



Universidad
de Alcalá

COMISIÓN DE ESTUDIOS OFICIALES
DE POSGRADO Y DOCTORADO

ACTA DE EVALUACIÓN DE LA TESIS DOCTORAL

Año académico 2018/19

DOCTORANDO: **SOMOLINOS YAGÜE, ALVARO**
D.N.I./PASAPORTE: ****7287H

PROGRAMA DE DOCTORADO: **D442-INGENIERÍA DE LA INFORMACIÓN Y DEL CONOCIMIENTO**
DPTO. COORDINADOR DEL PROGRAMA: **CIENCIAS DE LA COMPUTACIÓN**
TITULACIÓN DE DOCTOR EN: **DOCTOR/A POR LA UNIVERSIDAD DE ALCALÁ**

En el día de hoy 07/06/19, reunido el tribunal de evaluación nombrado por la Comisión de Estudios Oficiales de Posgrado y Doctorado de la Universidad y constituido por los miembros que suscriben la presente Acta, el aspirante defendió su Tesis Doctoral, elaborada bajo la dirección de **IVAN GONZALEZ DIEGO //**.

Sobre el siguiente tema: *DISEÑO E IMPLEMENTACIÓN DE UN MOTOR GRÁFICO PARA LA CREACIÓN Y VISUALIZACIÓN DE OBJETOS COMPLEJOS EN PROGRAMAS DE SIMULACION ELECTROMAGNÉTICA*

Finalizada la defensa y discusión de la tesis, el tribunal acordó otorgar la CALIFICACIÓN GLOBAL¹ de (**no apto, aprobado, notable y sobresaliente**): SOBRESALIENTE

Alcalá de Henares, ⁷ de Junio de 2019

EL PRESIDENTE

Fdo.: JUAN MANUEL RIUS CASALS

*Juan Manuel Ferrando
Bastaller*

EL SECRETARIO

Fdo.: LORENA LOZANO PLATA

EL VOCAL

Fdo.: RAFAEL GOMEZ ALCALA

Con fecha 24 de junio de 2019 la Comisión Delegada de la Comisión de Estudios Oficiales de Posgrado, a la vista de los votos emitidos de manera anónima por el tribunal que ha juzgado la tesis, resuelve:

- Conceder la Mención de "Cum Laude"
 No conceder la Mención de "Cum Laude"

La Secretaria de la Comisión Delegada

FIRMA DEL ALUMNO,

Fdo.: SOMOLINOS YAGÜE, ALVARO

¹ La calificación podrá ser "no apto" "aprobado" "notable" y "sobresaliente". El tribunal podrá otorgar la mención de "cum laude" si la calificación global es de sobresaliente y se emite en tal sentido el voto secreto positivo por unanimidad.

INCIDENCIAS / OBSERVACIONES:

Por motivos de salud el presidente del tribunal de tesis doctoral, Dr. D. José Manuel Rius Casals, no ha podido asistir, por lo que fue avisado de urgencia el presidente suplente, Dr. D. Miquel Ferrando Bataller.

En aplicación del art. 14.7 del RD. 99/2011 y el art. 14 del Reglamento de Elaboración, Autorización y Defensa de la Tesis Doctoral, la Comisión Delegada de la Comisión de Estudios Oficiales de Posgrado y Doctorado, en sesión pública de fecha 24 de junio, procedió al escrutinio de los votos emitidos por los miembros del tribunal de la tesis defendida por **SOMOLINOS YAGÜE, ALVARO**, el día 7 de junio de 2019, titulada, *DISEÑO E IMPLEMENTACIÓN DE UN MOTOR GRÁFICO PARA LA CREACIÓN Y VISUALIZACIÓN DE OBJETOS COMPLEJOS EN PROGRAMAS DE SIMULACION ELECTROMAGNÉTICA* para determinar, si a la misma, se le concede la mención "cum laude", arrojando como resultado el voto favorable de todos los miembros del tribunal.


Por lo tanto, la Comisión de Estudios Oficiales de Posgrado y Doctorado **resuelve otorgar** a dicha tesis la

MENCIÓN "CUM LAUDE"

Alcalá de Henares, 24 de junio de 2019
 EL VICERRECTOR DE INVESTIGACIÓN Y TRANSFERENCIA
 F. Javier de la Mata de la Mata

Copia por e-mail a:

Doctorando: SOMOLINOS YAGÜE, ALVARO
 Secretario del Tribunal: LORENA LOZANO PLATA
 Director de Tesis: IVAN GONZALEZ DIEGO

Código Seguro De Verificación:	MÉ0pgEÉDYmNGUmQbJApcyA==	Estado	Fecha y hora	
Firmado Por	Francisco Javier De La Mata De La Mata - Vicerrector de Investigación Y Transferencia	Firmado	26/06/2019 09:09:42	
Observaciones		Página	13/14	
Uri De Verificación	https://vfirma.uah.es/vfirma/code/MÉ0pgEÉDYmNGUmQbJApcyA==			



Universidad
de Alcalá

ESCUELA DE DOCTORADO
Servicio de Estudios Oficiales de
Posgrado

DILIGENCIA DE DEPÓSITO DE TESIS.

Comprobado que el expediente académico de D./D^a _____
reúne los requisitos exigidos para la presentación de la Tesis, de acuerdo a la normativa vigente, y habiendo
presentado la misma en formato: soporte electrónico impreso en papel, para el depósito de la
misma, en el Servicio de Estudios Oficiales de Posgrado, con el nº de páginas: _____ se procede, con
fecha de hoy a registrar el depósito de la tesis.

Alcalá de Henares a _____ de _____ de 20 _____



Fdo. El Funcionario



Universidad
de Alcalá

Programa de Doctorado en Ingeniería de la
Información y del Conocimiento

**Diseño e implementación de un
Motor Gráfico para la creación y
visualización de objetos complejos en
programas de simulación
electromagnética**

Tesis Doctoral presentada por
Álvaro Somolinos Yagüe

2019



Universidad
de Alcalá

Programa de Doctorado en Ingeniería de la
Información y del Conocimiento

**Diseño e implementación de un
Motor Gráfico para la creación y
visualización de objetos complejos en
programas de simulación
electromagnética**

Tesis Doctoral presentada por
Álvaro Somolinos Yagüe

Director

Iván González Diego

Alcalá de Henares, 7 de marzo de 2019

D. Manuel Felipe Cátedra Pérez, Catedrático de Universidad del Departamento de Ciencias de la Computación de la Universidad de Alcalá,

D. Iván González Diego, Profesor Titular de Universidad del Departamento de Ciencias de la Computación de la Universidad de Alcalá,

HACEN CONSTAR:

Que, una vez concluido el trabajo de tesis doctoral titulado: **“Diseño e implementación de un Motor Gráfico para la creación y visualización de objetos complejos en programas de simulación electromagnética”** realizado por D. Álvaro Somolinos Yagüe, dicho trabajo reúne los suficientes méritos teóricos, que se han contrastado adecuadamente mediante validaciones experimentales y que son altamente novedosos. Por todo ello consideran que procede su presentación y defensa pública.

Y para que así conste, firman la presente en Alcalá de Henares, a 22 de Febrero de 2019.

El Director de la Tesis



D. Iván González Diego



Universidad
de Alcalá

El Tutor de la Tesis



D. Manuel Felipe Cátedra Pérez



Universidad
de Alcalá

Dr. D. José Javier Martínez Herraiz, Profesor Titular de Universidad del Área de Ciencias de la Computación e Inteligencia Artificial, en calidad de Coordinador del Programa de Doctorado en Ingeniería de la Información y del Conocimiento de la Universidad de Alcalá.

CERTIFICO: Que la Comisión Académica del Programa ha aprobado la presentación de la Tesis Doctoral titulada “Diseño e implementación de un Motor Gráfico para la creación y visualización de objetos complejos en programas de simulación electromagnética” realizada por D. Álvaro Somolinos Yagüe, dirigida por el Dr. D. Iván González Diego.

La Tesis Doctoral reúne los requisitos científicos de originalidad y rigor metodológicos para ser defendida ante un tribunal. Esta Comisión ha tenido también en cuenta la evaluación positiva anual del doctorando, habiendo obtenido las correspondientes competencias establecidas en el Programa.

Y para que así conste, firmo la presente en Alcalá de Henares, a 11 de marzo de 2019.



Dr. José Javier Martínez Herraiz

A mi familia y amigos...

Agradecimientos

Este trabajo es el fruto de muchas horas de trabajo, quisiera expresar mi agradecimiento a todos los que en mayor o menor medida han participado en él a lo largo de su proceso de gestación.

En primer lugar, me gustaría agradecer al Dr. Manuel Felipe Cátedra y Dr. Iván González Diego por haberme dado la oportunidad de trabajar en este proyecto y por todos los consejos y el apoyo recibidos durante estos años que han hecho posible la realización de este trabajo.

También me gustaría agradecer la ayuda recibida del resto de compañeros del grupo de investigación de Electromagnetismo Computacional de la Universidad de Alcalá y de todos los que han pertenecido a él en algún momento durante la realización de esta tesis.

Quiero hacer una mención especial a mis compañeros de trabajo, en especial a Gustavo Romero Vázquez y Javier Moreno Garrido, por su gran ayuda y su contribución al desarrollo y mejora de este trabajo.

Finalmente, hay incontables contribuyentes a este trabajo, todas las dudas que he resuelto en internet, he intentado referenciar los más importantes pero seguro que he omitido alguno. Desde aquí les quiero dar las gracias a todos ellos por compartir su saber con el mundo.

Resumen

En esta tesis se presenta el diseño y desarrollo de una nueva herramienta gráfica para el conjunto de software de simulación electromagnética newFASANT. El sistema presentado permite el modelado geométrico de escenarios complejos en 3D, la simulación electromagnética y la visualización de resultados.

La empresa NEWFASANT S.L. desarrolla desde hace años herramientas software orientadas al análisis electromagnético. Estas herramientas permiten reducir notablemente los costes de diseño y fabricación de elementos radiantes o sus entornos.

La parte de diseño del escenario de simulación es muy importante, es necesaria una herramienta potente para trabajar con los modelos geométricos en 3D y que se ajusten a las especificaciones del usuario. A su vez, el éxito del programa depende en gran medida de lo fácil e intuitivo que resulte trabajar con su interfaz de usuario.

Para la nueva herramienta, se ha creado un motor gráfico que permite trabajar con curvas y superficies NURBS de manera sencilla. Los núcleos de simulación electromagnética utilizan este tipo de superficies debido al alto rendimiento que proporcionan, ya que permiten definir formas arbitrarias de manera exacta utilizando muy poca información.

La nueva herramienta se ha desarrollado en Java utilizando Swing y se ha escogido Java3D para desarrollar el motor gráfico. Dichas librerías permiten la integración del escenario 3D en la herramienta y proporcionan las funcionalidades básicas de visualización. Por encima de esto se ha desarrollado la funcionalidad necesaria para trabajar con curvas y superficies NURBS, la cual es independiente y podría adaptarse a otras plataformas como OpenGL o JOGL.

La parte de simulación electromagnética se estructura en distintos módulos (RCS, Antenas, Estructuras Periódicas, IR...) dependiendo del tipo de simulación que se va a realizar. El usuario crea o importa el modelo geométrico en la herramienta e introduce los datos de simulación, generalmente mediante pestañas o interactuando con el escenario 3D. Posteriormente, la interfaz de usuario se encarga de escribir los ficheros de entrada y ejecutar los programas de simulación. Cuando el proceso ha terminado, se leen los ficheros de salida y la herramienta permite visualizar y exportar los resultados de forma atractiva.

Finalmente y como validación de la herramienta propuesta, se presenta el diseño de un reflectarray conformado sobre una superficie parabólica que utiliza la técnica VRT para

generar una discriminación en polarización circular. Para comprobar la validez del diseño, también se ha creado un demostrador, que ha sido fabricado y medido para comparar los resultados.

Palabras clave: Java3D, NURBS, Modelado Geométrico, Interfaz de Usuario.

Abstract

This thesis presents the design and development of a new graphical user interface (GUI) for the newFASANT electromagnetic simulation software. The developed system allows geometric modeling integration of complex 3D scenarios, electromagnetic simulations and results visualization in the same application.

The NEWFASANT S.L company has developed several software tools of electromagnetic analysis. These tools allows to reduce the design and manufacturing costs of the radiating elements and their environments.

The design process is very important, a powerful tool is needed to create 3D models of real environments that can be simulated. In turn, the program's success depends on how easy and intuitive it's to work with the user interface.

For the application, a new graphics engine has been developed, which allows to work with NURBS curves and surfaces in a simple way. The electromagnetic simulation kernel works with these surfaces due to the high performance, they allow to define accurately arbitrary shapes with less information.

The user interface has been developed in Java using Swing and Java3D to develop the graphics engine. These libraries allow the integration of the 3D scenario in the GUI and provide the basic visualization tools. It has also developed the necessary functionality to work with NURBS, it does not depend of Java3D and could be adapted to other platforms like OpenGL or JOGL.

The electromagnetic simulation process is divided into different modules (RCS, Antennas, Periodical Structures, IR ...) depending on the simulation type. The user creates or imports the geometric model in the GUI and selects the simulation parameters, usually by tabs or interacting with the 3D scene. Subsequently, the application writes the input files and execute the simulation program. When the process has finished, output files are read and the user interface displays the results in an attractive way.

Finally and as a validation of the proposed tool, a parabolic reflectarray designed to generate a circular polarization discrimination using the VRT technique is presented. To verify the design, a demonstrator has also been created, which has been manufactured and measured to compare the results.

Keywords: Java3D, NURBS, Computer-aided design (CAD), User Interface (GUI).

Índice general

Resumen	xiii
Abstract	xv
Índice general	xvii
Índice de figuras	xxiii
Índice de tablas	xxix
Índice de algoritmos	xxxii
1 Introducción	1
1.1 El inicio	1
1.2 Motivación de la tesis	2
1.3 Objetivos de la tesis	3
1.4 Antecedentes históricos	4
1.4.1 Herramientas de simulación electromagnética	4
1.4.2 Sistemas gráficos 3D	6
1.5 Estructura de la tesis	8
Bibliografía	9
2 Motor Gráfico	13
2.1 Introducción	13
2.2 Java 3D	13
2.2.1 Historia	14
2.2.2 Scene Graph	15
2.2.3 Clases de Java 3D	17

2.3	NURBS	18
2.3.1	Curvas NURBS	19
2.3.1.1	Curvas de Bézier	19
2.3.1.2	Curvas B-splines	22
2.3.1.3	Curvas Cónicas	24
2.3.1.4	Curvas NURBS	25
2.3.2	Superficies NURBS	27
2.3.2.1	Superficies NURBS Recortadas (Trimmed)	28
2.4	Curvas y Superficies NURBS en JAVA	29
2.4.1	Clase ControlPoint	30
2.4.2	Clase KnotVector	31
2.4.3	Clase NurbsCurve	31
2.4.4	Clase NurbsSurface	32
2.4.5	Clases Trimms3D y TrimmsUV	33
2.5	Algoritmos de Renderizado de Curvas y Superficies NURBS	34
2.5.1	Java 3D Geometry	35
2.5.2	Renderizado de Curvas	36
2.5.3	Renderizado de Superficies	38
2.5.3.1	Renderizado de superficies mediante líneas	38
2.5.3.2	Renderizado de superficies mediante polígonos	39
2.6	Transformaciones Geométricas	44
2.6.1	Matrices de transformación	45
2.6.2	Traslación	45
2.6.3	Escalado	46
2.6.4	Rotación	47
2.6.5	Simetría	49
2.6.6	Transformaciones en curvas y superficies NURBS	50
2.6.7	Invertir normales	51
2.7	Primitivas	51
2.7.1	Curvas NURBS	52
2.7.1.1	Línea	52
2.7.1.2	Circunferencia	52
2.7.1.3	Elipse	53

2.7.1.4	Parábola	54
2.7.1.5	Hipérbola	55
2.7.2	Superficies NURBS	57
2.7.2.1	Faceta	57
2.7.2.2	Disco	58
2.7.2.3	Cubo	58
2.7.2.4	Cilindro	59
2.7.2.5	Cono	60
2.7.2.6	Esfera	61
2.8	Búsqueda de Puntos en Coordenadas Paramétricas	62
2.8.1	Método del Gradiente Conjugado	63
2.8.2	Clase Gradient	64
2.8.3	Función a Minimizar	65
2.8.4	Búsqueda de Semillas	67
2.9	Algoritmos de Modelado de Superficies NURBS	68
2.9.1	Interpolación Global	68
2.9.2	Extrusión	71
2.9.3	Revolución	72
2.9.4	Skinned	74
2.9.5	Coons	76
	Bibliografía	78
3	Interfaz de Usuario: Diseño 3D	81
3.1	Introducción	81
3.2	Características Principales	82
3.3	Estructura del programa	83
3.4	Java Swing	85
3.4.1	Características de una aplicación Swing	86
3.4.2	Componentes principales de Swing	87
3.4.3	Netbeans IDE y Swing	88
3.5	La ventana principal: MainFrame	89
3.6	Panel de Geometría: Panel3D	91
3.6.1	Objetos Geométricos	92

3.6.1.1	PointShape	93
3.6.1.2	CurveShape	93
3.6.1.3	SurfaceShape	94
3.6.1.4	Objeto3D	97
3.6.2	Layers	97
3.6.3	Selección	98
3.6.4	Cámara	100
3.6.5	Ejes	101
3.6.6	Plano de Referencia	101
3.6.7	Pick	102
3.7	Consola de comandos	104
3.7.1	Clase ConsolePanel	105
3.7.2	Clase Command	106
3.7.3	Historial de Comandos	107
3.8	Parametrización de la geometría	108
3.8.1	Clase HashMapParameter	111
3.8.2	Evaluación de Funciones	112
3.9	Ficheros Geométricos	112
3.9.1	Formato NUR	113
3.9.2	Formato IGES	117
3.9.3	Formato MESH	119
	Bibliografía	121
4	Interfaz de Usuario: Simulación Electromagnética	123
4.1	Introducción	123
4.2	Módulos de Simulación Electromagnética	123
4.2.1	Estructura del programa	125
4.2.2	El Proyecto de Simulación	125
4.3	El núcleo electromagnético MONURBS	126
4.4	Módulo MOM	127
4.4.1	Ejemplo: Simulación de una bocina	128
4.4.2	Materiales	137
4.4.3	Parámetros de Simulación	140

4.4.3.1	Configuración de la simulación	140
4.4.3.2	Configuración del núcleo electromagnético	141
4.4.4	Tipos de Alimentación	142
4.4.4.1	Onda Plana	142
4.4.4.2	Dipolos	143
4.4.4.3	Diagramas de Radiación	143
4.4.4.4	Campo Impreso	145
4.4.4.5	Guía onda	145
4.4.4.6	Multipolo	146
4.4.5	Parámetros de Observación	146
4.4.5.1	Campo Cercano	146
4.4.5.2	Campo Lejano	148
4.4.6	Mallado	149
4.4.6.1	Características de la implementación	150
4.4.6.2	Visualización de la malla	151
4.4.7	Simulación	152
4.4.8	Resultados	153
4.4.8.1	Campo Lejano	153
4.4.8.2	Diagrama de Radiación	155
4.4.8.3	Campo Cercano	156
4.4.8.4	Visualización de Corrientes y Cargas sobre la Geometría	157
	Bibliografía	158
5	Resultados: Diseño de un Reflectarray Parabólico	161
5.1	Introducción	161
5.2	Descripción de la Antena	162
5.3	Diseño de la Celda del Reflectarray	166
5.4	Diseño del Reflectarray Parabólico	174
5.5	Diseño y Fabricación de un Demostrador	181
	Bibliografía	186
6	Conclusiones y Futuras Líneas de Trabajo	187
6.1	Conclusiones	187
6.2	Futuras Líneas de Trabajo	188

Índice de figuras

2.1	Java 3D	14
2.2	Scene Graph de un programa en Java 3D	16
2.3	Representación de la geometría en Java 3D	17
2.4	Modelo de barco formado por 353 superficies NURBS	18
2.5	Curva de Bézier	20
2.6	Polinomios de Bernstein	21
2.7	Ejemplo de evaluación de una curva B-spline	23
2.8	Tipos de curvas cónicas	24
2.9	Ejemplo de curva cónica	25
2.10	Ejemplo de curva NURBS	26
2.11	Superficie NURBS	28
2.12	Superficie NURBS Trimmed	28
2.13	Estructura del paquete 'NurbsLib'	30
2.14	Clase 'ControlPoint'	30
2.15	Clase 'KnotVector'	31
2.16	Clase 'NurbsCurve'	31
2.17	Clase 'NurbsSurface'	32
2.18	Clases 'Trimms3D' y 'TrimmsUV'	34
2.19	Renderizado de curvas adaptativo	36
2.20	Resultados del algoritmo adaptativo	38
2.21	Renderizado de superficies	38
2.22	Curvas de borde de una superficie NURBS	39
2.23	Malla de polígonos de una superficie NURBS	40
2.24	Matriz de renderizado de una superficie NURBS	41
2.25	Resultados del algoritmo de renderizado de superficies	44

2.26	Transformaciones geométricas: Translación	46
2.27	Transformaciones geométricas: Escalado	47
2.28	Transformaciones geométricas: Rotación	47
2.29	Transformaciones geométricas: Simetría	49
2.30	Curvas NURBS: Línea	52
2.31	Curvas NURBS: Circunferencia	52
2.32	Curvas NURBS: Elipse	53
2.33	Curvas NURBS: Parábola	54
2.34	Curvas NURBS: Hipérbola	55
2.35	Superficies NURBS: Faceta	57
2.36	Superficies NURBS: Disco	58
2.37	Superficies NURBS: Cubo	59
2.38	Superficies NURBS: Cilindro	60
2.39	Superficies NURBS: Cono	61
2.40	Superficies NURBS: Esfera	62
2.41	Point Inversion: Búsqueda de coordenadas paramétricas en NURBS	63
2.42	Clase 'Gradient'	64
2.43	Interpolación Global de Curvas NURBS	69
2.44	Interpolación Global de Superficies NURBS	71
2.45	Extrusión de una curva NURBS	72
2.46	Revolución de una curva NURBS	74
2.47	Superficie 'skinned' formada por círculos con distinto radio	76
2.48	Superficie 'coons' formada por cuatro curvas de borde	78
3.1	Interfaz de Usuario	82
3.2	Estructura del programa	84
3.3	Java Swing	85
3.4	Netbeans IDE: Diseñador de ventanas con Swing	88
3.5	Ventana principal: MainFrame	89
3.6	Componentes Swing de la ventana principal	90
3.7	Atributos de la clase 'MainFrame'	90
3.8	Estructura del árbol de Java3D en la interfaz	91
3.9	Interfaz 'ObjectInterface'	92

3.10	Objetos de tipo 'PointShape'	93
3.11	Objetos de tipo 'CurveShape'	94
3.12	Objetos de tipo 'SurfaceShape'	94
3.13	Ejemplo de objetos 'NurbsSurfaceShape'	96
3.14	Ejemplo de 'MeshSurfaceShape'	96
3.15	Ejemplo de 'Objeto3D' formado por 8 superficies NURBS	97
3.16	Ventana de Layers en la interfaz de usuario	98
3.17	Barra de estado: Tipos de selección	99
3.18	Lista para seleccionar varias superficies que coinciden	99
3.19	Funciones para controlar la cámara	100
3.20	Ejes cartesianos	101
3.21	Plano de Referencia	102
3.22	Barra de estado: Tipos de pick	102
3.23	Ejemplo de 'Pick' en una esfera	103
3.24	Consola de comandos	104
3.25	Clase 'ConsolePanel'	105
3.26	Clase 'Command'	106
3.27	Ejemplo de comando: box	107
3.28	Historial de comandos	107
3.29	Pestaña para editar el historial de comandos	108
3.30	Pestaña para definir parámetros variables	109
3.31	Pestaña para seleccionar la variación paramétrica y el paso	110
3.32	Comando 'box' usando el parámetro $myHeight = \{1, 2, 3\}$ en el segundo paso	110
3.33	Resultado de la variación paramétrica	111
3.34	Clase 'HashMapParameter'	111
3.35	Parámetros funcionales definidos en la consola	112
3.36	Aircraft.nur: 491 superficies NURBS	116
3.37	Tank.igs: 7590 superficies NURBS	119
3.38	Reflector.msh: 304.729 elementos	120
4.1	Módulos de simulación electromagnética	124
4.2	Estructura del paquete Modules	125

