VISUALIZACIÓN Y DEFORMACIÓN DE OBJETOS VIRTUALES 3D

Raúl Andrés Gómez Ruiz
Faber Ramiro Montero Calderón

Universidad del Cauca
Facultad de Ingeniería Electrónica y Telecomunicaciones
Departamento de Electrónica, Instrumentación y Control
Ingeniería en Automática Industrial
Popayán, Abril de 2012

VISUALIZACIÓN Y DEFORMACIÓN DE OBJETOS VIRTUALES 3D

Monografía presentada como requisito parcial para optar por el título de Ingeniero en Automática Industrial

Raúl Andrés Gómez Ruiz
Faber Ramiro Montero Calderón

Director: Elena Muñoz España

Universidad del Cauca
Facultad de Ingeniería Electrónica y Telecomunicaciones
Departamento de Electrónica, Instrumentación y Control
Ingeniería en Automática Industrial
Popayán, Abril de 2012

Nota de aceptación: _____

_____

_____

_____

_____

_____

_____
Firma del Presidente del Jurado

_____
Firma del Jurado

_____
Firma del Jurado

Popayán, Abril de 2012

Agradecimientos

A nuestras familias por su apoyo y comprensión.

A la ingeniera Elena Muñoz España por su instrucción como directora y docente.

A la Universidad del Cauca por formarnos como profesionales.

Resumen

En este proyecto se presenta el desarrollo de una aplicación software 3d para deformación de objetos basada en software libre. Esta aplicación permitirá al usuario deformar un objeto mediante acciones de presión y estiramiento sobre un punto de la superficie del objeto 3d además de que este objeto recupere su forma original después de deformarlo.

El funcionamiento de la aplicación comienza con la importación de un objeto rígido desde los formatos vtk, vtp, obj o ply, luego se hace una conversión de este objeto a un modelo deformable, posteriormente se recurre al uso de una herramienta 3d que es manejada por el mouse con el propósito de deformar el objeto.

Para la construcción de esta aplicación se emplearon herramientas de software libre. Estas herramientas son librerías bajo el lenguaje C++ que básicamente conforman tres aspectos fundamentales en el desarrollo de la aplicación, estos tres aspectos son: visualización, detección de colisiones e interfaz gráfica de usuario. Las librerías utilizadas para la implementación de la aplicación son: Las librerías VTK encargadas de la visualización 3d, VCollide que hace posible la detección de colisiones entre dos objetos 3d, y Qt útil para el desarrollo de interfaces gráficas de usuario.

Abstract

This project presents the development of a software application for 3D object deformation based on free software. This application allows the user to deform an object by using pressure and stretching actions on a point on the surface of the 3D object that this object also regains its original shape after stretching.

The operation of the application begins with the importation of a rigid object from vtk formats, vtp, obj or ply, then a conversion is made of this object to a deformable model then is used to using a 3D tool that is managed by the mouse for the purpose of deforming the object.

To build this application is free software tools used. These tools are libraries in C + + language that basically made up of three fundamental aspects in the development of the application, these three aspects are: visualization, collision detection and graphical user interface. The libraries used for the implementation of the application are: VTK libraries responsible for 3D visualization, which enables VCollide collision detection between two 3D objects, and Qt useful for developing graphical user interfaces.

Contenido

LISTA DE FIGURAS

LISTA DE TABLAS

1. INTRODUCCIÓN

En el área de visualización y gráficos por computadora, se ha trabajado con modelos deformables desde hace ya varios años debido a que permiten animar y visualizar el comportamiento físico de objetos 3d [**1**], [**2**]. Entre las aplicaciones que emplean estos modelos se encuentran los entornos de simulación quirúrgica, deformación de materiales, juegos de video, modelado de telas en personajes animados, entre otros [**3**], [**4**], [**5**].

En el desarrollo del proyecto se han tenido en cuenta diferentes investigaciones donde se aborda el tema de deformación de objetos. Entre las principales se encuentran:

- "Interacción con Objetos Deformables" [**2**] describe la deformación de una malla de cuadriláteros en un espacio tridimensional. La deformación de la malla está basada en un sistema masa resorte amortiguador entre sus vértices, que utiliza el método de diferencias finitas para resolver la ecuación del modelo. La deformación se realiza sobre un objeto con geometría establecida previamente, es decir que solo se aplica a un objeto en particular. El modelo masa resorte amortiguador no tiene fuerzas internas que permitan simular un malla más rígida. El contacto de este modelo deformable está permitido sólo sobre los vértices de la malla. Esta herramienta se implementó con OpenGL.

- "Manipulación Tridimensional de Objetos Deformables Virtuales" [**5**] describe un sistema para la manipulación en tiempo real de objetos deformables virtuales sin retroalimentación de fuerzas. Los objetos se construyen a partir de mallas formadas por vértices que se le asocian tres vértices vecinos. Implementa un modelo masa resorte amortiguador, sin fuerzas internas en el objeto. La deformación se realiza sobre la superficie de un objeto con geometría fija. Esta herramienta se implementó con OpenGL.

- "Animación de Modelos Deformables" [**6**] describe la animación de modelos deformables elásticos e inelásticos, basado en mallas formadas por vértices conectados a tres vértices vecinos. Para la animación se utilizó el modelo de masa resorte amortiguador. El sistema no permite deformar objetos en tiempo real. Esta herramienta se implementó con OpenGL.

- "Implementación de Objetos Deformables en un Simulador Dinámico" [**7**] describe el comportamiento en tiempo real del modelo deformable masa resorte implementado con las librerías de código abierto OpenTissue dentro de un simulador virtual. El modelo deformable considera una malla superficial con fuerzas internas. La deformación se realiza sobre objetos en formato genérico .obj.

- "Visualización 3D de Deformación y Corte de Objetos Virtuales Basada en Descomposición Ortogonal" [**8**] presenta la generación de un mallado inercial. Un mallado inercial es un conjunto de puntos a los cuales se les asigna una masa, por tanto son interconectados por resortes y amortiguadores a lo largo de las aristas que los unen.

- "ParSys" [**3**] describe un nuevo modelo deformable que permite deformar objetos 3d basándose en un algoritmo de volúmenes ligados. Este modelo deformable se enfoca principalmente en la velocidad por encima de la precisión, conservando características propias de los modelos basados en física. En este modelo existe conexiones entre

partículas, estas conexiones determinan las fuerzas internas que deberán aplicarse al objeto deformable y son las que permitirán al modelo recuperar su forma original.

En el presente proyecto se realiza la implementación de un algoritmo para la modelación de las deformaciones, es decir un modelo deformable que permita realizar un cambio en la forma de los objetos, se experimenta con deformaciones elásticas de estiramiento o presión ejercidas sólo en un punto de la superficie del objeto. El algoritmo seleccionado permite favorecer la velocidad de renderizado, en el sentido que dicho algoritmo no afecta considerablemente el desempeño visual de la aplicación. Respecto a la evaluación de la aplicación se realiza en dos partes: La primera parte evalúa el algoritmo de deformación para observar deformación, tiempo de repuesta y renderización; la segunda parte es una evaluación de la aplicación final, donde se verifica que el sistema cumple con su función.

En este proyecto se escogió un modelo basado en resortes debido a sus características físicas, pues representa un comportamiento elástico, además de ser una configuración simplificada que no ocasiona tanta carga computacional resultado de un sistema de ecuaciones poco complejas [**7**].

Como sistema de detección de colisiones se ha escogido la librería VCollide debido que en comparación con otras librerías registra el menor tiempo promedio de ejecución y por la facilidad de procesamiento con mallas triangulares [**9**].

Como herramienta principal se ha escogido las librerías VTK porque son de código y distribución libre que no requieren licencia. VTK tiene un mayor nivel de abstracción que otras librerías de renderización como OpenGL o PEX soportando diferentes lenguajes de programación como C++, Tcl, Java y Piton además de poderse compilar para Linux, Unix y Windows, asimismo soportan distintos tipos de hardware gráfico [**10**], [**11**].

La funcionalidad de la aplicación desarrollada es básicamente importar un objeto rígido en los formatos vtk, vtp, obj o ply y convertir el objeto importado en un objeto deformable. La deformación consiste en presionar o estirar un punto de la superficie del objeto y que este objeto al ser deformado recupere su forma original.

El trabajo está organizado en seis capítulos y CUATRO ANEXOS complementarios. En el primer capítulo se presenta un resumen del trabajo desarrollado y la organización del presente documento. En el segundo capítulo se expone una introducción de los temas de visualización de objetos 3d, modelos deformables y detección de colisiones. En el capítulo tres se hace un análisis de requisitos empleando la ingeniería de requerimientos con el fin de identificar las funcionalidades y restricciones de la aplicación. En el capítulo cuatro se plantea un sistema de resortes que definirá el modelo deformable así mismo se expondrá la descripción estructural de la aplicación. En el capítulo cinco se presentan los resultados y pruebas de la aplicación para un análisis de su funcionalidad. El capítulo seis, contiene una exposición de las conclusiones y futuros trabajos que se pueden abordar sobre temas relacionados con el desarrollo presentado. Finalmente se incluyen todas las referencias del presente trabajo.

En el anexo A se presenta una guía de instalación de la aplicación. El anexo B contiene el manual de usuario de la aplicación exponiendo el manejo de la interfaz grafica de usuario. El anexo C expone la descripción estructural de una aplicación implementada con la clase vtkCollisionDetectionFilter como sistema de detección de colisiones. El anexo D muestra

ejemplos de pequeñas aplicaciones en VTK, aplicaciones que son base en la implementación de la aplicación software 3d.

2. FUNDAMENTOS TEÓRICOS

Este capítulo presenta los conceptos generales sobre visualización de objetos 3d, modelos deformables y detección de colisiones. Además se expone los distintos métodos de deformación de objetos.

2.1. VISUALIZACIÓN DE OBJETOS 3D

Los gráficos por computador permiten simular la percepción del mundo real tridimensional como una forma de comunicación para el uso de muchas aplicaciones. Todo esto es gracias a un proceso hardware y software que permite crear una ilusión óptica en pantalla para la visualización 3D [**12**], [**11**].

En el universo real, es posible notar tres dimensiones de los objetos que nos circundan. En la pantalla, la profundidad no existe, es la dimensión que se simula al graficar objetos 3D. Para realizar la simulación de esta tercera dimensión, se suman los efectos visuales y se proyectan los objetos 3D sobre 2D. Además se debe considerar que un modelo 3D, se encarga de reemplazar una idealización matemática, la cual ha sido empleada para representar la información de un objeto virtual [**12**].

La visualización gráfica representa un tipo de conexión, mediante la cual se puede interpretar la transformación de datos o información en términos visuales o imágenes. Por lo que la visualización estará muy conectada con uno de los principales sentidos del hombre, la visión, y también con la capacidad que tiene el cerebro humano. Lo que resulta un mecanismo simple, pero muy efectivo de enlazar información engorrosa y voluminosa. La visualización usa las destrezas naturales del sistema visual humano. Las simbologías visuales son muy sencillas de entender; no sólo las de 2D, sino que se puede obtener una imagen cerebral en 3D de un objeto. Lo que conduce a la visualización interactiva, para obtener un objeto, y así conseguir una idea tridimensional del mismo [**11**].

Los usos iníciales de la visualización se dieron en ingeniería y trabajos científicos. Desde que se introdujo la computadora personal, se empezó a usar como herramienta de cálculo y simulación de procesos físicos como, trayectorias en balística, movimiento de fluidos, mecánica estructural. Las aplicaciones de la visualización no cambian, en cualquier campo de tecnología existente, pues por medio de ésta se posibilita la representación de los resultados de experimentos, simulaciones, y datos medidos; y usar de esta manera las imágenes para comunicar, entender, y entretener [**11**].

2.1.1. Objeto 3D

Un objeto 3d es una estructura geométrica compuesta por una configuración de polígonos que describe la superficie de un sólido [**12**].

La configuración de polígonos, definido como malla, se constituye de otros componentes conocidos como primitivas gráficas obtenidas a partir de una colección de datos, éstas primitivas son: punto, línea y polígono [**11**].

Punto: Terna ordenada de números reales (x, y, z), que representa una localización en el espacio.

Línea: Conexión entre dos puntos.

Polígono: Figura geométrica determinada por segmentos de rectas que confinan un área.

El polígono entonces está formado por líneas (aristas) y puntos (vértices). Su clasificación se define por el numero de aristas, el más básico de todos es el triángulo que se compone de tres aristas y tres vértices y del cual se derivan todos los demás polígonos [**12**].

Partiendo del polígono base entonces se define una malla triangular como una estructura constituida por triángulos unidos por sus aristas para conformar una superficie, así pues la faceta de cada uno de los triángulos permite la construcción de un objeto 3d (Figura 2.1) [**2**].

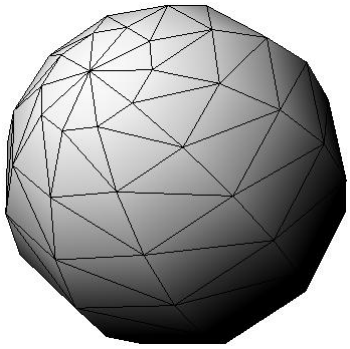


Figura 2.1: Objeto 3D

2.1.2. Renderización

La visualización como ya se ha dicho, es la técnica de modificar los datos en una sucesión de primitivas gráficas, es decir representa el proceso de generación de imágenes mediante un ordenador, o el proceso de conversión de datos gráficos a una imagen. Por lo que su uso hace énfasis en convertir estas primitivas en imágenes y animaciones. La renderización es un proceso de cálculo matemático que genera imágenes a partir de datos gráficos para que puedan ser representados en una pantalla [**11**].

En el mundo de la renderización existen dos tipos: uno representa la renderización lenta (*prerendering*), utilizado en películas e imágenes de calidad superior, y el otro el *real-time rendering*, empleado para simulaciones dadas en tiempo real y juegos; este último tiene como propiedad fundamental que debe darse mediante una transformación rápida para crear la ilusión de realismo [**12**].

La mayor parte de procesos de renderización se modifican desde los programas de dibujo en 2D hasta las más complejas técnicas en 3D. El propósito de la renderización, no está en el realismo alcanzado, como si lo es el contenido de información, aunque cabe mencionar que esto depende, de la aplicación. Además, es importante alcanzar representaciones interactivas, en las que se logre interactuar con los datos [**11**].

Para visualizar objetos 3d entonces el trabajo de la renderización es procesar la geometría del objeto para proyectarse en el área 2d de la pantalla y así obtener una imagen (Figura 2.2) [**12**].

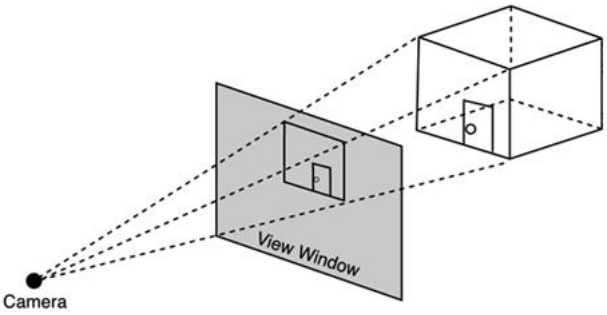


Figura 2.2: Proyección de un objeto 3D sobre la pantalla

Fuente: [**12**]

2.1.3. Cámaras

Los actores representan objetos (geometría y propiedades), que con rayos de luz emitidos por fuentes luminosas producen cualidades en sus superficies. En la escena se tiene la interacción con la luz, lo que produce un color de composición, el cual se conoce como un color producto del efecto de ambiente, la combinación de la luz, el efecto especular, y la superficie del objeto. Ahora lo que se requiere completamente para renderizar una escena es una cámara. Hay un conjunto importante de factores que determinan cómo se proyecta la escena 3D sobre un plano para formar la imagen 2D (Figura 2.3). Estos factores serán entonces la orientación, la posición, y punto focal de la cámara, los dos últimos definen la posición de la cámara y el punto al que enfoca, la orientación de la cámara se controla mediante su posición y punto focal, además del vector que indica la vista superior de la cámara [**11**].

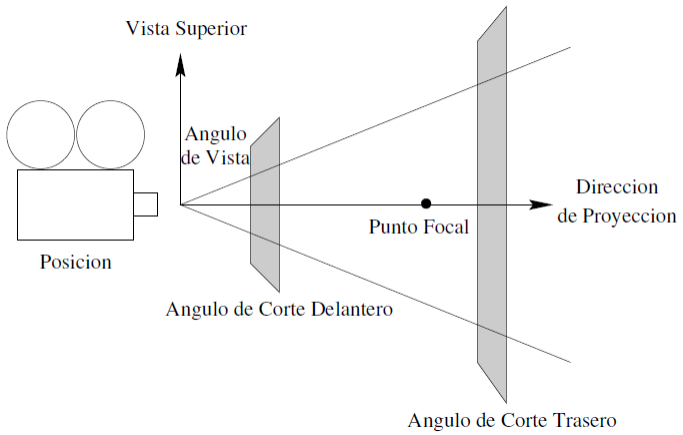


Figura 2.3: Atributos de la cámara

Fuente: [**11**]

El vector situado desde la posición de la cámara hasta el punto focal se llama dirección de proyección. Situándose entonces el plano de imagen de la cámara en el punto focal, y es, por lo general, perpendicular al vector de proyección. La técnica de proyección verifica cómo se ubican los actores en el plano de la imagen. Además que en proyección ortográfica o paralela, todos los rayos de luz que entran en la cámara son paralelos al vector de proyección. La proyección en perspectiva se da al pasar todos los rayos de luz por un centro de proyección. Siendo por lo general perpendiculares a él los planos de

corte delantero y trasero que cortan al vector de proyección y de esta manera ser usados para eliminar datos que se hallan en la cámara, o muy cerca, o bien, muy lejos de ella. Como efecto de esto sólo los actores o porciones de actores que están entre los dos planos de corte pueden conseguir ser visibles. Por lo general estos planos son perpendiculares a la dirección de proyección [**11**].

