Universidad de las Ciencias Informáticas

Facultad 5

Título: Generación en tiempo real de pequeñas deformaciones por colisiones utilizando bump maps.

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

Autores: Erislandys Igarza Castro

Yury Antonio Rodríguez Cruz

Tutores: Ing. Hassán Lombera Rodríguez

Ing. Yoander Cabrera Díaz

Ciudad de la Habana, 15 de junio de 2009

"Año del 50 aniversario del triunfo de la Revolución"

DATOS DE CONTACTO DE LOS TUTORES

Nombre y Apellidos: *Hassán Lombera Rodríguez*

Edad: 24

Ciudadanía: *cubana*

Institución: *Universidad de las Ciencias Informáticas (UCI)*

Título: *Ingeniero en Ciencias Informáticas*

Categoría Docente: *Profesor en adiestramiento*

E-mail: *hlombera@uci.cu*

Ocupación Actual: *Profesor del Departamento de Gráfico por Computadoras*

Nombre y Apellidos: *Yoander Cabrera Díaz*

Edad: 25

Ciudadanía: *cubana*

Institución: *Universidad de las Ciencias Informáticas (UCI)*

Título: *Ingeniero en Ciencias Informáticas*

Categoría Docente: *Profesor Instructor*

E-mail: *ycabrerad@uci.cu*

Ocupación Actual: *Profesor del Departamento de Gráfico por Computadoras*

Una vez antes de entrar en la UCI una buena amiga me dijo:"La Universidad cambiara tu vida por completo". Hoy cuando concluyo con mi vida de estudiante en la UCI con la mayoría de mis metas cumplidas y todo un cúmulo de buenas experiencias, doy las gracias a todas las personas que de una forma u otra han hecho que la advertencia de mi amiga se convirtiera en realidad de la forma más positiva posible.

Agradezco especialmente a toda mi familia donde el amor de padres tíos abuela y familia en general siempre fue y será el motor impulsor en mi vida.

Gracias a mis tutores los que antes que todo son mis amigos, a mi "aguerrido" compañero de tesis y a tantas otras personas especiales que han estado a mi lado ayudándome, aguantándome y otras muchas queriéndome, lo que en su mayoría ha sido mutuo.

Una de las primeras cosas que escuché de un buen profesor aquí en la UCI fue "Si ustedes se pudieran ver por un hueguito como serán en un futuro no lo creerían", tenía mucha razón nunca pensé entre otras cosas, contar con tantas especiales.

A todos muchas gracias.

Eri

Primeramente quisiera agradecerles a mis tutores, por la dedicación y la confianza en nosotros, en especial a Hassan por tanta paciencia y ante todo por ser mi amigo.

Un agradecimiento especial a María Antonia, por ayudarme a realizar mi sueño, estudiar en la UCI.

Mil gracias a nuestro Comandante Fidel por crear esta universidad que me ha formado no solo como intelectual sino también como un hombre de bien.

A Luisito, por su dedicación y apoyo.

A todos mis amigos, por confiar siempre en mí, A la FEU por haberme enseñado tanto. A mis compañeros del proyecto Juego Consola que tanto significa para mí.

A Omar, Mariyta, Pedro y Sandra, por su apoyo y acogerme siempre como una familia más.

A mi vida Raque, por su paciencia, comprensión y amor.

A mi abuela Aya, por enseñarme desde pequeño todo lo que sé.

A Tata, por el cariño que siempre me ha dado.

A mi hermanito y mi hermanita por quererme tanto.

A mi papá, por ser mi guía y mostrarme el camino de la verdad.

A mi mamá, por haberme querido tanto y enseñarme que solo el amor engendra la maravilla.

Quisiera dedicarle este trabajo principalmente:

A mi mamá, a quien quiero con el alma. A mi papá, por ser todo para mí.

A mi hermana y mi hermano, por constituir otra parte de mi cuerpo.

A mi abuela Aya, por salvarme la vida un día, gracias a ti estoy aquí.

A mi abuela Tata, por malcriarme y estar siempre pendiente de mí.

A mi amiga Raque como un día le dije, y ahora más que eso, MI AMOR

Yury

RESUMEN

La presente investigación propone un novedoso método para la generación de pequeñas deformaciones sobre superficies de objetos sólidos sin transformar la geometría de los mismos, con la utilización de la técnica *bump mapping,* apoyada con *normal mapping*. Dicho método propicia realismo en las escenas, con eficiencia del *rendering* con bajo costo computacional, pues se libera la Unidad de Procesamiento Central del cálculo de la deformación subyacente durante la colisión a partir de la explotación de la Unidad de Procesamiento Gráfico para este propósito.

Para la compresión del método se exponen modelos y formas de representar los objetos deformables. Del mismo modo, se describen también su incorporación en el mundo de los gráficos por computadora y su representación. Se realiza una breve explicación física de las deformaciones de objetos y de la resistencia de materiales, aplicándolos a la computación gráfica. Se incluye tópicos sobre la luz, los modelos de iluminación con el uso de las tarjetas gráficas y cómo puede ser su explotación en el mundo de los videojuegos. Se hace referencia a la utilización del mapeado de textura y las técnicas de gráficos computacionales basadas en iluminación, así como sus resultados visuales en las escenas virtuales. Se menciona además el entorno de desarrollo seleccionado para realizar la implementación, abundando sobre sus características. Posteriormente, se hace referencia a las bibliotecas gráficas capaces de satisfacer las necesidades de desarrollo. De igual forma se analizarán los motores físicos o bibliotecas físicas de alto rendimiento disponibles, pues constituyen el núcleo para una simulación realista.

Para mostrar la funcionalidad del método se utilizó el motor gráfico *Graphics Three Dimensional Engine* y el físico *Open Dynamics Engine*. La descripción de las clases necesarias para demostrar la utilidad del método, así como los patrones utilizados para su diseño, son temas que también se abordan como parte de la investigación. Por último se presentan los resultados obtenidos y los aportes más significativos de los mismos.

Palabras Clave:

pequeñas deformaciones, *bump mapping, normal mapping.*

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

Un hito importante en el inicio de los videojuegos tuvo lugar en 1971 cuando Nolan Bushnell comercializó *"Computer Space".* Durante los años siguientes se implantaron numerosos avances técnicos que tributaron directamente al desarrollo de los videojuegos como: los microprocesadores y los chips de memoria. A principios de los años 90 en la llamada "Revolución del 3D", las videoconsolas dieron otro importante salto técnico, hasta llegar a la actualidad donde los desarrolladores de esta industria luchan por alcanzar mayor interactividad y realismo en las escenas.

La simulación de objetos deformables puede incrementar de forma considerable el nivel de aceptación de muchas aplicaciones en el campo de la Computación Gráfica. Los mismos han sido estudiados por un período cercano a dos décadas pero con objetivos bien diferentes, que han guiado el desarrollo de diversos modelos deformables para la obtención de varias animaciones como: la simulación de ropa y las expresiones faciales.

En las animaciones computacionales para los largometrajes, el realismo físico es con frecuencia el aspecto más importante. Sin embargo, en aplicaciones interactivas como los videojuegos, la eficiencia y la robustez para lograr comportamientos creíbles son los aspectos dominantes.

La deformación de estos objetos puede ocurrir entre otras causas por la acción de fuerzas externas ejercidas durante la colisión con el entorno. Dentro de las técnicas existentes para simular estos objetos se encuentran las que se han centrado en calcular una deformación global de manera estable, precisa y eficiente, mientras otras han apostado por la simulación eficiente de los efectos en los puntos de contacto. Así, tanto los algoritmos de detección de colisiones como los modelos de respuesta para las mismas juegan un papel central en la simulación de objetos deformables.

Estas simulaciones suelen ser muy costosas en término computacional. Por lo general, en una simulación de tiempo real la Unidad de Procesamiento Central contiene amplia carga debido a la lógica del juego, el procesamiento de datos de entrada y salida, la simulación física, la inteligencia artificial, el recibo y envío de paquetes por la red.

Desde finales de la última década del siglo pasado, las industrias dedicadas a la fabricación de computadoras y equipos eléctricos de videojuegos, han puesto su atención en la Unidad de Procesamiento Gráfico para optimizar el *rendering* de las escenas. Los *shaders* facilitan la programación gráfica, permiten mayor flexibilidad en el uso de las características avanzadas de la tarjeta gráfica a la vez que aligeran la carga de la Unidad de Procesamiento Central. La técnica que ha provocado un significativo incremento del realismo en los Entornos Virtuales es el Mapeado de Textura combinado con técnicas de gráficos computacionales basadas en iluminación, por medio de este se pueden representar apariencias sumamente complejas sin la necesidad de incrementar la geometría. Dentro del las técnicas basadas en iluminación existen una gran variedad de modelos específicos, cada uno con su propósito bien definido. Es posible citar por ejemplo los: *light mapping, environment mapping, displacement mapping, mip mapping, normal mapping, relief mapping, parallax mapping* y *bump mapping*, los cuales tienen en común la utilización de la proyección y reflexión de la luz como efecto óptico. El modelo *bump mapping* aporta más realismo a las imágenes virtuales, sin transformar la geometría, mediante algoritmos de sombreado y basado en la transformación del vector normal de la reflexión de la luz. Utilizando la Unidad de Procesamiento Gráfico son cambiados los colores y la iluminación en cada *texel*, provocando así un efecto visual de tridimensionalidad en una textura bidimensional, cuando en realidad este no existe.

Los sistemas complejos requieren con frecuencia modelos híbridos, que posibiliten alcanzar los objetivos propuestos con una solución más eficiente. Precisamente, esta estrategia será utilizada en la presente investigación, en la cual conceptos como resistencia de materiales, los cuerpos rígidos, modelo de detección de colisiones, los *shaders* y el mapeado de textura con el enriquecimiento que le aportan las técnicas gráficas de computación basadas en iluminación, podrán combinarse de manera que produzcan un efecto visual de pequeñas deformaciones y melladuras en superficies de objetos sólidos debido a las colisiones. Las mismas serán producidas en tiempo real garantizando así un nivel de realismo adecuado y una respuesta inmediata en virtud del principio newtoniano causa-efecto.

Situación Problemática

El proyecto “Juegos Consola” de la Facultad 5 perteneciente a la Universidad de las Ciencias Informáticas, especializado en el desarrollo de juegos virtuales, actualmente cuenta con una segunda versión de un juego de carreras de automóviles. La misma, no contiene efectos especiales como las deformaciones en superficies de objetos en tiempo real, que permitan incluirle al juego un realismo más perfeccionado ante

las colisiones. Esto sin dudas lo convertiría en un producto más competitivo y atractivo. Por esta razón se impone la búsqueda de una solución para esta problemática.

Problema Científico

¿Cómo realizar en tiempo real la simulación de abolladuras y melladuras creíbles en superficies de objetos debido a las colisiones en un videojuego?

Objeto de Estudio

Las técnicas de tiempo real para la deformación de objetos, empleadas en los videojuegos.

Objetivo General

Desarrollar un método basado en la técnica de *bump mapping*, para la simulación de pequeñas deformaciones sobre superficies de objetos, debido a las colisiones.

Objetivos Específicos

1- Detectar las colisiones mediante la utilización del motor físico *Open Dynamics Engine*.

2- Calcular la deformación del *bump map* durante la colisión, utilizando la unidad de procesamiento gráfico.

3- Representar pequeñas deformaciones mediante la técnica de *bump mapping* en sustitución de las deformaciones de mallas.

4- Probar la utilidad del método propuesto mediante la ejecución en tiempo real de un ejemplo.

Campo de Acción

Las técnicas de tiempo real empleadas en los videojuegos para la simulación de pequeñas deformaciones en las superficies de objetos.

Idea a Defender

La técnica de *bump mapping* utilizando *shaders*, aplicada a la simulación de las abolladuras y melladuras en la superficie de objetos sólidos, aportará realismo a las escenas de los videojuegos y mayor velocidad de cómputo para simular las mismas.

Tareas Investigativas

1. Análisis de logros y limitaciones en los enfoques existentes sobre las deformaciones en objetos.
2. Evaluación del contenido de la información obtenida sobre las deformaciones en objetos.
3. Diagnóstico de las tendencias actuales y tomar posición al respecto.
4. Análisis de las bases teóricas del motor físico *Open Dynamics Engine.*
5. Dominio del motor físico *Open Dynamics Engine* para la detección de colisiones.
6. Caracterización de los modelos de deformación de sólidos para la selección del más adecuado.
7. Análisis de la técnica de *bump mapping* para su total comprensión.
8. Caracterización de los modelos de iluminación local, global y sombreados poligonales para la selección del más adecuado.
9. Selección de las tecnologías y el lenguaje de programación para el hardware gráfico.
10. Diseño e implementación de la arquitectura de clases que de solución al problema científico.
11. Diseño e implementación de un entorno virtual de prueba para la demostración del cumplimiento de los objetivos mediante un ejemplo.

Con la realización de este trabajo se pretende proveer a la comunidad de desarrolladores de videojuegos en la Universidad de las Ciencias Informáticas, de un novedoso método para la generación eficiente de abolladuras y melladuras creíbles sobre superficies de objetos con la utilización de la técnica *bump mapping* en sustitución de las modificaciones de mallas. Dicho método propiciaría un incremento del realismo en las escenas, con eficiencia del *rendering* a menor costo computacional, pues se libera la Unidad de Procesamiento Central del cálculo de la deformación subyacente durante la colisión a partir de la explotación de la Unidad de Procesamiento Gráfico para este propósito.

El trabajo está estructurado en cuatro capítulos.

En el Capítulo 1 denominado *Fundamentación Teórica*, se introducen todo los conceptos necesarios para la comprensión del presente trabajo y se recogen los antecedentes del tema tratado a nivel internacional,

nacional y de la UCI. Se incluye además un estudio de las tendencias, técnicas, tecnologías, metodologías y software usados en la actualidad, en los que se apoya la solución propuesta.

En el Capítulo 2 titulado *Solución Técnica* se explicará en qué consiste el método propuesto, así como los pasos a seguir para la utilización del mismo.

En el Capítulo 3 nombrado *Descripción de la Solución Propuesta*, utilizando el Lenguaje Unificado de Modelado (UML) quedará plasmada la solución ingenieril de dicha propuesta de solución.

En el Capítulo 4 titulado *Resultados,* expondrá el alcance del método, sus potencialidades, las dificultades y los aportes realizados al mismo.

CAPÍTULO 1. FUNDAMENTACIÓN TEÓRICA

En el transcurso de los años las deformaciones han invadido una amplia gama de campos tales como: la medicina, el diseño industrial, la Física, la Química, y en el campo nuclear, hasta llegar a la industria del entretenimiento. El desarrollo del *hardware* ha traído como resultado la evolución de los simuladores en tiempo real y los videojuegos, en cuanto a calidad y alta precisión de las deformaciones de objetos en tiempo real y los efectos especiales asociados a estos.

En el presente capítulo se expondrán modelos y formas de representar los objetos deformables. Del mismo modo, será también descrita su incorporación en el mundo de los gráficos por computadora y su representación. Se abordará una breve explicación física de las deformaciones de objetos y de la resistencia de materiales, aplicándolos a la computación gráfica.

Se incluyen además tópicos sobre la luz, los modelos de iluminación con el uso de las tarjetas gráficas y cómo pudiera ser su explotación en el mundo de los videojuegos. Se abordará sobre las utilizaciones actuales de las técnicas de mapeado de textura y las Técnicas de Gráficos Computacionales Basadas en Iluminaciones (TGCBI) así como sus resultados visuales en las escenas virtuales. Se hará referencia al entorno de desarrollo escogido para realizar la implementación, abundando sobre sus características. Posteriormente, se hará referencia a las bibliotecas gráficas capaces de satisfacer las necesidades de desarrollo. De igual forma se analizarán los motores físicos o bibliotecas físicas de alto rendimiento disponibles, pues constituyen el núcleo para una simulación realista.

1.1. Objetos deformables

Cuando se habla de objetos deformables, se refiere a objetos que, debido a sus propiedades físicas, su forma puede ser modificada por la acción de otro cuerpo o agente externo (gravedad, viento, pinza, mano), en fin, por todo aquel objeto que se deforme por un choque o colisión en el entorno virtual, (ver figura 1). La idea de la deformación es aparentemente simple, bastaría sólo con determinar qué parte del objeto sufre cambios ante una colisión y calcular la nueva posición de cada vértice del área afectada, sin

embargo, esa nueva posición depende de una serie de parámetros físicos o geométricos, dependiendo de la técnica de deformación usada, lo cual requiere de un cálculo muy costoso (1).

Figura 1: Simulación eficiente de tela (2).

El modelado y la simulación de objetos deformables tienen una larga historia en las ciencias de los materiales y la ingeniería. En los Gráficos por Computadoras (GxC), los objetos deformables se han estudiado durante casi dos décadas, pero con diferentes objetivos. En aplicaciones gráficas, la principal preocupación suele ser la eficiencia computacional al generar comportamientos creíbles, más que la predicción precisa de los resultados exactos. En la medida en que la simulación se ve realista, la simplificación se considera aceptable. Los modelos deformables se han utilizado en amplias variedades de aplicaciones gráficas, incluyendo las animaciones de tela, expresiones faciales, y en general, en el modelado de los objetos no rígidos.

Otras de las aplicaciones de los modelos deformables están en los sistemas en tiempo real tales como: la cirugía, la formación virtual y la escultura, donde el usuario modifica interactivamente modelos deformables. En los sistemas de tiempo real como los simuladores, la velocidad y la estabilidad son las dos principales preocupaciones. Un enfoque hacia el cumplimiento de los objetivos y la robustez del método propuesto es utilizar los modelos físicos simplificados. Sin embargo es difícil e importante expresar las propiedades de los materiales con estos enfoques.

En la actualidad, para realizar las deformaciones de objetos en tiempo real, es utilizada la tradicional deformación de malla o la utilización de los puntos como primitiva gráfica. Los autores Kaufmann y

Teschner, muestran cómo pudieran realizarse deformaciones de mallas en tiempo real (3) (4). Mientras que el autor Grossman, escoge la revolucionaria idea de utilizar los puntos como primitiva gráfica, obteniendo magníficos resultados (5). En cualquiera de los casos, el cálculo de la nueva posición para cada punto o vértice de la malla es sumamente costoso, más aún cuando se está en presencia de escenarios muy complejos con miles de millones de polígonos. A lo largo de la historia se han realizado diversas demostraciones de la certeza y exactitud de ambos métodos, los cuales a pesar del avance tecnológico del *hardware* siguen siendo muy costosos para los sistemas gráficos en tiempo real.

Actualmente en los simuladores y videojuegos es combinado un grupo de modelos, métodos y técnicas, que juegan un papel fundamental en la simulación virtual de los efectos especiales debido a una colisión. En ocasiones en los videojuegos son mezcladas las deformaciones de mallas prediseñadas con los sistemas de partículas, para así optimizar el uso del *hardware* y tener una representación visual adecuada del mundo físico real (2). Mientras que en los simuladores como los quirúrgicos la malla es deformada en tiempo real, lo cual es mucho más complicado y costoso, pero necesario por motivos vitales.

En la Facultad 5 (F5) de la Universidad de las Ciencias Informáticas (UCI), se han realizado algunos trabajos al respecto: la tesis de Raissel Ramírez Orozco enfocada en el simulador quirúrgico (1), y la continuación de la misma realizada por Yerandi Marcheco Díaz las deformaciones específicamente con el algoritmo *Chainmail*, para el propio simulador (6). Ambos transforman la geometría de los cuerpos.

1.1.1. Representación de las deformaciones

En la medida en que se tratan de simular los efectos de deformación con más exactitud, se hacen más costosos los cálculos computacionales asociados. Con los avances tecnológicos actuales, la demora para realizar estos cálculos no resulta razonable para algunas aplicaciones como los videojuegos, donde lo primordial es obtener un tiempo de respuesta adecuado. Por esta razón, en ocasiones resulta conveniente sacrificar la precisión con que ocurre un fenómeno con tal de alcanzar un resultado aceptable en un tiempo razonable.

En busca de la optimización para representar las deformaciones en los videojuegos, se han propuesto métodos según el nivel de detalle de las deformaciones. Estos niveles de detalles se caracterizan por: las deformaciones en el nivel de micro-estructuras utilizando *Bidirectional Reflectance Distribution Function*

(BRDF), donde en la investigación de Remo Ziegler se realiza una perfecta demostración mediante BRDF (7). Representa mínimas deformaciones en los objetos sin necesidad de realizar cálculos complejos o muy extensos, permitiendo un uso moderado del hardware.

Varios autores han presentado artículos para las transformaciones en la meso-estructura, en las cuales se realizan pequeñas deformaciones en la superficie de los objetos. Morgan McGuire propone un nuevo método para pequeñas deformaciones en superficies de objetos sin transformar la geometría de los mismos (8). Mediante la utilización del mapeado de textura, enriquecido con técnica de iluminación *bump mapping* y *normal mapping,* representa pequeñas deformaciones. Por otra parte los autores Shen y Willis muestran la combinación de la imagen original proyectando el mapeado de textura en el volumen del objeto, teniendo resultados eficientes en cuanto al *rendering* de las imágenes (9).

Amplia variedad de algoritmos de deformación en la macro-estructuras han sido propuestos en la literatura del pasado. Ejemplo de esto se encuentra en la investigación de Matthias Müller, donde se propone un método para la deformación de materiales usando mallas volumétricas (10). Mientras que Rezk-Salama propuso una forma de utilizar la Unidad de Procesamiento Gráfico (GPU) para acelerar las deformaciones, moviendo el cálculo de la alteración de la malla al hardware gráfico (11). Importantes contribuciones en este mismo sentido realizaron los autores Govindaraju y Mosegaard en (12) (13) respectivamente. En cualquiera de los casos es necesario el análisis y procesamiento de los estados y parámetros físicos de los materiales, así como la simulación de las colisiones entre los objetos.

1.1.2. Modelos físico-matemáticos subyacentes.

La mayoría de las aplicaciones de simulación tanto 2D como 3D en las que intervienen deformaciones, han sido modeladas utilizando los Métodos basados en Mallas Lagrangianas, tales como: Sistemas Masa-Resorte, el Método de Elementos Finitos (MEF) o *Boundary Element Method* (BEM) conjugados con la teoría de la elasticidad. Sin embargo, las aplicaciones basadas en puntos se han convertido en una popular técnica. Una razón clave de este nuevo interés en los puntos, es que la complejidad poligonal de los modelos gráficos se ha incrementado dramáticamente en la última década, lo que lógicamente hace mucho más complejos los enmallados tradicionales (1).

El uso de los puntos como primitiva gráfica, es un método más en la búsqueda de velocidad y simplicidad del *rendering*. Dibujar un punto es extremadamente fácil, este no necesita cortar polígonos, computarizar conversiones, ni mapear texturas. El dibujo es similar a la imagen original, necesita gran cantidad de memoria para representar objetos complejos, esta es una manera de evitar el largo tiempo de *rendering* de los polígonos. Por otra parte al no utilizar las diferentes modalidades de las TGCBI y los *shaders*, no explotan las potencialidades del *hardware* gráfico y se queda atrás en las representaciones más realistas (5).

Si bien la mayoría de las técnicas son basadas en parámetros físicos, no se puede obviar la existencia de técnicas puramente geométricas, que también han sido empleadas con distinto grado de éxito, sobre todo en el campo de la eficiencia computacional. Los métodos basados en la geometría modifican la forma de los objetos indirectamente, a través de puntos de control o ajustando parámetros en la función que define la superficie. Estos métodos garantizan un buen rendimiento, pero no modelan las propiedades físicas del material, ni emplean leyes físicas para calcular la nueva forma del cuerpo. Entre ellas se destacan los *Splines* y Ajustes de Curvas, Deformaciones de Libre Forma (DLF), el cual fue introducido por T. Sederberg y S. Parry en 1986. Este un método muy rápido para representar y modelar objetos flexibles basados en deformaciones del espacio que los contiene (14).