4.3	Directorio de un proyecto de simulación	126
4.4	Reflector analizado con MONURBS	127
4.5	Interfaz de usuario con el módulo MOM	128
4.6	Pestaña para introducir los parámetros de una bocina piramidal	129
4.7	Bocina añadida al panel junto a la previsualización de otra bocina	129
4.8	Pestaña de 'Simulation Parameters'	130
4.9	Pestaña de 'Solver Parameters'	131
4.10	Pestaña de 'Observation Directions'	131
4.11	Pestaña de 'Mesh Parameters'	132
4.12	Proceso de mallado	132
4.13	Resultados del proceso de mallado	133
4.14	Pestaña de 'Calculate'	133
4.15	Proceso de simulación	134
4.16	Resultados de la simulación: Ficheros de texto	135
4.17	Resultados de la simulación: Gráficas	136
4.18	Resultados de la simulación: Diagrama de radiación	136
4.19	Resultados de la simulación: Corrientes	137
4.20	Definición de materiales	138
4.21	Parámetros de los materiales	138
4.22	Pestaña para asignar material a una superficie	139
4.23	Pestaña de parámetros de simulación	140
4.24	Pestaña de configuración del núcleo electromagnético	141
4.25	Menú Source	142
4.26	Alimentación por onda plana	143
4.27	Alimentación por dipolo	143
4.28	Alimentación por diagrama de radiación	144
4.29	Alimentación por campo impreso	145
4.30	Alimentación por guía onda	145
4.31	Alimentación por multipolo	146
4.32	Puntos de observación en campo cercano	147
4.33	Resultados de los puntos de observación en campo cercano	148
4.34	Coordenadas esféricas	148
4.35	Puntos de observación en campo cercano	149

4.36	Pestaña de parámetros de mallado	150
4.37	Proceso de mallado	151
4.38	Visualización de la malla	152
4.39	Pestaña de parámetros de ejecución	152
4.40	Fichero de campo lejano	154
4.41	Gráfica de campo lejano	154
4.42	Diagrama de radiación	156
4.43	Fichero de campo cercano	157
4.44	Resultados de campo cercano	157
4.45	Visualización de corrientes sobre la geometría	158
5.1	Superficie parabólica de la antena reflectarray	162
5.2	Diagrama de radiación de la bocina con polarización LHCP	163
5.3	Comparativa entre la bocina real y el diagrama de radiación	164
5.4	Mallado del reflector parabólico	164
5.5	Diagrama de radiación del reflector parabólico	165
5.6	Resultados del reflector parabólico alimentado con LHCP	165
5.7	Distribución de corrientes del reflector parabólico	166
5.8	Estructura de la celda del reflectarray	166
5.9	Simulación de un dipolo paramétrico	167
5.10	Definición de parámetros del dipolo	168
5.11	Curva de fases del dipolo paramétrico	168
5.12	Simulación de una cruz paramétrica	169
5.13	Curva de fases de la cruz paramétrica	170
5.14	Simulación de una cruz paramétrica variando el ángulo de rotación	171
5.15	Fases en polarización circular de la cruz rotada (VRT)	171
5.16	Fases en circulares de la base de datos sin optimizar	172
5.17	Amplitud en circulares de la base de datos sin optimizar	172
5.18	Problemas de fase al rotar los elementos	173
5.19	Errores de fase de las cruces optimizadas	173
5.20	Fase en circulares de la base de datos optimizada	174
5.21	Amplitud en circulares de la base de datos optimizada	174
5.22	Alimentación del reflectarray parabólico	175

5.23	Parámetros del reflectarray parabólico	176
5.24	Esquema de funcionamiento del reflectarray	176
5.25	Layout del reflectarray parabólico	178
5.26	Mallado del reflectarray parabólico	178
5.27	Resultados del reflectarray parabólico alimentado con LHCP y RHCP . . .	179
5.28	Resultados del reflectarray parabólico alimentado con LHCP y RHCP (2) .	179
5.29	Alimentación con 3 bocinas	180
5.30	Resultados del reflectarray parabólico alimentado con tres bocinas	180
5.31	Resultados ampliados del reflectarray parabólico alimentado con tres bocinas	180
5.32	Demostrador generado en la interfaz de usuario	181
5.33	Resultados del demostrador en la interfaz de usuario	182
5.34	Resultados ampliados del demostrador	182
5.35	Fichero de geometría que contiene el layout del demostrador	183
5.36	Fotografías del demostrador en la cámara anecoica de la UPM	184
5.37	Medidas del demostrador RHCP y LHCP	185
5.38	Resultados ampliados del demostrador	185

Índice de tablas

3.1	Tipos de eventos en Swing	86
3.2	Formato de fichero .nur	113
3.3	Formato de fichero .nur: Curva NURBS	114
3.4	Formato de fichero .nur: Superficie NURBS	114
3.5	Formato de fichero .nur: Superficies NURBS trimadas	115
3.6	Formato de fichero .nur: Malla	115
3.7	Formato de fichero .nur: Objeto 3D	116
3.8	Formato de fichero IGES	117
3.9	Formato de fichero MESH	120
4.1	Formato de fichero .dia: Tipo 3DE	144
5.1	Materiales del reflectarray	167
5.2	Fichero de <i>script</i> para crear la geometría de la cruz paramétrica	169
5.3	Tamaño de las cruces optimizadas	173

Índice de algoritmos

2.1	Renderizado de curvas estático	36
2.2	Renderizado de curvas adaptativo	37
2.3	Comprueba si un punto está en la zona válida de la superficie NURBS . . .	40
2.4	Proyección de un punto en un segmento	42
2.5	Renderizado de superficies (1)	43
2.6	Renderizado de superficies (2)	43
2.7	Transformación de una superficie NURBS	51
2.8	Curvas NURBS: Línea	52
2.9	Curvas NURBS: Circunferencia	53
2.10	Curvas NURBS: Elipse	54
2.11	Curvas NURBS: Parábola	55
2.12	Curvas NURBS: Hipérbola	56
2.13	Superficies NURBS: Faceta	57
2.14	Superficies NURBS: Disco	58
2.15	Superficies NURBS: Cubo	59
2.16	Superficies NURBS: Cilindro	60
2.17	Superficies NURBS: Cono	61
2.18	Superficies NURBS: Esfera	62
2.19	NurbsCurveGradient: Función y Derivada	66
2.20	NurbsSurfaceGradient: Función y Derivada	67
2.21	Interpolación Global de Curvas NURBS	69
2.22	Interpolación Global de Superficies NURBS	70
2.23	Extrusión de una curva NURBS	72
2.24	Revolución de una curva NURBS	73
2.25	Generar curvas compatibles	74
2.26	Algoritmo para generar la superficie 'Skinned'	75
2.27	Algoritmo para generar la superficie 'Coons'	77
5.1	Función de usuario para generar el reflectarray 'circularBeam.java'	177

Capítulo 1

Introducción

1.1 El inicio

Desde el año 1995 aproximadamente, el ahora llamado Grupo de Electromagnetismo Computacional (GEC) [1], ha estado trabajando y realizando una gran aportación al mundo de las herramientas SW para el análisis electromagnético.

En su inicio estas herramientas fueron concebidas para la realización de dichos análisis por parte de personal especializado y familiarizado con la materia, que a su vez tenía que dedicar una cantidad considerable de tiempo para lanzar una simulación. Esto es, preparar todos los archivos de entrada a mano y construir la geometría desde otros programas, para pasarlos más tarde al núcleo mediante línea de comandos.

Posteriormente se dio un impulso para hacer más abiertas y amigables dichas herramientas mediante la programación de interfaces gráficas de ventanas, en las que se definió el esquema básico de funcionamiento de las mismas.

Las primeras interfaces estaban programadas en FORTRAN [2], permitían escribir automáticamente los ficheros de configuración y lanzar la simulación. Aunque son potentes y permiten ahorrar mucho tiempo, son mejorables en su aspecto y es muy difícil añadir nuevas funcionalidades como el modelado geométrico, por lo que pronto quedaron obsoletas.

Después se creó una interfaz de usuario basada en JAVA [3], que permitía visualizar modelos creados con otros programas y construir geometría mediante primitivas sencillas. También permitía configurar todos los parámetros, lanzar la simulación y visualizar resultados desde la propia interfaz. Para visualizar la geometría se utilizaban las librerías nativas de JAVA 3D [4].

Debido al éxito de esta interfaz y la potencia de los núcleos de simulación, en 2010 se crea la empresa NEWFASANT S.L. para continuar con el desarrollo y comercializar el software.

En la actualidad, debido a los avances tecnológicos, las herramientas de simulación electromagnética se han convertido en imprescindibles para realizar estudios de los sistemas y así optimizarlos antes de fabricar los prototipos. Es necesario disponer de herramientas que garanticen que los resultados obtenidos en las simulaciones sean muy parecidos a las prestaciones reales.

Estas herramientas permiten reducir notablemente los costes de diseño y fabricación de elementos radiantes o sus entornos. Así pues, la optimización de los productos se lleva a cabo mediante una fase de diseño y batería de simulaciones hasta conseguir que los resultados obtenidos para los prototipos cumplan las especificaciones requeridas. Tras la validación mediante simulaciones, se pasa a la fase de fabricación y medidas, volviendo a rediseñar-simular cuando las medidas no se ajustan a las necesidades o finalizando la fase de diseño de los sistemas.

En newFASANT [5] se han desarrollado herramientas de simulación que han permitido minimizar los costes de diseño a numerosas empresas en diferentes disciplinas de trabajo, como son Telefónica, SAAB, Volvo, Iberia, EADS, ESA, etc., sirviendo estos clientes como 'feedback' para la validación y continua mejora del software.

1.2 Motivación de la tesis

La fase de diseño es muy importante, es necesaria una potente herramienta que sea capaz de crear modelos en 3D que se ajusten a las especificaciones del usuario y que puedan ser simulados. Se han investigado los beneficios de usar modelados geométricos eficientes para optimizar las simulaciones. Debido al alto rendimiento conseguido con las superficies NURBS [6], tanto en términos de eficiencia como de precisión, se han podido llevar a cabo numerosos estudios para que los métodos numéricos de análisis trabajen directamente con este tipo de superficies [7–10].

Actualmente la herramienta es muy potente a la hora de realizar el análisis electromagnético pero carece de las ayudas necesarias para realizar un modelado complejo de la geometría a analizar. Los usuarios tienen que recurrir a un programa de modelado externo e importar el modelo en la herramienta. Esto supone una desventaja frente a herramientas de la competencia que incluyen todo el proceso de diseño.

Por otro lado la interfaz actual es muy difícil de mantener y ampliar, esto se debe principalmente a una mala estructuración del código y al desconocimiento de todos los requisitos del programa en su fase de diseño.

La intención de esta tesis es dar solución a estos problemas y a otros que tienen las empresas que desarrollan software de simulación científica que requieren de modelado geométrico:

- El coste de incluir una solución propietaria externa en cada paquete software es muy

grande para la pequeña y mediana empresa. Además suele ser anual o estar asociado a cada licencia.

- Dificultad de adaptarlo a las necesidades específicas de la empresa o dificultad de poder agregar nuevas funcionalidades.
- Dificultad para solucionar errores, muchas veces hay que esperar una actualización del software externo que solucione el problema.

Por ello, debido a los requisitos específicos de modelado para la simulación electromagnética, se ha apostado por la investigación para desarrollar un nuevo entorno de modelado geométrico propio que permita integrar la fase de diseño en la herramienta.

1.3 Objetivos de la tesis

El objetivo principal de la tesis es implementar una nueva interfaz gráfica para modelar escenarios complejos utilizando superficies NURBS. La interfaz integrará además todo el proceso de simulación y la llamada a los núcleos será transparente para el usuario, además permitirá la visualización de distintos tipos de resultados en forma de gráficas o mediante un panel 3D secundario. Posteriormente se incluirá algún ejemplo de validación de la interfaz.

Para cumplir el objetivo principal se tendrán en cuenta los siguientes apartados:

- Se implementará un motor gráfico en JAVA 3D. Este sistema se implementará como una API que el programa principal utilizará para dibujar los objetos en pantalla e interactuar con ellos.
- La mayor parte de la información se mostrará en la ventana principal y se intentará minimizar el uso de ventanas secundarias. El usuario interactuará con la geometría a través de una consola de comandos, esto permitirá utilizar el ratón para cazar puntos y construir geometría.
- La construcción de geometría debe permitir simulaciones paramétricas. Para ello se definirán una serie de variables que se usarán dentro de los comandos para crear geometría o realizar operaciones geométricas. Estas variables tomarán diferentes valores en cada simulación.
- La interfaz permitirá importar y exportar los formatos CAD (Computer Aided Design) más comunes como pueden ser IGES, DXF, STEP, STL y MSH [11].
- La interfaz se dividirá en distintos módulos para utilizar los distintos núcleos de simulación existentes (RCS, Antenas, Estructuras periódicas, GTD, ...).
- El sistema será multiplataforma para que pueda ser utilizado en entornos Windows, MacOS y GNU/Linux.

Por último, con vistas al futuro mantenimiento y ampliación del proyecto, se estudiará detenidamente la estructuración del código. Se tendrá especial cuidado en que sea sencillo incluir nuevos módulos y operaciones geométricas, así como nuevos tipos de formato de archivo y nuevas funcionalidades.

1.4 Antecedentes históricos

Como se ha podido observar en el primer apartado, la empresa NEWFASANT S.L. junto con el Grupo de Electromagnetismo Computacional (GEC) de la Universidad de Alcalá, han estado trabajando durante muchos años en el desarrollo de métodos numéricos para el análisis electromagnético. Esta tesis presenta el diseño y desarrollo de una nueva interfaz gráfica que de soporte a los distintos métodos de análisis desarrollados hasta el momento, abarcando desde el diseño de la estructura y la ejecución de la simulación, hasta la visualización de los resultados obtenidos.

1.4.1 Herramientas de simulación electromagnética

A partir de los años 80 y durante décadas, las herramientas de simulación empleadas en el mundo electromagnético computacional han tenido una gran demanda, pero hoy en día, gracias a los avances tecnológicos relacionados sobre todo con la computación y los métodos numéricos, ha hecho posible que, con mayor facilidad, se pueda investigar y realizar simulaciones electromagnéticas. Estas herramientas software abarcan un amplio margen de análisis y diseño, como son: el estudio de la compatibilidad electromagnética entre equipos en diferentes entornos, el análisis y diseño de antenas y circuitos, componentes pasivos de microondas, cálculo de RCS (Radar Cross Section) y de imágenes ISAR (Inverse Synthetic Aperture Radar), análisis de antenas embarcadas sobre estructuras complejas, análisis doppler, radio propagación tanto en entornos exteriores como en interiores, análisis de sistemas de radio, estudio del acoplo entre antenas, etc.

En el mundo de la ingeniería, de la investigación e incluso de los estudiantes de posgrado, dichas herramientas de simulación de campos electromagnéticos se han vuelto necesarias para la toma de decisiones. Estas herramientas son una alternativa importante a la realización de medidas en entornos reales, debido a que permiten analizar dinámicamente escenarios complejos del mundo real de forma precisa, con un bajo coste económico y un ahorro de tiempo considerable.

Para realizar los análisis y diseños de estas estructuras existen dos alternativas: por un lado, realizar medidas usando una estructura real o un modelo a escala, y por otro lado, emplear una herramienta software basada en técnicas numéricas que permita simular la estructura y predecir su comportamiento real. La toma de medidas, utilizando la estructura real o un prototipo de la misma realizado a escala, tiene asociado un elevado valor

económico. Esto se debe al coste para fabricar el prototipo, sumado al alto coste de las instalaciones de medición y mantenimiento de las mismas, así como las horas necesarias para llevar a cabo dichas medidas.

Las herramientas software suponen una gran alternativa para la industria y la investigación, debido a que con un menor coste económico y temporal se pueden construir escenarios y diseños reales para su análisis, permitiendo ajustar las características del diseño y detectar los problemas con antelación.

Cualquier método de análisis electromagnético se basa en la resolución de las ecuaciones de Maxwell [12]. Las ecuaciones de Maxwell son un conjunto de cuatro ecuaciones que describen por completo los fenómenos electromagnéticos. La gran contribución de James Clerk Maxwell fue reunir en estas ecuaciones largos años de resultados experimentales, debidos a Coulomb, Gauss, Ampere, Faraday y otros, introduciendo los conceptos de campo y corriente de desplazamiento, y unificando los campos eléctricos y magnéticos en un solo concepto: el campo electromagnético.

Las herramientas software de simulación electromagnética están basadas en dos tipos de técnicas: técnicas rigurosas y técnicas asintóticas. Las técnicas rigurosas, tal como el Método de los Momentos (MoM) [13], el Método de las Diferencias Finitas en el dominio del Tiempo (FDTD) [14], o el Método de los Elementos Finitos (FEM) [15], están basadas en la discretización de la geometría en elementos diferenciales. Las técnicas rigurosas, para estructuras eléctricamente grandes, requieren unos recursos computacionales muy elevados y suelen estar limitadas por estos recursos, aunque gracias a las nuevas investigaciones, estas limitaciones cada vez son menores. La principal ventaja de estas técnicas es que se obtiene una solución numéricamente exacta del problema.

Las técnicas asintóticas tienen la principal ventaja de que su coste computacional es independiente de la frecuencia de análisis, la precisión de los resultados se incrementa al incrementar la frecuencia. Estos métodos son especialmente adecuados para analizar problemas eléctricamente grandes, particularmente en el proceso de diseño donde el número de análisis puede ser muy alto. Las técnicas más populares son Óptica Geométrica y la Teoría Geométrica de la Difracción (GO/GTD) [16], y la Óptica Física junto con la Teoría Física de la Difracción (PO/PTD) [17].

La precisión de ambas técnicas está íntimamente relacionada con la fidelidad del modelo geométrico comparado con el escenario real. Esta relación de dependencia directa hace que la exactitud en los resultados obtenidos mejore cuanto más se aproxime el modelo geométrico a la estructura real. Una primera aproximación consiste en simplificar el modelo geométrico empleando estructuras canónicas para hacer más sencillo el tratamiento geométrico asociado con las técnicas rigurosas y asintóticas. La principal desventaja de esta opción es una pérdida de la precisión en los resultados. La segunda opción es obtener un modelo exacto del escenario que proporcione resultados que sean tan precisos como sea posible. En este caso, el precio es el coste computacional asociado con la solución. No

obstante, si se combinan los métodos electromagnéticos con alguna técnica de tratamiento geométrico o de aceleración de trazado de rayos, se puede alcanzar la solución de problemas complejos con una carga computacional razonable. Por tanto, el papel que juega en este tipo de análisis el modelado geométrico de la estructura real, es tan importante como la implementación de las técnicas numéricas eficientes. Esta es la principal causa por la que se utilizan superficies NURBS (Non-Uniform Rational B-Spline) para el modelado de las estructuras reales. Las superficies NURBS han sido las superficies paramétricas más populares en el mundo del electromagnetismo computacional, ya que se ajustan perfectamente a la estructura real requiriendo almacenar muy poca información para definirla. Además, estas superficies representan el enlace perfecto desde el mundo del electromagnetismo a otros mundos. De este modo queda perfectamente justificado el uso de este tipo de superficies en el proceso de modelado de la estructura.

En la actualidad existen numerosos programas de simulación electromagnética que abarcan todo el proceso de diseño y simulación de una estructura y suelen contener varios de estos métodos de simulación. Las herramientas más utilizadas son FEKO, CST y HFSS [18–20].

1.4.2 Sistemas gráficos 3D

El software de gráficos 3D por computadora comenzó a aparecer a finales de los años 70. El primer ejemplo conocido es 3D Art Graphics, un conjunto de efectos gráficos 3D creado por Kazumasa Mitazawa y lanzado en el mercado el junio de 1978 para el Apple II.

En 1981 Jim Clark fundó Silicon Graphics Inc. (SGI) [21], dedicada a la fabricación tanto de equipos de cómputo de alto nivel y software. Su invento llamado Geometry Engine, se trataba de una serie de componentes en un procesador VLSI que permitían calcular las operaciones requeridas para mostrar un escenario en 3D, como transformaciones de matrices, recortes, etc.

Posteriormente, con el nacimiento de las estaciones de trabajo (LISP, Paintbox Computers, Silicon Graphics) comenzó a expandirse la industria. Empezaron a aparecer las primeras tarjetas gráficas, que contienen un procesador especializado dedicado específicamente a estos cálculos y los primeros motores gráficos para dibujar en 3D.

Los polígonos tridimensionales son la base de prácticamente todos los gráficos 3D por ordenador. La mayoría de los motores gráficos están basados en el almacenaje de puntos, por medio de coordenadas tridimensionales (x, y, z) , líneas y polígonos que pueden ser triangulares, cuadrangulares, etc [22].

La generación de gráficos 3D actual va más allá del almacenaje de polígonos en la memoria de la computadora. Las gráficas de hoy en día son capaces de realizar operaciones de iluminación, sombreado, mapeado de texturas, rasterización, etc. A su vez son capaces de realizar miles de operaciones geométricas en tiempo real [23].

En la década de los 80 comenzaron a aparecer los primeros programas de diseño 3D. La empresa Autodesk Inc. [24] se fundó en 1982 con la intención de diseñar software para PC's con su paquete estrella AutoCAD. El primer software de Autodesk dedicado enteramente a la animación 3D fue 3D Studio para DOS en 1990.

En la década de los 90 aparecen numerosos paquetes de modelado y animación 3D, los cuatro más destacados son:

- Maya (Autodesk). Es el software de modelado más popular, ya que es utilizado por multitud de importantes estudios de efectos visuales en combinación con RenderMan, el motor de renderizado fotorrealista de Pixar.
- 3D Studio Max (Autodesk). Es el líder en el desarrollo 3D del videojuego y es muy utilizado a nivel amateur.
- Softimage XSI (Autodesk). En 1987, Softimage Inc, una compañía situada en Montreal, escribió Softimage|3D, que se convirtió rápidamente en el programa de 3D más popular de ese período. En 1994, Microsoft compró Softimage Inc. y comenzaron a reescribir SoftImage|3D para Windows NT. Actualmente, es uno de los productos de la compañía Autodesk.
- Lightwave 3D (Newtek). Es utilizado en multitud de estudios para efectos visuales y animación de cine y televisión.

Actualmente existen otras aplicaciones informáticas muy potentes, pero su uso está menos extendido, las más destacadas son:

- Blender. Es un programa libre de modelado, animación, iluminación y renderizado, con simulación de partículas y física de fluidos, cuerpos rígidos y suaves en tiempo real (necesarios para su motor de juegos), con posibilidad de edición y composición de imágenes y vídeo [25].
- Rhinoceros 3D. Un potente modelador bajo NURBS [26].

Los gráficos 3D se han convertido en algo muy popular, particularmente en videojuegos, al punto que se han creado interfaces de programación de aplicaciones (API) especializadas para facilitar los procesos en todas las etapas de la generación de gráficos por computadora. Estas interfaces han demostrado ser vitales para los desarrolladores de hardware para gráficos por computadora, ya que proveen un camino al programador para acceder al hardware de manera abstracta, aprovechando las ventajas de tal placa de vídeo.

Las siguientes interfaces para gráficos por computadora son particularmente populares:

- OpenGL [27]. Es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. La implementación más utilizada es la de C/C++, pero actualmente existen bindings a

otros lenguajes que tienen una gran acogida como JAVA con JOGL o LWJGL. La librería JAVA 3D actual que se utiliza en esta tesis es un API de alto nivel construida sobre JOGL.

- Direct3D [28]. Es parte de DirectX (conjunto de bibliotecas para multimedia), propiedad de Microsoft. Está disponible tanto en los sistemas Windows de 32 y 64 bits, como para sus consolas Xbox y Xbox 360.

Utilizando las API anteriores que pueden ser clasificadas como de 'bajo nivel', se han desarrollado los llamados núcleos geométricos que proveen de funciones de 'alto nivel' para crear geometría en 3D. La mayoría de estos motores utilizan el lenguaje C/C++.

Por un lado están los motores gráficos que se utilizan en videojuegos, la mayoría propietarios, que proveen de rutinas de programación que permiten el diseño, la creación y la representación de un videojuego. Desarrolladoras grandes de videojuegos como Epic, Valve y Crytek han lanzado al público sus motores o SDKs para que los usuarios interesados en el desarrollo de videojuegos puedan descubrir cómo se elaboran y así tener una introducción amplia a la industria y el desarrollo. Los más famosos son Unreal Engine [29], Source [30], CryEngine [31] o Unity [32].

Por otro lado están los núcleos geométricos utilizados para crear aplicaciones de diseño 3D, en este caso la mayoría también son propietarios. Dos de los núcleos de modelado gráfico más utilizados en herramientas de análisis electromagnético son ACIS [33] de Dassaults Systems y Parasolid [34] de Siemens. Ambos proporcionan un conjunto de librerías en C++/.NET que permiten implementar prácticamente cualquier opción de diseño CAD. En cuanto a software libre existe OpenCASCADE [35], que proporciona funciones para el modelado 3D de superficies y sólidos, visualización, intercambio de datos y desarrollo rápido de aplicaciones.

1.5 Estructura de la tesis

La presente tesis está compuesta por seis capítulos principales, en los que se expone gradualmente desde los motivos para llevar a cabo este trabajo al desarrollo completo del motor gráfico y la nueva interfaz de usuario, así como su posterior validación.