2.1.4. Cuadros por segundo

Frame o cuadro se denomina a cada imagen mostrada en pantalla, y de este modo cada pasada del *rendering* gráfico terminará en un *frame*. Los cuadros por segundo o *frames per second* (FPS), son una medida muy empleada con la cual se determina el rendimiento de una aplicación 3D en tiempo de ejecución, en consecuencia se puede presentar la situación en que la placa gráfica no pueda ejecutar a tiempo todos los cálculos matemáticos para poder renderizar. Si sucede de esta manera, los FPS se disminuirán y por tanto la aplicación 3D se verá afectada [**12**].

Cuando los FPS se reducen, entonces es en ese momento cuando el usuario pierde cuadros o escenas de lo que está pasando en el mundo virtual. Por citar el caso en el que un automóvil se esté moviendo a una cierta velocidad del mundo virtual y los FPS se reducen sensiblemente, entonces el automóvil no disminuirá su velocidad, pero el usuario no podrá ver el movimiento completo, es decir, se saltará unos cuantos cuadros de la animación. Un muy buen valor de FPS ronda los 30 cuadros. En el caso de la televisión o el cine, en los cuales para lograr la ilusión de movimiento, se deben alcanzar 24 FPS, pero en el caso que el valor llegase a ser menor, se vería únicamente un parpadeo de la imagen [**12**].

2.2. MODELOS DEFORMABLES

En el campo de la informática gráfica, los modelos deformables son objetos constituidos por un conjunto de polígonos que se representan en un entorno virtual con ciertas propiedades que permiten la modificación de su forma [**4**].

Existen diversos modelos deformables que emulan el cambio de forma de un objeto bajo la acción de una fuerza externa en un área del objeto, que de acuerdo a sus características, resulta en unos sistemas con un comportamiento más real que otros y por ende un tiempo de cómputo de acuerdo al realismo en algunos casos [**7**].

2.2.1. Métodos geométricos

Los modelos deformables basados en métodos geométricos presentan la particularidad de funcionar con poco realismo debido a varios factores como imprecisión, mallas irregulares, recuperación de forma y la incapacidad de interactuar directamente sobre los objetos.

2.2.1.1. *Splines* deformables

Este método se basa en cambiar la forma de un volumen a partir de un conjunto de puntos, denominados puntos de control, que permiten obtener superficies curvas al mover estos puntos (Figura 2.4) [**4**].

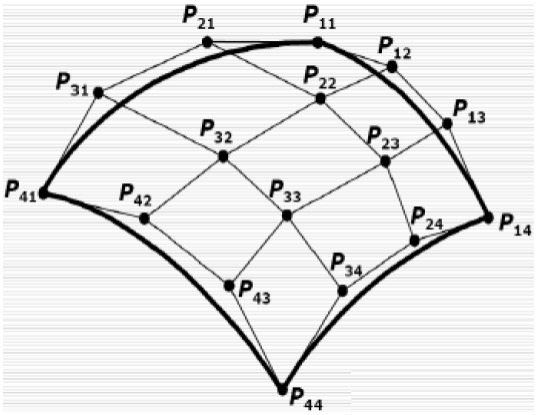


Figura 2.4: Curva spline con sus puntos de control

Fuente: [**4**]

Con los *splines* se obtienen curvas suavizadas y debido a esto se requiere que la superficie que conforma el objeto tenga un número elevado de polígonos, es decir alta resolución, lo que significa un gran impacto sobre el coste computacional. Adicional a esto no hay precisión en la deformación como resultado de discontinuidades en el modelo causando que los puntos de la superficie se muevan incorrectamente [**4**], [**5**].

2.2.1.2. Modelo *chain-mail*

Este método fue desarrollado originalmente por Sarah Gibson [**13**], trabaja por medio de eslabones enlazados entre sí (Figura 2.5), semejante a la estructura de una cadena, de tal manera que si se ejerce una acción sobre un eslabón, los eslabones contiguos se verán afectados por esa acción. Cada eslabón tiene cierto grado de movilidad donde no afecta a los eslabones vecinos, determinando así el nivel de flexibilidad del objeto [**4**].

Este método presenta el inconveniente de no recuperar totalmente el objeto a su forma original al producirse la deformación, además de limitarse a funcionar en mallas uniformes [**4**], [**5**].

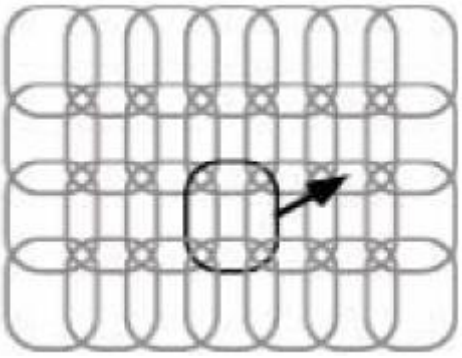


Figura 2.5: Modelo Chain-Mail

Fuente: [**4**]

2.2.1.3. Deformación de libre forma

El método de deformación de libre forma o FFD (*Free Form Deformation*) desarrollado por Thomas Sederberg y Scott Parry [**14**], es similar al método de *splines* en cuanto a la deformación por medio de puntos de control, pero estos puntos están organizados en una rejilla (arreglo tridimensional) que conforman un espacio en forma de paralelepípedo (Figura 2.6). En este espacio se alberga el objeto a deformar cuyos puntos son modificados de acuerdo a funciones paramétricas asignadas a la rejilla, de esta manera si se deforma el espacio definido también lo hará el objeto [**4**].

La desventaja de este método es que no se ejerce una acción directa sobre la superficie del objeto [**5**].

Figura 2.6: Rejilla tridimensional

Fuente: [**4**]

2.2.2. Métodos basados en física

Se usan en modelos deformables, los cuales responden de una manera natural a las fuerzas aplicadas, los medios de comunicación ambiental, las limitaciones, y las colisiones con otros objetos en un entorno físico simulado, ofreciendo de esta manera un realismo sin igual en el modelado de fenómenos naturales, basado en los principios que rigen las leyes de la física Newtoniana [**15**].

2.2.2.1. Método de elementos finitos

El método consiste en dividir los objetos deformables en un conjunto finito de elementos interconectados entre sí por una serie de puntos llamados nodos. Las propiedades físicas del objeto son interpoladas para cada elemento usando ecuaciones diferenciales, de manera que la mecánica continua del objeto es expresada en términos de un conjunto de elementos [**16**].

A pesar de que este método genera simulaciones más realistas físicamente es costoso computacionalmente debido a su complejidad y cantidad de cálculos requeridos [**4**], [**3**].

2.2.2.2. Método de elementos de contorno

Es un método similar al FEM, la diferencia radica en que sus cálculos son realizados sobre la superficie del objeto elástico y no en todo el volumen [**4**].

2.2.2.3. Modelo masa-resorte

Este método consiste en un conjunto de puntos que se les atribuye una masa y se interconectan entre sí por medio de resortes (Figura 2.7). Estos puntos están distribuidos de acuerdo a una estructura definida, de forma que si un punto es afectado por una fuerza, este punto afectara a sus vecinos [**4**].

El desarrollo pionero de este método fue realizado por Demetri Terzopoulos y Kurt Fleischer [**15**].

Este método muestra un comportamiento realista ya que está basado en descripciones físicas como la segunda ley de Newton y la ley de Hooke, sin embargo este comportamiento depende de la resolución de la malla [**4**].

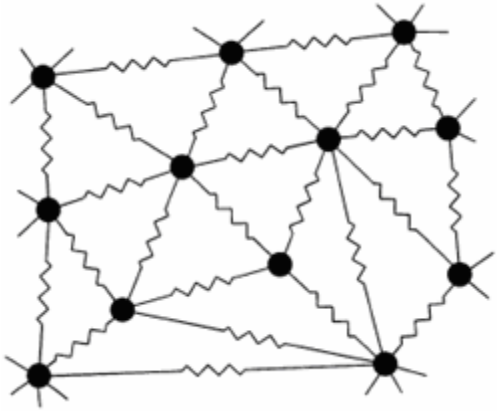


Figura 2.7: Modelo Masa-Resorte

Fuente: [**7**]

2.3. SISTEMA MASA RESORTE AMORTIGUADOR

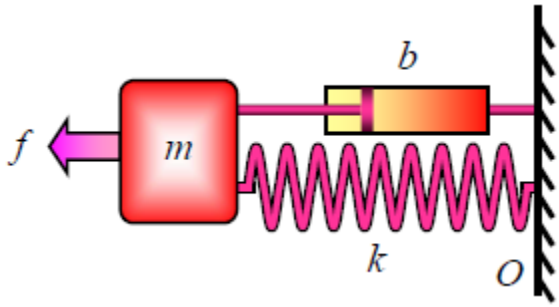


Figura 2.8: Sistema masa resorte amortiguador

Fuente: [**5**]

Un resorte lineal corresponde a una pieza mecánica que permite ser deformada por una fuerza externa, de manera que tal deformación, sea directamente proporcional a la fuerza aplicada [**17**].

La mayoría de sistemas físicos se componen de algún amortiguamiento, ya sea de tipo seco, magnético, viscoso, etcétera. Este no sólo retrasa el movimiento, sino que puede suceder el caso en el que lo frene (Figura 2.8) [**17**].

Con un sistema sin amortiguamiento la energía mecánica del sistema se mantiene, en consecuencia el movimiento se prolongará indefinidamente. El uso de un amortiguador, hace que el sistema llegue al reposo. Este modelo produce una fuerza de amortiguamiento opuesta al movimiento y será directamente proporcional a la magnitud de la velocidad [**2**]. De forma que se tiene,

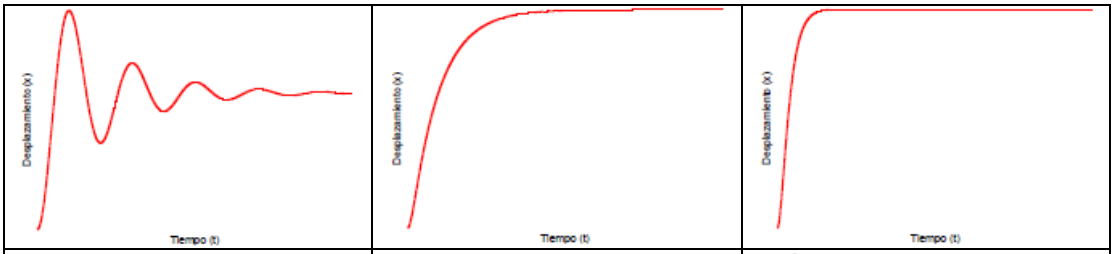
$$F = m\ddot{x} + b\dot{x} + kx$$

Siendo:

$\ddot{x}$ aceleración, $\dot{x}$ velocidad, x posición, b constante de amortiguamiento, m masa y k constante de elasticidad.

Las características del movimiento resultante del sistema, vendrán dadas por el factor de amortiguamiento, por tanto la masa tenderá en algún momento a detenerse debido al decaimiento de energía mecánica [**2**].

Existen tres tipos de amortiguamiento, según sea la solución a la ecuación diferencial del sistema: Subamortiguado, sobreamortiguado y críticamente amortiguado.



Subamortiguado	Sobreamortiguado	Críticamente amortiguado

Figura 2.9: Tipos de amortiguamiento

Fuente: [**5**]

2.4. DETECCIÓN DE COLISIONES

“Una colisión es una configuración en la que dos objetos ocupan parte de una misma porción del espacio al mismo tiempo” [**18**].

Un sistema de colisiones consiste entonces en calcular la intersección entre primitivas geométricas de dos objetos, es decir que todos los polígonos que conforman la superficie de un objeto intercepten satisfactoriamente la superficie de otro objeto [**19**].

2.4.1. Esfera envolvente

Esta técnica también llamada BS (*Bounding Sphere*) consiste en tomar objetos a colisionar y envolverlos en esferas, la colisión se da cuando se compara la distancia de los centros de cada esfera, siendo esta distancia menor a la suma de los radios de las esferas [**18**].

Esta técnica tiene la desventaja de no ajustarse a objetos de distinta geometría generando así falsos reportes [**18**].

2.4.2. Caja envolvente alineada con los ejes

Normalmente conocida como AABB (*Axis Aligned Bounding Box*) consiste en envolver los objetos colisionables en cajas que están alineadas con el sistema de coordenadas x, y y z. La colisión se realiza cuando se compara las posiciones de los ejes coordenados con respecto al centro de cada objeto [**20**].

Esta técnica disminuye la aparición de falsos reportes de colisiones con respecto a la técnica BS o esfera envolvente, sin embargo genera falsos reportes si los objetos se encuentran inclinados o rotados además de no ajustarse a muchos objetos de diferentes geometrías [**20**].

2.4.3. Caja envolvente orientada

Esta técnica conocida como OBB (*Oriented Bounding Box*) se basa en encerrar los objetos a colisionar dentro de cajas que se orientan según la alineación de los objetos [**20**].

Esta técnica está limitada para objetos cuya forma se adapten a geometrías rectangulares [**20**].

2.4.4. Rejilla de vóxeles

Este método se fundamenta en la partición del espacio en celdas rectangulares y uniformes, relacionando así una celda a un objeto. La colisión existe cuando los objetos ocupen la misma celda [**18**].

2.4.5. Octrees

Esta técnica consiste en una partición jerárquica del espacio, es decir una división que consiste en fraccionar un volumen ocho veces, estos ocho volúmenes son particionados en otros ocho volúmenes y así sucesivamente creando de esta forma una división recursiva [**18**].

2.4.6. K-d trees

Esta técnica se basa en dividir cada región en dos partes por un plano alineado con los ejes, permitiendo realizar divisiones uniformes o variables del espacio. Las divisiones variables, aunque necesitan mayor almacenamiento para la estructura, se adaptan mejor a los objetos [**18**].

2.4.7. BSP trees

Esta técnica es una estructura jerárquica que subdivide el espacio en celdas convexas. “Cada nodo interno del árbol divide la región convexa asociada con el nodo en dos regiones, utilizando para ello un plano con orientación y posición arbitrarios. Es análogo a un k-d tree variable sin la restricción de que los planos estén orientados ortogonalmente con los ejes.” [**18**].

3. DISEÑO DE LA APLICACIÓN SOFTWARE 3D

Este capítulo describe inicialmente los alcances y limitaciones de la aplicación. Luego se hace un análisis de los requerimientos del software para obtener una descripción del proceso de funcionamiento de la aplicación.

3.1. ALCANCES Y LIMITACIONES DE LA APLICACIÓN

Para el desarrollo de la aplicación software se establecen tres herramientas a utilizar: Las librerías VTK, librerías VCollide y librerías QT. La razón principal es que son herramientas de código y distribución libre que no requieren licencia lo cual va acorde al objetivo de crear una herramienta de software libre.

3.1.1. Alcances

La aplicación está en la capacidad de importar objetos 3d rígidos en los formatos vtk, vtp, obj y ply.

Permite al usuario ejercer una acción directa de presión o estiramiento al objeto de tal manera que éste se deforme y recupere su forma original.

El software cuenta con una interfaz gráfica de usuario amigable permitiendo que la aplicación sea de fácil uso.

3.1.2. Limitaciones

La deformación se realiza ejerciendo una acción de presionar o estirar sobre un punto de la superficie del objeto.

La aplicación permitirá deformar un objeto 3d a la vez.

Al objeto deformable se le admite una deformación a la vez.

3.2. REQUERIMIENTOS DE LA APLICACIÓN

El desarrollo de este proyecto se basa en el proceso de la ingeniería de requerimientos. En la figura 3.1 se presenta las fases del proceso de IR: Obtención y Análisis, Especificación y Validación [**21**].

3.2.1. Obtención y análisis

Definición de los alcances y limitaciones de la aplicación, ya definido en la sección 3.1.

3.2.2. Especificación

En esta etapa se definen y clasifican los requerimientos del sistema.

3.2.2.1. Requerimientos funcionales

En la Tabla 3.1 se relacionan los requerimientos funcionales que la aplicación debe tener. En la primera columna se indica el código asociado a cada requerimiento, en la segunda

columna se especifica el tipo de requerimiento y en la tercera columna se describe el requerimiento.

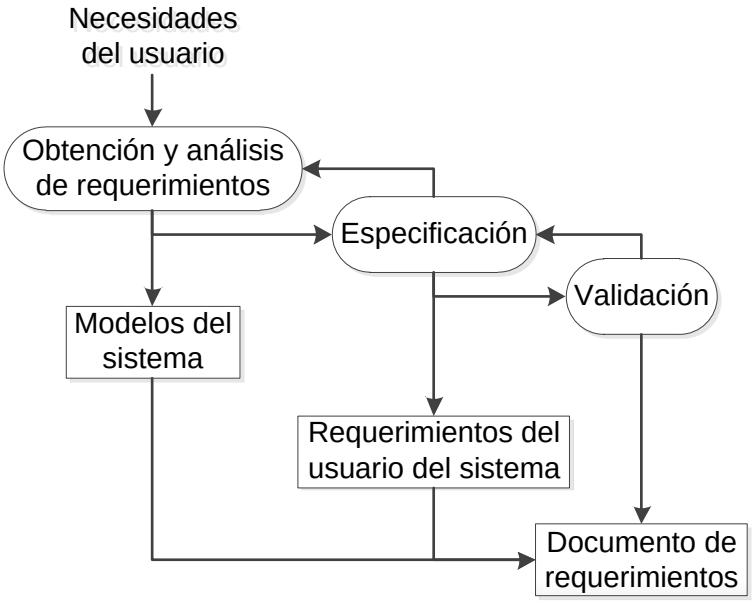


Figura 3.1: Proceso de Ingeniería de Requerimientos

Fuente: Modificado de [**22**]

3.2.2.2. Requerimientos no funcionales

En la Tabla 3.2 se relaciona la lista de requerimientos no funcionales generales que debe tener en cuenta la aplicación. En la primera columna se indica el código asociado a cada requerimiento, en la segunda columna se especifica el tipo de requerimiento y en la tercera columna se describe el requerimiento.

