Esta pantalla contiene dos bitácoras de mensajes: la primera muestra las peticiones del cliente (mensajes recibidos) y la segunda muestra las respuestas del servidor (mensajes enviados). En ambas bitácoras se utiliza un formato de visualización con dos valores, un entero que indica el tipo de mensaje y un texto que constituye el cuerpo del mensaje. La Tabla 5 indica los mensajes predefinidos que el servidor es capaz de reconocer y enviar.

Identificador	Descripción
0	Mensaje vacío
1	Contiene una lista de archivos y directorios
2	Contiene la ruta de la carpeta actual
3	Contiene la ruta del volumen solicitado
4	Contiene el encabezado del volumen a enviar, compuesto por las dimensiones del volumen, la cantidad de bytes del volumen, la cantidad de paquetes de volúmenes
5	Contiene un mensaje de error
6	Contiene información de un paquete de volumen
7	Envía un mensaje de solicitud para obtener la ruta de la carpeta actual en el servidor
8	Contiene una colección de tiempos en flotantes
9	Envía un mensaje de solicitud para obtener los archivos y directorios de la ruta actual en el servidor

Tabla 5: Descripción de los mensajes aceptados por el servidor.

La clase **NetworkConnection** gestiona todo lo relacionado con sesiones de red: creación y unión en una sesión (CreateSession() y JoinSession()), envío y recepción de paquetes (Send() y Receive()). En la Figura 70 puede apreciarse todo el conjunto de atributos y métodos que conforman dicha clase.

El servidor debe invocar al método CreateSession(), mientras que el cliente invocará a JoinSession(). Para que el cliente pueda unirse en una sesión, debe existir una sesión creada, en caso de existir se intentará conectar a la primera sesión disponible.

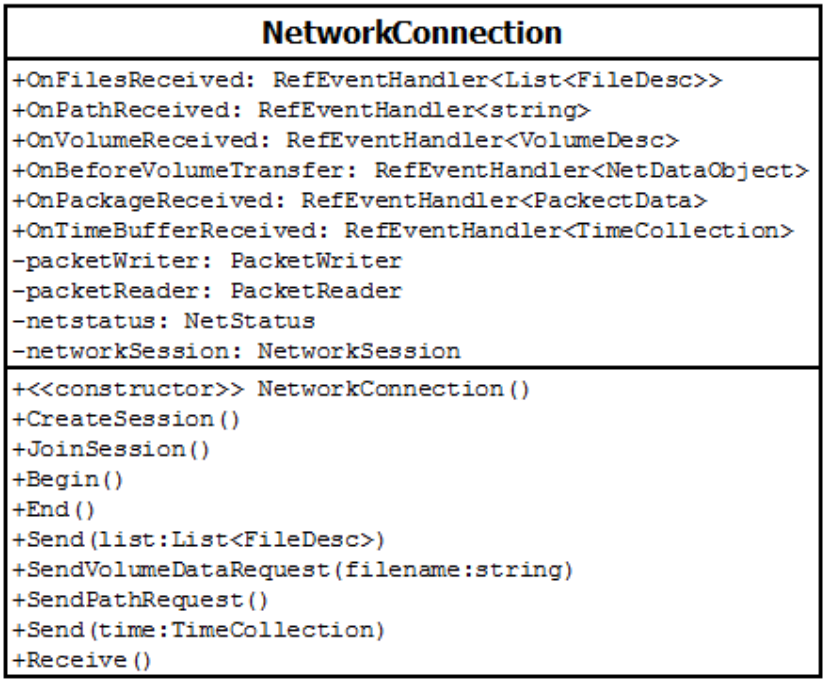


Figura 70: Diagrama de clases de la clase NetworkConnection.

Cuando un paquete de red es recibido, su identificador es leído mediante el objeto *packetReader* y de acuerdo a ello se leerán los datos del cuerpo del mensaje. Aunado a esto se cuenta con una cantidad de eventos que son invocados luego de recibir correctamente los paquetes de red, con la finalidad de que las entidades externas realicen tareas específicas de acuerdo al tipo de paquete recibido.

El evento *OnFilesReceived* es levantado luego de recibir los directorios y archivos del servidor, *OnPathReceived* es levantado luego de recibir la ruta actual del servidor, *OnVolumeReceived* es invocado luego de recibir el volumen, *OnBeforeVolumeTransfer* es levantado al recibir la cabecera del volumen a enviar, *OnPackageReceived* es invocado luego de recibir un fragmento del volumen.

El atributo *netstatus* es un enumerado que indica el estado de la conexión de red, *networkSession* alberga la sesión de red, *packetWriter* y *packetReader* permiten escribir y leer paquetes de red por medio de una conexión de red.

4.9 Selección de archivos mediante el Explorador de Archivos

Se desarrolló una interfaz unificada (ver Figura 71) para la selección de archivos para ambas plataformas. En el caso de la plataforma Xbox es necesario tener establecida una sesión de red para poder navegar por las carpetas del servidor.

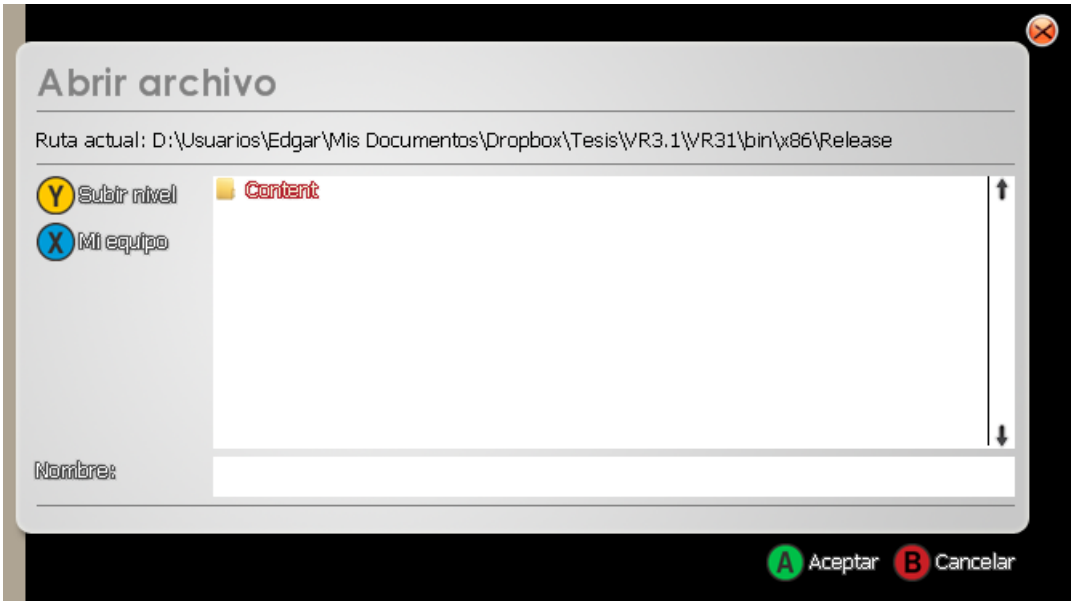


Figura 71: Ventana OpenDialogScreen.

Se desarrolló una interfaz de programación (**IExplorer**) que permite canalizar la obtención de carpetas y archivos para ambas plataformas. Es posible especificar extensiones de archivos aceptados, la ruta de la carpeta actual, entre otros. Las clases concretas son **NetExplorer, WinExplorer** y **XboxExplorer** para las plataformas Xbox y Windows respectivamente, como se aprecia en la Figura 72.

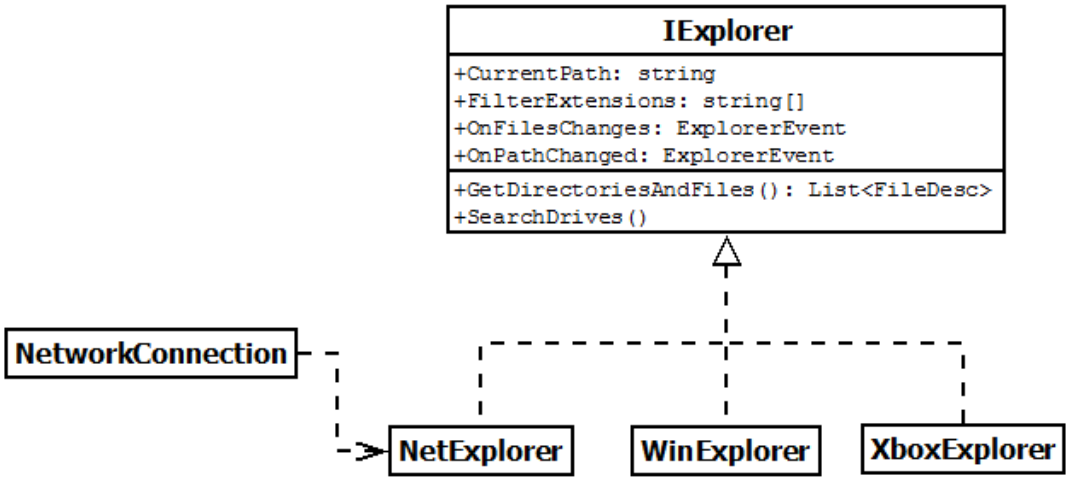


Figura 72: Diagrama de clases de la clase IExplorer.

4.10 Uso de la aplicación en el Xbox 360

Una vez culminado el desarrollo de la aplicación, se genera el ejecutable tanto para la versión PC como para el Xbox. Mediante XNA *Game Studio* se puede transferir el ejecutable y los *assets* de la aplicación a la consola Xbox. Antes de hacerlo se necesita establecer una conexión entre la consola y el PC utilizando XNA *Game Studio Device Center*. El XNA *Game Studio Devices* permite establecer conexión con Zune y con el Xbox 360 como lo indica la Figura 73.

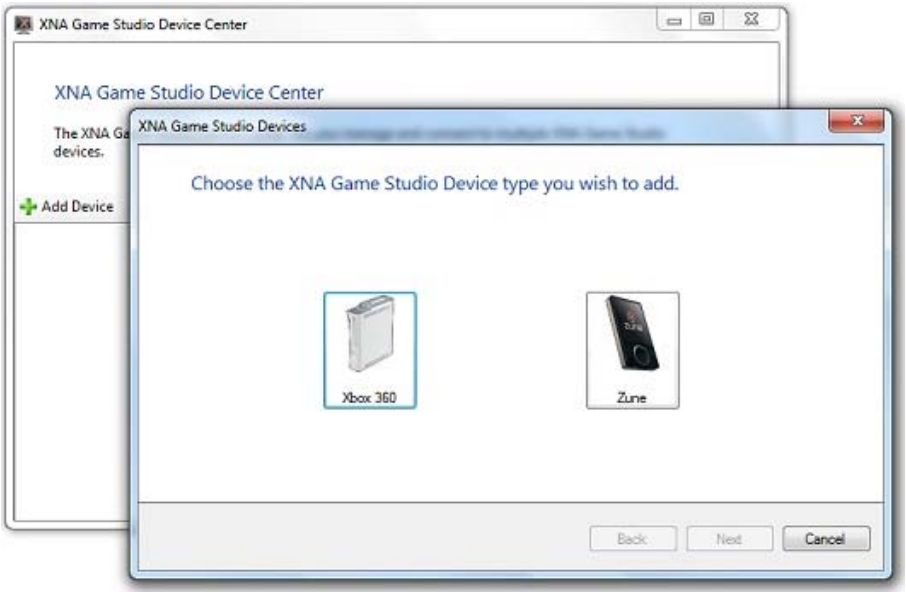


Figura 73: Ventana de XNA Game Studio Device Center en Windows.

