UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE CIENCIAS

VISUALIZACIÓN TRIDIMENSIONAL INTERACTIVA Y EN TIEMPO REAL DE YACIMIENTOS DE HIDROCARBUROS

REPORTE DE TRABAJO PROFESIONAL

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A :

ALAN VILLEGAS RENDÓN

FACULTAD DE CIENCIAS
UNAM

DIRECTOR: DR. VÍCTOR HUGO ARANA ORTIZ

2007

FACULTAD DE CIENCIAS

División de Estudios Profesionales

ACT. MAURICIO AGUILAR GONZÁLEZ
Jefe de la División de Estudios Profesionales
Facultad de Ciencias
P r e s e n t e .

Por este medio hacemos de su conocimiento que hemos revisado el trabajo escrito titulado:

"Visualización tridimensional interactiva y en tiempo real de yacimientos de hidrocarburos"

realizado por **Villegas Rendón Alan,** con número de cuenta **095130884,** quien opta por titularse en la opción de **Trabajo Profesional** de la Licenciatura en **Ciencias de la Computación**. Dicho trabajo cuenta con nuestro voto aprobatorio.

Propietario	Dr.	Pablo Barrera Sánchez
Propietario	Mat.	María Concepción Ana Luisa Solís González-Cosío
Tutor(a) Propietario	Dr.	Víctor Hugo Arana Ortiz
Suplente	M. en C.	María Guadalupe Elena Ibargüengoitia González
Suplente	Dr.	José de Jesús Galaviz Casas

A t e n t a m e n t e
"POR MI RAZA HABLARÁ EL ESPÍRITU"
Ciudad Universitaria, D.F., a 15 de octubre del 2007.
EL COORDINADOR DEL COMITÉ DE TITULACIÓN
DE LA LICENCIATURA EN CIENCIAS DE LA COMPUTACION

FACULTAD DE CIENCIAS
CONSEJO DEPARTAMENTAL
DE
MATEMÁTICAS

DR. JOSÉ DE JESÚS GALAVIZ CASAS

Señor sinodal: antes de firmar este documento, solicite al estudiante que le muestre la versión digital de su trabajo y verifique que la misma incluya todas las observaciones y correcciones que usted hizo sobre el mismo.

a mi hermano
Jonathan Villegas Rendón

a mi hermana
Mariana Yalin Villegas Rendón

AGRADECIMIENTOS

A mi Amigo Eterno por permitirme vivir esta vida.

A mi familia por su constante apoyo.

A mi querida amiga Liliana por todos estos años que hemos disfrutado juntos.

A todos mis amigos y en especial a aza, pipo, chester, fito, jorge, morgado y greñas, por compartir alegrías, tristezas, enojos, debralles y claro esta, un buen juego de haki.

A mis amigas vero, lore, vicky y Paola por las grandes platicas que hemos compartido.

A toda la G4 por ser la mejor generación que me pudo haber tocado.

A mis profesores por compartir su conocimiento, especialmente a José y a Jaime por ser mis amigos.

A todo el GSNY por las facilidades otorgadas para la realización de este trabajo.

A mi tutor y sinodales por todo su apoyo brindado.

A todos gracias.

Índice general

Índice de figuras v

Índice de secuencias vii

1. Introducción 1
- 1.1. Descripción del Proyecto . . . 1
  - 1.1.1. Objetivo . . . 2
- 1.2. Fundamentos . . . 3
  - 1.2.1. Motivación para la simulación . . . 4
  - 1.2.2. Como trabaja un simulador . . . 4
- 1.3. Motivación para la visualización . . . 6
- 1.4. Alcance . . . 7

2. Requerimientos de Visualización 9
- 2.1. Requerimientos de usuario . . . 9
  - 2.1.1. Objetivo . . . 9
  - 2.1.2. Ambiente . . . 10
  - 2.1.3. Actores . . . 10
  - 2.1.4. Necesidades de rendimiento capacidad . . . 10
- 2.2. Requerimientos de software . . . 10
  - 2.2.1. Requerimientos funcionales . . . 10
  - 2.2.2. Requerimientos no funcionales . . . 13

3. Modelado UML 15
- 3.1. Diseño conceptual . . . 15
  - 3.1.1. Módulo de Caracterización de Fluidos (PVT) . . . 15
  - 3.1.2. Módulo de Mallas Numéricas (GRID) . . . 16
  - 3.1.3. VFP y Modelación de Pozos-Yacimientos (VFP) . . . 16
  - 3.1.4. Simulador de Yacimientos (Reservoir Simulator) . . . 17
  - 3.1.5. Ajuste de Historia (History Matching) . . . 17

3.1.6. Análisis Económico (Economic Analysis) . . . 17
3.2. Arquitectura del Simulador Numérico de Yacimientos Multipropósito . . . 17
3.2.1. Shared Memory . . . 17
3.2.2. Simulation Thread . . . 18
3.2.3. Sequential Simulation Engine . . . 19
3.2.4. Link . . . 19
3.2.5. IOManager . . . 19
3.2.6. 2D Graphics . . . 19
3.2.7. 3D Visualization . . . 19
3.2.8. Graphic User Interface . . . 19
3.3. Diseño del módulo de Visualización 3D . . . 20
3.3.1. Diagramas de secuencia . . . 22

4. Diseño Detallado de Algoritmos 47
4.1. Clase "Conector" . . . 47
4.2. Clase "VisMesh" . . . 47
4.3. Clase "VisMeshO" . . . 48
4.3.1. Variables principales . . . 48
4.3.2. Funciones principales . . . 50
4.4. Clase "VisMeshR" . . . 56
4.4.1. Variables principales . . . 57
4.4.2. Funciones principales . . . 58
4.5. Clase "PointsMeshO" . . . 62
4.5.1. Variables principales . . . 63
4.5.2. Funciones principales . . . 63
4.6. Clase "PointsMeshR" . . . 64
4.6.1. Variables principales . . . 64
4.6.2. Funciones principales . . . 65
4.7. Clase "ReferenceMesh" . . . 67
4.7.1. Variables principales . . . 68
4.7.2. Funciones principales . . . 68
4.8. Clase "TrackBall" . . . 71
4.9. Clase "ColorLUT" . . . 72
4.9.1. Variables principales . . . 72
4.9.2. Funciones principales . . . 72
4.10. Clase "MapColor" . . . 74
4.10.1. Variables principales . . . 74
4.10.2. Funciones principales . . . 74
4.11. Clase "Well" . . . 75
4.11.1. Variables principales . . . 75
4.11.2. Funciones principales . . . 76

4.12. Clase "Bubbles" . . . 78
4.12.1. Variables principales . . . 78
4.12.2. Funciones principales . . . 79
4.13. Clase "CutPlanes" . . . 81
4.13.1. Variables principales . . . 82
4.13.2. Funciones principales . . . 82
4.14. Clase "Point3D" . . . 85
4.14.1. Variables principales . . . 85
4.14.2. Funciones principales . . . 85
4.15. Clase "Plane3D" . . . 86
4.15.1. Variables principales . . . 86
4.15.2. Funciones principales . . . 87
4.16. Clase "IsolinesMO" . . . 87
4.16.1. Variables principales . . . 88
4.16.2. Funciones principales . . . 89
4.17. Clase "ManagerProp" . . . 91
4.17.1. Variables principales . . . 92
4.17.2. Funciones principales . . . 93

5. Resultados **99**
5.1. Interacción con la malla . . . 99
5.2. Opciones de visibilidad . . . 100
5.3. Opciones de color y transparencia . . . 103
5.4. Exploración de celdas internas . . . 105
5.5. Modificación de la altura de las celdas . . . 105
5.6. Interpolación y degradado de colores . . . 105
5.7. Opciones de pozos . . . 106
5.8. Burbujas . . . 109
5.9. Representación ideal de medios . . . 109
5.10. Curva de saturaciones . . . 110
5.11. Cortes mediante un plano y mediante pozos . . . 111
5.12. Isolíneas . . . 113
5.13. Interfaz de prueba . . . 114

6. Conclusiones y Trabajo a Futuro **117**
6.1. Conclusiones . . . 117
6.2. Trabajo a futuro . . . 118

A. Sistema Visual Humano **121**
A.1. La Luz 121
A.2. Sistema Visual Humano 123
A.2.1. Estructura del ojo humano 123
A.2.2. Propiedades del Sistema Visual Humano 125

B. Fundamentos de Color **131**
B.1. Modelo de color RGB 133
B.2. Modelo de color CMY 135
B.3. Modelo de color HSI 135
B.4. Conversión de RGB a HSI 137
B.5. Conversión de HSI a RGB 137

C. Acrónimos **139**

D. Glosario **141**

Bibliografía **143**

Índice de figuras

1.1. Estructura y alcance del proyecto SNYM. 8

3.1. Diseño conceptual del sistema SNYM 16
3.2. Arquitectura del simulador numérico de yacimientos 18
3.3. Diagrama de clases del módulo de visualización 3D 21

4.1. Esquema de la representación de una celda ortogonal. 49
4.2. Numeración de líneas. 50
4.3. Curva de saturaciones de una celda. 55
4.4. Esquema de la representación de una malla Radial. 57
4.5. Representación de las divisiones internas de las caras 1 y 3. 58
4.6. Numeración de líneas de una celda radial. 59
4.7. Radios fronteras en una rebanada de una malla radial. 66
4.8. Uso de senos y cosenos 66
4.9. Proceso de construcción de una esfera. 80
4.10. Corte de una celda ortogonal mediante el uso de un plano. 81
4.11. Construcción de Isolíneas. 91
4.12. Vecinas de una celda determinada. 95
4.13. Estructura del arreglo que contiene los datos de una propiedad. 97

5.1. Esfera de referencia para las rotaciones. 100
5.2. Translación y cambio de escala. 101
5.3. Vistas predeterminadas. 101
5.4. Selección de celda. 102
5.5. Líneas ocultas y visibles. Pozos ocultos y visibles. 102
5.6. Modelos de color disponibles. 103
5.7. Cambio de los colores mínimo y máximo. Cambio de los colores ternarios. . 104
5.8. Transparencia. 104
5.9. Exploración de celdas internas. 105
5.10. Modificación de la altura de las celdas. 106
5.11. Degradado de colores. 107

5.12. Opciones de pozos. 108
5.13. Opción de grosor de pozos. 108
5.14. Burbujas. 109
5.15. Representación ideal del medio fracturado. 110
5.16. Curva de saturaciones. 111
5.17. Cortes por medio de un plano. 112
5.18. Cortes por pozo. 113
5.19. Isolíneas. 114
5.20. Interfaz de prueba. 115

A.1. Espectro electromagnético 122
A.2. Longitud de onda 122
A.3. Estructura del ojo humano 123
A.4. Distribución de conos y bastones sobre la retina 125
A.5. Curvas de respuesta relativa de los conos 126
A.6. Respuesta del ojo a la intensidad de la luz 126
A.7. a)Saltos iguales en escala de gris b) Intensidad de los saltos 127
A.8. a)Pasos de acuerdo a la respuesta logarítmica del ojo. b) Intensidad de los saltos 128
A.9. Contraste simultaneo 128
A.10.Efecto de las bandas de Mach a) Pasos de acuerdo a la respuesta logaritmica del ojo. b) Intensidad de los pasos c) Brillo percibido en los pasos de la imagen 129
A.11.Patrón ascendente de ondas 130

B.1. Funciones CIE de correspondencia de color. 132
B.2. Diagrama de cromaticidad CIE 133
B.3. Representación del modelo de color RGB 134
B.4. Representación del modelo de color HSI 136

Índice de secuencias

3.1. Crear Malla. 23
3.2. Líneas visibles. 24
3.3. Rotar. 24
3.4. Escalar. 25
3.5. Trasladar. 25
3.6. Estado Visual Inicial. 26
3.7. Vistas. 26
3.8. Modelo de Color. 27
3.9. Cambio Color Mínimo y Máximo. 27
3.10. Seleccionar propiedad. 28
3.11. Seleccionar Medio. 28
3.12. Seleccionar paso de simulación. 29
3.13. Activar Transparencia. 29
3.14. Modificar Nivel de Transparencia. 30
3.15. Seleccionar una Celda. 30
3.16. Exploración de Celdas Internas. 31
3.17. Modificar la Altura de las Celdas. 32
3.18. Interpolación y Degradado de Colores. 33
3.19. Mostrar y Ocultar Pozos. 33
3.20. Mostrar/Ocultar Nombres de Pozos. 34
3.21. Modificar Nombres de Pozos. 34
3.22. Modificar Grosor de Pozos. 35
3.23. Burbujas (Información de Pozos). 36
3.24. Representación ideal de los medios. 37
3.25. Curva de Saturaciones. 38
3.26. Cortes con Plano. 39
3.27. Mover Plano de Cortes. 40
3.28. Cortes por Pozos. 41
3.29. Invertir Dirección del Plano de Corte por Pozos. 42
3.30. Isolíneas. 43

3.31. PaintGL (equivalente a updateGL()). 44
3.32. displayMesh(). 45
3.33. paintMesh(). 46

Resumen

Un Simulador Numérico de Yacimientos (SNY) combina la física, la matemática y la ingeniería de yacimientos en una herramienta computacional, que es capaz de predecir el comportamiento de un yacimiento de hidrocarburos bajo diferentes condiciones de explotación.

Un SNY genera una gran cantidad de información que debe ser analizada por el especialista. La representación visual de esta información, ofrece una forma de analizar los resultados de una forma rapida y fiable, para la mejor toma de decisiones técnicas por parte del especialista.

Tomando en cuenta las caracteristicas de la información generada por un SNY, la visualización tridimensional del yacimiento es la opción más natural para la representación de la información.

El objetivo del proyecto es la construcción de un simulador numérico considerando los mejores algoritmos publicados en el campo del flujo de fluidos en medios porosos y las mejores técnicas de visualización.

El módulo de visualización tridimensional es el componente que permite el despliegue gráfico de la representación del yacimiento. Todo el tratamiento de la información para su adecuado análisis es manejado por este componente.

Este reporte se encuentra dividido siguiendo el esquema de proceso de desarrollo de cualquier software, pero con el debido ajuste para una fácil lectura y comprensión y con la finalidad de colocar solo la información necesaria, sin caer en redundancia de información.

El capítulo 1 ofrece un fondo de referencia para la adecuada comprensión del resto de los capítulos.

En el capítulo 2 se exponen las necesidades funcionales del módulo desarrollado.

El capítulo 3 inicia mostrando el modelo general del proyecto y posteriormente se centra en el modelado del módulo de visualización.

En el capítulo 4 se explica el diseño detallado y la implementación de los principales algoritmos del módulo.

En el capítulo 5 se muestran los resultados ordenados por algoritmos.

Finalmente, en el capítulo 6 se dan las conclusiones.

Capítulo 1

Introducción

Este trabajo consiste en un reporte de las actividades realizadas en el Grupo de Simulación Numérica de Yacimientos (**GSNY**), en el periodo de septiembre del 2004 a noviembre del 2006. El GSNY fue formado a partir del convenio realizado por PEMEX y la Facultad de Ingeniería de la UNAM, para el desarrollo del Simulador Numérico de Yacimientos Multipropósito (**SNYM**).

A continuación se iniciará con una breve descripción del proyecto, se proporcionará el objetivo del proyecto y se verán los fundamentos de lo que es la simulación de yacimientos, con el objetivo de que el lector se familiarice con el tema sobre el que se basa el trabajo realizado.

1.1. Descripción del Proyecto

El proyecto consiste en la construcción de un simulador numérico multipropósitos de uso intensivo para yacimientos de hidrocarburos, con capacidades de pre y post procesamiento de información, satisfaciendo necesidades no cubiertas por los simuladores comerciales actuales.

El proyecto total estará conformado por los siguientes componentes.

1. Caracterización de Fluidos.
2. Modelación Pozo-Yacimiento.
3. Generación de Mallas Numéricas.
4. Simulador de yacimientos.
5. Ajuste de Historia.
6. Análisis económico.

Estos componentes deberán diseñarse con la finalidad de alcanzar un alto grado de portabilidad, modularidad, eficiencia, extensibilidad, mantenimiento y una óptima integración con los demás componentes. Produciendo de esta forma un producto de alta calidad. Para el desarrollo se utilizarán metodologías de ingeniería de software, tecnologías y herramientas de efectividad reconocida.

El proyecto contará con las siguientes características.

- Capacidad de trabajar en un entorno de cómputo distribuido incluyendo algoritmos paralelizados y Descomposición de Dominio.
- Capacidad de trabajar en multiplataforma.
- Mantenimiento y actualización perfectibles.

El proyecto utiliza los siguientes lenguajes de programación:

Fortran 90: Utilizado para el simulador numérico y otros componentes de cálculo numérico.

C++: Utilizado para las interfaces y la visualización.

El proyecto utiliza las siguientes herramientas:

Qt: Qt es una biblioteca multiplataforma para desarrollar interfaces gráficas de usuario. [QT_Producto]

OpenGL: OpenGL estrictamente hablando es una interfaz de software a hardware gráfico. En esencia, es una biblioteca de modelado y de gráficos 3D que es altamente portable y muy rápida [Wright, 2005], [OpenGL].

1.1.1. Objetivo

El principal objetivo del proyecto es la construcción de un simulador numérico considerando los mejores algoritmos publicados en el campo del flujo de fluidos en medios porosos y las mejores técnicas de visualización.

Las características principales del simulador son las siguientes: flujo isotérmico de aceite gas y agua, discretización en diferencias finitas en tres dimensiones en coordenadas radiales y cartesianas, equilibrio inicial gravitacional y capilar, y formulación totalmente implícita.

1.2. Fundamentos

Los modelos son usados para describir procesos que suceden en todas las ramas de la ciencia y tecnología. En un modelo matemático, el sistema o fenómeno a ser modelado es expresado en términos de ecuaciones. Este es el caso del flujo de fluidos en los yacimientos petroleros, cuando se desea modelar matemáticamente el comportamiento a escala de campo. Para realizar este análisis se debe recurrir a un simulador numérico de yacimientos.

Los primeros simuladores de yacimientos fueron construidos como herramientas de diagnóstico para entender a los yacimientos que sorprendían a los ingenieros, debido a que no se comportaban como tradicionalmente lo hacían. Estos consistían de modelos físicos, similares a una pecera llena de arena, donde se representaba el flujo del fluido. Los primeros modelos y ecuaciones fueron documentados en los años 30 y algunas cosas no han cambiado desde entonces. Los simuladores actuales, generalmente resuelven las mismas ecuaciones estudiadas en los 30's, balance de materiales y la Ley de Darcy [1]. En los años 60, con la llegada de las computadoras digitales, algunas cosas cambiaron dramáticamente, el modelado de yacimientos avanzó de peceras de arena a simuladores numéricos. En los simuladores numéricos, los yacimientos son representados por una serie de bloques o celdas interconectadas y el flujo entre celdas es resuelto numéricamente. Al inicio, las computadoras tenían poca memoria por lo que se requería una simplificación de los modelos y una pequeña cantidad de datos de entrada. Conforme el poder de cómputo aumentaba, los ingenieros crearon modelos más grandes y geológicamente más realistas, modelos que requerían una gran cantidad de datos de entrada. El incremento en los modelos ha sido seguido por la creación de programas de simulación eficientes y cada vez más complejos, conjuntamente con interfaces de usuario amigables y paquetes de análisis de resultados.

En la actualidad la simulación numérica ha llegado a ser una herramienta de administración de yacimientos. El proceso de la administración de un yacimiento o un activo petrolero integra los siguientes puntos:

1. Adquisición de información.
2. Validación de información.
3. Integración de la información en un modelo de yacimiento.
4. Comportamiento del modelo de yacimiento con un simulador numérico de yacimientos.
5. Calibración del modelo del yacimiento (ajuste de historia).

[1] La ley de Darcy declara que la velocidad del flujo del fluido es proporcional al gradiente de la presión y la permeabilidad, y es inversamente proporcional a la viscosidad.

6. Acoplamiento del modelo del yacimiento con las instalaciones superficiales.

7. Realización de pronósticos de producción.

De los siete puntos anteriores, la simulación numérica de yacimientos es la herramienta fundamental para la realización de los cinco últimos. De aquí la importancia de contar con un simulador numérico de yacimientos propio para la administración efectiva de los mismos.

1.2.1. Motivación para la simulación

Un simulador numérico es una herramienta para predecir el comportamiento de un yacimiento, y en las manos del ingeniero experto puede reproducir el comportamiento de un yacimiento. Un simulador puede predecir la producción bajo condiciones de operaciones actuales, las reacciones del yacimiento a los cambios en las condiciones, tal como el incremento en las tasas de producción; la producción de mas o de diferentes pozos; la respuesta a la inyección de agua, vapor, acido o espuma; los efectos de hundimientos; y la producción de pozos horizontales de diferentes longitudes y orientaciones.

La simulación de yacimientos puede ser realizada por ingenieros de yacimientos de compañías petroleras o por contratistas consultores de ingeniería. En ambos casos el simulador es una herramienta que permite a los ingenieros responder preguntas y ofrecer recomendaciones para mejorar prácticas operacionales.

Para realizar pronosticos sensatos es necesario hacerse algunas preguntas, por ejemplo, ¿Donde deberían ser colocados los pozos para maximizar la extracción por peso (o dólar) o la inversión adicional? ¿Cuántos pozos son requeridos para producir gas suficiente para conocer un calendario de liberación de contratos? ¿Se debería extraer el petróleo por producción natural (producción primaria) o por inyección de agua? ¿Cuál es la longitud optima de un pozo horizontal? ¿El dióxido de carbono (CO2) es viable para la inyección? En todos los casos, el estudio de la simulación debería dar como resultado en recomendaciones para la intervención. Esto puede incluir nuevas estrategias para la adquisición de datos o un plan de perforación, localización y dirección de pozos y una estrategia final para cada pozo.

1.2.2. Como trabaja un simulador

El principal objetivo de la simulación de yacimientos es proporcionar al ingeniero de diseño de explotación una herramienta confiable para predecir el comportamiento de los yacimientos de hidrocarburos bajo diferentes condiciones de operación. El modelar el comportamiento de un yacimiento de hidrocarburos bajo diferentes esquemas de producción reduce el riesgo asociado a la elección del plan de explotación y por lo tanto minimiza los flujos de efectivo negativos.

Una vez que los objetivos de la simulación han sido determinados, el próximo paso es describir el yacimiento en términos del volumen de gas o aceite en el lugar, la cantidad que se puede recuperar y la tasa a la cual puede ser recuperado. Para estimar las reservas recuperables se crea un modelo estático a través de esfuerzos combinados de geólogos, geofísicos, petrofísicos e ingenieros de yacimientos.

El simulador calcula el flujo del fluido a través del yacimiento usando las ecuaciones fundamentales de flujo de fluido expresadas en derivadas parciales, estas ecuaciones son obtenidas de las ecuaciones de continuidad, de flujo y de estado. La ecuación de continuidad expresa la conservación de la masa, la ecuación de flujo es la ley de Darcy y la ecuación de estado describe la relación de la presión-volumen o de la presión-densidad de varios fluidos presentes en el yacimiento. Posteriormente estas ecuaciones son escritas en la forma de diferencias finitas, en las cuales el yacimiento es tratado como una colección de bloques numerados y el periodo de producción del yacimiento se divide dentro de un número de pasos de tiempo. Matemáticamente, el problemas es discretizado en el espacio y en el tiempo.

Ejemplos de simuladores comerciales son: CMG, Atos, Éxodos, Sara, Petrel y la familia de simuladores Eclipce.

Todos los simuladores comerciales conciben un yacimiento dividido dentro de un número de bloques individuales llamados bloques de malla o celdas. Cada bloque corresponde a un volumen en el yacimiento y debe de tener propiedades de roca y de fluido representativos del yacimiento en esa región. Algunas propiedades de roca y de fluido son por ejemplo: permeabilidad, saturación, porosidad, presión, viscosidad, compresibilidad, densidad, etc. Tradicionalmente, los bloques de la malla, que llamaremos celdas, son paralelepipedos. Esta configuración asegura que la malla permanece ortogonal y que se acopla a los modelos matemáticos usados en los simuladores.

Sin embargo, este enfoque no hace fácil representar complejas estructuras tal como fallas no verticales o superficies erosionadas. La geometría de punto de esquina resuelve este problema. En una malla de punto de esquina, las esquinas no necesitan ser ortogonales. Las mallas de tres dimensiones son construidas a partir de una malla de dos dimensiones, colocando esta malla en la cima de la superficie del yacimiento y proyectándola verticalmente a lo largo de planos específicos en las capas inferiores. Los programas interactivos para la creación de mallas ayudan a construir mallas complejas eficientemente. En este punto la visualización del modelo tridimensional permite encontrar inconsistencias en los modelos y de este modo poder corregirlas antes de introducir el modelo al simulador. Una vez que la malla ha sido construida, el siguiente paso es asignar propiedades de roca y de fluido a cada celda de la malla. A cada celda se le asigna un único valor para cada propiedad del yacimiento.

Una vez que el modelo se ha terminado, al inicio de la simulación, el simulador requiere condiciones de frontera y condiciones iniciales para el comportamiento del fluido. Entonces para un tiempo dado posterior, conocido como paso de tiempo, el simulador calcula la

nueva distribución de presiones y saturaciones que indican la tasa del flujo del fluido. Este proceso es repetido un número de pasos de tiempo y de esta forma se calcula la historia de la presión y la tasa de flujo para cada celda.

Los resultados de la simulación pueden ser visualizados, permitiendo una rápida evaluación de la simulación ejecutada y proporcionando una comprensión inmediata de la recuperación o explotación del yacimiento y de los procesos físicos ocurridos en el mismo. Como se ha visto, la visualización del modelo en tres dimensiones, es un recurso fiable para revisar los modelos de los yacimientos antes de introducirlos a un simulador, para encontrar inconsistencias en el modelo y corregirlas. Después de la simulación, los resultados pueden ser visualizados, permitiendo una rápida evaluación de los mismos, se puede hacer una rápida comparación de simulaciones y provee una comprensión del comportamiento del yacimiento.

Estas técnicas y programas para cargar datos, calcular simulaciones y visualizar los resultados, permiten a los ingenieros el uso de los simuladores como guías de gestión de yacimientos, para tomar decisiones a lo largo de la vida del yacimiento [Adamson, 1996], [Peña, 2006].

1.3. Motivación para la visualización

Hemos visto como los simuladores han llegado a ser una herramienta de administración de yacimientos, ya que reproducen el comportamiento pasado de un yacimiento y predicen su comportamiento futuro.

También hemos visto que una simulación consiste de varios pasos de tiempo, donde, cada uno de ellos contiene todas las propiedades para cada una de las celdas en toda la malla.

Una simulación se divide en dos partes, en la primera se realiza un ajuste de historia con toda la información disponible del yacimiento hasta el momento de realizar la simulación. La segunda parte es en donde se predice el comportamiento futuro del yacimiento. Además, cabe mencionar, que se debe tener una continua calibración del simulador para que los resultados sean lo más realista posible.

Si tomamos en cuenta que en la actualidad, una simulación promedio real contiene un periodo de simulación del yacimiento que va de los 15 hasta los 30 años, en donde de 5 a 10 años pertenecen a un ajuste de historia (dependiendo de la información histórica del yacimiento) y el resto es la predicción del comportamiento futuro; y en donde cada paso de tiempo es tomado en un periodo de uno o dos meses, tenemos que, por ejemplo, una simulación real consiste, en promedio, de 240 pasos de tiempo. Si aunado a esto tomamos en cuenta que las mallas actuales contienen entre cien mil y un millón de celdas dependiendo de la complejidad del yacimiento y se calculan en promedio 20 propiedades, tenemos que, una simulación real completa contiene en promedio, 240

pasos, 20 propiedades y 500,000 celdas, resultando un total de 2,400,000,000(dos mil cuatrocientos millones) de datos.

Si quisiéramos que un experto analice esta cantidad de información en formato crudo, seria prácticamente imposible para esté, formar un juicio adecuado de las condiciones y el comportamiento del yacimiento.

Aquí es donde comprendemos que el experto necesita de herramientas que le ayuden a evaluar y analizar los datos de la manera más optima posible.
Las graficas bidimensionales proveen una forma básica de conocer el comportamiento promedio de todo el yacimiento, pero no ofrecen información concisa sobre áreas específicas del mismo.
La visualización tridimensional del yacimiento permite una rápida evaluación del comportamiento total del yacimiento, así como de áreas específicas del mismo, además, proporciona una comprensión inmediata de la recuperación o explotación del yacimiento y de los procesos físicos ocurridos dentro de él. La visualización 3D también ofrece al experto la posibilidad de explorar con detenimiento áreas específicas del yacimiento.

Gracias a herramientas de este tipo, los expertos pueden utilizar los simuladores como guías de gestión de yacimientos, y así, tomar decisiones a lo largo de la vida de los mismos.

1.4. Alcance

La Figura 1.1 muestra en forma esquemática la estructura y alcance del proyecto SNYM. Los trabajos hechos en la versión 2004 se muestran en color negro. Las actividades hechas en la versión 2005 se muestran en color rojo. Las actividades realizadas en la versión 2006 se presentan en color azul. Las actividades en color amarillo se encuentran en fase de validación. Las actividades que se realizarán en el futuro se muestran en verde.

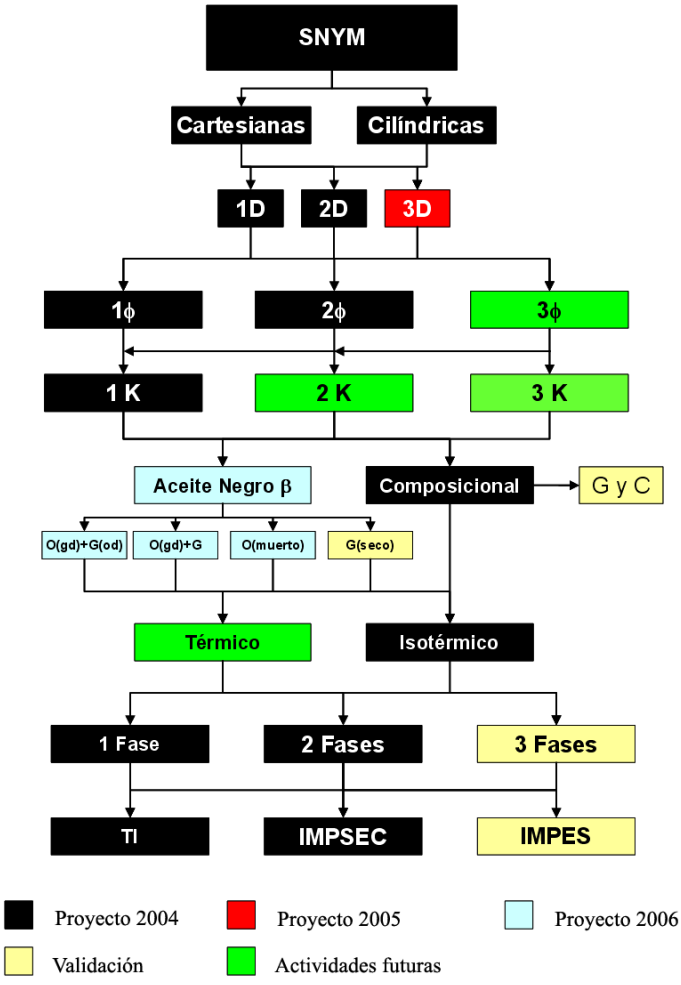


Figura 1.1: Estructura y alcance del proyecto SNYM.

Capítulo 2

Requerimientos de Visualización

No nos extenderemos en explicar porque es importante obtener unos requisitos claros, concretos, no ambiguos, suficientes y estables, baste decir que de los proyectos que fracasan, el 50 % se debe directa o indirectamente a defectos en los requisitos, ya que estos repercuten en todas las fases de desarrollo de un sistema [Bañares, 2004].

Este capítulo se encuentra dividido en dos partes igualmente importantes: los requerimientos de usuario y los requerimientos de software. Con los requerimientos de usuario tenemos una clara idea de quién va a usar el software, para qué lo vaya a usar y en dónde lo va a usar. Con los requerimientos de software sabremos qué es lo que el usuario desea, y a partir de ahí qué necesita para poder ejecutar el sistema.

2.1. Requerimientos de usuario

2.1.1. Objetivo

El propósito del proyecto es la construcción de un simulador considerando los mejores algoritmos publicados en el campo del flujo de fluidos en medios porosos y las mejores técnicas de visualización.

El módulo de visualización tiene como objetivos primordiales:

- El alto desempeño del módulo.
- La optimización de los algoritmos usados.
- La interacción y compatibilidad entre funcionalidades.
- Desarrollo progresivo en los componentes.
- Extensión, actualización y mantenimiento perfectibles.
- Usablilidad, flexibilidad, robustez y eficiencia de uso.

2.1.2. Ambiente

Ambiente de negocios: Se genera una gran cantidad de información que debe ser analizada de forma rápida. La aplicación debe ser familiar al usuario que utilice otro software.

Ambiente físico: Una o varias personas laboran dentro de una oficina.

Ambiente tecnológico: La mayoría de usuarios cuentan con equipos P-IV. Monitores de 17" y 19" con resoluciones de 1024x768 y 1280x800. Conexión a red de hasta 100 Mbps. La mayoría de usuarios cuenta con sistema operativo Windows XP y solo algunos con UNIX o Linux.

2.1.3. Actores

Los actores son ingenieros petroleros especializados en el área de yacimientos.

Las necesidades de los actores son:

- Encontrar fácilmente información.
- Conocer propiedades del yacimiento.
- Poder crear presentaciones de los resultados.
- Ver métricas comparativas.

2.1.4. Necesidades de rendimiento capacidad

Se espera correr mallas de 100x100x20 y con 100 pasos de simulación.

2.2. Requerimientos de software

Se han dividido los requerimientos de software en funcionales y no funcionales, para tener una mejor comprensión de las necesidades del usuario y de los requisitos del sistema.

2.2.1. Requerimientos funcionales

A continuación se listan, de forma priorizada, los requerimientos funcionales únicamente del módulo de visualización.

1. Crear la visualización tridimensional de una malla numérica ortogonal, para la representación de un yacimiento de hidrocarburos.

2. Crear la visualización tridimensional de una malla numérica radial, para la representación de un yacimiento de hidrocarburos.

3. Los datos de la simulación podrán ser cargados de una simulación ya realizada o podrán ser generados al ejecutar una simulación.

4. Cuando los datos sean cargados de un archivo, la malla se colocará en un estado inicial con los datos del primer paso.

5. Cuando los datos sean generados en una simulación en tiempo real, la malla inicial se encontrará vacía, solo indicando las divisiones de cada celda. A cada paso simulado se actualizará la visualización con la información del paso calculado. El usuario podrá revisar cualquier paso ya simulado al mismo tiempo que se calcula el siguiente paso de la simulación.

6. La malla podrá contener celdas activas e inactivas, las últimas no deberán mostrarse en la simulación. El usuario podrá configurar las celdas activas e inactivas.

7. Cada celda esta separada de sus vecinas mediante líneas, el usuario podrá mostrar u ocultar estas líneas.

8. El usuario podrá interactuar con la malla mediante rotaciones, traslaciones y escalas.

9. El sistema contará con un estado visual inicial de la malla respecto a la rotación, la traslación y la escala. Se podrá volver al estado visual inicial en cualquier momento.

10. La visualización deberá contar con 6 vistas predefinidas de la malla y el usuario podrá elegir cualquiera de ellas. Las vistas mostrarán la malla desde la parte inferior, superior, izquierda, derecha, frontal y trasera. El sistema deberá recordar el estado de la malla antes de cambiar a alguna vista y se deberá poder regresar a el.

11. Se proveerá de un indicador tipo ejes, que muestre la posición de la malla en todo momento.

12. Se representarán los valores de cada celda mediante colores. Se deberán definir tres esquemas para realizar el mapeo de colores. Uno usara el modelo de color RGB, otro usara el modelo HSI y otro usara un modelo basado en tres colores elegidos por el usuario. El usuario podrá elegir cualquiera de los tres esquemas.

13. Se deberá proveer del indicador necesario para mostrar el mapeo de los datos en colores.

14. Para cada propiedad, los colores mínimo y máximo pueden ser modificados por el usuario. El sistema deberá recordar los colores colocados para cada propiedad.

15. El usuario podrá elegir en cualquier momento la propiedad mostrada por la visualización.

16. El usuario podrá elegir el medio mostrado por la visualización.

17. El usuario podrá elegir el paso de simulación que la visualización mostrara.

18. La malla contará con transparencia, y el usuario podrá activarla cuando lo desee, además podrá modificar el nivel de la misma.

19. Se deberá contar con una forma de mostrar los datos de una celda en particular.

20. Se deberán poder explorar las celdas internas de la malla.

21. La altura de las celdas podrá ser modificada a petición del usuario. Se deberá poder regresar a la altura original de una forma sencilla.

22. Se requiere de un degradado de colores en la malla basado en la interpolación de datos de una celda con los datos de sus vecinas.

23. La visualización mostrará los pozos mediante tubos. El pozo se mostrará con un color determinado dependiendo del tipo de pozo que sea.

24. El pozo contará con un indicador a modo de flecha, que mostrará si es inyector o extractor según la flecha se encuentre hacia abajo o hacia arriba correspondientemente.

25. El pozo podrá mostrar una etiqueta con su nombre, esta podrá hacerse visible o no a petición del usuario. El nombre del pozo también deberá poder modificarse.

26. El grosor del pozo depende del grosor real del mismo, pero el usuario podrá escalarlo para una mejor apreciación. Se deberá poder regresar al grosor inicial del pozo.

27. El usuario podrá hacer visibles e invisibles los pozos y todos sus atributos cuando lo desee.

28. Las propiedades y el comportamiento del pozo se mostrarán mediante esferas que se encontraran sobre el mismo. El usuario podrá elegir mostrarlas o no, además de que podrá seleccionar las propiedades que desea ver. El color de las esferas dependerá de la propiedad que estas muestren. El usuario podrá elegir mostrar u ocultar un cuadro de información de las esferas. El usuario podrá mostrar u ocultar la etiqueta de las esferas.

29. Cuando la simulación cuente con más de un medio, se deberá poder mostrar la representación ideal de los medios de una celda seleccionada por el usuario.

30. Se mostrará la curva de saturaciones sobre una pila de celdas. Las celdas serán seleccionadas por el usuario y la pila con la curva de saturaciones se mostrara por encima de la malla. El usuario podrá seleccionar todas las pilas de celdas de la malla.

31. Se podrán realizar cortes a la malla mediante el uso de un plano, la malla solo mostrará lo que quede de cierto lado del plano. El plano de corte podrá ser rotado por el usuario.

32. Se podrán realizar cortes a la malla mediante el uso de los pozos. Cada vez que se seleccionen dos pozos diferentes se creara un corte que pase exactamente por los pozos seleccionados. Los cortes serán acumulativos. Se proveerá de una forma de invertir la dirección del plano, de modo que el corte quede a la inversa, esto es, la parte visible será eliminada y la parte que antes no se encontraba será la visible.

33. Se podrán generar las isolíneas de una capa de la malla. El usuario podrá elegir la capa.

2.2.2. Requerimientos no funcionales

A continuación se proporcionaran los requerimientos no funcionales más importantes, desde el punto de vista del módulo de visualización 3D.

Requerimientos de usabilidad

La interfaz será tan familiar como sea posible, se seguirán guías UI. Se deberá contar con ayuda en línea o manual de usuario y se incluirán principios de interacción instructiva.

Requerimientos de desempeño y escalabilidad

El sistema debe ser ejecutado en tiempo real, es decir, a cada paso de simulación se debe mostrar la información calculada.

El tiempo de respuesta debe ser muy breve para el manejo de la malla en la visualización.

El sistema debe poder ser fácilmente escalado, con un fácil progreso de los componentes y una fácil inclusión de nuevas funcionalidades.

Requerimientos de mantenimiento y actualización

El sistema deberá poder ser desarrollado progresivamente. La localización y corrección de errores deberá de ser eficiente, rápida y manejable.

Requerimientos ambientales

El sistema deberá ser ejecutado en sistemas Windows 2000 o superior, y sistemas UNIX. El hardware deberá ser Pentium III o superior, con 512 Mb ram o superior con 20Mb libres en disco duro para el sistema. Se necesita la biblioteca de Qt, provista por el sistema. Se necesita de OpenGL en el sistema.

Capítulo 3

Modelado UML

En este capítulo se describirá el módulo en varios niveles de abstracción, iniciando con el diseño conceptual de todo el sistema, y descendiendo hasta el módulo de visualización, el cual se expondrá a detalle. Para la descripción se usará **UML** [1] por ser un lenguaje de modelado estándar en la industria de desarrollo de software.

3.1. Diseño conceptual

En el Diseño Conceptual que se muestra en la Figura 3.1, elaborado por la administración de desarrollo, se observan los módulos que conformarán el sistema **SNYM** (Simulador Numérico de Yacimientos Multipropósito), la etapa del proceso de simulación a la que pertenecen y el flujo de información entre ellos.

A continuación se provee una breve explicación de cada Componente del Sistema, proporcionada por la Administración de Desarrollo del sistema SNYM.

3.1.1. Módulo de Caracterización de Fluidos (PVT)

Aplicación encargada de la caracterización del Hidrocarburo, incluye en sus funcionalidades el manejo de componentes conocidos, el cálculo de las propiedades de pseudocomponentes, la reproducción de experimentos PVT y el ajuste a la ecuación de estado. Sus resultados se relacionan directamente con la modelación de pozos, así como con la simulación numérica de yacimientos.