En el Capítulo 1 se realiza una introducción y se exponen los objetivos de la presente tesis. A su vez, se exponen los motivos que han llevado a la realización de este trabajo y se realiza un estudio del estado del arte actual.

En el Capítulo 2 se presenta el modelo geométrico que se va a utilizar, basado en las curvas y superficies NURBS y mostrando todas sus ventajas. Posteriormente se diseña e implementa el motor gráfico que se va a utilizar en la nueva interfaz de usuario.

En el Capítulo 3 se describe la estructura de la nueva interfaz de usuario y se integran las librerías desarrolladas en el capítulo anterior con el API de Java3D. Después, se

implementan todas las funcionalidades necesarias para trabajar con la geometría.

En el Capítulo 4 se describe la estructura de los módulos de simulación electromagnética de la interfaz de usuario, mostrando en detalle la implementación del módulo MOM y cómo se realiza el proceso de recogida de parámetros, ejecución de las simulaciones y visualización de resultados en la nueva interfaz.

En el Capítulo 5 se muestra el diseño completo de un reflectarray parabólico con discriminación en polarización circular. Todo el diseño se ha realizado utilizando la interfaz de usuario propuesta en esta tesis y corresponde a un caso real realizado durante mi estancia en la ETSI de Telecomunicación de la Universidad Politécnica de Madrid.

Para terminar el trabajo, en el Capítulo 6 se resumen las conclusiones obtenidas de la presente tesis, así como las principales líneas futuras de trabajo para seguir manteniendo y ampliando la herramienta.

Bibliografía

- [1] “Grupo de Electromagnetismo Computacional.” [Online]. Disponible en: <https://www.uah.es/es/investigacion/unidades-de-investigacion/grupos-de-investigacion/Electromagnetismo-Computacional/>
- [2] “Fortran 90.” [Online]. Disponible en: <http://www.fortran90.org/>
- [3] “Java 8 Standard Edition.” [Online]. Disponible en: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- [4] “Java 3D API.” [Online]. Disponible en: <http://www.java3d.org/index.html>
- [5] “Página web de NEWFASANT.” [Online]. Disponible en: <https://www.fasant.com/>
- [6] W. T. Les A. Piegl, *The NURBS Book (Monographs in Visual Communication)*. Springer, 2013.
- [7] M. Domingo, F. Rivas, J. Pérez, R. P. Torres, y M. F. Cátedra, “Computation of the RCS of complex bodies modeled using NURBS surfaces,” *IEEE Antennas and Propagation Magazine*, 1995.
- [8] F. S. d. Adana, I. González, O. Gutiérrez, L. Lozano, y M. F. Cátedra, “Method Based on Physical Optics for the Computation Radar Cross Section Including Diffraction and Double Effects of Metallic and Absorbing Bodies Modeled with Parameter surfaces,” *IEEE Transactions on Antennas and Propagation*, 2004.
- [9] I. González, E. García, F. S. d. Adana, y M. F. Cátedra, “Monurbs: A Parallelized Fast Multipole Multilevel Code For Analysing Complex Bodies Modelled By NURBS Surfaces,” *Applied Computational Electromagnetics Society Journal*, 2008.

- [10] C. Delgado, E. García, F. Cátedra, y R. Mitra, "Generation of Characteristic Basis Functions Defined over Large Surfaces Multilevel Approach," *IEEE Transactions on Antennas and Propagation*, 2009.
- [11] "Tipos de ficheros CAD." [Online]. Disponible en: <http://www.studioseed.net/blog/tipos-de-ficheros-cad/>
- [12] *A Dynamical Theory of the Electromagnetic Field*. James Clerk Maxwell, 1864. [Online]. Disponible en: https://en.wikisource.org/wiki/A_Dynamical_Theory_of_the_Electromagnetic_Field
- [13] R. F. Harrington, *Field Computation by Moment Methods*. Wiley, 1993.
- [14] A. Taflové y S. C. Hagness, *Computational Electrodynamics: The Finite-difference Time-domain Method*. Artech House, 2005.
- [15] O. C. Zienkiewics, *The Finite Element Method: Its Basis and Fundamentals*. Elsevier/Butterworth-Heinemann, 2005.
- [16] J. B. Keller, "Geometrical Theory of Diffraction," *Optical Society of America*, 1962.
- [17] J. S. Asvestas, "The Physical Optics method in Electromagnetic Scattering," *Journal of Mathematical Physics*, 1980.
- [18] "FEKO - EM Simulation Software." [Online]. Disponible en: <https://www.feko.info/>
- [19] "CST - Computer Simulation Technology." [Online]. Disponible en: <https://www.cst.com/>
- [20] "ANSYS HFSS - High Frequency Electromagnetic Field Simulation." [Online]. Disponible en: <http://www.ansys.com/products/electronics/ansys-hfss>
- [21] "Silicon Graphics - SGI." [Online]. Disponible en: https://es.wikipedia.org/wiki/Silicon_Graphics
- [22] "Gráficos 3D por computadora." [Online]. Disponible en: https://es.wikipedia.org/wiki/Gr%C3%A1ficos_3D_por_computadora
- [23] "Motores gráficos actuales." [Online]. Disponible en: <http://blogthinkbig.com/motores-graficos/>
- [24] "Autodesk Inc." [Online]. Disponible en: <https://www.autodesk.es/>
- [25] "Blender." [Online]. Disponible en: <https://www.blender.org/>
- [26] "Rhinceros." [Online]. Disponible en: <https://www.rhino3d.com/es/>
- [27] "OpenGL - Open Graphics Library." [Online]. Disponible en: <https://www.opengl.org/>

-
- [28] “Microsoft Direct3D.” [Online]. Disponible en: [https://msdn.microsoft.com/en-us/library/windows/desktop/hh309466\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/hh309466(v=vs.85).aspx)
- [29] “Unreal Engine.” [Online]. Disponible en: <https://www.unrealengine.com/products/unreal-engine-4>
- [30] “Source Engine.” [Online]. Disponible en: https://partner.steamgames.com/documentation/source_games
- [31] “CryEngine.” [Online]. Disponible en: <https://cryengine.com/features>
- [32] “Unity.” [Online]. Disponible en: <https://unity3d.com/get-unity>
- [33] “3D ACIS Modeling.” [Online]. Disponible en: <https://www.spatial.com/products/3d-acis-modeling>
- [34] “Parasolid.” [Online]. Disponible en: <https://www.plm.automation.siemens.com/es/products/open/parasolid/>
- [35] “Open CASCADE Technology.” [Online]. Disponible en: <https://www.opencascade.com/>

Capítulo 2

Motor Gráfico

2.1 Introducción

En este capítulo se describe el diseño y posterior desarrollo del motor gráfico de la interfaz. El motor gráfico es la parte más importante de la aplicación, ya que todo el modelado geométrico va a depender de él. Para el desarrollo del motor gráfico se utilizará la tecnología Java 3D [1], lo que posteriormente permitirá integrarlo en el programa de una manera sencilla.

En la primera parte, se realiza una descripción detallada de la tecnología escogida y su funcionamiento.

Después, se explica el funcionamiento de las curvas y superficies NURBS y los algoritmos utilizados.

En la tercera parte, se describe la implementación de este tipo de curvas y superficies en Java. Como las librerías de visualización no trabajan directamente con NURBS, se han desarrollado los algoritmos necesarios para visualizar las curvas y superficies NURBS en Java 3D.

2.2 Java 3D

Java 3D es una interfaz de programación utilizada para realizar aplicaciones con gráficos en tres dimensiones. Es similar a la biblioteca gráfica OpenGL, pero Java 3D tiene la característica de ser orientado a objetos.

Esta API proporciona una colección de constructores de alto-nivel para crear y manipular geometrías 3D y estructuras para dibujar esta geometría. Java 3D proporciona las funciones para creación de imágenes, visualizaciones, animaciones y programas de aplicaciones gráficas 3D interactivas [2].

Estos objetos geométricos residen en un universo virtual, que luego es renderizado. El



Figura 2.1: Java 3D

API está diseñado con flexibilidad para crear universos virtuales precisos de una amplia variedad de tamaños, desde astronómicos a subatómicos.

A pesar de toda esta funcionalidad, el API es sencillo de usar. Los detalles de renderizado se manejan automáticamente. Aprovechándose de los 'threads', Java 3D es capaz de renderizar en paralelo. El renderizador también puede optimizarse automáticamente para mejorar el rendimiento.

Un programa Java 3D crea ejemplares de objetos Java 3D y los sitúa en un estructura de datos del escenario gráfico. Este escenario gráfico es una composición de objetos en una estructura de árbol que especifica completamente el contenido de un universo virtual, y cómo va a ser renderizado [3].

Los programas Java 3D pueden escribirse para ser ejecutados como aplicaciones solitarias, por ejemplo un videojuego, o se pueden integrar en aplicaciones de ventanas creadas con Java Swing mediante la clase Canvas3D. También pueden crearse applets en navegadores que hayan sido extendidos para soportar Java 3D.

2.2.1 Historia

Intel, Silicon Graphics, Apple y Sun decidieron colaborar para el desarrollo del modo de representación 'scene graph' en sus APIs en 1996, y debido a esto, se creó un proyecto conjunto que logró convertirse en Java 3D. La versión Beta fue lanzada en marzo de 1998 y la primera versión oficial en Diciembre de ese mismo año, como una extensión estándar del JDK 2 de Java.

Desde mediados de 2003 hasta 2004 el desarrollo y producción de Java 3D fue discontinuado. En verano de 2004, el desarrollo de Java 3D fue abandonado por el resto de participantes, dejando a Sun a cargo de su continuidad y desarrollo, su última versión es la 1.5.2.

En enero de 2008, se anunció que las mejoras de Java 3D iban a ser puestas en 'stand by', para permitir el desarrollo de un motor gráfico 3D para Java FX [4].

Desde febrero de 2008 todo el código fuente de Java 3D ha sido liberado bajo la 2ª versión GPL. A partir de febrero de 2012 el proyecto Java 3D ha renacido con fuerza.

La nueva versión de Java 3D 1.6.0 utiliza JOGL 2.0 [5] para el renderizado y mediante OpenGL es acelerado por hardware.

2.2.2 Scene Graph

Un universo virtual de Java3D es definido por un grafo, que se conoce como Scene Graph [6]. El Scene Graph es un ordenamiento de objetos 3D en una estructura de árbol, que se crea instanciando objetos de las clases de Java3D. Con este grafo se define la geometría, sonido, luz, ubicación, orientación y apariencia de objetos visuales.

La relación más común es 'padre-hijo'. Un nodo Group puede tener cualquier número de hijos, pero sólo un padre. Un nodo hoja sólo puede tener un padre y no puede tener hijos. La otra relación es una referencia. Una referencia asocia un objeto NodeComponent con un nodo del escenario gráfico. Los objetos NodeComponent definen la geometría y los atributos de apariencia usados para renderizar los objetos visuales.

La línea básica de desarrollo de un programa Java 3D consiste en siete pasos [7] (a los que la especificación del API Java 3D se refiere como Receta) presentados a continuación. Esta receta se usa para crear la mayoría de programas en Java 3D.

1. Crear un objeto Canvas3D.
2. Crear un objeto VirtualUniverse.
3. Crear un objeto Locale, adjuntarlo al objeto VirtualUniverse.
4. Construir la rama de vista gráfica.
 - (a) Crear un objeto View.
 - (b) Crear un objeto ViewPlatform.
 - (c) Crear un objeto PhysicalBody.
 - (d) Crear un objeto PhysicalEnvironment.
 - (e) Adjuntar los objetos ViewPlatform, PhysicalBody, PhysicalEnvironment y Canvas3D al objeto View.
5. Construir la(s) rama(s) gráfica(s) de contenido.
6. Compilar la(s) rama(s) gráfica(s).
7. Insertar las ramas dentro de un objeto Locale.

Siguiendo los pasos anteriores en la figura 2.2¹ se puede ver la estructura general de un programa en Java 3D.

¹Fuente: https://docs.oracle.com/cd/E17802_01/j2se/javase/technologies/desktop/java3d/forDevelopers/j3dguide/j3dTOC.doc.html

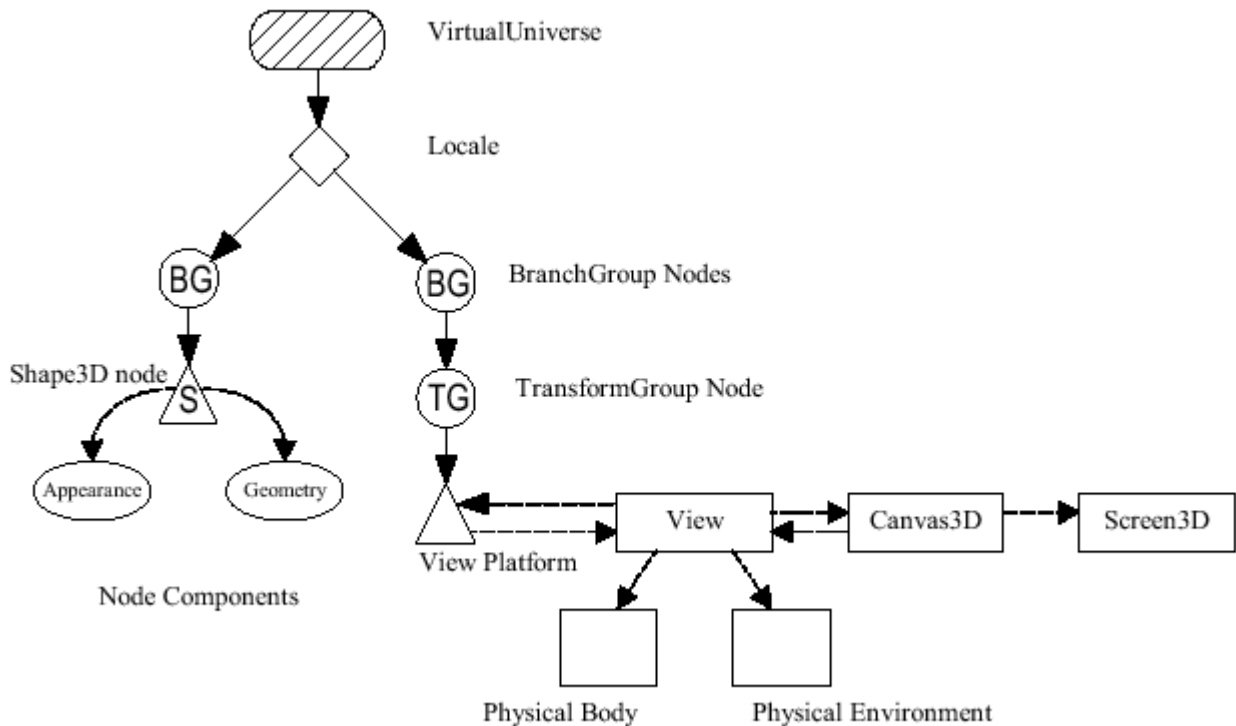


Figura 2.2: Scene Graph de un programa en Java 3D

Como se puede observar, el árbol tiene dos partes diferenciadas. En la parte izquierda se encuentra la rama de geometría de donde cuelgan todos los objetos Shape3D que van a ser renderizados. En la parte derecha se encuentra la rama de vista gráfica que define como se van a ver los objetos.

Los programas Java 3D escritos usando la receta básica tienen ramas de vista gráfica con idéntica estructura. La regularidad de la estructura de las ramas de vista gráfica también se encuentra en la clase SimpleUniverse. Los ejemplares de esta clase realizan los pasos 2, 3 y 4 de la receta básica.

El objeto SimpleUniverse crea una rama de vista gráfica completa para un universo virtual. Esta rama incluye un plato de imagen. Un plato de imagen es el rectángulo donde se proyecta el contenido para formar la imagen renderizada. El objeto Canvas3D, que proporciona una imagen en una ventana de nuestra pantalla, se corresponde con el plato de imagen.

La figura 2.3² muestra la relación entre el plato de imagen, la posición del ojo, y el universo virtual. La posición del ojo está detrás del plato de imagen. Los objetos visuales delante del plato de imagen son renderizados en el plato de imagen. El renderizado puede ser como una proyección de los objetos visuales sobre el plato de imagen. Esta idea se ilustra con los cuatro proyectores de la imagen (líneas punteadas).

²Fuente: http://java.sun.com/developer/onlineTraining/java3d/j3d_tutorial_ch1.pdf

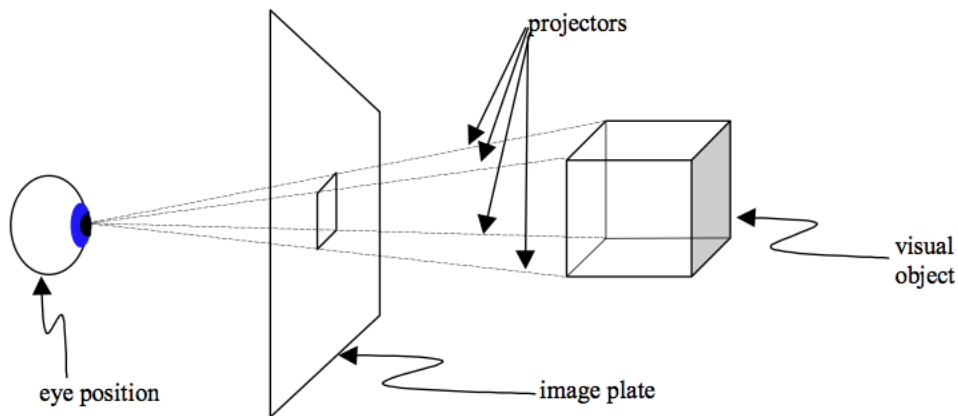


Figura 2.3: Representación de la geometría en Java 3D

2.2.3 Clases de Java 3D

Estas son las clases más importantes de Java 3D:

- **Canvas3D:** La clase Canvas3D extiende de Canvas de AWT y permite añadir el escenario 3D a una ventana o panel de Java.
- **BranchGroup:** Esta clase se usa para formar el escenario gráfico. Puede tener varios hijos de tipo Group o Leaf. El objeto de tipo BranchGroup es el único que puede ser añadido o eliminado en tiempo de ejecución.
- **TransformGroup:** Los objetos TransformGroup contienen transformaciones geométricas como traslaciones y rotaciones. La transformación se crea en un objeto Transform3D que contiene una Matriz 4x4 que representa la operación. Al ser de tipo Group aplica la transformación a todos los nodos hijos que cuelguen de él.
- **Shape3D:** Un nodo Shape3D define un objeto visual. Es una de las subclases de la clase Leaf; por lo tanto, los objetos Shape3D sólo pueden ser hojas en un escenario gráfico. El objeto Shape3D contiene la información sobre la forma y el color de un objeto. Esta información está almacenada en los objetos Geometry y Appearance referidos por el objeto Shape3D.
- **Behavior:** El propósito del objeto Behavior en un escenario gráfico es modificar el propio escenario gráfico, o los objetos que hay dentro de él, en respuesta a algunos estímulos. Un estímulo puede ser una pulsación de tecla, un movimiento del ratón, la colisión de objetos, el paso del tiempo, algún otro evento, o una combinación de estos. Las clases que implementan la selección de objetos o el movimiento de la cámara con el ratón extienden de la clase Behavior.

2.3 NURBS

NURBS (Non-Uniform Rational Basis-Spline) es un modelo matemático para definir curvas y superficies. La utilización de NURBS permite representar modelos complejos de manera precisa con muy poca información. Algunos motivos de su éxito son:

- Las NURBS pueden representar con precisión objetos geométricos estándar tales como líneas, círculos, elipses, esferas y toroides, así como formas geométricas libres como carrocerías de coches y cuerpos humanos.
- La cantidad de información que requiere la representación de una forma geométrica en NURBS es muy inferior a la que necesitan por separado las aproximaciones comunes.
- Existen varias formas estándar industriales para intercambiar la geometría NURBS. Los usuarios pueden y deberían ser capaces de transportar todos sus modelos geométricos entre los diferentes programas de modelado, renderizado, animación e ingeniería de análisis que hay en el mercado.
- Las NURBS tienen una definición precisa y muy conocida. Los algoritmos matemáticos de las superficies NURBS son rápidos y numéricamente estables.
- Tanto las superficies como las curvas NURBS mantienen la propiedad de invariancia ante diversas transformaciones geométricas, como la translación, rotación, escala, simetría, etc. Las transformaciones geométricas se aplican a sus puntos de control como se verá posteriormente.

Estas características y la importancia de la precisión en el modelado de la geometría para los métodos de análisis electromagnético son los principales motivos por los que la interfaz desarrollada en esta tesis trabaja con superficies NURBS.

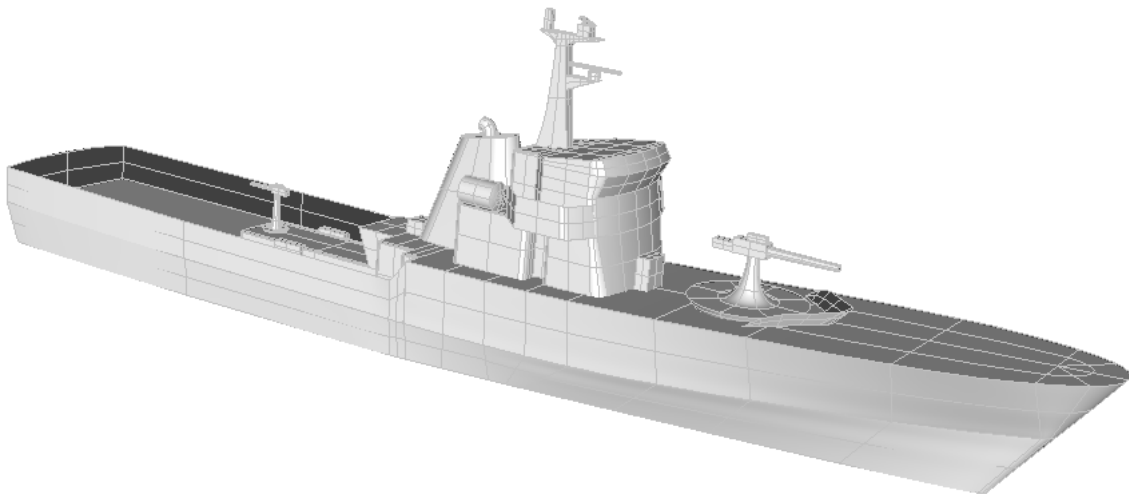


Figura 2.4: Modelo de barco formado por 353 superficies NURBS

En los siguientes apartados se va a describir de manera general la formulación matemática de las curvas y superficies NURBS. Se pretende dar una idea general de los principales algoritmos y formulaciones necesarios para modelar con estas entidades, sin acompañarlos con extensas demostraciones teóricas o explicaciones detalladas. Para una descripción más profunda se recomienda leer las referencias [8, 9].

2.3.1 Curvas NURBS

Para describir las curvas NURBS en toda su generalidad, es interesante introducir previamente un conjunto de curvas más simples para explicar sus propiedades. En concreto, el orden será el siguiente:

1. **Curvas de Bézier.** Tipo de curvas que se definen mediante un polígono de control con un número fijo de puntos en función del grado.
2. **B-Splines.** Son una extensión de las curvas de Bézier a partir del concepto de curva definida a trozos. Incorporan los nodos (knots)³. Se representan mediante el polígono de Boor.
3. **Cónicas.** Curvas con término racional que describen al conjunto de: parábolas, hipérbolas, elipses, círculos, etc.
4. **Curvas NURBS.** B-splines con término racional. Combinan las propiedades de las B-splines con las de las cónicas.

2.3.1.1 Curvas de Bézier

Se denomina curva de Bézier a un método de definición de una curva en serie de potencias. El método consiste en definir una serie de puntos de control, a partir de los cuales se calculan los puntos de la curva. La poligonal que forman los puntos de control se conoce como polígono de control.

³Se llama knots al conjunto de parámetros K_i pertenecientes a R tal que $K_i \in [0, 1]$, que forman una serie de valores posibles del parámetro de la curva y que sirven para su definición.

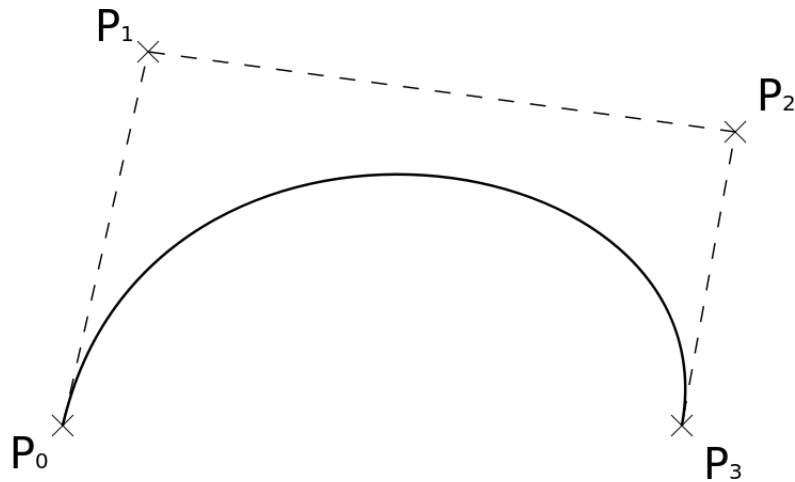


Figura 2.5: Curva de Bézier

Los puntos por los que pasa la curva se obtienen a partir de la coordenada paramétrica u definida en el intervalo $[0, 1]$ que define el espacio paramétrico.