Con el fin de establecer la conexión es necesario descargar y ejecutar XNA *Game Studio Connect* desde el Xbox *LIVE Marketplace*. Para lograrlo se deben seguir los siguientes pasos:

1. **Iniciar sesión en Xbox Live**: Para iniciar sesión se necesita una membrecía de tipo Xbox *Live Silver* o Xbox *Live Gold*, una membresía a XNA *Creators Club* o una membresía a *App Hub Trial*, y un disco duro conectado a la consola.
2. **Descargar XNA Game Studio Connect**: Se debe descargar XNA Game Studio Connect desde el Xbox LIVE Marketplace e instalarlo en la consola. Para descargarlo seleccione la pestaña *Games* en la interfaz principal de la consola y luego seleccione *Game Marketplace*. Dentro del *Game Marketplace* se encontrará el XNA *Game Studio Connect* en la pestaña *Games*, para facilitar su búsqueda se puede acceder desde la opción de búsqueda alfabética *Titles A to Z* como se indica en la Figura 74 y seleccionar luego la letra *X*. La aplicación se descargará en el disco duro, y una vez descargada se busca desde la pestaña *Games* de la interfaz principal (catálogo de juegos) y al ejecutarla se instala automáticamente.

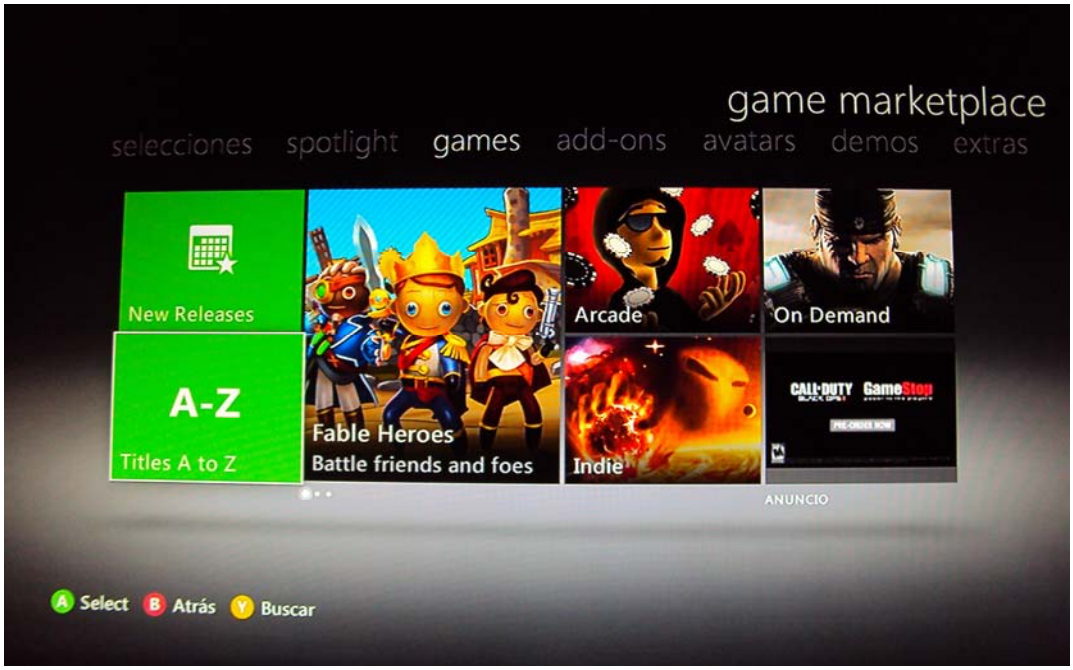


Figura 74: Pantalla del Game Marketplace en el Xbox.

3. **Establecer conexión**: La primera vez que se establece una conexión entre la computadora y el Xbox a través del *XNA Game Studio Connect,* el Xbox proporcionará una clave de acceso como se indica en la Figura 75. Esta clave debe ser ingresada en XNA *Game Studio Device Center* desde la computadora. A continuación se carga una pantalla de espera mientras se establece la conexión (Figura 76).

Figura 75: Configuración de clave para establecer una conexión a través del XNA Game Studio Connect.

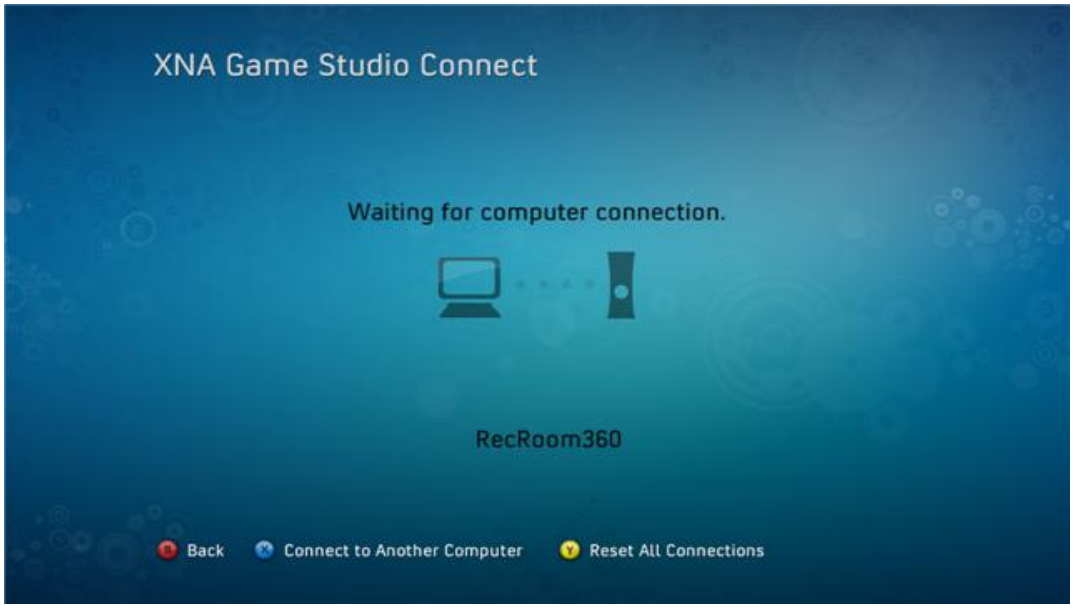


Figura 76: Estableciendo conexión desde XNA Game Studio Connect.

4. **Uso de la aplicación**: Luego de establecer la conexión, es necesario ejecutar la aplicación desde *Visual Studio* para utilizar la aplicación en la consola de Xbox.

A continuación en el próximo capítulo se describen diversas pruebas de rendimiento y de calidad de imagen que se realizaron a la aplicación desarrollada.

CAPÍTULO V. Pruebas y Resultados

Una vez finalizada la etapa de diseño y desarrollo es necesario poner a prueba el sistema para medir el rendimiento y la calidad de las imágenes obtenidas. A continuación se detalla el ambiente de pruebas, los volúmenes utilizados y los resultados obtenidos.

5.1 Descripción del ambiente de pruebas

En esta sección se especifica el entorno de hardware y software en el cual se realizaron las pruebas.

5.1.1 Requerimientos de hardware

Para las pruebas se utilizó una consola de Xbox 360 con un disco duro y diversos computadores personales como se muestra en la Tabla 6. Allí se pueden apreciar algunas de las especificaciones técnicas de cada equipo como el tipo de procesador, la memoria RAM y el sistema operativo.

Equipo	Plataforma	Procesador	Memoria RAM	Sistema Operativo
1	Xbox	IBM 3.2GHz	512MB*	-
2	PC	Intel Core i5 3.66GHz	8 GB	Windows 7
3	PC	Intel Core 2 Quad 2.40 GHz	3,24 GB	Windows XP

Tabla 6: Descripción de los equipos utilizados en las pruebas de rendimiento.

En la Tabla 7 se indican las tarjetas gráficas utilizadas para cada equipo y la memoria de video con la que cuentan cada una de ellas. En el caso del Xbox 360, la memoria es la misma tanto para video como para RAM, en otras palabras la memoria se comparte entre las tareas de procesamiento y de despliegue gráfico, y es asignada de acuerdo a los requerimientos de la aplicación.

Equipo	Tarjeta Gráfica	# de Núcleos	Memoria de video
1	ATI C1 Xenos	48	512MB*
2	NVIDIA 460 GTX	336	768MB
3	NVIDIA 8800 GTS	96	640MB

Tabla 7: Descripción de las tarjetas gráficas utilizadas en las pruebas de rendimiento.

* Memoria compartida entre las tareas de procesamiento y de despliegue gráfico.

5.1.2 Requerimientos de software

Los requerimientos de software son los siguientes:

- Sistema Operativo Windows XP o superior.
- DirectX 9.0 para el despliegue gráfico con XNA. Para el despliegue de efectos se utilizó HLSL con soporte a los procesadores de vértices y de píxeles.
- XNA 3.1 para el manejo de la interfaz gráfica, el despliegue de volúmenes y la transferencia de datos por red.
- En el caso de ejecutar la aplicación en el Xbox se requiere el acceso a Internet para poder conectarse al servicio de *Xbox Live*.

5.1.3 Volúmenes

Para las pruebas se utilizaron tres volúmenes:

- **Volumen A (motor)**: Es una Tomografía Computarizada de un motor. Tiene una dimensión de 256x256x128 donde cada vóxel tiene 8 bits de precisión y ocupa 16 MB de memoria. En la Figura 77 se puede apreciar una imagen del volumen A.

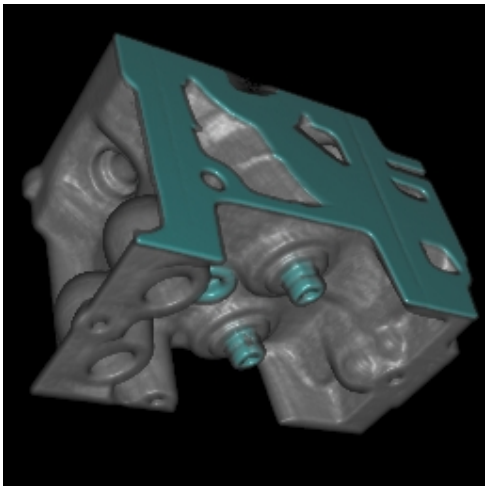


Figura 77: Captura del despliegue de un volumen TC de un motor.

- **Volumen B (rana)**: Es una imagen de Resonancia Magnética de una rana. Tiene una dimensión de 256x256x44 donde cada vóxel tiene 8 bits de precisión y ocupa 2,75 MB de memoria. La Figura 78 ilustra el volumen B.

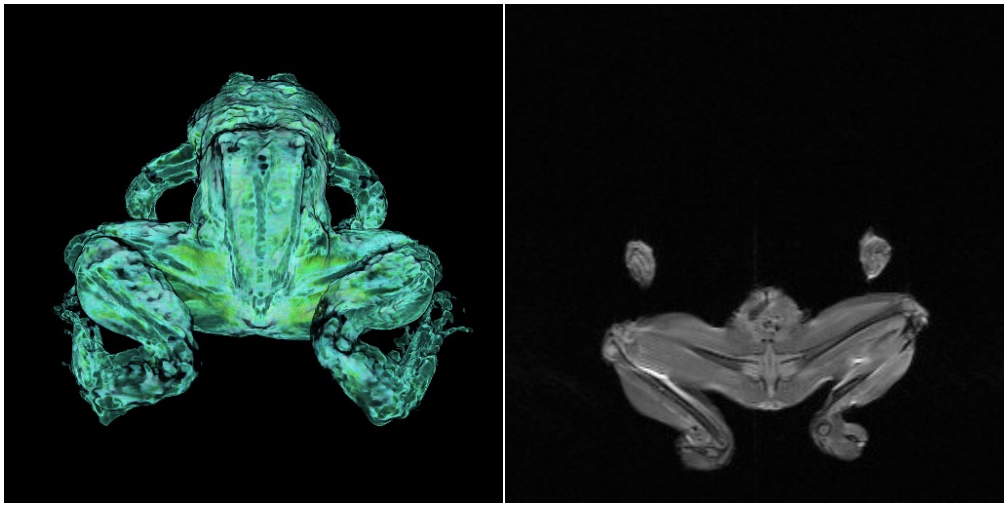


Figura 78: A la izquierda, captura del despliegue de un volumen MRI de una rana. A la derecha un corte del volumen.