[1] UML (Unified Modeling Language - Lenguaje de modelado unificado) es un lenguaje que permite modelar, construir y documentar los elementos que forman un sistema de software. Se ha convertido en el estándar de facto de la industria, debido en parte, a que ha sido concebido por los autores de los tres métodos más usados en la ingeniería de software: Grady Booch, Ivar Jacobson y Jim Rumbaugh, y en parte en que forma una herramienta compartida entre todos los ingenieros de software, ya que incorpora las principales ventajas de cada uno de los métodos en los que se basa. [Bañares, 2004], [Booch, 1999], [UML], [AgilMod]

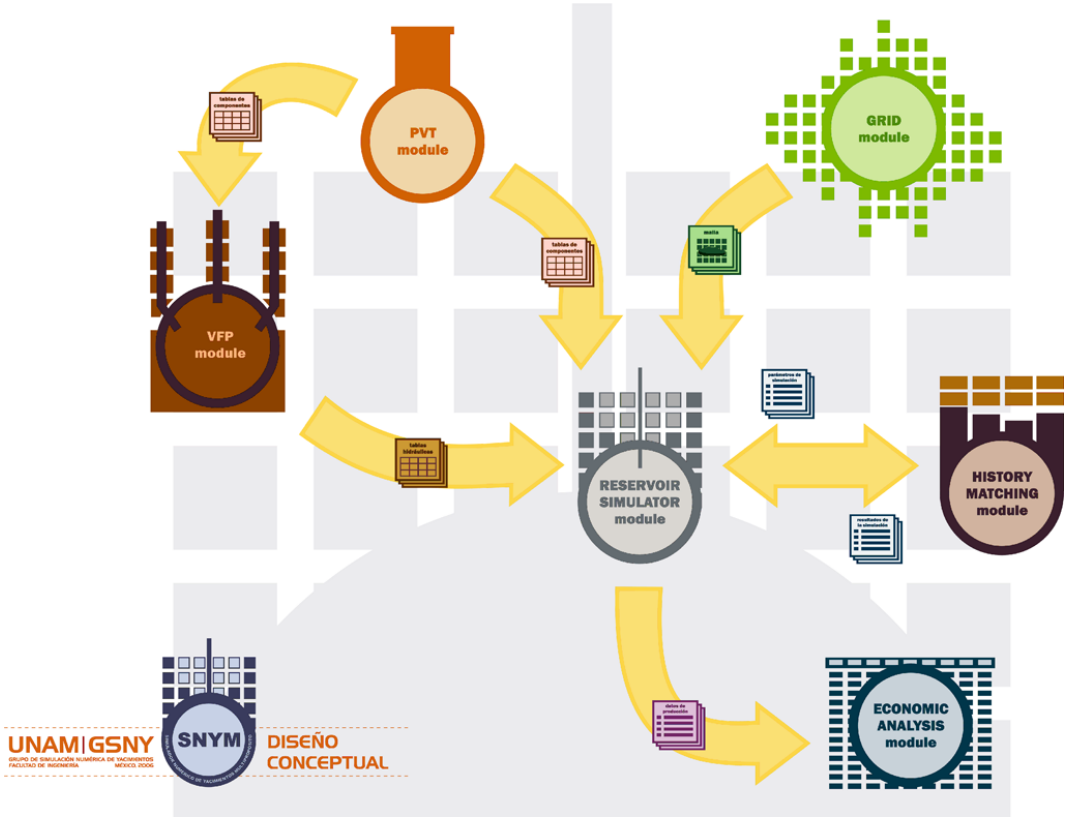


Figura 3.1: Diseño conceptual del sistema SNYM

3.1.2. Módulo de Mallas Numéricas (GRID)

El Módulo de Mallas Numéricas se encargará de la aplicación de algoritmos algebraicos, de las técnicas matemáticas más avanzadas y de la implementación de los algoritmos mas eficiente para la generación de mallas, cuyas características sean propicias para la simulación numérica de yacimientos.

3.1.3. VFP y Modelación de Pozos-Yacimientos (VFP)

Esta aplicación se encargará de la modelación matemática de la dinámica de distintos tipos de pozos bajo condiciones de explotación, con el fin de generar tablas hidráulicas utilizadas en los procesos de simulación de yacimientos. Adicionalmente contará con capacidades de ajuste para correlaciones PVT y de Flujo Multifásico.

3.1.4. Simulador de Yacimientos (Reservoir Simulator)

Módulo central del proyecto encargado de simular la dinámica de yacimientos de hidrocarburos. Esta aplicación estará capacitada para simular los distintos modelos de yacimientos, como son el composicional, de aceite negro y el de gas, utilizando motores numéricos especializados. En esta aplicación se concibe además el desarrollo de motores numéricos paralelos utilizando técnicas de cómputo intensivo para computadoras de alto rendimiento.

3.1.5. Ajuste de Historia (History Matching)

Siendo la predicción de comportamientos futuros parte esencial de la explotación de yacimientos, el proceso de ajuste de historia resulta de gran importancia ya que permite adecuar los parámetros de operación del simulador al comportamiento histórico del yacimiento, logrando así una mayor capacidad de predicción. Los procesos necesarios para realizar tal ajuste, serán cubiertos por este módulo.

3.1.6. Análisis Económico (Economic Analysis)

Con el fin de contar con un sistema completo, los aspectos económicos no se dejarán de lado, por lo que se desarrollará una aplicación especializada para tales propósitos, que permita realizar con eficiencia los análisis necesarios que coadyuven en la toma de decisiones de tal forma que éstas resulten más acertadas.

3.2. Arquitectura del Simulador Numérico de Yacimientos Multipropósito

La arquitectura del Simulador Numérico de Yacimientos Multipropósito, diseñada por la Administración de Desarrollo, se ha modelado siguiendo una estructura modular que agrupa las tareas según su propósito, dando como resultado la constitución de componentes independientes. La independencia de componentes es básica para la adecuada evolución de un sistema de estas dimensiones, además es indispensable para el correcto manejo de errores, para facilitar el mantenimiento del software y para gestionar cambios del mismo.

La relación que guardan estos componentes puede ser observada en la Figura 3.2, imagen proporcionada por la Administración de Desarrollo.

A continuación se provee una breve explicación de cada Componente del Simulador de Yacimientos, proporcionada por la Administración de Desarrollo del sistema SNYM.

3.2.1. Shared Memory

Este componente es el encargado de la gestión de todas las estructuras de memoria necesarias para el almacenamiento de parámetros de operación y de resultados arrojados

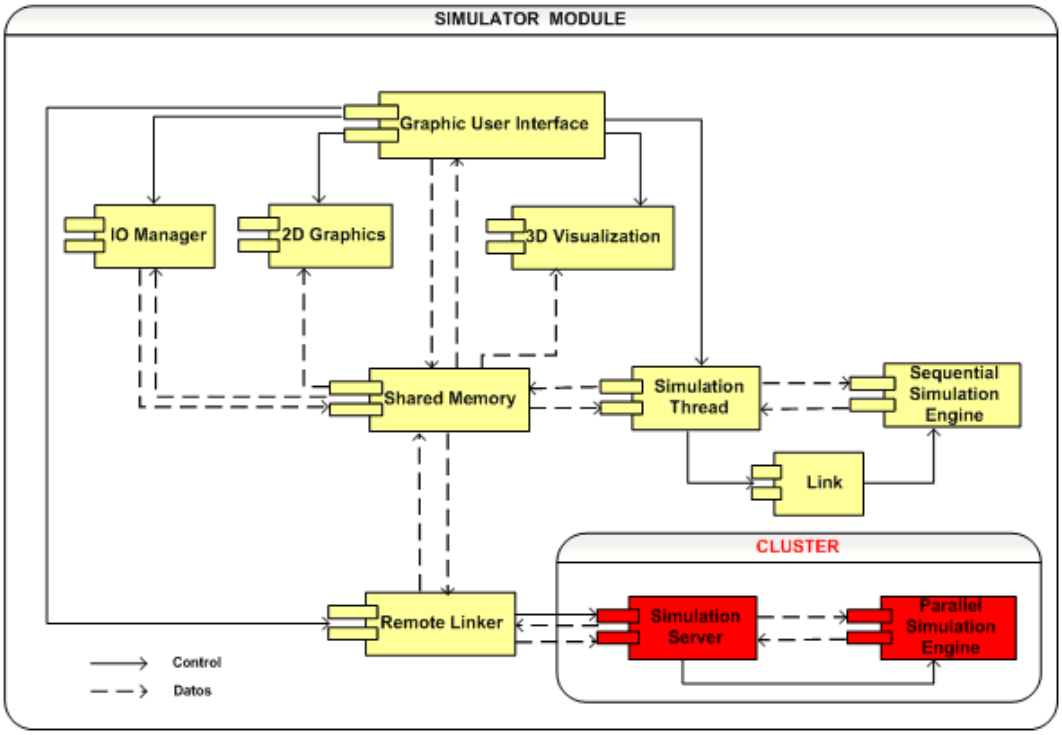


Figura 3.2: Arquitectura del simulador numérico de yacimientos

por el simulador numérico, y que son compartidas entre los demás componentes de la aplicación. Además de la creación, alojamiento y destrucción, también provee de métodos de acceso a tales estructuras. Además incorpora variables de estado compartidas, a través de las cuales se puede determinar en todo momento las acciones realizadas por el usuario.

3.2.2. Simulation Thread

Este componente tiene dos funciones principales, por un lado sirve de enlace entre el motor de simulación y el resto de la aplicación, y por el otro, se encarga de gestionar las estructuras internas necesarias para que la simulación se ejecute de manera concurrente. El enlace consiste en comunicar, a través de parámetros explícitos, las estructuras donde se almacenan todos los parámetros de operación del simulador, así como aquellas donde se almacenarán los resultados. Este modelo de integración es necesario tanto por las distintas plataformas de desarrollo, como por los paradigmas de programación utilizados: orientada a objetos en la aplicación y estructurada en las subrutinas de cálculo.

3.2.3. Sequential Simulation Engine

Conjunto de algoritmos de cálculo, implementados en Fortran 95, encargados de la modelación matemática de los procesos involucrados en la simulación numérica de yacimientos y su dinámica a través de cálculos numéricos intensivos.

3.2.4. Link

Este componente sirve de enlace entre las subrutinas de cálculo numérico y el resto de la aplicación. Por cuestiones de modelado se representa como un componente externo a Simulation Engine y a Simulation Thread, sin embargo está concebido como el conjunto de métodos y funciones necesarios para enlazar las subrutinas de cálculo con los demás componentes de la aplicación, conjunto que se encuentra distribuido tanto en los métodos del encapsulador como de las subrutinas escritas en Fortran diseñadas para tal propósito.

3.2.5. IOManager

Componente encargado de realizar las operaciones de lectura/escritura de archivos de parámetros y de resultados que conforman un proyecto. La gestión de los archivos, la estructuración de la información, así como la verificación de su integridad son tareas de este componente.

3.2.6. 2D Graphics

Para efecto de los análisis de resultados, la aplicación provee de gráficas bidimensionales en las que se muestran resultados. La creación de estas gráficas y el tratamiento de la información para las mismas se realizan en este componente.

3.2.7. 3D Visualization

Componente que permite el despliegue gráfico de la representación tridimensional del yacimiento, así como los mecanismos necesarios para controlar su aspecto.

El despliegue y las complejas operaciones para su control se realizan por medio del componente de visualización. Todo el tratamiento de la información para su adecuado análisis es manejado por este componente.

3.2.8. Graphic User Interface

Finalmente el componente que integra, coordina y controla a todos los anteriores, dependiendo de las acciones del usuario es la interfaz gráfica. Si bien sus funciones principales son la comunicación de la aplicación con el usuario, gestionar el entorno gráfico y atender y responder a las acciones del usuario, este componente también se encarga de controlar

el flujo de ejecución de la aplicación, así como de la validación de la consistencia entre la información del proyecto y de los procesos involucrados en su generación.

3.3. Diseño del módulo de Visualización 3D

Ahora que poseemos la información necesario sobre la cual se desarrolla el módulo de visualización, podremos comprender mejor las decisiones tomadas en el modelado del módulo.

Para iniciar se mostrará el diagrama de clases contenido en el único paquete del módulo. Este paquete contiene todas las clases dentro del módulo, esto se observa fácilmente en la Figura 3.3. El diagrama de clases muestra la interacción entre las diferentes clases del módulo. La clase principal del módulo es la clase Conector, esta clase es la encargada de recibir y gestionar todas las señales externas al módulo, por medio de esta clase se tiene acceso a todos los métodos y acciones contenidos en el módulo.

A continuación se proveerá de una breve explicación de cada clase del módulo de Visualización 3D. La explicación será muy breve, pues en el siguiente capítulo se detallara la funcionalidad y los algoritmos involucrados en cada clase. Esta información solo es necesaria para una correcta comprensión de los diagramas de secuencia mostrados en el presente capítulo.

Conector

Es la clase encargada de gestionar todas las peticiones externas, ya sea directamente del usuario o a través de la interfaz. Controla toda la información entre el módulo y los módulos externos.

VisMesh

Clase que define las propiedades de una malla genérica, sirviendo como base para las clases que heredan de ella.

VidMeshO

Clase que define las propiedades de una malla ortogonal.

VisMeshR

Clase que define las propiedades de una malla radial.

ManagerProp

Clase encargada de manejar todos los datos de las propiedades de la malla y de realizar los cálculos que se relacionen con estas propiedades. Además contiene variables relacionadas con la simulación actual.

ColorLUT

Clase encargada de gestionar los colores que utilizara la malla en el mapeo de las propiedades a colores.

MapColor

Clase que contiene los algoritmos necesarios para pasar de un modelo de color a otro.

TrackBall

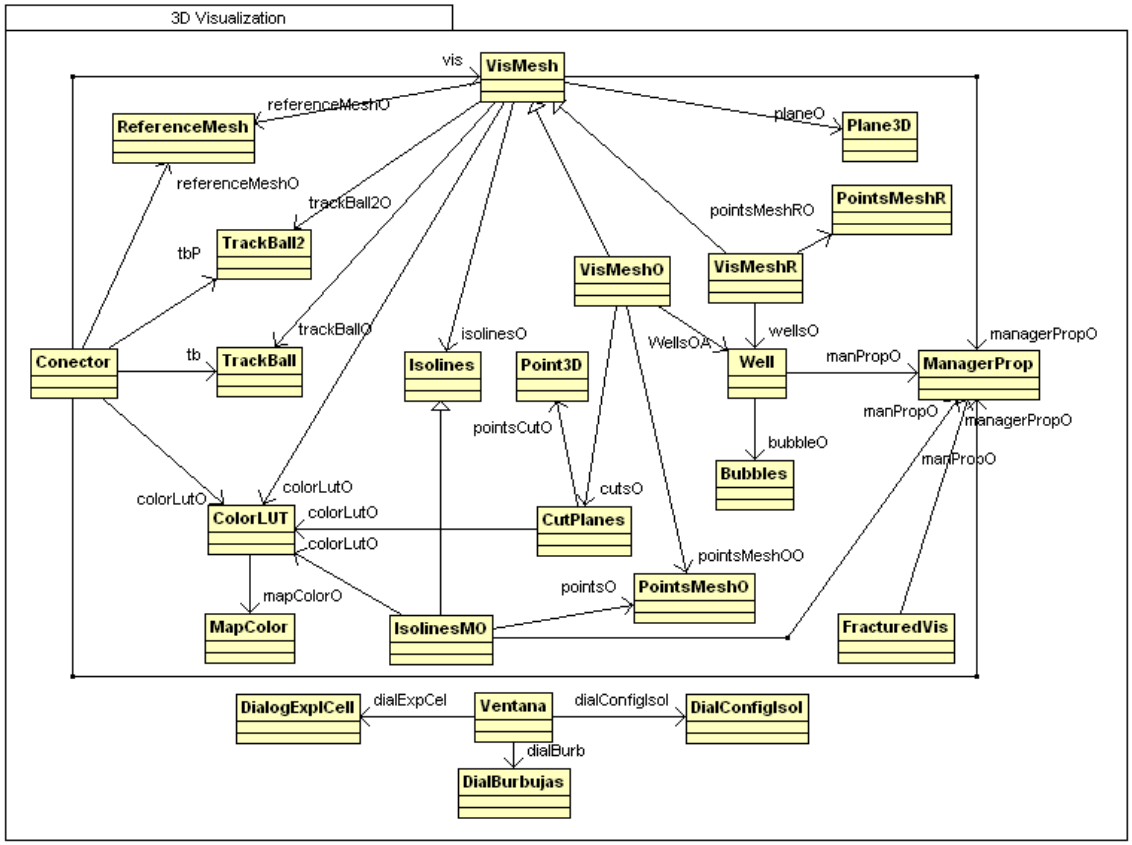


Figura 3.3: Diagrama de clases del módulo de visualización 3D

Clase encargada de las rotaciones de la malla.

TrackBall2

Clase encargada de las rotaciones de los planos de corte.

ReferenceMesh

Contiene una malla lógica de referencia y los algoritmos necesarios para producir cortes por bloques y explorar la malla interna.

PointsMeshO

Clase que contiene las coordenadas de cada punto de cada celda de una malla ortogonal. Además de los algoritmos necesarios para crear estas coordenadas y para el manejo de las mismas.

PointsMeshR

Clase que contiene las coordenadas de cada punto de cada celda de una malla radial. Además de los algoritmos necesarios para crear estas coordenadas y para el manejo de las

mismas.

Isolines

Interfaz que define las funciones para generar isolíneas.

IsolinesMO

Clase que contiene los algoritmos necesarios para generar las isolíneas de una malla ortogonal.

Point3D

Clase que define un punto en tres dimensiones y los algoritmos para su uso y manejo.

Plane3D

Clase que define un plano en tres dimensiones, la ecuación que lo gobierna y algoritmos para su uso. Además contiene algoritmos de distancia de un punto al plano.

CutPlanes

Clase que define algoritmos de cálculo para el corte de una celda ortogonal por medio de un plano euclidiano.

Well

Clase que define un pozo, sus características y su manejo, además se encarga de visualizar el pozo y sus propiedades.

Bubbles

Clase que define las propiedades de una esfera o burbuja y que se encarga de visualizarla.

FracturedVis

Clase encargada de crear y visualizar la representación ideal del medio fracturado.

Ventana

Clase que crea una ventana necesaria para probar todos los elementos y características del módulo.

DialogExplCell

Clase que crea un dialogo usado para probar los algoritmos de la exploración de las celdas internas.

DialBurbujas

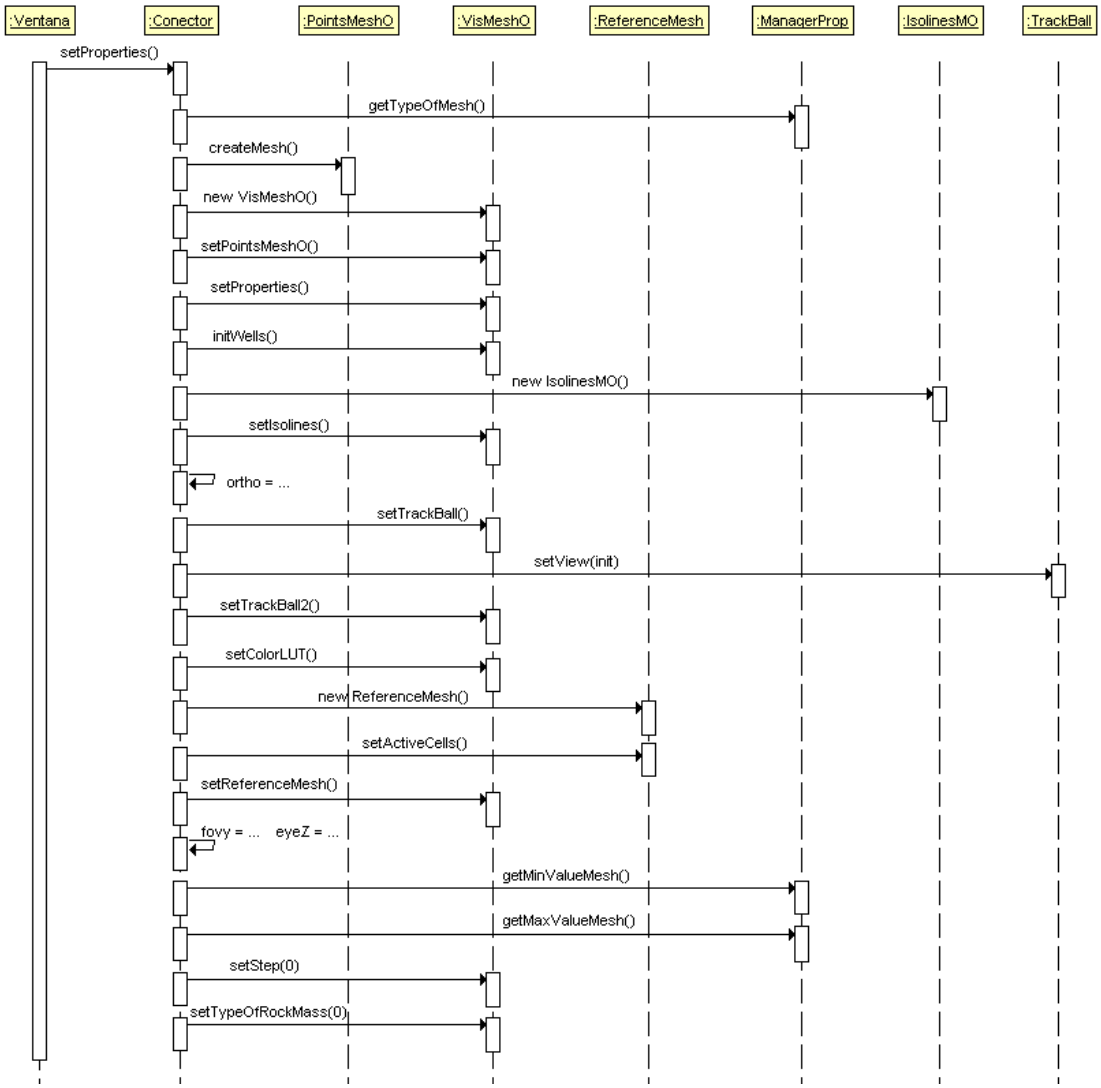
Clase que crea un dialogo usado para probar los algoritmos de la funcionalidad de burbujas.

DialConfigIsol

Clase que crea un dialogo usado para configurar y probar los algoritmos de isolíneas.

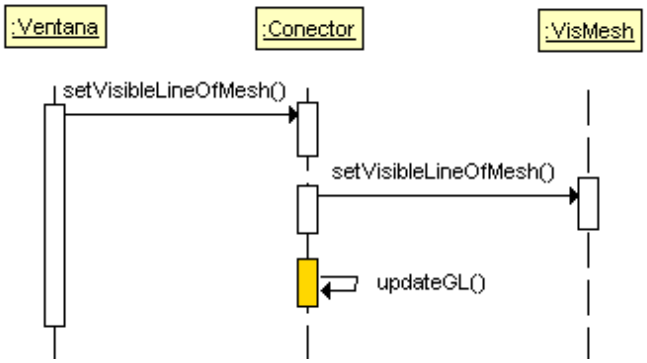
3.3.1. Diagramas de secuencia

A continuación se exponen los diagramas de secuencia de cada una de las funcionalidades del sistema, cada diagrama pretende mostrar la solución a un requerimiento funcional. Aunado al diagrama de secuencia se proporciona una explicación del diagrama a modo de resumir el caso de uso contenido en el diagrama.



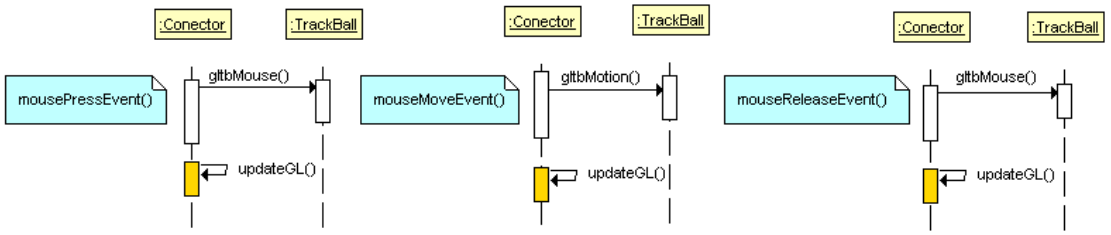
Ya sea que el usuario inicie una nueva simulación o cargué una preestablecida, la interfaz le indica a la visualización que hay una nueva simulación mediante el uso de la función setProperties(), inmediatamente después, la visualización accede al tipo de malla y crea una malla del tipo adecuado e inicializa todo lo necesario para tener una visualización completa y correcta. En este diagrama se muestra la creación de una malla ortogonal, pero el proceso para una malla radial es exactamente el mismo.

Secuencia 3.1: Crear Malla.



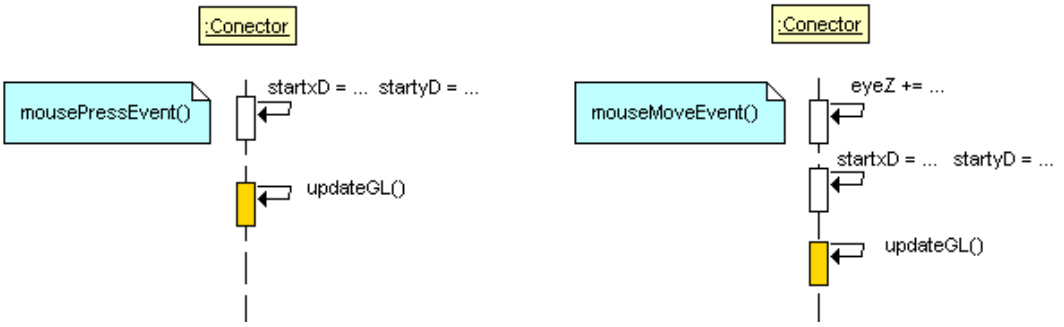
El usuario utiliza el control correspondiente en la interfaz de usuario, esta ejecuta la función setVisibleLineOfMesh() de la clase conector, este a su vez hace la llamada correspondiente a la clase VisMesh y esta modifica la variable correspondiente. Al final se repinta la visualización con el cambio realizado. La línea de vida de la función updateGL() se encuentra de un color diferente indicando que esa función se encuentra de forma expandida en otro diagrama, el diagrama correspondiente muestra una etiqueta del mismo color con el nombre de la función en la esquina superior izquierda.

Secuencia 3.2: Líneas visibles.



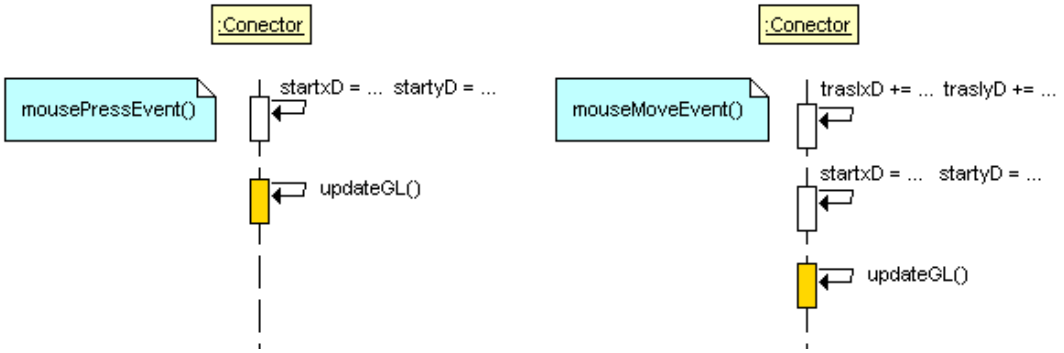
Para rotar la malla, el usuario presiona el botón izquierdo del ratón sobre el área de visualización y mueve el ratón sin soltar el botón, la malla seguirá el movimiento del ratón. Al soltar el botón del ratón el movimiento de la malla cesara.

Secuencia 3.3: Rotar.



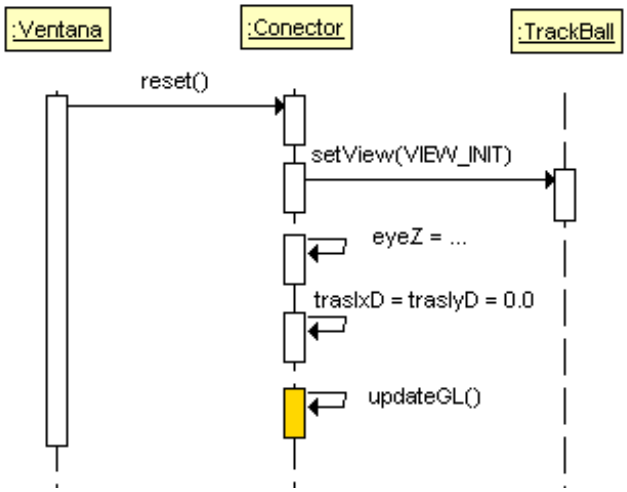
Para escalar la malla, el usuario presiona el botón derecho del ratón y realiza un movimiento ascendente o descendente del ratón, dependiendo si desea alejar o acercar la malla correspondientemente. Al soltar el ratón se realizan las mismas acciones que en la secuencia de rotar.

Secuencia 3.4: Escalar.



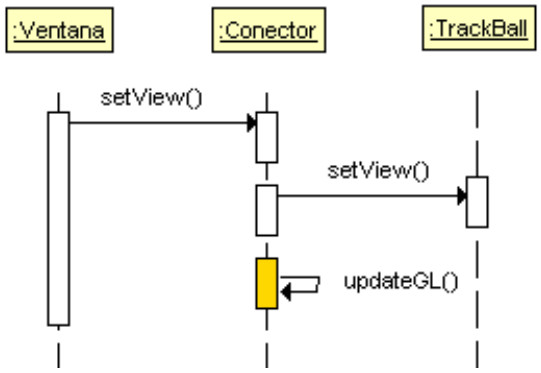
Para trasladar la malla, el usuario presiona el botón central del ratón ó el botón izquierdo y derecho al mismo tiempo. La malla seguirá el movimiento del ratón. Al soltar el ratón se realizan las mismas acciones que en la secuencia de rotar.

Secuencia 3.5: Trasladar.



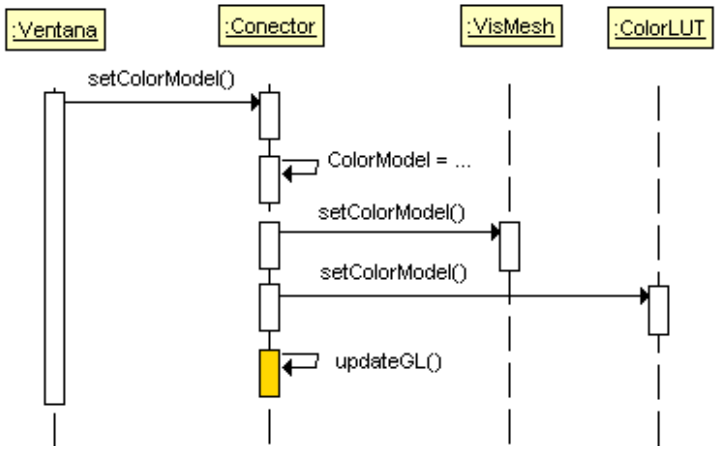
Cuando el usuario desea colocar la malla en la posición visual inicial, es decir, con el tamaño y la rotación por defecto, solo acciona el control adecuado en la interfaz y la malla se colocará en el estado visual por defecto.

Secuencia 3.6: Estado Visual Inicial.



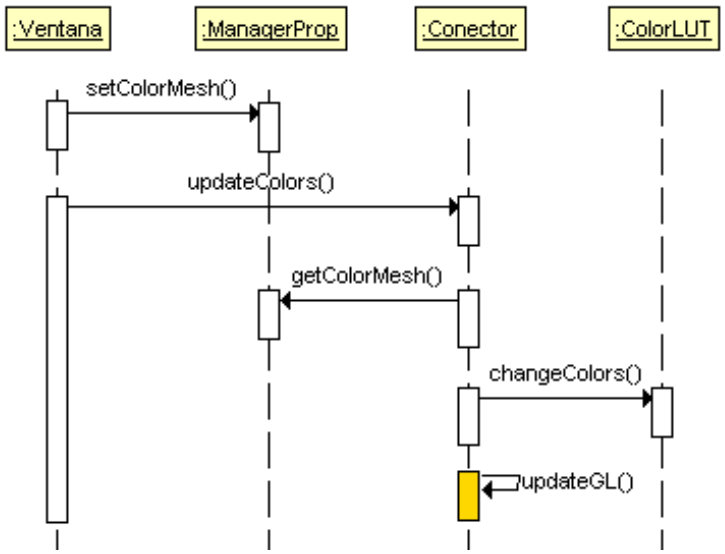
Para seleccionar una vista preestablecida de la malla, el usuario selecciona el control adecuado, se guarda la posición actual de la malla, se actualizan los valores adecuados y se actualiza la malla con la posición adecuada. El usuario podrá elegir entre ocho vistas preestablecidas, las vistas mostraran la malla desde la parte inferior, superior, izquierda, derecha, frontal, trasera, la vista inicial por defecto y la vista inmediata anterior a la selección de cualquiera de las siete vistas anteriores o una serie continua de cambios de vistas.

Secuencia 3.7: Vistas.



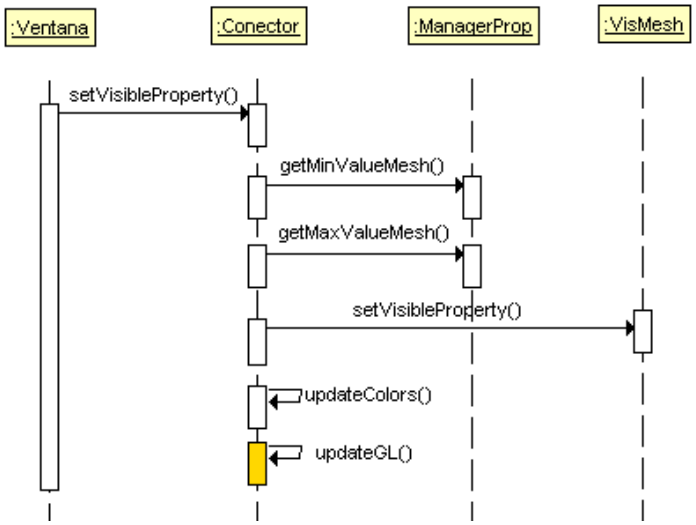
Para modificar el modelo de color, el usuario utiliza el control adecuado en la interfaz y selecciona uno de los tres modelos de color disponibles, entonces la interfaz hace la llamada a Conector y este a su vez coloca la opción adecuada en la malla y en la tabla de colores, posteriormente se actualiza la visualización con el cambio adecuado.

Secuencia 3.8: Modelo de Color.



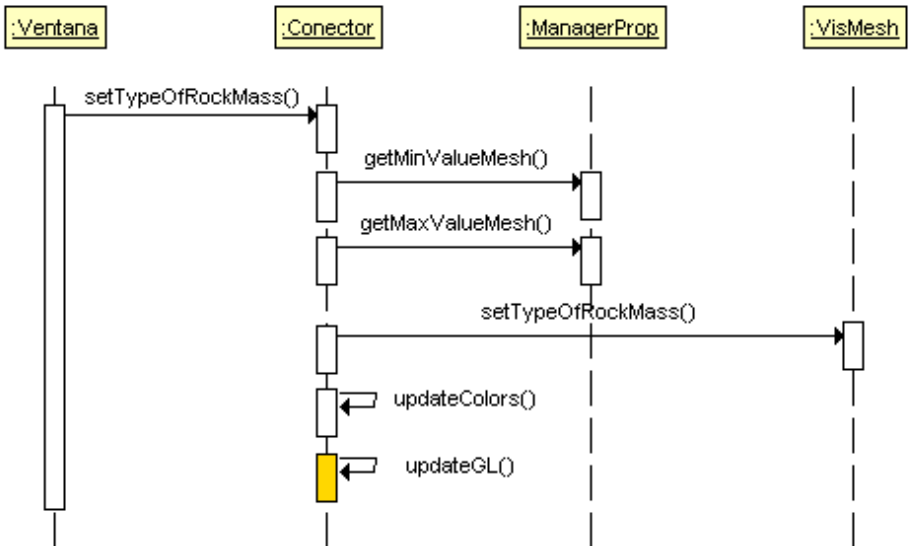
El usuario puede modificar el color mínimo y máximo de cualquier propiedad disponible accionando el control adecuado y seleccionando el color deseado del dialogo de selección de colores. El color elegido es guardado junto con las propiedades y posteriormente se actualizan los colores, puesto que la gama a representar seguramente habrá cambiado. Al final se actualiza la visualización.

Secuencia 3.9: Cambio Color Mínimo y Máximo.



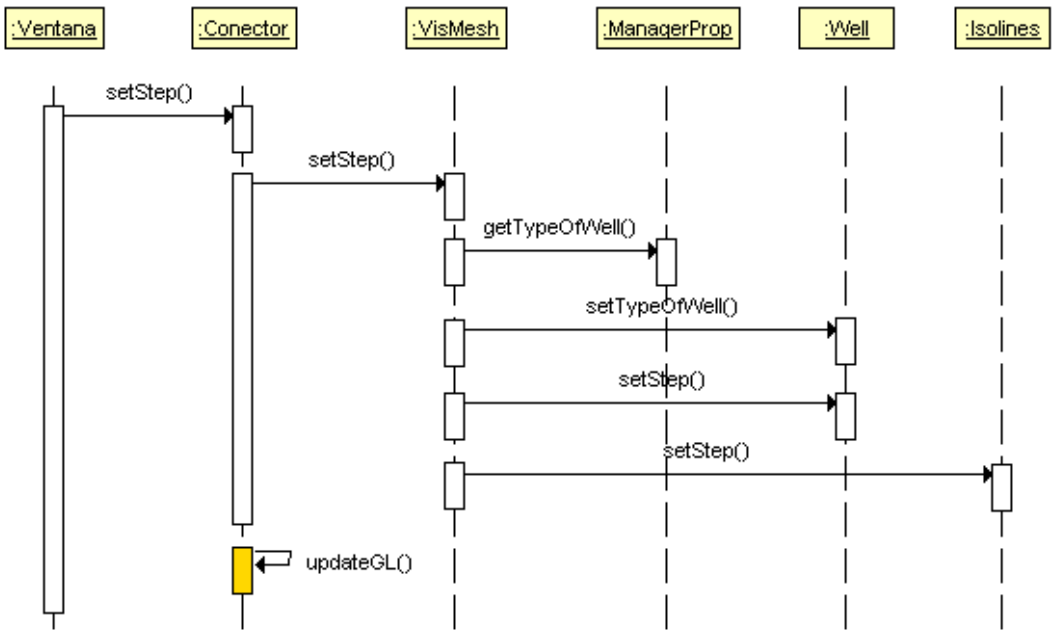
El usuario puede seleccionar la propiedad deseada mediante el control adecuado en la interfaz de usuario. La visualización actualizará la propiedad y los datos asociados a ella. Finalmente se repinta la visualización con el cambio realizado.

Secuencia 3.10: Seleccionar propiedad.



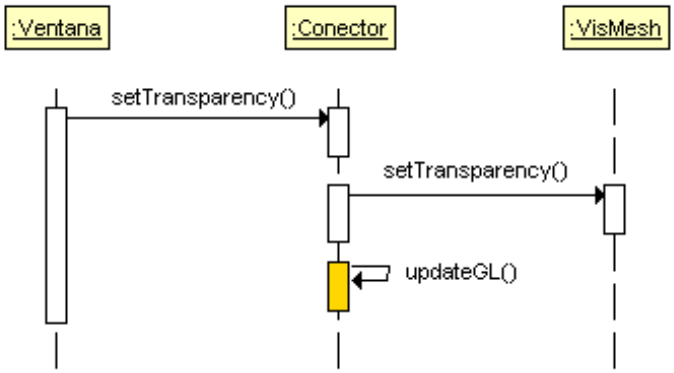
El usuario puede seleccionar el medio deseado mediante el control adecuado en la interfaz de usuario. La visualización actualizará el medio y los datos asociados a él. Finalmente se repinta la visualización con el cambio realizado.

Secuencia 3.11: Seleccionar Medio.



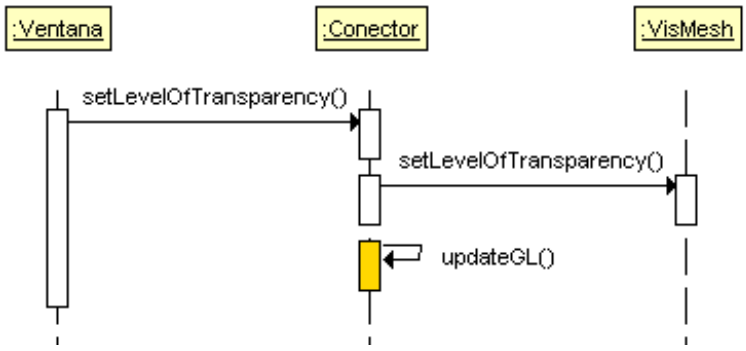
El usuario puede seleccionar el paso deseado mediante el control adecuado en la interfaz de usuario. La visualización actualizará el paso y los datos asociados a este. La actualización del tipo de pozos es necesaria, pues estos pueden cambiar de extractores a inyectores, pueden cerrarse o pueden surgir nuevos pozos en el transcurso del tiempo. Finalmente se repinta la visualización con el cambio realizado.

Secuencia 3.12: Seleccionar paso de simulación.



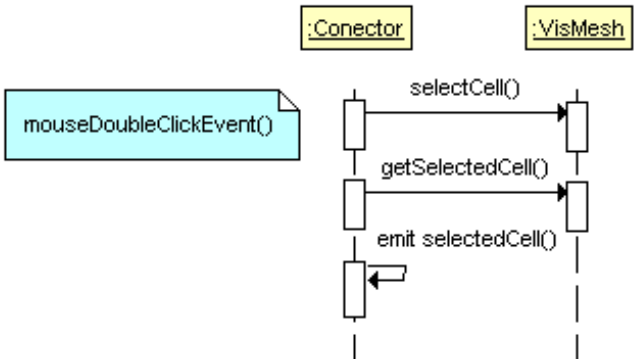
El usuario activa o inactiva la transparencia con una acción mínima activando el control adecuado en la interfaz. La visualización guarda la opción dada y actualiza el cambio.

Secuencia 3.13: Activar Transparencia.



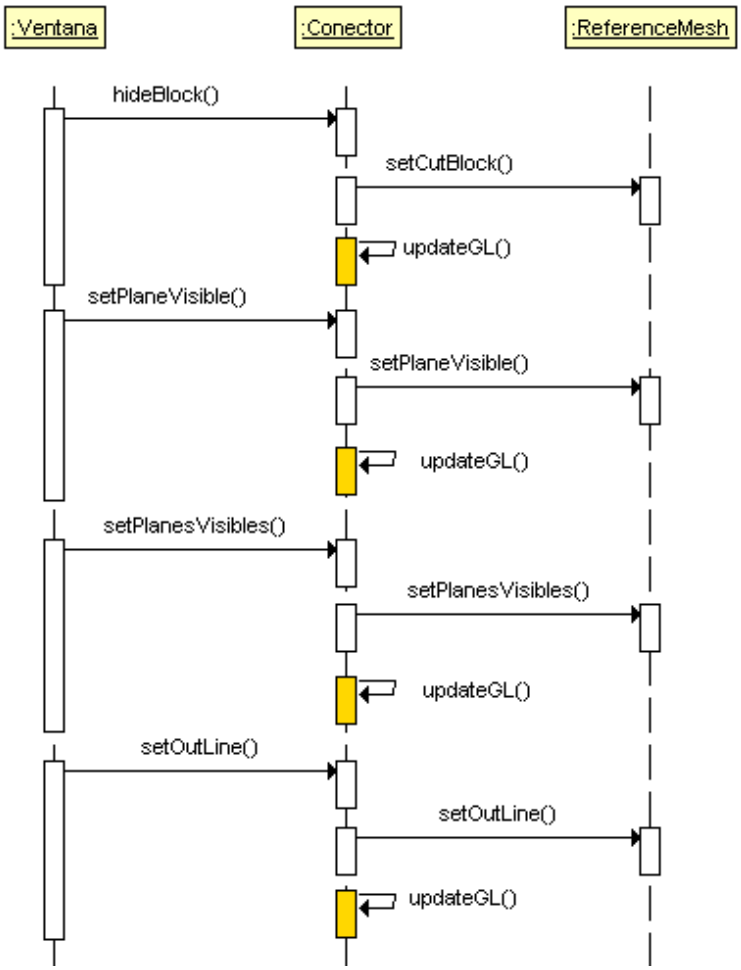
El usuario modifica el nivel de transparencia con una acción mínima activando el control adecuado en la interfaz y seleccionando el nivel deseado. La visualización guarda la opción dada y actualiza el cambio.