1.1.3. Introducción a la Resistencia de Materiales.

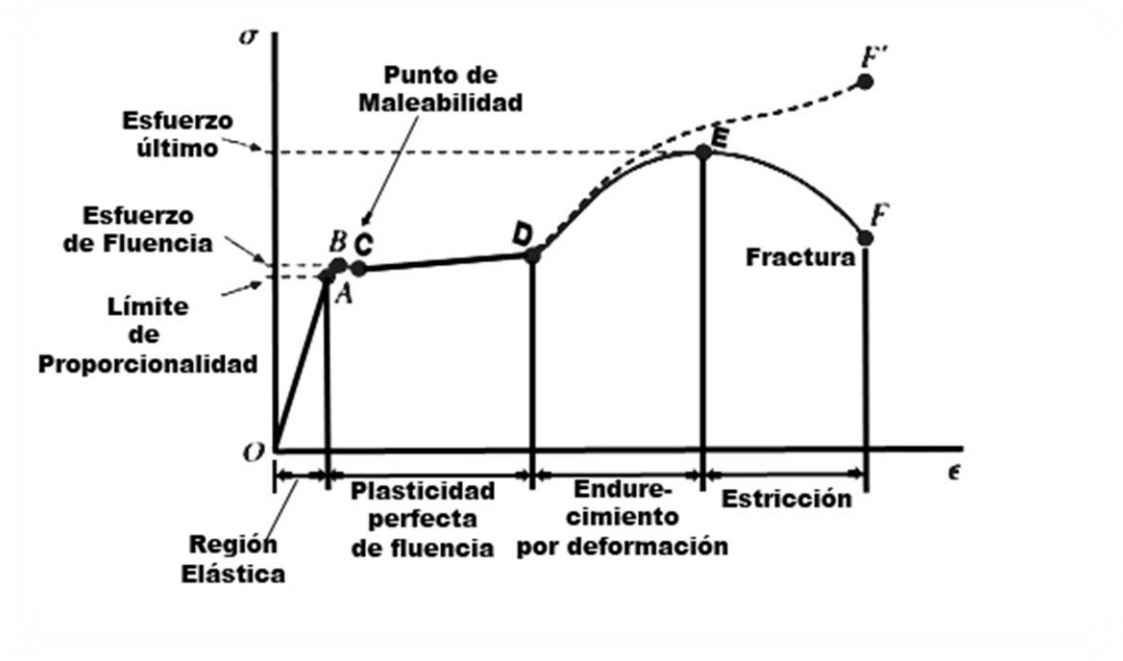
La mecánica de materiales es una rama de la mecánica aplicada que estudia el comportamiento de los cuerpos sólidos sometidos a diversas cargas. Este campo de estudio tiene otros nombres, como resistencia de materiales y mecánica de cuerpos deformables. El objetivo principal de la mecánica de materiales es determinar los esfuerzos, deformaciones unitarias, desplazamientos en sus estructuras y sus componentes debido a las cargas que actúan sobre ellos. Personajes como Leonardo da Vinci y Galileo Galilei efectuaron experimentos para determinar la resistencia de alambres, barras y vigas, aunque no formularon teorías adecuadas para explicar los resultados de sus pruebas. En contraste, Leonard Euler, famoso matemático, concibió la teoría matemática de las columnas y calculó, en 1744, la carga crítica de una columna, mucho antes de que existieran pruebas experimentales que demostraran la importancia de sus resultados. Sin las pruebas adecuadas que respaldaran su teoría, los resultados de

Euler permanecieron sin aplicación durante más de 100 años, aunque hoy son la base del diseño y el análisis de las estructuras constructivas. (15)

Los conceptos fundamentales en la resistencia de materiales son presión y deformación unitaria. La presión es unidad de fuerza por unidad de área y se representa con la letra σ (sigma) $\boldsymbol{\sigma} = d\mathbf{P} / d\mathbf{A}$. En general las presiones σ que actúan sobre una superficie plana pueden ser uniformes en el área o pueden variar de intensidad de un punto a otro. Mientras tanto, las deformaciones unitarias, son las que se producen en los materiales debido a una presión.

La curva de Presión vs. Deformación (Pvs.D) caracteriza el comportamiento de un material. Puede ser también utilizada para describir cualitativamente y clasificar un material. Las regiones típicas que pueden ser observadas en una curva de este tipo se muestran en la figura 2.

Figura 2: Curva de Presión vs Deformación (15).



Es necesario enfatizar que las dimensiones de cada región en una curva de Pvs.D es dependiente del material en cuestión y no todos los materiales exhiben todas las regiones.

Cuando una pieza se somete a una fuerza de tensión, se produce una deformación del material. Si el material vuelve a sus dimensiones originales cuando la fuerza cesa se dice que el material ha sufrido una deformación elástica. El número de deformaciones elásticas en un material es limitado, sus átomos son desplazados de su posición original, pero no hasta el extremo de que tomen nuevas posiciones fijas (15). Así cuando la fuerza cesa, los átomos vuelven a sus antiguas posiciones, adquiriendo su forma original. Si el material es deformado hasta el punto que los átomos no pueden recuperar sus posiciones, se dice que ha experimentado una deformación plástica.

Región elástica

En el contexto del comportamiento de los materiales, un componente estructural se dice que se comporta elásticamente si durante un proceso de carga y descarga sobre el mismo la deformación ocurrida es

reversible. Por el contrario, si el material no retorna a su estado inicial en el proceso de descarga, se dice que se comportó inelásticamente. La mayor parte de la región elástica mostrada en la figura 2 corresponde a una respuesta linealmente elástica (presión linealmente proporcional a la deformación). La ley de Robert Hooke plantea que la relación lineal entre la presión y la deformación unitaria, se expresa con la ecuación $\boldsymbol{\sigma} = \mathbf{E} \cdot \boldsymbol{\varepsilon}$ donde $\boldsymbol{\varepsilon}$ es la deformación unitaria axial y $\mathbf{E}$ una constante de proporcionalidad llamada módulo elástico. El módulo elástico es la pendiente del diagrama presión-deformación unitaria en la región linealmente elástica (15).

Es también posible que los materiales tengan un comportamiento elástico pero no linealmente elástico. En la figura 2 se muestra el límite de la proporcionalidad donde la presión se incrementa de manera no lineal cuando crece la deformación. Existen además algunos materiales para los cuales la región elástica es en su mayoría no lineal a la deformación.

Región de Plasticidad Perfecta

Cualquier incremento en la deformación más allá del punto de fluencia causará que el material se deforme permanentemente. Además en esta región, la deformación será relativamente larga para pequeños y casi imperceptibles incrementos de presión. Este proceso es con frecuencia conocido como plasticidad perfecta. El material se vuelve perfectamente plástico, para su mejor comprensión, se deforma sin que aumente la carga aplicada.

Región de Endurecimiento por Deformación

Cuando la carga es llevada mas allá de la región de fluencia, se necesita un incremento de esta para que ocurra una deformación adicional. Este efecto es conocido como endurecimiento por deformación y está asociado a un incremento de la resistencia del material a la deformación. Finalmente, la curva de Pvs.D alcanza un máximo en un punto conocido como “Esfuerzo máximo”. Para muchos materiales la reducción del área de su sección transversal no es perceptible a simple vista hasta que este punto no se sobrepasa.

Estricción y Fractura

Cuando la carga es continuada más allá del esfuerzo máximo, el área de la sección transversal disminuye drásticamente en una región localizada del material. Debido a la disminución, la capacidad de carga de la región se reduce. La carga cae al igual que la tensión, hasta que el material alcanza el punto de fractura.

1.2. Modelos de Iluminación

En la naturaleza se pueden ver las cosas porque reflejan la luz proveniente de una fuente, o porque ellos mismos son una fuente. El ser humano no está capacitado para ver un objeto a menos que esté iluminado o emita luz. Las sombras ayudan a definir las relaciones espaciales entre objetos en una escena. Un aspecto a tomar en cuenta para mejorar el realismo en los entornos virtuales, es el correcto manejo de luces y sombras.

Una escena carece de realismo sin una iluminación correcta. La misma se consigue mediante la ubicación de varias fuentes de luz que interactúan sobre los objetos, teniendo en cuenta las propiedades de los materiales de que estén constituidos. La luz es importante en la creación de videojuegos, pues promueve la sensación de tridimensionalidad de los objetos. La forma en que el objeto refleja la luz da una idea del material del cual está formado (16).

1.2.1. La Luz

El entorno está rodeado de fenómenos ópticos. Cuando la luz es emitida desde un foco luminoso, esta es reflejada desde innumerables objetos antes de alcanzar los ojos del espectador. Cada vez que es reflejada, una parte de la energía es absorbida por la superficie, otra parte es dispersada en direcciones aleatorias y el resto se dirige a otra superficie o a los ojos del espectador (16). El proceso anterior se repite hasta que la energía se reduce a cero o el espectador percibe la luz. Fue Newton quien mostró que la luz blanca es una superposición de ondas con diferentes frecuencias, descomponiéndola de forma artificial mediante un prisma.

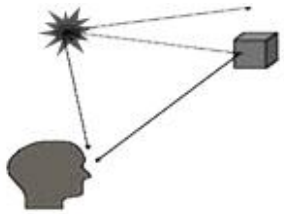
Cuando la luz incide sobre una superficie muy lisa, esta se refleja especularmente. Cuando el área se torna rugosa, la luz se refleja no solamente con una componente especular sino que también surge luz reflejada en forma difusa. Es a través de la luz que se puede ver la rugosidad de superficies que en

primera instancia parecerían lisas. No basta conocer solo el origen de esa luz o cuáles son sus fuentes, también es necesario tomar en consideración cómo se modifica hasta llegar a nuestros ojos (16).

1.2.2. Fenómeno Básico de la Iluminación

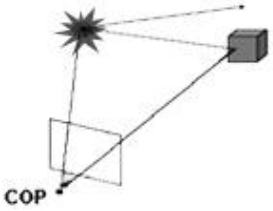
Desde una perspectiva física, la superficie de un objeto puede emitir luz de manera natural o reflejar luz de otras superficies que la iluminan, (ver figura 3). El color que se ve en un punto de una superficie está determinado por las múltiples interacciones entre las fuentes de luz y superficies reflectoras. Particularmente, los objetos que no emiten su propia luz, reciben tres tipos de luz diferentes: la luz ambiental, la luz difusa y la luz especular, conocidas también como: componente ambiental, difusa y especular (16).

Figura 3: Percepción de la luz por el ojo humano (16).



En términos de GxC, se reemplaza el observador por el plano de proyección, (ver figura 4).

Figura 4: Plano de proyección (16).



El plano de proyección y su mapeo a la pantalla significa un número particular de *pixels* de despliegue. El color de la fuente de luz y las superficies, determinan el color de uno o más *pixels* en el *frame buffer*. Se debe considerar solo aquellos rayos que dejan las fuentes y llegan al ojo del observador. La mayoría de los rayos que dejan la fuente no contribuyen a la imagen.

El sombreado de los objetos también depende de la orientación de las superficies, caracterizado por el vector normal a cada punto. Las interacciones entre luz y materiales se pueden clasificar en tres grupos que se muestran en la figura 5.

Figura 5: Interacción en luz y material (16).

Las superficies especulares se ven relumbrantes porque la mayoría de la luz reflejada ocurre en un rango de ángulos cercanos al ángulo de reflexión. Los espejos son superficies especulares perfectas. La reflexión del rayo de luz entrante puede absorberse parcialmente, pero toda la luz reflejada aparece en un solo ángulo, obedeciendo la regla de que el ángulo de incidencia es igual al ángulo de reflexión.

Las superficies difusas se caracterizan por reflejar la luz en todas las direcciones. Paredes pintadas con mate son reflectores difusos. Superficies difusas perfectas dispersan luz de manera igual en todas las direcciones y tienen la misma apariencia a todos los observadores.

Las superficies translucidas permiten que parte de la luz penetre la superficie y emerja de otra ubicación del objeto. El proceso de refracción caracteriza el vidrio y el agua. Cierta luz incidente puede también reflejarse en la superficie.

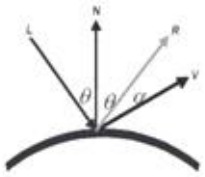
1.2.3. Modelo de Iluminación Local

Los modelos de iluminación se utilizan para calcular el color o intensidad de la luz que se percibe desde un punto determinado en la superficie de un objeto. Son métodos simplificados que se basan en las propiedades ópticas de las superficies, las condiciones de la luz ambiente o de fondo, y las características de las fuentes de luz. Las propiedades ópticas de las superficies se especifican mediante parámetros que permiten controlar la cantidad de absorción y reflexión de la luz incidente. El modelo de iluminación debe

contemplar algunos de los comportamientos físicos más notorios de las superficies reales en interacción con la luz visible.

La figura 6 muestra las variables vectoriales principales que dan origen a dichas componentes. L (vector de incidencia de la luz), N (vector normal a la superficie), R (vector de reflexión) y V (vector del observador) (16).

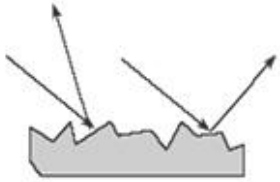
Figura 6: Variables vectoriales del mundo de la Iluminación (16).



La intensidad de la luz reflejada depende de qué tan grande sea la componente del vector normal de la superficie *N* en la dirección de la fuente de luz puntual, es decir, de qué tan parecidos sean el vector normal a la superficie del objeto en un punto determinado y el vector que representa la ubicación de la fuente de luz *L* en el espacio de coordenadas de la escena. Esto hace que los polígonos de un objeto que se encuentren expuestos más directamente a la fuente de luz se vean más brillantes que los que reciben la iluminación de una manera más sesgada.

Un reflector difuso perfecto esparce la luz que refleja de manera igual en todas las direcciones, viéndose igual para todos los observadores. Sin embargo, la cantidad de luz reflejada depende del material dado, que parte de que la luz es absorbida. Reflexiones difusas son caracterizadas por superficies rugosas.

Figura 7: Superficie rugosa (16).



El proceso de calcular el componente especular es más costoso y lento que el cálculo de la intensidad difusa y ambiental, pero adiciona un alto realismo a la escena, brindando apariencias metálicas, rugosas, húmedas, así como muchos otros efectos visuales en los objetos. La iluminación de las escenas, no solo depende de las propiedades de los focos luminosos, sino también del modelo usado para calcular los componentes de la luz. Estos modelos han sido optimizados en los últimos tiempos, explotándose las tarjetas gráficas mediante los *shaders*.

1.3. Shaders

Los *shaders* en tiempo real están en el corazón de todos los nuevos efectos visuales y continuarán siendo la fundación de una experiencia gráfica asombrosa para el futuro. Con la introducción de los lenguajes de sombreado de alto nivel de Directx y *Open Graphics Library* (OpenGL), la complejidad de los *shaders* en tiempo real ha aumentado en gran medida, la rápida evolución de la capacidad de las tarjetas gráficas, ha causado una explosión en la cantidad de contenido del *shader*, necesaria para los proyectos en tiempo real.

1.3.1. Unidad de Procesamiento Gráfico

El GPU es un procesador dedicado exclusivamente al procesamiento gráfico, para aligerar la carga de trabajo de la Unidad de Procesamiento Central (CPU). Normalmente se utiliza la GPU para ofrecer múltiples canales de ejecución, con algo de Memoria de Acceso Aleatorio (VRAM) para los *buffer*, el texturizado y alguna instrucción en específico.

Una GPU está altamente segmentada, lo que indica que posee gran cantidad de unidades funcionales. Estas unidades funcionales se pueden dividir principalmente en dos: aquellas que procesan vértices, y las que procesan *pixels*. Por tanto, se establecen el vértice y el *pixels* como las principales unidades que maneja.

La limitante fundamental que tenía el GPU, era que su estructura no se podía cambiar. Cuando se creaban estas tarjetas, los ingenieros prefijaban un conjunto de instrucciones y algoritmos en el chip de video. Los fabricantes no brindaban la posibilidad de ejecutar en estos chips, otras instrucciones que no

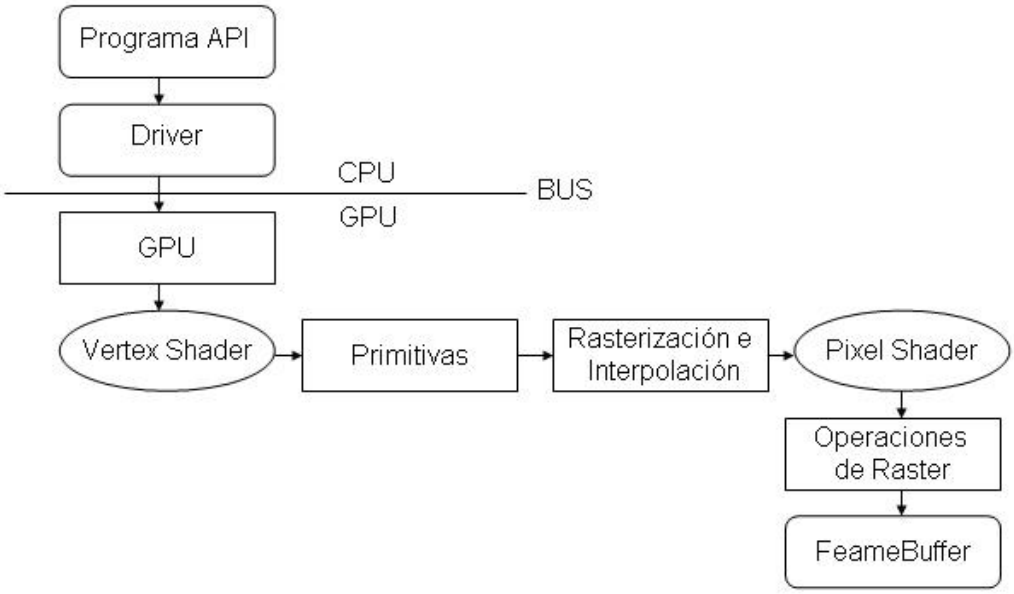
fueran las codificadas en el momento de su creación. A esta estructura estática se le denomina sistema de funciones fijas, las mismas son manejadas a través del *pipeline* gráfico.

En los últimos 10 años se ha elaborado una alternativa a las funciones fijas, para aumentar la flexibilidad gráfica en las tarjetas de video. En dos momentos claves del *pipeline* gráfico se ha concebido introducir códigos, que permitan ejecutar algoritmos no diseñados inicialmente en la tarjeta gráfica. A estos códigos se le llaman *shader*, y según su funcionamiento y lugar de ejecución se dividen en *Vertex Shader* y *Pixel Shader.* De esta forma, el uso de los *shader* permite introducir nuevos algoritmos en las tarjetas de video. Dando la posibilidad de realizar avances en la visualización virtual, sin la necesidad de esperar a que estos algoritmos sean codificados en el chip de video, para obtener así aceleración por *hardware*.

Los *Vertex Shader* son los encargados de transformar todos los vértices de la escena. En ellos se ejecutan las transformaciones de espacio-objeto a espacio-mundo, de cámara, y finalmente se obtiene la posición en la pantalla. Además, en ellos se incluyen todas las demás operaciones a nivel de vértices como son el cómputo de coordenadas de texturas, la iluminación *Per-Vertex*, y la *Per-pixel.*

La figura 8 muestra cómo trabaja el *pipeline* gráfico, donde la interfaz de Directx u OpenGL pasa los *vertex shader* a las funciones fijas, la GPU recibe estos datos o comandos transformando los vértices recibidos y realiza otras operaciones a este mismo nivel, el interpolador recibe la posición en la pantalla de cada uno de los vértices y genera los triángulos correspondientes. Al dibujar estos triángulos se calcula la posición de cada uno de los *pixels* que lo conforman (17).

Figura 8: Pipeline Gráfico Conceptual usando *Shaders* (17).



Cada uno de estos *pixels* se introduce en el *Pixel Shader* para que este calcule el color del mismo en la pantalla. De esta forma en el *Pixel Shader* se realizan todas las operaciones a nivel de *pixel* como es el cálculo de la iluminación *Per-Pixel*, el mapeo de texturas, y el uso de los mapas de normales para la determinación de la normal del *pixel*.

1.3.2. Lenguajes de Programación en Shaders

OpenGL es la biblioteca gráfica 3D por excelencia, puede utilizarse en Linux, UNIX, Mac OS, Windows e incluso en móviles. Desarrollada originalmente por *Silicon Graphics Inc.* (SGI), es una biblioteca de gráficos que ofrece al programador una interfaz de acceso a las tarjetas gráficas mediante *Application Program Interface* (API), la cual ha crecido a la par del *hardware*. Al igual que Directx, el cual posee el conocido *High Level Shading Language* (HLSL), OpenGL tiene el *GL Shading Language* (GLSL), ambos son utilizados para programar sobre el GPU (16).

HLSL no es más que una capa adicional que reside conceptualmente sobre la API existente. El lenguaje de programación HLSL de Microsoft para la GPU en Directx 9.0 trabaja solamente en Windows y consigue

altas marcas de calidad. El lenguaje de programación de alto nivel puede crear animaciones ultra realistas y efectos visuales sorprendentes, sin tener que preocuparse por el tipo específico de *hardware*, está basado en Directx 3D y no es multiplataforma, a pesar de ello es muy popular en la comunidad de programadores gráficos.

Por el contrario, la técnica adoptada en OpenGL consiste en incrustar el compilador para el lenguaje de *shaders* GLSL en el mismo driver OpenGL. Esta técnica facilita a los desarrolladores de aplicaciones que van hacer uso de lenguajes de alto nivel al estilo C o C++ la implementación de los *shaders* directamente en el subsistema gráfico.

Al contrario de los *shaders* HLSL donde todo funciona con arquitectura PC, un *shader* OpenGL es compatible con todas las plataformas y sistemas que sean compatibles con OpenGL. Permite redefinir el comportamiento por defecto del procesador de vértices y de fragmentos, presente en los sistemas OpenGL modernos. Aunque esta biblioteca no posee tanta popularidad como Directx, es la preferida de los programadores gráficos como Carmack, con la que ha desarrollado los motores del *Quake* y del *Doom* (16).

Hace poco más de una década, los programadores gráficos vienen trabajando en el lenguaje Cg (del inglés, *C for Graphics*). Este lenguaje ha sido potenciado por la NVIDIA, y los últimos tiempos ha sido el preferido de muchos programadores de la comunidad gráfica. Es un lenguaje de código abierto, multiplataforma, y se pueden desarrollar programas con las primitivas gráficas de OpenGL o Directx para múltiples sistemas operativos e incluso consolas. Los programas en Cg además de especializarse en lo que se conoce como *shading language,* se pueden realizar simulaciones físicas, composiciones, y otras tareas que no tengan que ver necesariamente con sombreado.

Un aporte significativo en busca del rendimiento computacional lo aporta C. Rezk-Salama, con la representación directa del volumen, donde en cada punto describe la absorción y emisión de la luz, explotando el GPU mediante los *shaders*. Como consecuencia, en los últimos años, se han desarrollado varios algoritmos eficientes para explotar las tarjetas gráficas con fines generales de texturizado para una rápida interpolación.

1.3.3. Mapeado de Textura y Técnicas de Gráficos Computacionales Basadas en Iluminación.

Los *shader* ofrecen una forma más detallada de trabajar los efectos especiales como la iluminación, logrando mayor realismo en los objetos. Para el cumplimiento de este objetivo, se deben utilizar técnicas capaces de darle a los objetos virtuales la mayor realidad posible. Emplear las TGCBI contribuye a un mayor realismo y a la mejora de cómputo por la enorme disminución del número de polígonos, obteniendo los mismos resultados visuales, (ver figura 9).

Figura 9: *Parallax occlusion mapping* contra geometría actual (18).

Dibujado con 1100 polígonos utilizando parallax mapping

Dibujado con 1.5 millones de polígonos

El mapeado de texturas consiste en aplicar una serie de dibujos o plantillas a las caras de un objeto tridimensional. Por ejemplo, pegando la foto del rostro de una persona en una esfera plana, la se habrá convertido en una cabeza. Lo mismo ocurre si se aplica la textura de la madera a un cilindro: se obtendrá un tronco bastante real (19).

Para que estas texturas se ajusten debidamente al objeto donde se pegan, se suele aplicar una corrección de perspectiva o textura inversa, que estira o comprime la textura según su posición, evitando cualquier anomalía y pérdida de realidad. Básicamente se trata de buscar el píxel de la pantalla que se corresponde con cada píxel de la textura (19).

Las coordenadas de la textura son asignadas a los vértices del objeto en cuestión, según diversos procedimientos. Se identifican cada uno de los *pixels* de dicho objeto durante el dibujado con uno de los *texels* de la textura para sustituir. También se pueden alterar algunas de las características de superficie del pixel original, tales como: el color, el brillo o la transparencia, por aquellas que indiquen las propias texturas. En función del tamaño visible del objeto, la resolución de dibujado y la de la propia textura, podrá resultar que un *pixel* del objeto se corresponda con varios *texels*, promediando los valores de éstos, o que un *texel* haga lo propio con varios *pixels*, aplicando entonces sus valores a todos los afectados (19).

En una escena de calidad, la iluminación y las texturas suelen tener mayor importancia que la propia geometría que sustenta los modelos. Existen varias TGCBI: *light mapping, environment mapping, displacement mapping, mip mapping, normal mapping, relief mapping, parallax mapping* y *bump mapping*, donde no todas tienen como resultado el efecto visual de relieves en las superficies de los objetos, aunque si aumentan considerablemente el nivel de realismo y detalle. Para lograr el efecto de relieve pueden utilizarse: *displacement mapping, normal mapping, relief mapping, parallax mapping y bump mapping*, las que serán explicadas en el presente capítulo. Las características y ventajas del mapeado de texturas combinada con TGCBI son obvias, permite aumentar enormemente la complejidad visual de un modelo sin necesidad de aumentar su complejidad geométrica.