- **Volumen C (cabeza)**: Es una imagen de Resonancia Magnética simulada de un fragmento de cabeza humana. Tiene una dimensión de 181x217x181 donde cada vóxel tiene 16 bits de precisión y ocupa 13 MB de memoria. La Figura 79 muestra una imagen del volumen.

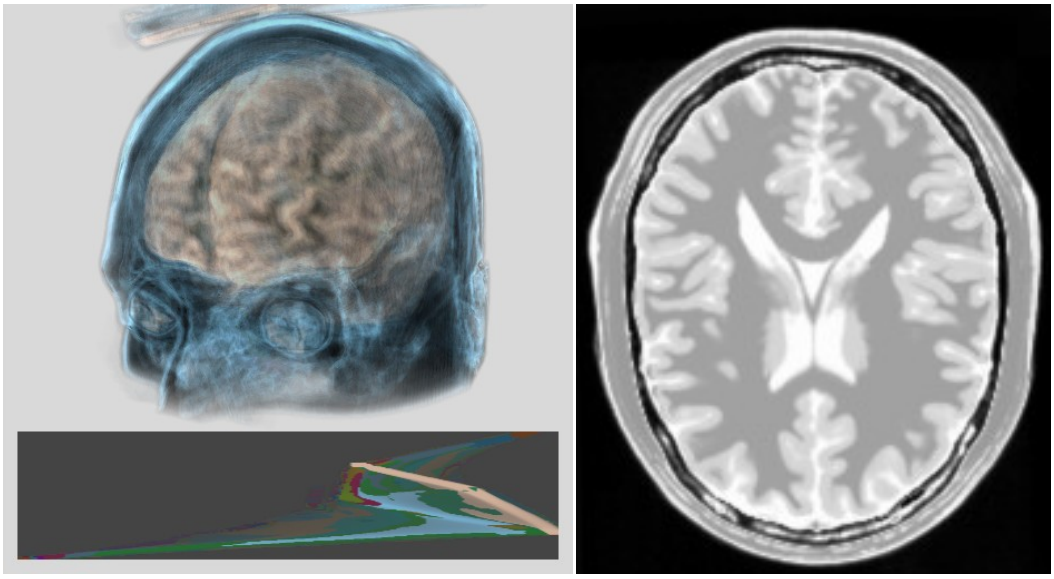


Figura 79: Arriba la izquierda, captura del despliegue de un volumen MRI simulado de un fragmento de cabeza humana. Abajo a la izquierda, captura de la función de transferencia utilizada. A la derecha un corte del volumen.

5.2 Resultados Cuantitativos

A continuación se hicieron una serie de mediciones de tiempo de despliegue y cuadros por segundo (*FPS – Frames per seconds*) para los tres volúmenes vistos anteriormente. Cada prueba se ejecutó cinco veces para obtener resultados más precisos. Se utilizaron dos técnicas de despliegue: *Ray Casting* y Planos alineados al Viewport, en ambas se utiliza el mismo ángulo de visión y la misma resolución de pantalla.

También se establecen aquí comparaciones entre técnicas y equipos, en cuanto a su rendimiento de acuerdo a los resultados obtenidos. Así mismo, se realizaron pruebas con respecto al tiempo de transmisión en segundos de paquetes a través de la red para cuatro volúmenes de prueba.

5.2.1 Consideraciones previas

Antes de medir resultados fue necesario configurar la función de transferencia para cada volumen. Se utilizan tres configuraciones: la primera es la función identidad, común a los tres volúmenes. Las otras dos configuraciones se eligen de manera arbitraria para cada volumen, procurando resaltar la mayor cantidad de vóxeles posibles para que el procesamiento sea más pesado. Estas dos funciones de transferencia de los volúmenes están descritas en la Tabla 8. Para cada función se definen los puntos FT(y) como una tupla ordenada de 5 elementos, donde los primeros 4 elementos representan el color en el modelo RGBA y el 5to elemento representa el isovalor.

Volumen	Función de Transferencia 2	Función de Transferencia 3
A	FT(0)={0,0,0,0,0} FT(1)={4,59,4,19,43} FT(2)={255,137,137,0,139} FT(3)={255,255,255,5,255}	FT(0)={0,0,0,0,0} FT(1)={4,59,242,5,7} FT(2)={255,31,137,4,117} FT(3)={255,255,255,26,255}
B	FT(0)={0,0,0,0,0} FT(1)={48,6,6,0,31} FT(2)={0,126,20,15,45} FT(3)={255,255,255,41,255}	FT(0)={0,0,0,0,0} FT(1)={48,165,6,0,42} FT(2)={172,0,172,147,146} FT(3)={255,255,255,0,255}
C	FT(0)={0,0,0,0,0} FT(1)={0,0,96,5,13} FT(2)={191,15,15,168,25} FT(3)={255,255,255,123,255}	FT(0)={0,0,0,0,0} FT(1)={174,5,5,17,3} FT(2)={0,0,96,5,13} FT(3)={15,15,15,11,21} FT(4)={191,15,15,168,25} FT(5)={255,255,255,123,255}

Tabla 8: Descripción de las funciones de transferencia utilizadas.

5.2.2 Resultados obtenidos utilizando la técnica de Planos Alineados al Viewport

A continuación se presentan en la Tabla 9 los resultados de las pruebas realizadas en el Volumen A (motor) con respecto al tiempo de despliegue (TD) y los frames por segundo, para las tres funciones de transferencia diferentes mencionadas anteriormente.

Volumen A		Función de Transferencia 1		Función de Transferencia 2		Función de Transferencia 3	
		TD (seg)	FPS	TD (seg)	FPS	TD (seg)	FPS
Equipo 1	Prueba 1	0,034215228	29,233	0,03414285	29,261	0,03415493	29,261
	Prueba 2	0,034163137	29,261	0,034128018	29,257	0,03420873	29,21
	Prueba 3	0,034155432	29,232	0,03413799	29,29	0,0341559	29,261
	Prueba 4	0,034144128	29,261	0,034137512	29,29	0,03416736	29,22
	Prueba 5	0,034150651	29,261	0,034164612	29,236	0,0342106	29,232
Equipo 2	Prueba 1	0,00908125	110	0,008994884	111,262	0,00908995	110
	Prueba 2	0,009090106	110,22	0,008986811	111,222	0,00909884	110,11
	Prueba 3	0,00909893	110,22	0,009003512	111,333	0,00908121	110,162
	Prueba 4	0,00908096	110,22	0,008995091	111,29	0,00908995	110,11
	Prueba 5	0,009090149	110,22	0,008985601	111,237	0,00909895	110,11
Equipo 3	Prueba 1	0,015209064	65,76	0,015211521	65,737	0,01521308	65,736
	Prueba 2	0,015209666	65,782	0,015211721	65,785	0,01521261	65,802
	Prueba 3	0,015209384	65,802	0,015211824	65,736	0,01521294	65,772
	Prueba 4	0,01520948	65,742	0,015211586	65,736	0,01521265	65,76
	Prueba 5	0,015209277	65,736	0,015212323	65,736	0,01521355	65,737

Tabla 9: Resultados de las cinco pruebas realizadas sobre el Volumen A (motor) con Planos Alineados al Viewport.

Las pruebas realizadas sobre el Volumen B (rana) se muestran en la Tabla 10.

Volumen B		Función de Transferencia 1		Función de Transferencia 2		Función de Transferencia 3	
		TD (seg)	FPS	TD (seg)	FPS	TD (seg)	FPS
Equipo 1	Prueba 1	0,033833736	29,55	0,033819747	29,547	0,03383774	29,527
	Prueba 2	0,033819611	29,54	0,033857249	29,544	0,03382011	29,522
	Prueba 3	0,033835158	29,555	0,033838854	29,555	0,03382701	29,559
	Prueba 4	0,033843992	29,522	0,033850539	29,55	0,03383711	29,538
	Prueba 5	0,033835803	29,533	0,033827207	29,525	0,03382332	29,55
Equipo 2	Prueba 1	0,009263252	108	0,008638839	115,768	0,00863197	115,847
	Prueba 2	0,00927515	107,513	0,008630351	115,768	0,0086407	115,884
	Prueba 3	0,009280339	107,57	0,008646396	115,723	0,00862335	115,859
	Prueba 4	0,009264022	108	0,008638961	115,745	0,00863238	115,884
	Prueba 5	0,009271102	108	0,00863064	115,397	0,00864095	115,558
Equipo 3	Prueba 1	0,015176113	65,934	0,015176812	65,868	0,01517548	65,868
	Prueba 2	0,015174267	65,934	0,015175429	65,868	0,01517542	65,872
	Prueba 3	0,015174609	65,934	0,015176185	65,868	0,01517634	65,868
	Prueba 4	0,015176048	65,917	0,01517559	65,868	0,01517532	65,868
	Prueba 5	0,015175577	65,868	0,015176798	65,868	0,01517566	65,868

Tabla 10: Resultados de las cinco pruebas realizadas sobre el Volumen B (rana) con Planos Alineados al Viewport.

Finalmente, la Tabla 11 indica los resultados obtenidos tras hacer las pruebas sobre el Volumen C (cabeza).

Volumen C		Función de Transferencia 1		Función de Transferencia 2		Función de Transferencia 3	
		TD (seg)	FPS	TD (seg)	FPS	TD (seg)	FPS
Equipo 1	Prueba 1	0,028482012	35,14	0,028439529	35,166	0,02848067	35,14
	Prueba 2	0,02846316	35,103	0,028471392	35,14	0,02846663	35,07
	Prueba 3	0,028471833	35,103	0,028456971	35,14	0,0284717	35,07
	Prueba 4	0,028471752	35,105	0,028453081	35,07	0,02849505	35,07
	Prueba 5	0,028479846	35,174	0,02845408	35,099	0,02847706	35,105
Equipo 2	Prueba 1	0,007776888	128,629	0,007780357	128,513	0,00781154	128
	Prueba 2	0,007777041	128,742	0,007772285	128,612	0,00781931	128
	Prueba 3	0,00777708	128,642	0,007787424	128,533	0,00780433	128
	Prueba 4	0,007776977	128,742	0,007779161	128,642	0,00781175	128
	Prueba 5	0,007776993	128,742	0,007772114	128,619	0,00781922	128
Equipo 3	Prueba 1	0,012836512	77,922	0,012836871	77,922	0,011697	77,922
	Prueba 2	0,012836557	77,922	0,012836398	77,922	0,01169309	77,875
	Prueba 3	0,012837345	77,907	0,012837032	77,922	0,01169917	77,844
	Prueba 4	0,012836286	77,844	0,012836905	77,922	0,0116989	77,92
	Prueba 5	0,012837159	77,865	0,0128369	77,922	0,01169399	77,92

Tabla 11: Resultados de las cinco pruebas realizadas sobre el Volumen C (cabeza) con Planos Alineados al Viewport.

A continuación se presenta en la Tabla 12 los diferentes resultados obtenidos tras promediar las cinco pruebas de acuerdo a las diferentes funciones de transferencia y los diferentes equipos, así podemos comparar los resultados de tiempo de despliegue en los tres volúmenes para cada equipo.

		Motor (seg)	Rana (seg)	Cabeza (seg)
Equipo 1	Función de Transferencia 1	0,034165715	0,03383366	0,02847372
	Función de Transferencia 2	0,034142196	0,033838719	0,028455011
	Función de Transferencia 3	0,034179504	0,033829055	0,028478222
Equipo 2	Función de Transferencia 1	0,009088279	0,009270773	0,007776996
	Función de Transferencia 2	0,00899318	0,008637037	0,007778268
	Función de Transferencia 3	0,009091781	0,00863387	0,007813229
Equipo 3	Función de Transferencia 1	0,015209374	0,015175323	0,012836772
	Función de Transferencia 2	0,015211795	0,015176163	0,012836821
	Función de Transferencia 3	0,015212965	0,015175644	0,011696431

Tabla 12: Tiempos promedios de despliegue a las diferentes funciones de transferencia para la técnica Planos Alineados al *Viewport*.