Secuencia 3.14: Modificar Nivel de Transparencia.



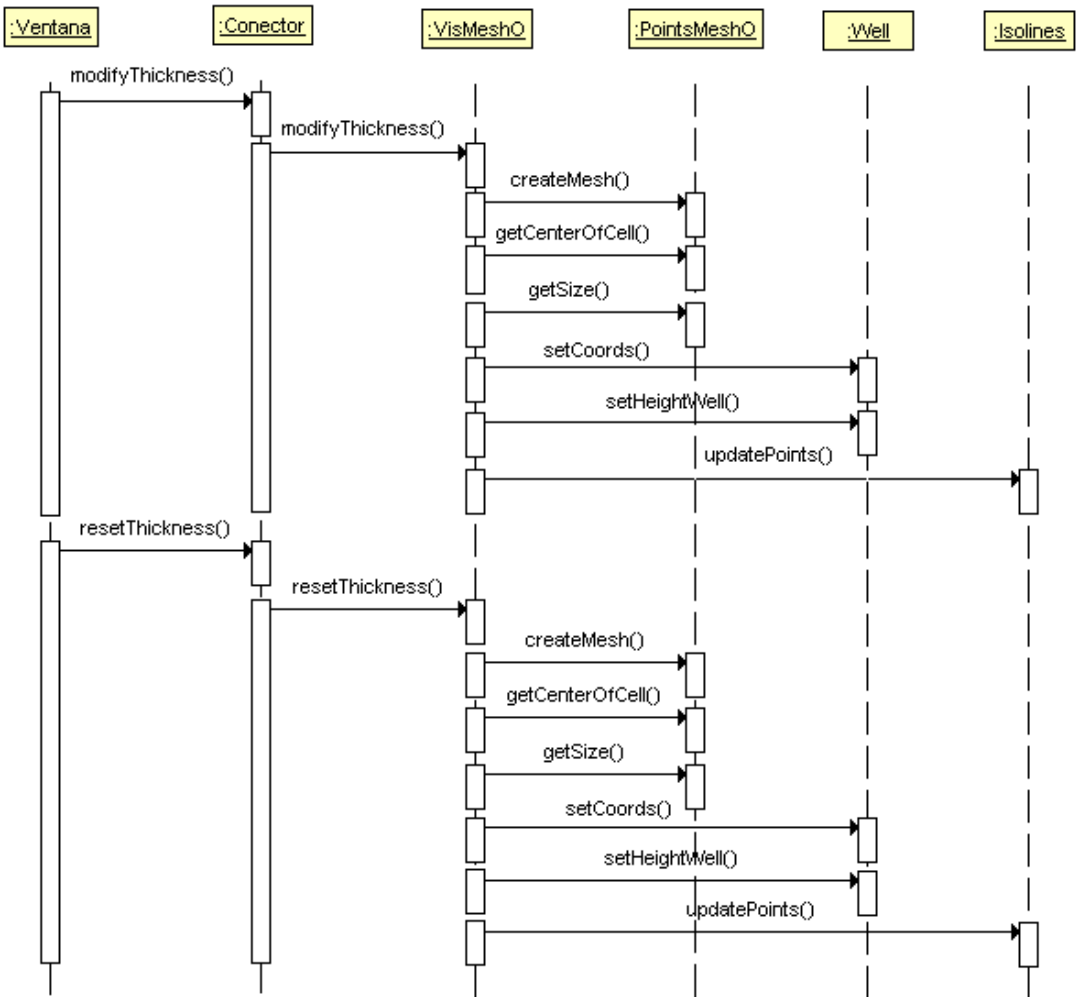
Para mostrar los datos de una celda particular, el usuario tiene que seleccionarla primero. Para seleccionar una celda, el usuario primero tiene que activar el modo correspondiente si es que no esta activo. Posteriormente sitúa el ratón sobre la celda deseada y realiza un doble clic del botón izquierdo del ratón. Los datos de la celda seleccionada se mostraran en el área de visualización y se harán los cambios correspondientes en los lugares necesarios, por ejemplo, en los datos mostrados en la representación ideal de los medios.

Secuencia 3.15: Seleccionar una Celda.



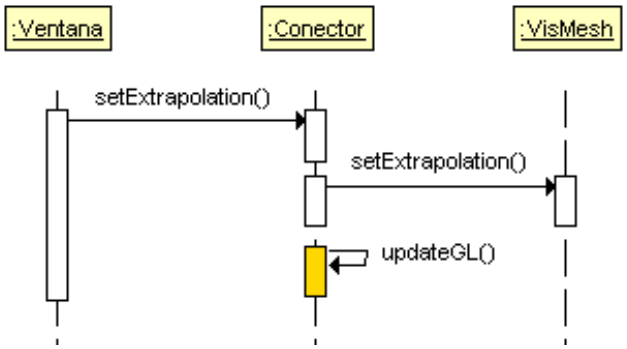
Para poder explorar las celdas internas del modo más libre y completo se proporcionan al usuario diversas opciones para ocultar bloques de celdas en los planos X, Y y Z, así como un conjunto de bloques. Además se da la opción de mostrar u ocultar la línea de las celdas ocultas, con el fin de que el usuario no pierda el tamaño ni la figura de la malla completa. El usuario podrá interactuar con los controles provistos y los cambios se verán reflejados conforme se vayan realizando.

Secuencia 3.16: Exploración de Celdas Internas.



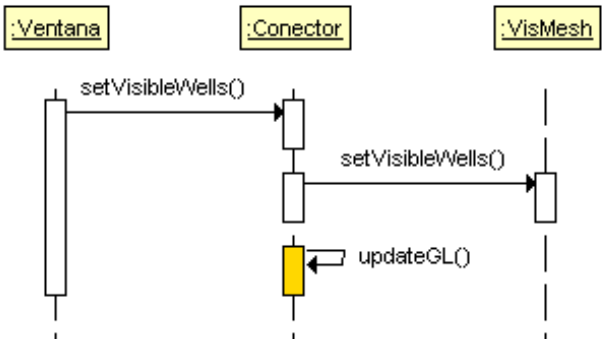
El usuario puede modificar la altura (grosor) de las celdas interactivamente. Conforme el usuario accione el control adecuado, la altura de las celdas se incrementara o se reducirá. El usuario vera constantemente los cambios realizados. Para colocar la altura inicial de las celdas, el usuario accionara el control adecuado y el cambio se vera en la visualización.

Secuencia 3.17: Modificar la Altura de las Celdas.



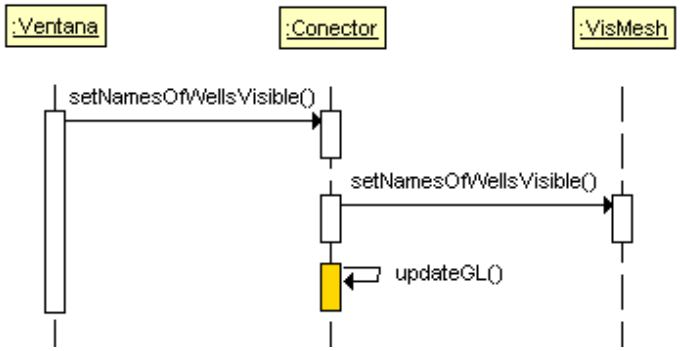
Cada celda de la malla contiene un solo dato por propiedad, medio y paso de tiempo, esto se ve reflejado en un solo color por celda. Cuando el usuario acciona el control de degradado de colores, se realiza una interpolación para cada vértice de cada celda interna, y se realiza una extrapolación para los vértices frontera. De esta forma, a cada vértice se le asigna un dato y por lo tanto un color, el resultado final es un cambio suave de colores entre celdas.

Secuencia 3.18: Interpolación y Degradado de Colores.



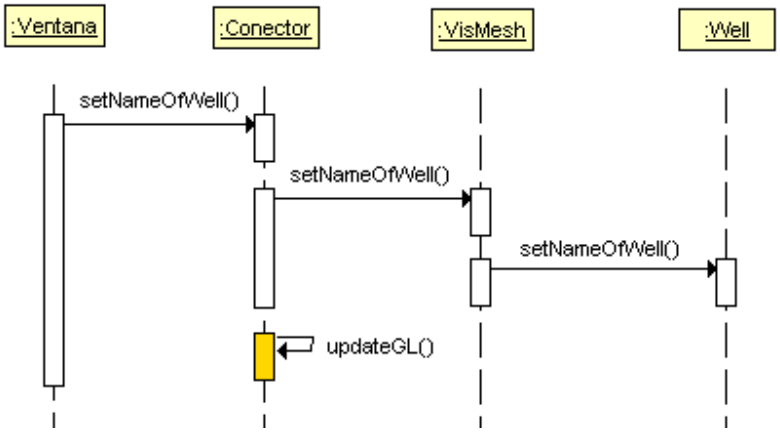
El usuario puede mostrar u ocultar los pozos y todas sus propiedades accionando el control adecuado. La visualización actualiza el cambio y se repinta.

Secuencia 3.19: Mostrar y Ocultar Pozos.



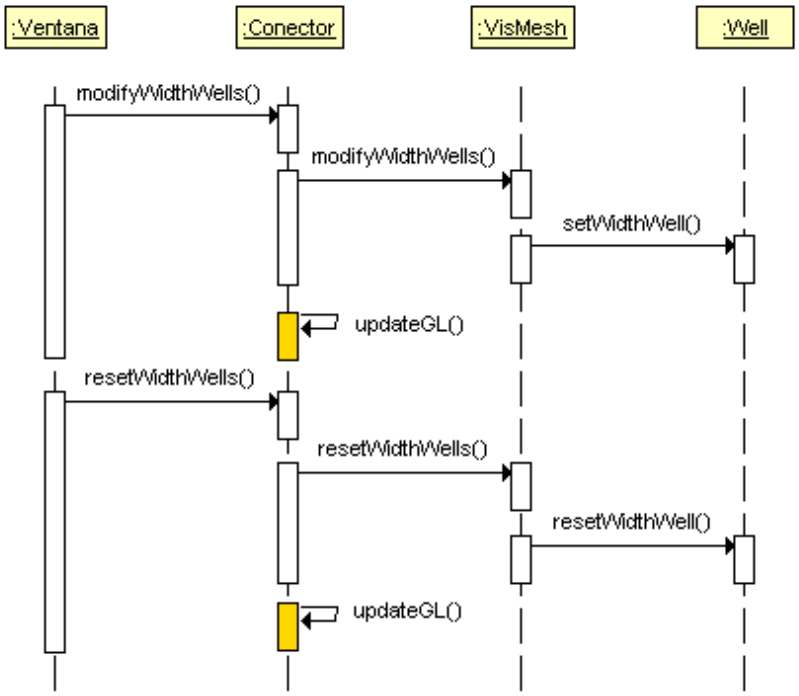
El usuario puede ocultar el nombre de los pozos accionando el control adecuado. Si los pozos se encuentran ocultos el cambio no será percibido, pero si será realizado. Así cuando los pozos sean mostrados nuevamente, serán mostrados con el cambio efectuado.

Secuencia 3.20: Mostrar/Ocultar Nombres de Pozos.



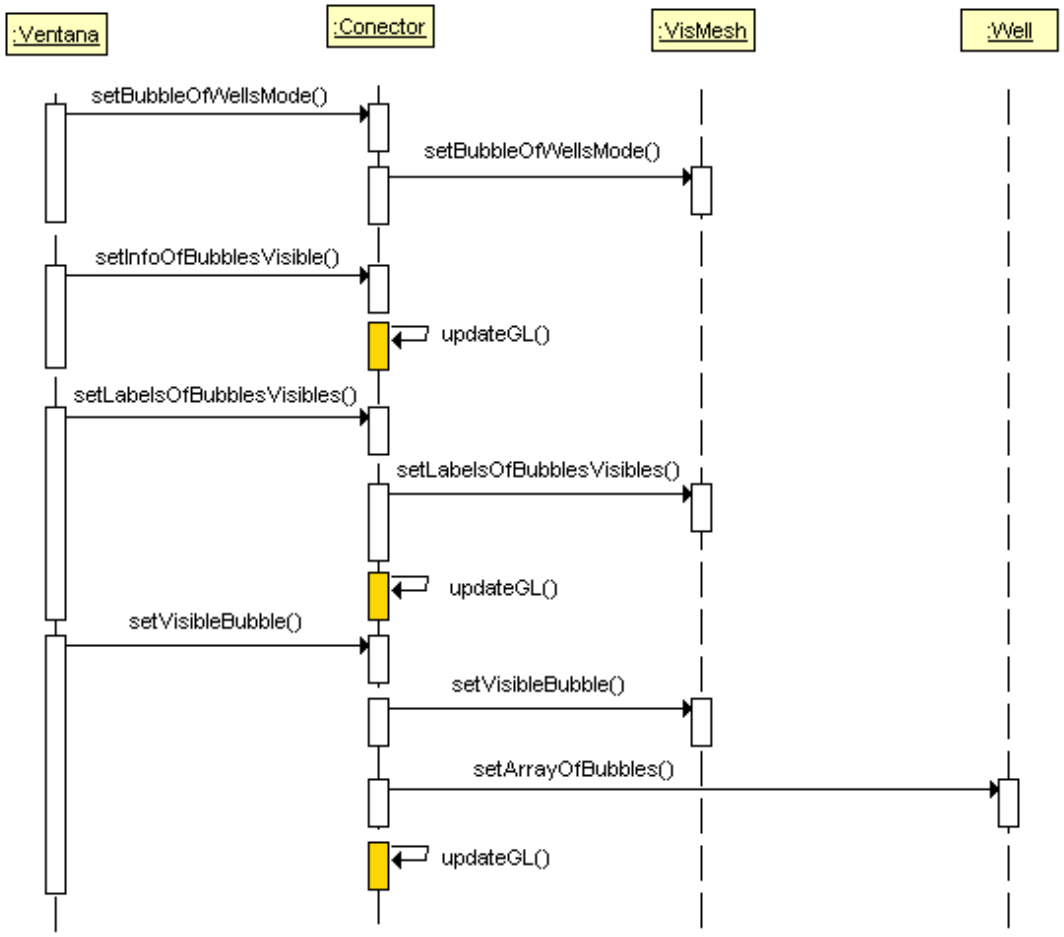
El usuario puede modificar el nombre de uno o varios pozos mediante los controles adecuados, cuando los cambios hayan sido realizados, se actualizará la visualización mostrando los cambios efectuados.

Secuencia 3.21: Modificar Nombres de Pozos.



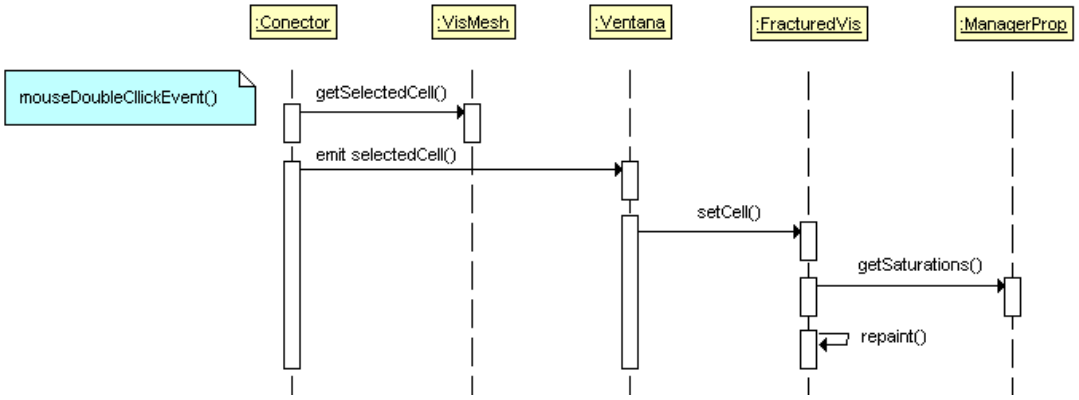
El usuario puede modificar el grosor (o ancho) de los pozos utilizando el control adecuado. También puede colocar el grosor inicial utilizando el control provisto y con una acción mínima. La visualización actualizará los cambios conforme se vayan realizando.

Secuencia 3.22: Modificar Grosor de Pozos.



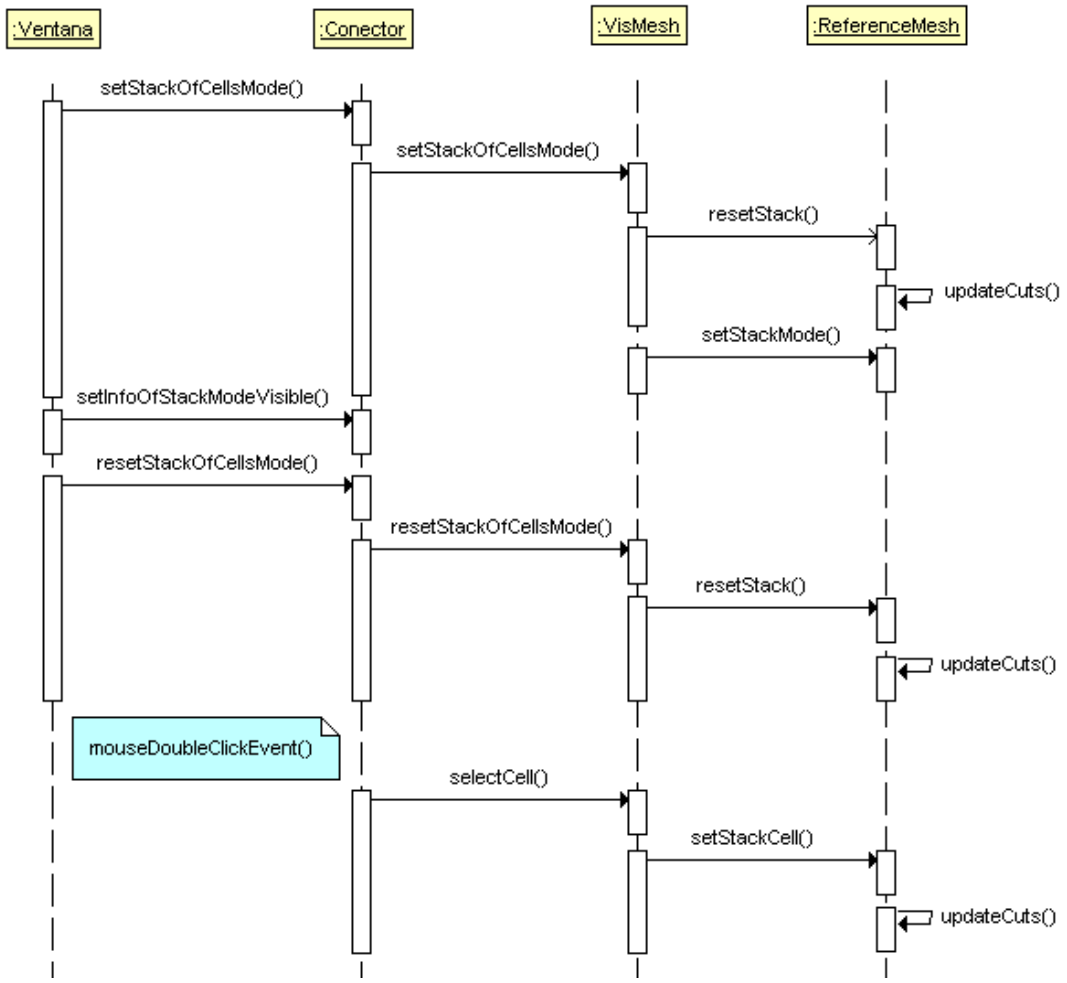
El usuario podrá mostrar las propiedades de los pozos mediante esferas (burbujas) accionando el control adecuado. También podrá seleccionar las propiedades que serán mostradas, así también, podrá mostrar u ocultar una etiqueta con la información de las propiedades y una etiqueta junto a la esfera con la información de esa propiedad en particular. Cada cambio realizado será actualizado en la visualización al momento de llevarse acabo.

Secuencia 3.23: Burbujas (Información de Pozos).



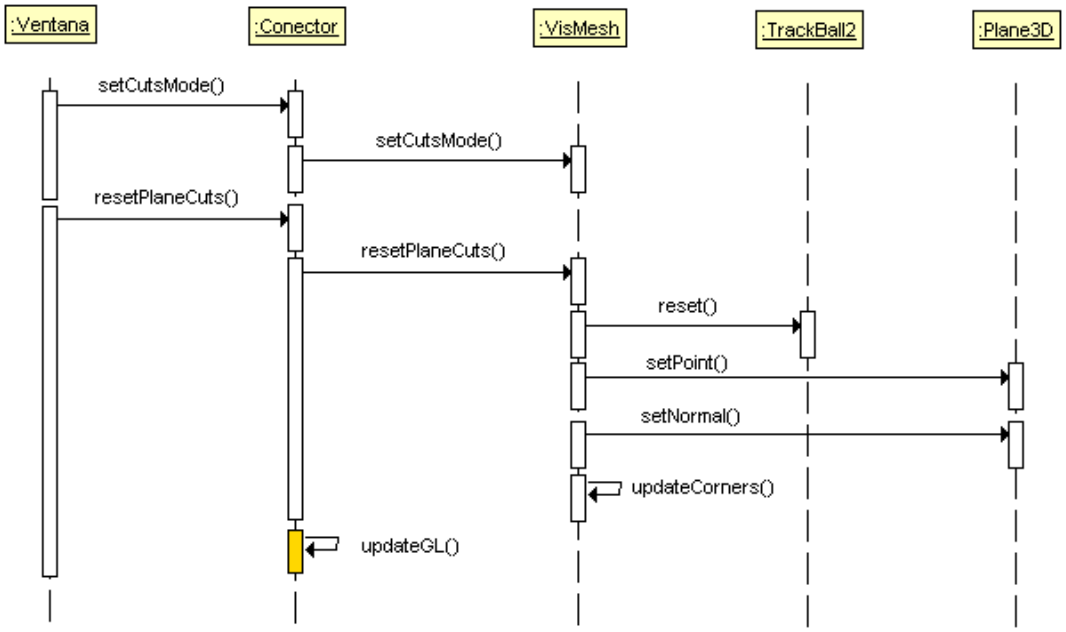
El usuario puede mostrar u ocultar la representación ideal de los medios, utilizando el control adecuado. Cuando dicha representación se encuentre activa, el usuario puede modificar la celda representada, seleccionando la celda que desee en la malla del área de visualización.

Secuencia 3.24: Representación ideal de los medios.



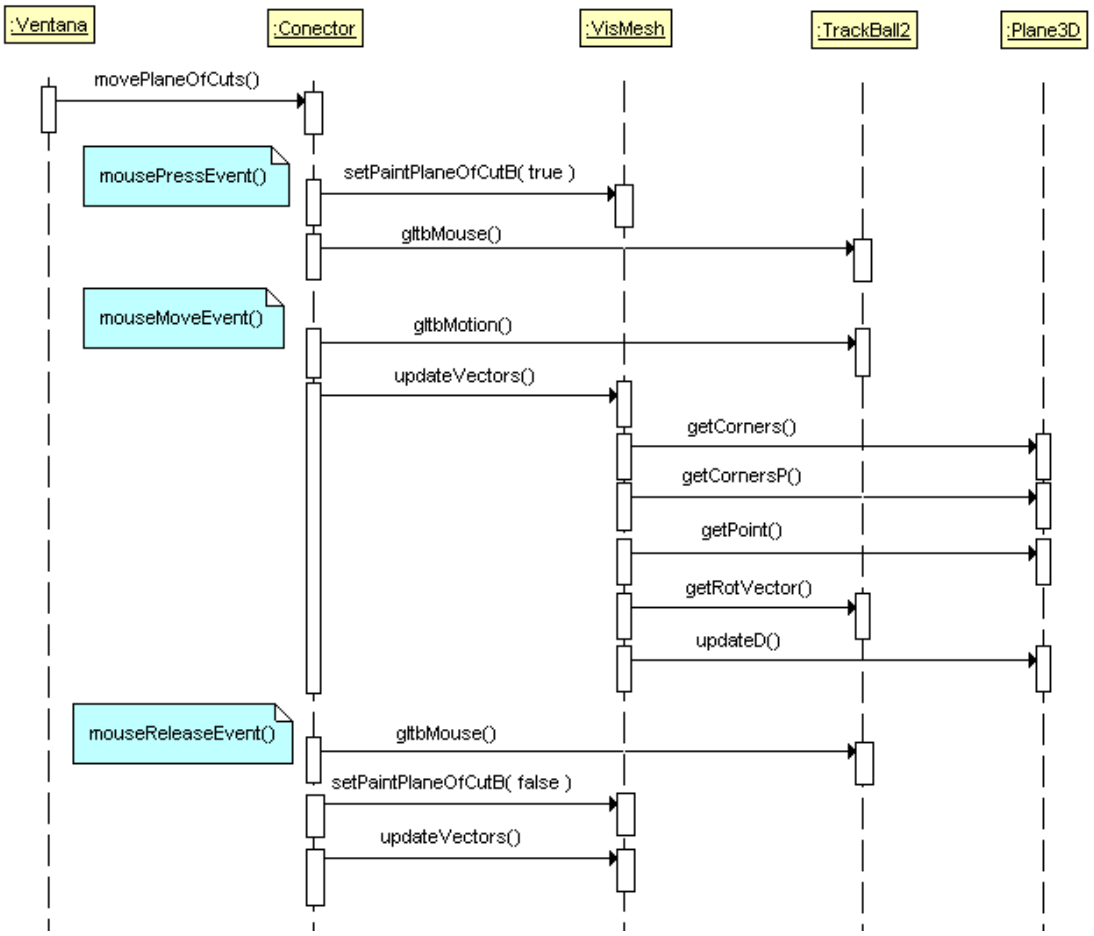
Para que el usuario pueda mostrar la curva de saturaciones de una pila de celdas, primero tiene que activar el modo con el uso del control adecuado. Posteriormente podrá seleccionar una pila de celdas de la malla posicionando el ratón sobre la celda deseada y dando un doble clic con el botón izquierdo del ratón. La pila de celdas seleccionada se mostrara sobre la malla con la gráfica de la curva de saturaciones. El usuario podrá seleccionar tantas pilas de celdas como desee. Si el usuario selecciona una pila de celdas con la curva de saturaciones, está regresara a su posición inicial. Dado que en algún momento puede ser engorroso colocar todas las pilas en su lugar, se da la opción al usuario de reestablecer todas las pilas a su lugar inicial con una acción mínima utilizando el control adecuado.

Secuencia 3.25: Curva de Saturaciones.



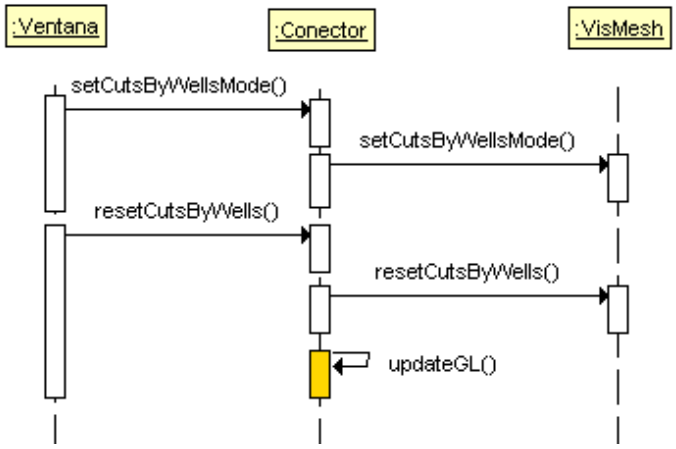
El usuario podrá realizar cortes a la malla utilizando un plano, para realizar los cortes primero deberá activar el modo correspondiente, con el control adecuado. El usuario podrá colocar el plano en su posición original con una acción mínima activando el control adecuado.

Secuencia 3.26: Cortes con Plano.



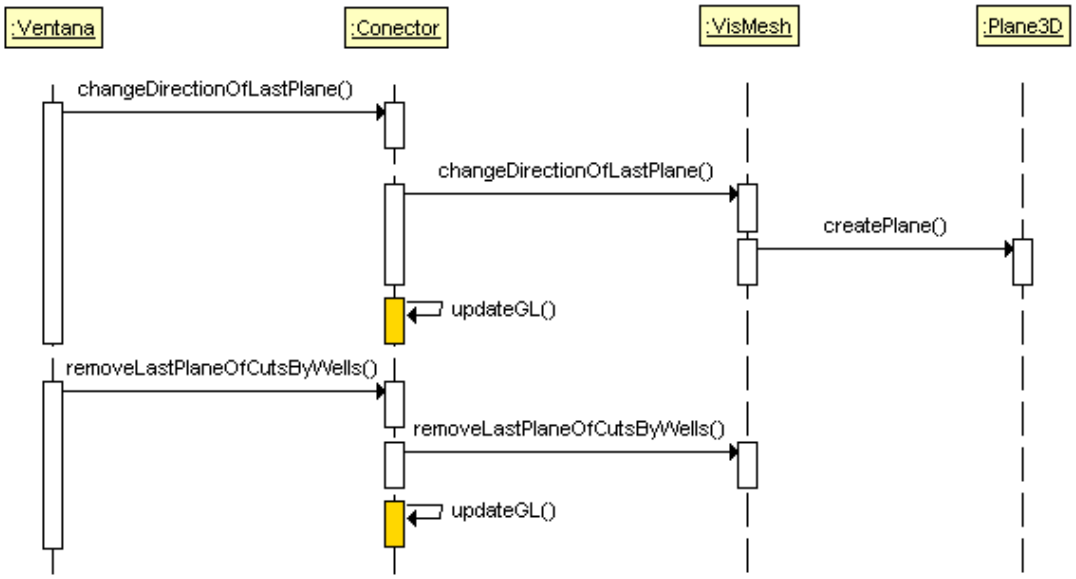
Al realizar un corte de la malla con el uso de un plano, el usuario podrá rotar el plano de corte, para hacerlo primero deberá activar el modo correspondiente y posteriormente usará el ratón de la misma forma que es usado para rotar la malla, pero estará rotando el plano de corte. La visualización irá actualizando el corte conforme se vaya realizando el mismo. Si el usuario desea rotar la malla deberá inactivar el modo para rotar el plano de corte y así podrá realizar rotaciones sobre la malla normalmente.

Secuencia 3.27: Mover Plano de Cortes.



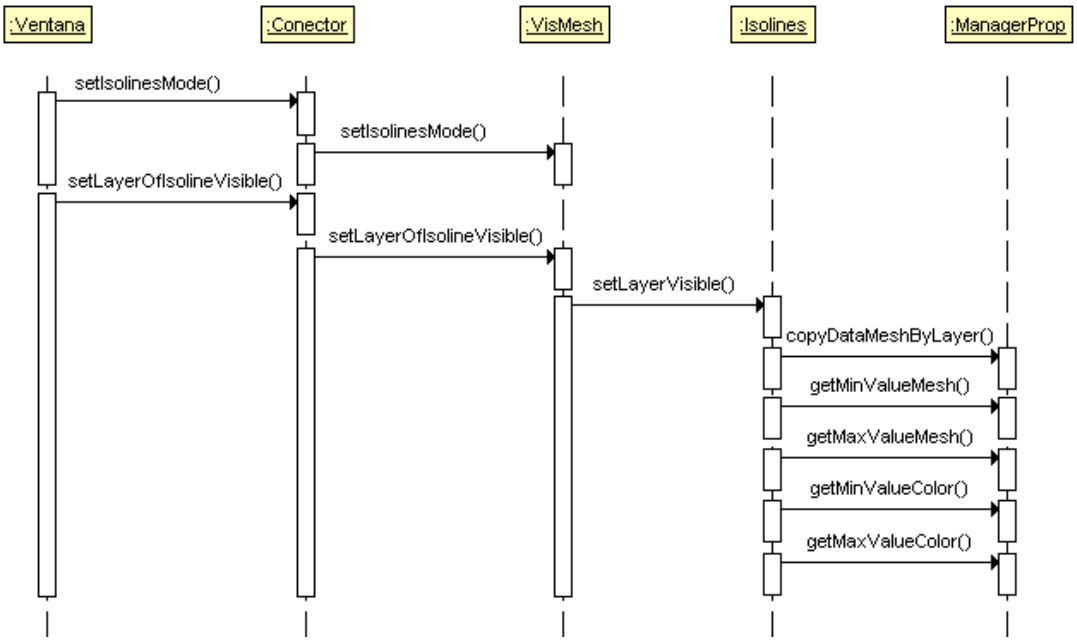
El usuario podrá realizar cortes a la malla con ayuda de los pozos, al seleccionar dos pozos se creará un plano de corte que atraviesa la malla y pasa por ambos pozos. El usuario puede hacer tantos cortes como el número de pozos lo permita. Se podrán eliminar todos los planos de corte creados por medio del control adecuado y con una acción mínima. La visualización se actualizará a cada corte realizado y al eliminar todos los planos de corte.

Secuencia 3.28: Cortes por Pozos.



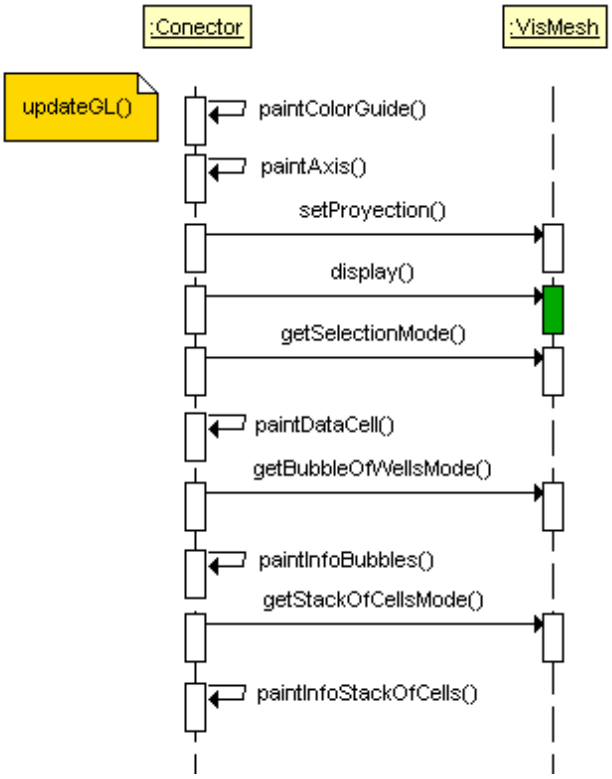
El usuario puede modificar la dirección del plano del último corte realizado utilizando el control adecuado. También puede eliminar el último plano de corte realizado. La visualización se actualizará a cada cambio efectuado.

Secuencia 3.29: Invertir Dirección del Plano de Corte por Pozos.



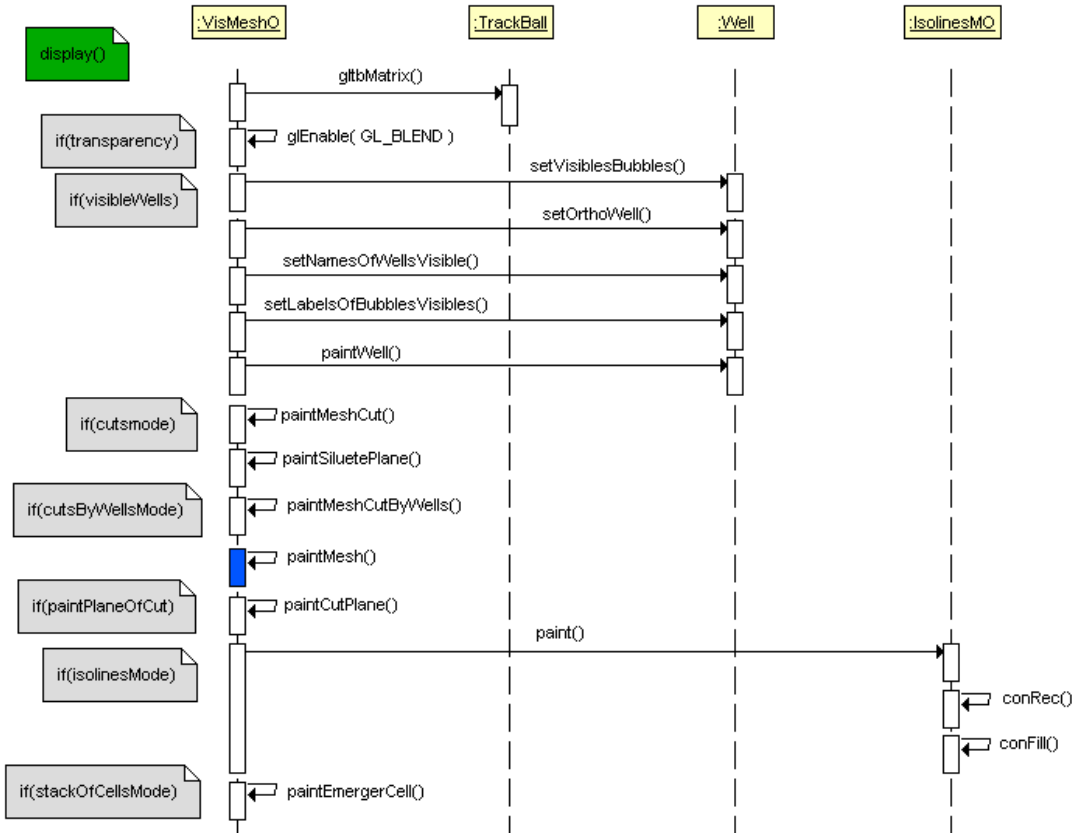
El usuario podrá mostrar las isolíneas de una capa determinada activando el modo correspondiente. También podrá modificar la capa sobre la cual se obtienen las isolíneas mediante el uso del control adecuado. La visualización actualizará cada cambio realizado.

Secuencia 3.30: Isolíneas.



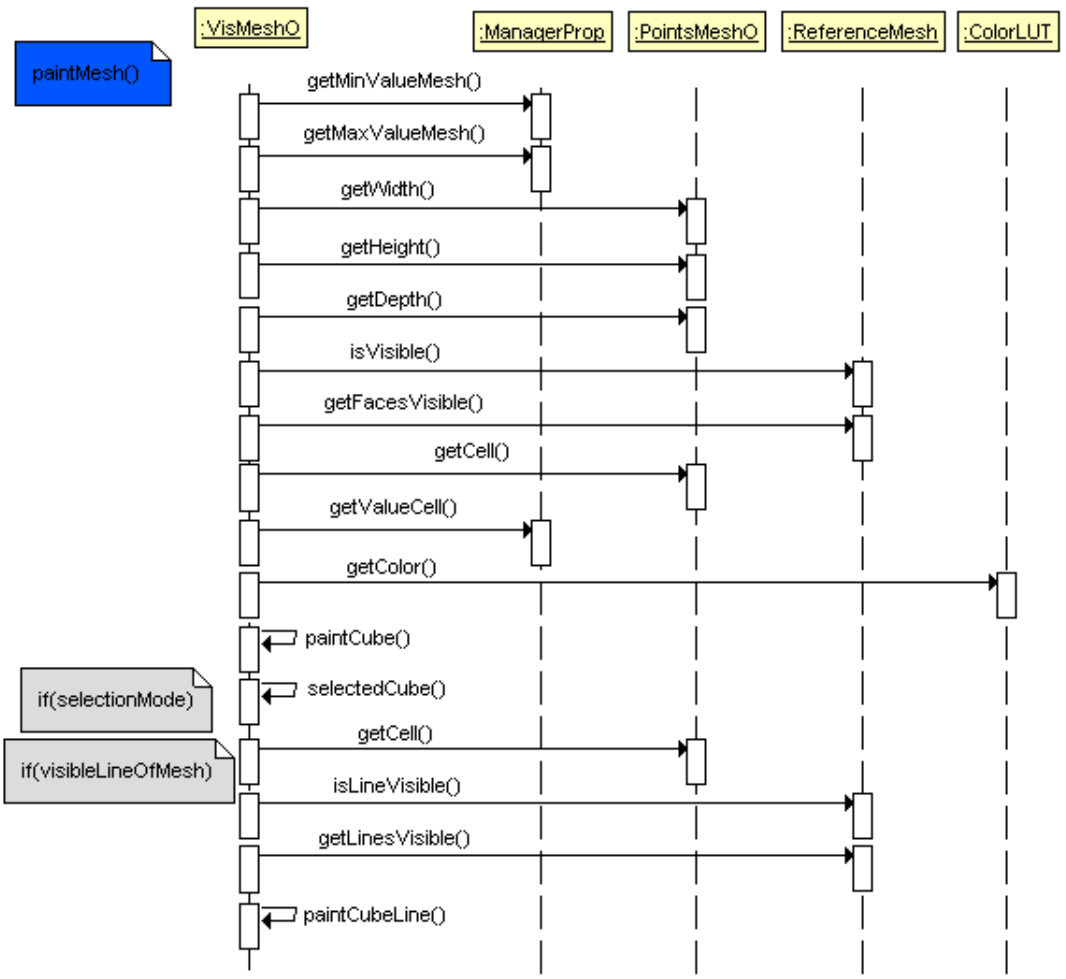
A cada actualización de la visualización la serie de tareas que se realizan siempre son las mismas. Se pinta la guía de los colores y los ejes que indican la dirección de la malla. Se manda visualizar la malla y dependiendo del modo seleccionado se muestra una guía que muestra información respecto del modo o de los modos activos.

Secuencia 3.31: PaintGL (equivalente a updateGL()).



Cada vez que se visualiza la malla se realizan una serie de tareas que son básicamente las mismas para una malla radial y una malla ortogonal. Primero se accede al estado de rotación de la malla, se activa la transparencia si el usuario lo indico, se colocan los estados de las propiedades de los pozos tal y como fueron seleccionadas por el usuario y se mandan pintar los pozos. Después se manda pintar la malla dependiendo del modo correspondiente y se pintan los indicadores necesarios dependiendo del modo.

Secuencia 3.32: displayMesh().



Para pintar la malla se toma en cuenta el modo en el que se encuentre la visualización, pero comúnmente se realizan las siguientes acciones. Se obtienen los datos necesarios para realizar el cálculo del mapeo de las propiedades en colores. Posteriormente se recorre la malla en ancho, alto y profundo y por cada celda se decide si se muestra o no. Si la celda es mostrada, se decide que caras de la misma serán pintadas, se obtienen las coordenadas de la celda y por ende de cada cara, y se decide de que color o colores serán pintadas. Se pinta la celda con los datos obtenidos y dependiendo del modo. Posteriormente se pintan las líneas de la celda si estas están activas y solo se pintan las líneas de las caras visibles. Para cada modo se hacen los ajustes necesarios. Los pasos para pintar una malla radial son los mismos, pero con los ajustes necesarios.