Número	Nombre	Requerimientos funcionales
RF-01	Importar objeto 3D	La aplicación debe permitirle al usuario final importar el objeto que desea deformar
RF-02	Deformar objeto	Se debe permitir al usuario deformar el objeto importado, haciendo uso de un cursor 3d
RF-03	Restaurar objeto	La aplicación debe estar en la capacidad de restaurar el objeto a su forma original
RF-04	Generar gráfica	El sistema debe permitirle al usuario generar una gráfica de tiempo de respuesta
RF-05	Modificar parámetros del sistema	El usuario debe poder modificar los valores de la masa, el resorte y el amortiguador para la etapa de recuperación de forma

Tabla 3.1: Requerimientos funcionales

Número	Tipo	Requerimiento
RNF-01	Escalabilidad	El sistema debe permitir en un futuro el desarrollo de nuevas funcionalidades, además de modificar o eliminar las existentes

RNF-02	Facilidad de uso	Las interfaces de la aplicación deben ser atractivas y amigables para el usuario final, de tal forma que los nuevos usuarios desarrollen una interacción efectiva con el sistema. Además la aplicación debe presentar mensajes en ventanas emergentes que permitan al usuario visualizar algún tipo de procesamiento e información
RNF-03	Mantenibilidad	Permitir modificar la aplicación desarrollada para poder corregir fallos del sistema, mejorar su funcionamiento u otros atributos o adaptarse a futuros cambios en el entorno
RNF-04	Implementación	Las herramientas utilizadas para el desarrollo de la aplicación deben ser de software libre

Tabla 3.2: Requerimientos no funcionales

3.2.3. Casos de uso

Teniendo conocimiento de los requerimientos funcionales se hace una descripción del proceso de funcionamiento de la aplicación.

Acción del actor		Respuesta del sistema	
1.	El caso de uso comienza cuando el usuario corre la aplicación.		
2.	Importar un objeto.	3.	Procesa los datos del objeto como filtrado a polígonos triangulares y eliminación de vértices y vectores redundantes.
4.	Acciona la conversión del objeto importado.	5.	Reorganiza vértices y transforma el objeto rígido a objeto modificable.
6.	Mueve el cursor hasta hacer contacto con el objeto.	7.	Posiciona los vértices de acuerdo al modelo deformable para luego renderizar el comportamiento de deformación.
8.	Modificar parámetros de la etapa de restauración de forma del objeto.	9.	Posicionar los vértices de acuerdo a la dinámica del sistema.
10.	Seleccionar intervalo de tiempo para graficar el comportamiento de restauración.	11.	Despliega una gráfica que permite observar el tiempo de respuesta de restauración de forma del objeto.

Tabla 3.3: Descripción caso de uso

La Figura 3.2 presenta el diagrama de casos de uso general de la aplicación.

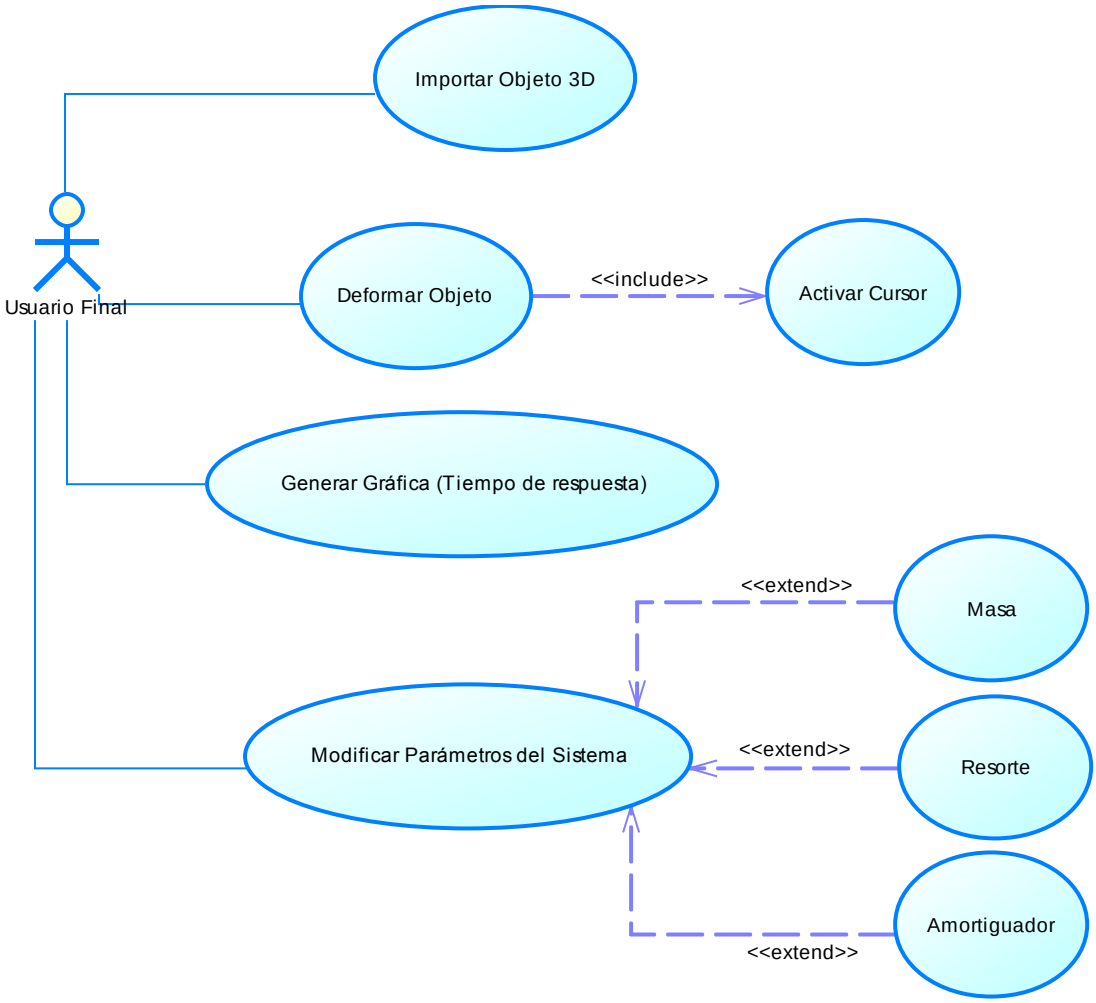


Figura 3.2: Diagrama casos de uso general del sistema

4. IMPLEMENTACIÓN DE LA APLICACIÓN SOFTWARE 3D

Este capítulo describe el diseño de la aplicación desarrollada de forma abstracta, teniendo en cuenta los requerimientos funcionales que se plasmaron con anterioridad. Además se hará una breve explicación de las clases utilizadas en el desarrollo del programa, para concluir con diagramas UML que permitan al lector entender con mayor facilidad la estructura de la aplicación presentada.

4.1. CLASES DE LA APLICACIÓN

En la aplicación software 3d se manejan dos paquetes, un paquete para el manejo de las clases base que provee VTK, VCollide y QT, y el otro para el manejo de las clases propias de la aplicación (Figura 4.1).

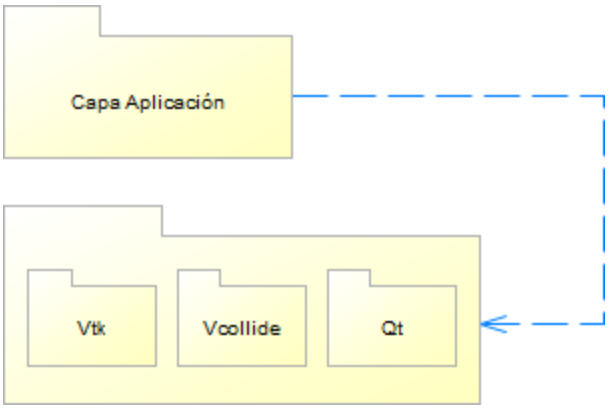


Figura 4.1: Diagrama de paquetes

The Visualization ToolKit (VTK) [**23**] son librerías de código libre desarrolladas por Kitware para el desarrollo de gráficas en 3D, procesamiento de imágenes y visualización. El diseño de esta librería está fuertemente basado en el paradigma orientado a objetos y su modelo gráfico está en un nivel más alto de abstracción que el de otras librerías de renderización como OpenGL o PEX, por lo cual es más fácil implementar aplicaciones de visualización 3d [**10**].

VCollide [**24**] son librerías de código abierto implementadas en C++ por el grupo GAMMA (*Geometric Algorithms for Modeling, Motion, and Animation*) de la Universidad de Carolina del Norte. Estas librerías trabajan en escenarios que contengan gran cantidad de objetos geométricos además de tener la facilidad de funcionar con mallas triangulares [**9**].

Qt [**25**] es una herramienta para el desarrollo de interfaces gráficas de usuario en aplicaciones que pueden funcionar para diferentes plataformas. Su entorno de trabajo está basado en C++ lo cual permite construir interfaces de manera rápida y sencilla.

4.1.1. Clases base de la aplicación

La Figura 4.2 muestra el conjunto de clases base utilizadas en la aplicación software 3d.

Figura 4.2: Clases base

A continuación se hará una breve descripción de las clases de VTK utilizadas en el desarrollo de la aplicación.

vtkProgrammableFilter: Es un filtro que puede ser programado por el usuario. Para utilizar el filtro se define una función que recupera la entrada del tipo correcto, crea los datos, y luego manipula la salida del filtro. El uso de este filtro evita la necesidad de creación de subclases.

vtkPolyDataNormals: “Calcula la normal del punto para una malla poligonal, este filtro puede reordenar los polígonos para asegurar la consistencia de orientación a través de los polígonos vecinos. Quita los bordes afilados y duplica puntos con normales separadas para obtener una superficie mejor definida” [**26**].

vtkPolyData: “Representa un conjunto de datos como líneas, polígonos y triángulos” [**27**].

vtkPLYReader: Lee los datos poligonales en formato ply.

vtkLineSource: Es un objeto que crea una línea definida por dos puntos finales. El número de segmentos que componen la línea es controlado por el ajuste de la resolución del objeto.

vtkModifiedBSPTree: Localizador basado en búsquedas.

vtkPolyDataMapper: “Genera primitivas gráficas a partir de datos poligonales” [**10**].

vtkDoubleArray: Es un arreglo de valores de tipo *double*. Proporciona métodos para la inserción y recuperación de valores y automáticamente se redimensionará para guardar los nuevos datos.

vtkIdList: Lista de identificadores de puntos o celdas.

vtkRenderer: “Coordina la renderización de luces, cámara y actores” [**26**].

vtkPolyDataReader: Lee datos poligonales de archivos vtk.

vtkOBJReader: Sirve para leer archivos obj.

vtkCell: Es una clase abstracta que especifica las interfaces para las celdas de datos. Las celdas de datos son simples elementos topológicos como puntos, líneas, polígonos y tetraedros de los cuales se componen los conjuntos de datos de visualización. En algunos casos, estos conjuntos de datos pueden representar de forma explícita celdas, y en otros casos, los conjuntos de datos están, implícitamente, compuesto de celdas.

vtkRenderWindowInteractor: “Provee una plataforma independiente de mecanismos de interacción para eventos del mouse, teclado y tiempo” [**26**].

vtkXMLPolyDataReader: Lee el formato de archivo XML VTK PolyData. Un archivo de datos poligonales puede ser leído para producir una salida. La extensión estándar del archivo de este lector es el formato “vtp”. Este lector también se utiliza para leer una sola pieza del formato de archivo paralelo.

vtkMatrix4x4: “Representa y manipula las matrices 4x4 de transformación. En concreto trabaja con matrices que se encuentran en 3D usando coordenadas homogéneas” [**27**].

vtkTriangleFilter: “Genera triángulos sobre un objeto 3D, este filtro también pasa por cada vértice y línea de ser necesario” [**27**].

vtkCleanPolyData: Es un filtro que toma los datos poligonales como entrada y genera datos poligonales como salida. Además puede combinar puntos duplicados, eliminar los puntos que no se utilizan, y si está habilitado, transformar celdas degeneradas en formas apropiadas (por ejemplo, un triángulo se convierte en una línea si dos puntos del triángulo se mezclan).

vtkActor: “Representa un objeto renderizado en la escena. Sus propiedades y posición están dados en coordenadas del mundo real” [**11**].

vtkConeSource: “Fuente que genera geometría de datos” [**26**].

vtkCylinderSource: “Fuente que genera geometría de datos” [**26**].

vtkAppendFilter: Añade uno o más conjuntos de datos dentro de una red estructurada única.

vtkCoordinate: Realiza transformación de coordenadas y representa la posición en una variedad de sistemas de coordenadas vtk.

vtkDataSetMapper: Asignación de conjunto de datos para primitivas gráficas.

vtkFollower: “Instancias que siempre siguen a la cámara activa. Esto es útil cuando se diseñan textos que deben ser legibles desde cualquier posición de la cámara en la escena” [**11**].

vtkPlane: Proporciona métodos para cálculos de distintos planos. Estos incluyen proyección de puntos sobre un plano, evaluación de la ecuación del plano, y devolución de un plano normal.

vtkBoundedPlanePointPlacer: Sistema de ubicación que limita alguna manipulación sobre un plano finito.

vtkGeometryFilter: Es un filtro de propósito general para extraer la geometría (y los datos asociados) de cualquier tipo de conjunto de datos.

vtkTable: Estructura de datos básica para el almacenamiento de columnas de datos.

vtkFloatArray: Es un arreglo de valores de tipo float. Proporciona métodos para la inserción y recuperación de valores y automáticamente se redimensiona para guardar los datos nuevos.

vtkChartXY: Esta clase nos provee los servicios necesarios para la elaboración de gráficos en 2D (XY).

vtkContextView: Muestra una escena 2D en una vantana.

vtkPlot: Clase abstracta para el manejo de gráficas 2D. Esta es la clase base para todos los tipos de gráficos utilizados en vtkChart.

vtkPoints: Representa y manipula puntos 3D.

4.1.2. Clases propias de la aplicación

Las clases propias de la aplicación se dividen en dos tipos: clases del sistema de deformación y clases manejo de interfaz.

Clases del sistema de deformación

Estas clases están encargadas del funcionamiento del conjunto total del sistema de deformación.

A continuación se describen brevemente las clases implementadas para la aplicación.

Cursor3D: Configurar el tamaño del cursor de acuerdo a las dimensiones del objeto importado además de ponerlo en escena.

DeteccionColisiones: Incorpora los dos objetos a colisionar y reporta el lugar exacto donde se produjo el contacto en el objeto deformable.

Eventos: Captura lo que pasa en escena además de las acciones realizadas con el mouse y el teclado.

Graficacion: Permite graficar el tiempo de respuesta de la restauración del objeto deformable.

MotorDeformacion: Esta clase se encarga de proveer la información necesaria para poder realizar la deformación y restauración del objeto deformable.

PropagacionCeldas: Almacena la organización de los triángulos del objeto de acuerdo a una configuración de propagación.

PropagacionMovimiento: Configura la propagación del movimiento teniendo en cuenta la propagación de las celdas y de los vértices.

PropagacionVertices: Almacena la organización de los vértices del objeto de acuerdo a una configuración de propagación.

Rayo: Configura el señalador del cursor y lo pone en escena.

ReformadorSuperficial: Hace posible la modificación de la posición de los vértices de un objeto 3d.

Figura 4.3: Clases propias del sistema de deformación

Clases manejo de interfaz

Estas clases nos proporcionan los servicios necesarios para poder construir la interfaz con la que el usuario final interactuará.

Figura 4.4: Clases propias manejo de interfaz

A continuación se describen las clases implementadas para el manejo de interfaz.

AcercaDe: Esta clase provee los servicios necesarios para el despliegue de una ventana con la información del proyecto realizado.

Ayuda: Proporciona lo necesario para la exhibición de una ventana con información del manejo de la aplicación.

BarraProgreso: Muestra al usuario final una barra de progreso que indica que hay un procesamiento de datos.

GUI: Es la encargada de suministrar lo necesario para la ejecución de la ventana principal.

Importando: Muestra al usuario final un indicador de que un objeto está en proceso de importación.

Main: Clase principal para la ejecución de la interfaz gráfica de usuario.

Parametros: Esta clase proporciona los servicios necesarios para la creación de la ventana que permite la modificación de los parámetros del sistema masa resorte amortiguador.

TiempoGrafica: Ventana que permite definir el intervalo de tiempo para graficar.

4.2. ESTRUCTURA GENERAL DE LA APLICACIÓN SOFTWARE 3D

La figura 4.5 muestra la estructura de la aplicación software con sus respectivos componentes. Este diagrama representa la composición de la aplicación software conformada por: Acondicionamiento, Cursor 3d, Colisiones, Deformación, Interacción y Representación visual.

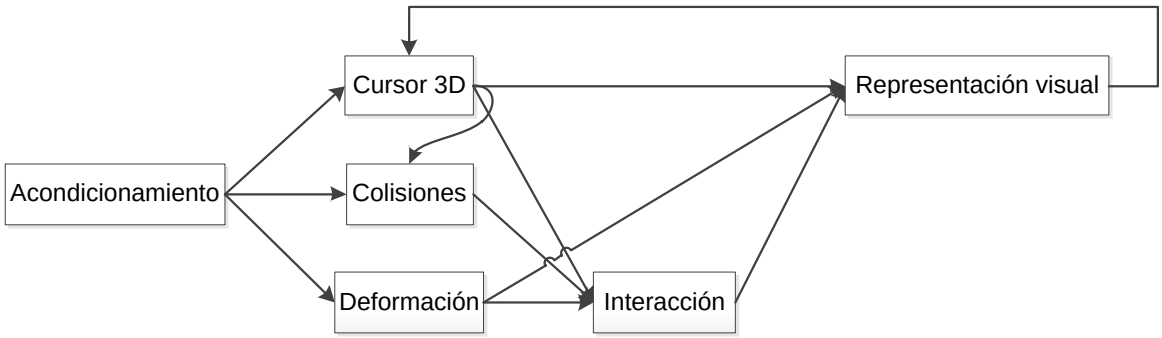


Figura 4.5: Componentes de la aplicación software

A continuación se describen cada uno de los componentes de la aplicación.

Acondicionamiento: Encargado de importar y preparar el objeto.

Cursor 3d: Herramienta que permite al usuario deformar el objeto 3d.

Colisiones: Es el encargado de verificar si el cursor 3d está en contacto con el objeto a deformar, así como el instante y punto exacto donde se dio dicho contacto.

Deformación: Es el encargado de calcular la deformación del objeto 3d ante una acción generada por el usuario desde el mouse.

Interacción: Permite al usuario interactuar con la escena.

Representación visual: Despliega la escena virtual en la interfaz gráfica de usuario.