Como se observa en la Tabla 12, los resultados en tiempo de despliegue son muy similares en las tres funciones de transferencia. Si promediamos esos valores podremos comparar el rendimiento obtenido de acuerdo a los tres volúmenes utilizados. En la Figura 80 podemos apreciar que, a excepción del Volumen C (cabeza), los resultados obtenidos son similares entre los volúmenes A (motor) y B (rana). Esto ocurre porque el Volumen C (cabeza) tiene menos vóxeles que los otros dos, y por lo tanto, el tiempo de despliegue también disminuye.

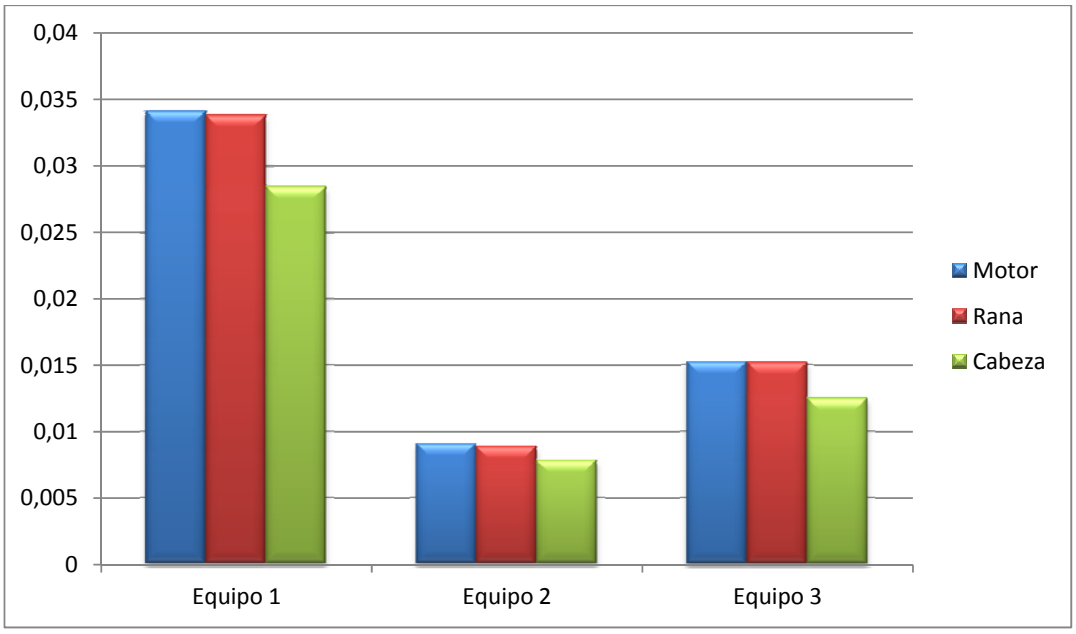


Figura 80: Gráfica comparativa de los tiempos de despliegue obtenidos con la técnica Planos Alineados al Viewport, por cada volumen, en los diferentes equipos de prueba.

De esta forma es posible notar que las pruebas ejecutadas en el Equipo 1 arrojan resultados en tiempo de despliegue muy por encima de los otros dos equipos. Igualmente, se puede evidenciar que el Equipo 2 es el que requiere de menos tiempo para el despliegue.

Con respecto a los *Frames* por segundo (FPS) se puede hacer un análisis similar siendo que la cantidad de FPS desplegados en cada prueba es muy similar, e inclusive es la misma en algunos casos. Así en la Tabla 13 podemos apreciar la cantidad de FPS obtenidos de acuerdo a los tres volúmenes y las tres funciones de transferencia en cada equipo.

		Motor	Rana	Cabeza
Equipo 1	Función de Transferencia 1	29,2	29,6	35,4
	Función de Transferencia 2	29,4	29,4	35,2
	Función de Transferencia 3	29,2	29,6	35
Equipo 2	Función de Transferencia 1	110,176	107,8166	128,6994
	Función de Transferencia 2	111,2688	115,6802	128,5838
	Función de Transferencia 3	110,0984	115,8064	128
Equipo 3	Función de Transferencia 1	65,7644	65,9174	77,892
	Función de Transferencia 2	65,746	65,868	77,922
	Función de Transferencia 3	65,7614	65,8688	77,8962

Tabla 13: Tiempos promedios de los FPS de acuerdo a las diferentes funciones de transferencia para la técnica Planos Alineados al Viewport.

Como se observa en la Tabla 13, los valores son muy similares en cualquiera de las tres funciones de transferencia, de acuerdo al volumen desplegado y al equipo utilizado para su procesamiento. Si promediamos estos resultados podremos observar que los valores son similares también entre volúmenes, de forma tal que solo podemos comparar la diferencia de rendimiento entre los tres equipos como se muestra en la Figura 81.

En esta misma gráfica es notable el hecho de que el volumen C (cabeza) presenta mejores resultados en los tres equipos con respecto a los otros volúmenes. Como se dijo anteriormente, la cantidad de vóxeles es menor en el Volumen C y por lo tanto la respuesta en el despliegue es mucho mayor. Por otro lado, los resultados obtenidos con el volumen A (motor) y B (rana) son bastante similares.

De manera general, se puede observar que los resultados obtenidos con el Equipo 2 son los de más alto rendimiento (mayor cantidad de FPS), mientras que, una vez más el Equipo 1 presenta el menor rendimiento. Es visible así la relación entre el tiempo de despliegue y la cantidad de *frames* por segundo, a menor tiempo de despliegue, mayor cantidad de *frames* por segundo y con ello, mayor rendimiento.

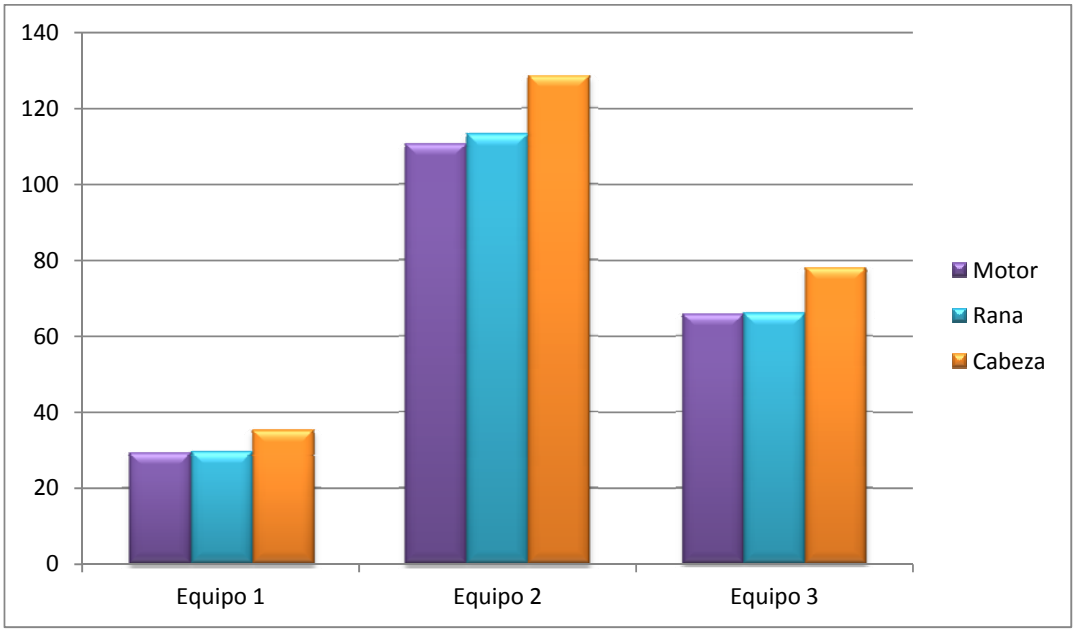


Figura 81: Gráfica comparativa de la cantidad de *Frames* por segundo obtenidos con la técnica Planos Alineados al Viewport, por cada volumen en los diferentes equipos de prueba.

5.2.3 Resultados obtenidos con Raycasting

A continuación se expresan en la Tabla 14 los resultados de las pruebas realizadas en el Volumen A (motor) con respecto al tiempo de despliegue (TD) y los frames por segundo, para las tres funciones de transferencia diferentes.

Volumen A		Función de Transferencia 1		Función de Transferencia 2		Función de Transferencia 3	
		TD (seg)	FPS	TD (seg)	FPS	TD (seg)	FPS
Equipo 1	Prueba 1	0,046731709	21,383	0,050077658	19,96	0,04906066	20,389
	Prueba 2	0,046731622	21,378	0,050078702	19,96	0,04906042	20,38
	Prueba 3	0,046730443	21,38	0,050078128	19,949	0,04906091	20,38
	Prueba 4	0,046731714	21,378	0,050077404	19,94	0,04906035	20,38
	Prueba 5	0,046731012	21,373	0,050077868	19,948	0,04906061	20,38
Equipo 2	Prueba 1	0,005430252	184,319	0,006282767	159,159	0,00621529	161,161
	Prueba 2	0,005427945	184	0,006275324	158,841	0,00620382	160,678
	Prueba 3	0,005426388	183,448	0,006288565	158,841	0,0062148	160,741
	Prueba 4	0,005429346	183,816	0,00627624	159	0,00621001	160,678
	Prueba 5	0,00543385	183,816	0,006283892	159	0,00620279	160,839
Equipo 3	Prueba 1	0,009796417	97,291	0,011139294	86,913	0,01092062	88
	Prueba 2	0,009803882	97,413	0,011142302	86,913	0,01091275	87,912
	Prueba 3	0,009815775	97,216	0,011147793	86,913	0,0109091	88
	Prueba 4	0,009805938	97,194	0,011155494	86,913	0,01091047	88
	Prueba 5	0,009796371	97,388	0,011155192	86,913	0,01091202	88

Tabla 14: Resultados de las cinco pruebas realizadas sobre el Volumen A (motor) con Raycasting.

Las pruebas realizadas sobre el Volumen B (rana) están tabuladas en la Tabla 15.

Volumen B		Función de Transferencia 1		Función de Transferencia 2		Función de Transferencia 3	
		TD (seg)	FPS	TD (seg)	FPS	TD (seg)	FPS
Equipo 1	Prueba 1	0,042360884	23,592	0,053098646	18,852	0,05188762	19,283
	Prueba 2	0,042366594	23,592	0,053097467	18,829	0,0518889	19,264
	Prueba 3	0,042361415	23,602	0,053092824	18,829	0,05189268	19,279
	Prueba 4	0,042360186	23,598	0,053098378	18,829	0,05189361	19,266
	Prueba 5	0,042363589	23,607	0,053094172	18,829	0,05189052	19,281
Equipo 2	Prueba 1	0,005057873	198	0,005826006	171,787	0,00554972	180
	Prueba 2	0,005050858	198	0,005824992	171,656	0,00556043	180
	Prueba 3	0,005055103	198,198	0,005831473	171,723	0,00554965	180
	Prueba 4	0,005047093	198	0,005821712	171,665	0,00556022	180
	Prueba 5	0,00505274	198	0,005826363	171,687	0,00554982	180
Equipo 3	Prueba 1	0,009786623	88	0,011156468	87,912	0,01072425	91,816
	Prueba 2	0,009786428	102,204	0,011170059	87,912	0,01072911	91,816
	Prueba 3	0,009780775	102,11	0,011181636	87,824	0,0107098	91,554
	Prueba 4	0,009791172	102,212	0,011171727	87,912	0,01070503	91,696
	Prueba 5	0,009811673	102	0,011194964	89,434	0,01070721	91,632

Tabla 15: Resultados de las cinco pruebas realizadas sobre el Volumen B (rana) con Raycasting.

Finalmente, la Tabla 16 indica los resultados obtenidos tras realizar las pruebas sobre el Volumen C (cabeza).