1.3.4. Bump Mapping

Esta técnica consiste en dar un aspecto rugoso a las superficies de los objetos. Esto puede ser claramente aplicado cuando se desea dar un efecto de relieve en el objeto. La técnica del *bump mapping* se utiliza para aumentar el nivel de detalle sin acrecentar el número de polígonos. Modifica las normales de las texturas dependiendo de los tonos de grises de cada *pixel* en el *bump map,* si la zona es clara u oscura sin modificar en ningún momento la topología ni la geometría del objeto, (ver figura 10).

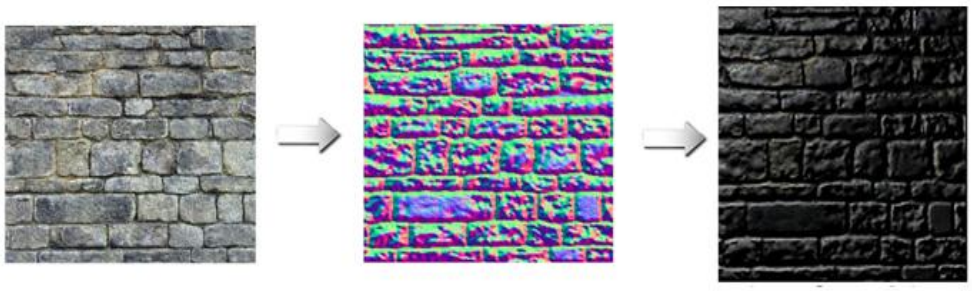
Figura 10: Resultado de la aplicación del *bump mapping*.

1.3.5. Normal Mapping

Normal mapping permite dar un relieve mucho más detallado a la superficie de un objeto. Es una evolución del *bump mapping* y se aplica en videojuegos a los que les dota un mayor realismo, así como en películas de animación para agilizar los cálculos y reducir el número de polígonos con los que en un principio contaba los objetos. Esta técnica no aplica la información por *pixel* en tonos de grises, el cual representaría la altura, sino en colores RGB dando más fidelidad al original que se quiere imitar y trabaja con los tres ejes X, Y, Z.

Normal mapping se encuentra normalmente en dos variedades: objeto-espacio y espacio-tangente. Se diferencian en sistemas de coordenadas en el que las normales se miden y se almacenan. Uno de los usos más interesantes de esta técnica es en gran medida mejorar la apariencia de un objeto que tiene un modelo de explotación que carece de polígonos, (ver figura 11).

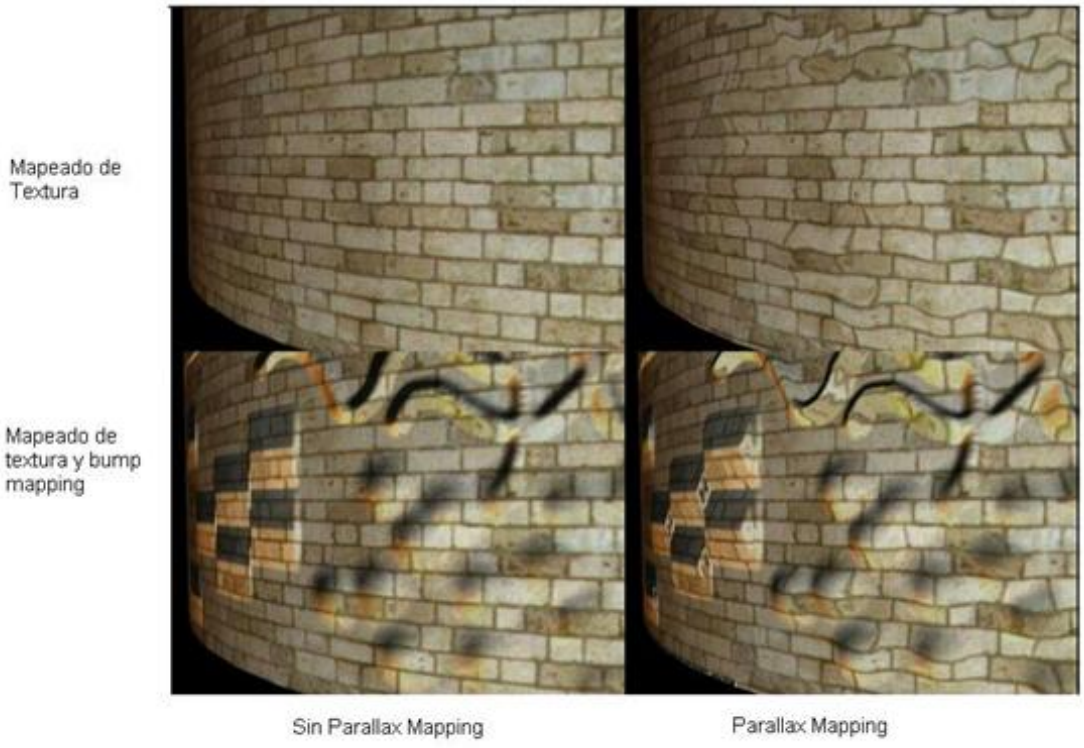
Figura 11: Resultado de la aplicación del *normal mapping*.



1.3.6. Parallax Mapping

El *parallax mapping* también llamado mapas de desplazamiento virtual, es una mejora de las técnicas de *bump mapping* o *normal mapping* que se aplica a las de mapeo de texturas 3D. Esta tiene en cuenta el ángulo con el que se mira a la superficie e introduce variaciones en la iluminación. En este caso la geometría de la superficie se mantiene intacta, pero el efecto está mejor logrado, (ver figura 12). Es una técnica correcta de aproximación de la aparición de superficies irregulares, modificando la textura a coordinar para cada *pixel*. Se puede simular bien mediante la construcción de un modelo geométrico de una superficie, pero para extraer todos los polígonos resulta computacionalmente costoso.

Figura 12: Comparación visual de varias técnicas de *rendering* (20).



A esta técnica introducida en el 2001 por varios autores japoneses entre los que se destaca Tomomichi Kaneko presentando resultados significativos. Otros la han popularizado realizado aportes significativos,

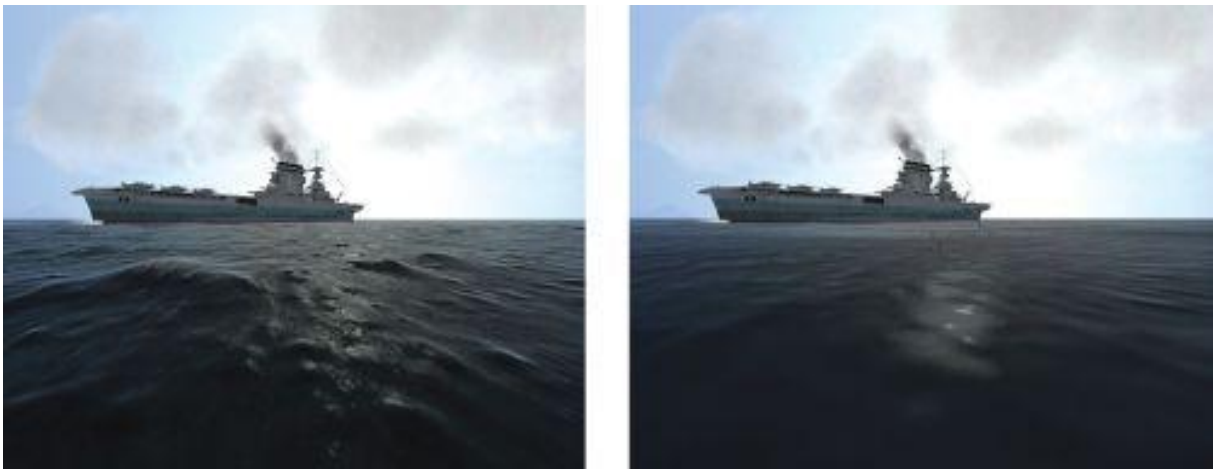
como por ejemplo: Terry Welsh en el 2003, proporciona un algoritmo para ejecutarlo en las tarjetas gráficas, obteniendo resultados eficientemente con el *parallax mapping*.

Estos análisis y algoritmos descritos por Terry Welsh y Kaneko no tienen en cuenta la oclusión. Con posterioridad se han realizado mejoras al algoritmo iterativo. Estos incorporan enfoques para permitir la oclusión y una silueta exacta del dibujado de las escenas, un ejemplo de esto es el trabajo realizado por Tatarchuk (18) basado en los aportes de Brawley (21), juntos publicaron *"Parallax Occlusion Mapping".*

1.3.7. Displacement Mapping

Esta técnica es una de las más avanzadas en la búsqueda de superficies con relieves cada vez más creíbles. Así, en lugar de utilizar las trampas del *parallax* o *normal mapping*, *displacement mapping* modifica realmente la superficie, por lo que el detalle es real, no una simple ilusión, (ver figura 13). Esto conlleva a que la sombra proyectada por el objeto o superficie tenga en cuenta la rugosidad de la misma (22).

Figura 13: El agua a la izquierda con el *displacement mapping*, a la derecha sin la misma (22).



1.3.8. Relief Mapping

En los gráficos computacionales *relief mapping* es una técnica alternativa de *parallax mapping*, mucho más precisa, puede apoyar el libre sombreado y *normal mapping*. Es posible describirla como una corta distancia de trazado de rayo hecha en un *pixel shader*. *relief mapping* apoya la representación de detalles en las superficies 3D, (ver figura 14). Esta técnica puede corregir puntos de vista de superficies y escenas 3D. Simula la apariencia de los detalles de la superficie geométrica. Con el sombreado de fragmentos individuales, de acuerdo a cierta profundidad y a la información de la superficie normal que está asignada en los modelos poligonales. Los *bump maps* y *normal maps* pueden ser almacenados en una única textura RGBA (32-bit por *Texel*) denominada textura de relieve. Todavía no es común encontrar esta técnica en los juegos de video, debido a su lentitud por la necesidad de una gran cantidad de procesamiento de *per-pixel* (23).

Figura 14: Tetera presentando diferentes texturas con *relief mapping* (23).

1.4. Motor Gráfico

En GxC el término motor de juego hace referencia al componente principal, al alma de un videojuego. Típicamente manipula el dibujado de las escenas y otras tecnologías necesarias, pero pudiera también manipular cuestiones adicionales como la inteligencia artificial y la detección de colisiones entre objetos. Los elementos más comunes que un motor de juego provee son las utilidades gráficas de dibujado ya sean 2D o 3D. Los motores que solamente proveen dibujado 3D en tiempo real son con frecuencia llamados motores gráficos (24).

Entre los motores gráficos comerciales y de código abierto más utilizados por la comunidad gráfica internacional se pueden citar:

Tabla 1: Prestigiosos motores gráficos en la comunidad gráfica internacional (24).

Comerciales	**Código Abierto**
Torque Game Engine	*Object Oriented Graphics Rendering Engine*
TV3D SDK 6.5	*Graphics Three Dimensional Engine*
3DGameStudio	*Irrlicht*
C4 Engine	*Crystal Space*
Unity	*Panda3D*
NeoAxis Engine	*JME*
DX Studio v2.2	*Reality Factory*
3Impact	*The Nebula Device 2*
Beyond Virtual	*RealmForge*
Deep Creator	*Blender Game Engine*

Debido a que el videojuego “Rápido y Curioso” está realizado con *Graphics Three Dimensional Engine* (G3D), a continuación se analizarán las principales características de este motor de juego.

1.4.1. Graphics Three Dimensional Engine

G3D está implementado en C++ y de código abierto bajo la licencia BSD y multiplataforma. Ha sido usado en juegos comerciales, artículos investigativos, simuladores militares y en las universidades. Soporta *rendering* en tiempo real y desconectado, además de realizar cálculos de propósito general basados en los GPU. G3D facilita un conjunto de rutinas y estructuras comunes que son necesarias en casi todos los programas de gráficos. Propicia que las bibliotecas de bajo nivel como OpenGL y los *sockets* sean más fáciles de usar y sin limitar su funcionalidad o rendimiento. Algunas características en G3D son útiles para cualquier programa, sin importar si este ejecuta cálculos 3D o corre sobre un procesador gráfico. Cuando se usa como una biblioteca utilitaria no es necesario el *framework* para la interfaz de usuario gráfica. Su última versión 7.01 constituye una completa solución gráfica para la construcción de juegos 3D y simuladores. Contiene poderosas características como una GUI parametrizable, soporta la carga de

numerosos formatos de modelos 3D, así como eficientes mecanismos para el trabajo con *shaders*. Es utilizada en muchas universidades prestigiosas, apareciendo además en numerosos juegos comerciales (24).

1.5. Bibliotecas Físicas

El corazón de la simulación de un sistema articulado de cuerpos rígidos es una biblioteca física de alto rendimiento o motor físico. Estos proveen una simulación basada en la física Newtoniana para sistemas de cuerpos rígidos. Usa variables como la masa, la velocidad, la fricción y la resistencia al viento. Permitiendo simular y predecir efectos bajo diferentes condiciones, con un alto grado de aproximación a lo que ocurre en la vida real. Por lo general se clasifican en dos tipos, de tiempo real y de alta precisión.

Los motores físicos de alta precisión requieren más poder de procesamiento para calcular con precisión los cálculos físicos. Comúnmente son utilizados por científicos en el estudio de estrategias de manipulación, control de sistemas robóticos y en películas de animados hechas en computadoras. En los videojuegos u otras formas de computación interactiva, el motor físico está llamado a simplificar sus cálculos y bajar su precisión. Actuando en un tiempo razonable, siendo apropiado durante la ejecución del juego. Estos son los reconocidos motores físicos de tiempo real. Por tales motivos existen una gran variedad, con alta calidad y disponibles tanto comerciales como código abierto.

Vale destacar que en febrero del 2006 se hizo el lanzamiento de la primera unidad de procesamiento físico *Physics Processing Unit* (PPU), perteneciente a la Compañía Ageia. Funcionan como las tarjetas gráficas, pero en este caso liberan al CPU del procesamiento físico. La tarjeta fue más efectiva en el aceleramiento de sistemas de partículas, sin embargo la mejora del rendimiento para cuerpos rígidos fue considerable.

1.5.1. Motores Físicos en tiempo real

El motor físico ODE desarrollado por Russel Smith y lanzada bajo la licencia BSD de código abierto fue encontrado y resultó ser muy apropiado para estos criterios. ODE tiene completa calidad industrial, presenta una detección de colisiones integrada (aunque se le puede acoplar un sistema de colisiones

propio) y una API en C/C++. En adición a numerosos juegos de computadoras comerciales, ODE ha sido utilizado en investigaciones biomecánicas y robóticas. Su ecuación para el movimiento de cuerpos rígidos es derivada del modelo de velocidad basado en los multiplicadores de Lagrange. Es rápido, robusto y estable. Además el usuario tiene la completa libertad de cambiar la estructura del sistema aún cuando la simulación esté corriendo. Enfatiza velocidad y estabilidad por encima de la precisión física (24). En la tabla 2 se muestran los motores físicos de tiempo real más conocidos.

Tabla 2: Motores Físicos en tiempo Real (24).

Código Abierto	*Código Cerrado*	*Comerciales*
Open Dynamics Engine	*Newton Game Dynamics*	*Havok (propiedad de Intel)*
Bullet	*Tokamak physics engine*	*nV Physics SDK*
OPAL	*PhysX (antiguamente NovodeX incorporando Meqon)*	*Endorphin*

1.6. Herramientas de Desarrollo

1.6.1. Microsoft Visual Studio 2008

Para el desarrollo de esta aplicación de consola Win32 escrita en C++ se seleccionó el entorno integrado de desarrollo Visual Studio 2008, dentro de este específicamente su compilador de C++ nativo. Visual Studio 2008 define un conjunto de herramientas de desarrollo profesionales para programadores individuales o para aquellos que están trabajando en pequeños equipos y que están construyendo aplicaciones. Dicha herramienta permite disfrutar de un entorno de desarrollo altamente productivo que incluye lenguajes de programación y editores de código mejorados. Dicha edición añade funcionalidad a las características básicas incluyendo herramientas para la depuración, así como un IDE completo y sin restricciones.

1.6.2. Lenguaje de Programación

Se utilizará como lenguaje de programación el C++ por ser un lenguaje libre, portable y robusto soportado bajo el paradigma de la Programación Orientada a Objeto e ideal para el desarrollo de gráficos.

1.6.3. Visual Paradigm

Visual Paradigm para UML es una herramienta profesional que soporta el ciclo de vida completo del desarrollo de software: análisis y diseño orientados a objetos, construcción, pruebas y despliegue. El software de modelado UML ayuda a una más rápida construcción de aplicaciones de calidad, mejores y a un menor coste. Permite dibujar todos los tipos de diagramas de clases, código inverso, generar código desde diagramas y generar documentación. La herramienta UML CASE también proporciona abundantes tutoriales, demostraciones interactivas y de proyectos UML.

Soporta un conjunto de lenguajes, tanto en la generación de código y como de ingeniería inversa sobre Java, C++, CORBA IDL, PHP, XML Schema, Ada y Python. Además, apoya la generación de código C#, VB. NET, código script de Flash, Delphi, Perl, Ruby, entre otros. Ingeniería Inversa también apoya clase Java, .NET, .dll y .exe, JDBC.

Después de realizado un estudio de esta herramienta, se determina que es la ideal para el trabajo con la Ingeniería de Software debido a que es muy potente para el propósito ingenieril además tiene como ventaja que posee versiones multiplataforma, buena integración con IDEs, incluye localización en castellano, muy personalizable y soporta muchos más lenguajes de programación. Todo esto favorece un buen desarrollo del producto por lo cual se obtendrá una mayor calidad en el software.

CAPÍTULO 2. DESCRIPCIÓN DE LA SOLUCIÓN TÉCNICA

La propuesta de solución de la presente investigación consiste en emplear las técnicas de mapeado de textura en combinación con las TGCBI como: *bump mapping y normal mapping*; para la representación de pequeñas deformaciones en objetos debido a colisiones. Esta propuesta lleva consigo un método bien definido para lograr dicho propósito. Para mostrar los resultados se utilizarán el motor gráfico G3D y el físico ODE.

El proceso comienza cuando el motor físico dispara la señal de la ocurrencia de una colisión. Previamente se debe garantizar que todos los objetos pensados para que en ellos ocurra este efecto, estén debidamente texturizados. Se le aplicarán a cada uno las técnicas de *bump mapping y normal mapping,* apoyándose del *parallax mapping* para obtener un mejor efecto visual. A cada *texel* en el *bump map* debe corresponderle una única ubicación en la superficie del objeto. Garantizando que al actualizar el *bump map* con los correspondientes cambios, la deformación se muestre en el lugar exacto donde ha ocurrido la colisión.

Cuando se produzca la señal de colisión, se determina si ha ocurrido la deformación en determinado objeto. Lo cual se logra mediante el modelo simplificado de deformación, trabajando con las propiedades físicas del material por el que está compuesto el objeto, conjuntamente con los estados de las variables, proporcionadas por el motor físico en el momento de la colisión.

Si ha ocurrido la deformación, se procede al cálculo de la misma. Mediante un mapa de direcciones, donde se guardan las posiciones de los *texels* afectados en la textura. Se realiza el cálculo de la diferencia de profundidad en cada *pixel* de los objetos en colisión. Posteriormente, con este mapa de direcciones, se efectuaría la actualización del *bump map* cargado en la memoria, correspondiente a la parte de la malla del objeto que ha colisionado. Finalmente se obtiene un nuevo *bump map* para el objeto, aplicándole el proceso de *normal mapping* al mismo se mostraría la deformación correspondiente.

2.1. Descripción de la solución propuesta

El pre-procesamiento de la deformación comienza cuando el motor físico dispara la señal correspondiente a la ocurrencia de una colisión. Dicho pre-procesamiento consiste en determinar si ha ocurrido una deformación permanente en un determinado objeto, teniendo en cuenta los parámetros físicos de cada uno de los materiales y los valores de las variables dinámicas manejadas por el motor físico. Esta determinación se realiza en su totalidad, en virtud de una disciplina de la Ingeniería Mecánica conocida como Resistencia de Materiales. Resulta meritorio aclarar que, en caso de ser negativa, no se ejecutaría ninguno de los pasos ni procesos que a continuación se describen. Así, en caso contrario, se prepararían las condiciones para el cálculo de dicha deformación, mediante un conjunto de pasos algorítmicos que garantizan su cálculo acertado. En un principio, se hace penetrar un objeto dentro del otro una distancia igual a la señalada por el motor físico, esta es la profundidad que un cuerpo penetra al otro. Luego se calcula el cubo de contención para los dos objetos participantes en la colisión, con el objetivo de determinar los parámetros necesarios para establecer una proyección ortográfica en la escena. A continuación se ubica la cámara adecuadamente en dirección contraria a la normal de colisión. Como la deformación a calcular puede ser tan grande como el menor de los objetos, se reajusta el campo de visión de la cámara al menor de estos, lo cual mejora notablemente el tiempo de ejecución del algoritmo.

Una vez ejecutados todos los pasos del pre-procesamiento, el GPU estará listo para el cálculo de la deformación, utilizando shaders durante el rendering en serie para el back buffer de los objetos que colisionan. Se utilizan dos buffers de color y dos de profundidad para almacenar información del texel afectado y calcular la diferencia de profundidad en cada pixel respectivamente. Cuando se tienen estos dos parámetros de la deformación, se actualiza el *bump map*, sustituyendo en cada *texel* afectado previamente calculado el nuevo valor de profundidad anteriormente determinado.

Vale destacar que dicho proceso se ejecuta para ambos objetos en colisión, de manera que se puedan calcular acertadamente las deformaciones respectivas. Una vez finalizado esto, a partir de los nuevos valores de profundidad del *bump map* de cada cuerpo, se re-calculan sus respectivos mapas de normales, con el objetivo de aplicar nuevamente las técnicas de *normal mapping*. Finalmente, se obtiene así una

deformación que no incluye modificación alguna en las geometrías de los objetos. La técnica de parallax mapping se utiliza para obtener mejor calidad visual.

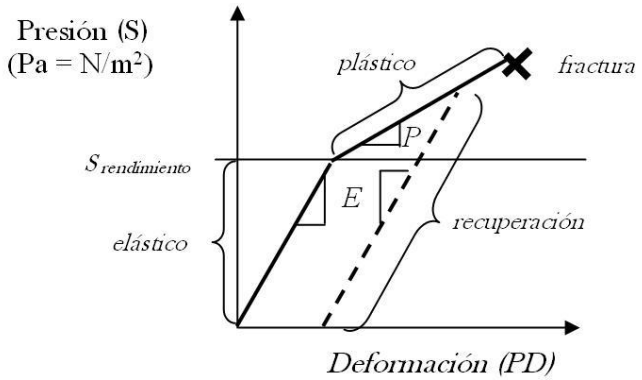
2.2. Simulación

Para lograr una representación creíble de pequeñas deformaciones en un objeto debido a una colisión, es necesario apoyarse en un modelo que permita trabajar con las variables necesarias para dicho objetivo. El modelo escogido está basado en la curva de Pvs.D presentada en la figura 2. Esta curva representa los cuatro estados por los que transita un material antes de fracturarse. El objetivo es simplemente, modelar físicamente las pequeñas deformaciones generadas por choques. Esta variedad de deformación solo ocurre en la región de plasticidad perfecta. Es necesario simplificar el modelo Pvs.D para lograr un resultado más eficiente.

2.2.1. Modelo Simplificado de Deformación.

En el artículo realizado por Crandall (25) se describe perfectamente el modelo simplificado de deformación basado en la curva de Pvs.D. Se tomarán en cuenta solamente las regiones elásticas y plásticas. Las mismas son las únicas que intervienen en el proceso de decisión de deformación permanente y cálculo de la pequeña deformación. Por simplicidad se modela la deformación plástica como una recta con pendiente ligeramente menor a la de la región elástica, como se muestra en la gráfica de la figura 15. Aunque es conocido que existen materiales con comportamiento no lineal en esta región.

Figura 15: Modelo Simplificado de la curva de Presión vs. Deformación (25).



Cuando la tensión disminuye, los materiales experimentan una fase de recuperación. Durante esta fase el desplazamiento decrece con el módulo de Young hasta que la tensión se hace cero. Es necesario notar que si el valor de la presión no excede nunca el punto límite de proporcionalidad, no habrá una deformación permanente pues el desplazamiento regresa a lo largo de la curva original. Si el valor presión llegara a sobrepasar el punto límite de proporcionalidad, el material experimentaría una deformación plástica permanente. En este proceso el material experimenta una fase de recuperación.