Se llama orden de una curva de Bézier a la cantidad de puntos de control. Para dos puntos de control, la curva es de segundo orden y primer grado, por cada punto de control que se agrega, se aumenta en uno el grado del polinomio.

$$\text{orden} = \text{grado} + 1 \quad (2.1)$$

Una curva de Bézier se puede expresar en términos de polinomios de Bernstein como se muestra en la ecuación (2.2).

$$\vec{C}(u) = \sum_{i=0}^n \vec{b}_i B_i^n(u) \quad 0 \leq u \leq 1 \quad (2.2)$$

donde n es el grado y \vec{b}_i son los puntos de control de la curva, que en el espacio real definen el polígono de control. Las funciones $B_i^n(u)$ son los polinomios de Bernstein definidos por la ecuación:

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i} \quad (2.3)$$

Esa fórmula permite analizar las propiedades pero es inadecuada para el cómputo por la presencia de números combinatorios, en su lugar se utiliza el triángulo de Pascal y se calculan en forma recursiva.

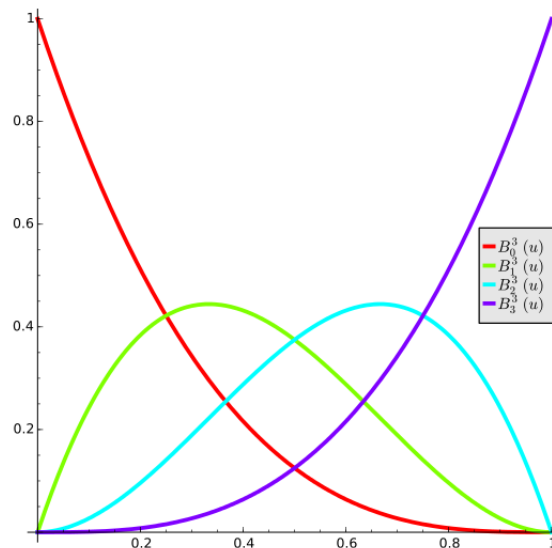


Figura 2.6: Polinomios de Bernstein

En la figura se representan los cuatro polinomios de Bernstein de tercer grado. Puede verse la simetría alrededor de $u = 0,5$ que podría deducirse de las simetrías de los números combinatorios. La simetría implica que se puede revertir la lista de puntos de control y la curva resultará igual. Cada punto de control se corresponde con un polinomio de Bernstein que actúa entonces como función de forma y miden la influencia de cada punto de control en cada punto de la curva. B_0^n toma el valor 1 en $u = 0$ mientras que B_n^n toma el valor 1 en $u = 1$ (en la figura, con polinomios de tercer grado, $n = 3$), eso implica que la curva pasa por los puntos de control inicial y final. Además ningún B toma el valor unitario en ningún otro valor de u , por lo que P_1 y P_n son los únicos puntos de control que coinciden con la curva.

Las curvas de Bézier tienen las siguientes propiedades:

- La curva de Bézier se encuentra en el interior de la envolvente convexa de los puntos de control.
- La curva es continua y derivable, de grado C^{n-1} en todo el dominio.
- El control de la curva es global. Modificar un punto de control implica modificar completamente la curva.
- Para efectuar una transformación afín de la curva es suficiente efectuar la transformación sobre todos los puntos de control.
- La curva comienza en el punto P_0 y termina en el P_n . Esta peculiaridad es llamada interpolación del punto final.
- La curva es un segmento recto si, y sólo si, todos los puntos de control están alineados.

2.3.1.2 Curvas B-splines

Tal como se ha descrito en la sección anterior, el grado de una curva de Bézier determina el número de puntos que contiene el polígono de control. Debido a esto, cuando se necesita aumentar el número de puntos para tener más flexibilidad en la definición de la curva, se debe aumentar el grado. A efectos prácticos y de coste computacional, grados excesivamente altos ($n > 10$) no son aconsejables. La extensión natural de estas curvas para obtener otras más flexibles, es la de las curvas polinómicas a trozos o splines.

La B de B-splines se refiere a “base”: se utiliza una base, en general polinómica. Las B-splines ganaron la batalla sobre todos los otros tipos definidos y hoy la variante: Non-Uniform Rational B-Spline o NURBS es la más utilizada. Se trata de curvas no-interpolantes, en general cúbicas, unidas con continuidad C^2 y cuyos puntos de control poseen control local. Pueden pensarse como un método de unión de curvas de Bézier de grado n con continuidad C^{n-1} . Normalmente, las B-splines y las NURBS se definen mediante las blending functions (como los polinomios de Bernstein para las curvas de Bézier) usando el algoritmo recursivo de Cox-De Boor [10] para calcularlas.

Definición

Una curva B-spline viene definida por su grado, su polígono de control y su secuencia de knots. Como una B-spline es una curva formada por diversos segmentos polinómicos, el grado de la curva será el mismo grado que tendrá cada uno de los segmentos.

El polígono de control es un conjunto de puntos en el espacio donde algunos de ellos pueden estar repetidos. La curva pasará por el primer y por el último punto y se acercará, de manera suave, a todos los otros puntos.

La secuencia de knots, es un conjunto de números reales en forma de lista no decreciente. Normalmente se definen en el intervalo $[0, 1]$ y, por tanto, el primero valdrá 0 y el último 1. Si un mismo valor está repetido r veces, se dice que tiene una multiplicidad r . Diferentes multiplicidades implicarán diferentes propiedades en relación al punto de control que corresponde al knot. En concreto, una multiplicidad igual al orden supondrá que la curva interpole al punto correspondiente. El primer knot de la lista y el último tendrán multiplicidad igual al orden para asegurar la interpolación de los puntos extremos.

El número de puntos y el de knots deben cumplir que:

$$N_k = N_p + o \quad \left| \begin{array}{l} N_k = \text{Número de knots} \\ N_p = \text{Número de puntos de control} \\ o = \text{orden} \end{array} \right. \quad (2.4)$$

Para evaluar una curva de este tipo se pueden usar varios algoritmos. Uno de ellos proviene de la inserción de knots y se enuncia de la siguiente manera: Dada una curva

definida por su grado n , polígono de control $\vec{P}_1, \dots, \vec{P}_L$ y knots u_0, \dots, u_{L+n} , para evaluarla en u se tendrá que encontrar u_I tal que $u \in [u_I, u_{I+1}]$. El valor de la curva en u será:

$$\vec{P}_i^k(u) = \frac{u_{i+n-k} - u}{u_{i+n-k} - u_{i-1}} \vec{P}_{i-1}^{k-1}(u) + \frac{u - u_{i-1}}{u_{i+n-k} - u_{i-1}} \vec{P}_i^{k-1}(u) \quad \left| \begin{array}{l} k = 1, \dots, n - r \\ i = I - n + k + 1, \dots, I + 1 \\ \vec{P}_i^0(u) = \vec{P}_i \end{array} \right. \quad (2.5)$$

entonces:

$$\vec{s}(u) = \vec{P}_{I+1}^{n-r}(u) \quad (2.6)$$

Donde r es la multiplicidad del knot $u_I = u$. Si no existe ese knot, entonces $r = 0$. Se deduce de esta ecuación que la evaluación de la curva pasa por calcular recursivamente una serie de interpolaciones lineales.

Ejemplo

Para ilustrar los puntos que intervienen en la evaluación de la curva para un valor del parámetro dado, se va a dar un pequeño ejemplo de una curva B-spline cuadrática ($n = 2$), definida mediante un polígono de control de 4 puntos. El polígono de control y la secuencia de knots se pueden ver en la siguiente figura:

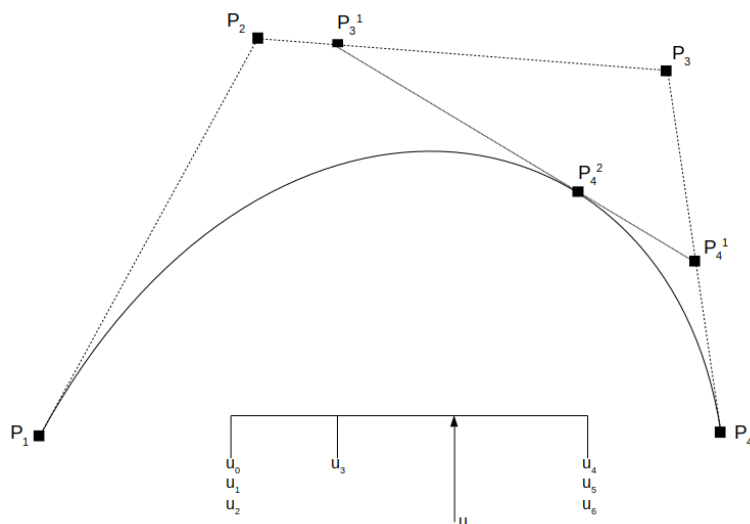


Figura 2.7: Ejemplo de evaluación de una curva B-spline

Suponiendo que $u \in [u_3, u_4]$, entonces $I = 3$ y la curva $\vec{s}(u) = \vec{P}_4^2$. El cálculo es el siguiente:

$$\vec{P}_3^1 = (1 - \alpha)\vec{P}_2 + \alpha\vec{P}_3 \quad \alpha = \frac{u - u_2}{u_4 - u_2} \quad (2.7)$$

$$\vec{P}_4^1 = (1 - \alpha)\vec{P}_3 + \alpha\vec{P}_4 \quad \alpha = \frac{u - u_3}{u_5 - u_3} \quad (2.8)$$

$$\vec{P}_4^2 = (1 - \alpha)\vec{P}_3^1 + \alpha\vec{P}_4^1 \quad \alpha = \frac{u - u_3}{u_4 - u_3} \quad (2.9)$$

Se comprueba que para una curva cuadrática, la función recursiva se evalúa en dos niveles y que los puntos de control que afectan al resultado son:

$$\begin{array}{ccc} \vec{P}_2 & \vec{P}_3^1 & \vec{P}_4^2 \\ \vec{P}_3 & \vec{P}_4^1 & \\ \vec{P}_4 & & \end{array}$$

y que los knots afectados son u_2, u_3, u_4, u_5 . A medida que aumenta el grado de la curva, aumentan los niveles de recursividad y también el número de puntos de control que intervienen en la solución.

2.3.1.3 Curvas Cónicas

Las curvas cónicas se obtienen al cortar un cono por medio de un plano (que en general no pasa por el vértice, ni es tangente a la superficie). Dependiendo del ángulo se obtienen las distintas curvas. Si el plano es perpendicular al eje se obtiene una circunferencia, si comienza a inclinarse respecto del eje se obtienen elipses de excentricidad cada vez mayor, hasta que, al ser el plano paralelo a la generatriz, se obtiene una parábola, para una inclinación mayor se obtienen hipérbolas.

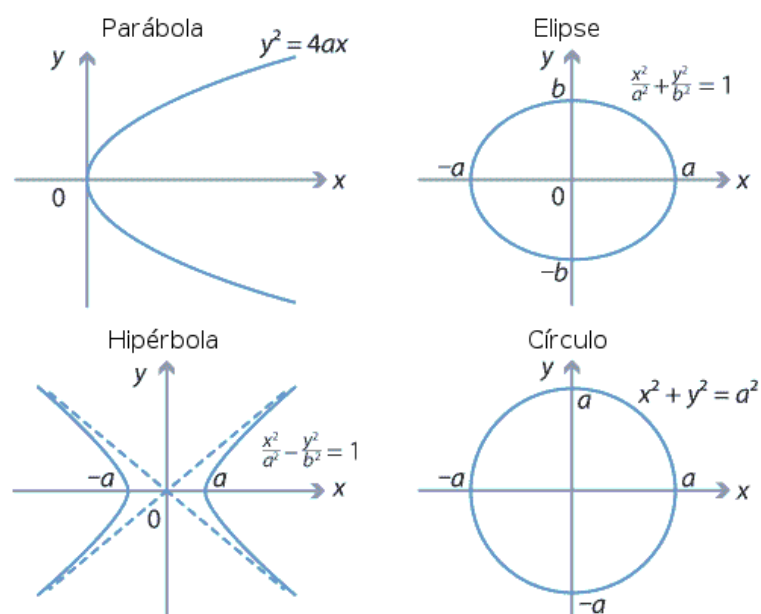


Figura 2.8: Tipos de curvas cónicas

Las curvas de Bézier de segundo grado (no-rationales) son siempre parábolas y no hay ninguna forma de que una curva de Bézier (de cualquier grado) represente una cónica diferente.

Para solventar esto, se introducen las curvas racionales. Las curvas racionales son el resultado de la proyección perspectiva de una curva con una dimensión más. La curva se define en coordenadas homogéneas $R^{d+1} \{x(u), y(u), z(u), w(u)\}$ pero se dibuja en el espacio proyectivo P^d dividiendo por el polinomio $w(u)$; de ahí la denominación de “racional”: son cocientes de polinomios.

Otra propiedad interesante es que las curvas así definidas poseen invariancia proyectiva. Como se ha visto, las curvas definidas mediante combinación afín tienen invariancia afín y por lo tanto lineal. Las transformaciones proyectivas son transformaciones lineales en una dimensión superior. Por lo tanto, una curva definida en una dimensión superior se puede transformar linealmente en R^{d+1} mediante la transformación (lineal) de sus puntos de control, para realizar la división perspectiva por $w(u)$ al final del proceso.

Dados los puntos y las tangentes inicial y final de una curva de Bézier de segundo grado, el punto de control central estará en la intersección de las tangentes. Para una curva no racional ya está todo dicho, pero para una curva racional aún se puede modificar la curva mediante los pesos.

La figura muestra dos formas de construir arcos de circunferencia con curvas racionales de Bézier de segundo grado. La primera imagen muestra un método general, la segunda está particularizada para un cuarto de circunferencia.

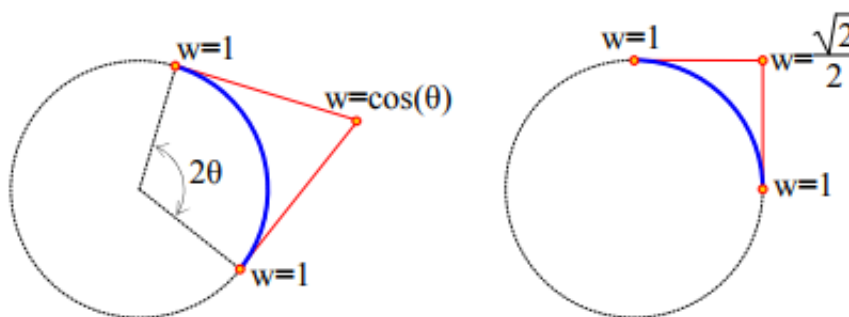


Figura 2.9: Ejemplo de curva cónica

2.3.1.4 Curvas NURBS

Una curva NURBS es una B-spline con término racional. Las expresiones para evaluar una NURBS son equivalentes a las de evaluación de las B-splines, añadiendo los pesos w_i . La siguiente fórmula es equivalente a la ecuación (2.5), añadiendo los componentes del término racional. El cálculo de los pesos w_i^k se realiza también de manera recursiva. Para evaluar la curva en u se debe encontrar u_I tal que $u \in [u_I, u_{I+1}]$.

$$\vec{P}_i^k(u) = \frac{(1 - \alpha_i^k)w_{i-1}^{k-1}\vec{P}_{i-1}^{k-1}(u) + \alpha_i^k w_i^{k-1}\vec{P}_i^{k-1}(u)}{w_i^k} \quad \left| \begin{array}{l} k = 1, \dots, n - r \\ i = I - n + k + 1, \dots, I + 1 \\ \vec{P}_i^0(u) = \vec{P}_i \\ w_i^0 = w_i \end{array} \right. \quad (2.10)$$

$$\alpha_i^k = \frac{u - u_{i-1}}{u_{i+n-k} - u_{i-1}}$$

$$w_i^k = (1 - \alpha_i^k)w_{i-1}^{k-1} + \alpha_i^k w_i^{k-1}$$

entonces:

$$\vec{s}(u) = \vec{P}_{I+1}^{n-r}(u) \quad (2.11)$$

Ejemplo

Para ilustrar la definición de una curva racional mediante NURBS, se ofrece el ejemplo de la siguiente figura. Los valores de las coordenadas están dados para el círculo unitario centrado en el origen. Cualquier otro círculo se obtiene mediante traslaciones y escalados. Una elipse tendría una definición equivalente, sin más que transformar el cuadrado que forman los puntos de control en un rectángulo.

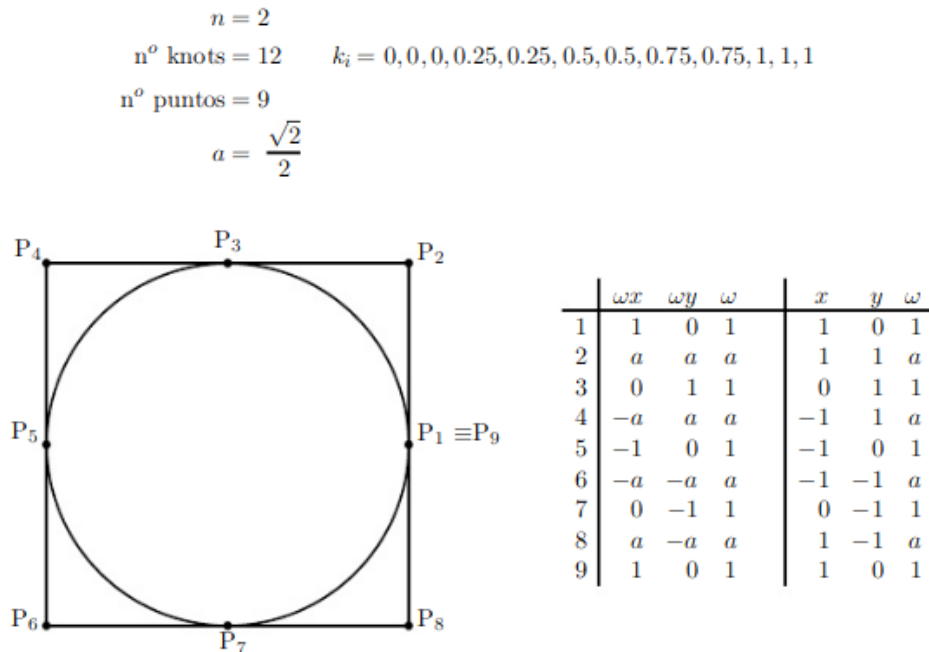


Figura 2.10: Ejemplo de curva NURBS

El cambio de los pesos de los puntos de control de una curva NURBS afectan, en general, a la forma de la curva. En concreto, el incremento del peso de un punto tenderá a atraer la curva hacia ese punto.

Se dice que la curva está en la forma estándar si los pesos de los dos puntos extremos son iguales a uno. Nos interesa, en general, que la curva esté en esta forma para poder realizar algunas operaciones como creación de superficies a partir de su contorno. Toda curva se puede convertir a su forma estándar mediante:

$$\hat{w}_i = \left(\sqrt[n]{\frac{w_1}{w_{n+1}}} \right)^{i-1} \quad i = 1, \dots, n+1 \quad (2.12)$$

$$\hat{w}_i = \frac{1}{w_1} w_i \quad i = 1, \dots, n+1 \quad (2.13)$$

Posteriormente se describirá la implementación de este tipo de curvas y los algoritmos necesarios para evaluarlas. A continuación se muestran las superficies NURBS, que no es más que una extensión de las curvas en dos direcciones: u y v .

2.3.2 Superficies NURBS

Si se realiza la extensión de las curvas NURBS en dos direcciones, se obtienen las superficies NURBS. Estas son el producto tensorial de dos curvas.

Las superficies se definen mediante los puntos de control, el grado y la lista de knots en cada una de las direcciones u y v . Tanto el grado como el número de knots, no tienen que coincidir para u y v . Los puntos de control se definen mediante una matriz en el espacio que, en general, tiene diferente número de puntos según cada una de las direcciones.

La expresión general para evaluar una superficie NURBS es la siguiente:

$$\vec{s}(u, v) = \frac{\sum_i \sum_j w_{ij} \vec{P}_{ij} N_i^m(u) N_j^n(v)}{\sum_i \sum_j w_{ij} N_i^m(u) N_j^n(v)} \quad (2.14)$$

La mayoría de las operaciones a realizar sobre estas superficies, incluida la evaluación, pueden realizarse primero sobre un sentido, obtener como resultado una curva y aplicar la operación sobre el otro sentido. De esta manera se reduce el problema de operar sobre una superficie al problema de operar sobre un conjunto de curvas.

Ejemplo: Dada una superficie con m puntos en la dirección u y n puntos en la dirección v , que se quiere evaluar en (u_i, v_i) , hay que evaluar las m curvas formadas con los puntos de la matriz de control tomados como columnas y evaluarlas en $v = v_i$. El resultado es un conjunto de m puntos que forman una curva que hay que evaluar para $u = u_i$.

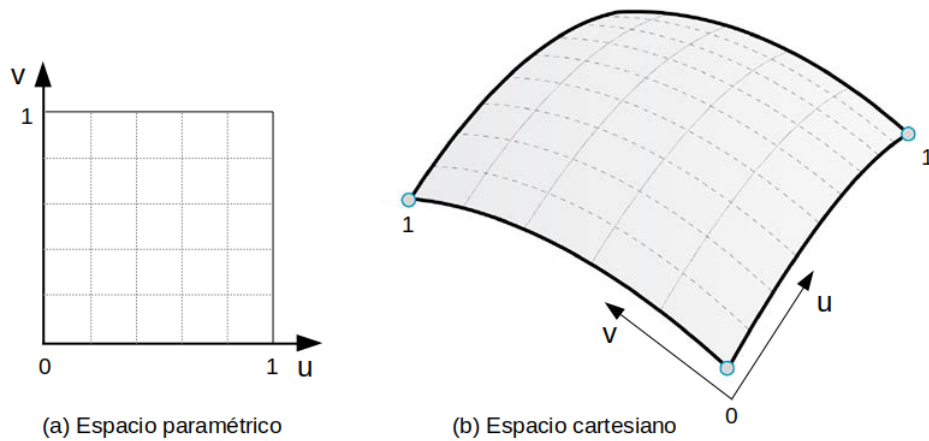


Figura 2.11: Superficie NURBS

En la imagen de la izquierda (a) se puede ver el espacio paramétrico en las dos direcciones u y v con la matriz de puntos de control. A la derecha (b) se encuentra la misma matriz de puntos de control en el espacio cartesiano y la superficie NURBS.

2.3.2.1 Superficies NURBS Recortadas (Trimmed)

Las superficies NURBS recortadas⁴ se definen mediante una superficie NURBS como base y una o varias curvas recortadoras. Según el sentido de la curva, este será un contorno exterior o un agujero. Las curvas recortadoras se definen en el espacio paramétrico u y v de la superficie.

Los contornos o agujeros delimitan el espacio visible de la superficie, se pueden crear utilizando una o varias curvas NURBS para formar un bucle cerrado (loop). En general, una superficie NURBS recortada solo puede tener un bucle que defina su contorno y uno o varios bucles para formar agujeros.

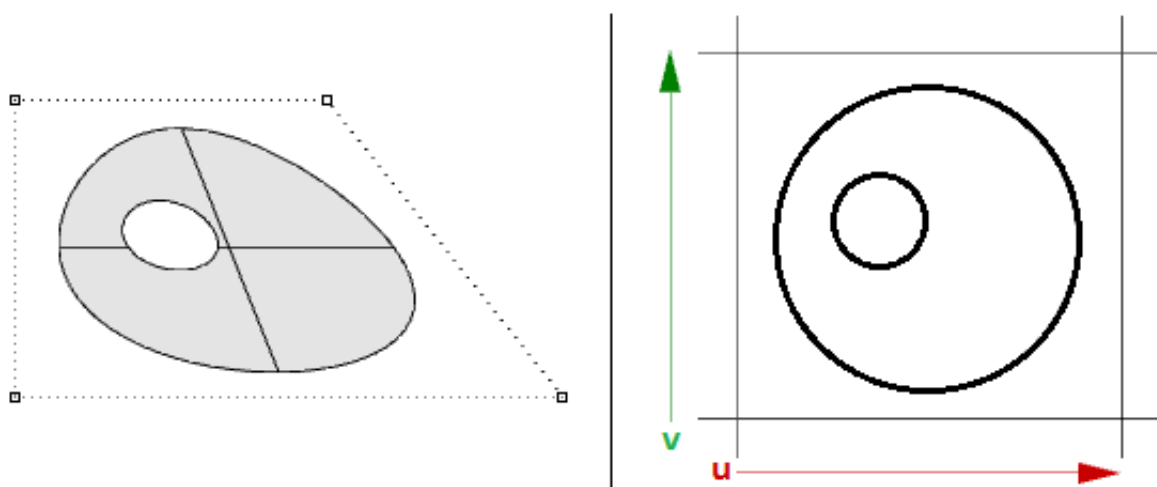


Figura 2.12: Superficie NURBS Trimmed

⁴En modelado geométrico también se suele usar el término *trimada* para referirse a este tipo de superficies.