Volumen C		Función de Transferencia 1		Función de Transferencia 2		Función de Transferencia 3	
		TD (seg)	FPS	TD (seg)	FPS	TD (seg)	FPS
Equipo 1	Prueba 1	0,050734531	19,683	0,053575598	18,639	0,05314118	18,799
	Prueba 2	0,050731865	19,7	0,053573802	18,658	0,05313863	18,829
	Prueba 3	0,050734549	19,7	0,053572897	18,63	0,05313098	18,817
	Prueba 4	0,050735761	19,7	0,053574466	18,62	0,05312354	18,814
	Prueba 5	0,050732877	19,71	0,05357561	18,627	0,05313263	18,81
Equipo 2	Prueba 1	0,005560232	180	0,006299201	159,208	0,00639466	156,756
	Prueba 2	0,005549783	180,18	0,006306346	159,318	0,00640485	156,843
	Prueba 3	0,005555069	180	0,006292406	159,159	0,00639754	156,729
	Prueba 4	0,005560555	180	0,00630572	159,159	0,00639186	156,748
	Prueba 5	0,005549279	180	0,006300624	159,318	0,00640108	156,843
Equipo 3	Prueba 1	0,010508716	94,188	0,011591357	85,34	0,01180691	84,521
	Prueba 2	0,010509228	94,282	0,011590539	85,34	0,01180273	84,521
	Prueba 3	0,010512389	94,204	0,011584057	85	0,0117939	84,505
	Prueba 4	0,010514654	94,127	0,011586044	85,085	0,0117996	84,57
	Prueba 5	0,010500845	93,898	0,011591371	85	0,01180229	84,575

Tabla 16: Resultados de las cinco pruebas realizadas sobre el Volumen C (cabeza) con Raycasting.

A continuación se presenta en la Tabla 17 los diferentes resultados obtenidos con la técnica de Raycasting tras promediar las cinco pruebas realizadas de acuerdo a las diferentes funciones de transferencia y los diferentes equipos, con la intención de comparar los resultados de tiempo de despliegue en los tres volúmenes para cada equipo.

		Motor (seg)	Rana (seg)	Cabeza (seg)
Equipo 1	Función de Transferencia 1	0,0467313	0,042362534	0,050733917
	Función de Transferencia 2	0,050077952	0,053096297	0,053574475
	Función de Transferencia 3	0,049060588	0,051890668	0,053133391
Equipo 2	Función de Transferencia 1	0,005429556	0,005052733	0,005554984
	Función de Transferencia 2	0,006281357	0,005826109	0,006300859
	Función de Transferencia 3	0,00620934	0,005553968	0,006397999
Equipo 3	Función de Transferencia 1	0,009803677	0,009791334	0,010509166
	Función de Transferencia 2	0,011148015	0,011174971	0,011588674
	Función de Transferencia 3	0,010912989	0,010715078	0,011801083

Tabla 17: Tiempo promedios de despliegue de acuerdo a las diferentes funciones de transferencia para la técnica Raycasting.

En este caso, los valores son bastante equivalentes entre volúmenes, a excepción quizá del Volumen C (cabeza) que tiene una respuesta en tiempo de procesamiento un poco más lenta en todos los equipos con respecto a los otros dos volúmenes. Al igual que en el caso de la técnica de Planos Alineados al Viewport, el Equipo 2 es quien presenta mayor rendimiento, mientras que el Equipo 1 tarda más, y con esta técnica demora incluso mucho más que con la anterior.

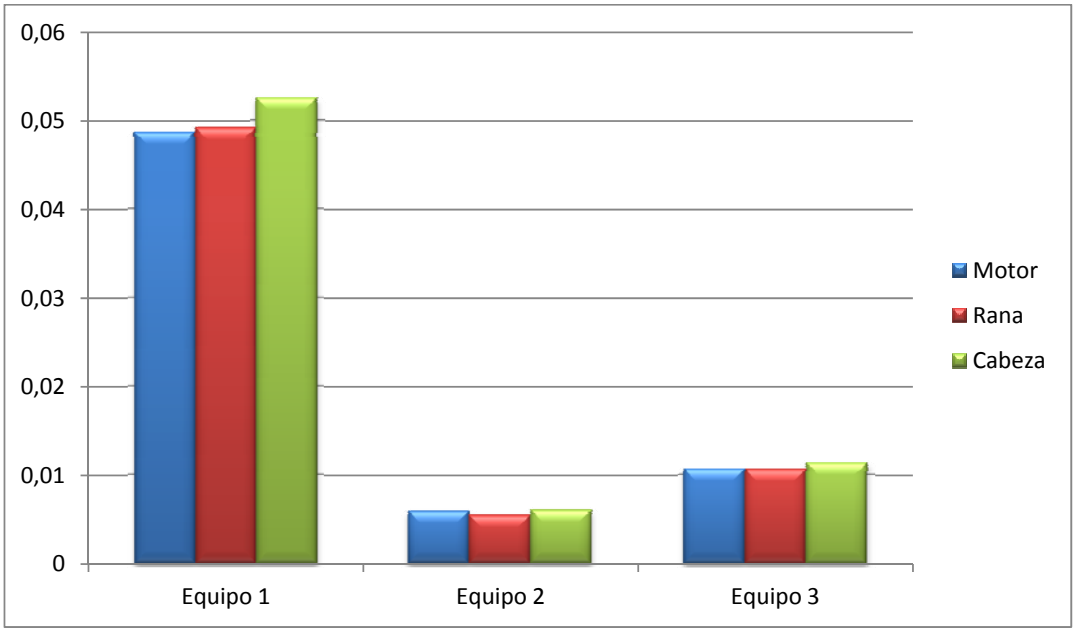


Figura 82: Gráfica comparativa de los tiempos de despliegue obtenidos con la técnica Raycasting, por cada volumen en los diferentes equipos de prueba.

En el caso de los *Frames* por segundo (FPS) los resultados no son tan similares como ocurre con la técnica de Planos Alineados *Viewport* debido a que las diferentes funciones de transferencia pueden concentrar mayor o menor cantidad de vóxeles, por lo cual los resultados son variantes.

		Motor	Rana	Cabeza
Equipo 1	Función de Transferencia 1	21,4	23,6	19,8
	Función de Transferencia 2	20	18,8	18,8
	Función de Transferencia 3	20,4	19,8	19
Equipo 2	Función de Transferencia 1	183,8798	198,0396	180,036
	Función de Transferencia 2	158,9682	171,7036	159,2324
	Función de Transferencia 3	160,8194	180	156,7838
Equipo 3	Función de Transferencia 1	97,3004	99,3052	94,1398
	Función de Transferencia 2	86,913	88,1988	85,153
	Función de Transferencia 3	87,9824	91,7028	84,5384

Tabla 18: FPS desplegados de acuerdo a las diferentes funciones de transferencia para la técnica Raycasting.

En la gráfica mostrada en la Figura 83 podemos ver como el Equipo 1 sigue siendo el que alcanza menor cantidad de FPS mientras que el Equipo 2 supera los 160 FPS en todos los casos. En los tres equipos el Volumen B alcanzó la mayor cantidad de FPS.

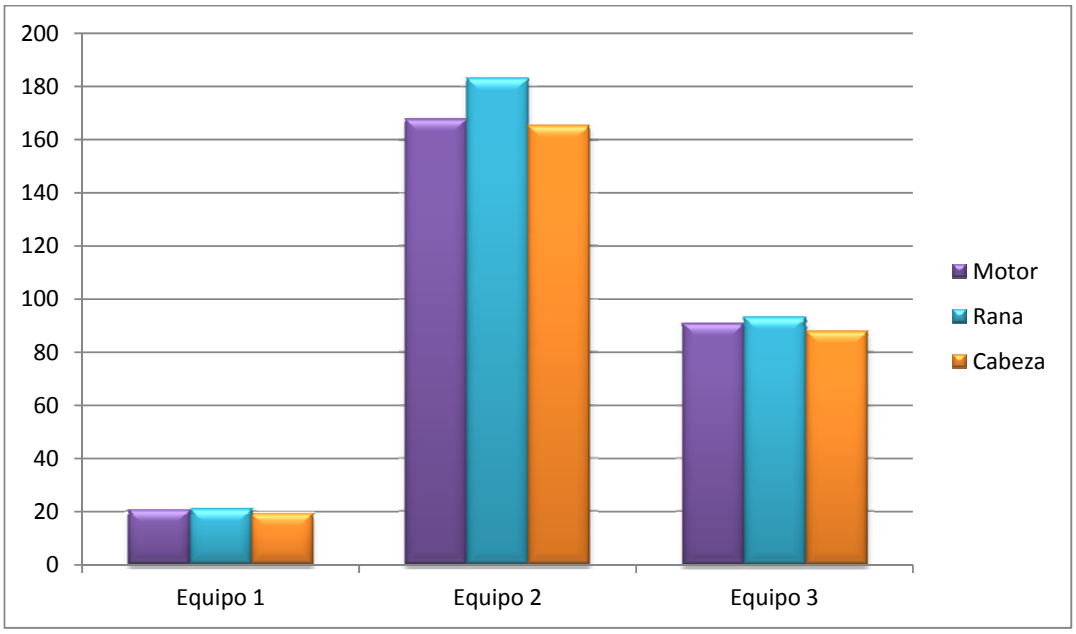


Figura 83: Gráfica comparativa de la cantidad de frames por segundo obtenidos con la técnica de RayCasting, por cada volumen en los diferentes equipos de prueba.

5.2.4 Rendimiento de los Equipos

Si promediamos los resultados obtenidos con ambas técnicas, independientemente de los volúmenes podremos comparar el desempeño de los equipos de prueba utilizados y las técnicas de Raycasting y Planos Alineados al Viewport.

		Viewport	Ray Casting
Equipo 1	Tiempo de Procesamiento (seg)	0,03215509	0,05007346
	FPS	31,3333333	20,1777778
Equipo 2	Tiempo de Procesamiento (seg)	0,00856482	0,00584521
	FPS	117,347733	172,162533
Equipo 3	Tiempo de Procesamiento (seg)	0,01428125	0,01082722
	FPS	69,8484667	90,5815333

Tabla 19: Comparación de rendimiento entre los equipos de prueba y las técnicas de despliegue de volúmenes utilizados.

En la Tabla 19 podemos apreciar los resultados tanto en tiempo de despliegue como en f*rames* por segundo de acuerdo a la técnica utilizada. En la Figura 84 podemos ver como en efecto el Equipo 1 tiene el mayor tiempo de despliegue, siendo los equipos 2 y 3 mucho más rápidos.

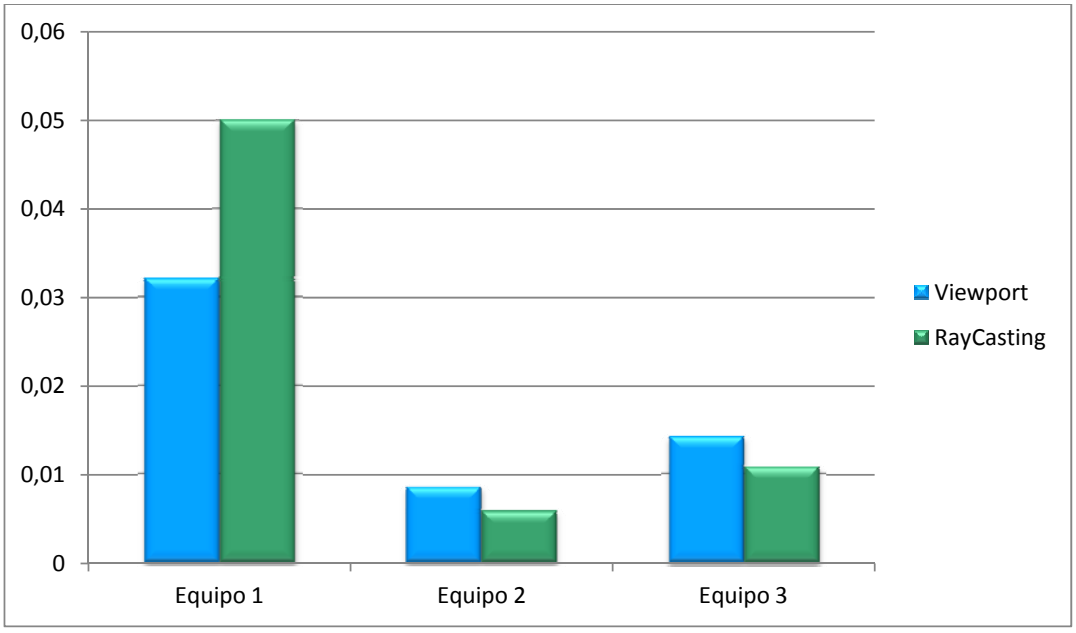


Figura 84: Relación de rendimiento en función del tiempo de procesamiento.