Debido la rápida ocurrencia de acciones en los videojuegos, las deformaciones elásticas no constituyen resultados visibles por los usuarios. Por estos motivos, y en aras de lograr mayor eficiencia y optimización, la presente investigación solamente simula las deformaciones permanentes, no las deformaciones elásticas temporales. Exclusivamente se trata este punto límite de proporcionalidad que se encuentra entre las dos regiones para determinar la ocurrencia de la deformación.

En la figura 15, el eje vertical representa la presión (S) a la que está sometido un cuerpo en un instante determinado, esta depende de magnitudes físicas determinadas durante la colisión con unidades de medida en pascal $(Pa = N / m^2)$. El eje horizontal representa el porcentaje de deformación (PD) que experimenta un cuerpo durante la acción de la presión. Los parámetros E y P son los coeficientes de elasticidad y plasticidad respectivamente, estos parámetros son únicos para cada material o compuesto sólido, para los metales el coeficiente elástico se encuentra en el orden de $10^{10}\ N/m^2$.

Siempre que ocurre una deformación permanente en el material, hay una parte de este que recupera su forma original en proporcionalidad con el coeficiente elástico y la presión ejercida en el material, mediante la ecuación de la pendiente para una recta, $m = \frac{y_1 - y_0}{x_1 - x_0}$, se deriva que:

$$recuperación = \frac{S}{E}$$

Para calcular el porciento de deformación total ocurrida en un cuerpo, se debe restar el porciento de deformación recuperada al porciento de deformación ocurrido en los procesos de elasticidad y plasticidad:

$$PD_{Total} = (\Delta \mathrm{PD}_P + \Delta \mathrm{PD}_E) - recuperación$$

Cuando ocurre una deformación permanente, para obtener el porciento de deformación en el proceso elástico y plástico, también se relacionan estos con el principio matemático de la pendiente:

$$E = \frac{S_{rendimiento}}{\Delta PD_E}\ ,\ P = \frac{S - S_{rendimiento}}{\Delta PD_P}$$

Por lo qué:

$$PD_{Total} = \frac{S - S_{rendimiento}}{P} + \frac{S_{rendimiento}}{E} - \frac{S}{E}$$

$$PD_{Total} = \frac{S - S_{rendimiento}}{P} + \frac{S_{rendimiento} - S}{E}$$

$$PD_{Total} = \frac{S - S_{rendimiento}}{P} + \frac{(-1) * S_{rendimiento} - S}{(-1) * E}$$

$$PD_{Total} = \frac{S - S_{rendimiento}}{P} - \frac{S - S_{rendimiento}}{E}$$

$$PD_{Total} = (S - S_{rendimiento}) * \left(\frac{1}{P} - \frac{1}{E}\right)$$

Los fenómenos del choque casi siempre están acompañados de la pérdida de energía, que puede calcularse restando la energía mecánica inmediatamente después del choque a la energía mecánica antes del mismo. Esta pérdida de energía tiene lugar mediante la generación de calor durante la deformación inelástica del material, por razón de la generación y disipación de ondas elásticas en el interior de los cuerpos y por la generación de energía acústica. Según esta teoría clásica, un valor de restitución $(\varepsilon = 1)$, significa que, la capacidad de dos cuerpos para restaurarse es igual a su tendencia a deformarse. Esta condición es la del choque elástico, sin pérdida de energía. En cambio, un valor $\varepsilon = 0$ caracteriza al choque inelástico o plástico en el cual la pérdida de energía es máxima. Todos los casos de choque se encuentran en estos dos casos extremos. También debe indicarse que se asocia un coeficiente de restitución a cada par de cuerpos en contacto.

Por lo que se puede decir que el coeficiente de restitución ε y el límite de proporcionalidad $S_{rendimiento}$ guardan una estrecha relación. La variable $S_{rendimiento}$ significa el valor límite para el cual el material deja de comportarse elásticamente y comienza el proceso plástico.

Cuando toda la energía es restituida $(\varepsilon = 1)$ el material se comporta todo el tiempo elásticamente sin deformarse permanentemente, mientras que este valor decrece el material puede alcanzar con más facilidad la deformación plástica, con el valor $(\varepsilon = 0)$ cualquier choque por muy pequeño que fuese deformaría el material permanentemente. Por lo que se puede decir que la energía que no se restituya en el cuerpo, es la causante de provocar una deformación en el mismo. En la figura 16 se muestra como es el comportamiento de este coeficiente en algunos materiales.

Figura 16: Relación del coeficiente de restitución y la velocidad relativa al choque (26).

Con lo antes explicado se puede llegar a la conclusión matemática:

$$\lim_{\varepsilon \to 1} S_{rendimiento} = \infty$$

$$\lim_{\varepsilon \to 0} S_{rendimiento} = 0$$

Por lo que:

$$S_{rendimiento} = Y_i = \frac{1}{1-\varepsilon} - 1 \; ; \; i \; objeto \; en \; cuestión$$

Para lograr la simplicidad en el cálculo de la deformación, se necesita equiparar las magnitudes físicas de la presión (N/m^2) con la del impulso $(N * s)$. Mediante estas consideraciones, se procede al cambio de *S* por el impulso (*j*) del instante en que los dos cuerpos colisionan entre sí (8).

Este impulso es calculado de diferentes formas según el motor físico utilizado, se propone para el cálculo de este impulso la ecuación descrita en (27).

$$j = -N_C \frac{[1+\min(\varepsilon_A,\varepsilon_B)]N_C(v_B - v_A)}{m_A^{-1}+m_B^{-1}} \quad (27)$$

Donde N_C es la normal de colisión, ε_A y ε_B son los coeficientes de restitución de cada cuerpo. Los parámetros v y m son las velocidades y masas de los cuerpos en colisión respectivamente

Con estas interpretaciones de $S_{rendimiento}$ se tiene que la deformación total es:

$$PD_{Total} = (j - Y_i) * \left(\frac{1}{P} - \frac{1}{E}\right)$$

Para:

$$k_i = \frac{1}{P} - \frac{1}{E} \; ; \; i \; objeto \; en \; cuestión$$

La ecuación final para el porciento de la deformación será:

$$PD_{Total} = k_i \max(0, \|j_{\varepsilon=1}\| - Y_i) \quad (8)$$

2.2.2. Simulación Física.

Para la simulación física y detección de colisiones se utilizó el motor físico ODE. En la fase de la detección de colisiones, se aplica el procedimiento para determinar si dicha colisión podrá deformar los objetos, dadas las propiedades del material se calcula un umbral para su deformación. Solo si el impulso de la colisión se encuentra por encima de este umbral ocurrirá una deformación en el objeto. Si el resultado es afirmativo se ejecuta el procedimiento destinado al cálculo de la deformación del *bump map.*

Este umbral impide que se realicen cálculos y cambios innecesarios en el *bump map* durante las colisiones débiles, posibilitando mayor eficiencia cuando existen demasiadas colisiones. Asegurando así, que se deforme el objeto sólo cuando el impacto es suficientemente fuerte para dejar una marca. No se deforman los objetos durante la fase de resolución de la colisión, con el fin de aumentar la velocidad de

simulación. Un objeto pesado puede deformar una superficie suave simplemente descansando en ella, pero es necesario omitir este caso para favorecer la velocidad de simulación.

Se pueden ajustar las normales de contacto y colisión por el mapa de normales cuando se calcula el impulso. Esto cambia las direcciones y velocidades resultantes de los objetos después de la colisión, simulando las reacciones físicas entre los objetos. Cuando se utiliza un motor físico que trae consigo la definición de estas transformaciones no es necesario preocuparse por esto. En este caso se utiliza ODE, el cual se torna conveniente y poderoso para la deformación. Este método puede ser trabajado en cualquier otro tipo de motor físico teniendo en cuenta siempre si es predictivo o no.

2.3. Texturizado y Rendering

En el proceso de texturizado o mapeado de textura, las coordenadas son asignadas a los vértices del objeto en cuestión. Según diversos procedimientos, se identifican cada uno de los *pixels* de dicho objeto durante el *rendering* con uno de los *texels* de la textura.

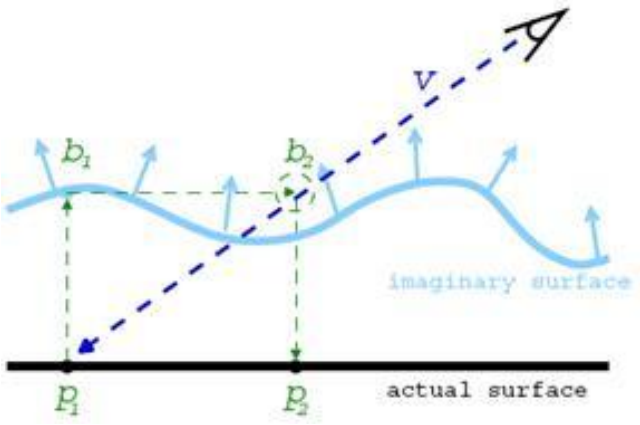
Para el *rendering* se utiliza la TGCBI *parallax bump mapping* con las limitaciones descritas en los métodos descritos en (28) (20), aproximándose ambos a la auto-oclusión y sombreado de las superficies rugosas. Este algoritmo funciona de la siguiente forma:

1. Para cada punto p_1 comienza en *rendering,* desde el vector de visión v al punto p_1. Al punto p_1 le corresponde el punto b_1 con el valor de altura almacenado en el *bump map.*
2. Desplazarse hacia atrás en dirección del vector v hasta que la altura sea igual a la del punto b_1. Suponiendo que el valor de la altura en el punto b_2 es localmente similar a la de b_1.
3. Se proyecta el punto b_2 en el punto p_2 de la superficie del objeto.
4. Se establecen los valores de la textura y la normal en el punto p_1 utilizando los valores del punto p_2.

Este método funciona bien si las superficies no tienen bordes filosos, teniendo bajas frecuencias. Altas frecuencias violan la suposición de que los valores de altura en el *bump map* son localmente similares. En este caso, dan la apariencia de dos superficies flotando una sobre la otra (8).

Se empaquetan el *normal map* y el *bump map* en una simple textura con los valores [*R,G,B,A*] = [N_x,N_y,N_z,*h*]. Donde *N* es la normal de la tangente del espacio y *h* es el desplazamiento en la superficie, (ver figura 17).

Figura 17: Método *parallax bump mapping* (8).



2.3.1. Parametrización.

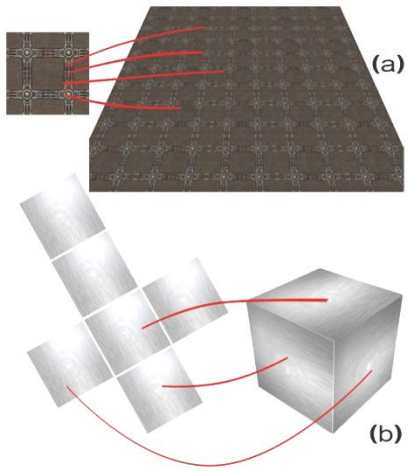
La parametrización es un conjunto de patrones, que aplicados a diversos factores, permiten obtener un resultado dentro de una cuota definida. En el presente caso, es preciso aplicarlo al mapeado de textura usado sobre un modelo o escena 3D.

Con frecuencia, los diseñadores gráficos reutilizan las texturas, aprovechando que estas son cargadas en memoria por las aplicaciones gráficas, así se obtiene un mejor aprovechamiento de los recursos del sistema. Ejemplo de ello es el texturizado del piso de una escena con baldosas, en el que se utiliza la textura reiteradamente hasta cubrir toda el área. Haciendo uso de técnicas de mapeado de texturas, se le asigna al objeto su respectivo *bump map,* donde el mismo se encuentra en varios puntos de la geometría (de uno a muchos).

Para el correcto funcionamiento del método que se desarrolla, es preciso tener un *bump map* y una textura para cada polígono. Si los diseñadores aplican la reutilización de texturas y por consiguiente el mismo *bump map* en varios puntos del modelo o entorno 3D, se tendrían varios puntos de la geometría asociada a un único *texel,* (ver figura 18(a)). En caso de colisión, al modificar el *bump map*, afectaría

visualmente todos los lugares donde ha sido mapeado, obteniendo un resultado indeseado. Se plantea como solución a este problema la realización del mapeado de textura por parte de los artistas gráficos, utilizando una única textura y *bump map* para cada punto de la geometría (de uno a uno), (ver figura 18(b)). Quedando de esta forma parametrizado el mapeado de textura del entorno o modelo 3D.

Figura 18: Parametrización del texturizado.



Otra de las formas de resolver este inconveniente, es en tiempo de ejecución, asignarle un *bump map* a cada malla sin importar que la textura esté repetida, aunque no es la mejor forma de hacerlo. Teniendo en cuenta lo anteriormente planteado, es indispensable la uniformidad en el texturizado cumpliendo los parámetros descritos.

2.3.2. Bump Mapping.

El *bump map* se utiliza en el cálculo de la pendiente de cada *pixel* y en la determinación de la ruta de altura. Inicialmente se hacen los cálculos al vector del punto alineado al eje de U y V del *bump map,* ajustándolo correctamente a la normal. Luego es obtenida la transformación de la normal de la luz de U y V, comparando el vector de dirección del punto en el mapa con la normal de la luz de X, Y, Z de los puntos calculados.

Si el punto está más expuesto a la luz, entonces es más brillante, frente a los puntos más alejados de la luz, realmente se obtiene rápidamente más oscuridad. Esta técnica utiliza como método matemático el método de Blinn, desarrollado en 1978 por James Blinn, el cual se encarga de simular arrugas en una superficie sin necesidad de modificar geométricamente el modelo. Para esto se utiliza una imagen en escala de grises en la que los colores claros significaban protuberancias y los oscuros huecos (22).

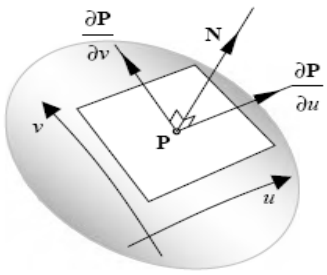
Para la creación del *bum map* se inicia por reconocer algún tipo de vector de la función binaria $P(\mathrm{u},\mathrm{v})$ que describe en una superficie en 3D, donde los parámetros (u,v) identifican un único punto $(\mathrm{x},\mathrm{y},\mathrm{z})$ en la superficie. Por ejemplo, $P(\mathrm{u},\mathrm{v})$ puede describir un plano, un cilindro, una esfera, un toroide, o un parche paramétrico.

A partir de ahora, se referencia la función como *P* y se recuerda que sus parámetros de entrada son (u,v).

Diferenciando parcialmente P con respecto a (u,v), se obtienen dos vectores $\partial \mathrm{P}/\partial \mathrm{u}$ y $\partial \mathrm{P}/\partial \mathrm{v}$, que se encuentran en el plano tangente que contiene P, (ver figura 19). Al tomar su cruce de productos se obtiene de la superficie normal *N* a *P*:

$$N = \frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v}$$

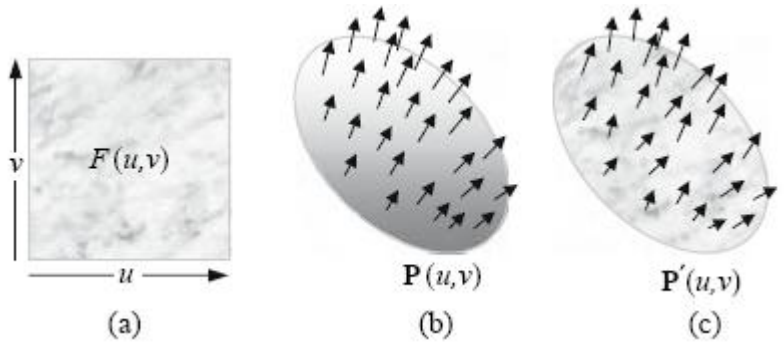
Figura 19: Ilustración de los vectores (29).



El siguiente paso es desplazar la superficie de manera efectiva a lo largo de N por un valor almacenado en una función escalar (*bump map*) $F(u,v)$. La figura 20 (a) muestra un mapa indexado por u y v. La

figura 20 (b) muestra una superficie definida por un vector de la función $P(u,v)$ con alguno de sus vectores normales. La figura 20 (c) muestra la superficie perturbada $P'(u,v)$ después el vector normal se ha perturbado por $F(u,v)$.

Figura 20: Vector Normal en cada punto (29).



Antes de que el desplazamiento se realice, N es normalizado para el proceso coherente:

$$\frac{N}{\|N\|}$$

Por tanto, el desplazamiento del punto P' se define como:

$$P' = P + F\frac{N}{\|N\|}$$

Estos nuevos puntos son interpretados para la superficie secundaria perturbada. Pero en la interpretación requiere el acceso a las superficies normales asociadas con P, que se define utilizando:

$$N' = \frac{\partial P'}{\partial u} \times \frac{\partial P'}{\partial v} \quad (1)$$

Las derivadas parciales en la ecuación 1 se amplían usando la regla de la cadena:

$$\frac{\partial P'}{\partial u} = \frac{\partial P}{\partial u} + \frac{\partial F}{\partial u}\left(\frac{N}{\|N\|}\right) + F\left(\frac{\partial \frac{N}{\|N\|}}{\partial u}\right)$$

Y

$$\frac{\partial P'}{\partial v} = \frac{\partial P}{\partial v} + \frac{\partial F}{\partial v}\left(\frac{N}{||N||}\right) + F\left(\frac{\partial \frac{N}{||N||}}{\partial v}\right)$$

Para simplificar estas derivadas parciales, se supone que la magnitud relativa de F es insignificante y puede ser ignorada. Así,

$$\frac{\partial P'}{\partial u} = \frac{\partial P}{\partial u} + \frac{\partial F}{\partial u}\left(\frac{N}{||N||}\right)$$

Y

$$\frac{\partial P'}{\partial v} = \frac{\partial P}{\partial v} + \frac{\partial F}{\partial v}\left(\frac{N}{||N||}\right)$$

Permite que la ecuación 1 se escriba como:

$$N' = \left(\frac{\partial P}{\partial u} + \frac{\partial F}{\partial u}\left(\frac{N}{||N||}\right)\right) \times \left(\frac{\partial P}{\partial v} + \frac{\partial F}{\partial v}\left(\frac{N}{||N||}\right)\right) \quad (2)$$

Básicamente la ecuación 2 tiene la forma de 4 vectores:

$$(a + b) \times (c + d)$$

Que cuando se amplía queda:

$$(a + b) \times (c + d) = a \times c + b \times c + a \times d + b \times d$$

Ampliando la ecuación (2) a lo largo de las líneas similares, se obtiene:

$$N' = \frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v} + \frac{\partial F}{\partial u}\left(\frac{N}{||N||} \times \frac{\partial P}{\partial v}\right) + \frac{\partial F}{\partial u}\left(\frac{\partial P}{\partial v} \times \frac{N}{||N||}\right) + \frac{\partial F}{\partial u}\frac{\partial F}{\partial v}\frac{N}{||N||} \times \frac{N}{||N||}$$

Pero

$$\frac{N}{\|N\|} \times \frac{N}{\|N\|} = 0$$

Y

$$\frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v} = N$$

Por tanto

$$N' = N + \frac{\partial F}{\partial u}\left(\frac{N}{\|N\|} \times \frac{\partial P}{\partial v}\right) + \frac{\partial F}{\partial u}\left(\frac{\partial P}{\partial v} \times \frac{N}{\|N\|}\right)$$

O

$$N' = N + \frac{\frac{\partial F}{\partial u}\left(N \times \frac{\partial P}{\partial v}\right) + \frac{\partial F}{\partial u}\left(\frac{\partial P}{\partial v} \times N\right)}{\|N\|}$$

O

$$N' = N + \frac{\frac{\partial F}{\partial u}\left(N \times \frac{\partial P}{\partial v}\right) - \frac{\partial F}{\partial u}\left(N \times \frac{\partial P}{\partial v}\right)}{\|N\|} \quad (3)$$

Blinn propone dos maneras de interpretar la ecuación 3: la primera interpretación, y probablemente la más obvia, es N' como la suma de dos vectores:

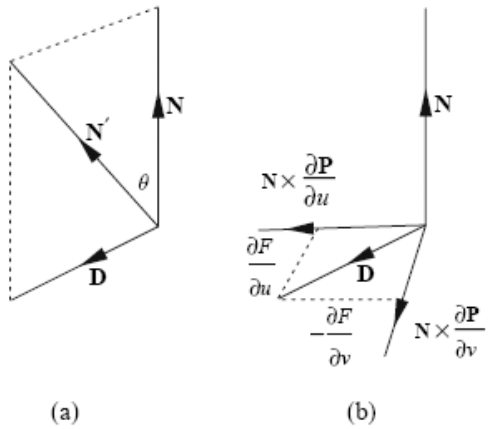
$$N' = N + D$$

Donde:

$$D = \frac{\frac{\partial F}{\partial u}\left(N \times \frac{\partial P}{\partial v}\right) - \frac{\partial F}{\partial u}\left(N \times \frac{\partial P}{\partial v}\right)}{\|N\|} \quad (4)$$

La figura 21 (a), muestra además este vector, y la figura 21 (b) muestra el vector entre productos asociados con D.

Figura 21: Representación de vectores (29).



La figura 21 (b) muestra que con la superficie N, el vector normal, $N \times {}^{\partial P}/_{\partial u}$ y $N \times {}^{\partial P}/_{\partial v}$ deberá situarse en el plano tangente a la superficie. Cuando estos vectores son escalados por las derivadas parciales del *Bump map* F, son sumados para obtener D, utilizado posteriormente para modificar N.

La altura del campo F fue utilizado inicialmente para escalar la unidad del vector normal ${}^{N}/_{\|N\|}$, es utilizado eventualmente para modificar N por D. Sin embargo, se puede guardar una gran cantidad de trabajo si el *bump map* toma la forma de compensar una función del vector $D(u, v$), entonces todo lo que se tiene que hacer es añadirlo a $N(\, u, v)$. Esta técnica se denomina mapa de compensación.

La segunda interpretación de Blinn sobre la ecuación 3 es imaginar que N es rotado alrededor de algún eje $N´$. El eje se encuentra en el plano tangente y se calcula utilizando $N \times N´$. Pero, como

$$N´ = N + D$$

Entonces:

$$N \times N´ = N \times (N + D) = N \times N + N \times D = N \times D \quad (5)$$

Si se sustituye la ecuación 4 en la ecuación 5 se obtiene:

$$N \times N' = N \times D = N \times \left(\frac{\frac{\partial F}{\partial u}\left(N \times \frac{\partial P}{\partial v}\right) - \frac{\partial F}{\partial v}\left(N \times \frac{\partial P}{\partial u}\right)}{\|N\|} \right)$$

$$N \times N' = \frac{\frac{\partial F}{\partial u}\left(N \times \left(N \times \frac{\partial P}{\partial v}\right)\right) - \frac{\partial F}{\partial v}\left(N \times \left(N \times \frac{\partial P}{\partial u}\right)\right)}{\|N\|} \quad (6)$$

La ecuación 6 ahora contiene un producto triple, lo que propone la siguiente identidad:

$$a \times (b \times c) = (a \cdot c)b - (a \cdot b)c$$

Ampliando la ecuación 6 se produce que:

$$N \times N' = \frac{\frac{\partial F}{\partial u}\left(\left(N \cdot \frac{\partial P}{\partial v}\right)N - (N \cdot N)\frac{\partial P}{\partial v}\right) - \frac{\partial F}{\partial v}\left(\left(N \cdot \frac{\partial P}{\partial u}\right)N - (N \cdot N)\frac{\partial P}{\partial u}\right)}{\|N\|}$$

Pero:

$$N \cdot N = \|N\|^2$$

Y

$$N \cdot \frac{\partial P}{\partial v} = 0 \; y \; N \cdot \frac{\partial P}{\partial u} = 0$$

Por tanto:

$$N \times N' = \frac{\frac{\partial F}{\partial u}\left(-\|N\|^2 \frac{\partial P}{\partial v}\right) - \frac{\partial F}{\partial v}\left(-\|N\|^2 \frac{\partial P}{\partial u}\right)}{\|N\|} = \|N\|\left(\frac{\partial F}{\partial v}\frac{\partial P}{\partial u} - \frac{\partial F}{\partial u}\frac{\partial P}{\partial v}\right)$$

Lo cual se puede expresar como:

$$N \times N' = \|N\|A \; (7)$$

Donde:

$$A = \frac{\partial F}{\partial v}\frac{\partial P}{\partial u} - \frac{\partial F}{\partial u}\frac{\partial P}{\partial v} = 0$$

Debido al cruce de productos en la ecuación 7, A debe ser perpendicular al plano que contenga N y $N´$. Por lo tanto, se puede imaginar que N ha sido rotado desde A hasta N.