4.2.1. Descripción general de la aplicación

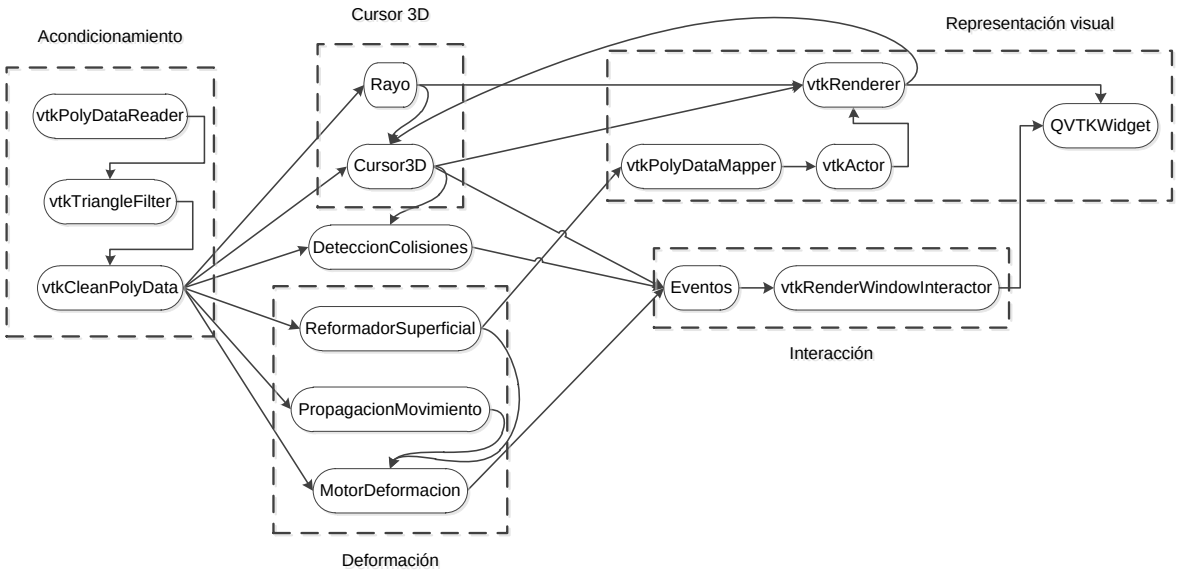


Figura 4.6: Estructura general de la aplicación

La Figura 4.6 ilustra el procedimiento general para el funcionamiento de la aplicación. La clase **vtkPolyDataReader** es la encargada de importar los objetos 3d, en este caso el formato vtk. El objeto que se importe es procesado por el filtro **vtkTriangleFilter** con el fin de garantizar un mallado triangular ya que el sistema de colisiones solo admite polígonos triangulares. La clase **vtkCleanPolyData** elimina vértices redundantes que pueda tener el objeto, este proceso es importante para no tener inconvenientes con el reporte de colisiones, el reformador superficial y propagación de movimiento. La clase **Rayo** interseca una línea sobre la superficie del objeto y genera un actor para ser renderizado en la escena. **Cursor3D** configura el tamaño de la representación del cursor de acuerdo a las dimensiones del objeto importado, orienta el cursor en la escena de acuerdo a los datos de orientación de la cámara proporcionado por **vtkRenderer**, configura la orientación de la línea proporcionado por **Rayo** y genera un actor para ser renderizado en la escena. **DeteccionColisiones** extrae la información de los triángulos tanto del objeto 3d como del cursor para evaluar colisiones entre estos objetos, con estos conjuntos de triángulos añadidos al sistema de colisiones se genera el reporte de colisiones. **ReformadorSuperficial** convierte el objeto rígido a objeto modificable, es decir que se puede modificar la posición de todos sus vértices. **PropagacionMovimiento** organiza y almacena las id de los vértices. **MotorDeformacion** modifica los vértices del objeto modificable de acuerdo al modelo deformable y qué vértices son afectados de acuerdo a la propagación de movimiento, además de regresar esos vértices a su posición original en

el proceso de restauración. **Eventos** ejecuta los métodos de las clases involucradas, en **Cursor3D** la clase **Eventos** captura la posición en pantalla del cursor convencional y estos datos son transmitidos al **Cursor3D** para posicionar el cursor en la escena. En **DeteccionColisiones** actualiza la matriz de transformación del cursor y captura los reportes generados, en **MotorDeformacion** ejecuta los métodos de deformar y restaurar forma. **vtkRenderWindowInteractor** permite interactuar con la escena, principalmente hace posible el cambio de vistas rotando la cámara, **vtkPolyDataMapper** genera las primitivas gráficas a partir de los datos de la clase **ReformadorSuperficial**, **vtkActor** hace la representación visual del objeto modificable, **vtkRenderer** renderiza los actores de **Rayo**, **Cursor3D** y el actor del objeto modificable, y **QVTKWidget** permite la visualización del escenario en una ventana creada en Qt.

4.3. COMPONENTE ACONDICIONAMIENTO

Este componente se encarga de importar y adecuar el objeto importado con el fin de evitar inconvenientes en los componentes colisiones y deformación.

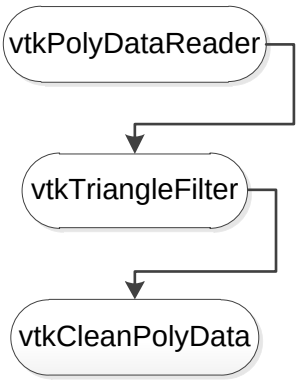


Figura 4.7: Estructura del componente Acondicionamiento

A continuación se describe el procedimiento en código de la Figura 4.7.

```
vtk->SetFileName("Archivo.vtk");
Filtro->SetInputConnection( vtk->GetOutputPort() );
Limpieza->SetInputConnection( Filtro->GetOutputPort() );
```

Siendo vtk una instancia de la clase vtkPolyDataReader, Filtro una instancia de la clase vtkTriangleFilter y Limpieza una instancia de la clase vtkCleanPolyData.

4.4. COMPONENTE CURSOR 3D

En esta aplicación se requiere de un objeto rígido el cual se usará como una herramienta para que el usuario pueda presionar o estirar un punto de la superficie del objeto deformable. De esta sección en adelante llamaremos a esta herramienta objeto cursor 3d que es un elemento fundamental de la aplicación.

Este componente permite que el objeto cursor 3d se posicione y oriente en el escenario virtual de manera que su uso sea amigable para el usuario.

Respecto al uso de una herramienta en un escenario virtual se desea que dicha herramienta se mantenga visible en pantalla si se hace un cambio de posición de la vista en el escenario, además de mantener la orientación de movimiento sin importar en que parte del escenario se encuentre. Para explicar lo anterior se recurre a la Figura 4.8 donde se ilustra el cambio de vistas alrededor de un punto. La cámara es lo que se ve en pantalla, el punto focal es el punto de observación de la cámara además de ser el punto de rotación de la misma, acimut es la rotación sobre el eje vertical y la elevación es la rotación sobre el eje horizontal.

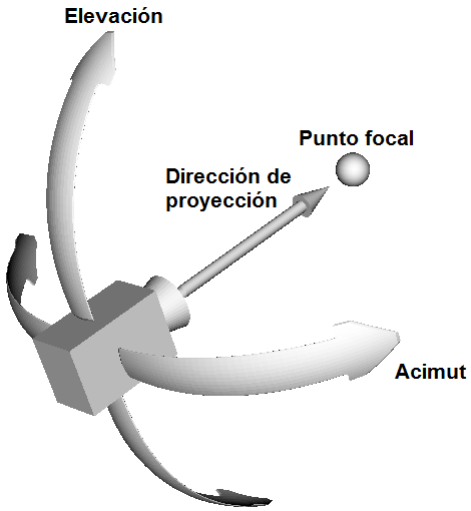


Figura 4.8: Rotación cámara

Fuente: Modificado de [**28**]

Se asume una vista que se encuentra en una posición cualquiera en el escenario y se tiene como punto focal el centro de un objeto 3d, inicialmente se considera una herramienta ubicada entre el objeto y la cámara, es decir que se está viendo en pantalla la herramienta en frente del objeto. Si se hace una elevación de cámara de 180° ahora el objeto está en frente de la herramienta. Ahora surge otro problema y es que si inicialmente se movía esta herramienta hacia arriba moviendo el mouse en el mismo sentido, la nueva posición implica que para hacer un movimiento hacia arriba de la herramienta desde la perspectiva del usuario se tendría que mover el mouse hacia abajo y esto es un problema ya que el usuario pierde la orientación de cómo manejar este instrumento. Para lograr el desplazamiento sin perder la orientación se debe mover el objeto cursor 3d sobre un plano en el espacio ubicado entre el objeto y la cámara además de que este plano siempre esté paralelo a la pantalla sin importar la rotación de la cámara, en las secciones siguientes se explica más detalladamente este procedimiento.

Construcción del cursor 3d

Como se van a importar objetos de diferentes tamaños, es importante definir las dimensiones del cursor. La creación visual del cursor dependerá entonces de las dimensiones del objeto para mantener la proporción entre el objeto 3d y el objeto cursor 3d de manera que no importa el tamaño del objeto que se importe al entorno, siempre se verá un objeto cursor 3d del mismo tamaño en la escena.

La Figura 4.9 muestra el esquema de la clase Cursor3D que genera el actor del objeto cursor 3d a partir del tamaño del objeto. Actor es la representación visual de un objeto en un escenario virtual.

Figura 4.9: Clase Cursor3D

A continuación se representa en código el esquema de la Figura 4.9.

```
cursor->SetEntrada( Limpieza->GetOutput() );
cursor->GetActor();
```

Siendo cursor una instancia de la clase Cursor3D.

La construcción del objeto cursor 3d se hace con dos formas básicas: un cilindro y un cono, para dar la apariencia de un lápiz.

La Figura 4.10 ilustra el procedimiento para generar el actor del objeto cursor 3d. Objeto entrega los datos que determinan el tamaño del objeto 3d, vtkCylinderSource y vtkConeSource generan los datos geométricos de cilindro y cono respectivamente definiendo las dimensiones de cada uno de acuerdo al tamaño del objeto, vtkAppendFilter unifica los datos de cilindro y cono para generar un solo conjunto de datos geométricos, vtkDataSetMapper mapea estos datos geométricos para convertirlos en primitivas, vtkActor se encarga de la representación visual del objeto cursor 3d para añadirlo a la escena.

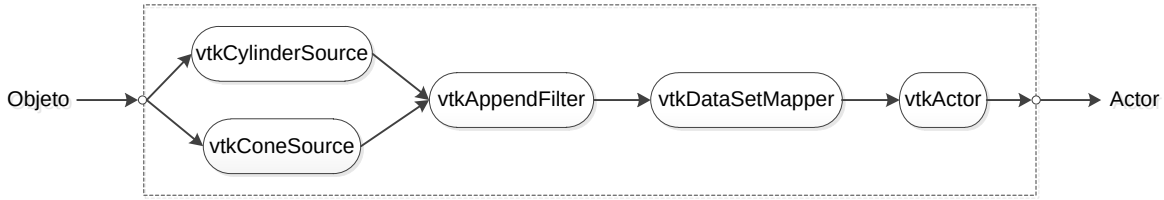


Figura 4.10: Procedimiento para generar el actor del cursor 3d

Desplazamiento del cursor 3d sobre el escenario

Hasta ahora solo se tiene la representación del objeto cursor 3d para ser visualizado en la escena, por consiguiente se debe hacer que esta representación se mueva sobre el espacio. El desplazar el objeto cursor 3d siguiendo los movimientos del mouse y que este movimiento se dé entre el objeto 3d y la cámara es un proceso difícil de ejecutar cuando constantemente se cambia la vista del objeto al rotar la cámara sobre los ejes vertical y horizontal. Con el fin de lograr el desplazamiento mencionado se debe mover el objeto cursor 3d sobre un plano en el espacio ubicado entre el objeto y la cámara además de que este plano siempre esté paralelo a la pantalla sin importar la rotación de la cámara, la Figura 4.11 ilustra lo mencionado anteriormente.

Un sistema de cursor convencional en cualquier aplicación informática es medido en coordenadas de pantalla, es decir un plano que representa el área de visualización de un monitor con valores dados en píxeles. Para poder desplazar el objeto cursor 3d se deben

capturar dichos valores y transmitirlos al actor del objeto cursor 3d, pero un escenario 3d es gobernado por coordenadas cartesianas espaciales por lo tanto se debe transformar la posición en pantalla del cursor dado en píxeles a coordenadas tridimensionales. De ahí que para obtener las coordenadas en el espacio se debe proyectar un punto de la pantalla sobre el plano ubicado entre el objeto y la cámara.

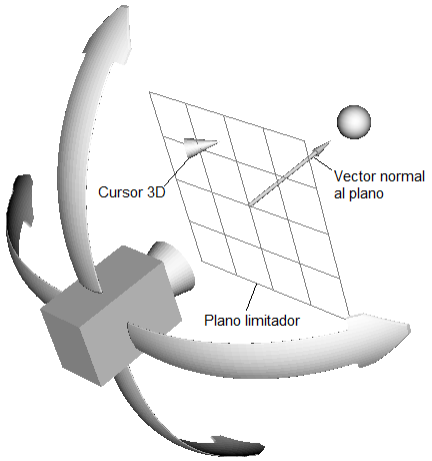


Figura 4.11: Desplazamiento del cursor 3d sobre un plano

Fuente: Modificado de [**28**]

La Figura 4.12 muestra los elementos de entrada necesarios para el desplazamiento del objeto cursor 3d sobre el escenario virtual a partir de las coordenadas de pantalla y un sistema de renderización que controla la cámara.

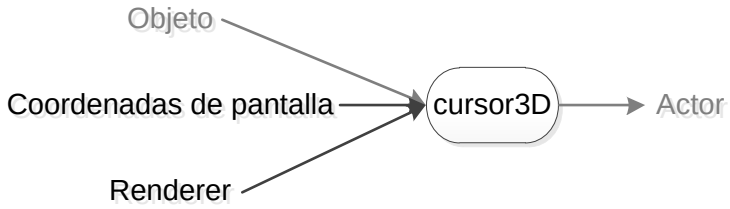


Figura 4.12: Elementos de entrada para desplazamiento del cursor 3d

A continuación se representa en código el esquema de la Figura 4.12.

```
cursor->SetPosicion(Coordenadas_de_pantalla);
cursor->SetRenderer(renderer);
```

Siendo renderer una instancia de la clase vtkRenderer.

La Figura 4.13 indica el procedimiento para mover el actor del objeto cursor 3d sobre la escena virtual. Renderer proporciona datos de la dirección de la cámara, vtkPlane genera el plano al cual se le va a proyectar un punto, la dirección del plano se establece a partir de la dirección de la cámara de manera que este plano se mantendrá paralelamente al plano de la pantalla, vtkBoundedPlanePointPlacer proyecta un punto en pantalla sobre el plano y entrega un valor en coordenadas cartesianas espaciales, vtkMatrix4x4 es una

matriz de transformación que posiciona el actor a partir del valor entregado por vtkBoundedPlanePointPlacer.

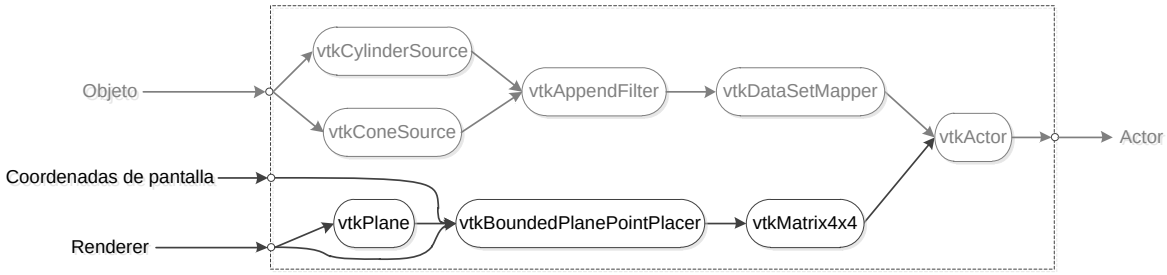


Figura 4.13: Procedimiento para desplazar el cursor 3d

La figura 4.14 muestra el esquema detallado de la conversión de coordenadas de pantalla a coordenadas cartesianas. El elemento Plano es una instancia de la clase vtkPlane que evalúa la ecuación de un plano en el espacio, en este plano se hará una proyección que permita identificar la posición cartesiana de un punto sobre la superficie de este plano, el elemento "Coordenadas de pantalla" son las coordenadas (*x*, *y*) dadas en píxeles proporcionadas por la posición del cursor al recorrer la pantalla y el elemento Renderer calcula la transformación entre coordenadas. La clase vtkBoundedPlanePointPlacer proyecta la posición del cursor en pantalla sobre el plano y de esta manera se encuentra las coordenadas cartesianas en el espacio.

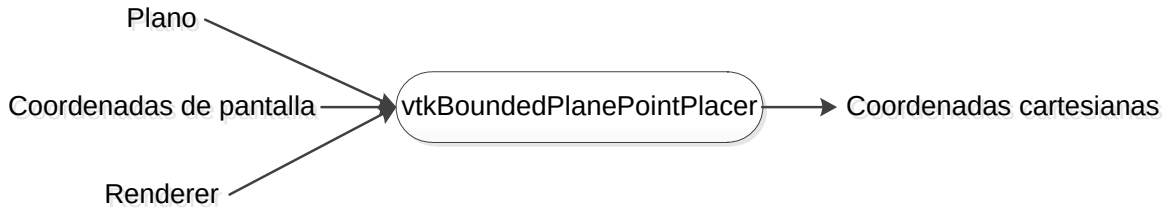


Figura 4.14: Conversión de coordenadas de pantalla a coordenadas cartesianas

Orientación del cursor 3d

Hasta esta etapa se puede mover el cursor pero sin tener en cuenta la orientación del mismo, en consecuencia la dirección del objeto cursor 3d, desde la perspectiva del usuario, cambiará a medida que se rote la cámara. Lo ideal es que sin importar en que parte del escenario se encuentre la cámara, el objeto cursor 3d siempre esté apuntando hacia el frente.