Secuencia 3.33: paintMesh().

Capítulo 4

Diseño Detallado de Algoritmos

En el presente capítulo se proporciona una explicación detallada de cada clase y los algoritmos principales que la conforman. Cuando sea necesario se incluyen diagramas, formulas, pseudocódigo o código.

4.1. Clase "Conector"

Esta clase es la encargada de hacer la conexión entre un módulo externo y toda la visualización, a excepción de la clase ManagerProp, todas las funcionalidades de la visualización se manejan desde esta clase. Ya que la función principal de esta clase es la de gestionar peticiones al módulo, la mayor parte de sus funciones solo realizan llamadas a funciones de otras clases.

También es la clase que contiene el widget principal de la visualización y por ende maneja todos los eventos del ratón y los eventos de ventana propios del área de visualización.

Esta clase también maneja la información de los colores que el usuario percibe y muestra esta información mediante una barra de colores o un triángulo de colores dependiendo del modelo de color elegido por el usuario. Así mismo, maneja la información de las burbujas, de la curva de saturaciones, muestra los datos de la celda seleccionada y muestra los ejes.

4.2. Clase "VisMesh"

Clase encargada de implementar las funciones comunes a los dos tipos de mallas, además de definir las funciones que cada tipo de malla especifica deberá implementar. Gestiona el acceso y el uso de los diferentes modos de visualización. Inicia y gestiona todos los arreglos y objetos usados por ambos tipos de mallas.

Los objetos y arreglos que maneja esta clase son los siguientes: un objeto de la clase ColorLUT, uno de la clase ManagerProp, uno de TrackBall, uno de TrackBall2, uno de

ReferenceMesh, uno de Isolines, uno de Plane3D y un arreglo de objetos del tipo Plane3D. Contiene variables para el nombre de las propiedades de los pozos, para los datos de extrapolación, para el color de una celda, para la celda seleccionada, para los pozos seleccionados, para contener todos los modos, para la visibilidad de pozos, de líneas de las celdas, para saber el modelo de color, la propiedad, el medio, el número de pasos, el número de medios, el número de propiedades, el nivel de transparencia, la posición visual de la malla y para conocer el estado actual de la malla.

Como se puede apreciar, esta clase maneja una gran cantidad de información y un considerable número de estados de la malla.

4.3. Clase "VisMeshO"

Esta es la clase encargada de pintar una malla ortogonal a partir de los datos contenidos en pointsMeshO. Para pintar la malla, esta clase pinta de celda en celda, hasta completar la malla, tomando en cuenta los datos obtenidos de la clase ReferenceMesh que indica cual celda pintar y cual no, además de las caras y líneas visibles. Además de pintar una malla normal, esta clase se encarga de pintar la malla en los diferentes modos, con el auxilio de las clases especializadas.

Los algoritmos de pintado se basan completamente en el orden del arreglo que contiene los datos de los vértices de cada celda, en el orden del arreglo de índices y en los arreglos de caras y de líneas visibles, por este motivo se explicarán las variables principales y posteriormente los métodos y algoritmos principales de la clase.

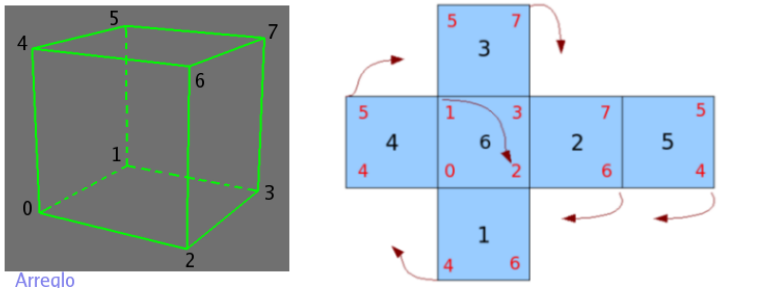
4.3.1. Variables principales

a) **PointsMeshO* pointsMeshOO:** Es un objeto de la clase pointsMeshO, contiene los vértices de todas las celdas además del tamaño de la malla y otros datos de la misma.

b) **double* rgbDA:** Es un arreglo en el cual se guarda el color con el que será pintada cada celda.

c) **double* pointsCubeDA:** Es el arreglo en el cual se guardan los datos de la posición de la celda que se va a pintar. Los datos de este arreglo se encuentran estrictamente en el orden mostrado en la Figura 4.1.

   Como se puede apreciar en la Figura 4.1, la celda se encuentra formada por 8 vértices, cada vértice es de la forma XYZ y por lo tanto ocupa tres posiciones continuas en el arreglo, dando un total de 24 datos continuos. En el esquema se muestran los índices del arreglo y los índices de los vértices que corresponden a cada triada de datos. Aunado a estos datos, en la figura se muestra una tabla y dos esquemas, posteriormente explicaremos estos diagramas con más detalle.



Caras	Vertices	Indice del arreglo	Celdas vecinas
1	4, 0, 2, 6	12, 0, 6, 18	x y-1 z
2	6, 2, 3, 7	18, 6, 9, 21	x+1 y z
3	7, 3, 1, 5	21, 9, 3, 15	x y+1 z
4	5, 1, 0, 4	15, 3, 0, 12	x-1 y z
5	4, 6, 7, 5	12, 18, 21, 15	x y z+1
6	1, 3, 2, 0	3, 9, 6, 0	x y z-1

Figura 4.1: Esquema de la representación de una celda ortogonal.

d) int indexArrayIA:** Este es un arreglo matricial de 6 X 4 en el cual se guardan los índices de los vértices que conforman cada cara de la celda, en la tabla contenida en la Figura 4.1 se encuentra una subtabla con el título de Índice del arreglo, esta corresponde exactamente al contenido de este arreglo. Como se puede apreciar en la imagen, el contenido del arreglo indica de que posición del arreglo pointsCubesDA se deben de tomar los datos de cada vértice de cada cara, además de esto también indica el orden para tomarlos. Por ejemplo, la cara 6, que es la cara inferior del cubo, se encuentra formada por los vértices 1, 3, 2 y 0, y los índices son el 3, 9, 6 y 0, de esta forma ya sabemos de donde tomar los datos y en que orden.

e) int indexsLinesIA:** Este es un arreglo de 12 X 2, y lo utilizamos para guardar los índices de los puntos que forman las líneas que, a su vez, conforman a nuestra celda. La idea es muy similar a la del arreglo anterior, por ejemplo la línea 2 se encuentra formada por los vértices 0 y 2, los que a su vez se encuentran en la posición 0 y 6, estos dos últimos datos son los que contiene este arreglo bidimensional en su segunda entrada. La Figura 4.2 muestra el diagrama de una celda con la numeración de las líneas que se sigue en este arreglo.

f) bool* visibleFacesBA: Es un arreglo de boléanos en el cual guardaremos como true (verdadero) la cara que se podrá visualizar y por lo tanto se pintara, y como false (falso) aquella cara que no debe de verse y que por lo tanto no se pintaran. El orden

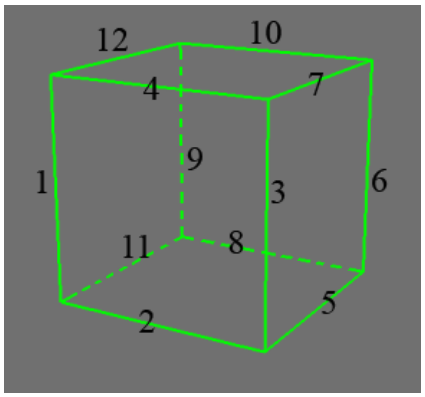


Figura 4.2: Numeración de líneas.

de las caras es el que se muestra en la Figura 4.1.

g) **int** indexsFacesVLineIA:** Al igual que el arreglo anterior, este arreglo sirve para guardar cuales líneas serán pintadas y cuales no. El orden de las líneas es el que se muestra en la Figura 4.2.

4.3.2. Funciones principales

a) **int setPointsMeshO(PointsMeshO* pointsV)**

Esta función es utilizada para colocar la instancia de la clase PointsMeshO que contiene los vértices de todas las celdas de la malla. Primero se checa que el objeto sea valido, si es así, guardamos el objeto en la variable pointsMeshOO y posteriormente mandamos este objeto a la clase de Isolíneas, puesto que esta también utiliza las coordenadas de las celdas. Finalmente calculamos las esquinas que forman el plano que se utilizará en algún momento para cortar la malla.

b) **void paintCubeLine(double r, double g, double b, double ancho)**

Esta función se encarga de pintar las líneas de una celda con el color y el tamaño indicados en los parámetros. Utiliza el arreglo pointsCubeDA para obtener los vértices, el arreglo visibleFacesBA para saber cuales líneas pintar y cuales no. Además utiliza los arreglos indexsLinesIA e indexsFacesVLineIA para saber como tomar esos datos. El algoritmo es el siguiente:

```
1: for i = 0 hasta 11 do
2:     if la línea i es visible then
3:         pintar la línea i
```

```
4:   end if
5: end for
```

c) void paintCube(double r, double g, double b)

Se encarga de pintar una celda no seleccionada, esta función recibirá tres parámetros que representan el color que tendrá la celda, además hace uso de la variable levelOfTransparencyD para colocar el nivel de transparencia de la celda en caso de que se active la transparencia en la visualización. Esta función utiliza el arreglo visibleFacesBA para saber cual cara pintar, usa el arreglo pointsCubeDA e indexsArrayIA para saber que vértices pintar y en que orden. Se utilizan QUADS de openGL para pintar cada cara, de esta forma al pintar cuatro vértices, automáticamente se pinta una cara, por lo que no es necesario indicar explícitamente que se esta pintando una cara. El algoritmo es el siguiente.

```
1: for j = 0 hasta 5 do
2:   if la cara j es visible then
3:     for i = 0 hasta 3 do
4:       pintar el vértice j,i
5:     end for
6:   end if
7: end for
```

d) void paintCube(double min, double step, int cx, int cy, int cz)

Esta función pinta una celda en el modo de extrapolación. Con los parámetros obtiene los datos extrapolados de la celda pidiéndoselos a la clase ManagerProp y los coloca en el arreglo extrapDataDA. Posteriormente pinta la celda de una forma muy parecida a la función anterior, pero con la particularidad de que calcula un color para cada vértice de cada cara. El algoritmo es el siguiente:

```
1: Obtener los datos extrapolados.
2: for j = 0 hasta 5 do
3:   if la cara j es visible then
4:     for i = 0 hasta 3 do
5:       calcular el color para el vértice j,i
6:       pintar el vértice j,i
7:     end for
8:   end if
9: end for
```

e) void paintMesh()

Esta función es muy importante ya que se encarga de crear la malla ortogonal a partir de las funciones anteriores (paintCubeLine() y paintCube()). El algoritmo es el siguiente:

```
1: for z = 0 hasta la altura de la malla do
2:   for y = 0 hasta la profundidad de la malla do
3:     for x = 0 hasta el ancho de la malla do
4:       if la celda x, y, z es visible then
5:         obtener las caras visibles
6:         obtener las coordenadas de la celda x, y, z
7:         obtener el color dependiendo del modelo de color
8:         pintar la celda
9:         if esta activo el modo de selección then
10:          pintar la celda en modo de selección
11:        end if
12:      end if
13:      if las líneas de la malla son visibles then
14:        obtener las coordenadas de la celda x, y, z
15:        if las líneas de la celda son visibles then
16:          obtener las líneas visibles
17:          if esta activo el modo de selección then
18:            pintar las líneas en modo de selección
19:          else
20:            pintar las líneas de la celda
21:          end if
22:        end if
23:      end if
24:    end for
25:  end for
26: end for
```

f) void paintMeshCut()

Esta función se encarga de pintar la malla bajo el modo de cortes con plano. Utiliza la clase especializada de cortes, la clase CutPlanes. El algoritmo es muy similar al anterior, pero omite la sección del modo de selección. El algoritmo es el siguiente:

```
1: Colocar la tabla de colores al objeto de cortes.
2: Colocar el estado del modo de extrapolación al objeto de cortes.
3: for z = 0 hasta la altura de la malla do
4:   for y = 0 hasta la profundidad de la malla do
```

```
5:      for x = 0 hasta el ancho de la malla do
6:        if la celda x, y, z es activa then
7:          obtener las coordenadas de la celda x, y, z
8:          calcular si el plano de corte atraviesa la celda
9:          if la celda es visible dependiendo del plano de corte then
10:           obtener el color dependiendo del modelo de color
11:           if el plano atraviesa la celda then
12:             pintar la celda usando la clase de cortes
13:           else
14:             pintar la celda
15:           end if
16:         end if
17:         if las líneas de la malla son visibles then
18:           obtener las coordenadas de la celda x, y, z
19:           if la celda es visibles dependiendo del plano de corte, pero el plano
              no la atraviesa then
20:             pintar las líneas de la celda
21:           end if
22:         end if
23:       end if
24:     end for
25:   end for
26: end for
```

g) void paintMeshCutByWells()

Esta función se encarga de pintar la malla bajo el modo de cortes por pozo. Utiliza la clase especializada de cortes, la clase CutPlanes. El algoritmo es muy similar al anterior, pero añade un nuevo ciclo para todos los planos generados. El algoritmo es el siguiente:

```
1: if no hay planos de corte then
2:   pintar la malla usando la función paintMesh()
3: end if
4: Colocar la tabla de colores al objeto de cortes.
5: Colocar el estado del modo de extrapolación al objeto de cortes.
6: for z = 0 hasta la altura de la malla do
7:   for y = 0 hasta la profundidad de la malla do
8:     for x = 0 hasta el ancho de la malla do
9:       if la celda x, y, z es activa then
10:        obtener las coordenadas de la celda x, y, z
11:        for cada plano de corte do
```

```
12:              calcular si el plano de corte atraviesa la celda
13:              if la celda es visible dependiendo del plano de corte then
14:                obtener el color dependiendo del modelo de color
15:                if el plano atraviesa la celda then
16:                  pintar la celda usando la clase de cortes
17:                else
18:                  pintar la celda
19:                end if
20:              end if
21:              if las líneas de la malla son visibles then
22:                obtener las coordenadas de la celda x, y, z
23:                if la celda es visibles dependiendo del plano de corte, pero el plano
                   no la atraviesa then
24:                  pintar las líneas de la celda
25:                end if
26:              end if
27:            end for
28:          end if
29:        end for
30:      end for
31:  end for
```

h) void display()

Función encargada de gestionar los diferentes modos de pintar la malla. Además esta encargada de administrar la visualización de los pozos.

Primero coloca la rotación de la malla con ayuda de trackBall, posteriormente activa la transparencia si la opción esta habilitada. A continuación manda pintar cada pozo y posteriormente pinta la malla de acuerdo al modo seleccionado por el usuario. En seguida le indica a las Isolíneas, si están habilitadas, que se pinten. Si la curva de saturaciones esta habilitada, se pinta. Al final pinta la esfera de referencia si así lo requirió el usuario y deshabilita la transparencia.

i) int modifyThickness(int p)

Función usada para modificar el grosor de las capas de la malla. Para poder implementar este método se hace uso de la clase pointsMeshO, dado que en lugar de crear algoritmos de escalado que sean complejos o lentos, se decidio crear una nueva malla con las nuevas dimensiones. Esto que podría parecer una solución mala es en realidad la mejor opción disponible, puesto que, ya que el usuario ha elegido un grosor adecuado, difícilmente lo modificara a lo largo de todo el proceso de simulación o de exploración de datos. El algoritmo es el siguiente:

1: Calcular el nuevo grosor de las celdas.
2: Crear una nueva malla con las nuevas dimensiones.
3: Calcular las nuevas dimensiones de los pozos.
4: **for** cada pozo **do**
5: calcular la nueva posición de salida del pozo
6: colocarle al pozo sus nuevas coordenadas
7: colocarle al pozo sus nuevas dimensiones
8: **end for**
9: Actualizar los puntos en las isolíneas.

j) void paintCubeStack(int h, double dz, double min, double step, int cx, int cy, int cz)

Función encargada de pintar la curva de saturaciones de una sola celda. El algoritmo toma la celda y la descompone en caras, posteriormente se aplica el mismo algoritmo a cada una de las cuatro caras alrededor de la celda. La cara superior y la cara inferior se pintan con el color normal de la celda. La forma en como trabaja el algoritmo se muestra en la Figura 4.3.

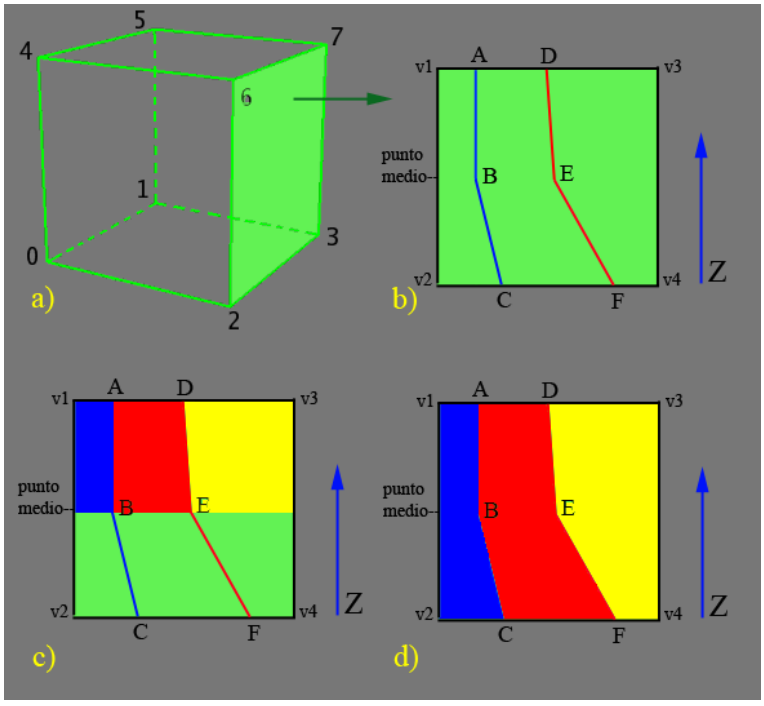


Figura 4.3: Curva de saturaciones de una celda.

Como se puede apreciar para poder generar la curva, primero tomamos una cara de la celda. Sobre esta cara encontramos seis puntos A, B, C, D, E y F, estos puntos corresponden al valor de las saturaciones de agua, de aceite y de gas de la celda, conjuntamente con los mismos valores de sus vecinas superior e inferior. Una vez localizados estos puntos, pintamos las curvas por medio de cuadrilateros de OpenGL. Cada curva utiliza 2 cuadrilateros solo para asegurarnos que sean pintados correctamente. Este procedimiento es aplicado para las 4 caras alrededor de la celda.

k) void paintEmergerCell(GLenum mode)

Función encargada de mostrar la curva de saturaciones en las pilas de celdas seleccionadas por el usuario. Esta función decide cuales celdas pintar y cuales no, hace el recorrido de toda la malla y a las celdas adecuadas las manda pintar usando la función paintCubeStack(). El algoritmo es el siguiente:

```
1: Obtener las dimensiones de la malla.
2: for y = 0 hasta la profundidad de la malla do
3:   for x = 0 hasta el ancho de la malla do
4:     if la pila de celdas en la posición (x, y) esta seleccionada then
5:       for z = 0 hasta la altura de la malla do
6:         if la celda x, y, z es activa then
7:           obtener los valores de la curva de saturaciones de la celda x, y, z
8:           obtener las coordenadas de la celda x, y, z
9:           pintar la celda y su curva de saturaciones
10:        end if
11:      end for
12:    end if
13:  end for
14: end for
```

4.4. Clase "VisMeshR"

Esta es la clase encargada de pintar una malla radial a partir de los datos contenidos en pointsMeshR. Para pintar la malla, esta clase pinta de celda en celda, hasta completar la malla, tomando en cuenta los datos obtenidos de la clase ReferenceMesh que indica cual celda pintar y cual no, además de las caras y líneas visibles. Además de pintar una malla normal, esta clase se encarga de pintar la malla en los diferentes modos, con ayuda de las clases especializadas.

Los algoritmos de pintado se basan completamente en el orden del arreglo que contiene los datos de los vértices de cada celda, en el orden del arreglo de índices y en los arreglos de caras y de líneas visibles, por este motivo se explicarán las variables principales y

posteriormente los métodos y algoritmos principales de la clase.

4.4.1. Variables principales

a) **PointsMeshR* pointsMeshRO:** Es un objeto de la clase pointsMeshR. El cual contiene todas las coordenadas de los vértices de la malla, además de otros datos de la misma.

b) **double* rgbDA:** Es un arreglo en el cual se guarda el color con el que será pintada cada celda.

c) **double* layersZDA:** Es un arreglo que contiene las coordenadas en Z de cada capa de la malla.

d) **double** pointsCubeDA:** Es el arreglo en el cual se guardan los vértices de la celda que se va a pintar. En la Figura 4.4 se muestra la representación de una celda radial, la celda descompuesta en caras y una tabla con el formato del arreglo.

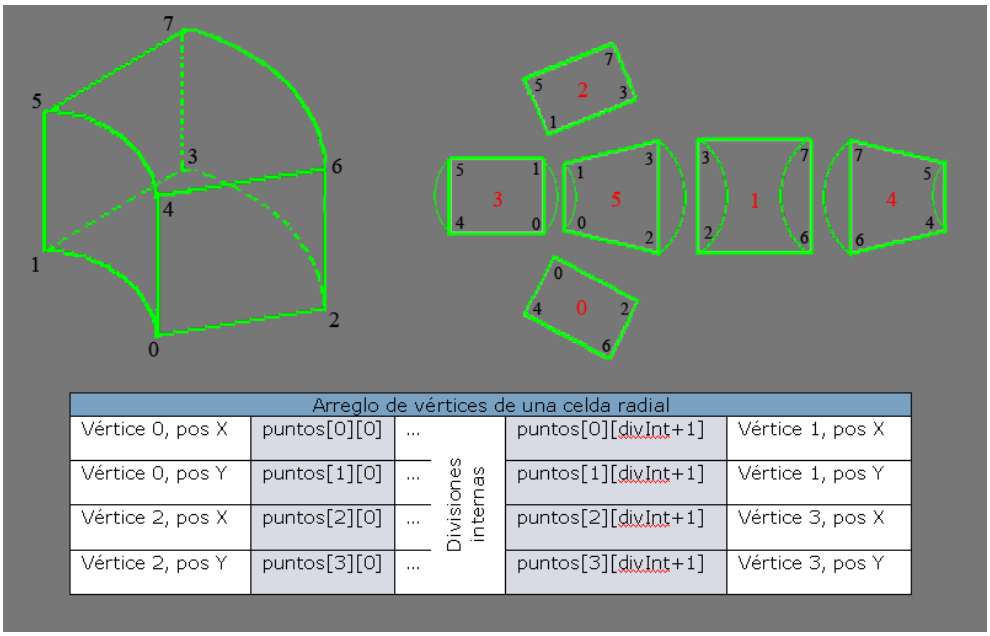


Figura 4.4: Esquema de la representación de una malla Radial.

Como se puede apreciar en la tabla de la Figura 4.4, el arreglo pointsCubeDA es en realidad una matriz de cuatro renglones por un número no fijo de columnas, este número es determinado por el número de divisiones internas que necesite la malla dependiendo del tamaño de la misma. Este número de divisiones internas es necesario para poder mostrar la celda con la curvatura característica de este tipo de mallas.

e) **bool* visibleFacesBA:** Es un arreglo de boléanos en el cual se guardan como true las caras que se podrán visualizar, y como false aquellas caras que no se pintaran.

f) **int divIntI:** Es un entero que indica el número de divisiones internas de cada celda. En la Figura 4.5 se muestra una celda radial con las divisiones internas necesarias para dar la impresión de curvatura en las cara 1 y 3.

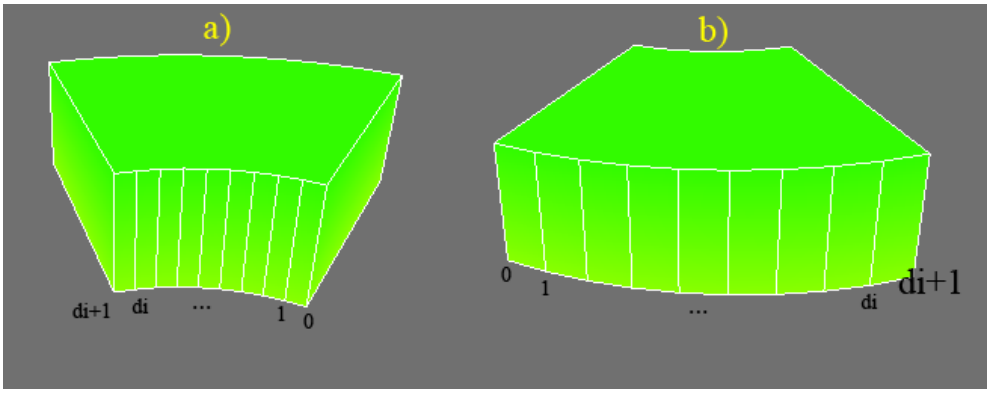


Figura 4.5: Representación de las divisiones internas de las caras 1 y 3.

4.4.2. Funciones principales

a) **int setPointsMeshR(PointsMeshR* pointsV)**

Esta función se utiliza para establecer la instancia de la clase PointsMeshR, instancia que deberá contener los vértices de las celdas de la malla. Primero se checa que el objeto sea valido, si es así, guardamos el objeto en la variable pointsMeshRO y posteriormente obtenemos de este objeto un apuntador al arreglo de las coordenadas de las capas. Finalmente obtenemos el tamaño de la malla.

b) **paintCubeLine(double r, double g, double b, double ancho, int lugZ)**

Función encargada de pintar las líneas normales de una celda radial. En la Figura 4.6 se muestra el esquema de una celda radial con la numeración de sus líneas. El algoritmo hace uso de esta numeración.

El algoritmo es el siguiente:

1: Colocar el color de las líneas.
2: Colocar el grosor de las líneas.
3: **for** la cara superior y la cara inferior **do**
4: **if** la línea 1 o 5, según corresponda, es visible **then**

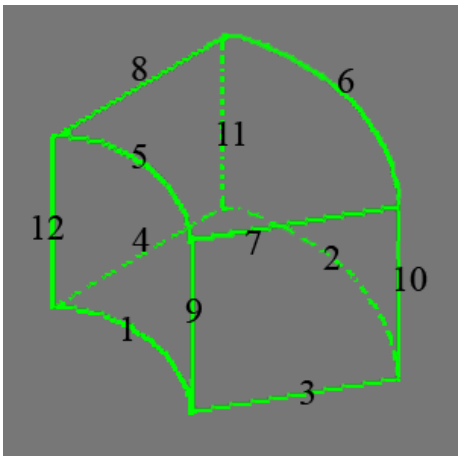


Figura 4.6: Numeración de líneas de una celda radial.

```
5:      inicia la lista de vértices
6:      for i=0 hasta divIntI+1 do
7:        añadir el vértice a la lista
8:      end for
9:      pintar los vértices de la lista como una serie de líneas unidas.
10:   end if
11:   if la línea 2 o 6, según corresponda, es visible then
12:     inicia la lista de vértices
13:     for i=0 hasta divIntI+1 do
14:       añadir el vértice a la lista
15:     end for
16:     pintar los vértices de la lista como una serie de líneas unidas.
17:   end if
18:   if la línea 3 o 7, según corresponda, es visible then
19:     pintar la línea
20:   end if
21:   if la línea 4 u 8, según corresponda, es visible then
22:     pintar la línea
23:   end if
24: end for
25: if la línea 9 es visible then
26:   pintar la línea
27: end if
28: if la línea 10 es visible then
```

```
29:    pintar la línea
30: end if
31: if la línea 11 es visible then
32:    pintar la línea
33: end if
34: if la línea 12 es visible then
35:    pintar la línea
36: end if
```

c) void paintCube(double r, double g, double b, int lugZ)

Función con la cual pintaremos la celda de la malla, esta función recibirá tres parámetros que nos representan el color que tendrá la celda. El cuarto parámetro nos indica la posición de la celda en la capa Z, utilizaremos el arreglo layersZDA y este parámetro para conocer las coordenadas en Z de la celda. El algoritmo hace referencia al número de caras utilizado en el esquema de la Figura 4.4. El algoritmo es el siguiente:

```
1: Colocar el color de la celda.
2: if la cara 1 es visible then
3:    inicia la lista de vértices
4:    for i=0 hasta divIntI+1 do
5:       añadir el vértice inferior a la lista
6:       añadir el vértice superior a la lista
7:    end for
8:    pintar los vértices de la lista como una serie de cuadros unidos
9: end if
10: Se repite de 2 a 9 para las caras 3, 4 y 5.
11: if la cara 0 es visible then
12:    pintar la cara usando los 4 vértices conocidos
13: end if
14: Se repite de 11 a 13 para la cara 2.
```

d) void paintMesh()

Esta función se encarga de crear la malla ortogonal a partir de las funciones anteriores (paintCubeLine() y paintCube()). El algoritmo es el siguiente:

```
1: for z=0 hasta el número de capas de la malla do
2:    for y=0 hasta el número de rebanadas de la malla do
3:       for x=0 hasta el número de radios de la malla do
4:          if la celda x, y, z es visible then
5:             obtener las caras visibles
```

```
6:            obtener las coordenadas de la celda x, y, z
7:            obtener el color dependiendo del modelo de color
8:            pintar la celda
9:            if esta activo el modo de selección then
10:             pintar la celda en modo de selección
11:           end if
12:         end if
13:         if las líneas de la malla son visibles then
14:           obtener las coordenadas de la celda x, y, z
15:           if las líneas de la celda son visibles then
16:             obtener las líneas visibles
17:             if esta activo el modo de selección then
18:               pintar las líneas en modo de selección
19:             else
20:               pintar las líneas de la celda
21:             end if
22:           end if
23:         end if
24:       end for
25:     end for
26: end for
```

e) **void display()**

Función encargada de gestionar la forma de pintar la malla y el pozo.

Primero coloca la rotación de la malla con ayuda de trackBall, posteriormente activa la transparencia si la opción esta habilitada. A continuación manda pintar el pozo y posteriormente pinta la malla. Si el stack de celdas esta habilitado se pinta. Al final pinta la esfera de referencia si así lo requirió el usuario y deshabilita la transparencia.

f) **int modifyThickness(int p)**

Esta función modifica la altura de las celdas. Únicamente se modifica el contenido del arreglo layersZDA por medio de la clase PointsMeshR y posteriormente se modifica la nueva altura y posición del pozo.

g) **void paintCubeStack(int h, double dz, double min, double step, int cx, int cy, int cz)**

Función encargada de pintar la curva de saturaciones de una sola celda. Este algoritmo es exactamente el mismo utilizado en la clase VisMeshO, pero solamente es aplicado en las caras 0 y 2 de la celda. La forma en como trabaja el algoritmo se muestra en la Figura 4.3. Las otras caras de la celda son pintadas normalmente pero a la altura de las caras 0 y 2.

h) void paintEmergerCell(GLenum mode)

Función encargada de mostrar la curva de saturaciones en las pilas de celdas seleccionadas por el usuario. Esta función decide cuales celdas pintar y cuales no, hace el recorrido de toda la malla y a las celdas adecuadas las manda pintar usando la función paintCubeStack(). El algoritmo es el siguiente:

```
1: Obtener las dimensiones de la malla.
2: for y=0 hasta el número de rebanadas de la malla do
3:   for x=0 hasta el número de radios de la malla do
4:     if la pila de celdas en la posición x, y esta seleccionada then
5:       for z=0 hasta el número de capas de la malla do
6:         if la celda x, y, z es activa then
7:           obtener los valores de la curva de saturaciones de la celda x, y, z
8:           obtener las coordenadas de la celda x, y, z
9:           pintar la celda y su curva de saturaciones
10:        end if
11:      end for
12:    end if
13:  end for
14: end for
```

i) int modifyWidthWells(int p)

Modifica el ancho del pozo. Esta función es particularmente delicada, pues al modificar el ancho del pozo se tiene que modificar toda la malla, puesto que la construcción de la misma se basa en el ancho del pozo. El algoritmo es el siguiente:

```
1: Obtener la escala del nuevo grosor del pozo.
2: Colocar el ancho original de la malla.
3: Obtener el ancho de una capa de la malla.
4: Crear una nueva malla con los datos nuevos.
5: Obtener las nuevas coordenadas de las capas de la malla.
6: Colocar el ancho de la malla que tenia antes de iniciar el algoritmo.
7: Colocar el nuevo ancho al pozo.
```

4.5. Clase "PointsMeshO"

Esta clase contiene las propiedades geométricas de una malla ortogonal. La malla es guardada en memoria como un arreglo de celdas, cada celda contiene un arreglo de puntos que definen sus vértices. De esta forma tenemos un arreglo de arreglos, es decir una matriz. Esta clase conserva datos como el tamaño de la malla, el centroide de la misma, los valores

mínimos y máximos en cada eje coordenado, la posición de los nodos y el tipo de los mismos. Cada que se quiere visualizar la malla o una parte de la misma, se le solicita la información necesaria a esta clase.

4.5.1. Variables principales

a) double pointsDA:** Matriz que contiene todas las coordenadas de todas las celdas.

b) int sizeXI: Es el número de celdas en el ancho de la malla, sobre el eje X.

c) int sizeYI: Es el número de celdas en el profundo de la malla, sobre el eje Y.

d) int sizeZI: Es el número de celdas en el alto de la malla, sobre el eje Z.

e) int numberOfCellsI: Es el número de celdas totales de la malla.

f) double deltaXD: Es el ancho de una celda. Cada celda de la malla tiene este ancho.

g) double deltaYD: Es el grosor de una celda.

h) double deltaZD: Es el alto de una celda.

4.5.2. Funciones principales

a) int createMeshP(int sx, int sy, int sz, double dx, double dy, double dz)

Datos de entrada: Número de celdas en X, Y, Z, ancho, alto y profundo de las celdas. La función crea un arreglo, de tamaño igual al número total de celdas, y cada entrada de este arreglo contiene un arreglo de 24 números, que indican las coordenadas de los 8 vértices de la celda. Cada celda es construida con el formato de la tabla de la Figura 4.1. De este modo tenemos una matriz de la forma:

C0	X0	Y0	Z0	X1	Y1	Z1	...	X6	Y6	Z6	X7	Y7	Z7
⋮	⋮												
CN	X0	Y0	Z0	X1	Y1	Z1	...	X6	Y6	Z6	X7	Y7	Z7

Donde C0 representa el renglón de la celda cero y CN representa el renglón de la celda N, donde N es igual a numberOfCellsI - 1. Como se puede apreciar en esta tabla, aunado a la representación de la Figura 4.1., cada celda esta constituida por 8 vértices y cada vértice esta formado por tres valores que representan su coordenada en el espacio tridimensional. El algoritmo es el siguiente:

1: **if** el arreglo de puntos tiene memoria asignada **then**
2: Liberar la memoria del arreglo

3: **end if**
4: Asignar la cantidad de memoria adecuada al arreglo.
5: A cada celda asignarle las coordenadas en la posición adecuada.

b) double* getCell(int posx, int posy, int posz)

La función devuelve un arreglo de 24 posiciones. El contenido del arreglo son los 8 vértices de la celda pedida. La celda es localizada mediante la formula:

$$pos = (int)(posx + (posy * sizeXI) + (posz * (sizeXI * sizeYI)))$$

y el arreglo devuelto es:

$$pointsDA[pos]$$

4.6. Clase "PointsMeshR"

Esta clase contiene las propiedades geométricas de una malla radial.

4.6.1. Variables principales

a) double* pointsDA:** Arreglo tridimensional donde se colocan las coordenadas de la capa inferior de la malla radial.

b) double* layersZDA: Arreglo donde se colocan las alturas de las capas.

c) int radsI: El número de radios, equivalente a la profundidad de la malla.

d) int slicesI: El número de rebanadas, equivalente al ancho de la malla.

e) int layersI: El número de capas, determina la altura de la malla.

f) int divIntI: Variable usada para saber el número de secciones internas que contendrá cada celda con el fin de simular la curvatura de las celdas. En la Figura 4.5 se muestra el ejemplo de una celda con las divisiones internas y su numeración.

g) double radIniD: Equivalente al radio del pozo. Es el radio de la celda más interna de la malla.

h) double radFinD: Es el radio de la celda más externa de la malla.

i) double deltaLayersD: Es la altura de cada celda.

4.6.2. Funciones principales

a) int createMesh(double rw, double re, int rs, int nSlices, int nLayers, double wLayers)

Función encargada de calcular las coordenadas de todas las celdas de la malla. Para utilizar la menor cantidad de memoria, sin afectar el desempeño de la visualización, solo se calculan las coordenadas de la capa inferior de la malla. Para formar la totalidad de la malla estas coordenadas son repetidas en cada capa modificando solo la coordenada Z. El cambio en la coordenada Z es tomando del arreglo layersZDA. Para poder crear la malla se usaron ecuaciones para el cálculo del tamaño del radio de cada sección, como resultado final se tiene un valor de frontera para cada radio, las ecuaciones son las siguientes:

$$\alpha = \left[\frac{re}{rw}\right]^{\frac{1}{NR}} \tag{4.1}$$

$$r_1 = \frac{\alpha * ln(\alpha)}{\alpha - 1} * rw \tag{4.2}$$

$$r_{i+1} = \alpha * r_i \tag{4.3}$$

$$F_0 = r_w \tag{4.4}$$

$$F_i = \frac{r_i - r_{i-1}}{ln(\alpha)} \tag{4.5}$$

En la Figura 4.7 se muestra una rebanada de una malla radial con la numeración de las secciones o fronteras, cada F es la calculada por la formula 4.5.

Para calcular las coordenadas de los vértices se usan senos y cosenos como se muestra en la Figura 4.8, el tamaño de la línea se va sustituyendo por el tamaño de las fronteras, es decir por F0, F1, etc.

El algoritmo para la creación de una malla radial es el siguiente:

```
1: Limpiar la memoria a utilizar.
2: Almacenar la memoria para los arreglos utilizados.
3: Calcular los radios fronteras de acuerdo a las ecuaciones dadas.
4: Calcular el número de divisiones internas de una celda.
5: Calcular el número de grados de cada rebanada y de cada división interna.
6: for di=0 hasta 360 con incremento de gradoCelda do
7:   for j=0 hasta el número de radios do
8:     for l=0 hasta el número de divisiones internas más dos do
9:       calcular los radianes del paso
10:      calcular las coordenadas del vértice usando senos y cosenos
11:    end for
```

12: **end for**
13: **end for**

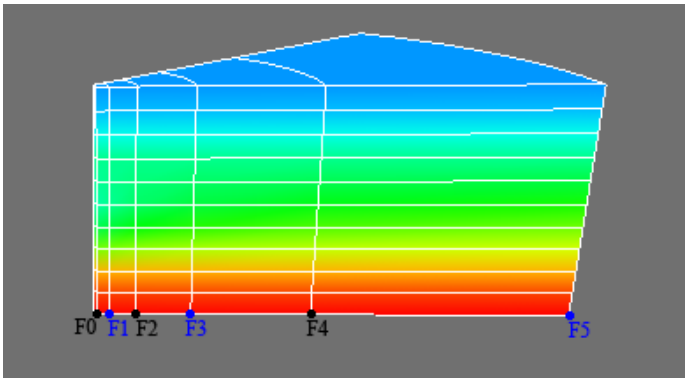


Figura 4.7: Radios fronteras en una rebanada de una malla radial.

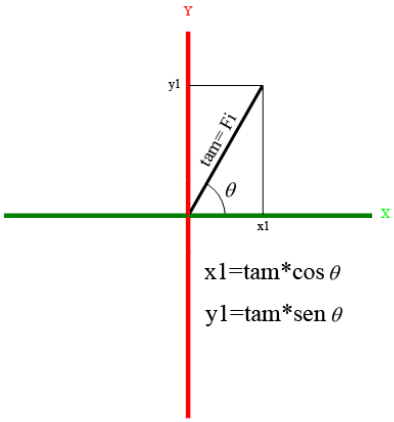


Figura 4.8: Uso de senos y cosenos

b) int setPercentLayerZ(int p)

Esta función genera las nuevas alturas de las celdas dependiendo del parámetro de escalado. El algoritmo es el siguiente:

1: Calcular la nueva altura de una celda, usando la formula: $deltaN = delta * \frac{p}{100}$.
2: Calcular la posición de la capa base, es decir, de la capa inferior.

3: Colocar el valor de la posición de la capa base en la posición cero del arreglo layersZDA.
4: **for** i=0 hasta el número de capas aplicar **do**
5: layersZDA[i+1] = layersZDA[i]+deltaN
6: **end for**

c) double getCell(int ra, int sl, int la)**

Esta función devuelve la celda de la posición pedida. Primero se calcula la posición de la celda en el arreglo pointsDA mediante la formula:

$$lugar = (sl * radsI) + ra;$$

Y posteriormente se devuelve el apuntador del arreglo en la posición calculada, es decir se devuelve:

$$pointsDA[lugar]$$

d) double* getLayersZ()

Esta función devuelve la variable $layersZDA$.

4.7. Clase "ReferenceMesh"

Esta clase crea una malla lógica que servirá de referencia para saber cuales son las celdas activas, inactivas, las visibles y las que no, además de ser la encargada de realizar la exploración de celdas internas. Para que la visualización de la malla sea más rápida, las celdas internas no son mostradas, es decir, solo se pinta el caparazón de la malla.

Para construir la malla lógica se necesita de las dimensiones de la malla y las celdas activas, con esta información se crea un arreglo tridimensional y tres arreglos que servirán para contener la información de las celdas y de los planos visibles.

La idea en la que se basan los algoritmos de esta clase es tener la información codificada dentro de los bits de la clave numérica asignada a cada tipo de celda. Los tipos de celdas, sus claves y su información codificada se muestra en la siguiente tabla:

Tipo de celda	Clave	Bits
INACTIVA	0	00000000
INT_INACTIVA	2	00000010
INT_NO_VISIBLE	4	00000100
INT_VISIBLE	5	00000101
FRONTERA_NO_VISIBLE	8	00001000
FRONTER_VISIBLE	9	00001001

Como se puede apreciar en los bits de cada tipo de celda, solo las celdas visibles cuentan con el último bit prendido. De esta forma es muy fácil saber cual es una celda visible con solo comprobar el estado de ese último bit de la siguiente forma: mesh[x][y][z]&1. Esta operación regresa 1 si es visible y cero si no.

La malla es creada al inicio como una malla llena, y después se deja hueca, colocando el tipo interno o externo a la celda según corresponda, por omisión las celdas internas no son visibles y las externas si lo son.

Los siguientes puntos resumen todos los estados de una celda por los que se puede preguntar y la forma de responder:

1. Una celda es activa si es diferente de cero.
2. Una celda es visible si su último bit esta activo, es decir es uno.
3. Las líneas de una celda son visibles si la celda lo es o si las silueta de la malla se ve y la celda es frontera, aunque la celda no sea visible.
4. Una cara de una celda es visible si la celda es activa y visible y además si su vecina es una celda frontera no visible o si es interna inactiva o si es inactiva.
5. Las líneas de una cara de una celda son visibles si la cara a la que pertenecen es visible, o si la silueta es visible y la cara tiene una vecina inactiva.