Ahora

$$N \times N´ = N \times D$$

Y

$$N \times N´ = \|N\|A$$

Entonces

$$N \times D = \|N\|A$$

Además

$$\|N \times D\| = \|N\|\|A\| \ (8)$$

Pero como N es perpendicular ha D, se tiene:

$$\|N \times D\| = \|N\|\|D\| \sin 90° = \|N\|\|D\| \ (9)$$

La ecuación 8 y la 9 implican que

$$\|D\| = \|A\|$$

N se rota con un ángulo θ, en la figura 21(a), donde:

$$\tan\theta = \frac{\|D\|}{\|N\|} = \frac{\|N\|}{\|A\|}$$

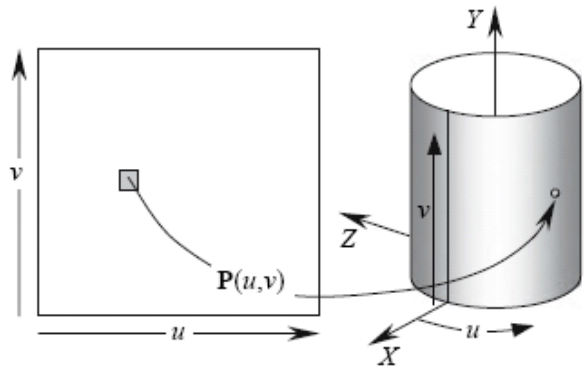
O

$$\theta = tan^{-1}\left(\frac{\left\|\frac{\partial F}{\partial v}\frac{\partial P}{\partial u} - \frac{\partial F}{\partial u}\frac{\partial P}{\partial v}\right\|}{\left\|\frac{\partial F}{\partial v} \times \frac{\partial P}{\partial u}\right\|}\right)$$

Permite desarrollar una estrategia de perturbación basada en las rotaciones, más compensada.

Ahora se ilustra el análisis anterior con un ejemplo. Para mantener las matemáticas simples, se usará un cilindro con vector función P, como se muestra en la figura 22.

Figura 22: Vector función (29).



Por ejemplo, si el radio y la altura del cilindro es igual a 1 y $(u, v) \in [0,2\pi]$, Entonces:

El componente x es dado por el $\cos u$,

El componente y es dado por v,

El componente z es dado por $-\sin u$,

Y

$$P(u, v) = \cos ui + vj - \sin uk \quad (10)$$

Para la ecuación 10 esto hace que:

$$\frac{\partial P}{\partial u} = -\sin ui - \cos uk$$

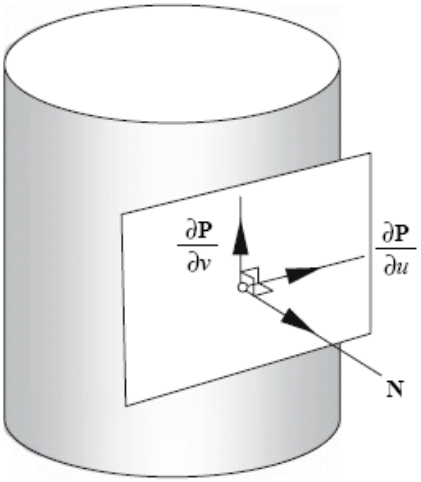
$$\frac{\partial P}{\partial v} = j$$

$$N = \frac{\partial P}{\partial u} \times \frac{\partial P}{\partial v} = \begin{vmatrix} i & j & k \\ -\sin u & 0 & -\cos u \\ 0 & 1 & 0 \end{vmatrix}$$

$$N = \cos u i - \sin u k$$

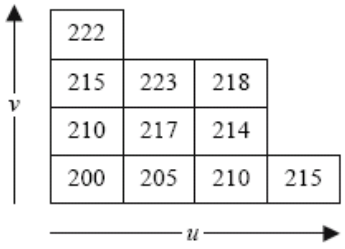
La figura 23 ilustra la orientación de estos tres vectores $\frac{\partial P}{\partial u}, \frac{\partial P}{\partial v}, y\ N$

Figura 23: Los tres vectores de cada punto (29).



Como la altura del campo $F(u, v) \in [0,255]$ contiene los valores que se muestran en la figura 24.

Figura 24: Campo (29).



Se muestra lo que ocurre en el punto $u = v = 0$

Para empezar:

$$\frac{\partial P}{\partial u} = -k$$

$$\frac{\partial P}{\partial v} = j$$

$$N = i$$

Para la figura 24 se tiene que:

$$\frac{\partial F}{\partial u} = 205 - 200 = 5$$

$$\frac{\partial F}{\partial v} = 210 - 200 = 10$$

En el cálculo de $^{\partial F}/_{\partial u}$ y $^{\partial F}/_{\partial v}$, se ignoran incrementos en ∂u y ∂v, debido a que esto puede ser compensado por la introducción de un factor de escala, que también controla el impacto del *bump map* en N. Este factor de escala es λ con un valor inicial de 1. Modificando la ecuación 3 con λ se produce:

$$N' = N + \lambda\left(\frac{\frac{\partial F}{\partial u}\left(N \times \frac{\partial P}{\partial v}\right) - \frac{\partial F}{\partial v}\left(N \times \frac{\partial P}{\partial u}\right)}{\|N\|}\right) \quad (11)$$

Ahora si se calcula $N \times {}^{\partial P}/_{\partial v}$ y $N \times {}^{\partial P}/_{\partial u}$:

$$N \times \frac{\partial P}{\partial v} = \begin{vmatrix} i & j & k \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{vmatrix} = k$$

$$N \times \frac{\partial P}{\partial u} = \begin{vmatrix} i & j & k \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{vmatrix} = j$$

Sustituyendo todos los términos en la ecuación 11, se obtiene:

$$N' = i + \lambda(5k - 10j)$$

Y si $\lambda = 1$

$$N´ = i - 10j + 5k$$

Si esto se considera inaceptable visualmente, λ puede ser ajustada a un determinado valor. De hecho, se puede hacer $\lambda = 00.1$ y a continuación, calcular nuevamente $N´$:

$$N´ = i - 0.1j + 0.05k$$

Por tanto, inicialmente $N = i$, y después de haber sido perturbado, será $N´ = i - 0\ 1j + 0.05k$ .Luego se calcula el ángulo θ entre estos vectores utilizando el producto por punto:

$$\theta = cos^{-1}\frac{(N \cdot N´)}{\|N\|\|N´\|}$$

$$\theta = cos^{-1}\left(\frac{(i) \cdot (i - 0.1j + 0.05k)}{1.0125}\right) = cos^{-1}\left(\frac{1}{1.0125}\right) = 9.01°$$

Como se mencionó anteriormente, es posible convertir la altura del campo original para compensar el mapa que contiene los vectores de la perturbación N, ahorrando considerables cálculos repetitivos.

2.3.3. Normal Mapping.

Partiendo de la información de altura o profundidad para cada *texel* brindada por *bump map*. El *normal map* se obtiene mediante un proceso automático en el que para cada *texel* del *bump map*, se prueba la altura de su inmediato derecho y su inmediato superior. El vector normal es la versión normalizada del producto cruzado de dos vectores de diferencia. El primer vector es *(1,0,Hr,Hg)*, en el que Hg es la altura del *texel* dado y *Hr* la del *texel* inmediato derecho al *texel* en cuestión. El segundo vector de diferencia es *(0,1,Ha,Hg)*, donde Ha es la altura del *texel* adyacente superior. El producto cruzado de estos dos vectores es un tercer vector apuntando fuera de la superficie del campo de altura. La normal resultante es:

$$\text{normal} = \frac{(\text{Hg} - \text{Ha}, \text{Hg} - \text{Hr}, 1)}{\sqrt{(\text{Hg} - \text{Ha})^2 + (\text{Hg} - \text{Hr})^2 + 1)}}$$

Esta normal es almacenada en una textura *RGB*, donde rojo, verde, y azul concuerden con *x, y, z*. Debido a la naturaleza del proceso de conversión de *bump map* a *normal map*, el componente de la *z* es siempre positivo y se guarda en el componente azul (ver anexo 1).

2.3.4. Parallax Mapping.

Esta técnica es ejecutada por el desplazamiento de las coordenadas de textura en un punto del polígono brindando por una función del ángulo de visión en el espacio tangente (el ángulo relativo a la superficie normal) y el valor de la altura de ruta en ese momento. Por el criterio de la función del ángulo de visión, las coordenadas de texturas más pronunciadas son más desplazadas y así dan la ilusión de profundidad (22).

Para el usuario final significa que las texturas utilizadas para simular un entorno serán más reales, arrojando como resultado mayor realismo en los escenarios de simulación. Esta técnica es un proceso de un solo paso que no tiene en cuenta la oclusión. Actualmente se han realizado mejoras al algoritmo iterativo que incorporan enfoques para permitir la oclusión y una silueta exacta de *rendering*.

Para la solución que se propone no es estrictamente necesario utilizar este método. Si se decide su incorporación a la solución del problema, bastaría con seleccionar cuál sería el algoritmo a ejecutar por el *parallax bump mapping.* Se propone el algoritmo con las limitaciones descritas en las investigaciones (28) (20), aproximándose ambos a la auto-oclusión y sombreado de las superficies rugosas. También pudiera utilizarse el método con *occlusion mapping* con las restricciones especificadas en (21) (18), en caso de disponer mejores prestaciones de la tarjeta gráfica.

2.4. Deformación del Objeto debido a la Colisión

Cuando dos objetos colisionan, sus parámetros físicos son alterados, modelándolos como cuerpos rígidos. Cada objeto es deformado cuando el impulso de la colisión sobrepasa el umbral definido por el coeficiente de restitución. El hecho de realizar la deformación no afecta de ningún modo el cálculo de la simulación en el motor físico.

En caso de que ocurriese una deformación plástica, se debe tener en cuenta el tamaño, la forma y la elasticidad del objeto que lo causó. Por ejemplo, una esfera deja una marca diferente de la que puede

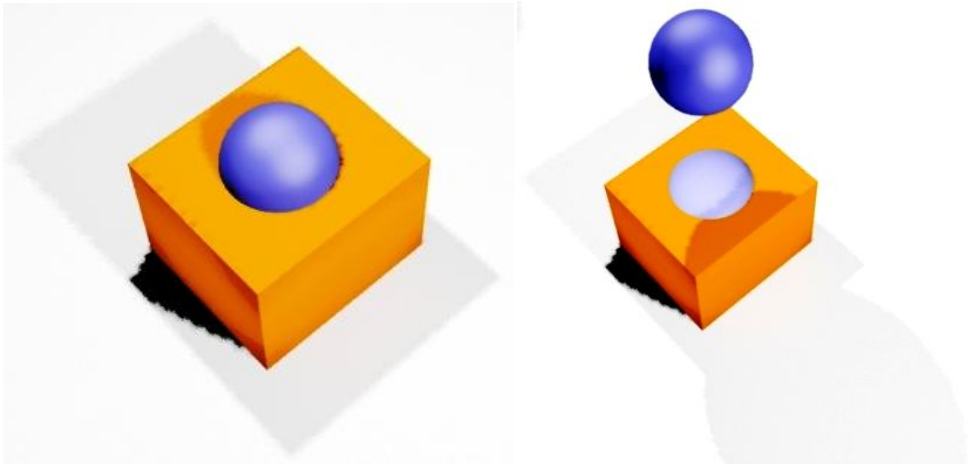
dejar un cubo al interactuar con un objeto plano. También una bola de acero dejará una marca en un cubo de arcilla, pero un cubo de arcilla no puede dejar una marca en la bola de acero.

Los autores Akeley y Jermoluk fueron los primeros en utilizar el *z-buffer* y el *stencil buffers,* para calcular la intersección de los objetos y operaciones de construcción de geometría sólida con propósitos de *rendering*. El método utiliza un mapa de direcciones para tomar las coordenadas de las texturas, una técnica adecuada para la elección de la proyección y el campo de visión de la cámara, de forma que toda la intersección esté a la vista. Por otra parte Govindaraju (12) también utiliza el *z-buffer* y el *stencil buffer* para detectar la penetración. En el mismo, solo se trabaja la detección de colisiones, sin buscar el contorno de la penetración, ni mapear la textura en el espacio que ha penetrado.

Utilizando los aportes anteriormente mencionados, Morgan McGuire emplea la tarjeta gráfica para calcular la deformación que ha tenido un objeto al ocurrir una colisión. Esta deformación no es más que la aproximación del tamaño y la forma de la misma, causada por el objeto que penetra dentro de otro durante la colisión. Intuitivamente, esta coincidencia está dada debido a cómo debe retroceder la superficie de un objeto para dar cabida al objeto penetrante.

Después de haber calculado la deformación en el GPU, se actualiza el *bump map* de los objetos. Esta actualización se ejecuta en el CPU y puede realizarse de varias formas, todo depende de las posibilidades que brinde el motor gráfico seleccionado. Así quedan representadas las pequeñas deformaciones en los objetos, sin necesidad de alterar la malla del mismo para lograr este propósito, ver la figura 25.

Figura 25: Pequeña deformación.



2.4.1. Cálculo de la Deformación.

Antes de realizar el cálculo de la deformación, deben estar creadas las bases para el correcto funcionamiento del algoritmo. Esto consiste en determinar si ha ocurrido una deformación permanente en un determinado objeto, teniendo en cuenta los parámetros físicos de cada uno de los materiales y los valores de las variables dinámicas manejadas por el motor físico. En un principio, se hace penetrar un objeto dentro del otro una distancia igual a la señalada por el motor físico, esta es la profundidad que un cuerpo penetra al otro. Luego se calcula el cubo de contención para los dos objetos participantes en la colisión, con el objetivo de determinar los parámetros necesarios para establecer una proyección ortográfica en la escena. A continuación se ubica la cámara adecuadamente en dirección contraria a la normal de colisión. Como la deformación a calcular puede ser tan grande como el menor de los objetos, se reajusta el campo de visión de la cámara al menor de estos, lo cual mejora notablemente el tiempo de ejecución del algoritmo.

Una vez ejecutados todos los pasos antes mencionados, el GPU estará listo para el cálculo de la deformación utilizando *shaders* durante el *rendering* en serie para el *back buffer* de los objetos que colisionan. Se utilizan entonces dos buffers de color y dos de profundidad para almacenar información del *texel* afectado y calcular la diferencia de profundidad en cada *pixel*.

El cálculo de la deformación consiste en utilizar algunas variables controladas por el motor físico durante la colisión. Con técnicas de *rendering* se genera un *bump map* por cada objeto en colisión, propiciando la futura actualización de la deformación. Se supone que el motor físico proporcione el punto del espacio donde ha ocurrido la colisión P_c, la normal de la colisión N_c, y la profundidad de la penetración d_c. Estas variables y otras utilizadas a lo largo del epígrafe se encuentran comentadas en la tabla 3.

Tabla 3: Variables de la colisión y el *bump map* (8).

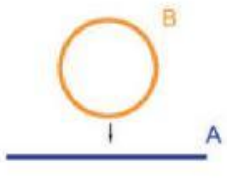
$C_A[i,j]$	Buffer de color para cada píxel [i,j] usado en el mapa de direcciones del objeto A.
$D_A[i,j]$	Buffer de profundidad (*z-buffer)* para cada píxel [i,j] del objeto A.
$\Delta D[i,j]$	Diferencia de la profundidad de la penetración en cada píxel [i,j].
$h_A[i,j]$	Elevación de la tangente del *bump map* del objeto A antes de la

	deformación.
$h'_A[i,j]$	Elevación del *bump map* del objeto A después de la deformación.
N_C	Vector Normal de la colisión.
P_C	Vector del punto de colisión en el espacio.
d_C	Profundidad de la penetración de la colisión.
t_A	Grosor del objeto A, o de sus muros si es hueco.
PD_A	Porciento de deformación en la superficie del objeto A.
s	Distancia entre la colisión y el plano cercano de la cámara ortogonal.
z_{near}	Coordenada z del plano cercano a la cámara ortogonal.
z_{far}	Coordenada z del plano lejano a la cámara ortogonal.
K	Profundidad máxima del *bump map.*
k'	Profundidad máxima del *bump map* cuando son alterados los valores del buffer de profundidad (*z-buffer*).

Se realiza el cálculo de la deformación en cada objeto por separado, debido a que se tiene en cuenta la perspectiva del objeto al que se está analizando. El método se repite para cada objeto en colisión. Es necesario aclarar que para cada variable del objeto *A*, debe existir una para el objeto *B*, por ejemplo: en cada iteración del método deben declararse dos atributos del *buffer* de colores, una para *A* y otro para *B*, $C_A[i,j]$ y $C_B[i,j]$. Así para cada atributo que tenga subíndice *A* debe existir uno con *B*.

Para entender mejor el procedimiento, se explica el método a través de un ejemplo. Suponiendo que se tienen dos objetos, un plano azul *A* y una esfera naranja *B* como se muestra en la figura 26, donde se analizará la deformación en el plano *A*.

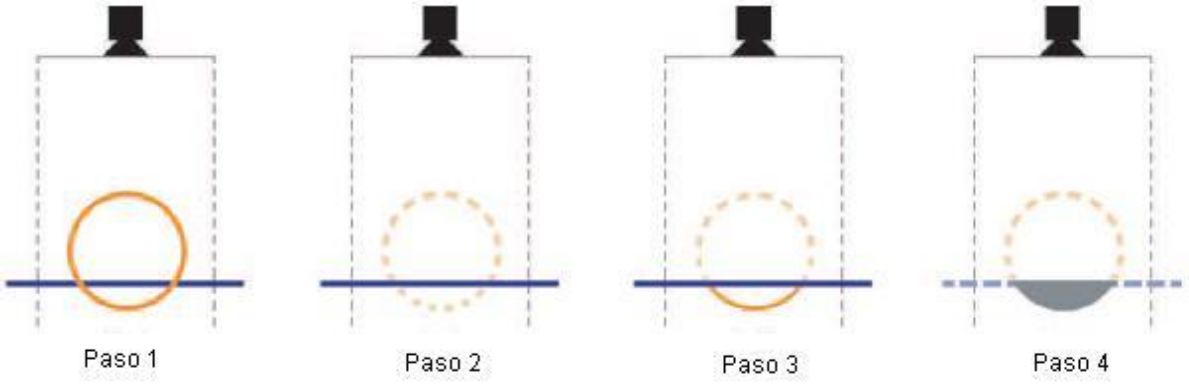
Figura 26: Plano azul A y esfera naranja B cayendo por la fuerza de gravedad (8).



Inmediatamente después de la colisión de los objetos, se obtiene el porciento de deformación PD_A ocurrido en el plano *A*, el cual si su resultado es positivo pasaría al cálculo de la deformación. El cálculo requiere de cuatro pasos lógicos, (ver figura 27):

1. Penetrar el objeto una distancia igual a la señalada por el motor físico, en el sentido contrario a la normal de colisión.
2. Realizar el *rendering* de la cara delantera del plano *A*.
3. Realizar el *rendering* de la cara trasera de la esfera B, solo los *pixels* que son más profundos que la cara del plano *A*.
4. La resta de las profundidades de cada uno de los píxeles de las caras, dará como resultado la diferencia de profundidad entre los dos objetos utilizándola para actualizar el *bump map.* Llevar el objeto a su posición original.

Figura 27: Pasos para el cálculo de la deformación (8).



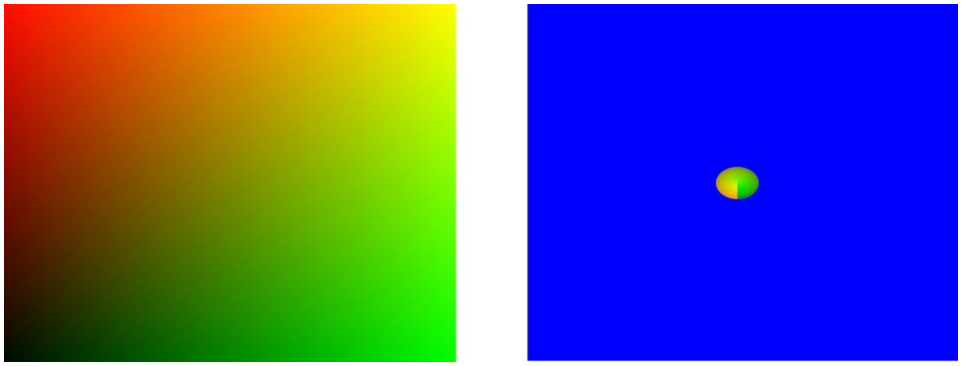
Para una mejor compresión, a continuación son explicados cada uno de estos casos:

Primeramente se traslada el objeto una distancia d_C en dirección contraria de la normal de colisión del punto inicial de contacto. La cámara en estos momentos debe estar posicionada en $P_C + N_C * s$ con el vector de visión en sentido contrario de la normal de colisión. Se selecciona *s* para que el volumen de visión de la cámara ortogonal contenga el cubo de contención de los objetos. Con esta configuración la cámara mira hacia el plano *A*, y la esfera *B* está más cerca de la cámara que *A*, excepto por donde los dos objetos interceptan. Como el solapamiento debe ser tan grande como el menor de los dos objetos, el campo de visión de la cámara se ajusta al objeto más pequeño.

El rango del valor para cada píxel del *bump map* se encuentra entre 0 y 1, el 0 es el valor de máxima abolladura, mientras que 1 es el máximo bulto que se puede formar en la superficie. Seguidamente se realiza una correspondencia con estos valores mínimo y máximo de profundidad de los objetos en la escena. El rango en que pueden estar los valores de profundidad en la escena es $\left[-\frac{k}{2}, \frac{k}{2}\right]$. El rango *k* puede ser elegido para cada objeto independientemente de la escala de la escena.

Luego se determina que porción del *bump map* necesita ser cambiada utilizando un mapa de direcciones. Cuando se dibujan los dos objetos, los colores de cada *pixel* llevan la coordenadas *(x, y)* de la textura en los canales rojo y verde (ver figura 28). Correspondiéndose el color del *pixel* con la dirección del *texel* del *bump map*. El canal azul siempre estará en cero, por lo que es una máscara para los objetos. Seguidamente se ejecutan los siguientes pasos, sin iluminación ni *parallax bump mapping*.

Figura 28: Mapa de direcciones.



1. Borrar el *frame buffer* con el color [0,0,1].
2. Se dibuja la cara delantera del plano *A* (paso 2 de la figura 28), donde a cada píxel de la imagen generada le corresponde un único *texel* en la textura.
3. Leer de nuevo la profundidad del *buffer* D_A y el *buffer* de colores C_A (ahora tiene como profundidad mayor la de los puntos de *A*).
4. Borrar el *buffer* de colores, (dejando los valores en el *buffer* de profundidad) y establecer mediante una prueba de profundidad. Seguir al próximo paso si el nuevo *pixel* está más lejos de la cámara que el anterior.

5. Dibujar el fondo de la cara del objeto *B* (paso 3 de la figura 28), con el color igual a la coordenada de la textura del *bump map.* Solo los *pixels* donde *B* es menor que *A*. Siempre que B penetre a *A* son pintados estos *pixel*s para la prueba de profundidad.
6. Leer de nuevo la profundidad del *buffer* D_B (contienen los puntos mínimos del área de la superposición) y el *buffer* de colores C_B.

Ver la figura del anexo 2, donde se puede observar un diagrama que representa el contenido de los *buffer* utilizados durante estos pasos. En la solución se utilizan cuatro *buffer* para calcular la profundidad de la deformación, dos para el mapa de direcciones de las coordenadas de la textura y otros dos para guardar la profundidad en cada *pixel*. La diferencia entre los *buffer* de profundidad, es la medida en cuánto se penetraron ambas geometrías. Debido a que la representación de los objetos está realizada con *bump mapping,* no solo se transforma el reflejo en cada punto de la geometría, sino también el *bump map*. Para cada *pixel*, el *vertex-shader* utiliza valores del mapa de direcciones, para obtener las coordenadas exactas en las que se debe actualizar el *bump map* correspondiente a cada objeto. Manipulando el valor de profundidad con la cámara ortográfica, se producen valores de profundidad lineal, no valores hiperbólicos como los obtenidos con la proyección de perspectiva.

Para alterar el *bump map* de los objetos con los *buffer* de profundidad, se necesita convertir los valores del *bump map* en valores de profundidad. Primero se multiplican los valores del *bump map* por el máximo valor de profundidad y se divide por la distancia entre el plano cercano y el plano lejano de la cámara (z_{far}– z_{near}), para obtener el valor de profundidad correspondiente. Sin embargo no se puede utilizar el valor *K* como máxima profundidad del *bump map*. Se selecciona el valor de *k'*, el mismo es más pequeño que *K*, para su uso posterior en la actualización del *bump map*. Como resultado, los valores almacenados en el *buffer* de profundidad contiene la información del *bump map* junto con la profundidad de la geometría. El *bump map* se escala, pues la deformación resultante no es demasiado profunda.