La figura 4.15 muestra el procedimiento para mantener la dirección del objeto cursor 3d siempre hacia el frente. vtkFollower se encarga de mantener la orientación del cursor cuando se mueve la cámara en cualquier dirección, vtkFollower entrega datos de orientación a la matriz del actor para establecer la dirección del objeto cursor 3d.

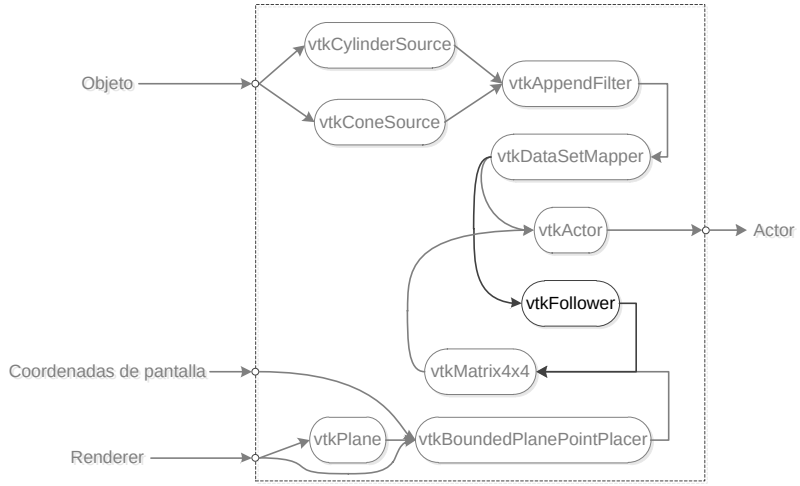


Figura 4.15: Procedimiento para orientar el cursor 3d

Señalador del cursor 3d

Otro aspecto fundamental a tener en cuenta en la aplicación es tocar el punto exacto de la superficie del objeto deformable. Como en muchos escenarios virtuales, la percepción de profundidad no es completa y esto causa que el usuario no domine plenamente el manejo del cursor al ejecutar un movimiento en profundidad, es decir que el usuario no hace contacto en el punto deseado. Para solucionar este inconveniente se hace necesario implementar una señal guía que facilite este trabajo.

La Figura 4.16 ilustra el elemento de entrada incorporado al componente Cursor3D que permite al objeto cursor 3d tener la funcionalidad de señalar el punto de la superficie del objeto deformable que se quiere contactar, es como si apuntáramos con un láser la superficie de dicho objeto.

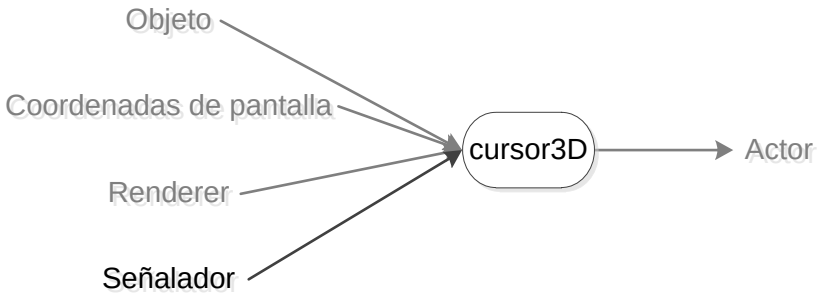


Figura 4.16: Elemento señalador

A continuación se representa en código la inclusión del señalador al objeto cursor 3d.

```
cursor->SetSenalador(senalador);
```

Siendo senalador una instancia de la clase Rayo.

La Figura 4.17 muestra la clase Rayo y los elementos necesarios para poder generar un señalador al objeto cursor 3d. Objeto es el elemento objetivo al cual se le va a proyectar un punto sobre su superficie, Pi y Pf son los puntos que permitirán crear una línea, la

señal guía que se mostrará en el escenario, siendo Pi punto inicial y Pf punto final respectivamente. Actor es la representación visual de la línea en el escenario.

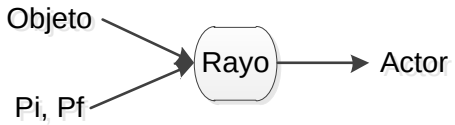


Figura 4.17: Clase Rayo

A continuación se representa en código el esquema de la Figura 4.17.

```
senalador->SetSuperficie( Limpieza->GetOutput() );
senalador->Proyeccion(Pi, Pf);
senalador->GetActor();
```

La Figura 4.18 explica el procedimiento para la generación del señalador incorporado al objeto cursor 3d. vtkModifiedBSPTree calcula el punto exacto donde la línea se interseca con la superficie del objeto 3d, los datos de la línea son entregados por el objeto cursor 3d, vtkLineSource creará una línea a partir de los datos Pi y Pf, Pi es proporcionado por el centro del objeto cursor 3d, Pf es proporcionado por vtkModifiedBSPTree, vtkProgrammableFilter permite que la línea generada por vtkLineSource sea modificada, es decir que varíe su longitud, vtkPolyDataMapper mapea los datos geométricos de la línea para convertirlos en una primitiva y vtkActor se hace cargo de representar visualmente esta línea en el escenario.

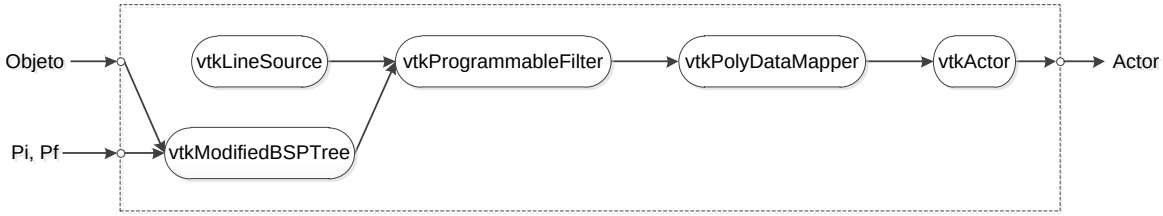


Figura 4.18: Procedimiento para generar una señal guía

La figura 4.19 muestra la visualización del objeto cursor 3d con el señalador sobre la escena virtual.

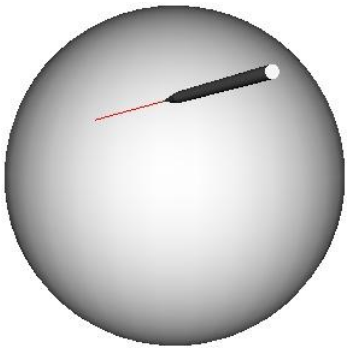


Figura 4.19: Cursor 3D con señalador

4.5. COMPONENTE COLISIONES

Para poder realizar deformación sobre el objeto 3d es necesario tener un contacto con el cursor 3d, es por eso que se requiere de un sistema que detecte cuándo estos objetos colisionan entre sí. Por tal motivo se implementó la clase DeteccionColisiones que permite un contacto directo entre el cursor y el objeto 3d.

La Figura 4.20, muestra los elementos necesarios para la clase DeteccionColisiones. Objeto y cursor son los objetos a colisionar, el elemento Matriz indica a la instancia de la clase la posición y orientación del cursor, Dato booleano permite saber si la colisión entre estos objetos se ha dado.

Figura 4.20: Clase del componente de colisiones

A continuación se representa en código el esquema de la Figura 4.20.

```
colisiones->SetObjeto( Limpieza->GetOutput() );
colisiones->SetObjeto( cursor->GetSalida() );
colisiones->ActualizarMatriz(cursor->GetMatriz()->Element);
colisiones->Contacto();
```

Algo que destacar en esta etapa del sistema de deformación es la importancia de no generar demasiada carga computacional en la detección de colisiones con el fin de no afectar el rendimiento de la aplicación.

Por la facilidad que ofrece para trabajar con mallados triangulares, por el tipo de reporte de colisión y con la finalidad de reducir el tiempo promedio de ejecución en comparación con otras librerías, se seleccionó VCollide para el análisis de detección de colisiones [**9**].

VCollide utiliza una arquitectura de detección de colisiones en tres etapas [**29**]:

- Un test N-body encuentra pares de objetos posiblemente colisionados.
- Un test jerárquico de OBBs (*Oriented Bounding Boxes*) encuentra posibles pares de triángulos colisionados.
- Un test exacto determina si un par de triángulos están solapados.

La figura 4.21 tiene como objetivo mostrar cómo la clase externa DeteccionColisiones adapta los datos de entrada para que la clase interna VCollide pueda recibirlos y procesarlos de manera adecuada.

Figura 4.21: Estructura interna de DeteccionColisiones

Ahora que se pueden detectar colisiones es fundamental saber en qué parte del objeto se produjo la colisión. La Figura 4.22 ilustra el esquema total de la clase DeteccionColisiones que permite identificar el instante y lugar preciso de contacto.

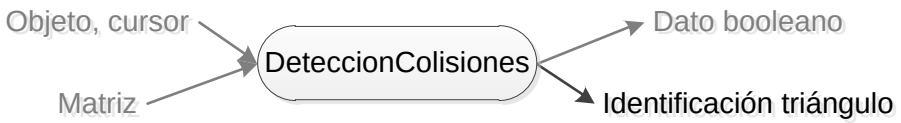


Figura 4.22: Clase DeteccionColisiones

A continuación se representa en código el reporte del triángulo que presenta la clase DeteccionColisiones.

```
colisiones->GetIdCeldaContacto();
```

La figura 4.23 ilustra que VCollide genera el reporte entregando el valor de identificación del triángulo que se contacta al existir una colisión. La clase externa DeteccionColisiones extrae ese valor permitiendo así saber el punto preciso de colisión.

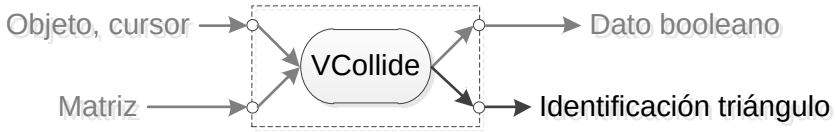


Figura 4.23: Reporte de colisión de VCollide

La clase VCollide se configuró para reportar un solo triangulo, de manera que este sistema entregará el valor del primer contacto que se registre entre el cursor y el objeto deformable.

4.6. COMPONENTE DEFORMACIÓN

Este componente se encarga de deformar el objeto cuando presionamos o estiramos un punto de la superficie del objeto con el objeto cursor 3D y restaura el objeto deformado a su forma original. En la Figura 4.24 se presenta la estructura de este componente.

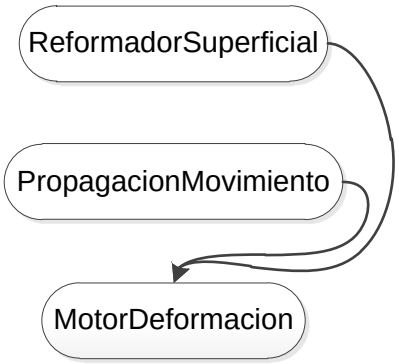


Figura 4.24: Estructura del motor de deformación

A continuación se representa en código el esquema de la Figura 4.24.

```
deformacion->SetReformador(reformador);
deformacion->SetPropagacionVerts(propagacion->GetPropagacionVertices());
```

Siendo deformacion una instancia de la clase MotorDeformacion, reformador es una instancia de la clase ReformadorSuperficial y propagacion es una instancia de la clase PropagacionMovimiento.

La estructura del componente Deformación se constituye de tres clases: ReformadorSuperficial, PropagacionMovimiento y MotorDeformacion. ReformadorSuperficial contiene los datos de las coordenadas de los vértices del objeto 3d por tanto la malla del objeto está formada por esta información, estos datos podrán ser modificados por la clase MotorDeformacion. PropagacionMovimiento entrega información de los vértices que serán afectados por la deformación en los diferentes niveles de propagación. MotorDeformacion define el cómo se deben modificar las coordenadas de los vértices de acuerdo al modelo deformable.

4.6.1. Clase ReformadorSuperficial

La clase ReformadorSuperficial hace posible la modificación de posición de todos los vértices del objeto 3d de manera que se pueda amoldar cualquier objeto en el escenario virtual.

La Figura 4.25 ilustra la clase ReformadorSuperficial que se encarga de convertir un objeto rígido a un objeto modificable.

Figura 4.25: Clase ReformadorSuperficial

A continuación se representa en código el esquema de la Figura 4.25.

```
reformador->SetEntrada( Limpieza->GetOutput() );
reformador->GetSalida();
```

La Figura 4.26 muestra la estructura interna de la clase ReformadorSuperficial, vtkProgrammableFilter permite modificar la posición de todos los vértices del objeto 3d de

tal forma que se pueda moldear su superficie, la clase vtkDoubleArray es un arreglo dinámico de tipo *double* donde se almacenan los datos de las coordenadas de los vértices del objeto 3d y este arreglo es utilizado por vtkProgrammableFilter para modificar los valores de los vértices, la clase vtkPolyDataNormals quita los bordes afilados de la superficie del objeto 3d recuperando su apariencia original, esto debido a que la clase vtkProgrammableFilter cambia la apariencia de la superficie del objeto eliminando efectos de sombreado y suavizado de bordes.

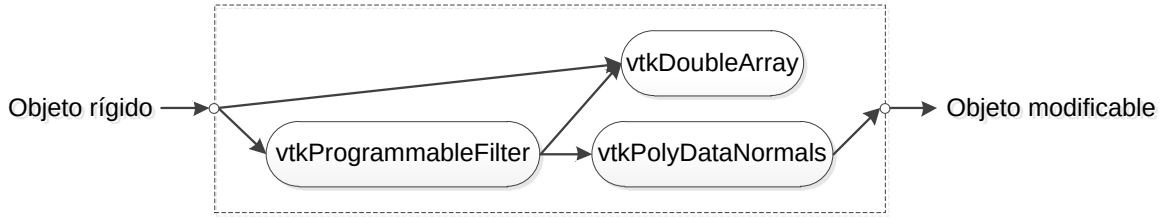


Figura 4.26: Estructura interna ReformadorSuperficial

4.6.2. Clase PropagacionMovimiento

Propagación de movimiento

Cuando se arroja una piedra al agua, las ondas se propagan por todas las direcciones a partir del punto de contacto, en un objeto elástico al ejercer una fuerza sobre un punto, esta fuerza tiende a propagarse alrededor de ese punto.

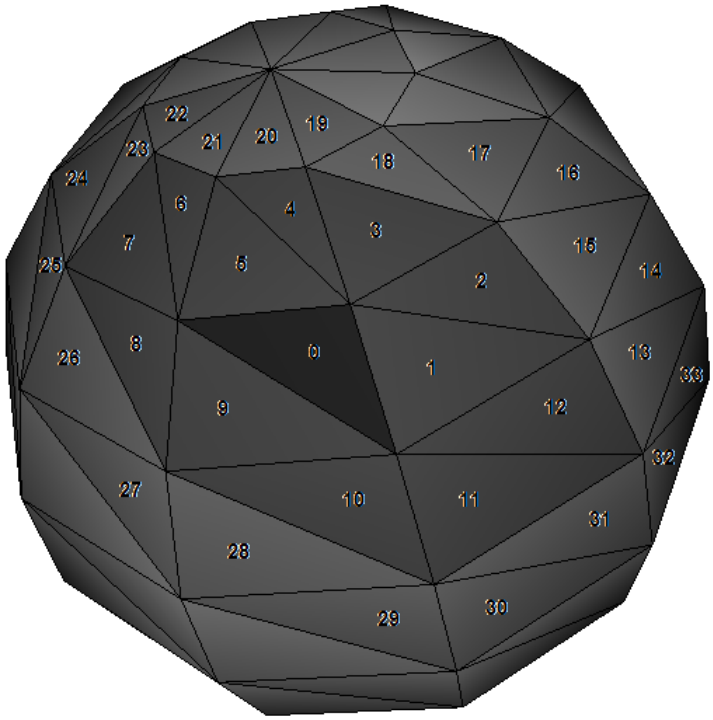


Figura 4.27: Propagación en una malla triangular

En el caso de un objeto 3d, el punto de contacto sería un triángulo, representar el comportamiento de propagación alrededor de este triángulo consiste en mover los demás

triángulos que lo rodean. La propagación en todas las direcciones sobre una estructura triangular no es uniforme debido a su geometría, pero esto mejora si el objeto tiene mayor resolución.

A continuación se explica cómo funciona la propagación en una estructura triangular. Suponer que se ejerce una acción de movimiento sobre el triángulo 0 (Figura 4.27), la propagación debe ejercerse sobre todas las direcciones por tanto los triángulos a mover en primera instancia son los triángulos que rodean al triangulo afectado, es decir que los primeros triángulos a mover son del 1 al 12, estos triángulos pertenecen al primer nivel de propagación, los triángulos del 13 al 33 pertenecen al segundo nivel de propagación, este comportamiento continua de esta manera de forma sucesiva hasta el nivel n. Todo esto se cumple para cada uno de los triángulos que conforman el objeto.

La dinámica de propagación del objeto con triángulos permite obtener la propagación por vértices que básicamente es el mismo sistema, es decir partiendo de los vértices del triángulo 0 se obtienen los vértices de propagación de primer nivel de los triángulos 1 al 12, los vértices de propagación del segundo nivel de los triángulos del 13 al 33 y así sucesivamente.

El propósito de los niveles de propagación por vértices es en definitiva asignar una relación de movimiento a los vértices según el nivel en que se encuentren.

Estructura de la clase

Si la deformación está basada en un sistema de resortes donde los vértices del objeto están interconectados por resortes, el movimiento de cada vértice afectará a sus vecinos generando así una propagación de movimiento. Para cambiar la forma de un objeto 3d se debe modificar la posición de sus vértices, pero esta modificación debe ejecutarse siguiendo un orden consecutivo de propagación. La Figura 4.28 muestra cómo la clase PropagacionMovimiento toma un objeto y organiza sus vértices de tal forma que exista una propagación.

Figura 4.28: Clase PropagacionMovimiento

A continuación se representa en código el esquema de la Figura 4.28.

```
propagacion->SetEntrada( Limpieza->GetOutput() );
propagacion->GetPropagacionVertices();
```

En [**2**] se hace uso de un algoritmo que recorre toda la malla iniciando desde el vértice afectado por una acción aplicada, afectando consecutivamente a cada uno de sus vértices vecinos, en este trabajo se propone implementar un algoritmo que identifique los vértices adyacentes a cada triángulo del objeto y almacenarlos en memoria.