4.7.1. Variables principales

a) int* rmeshIA:** Arreglo tridimensional que contiene las claves de cada celda de la malla.

b) bool stackStatesBA:** Arreglo que guarda los estados de las pilas de celdas para el algoritmo de curva de saturaciones.

c) bool* planesVxBA: Arreglo que guarda los estados de los planos en el eje X.

d) bool* planesVyBA: Arreglo que guarda los estados de los planos en el eje Y.

e) bool* planesVzBA: Arreglo que guarda los estados de los planos en el eje Z.

4.7.2. Funciones principales

a) int setActiveCells(int w, int h, int d, int* act, char typeOfMesh, bool special = false)

Función encargada de reservar la memoria necesaria para todos los arreglos usados. Se encarga, además, de colocar las celdas activas e inactivas y de la primera clasificación de las celdas.

El algoritmo es el siguiente:

```
1: Reservar la memoria necesaria.
2: Iniciar pos=0
3: for k = 1 hasta h do
4:   for j = 1 hasta d do
5:     for i = 1 hasta w do
6:       rmeshIA[i][j][k]=act[pos++]
7:     end for
8:   end for
9: end for
10: if la malla es radial then
11:   for k = 1 hasta h do
12:     for i = 1 hasta w do
13:       rmeshIA[i][0][k]=rmeshIA[i][d][k]
14:       rmeshIA[i][d+1][k]=rmeshIA[i][1][k]
15:     end for
16:   end for
17: end if
18: for k = 1 hasta h do
19:   for j = 1 hasta d do
20:     for i = 1 hasta w do
21:       if la celda i, j, k es activa then
22:         if alguna de sus vecinas es inactiva then
23:           hacer a la celda i, j, k del tipo 9
24:         else
25:           hacer a la celda i, j, k del tipo 4
26:         end if
27:       end if
28:     end for
29:   end for
30: end for
31: Repetir de 10 a 17.
```

En este algoritmo, de la línea 2 a la 9, únicamente se copia el contenido del arreglo unidimensional que es dado como parámetro al arreglo tridimensional de la clase, dejando una capa alrededor de la malla, esta capa será de celdas inactivas y es esencial para un adecuado funcionamiento de todos los algoritmos.

De la línea 10 a la 17 y en la 31, se copia el contenido del segundo plano al último y del penúltimo al primero para simular la continuidad de una malla radial.

De la línea 18 y hasta la 30, se realiza la primer clasificación de las celdas dando como resultado el ahuecamiento de la malla, de esta forma solo serán visibles las caras externas de la malla, ahorrando una gran cantidad de tiempo al visualizar la malla.

b) int setCutBlock(int plane, int posInitial, int posFinal = -1)

Función encargada de ocultar un bloque de la malla. El bloque oculto ira a lo largo del eje pedido por *plane* e iniciará en $posInitial$ y terminara en $posFinal$. Si $posFinal$ es igual a -1, entonces se toma como posición final el máximo valor de ese eje, es decir, el tamaño de la malla en el eje *plane*. Esta función solo copia los valores a las variables adecuadas y posteriormente actualiza los cortes usando la función $updateCuts()$.

c) int setPlaneVisible(int plane, bool visible, int pos)

Esta función se encarga de colocar un plano como visible o no visible dependiendo de la opción visible. Solo copia el estado dado en la posición pedida de uno de los arreglos planesVxBA, planesVyBA o planesVzBA. Posteriormente actualiza los cortes usando la función updateCuts().

d) updateCuts()

Esta función es la encargada de realizar la exploración de las celdas internas, mediante un algoritmo de ocultamiento de celdas en bloques.

La forma en como trabaja el algoritmo es la siguiente:

Se cuenta con la posición inicia y final del corte para cada uno de los ejes o direcciones de la malla. Con esta información se hace un recorrido de la malla completa y si alguna celda activa cae fuera de estos límites se vuelve interna inactiva o frontera no visible según sea el caso. Si cae dentro de los límites la celda se vuelve frontera visible o interna visible según sea el caso.

El siguiente paso es recorrer nuevamente la malla pero solo la parte que esta dentro de los límites indicados en las posiciones iniciales y finales de los cortes. Entonces si alguna celda se encuentra en una posición igual a la de alguno de los tres arreglos que no sea visible, esa celda se vuelve interna no activa o frontera no visible según sea el caso.

En este momento se toman en cuenta las pilas de celdas seleccionadas y si alguna celda coincide con una pila, esa celda se vuelve frontera no visible o interna inactiva según sea el caso.

Ahora podemos aplicar el algoritmo de ahuecado de la malla. Para poder dejar la malla completamente hueca no servimos de un estado auxiliar no mostrado llamado estado 17, ya que es el siguiente estado que es posible usar, sus bits son los siguientes: 00100001. Como se puede ver, este estado es visible. De esta forma, si una celda se ve

y tiene alguna vecina que no se vea, comparamos si es interna no visible y entonces le colocamos el estado 17 y si no es de este tipo simplemente decimos que no se ve.

Posteriormente recorremos la malla nuevamente pero ahora solo cambiamos el estado 17 por el estado interna no visible. De esta forma no modificaremos el resultado real al hacer no visible una celda en el momento no adecuado.

A cada paso, si la malla es radial, se realiza la copia del segundo y penúltimo plano al último y primer plano. De modo que siempre se mantenga la continuidad de la malla radial.

4.8. Clase “TrackBall”

Esta clase implementa el algoritmo de la “Esfera virtual”, usado para rotar objetos en OpenGL de una forma muy intuitiva [Chen, 1988], [Projtex], [Blythe, 2005]. Este algoritmo consiste en imaginar que el objeto a rotar se encuentra dentro de una esfera de cristal y que en lugar de rotar el objeto rotamos la esfera y el objeto con ella, de esta forma podemos rotar el objeto imaginando que rotamos la esfera, lo cual resulta muy intuitivo.

En general, la forma en que trabaja el algoritmo es la siguiente:

1. Se dice cual botón del ratón será tomado para las operaciones.

2. Se dice el tamaño de la ventana sobre la que se moverá el ratón.

3. Se dice el momento en el cual se iniciará el movimiento del objeto, en nuestro caso el click del botón del ratón. Se usa la función “startMotion” para este fin.

4. De este click se toman las coordenadas al momento de presionar el botón.

5. Se va siguiendo el movimiento del ratón mediante la función “motion” y cuando se libera el botón se detienen las operaciones usando la función “stopMotion”. Conforme se va desplazando el ratón se hacen llamadas a la función “Matrix”, la cual realiza la rotación adecuada para el movimiento dado.

Cuando se llama a la función “startMotion” se realiza la proyección de la posición del ratón sobre la esfera, y se guarda esa proyección, cuando se llama a la función “motion” se realiza una nueva proyección con la nueva posición del ratón y con estas dos proyecciones se obtienen el ángulo y los ejes de rotación y se guarda la nueva proyección sustituyendo la anterior. Para cada nueva posición del ratón se realiza el mismo procedimiento hasta que sea detenido con la función “stopMotion”.

4.9. Clase "ColorLUT"

Implementa una tabla LUT (Look Up Table) de colores, una tabla LUT es básicamente una tabla de conversiones, la tabla utilizada se construye usando un color mínimo, un color máximo, y el tamaño de la tabla. Se crea un degradado de colores entre el color mínimo y máximo que llena la tabla usando el modelo de color RGB y el modelo HSI. De esta forma la tabla recibe de entrada una posición y regresa el color correspondiente.

4.9.1. Variables principales

a) **int sizeTotalTable:** Contiene el tamaño total de la tabla = (LUTSIZE*LUTSIZE)

b) **int colorModelI:** Contiene el modelo de color actual. 0=hsi, 1=rgb. El color pedido se regresara en el modelo de color indicado por esta variable.

c) **MapColor* mapColorO:** Objeto de la clase MapColor usado para llenar la tabla.

d) **double* colorMinDA:** El color mínimo actual. Es usado para llenar la tabla.

e) **double* colorMaxDA:** El color máximo actual. Es usado para llenar la tabla.

f) **double** rgbLUT:** Es el arreglo bidimensional que contiene la tabla de colores en el formato RGB.

g) **double** hsiLUT:** Es el arreglo bidimensional que contiene la tabla de colores en el formato HSI.

4.9.2. Funciones principales

a) **ColorLUT()**

Constructora de la clase. Aloja la memoria necesaria y llena las 2 tablas con los colores por omisión. El tamaño de la tabla es igual al cuadrado de LUTSIZE. Inicia el color mínimo en azul y el máximo en rojo, las 2 tablas en ceros y al final manda llamar a la función updateLUT().

b) **int setColorModel(int model)**

Coloca el modelo de color por defecto. El modelo de color es 0 para RGB y 1 para HSI.

c) **double* getColor(int pos, double* arr)**

Recibe la posición de la tabla y regresa el arreglo con el color correspondiente en el modelo de color por defecto.

Primero revisa que arr sea valido, si no lo es se crea uno nuevo. Después se llena el arreglo con ceros y se checa que la posición sea valida, si no lo es se regresa el arreglo, si lo es se copia el contenido de la posición pos de la tabla RGB o HSI dependiendo del modelo de color actual al arreglo arr y se regresa el arreglo.

d) int changeColors(double* colorMin, double* colorMax)

Modifica la tabla LUT para que el degradado de colores se encuentre entre los colores recibidos. Primero se checa que los arreglos sean validos y que el contenido se encuentre en el rango de 0 a 255. Posteriormente si el color mínimo es diferente del nuevo color mínimo se cambia ese color, y de igual forma para el color máximo. Si alguno de los dos fue modificado se llama a la función updateLUT().

e) updateLUT()

Actualiza la tabla para contener los colores correspondientes. El algoritmo es el siguiente:

1: Colocar en mapColorO como dato mínimo el valor de cero.
2: Colocar en mapColorO como dato máximo el valor de sizeTotalTable-1.
3: Colocar en mapColorO el color mínimo y máximo tomándolos de las variables de la clase.
4: **for** i=0 hasta sizeTotalTable-1 **do**
5: aux = mapColorO→getHsiMap(i, aux)
6: copiar aux en la posición i de la tabla HSI
7: aux = mapColorO→getRgbMap(i, aux)
8: copiar aux en la posición i de la tabla RGB
9: **end for**

Uso de la clase ColorLUT:

El usuario de la tabla deberá contener un dato mínimo, uno máximo y el dato que se desea convertir a color. Además deberá usar el tamaño de la tabla definido.

Con estos datos solo tendrá que usar la siguientes formulas para calcular la posición que pedirá de la tabla.

$$paso = \frac{max - min}{(LUTSIZE * LUTSIZE) - 1}$$
$$pos = \frac{dato - min}{paso}$$

Posteriormente usará la función getColor(), con la posición calculada.

4.10. Clase "MapColor"

Esta clase realiza la conversión de números a colores, en los modelos de color RGB y HSI. Para realizar esta conversión utiliza un color mínimo y un color máximo, el dato a cambiar y la cota del dato, es decir el valor mínimo y máximo donde se puede encontrar el dato.

4.10.1. Variables principales

a) double datMinD: Cota mínima para el valor del dato a convertir.

b) double datMaxD: Cota máxima para el valor del dato a convertir.

c) double* rgbMinDA: Cota mínima del rango de colores resultados de la conversión.

d) double* rgbMaxDA: Cota máxima del rango de colores resultados de la conversión.

4.10.2. Funciones principales

a) double* getRgbMap(double dato, double datmin, double datmax, double* rgbmin, double* rgbmax, double* rgbres = NULL)

Esta función hace un degradado simple entre dos colores, el mínimo y el máximo.

b) double* getHsiMap(double dato, double datmin, double datmax, double* rgbmin, double* rgbmax, double* hsires = NULL)

Esta función hace un degradado usando el modelo HSI, pasando por todos los colores que queden dentro del rango dado por los colores mínimo y máximo, el modelo HSI es una representación de la parte visible del espectro electromagnético y va desde el amarillo hasta el rojo, pasando por el verde, el cian, el azul y el magenta.

La forma en como se localiza el color correspondiente al valor dado, es similar a la realizada por la tabla LUT, se encuentra la distancia que hay entre el valor máximo y el valor mínimo y se normaliza, ahora que la distancia se encuentra en el rango de 0 a 1 se procede a localizar el color correspondiente, para el modelo de color RGB, se normaliza la distancia entre el color mínimo y el máximo y se localiza el color en la posición encontrada anteriormente, y ese color es el resultado.

Para el modelo HSI se realiza algo muy parecido, pero tomando en cuenta que los colores vienen dados por su matiz, su saturación y su brillo.

c) double* rgbToHsi(double* rgb, double* hsi = 0)

Esta función cambia un color en formato RGB a un color en el formato HSI.

Las ecuaciones usadas para realizar la conversion son proporcionadas en la sección B.4 del Apéndice B.

d) double* hsiToRgb(double* hsi, double* rgb = 0)

Esta función cambia un color en formato HSI a un color en el formato RGB.

Las ecuaciones usadas para realizar la conversion son proporcionadas en la sección B.5 del Apéndice B.

e) double degreeToRadian(double grados)

Esta función convierte los grados a radianes, utilizando la formula:

$$Radianes = \frac{grados * PI}{180{,}0}$$

f) double radianToDegree(double radianes)

Esta función convierte los radianes a grados, utilizando la formula:

$$Grados = \frac{180{,}0 * radianes}{PI}$$

4.11. Clase “Well”

Clase encargada de gestionar todo lo referente a los pozos. Cada instancia de esta clase contiene uno y solo un pozo, por lo que se tienen que crear un número de instancias de clase igual al número de pozos que contiene la malla.

4.11.1. Variables principales

a) GLUquadricObj* objs1GL: Objeto usado para pintar el cilindro que representara el pozo.

b) GLUquadricObj objs2GL:** Objetos usados para pintar dos discos a cada lado del cilindro, discos que serán las tapas de los pozos.

c) GLUquadricObj* fsGL: Objeto utilizado para pintar una parte de la flecha del pozo.

d) GLUquadricObj* tsGL: Objeto utilizado para pintar una parte de la flecha del pozo.

e) GLUquadricObj* dsGL: Objeto utilizado para pintar los disparos del pozo.

f) ManagerProp* manPropO: Objeto utilizado para extraer información referente a las propiedades del pozo.

g) bool* visibleBubbBA: Arreglo que contiene el estado de visibilidad de las propiedades del pozo.

h) char* aux: Arreglo de caracteres utilizado para imprimir adecuadamente los datos y el texto.

i) double colors:** Matriz que contiene los colores utilizados para pintar cada burbuja.

j) Bubbles bubbleOA:** Arreglo de objetos del tipo Bubbles, cada objeto pintara una burbuja.

k) QGLWidget* conector: Objeto que contiene el widget principal de la visualización, utilizado para mostrar el texto.

4.11.2. Funciones principales

a) Well(double coordX, double coordY, double coordZ, double shotX, double shotY, double shotZ, double radius, double heightWell, double widthWell, double deltaX, double deltaY, double deltaZ, int beginType, double ortho, int numberOfWell, QGLWidget* obj)

Función constructora que se utiliza para cargar los datos que se necesitan para poder pintar cada uno de los pozos. Se reciben las coordenadas del pozo, cuantos disparos va a tener en los ejes x, y, z, el radio que tiene el pozo, el alto y el ancho del pozo, así como su tipo inicial.

Primero inicia todos los apuntadores a NULL y realiza la petición de memoria adecuada para cada objeto y arreglo. Posteriormente guarda todos los parámetros en las variables adecuadas e inicia los arreglos en el estado correcto.

b) void paintWell()

Esta función es la que se encarga de pintar los pozos y realiza el llamado a las funciones encargadas de pintar los disparos y las flechas. El algoritmo de pintado es el siguiente:

1: Calcular el color del pozo de acuerdo a su tipo.
2: Realizar la traslación adecuada para colocar al pozo en su sitio.
3: Pintar el cilindro y los discos que representan al pozo.
4: Mandar pintar los disparos, la flecha y las burbujas.

c) void paintShots( )

Función que se utiliza para el pintado de los disparos de cada uno de los pozos, esta función es llamada por $paintWell()$. El algoritmo de pintado es el siguiente:

1: Colocar el color verde para pintar los disparos.
2: Realizar la rotación y la traslación adecuadas para colocar los disparos en su lugar.
3: **for** $j = 1$ hasta el número de disparos **do**
4: pintar los dos conos que representan el disparo.

```
5:    realizar la traslación adecuada para poder pintar el siguiente disparo.
6: end for
```

d) void paintArrow( )

Esta función es la que se encarga de pintar la flecha que nos indica si un pozo es extractor o productor. La dirección de la flecha varía de acuerdo al valor de la variable typeI. El algoritmo de pintado es el siguiente:

```
1: Realizar la traslación adecuada para colocar la flecha sobre el cilindro.
2: if el pozo es inyector then
3:    pintar la flecha hacia abajo
4: else
5:    pintar la flecha hacia arriba
6: end if
7: if el nombre del pozo es visible then
8:    realizar la traslación adecuada para colocar el nombre del pozo sobre la flecha
9:    pintar el nombre del pozo
10: end if
```

e) void paintBubbles( )

Función encargada de mostrar las propiedades de los pozos mediante el uso de esferas a las que denominamos burbujas. Cada esfera tiene un color y un tamaño que dependen de la propiedad que representan y del valor de esa propiedad. Esta función utiliza a objetos de la clase Bubble. El algoritmo de pintado es el siguiente:

```
1: Realizar la traslación adecuada para colocar a las esferas sobre el pozo.
2: for k = 0 hasta 8 do
3:    if la burbuja k es visible then
4:       if la propiedad de pozos k+1 es visible then
5:          pintar la burbuja k
6:          if la etiqueta de la burbuja k es visible then
7:             obtener la nomenclatura de la propiedad k+1
8:             obtener el valor de la propiedad k+1
9:             obtener la medida de la propiedad k+1
10:            pintar la etiqueta de la celda con los datos obtenidos
11:         end if
12:         realizar la traslación adecuada para poder pintar la siguiente burbuja
13:      end if
14:   end if
15: end for
```

f) int setWidthWell(int p)

Con esta función se manipula el ancho del pozo recibiendo el valor del nuevo grosor. Con este dato se calcula la escala necesaria para pintar el pozo del tamaño adecuado y se guarda esa escala.

g) void setProperties(ManagerProp * p)

Función la cual recibe un objeto de ManagerProp para poder saber las propiedades que se requiere pintar o representar con cada pozo. El objeto es guardado en la variable manPropO.

h) int setNameOfWell(QString name)

Función que establece el nombre del pozo modificando la variable nameOfWellQt.

4.12. Clase "Bubbles"

Clase encargada de pintar la esfera que representa una propiedad del pozo. Se decidió crear una clase específica para pintar una esfera, para poder realizar un degradado de colores sobre la misma, y de esta forma el usuario tenga una percepción visual de volumen.

4.12.1. Variables principales

a) double propertyD: Es la variable que indica el tamaño de la esfera, obtenido directamente del valor de la propiedad que representa la esfera.

b) double color1: La banda roja del color de la esfera.

c) double color2: La banda verde del color de la esfera.

d) double color3: La banda azul del color de la esfera.

e) double* r0: Arreglo que contiene las posiciones reales de las rebanadas de la esfera en X.

f) double* y0: Arreglo que contiene las posiciones reales de las rebanadas de la esfera en Y.

g) double* r2: Arreglo que contiene las divisiones de una esfera unitaria para el cálculo del degradado de colores.

h) double* y2: Arreglo que contiene las divisiones de una esfera unitaria para el cálculo del degradado de colores.

i) int sliceAI: El número de rebanadas de la esfera.

j) int sliceBI: El número de capas de la esfera.

4.12.2. Funciones principales

a) Bubbles(double propertyValue,double r, double g, double b)

Constructora de la clase, coloca el color de la esfera y su tamaño inicial. Reserva la memoria necesaria, posteriormente inicia los arreglos mediante el siguiente algoritmo:

1: $da = \frac{PI}{sliceAI}$
2: $db = \frac{2PI}{sliceBI}$
3: **for** $i = 0$ hasta sliceAI **do**
4: r0[i]=sin(i*da)*propertyD
5: y0[i]=cos(i*da)*propertyD
6: r2[i]=sin(i*da)
7: y2[i]=cos(i*da)
8: **end for**

Como se puede apreciar se hace uso de senos y cosenos para el calculo de la posición en (x, y) de un vector con un ángulo dado, tal y como se explica en la Figura 4.8.

b) void Bubble()

Esta función es la encargada de pintar la esfera con el color y el tamaño adecuados. La esfera es creada como un conjunto de pequeños cuadriláteros no ortogonales, en la Figura 4.9 se muestra el proceso para construir una esfera. Las líneas en las esferas a, b y c son solo para indicar como es el proceso de creación de la esfera. La sección b muestra una tercera parte de la esfera, la sección c muestra la esfera completa y la sección d muestra una esfera como se aprecia en la visualización, como se puede notar, al modificar su tamaño se pierden totalmente las secciones y se observa una esfera lisa. Las secciones e, f y g muestran la creación de un polígono en particular, cada polígono de la esfera es creado de la misma forma.

El algoritmo es el siguiente:

1: **for** $i = 0$ hasta que i sea menor que sliceAI **do**
2: x0 = 0.0
3: z0 = r0[i]
4: x2 = 0.0
5: z2 = r0[i+1]
6: **for** $j = 0$ hasta que j sea menor que sliceBI **do**
7: $x1 = r0[i] * sin((j + 1) * db)$
8: $z1 = r0[i] * cos((j + 1) * db)$
9: $x3 = r0[i + 1] * sin((j + 1) * db)$
10: $z3 = r0[i + 1] * cos((j + 1) * db)$

```
11:       calcular el color del primer vértice
12:       glVertex3d( x0, y0[i], z0 )
13:       calcular el color del segundo vértice
14:       glVertex3d( x1, y0[i], z1 )
15:       calcular el color del tercer vértice
16:       glVertex3d( x3, y0[i+1], z3 )
17:       calcular el color del cuarto vértice
18:       glVertex3d( x2, y0[i+1], z2 )
19:       x0 = x1
20:       z0 = z1
21:       x2 = x3
22:       z2 = z3
23:    end for
24: end for
```

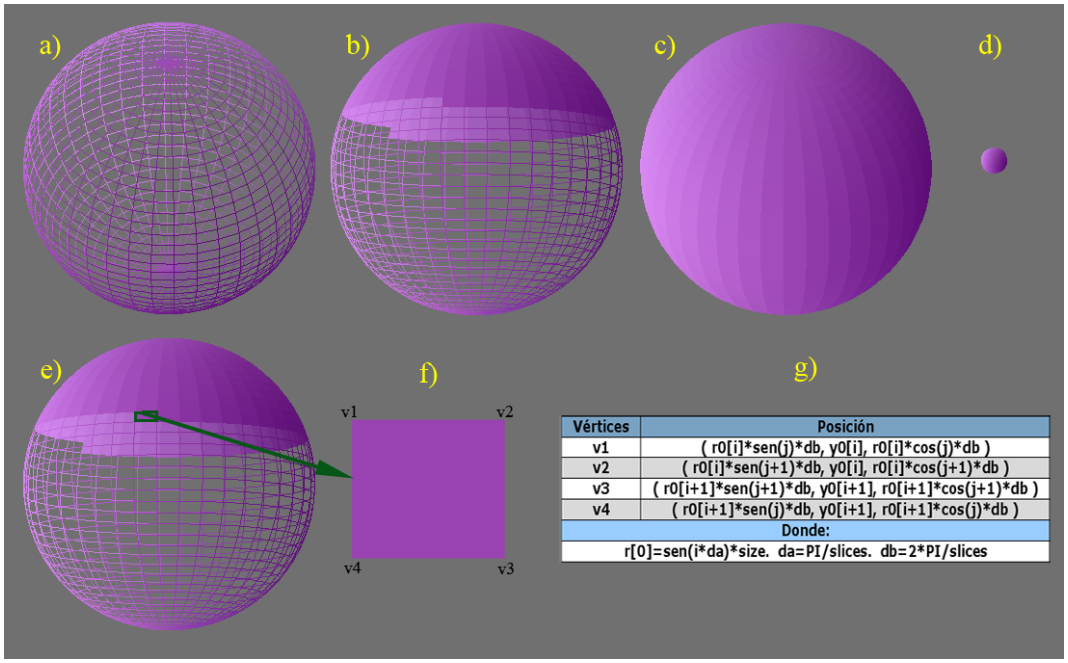


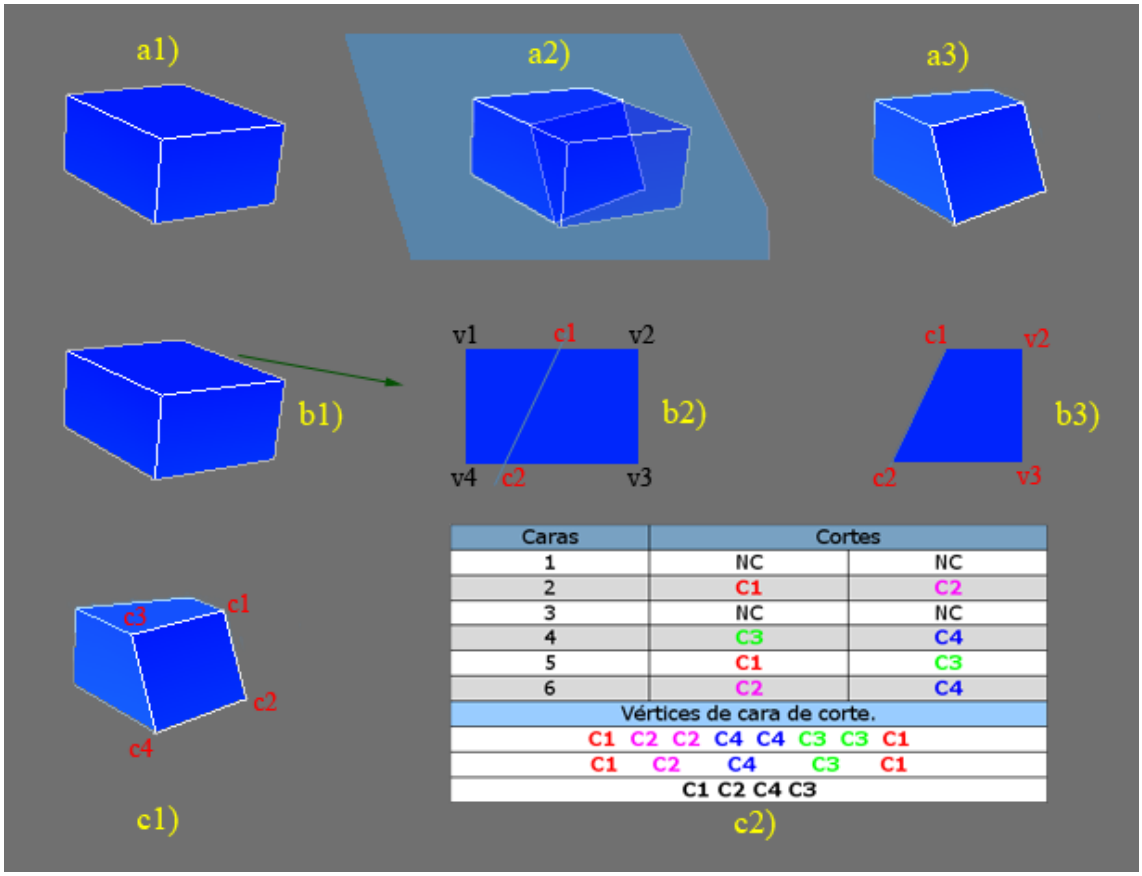
Figura 4.9: Proceso de construcción de una esfera.

c) int setProperty(double property)

Esta función modifica la variable propertyD para colocar el nuevo tamaño de la esfera.

4.13. Clase “CutPlanes”

Esta clase es la encargada de realizar el corte a una sola celda mediante el uso de un plano y de pintar el resultado. La Figura 4.10 muestra un ejemplo del proceso de corte. En el primer renglón de la imagen se muestran tres celdas, la primera es la celda original, la central es la celda con el plano de corte y la última es la celda con el corte. Para realizar éste o algún otro corte, primero se descompone la celda en caras, se toma cada cara y se localiza el corte sobre cada una de ellas. Cada cara es pintada con el corte como en la seccion b3 de la Figura 4.10. Posteriormente se pinta la cara que se encuentra sobre el plano, es decir, el corte de la celda. Ya que seguramente esta cara de corte no coincide con ninguna otra, si no que es una nueva cara, se tienen que encontrar todos sus vértices y ordenarlos. Este proceso se muestra en la tabla c2 de la Figura 4.10.



Caras	Cortes	
1	NC	NC
2	C1	C2
3	NC	NC
4	C3	C4
5	C1	C3
6	C2	C4
Vértices de cara de corte.		
C1 C2 C2 C4 C4 C3 C3 C1		
C1 C2 C4 C3 C1		
C1 C2 C4 C3		

Figura 4.10: Corte de una celda ortogonal mediante el uso de un plano.

4.13.1. Variables principales

a) ColorLUT* colorLutO: Instancia de la clase ColorLUT necesaria para aplicarle el color a la celda.

b) Point3D* pointsCutOA:** Arreglo donde se colocaran los vértices de los cortes y que formaran la cara de corte.

c) Point3D pointsFaceOA:** Arreglo de puntos que servirá para pintar cada cara que es cortada por el plano.

d) double* vertexDistDA: Arreglo que servirá para guardar la distancia de cada vértice al plano.

4.13.2. Funciones principales

a) void cutCell(double* p, double* N, double* cell, double* rgb, double alpha, bool linV)

Función principal de la clase. Recibe el punto origen del plano, el vector normal del plano, las coordenadas de la celda, el color de la celda, el nivel de transparencia y un booleano que indica si la línea de corte es visible o no. El algoritmo es el siguiente:

```
1: Iniciar los arreglos apropiadamente.
2: Calcular el tamaño del vector normal del plano.
3: for k = 0 hasta 5 do
4:   for i = 0 hasta 3 do
5:     obtener la distancia del vértice i de la cara k al plano
6:     if la distancia es menor que cero then
7:       vertexVisibleBA[i] = true
8:     else
9:       vertexVisibleBA[i] = false
10:    end if
11:  end for
12:  if los cuatro vértices son visibles then
13:    pintar la cara completa
14:  else if al menos un vértice no se ve then
15:    for cada arista en la cara do
16:      if son visibles ambos vértices then
17:        agregar ambos vértices a pointsFaceOA
18:      else if se ve alguno de los dos vértices then
19:        if se ve el primer vértice then
20:          agregar el vértice a pointsFaceOA
```

```
21:            end if
22:            localizar el corte mediante la función locatecut()
23:            agregar el vértice de corte a pointsFaceOA
24:            agregar el vértice de corte a pointsCutOA
25:            if se ve el segundo vértice then
26:              agregar el vértice a pointsFaceOA
27:            end if
28:          end if
29:        end for
30:        remover los vértices iguales de pointsFaceOA usando la función removeE-
           quals()
31:        pintar la cara mediante la función paintFace()
32:      end if
33:  end for
34:  Lamar a la función adjustFaceCut() con pointsCutOA como parámetro.
```

b) **bool locateCut(double* cell, int pos1, int pos2, double* p, double* N, double d, double* res)**

Localiza el punto de corte del plano, formado por el punto p y el vector N, y la línea que pasa por el vértice pos1 y el vértice pos2 contenidos en la celda cell. El resultado es guardado en res. La formula [Lehmann, 2002] utilizada para localizar el corte es la siguiente:

Dados:
El plano

$$P = Ax + By + Cz + D = 0$$

La línea que pasa por los puntos:

$$P1(x1, y1, z1)$$
$$P2(x2, y2, z2)$$

Obtenemos:

$$m = A * x1 + B * y1 + C * z1 + D$$
$$n = A * (x2 - x1) + B * (y2 - y1) + C * (z2 - z1)$$
$$t = -m/n$$

El punto $Q(q1, q2, q3)$ de intersección viene dado por:

$$q1 = x1 + ((x2 - x1) * t)$$
$$q2 = y1 + ((y2 - y1) * t)$$
$$q3 = z1 + ((z2 - z1) * t)$$

Si n es cero, la línea es paralela al plano o vive en el.

c) void paintFace(Point3D arr, int num, double* rgb, double alpha, bool line)**

Esta función pinta un polígono usando los vértices contenidos en el arreglo arr con el color rgb, y la transparencia alpha. El algoritmo es el siguiente:

```
1: for i = 0 hasta num-1 do
2:    usando polígonos, pintar el vértice usando el punto arr[i]
3: end for
4: if line es verdadero then
5:    for i = 0 hasta num-1 do
6:       usando líneas, pintar el vértice usando el punto arr[i]
7:    end for
8: end if
```

d) void adjustFaceCut(Point3D* arr, int* numCuts, double* rgb, double alpha, bool line, double** extraP)**

Esta función aplica el algoritmo mostrado en la tabla incluida en la Figura 4.10. Además de reordenar los elementos del arreglo arr elimina los vértices repetidos y al final manda pintar la cara resultante usando la función paintFace(). El arreglo arr contiene una matriz parecida a la tabla de la Figura 4.10, solo que si la cara no contiene cortes, el arreglo en esa posición es vació y el número de cortes en esa posición es cero. El algoritmo es el siguiente:

```
1: Localizar la primera cara que contenga cortes.
2: Copiar el primer punto dentro de pointsFaceOA. Llamaremos a este punto PO
3: Eliminar del arreglo el punto copiado.
4: repeat
5:    Localizar el punto PO dentro de arr
6:    if el punto fue localizado then
7:       borrar el punto y recordar la cara en que fue localizado.
8:       tomar el siguiente punto de esa cara y llamarlo PO
9:       copiar a PO dentro de pointsFaceOA
10:   end if
11: until se sigan localizando puntos dentro de arr
12: Pintar la cara utilizando la función paintFace()
```

e) int removeEquals(Point3D arr, int num, double* extrap)**

Esta función realiza una comparación de un punto contra todos los demás, si este punto es igual a otro, se borra este último. Y se realiza lo mismo para cada uno de los puntos. El algoritmo es el siguiente:

1: Crear un arreglo auxiliar del mismo tamaño que arr.
2: Copiar el primer elemento dentro del arreglo auxiliar.
3: **for** $j = 1$ hasta num-1 **do**
4: **if** el elemento arr[j] es diferente de todos los elementos del arreglo aux **then**
5: copiar arr[j] dentro de aux
6: **end if**
7: **end for**

4.14. Clase "Point3D"

Esta clase crea un punto en el espacio.

4.14.1. Variables principales

a) double x: La posición del punto en la coordenada x.

b) double y: La posición del punto en la coordenada y.

c) double z: La posición del punto en la coordenada z.

4.14.2. Funciones principales

a) Point3D(double px, double py, double pz)

Constructora de la clase. Crea un punto con coordenada (px, py, pz).

b) void setX(double px)

Coloca la coordenada x del punto al valor px.

c) void setY(double py)

Coloca la coordenada y del punto al valor py.

d) void setZ(double pz)

Coloca la coordenada z del punto al valor pz.

e) double getX( )

Regresa el valor de la coordenada x del punto.

f) double getY( )

Regresa el valor de la coordenada y del punto.

g) double getZ( )

Regresa el valor de la coordenada z del punto.

h) bool equal(Point3D* p)

Indica si dos puntos se encuentran en la misma posición con un error de una decima de millón. Es decir para que dos puntos sean "iguales" no es necesario que tengan exactamente el mismo número, se realiza de esta forma para evitar errores por redondeo.

1: $doublee = 0{,}0000001$
2: **if** $x - e > p.x || p.x > x + e$ **then**
3: return false
4: **end if**
5: **if** $y - e > p.y || p.y > y + e$ **then**
6: return false
7: **end if**
8: **if** $z - e > p.z || p.z > z + e$ **then**
9: return false
10: **end if**
11: return true

4.15. Clase "Plane3D"

Esta clase define un plano y funciones para calcular la distancia de un punto al plano, de acceso y modificación de sus elementos. Un plano queda definido por un punto y un vector normal o por tres puntos diferentes [Lehmann, 2002]. La ecuación general del plano es: $Ax + By + Cz + D = 0$
Donde (x, y, z) es un punto sobre el plano y ABC es el vector normal al plano. El coeficiente D puede ser calculado mediante: $D = -Ax * -By * -Cz$

4.15.1. Variables principales

a) double D: Coeficiente D del plano.

b) double* pA: Arreglo que contiene el punto sobre el plano.

c) double* normal: Arreglo que contiene el vector normal al plano.

4.15.2. Funciones principales

a) Plane3D(double px, double py, double pz, double nx, double ny, double nz)

Constructora que genera un plano a partir de un punto sobre el plano y el vector normal al plano. Reserva la memoria necesaria y copia los valores del punto y el vector en el arreglo correspondiente. Finalmente calcula el coeficiente D.

b) Plane3D(double px1, double py1, double pz1, double px2, double py2, double pz2, double px3, double py3, double pz3)

Constructora que genera un plano a partir de tres puntos sobre el mismo. Reserva la memoria necesaria. Toma el primer punto como el punto sobre el plano. Posteriormente calcula el vector normal usando los tres puntos y el coeficiente D de la siguiente forma:

$$\begin{aligned} A &= (py1 * (pz2 - pz3)) + (py2 * (pz3 - pz1)) + (py3 * (pz1 - pz2)) \\ B &= (pz1 * (px2 - px3)) + (pz2 * (px3 - px1)) + (pz3 * (px1 - px2)) \\ C &= (px1 * (py2 - py3)) + (px2 * (py3 - py1)) + (px3 * (py1 - py2)) \\ D &= (px1 * ((py2 * pz3) - (py3 * pz2))) + (px2 * ((py3 * pz1) - \\ &\quad (py1 * pz3))) + (px3 * ((py1 * pz2) - (py2 * pz1))) \end{aligned}$$

c) double distPointToPlane(double px, double py, double pz)

Esta función calcula la distancia de un punto al plano. La distancia de un punto al plano es:

$$\frac{|AP*n|}{|n|}$$

Donde $AP = P - A$, P es el punto dado, A es un punto sobre el plano, y n es la normal al plano. En esta función se desarrolla la formula del plano, es decir, dado $P = (x, y, z)$ se encuentra:

$$val = Ax + By + Cz + D$$

Este valor es el resultado regresado. Aunque no es exactamente la distancia, para el fin deseado es equivalente. El código es el siguiente:

```
1: return ( ( px * normal[0] ) + ( py * normal[1] ) + ( pz * normal[2] ) + D )
```

4.16. Clase "IsolinesMO"

Esta clase crea las isolíneas de una propiedad, medio y capa de la malla seleccionadas por el usuario. Las isolíneas muestrean los valores de la propiedad y crean un número de rangos igual al número de contornos, por cada contorno se genera una Isolínea de un color, en realidad se genera un isocontorno lleno de un color especifico.

4.16.1. Variables principales

a) **int propertyI:** La propiedad a la que se le van a obtener las isolíneas.

b) **int typeOfRockMassI:** El medio seleccionado.

c) **int stepI:** El paso de simulación seleccionado.

d) **int layerI:** La capa de la malla de la que se extraerán los datos.

e) **int contournsI:** El número de contornos deseados.

f) **int* activeCellsIA:** Un arreglo con las celdas activas e inactivas de la malla.

g) **double minValueD:** El valor mínimo de la propiedad.

h) **double maxValueD:** El valor máximo de la propiedad.

i) **double* colorMinDA:** Un arreglo con el color mínimo en formato RGB.

j) **double* colorMaxDA:** Un arreglo con el color máximo en formato RGB

k) **double* pointsXDA:** Arreglo con las posiciones en X de los vértices de la malla.

l) **double* pointsYDA:** Arreglo con las posiciones en Y de los vértices de la malla.

m) **double* contournsPDA:** Arreglo que contiene los rangos de cada contorno.

n) **double** meshODA:** Arreglo que contiene los datos actuales usados para calcular los contornos.

o) **double** meshRDA:** Arreglo auxiliar que contiene los datos de la capa usados para calcular los contornos.

p) **double** pointsXYDA:** Matriz que contiene los vértices de la capa actual.

q) **double* rgbDA:** Arreglo que contendrá el color en formato RGB.

r) **PointsMeshO* pointsO:** Instancia de la clase PointsMeshO necesaria para acceder a los vértices de la malla.

s) **ManagerProp* managerPropO:** Instancia de la clase ManagerProp necesaria para acceder a los datos de las propiedades.

t) **ColorLUT* colorLutO:** Instancia de la clase ColorLUT necesaria para obtener el color de un dato determinado.

4.16.2. Funciones principales

a) IsolinesMO(PointsMeshO* p, ManagerProp* m, ColorLUT* cl)

Constructora de la clase. Copia las instancias de los objetos. Reserva la memoria de todos los arreglos usados e inicia los valores de los mismos.

b) void setStep(int s)

Cambia el paso de simulación. Posteriormente manda llamar a la función searchInfo().

c) setProperty(int p)

Cambia la propiedad seleccionada. Posteriormente manda llamar a la función searchInfo().

d) setLayerVisible(int l)

Cambia la capa seleccionada. Posteriormente manda llamar a la función searchInfo().

e) setNumOfContourns(int cn)

Cambia el número de contornos que serán generados. Libera la memoria del arreglo de contornos y enseguida reserva la memoria necesaria para contener el nuevo tamaño del arreglo. Posteriormente inicia cada valor del arreglo mediante el siguiente algoritmo:

1: calcular $paso = \frac{maxValueD - minValueD}{cn-1}$
2: **for** $i = 0$ hasta $cn - 1$ **do**
3: contournsPDA[i]=minValueD+(paso*i)
4: **end for**

f) void updatePoints(PointsMeshO* p)

Esta función actualiza la instancia de la clase que maneja los puntos y recalcula la posición y el tamaño de las isolíneas.

g) void paint()

Función encargada de mostrar todo lo relacionado con las isolíneas. Calcula el rango de pintado de las isolíneas y manda llamar a la función conFill().

h) void searchInfo()

Función encargada de actualizar los datos necesarios para mostrar adecuadamente las isolíneas.

Obtiene los datos de la capa actual, con el adecuado paso, propiedad y medio. Posteriormente obtiene el valor mínimo y el valor máximo de la propiedad actual, así como

su color mínimo y máximo. Finalmente llena el arreglo de contornos usando el mismo algoritmo que el usado en la función setNumOfContourns().

i) double getPoint(double x1, double x2, double p1, double p2)

Función usada en la función conFill(). Regresa la posición correspondiente según la coordenada x1 y x2 con la distancia $p1$ y $p2$ usando coordenadas baricéntricas. La ecuación usada es la siguiente:

$$\frac{(p2*x1)-(p1*x2)}{p2-p1}$$

j) void conFill(double **d, int ilb, int iub, int jlb, int jub, double *x, double *y, int nc, double *cont)

Encuentra las isolíneas de la matriz de datos d en el rango de índices de ilb, iub en X, y jlb, jub en Y. Mostrando la información en la coordenadas x e y.