Como consecuencia de lo anterior, vemos como en la Figura 85 el Equipo 2 presenta la mayor cantidad de FPS, siendo aún mayor cuando se utiliza la técnica de RayCasting. De igual modo ocurre en el Equipo 3, quien le sigue en rendimiento. Esto ocurre porque la cantidad de cortes utilizados en la técnica Planos Alineados al Viewport es tomada de la dimensión mayor que tenga el volumen, mientras que la cantidad de vóxeles a procesar en el RayCasting puede ser menor, bien sea porque se utilizan menos o porque algunos son descartados, como por ejemplo los vóxeles transparentes o al llegar a un umbral de opacidad que no permite que se continúe la travesía del rayo.

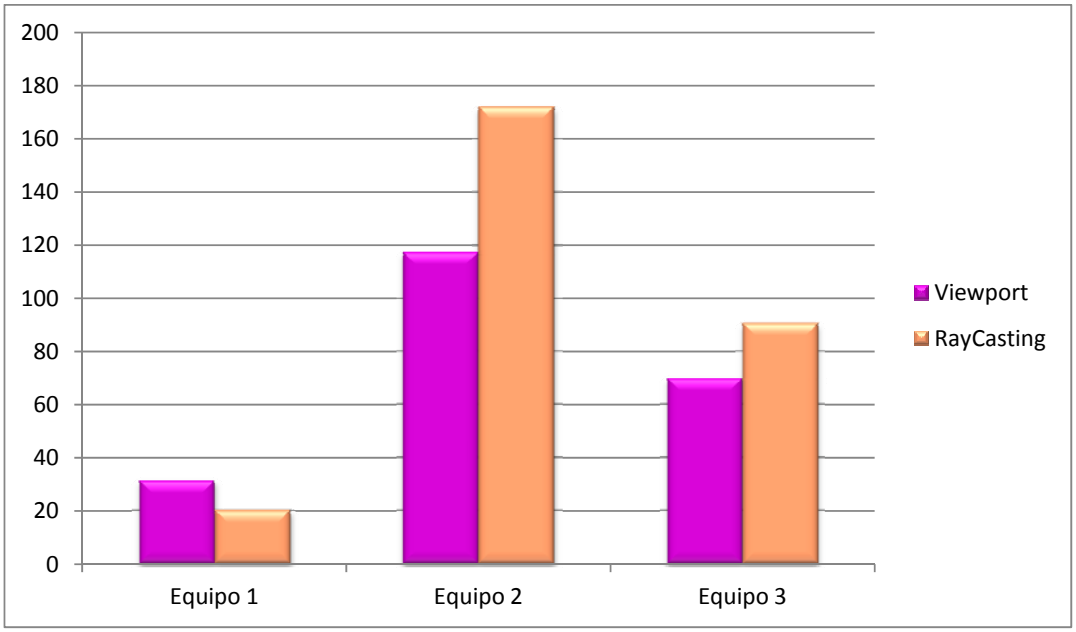


Figura 85: Comparación de rendimiento en función de los *Frames* por segundo (FPS).

5.2.5 Resultados obtenidos utilizando una conexión de red

Para estas pruebas se utilizaron cuatro volúmenes:

- **Volumen A (cubo)**: Es un volumen simulado de un cubo. Tiene una dimensión de 64x64x64 donde cada vóxel tiene 8 bits de precisión y ocupa 0,25 MB de memoria. En la Figura 86 se puede apreciar una imagen del volumen A.

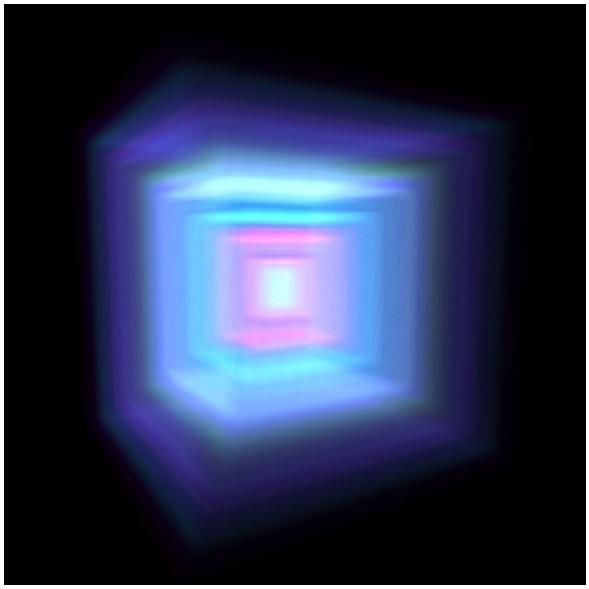


Figura 86: Captura del despliegue de un volumen simulado de un cubo.

- **Volumen B (rana)**: Se utiliza el mismo Volumen B de las pruebas anteriores. Es una imagen de Resonancia Magnética de una rana. Tiene una dimensión de 256x256x44 donde cada vóxel tiene 8 bits de precisión y ocupa 2,75 MB de memoria. La Figura 78 ilustra el volumen B.
- **Volumen C (cabeza)**: Se utiliza el mismo Volumen C de las pruebas anteriores. Es una imagen de Resonancia Magnética simulada de un fragmento de cabeza humana. Tiene una dimensión de 181x217x181 donde cada vóxel tiene 16 bits de precisión y ocupa 13 MB de memoria. La Figura 79 muestra una imagen del volumen.
- **Volumen D (pez)**: Es una Tomografía Computarizada de un pez. Tiene una dimensión de 256x256x512 donde cada vóxel tiene 16 bits de precisión y ocupa 64 MB. La Figura 87 muestra una imagen del volumen.

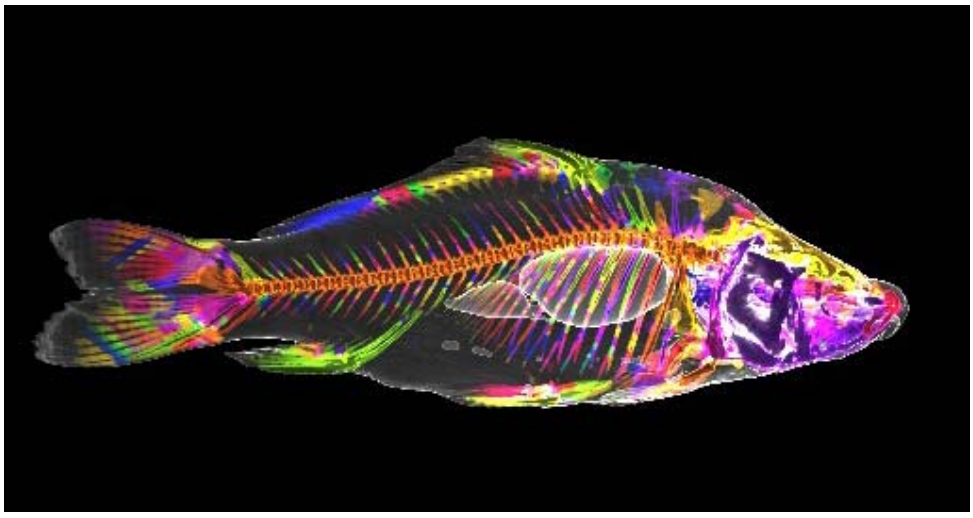


Figura 87: Captura del despliegue de un volumen TC de un pez.

A continuación podemos ver en la Tabla 20 los resultados arrojados tras registrar el tiempo en segundos de transferencia de paquetes a través de la red en los cuatro volúmenes. Para ello se hicieron tres pruebas con resultados bastante similares, en especial en los volúmenes A y D, como se observa en la Tabla 20.

	Cubo	Rana	Cabeza	Pez
Prueba 1	9,2988739	102,9214546	624,2641558	3155,510393
Prueba 2	9,2829149	102,0495845	619,9638832	3205,510783
Prueba 3	9,2888223	114,5928503	708,9784351	3220,508743

Tabla 20: Resultados en tiempo (seg.) de transmisión de paquetes en la red.

Al promediar estos resultados podemos ver en la Figura 88 que el Volumen A (cubo) es el que tiene menor tiempo en la transmisión de datos, y el Volumen D (pez) es el que tiene mayor tiempo. Esto ocurre debido a la diferencia de tamaño del volumen, mientras que el Volumen A ocupa 0,25 MB el Volumen D ocupa 64 MB.

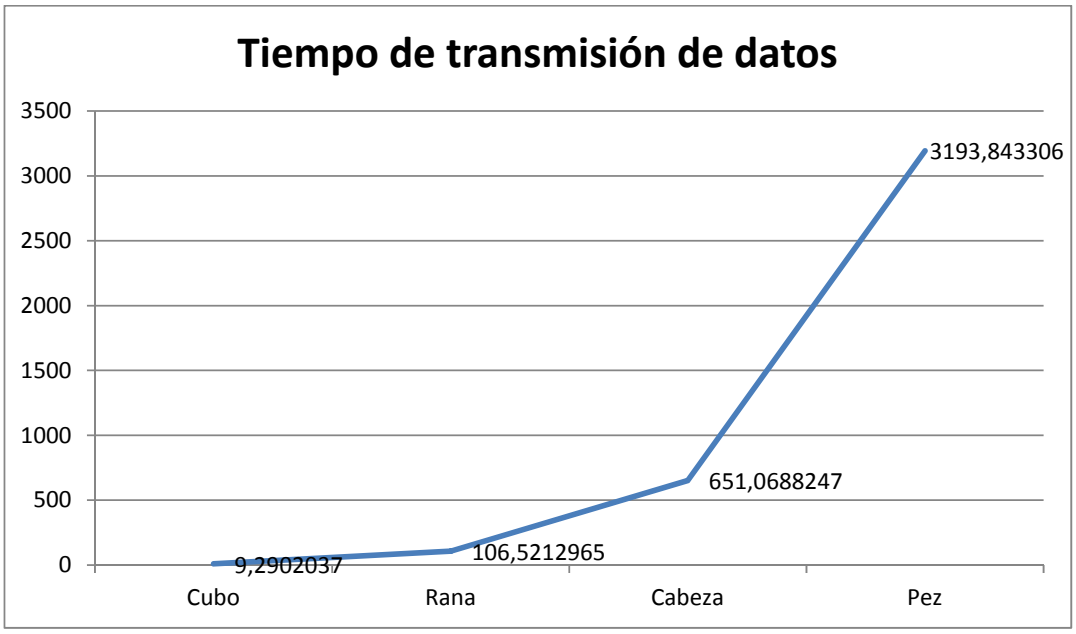


Figura 88: Resultados en la transmisión de paquetes en los diferentes volúmenes de prueba.

5.3 Resultados Cualitativos

A continuación se presentan los resultados visuales de la aplicación mediante capturas de pantalla. Para la plataforma Xbox se utilizó una capturadora de video EasyCap, la cual obtiene la imagen mediante un conector RCA o conector S-Video y es traspasado a una PC mediante un puerto USB.

Esta capturadora tiene una resolución máxima de 640x480 en los dispositivos con codificación NTSC. Por lo tanto, para equiparar los resultados fue necesario establecer la misma resolución de la capturadora en la plataforma Windows.

Se mostrarán sólo algunas capturas, en algunas de ellas pueden observarse los mejores resultados visuales obtenidos o mientras que otras se usaron para establecer comparaciones entre técnicas o equipos (aquellas en donde las diferencias se hacen más notorias). Para estas pruebas se utilizaron los volúmenes A, B y C de las pruebas cuantitativas y algunas de las funciones de transferencias antes descritas.