La organización de estos vértices se almacena en un arreglo dinámico de tipo entero con el fin de que el sistema de deformación acceda a estos datos y pueda ejecutar adecuadamente la dinámica de deformación.

La Figura 4.29 ilustra el procedimiento para generar el arreglo de vértices de la clase PropagacionMovimiento. El objeto 3d proporciona los datos de celdas y vértices de tal forma que la clase PropagacionCeldas explora toda la superficie del objeto encontrando los triángulos vecinos de cada triángulo y organizándolos de la forma explicada en la sección propagación de movimiento, esta organización es almacenada en un arreglo dinámico de tipo entero. La Tabla 4.1 muestra cómo los triángulos son organizados de acuerdo al nivel de propagación en que se encuentren. La clase PropagacionVertices hace uso de este arreglo de celdas para reflejar la misma organización con vértices. La razón del uso de los datos del objeto en la clase PropagacionVertices es para extraer los vértices de acuerdo a dicha organización y almacenarlos en el arreglo dinámico.

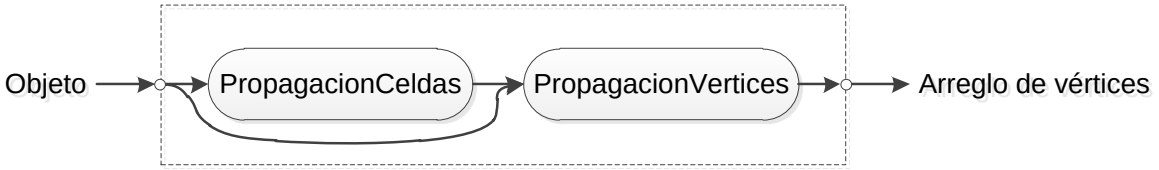


Figura 4.29: Procedimiento para generar arreglo de vértices

Nivel	Celdas																				
1	1	2	3	4	5	6	7	8	9	10	11	12	-	-	-	-	-	-	-	-	-
2	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…
n	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Tabla 4.1: Arreglo de celdas

La Figura 4.30 muestra la clase vtkIdList que permite almacenar la organización de las celdas, esto mismo se aplica para la organización de los vértices.

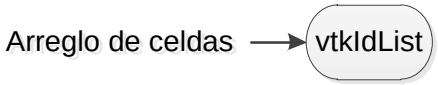


Figura 4.30: Almacenamiento del arreglo de celdas

4.6.3. Clase MotorDeformacion

La clase MotorDeformacion se conforma principalmente de dos métodos (Figura 4.31): el método Deformar se encarga de que el objeto se deforme de acuerdo al modelo deformable y el método Restaurar hace que el objeto recupere su forma original.

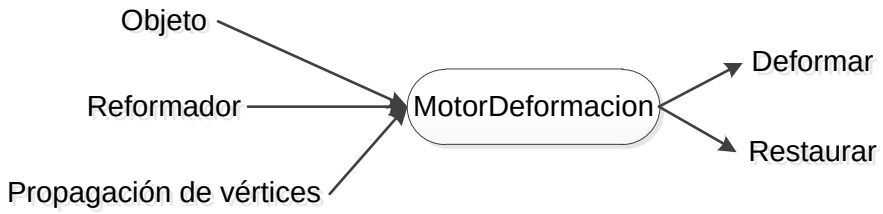


Figura 4.31: Clase MotorDeformacion

A continuación se representa en código el esquema de la Figura 4.31.

```
deformacion->SetEntrada( Limpieza->GetOutput() );
deformacion->SetReformador(reformador);
deformacion->SetPropagacionVerts(propagacion->GetPropagacionVertices());
deformacion->Deformar();
deformacion->RestaurarForma();
```

Modelo deformable

En este modelo se emplearán resortes ideales, es decir que se despreciarán sus características internas de masa y amortiguamiento.

En [**2**] se usa un algoritmo que interconecta nodos usando una técnica de sólidos en revolución, en este proyecto se propone el siguiente sistema de resortes desarrollado a continuación.

A continuación se hará un análisis de un sistema de resortes en serie para encontrar un modelo matemático que explique el comportamiento del sistema debido a una fuerza aplicada.

En la Figura 4.32 se puede observar dos resortes unidos al nodo x2 y que en el extremo derecho del sistema actúa una fuerza que jala hacia la derecha los dos resortes, por consiguiente se efectuará un movimiento traslacional de x1 y x2.

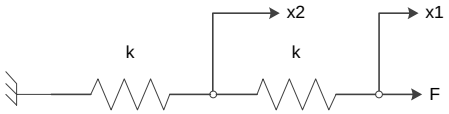


Figura 4.32: Resortes en serie

El análisis respecto al nodo x2 se describe mediante la siguiente ecuación.

$$kx_2 + k(x_2 - x_1) = 0$$

Donde k es la contante del resorte. Despejando x2 obtenemos

$$x_2 = \frac{1}{2}x_1$$

Entonces de acuerdo a este resultado se concluye que x2 se desplaza a la mitad de x1. Esta relación es importante para la aplicación del modelo deformable porque explica como los vértices del objeto se desplazan al ser afectados por el movimiento de un vértice al que se le ha aplicado una acción externa.

El análisis del nodo x1 es descrito a continuación.

$$k(x_1 - x_2) = F$$

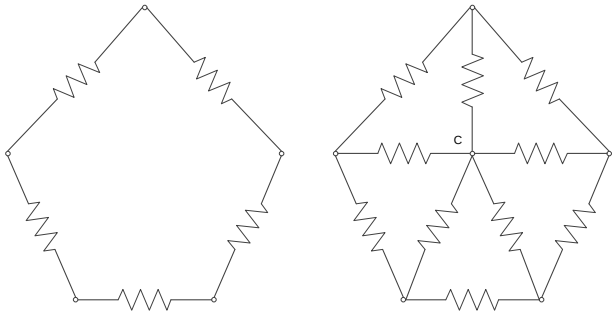
Al reemplazar x2 en la ecuación anterior tenemos

$$F = \frac{1}{2}kx_1$$

Lo cual nos relaciona la fuerza con el desplazamiento neto x1.

Para el siguiente análisis se debe tener en cuenta algo muy importante respecto al modelo deformable, si el sistema consiste solo en conexiones de resortes, en la superficie se perdería realismo debido a la ausencia de fuerzas internas en el objeto, es por eso que el modelo debe diseñarse con resortes conectados al centro geométrico del objeto para dar la sensación de deformación de volumen.

Suponer que se tiene la sección transversal de un objeto cuyos vértices de la superficie estén conectados con resortes (Figura 4.33 (a)), el objeto carece de fuerzas internas. En la Figura 4.33 (b) se conectan resortes al centro geométrico del objeto, C, de manera que se ejerzan fuerzas adicionales al interior del mismo como si se ejerciera una presión sobre las paredes internas de la superficie para dar una sensación de estar deformando un objeto volumétrico relleno.



(a) (b)

Figura 4.33: (a) sistema sin fuerzas internas, (b) sistema con fuerzas internas

Con la explicación anterior entonces se modela un sistema que incluya conexiones al interior del objeto, ese sistema se muestra en la Figura 4.34 donde C se asume que es el centro geométrico del objeto lo cual es una configuración de resortes en paralelo.

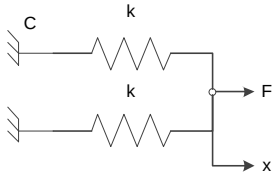


Figura 4.34: Resortes en paralelo

El análisis respecto al nodo x da la siguiente ecuación,

$$kx + kx = F$$

Que es la suma de las fuerzas de los resortes actuando sobre x tal como se muestra a continuación.

$$F = 2kx$$

Ahora el análisis se hará para dos nodos cuyo sistema es una combinación de las configuraciones en serie y paralelo. La descripción de este sistema se basa en

interconectar estos nodos y que cada nodo esté conectado al punto común C (Figura 4.35).

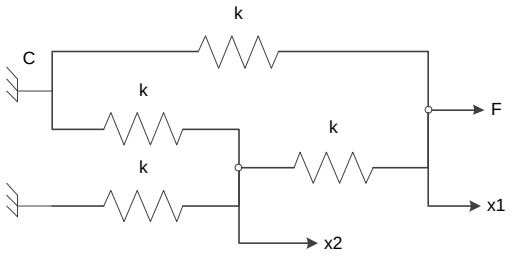


Figura 4.35: Sistema de dos nodos

El análisis para el nodo x2 es,

$$2kx_2 + k(x_2 - x_1) = 0$$

Despejando x2 da,

$$x_2 = \frac{1}{3}x_1$$

Entonces se encuentra la relación con respecto al nodo x1.

Ahora el análisis para x1 da,

$$k(x_1 - x_2) + kx_1 = F$$

Reemplazando x2 en la ecuación anterior resulta,

$$F = \frac{5}{3}kx_1$$

A continuación se hará un análisis para un sistema de tres y cuatro nodos de la misma manera cono se hizo con el anterior sistema con el propósito de observar la propagación de movimiento que tienen los vértices cuando el vértice donde actúa la fuerza se mueve.

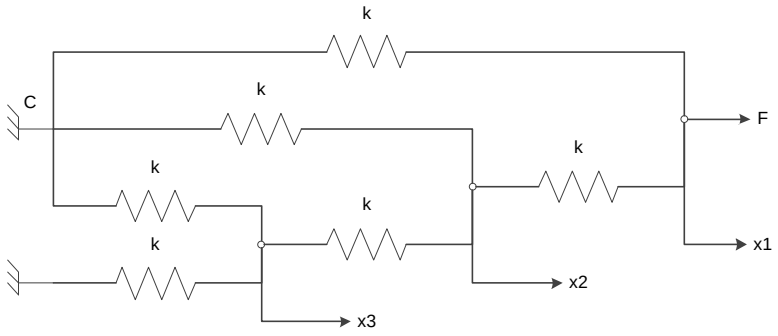


Figura 4.36: Sistema de tres nodos

$$x_3 = \frac{1}{3}x_2$$

$$k(x_2 - x_3) + kx_2 + k(x_2 - x_1) = 0$$

$$x_2 = \frac{3}{8}x_1$$

$$F = \frac{13}{8}kx_1$$

Análisis para el sistema de cuatro nodos (Figura 4.37).

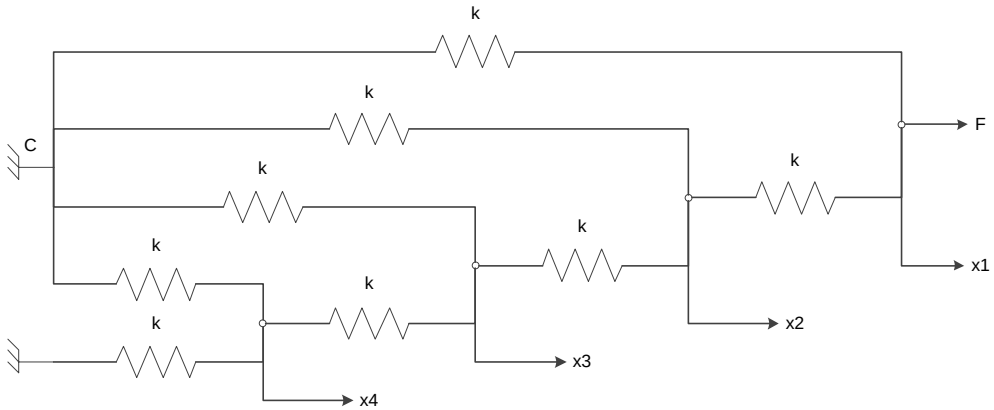


Figura 4.37: Sistema de cuatro nodos

$$x_4 = \frac{1}{3}x_3$$

$$x_3 = \frac{3}{8}x_2$$

$$x_2 = \frac{8}{21}x_1$$

$$F = \frac{34}{21}kx_1$$

$$x_2 = \frac{8}{21}x_1, \quad x_3 = \frac{3}{21}x_1, \quad x_4 = \frac{1}{21}x_1 \qquad (4.1)$$

De las ecuaciones 4.1 se ve la relación que existe entre los nodos x2 al x4 respecto al desplazamiento de x1. La descripción de estas relaciones se advierte claramente cómo el desplazamiento de los nodos se atenúa a medida que se alejan del nodo de acción, es decir que los nodos más lejanos no se desplazan tanto como los cercanos.

Hasta ahora se ha hecho un análisis para cuatro nodos pero en un modelo 3d se pueden encontrar muchos más nodos, entonces se debe encontrar una ecuación para *n* nodos. Para eso se retoman las ecuaciones resultantes de los sistemas desde un nodo hasta cuatro nodos hechas anteriormente.

$$F = \frac{2}{1}kx, \quad F = \frac{5}{3}kx_1, \quad F = \frac{13}{8}kx_1, \quad F = \frac{34}{21}kx_1$$

De las ecuaciones anteriores, si se escriben los números en orden ascendente empezando por el denominador de la primera ecuación, luego el numerador y así sucesivamente en cada una de las ecuaciones, se obtiene,

1, 2, 3, 5, 8, 13, 21, 34.

Esta secuencia de números corresponde a la sucesión de Fibonacci,

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,...

Entonces para *n* términos se tiene la siguiente definición recursiva que genera esta sucesión.

$$a_n = \begin{cases} 0, & si\ n = 0 \\ 1, & si\ n = 1 \\ a_{n-1} + a_{n-2}, & si\ n > 1 \end{cases}$$

Ahora si se observa la ecuación 4.1, en cada nodo se ve que existe un denominador común. Se necesita encontrar una ecuación que permita hallar este valor con datos relacionados al sistema.

Una fórmula que es de mucha utilidad para encontrar cualquier número de Fibonacci es la proporción áurea [**30**].

$$f(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$$

Siendo φ el llamado número de oro [**30**].

$$\varphi = \frac{1+\sqrt{5}}{2}$$

Ahora si se retoma el sistema de cuatro nodos, el número que interesa encontrar es el 21 debido a la ecuación 4.1, este número corresponde al termino 8 de la sucesión de Fibonacci, en consecuencia este término es el numero de nodos por dos.

$$n = 4 \times 2$$

Ahora reemplazando n en la formula de la proporción áurea,

$$f(n) = 21$$

Que es el valor de interés.

En el numerador de cada uno de los vértices se observa que son los términos pares de la sucesión de Fibonacci, en este caso los términos 2, 4 y 6 correspondientes a 1, 3 y 8.

Todo esto se aplica para cualquier número de nodos. Por tanto es fácil encontrar la relación de movimiento entre los distintos nodos respecto al desplazamiento del nodo a mover.

Iteración del proceso de deformación

La Figura 4.38 muestra cómo funciona el proceso de deformación basado en el desplazamiento del triángulo de contacto, si se tiene un objeto 3d con cuatro niveles de propagación, de acuerdo a esta información, cada nivel de vértices se desplazará en relación al movimiento de los vértices del triangulo contactado.

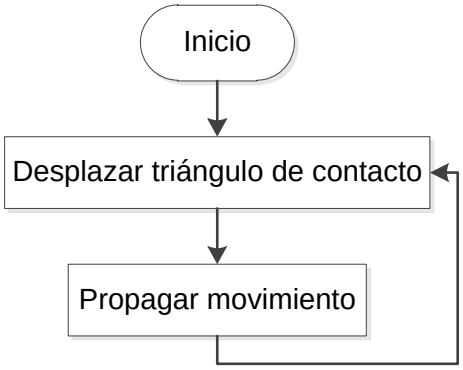


Figura 4.38: Proceso de deformación

Restauración de forma

La figura 4.39 muestra los elementos para restaurar el objeto a su forma original. El elemento vértices propagados proporciona información de los vértices afectados en la malla. El elemento sistema discreto MBK contiene la ecuación diferencial discretizada del sistema masa resorte amortiguador que ejecutará la dinámica de restauración.

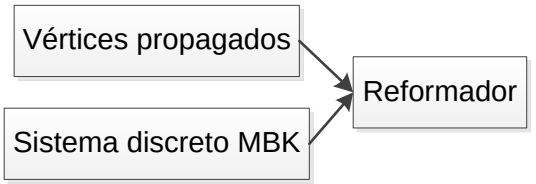


Figura 4.39: Estructura del método restauración de forma

Para que el rendimiento de un sistema de deformación sea óptimo es importante elegir un método de discretizacion rápido y preciso, ya que el tiempo que se requiere para calcular la deformación del objeto está relacionado proporcionalmente a la cantidad de operaciones que utilice el método [**6**].

La discretización por diferencias finitas es un método simple, rápido y preciso comparado con otros métodos y ofrece un margen de error pequeño [**6**]. Estas características hacen este método conveniente para realizar la discretización del algoritmo masa resorte amortiguador implementado en este proyecto.

A continuación se resuelve la ecuación del sistema masa resorte amortiguador por medio del método de diferencias finitas.

$$F = m\ddot{x} + b\dot{x} + kx$$

$$\dot{x} = \frac{x(t) - x(t-1)}{\Delta t}$$

$$\ddot{x} = \frac{V_f - V_i}{\Delta t}$$

$$F = m\frac{V_f - V_i}{\Delta t} + b\frac{x(t) - x(t-1)}{\Delta t} + kx(t)$$

$$V_f = \frac{x(t+1) - x(t)}{\Delta t}$$

$$V_i = \frac{x(t) - x(t-1)}{\Delta t}$$

$$x(t+1) = \left(2 - \frac{b}{m}\Delta t - \frac{k}{m}\Delta t^2\right)x(t) - \left(1 - \frac{b}{m}\Delta t\right)x(t-1) + \frac{F}{m}\Delta t^2$$

$$x(t) = \left(2 - \frac{b}{m}\Delta t - \frac{k}{m}\Delta t^2\right)x(t-1) - \left(1 - \frac{b}{m}\Delta t\right)x(t-2) + \frac{F}{m}\Delta t^2$$

$$A = \left(2 - \frac{b}{m}\Delta t - \frac{k}{m}\Delta t^2\right)$$

$$B = \left(1 - \frac{b}{m}\Delta t\right)$$

$$C = \frac{kx}{m}\Delta t^2$$

$$x(t) = Ax(t-1) - Bx(t-2) + C$$

En código:

```
A = 2-(b/m)*h-(k/m)*h*h;
B = (b/m)*h-1;
C = (k*x/m)h*h;
```

Inicialización de estados

```
xk1 = coordenadas_vertice;
xk2 = coordenadas_vertice;
```

Dinámica del sistema

```
xk = A*xk1+B*xk2+C;
```

Actualización de estados

```
xk2 = xk1;
xk1 = xk;
```

4.7. COMPONENTE REPRESENTACIÓN VISUAL

Este componente se encarga del despliegue visual de la escena, colocándola en una ventana.