El número de contornos a encontrar son nc y se encuentran ordenados de menor a mayor en $cont$.

En la Figura 4.11 se muestra la forma en que trabaja el algoritmo de isolíneas.

Lo primero que realiza el algoritmo es crear una capa con los datos en los vértices de las celdas, utilizando la capa original que tiene los datos en el centro de cada celda, como se muestra en las secciones a y b de la Figura 4.11.

Posteriormente para cada celda se aplica el siguiente proceso.

Se toma la celda y se divide en cuatro triángulos, como en la sección c, el dato del centro de la celda es encontrado promediando los cuatro datos originales. Posteriormente para cada triángulo se buscan todos los contornos que se encuentren en el triángulo. Finalmente, para cada contorno se sigue el siguiente proceso. Se toma una línea del triángulo y se obtienen los puntos por donde pasa el contorno, utilizando la función getPoint() y los casos mostrados en la sección h. Para cada línea se obtienen los puntos del contorno y se pinta el polígono que se genere con todos los puntos obtenidos de las tres líneas del triángulo. En la sección f de la Figura 4.11 se muestran varios triangulos con uno, dos y tres contornos. Al pintar en cada triángulo los contornos encontrados se generan los isocontornos deseados.

El algoritmo es el siguiente:

```
1: Calcular el paso para los colores de cada contorno.
2: for cada celda en la malla do
3:   if los cuatro vértices contienen información then
4:     calcular el valor mínimo y máximo de los valores de los cuatro vértices
5:     for cada contorno do
```

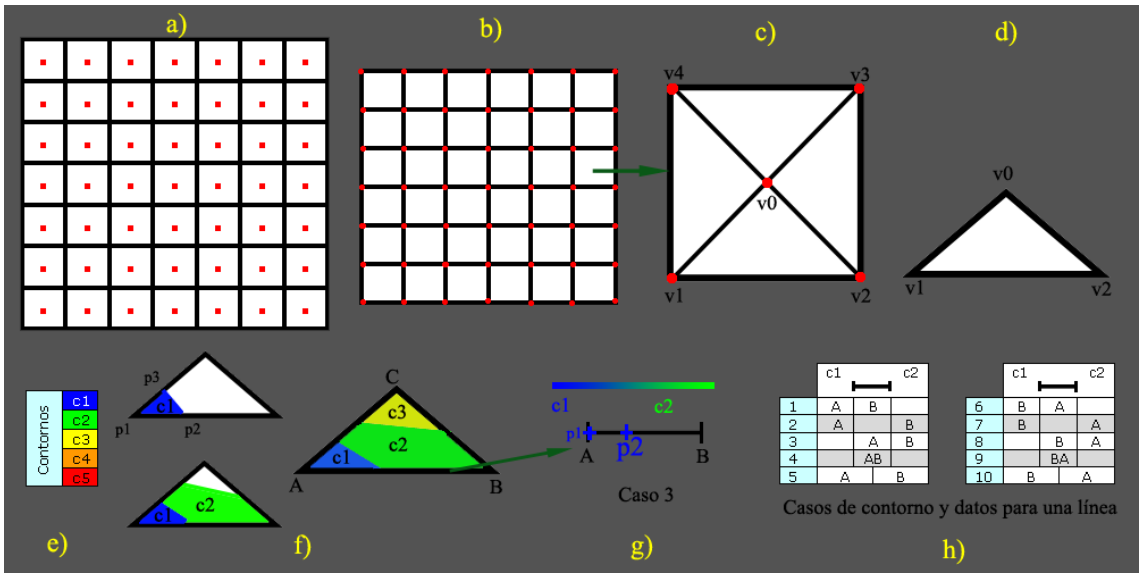


Figura 4.11: Construcción de Isolíneas.

```
6:        if hay algún dato en el contorno actual (usar min, max y el contorno
          actual) then
7:          obtener el color del contorno actual
8:          for cada triángulo en la celda actual do
9:            for cada línea del triángulo actual do
10:             encontrar los puntos por donde pasa el contorno
11:           end for
12:           pintar el polígono con los puntos encontrados
13:         end for
14:       end if
15:     end for
16:   end if
17: end for
```

4.17. Clase “ManagerProp”

Clase que contiene todos los datos que serán visualizados. Se encarga de gestionar toda la información que utilizará el módulo de visualización 3D. Además de ser usada por el módulo de visualización 3D, ésta clase también es usada por el módulo de graficas.

Esta clase contiene todos los datos que serán visualizados, así como los colores de las

propiedades, y todos los datos necesarios para una correcta visualización. La mayoría de los datos son obtenidos de SharedMem y se les da un orden, este orden es esencial para que el usuario pueda elegir la propiedad a visualizar, el medio, el paso y todas las opciones disponibles.

La clase maneja una tabla de propiedades por malla y por pozo, una tabla de posiciones para esas propiedades, una tabla de nombres, una de códigos, una de medidas, una de colores, una de valores mínimos y una de máximos. Cada una de estas tablas lleva un orden específico y coherente con todas las demás. La tabla de posiciones es la que realiza el mapeo de los datos obtenidos de sharedMem con la posición correcta que usará la visualización.

La clase también maneja toda la información de los pozos y las celdas activas.

La clase contiene todas las funciones necesarias para acceder a la información disponible, y modificar la información que puede ser modificada, si algún dato de esta clase es erróneo la visualización no será coherente o no se mostrara.

4.17.1. Variables principales

a) **SharedMem* dataShO:** Instancia de la clase SharedMem. De esta clase se obtienen los apuntadores de los arreglos que contienen la información, además de todos los datos indispensables para la construcción del módulo.

b) **QString* codeOfPropsSA:** Los códigos de las propiedades de la malla, por ejemplo PO.

c) **QString* namesOfPropsSA:** La traducción de los códigos a nombres en el mismo orden, por ejemplo Presion.

d) **QString* codeOfPropWellsSA:** Los códigos de las propiedades de los pozos.

e) **QString* namesOfPropWellsSA:** La traducción de los códigos a nombres.

f) **QString* namesOfTypesOfRockMassSA:** El nombre de los medios.

g) **double** colorsDA:** El arreglo que contiene el color mínimo y máximo con el que se representará cada propiedad.

h) **double** propertiesOfWellsDA:** Matriz que contiene todos los datos de todas las propiedades de los pozos.

i) **double** propertiesOfMeshDA:** Matriz que contiene todos los datos de todas las propiedades de la malla.

j) **int* activeCellsIA:** Arreglo unidimensional que contiene que celdas son activas y cuales son inactivas.

k) int* typesOfWellsArrayDA: Arreglo que contiene el tipo de pozo para cada pozo en todos los pasos.

l) int totalStepsI: El número de pasos totales de simulación.

m) int actualStepsI: El número de pasos que se han simulado. En una simulación que es cargada, esta variable debe ser igual a totalStepsI.

n) int actualPropsI: El número de propiedades de la malla que pueden ser visualizadas.

o) int actualPropsOfWellsI: El número de propiedades de los pozos que pueden ser usadas.

p) int numberOfWellsI: El número de pozos que existen en la simulación.

q) int actualTypesOfRockMassI: El número de medios que existen en la simulación.

r) int numberOfCellsI: El número de celdas totales de la malla.

s) int numberOfActiveCellsI: El número de celdas activas de la malla.

t) int sizeXI: El número de celdas en el eje X.

u) int sizeYI: El número de celdas en el eje Y.

v) int sizeZI: El número de celdas en el eje Z.

w) double deltaXD: El tamaño de una celda sobre el eje X.

x) double deltaYD: El tamaño de una celda sobre el eje Y.

y) double deltaZD: El tamaño de una celda sobre el eje Z.

z) char typeOfMeshC: El tipo de malla que se esta simulando.

Esta clase contiene una gran cantidad de funciones, casi en su totalidad son para colocar y extraer datos. Estas funciones solo regresan el dato pedido o realizan la asignación debida, por lo que colocaremos solo las funciones más representativas o aquellas que presenten algún algoritmo de alguna dificultad.

4.17.2. Funciones principales

a) int setSharedMem(SharedMem* data)

Coloca la instancia de la clase SharedMem. Toma los datos necesarios de la instancia e inicia todas las variables usadas. El número de pasos actuales es iniciado en -1, si el proyecto es cargado de uno ya simulado, se tiene que colocar el número de pasos

al total, pero si la visualización es en tiempo real se tiene que ir colocando cada paso conforme se va simulando mediante el uso de la función setNumberOfSteps(). Lo que realiza esta función se explica con más detalle a continuación:

1: Limpia todas las variables y arreglos.
2: Inicia todas las variables a valores por defecto y los arreglo a NULL.
3: Pide y guarda el número de pasos de simulación, el número de medios, el número de pozos y el tipo de malla.
4: Pide y guarda el tamaño de la malla. Es decir, el número de celdas en X, Y y Z.
5: Pide y guarda el tamaño de una celda.
6: Reserva memoria para todos los arreglos necesarios.
7: Copia las celdas activas e inactivas y cuenta las celdas activas.
8: Copia los datos de los pozos.
9: Inicia los arreglos de posiciones a -1 y los arreglos de mínimos y máximos a DOUBLE_MAX_VIS y a DOUBLE_MIN_VIS correspondientemente.
10: Pide cada una de las propiedades de la malla y coloca el apuntador devuelto en la posición correcta en el arreglo propertiesOfMeshDA. Cuenta el número de propiedades validas.
11: Pide cada una de las propiedades de los pozos y coloca el apuntador devuelto en la posición correcta en el arreglo propertiesOfWellsDA. Cuenta el número de propiedades validas.
12: Calcula los valores mínimos y máximos para cada una de las propiedades de la malla y de los pozos y guarda los resultados en el arreglo adecuado, en la posición correcta.

b) int setNumberOfSteps(int actualSteps)

Coloca el número de pasos simulados, en caso de que la simulación sea en tiempo real, en cada paso simulado se tendrá que hacer una llamada a esta función. En caso de que la simulación se cargue de un archivo, basta con realizar una llamada con el número total de pasos de simulación inmediatamente después de llamar a setSharedMem. Después de guardar el dato dado calcula los valores mínimo y máximo para cada una de las propiedades de la malla y de los pozos.

c) int* copyActiveCells(int* data = NULL)

Esta función copia las celdas activas e inactivas dentro del arreglo data y regresa el mismo apuntador. El algoritmo es el siguiente:

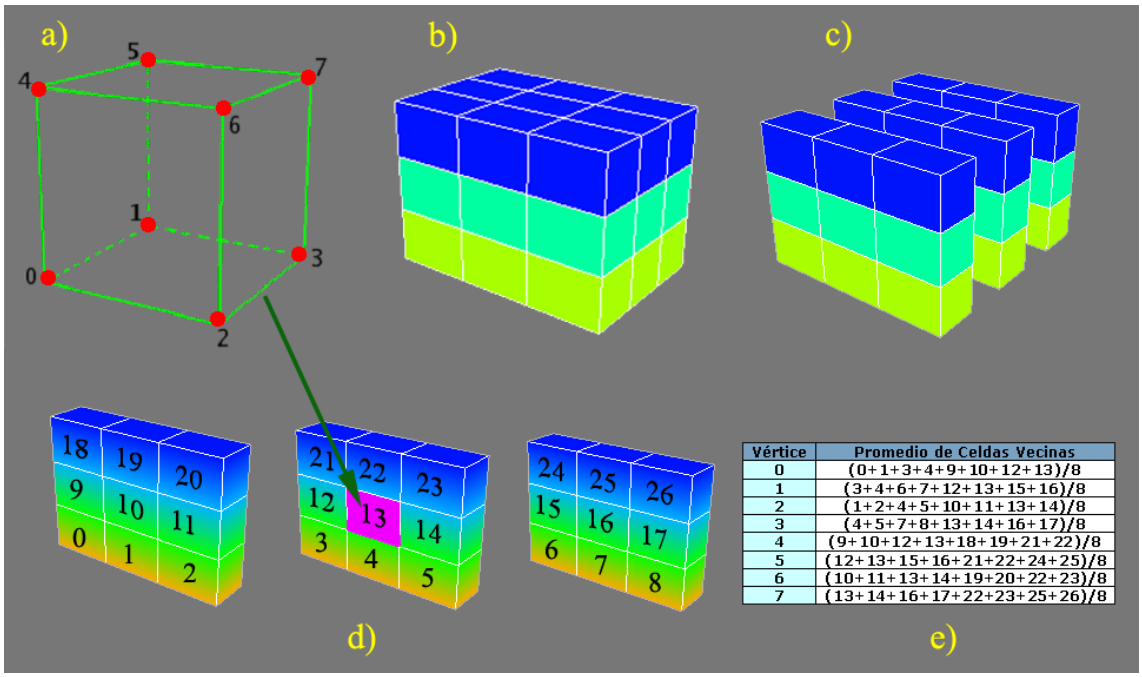
1: **if** data es nulo **then**
2: reservar memoria para el arreglo data
3: **end if**
4: **for** $i = 0$ hasta el número de celdas menos uno **do**

5: data[i]=activeCellsIA[i]
6: **end for**

d) **double* getValuesExtrapCell(int prop, int rockMass, int step, int cx, int cy, int cz, double* arr = NULL)**

Esta función regresa los datos extrapolados o interpolados de cada nodo de la celda en la posición (cx, cy, cz), en una determinada propiedad, medio y paso.

En la Figura 4.12 se muestra una celda y sus 26 vecinas. La sección *a* muestra la numeración de los vértices de la celda. Las secciones *b* y *c* muestran las celdas con el color original, la sección *c* muestra las celdas separadas por rebanadas. En la sección *d* se muestra cada rebanada con la numeración usada para calcular el promedio de valores para cada vértice de la celda 13. El color de las celdas de la sección *d* es el que tienen las celdas al aplicarles la interpolación de datos. Finalmente, la sección *e* muestra una tabla con el vértice y la manera de calcular el valor que le corresponde.



Vértice	Promedio de Celdas Vecinas
0	(0+1+3+4+9+10+12+13)/8
1	(3+4+6+7+12+13+15+16)/8
2	(1+2+4+5+10+11+13+14)/8
3	(4+5+7+8+13+14+16+17)/8
4	(9+10+12+13+18+19+21+22)/8
5	(12+13+15+16+21+22+24+25)/8
6	(10+11+13+14+19+20+22+23)/8
7	(13+14+16+17+22+23+25+26)/8

Figura 4.12: Vecinas de una celda determinada.

El algoritmo es el siguiente:

1: Obtener los rangos validos de la posición de la celda y de sus vecinas.

2: Obtener el valor del dato de la celda (cx, cy,cz).
3: Copiar el valor de la celda a todo el arreglo arrExtrapAuxDA.
4: Obtener el valor de cada una de las celdas vecinas de la celda pedida y colocarlo dentro del arreglo arrExtrapAuxDA en la posición adecuada.
5: **for** cada uno de los vértices de la celda **do**
6: sumar el valor del dato de sus ocho celdas vecinas
7: dividir entre ocho el resultado de la suma
8: guardar el resultado dentro del arreglo arr en la posición adecuada
9: **end for**

e) double getValueCell(int prop, int rockMass, int cx, int cy, int cz, int step)

Regresa el valor de la propiedad prop del medio rockMass en el paso step para la celda (cx, cy, cz).

Primero se localiza la posición de la propiedad dentro del arreglo de datos usando el arreglo de posiciones, de la siguiente forma:
$lugar = positionOfPropsIA[prop]$

Posteriormente se accede a la posición pedida dentro de la matriz de la siguiente forma:

$$return\ propertiesOfMeshDA[lugar][(step * numberOfCellsI * actualTypesOfRockMassI) + (numberOfCellsI * rockMass) + (cz * sizeXI * sizeYI) + (cy * sizeXI) + cx]$$

Esto debe ser realizado de esta forma ya que el arreglo que contiene los datos de una propiedad es unidimensional, es decir, el arreglo contiene los datos de toda la malla para todos los medios y todos los pasos. La forma de guardar los datos dentro del arreglo se muestra gráficamente en la Figura 4.13.

f) double getValuePropWell(int prop, int well, int step)

Regresa el valor de la propiedad prop, del pozo well y el paso step.

Se accede al valor de la propiedad dentro de la matriz de datos de pozos de la siguiente forma:
$propertiesOfWellsDA[positionOfPropWellsIA[prop]][(well * totalStepsI) + step]$

La estructura de la matriz de datos de pozos es muy similar a la matriz de datos de propiedades de la malla. Se tiene un arreglo de propiedades y cada posición de este arreglo apunta a un arreglo unidimensional con todos los datos de esa propiedad. Este arreglo contiene para cada pozo los valores de la propiedad para todos los pasos y posteriormente los valores del siguiente pozo para todos los pasos y así sucesivamente.

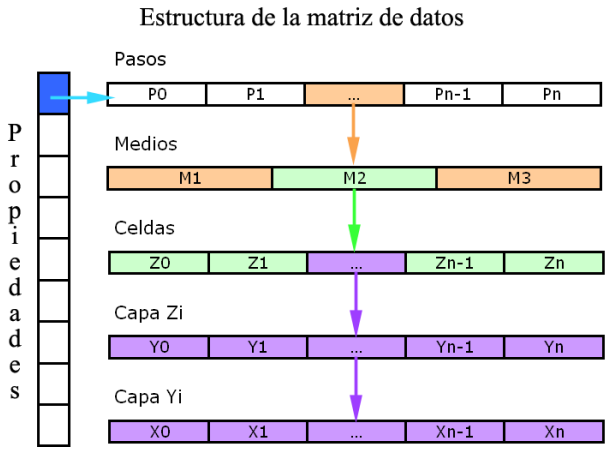


Figura 4.13: Estructura del arreglo que contiene los datos de una propiedad.

g) **double* getValuesStack(int rockMass, int cx, int cy, int cz, int step, double *arr)**

Esta función devuelve los valores A, B, C, D, E y F de la Figura 4.3, es decir, devuelve los valores de las saturaciones de agua y aceite usadas para dibujar la curva de saturaciones en la pila de celdas.

Los valores A, B y C corresponden a los valores de la saturación de agua de las celdas $(cx, cy, cz+1)$, (cx, cy, cz) y $(cx, cy, cz-1)$ en ese orden. Los valores D, E y F corresponden a los valores de la saturación de aceite de las celdas $(cx, cy, cz+1)$, (cx, cy, cz) y $(cx, cy, cz-1)$ en ese orden.

Capítulo 5

Resultados

En los capítulos precedentes hemos visto los requerimientos para el módulo de visualización tridimensional, el modelado de las funcionalidades y finalmente el diseño detallado de los algoritmos que las implementan. En este capítulo se expondrán los resultados de las funcionalidades y algoritmos diseñados. Ya que los resultados son propiamente visuales, estos son mostrados mediante imágenes. Además de las imágenes se ofrece una breve explicación de la funcionalidad o algoritmo al cual se hace referencia, explicación a modo de resumen de lo que se vió en los capítulos anteriores.

Las funcionalidades se encuentran organizadas, como en todos los demás capítulos, siguiendo el orden dado en los requerimientos funcionales del capítulo 2 y agrupadas por funcionalidad.

5.1. Interacción con la malla

Para rotar la malla se implementó el método más intuitivo encontrado en la literatura especializada, dicho método es llamado la esfera virtual.

La idea de este método es proyectar los movimientos del ratón sobre una esfera hipotética que llena la ventana 3D, de esta forma el usuario puede imaginar ver un objeto encerrado en una esfera de cristal, la rotación es entonces realizada balanceando o haciendo rodar la esfera y consigo al objeto 3D. Los movimientos arriba y abajo e izquierda y derecha en el centro del circulo(o equivalentemente, de la malla), producen rotaciones en los ejes X y Y respectivamente. Mientras que mover el ratón alrededor de la esfera (lo mas alejado de la malla), produce rotaciones alrededor del eje Z. Gracias a estos movimientos, la malla puede ser rotada muy fácil e intuitivamente, logrando una mejor experiencia para el usuario.

La Figura 5.1 muestra una malla en diferentes posiciones, encerrada por una esfera. Esta esfera es la que el usuario puede imaginar al momento de realizar las rotaciones. Cabe señalar que la esfera de referencia puede ser mostrada a criterio del usuario.

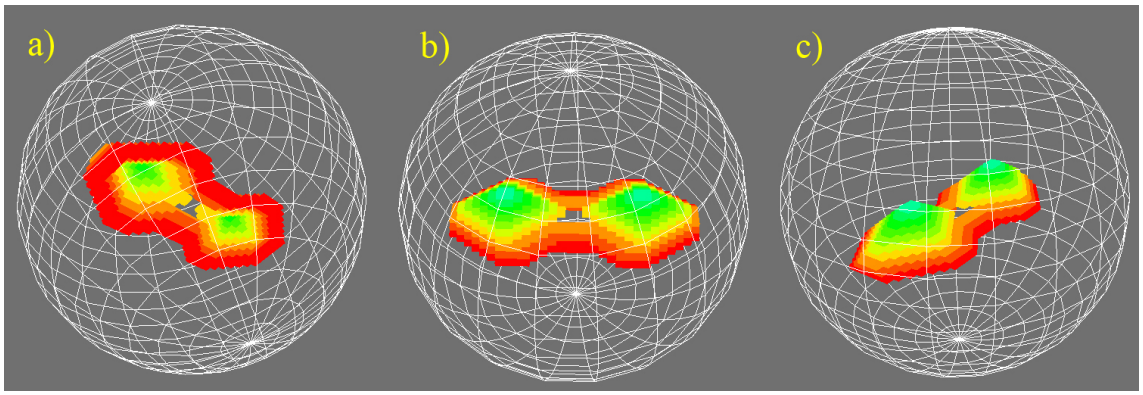


Figura 5.1: Esfera de referencia para las rotaciones.

Para escalar la malla basta con presionar el botón derecho del ratón y realizar un movimiento ascendente, si lo que se desea es alejar la malla; y un movimiento descendente, si lo que se busca es acercar la malla.

La translación de la malla se realiza presionando el botón central del ratón o en su defecto los botones, izquierdo y derecho, al mismo tiempo, deslizando el ratón hacia la dirección que se desea trasladar la malla. La Figura 5.2 muestra una malla que ha sido trasladada y escalada.

Se cuentan con una serie de vistas predeterminadas de la malla, con las cuales se pueden apreciar rápidamente los principales puntos de vista del yacimiento. Así mismo se cuenta con la opción de colocar la malla en su ángulo inicial. La Figura 5.3 muestra una malla con cada una de las vistas disponibles.

Para poder mostrar información de una celda particular se creo la funcionalidad de selección, cuando esta activo el control de esta funcionalidad, el usuario puede elegir cualquier celda del yacimiento mediante un doble clic del ratón sobre la celda deseada, mostrando datos específicos de la celda. La Figura 5.4 muestra la selección de una celda y la información desplegada.

5.2. Opciones de visibilidad

Se dispone de una opción para habilitar y deshabilitar las líneas de la malla, con lo cual se puede obtener una apreciación total y continua del yacimiento. De la misma forma se puede elegir mostrar u ocultar los pozos. La Figura 5.5 muestra estas opciones aplicadas tanto a una malla ortogonal como a una radial.

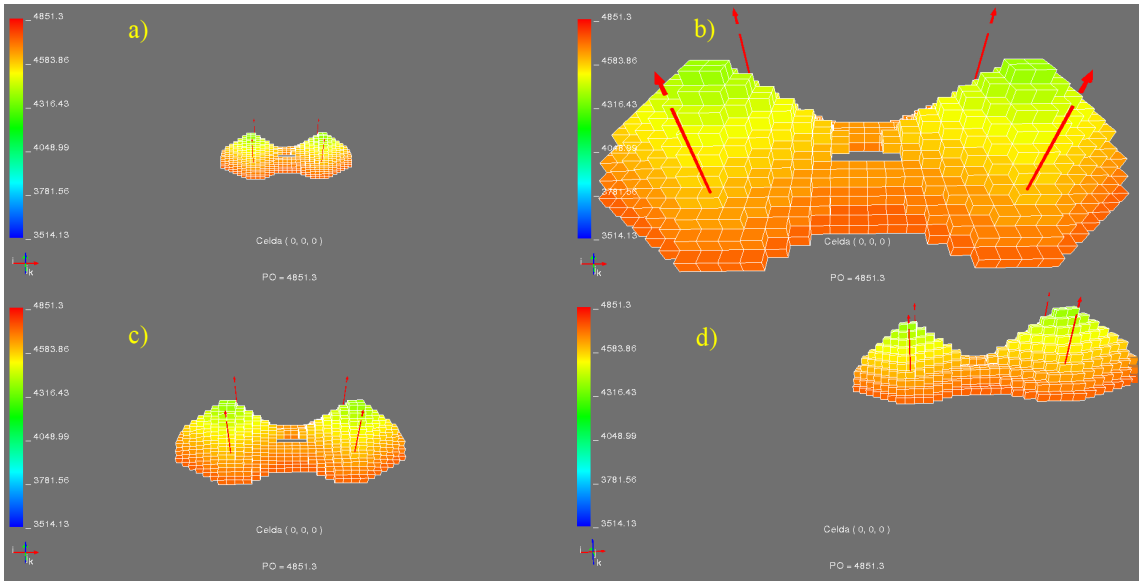


Figura 5.2: Translación y cambio de escala.

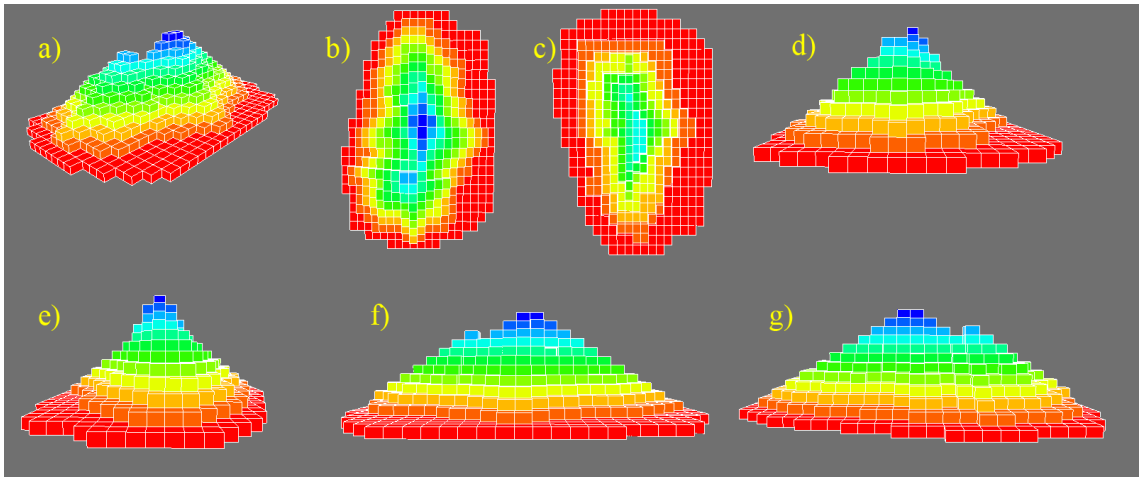


Figura 5.3: Vistas predeterminadas.

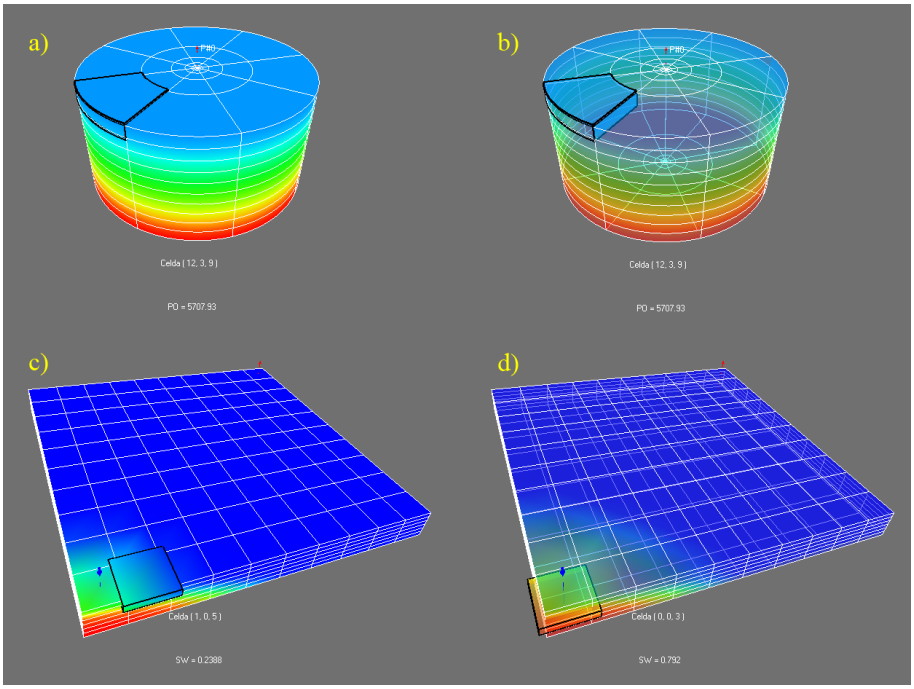


Figura 5.4: Selección de celda.

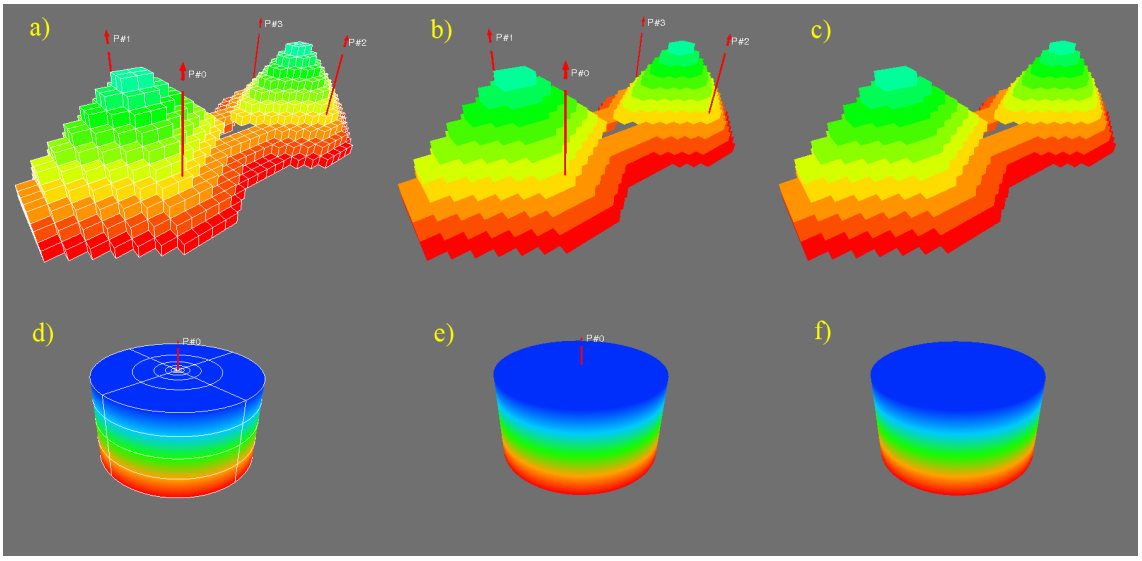


Figura 5.5: Líneas ocultas y visibles. Pozos ocultos y visibles.

5.3. Opciones de color y transparencia

Para representar los valores calculados en la simulación se utiliza una escala de colores, la cual realiza un mapeo de los valores encontrados entre el mínimo y el máximo para la propiedad mostrada en ese momento. Este mapeo puede ser calculado usando los modelos de color HSI, RGB y Ternario. La Figura 5.6 muestra los tres modelos de color aplicados a una misma malla.

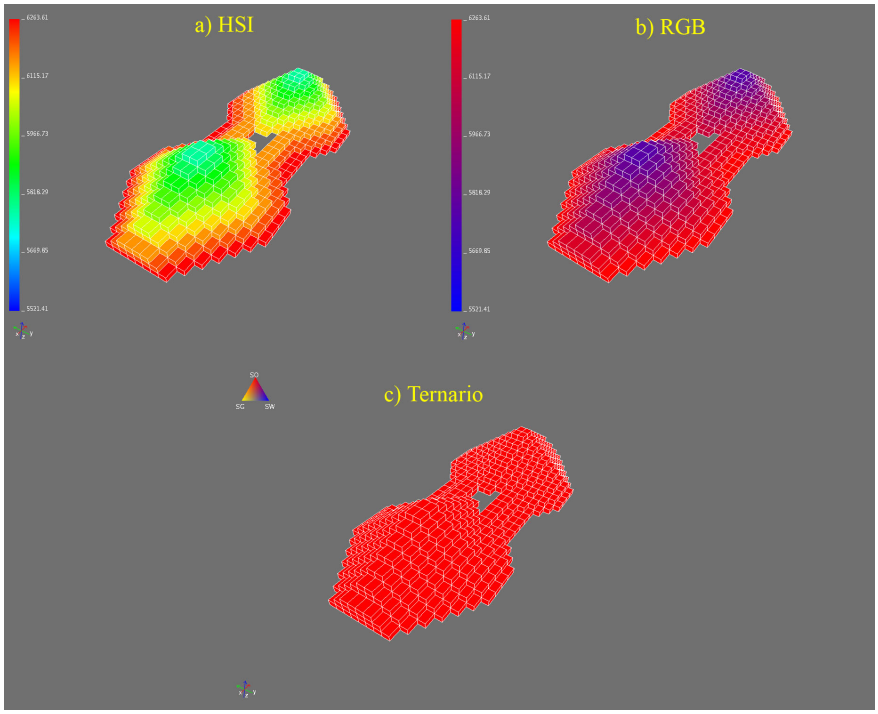


Figura 5.6: Modelos de color disponibles.

Cada propiedad tiene asignado un color mínimo y uno máximo, los cuales pueden ser modificados por el usuario. Estos colores son usados por los modelos HSI y RGB, el modelo Ternario necesita tres colores que se encuentran ligados a las saturaciones calculadas, dichos colores también pueden ser modificados. La Figura 5.7 muestra una misma malla con diferentes colores mínimos y máximos. Además muestra el cambio de colores en la representación ternaria.

Se cuenta con la opción de mostrar la malla con un nivel de transparencia elegido por el usuario, con la posibilidad de habilitar y deshabilitar esta funcionalidad. La Figura 5.8 muestra la transparencia en una malla ortogonal y una radial.

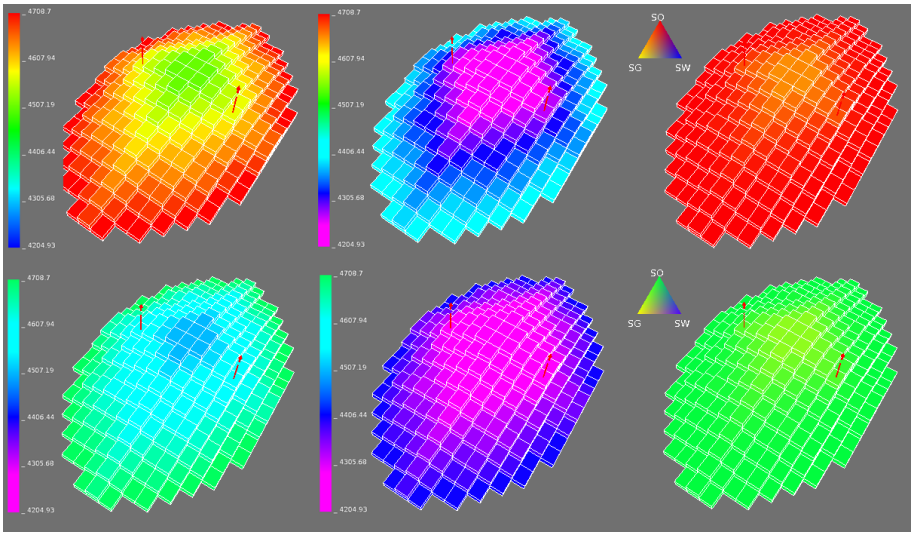


Figura 5.7: Cambio de los colores mínimo y máximo. Cambio de los colores ternarios.

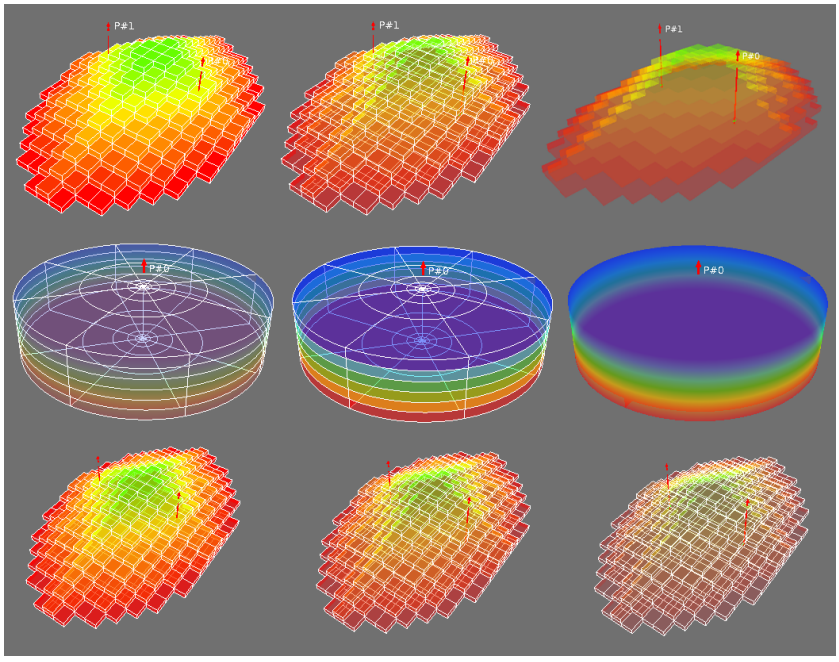


Figura 5.8: Transparencia.

5.4. Exploración de celdas internas

La funcionalidad de exploración de celdas contiene ocultamiento por bloques y ocultamiento de cualquier sección a lo largo de algún plano especifico dentro de las dimensiones de la malla. La Figura 5.9 muestra varias configuraciones posibles de una malla ortogonal y una malla radial. Además se muestra la interfaz con que fueron realizados estos cortes. Cabe señalar que esta interfaz fue creada solo para probar la funcionalidad del algoritmo y no es la interfaz con la que cuenta la aplicación.

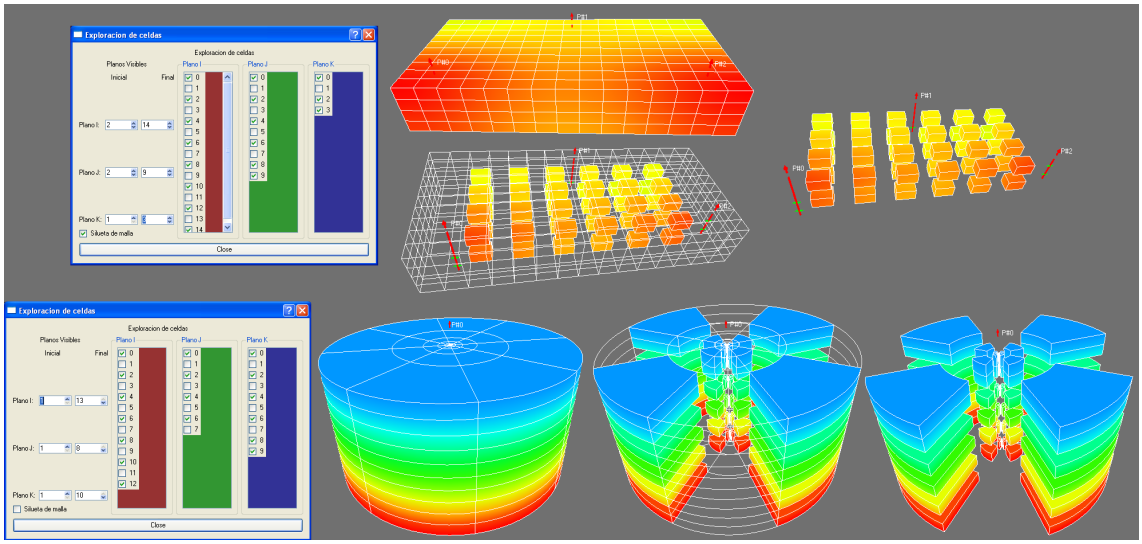


Figura 5.9: Exploración de celdas internas.

5.5. Modificación de la altura de las celdas

Otra de las funcionalidades con las que se cuenta es la de poder modificar la altura de las celdas logrando mostrar detalles que de otra forma permanecerían ocultos. También es posible regresar la malla a su altura original con una acción mínima de parte del usuario. La Figura 5.10 muestra dos tipos de mallas, cada una con tres diferentes alturas de sus celdas.

5.6. Interpolación y degradado de colores

La funcionalidad de degradado de colores fue creada para poder mostrar la malla con valores continuos, y de esta forma tener un enfoque continuo del flujo en el yacimiento.

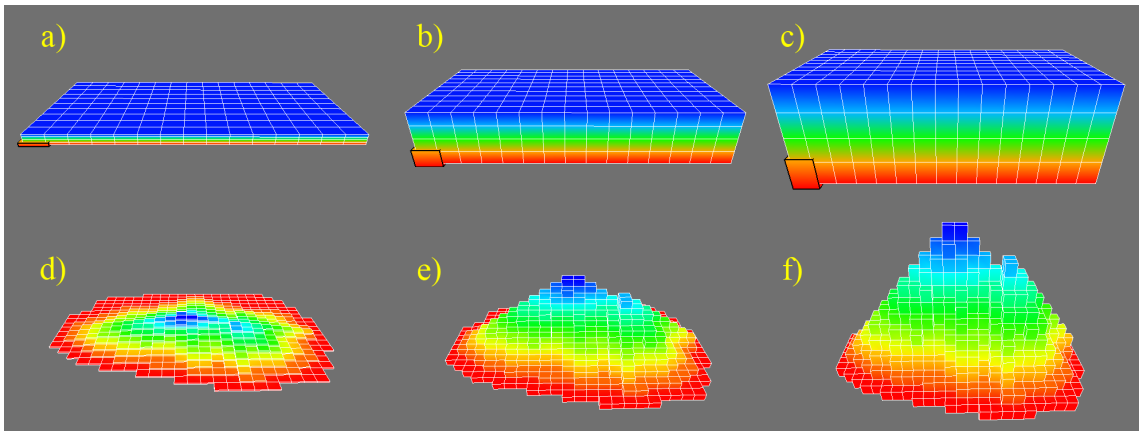


Figura 5.10: Modificación de la altura de las celdas.

La idea centrar es permitir mostrar cada celda como un componente conectado con las celdas que la rodean, mostrando un degradado entre una celda y sus vecinas, creando así una apreciación continua del yacimiento. Para mostrar una malla con interpolación solo se necesita habilitar el control adecuado, minimizando de esta forma, las acciones del usuario. La Figura 5.11 muestra varias mallas con y sin interpolación, permitiendo hacer una comparación subjetiva del método. En las secciones g y h se muestra la misma malla con la interpolación de dos propiedades distintas.