El algoritmo de renderizado propuesto se encarga de recortar la superficie NURBS para visualizarla correctamente. Debido a que es muy complicado para el usuario obtener la superficie deseada, dibujando las curvas recortadoras en el espacio paramétrico u y v de la superficie, se han desarrollado los algoritmos necesarios para obtenerlas a partir de las curvas en 3D.

2.4 Curvas y Superficies NURBS en JAVA

Para poder visualizar curvas y superficies NURBS en Java 3D hay que implementar una serie de algoritmos en Java para poder trabajar con este tipo de objetos. En este apartado se describe la implementación del paquete 'NurbsLib' que se ha desarrollado para la nueva interfaz.

En el libro *The NURBS Book* [9] se encuentran definidos, de manera teórica, muchos de los algoritmos necesarios para trabajar con curvas y superficies NURBS. También, se pueden encontrar varios proyectos en fase de desarrollo con alguna implementación de los algoritmos en Java. Los más importantes son JGeom [11], iGeo [12] y GeomSS [13]. Desgraciadamente, estos proyectos están abandonados, pero han resultado muy útiles como referencia.

El paquete 'NurbsLib' contiene las siguientes funcionalidades:

- Objetos para definir curvas y superficies NURBS *trimadas*.
- Algoritmos de renderizado para visualizar NURBS en Java 3D.
- Transformaciones geométricas.
- Algoritmos para crear distintos tipos de primitivas.
- Búsqueda de puntos en superficies y curvas paramétricas.
- Algoritmos de modelado de superficies NURBS.

El paquete 'NurbsLib' se engloba dentro del paquete 'Geometry' que contiene el motor gráfico de la interfaz. En el paquete 'Geometry' es donde se realiza la conexión de los objetos NURBS con objetos de JAVA 3D para que puedan ser visualizados. Esta conexión se describe en el siguiente capítulo, ya que pertenece a la interfaz de usuario.

En la figura 2.13 se muestra la estructura del paquete 'NurbsLib' y como se relacionan los distintos elementos.

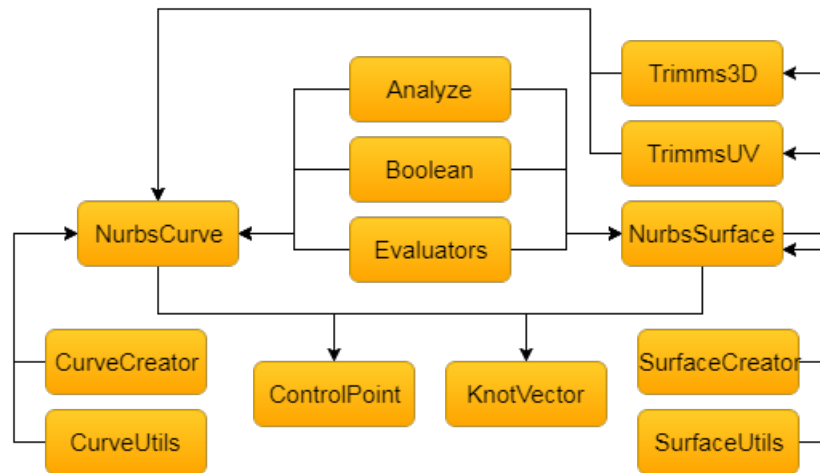


Figura 2.13: Estructura del paquete 'NurbsLib'

Las clases 'NurbsCurve' y 'NurbsSurface' contienen la información de una curva o superficie NURBS. Se puede observar que dependen de la clase 'ControlPoint' que almacena un punto de control y 'KnotVector' que contiene la estructura de un vector de nodos. El resto de clases implementan las distintas funcionalidades para trabajar con objetos de tipo 'NurbsCurve' o 'NurbsSurface'.

A continuación se describen las clases y algoritmos más importantes que forman el paquete 'NurbsLib'.

2.4.1 Clase ControlPoint

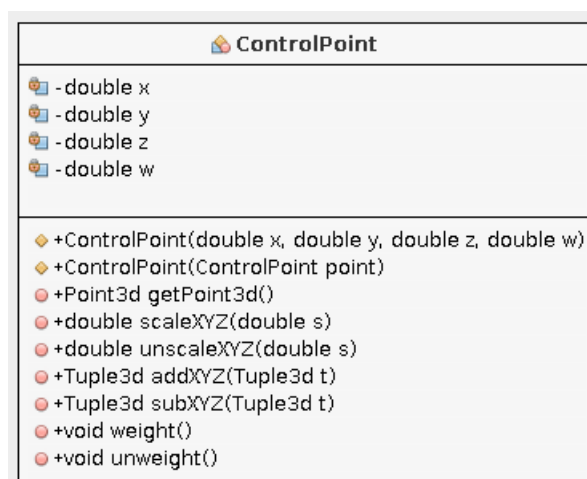


Figura 2.14: Clase 'ControlPoint'

La clase 'ControlPoint' contiene la información de un punto de control. Un punto de control está definido por las coordenadas XYZ que indican su posición y su peso w .

Los métodos implementan las operaciones comunes sobre un punto de control. La operación *weight* consiste en multiplicar las coordenadas XYZ por el peso w .

2.4.2 Clase KnotVector

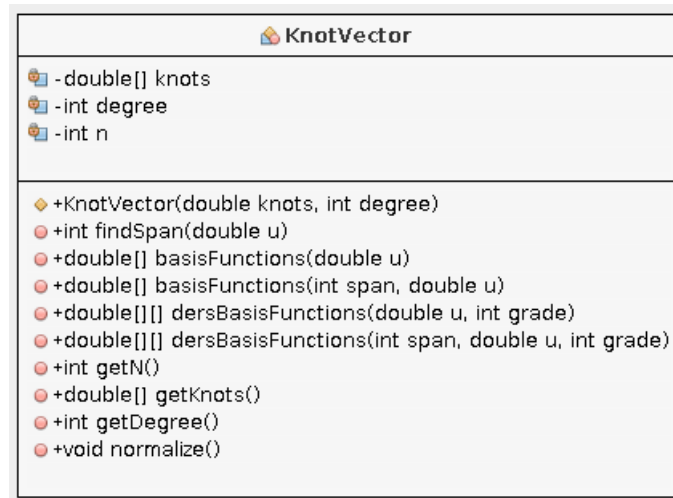


Figura 2.15: Clase 'KnotVector'

La clase 'KnotVector' contiene la información de los knots y el grado de la curva o superficie NURBS. En una superficie hay un objeto 'KnotVector' para cada dirección (u y v).

Esta clase contiene los métodos para calcular las funciones base de una NURBS así como sus derivadas. Las funciones base se utilizan, entre otras cosas, para calcular los puntos en coordenadas cartesianas de una curva o superficie NURBS.

2.4.3 Clase NurbsCurve

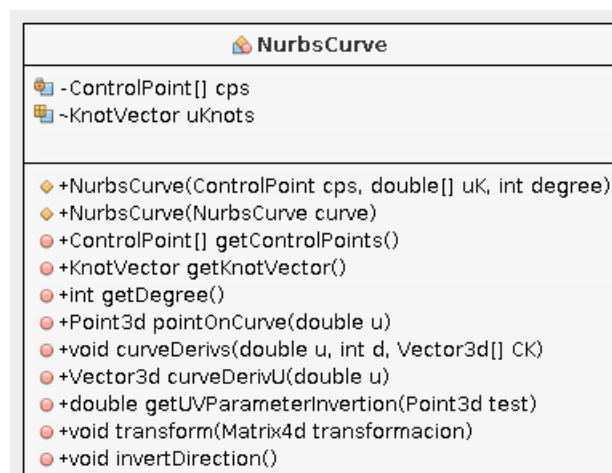


Figura 2.16: Clase 'NurbsCurve'

La clase 'NurbsCurve' representa una curva NURBS. Como se ha descrito anteriormente, una curva NURBS está formada por un vector de puntos de control y un vector de knots.

El método *pointOnCurve* permite obtener los puntos por donde pasa la curva en coordenadas cartesianas a partir de un valor del parámetro u definido en el intervalo $[0, 1]$. Esta función se utiliza posteriormente para generar el renderizado de la curva en Java 3D.

El método *curveDerivs* calcula las derivadas de una curva NURBS. La primera derivada indica la dirección de la curva para el valor del parámetro u .

El método *getUVParameterInversion* realiza la operación contraria a *pointOnCurve*. Dado el punto 'test' en coordenadas cartesianas obtiene el valor del parámetro u . Si el punto no pertenece a la curva devuelve el valor del punto mas cercano a este. Desgraciadamente, no se puede obtener el valor del parámetro u de forma analítica, el algoritmo implementado se basa en el método del gradiente conjugado que se explicará más adelante.

El método *transform* aplica un matriz de transformación a la curva. Para aplicar una matriz de transformación a una curva NURBS solo hay que transformar sus puntos de control. El proceso se describe detalladamente en la sección de 'Transformaciones Geométricas'.

El método *invertDirection* invierte la dirección de la curva NURBS. Para ello se invierte el vector de puntos de control y los knots.

2.4.4 Clase NurbsSurface

NurbsSurface	
#KnotVector	uKnots
#KnotVector	vKnots
#ControlPoint[][]	cpnet
#TrimmsUV	parametricTrimms
#Trimms3D	cartesianTrimms
+boolean	closedU
+boolean	closedV
+int	borde_deg
◆	+NurbsSurface(ControlPoint[][] cps, double[] uK, double[] vK, int p, int q)
◆	+NurbsSurface(NurbsSurface surface)
🔧	-void validate()
●	+ControlPoint[][] getControlPoints()
●	+Point3d pointOnSurface(double u, double v)
●	+Vector3d[] surfaceDerivs(double u, double v, int d)
●	+Vector3d surfaceDerivU(double u, double v)
●	+Vector3d surfaceDerivW(double u, double v)
●	+Vector3d getNormalVector(double u, double v)
●	+UVCoord2d getUVParameterInversion(Point3d test)
●	+KnotVector getUKnotVector()
●	+KnotVector getVKnotVector()
●	+TrimmsUV getParametricTrimms()
●	+Trimms3D getCartesianTrimms()
●	+void transform(Matrix4d transformacion)
●	+boolean isTrimmed()
●	+void invertNormals()

Figura 2.17: Clase 'NurbsSurface'

La clase 'NurbsSurface' representa una superficie NURBS. Se puede ver la clase como una extensión de 'NurbsCurve' en las direcciones paramétricas u y v . Una superficie NURBS está formada por una matriz de puntos de control y un vector de knots para cada dirección. Los objetos 'TrimmsUV' y 'Trimms3D' almacenan las curvas recortadoras para representar superficies trimadas. Las variables 'closedU', 'closedV' y 'borde_deg' son propiedades de la superficie NURBS e indican si es cerrada o degenerada en algún borde.

Una superficie NURBS es cerrada, por ejemplo en la dirección u , si el punto en cartesianas para $u = 0$ coincide con el de $u = 1$ para cualquier valor de v ; es equivalente a que la primera y la última fila de la matriz de puntos de control sean iguales.

Una superficie NURBS es degenerada en un borde si todos los puntos de ese borde son el mismo. Como puede haber varios bordes degenerados la variable 'borde_deg' es una máscara de 4 bits donde cada bit indica si un borde es degenerado o no.

El método *validate* se encarga de comprobar la ecuación (2.4) y calcular el valor de 'closedU', 'closedV' y 'borde_deg'.

El método *pointOnSurface* permite obtener los puntos de la superficie en coordenadas cartesianas a partir de los valores paramétricos u y v definidos en el intervalo $[0, 1]$. De la misma manera, esta función se utiliza para generar el renderizado de la superficie en Java 3D.

El método *surfaceDerivs* calcula las derivadas de una superficie NURBS. La primera derivada indica la dirección de la normal de la superficie en el punto u, v .

El método *getUVParameterInversion* obtiene los valores de u y v para un punto 'test' en coordenadas cartesianas.

El método *transform* aplica una matriz de transformación a la superficie. De la misma manera que en la curva, para aplicar una matriz de transformación a una superficie NURBS solo hay que transformar sus puntos de control.

El método *invertNormals* invierte las normales de la superficie. Para invertir las normales hay que cambiar la dirección u por v en la matriz de puntos de control y en los vectores de knots.

Los métodos *getParametricTrimms* y *getCartesianTrimms* sirven para obtener las curvas recortadoras de la superficie. Su funcionamiento se muestra a continuación.

2.4.5 Clases Trimms3D y TrimmsUV

Las clases 'Trimms3D' y 'TrimmsUV' contienen las curvas recortadoras de una superficie NURBS. Cada curva recortadora se almacena en coordenadas cartesianas en 'Trimms3D' y en coordenadas paramétricas en 'TrimmsUV'. Las curvas en coordenadas paramétricas se utilizan para renderizar la superficie.

En la siguiente figura se muestra la estructura de ambas clases, como se puede observar, las dos tienen un diseño similar.

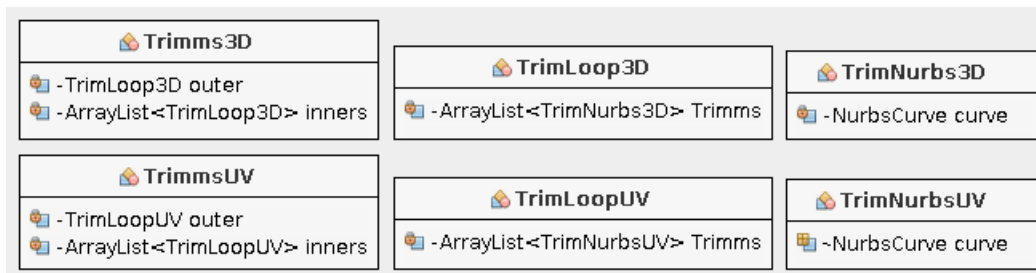


Figura 2.18: Clases 'Trimms3D' y 'TrimmsUV'

La clase 'Trimms3D' contiene un objeto de la clase 'TrimLoop3D' llamado *outer* que contiene las curvas que delimitan el borde exterior de la superficie. A su vez, puede contener uno o varios objetos 'TrimLoop3D' en el array llamado *inners* que formarán los agujeros de la superficie.

La clase 'TrimLoop3D' representa un bucle (loop) y puede contener una o varias curvas de tipo 'TrimNurbs3D', como se vio en la sección 2.3.2.1. Cada 'TrimNurbs3D' representa una curva de tipo 'NurbsCurve'.

2.5 Algoritmos de Renderizado de Curvas y Superficies NURBS

El renderizado de curvas y superficies NURBS consiste en generar, a partir de los datos de la NURBS, las estructuras necesarias para que puedan ser visualizadas por el API de Java3D.

En primer lugar, se describirá el algoritmo para el renderizado de curvas, que posteriormente se va a utilizar para visualizar superficies mediante líneas. En segundo lugar, se mostrará el algoritmo para renderizar una superficie NURBS mediante polígonos, en nuestro caso triángulos, a este proceso también se le conoce como teselado.

Los algoritmos de renderizado tienen que ser muy rápidos y generar el menor número de puntos posible, ya que el rendimiento de la aplicación y el número de objetos que se pueden dibujar simultáneamente dependerá enormemente de estos dos factores. Siempre existirá un compromiso entre la calidad de la visualización y el rendimiento, que hay que ajustar.

Los algoritmos que se han desarrollado son adaptativos, es decir, el número de puntos que se genera depende de la complejidad de la NURBS y del nivel de detalle.

En nuestra librería 'NurbsLib' los algoritmos de renderizado se encuentran dentro del paquete 'Evaluators' del diagrama mostrado en la figura 2.13.

2.5.1 Java 3D Geometry

En la sección 2.2.3 se puede ver que la clase 'Shape3D' es la encargada de visualizar la geometría. Dentro de esta se encuentra la clase 'Geometry' que es la encargada de definir la forma que tiene el objeto. La clase 'Geometry' es una clase abstracta y las clases que extienden de ella representan las distintas formas que tiene Java 3D para dibujar objetos.

Las clases que interesan extienden de 'GeometryArray' que representa una colección de vértices, colores, normales, texturas y atributos que permiten definir puntos, líneas y polígonos de diferentes maneras.

- **PointArray:** Dibuja un punto individual por cada punto en el array de vértices.
- **LineArray:** Dibuja una línea entre cada vértice del array.
- **TriangleArray:** Dibuja el array de vértices como triángulos individuales. El array se divide en grupos de 3 para cada triángulo.
- **QuadArray:** Dibuja el array de vértices como cuadrángulos individuales. Cada cuadrángulo se forma con grupos de 4 vértices.

Las tres primeras clases son las más rápidas para dibujar geometría con Java 3D y son las que se usan en el motor gráfico.

Las otras clases que extienden de 'GeometryArray' son 'GeometryStripArray' que permite dividir el conjunto de vértices en distintas secciones y 'IndexedGeometryArray' que permite definir un vector de índices que indican como se conectan los vértices. Dentro de estas clases se encuentran:

- **LineStripArray:** Funciona igual que 'LineArray' pero dibuja una línea distinta por cada número de vértices que se indica en el array de *strips*.
- **TriangleStripArray:** Permite definir un conjunto de triángulos donde cada nuevo vértice representa un nuevo triángulo que se conecta al anterior. El número de vértices en cada posición del array de *strips* indica conjuntos de triángulos separados.
- **TriangleFanArray:** Cada vértice nuevo crea un triángulo que se conecta con el vértice anterior y con el primer vértice.
- **IndexedPointArray:** Dibuja un punto por cada vértice que aparece referenciado en el vector de índices.
- **IndexedLineArray:** Se dibuja un segmento por cada par de vértices referenciados por el vector de índices.
- **IndexedTriangleArray:** Se dibuja un triángulo por cada 3 vértices que aparecen referenciados en los índices.
- **IndexedQuadArray:** Se dibuja un cuadrángulo por cada 4 vértices que aparecen referenciados en los índices.

2.5.2 Renderizado de Curvas

El renderizado de una curva NURBS se basa en extraer un array de vértices que se correspondan con los puntos de paso de la curva en el espacio real para, posteriormente, construir un objeto de tipo 'VertexArray' que permita dibujarla en Java 3D.

El método más sencillo para dibujar una curva NURBS consiste en utilizar la función *pointOnCurve* de la clase 'NurbsCurve' (2.4.4), tomando distintos valores de u en todo el espacio paramétrico⁵. Esto permite dibujar la curva con mayor o menor nivel de detalle dependiendo del número de divisiones que se realicen.

Algoritmo 2.1 Renderizado de curvas estático

```

ArrayList<Point3d> renderSimple(NurbsCurve curve, int div) {
    double[] knots = curve.getKnots();
    double u = knots[0];
    double max = knots[knots.length - 1];
    double step = (max - u) / div;
    ArrayList<Point3d> points = new ArrayList();
    for (int i = 0; i <= div; i++) {
        points.add(curve.pointOnCurve(u));
        u += step;
    }
    return points;
}

```

El algoritmo que se ha desarrollado para visualizar curvas NURBS es adaptativo, es decir, el número de puntos que se calculan varía según la complejidad de la curva. De esta forma, no hay que preocuparse por el número de divisiones para que la curva se visualice correctamente utilizando el menor número de puntos posible.

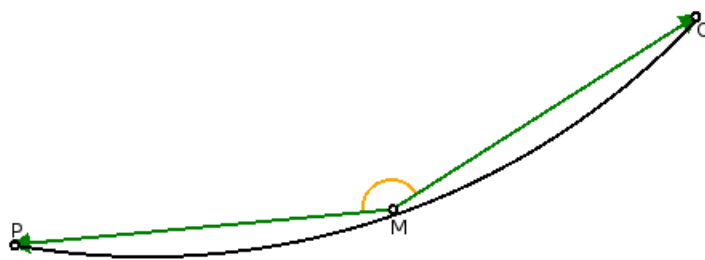


Figura 2.19: Renderizado de curvas adaptativo

El algoritmo es recursivo y funciona de la siguiente manera:

- Dados dos puntos P y Q que pertenecen a la curva, se calcula el punto medio entre ellos llamado M .
- Si los puntos P , Q y M son el mismo la función termina.

⁵El espacio paramétrico de una curva NURBS se encuentra definido en el intervalo entre el primer y el último valor de su vector de knots.

- Se comprueba si el ángulo que forman los vectores $\vec{v}_1 = MP$ y $\vec{v}_2 = MQ$ es mayor que $\pi - \varepsilon$ donde ε es el ángulo máximo entre dos puntos en radianes.
- Si se cumple, se buscan dos puntos aleatorios entre PM y entre MQ llamados M_1 y M_2 . Se repite el paso anterior con los puntos PMM_1 y MQM_2 y si se cumple se añade el punto M al vector de puntos y la función termina.
- Se repite todo el proceso recursivamente por la izquierda tomando los puntos PM y por la derecha tomando los puntos MQ .

Algoritmo 2.2 Renderizado de curvas adaptativo

```

public ArrayList<Point3d> renderAdaptive(NurbsCurve curve) {
    ArrayList<Point3d> points = new ArrayList();
    double uKnots[] = curve.getKnots();
    double uIni = uKnots[0];
    double uFin = uKnots[uKnots.length - 1];
    Point3d pIni = curve.pointOnCurve(uIni);
    Point3d pFin = curve.pointOnCurve(uFin);
    points.add(pIni);
    sample(curve, points, uIni, uFin);
    points.add(pFin);
    return points;
}

private void sample(NurbsCurve curve, ArrayList<Point3d> points, double p, double q) {
    Random r = new Random();
    double m = (p+q)*0.5;
    Point3d pp, pq, pm;
    pp = curve.pointOnCurve(p);
    pq = curve.pointOnCurve(q);
    pm = curve.pointOnCurve(m);
    if (equalPoints(pp, pq) && equalPoints(pp, pm)) {
        return;
    }
    if (checkAngle(pp, pq, pm)) {
        double m1 = ((m-p)*r.nextDouble()) + p;
        double m2 = ((q-m)*r.nextDouble()) + m;
        Point3d pm1, pm2;
        pm1 = curve.pointOnCurve(m1);
        pm2 = curve.pointOnCurve(m2);
        if (checkAngle(pp, pm, pm1) && checkAngle(pm, pq, pm2)) {
            points.add(pm);
            return;
        }
    }
    sample(curve, points, p, m);
    sample(curve, points, m, q);
}

private boolean checkAngle(Point3d pp, Point3d pq, Point3d pm) {
    Vector3d vPM = new Vector3d(pp.x-pm.x, pp.y-pm.y, pp.z-pm.z);
    Vector3d vQM = new Vector3d(pq.x-pm.x, pq.y-pm.y, pq.z-pm.z);

    return vPM.angle(vQM) > Math.PI-angleTolerance;
}

```

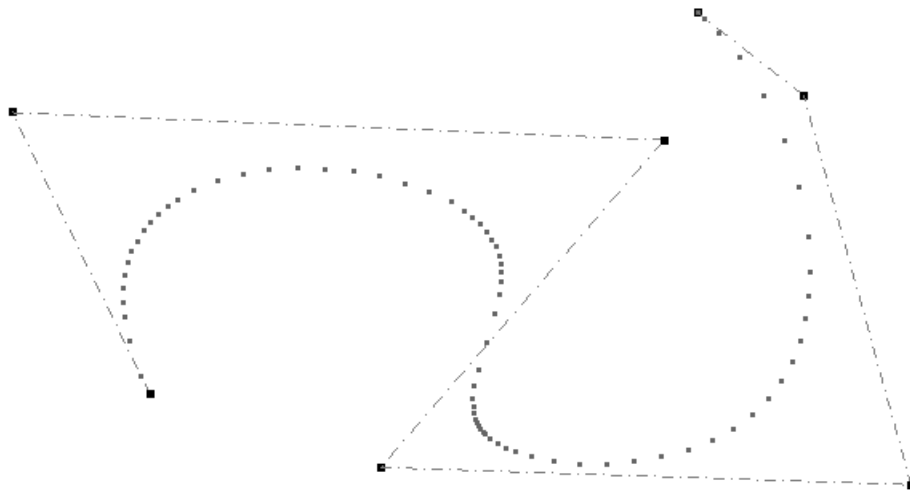
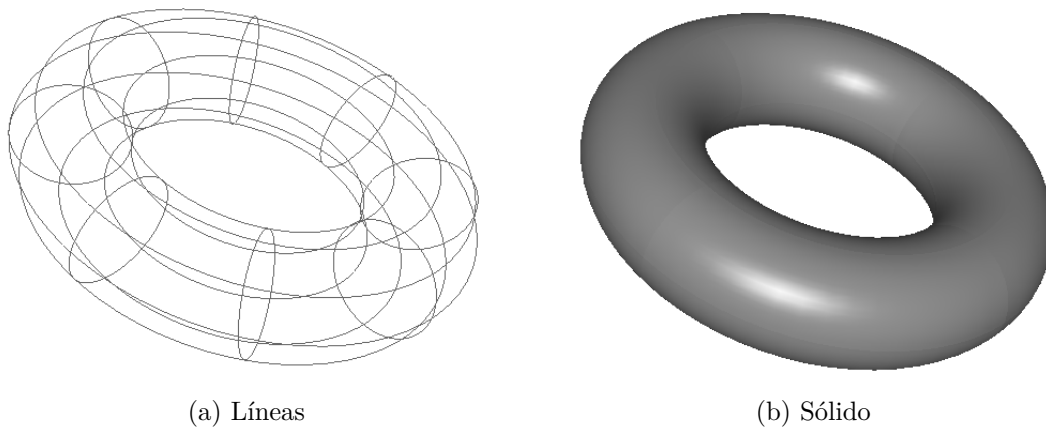


Figura 2.20: Resultados del algoritmo adaptativo

2.5.3 Renderizado de Superficies

Para visualizar una superficie NURBS se han desarrollado dos métodos. El primero permite visualizar una superficie mediante líneas y para ello se extraen los bordes de la superficie como curvas NURBS y se renderizan utilizando el algoritmo anterior. El segundo método consiste en realizar un mallado de la superficie para visualizarla como un objeto sólido.