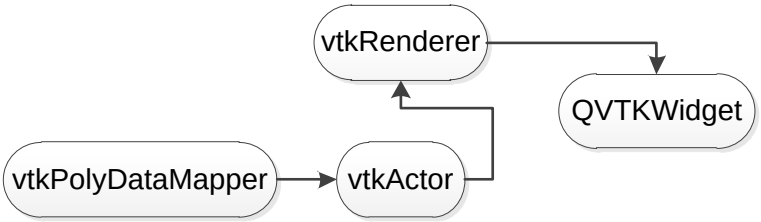


Figura 4.40: Estructura del componente Representación visual

A continuación se representa en código el esquema de la Figura 4.40.

```
actor->SetMapper(mapper);
renderer->AddActor(actor);
Visualizacion->GetRenderWindow()->AddRenderer(renderer);
```

Siendo mapper una instancia de la clase vtkPolyDataMapper, actor es una instancia de la clase vtkActor y Visualizacion es una instancia de la clase QVTKWidget.

4.8. COMPONENTE INTERACCIÓN

Este componente hace posible la interacción del usuario con la escena virtual.

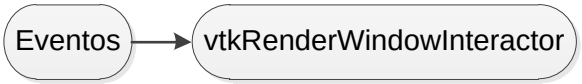


Figura 4.41: Estructura del componente interacción

A continuación se representa en código el esquema de la Figura 4.41.

```
rwi->SetInteractorStyle(interaccion);
```

interaccion es una instancia de la clase Eventos, rwi es una instancia de la clase vtkRenderWindowInteractor.

4.9. MODELADO UML

En esta parte se construyeron algunos diagramas UML con el fin de entender más a fondo la estructura de la aplicación creada. UML es una especificación de notación con que se construyen sistemas por medio de la programación orientada a objetos [**31**]. Es un lenguaje estándar diseñado para visualizar, especificar, construir y documentar software orientado a objetos.

4.9.1. Diagrama de clases

Este tipo de diagrama describe gráficamente las relaciones entre las clases software y las interfaces en un sistema. Generalmente tiene un conjunto de elementos que son estáticos, como clases, asociaciones, atributos, métodos, entre otros [**31**].

Una clase está representada por un rectángulo que dispone de tres apartados, el primero para indicar el nombre, el segundo para los atributos y el tercero para los métodos. Cada clase debe tener un nombre único, que la diferencie de las otras. Se puede observar a continuación en los diagramas de clases de las Figuras 4.42 y 4.43 donde se definen las características de cada una de las clases del sistema, además de sus relaciones de dependencia.

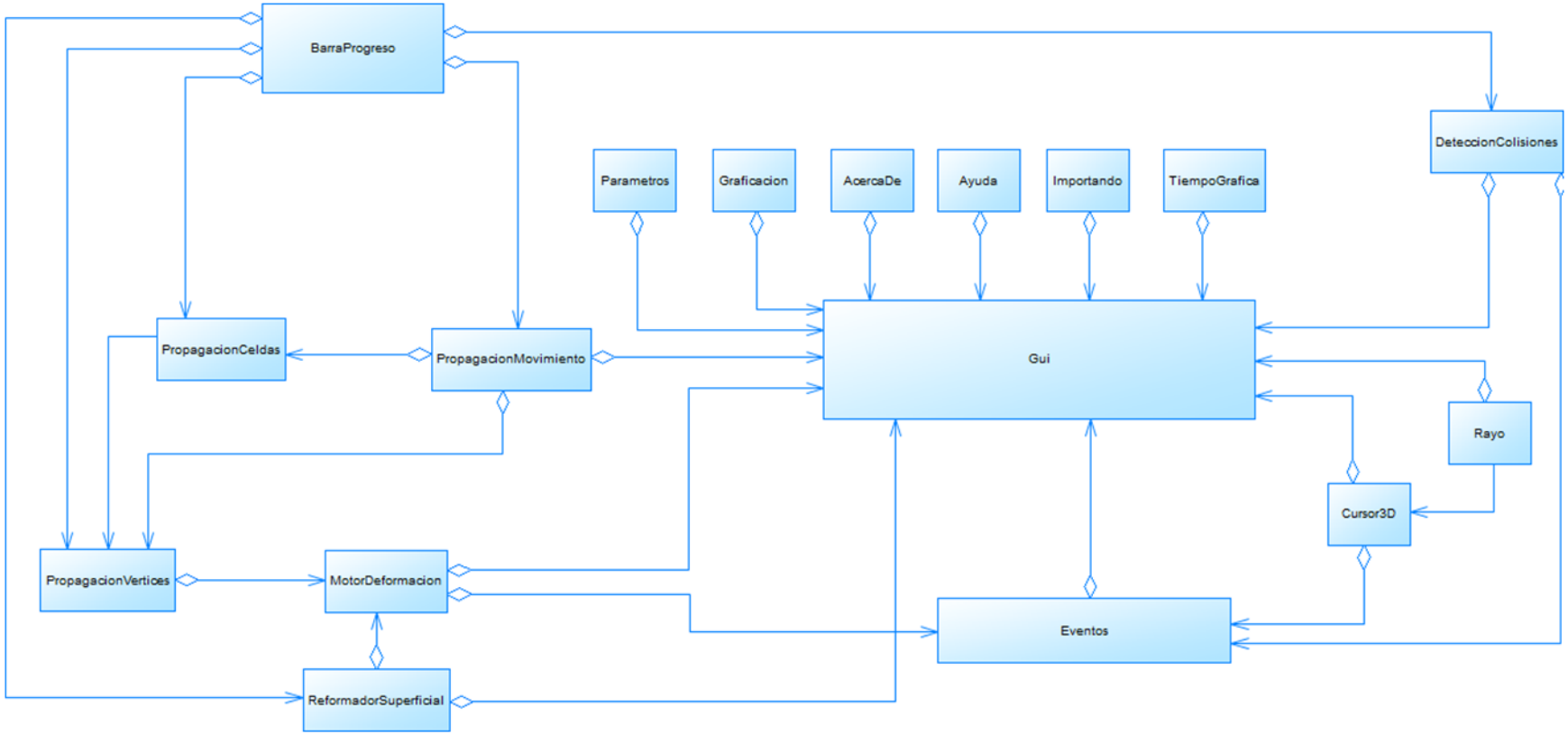


Figura 4.42: Diagrama de clases de relaciones del sistema

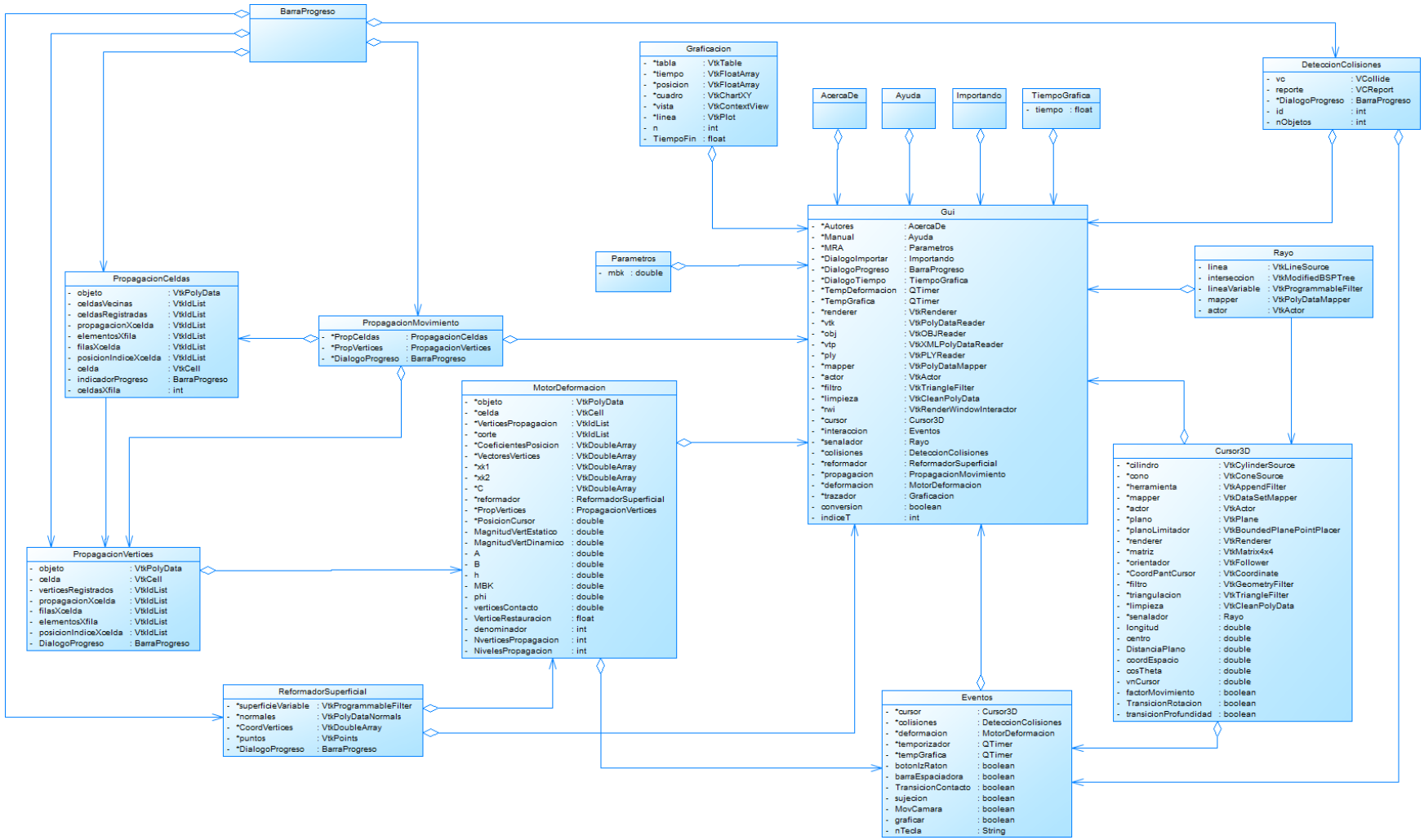


Figura 4.43: Diagrama de clases principal del sistema

5. RESULTADOS Y PRUEBAS

Las pruebas realizadas se hicieron con las siguientes características hardware.

Procesador: AMD Athlon Dual-Core 1.90 GHz.
Memoria: 2,0 GB
Tarjeta gráfica: GeForce 8200M G 256 MB

5.1. DEFORMACIÓN

En esta sección se realizan pruebas de deformación, presión y estiramiento de tres objetos diferentes, donde el usuario ejerce una acción sobre el objeto con una herramienta en forma de lápiz.

5.1.1. Deformación de una esfera

En esta sección se realiza una prueba sobre una esfera de 2310 triángulos y 1157 vértices.

Acción de presión

La figura 5.1 muestra el lápiz ejerciendo presión a una esfera, se aprecia como cierta región del objeto se hunde dando la impresión de estar presionando un objeto solido (no está hueco por dentro). Como ya se ha mencionado en capítulos anteriores, la acción del objeto rígido se hace sólo sobre un triangulo del objeto deformable y este triángulo arrastra los vértices que lo rodean. Es de esperar que para poder realizar esta acción de deformación se necesita detectar el lugar y momento de la colisión, por tanto se comprueba la colisión entre estos dos objetos.

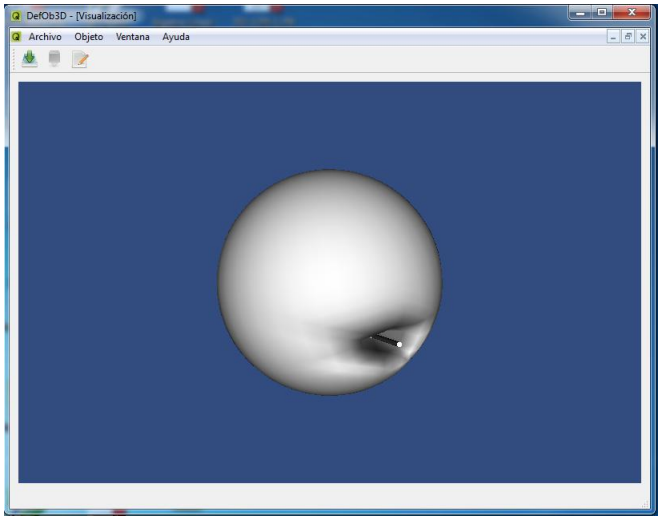


Figura 5.1: Presión sobre una esfera

Acción de estiramiento

La figura 5.2 muestra como se estira parte de la esfera al tomar un triángulo y arrastrarlo hasta cierto punto.

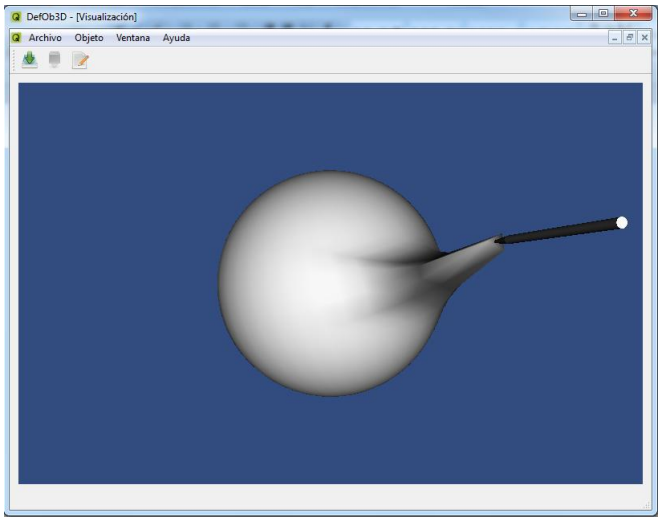


Figura 5.2: Estiramiento sobre una esfera

Tiempo de restauración

En esta etapa se realizan pruebas con esferas de resoluciones diferentes para ver como se ve afectado el tiempo de respuesta de restauración del objeto. Este indicador permite observar como el objeto deformable responde de acuerdo a la resolución del objeto ya que se requieren de más cálculos para procesar y por ende un retardo de tiempo en la recuperación de forma del objeto.

Se hicieron tres pruebas con una esfera cuya resolución es incrementada por cada ensayo. La tabla 5.1 muestra las características de resolución de los modelos de prueba.

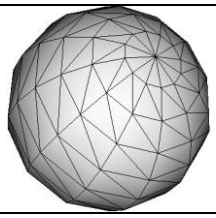
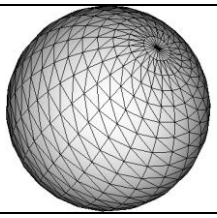
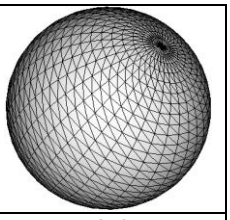
Imagen del modelo			
Modelo	Modelo 1	Modelo 2	Modelo 3
Triángulos	240	1056	2310
Vértices	122	530	1157

Tabla 5.1: Modelos de prueba

En la gráfica de restauración de forma (figura 5.3) se muestra la magnitud del estiramiento realizado a uno de los vértices del triángulo de contacto (Pos) con respecto al tiempo. La magnitud inicial del vértice es de 1 (cm), en este caso como el radio de la esfera es uno entonces todo vértice de la esfera tiene magnitud uno. Éste vértice se estira hasta 2.1 (cm) respecto al centro geométrico del objeto, luego se suelta y se observa un movimiento transitorio que tiende a situar la posición del vértice a su magnitud inicial de uno, la figura 5.3 muestra que al modelo 1 le toma cinco segundos recuperar su forma original.

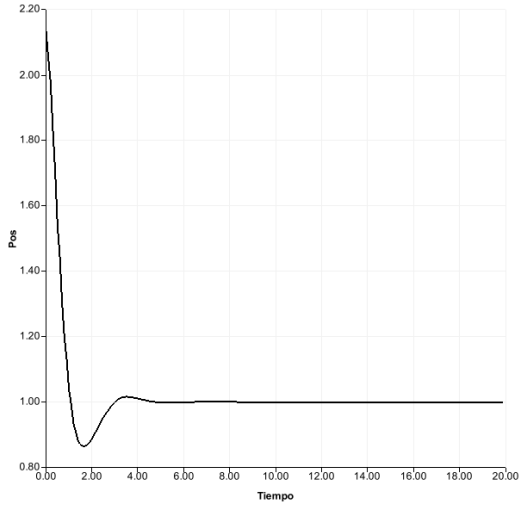


Figura 5.3: Tiempo de respuesta modelo 1

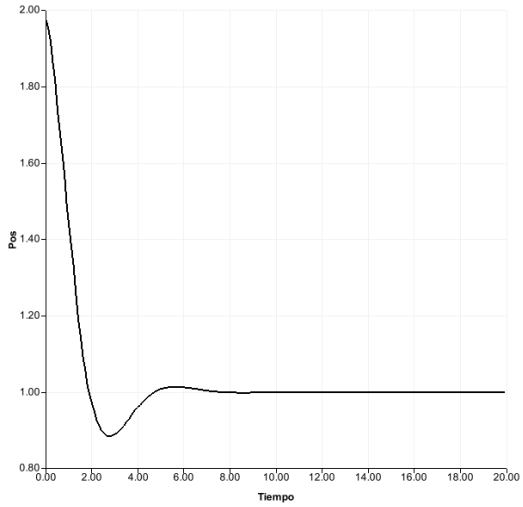


Figura 5.4: Tiempo de respuesta modelo 2

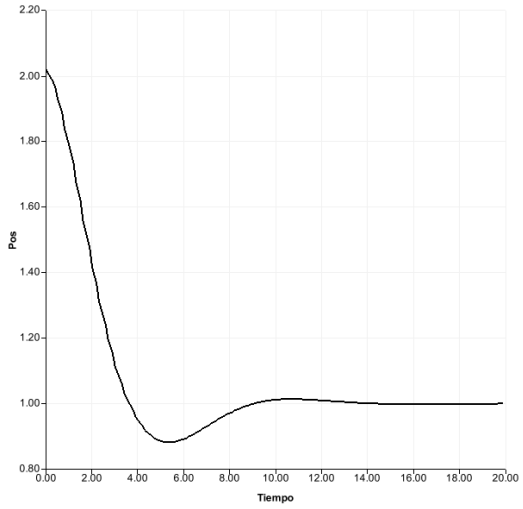


Figura 5.5: Tiempo de respuesta modelo 3

La figura 5.4 muestra que al modelo 2 le toma alrededor de nueve segundos recuperar su forma original en comparación de los cinco segundos que le toma al modelo 1 debido a que se deben procesar más vértices y sobre todo a que VTK actualiza toda la malla en la renderización.