5.7. Opciones de pozos

Los pozos cuentan con varias opciones para el manejo de los mismos y de sus etiquetas. Los pozos puedes ser ocultados a petición del usuario, si esta opción es utilizada se ocultara también su etiqueta y las burbujas, si es que están visibles. Si los pozos se encuentran visibles se puede ocultar su etiqueta. El nombre del pozo también puede ser modificado por el usuario. En la Figura 5.12 se muestran estas opciones.

El grosor del pozo puede ser modificado a petición del usuario. Si el grosor es modificado sobre una malla radial, entonces la malla también debe cambiar de tamaño. En la Figura 5.13 se muestra una malla radial con el grosor del pozo original y con el grosor del pozo modificado, esta última malla se encuentra recortada para poder apreciar mejor el cambio realizado, además se muestra una parte de la misma malla con todas sus celdas.

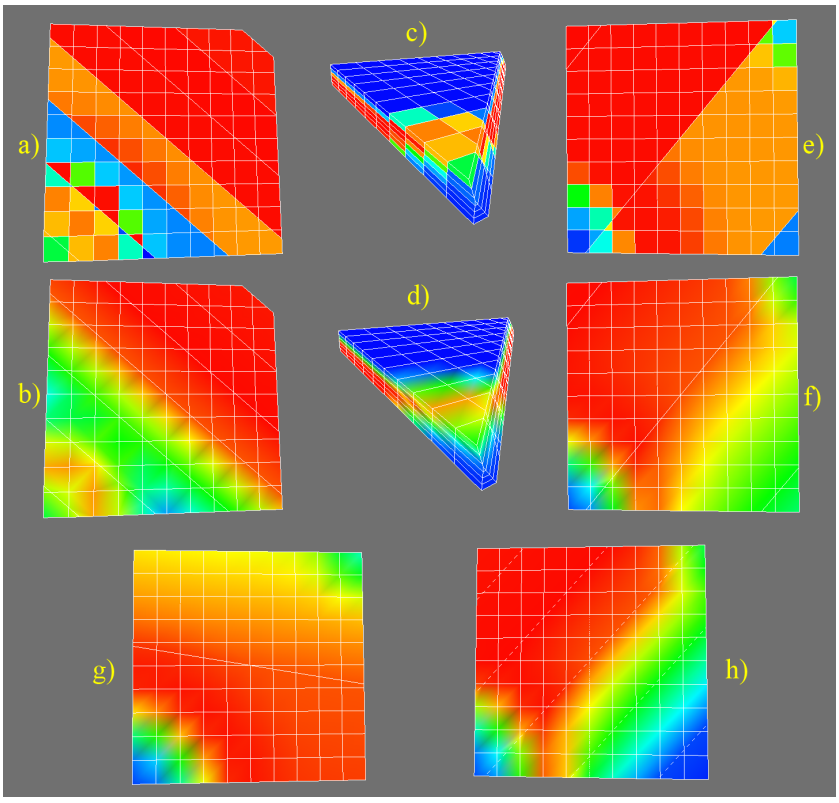


Figura 5.11: Degradado de colores.

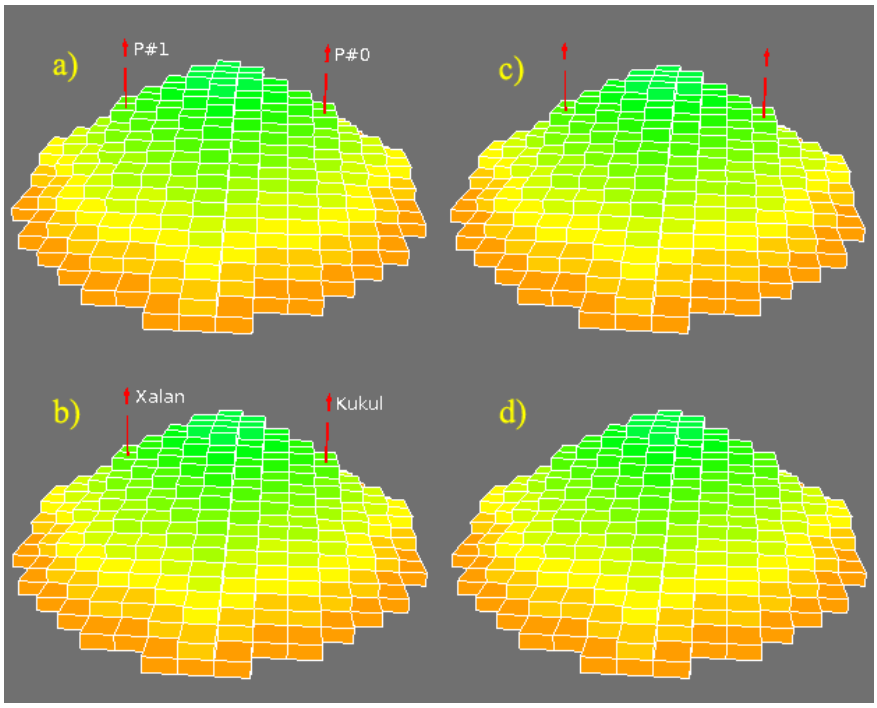


Figura 5.12: Opciones de pozos.

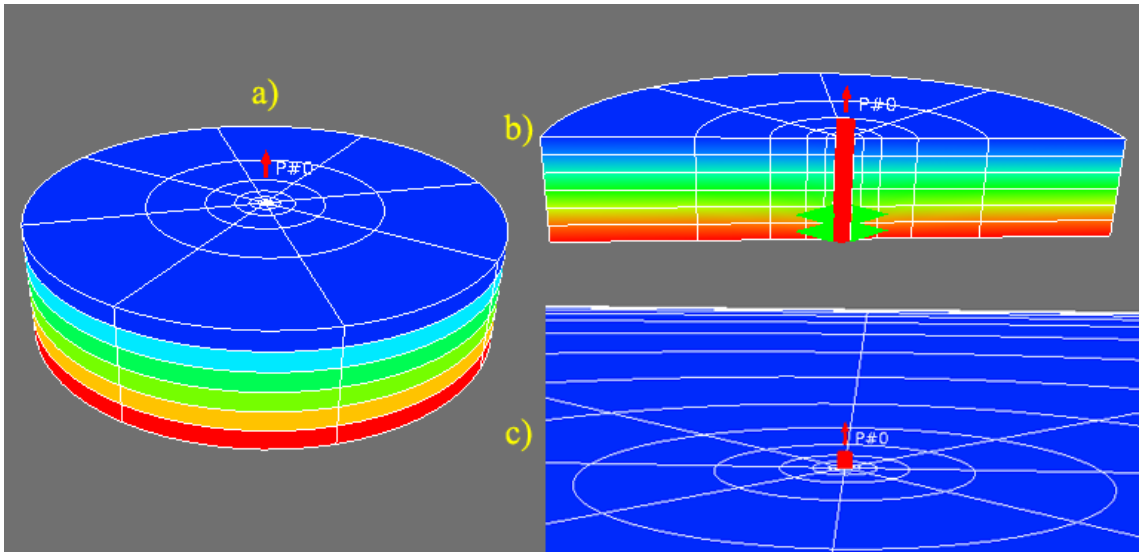


Figura 5.13: Opción de grosor de pozos.

5.8. Burbujas

La funcionalidad de Burbujas nos sirve para poder mostrar las propiedades de los pozos por medio de una apreciación cualitativa de referencia, utilizando indicadores en forma de esferas. El tamaño de las esferas depende del valor de la propiedad que representan. Estos indicadores aparecen en la parte superior de los pozos, una vez que se haya activado el control apropiado. Para poder mostrar información detallada de estos indicadores se debe dar doble clic sobre los mismos, o sobre el pozo.

Por omisión aparecen solamente los indicadores de las propiedades RGA, NP, GP, FW, con la opción de poder habilitar las propiedades QO, QG, QW, WP, RAG. En la Figura 5.14 se muestran diferentes configuraciones de burbujas.

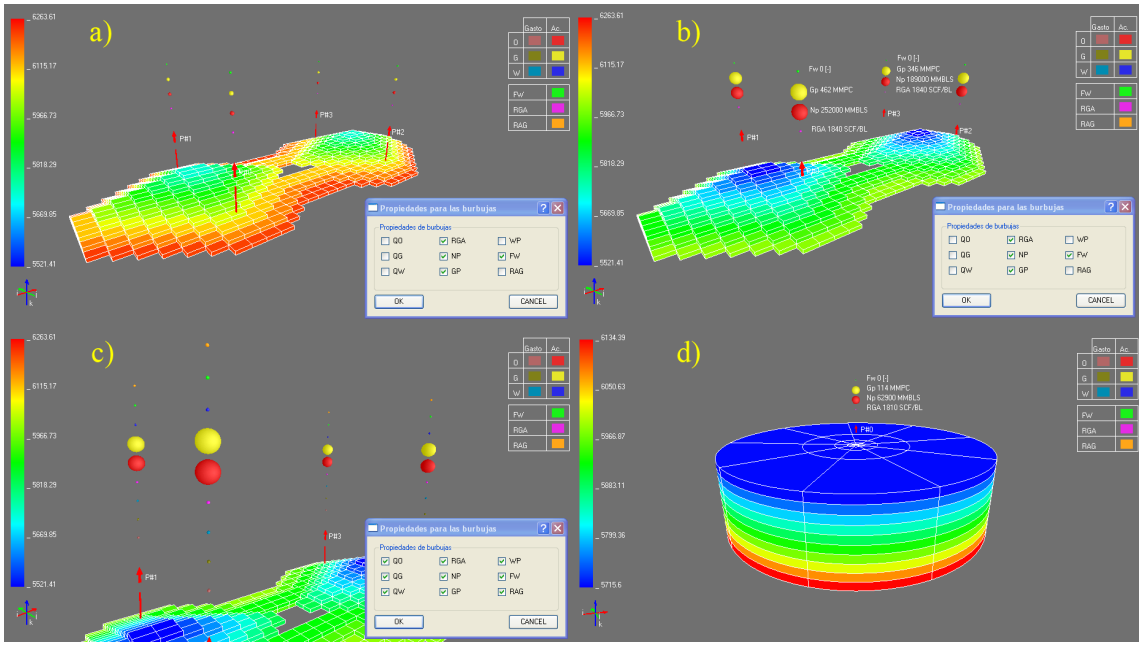


Figura 5.14: Burbujas.

5.9. Representación ideal de medios

Cuando la simulación cuenta con mas de un medio se puede elegir mostrar una ventana auxiliar con la representación ideal de los medios de una celda seleccionada.

En la ventana auxiliar se muestran todos los datos en una tabla, la cual se va modificando de acuerdo al número de medios con los que cuenta el proyecto. Los datos de las

saturaciones se muestran por medio de barras tridimensionales que modifican su tamaño dependiendo del valor que adquieren. Estas barras se modifican dinámicamente con el valor de las propiedades de acuerdo al paso visualizado en la malla tridimensional.

La representación de dos medios consiste de un cubo que contiene un conjunto de pequeños cubos. La representación visual del tercer medio consiste en la inclusión de un subconjunto de cubos más, dentro del subconjunto anterior. En la Figura 5.15 se muestra la representación ideal del medio fracturado con dos y tres medios.

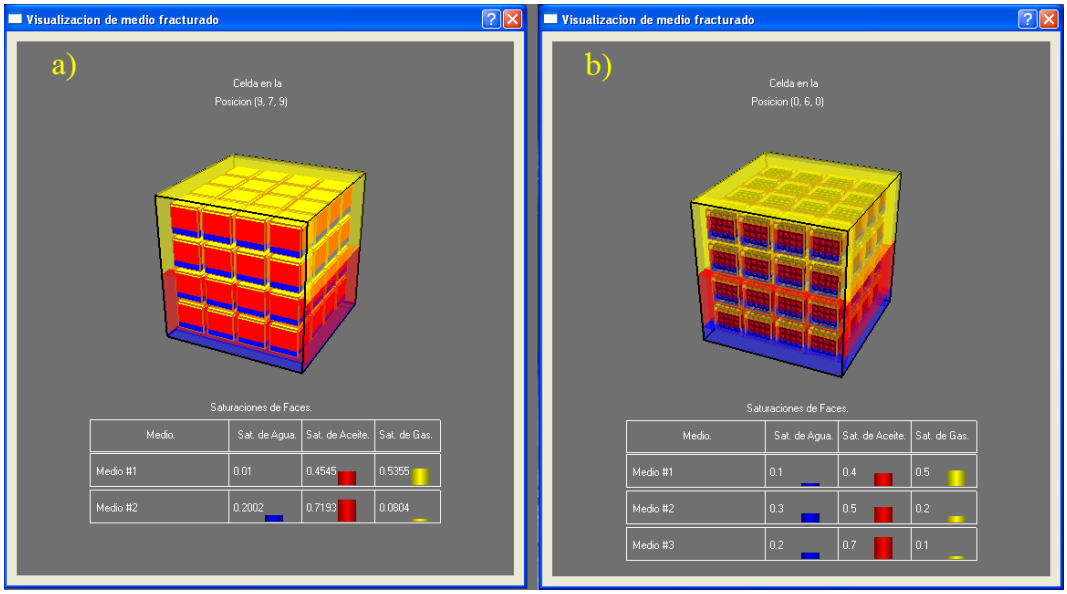


Figura 5.15: Representación ideal del medio fracturado.

5.10. Curva de saturaciones

En la funcionalidad que muestra una curva de saturaciones sobre una pila de celdas, cada pila es seleccionada por el usuario realizando un doble clic sobre alguna celda. Entonces la pila es colocada sobre la malla para tener una apreciación completa de cada una de las celdas que la componen. Esta funcionalidad solo se puede habilitar si las tres propiedades de las saturaciones (SO, SG y SW) se encuentran disponibles en el proyecto. La curva muestra la cantidad de saturación de agua, aceite y gas que contiene cada celda, pero uniendo estos datos con los de la celda vecina, dando continuidad a la gráfica mostrada. Esto permite formarse rápidamente una idea de cómo se encuentran estas saturaciones en el yacimiento. Los colores de cada saturación se encuentran ligados a los colores usados en el modelo ternario, por lo que es muy fácil e intuitivo usar ambas funcionalidades

conjuntamente. Para esta funcionalidad se cuenta con la opción de colocar en su posición original a todas las pilas mostradas, seleccionándolos de una en una con un doble clic, o todas a la vez accionando el control disponible. La Figura 5.16 muestra varios yacimientos con algunas pilas de celdas mostrando la curva de saturaciones.

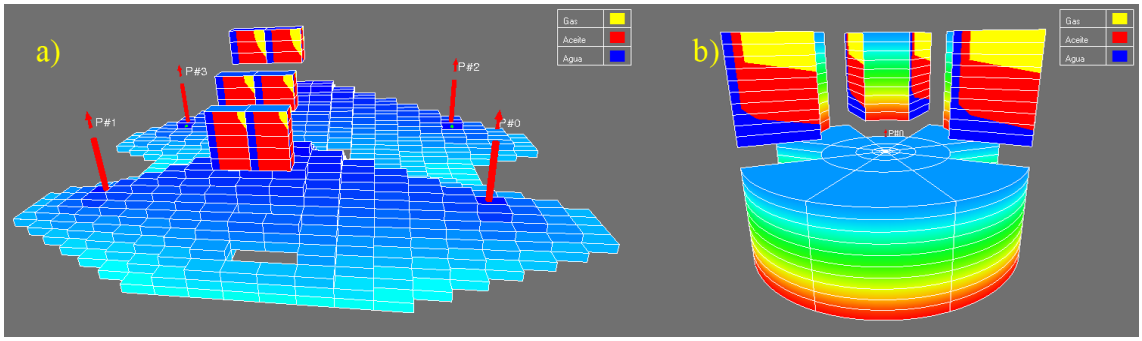


Figura 5.16: Curva de saturaciones.

5.11. Cortes mediante un plano y mediante pozos

La funcionalidad de cortes realiza cortes precisos en la malla tridimensional, ayudado por un plano de corte el cual sirve de cuchillo, este plano puede rotar en cualquier dirección, con lo cual se pueden llegar a tener cortes verticales, horizontales e incluso inclinados, sin importar las dimensiones de la malla que se esta visualizando. El corte inicial atraviesa por el centro de la malla en un plano paralelo al plano YZ. Si se requiere modificar el corte, se debe de habilitar el control para mover el plano de corte, con lo cual se mostrara un plano transparente al momento de presionar el botón izquierdo del ratón y desplazarlo. Para poder manipular la malla nuevamente, se necesita apagar el control mencionado. Si se quiere regresar el plano de corte a su posición inicial solo se tiene que habilitar el control necesario. En la Figura 5.17 se muestra una malla con varios cortes. En los cortes se muestra el plano de corte y el corte sin el plano. La sección d muestra la malla y el plano como se muestran al habilitar esta funcionalidad.

La funcionalidad de Cortes por Pozos, nos permite realizar cortes guiados por dos pozos de la malla, esto con la finalidad de poder tener una mejor apreciación de los datos que hay entre cada uno de los pozos y así ver su comportamiento que tienen al ir de un pozo a otro.

Para poder utilizar esta funcionalidad, primero se tiene que habilitar el control asignado a este modo, una vez hecho esto, lo primero que se tiene que realizar es escoger por medio de un doble clic del ratón el primer pozo sobre el cual se quiere realizar el corte (cabe señalar que el primer pozo seleccionado nos indicara cual será la parte derecha del corte,

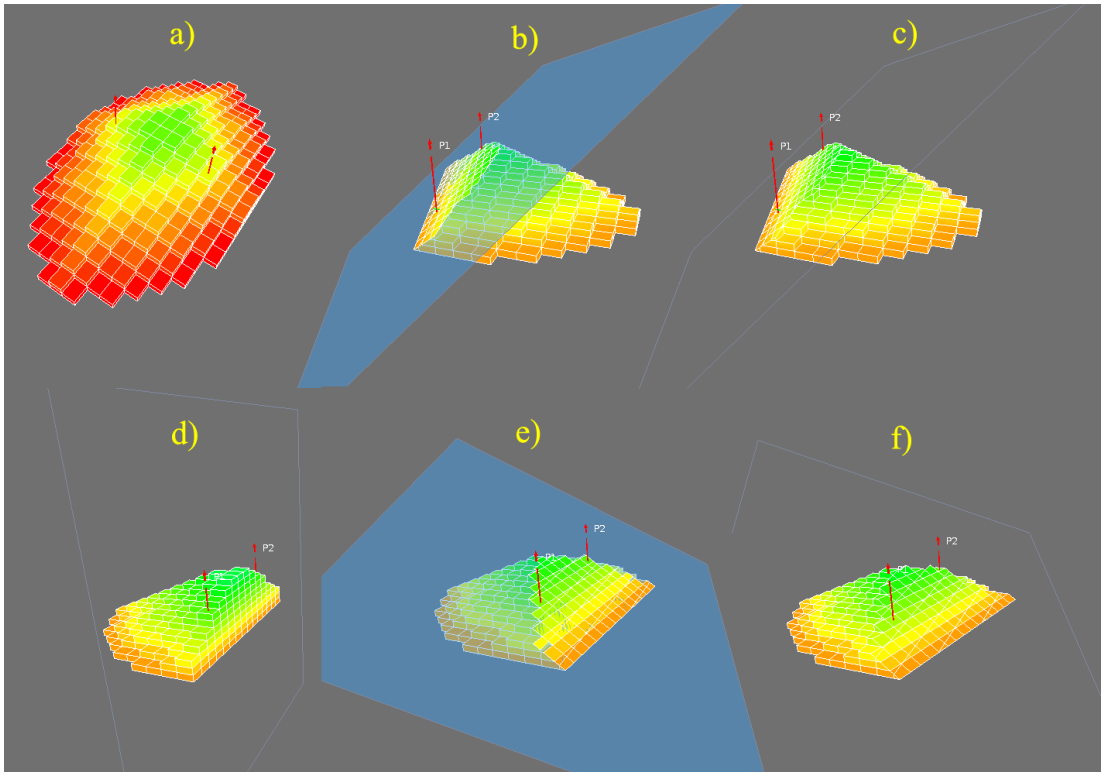


Figura 5.17: Cortes por medio de un plano.

indicándonos cual será la parte que se observara), la forma de saber que ya se tiene un pozo seleccionado es el ver que la flecha del disparo es de tamaño superior que las del resto. Una vez que se tiene ya seleccionado un pozo se le da doble clic sobre el siguiente pozo al que se le quiere aplicar el corte, con los dos pozos seleccionados se creará el corte que atraviesa la malla desde el primer pozo hasta el segundo, mostrando la parte de la malla que se encuentre en la parte posterior del corte. Se pueden realizar tantos cortes como pozos tenga la malla.

Si se quiere ver la parte inversa del corte que se ha creado, se habilita el control de *cambio de dirección de plano* con esto veremos el corte inverso del que teníamos con anterioridad. También se puede eliminar el último corte realizado, o se pueden eliminar todos los cortes hechos hasta ese momento. En la Figura 5.18 se muestran varios cortes por pozo aplicados a una misma malla. La sección *a* muestra la malla original, la sección *b* muestra la malla con un corte, la sección *c* muestra la malla con el mismo corte pero con la dirección contraria. En las secciones *d* y *e* se añadió un corte. Se muestra el último corte con ambas direcciones. Y finalmente en la sección *f* se muestra la malla con tres cortes acumulados.

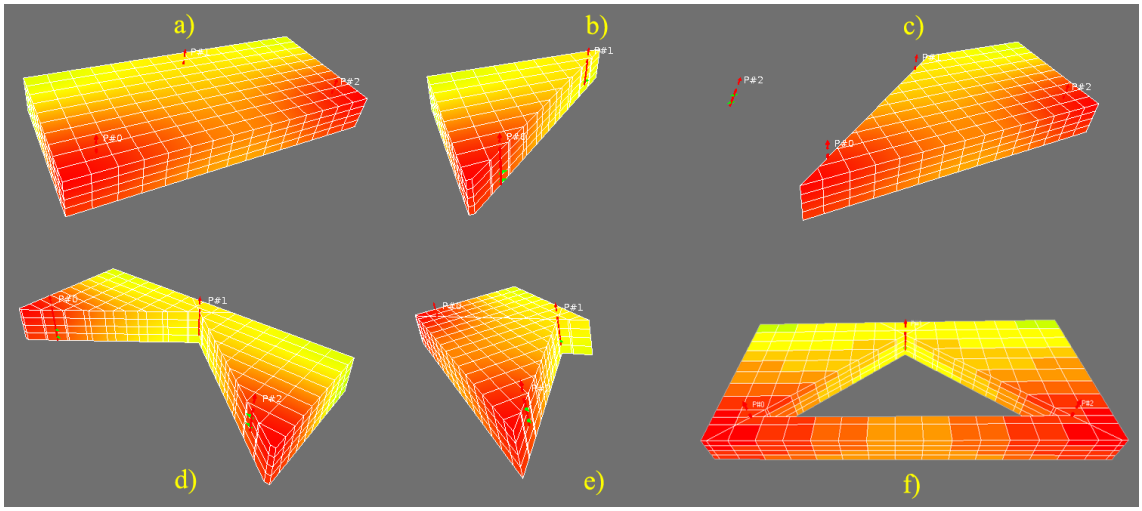


Figura 5.18: Cortes por pozo.

5.12. Isolíneas

Se creo la funcionalidad de Isolíneas para permitir al usuario encontrar información útil rápidamente. Las Isolíneas muestran un conjunto de datos como un contorno de igual color, de esta forma el usuario puede localizar sectores en el yacimiento con valores parecidos

en un rango establecido. Esta funcionalidad permite explorar cada capa de la malla con un rango variable para el conjunto de datos. El usuario puede elegir el numero de bloques entre este rango de valores, cada bloque será mostrado como un contorno con un color que toma de la escala de colores en el modelo HSI, por lo que los colores son fácilmente identificables con los valores que representan y la región representada en las Isolíneas puede ser fácilmente identificada en la malla. La Figura 5.19 muestra dos mallas y sus Isolíneas.

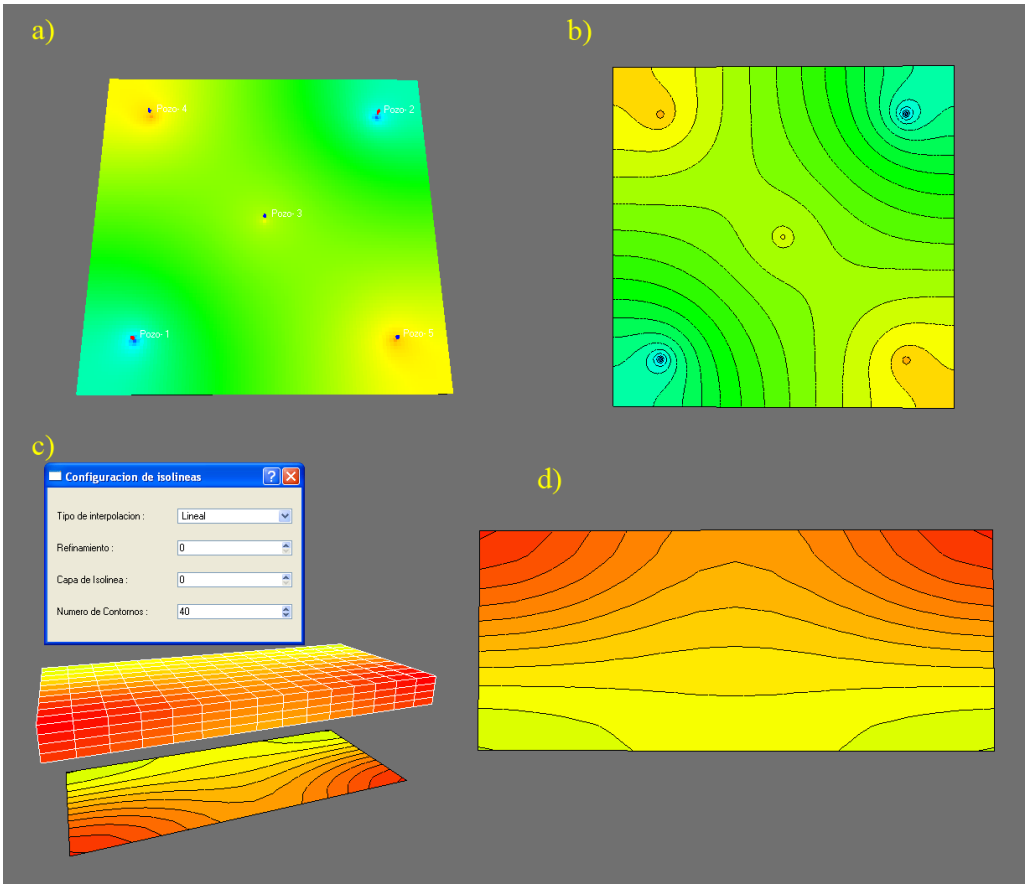


Figura 5.19: Isolíneas.

5.13. Interfaz de prueba

Se creo una interfaz de prueba para poder corroborar cada una de las funcionalidades del módulo. Esta interfaz fue realizada para cumplir las necesidades básicas del módulo, por lo cual no cumple con ningún requisito fuera del módulo y solo sigue una pauta

ergonómica básica. Todos los diálogos y controles que fueron creados para auxiliar en esta interfaz siguen la misma pauta de ergonomía, por lo que solo son aplicables a un uso interno del módulo. La Figura 5.20 muestra la interfaz de prueba creada y algunos de los diálogos.

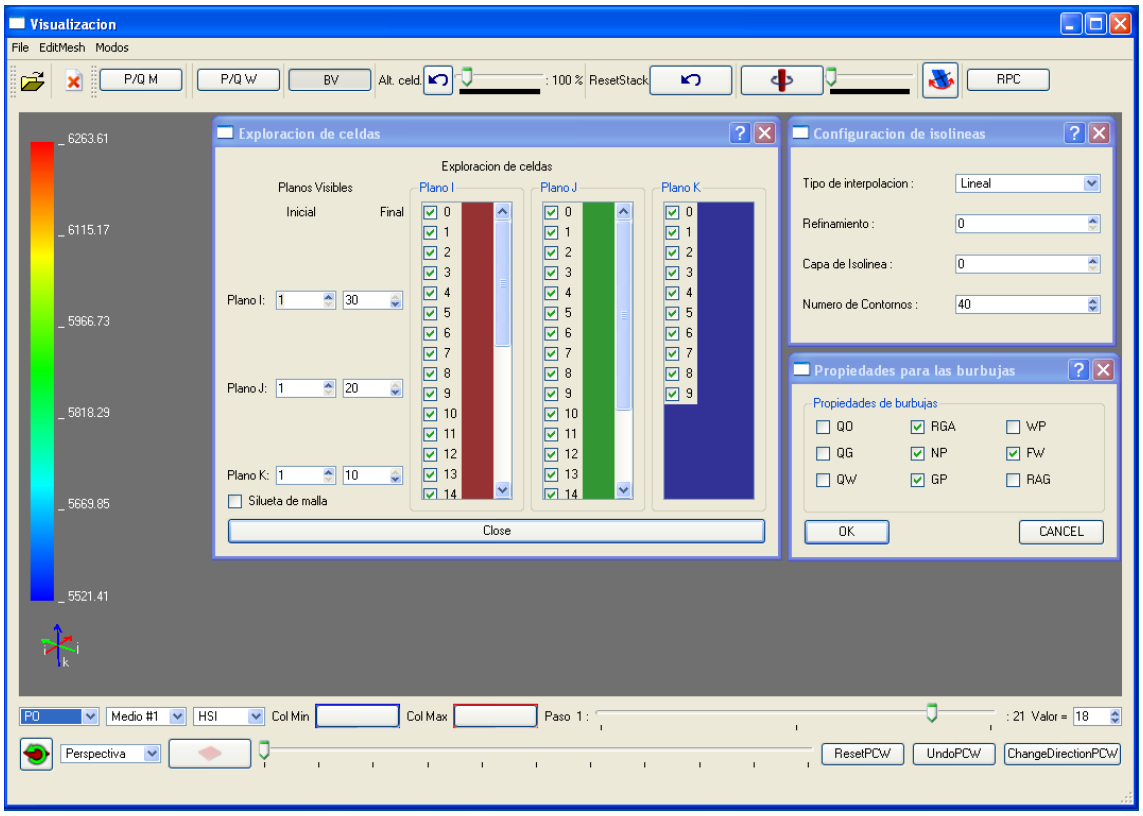


Figura 5.20: Interfaz de prueba.

Capítulo 6

Conclusiones y Trabajo a Futuro

6.1. Conclusiones

A lo largo de este trabajo hemos seguido el proceso de construcción del módulo de visualización tridimensional, desde los requerimientos hasta los resultados, pasando por los diferentes pasos del diseño y la implementación. Como se puede apreciar a lo largo de todo el proceso y especialmente en los resultados, se han cumplido cabalmente todos los requerimientos del módulo.

Desde el inicio del proyecto se pretendía crear un software de calidad [Piattini, 2001], por lo que los aspectos que siempre se tuvieron presentes al diseñar e implementar este y los demás módulos fueron los siguientes:

- Funcionalidad: Exactitud, conformidad.
- Fiabilidad: Tolerancia a fallos, facilidad de recuperación.
- Usabilidad: Comprensibilidad, facilidad de aprendizaje, facilidad de operación, atractivo.
- Eficiencia: Tiempo de respuesta, utilización de recursos.
- Mantenibilidad: Facilidad de análisis, facilidad de cambio, estabilidad.
- Transportabilidad: Facilidad de instalación.

Tomando en cuenta lo anterior, concluimos que:

1. Dado que el módulo ha sido construido a lo largo de cuatro ciclos de desarrollo, el módulo cumple con los requisitos de creación incremental: extensibilidad, actualización y mantenibilidad.

2. Las restricciones de tiempo imponían funcionalidad a eficiencia, pero, puesto que la mayoría de los algoritmos han sido desarrollados en varios ciclos de desarrollo, la mayor parte de ellos han estado sujetos a uno o varios pasos de optimización, además de que desde el diseño siempre se toma en cuenta la eficiencia.

3. Aunque el módulo ha realizado su parte con respecto a la fiabilidad y la transportabilidad, la mayor parte del trabajo realizado en este sentido se encuentra en otros módulos, esto es debido a la división natural de los módulos y de sus responsabilidades.

4. El software fué probado por especialistas petroleros dando como resultado una plena satisfacción objetiva y subjetiva. Se encontró que el aprendizaje necesario para poder utilizar el módulo fue mínimo y que el uso del mismo era fácil e intuitivo.

5. Por todo lo anteriormente dicho, la calidad del módulo es muy satisfactoria y se encuentra al nivel de los productos comerciales mas comúnmente usados, aunque como estos, el módulo debe permanecer en constante evolución.

6.2. Trabajo a futuro

Como trabajo a futuro se propones varias funcionalidades que podrían dar al módulo vistosidad y que permitirían obtener y analizar mayor información. Estas funcionalidades son solo un ejemplo de las posibilidades de extensión del módulo. Las funcionalidades son las siguientes:

1. **Flujo mediante partículas**
   El usuario selecciona un rango de valores para una propiedad y se crea una ventana con el flujo representado mediante partículas.

2. **Flujo de Vectores**
   Dentro de la malla, se mostrara el flujo del fluido mediante un conjunto de vectores.

3. **Isosuperficies**
   Dentro de la malla, se mostrara una iso-superficie que nos representé la propiedad que nos interesa observar, con esto se podrá tener una mejor apreciación de lo que esta pasando con esta propiedad.

4. **Pozos curvos y horizontales**
   Actualmente se cuenta únicamente con pozos verticales. Se diseñará e implementarán los algoritmos y estructuras necesarias para la representación de pozos horizontales.

5. **Análisis y representación de pozos.** Se construirá un módulo de análisis de pozos que permitirá al usuario observar y analizar diversas propiedades, por ejemplo:

- Efectos de cambios en las propiedades termo-mecánicas.
- Contribución de cambio de temperatura en el comportamiento del flujo del fluido.
- Comportamiento del pozo bajo tensión, incluyendo efectos de cambio de temperatura.
- Conducción de calor.

Apéndice A

Sistema Visual Humano

El humano es el usuario final, es el observador y el crítico de los resultados obtenidos en la visualización, por esto es importante entender cómo se lleva a cabo el proceso de la visión y los principales factores que intervienen en ella.

El sistema visual humano nos ofrece, mediante las imágenes percibidas, el conocimiento básico del mundo que nos rodea. Pero la visión requiere de una fuente de luz que ilumine lo que vemos, por lo que en este apéndice expondremos las ideas básicas para nuestro entendimiento sobre la naturaleza de la luz y sus propiedades, y posteriormente exploraremos el sistema visual humano, tratando de comprender las características y limitaciones del mismo.

A.1. La Luz

La luz, según el Diccionario de la Real Academia Española [RAE, 2002], es el agente físico que hace visibles los objetos. La luz constituye sólo una pequeña porción del espectro electromagnético, el conjunto de ondas electromagnéticas existentes en el universo [Halliday, 1997], [Adler, 1980]. En la Figura A.1 se muestra esquemáticamente el espectro electromagnético ordenado en función de sus frecuencias y longitudes de onda.

Las ondas electromagnéticas están formadas por un campo magnético y un campo eléctrico perpendiculares entre si, de esta forma, la radiación electromagnética es una combinación de campos magnéticos y eléctricos oscilantes, que se propagan a través del espacio. La radiación electromagnética, se puede comportar como ondas y como partículas, para nuestro interés solo basta con ver su comportamiento como ondas.

Una onda [Hartmann, 1991], es una forma de transferencia de energía a través del espacio o de un medio en la que existen cambios periódicos que implican la alteración de las partículas o un cambio periódico en la cantidad física. A las ondas se les caracteriza por su longitud de onda y por su frecuencias. La longitud de onda (ver Figura A.2) es la separación espacial existente entre dos puntos, cuyo estado de movimiento es idéntico. Lo

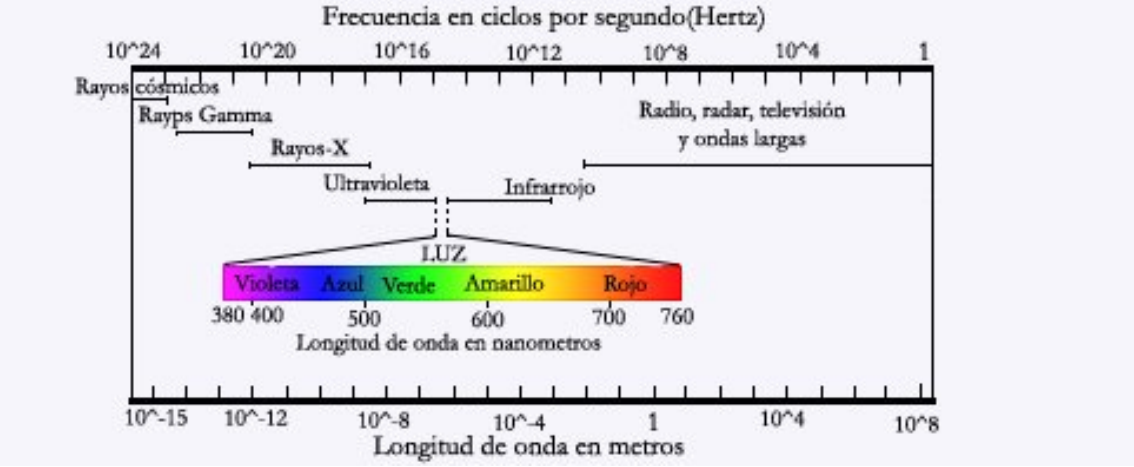


Figura A.1: Espectro electromagnético

mas sencillo para medirla es fijarse en la distancia existente entre dos crestas o dos valles de una onda. Por lo general se usa la letra griega lambda λ para designar la longitud de onda.

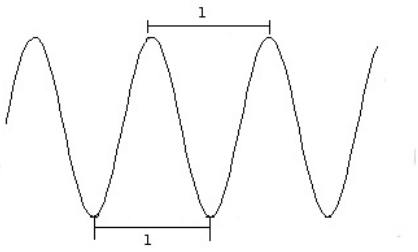


Figura A.2: Longitud de onda

La frecuencia es el número de ciclos u oscilaciones en una unidad de tiempo, por lo general un segundo, que se cuenta en un punto, en otras palabras, es el número de veces por segundo que la cresta de una onda (u otro punto especifico de la misma) pasa por un punto determinado. La unidad de medida de la frecuencia es el hercio(hz.) en honor al físico alemán Heinrich Rudolf Hertz, donde 1 hz. es un evento que tiene lugar una vez por segundo.

El espectro electromagnético, esta ordenado en función de sus frecuencias o longitudes de onda, en un espectro que se extiende desde ondas de frecuencias elevadas (longitudes de onda pequeñas) hasta frecuencias muy bajas(longitudes de onda grandes). Por orden

creciente de longitudes de onda, se ha confeccionado una escala denominada espectro electromagnético (ver Figura A.1). Este espectro va desde las ondas de radio, hasta los rayos gamma La luz visible es una pequeña franja que va desde los 780 nanómetros(NM) hasta los 380 NM, en esta fracción del espectro electromagnético, la longitud de onda de la radiación electromagnética se percibe como color. La luz se ve roja de 625 a 740 NM, se ve amarilla de 565 a 590 NM, se ve verde de 520 a 565 NM y se ve azul de 450 a 500 NM [Alonso, 1986]. Donde 1 NM es la billonésima parte de un metro.

A.2. Sistema Visual Humano

El sistema visual humano esta formado por el ojo y una porción del cerebro que procesa las señales neurales del ojo. El ojo es la cámara, y enfocaremos nuestra atención a este.

A.2.1. Estructura del ojo humano

Un corte horizontal con los componentes principales del ojo se muestra en la Figura A.3

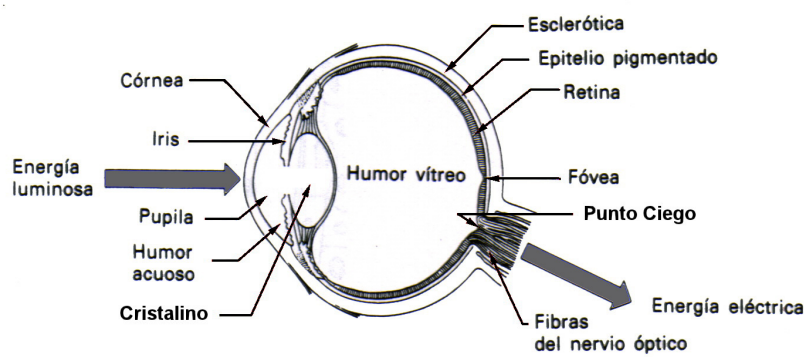


Figura A.3: Estructura del ojo humano

El ojo humano es una estructura esférica, con un diámetro aproximado de 20 mm. Este se encuentra envuelto por 3 capas de tejido o membranas: la capa externa, que consiste de la esclerótica y la córnea; la capa media, que se conforma de la coroides, el cuerpo ciliar y el iris; la capa interna, que es la retina [Gardner, 1989], [Adler, 1980], [Gonzalez, 2002], [Glassner, 1995].

La esclerótica es una membrana opaca, fibrosa, dura y está unida a los músculos del ojo. Sé continua con la córnea. La córnea es una membrana transparente muy resistente, cuya función principal es refractar la luz incidente. La córnea es la lente no ajustable del

ojo, tiene una forma redondeada que actúa como una lente convexa, refractando los rayos de luz, esta refracción forma el enfoque inicial de la luz que entra al ojo. La córnea es separada del cristalino por un fluido claro llamado humor acuoso.

El coroides, perteneciente a la capa media, llamada con frecuencia úvea, yace directamente bajo la esclerótica. El coroides contiene una red de vasos sanguíneos que sirven para nutrir al ojo. En su extremo, esta membrana se divide en el cuerpo ciliar y el iris. El iris es una estructura pigmentada suspendida entre la cornea y el cristalino, y tiene una abertura circular en el centro, la pupila. El tamaño de la pupila varia controlando la cantidad de luz que pasara por el cristalino. El cristalino lleva acabo el segundo enfoque de la luz, proyectando esta sobre la retina. Para poder enfocar distancias cortas y largas, el cristalino cambia de forma, en un proceso que se denomina acomodación, este proceso es controlado mediante el cuerpo ciliar, un grupo de músculos situados alrededor del iris. Los rayos que pasaron a través del cristalino pasan por otro liquido transparente y gelatinoso, el humor vítreo, sin sufrir ninguna desviación, y son finalmente enfocados sobre la retina. El humor vítreo mantiene la estructura del ojo mientras se forma la imagen de una forma adecuada.

La retina ésta compuesta sobre todo por células nerviosas receptoras sensibles a la luz, estas convierten la intensidad y color de la luz a señales neurales. Hay una parte de la retina que carece de estos receptores y se compone sólo de fibras del nervio óptico, por lo que es insensible a la luz, esta parte es la papila óptica o punto ciego. Hay dos tipos de receptores llamados: conos y bastones.