(a) Líneas

(b) Sólido

Figura 2.21: Renderizado de superficies

2.5.3.1 Renderizado de superficies mediante líneas

El primer paso para renderizar una superficie mediante líneas es extraer las curvas del borde exterior de la superficie NURBS. Para ello hay que diferenciar dos casos:

Si la superficie es trimada el borde serán las curvas que contiene el objeto *outer* de la clase 'Trimms3D'.

Si la superficie no es trimada hay que extraer las cuatro curvas del borde a partir de la matriz de puntos de control y de los nodos de la superficie NURBS.

Para obtener los puntos de control de las curvas hay que recorrer los bordes de la matriz de puntos de control en sentido antihorario, para que las curvas tengan la dirección correcta. También, para cada curva se extraen los nodos de la superficie de una forma determinada, según la dirección de esta. El procedimiento es el siguiente:

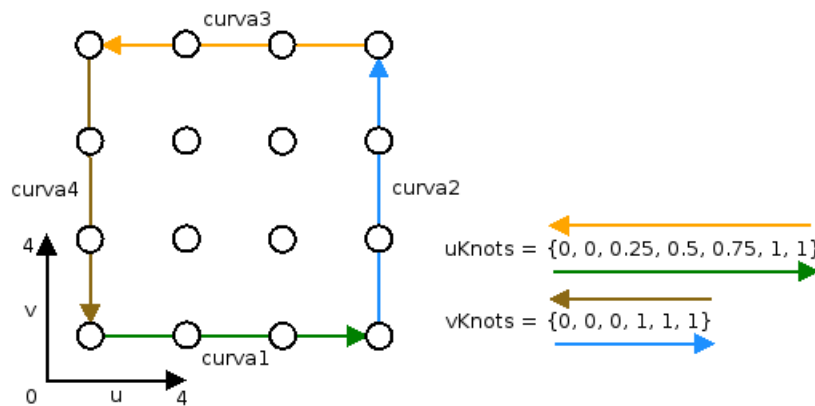


Figura 2.22: Curvas de borde de una superficie NURBS

Cuando los nodos se cogen de manera descendente, como en *curva3* y *curva4*, se resta el valor de cada nodo al valor máximo de ese vector de nodos, para que queden ordenados de menor a mayor.

También hay que tener en cuenta que un borde de una superficie NURBS puede ser degenerado, en ese caso coinciden todos los puntos de control en el mismo punto, por lo que no tiene sentido devolver esa curva.

El siguiente paso es comprobar si la superficie contiene agujeros, para ello se comprueba si el objeto *inners* de la clase 'Trimms3D' no está vacío, en este caso, se extraen las curvas de los agujeros y se dibujan.

2.5.3.2 Renderizado de superficies mediante polígonos

Para renderizar una superficie NURBS en forma de sólido es necesario generar una malla, en nuestro caso de triángulos.

Si la superficie NURBS es trimada, el algoritmo de renderizado se encargará de generar los triángulos en las partes visibles de la superficie, descritas en la Sección 2.3.2.1.

En la siguiente imagen se puede ver cómo está formada la malla de la superficie de la figura 2.21 antes de que se le aplique el material y la iluminación.

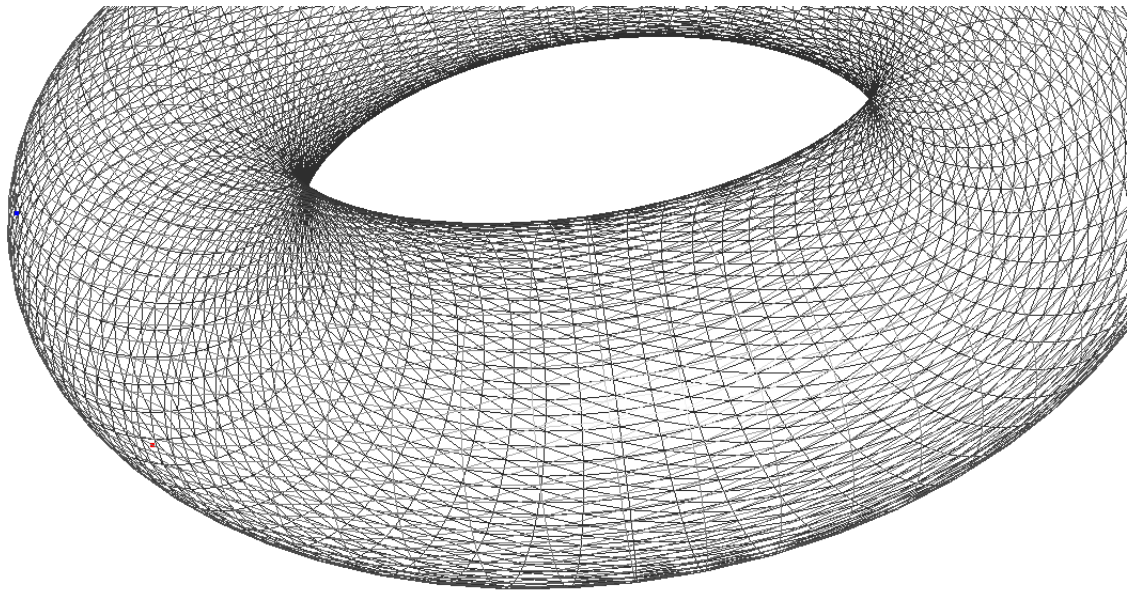


Figura 2.23: Malla de polígonos de una superficie NURBS

Para generar la malla, se divide el espacio paramétrico u, v de la superficie NURBS según un nivel de detalle.

El primer paso es comprobar si un punto u, v pertenece a la zona válida de la superficie, para ello se utilizan las curvas en coordenadas paramétricas de la clase 'TrimmsUV'. La idea es crear un polígono con cada bucle (loop) de la superficie y comprobar si el punto u, v está contenido dentro del polígono del borde exterior, y fuera de cada agujero. Para ello se utiliza la clase 'Polygon' de Java. El algoritmo devuelve un booleano que indica si el punto es válido.

Algoritmo 2.3 Comprueba si un punto está en la zona válida de la superficie NURBS

```

public boolean isVertexInBoundary(NurbsSurface surface, double u, double v) {
    boolean inOuter=true;
    boolean inInner=false;

    if (outerPolygon!=null) {
        inOuter = outerPolygon.contains(u, v);
    }
    for(Polygon p : innerPolygons) {
        if(p.contains(u, v)) {
            inInner=true;
            break;
        }
    }
    return inOuter&&!inInner;
}

```

El siguiente paso es construir la matriz de coordenadas u, v de la superficie, marcando los puntos válidos. Cuando se encuentra un punto válido se añade a un vector, y se guarda su índice en una matriz, si el punto no es válido la matriz almacena el valor -1. A su vez, se guardan las coordenadas paramétricas de todos los puntos analizados en otra matriz.

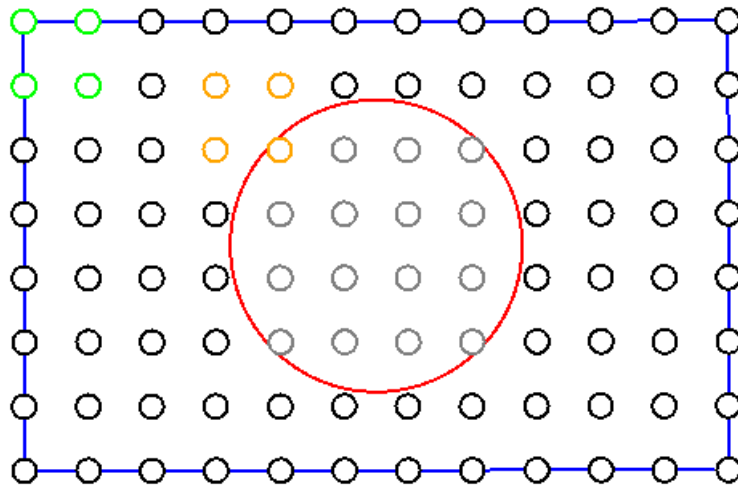


Figura 2.24: Matriz de renderizado de una superficie NURBS

Con la matriz de renderizado creada, se puede empezar a construir los parches de la malla, en nuestro caso el proceso construye parches cuadrados, que luego se dividirán en dos para formar triángulos.

En la figura aparecen los puntos marcados en negro, que pertenecen a la zona válida de la superficie. A su vez, los puntos marcados en gris pertenecen a la zona no válida, ya que están dentro de un bucle que forma un agujero (marcado en rojo).

Se recorre la matriz por filas i y columnas j , en cada paso se cogen los índices que se almacenaron anteriormente en $M[i][j]$, $M[i + 1][j]$, $M[i + 1][j + 1]$ y $M[i][j + 1]$. En este momento pueden ocurrir tres casos que se indican con distintos colores en la figura anterior.

1. El color verde indica el caso en que los cuatro índices son válidos, es decir, ninguno toma el valor -1 en la matriz. En este caso se añade un nuevo parche⁶.
2. El color gris indica que los cuatro puntos están en la zona no válida de la superficie NURBS. En este caso no se hace nada.
3. El color naranja indica que uno o más puntos del parche están en la zona no válida. En este caso para cada punto de la zona no válida hay que buscar el punto de intersección con la curva *trimmed*, añadirlo al vector de puntos y crear un nuevo índice para construir el parche.

Para buscar la intersección con la curva *trimmed*, hay que muestrear la curva en coordenadas paramétricas usando el algoritmo 2.2 y calcular la proyección del punto u, v en los segmentos formados por cada dos puntos de muestreo. El punto de intersección será el punto más cercano al punto u, v .

⁶La matriz tiene almacenados los valores de los índices del vector de puntos. Para añadir un parche se añaden esos cuatro valores a un vector de índices.

El siguiente algoritmo calcula la proyección del punto P_3 , en el segmento formado por P_1 y P_2 .

Algoritmo 2.4 Proyección de un punto en un segmento

```

public Point2d getProjectionOnSegment(Point2d p1, Point2d p2, Point2d p3) {

    double xDelta = p2.x - p1.x;
    double yDelta = p2.y - p1.y;

    double u = ((p3.x - p1.x)*xDelta + (p3.y - p1.y)*yDelta)
              / (xDelta*xDelta + yDelta*yDelta);
    Point2d closestPoint;
    if (u < 0) {
        closestPoint = p1;
    } else if (u > 1) {
        closestPoint = p2;
    } else {
        closestPoint = new Point2d(p1.x + u*xDelta, p1.y + u*yDelta);
    }

    return closestPoint;
}
  
```

Por último, hay que calcular las normales de cada parche, esto es importante para añadir iluminación en Java3D. Para ello hay que crear un vector de normales del mismo tamaño que el vector de índices. El vector contiene objetos de la clase 'Vector3d', cada uno de ellos indica la dirección de la normal para el índice correspondiente.

El proceso es el siguiente:

1. Se cogen los índices de un parche, es decir, se recorre el vector de índices cogiendo elementos de cuatro en cuatro.
2. Usando esos índices se extraen los puntos P_1, P_2, P_3 y P_4 en 3D del parche accediendo al vector de puntos.
3. La normal en P_1 es el producto vectorial de los vectores $\vec{v}_1 = P_2 - P_1$ y $\vec{v}_2 = P_4 - P_1$. Se repite el proceso para el resto de puntos.
4. En vez de almacenar una normal distinta para cada punto, se calcula la normal del parche entero sumando los vectores normales de los cuatro vértices. La normal se almacena para los cuatro índices del parche.

Finalmente, se muestra el código del algoritmo de renderizado de superficies NURBS para aclarar el proceso. El algoritmo se ha dividido en dos partes: La primera parte es el proceso de creación de la matriz UV que se muestra en la figura 2.24. La segunda parte es el proceso de mallado a partir de esa matriz.

Los argumentos de entrada del algoritmo son la superficie NURBS y las variables $segU$ y $segV$ que indican el número de divisiones del espacio paramétrico en u y v . Estas variables se usan para ajustar el nivel de detalle del renderizado.

Algoritmo 2.5 Renderizado de superficies (1)

```

//Matriz para almacenar los indices del vector de puntos en UV
int UVmatrix [][] = new int [segU][segV];
//Matriz para almacenar los valores UV para las intersecciones
double UV[][][] = new double [segU][segV][2];

for (int i = 0; i < segU; i++) {
    v = vKnots[0];
    for (int j = 0; j < segV; j++) {
        boolean trimmed = !isVertexInBoundary(surface, u, v);
        if (trimmed) {
            UVmatrix[i][j] = -1;
        } else {
            vertexList.add(surface.pointOnSurface(u, v));
            UVmatrix[i][j] = vertexList.size() - 1;
        }
        UV[i][j][0] = u;
        UV[i][j][1] = v;
        v += vStep;
    }
    u += uStep;
}

```

Algoritmo 2.6 Renderizado de superficies (2)

```

boolean trimmed, trimmed1, trimmed2, trimmed3, trimmed4;
for (int i=0; i<segU-1; i++) {
    for (int j=0; j<segV-1; j++) {
        trimmed1 = UVmatrix[i][j] == -1; //t1
        trimmed2 = UVmatrix[i+1][j] == -1; //t2
        trimmed3 = UVmatrix[i+1][j+1] == -1; //t3
        trimmed4 = UVmatrix[i][j+1] == -1; //t4
        trimmed = trimmed1 | trimmed2 | trimmed3 | trimmed4;
        if (!trimmed) {
            //Hay parche
            indexes.add(UVmatrix[i][j]);
            indexes.add(UVmatrix[i+1][j]);
            indexes.add(UVmatrix[i+1][j+1]);
            indexes.add(UVmatrix[i][j+1]);
        } else {
            if (trimmed1 && trimmed2 && trimmed3 && trimmed4) {
                continue;
            }
            Point2d i1=null, i2=null, i3=null, i4=null;
            if (trimmed1) {
                i1 = closestIntersection(surface, UV[i][j][0], UV[i][j][1]);
                if (i1==null) throw new Exception();
            }
            ... Se repite el proceso con trimmed2, trimmed3, trimmed4 ...
            if (trimmed1) {
                vertexList.add(surface.pointOnSurface(i1.x, i1.y));
                indexes.add(vertexList.size() - 1);
            } else {
                indexes.add(UVmatrix[i][j]);
            }
            ...
        }
    }
}

```

En la siguiente figura se muestra el resultado del Algoritmo 2.6 sobre una superficie rectangular con un agujero utilizando distintos niveles de detalle.

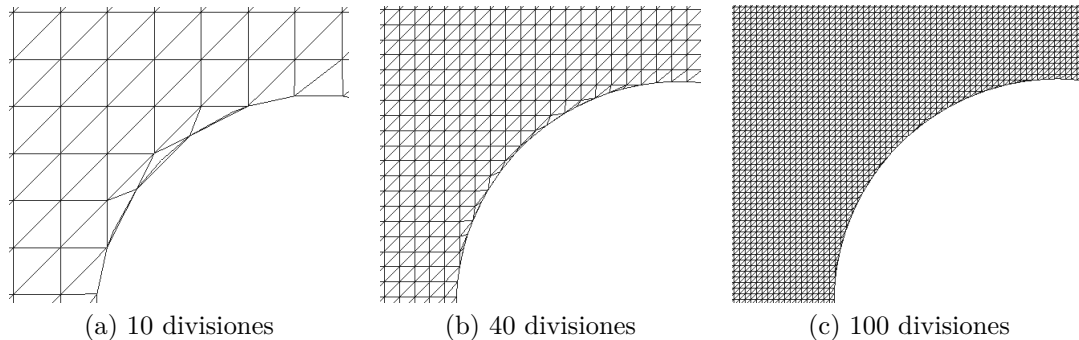


Figura 2.25: Resultados del algoritmo de renderizado de superficies

2.6 Transformaciones Geométricas

Una transformación geométrica es el resultado de un cambio (de posición, de tamaño, de forma, ...) producido en una figura dada F cuando pasa a ser F' . Las correspondencias entre los elementos de F y de F' originan los diferentes tipos de transformaciones. La relación que exista entre los elementos origen y transformados debe de ser biunívoca.

Una propiedad muy interesante de las curvas y superficies NURBS es que se pueden transformar linealmente mediante la transformación (lineal) de sus puntos de control.

En la mayoría de sistemas de visualización 3D las transformaciones geométricas se representan mediante matrices [14, 15]. Esto es posible gracias a las coordenadas homogéneas.

Hasta ahora, se han considerado las coordenadas 3D como tripletas (x, y, z) . Las coordenadas homogéneas introducen el término w para formar vectores (x, y, z, w) :

- Si $w = 0$ el vector representa una dirección.
- Si $w = 1$ el vector representa una posición en el espacio.

¿Qué diferencia hay? Bueno, para una rotación no hay diferencia. Cuando se rota un punto o una dirección, se obtiene el mismo resultado. Sin embargo para una traslación (mover un punto en una dirección), las cosas son diferentes. Una traslación de una dirección con $w = 0$ no hace nada, ya que no tiene sentido trasladar una dirección. Las coordenadas homogéneas usan una única fórmula matemática para lidiar con estos dos casos.

A lo largo de esta sección se usará la formulación matricial para representar matemáticamente las operaciones de transformaciones geométricas. De esta forma se reducen los cálculos, porque es posible aplicar una matriz a una posición o dirección para obtener su posición transformada.

2.6.1 Matrices de transformación

En términos simples, una matriz es un arreglo de números con un número predefinido de filas y columnas. Por ejemplo, una matriz de 2x3 se ve así:

$$\begin{bmatrix} 2 & 5 & 7 \\ 4 & 3 & 1 \end{bmatrix}$$

En computación gráfica se usan matrices de 4x4. Estas matrices permitirán transformar los vectores (x, y, z, w) multiplicando el vértice con la matriz:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} ax + by + cz + dw \\ ex + fy + gz + hw \\ ix + jy + kz + lw \\ mx + ny + oz + pw \end{bmatrix}$$

En Java3D se utiliza la clase 'Matrix4d' para representar una matriz de transformación, para realizar la operación anterior se utiliza el método *transform* de esta clase.

Conviene conocer la matriz identidad, que no hace nada. Normalmente, cuando se crea una matriz para realizar una transformación geométrica, se inicializa a la matriz identidad y posteriormente se asignan los valores de la matriz.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.6.2 Translación

La operación de translación consiste en modificar la posición de un objeto, de forma que su tamaño y orientación no se vean alterados. Al aplicar una translación sobre un objeto, este se moverá de una posición a otra con una trayectoria en línea recta, tal y como se muestra en la figura [2.26](#).

Matemáticamente la translación se indica como la suma de cada coordenada (x, y, z) con cada coordenada del vector desplazamiento (d_x, d_y, d_z) .

$$\begin{aligned} x' &= x + d_x \\ y' &= y + d_y \\ z' &= z + d_z \end{aligned}$$

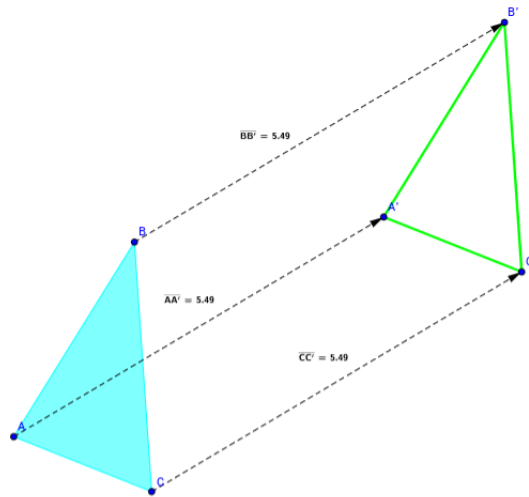


Figura 2.26: Transformaciones geométricas: Translación

La operación de translación se representa mediante la siguiente matriz de transformación:

$$T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Para ver un ejemplo de como funciona, si se quiere trasladar el punto $(2, 5, 7)$ 10 unidades en la dirección X. Se tiene que el vector de translación es $(10, 0, 0)$. Si se construye la matriz de translación con ese vector y se multiplica por el punto en coordenadas homogéneas, se obtiene como resultado el punto $(12, 5, 7)$.

$$\begin{bmatrix} 1 & 0 & 0 & 10 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 2 \\ 5 \\ 7 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 + 0 + 0 + 10 \\ 0 + 5 + 0 + 0 \\ 0 + 0 + 7 + 0 \\ 0 + 0 + 0 + 1 \end{bmatrix} = \begin{bmatrix} 12 \\ 5 \\ 7 \\ 1 \end{bmatrix}$$

2.6.3 Escalado

La operación de escalado consiste en aumentar o reducir el tamaño de un objeto multiplicando las coordenadas de cada uno de sus puntos por un factor. El factor de escalado debe ser positivo, si está entre 0 y 1 se reduce el tamaño del objeto, si es mayor que 1 se aumenta.

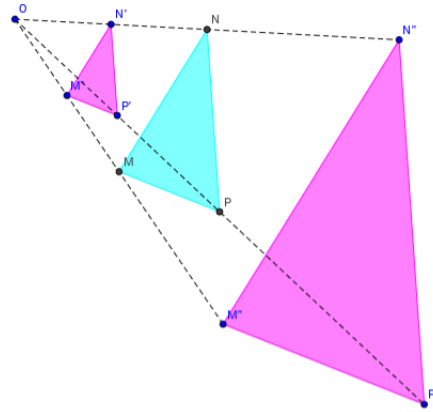


Figura 2.27: Transformaciones geométricas: Escalado

La operación de escalado se representa mediante la siguiente matriz de transformación:

$$S(f_x, f_y, f_z) = \begin{bmatrix} f_x & 0 & 0 & 0 \\ 0 & f_y & 0 & 0 \\ 0 & 0 & f_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Si $f_x = f_y = f_z$ se dice que el escalado es uniforme. Cuando estos factores son distintos se conoce como escalado no uniforme, que puede ser en una o dos dimensiones. Como se puede observar, cada factor multiplica una coordenada (x, y, z) del punto.

2.6.4 Rotación

La operación de rotación consiste en cambiar la orientación de un objeto con respecto a un eje llamado eje de rotación. En rotación, el ángulo a través del cual la figura se rota es llamado el ángulo de rotación.

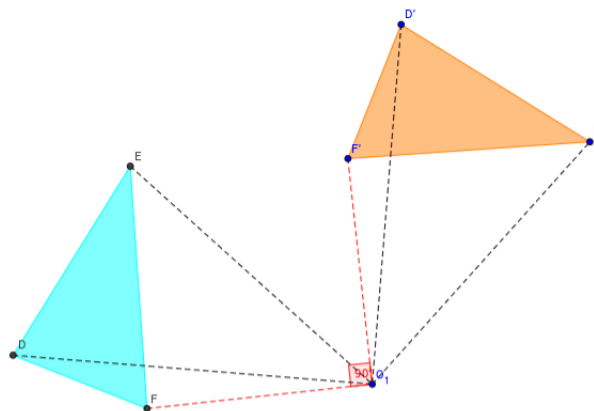


Figura 2.28: Transformaciones geométricas: Rotación

Si la rotación se realiza sobre uno de los ejes del sistema de coordenadas XYZ, la matriz de rotación es muy sencilla y se representa de la siguiente manera:

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & \sin \alpha & 0 \\ 0 & -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\alpha) = \begin{bmatrix} \cos \alpha & 0 & -\sin \alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin \alpha & 0 & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z(\alpha) = \begin{bmatrix} \cos \alpha & \sin \alpha & 0 & 0 \\ -\sin \alpha & \cos \alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

La operación de rotación también se puede realizar sobre un eje arbitrario y el ángulo de rotación α . Cuando el eje de rotación no coincide con ningún eje de coordenadas, se puede definir una matriz genérica de rotación compuesta por una combinación de traslaciones y rotaciones sobre los ejes. A esta operación se la conoce como rotación general.