El *bump* y el *normal map* se encuentran en el espacio tangente, mientras que la diferencia de profundidad de la colisión ΔD esta en el espacio de colisión. El valor de la profundidad de todos los ejes está definido por el vector director de la cámara $-N_c$. Por tanto, cuando se actualiza el *bump map* basado en ΔD se introduce un error, a menos que N_c sea perfectamente paralelo al eje *z* del *pixel* que está siendo deformado. La deformación tendrá la correcta profundidad, pero esto podría ser en la dirección equivocada. En el extremo que una esfera puede cepillar el borde de un objeto y crear un profundo surco

perpendicular a la dirección de la colisión. Se puede proyectar ΔD en N, multiplicando $N * N_c$. Alternativamente se podría dividir ΔD por el producto escalar para crear una deformación correcta de profundidad a lo largo de N_c, pero podría ser muy profunda a lo largo de un eje ortogonal.

2.4.2. Actualizar el Bump Map.

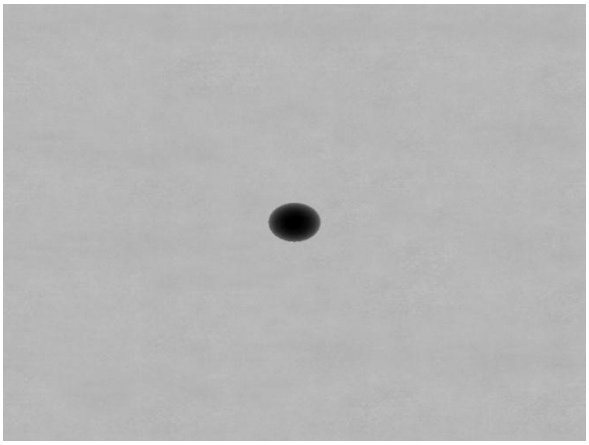
Se quiere deformar cualquier *pixel [i,j]* donde existe una superposición entre *A* y *B.* Se produce solo en los *pixels* donde los objetos han sido dibujados, $(C_A[i; j] = (r_A, g_A, b_A)$ y $C_B[i, j] = (r_B, g_B, b_B))$, $b_A = b_B = 0$ y en el cual exista una diferencia entre los valores de profundidad de los dos objetos ($\Delta D[i, j] \neq 0$). Para cada *pixel* en el que ocurra una superposición entre *A* y *B*, se disminuye el valor de $h_A[r_A,g_A]$ por $\Delta D[i, j]$. Sin embargo, se debe tener en cuenta que no se puede simplemente iterar en cada *pixel [i, j]*, porque el *texel* $h_A[r_A,g_A]$ podría afectar múltiples *pixels* de C_A durante el *rendering*. Esto causa múltiples *pixels* para un mismo *texel.* Si es ese el caso, entonces el cambio del valor de $h_A[r_A,g_A]$ por cada *pixel* del mapa causa un resultado inesperado. La deformación debe realizarse solo una vez en cualquier *texel*, y se tiene que elegir un *pixel* del mapa para el *texel,* ignorando cualquier otro, y modificar el valor del *texel* en consecuencia.

El primer paso para la actualización del *bump map* sería dibujar h_A por los valores del los *pixels* en cada posición del *z-buffer*. Luego ejecutar los siguientes pasos para cada pixel.

1. Obtener los valores de las componentes guardada en los *buffer* de cada objeto $(r_A,g_A,b_A) = C_A[i,j]$, $(r_B,g_B,b_B) = C_B[i,j]$; y se calcula la profundidad de penetración en el área de colisión $\Delta D[i,j] = D_B[i,j] - D_A[i,j]$.
2. Si la componente azul de cualquiera de los dos objetos es distinta de cero ($b_A \neq 0$ ó $b_B \neq 0$) ó la profundidad de penetración es igual a cero $(\Delta D[i,j] = 0)$, no hay solapamiento en este *pixel* y por lo tanto se pasaría al siguiente *pixel*, omitiendo los pasos posteriores.
3. Se obtiene el valor del *pixel* con las componentes roja y verde antes obtenida $h_A = h_A[r_A,g_A]$.
4. Calcular el nuevo valor de la elevación $h'_A = h_A - (\Delta D[i,j] * (z_{lejano} - z_{cercano}))/ k$. Este nuevo valor de la elevación se escala en el rango entre [0 ,1].
5. Pintar un punto en la posición $[r_A,g_A]$ del *buffer* con el color (h'_A, h'_A, h'_A) y profundidad h'_A.

Cuando este algoritmo se ha ejecutado para todos los pixels *[i,j],* el *back buffer* contiene el *bump map* de A con la actualización de los valores de la altura señalados en las posiciones correctas. Si se multiplican los *pixels* $C_A[i,j]$ donde el mapeado está en el mismo *texel* $h_A[r_A,g_A]$, se tendrán solo las deformaciones donde el resultado de los *pixels* correspondan con $\Delta D[i,j]$. Posteriormente, se obtiene el *bump map* de *A* leyendo del *back buffer*, (ver figura 29). Es necesario destacar, que este *bump map* obtenido del *back buffer* no tiene las mismas dimensiones que el *bump map* original del objeto, por lo que hay que cambiar en el original solo la parte que ha sido afectada.

Figura 29: Deformación del *bump map* del objeto *A*.

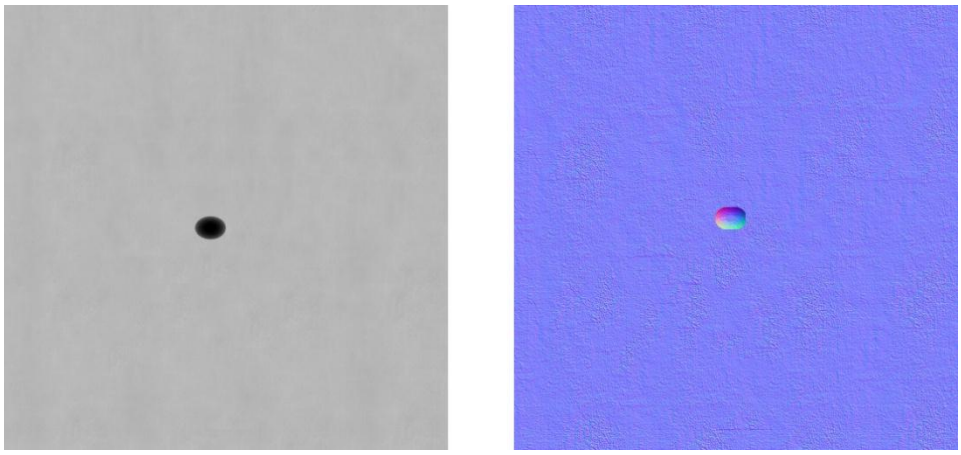


No se puede establecer la actualización del *bump map* de *A*, porque se necesita para la deformación del objeto B en caso de que este tenga también que deformarse. Se mueve A hacia la posición original y se repite el proceso para el objeto *B*. La única diferencia esta vez, es que se mueve el objeto *B* hacia *A* basado en el grosor del objeto *B* t_B, también el porciento de deformación en este caso es el de *B* y ahora es h_B en vez de h_A. Además hay que tener en cuenta para este segundo caso, que la normal de colisión debe ser orientada en sentido contrario a su dirección actual, para que la cámara y los cálculos posteriores se realicen naturalmente. Una vez que se tenga la actualización del *bump map* de *B*, esta se mueve hacia su posición original y luego con los nuevos *bump map* de ambos objetos se procede a la actualización de los mismos.

Para mostrar la deformación en 3D, no basta con actualizar el *bump map*. Para esto se necesita obtener el *normal map* del nuevo *bump map,* (ver figura 30). Este proceso es sumamente lento, dependiendo

siempre de las resoluciones de las imágenes a procesar. Cada *pixel* del *bump map* se procesa para obtener su correspondiente en el *normal map.* Por lo que en una imagen de alta resolución de 1024x1024, se procesarían 1 048 576 *pixels,* lo que constituye un alto costo computacional. Luego se tendría que convertir este *normal map* en una textura. Es válido aclarar que las dimensiones de las imágenes que se están procesando, intervienen significativamente en el rendimiento de actualización de la deformación. Se conoce que las texturas, para su correcto funcionamiento deben tener dimensiones que sean potencias de dos. De no cumplir con estos requisitos, las imágenes deben ser reconstruidas, por lo que el tiempo de respuesta de la actualización de la deformación se verá afectado. De lo contrario se tendría una respuesta satisfactoria, obviando la necesidad de reconstrucción de la imagen.

Figura 30: Conversión del *bump map* al *normal map*.



CAPÍTULO 3. PROPUESTA DE SOLUCIÓN.

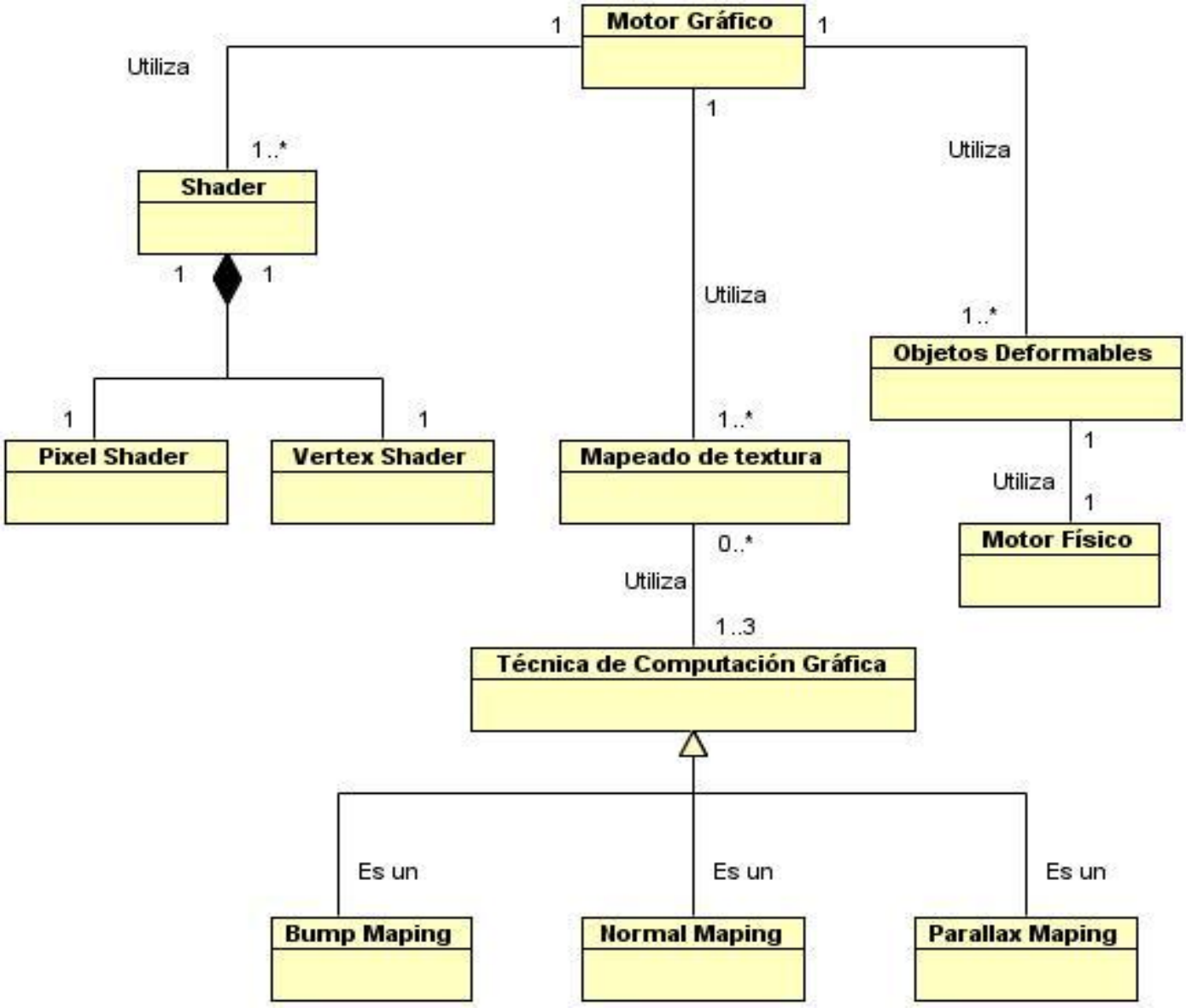
En este apartado se hace una descripción de la solución propuesta a través del modelo de dominio, representando los principales conceptos asociados al trabajo de diploma. Se hace referencia y se detallan los requisitos funcionales y no funcionales, casos de usos que se generan a partir de los requisitos funcionales y una explicación de cada uno de ellos a través de las descripciones textuales.

3.1. Modelo de Dominio

3.1.1. Diagrama de Clases del Dominio

Luego haber realizado un estudio, se llega a la conclusión, de que al no tener una estructura definida de los procesos del negocio (fronteras bien establecidas, donde se logren ver claramente quiénes son las personas que lo inician, quiénes son los beneficiados, pero además quiénes son las personas que desarrollan las actividades en cada uno de estos procesos), se plantea un modelo de dominio. Se realizará a través de un diagrama de clases de UML e identificando los conceptos representados en el diagrama en un glosario de términos; logrando que todo el interesado adquiera un entendimiento mínimo del contexto en que se desenvuelve el presente trabajo, (ver figura 31).

Figura 31: Modelo Conceptual de las clases del dominio.



3.1.2. Conceptos principales del Modelo de Dominio

Objetos deformables: Se refiere a objetos que, debido a sus propiedades físicas, su forma puede ser modificada por la acción de otro cuerpo o agente externo.

Shaders: Abstracción de hardware programable del *pipeline* de OpenGL

- *Vertex Shader*: Programa que actúa sobre las coordenadas, color, textura, entre otros, de un vértice.
- *Pixel Shader:* Programa que *a*ctúa sobre el color de cada *pixel* (*texel* para ser más preciso).

Mapeado de textura: Consiste en aplicar una serie de dibujos o plantillas a las caras de un objeto tridimensional.

Técnica de Computación Gráfica: Rendering tridimensional basada en la iluminación, diseñada para añadir nivel de detalle y sensación de tridimensionalidad a superficies sin aumentar la cantidad de polígonos aplicable a las técnicas de mapeo de texturas 3D.

- *Bump mapping:* Es una técnica de gráficos computacionales 3D que consiste en dar un aspecto rugoso a las superficies de los objetos calculado sobre la base de una textura de un solo canal (escala de grises).
- *Normal mapping:* Es una evolución del *bump mapping* que permite dar un relieve mucho más detallado a la superficie, la fuente de las normales es una imagen multicanal (RGB).
- *Parallax mapping:* Es una mejora del *bump mapping* y *normal mapping*, se aplica a las técnicas de mapeo de texturas 3D y tiene en cuenta el ángulo con el que se observa a la superficie e introduce variaciones en la iluminación.

Motor gráfico: Manipula el dibujado de las escenas y otras tecnologías necesarias, pero pudiera también manipular cuestiones adicionales como la Inteligencia Artificial y la Detección de Colisiones.

Motor físico: Permite simular y predecir efectos bajo diferentes condiciones con un alto grado de aproximación a lo que ocurre en la vida real.

3.2. Requerimientos

La Ingeniería de Requerimientos (IR) cumple un papel primordial en el proceso de producción de software, debido a su enfoque en un área fundamental: la definición de lo que se desea producir. Su principal tarea consiste en la generación de especificaciones correctas que describan con claridad, sin ambigüedades, en

forma consistente y compacta, las necesidades de los usuarios o clientes; de esta manera, se pretende minimizar los problemas relacionados por la mala gestión de los requerimientos en el desarrollo de sistemas. Los requerimientos pueden dividirse en funcionales y no funcionales.

3.2.1. Requerimientos Funcionales

Tabla 4: Requisitos Funcionales

Referencia	***Requisito Funcional***
R 1	Pre-Procesar deformación.
R 2	Calcular Deformación.
R 3	Actualizar *bump map* y *normal map*.

3.2.2. Requerimientos No Funcionales

Soporte:

- Para su correcta comprensión debe poseer una documentación del código y un demo ejemplificando el uso de los distintos algoritmos que posee.

Portabilidad:

- El Producto creado debe estar concebido para ser multiplataforma con ninguno o pocos cambios en su código fuente. Todas las herramientas de soporte utilizadas para la construcción del mismo deberán cumplir con dicha restricción.

Hardware:

- Tarjeta Gráfica que soporte *Vertex Shader* y *Pixel Shader*.
- Microprocesador Pentium 4 o superior.
- Memoria RAM 512MB o superior.

3.3. Descripción del Sistema. Modelos de Casos de Uso del Sistema

Posterior a la realización de una pequeña descripción del sistema, se da paso a la descripción del modelo de casos de uso del sistema usando las ventajas que brinda el lenguaje de modelado UML, se formulan las funcionalidades del sistema y representación mediante un diagrama, para ello es imprescindible definir los actores y los casos de uso que representarán las responsabilidades del mismo.

3.3.1. Determinación y justificación de los actores del sistema

Tabla 5: Justificación de Actores del Sistema

Actores	*Justificación*
Colisión	Representa al motor físico que detecta el momento de ocurrencia de la colisión.

3.3.2. Casos de Uso del Sistema

Un caso de uso constituye una técnica utilizada para describir el comportamiento del sistema, a través de un documento narrativo que define la secuencia de acciones que obtienen resultados de valor para un actor que utiliza un sistema para completar un proceso, sin importar los detalles de la implementación.

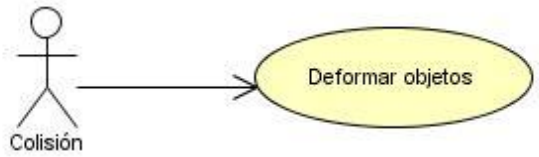
Tabla 6: Prioridad de los Casos de Uso.

Caso de Uso del Sistema	*Prioridad*
Deformar objetos	Crítico

3.3.3. Diagramas de Casos de Uso del Sistema

Los diagramas de casos de uso sirven para especificar la comunicación y el comportamiento de un sistema mediante su interacción con los usuarios y/u otros sistemas. O lo que es igual, un diagrama que muestra la relación entre los actores y los casos de uso en un sistema. Los diagramas de casos de uso se utilizan para ilustrar los requerimientos del sistema al mostrar cómo reacciona una respuesta a eventos que se producen en el mismo, (ver figura 32).

Figura 32: Diagrama de Casos de Uso del Sistema.



3.3.4. Expansión de los Casos de Uso del Sistema

Tabla 7: Descripción del CU "Deformar Objetos"

Descripción del Caso de Uso			
Nombre del Caso de Uso.	Deformar Objetos		
Actor(es).	Colisión.		
Propósito.	Determinar si ha ocurrido o no una deformación y preparar las bases.		
Descripción	El caso de uso se inicia cuando se dispara la señal de colisión, el motor físico recopila toda la información generada por esta y se determina si ocurre o no deformación.		
Referencias	RF- 1 RF- 2 RF- 3		
Pre - condiciones	Choque entre dos o más cuerpos.		
Pos - condiciones	La deformación ha sido representada.		
Flujo de Eventos			
	Actor		***Respuesta del Sistema***
1	Se dispara la señal de colisión.	2	Calcular si ha ocurrido deformación durante la colisión.
		3	Penetrar un objeto dentro del otro una distancia igual a la señalada por el motor físico.
		4	Calcular el cubo de contención para los dos

			objetos participantes en la colisión.
		5	Ubicar la cámara adecuadamente en dirección contraria a la normal de colisión.
		6	Reajustar el campo de visión de la cámara al menor de los dos objetos.
		7	Dibujar en un buffer todos los pixels de la cara frontal del objeto que se le está calculando su deformación guardando en las componentes rojas y verdes las posiciones de cada uno de estos pixel.
		8	Guardar en un buffer la profundidad de cada uno de estos pixels.
		9	Dibujar en un buffer los pixels de la cara trasera del segundo objeto, pero solo los que su profundidad sea mayor que la profundidad de los pixels del objeto anteriormente analizado.
		10	Guardar en un buffer la profundidad de cada uno de estos pixels del segundo objeto.
		11	Tomar las diferencias de profundidades en cada pixel.
		12	Obtener el valor de los texels que deben ser actualizados en el *bump map* original del cuerpo.
		13	Actualizar los texel del *bump map* del objeto analizado.
Flujo Alternativo			
	Actor		***Respuesta del Sistema***
		2.1	Si no ha ocurrido la deformación terminar el caso de uso.

3.4. Patrones.

3.4.1. Patrones de Diseño.

Se tuvo en cuenta algunos de los Patrones GRASP como son el patrón *Singleton* (perteneciente al patrón Creador), se utilizaron en las clases *CollisionManager* y *OdeWorld,* garantizando que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella. Además se utilizó el patrón de asignación de responsabilidad que según Booch y Rambaugh la responsabilidad es un "contrato u obligación de un tipo de clase", dentro de este se utilizó el patrón Experto, el cual plantea que se debe asignar la responsabilidad al experto en información que en este caso sería la clase que cuenta con la información necesaria para cumplir la responsabilidad. Un ejemplo es en la clase *PhysicMapper* donde su responsabilidad es la construcción del modelo físico de los objetos a partir de su geometría. Se utiliza además el patrón de Alta Cohesión que asigna responsabilidades de manera que la cohesión permanezca alta, mantiene la complejidad manejable.

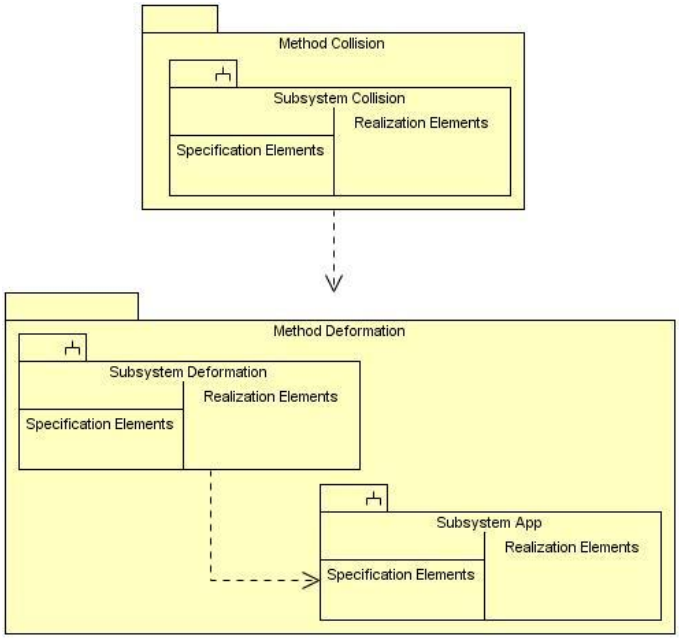
3.5. Diagramas de Clases del Diseño.

El modelo de diseño se adapta al entorno de implementación elegido que para este caso es .Net. Debe adaptarse para reutilizar *sistemas heredados* u otros marcos de trabajo desarrollados para el proyecto. Por tanto, funciona como esquema para la implementación.

Un diagrama de clases muestra un conjunto de clases, interfaces y colaboraciones, así como sus relaciones. Se utilizan para modelar la vista de diseño estática de un sistema principalmente, esto incluye modelar el vocabulario del sistema, modelar las colaboraciones o modelar esquemas. Son la base para los diagramas de componentes. Los mismos son fundamentales no sólo para visualizar, especificar y documentar modelos estructurales, sino también para construir sistemas ejecutables, aplicando ingeniería directa e inversa, (ver figura 33).

Para mayor referencia del Diseño de Clases Subsistemas consultar anexos 3, 4, 5.

Figura 33: Diagrama de Subsistema de Diseño.



3.6. Conclusiones parciales.

En este capítulo se dio paso a desarrollar la propuesta de la solución planteada, obteniéndose a partir de un análisis las funcionalidades que debe poseer el sistema, las cuales se representaron mediante un Diagrama de Casos de Uso describiéndose, paso a paso todas las acciones de los actores del sistema con los casos de uso con los que interactúan. Se identificaron los actores. Teniendo en cuenta todas estas características se puede comenzar a construir el sistema poniendo en práctica el cumplimiento de los requisitos tanto funcionales como no funcionales planteados en este capítulo. Además se mostró el diseño detallado del método a implementar, así como los patrones de arquitectura y diseño utilizados. Además se presenta como queda el sistema expresado en componentes de implementación.