La cantidad de conos se encuentra entre 6 y 7 millones, y están localizados, en su mayoría, en una porción central de la retina llamada fovea y cuya depresión central es llamada fovéola. Cuando se mira un objeto específicamente, una línea que va del objeto visto a la fovéola indica el eje visual del ojo. Los bastones se encuentran en mayor cantidad que los conos, de 75 a 150 millones, distribuidos sobre la superficie de la retina, en zonas alejadas de la fovea. Los bastones son los responsables de la visión escotópica (visión a bajos niveles de intensidad de luz, como en la noche.)

La Figura A.4 muestra la distribución de conos y bastones sobre la retina. Esta figura esta basada en un trabajo desarrollado por Osterberg en 1935.

Los conos son los receptores del color, son poco sensibles a la intensidad de la luz y proporcionan visión fotópica (visión a altos niveles de intensidad de luz, como en el día). Hay tres tipos distintos de conos y cada uno responde a una distinta banda del espectro de luz visible, a la roja, a la verde y a la azul. Esto permite al ojo y al cerebro discriminar el color, mediante un proceso llamado visión cromática. Básicamente, cada cono responde de una forma diferente a un color arbitrario, entonces se genera un conjunto único de respuestas para cada color único de luz. Con estas señales de los tres tipos de conos, el cerebro tiene la información necesaria para formar una percepción distinta para un gran numero de colores diferentes. La Figura A.5 muestra las curvas de respuesta de los tres diferentes tipos de conos.

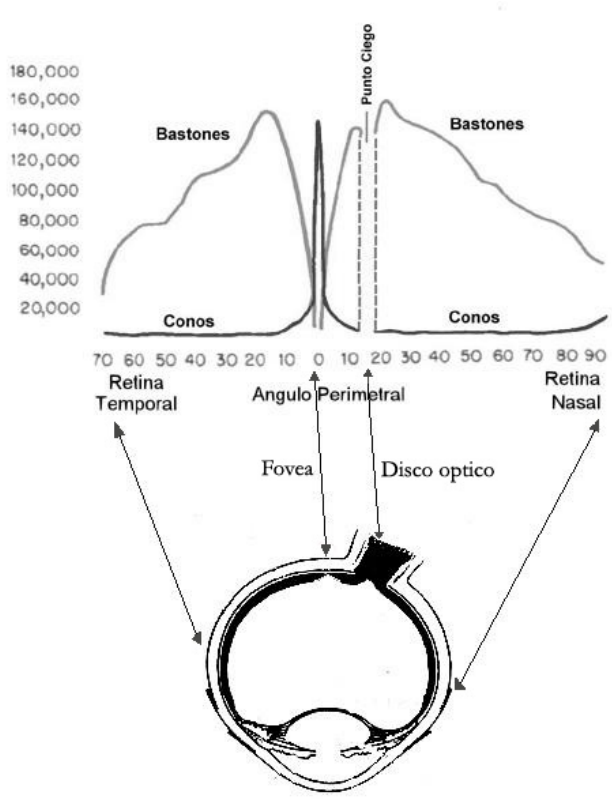


Figura A.4: Distribución de conos y bastones sobre la retina

A.2.2. Propiedades del Sistema Visual Humano

La diferencia entre los conos y los bastones y su distribución sobre la retina son responsables de diferentes aspectos de la visión. Ya que la sensación de color se encuentra concentrada en la fovea, nuestra percepción del color es mejor para objetos que miramos directamente. Por el contrario, tenemos una mínima percepción de color para objetos en nuestra visión lateral. Y ya que los bastones, altamente sensitivos, son muy abundantes fuera de la fovea, nuestra percepción de bajo nivel de luz es mejor en nuestra visión lateral. Cuando los conos y los bastones son excitados por alguna fuente de luz, se genera una reacción electroquímica que genera impulsos neurales que son enviados al cerebro mediante el nervio óptico. El nervio óptico es una extensión de la retina que conecta a esta con el cerebro. Los impulsos neurales son procesados por la corteza visual. La percepción de la visión es creada con el proceso de la corteza visual. Antes que la corteza visual procese la información recibida, en el ojo se llevan acabo diversos procesos de nuestro interés.

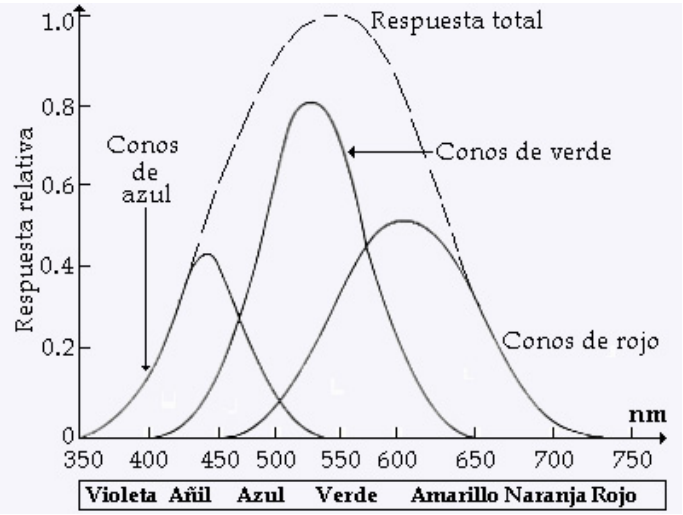


Figura A.5: Curvas de respuesta relativa de los conos

Los principales son: la forma en que los fotorreceptores responden a las diferencias en la intensidad de luz, la forma en que estos responden de un cambio de intensidad a otro, y las respuestas a los detalles finos.

La relación entre la intensidad de la luz que entra en el ojo y el brillo percibido no es una función lineal, a medida que la intensidad de lo observado cambia, el observador no percibe un cambio igual en el brillo. La respuesta actual del ojo a la intensidad, es más parecida a una función logarítmica [Gonzalez, 2002], [Baxes, 1998], [Glassner, 1995], como se muestra en la Figura A.6

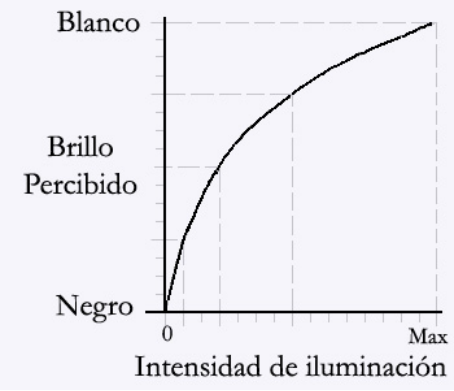


Figura A.6: Respuesta del ojo a la intensidad de la luz

En la Figura A.6 podemos ver que para que el ojo perciba la siguiente diferencia notable en el brillo, la intensidad de iluminación del objeto debe ser el doble. Como consecuencia, pequeños cambios de intensidad en regiones oscuras, son más perceptibles que cambios idénticos en regiones claras. Esta relación de intensidad con el brillo percibido es conocida como ley de Weber. La ley de Weber fue descrita primero por el fisiólogo germano E. H. Weber [1] y fue reformulada cuantitativamente por el gran fisiólogo experimental Gustav Fechner. La ley revela la influencia universal de los estímulos de fondo sobre la sensibilidad a los incrementos de intensidad.

Los siguientes ejemplos ilustran la ley de Weber.

La Figura A.7 muestra una imagen y una grafica, esta imagen esta compuesta por barras de distintos niveles de intensidad (niveles de gris), la intensidad asciende de izquierda a derecha en saltos iguales que van del negro, el menor nivel de intensidad, al blanco, el mayor nivel de intensidad. La grafica a su derecha muestra los pasos en el nivel de intensidad.

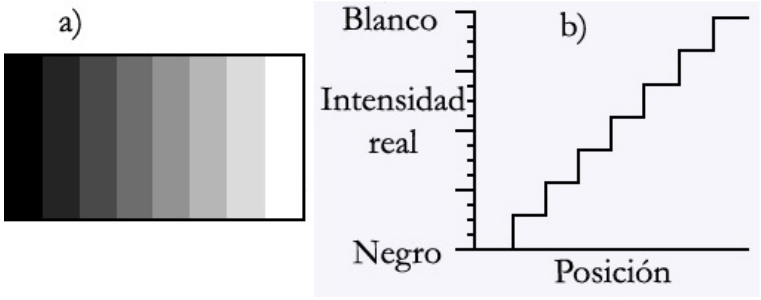


Figura A.7: a)Saltos iguales en escala de gris b) Intensidad de los saltos

Como se esperaba de la Figura A.6, los saltos en las zonas oscuras de la imagen son fácilmente distinguibles, mientras los pasos en la zona mas clara se vuelven indistinguibles. De aquí podemos observar dos cosas: la diferencia del brillo percibido en cada paso no es igual, y el ojo no ve iguales los incrementos de intensidad en las regiones oscuras y claras.

La Figura A.8 muestra otra imagen compuesta por barras de diferentes niveles de gris, pero a diferencia de la Figura A.7 la intensidad de estas barras asciende de acuerdo a la Figura A.6, en forma logarítmica. De esta forma, el brillo percibido de los saltos se observa igualmente espaciado e igualmente definido para las zonas oscuras y para las

[1] Weber, Ernst Heinrich (1795-1878). Fisiólogo alemán. Fue profesor en la Universidad de Leipzig(1821-71) y es conocido por su trabajo sobre el tacto y por la formulación de la ley de Weber (Weber's law). El aumento en el estimulo necesario para producir un incremento en la sensación no es fijo, sino depende de la intensidad del estimulo precedente.

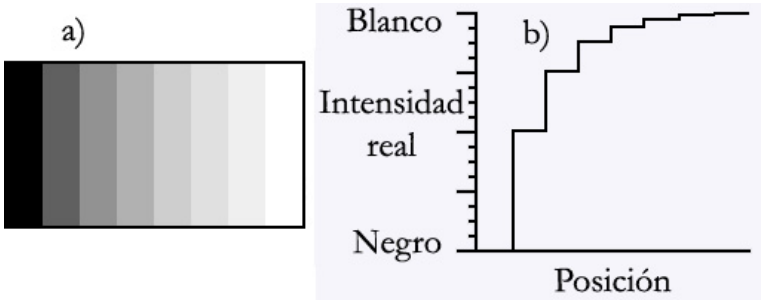


Figura A.8: a)Pasos de acuerdo a la respuesta logarítmica del ojo. b) Intensidad de los saltos

zonas claras.

El punto importante de la ley de Weber, es que el ojo es más sensible a cambios de intensidad en zonas oscuras de una imagen, que a cambios de intensidad iguales en zonas claras.

Existe una interacción entre fotorreceptores que es llamada inhibición lateral, esta causa algunos fenómenos de nuestro interés. Un fenómeno es llamado contraste simultaneo y otro es el efecto de las bandas de Mach.

El contraste simultaneo es una ilusión en donde una región que se encuentra rodeada por una región oscura se ve más clara de lo que en realidad es. Y una región rodeada por una región clara aparece más oscura.

Figura A.9: Contraste simultaneo

En la Figura A.9 se puede observar este efecto, los cuadros internos tienen la misma intensidad, pero el de la derecha se ve mas claro que el de la izquierda, esto es debido a que el cuadro exterior izquierdo es más oscuro que el cuadro exterior derecho y el sistema visual ajusta la intensidad vista de los cuadros interiores basándose en la carga de intensidad que rodea al objeto visto. De esta forma el cuadro interior izquierdo se ve más claro que el derecho porque el área que lo rodea es más oscura que el área que rodea al cuadro interior derecho.

El efecto de las bandas de Mach, es un fenómeno en el que el sistema visual varia

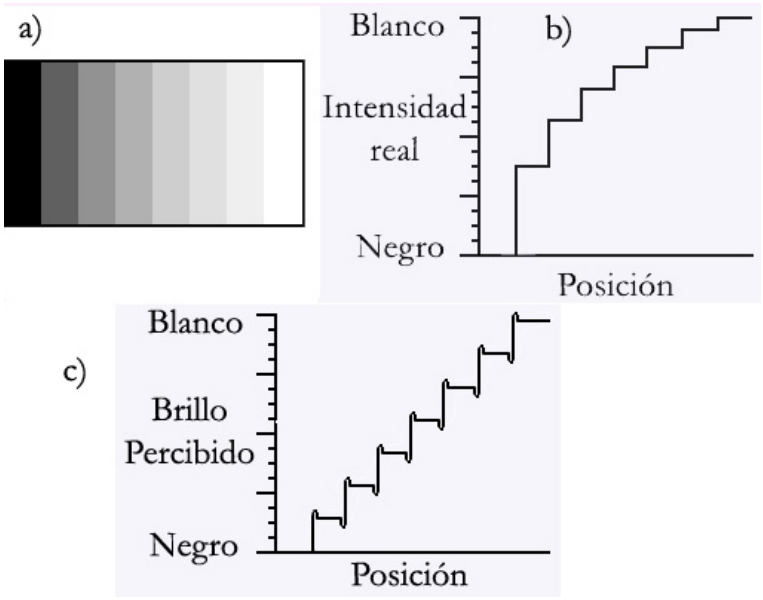


Figura A.10: Efecto de las bandas de Mach a) Pasos de acuerdo a la respuesta logaritmica del ojo. b) Intensidad de los pasos c) Brillo percibido en los pasos de la imagen

la intensidad en las uniones entre cambios de intensidad. La Figura A.10 ilustra este efecto. Cuando el ojo ve un cambio repentino en la intensidad, varia el brillo aparente en la transición, así, el brillo parece disminuir un poco antes del cambio, y aumenta un poco después de la transición, de esta forma el ojo realza los borde en los cambios de intensidad, y al resaltar los bordes en los cambios de intensidad, el ojo nos da mayor agudeza visual.

El sistema visual tiene limitaciones fundamentales como cualquier sistema óptico, el ojo tiene un limite en cuanto el tamaño y la diferencia de intensidad en una transición. Los factores limitantes son el número de fotorreceptores (conos y bastones) y la organización y distribución de estos en la retina (ver Figura A.4), la calidad óptica del ojo(cornea, humor acuoso, cristalino y humor vítreo), y la trasmisión y procesamiento de la información visual. Por lo general, la respuesta del ojo a la frecuencia falla cuando la transición en la intensidad vista se vuelve cada vez más y más pequeña, como se muestra en la Figura A.11. La diferencia entre los niveles de brillo en la transición también es un factor. De esta forma, cuando las transiciones son muy finas o el contraste es muy bajo, el ojo no percibe cambio alguno, sino que percibe una única cantidad de brillo en el área, como sucede en el extremo derecho de la Figura A.11.

De todos los fenómenos anteriores podemos resumir lo siguiente:

Los cambios en el brillo son más fácilmente percibidos en regiones oscuras. La intensidad de un objeto esta relacionada con la cantidad de brillo del área circundante. Los bordes

Figura A.11: Patrón ascendente de ondas

en los cambios de intensidad son resaltados. El ojo no puede observar detalles muy finos. Los detalles con alto contraste son más perceptibles que los que tienen bajo contraste.

Con esto en mente podemos explotar mejor el sistema visual humano con la finalidad de tomar mejores decisiones en futuros problemas, siempre pensando que el critico final será un ser humano.

Apéndice B

Fundamentos de Color

Los colores que los humanos perciben en un objeto, son determinados por la naturaleza de la luz reflejada por el objeto. La caracterización de la luz es central en la ciencia del color. Sí la luz es acromática, es decir sin color, su único atributo es la intensidad. Luz acromática es lo que vemos en una televisión de blanco y negro. La luz cromática, es decir con color, se encuentra aproximadamente en el rango de 380 a 760 nm en el espectro. Las cantidades básicas que son usadas para describir la calidad de una fuente de luz cromática son el resplandor (radiance), la luminiscencia (luminance) y el brillo (brightness).

El resplandor es la cantidad total de energía que fluye de la fuente de luz y se mide en watts (w). La luminiscencia es medida en lumens (lm) y da una medida de la cantidad de energía que un observador percibe de la fuente de luz. Por ejemplo, la luz de una fuente infrarroja tendrá una gran cantidad de energía, pero el observador no podrá percibirla, por lo que su luminiscencia tendrá un valor de cero. Finalmente, el brillo es un descriptor subjetivo que es prácticamente imposible de medir. Este envuelve la noción acromática de intensidad y es uno de los factores clave para describir la sensación de color.

La CIE [CIE] (Commission Internationale de l'Eclairag - La comisión internacional de iluminación) definió unas tablas o curvas (Figura B.1) llamadas *Funciones CIE de correspondencia de color*, estas curvas se basan en la *Teoría de los tres estímulos de la percepción de color* (the tristimulus theory of color perception), como se mostró en el Apéndice A, el ojo humano percibe los colores mediante la combinación de los estímulos recibidos por los tres tipos de conos con los que cuenta, cada uno de estos conos responde a una distinta banda del espectro de luz visible, a la roja, a la verde y a la azul. Esto permite al ojo y al cerebro discriminar el color, mediante un proceso llamado visión cromática. Graficando las respuestas de cada uno de los conos a diferentes longitudes de onda se obtienen las curvas de la Figura B.1.

Utilizando estas funciones, un color se expresa mediante la mezcla de los componentes XYZ. [Gonzalez, 2002], [Baxes, 1998], [Efford, 2000], [Pajares, 2002], [CIE_HyperPhysic],

[CIE_Ufasta].

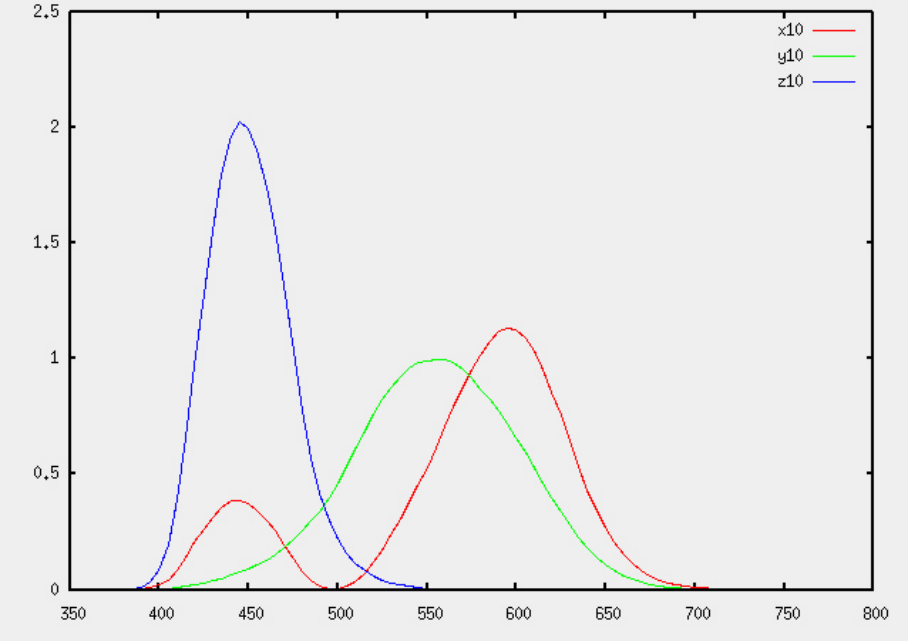


Figura B.1: Funciones CIE de correspondencia de color.

Dado que es más conveniente trabajar en un espacio de color bidimensional, se creo el *Diagrama de Cromaticidad CIE*, este diagrama se realiza proyectando el espacio CIE XYZ dentro del plano $X + Y + Z = 1$, esta proyección se define como:

$$\begin{aligned} x &= X/X + Y + Z \\ y &= Y/X + Y + Z \\ z &= Z/X + Y + Z = 1 - x - y \end{aligned} \tag{B.1}$$

La Figura B.2 muestra el resultado de esta proyección. Como se puede observar, el perímetro del diagrama marca la longitud de onda de la luz visible, a lo largo de este perímetro se encuentran todos los colores "puros" o totalmente saturados. Entre más se aleje un punto de la frontera y se acerque al punto de igual energía, el punto blanco, más luz blanca será añadida al color y este será menos saturado. El punto blanco es tomado como $x = \frac{1}{3}$ y $y = \frac{1}{3}$. La saturación en este punto es igual a cero.

Gracias a toda la base científica que el sistema CIE provee, se pueden especificar de una forma muy precisa todos los colores visibles, pero es difícil hacerlo. Una for-

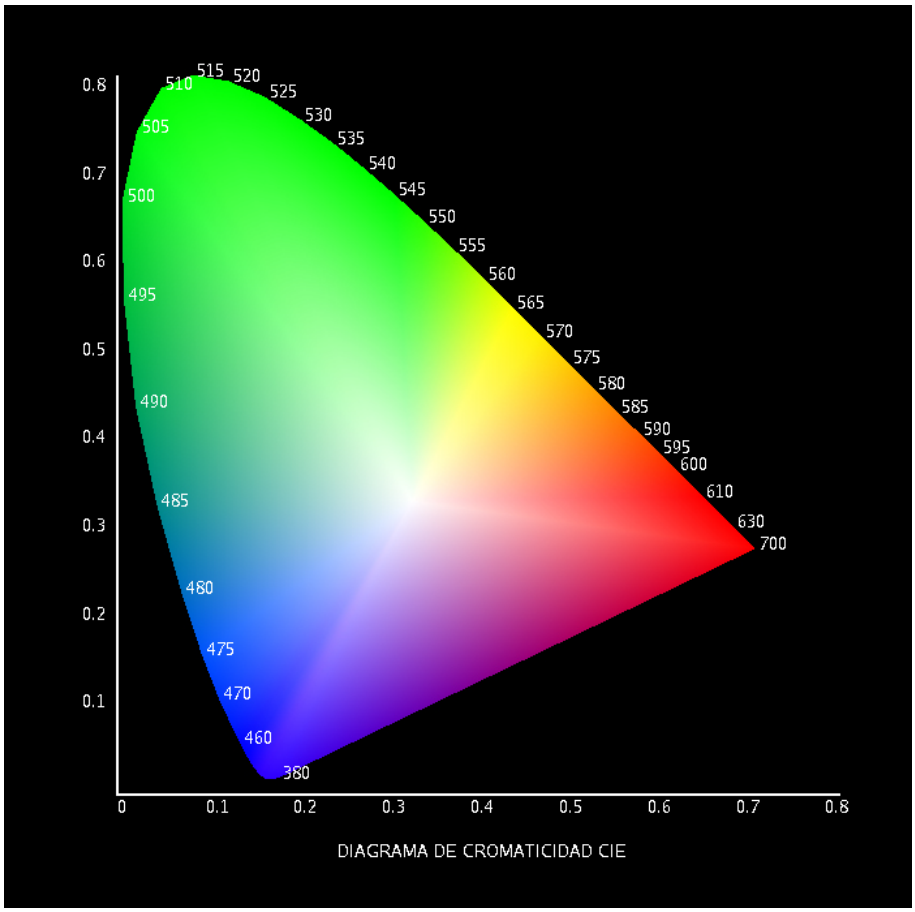


Figura B.2: Diagrama de cromaticidad CIE

ma fácil de especificar un color en algún estándar es mediante el uso de un modelo de color.

Un modelo de color, es una especificación de un sistema coordenado y un subespacio dentro de este sistema, donde cada color es representado por un solo punto.

En visualización, los modelos más comúnmente usados son el RGB, el CMY y el HSI.

B.1. Modelo de color RGB

En el modelo RGB (**R**ed, **G**reen, **B**lue), cada color se encuentra representado por los componentes espectrales primarios del rojo, el verde y el azul y se basa en un sistema cartesiano de coordenadas. El subespacio de color de interés es el cubo de la Figura B.3,

donde los valores **R**, **G**, **B** ocupan las tres esquinas sobre los ejes coordenados, el negro es el origen y el blanco es la esquina más alejada del origen. La escala de grises se encuentra en la línea que va del negro al blanco y que corresponde a valores iguales de rojo, de verde y de azul.

Los diversos colores son puntos dentro del cubo y son definidos por vectores que se extienden desde el origen. Por conveniencia, se asume que todos los valores de color han sido normalizados, de tal forma que el cubo RGB es un cubo unitario.

Los colores representados utilizando este modelo se encuentran conformados por tres valores, uno para cada color primario. El número de bits usados para representar el color en este espacio es llamado la profundidad del color.

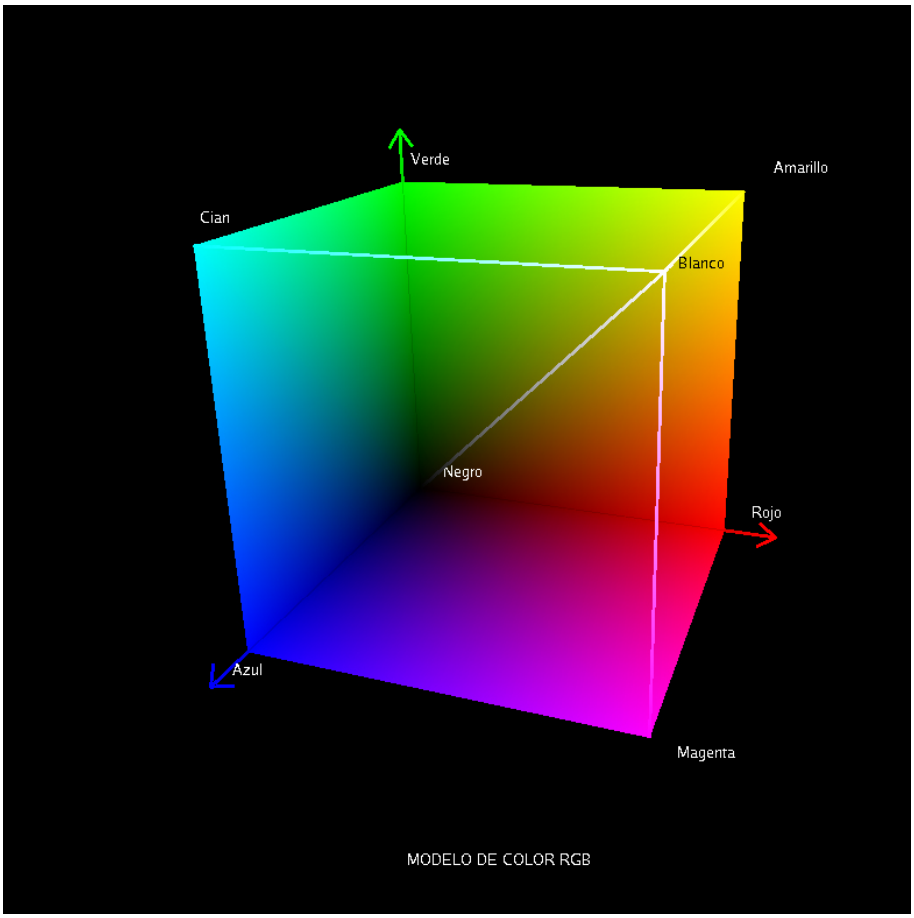


Figura B.3: Representación del modelo de color RGB

Este modelo de color es el que comúnmente se utiliza en los monitores, ya que se corresponde con la forma de mostrar un píxel en el monitor. El monitor utiliza tres fósforos que son iluminados al hacer incidir sobre ellos un haz de electrones, cada fósforo tiene un color especifico, uno es rojo, otro verde y otro azul.

B.2. Modelo de color CMY

Los colores cian, magenta y amarillo son los colores secundarios de luz o alternativamente los colores primarios de pigmentos. Por ejemplo, cuando una superficie cubierta de pigmento color cian es iluminada con luz blanca, la luz roja es absorbida, mientras que la luz verde y azul son reflejadas.

La mayoría de los dispositivos de impresión requieren de datos de entrada en formato CMY o necesitan hacer la conversión de RGB a CMY, esta conversión es desarrollada mediante la formula:

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{B.2}$$

donde R, G, B se encuentran en el rango [0,1].

En teoría la combinación de estos tres componentes debería producir negro, en la práctica es muy difícil y costoso, por lo que se usa una tinta adicional negra, creando el modelo de color CMYK (**C**yan, **M**agenta, **Y**ellow, blac**K**).

B.3. Modelo de color HSI

Como hemos visto, los modelos RGB y CMY son perfectos para su implementación en hardware, además de que el sistema RGB se ajusta perfectamente al modo en el que el sistema visual humano percibe los colores. Desafortunadamente estos modelos no describen los colores en términos prácticos para la interpretación humana. Cuando los humanos vemos un color, lo describimos mediante su matiz, su saturación y su brillo.

El modelo de color HSI (**H**ue, **S**aturation, **I**ntensity - Matiz, Saturación e Intensidad) es una herramienta ideal para desarrollar algoritmos basados en descriptores de color que son naturales e intuitivos a los humanos, que al final, son los desarrolladores y usuarios de los algoritmos.

El componente del matiz controla el espectro del color, desde el rojo hasta el violeta, pasando por el amarillo, el verde y el azul. El componente de la saturación controla la pureza del color, es decir que tanto color tiene. Por ejemplo, un rojo

profundo o fuerte es muy saturado, mientras que un rosa casi no tiene saturación rojo. El componente del brillo o intensidad controla que tan brillante u opaco se vera el color.

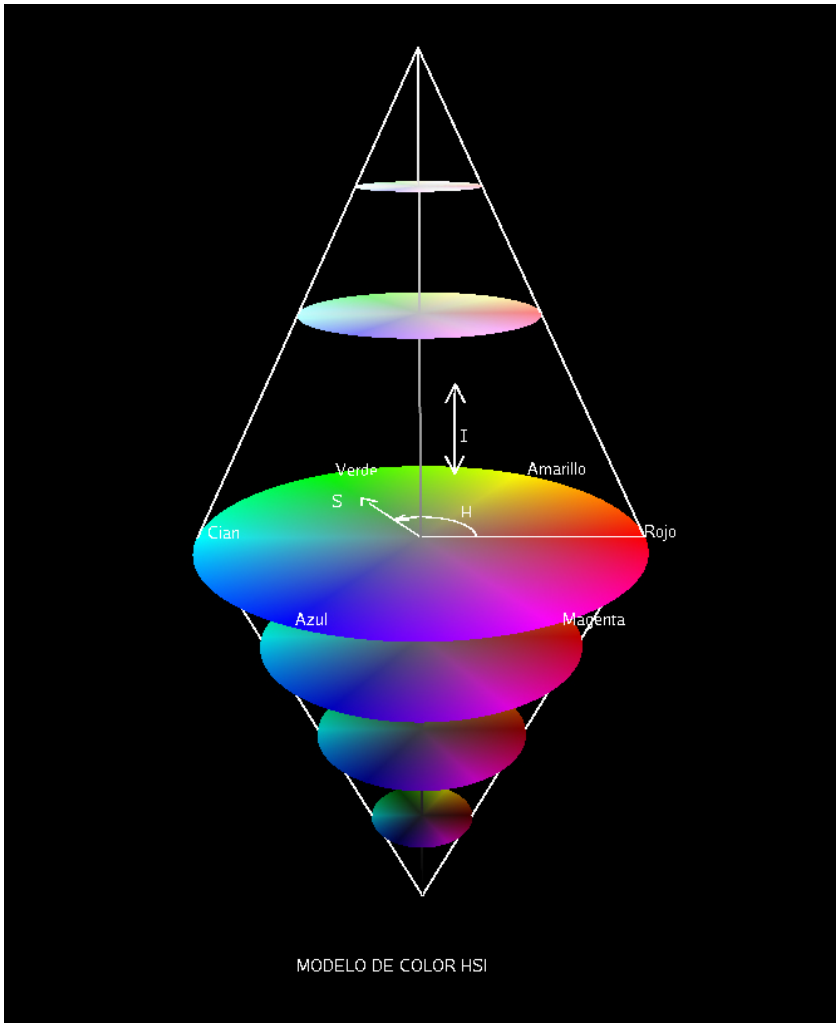


Figura B.4: Representación del modelo de color HSI

El modelo HSI es representado por un doble cono unido, como se muestra en la Figura B.4. La circunferencia del cono es el componente del matiz, alrededor de la cual se representan los colores puros del espectro. El matiz es determinado por un ángulo que parte de un punto de referencia. Usualmente un ángulo de 0 ° del eje rojo designa 0 matiz y este va incrementando en sentido contrario a las manecillas del reloj. La distancia del centro del

cono al extremo de la circunferencia es el eje de la saturación. Y la distancia del punto o pico del cono inferior al pico del cono superior forma el eje de la intensidad o brillo.

Otros espacio de color similares al HSI como el HSL, HSV y HSB por Lightness (iluminación), Value (valor) y Brightness (brillo), son sinónimos de Intensidad, pero representan de distinta forma como se ve el brillo del color y al igual que otros espacios de color parecidos, pero más particulares, son usados para aplicaciones específicas.

B.4. Conversión de RGB a HSI

A continuación se proporcionan las ecuaciones usadas para realizar la conversión de un color en el modelo RGB al modelo HSI [Gonzalez, 2002].

El componente H es obtenido usando la ecuación:

$$H = \begin{cases} \theta & \text{si} B \leq G \\ 360 - \theta & \text{si} B > G \end{cases} \tag{B.3}$$

Donde:

$$\theta = cos^{-1} \left\{ \frac{\frac{1}{2}[(R-G)+(R-B)]}{[(R-G)^2+(R-B)(G-B)]^{\frac{1}{2}}} \right\} \tag{B.4}$$

El componente S esta dado por:

$$S = 1 - \frac{3}{(R+G+B)}[min(R,G,B)] \tag{B.5}$$

Y finalmente el componente I lo obtenemos mediante la ecuación:

$$I = \frac{1}{3}(R+G+B) \tag{B.6}$$

En estas formulas se asume que los valores RGB se encuentran normalizados, es decir en el rango $[0,1]$ y que el ángulo θ inicia en el eje rojo del espacio de color HSI. El matiz se encuentra en el rango de $[0,360]$ y puede ser normalizado dividiéndolo por 360. La saturación y la intensidad se encuentran en el rango de $[0,1]$.

B.5. Conversión de HSI a RGB

A continuación se proveen las ecuaciones usadas para realizar la conversión de un color en el modelo HSI al modelo RGB, estas ecuaciones dependen del matiz, es decir del valor

del componente H en el color HSI. Hay tres intervalos de interés, de 0 ° a 119 °, de 120 ° a 239 ° y de 140 ° a 360 °. Como el color HSI de entrada se debe encontrar normalizado, el primer paso es multiplicar el componente H por 360 para colocarlo en el rango de $[0, 360]$. Las ecuaciones son las siguientes:

Para H en el rango $[0, 119]$ las ecuaciones son:

$$\begin{aligned} B &= I(1-S) \\ R &= I\left[1+\frac{S*cos(H)}{cos(60-H)}\right] \\ G &= 1-(R+B) \end{aligned} \tag{B.7}$$

Para H en el rango $[120, 239]$ las ecuaciones son:

$$\begin{aligned} H &= H-120 \\ R &= I(1-S) \\ G &= I\left[1+\frac{S*cos(H)}{cos(60-H)}\right] \\ B &= 1-(R+B) \end{aligned} \tag{B.8}$$

Para H en el rango $[140, 360]$ las ecuaciones son:

$$\begin{aligned} H &= H-240 \\ G &= I(1-S) \\ B &= I\left[1+\frac{S*cos(H)}{cos(60-H)}\right] \\ R &= 1-(G+B) \end{aligned} \tag{B.9}$$

En todos los casos, los resultados **R**, **G** y **B** se encuentran en el rango de $[0, 1]$.

Apéndice C

Acrónimos

2D: Dos dimensiones o bidimensional.

3D: tres dimensiones o tridimensional.

ANSI: American National Estándar Institute, Instituto Nacional Americano de Estándares.

API: Application Programming Interface, Interfaz de programación de aplicaciones.

ASCII: American Standar Code for Interchange of Information, Código estándar americano para el intercambio de Información.

GSNY: Grupo de Simulación Numérica de Yacimientos.

HSI: Modelo de color (Hue, Saturation, Intensity), (Matiz, Saturación, Intensidad).

IEEE: Institute of Electrical and Electronics Engineers, Instituto de Ingenieros Eléctricos y Electrónicos.

OO: Orientación a Objetos.

PEMEX: Petróleos Mexicanos.

POO: Programación Orientada a Objetos.

RGB: Modelo de color (Red, Green, Blue), (Rojo, Verde, Azul).

SNYM: Simulador Numérico de Yacimientos Multipropósito.

UI: Interface User, Interfaces de usuario.

UNAM: Universidad Nacional Autónoma de México.

UML: Unified Modeling Language, Lenguaje de modelado unificado.

Apéndice D

Glosario

API: Una API (del inglés Application Programming Interface - Interfaz de Programación de Aplicaciones) es el conjunto de funciones y procedimientos (o métodos si se refiere a programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción [Api].

Coordenadas Baricéntricas: El punto B es el baricentro de los puntos $A1, \ldots, Ar$ con pesos $a1, \ldots, ar$ si y sólo si

$$\sum_{i=1}^{r} a_i BA_i = 0$$

Si r=2 el baricentro se llama punto medio [Bari] [Baric].

Hidrocarburo: Compuesto resultante de la combinación del carbono con el hidrógeno.

ISO: Del griego iso que significa igual. International Organization for Standardization, Organización Internacional para la Estandarización.

Medio (Tipo de masa de roca - RockMass): Tipo de roca en la que se encuentra el yacimiento de hidrocarburo.

OpenGL: Estrictamente hablando es una interfaz de software a hardware grafico. En esencia, es una biblioteca de modelado y de gráficos 3D que es altamente portable y muy rápida [Wright, 2005] [OpenGL].

Pozo: Perforación profunda hecha para localizar o extraer hidrocarburos.

Qt: Es una biblioteca multiplataforma para desarrollar interfaces gráficas de usuario. Fue creada por la compañía noruega Trolltech. Utiliza el lenguaje de programación C++, pero permite usar también C, Python y Perl [Trolltech] [QT_Producto].

Simulador: Aparato que reproduce el comportamiento de un sistema en determinadas condiciones, aplicado generalmente para el entrenamiento de quienes deben manejar dicho sistema.

Widget: Literalmente artefacto, es un objeto visual que puede contener cero o mas objetos visuales.

Yacimiento: Sitio donde se halla naturalmente una roca, un mineral o un fósil.

Bibliografía

[Adamson, 1996] Adamson, Gordon & Crack, Martin & Gane, Brian & Gurpinar, Omer & Hardiman, Jim & Pointing, Dave *Simulation Throughout the Life of a Reservoir.* Art. 1996

[Adler, 1980] Adler. *Fisiología del ojo.* Editorial Médica Panamericana 1980

[Alonso, 1986] Alonso, Marcelo & Finn, Edward J. *Física. VolumenIII: Fundamentos Cuanticos y estadísticos.* Ed. Addison-Wesley iberoamericana. 1986

[Bañares, 2004] Bañares, Juan Palacio *Compendio de Ingeniería de Software 1.* Art. Dic. 2004

[Baxes, 1998] Baxes, Gregory A. *Digital Image Processing: Principles and applications.* Ed. John Wiley & Sons, Inc. 1994

[Blythe, 2005] Blyte, David & McReynolds, Tom *Advanced Graphics Programming Using OpenGL.* Ed. Morgan Kaufmann Publishers. 2005

[Booch, 1999] Booch, G. & Rumbaungh, I. & Jacobson, I. *El Lenguaje Unificado de Modelado.* Ed. Addison Wesley Iberoamericana. 1999

[Chen, 1988] Chen, Michel & Mountford, Joy S. & Sellen, Abigail *A Study in Interactive 3-D Rotation Using Control Devices.* Art. Computer Graphics, Vol. 22, Number 4, August 1988.

[RAE, 2002] Real Academia Española. *Diccionario de la Lengua Española. Vigésima Segunda Edición.* Ed. Espasa Calpe. 2002

[Efford, 2000] Efford, Nick. *Digital Image Processing a practical introduction using Java.* Ed. Addison-Wesley. 2000

[Ferre, SA] Ferré, Xavier & Sánchez, María Isabel *Desarrollo orientado a objetos con UML.* Art. Facultad de Informática - UPM

[Gardner, 1989] Gardner - Gray - Ó rahilly. *Anatomia 5ta edition.* Ed. Interamericana-McGraw-Hill 1989

[Glassner, 1995] Glassner, Andrews S. *Principles of Digital Image Synthesis. Volume one.* Ed. Morgan Kaufmann Publishers. 1995

[Gonzalez, 2002] Gonzalez, Rafael C. & Woods, Richard E. *Digital Image Processing. Second Edition.* Ed. Prentice Hall. 2002

[Halliday, 1997] Halliday, David & Resnick, Robert & Walker, Jearl. *Fundamentals of physics. Fifth edition. Part 4.* Ed. John Wiley & Sons, inc. 1997

[Hartmann, 1991] P. Hartmann-Petersen & J. N. Pigford *Diccionario de las Ciencias* Ed. Paraninfo. 1991

[Lehmann, 2002] Lehmann, Charles H. *Geometría Analítica.* Ed. Limusa, 2002

[Piattini, 2001] Piattini, Mario G. & Ruiz, Francisco & Usaola, Macario Polo & Villalba, José & Bastanchury, Teresa & Martínez, Miguel A. & Nistal, César *Mantenimiento del Software. Modelos, técnicas y métodos para la gestión del cambio.* Ed. Alfaomega Ra-Ma. 2001

[Pajares, 2002] Pajares, Gonzalo & de la Cruz García, Jesús M. *Visión por Computador. Imágenes Digitales y Aplicaciones.* Ed. Alfaomega. 2002

[Peña, 2006] Peña, Oscar & Reséndiz, Jesús Tadeo *Simulación Multifásica tridimensional en yacimientos naturalmente fracturados* Tesis 2006

[Russ, 2002] Russ, John C. *The Image Processing Handbook, Fourth edition.* Ed. CRC Press LLC. 2002

[Wright, 2005] Wright Jr., Richard S. & Lipchak, Benjamín *OpenGL Superbiblie, Third Edition.* Ed. Sams. 2005

[Api] *http://es.wikipedia.org/wiki/ Application_Programming_Interface*

[Bari] *http://pfortuny.sdf-eu.org/doc/Metodos/node24.html*

[Baric] *http://www.matematicas.unam.mx/gfgf/ga20052/ data/lecturas/lectura2.pdf*

[Conrec] *http://local.wasp.uwa.edu.au/ pbourke/papers/conrec/*

[CIE] *http://www.cie.co.at/cie/*

[CIE_HyperPhysic] *http://hyperphysics.phy-astr.gsu.edu/hbase/ vision/cie.html*

[CIE_Ufasta] *http://pub.ufasta.edu.ar/SISD/vision/cie.htm*

[Descartes] *http://descartes.cnice.mecd.es/*

[AgilMod] *http://www.agilemodeling.com/*

[OpenGL] *http://www.opengl.org*

[Projtex] *http://www.opengl.org/resources/code/samples/ mjktips/projtex/projtex.c*

[Trolltech] *http://trolltech.com/*

[QT_Producto] *http://trolltech.com/products/qt*

[UML] *http://www.uml.org/*