Sean $P = (x_1, y_1, z_1)$ y $Q = (x_2, y_2, z_2)$ dos puntos que definen el eje de rotación, el vector de dirección \vec{v} del eje de rotación es el siguiente:

$$\vec{v}(x, y, z) = (x_2 - x_1, y_2 - y_1, z_2 - z_1)$$

Si \vec{v} se encuentra sobre el eje X, es decir que $y = 0$ y $z = 0$, la rotación se realiza según la ecuación 2.15.

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & y_1 \\ 0 & 0 & 1 & z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha & 0 \\ 0 & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R(\alpha) = T \cdot R_x(\alpha) \cdot T^{-1} \tag{2.15}$$

En el caso contrario de que el vector \vec{v} no se encuentra sobre el eje X, la rotación se realiza según la ecuación 2.16.

$$T = \begin{bmatrix} 1 & 0 & 0 & x_2 \\ 0 & 1 & 0 & y_2 \\ 0 & 0 & 1 & z_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad d = \sqrt{y^2 + z^2}$$

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{z}{d} & \frac{y}{d} & 0 \\ 0 & \frac{-y}{d} & \frac{z}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y = \begin{bmatrix} \frac{d}{|\vec{v}|} & 0 & \frac{x}{|\vec{v}|} & 0 \\ 0 & 1 & 0 & 0 \\ \frac{-x}{|\vec{v}|} & 0 & \frac{d}{|\vec{v}|} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z(\alpha) = \begin{bmatrix} \cos \alpha & \sin \alpha & 0 & 0 \\ -\sin \alpha & \cos \alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R(\alpha) = (T \cdot R_x \cdot R_y \cdot R_z(\alpha) \cdot R_y^{-1} \cdot R_x^{-1} \cdot T^{-1})^{-1} \quad (2.16)$$

2.6.5 Simetría

La operación de simetría es una transformación respecto de un plano, llamado plano de simetría, en la que a cada punto de una figura se asocia a otro punto llamado imagen, que cumple las siguientes condiciones:

- La distancia de un punto y su imagen al plano de simetría, es la misma.
- El segmento que une un punto con su imagen, es perpendicular al plano de simetría.

En nuestro caso el plano de simetría se define mediante un punto y un vector normal al plano de simetría en ese punto.

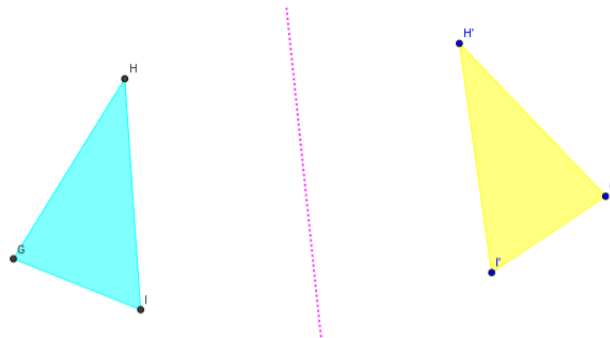


Figura 2.29: Transformaciones geométricas: Simetría

Si la simetría se realiza sobre uno de los ejes del sistema de coordenadas XYZ, la matriz de rotación se representa de la siguiente manera:

$$S_x = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$S_y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$S_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Cuando el plano de simetría es arbitrario la matriz de transformación se define de la siguiente manera:

Sea $P = (x_1, y_1, z_1)$ un punto perteneciente al plano de simetría y $Q = (x_2, y_2, z_2)$ otro punto que indica la dirección del vector normal del plano a partir de P . Ambos puntos definen el vector normal \vec{n} del plano de simetría:

$$\vec{n}(x, y, z) = (x_2 - x_1, y_2 - y_1, z_2 - z_1)$$

Dado el vector normal \vec{n} del plano de simetría la matriz de transformación es la siguiente:

$$S(\vec{n}) = \begin{bmatrix} 1 - 2 \cdot x^2 & -2 \cdot x \cdot y & -2 \cdot x \cdot z & 0 \\ -2 \cdot x \cdot y & 1 - 2 \cdot y^2 & -2 \cdot y \cdot z & 0 \\ -2 \cdot x \cdot z & -2 \cdot y \cdot z & 1 - 2 \cdot z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (2.17)$$

2.6.6 Transformaciones en curvas y superficies NURBS

Para aplicar una matriz de transformación a una curva o superficie NURBS solo hay que multiplicar sus puntos de control por la matriz de transformación, como se muestra en la sección 2.6.1.

En Java3D se utiliza la clase 'Matrix4d' para representar una matriz de transformación, el algoritmo 2.7 describe el proceso para aplicar una matriz de transformación a una

superficie NURBS. Si la superficie es trimada, hay que aplicar la matriz de transformación de la misma manera a sus curvas en coordenadas cartesianas.

Algoritmo 2.7 Transformación de una superficie NURBS

```

public void transform(Matrix4d transformacion) {
    ControlPoint [][] cpnet = this.getControlPoints();
    for (int i=0; i<cpnet.length; i++) {
        for (int j=0; j<cpnet[0].length; j++) {
            ControlPoint cp = cpnet[i][j];
            Point3d p = new Point3d(cp.x, cp.y, cp.z);
            transformacion.transform(p);
            cp.x = p.x;
            cp.y = p.y;
            cp.z = p.z;
        }
    }
    cartesianTrimms.transform(transformacion);
}

```

2.6.7 Invertir normales

En una superficie NURBS, para invertir sus vectores normales, solo hay que cambiar la dirección u por v de la siguiente manera:

1. Se hace la operación transpuesta de la matriz M de puntos de control, es decir, se cambian las filas por columnas.
2. Se intercambian el vector de knots de u y el vector de knots de v .
3. Si la superficie es trimada, hay que intercambiar u y v en los puntos de control de todas las curvas en coordenadas paramétricas.

En una curva NURBS se puede invertir la dirección de la curva con los siguientes pasos:

1. Se invierte el vector de puntos de control.
2. Se invierte el vector de knots, es necesario que esté normalizado, cada valor u del vector se cambia por $1 - u$.

2.7 Primitivas

En esta sección se describen varios ejemplos de curvas y superficies que se pueden crear mediante NURBS. Posteriormente, en el apartado de modelado geométrico, se mostrarán algoritmos complejos para generar superficies arbitrarias a partir de curvas.

2.7.1 Curvas NURBS

En la sección 2.3.1 se vio el funcionamiento de las curvas NURBS. A continuación se describen los algoritmos para generar los distintos tipos de curvas como pueden ser líneas, arcos, elipses, parábolas, etc.

2.7.1.1 Línea

La línea o segmento se forma a partir de dos puntos P y Q en coordenadas cartesianas.



Figura 2.30: Curvas NURBS: Línea

Para crear una línea se utiliza una curva NURBS de grado 1 y dos puntos de control, que coinciden con los extremos de la línea P y Q .

Algoritmo 2.8 Curvas NURBS: Línea

```
ControlPoint [] cps = new ControlPoint [2];
cps [0] = new ControlPoint (p1.x, p1.y, p1.z, 1);
cps [1] = new ControlPoint (p2.x, p2.y, p2.z, 1);
double [] knots = {0, 0, 1, 1};
NurbsCurve segment = new NurbsCurve (cps, knots, 1);
```

2.7.1.2 Circunferencia

La circunferencia es una curva plana y cerrada donde todos sus puntos están a igual distancia del centro. Para construir una circunferencia se necesita el centro en coordenadas cartesianas y el radio.

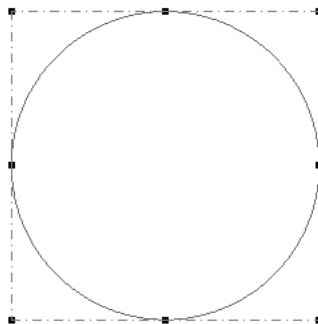


Figura 2.31: Curvas NURBS: Circunferencia

En la figura 2.31 se pueden ver los puntos de control de la circunferencia. Para construirla se utiliza una curva NURBS de grado 2. El peso de los puntos de control de las esquinas es $w = \frac{\sqrt{2}}{2}$.

Algoritmo 2.9 Curvas NURBS: Circunferencia

```

double w = Math.sqrt(2) / 2;
ControlPoint [] cps = new ControlPoint [9];
cps [0] = new ControlPoint(p.x+radius, p.y, p.z, 1);
cps [1] = new ControlPoint(p.x+radius, p.y+radius, p.z, w);
cps [2] = new ControlPoint(p.x, p.y+radius, p.z, 1);
cps [3] = new ControlPoint(p.x-radius, p.y+radius, p.z, w);
cps [4] = new ControlPoint(p.x-radius, p.y, p.z, 1);
cps [5] = new ControlPoint(p.x-radius, p.y-radius, p.z, w);
cps [6] = new ControlPoint(p.x, p.y-radius, p.z, 1);
cps [7] = new ControlPoint(p.x+radius, p.y-radius, p.z, w);
cps [8] = new ControlPoint(p.x+radius, p.y, p.z, 1);
double [] knots = {0, 0, 0, 0.25, 0.25, 0.5, 0.5, 0.75, 0.75, 1, 1, 1};
NurbsCurve circle = new NurbsCurve(cps, knots, 2);

```

También es posible construir un arco de circunferencia entre los ángulos Φ_1 y Φ_2 . El algoritmo se describe en A7.1 [9].

2.7.1.3 Elipse

Una elipse es la curva cerrada con dos ejes de simetría que resulta al cortar la superficie de un cono por un plano oblicuo al eje de simetría. Su construcción mediante curvas NURBS es similar a la circunferencia, solo que en este caso hacen falta dos radios.

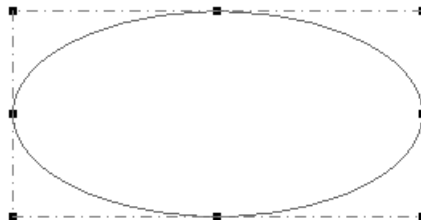


Figura 2.32: Curvas NURBS: Elipse

Al igual que en la circunferencia, se utiliza una curva NURBS de grado 2 y nueve puntos de control. El peso de los puntos de control de las esquinas es $w = \frac{\sqrt{2}}{2}$. Los dos radios de la elipse son *radius1* y *radius2*.

Algoritmo 2.10 Curvas NURBS: Elipse

```

double w = Math.sqrt(2) / 2;
ControlPoint [] cps = new ControlPoint [9];
cps [0] = new ControlPoint(p.x+radius1, p.y, p.z, 1);
cps [1] = new ControlPoint(p.x+radius1, p.y+radius2, p.z, w);
cps [2] = new ControlPoint(p.x, p.y+radius, p.z, 1);
cps [3] = new ControlPoint(p.x-radius1, p.y+radius2, p.z, w);
cps [4] = new ControlPoint(p.x-radius1, p.y, p.z, 1);
cps [5] = new ControlPoint(p.x-radius1, p.y-radius2, p.z, w);
cps [6] = new ControlPoint(p.x, p.y-radius, p.z, 1);
cps [7] = new ControlPoint(p.x+radius1, p.y-radius2, p.z, w);
cps [8] = new ControlPoint(p.x+radius1, p.y, p.z, 1);
double [] knots = {0, 0, 0, 0.25, 0.25, 0.5, 0.5, 0.75, 0.75, 1, 1, 1};
NurbsCurve ellipse = new NurbsCurve(cps, knots, 2);

```

2.7.1.4 Parábola

Una parábola se define como el lugar geométrico de los puntos de un plano que equidistan de una recta llamada directriz y un punto exterior a ella llamado foco.

La propiedad más interesante de la parábola es que la tangente refleja los rayos paralelos al eje de la parábola en dirección al foco. Las aplicaciones prácticas son muchas: las antenas embarcadas en satélites y radiotelescopios aprovechan el principio concentrando señales recibidas desde un emisor lejano en un receptor colocado en la posición del foco.

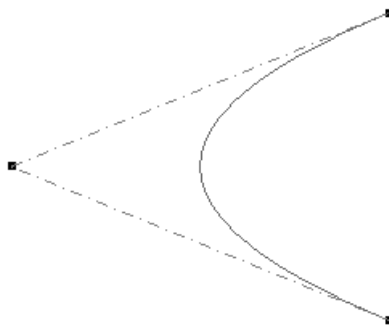


Figura 2.33: Curvas NURBS: Parábola

Hay múltiples formas para construir una parábola con diferentes parámetros de entrada. En diseño geométrico la que parece más interesante es la que necesita el centro de la directriz, la distancia al foco y un punto extremo de la parábola. La parábola se puede construir como una curva NURBS cuadrática (grado=2) con tres puntos de control.

Algoritmo 2.11 Curvas NURBS: Parábola

```

public static NurbsCurve createParabola(Point2d directrixCenter, Point2d p, double focus) {
    Point3d p0 = new Point3d(p.x - directrixCenter.x, p.y - directrixCenter.y, 0);
    Point3d pFocus = new Point3d(focus, 0, 0);
    Vector3d v0 = MathFunctions.differenceVector(pFocus, p0);
    Point3d p2 = new Point3d(p0.x, -1 * p0.y, 0);
    Vector3d v2 = MathFunctions.differenceVector(pFocus, p2);
    Vector3d vx = new Vector3d(-1, 0, 0);

    //Bisector para sacar las tangentes
    v0.normalize();
    v0.add(vx);
    v0.normalize();
    v2.normalize();
    v2.add(vx);
    v2.normalize();
    Point3d p1 = new Point3d();
    lineIntersect3D(p0, v0, p2, v2, p1);

    //Crear parabola
    p0.add(directrixCenter);
    p1.add(directrixCenter);
    p2.add(directrixCenter);
    ControlPoint[] cps = new ControlPoint[3];
    cps[0] = new ControlPoint(p0.x, p0.y, 0, 1);
    cps[1] = new ControlPoint(p1.x, p1.y, 0, 1);
    cps[2] = new ControlPoint(p2.x, p2.y, 0, 1);
    double[] knots = {0, 0, 0, 1, 1, 1};
    return new NurbsCurve(cps, knots, 2);
}

```

2.7.1.5 Hipérbola

Una hipérbola se define como el lugar geométrico de los puntos de un plano tales que el valor absoluto de la diferencia de sus distancias a dos puntos fijos, llamados focos, es igual a la distancia entre los vértices, la cual es una constante positiva.

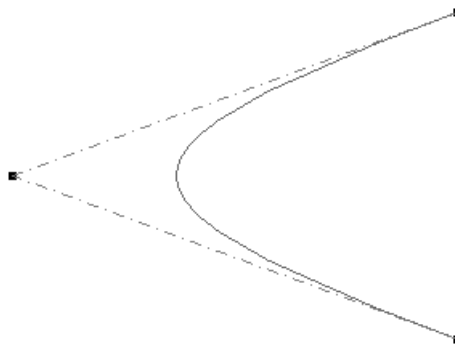


Figura 2.34: Curvas NURBS: Hipérbola

Al igual que con la parábola, hay varias maneras de construir una hipérbola. Para construir la hipérbola se necesita como parámetros de entrada el centro de la directriz, la distancia al foco y un punto extremo.

Se puede dibujar una hipérbola como una curva NURBS cuadrática (grado=2). La curva tiene tres puntos de control: P_0 , P_1 , P_2 . Los puntos P_0 y P_2 son los puntos finales de la curva mientras que P_1 está cerca del vértice de la curva.

El punto P_1 es el más difícil de calcular, se encuentra en el eje X, en la intersección de las tangentes de P_0 y P_2 . La tangente para un punto (x', y') en una hipérbola es:

$$y - y' = \frac{b \cdot x'}{(a^2 \cdot y') \cdot (x - x')}$$

Como se conoce que $y = 0$ en P_1 , se puede resolver para x de la siguiente manera:

$$x = \frac{-y'^2 \cdot a^2}{b^2 \cdot x'} + x'$$

Para dibujar la hipérbola, también hay que saber qué peso se le da a P_1 . La fórmula es $w = d/(e - d)$ donde d es la distancia entre el vértice y el punto M y e es la distancia entre P_1 y el punto M . El punto M es la coordenada x de P_0 .

Algoritmo 2.12 Curvas NURBS: Hipérbola

```

public static NurbsCurve createHyperbola(Point2d directrixCenter, Point2d p0, double focus) {
    //Se calculan los parametros a,b,c de la hiperbola
    p0.sub(directrixCenter);
    Point2d pFocus1 = new Point2d(focus, 0);
    Point2d pFocus2 = new Point2d(-focus, 0);
    double d1 = pFocus1.distance(p0);
    double d2 = pFocus2.distance(p0);
    double c = focus; //c^2 = a^2 + b^2
    double a = Math.abs(d1-d2)/2;
    double b = Math.sqrt((c*c) - (a*a));
    Point2d p2 = new Point2d(p0.x, -1 * p0.y);
    //Posición de P1
    double x = (-1 * p0.y * p0.y * a * a) / (b * b * p0.x) + p0.x;
    Point2d p1 = new Point2d(x, 0);
    if (p0.x < 0) { //Simetria
        p1.x = -x;
        a = -a;
    }
    //Peso de P1
    Point2d apex = new Point2d(a, 0);
    double point_M = p0.x;
    double d = point_M - apex.x;
    double e = point_M - p1.x;
    double weight = d / (e - d);
    //Crear hiperbola
    p0.add(directrixCenter);
    p1.add(directrixCenter);
    p2.add(directrixCenter);
    ControlPoint[] cps = new ControlPoint[3];
    cps[0] = new ControlPoint(p0.x, p0.y, 0, 1);
    cps[1] = new ControlPoint(p1.x, p1.y, 0, weight);
    cps[2] = new ControlPoint(p2.x, p2.y, 0, 1);
    double[] knots = new double[]{0., 0., 0., 1., 1., 1.};
    return new NurbsCurve(cps, knots, 2);
}
  
```

2.7.2 Superficies NURBS

En la sección 2.3.2 se describe el funcionamiento de las superficies NURBS. En este apartado se muestran distintos tipos de primitivas que se pueden crear mediante este tipo de superficies.

2.7.2.1 Faceta

Una faceta es la superficie más sencilla que se puede crear. Se define mediante una superficie NURBS de grado=1 y cuatro puntos de control que corresponden con las esquinas de la faceta.

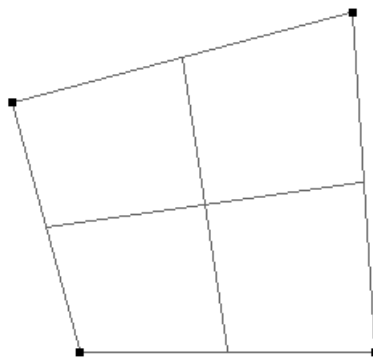


Figura 2.35: Superficies NURBS: Faceta

Se puede utilizar una faceta para construir un triángulo si se repite uno de sus puntos de control. También se utiliza para construir cualquier tipo de cuadrángulo.

Dados los cuatro puntos P_0 , P_1 , P_2 , P_3 se construye la superficie NURBS de la siguiente manera:

Algoritmo 2.13 Superficies NURBS: Faceta

```
ControlPoint [][] cps = new ControlPoint [2][2];
cps [0][0] = new ControlPoint (p0.x, p0.y, p0.z, 1);
cps [0][1] = new ControlPoint (p1.x, p1.y, p1.z, 1);
cps [1][0] = new ControlPoint (p2.x, p2.y, p2.z, 1);
cps [1][1] = new ControlPoint (p3.x, p3.y, p3.z, 1);
double uknots [] = {0, 0, 1, 1};
double vknots [] = {0, 0, 1, 1};
NurbsSurface facet = new NurbsSurface (cps, uknots, vknots, 1, 1);
```

Como se ha visto anteriormente, una superficie NURBS se define mediante una matriz de puntos de control *cps* y dos vectores de knots *uknots* y *vknots* para las direcciones *u* y *v* de la superficie. Los dos últimos parámetros corresponden al grado en *u* y el grado en *v* de la superficie.

2.7.2.2 Disco

Un disco es una superficie plana de frontera circular. El disco se define mediante cuatro superficies NURBS, cada superficie corresponde con un cuarto de disco.

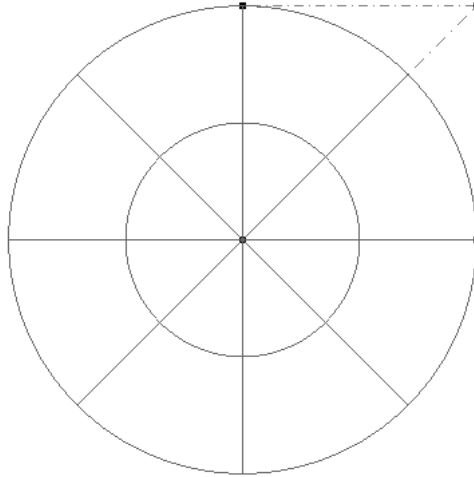


Figura 2.36: Superficies NURBS: Disco

Los parámetros de entrada son el centro en coordenadas cartesianas y el radio. Para construir un cuarto de disco se utiliza una matriz de puntos de control de 3x2 elementos.

La superficie NURBS tiene grado en $u=2$ y grado en $v=1$ y se construye de la siguiente manera:

Algoritmo 2.14 Superficies NURBS: Disco

```
ControlPoint cps [][] = new ControlPoint[3][2];
cps[0][0] = new ControlPoint(center.x, center.y, center.z, 1);
cps[0][1] = new ControlPoint(center.x, radius+center.y, center.z, 1);
cps[1][0] = new ControlPoint(center.x, center.y, center.z, Math.sqrt(2)/2.0);
cps[1][1] = new ControlPoint(radius+center.x, radius+center.y, center.z, Math.sqrt(2)/2.0);
cps[2][0] = new ControlPoint(center.x, center.y, center.z, 1);
cps[2][1] = new ControlPoint(radius+center.x, center.y, center.z, 1);
double u[] = {0, 0, 0, 1, 1, 1};
double v[] = {0, 0, 1, 1};
NurbsSurface disk1 = new NurbsSurface(cps, u, v, 2, 1);
```

Las primitivas anillo y elipse se construyen de manera similar al disco, pero utilizando dos radios.

2.7.2.3 Cubo

Cubo o hexaedro regular es un poliedro limitado por seis caras cuadradas congruentes. Es uno de los denominados sólidos platónicos.

Un cubo, además de ser un hexaedro, puede ser clasificado también como paralelepípedo, recto y rectangular, pues todas sus caras son de cuatro lados y paralelas dos a dos.

Incluso, se puede entender como un prisma recto, cuya base es un cuadrado y su altura equivalente al lado de la base.

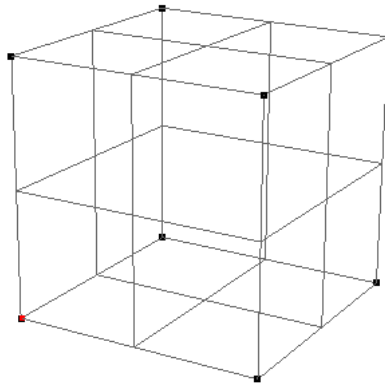


Figura 2.37: Superficies NURBS: Cubo

Para construir un cubo se necesitan los siguientes parámetros de entrada: un punto en coordenadas cartesianas que corresponde al punto de la base marcado en rojo y tres valores que serán la altura, anchura y profundidad.

Para generar la geometría construiremos seis facetas utilizando los siguientes puntos:

Algoritmo 2.15 Superficies NURBS: Cubo

```

Point3d p1 = pBase ;
Point3d p2 = new Point3d(pBase.x+width , pBase.y , pBase.z );
Point3d p3 = new Point3d(pBase.x , pBase.y+depth , pBase.z );
Point3d p4 = new Point3d(pBase.x+width , pBase.y+depth , pBase.z );

Point3d p5 = new Point3d(pBase.x , pBase.y , pBase.z+height );
Point3d p6 = new Point3d(pBase.x+width , pBase.y , pBase.z+height );
Point3d p7 = new Point3d(pBase.x , pBase.y+depth , pBase.z+height );
Point3d p8 = new Point3d(pBase.x+width , pBase.y+depth , pBase.z+height );

```

2.7.2.4 Cilindro

Un cilindro es una superficie de las denominadas cuádricas formada por el desplazamiento paralelo de una recta llamada generatriz a lo largo de una curva plana, denominada directriz del cilindro.

Si la directriz es un círculo y la generatriz es perpendicular a él, entonces la superficie obtenida, llamada cilindro circular recto, será de revolución y tendrá por lo tanto todos sus puntos situados a una distancia fija de una línea recta, el eje del cilindro. El sólido encerrado por esta superficie y por dos planos perpendiculares al eje también es llamado cilindro.