CAPÍTULO 4. RESULTADOS

El método que en esta investigación se presenta para el cálculo de las pequeñas deformaciones en la meso-estructura de los objetos, no es físicamente correcto, pero sí es creíble debido al resultado gráfico que se obtiene. La utilización conjunta de diferentes conceptos y técnicas como resistencia de materiales, *shaders* y el mapeado de textura en combinación con las técnicas de gráficos computacionales basadas en iluminación, permiten la representación de pequeñas deformaciones en los cuerpos rígidos, con nivel de realismo adecuado, garantizando el principio causa-efecto. A través del mismo se logran pequeñas deformaciones sin necesidad de transformar la geometría. Logrando eficiencia al utilizar el GPU para el cálculo de la deformación. Esto permite que se puedan representar pequeñas deformaciones en tiempo real con bajo costo computacional. Mientras que una deformación muy grande que implique cambio en la macro-estructura de los objetos no debería ser modelada por este método, se tendría que acudir a la tradicional deformación de mallas.

El método tiene varios pasos intermedios que garantizan eficiencia. Mediante la teoría de la resistencia de materiales se determina si los choques han sido plásticos o elásticos. Debido a que los choques elásticos no representan una deformación permanente en la superficie de los objetos, estos no se procesan, evitando cálculos innecesarios. Por otra parte, son simplificadas estas teorías aplicándose el método propuesto por Crandal, para las deformaciones plásticas. Internamente se calcula el impulso de la colisión con la ecuación propuesta por Eran Guendelman, el cual le da más simplicidad aún.

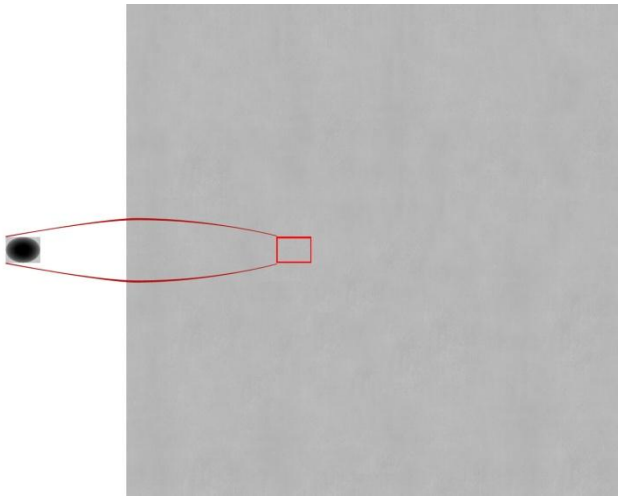
La mayor deformación que puede ocurrir es igual a la proyección del menor de los objetos. En el momento en que se calcula la deformación, es sumamente ventajoso tener el campo de visión de la cámara con proyección ortográfica. El campo de visión de la misma debe ajustarse al menor de los dos objetos que se encuentran colisionando. Esto permite que solo se procesen los *pixel* que se encuentran en este campo, impidiendo cálculos innecesarios.

El paso final del método de deformación consiste en actualizar el *bump map* del objeto. Morgan McGuire, en el trabajo que obtuvo segundo lugar en el evento SIGGRASF 2004, lo realiza de la siguiente forma. Primeramente dibuja el *bump map* original que previamente se ha cargado en memoria. Mediante el

vertex shader y utilizando los mapas de direcciones, se obtienen los puntos y la profundidad con la que se debe actualizar el *bump map* original. Esta actualización ocurre completamente en el CPU, debido a que las tarjetas gráficas actuales solo permiten tener cuatro texturas cargadas por cada *vertex shader.* Para realizar esta actualización, se necesitaría que cada *vertex shader* permita cargar cinco texturas. Las coordenadas de textura de cada objeto guardadas en un mapa de direcciones, las profundidades en cada una de estas coordenadas y el *bump map* original del objeto que se está analizando.

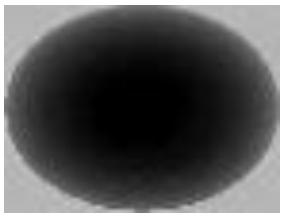
En aras de realizar este método con el menor número de pasos posible, se logró actualizar el *bump map* directamente en memoria, sin la necesidad de pasar por el proceso de *rendering*. Estos resultados se lograron gracias a las facilidades del motor gráfico utilizado, el cual permite cambiar los parámetros de los *pixels* de una imagen cargada en memoria. También facilita la copia de un fragmento de imagen sobre otra, obteniendo una nueva. Precisamente esta técnica, la cual es muy rápida, es la utilizada para la actualización del *bump map*, así se evitan el procesamiento innecesario en los *pixels* que no han sido afectados. Cambiando así solo la parte donde ha ocurrido la deformación, (ver figura 34). Prácticamente sería pegar el fragmento de la imagen del nuevo *bump map,* en el original. Mediante las coordenadas de textura donde comienza la deformación y las dimensiones del *bump map* original, se obtienen las coordenadas de imagen donde hay que pegar el nuevo *bump map*.

Figura 34: Actualización del *bump map*.



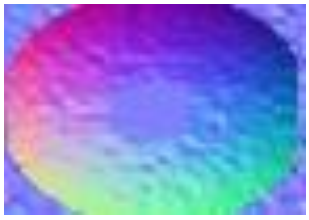
Como se mencionó anteriormente, al *bump map* hay que procesarlo para hallar su correspondiente *normal map.* Este procesamiento realiza varias operaciones para cada *pixel*, lo que implica que mientras mayor sea la imagen, mayor es su procesamiento. Precisamente para lograr eficiencia en este sentido, es necesario calcularle el *normal map* solamente al fragmento del *bump map* que ha sido actualizado, (ver figura 35).

Figura 35: Fragmento del *bump map* que ha sido actualizado.



Este procesamiento de normales depende de las dimensiones de la imagen. Para no alterar el resultado, el fragmento debe procesarse con las dimensiones del *bump map* original, (ver figura 36). Para lograr esta optimización, es necesario realizarle transformaciones al método de procesamiento de normales del anexo 1. Básicamente sería especificarle cuáles son las dimensiones del fragmento que se quiere procesar y cuáles son las coordenadas de comienzo, ver anexo 6.

Figura 36: Fragmento procesado por el algoritmo para convertir a *normal map*.



El resultado de este procesamiento no es más que la parte del *normal map* donde ha ocurrido la deformación. Para que queden completamente representados esos cambios, se pega este fragmento de *normal map* en el mapa original de normales del objeto, (ver figura 37). Por último se deben convertir estas nuevas imágenes del *bump map* y *normal map* en coordenadas de textura. De esta manera queda representada la deformación, (ver figura 38).

Figura 37: Actualización del *normal map*.

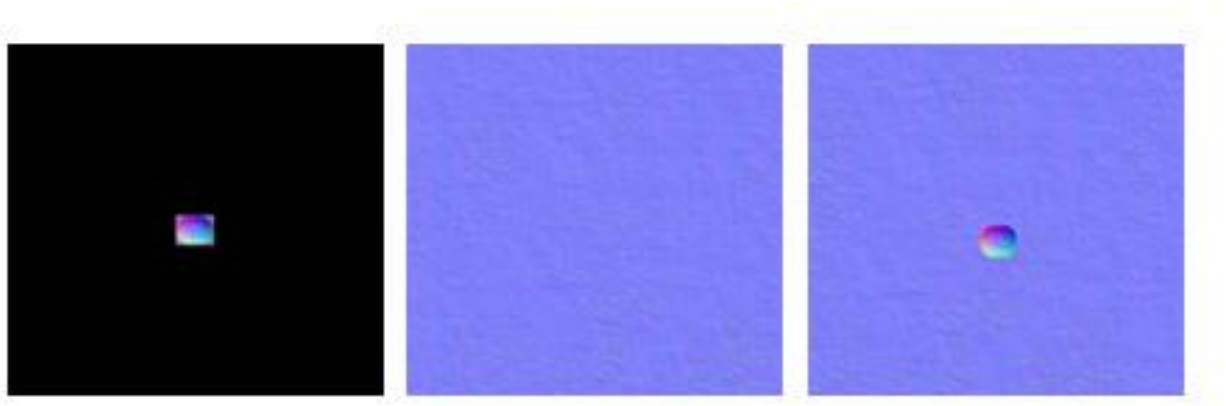
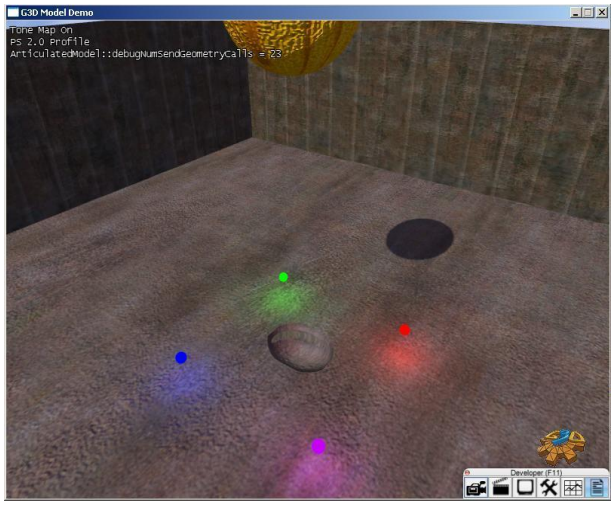


Figura 38: Representación de la deformación.



Otro de los resultados significativos de esta investigación, es la integración del método con el motor físico ODE, mientras que Morgan McGuire lo había realizado con OPCODE. Esto demuestra la gran independencia del método para su ejecución, logrando su funcionamiento con diferentes motores gráficos, físicos o de un lenguaje de *shader* específico. El motor físico con tan solo proporcionar la normal, el punto y la profundidad de colisión, ya se puede implementar con el este algoritmo. Sin más especificaciones el mismo solo se requiere de los pasos descritos en el seudocódigo que se presenta en el anexo 7.

CONCLUSIONES

Con la realización de la presente investigación se concluye que:

- El método cuenta con un alto nivel de independencia de los motores físicos, debido a la detección de las colisiones con ODE, además de poder trabajar con sus parámetros físicos para las decisiones de las deformaciones.
- Se calculó la deformación del *bump map* utilizando la unidad de procesamiento gráfico.
- Se logró la representación de las pequeñas deformaciones en los objetos debido a una colisión, utilizando las técnicas basadas en iluminación *bump mapping* y *normal mapping.*
- La estructura de clases propuestas para dar solución al problema científico permite cargar una escena con varios objetos, simulando su comportamiento físico, y en los cuales pueden realizarse pequeñas deformaciones en tiempo real.

Es necesario destacar que el proceso de desarrollo del sistema no supuso gastos de recursos, debido a que la infraestructura de producción estaba creada. Además, el soporte utilizado para la realización de la aplicación, es totalmente libre y multiplataforma por lo que no se incurrió en gastos referentes al pago de licencias y se siguieron las políticas de software libre.

RECOMENDACIONES

Durante el desarrollo del presente trabajo, han surgido ideas que podrían ser implementadas en un futuro, con el objetivo de lograr una aplicación más útil y efectiva, para lo cual se recomienda:

- Una vez que el *hardware* permita tener cinco o más texturas cargadas en cada *pixel shader,* debe realizarse el proceso de actualización en la tarjeta gráfica para lograr mayor eficiencia en respuesta a la deformación.
- Integrar el algoritmo propuesto en el videojuego “Rápido y Curioso”.
- Combinar esta técnica de pequeñas deformaciones con las de mallas prediseñadas, para un mayor realismo en las escenas.
- Perfeccionar la etapa de actualización del *bump map* mediante el estudio de los procesos en el *pipeline* gráfico.

REFERENCIAS BIBLIOGRÁFICAS

1. **Bretau Camejo, Osley and Ramírez Orozco, Raissel.** *Deformación de Objetos para Sistemas de Realidad Virtual.* Ciudad de La Habana : s.n., 2007.

2. *A PARTICLE-SYSTEM APPROACH TO REAL-TIME NON-LINEAR ANALYSIS.* **K., Martini.** Virginia : s.n., 2002.

3. *Flexible Simulation of Deformable Models Using Discontinuous Galerkin FEM.* **Kaufmann, Peter, et al.** Zurich : Eurographics, 2008.

4. *Collision Detection for Deformable Objects.* **Teschner, M., Kimmerle, S. and Heidelberger, B.** Zurich : s.n., 2005.

5. *Point Sample Rendering.* **Grossman, J.P. and Dally, W.** 2005.

6. **Marcheco Díaz, Yerandi and Ruiz Marine, Yausell.** *Módulo de Deformación Basado en el algoritmo Chainmail.* Ciudad de La Habana : s.n., 2008.

7. *A Bidirectional Light Field - Hologram Transform.* **Ziegler, Remo, et al.** 3, Zurich : Blackwell Publishing, 2007, Vol. 26.

8. *Real-Time Collision Deformations Using Graphics Hardware.* **McGuire, Morgan, Rice, Alexander and Wrotek, Pawel.** Los Angeles, CA : Journal of Graphics Tools, 2005. Vol. 10.

9. *Texture Mapping Volume Objects.* **Shen, P. and Willis, P.** Bath, UK : E. Trucco, M. Chantler, 2005.

10. *Real-Time Simulation of Deformation and Fracture of Stiff Materials.* **Müller, Matthias, et al.** UK : In Proceedings of the EurographicsWorkshop in Manchester, 2001.

11. *Fast volumetric deformation on general purpose hardware.* **Rezk-Salama, C., et al.** Alemania : In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, 2001.

12. *Interactive Collision Detection Between Complex Models in Large Environments using Graphics Hardware.* **Govindaraju, Naga K., et al.** North Carolina : M. Doggett, W. Heidrich, W. Mark, A. Schilling, 2003.

13. *Real-time Deformation of Detailed Geometry Based on Mappings to a Less Detailed Physical Simulation on the GPU.* **Mosegaard, J. and Sørensen, T. S.** Aarhus : R. Blach, E. Kjems, 2005.

14. *A ChainMail Algorithm for Direct Volumen Deformation in Virtual Endoscopic Simulation.* **Drägner, Ch.** Viena : Vienna University of Technology, 2005.

15. **Gere, James M.** *Mecánica de Materiales.* s.l. : Cengage Learning Editores, 2006. 9706864822.

16. **Céspedes Boch, Yirka and Cabrera Díaz, Yoander.** *Métodos Realistas de Iluminación Para Juegos 3D.* Ciudad de La Habana : s.n., 2007.

17. **Nodarse Valdés, Yaíma and Muguercia Torres, Lien.** *Módulo de Efectos Visuales Para Motores de Realidad, Virtual Luces y Sombras Dinámicas.* Ciudad de La Habana : s.n., 2007.

18. *Practical Dynamic Parallax Occlusion Mapping.* **Tatarchuk, Natalya.** Los Angeles, California : Siggraph, 2005.

19. **Sanz Sanfructuoso, Daniel and Carrasco de Pedro, Oscar.** Mapeado de Textura. *gsii.usal.es.* [Online] [Cited: Febrero 9, 2009.] http://gsii.usal.es/~corchado/igrafica/descargas/temas/Tema11.pdf..

20. *Parallax Mapping with Offset Limiting: A PerPixel Approximation of Uneven Surfaces.* **Welsh, Terry.** 2004.

21. *Parallax Occlusion Mapping.* **Brawley, Z. and Tatarchuk, N.** 2004.

22. **Galván Zald, Julio Antonio and Romero Naranjo, Maydelis.** *Aplicación de Shaders de Relieve a objetos 3D en entornos de Realidad Aumentada.* Ciudad De La Habana : s.n., 2008.

23. *Real-Time Relief Mapping on Arbitrary Polygonal Surfaces.* **Policarpo, F., Olivera, M. and Comba, J. L.** 2005.

24. **Pacheco Allende, Alberto Eliseo and Lombera Rodríguez, Hassán.** *Edición de sistemas articulados de cuerpos rígidos para las animaciones en los videojuegos.* Ciudad de La Habana : s.n., 2008.

25. *An Introduction to the Mechanics of Solids.* **CRANDALL, S. H, LARDNER, T. J and DAHL, N. C.** s.l. : SI Units., 1999.

26. **Vilardell, José, L.G., Kraige and J L, Meriam.** *Mecánica para ingenieros.* España : Reverte, 2000. 8429142592, 9788429142594.

27. *Nonconvex Rigid Bodies with Stacking.* **Guendelman, Eran, Bridson, Robert and Fedkiw, Ronald.** Los Angeles, California : ACM Trans. Graph., 2003.

28. *Detailed Shape Representation with Parallax Mapping.* **Kaneko, Tomomichi, et al.** Tokyo : ICAT, 2001.

29. **Vince, John.** *Vector Analysis for Computer Graphics.* Bournemouth : British Library, 2007. 978-1-84628-803-6.

BIBLIOGRAFÍA

A Bidirectional Light Field - Hologram Transform. **Ziegler, Remo, et al. 2007.** 3, Zurich : Blackwell Publishing, 2007, Vol. 26.

A ChainMail Algorithm for Direct Volumen Deformation in Virtual Endoscopic Simulation. **Drägner, Ch. 2005.** Viena : Vienna University of Technology, 2005.

A PARTICLE-SYSTEM APPROACH TO REAL-TIME NON-LINEAR ANALYSIS. **K., Martini. 2002.** Virginia : s.n., 2002.

A Practical and Robust Bump-mapping Technique forToday's GPUs. **Kilgard, Mark J. 2000.** California : s.n., 2000.

An Introduction to the Mechanics of Solids. **CRANDALL, S. H, LARDNER, T. J and DAHL, N. C. 1999.** s.l. : SI Units., 1999.

Artificial Fishes: Autonomous Locomotion, Perception, Behavior, and Learning in a Simulated Physical World. **Terzopoulos, Demetri, Tu, Xiaoyuan and Grzeszczuk, Radek. 1994.** 4, Toronto : Published in Artificial Life, 1994, Vol. 1. 327–351.

Bretau Camejo, Osley and Ramírez Orozco, Raissel. 2007. *Deformación de Objetos para Sistemas de Realidad Virtual.* Ciudad de La Habana : s.n., 2007.

Céspedes Boch, Yirka and Cabrera Díaz, Yoander. 2007. *Métodos Realistas de Iluminación Para Juegos 3D.* Ciudad de La Habana : s.n., 2007.

Collision Detection for Deformable Objects. **Teschner, M., Kimmerle, S. and Heidelberger, B. 2005.** Zurich : s.n., 2005.

Contact Handling for Deformable Point-Based Objects. **Keiser, Richard, et al. 2004.** Zurich : s.n., 2004.

Detailed Shape Representation with Parallax Mapping. **Kaneko, Tomomichi, et al. 2001.** Tokyo : ICAT, 2001.

Fast and Robust Angle Based Flattening. **Bogomyakov, Alexander, et al. 2005.** British Columbia : s.n., 2005.

Fast Simulation of Deformable Models in Contact Using Dynamic Deformation Textures. **Galoppo, Nico, et al. 2006.** Zurich : M.-P. Cani, J. O'Brien, 2006.

Fast volumetric deformation on general purpose hardware. **Rezk-Salama, C., et al. 2001.** Alemania : In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, 2001.

Flexible Simulation of Deformable Models Using Discontinuous Galerkin FEM. **Kaufmann, Peter, et al. 2008.** Zurich : Eurographics, 2008.

Galván Zald, Julio Antonio and Romero Naranjo, Maydelis. 2008. *Aplicación de Shaders de Relieve a objetos 3D en entornos de Realidad Aumentada.* Ciudad De La Habana : s.n., 2008.

Gere, James M. 2006. *Mecánica de Materiales.* s.l. : Cengage Learning Editores, 2006. 9706864822.

High-Performance Polygon Rendering. **Akeley, Kurt and Jermoluk, Tom. 1988.** 4, Mountain View, CA : Computer Graphics, 1988, Vol. 22. 94039-7311.

Interactive Collision Detection Between Complex Models in Large Environments using Graphics Hardware. **Govindaraju, Naga K., et al. 2003.** North Carolina : M. Doggett, W. Heidrich, W. Mark, A. Schilling, 2003.

Marcheco Díaz, Yerandi and Ruiz Marine, Yausell. 2008. *Módulo de Deformación Basado en el algoritmo Chainmail.* Ciudad de La Habana : s.n., 2008.

Nodarse Valdés, Yaíma and Muguercia Torres, Lien. 2007. *Módulo de Efectos Visuales Para Motores de Realidad, Virtual Luces y Sombras Dinámicas.* Ciudad de La Habana : s.n., 2007.

Nonconvex Rigid Bodies with Stacking. **Guendelman, Eran, Bridson, Robert and Fedkiw, Ronald. 2003.** Los Angeles, California : ACM Trans. Graph., 2003.

Pacheco Allende, Alberto Eliseo and Lombera Rodríguez, Hassán. 2008. *Edición de sistemas articulados de cuerpos rígidos para las animaciones en los videojuegos.* Ciudad de La Habana : s.n., 2008.

Parallax Mapping with Offset Limiting: A PerPixel Approximation of Uneven Surfaces. **Welsh, Terry. 2004.** 2004.

Parallax Occlusion Mapping. **Brawley, Z. and Tatarchuk, N. 2004.** 2004.

Per-Pixel Displacement Mapping with Distance Functions. **Donnelly, William. 2005.** Waterloo : Nvidia, 2005.

Point Sample Rendering. **Grossman, J.P. and Dally, W. 2005.** 2005.

Practical Dynamic Parallax Occlusion Mapping. **Tatarchuk, Natalya. 2005.** Los Angeles, California : Siggraph, 2005.

Pyramidal Displacement Mapping: A GPU based Artifacts-Free Ray Tracing through an Image Pyramid. **Hyunwoo, Ki, Cheol-Hi, Lee and Kyoungsu, Oh. 2007.** Seoul : Soongsil University, 2007.

Realistic, Hardware-accelerated Shading and Lighting. **Wolfgang, Heidrich and Hans-Peter, Seidel. 2000.** Saarbr¨ucken, Germany : Computer Graphics, 2000.

Real-Time Collision Deformations Using Graphics Hardware. **McGuire, Morgan, Rice, Alexander and Wrotek, Pawel. 2005.** Los Angeles, CA : Journal of Graphics Tools, 2005. Vol. 10.

Real-time Deformation of Detailed Geometry Based on Mappings to a Less Detailed Physical Simulation on the GPU. **Mosegaard, J. and Sørensen, T. S. 2005.** Aarhus : R. Blach, E. Kjems, 2005.

Real-Time Relief Mapping on Arbitrary Polygonal Surfaces. **Policarpo, F., Olivera, M. and Comba, J. L. 2005.** 2005.

Real-Time Simulation of Deformation and Fracture of Stiff Materials. **Müller, Matthias, et al. 2001.** UK : In Proceedings of the EurographicsWorkshop in Manchester, 2001.

Rodríguez González, Lester Oscar and Fernández Ruiz, Leonardo Rafael. 2008. *Técnica de Corte para Mallas Superficiales.* Ciudad de la Habana : s.n., 2008.

Sanz Sanfructuoso, Daniel and Carrasco de Pedro, Oscar. Mapeado de Textura. *gsii.usal.es.* [Online] [Cited: Febrero 9, 2009.] http://gsii.usal.es/~corchado/igrafica/descargas/temas/Tema11.pdf..

Texture Mapping Volume Objects. **Shen, P. and Willis, P. 2005.** Bath, UK : E. Trucco, M. Chantler, 2005.

Valiente Prieto, Dany Jesús and de la Paz Acosta, Robin. 2008. Desarrollo de técnicas de textura en entornos virtuales generaos con la herramienta de desarrollo “Scene Toolkit”. La Habana : s.n., 2008.

Vilardell, José, L.G., Kraige and J L, Meriam. 2000. *Mecánica para ingenieros.* España : Reverte, 2000. 8429142592, 9788429142594.

Vince, John. 2007. *Vector Analysis for Computer Graphics.* Bournemouth : British Library, 2007. 978-1-84628-803-6.

GLOSARIO DE TÉRMINOS

Buffer: es una ubicación de la memoria en una computadora o en un instrumento digital reservada para el almacenamiento temporal de información digital, mientras que está esperando ser procesada.

Frame buffer: memoria usada para retener uno o más *frames* para su posterior uso.

Hardware: componentes físicos de una computadora o de una red (a diferencia de los programas o elementos lógicos que los hacen funcionar).

***Pipeline*:** unidad de cálculo especializada. Calculan polígonos, sus coordenadas y posición. Existen pipelines de geometría (3D) y de imagen (2D).

Pixel: abreviatura de *"picture element"*. Es la unidad menor de almacenamiento de información de una imagen digital.

Pixel Shader: pequeños programas que se encargan del procesamiento de pixeles

Rendering: crear en forma automática una imagen de acuerdo al modelo tridimensional que existe en el ordenador.

Shader: conjunto de instrucciones gráficas destinadas para el acelerador gráfico, estas instrucciones dan el aspecto final de un objeto. Los shaders determinan materiales, efectos, color, luz, sombra.