Dependiendo de la técnica utilizada (RayCasting o Planos Alineados al Viewport) la configuración de los parámetros fueron diferentes. En el caso del RayCasting la distancia del muestreo influye en la precisión de la composición del volumen, mientras que en la técnica Planos Alineados al Viewport, la calidad de la imagen variará de acuerdo a la cantidad de cortes (planos) en que se subdivida el volumen.

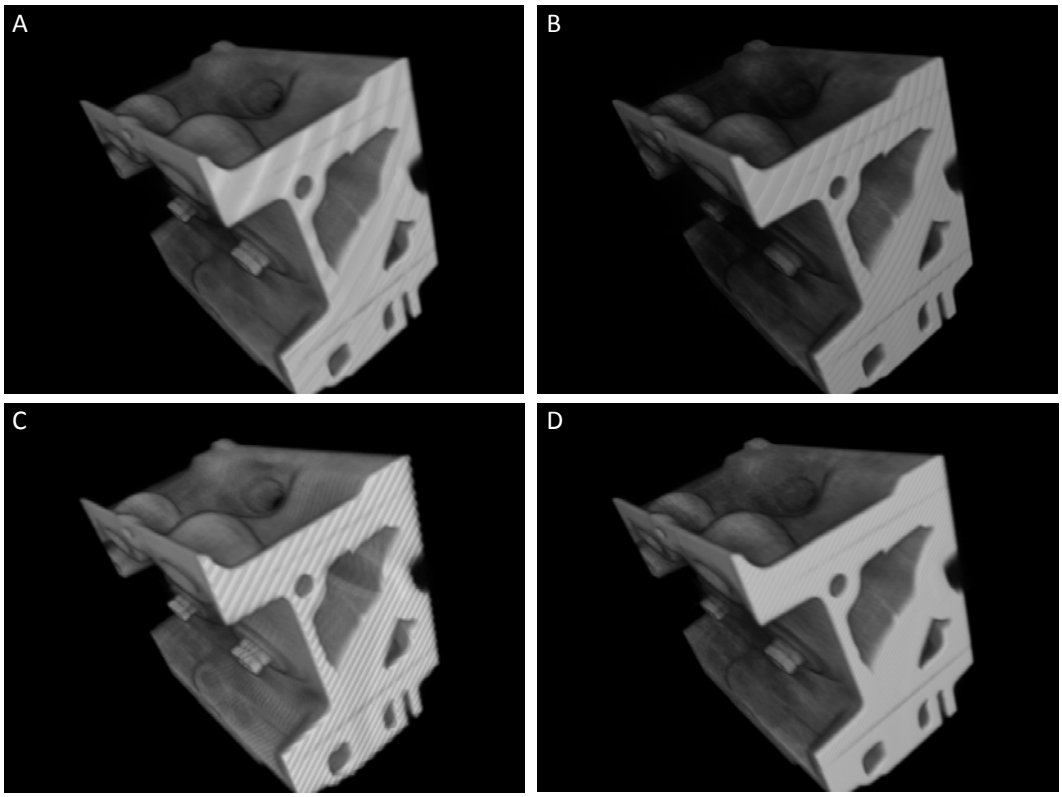


Figura 89: Resultados visuales del Volumen A en el Equipo 2. (A) Técnica Raycasting con un muestreo de 0.01 unidades. (B) Técnica Raycasting con un muestreo de 0.0045. (C) Técnica Planos Alineados al Viewport con 128 cortes. (D) Técnica Planos Alineados al Viewport con 256 cortes.

En la Figura 89 se puede apreciar como los resultados son más precisos con la técnica Raycasting cuando el valor de muestreo es menor, mientras que en la técnica Planos Alineados al Viewport, los mejores resultados visuales se obtienen cuando el número de cortes es mayor.

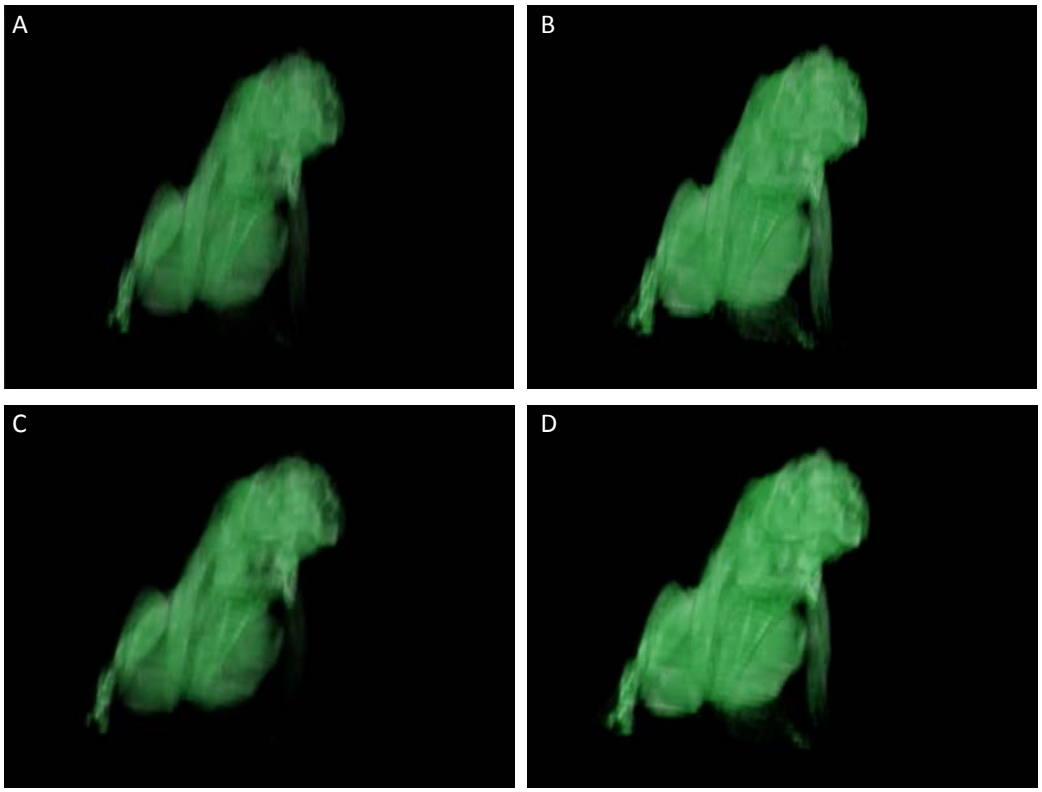


Figura 90: Resultados visuales del Volumen B con la Función de Transferencia 2. (A) Técnica Raycasting con un muestreo de 0.01 unidades en el Equipo 2. (B) Técnica Planos Alineados al Viewport con 256 cortes en el Equipo 2. (C) Técnica Raycasting con un muestreo de 0.01 unidades en el Equipo 1. (D) Técnica Planos Alineados al Viewport con 256 cortes en el Equipo 1.

En la Figura 90 podemos apreciar como los resultados visuales entre equipos suelen ser similares. Las figuras 90(B) y 90(D) contienen más información del volumen que las 90(A) y 90(C) por la cantidad de unidades de muestreo y de cortes del plano, sin embargo ambas técnicas ofrecen buenos resultados visuales en ambos equipos.

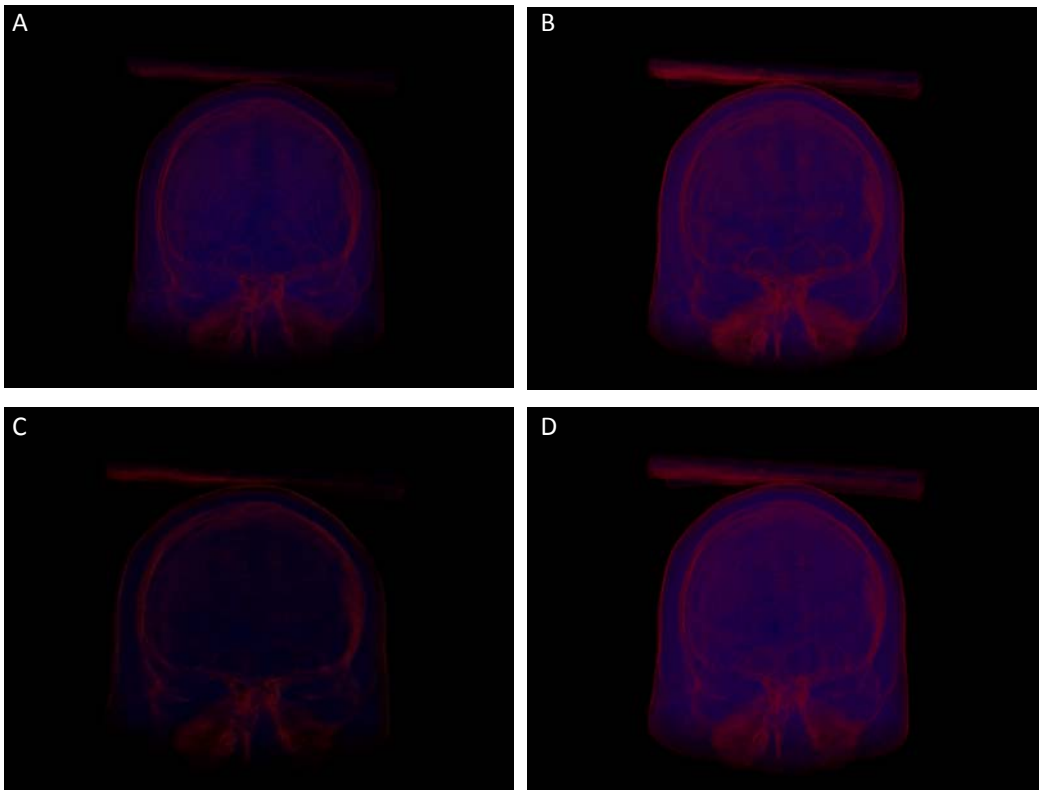


Figura 91: Resultados visuales del Volumen C con la Función de Transferencia 3. (A) Técnica Raycasting con un muestreo de 0.01 unidades en el Equipo 2. (B) Técnica Raycasting con un muestreo de 0.0045 en el Equipo 2. (C) Técnica Planos Alineados al Viewport con 256 cortes en el Equipo 1. (D) Técnica Planos Alineados al Viewport con 256 cortes en el Equipo 2.

Como el Xbox ha sido pensado para ser visualizado por medio de un televisor, aplica un proceso automático de corrección gamma durante el despliegue [23], [24], que en algunos casos puede aclarar la imagen, como en el caso de la Figura 90, en donde las muestras 90(A) y 90(B) capturadas del Equipo 2, son un poco más oscuras que las muestras 90(C) y 90(D) capturadas en el Xbox (Equipo 1).

Por otro lado, esta corrección gamma también podría alterar el brillo de la imagen oscureciéndola, como ocurre en la Figura 91(C) que se percibe más opaca que la Figura 91(D) que ha sido capturada del Equipo 2 a partir del mismo volumen, con la misma técnica de Planos Alineados al Viewport, con los mismos 256 cortes y la función de transferencia 3.

Esta diferencia entre los resultados obtenidos entre las plataformas Xbox (Equipo 1) y la PC (Equipo 2) con relación a la intensidad de color podemos resaltarla a partir de la Figura 91. En ella, todos los despliegues realizados en el Equipo 1 tienden a verse más oscuros que los desplegados en el Equipo 2, como podemos observar en la muestra 91(C) y 91(D).

Así mismo, cuando la calidad en la técnica de despliegue es mejor (despliegue Raycasting con menor unidad de muestreo o un despliegue con Planos Alineados al Viewport y mayor cantidad de cortes), la intensidad del color es mayor por el hecho de que se incluyen más detalles del volumen en el despliegue.

5.4 Mediciones de memoria

El Xbox tiene una cantidad de memoria compartida entre la memoria de video y memoria RAM. Siendo uno de los objetivos de la tesis determinar la cantidad máxima de memoria utilizable en el despliegue de un volumen, se hicieron algunas pruebas de medición.