Finalmente la figura 5.5 muestra que el modelo 3 le toma aproximadamente quince segundos recuperar su forma original.

5.1.2. Otras pruebas

También se realizaron pruebas con otros objetos para verificar que la aplicación no solo funciona con modelos de geometría esférica.

La Tabla 5.2 muestra las características de los objetos cubo, cilindro e hígado para la prueba de deformación.

Las figuras 5.6 a 5.9 ilustran las deformaciones de los objetos cubo, cilindro e hígado.

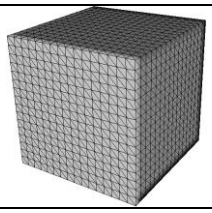
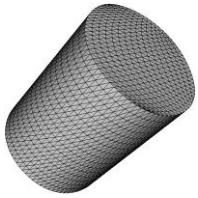
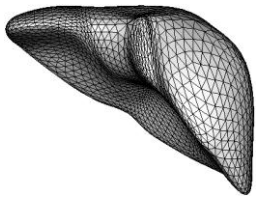
Modelo			
Triángulos	3072	4096	4384
vértices	1538	2050	2194
Formato	vtp	ply	vtk

Tabla 5.2: Características de los modelos cubo y cilindro

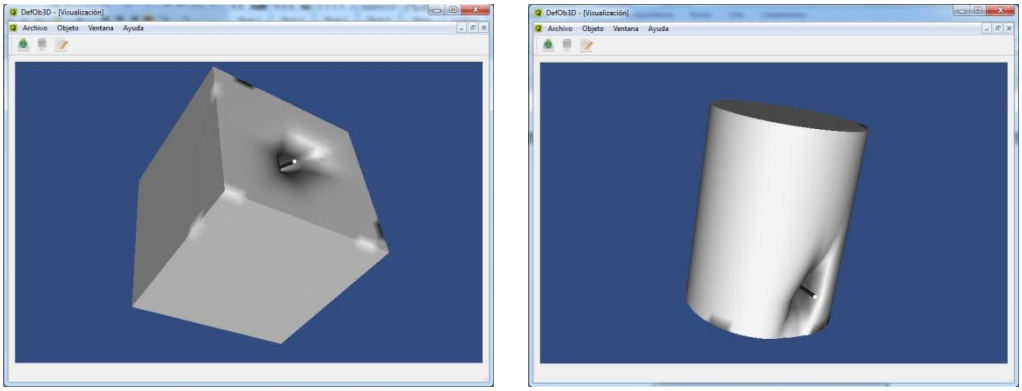


Figura 5.6: Presión sobre los objetos cubo y cilindro

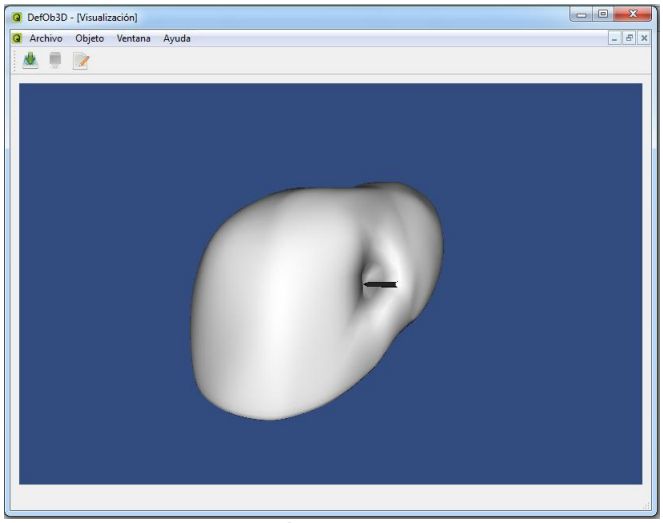


Figura 5.7: Presión sobre objeto irregular

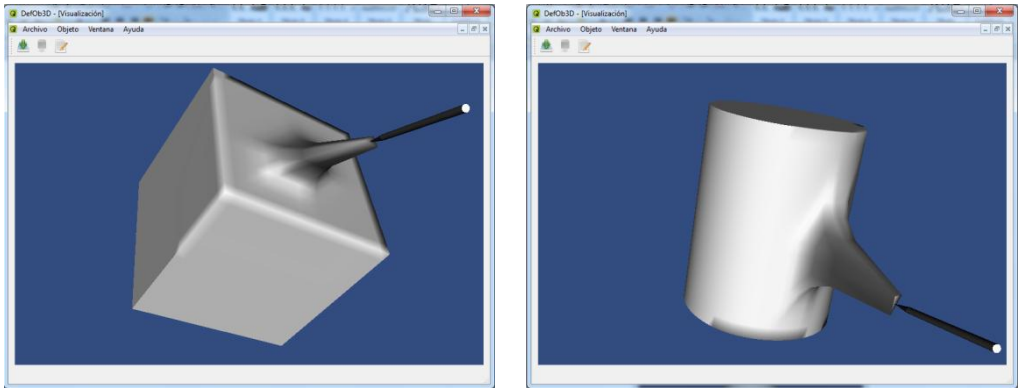


Figura 5.8: Estiramiento sobre los objetos cubo y cilindro

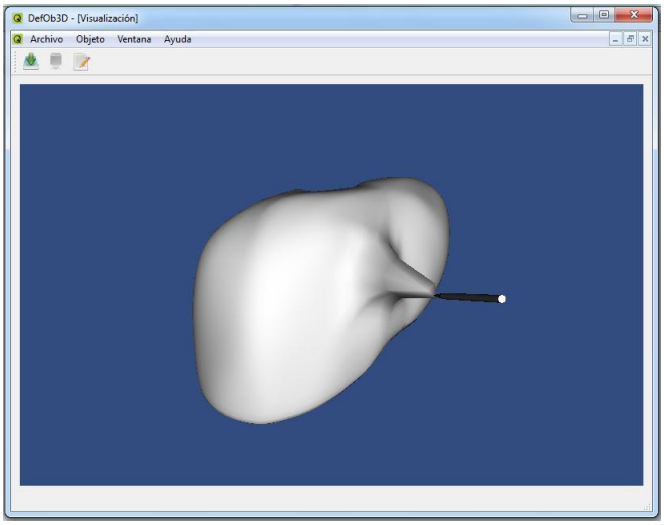


Figura 5.9: Estiramiento sobre objeto irregular

5.2. DESEMPEÑO DE VISUALIZACIÓN

El desempeño de la aplicación se ve afectado en gran parte por la renderización de objetos de alta resolución, en este caso se hacen pruebas de renderización de objetos de diferentes resoluciones para ver como se ve afectada la aplicación. La tabla 5.3 muestra que a medida que la resolución en los objetos aumenta los *frames* disminuyen. La caída de los *frames* es causa de la actualización de la malla del objeto al renderizar su deformación.

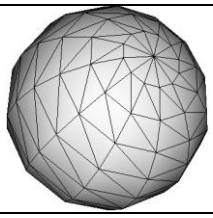
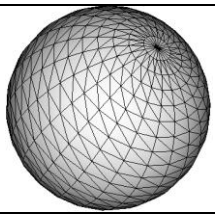
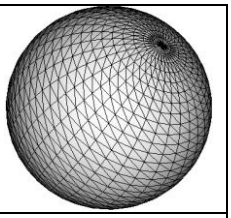
Modelo			
Triángulos	240	1056	2310
Vértices	122	530	1157
FPS	100	46	21

Tabla 5.3: Cuadros por segundo según la resolución

5.3. EVALUACIÓN DE COMPONENTES

La Tabla 5.4 registra la validación de los requerimientos funcionales y de los componentes involucrados en la aplicación software 3d según las pruebas realizadas. En la primera columna se especifican los componentes directamente relacionados, en la segunda columna se presenta un identificador de cada requerimiento, en la tercera columna se realiza la descripción de los requerimientos, y en la cuarta columna se describe si se cumple o no el requerimiento.

Componente	Id	Requerimientos funcionales	
Acondicionamiento	RF-01	La aplicación debe permitirle al usuario final importar el objeto que desea deformar	✓
Cursor3D	RF-02	Se debe permitir al usuario deformar el objeto importado, haciendo uso de un cursor 3d	✓
Colisiones			
Deformación			
Interacción			
Deformación	RF-03	La aplicación debe estar en la capacidad de restaurar el objeto a su forma original	✓
Representación visual	RF-04	El sistema debe permitirle al usuario generar una gráfica de tiempo de respuesta	✓
Deformación	RF-05	El usuario debe poder modificar los valores de la masa, el resorte y el amortiguador para el sistema	✓

Tabla 5.4: Evaluación de requerimientos

6. CONCLUSIONES Y TRABAJOS FUTUROS

6.1. CONCLUSIONES

En este proyecto se desarrolló una aplicación software 3d para deformación de objetos virtuales realizando un análisis de las herramientas requeridas que permitan la implementación de un entorno virtual donde se pueda representar gráficamente el comportamiento de deformación de un objeto 3d.

El estudio y análisis desarrollado en este proyecto sobre las librerías VTK dispone de un conocimiento de las clases usadas en la implementación de la aplicación de modo que este desarrollo sirva de guía para futuras aplicaciones que estén relacionas con este tipo de trabajos.

De las aplicaciones encontradas que hacen uso de las librerías VTK se encuentra un enfoque más al procesamiento y visualización de objetos rígidos tridimensionales, además de simulaciones con interacción programada observando escasamente interacciones directas del usuario con el entorno virtual que pudiera de cierta forma modificar en tiempo real el contenido del escenario virtual.

Por medio de pruebas de estiramiento y presión en un punto sobre la superficie de un objeto virtual se probó la efectividad de usar un elemento mecánico como el resorte para simular un comportamiento elástico en mallas triangulares.

Para una mejor percepción visual de la deformación del objeto es imprescindible tener una resolución alta, pero esto implica un decremento en el desempeño de la aplicación debido a que la cantidad de vértices a procesar es alta.

VCollide se establece como una herramienta de detección de colisiones eficiente debido a que no genera demasiada carga computacional en la aplicación, algo muy importante si se desea velocidad de renderizado.

6.2. TRABAJOS FUTUROS

Como trabajos futuros se plantea ampliar las opciones de la aplicación con otros modelos deformables además de integrar interfaces hápticas para mayor interacción. También se podría desarrollar un sistema de deformación que permita realizar cortes en la superficie de los objetos.

La aplicación sólo maneja un triángulo de contacto y como consecuencia el objeto es deformado por una sola herramienta, sería conveniente que la aplicación pudiera ampliar esta funcionalidad para múltiples triángulos de contacto de forma que varias herramientas puedan hacer la deformación de forma simultánea.

La funcionalidad de propagación a partir de un sólo triángulo de contacto también limita la aplicación a deformar sobre un punto de la superficie del objeto, sería conveniente modelar un sistema que permita deformar sobre áreas de la superficie que se amolden a volúmenes de objetos rígidos.

RECOMENDACIONES

En este capítulo se describen algunas sugerencias que ayudarán al desarrollo de trabajos futuros.

- Múltiples puntos de contacto.

Si se quiere que más de una herramienta haga contacto con el objeto, se debe configurar el sistema de colisiones para que funcione con más de dos objetos.

- Simulación de cortes.

Si se desea que el objeto pudiera sufrir algún tipo de ruptura en su superficie se deben realizar dos acciones: la primera acción consistiría en el aspecto visual que se encarga de eliminar los triángulos contactados para el corte, para esto se puede estudiar la clase vtkClipPolyData, la segunda acción consistiría en excluir los vértices de los triángulos eliminados en la dinámica de deformación.

- Propagación a partir de múltiples triángulos.

Si se quiere una propagación de más de un triángulo, se podría modificar el algoritmo de acceso al arreglo de vértices que identifique los niveles de propagación de cada triángulo y agrupe estos niveles.

- Apariencia de un objeto

Si se desea cambiar la apariencia de un objeto ya sea cambiando su color o textura, se puede estudiar la clase vtkTexture que maneja la textura de un objeto a partir de datos provenientes de imágenes.

REFERENCIAS BIBLIOGRÁFICAS

[1] Michael Hauth, "Visual Simulation of Deformable Models", Eberhard Karls Universität Tübingen, Tübingen, Tesis doctoral en ciencias exactas, 2004.

[2] Miriam Mecate, "Interacción con Objetos Deformables", Instituto Politécnico Nacional, México, Tesis de maestría en ciencias, 2008.

[3] Óscar Lopez, "ParSys", Universidad Politécnica de Valencia, España, Tesis doctoral, 2007.

[4] Osley Bretau y Raissel Ramírez, "Deformación de Objetos para Sistemas de Realidad Virtual", en XIII Convención y Feria Internacional de Informática, Habana, Cuba, 2009.

[5] Julio Moctezuma, "Manipulación Tridimensional de Objetos Deformables Virtuales", Instituto Politécnico Nacional, México, Tesis de maestría en ciencias, 2006.

[6] Claudia Ramírez, "Animación de Modelos Deformables", Instituto Politécnico Nacional, México, Tesis de maestría en ciencias, 2005.

[7] José Casao et al., "Implementación de Objetos Deformables en un Simulador Dinámico", en XXVIII Jornadas de Automática, Elche, España, 2007.

[8] Gabriel Sepúlveda, Vicente Parra, y Omar Domínguez, "Visualización 3D de Deformación y Corte de Objetos Virtuales Basada en Descomposición Ortogonal", Instituto Politécnico Nacional, México, Artículo 2008.

[9] María Pinto, José Sabater, y Jorge Esmeral, "Análisis de la Detección de Colisiones en un Entorno Virtual para Aplicaciones Hápticas de Asistencia Quirúrgica", Ingeniería e Investigación, vol. 31, no. 1, pp. 204-212, Abril 2011.

[10] Ignacio Berzal, "Desarrollo de Algoritmos de Procesamiento de Imágenes con VTK", Universidad Politécnica de Madrid, Madrid, Tesis profesional, 2004.

[11] Miguel Martín, "Esquemas Multiresolución para Compresión de Datos Volumétricos", Universidad de Valladolid, Valladolid, España, Proyecto fin de carrera 1999.

[12] Pablo Álvarez y Ignacio Rodriguez, "j3dEngine", Universidad Austral, Argentina, Proyecto de trabajo de grado, 2007.

[13] Sarah Gibson, “3D ChainMail: a Fast Algorithm for Deforming Volumetric Objects”, Mitsubishi Electric Research Laboratories, 1996.

[14] Thomas Sederberg y Scott Parry, “Free Form Deformation of Solid Geometric Models”, SIGGRAPH, 1986.

[15] Demetri Terzopoulos y Kurt Fleischer, “Deformable Models”, Springer-Verla, Palo Alto, 1988.

[16] Sarah Gibson y Brian Mirtich, “A Survey of Deformable Modeling in Computer Graphics”, Mitsubishi Electric Research Laboratories, 1997.

[17] Katsuhiko Ogata, Dinamica de Sistemas, Primera edición ed. México, México: Prentice Hall Hispanoamericana, 1987.

[18] Juan Jiménez, "Detección de Colisiones mediante Recubrimientos Simpliciales", Universidad de Granada, Granada, Tesis doctoral en informática, 2006.

[19] María del Carmen Ramírez, "Estudio e Implementación de un Algoritmo de Detección de Colisiones Basado en Esferas", Universidad de las Américas Puebla, Puebla, Tesis de maestría en ciencias, 2005.

[20] María Pinto, "Análisis e Implementación de una Interfaz Háptica en Entornos Virutales", Universidad Nacional de Colombia, Bogotá, Tesis de maestría en ingeniería en automatización industrual, 2009.

[21] Michael Arias, "La Ingeniería de Requerimientos y su Importancia en el Desarrollo de Proyectos de Software", InterSedes, vol. VI, no. 10, pp. 1-12, Julio 2007.

[22] Ian Sommerville, Ingeniería del Software, Séptima edición ed. Madrid, España: Pearson Educación, 2005.

[23] Visualization Toolkit. [En línea]. www.vtk.org. [Consultado: Noviembre, 2010]

[24] gamma. [En línea]. gamma.cs.unc.edu. [Consultado: Septiembre, 2011]

[25] (2008) Qt. [En línea]. qt.nokia.com. [Consultado: Junio, 2011]

[26] Job Acevedo, "Desarrollo de Aplicaciones Médicas Utilizando las Bibliotecas ITK y VTK", Benemerita Universidad Autonoma de Puebla, Puebla, Tesis profesional en ciencias de la computación, 2006.

[27] Helber Mayorca y Adriana Torres, "Entorno Gráfico de un Simulador Quirúrgico 3D", Universidad del Cauca, Popayán, Trabajo de grado en Ingeniería en Automática Industrial, 2011.

[28] Will Schroeder, Ken Martin, y Bill Lorensen, The Visualization Toolkit, Third edition ed. United States of America: Pearson Education, 2002.

[29] Daniel Ivorra, "Simulador Háptico para Entrenamiento de Técnicas de Laparoscopia", Universidad Miguel Hernández de Elche, Elche, España, Proyecto fin de carrera 2008.

[30] Francesc Comellas, Josep Fábrega, Anna Sánchez, y Oriol Serra, Matemática Discreta, Primera edición ed. Barcelona: Edicions UPC, 2001.

[31] Craig Larman, UML y Patrones, Primera edición ed. México, México: Prentice Hall Hispanoamericana, 1999.