Stencil buffers: buffer que se tiene para manejar el valor de los píxeles. Esta técnica generalmente es aplicada para generar Sombras y Reflexiones en aplicaciones en 3D.

Texel: (contracción del inglés *texture element*, o también *texture pixel*), es la unidad mínima de una textura aplicada a una superficie.

Vertex Shader: pequeños programas que se encargan del procesamiento de vértices.

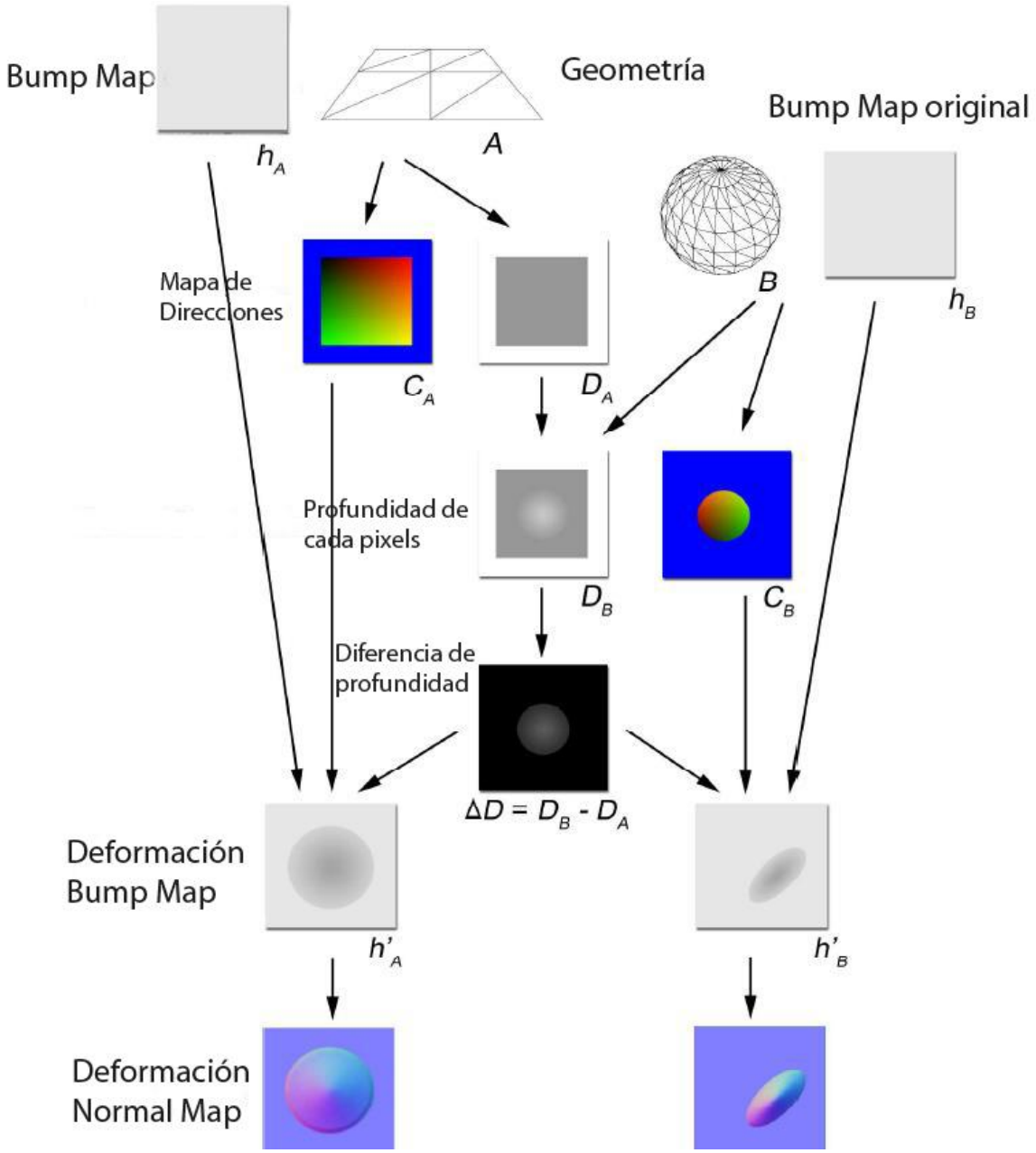
Z-buffer: también se conoce como *buffer* de profundidad, el cual permite reproducir correctamente la percepción de la profundidad.

ANEXOS

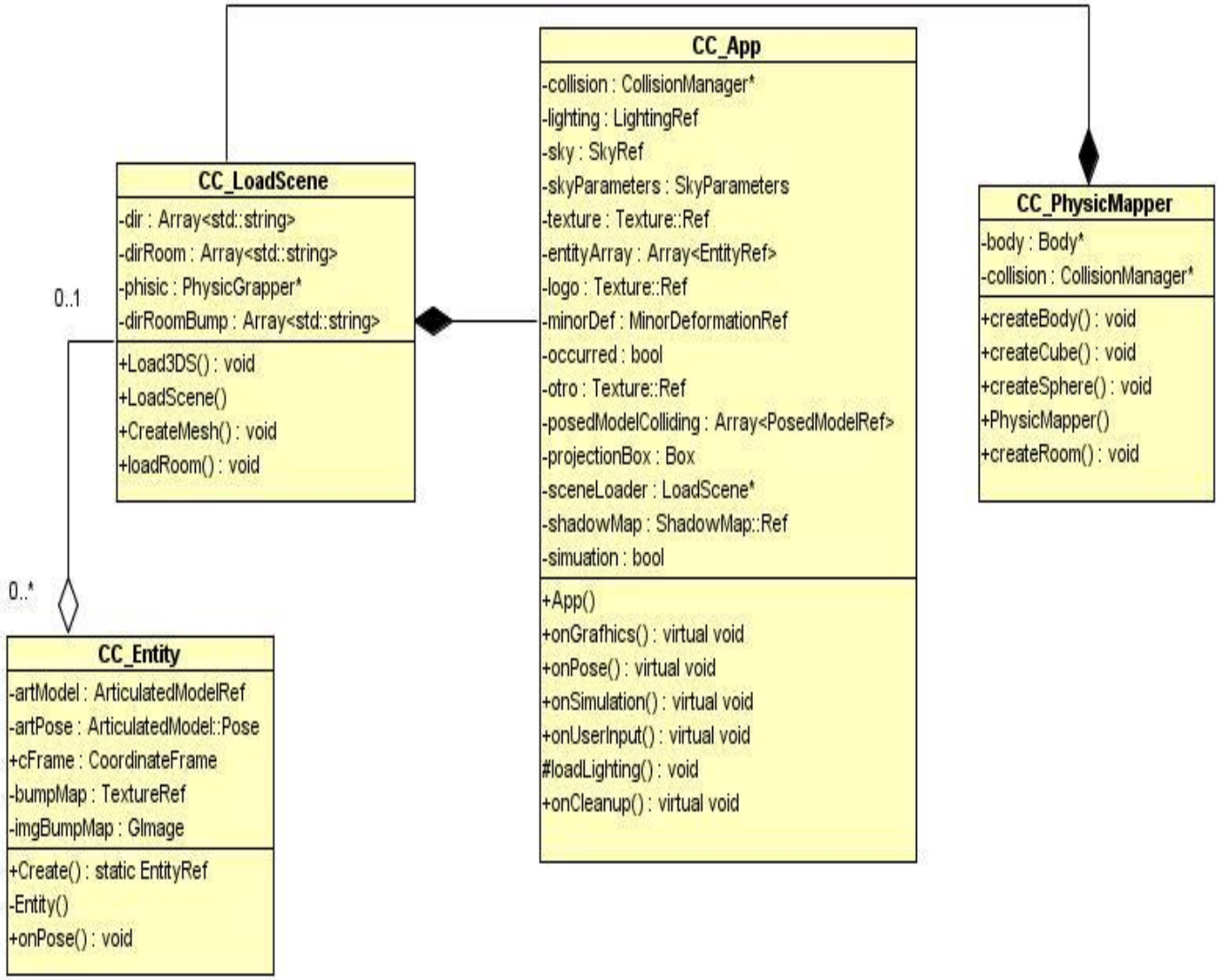
Anexo 1. Código de G3D para convertir el *bump map* en *normal map*.

```
void GImage::computeNormalMap(
    const GImage&       bump,
    GImage&             normal,
    bool                lowPassBump,
    bool                scaleHeightByNz) {

    const int w = bump.width;
    const int h = bump.height;
    const int stride = bump.channels;

    normal.resize(w, h, 4);

    const uint8* const B = bump.byte();
    Color4uint8* const N = normal.pixel4();

    for (int y = 0; y < h; ++y) {
        for (int x = 0; x < w; ++x) {
            // Index into normal map pixel
            int i = x + y * w;

            // Index into bump map *byte*
            int j = stride * i;


            Vector3 delta;

            // Get a value from B (with wrapping lookup) relative to (x, y)
            // and divide by 255
            #define height(DX, DY) ((B[((((DX + x + w) % w) + \
                                         ((DY + y + h) % h) * w) * stride]) / 255.0)


            // Sobel filter to compute the normal.
            //
            // Y Filter (X filter is the transpose)
            //  [ -1 -2 -1 ]
            //  [  0  0  0 ]
            //  [  1  2  1 ]

            // Write the Y value directly into the x-component so we don't have
            // to explicitly compute a cross product at the end.
            delta.y = -(height(-1, -1) *  1 + height( 0, -1) *  2 + height( 1, -1) *  1 +
                        height(-1,  1) * -1 + height( 0,  1) * -2 + height( 1,  1) * -1);

            delta.x = -(height(-1, -1) * -1 + height( 1, -1) * 1 +
                        height(-1,  0) * -2 + height( 1,  0) * 2 +
                        height(-1,  1) * -1 + height( 1,  1) * 1);

            delta.z = 1.0;


            delta = delta.direction();

            // Copy over the bump value into the alpha channel.
            float H = B[j] / 255.0;

            if (lowPassBump) {
                H = (height(-1, -1) + height( 0, -1) + height(1, -1) +
                        height(-1,  0) + height( 0,  0) + height(1,  0) +
                        height(-1,  1) + height( 0,  1) + height(1,  1)) / 9.0;
            }
            #undef height

            if (scaleHeightByNz) {
                // delta.z can't possibly be negative, so we avoid actually
                // computing the absolute value.
                H *= delta.z;
            }

            N[i].a = iRound(H * 255.0);

            // Pack into byte range
            delta = delta * 127.5 + Vector3(127.5, 127.5, 127.5);
            N[i].r = iClamp(iRound(delta.x), 0, 255);
            N[i].g = iClamp(iRound(delta.y), 0, 255);
            N[i].b = iClamp(iRound(delta.z), 0, 255);
        }
    }
}
```

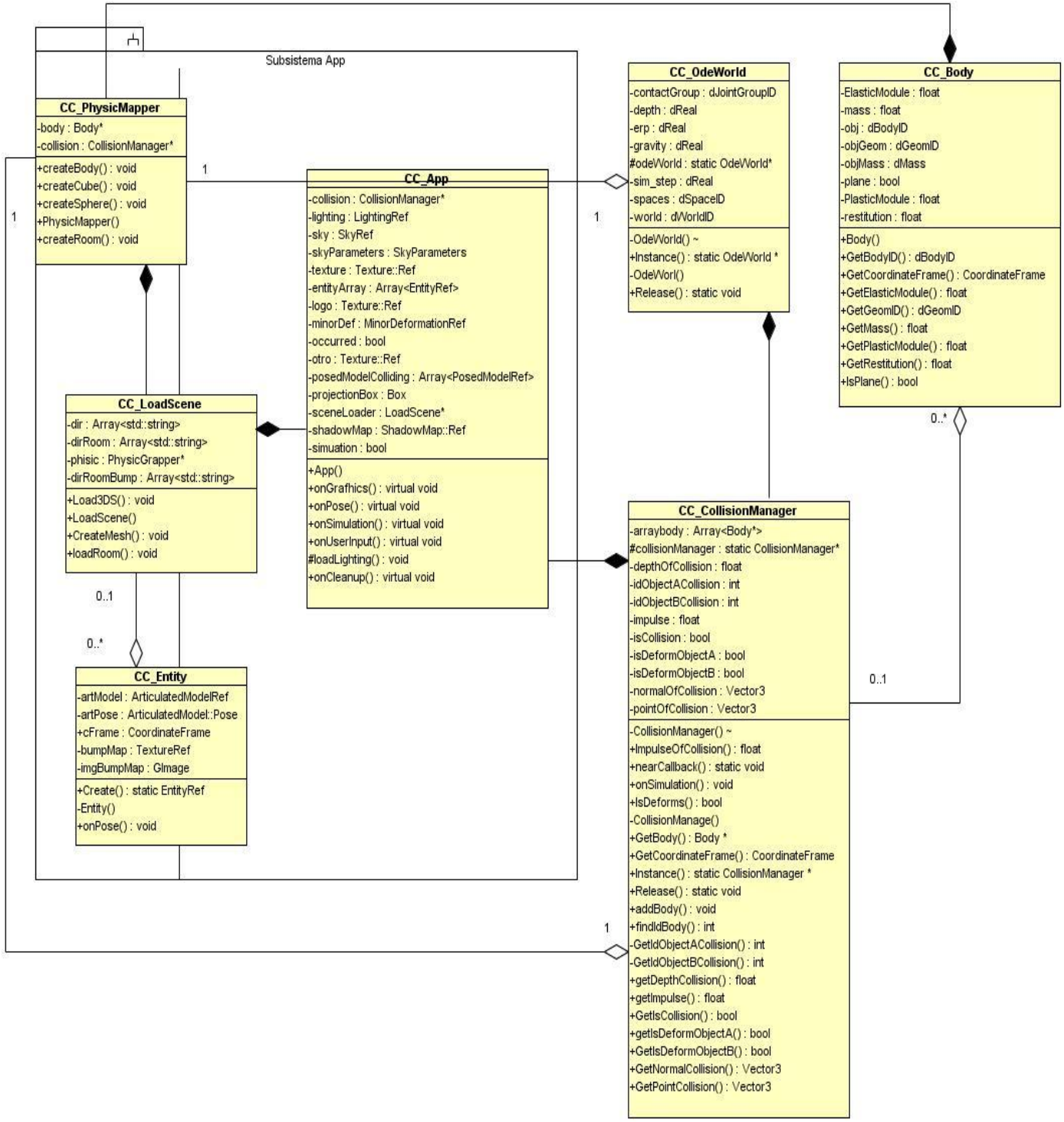
Anexo 2. Contenido de los *buffer* durante el cálculo de la deformación.



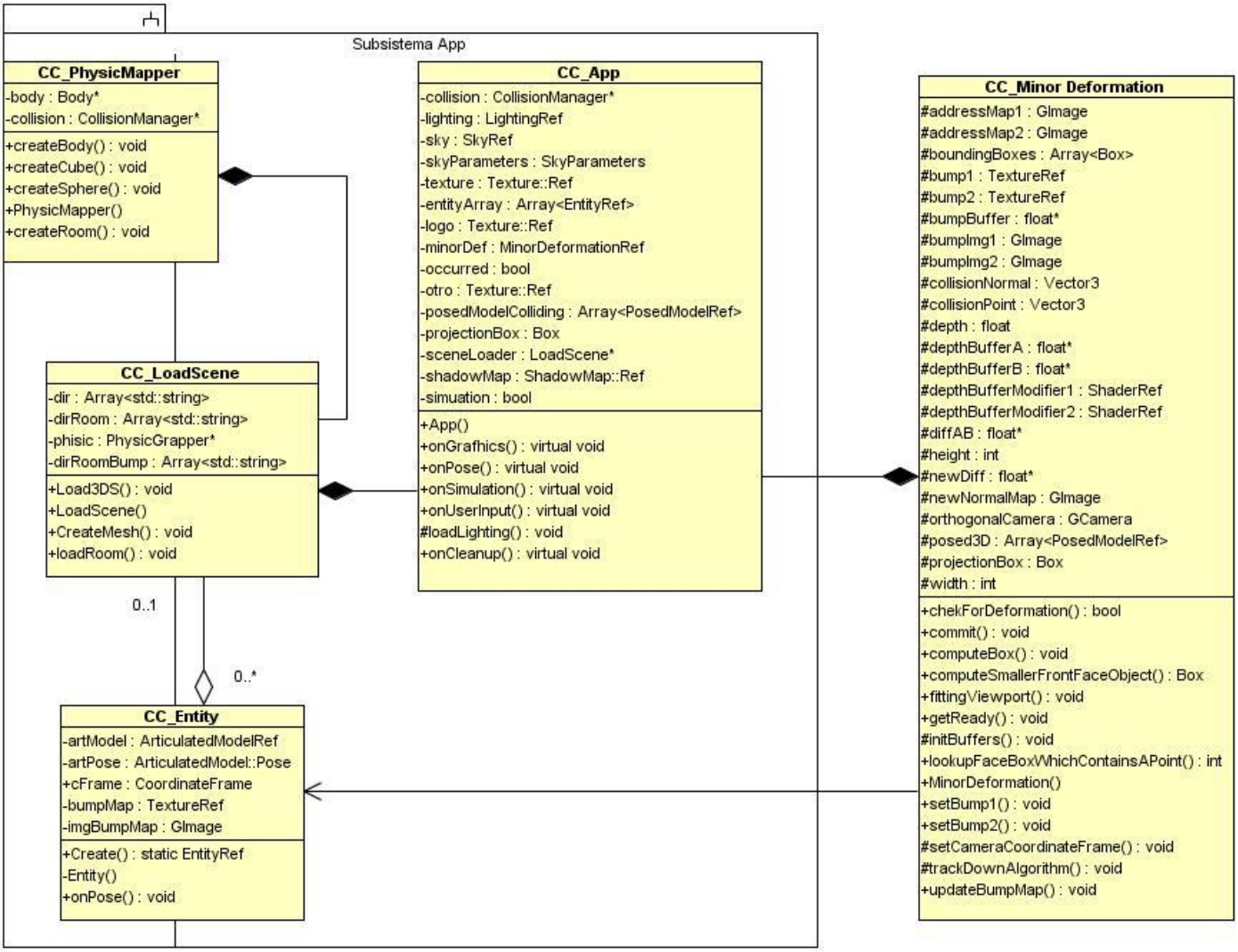
Anexo 3. Diagramas de Clases Subsistema de App.



Anexo 4. Diagrama de Clases Subsistema de Colisión.



Anexo 5. Diagrama de Clases Subsistema de Deformación.



Anexo 6. Código en C++ para procesar a mapa de normales, solamente un fragmento de una imagen.

```
void GImage::computeNormalMap(
        int width,
        int height,
        int channels,
        const uint8* src,
        GImage& normal,
        int startWidthIndex,
        int endWidthIndex,
        int startHeightIndex,
        int endHeightIndex,
        float whiteHeightInPixels,
        bool lowPassBump,
        boolscaleHeightByNz)

{

        if (whiteHeightInPixels == -1.0f)
        {
                // Default setting scales so that a gradient ramp
                // over the whole image becomes a 45-degree angle
                // Account for potentially non-square aspect ratios
                whiteHeightInPixels = G3D::max(width, height);
        }

    const int w = bump.width;
    const int h = bump.height;
    const int stride = bump.channels;

    normal.resize(w, h, 4);

    const uint8* const B = src;
    Color4uint8* const N = normal.pixel4();

    for (int y = startHeightIndex; y < endHeightIndex; ++y) {
        for (int startWidthIndex = 0; x < endWidthIndex; ++x) {
            // Index into normal map pixel
            int i = x + y * w;

            // Index into bump map *byte*
            int j = stride * i;


            Vector3 delta;

            // Get a value from B (with wrapping lookup) relative to (x, y)
            // and divide by 255
            #define height(DX, DY) ((B[(((DX + x + w) % w) + \
                                        ((DY + y + h) % h) * w) * stride]) / 255.0)


            // Sobel filter to compute the normal.
            //
            // Y Filter (X filter is the transpose)
            //  [ -1 -2 -1 ]
            //  [  0  0  0 ]
            //  [  1  2  1 ]

            // Write the Y value directly into the x-component so we don't have
            // to explicitly compute a cross product at the end.
            delta.y = -(height(-1, -1) *  1 + height( 0, -1) *  2 + height( 1, -1) *  1 +
                        height(-1,  1) * -1 + height( 0,  1) * -2 + height( 1,  1) * -1);

            delta.x = -(height(-1, -1) * -1 + height( 1, -1) * 1 +
                        height(-1,  0) * -2 + height( 1,  0) * 2 +
                        height(-1,  1) * -1 + height( 1,  1) * 1);

            delta.z = 1.0;


            delta = delta.direction();

            // Copy over the bump value into the alpha channel.
            float H = B[j] / 255.0;

            if (lowPassBump) {
                H = (height(-1, -1) + height( 0, -1) + height(1, -1) +
                        height(-1,  0) + height( 0,  0) + height(1,  0) +
                        height(-1,  1) + height( 0,  1) + height(1,  1)) / 9.0;
            }
            #undef height

            if (scaleHeightByNz) {
                // delta.z can't possibly be negative, so we avoid actually
                // computing the absolute value.
                H *= delta.z;
            }

            N[i].a = iRound(H * 255.0);

            // Pack into byte range
            delta = delta * 127.5 + Vector3(127.5, 127.5, 127.5);
            N[i].r = iClamp(iRound(delta.x), 0, 255);
            N[i].g = iClamp(iRound(delta.y), 0, 255);
            N[i].b = iClamp(iRound(delta.z), 0, 255);
        }
    }
}
```

Anexo 7. Seudo–código del algoritmo.

BumpDeformA (CPU)

$deform_A = t_A \cdot \kappa_A \cdot \max(0, ||j_{\varepsilon=1}|| - (1/(1-\varepsilon_A) - 1))$
$pos_A = N_C \cdot (deform_A - d_C)$

$pos_{cam} = P_C + N_C \cdot s_{cam}$
$look_{cam} = -N_C$
$up_{cam} = (0,1,0)$
if (abs($up_{cam} \cdot look_{cam}$) > 0.9)
  $up_{cam} = (1,0,0)$
create orthographic projection camera at pos_{cam} with look vector $look_{cam}$ and up vector up_{cam}
resize the camera's viewport to fit the smaller of A and B

glDisable(GL_LIGHTING)
glEnable(GL_DEPTH_TEST)
glDepthMask(GL_TRUE)
glEnable(GL_CULL_FACE)
glDrawBuffer(GL_BACK)
glReadBuffer(GL_BACK)
glClearColor(0, 0, 1, 1)
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

glDepthFunc(GL_LESS)
glCullFace(GL_BACK)
set fragment shader *RenderPassA*
render(A)
glReadPixels(x,y,w,h, GL_RGB, GL_UNSIGNED_BYTE, C_A)
glReadPixels(x,y,w,h, GL_DEPTH_COMPONENT24_ARB, GL_FLOAT, D_A)

glClear(GL_COLOR_BUFFER_BIT)

glDepthFunc(GL_GREATER)
glCullFace(GL_FRONT)
set fragment shader *RenderPassB*
render(B)
glReadPixels(x,y,w,h, GL_RGB, GL_UNSIGNED_BYTE, C_B)
glReadPixels(x,y,w,h, GL_DEPTH_COMPONENT24_ARB, GL_FLOAT, D_B)

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
glDepthFunc(GL_LESS)

glRasterPos2i(x,y)
glDrawPixels(w,h, GL_RGB, GL_UNSIGNED_BYTE, h_A)

glBegin(GL_POINTS)
for each pixel $[i,j]$:
  $(r_A, g_A, b_A) = C_A[i,j]$
  $(r_B, g_B, b_B) = C_B[i,j]$
  $\Delta D[i,j] = D_B[i,j] - D_A[i,j]$

  if ($b_A = 1$ or $b_B = 1$ or $\Delta D[i,j] = 0$)
    go to next pixel

  $h_A = h_A[r_A, g_A]$
  $h'_A = h_A - (\Delta D[i,j] \cdot (z_{far} - z_{near}))/k$
  clamp(h'_A, 0, 1)

  glColor3f(h'_A, h'_A, h'_A)
  glVertex3f(r_A, g_A, h'_A)
glEnd()

glCopyTexImage2D(h_A, 0, GL_RGB, x,y,w,h, 0)

position of $A = pos_A$

RenderPassA (GPU)

h = texture2D(h_A, gl_TexCoord[0].xy)
gl_FragDepth = gl_FragCoord.z - $h * k'/(z_{far} - z_{near})$
gl_FragColor = (gl_TexCoord[0].xy, 0)

RenderPassB (GPU)

h = texture2D(h_B, gl_TexCoord[0].xy)
gl_FragDepth = gl_FragCoord.z + $h * k'/(z_{far} - z_{near})$
gl_FragColor = (gl_TexCoord[0].xy, 0)