Para ello se utiliza el manejador de excepciones de .NET y se captura la excepción "*OutOfMemoryException*" que se dispara cuando se intenta almacenar un volumen que excede la capacidad en memoria del Xbox. De esta forma, el volumen máximo que fue cargado exitosamente fue el Volumen D (pez), que tiene una dimensión de 256x256x512 vóxeles con 16 bits de precisión.

CAPÍTULO VI. Conclusiones y Trabajos Futuros

En este Trabajo Especial de Grado se pudo desarrollar una aplicación para visualizar volúmenes en plataforma Xbox y Windows, utilizando el *framework* XNA. Efectivamente, este trabajo demostró que se pueden desarrollar aplicaciones de visualización científica para ambas plataformas, con la misma interfaz de usuario, solamente cambiando en el compilador cuál será la plataforma de ejecución.

La interfaz gráfica desarrollada permite al usuario cargar un volumen, bien sea desde una ubicación local o remota a través de una conexión en una red local. También es posible elegir la técnica de despliegue a utilizar (RayCasting o Planos alineados al Viewport) y configurar los parámetros inherentes a las mismas (unidad de muestreo, planos de corte, etc.). Así mismo, es posible crear y editar una función de transferencia por medio de la interfaz, agregando, editando o eliminado puntos de control.

Para la carga de volúmenes, se creó un método de búsqueda de directorios en un sistema de archivos que no solo permite la carga de un volumen a partir de una ubicación local sino también desde otro equipo dentro de una red local. Una conclusión a la que se llegó durante el desarrollo es que para lograr esto, en el caso del Xbox, se debe crear una aplicación aparte que haga las veces de un servidor, mientras que el Xbox hace de cliente. Dicho servidor le suministra al Xbox la información referente a las carpetas y archivos disponibles en el sistema donde se ejecuta el servidor, para que luego mediante una interfaz gráfica sencilla el usuario del Xbox pueda seleccionar qué volumen cargar.

Otra conclusión importante, es que para la implementación de una aplicación de visualización de volúmenes debe usarse el *framework* XNA 3.1, ya que éste permite exportar aplicaciones para ambas plataformas, sin limitar el tamaño máximo de las texturas. Esta es una limitación que se encontró en la versión 4.0 del *framework*, que impedía el despliegue de volúmenes de tamaño superior a 256^3 (32 MB), con la versión XNA 3.1 fue posible desplegar volúmenes de hasta 64 MB.

Se realizaron una serie de pruebas de rendimiento entre las dos técnicas implementadas y su desempeño tanto en la plataforma Windows como en el Xbox, teniendo como resultado que en ambas plataformas es posible desplegar de manera eficiente un volumen y obtener buenos resultados visuales. Sin embargo, el desempeño general de los computadores utilizados en las pruebas fue mejor al desempeño de la consola Xbox debido a la superioridad de los procesadores, las tarjetas gráficas y la capacidad de memoria RAM de los computadores con respecto al Xbox, a pesar del paralelismo implícito presente en todos los equipos de prueba.

Esta diferencia tecnológica quizás viene dada por el hecho de que el Xbox es una consola que salió antes de que se popularizaran los procesadores multinúcleo, por tanto su procesador es uno de los primeros en tener varios núcleos de procesamiento mientras que las computadoras usadas como equipos de prueba tienen procesadores con tecnología multinúcleo más reciente. Esto nos hace pensar que se podría establecer una comparación que arroje resultados más

justos, utilizando el Xbox con equipos que tengan procesadores de un solo núcleo, o bien comparar el rendimiento de equipos de múltiples núcleos contra el Xbox que saldrá próximamente, del cual se espera una mejor capacidad de hardware.

Con respecto a las técnicas de despliegue utilizadas, ambas técnicas presentan buenos resultados visuales. Sin embargo, en las PC el tiempo de despliegue es mayor con Planos Alineados al Viewport que con RayCasting. Mientras que en el Xbox, el tiempo de despliegue es mayor con RayCasting que con Planos Alineados al Viewport. Dado que la única diferencia entre estas pruebas es la plataforma objetivo, atribuimos dicha incoherencia en los resultados a las diferencias de hardware entre las plataformas y/o a posibles deficiencias del compilador de .NET para traducir el código adecuadamente para las plataformas utilizadas. En este último caso, tal vez sea posible mejorar el rendimiento optimizando el código manualmente para cada plataforma.

Por otro lado, el recolector de basura de .NET no tiene el mismo rendimiento en una PC que en el Xbox. En una PC, el recolector organiza la memoria mediante un enfoque por generaciones, haciendo este proceso menos complicado y trabajoso para una gran cantidad de objetos. Esto podría verse reflejado en algunos de los resultados obtenidos durante las pruebas en cuanto a la diferencia de rendimiento entre la consola y las PC.

Una limitante en la comparación de los resultados visuales fue la capacidad de la capturadora EasyCap con la que se realizaron las pruebas cualitativas, debido a que su resolución máxima de 640x480 no permitía capturar imágenes del despliegue con mayor nivel de detalle para una comparación visual más exhaustiva. Aunado a esto, esta capturadora no tiene la misma precisión que pueda tener una capturadora HDMI dado que la salida de video es RCA.

Otro factor influyente en los resultados visuales fue el proceso automático de corrección gamma aplicado por el Xbox durante el despliegue de los volúmenes, logrando que en algunos casos la imagen se percibiera más clara y en otros más oscura.

Se demostró la posibilidad de integrar la herramienta de visualización desarrollada con la mesa del Realidad Virtual del CCG conectando esta misma salida de video RCA presente en el Xbox, con el proyector del CCG para ser visualizado en la Mesa de Realidad Virtual y ser manipulado con el mando de control del Xbox.

Sin embargo, es posible ejecutar la aplicación en una computadora y visualizarla en la Mesa de Realidad Virtual sin hacer uso del Xbox, únicamente utilizando el mando de control del Xbox para el control de la interfaz en la manipulación de la aplicación. Haciendo uso de un receptor inalámbrico es posible la comunicación entre la computadora y el mando de control de Xbox, con lo cual se evitaría el uso del rastreador cableado a través del cual se interactúa actualmente con las aplicaciones que corren en la Mesa de Realidad Virtual.

A continuación se presentan algunas posibles mejoras a la aplicación desarrollada para generar mejores comparaciones y/o expandir las funcionalidades del sistema:

- Utilizar la programación paralela mediante Hilos para la carga de datos.
- Optimizar el código manualmente para cada plataforma, para evitar las desmejoras de rendimiento producto de las deficiencias del compilador de .NET Compact Framework.

- Realizar la corrección Gamma en la plataforma Xbox para obtener mejores resultados visuales.
- Utilizar una capturadora con salida de video HDMI para obtener mayor precisión en los resultados cualitativos.
- Utilizar un proyector HDMI en la conexión con la Mesa de Realidad Virtual para obtener mejores resultados visuales y mayor precisión.
- Implementar una versión de la aplicación utilizando otro lenguaje de programación y otro API para el despliegue gráfico, con el fin de comparar el rendimiento.

Referencias

[1] C. Harvey y W. Lorensen, «Marching Cubes: A high resolution 3D surface construction algorithm,» *Computer Graphics,* vol. 21, nº 4, Julio 1987.

[2] R. Carmona y O. Rodríguez, «Cubos Marchantes: Una Implementación Eficiente,» de *XXV Conferencia Latinoamericana de Informática*, La Asunción, Paraguay, 1999.

[3] J. Schulze, M. Kraus, U. Lang y T. Ertl, «Integrating Pre-Integration Into The Shear-Warp Algorithm,» de *Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume graphics*, Tokio, Japón, 2003.

[4] P. Williams y N. Max, «A volume density optical model,» de *Proceedings of the 1992 Workshop on Volume Visualization*, Octubre 1992.

[5] H. Anton, Calculus: A New Horizon, Combined, 6ta edición ed., John Wiley & Sons Inc., 1998.

[6] H. Hans-Christian, H. Tobias y S. Detlev, «Volume Rendering - Mathematical Models and Algorithmic Aspects,» Konrad-Zuse-Zentrum Berlin, 1993.

[7] M. Artner, T. Möller, V. I y M. Gröller, «High-Quality Volume Rendering with Resampling in the Frequency Domain,» de *Proceedings of Eurographics / IEEE VGTC Symposium on Visualization*, 2005.

[8] P. G. Lacroute, «Fast Volume Rendering Using Shear-Warp Factorization of the Viewing Transformation,» 1995.

[9] L. A. Westover, «Interactive Volume Rendering,» de *Proceedings of Volume Visualization Workshop*, 1989.

[10] T. T. Elvins, «A Survey of Algorithms for Volume Visualization,» *Computer Graphics,* vol. 26, nº 3, pp. 194-201, 1992.

[11] M. Levoy, «Efficient Ray Tracing of Volume Data,» *ACM Transactions on Graphics,* vol. 9, nº 3, pp. 245-261, 1990.

[12] J. Krüger y R. Westermann, «Acceleration techniques for GPU based Volume Rendering,» de *IEEE Visualization*, 2003.

[13] O. Wilson, A. Gelder y J. Wilhelms, «Direct Volume Rendering via 3D Textures,» 1994.

[14] K. Jaegers, XNA 4.0 Game Development by Example, Birmingham: Packt Publishing Ltd., 2010.

[15] Microsoft, «HLSL,» Agosto 2011. [En línea]. Available: http://msdn.microsoft.com/en-us/library/bb509638%28v=VS.85%29.aspx.

[16] B. Nitschke, Professional XNA Game Programming: For Xbox 360 and Windows, Indiana: Wiley Publishing, Inc., 2007.

[17] M. y. K. K. Shiratuddin, «Interactive Home Design in a Virtual Environment,» de *Conference on Applied Virtual Reality in Engineering and Construction Applications of Virtual Reality*, 2007.

[18] M. Pinho, L. Dias, C. Antunes, E. Khodjaoghlanian, G. Becker y L. Duarte, «A user interface model for navigation in VR environments,» de *CyberPsychology & Behavior*, 2002.

[19] S. O'Keeffe y M. Shiratuddin, «Utilizing XNA and Xbox 360 Game Console as a Sustainable Streetscape Spatial Review Tool in a Virtual Environment,» Hattiesburg, 2008.

[20] A. Kaplanyan, «Light Propagation Volumes in CryEngine3,» de *SIGGRAPH*, New Orleans, 2009.

[21] S. Hill y D. Collin, «Practical, Dynamic Visibility for Games,» de *GPU Pro 2: Advanced Rendering Techniques*, Wolfgang Engel, 2011, pp. 329-347.

[22] J. Andrews y N. Baker, Xbox 360 System Architecture, IEEE Computer Society, 2006.

[23] T. Sousa, N. Kasyan y N. Schulz, «CryEngine 3,» de *GPU Pro 3: Advanced Rendering Techinques*, A K Peters/CRC Press, 2012, pp. 140-142.

[24] M. Pettineo, «The Danger Zone: Correcting XNA's Gamma Correction,» 2009. [En línea]. Available: http://mynameismjp.wordpress.com/2009/12/31/correcting-xnas-gamma-correction/.

[25] C. Carter, Microsoft XNA Game Studio 3.0 Unleashed, Sams, 2009.

[26] H. Chaplin y A. Ruby, Smartbomb: The Quest for Art, Entertainment, and Big Bucks in the Videogame Revolution, Algonquin Books, Noviembre 2004.

[27] K. Engel, M. Kraus y T. Ertl, «High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading,» Alemania.

[28] E. LaMar, B. Hamann y K. I. Joy, «Multiresolution Techniques for Interactive Texture-Based Volume Visualization,» California.

[29] H. Schildt, Fundamentos de C# 3.0, Mc Graw Hill, 2009.

[30] E. Tabellion y A. Lamorlitte, «An Approximate Global Illumination System for Computer Generated Films,» de *SIGGraph'04*, 2004.