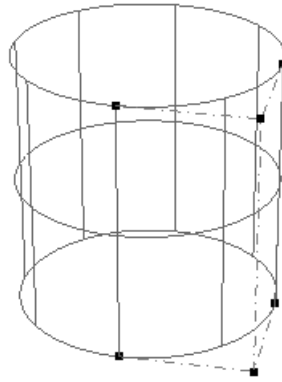


Figura 2.38: Superficies NURBS: Cilindro

El cilindro está formado por cuatro superficies NURBS. Cada superficie tiene una matriz de puntos de control de 3×2 elementos y un grado en $u=2$ y grado en $v=1$, al igual que el disco.

Los parámetros necesarios para construir el cilindro son: el punto central de la base en coordenadas cartesianas y dos valores que corresponden al radio y a la altura. En el siguiente ejemplo se construye una de las cuatro superficies NURBS del cilindro.

Algoritmo 2.16 Superficies NURBS: Cilindro

```

double w = Math.sqrt(2)/2.0;
ControlPoint cps [][] = new ControlPoint[3][2];
cps[0][0] = new ControlPoint(center.x, radius+center.y, height+center.z, 1);
cps[0][1] = new ControlPoint(center.x, radius+center.y, center.z, 1);
cps[1][0] = new ControlPoint(radius+center.x, radius+center.y, height+center.z, w);
cps[1][1] = new ControlPoint(radius+center.x, radius+center.y, center.z, w);
cps[2][0] = new ControlPoint(radius+center.x, center.y, height+center.z, 1);
cps[2][1] = new ControlPoint(radius+center.x, center.y, center.z, 1);
double u[] = {0, 0, 0, 1, 1, 1 };
double v[] = {0, 0, 1, 1};
NurbsSurface cylinder1 = new NurbsSurface(cps, u, v, 2, 1);

```

2.7.2.5 Cono

En geometría, un cono es un sólido de revolución generado por el giro de un triángulo rectángulo alrededor de uno de sus catetos. Al círculo conformado por el otro cateto se denomina base y al punto donde confluyen las generatrices se llama vértice o cúspide.

Para construir el cono, se utilizarán como parámetros el punto central de la base en coordenadas cartesianas y dos valores que corresponderán al radio inferior y superior. El cono será cerrado si el radio superior es igual a 0, si el radio superior es mayor que 0 el cono será abierto.

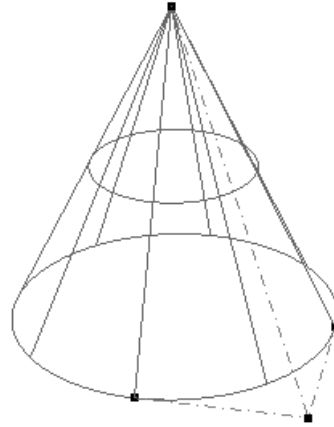


Figura 2.39: Superficies NURBS: Cono

Se puede considerar un cono como un caso especial del cilindro, ya que su construcción es similar. El cono está formado por cuatro superficies NURBS. Para construir una superficie del cono solo hay que utilizar el radio superior en los puntos de control del cilindro que tienen el parámetro *height*.

Algoritmo 2.17 Superficies NURBS: Cono

```

double w = Math.sqrt(2)/2.0;
ControlPoint cps [][] = new ControlPoint[3][2];
cps[0][0] = new ControlPoint(center.x, topRadius+center.y, height+center.z, 1);
cps[0][1] = new ControlPoint(center.x, bottomRadius+center.y, center.z, 1);
cps[1][0] = new ControlPoint(topRadius+center.x, topRadius+center.y, height+center.z, w);
cps[1][1] = new ControlPoint(bottomRadius+center.x, bottomRadius+center.y, center.z, w);
cps[2][0] = new ControlPoint(topRadius+center.x, center.y, height+center.z, 1);
cps[2][1] = new ControlPoint(bottomRadius+center.x, center.y, center.z, 1);
double u[] = {0, 0, 0, 1, 1, 1 };
double v[] = {0, 0, 1, 1};
NurbsSurface conel = new NurbsSurface(cps, u, v, 2, 1);

```

2.7.2.6 Esfera

Una superficie esférica es una superficie de revolución formada por el conjunto de los puntos del espacio que equidistan de un punto llamado centro.

Los parámetros de la esfera serán el centro en coordenadas cartesianas y el radio.

La esfera está formada por ocho superficies NURBS. Cada superficie tiene una matriz de puntos de control de 3x3 elementos y un grado en *u* y en *v* igual a 2.

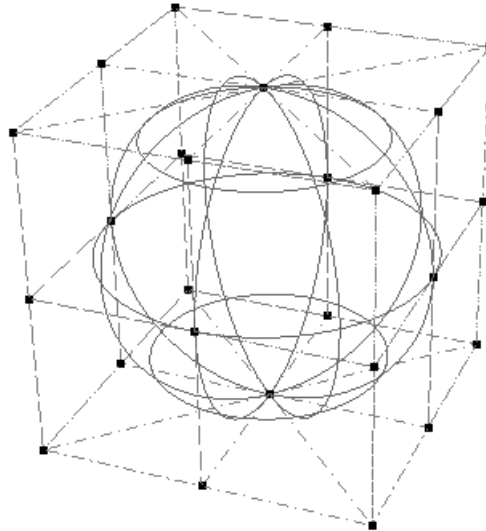


Figura 2.40: Superficies NURBS: Esfera

Cada cuadrante de la esfera se construye de la siguiente manera:

Algoritmo 2.18 Superficies NURBS: Esfera

```

double w = Math.sqrt(2)/2.0;
ControlPoint cps [][] = new ControlPoint[3][3];
cps[0][0] = new ControlPoint(center.x-radius, center.y, center.z, 1);
cps[0][1] = new ControlPoint(center.x-radius, center.y, center.z+radius, w);
cps[0][2] = new ControlPoint(center.x, center.y, center.z+radius, 1);
cps[1][0] = new ControlPoint(center.x-radius, center.y-radius, center.z, w);
cps[1][1] = new ControlPoint(center.x-radius, center.y-radius, center.z+radius, 0.5);
cps[1][2] = new ControlPoint(center.x, center.y, center.z+radius, w);
cps[2][0] = new ControlPoint(center.x, center.y-radius, center.z, 1);
cps[2][1] = new ControlPoint(center.x, center.y-radius, center.z+radius, w);
cps[2][2] = new ControlPoint(center.x, center.y, center.z+radius, 1);
double u[] = {0, 0, 0, 1, 1, 1};
double v[] = {0, 0, 0, 1, 1, 1};
NurbsSurface sphere1 = new NurbsSurface(cps, u, v, 2, 2);
  
```

2.8 Búsqueda de Puntos en Coordenadas Paramétricas

En la sección 2.3.2 se describen las superficies NURBS, en la figura 2.11 se puede observar la correspondencia entre el espacio paramétrico y el espacio real de la superficie.

En la clase 'NurbsSurface', descrita en la sección 2.4.4, existen dos métodos para convertir puntos entre el espacio paramétrico de la superficie y el espacio real:

- El método *pointOnSurface* permite obtener un punto de la superficie en coordenadas cartesianas a partir de las coordenadas paramétricas u y v . Este método, que se ha utilizado en los algoritmos de renderizado es analítico, es decir, existe una fórmula que permite obtener el punto en coordenadas cartesianas dado un valor de u y v .

- El método *getUVParameterInversion* realiza la operación contraria, esto es, dado un punto en coordenadas cartesianas busca el punto más cercano en la superficie y devuelve sus coordenadas paramétricas u y v . Desgraciadamente, no existe una función analítica que permita realizar esta operación.

En esta sección se muestra el algoritmo para realizar la operación *getUVParameterInversion* y se describe como se ha resuelto el problema en nuestro motor gráfico. Análogamente, el algoritmo es aplicable a una curva NURBS solo que en este caso se utiliza una sola dimensión u para obtener el resultado.

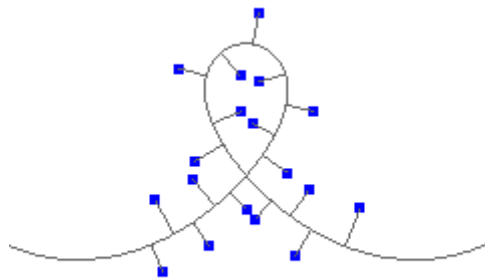


Figura 2.41: Point Inversion: Búsqueda de coordenadas paramétricas en NURBS

2.8.1 Método del Gradiente Conjugado

El método del gradiente es un método de descenso en el que se comienza a iterar en un punto arbitrario x_0 y se continúa siguiendo una dirección, obteniéndose una sucesión de puntos x_1, x_2, \dots hasta que se obtiene un punto lo suficientemente cercano a la solución x_* .

Con este método se puede minimizar una función de N variables siempre que la función se pueda aproximar por un modelo cuadrático:

$$f(x) = \frac{1}{2}xAx - bx + c$$

Este método es un método iterativo de tal manera que en la iteración $i + 1$ la solución x_{i+1} se obtiene a partir de la solución x_i de la iteración anterior de la siguiente manera:

$$x_{i+1} = x_i + \alpha_i \delta_i$$

Para obtener α_i en cada iteración se realiza una búsqueda lineal a lo largo de la dirección de descenso correspondiente. En cada iteración se elige la dirección δ_i para la que f decrece más rápidamente, que es la dirección contraria a $\nabla f(x_i)$.

$$\delta_{i+1} = -\nabla f(x_{i+1}) + \beta_i \delta_i$$

Las distintas variantes del método del gradiente conjugado dependen de como se calcula el parámetro β_i .

El algoritmo del gradiente conjugado se denomina 'frprmn' y se describe en la sección 10.6 de [16]. Para utilizarlo se ha implementado en JAVA mediante una clase abstracta llamada 'Gradient'. También es posible utilizar el algoritmo 'powell' de la sección 10.5 que permite calcular el mínimo de la función f sin utilizar la derivada, pero necesita más iteraciones que el gradiente conjugado.

2.8.2 Clase Gradient

La clase 'Gradient' implementa el algoritmo del gradiente conjugado en JAVA. Se utiliza una clase abstracta de la que posteriormente extienden las clases que utilicen el algoritmo.



Figura 2.42: Clase 'Gradient'

La implementación de la clase 'Gradient' se basa en el algoritmo 'frprmn.c' que se puede encontrar en [16] con las siguientes modificaciones:

- En JAVA los tipos de datos nativos *int* y *double* se pasan a las funciones por valor⁷. Las funciones del gradiente conjugado modifican internamente los parámetros de entrada, por lo tanto, hay que crear dos clases privadas llamadas 'Int' y 'Real' que

⁷La información de la variable se almacenan en una dirección de memoria diferente al recibirla en la función, por lo tanto si el valor de esa variable cambia no afecta la variable original, solo se modifica dentro del contexto de la función.

almacenen el valor de la variable, ya que los objetos en JAVA se pasan a las funciones por referencia⁸.

- Las funciones *func* y *dfunc* representan la función a minimizar y su derivada respectivamente. En C son punteros a funciones definidas como parámetros de entrada a la función *frprmn*. En JAVA se han definido como funciones abstractas, esto quiere decir que las clases que extiendan de 'Gradient' deberán definir la función a minimizar y su derivada.

2.8.3 Función a Minimizar

En la sección anterior se describe el método del gradiente conjugado que permite buscar el mínimo de una función. Para ello se necesita definir la función y su derivada.

En la búsqueda de puntos en coordenadas paramétricas la función a minimizar es el cuadrado de la distancia entre el punto que se quiere buscar en coordenadas cartesianas y un punto en la NURBS.

Curva NURBS

Para el caso de la curva NURBS la función depende de la coordenada paramétrica u y su ecuación es:

$$f(u) = (T - P(u))^2 = |\vec{v}|^2 \quad (2.18)$$

Donde T es el punto *Test* y $P(u)$ es la función *pointOnCurve* de la clase 'NurbsCurve' (2.4.3) que devuelve el punto en coordenadas cartesianas sobre la curva para el valor del parámetro u .

Se puede observar que aplicando el método del gradiente conjugado a esa función se obtiene el parámetro u para el cual la distancia entre T y $P(u)$ es mínima.

Para aplicar el método, se necesita la derivada $f'(u)$ cuya expresión es:

$$f'(u) = \frac{\partial f}{\partial u} = 2 \cdot \vec{v} \cdot \overrightarrow{\frac{\partial P(u)}{\partial u}} \quad (2.19)$$

Donde \vec{v} es el vector diferencia de $T - P(u)$ y $\overrightarrow{\frac{\partial P(u)}{\partial u}}$ es la primera derivada en la curva para la coordenada paramétrica u que se puede obtener con la función *curveDerivs* de la clase 'NurbsCurve'.

⁸La variable que se recibe como parámetro en la función apunta exactamente a la misma dirección de memoria que la variable original por lo que si dentro de la función se modifica su valor también se modifica la variable original.

Algoritmo 2.19 NurbsCurveGradient: Función y Derivada

```

@Override
protected double func(Real[] x) {
    double u = x[0].value;

    Point3d pcurve = curve.pointOnCurve(u);
    Vector3d vector = MathFunctions.differenceVector(test, pcurve);
    return MathFunctions.scalarProduct(vector, vector);
}
@Override
protected void dfunc(Real x[], Real y[]) {
    double u = x[0].value;

    Vector3d du = curve.curveDerivU(u);
    Point3d pcurve = curve.pointOnCurve(u);
    Vector3d vector = MathFunctions.differenceVector(pcurve, test);
    y[0] = new Real(2 * MathFunctions.scalarProduct(vector, du));
}

```

Superficie NURBS

En la superficie NURBS la función a minimizar depende de las coordenadas paramétricas u y v , su ecuación es la siguiente:

$$f(u, v) = (T - P(u, v))^2 = |\vec{v}|^2 \quad (2.20)$$

Al igual que en la curva, T es el punto que se quiere buscar y $P(u, v)$ es la función *pointOnSurface* de la clase 'NurbsSurface' (2.4.4). El vector \vec{v} es el vector diferencia $T - P(u, v)$ cuyo modulo es la distancia entre los puntos T y P en coordenadas cartesianas.

El método del gradiente conjugado necesita la derivada de la función respecto a u y v para saber que dirección toma cada parámetro al iterar. Se calculan mediante las siguientes expresiones:

$$f'_u(u, v) = \frac{\partial f}{\partial u} = 2 \cdot \vec{v} \cdot \overrightarrow{\frac{\partial P(u, v)}{\partial u}} \quad (2.21)$$

$$f'_v(u, v) = \frac{\partial f}{\partial v} = 2 \cdot \vec{v} \cdot \overrightarrow{\frac{\partial P(u, v)}{\partial v}} \quad (2.22)$$

Donde $\overrightarrow{\frac{\partial P(u, v)}{\partial u}}$ es la primera derivada de la superficie en u, v respecto a u que se obtiene de la función *surfaceDerivU* de la clase 'NurbsSurface' y $\overrightarrow{\frac{\partial P(u, v)}{\partial v}}$ es la primera derivada de la superficie en u, v respecto a v que se obtiene de la función *surfaceDerivV* de la misma clase.

Algoritmo 2.20 NurbsSurfaceGradient: Función y Derivada

```

@Override
protected double func(Real[] x) {
    double u = x[0].value;
    double v = x[1].value;

    Point3d psurface = surface.pointOnSurface(u, v);
    Vector3d vector = MathFunctions.differenceVector(test, psurface);
    return MathFunctions.scalarProduct(vector, vector);
}
@Override
protected void dfunc(Real x[], Real y[]) {
    double u = x[0].value;
    double v = x[1].value;

    Vector3d du = surface.surfaceDerivU(u, v);
    Vector3d dv = surface.surfaceDerivV(u, v);
    Point3d psurface = surface.pointOnSurface(u, v);
    Vector3d vector = MathFunctions.differenceVector(psurface, test);
    y[0] = new Real(2 * MathFunctions.scalarProduct(vector, du));
    y[1] = new Real(2 * MathFunctions.scalarProduct(vector, dv));
}

```

2.8.4 Búsqueda de Semillas

El método del gradiente conjugado necesita un punto x_0 para comenzar a iterar, a ese punto se le conoce como semilla. Si la semilla es un punto cercano a la solución, el algoritmo necesitará menos iteraciones para encontrarla. También es posible que dada una semilla el gradiente no encuentre solución óptima en el número máximo de iteraciones.

Para hacer más óptima la búsqueda con el método del gradiente conjugado, se hace un preprocesado de la curva o superficie NURBS donde se obtiene un conjunto de semillas en el que aplicar el algoritmo. El proceso es el siguiente:

- Se crea un array en el que se van insertando las semillas ordenadamente en base a su distancia con el punto T que se quiere encontrar.
- Se divide la curva o superficie NURBS de la misma manera que en el algoritmo 2.1. Para cada punto se calcula la distancia con el punto T y se insertan sus coordenadas paramétricas en el array anterior, quedando ordenadas por la distancia al punto de búsqueda.
- Se llama al gradiente sucesivamente con cada semilla del array hasta encontrar un punto que coincida con el punto T en cuyo caso termina o devolver el punto más cercano a la solución de todas las calculadas.

Para la superficie NURBS existe también un proceso alternativo con el que se han obtenido buenos resultados y suele ser más rápido. Se realiza de la siguiente manera:

- Se obtienen las curvas de borde de la superficie NURBS como puede observarse en la figura 2.22.

- Se busca la semilla más cercana al punto T dentro de la superficie NURBS con el método anterior.
- Se calcula el gradiente en las 4 curvas de borde y en el punto interior de la superficie y se devuelve la mejor solución.

2.9 Algoritmos de Modelado de Superficies NURBS

Generalmente, la creación de modelos geométricos complejos mediante superficies NURBS se realiza de forma incremental, permitiendo utilizar puntos y curvas auxiliares para construir superficies arbitrarias. En esta sección se describen los algoritmos, que han sido implementados en el motor gráfico, para crear superficies NURBS a partir de otros elementos como puntos, vectores o curvas.

2.9.1 Interpolación Global

El método de interpolación global permite construir una curva o superficie NURBS que pase por un conjunto de puntos. El método es global, esto quiere decir que cualquier cambio en uno de los puntos afecta a toda la superficie. Para realizar la interpolación global es necesario resolver un sistema de ecuaciones.

La interpolación global de superficies está basada en la de curvas, que se muestra a continuación.

Interpolación Global de Curvas

Suponer que se dispone de una serie de puntos $\{Q_k\}$ con $k = 0, \dots, n$ y se quiere interpolar una curva B-spline no racional de grado p . Si se asigna un valor al parámetro \bar{u}_k para cada Q_k , y se selecciona un vector de nodos apropiado $U = \{u_0, \dots, u_m\}$, se puede construir un sistema lineal de ecuaciones $(n + 1) \times (n + 1)$.

$$Q_k = C(\bar{u}_k) = \sum_{i=0}^n N_{i,p}(\bar{u}_k) P_i \quad (2.23)$$

Los puntos de control P_i , son las $(n + 1)$ incógnitas. La elección de los nodos \bar{u}_k afecta a la forma y a la parametrización de la curva, en el algoritmo se usa el método centripeto ya que proporciona los mejores resultados.

$$\bar{u}_k = \bar{u}_{k-1} + \frac{\sqrt{|Q_k - Q_{k-1}|}}{d} \quad \left\{ \begin{array}{l} \bar{u}_0 = 0 \\ \bar{u}_n = 1 \\ k = 1, \dots, n - 1 \\ d = \sum_{k=1}^n \sqrt{|Q_k - Q_{k-1}|} \end{array} \right. \quad (2.24)$$

En la siguiente figura se muestra el resultado de una curva NURBS interpolada utilizando este algoritmo.

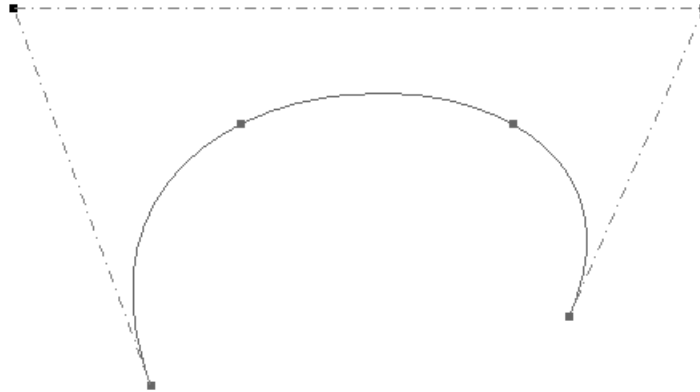


Figura 2.43: Interpolación Global de Curvas NURBS

A continuación se muestra el algoritmo implementado en el motor gráfico, para resolver el sistema de ecuaciones se utilizan las clases 'GMatrix' y 'GVector' de Java 3D.

Algoritmo 2.21 Interpolación Global de Curvas NURBS

```

public static NurbsCurve globalCurveInterpolation(Point3d[] points, int p) {
    int n = points.length - 1;
    double[] A = new double[(n + 1) * (n + 1)];
    double[] uk = centripetal(points);
    KnotVector uKnots = new KnotVector(uk, p);
    for (int i = 0; i <= n; i++) {
        int span = uKnots.findSpan(uk[i]);
        double[] tmp = uKnots.basisFunctions(span, uk[i]);
        System.arraycopy(tmp, 0, A, i * (n + 1) + span - p, tmp.length);
    }
    GMatrix a = new GMatrix(n + 1, n + 1, A);
    GVector perm = new GVector(n + 1);
    GMatrix lu = new GMatrix(n + 1, n + 1);
    a.LUD(lu, perm);
    ControlPoint[] cps = new ControlPoint[n + 1];
    for (int i = 0; i < cps.length; i++) {
        cps[i] = new ControlPoint(0, 0, 0, 1);
    }
    //Coordenada X
    GVector b = new GVector(n + 1);
    for (int j = 0; j <= n; j++) {
        b.setElement(j, points[j].x);
    }
    GVector sol = new GVector(n + 1);
    sol.LUDBackSolve(lu, b, perm);
    for (int j = 0; j <= n; j++) {
        cps[j].x = sol.getElement(j);
    }
    ... //Repetir para las coordenadas Y y Z
    return new NurbsCurve(cps, uKnots);
}

```

Interpolación Global de Superficies

Dada una serie de puntos $\{Q_{k,l}\}$ con $k = 0, \dots, n$ y $l = 0, \dots, m$, se puede construir la superficie B-Spline no racional de grado (p, q) interpolando esos puntos.

$$Q_{k,l} = S(\bar{u}_k, \bar{v}_l) = \sum_{i=0}^n \sum_{j=0}^m N_{i,p}(\bar{u}_k) N_{j,q}(\bar{v}_l) P_{i,j} \quad (2.25)$$

En el caso de la superficie los puntos de control $P_{i,j}$ pueden obtenerse como una secuencia de interpolación de curvas. Esto se consigue interpolando las filas de la matriz de puntos en la dirección u con curvas de grado p , obteniendo la matriz de puntos de control intermedios $R_{i,j}$, e interpolando por columnas en la dirección v con curvas de grado q , dando como resultado la matriz de puntos de control $P_{i,j}$. En el siguiente algoritmo puede observarse el procedimiento en detalle.

Algoritmo 2.22 Interpolación Global de Superficies NURBS

```

public static NurbsSurface globalSurfaceInterpolation(Point3d points [], int p, int q) {
    int n = points.length - 1;
    int m = points[0].length - 1;

    //Calculo de los nodos de la superficie interpolada A9.3
    double uv[][] = surfaceMeshParameters(points, n, m);
    KnotVector u = averaging(uv[0], p);
    KnotVector v = averaging(uv[1], q);

    //Primera interpolación en u para obtener Rij
    ControlPoint r[][] = new ControlPoint[m + 1][n + 1];
    for (int l = 0; l <= m; l++) {
        Point3d tmp[] = new Point3d[n + 1];
        for (int i = 0; i <= n; i++) {
            tmp[i] = points[i][l];
        }
        NurbsCurve curve = globalCurveInterpolation(tmp, p);
        r[l] = curve.getControlPoints();
        for (int i = 0; i < tmp.length; i++) {
            r[l][i] = new ControlPoint(tmp[i], 1);
        }
    }

    //Segunda interpolación en v para obtener Pij
    ControlPoint cp[][] = new ControlPoint[n + 1][m + 1];
    for (int i = 0; i <= n; i++) {
        Point3d tmp[] = new Point3d[m + 1];
        for (int j = 0; j <= m; j++) {
            tmp[j] = r[j][i].getPoint3d();
        }
        NurbsCurve curve = globalCurveInterpolation(tmp, q);
        cp[i] = curve.getControlPoints();
        for (int j = 0; j < tmp.length; j++) {
            cp[i][j] = new ControlPoint(tmp[j], 1);
        }
    }

    return new NurbsSurface(cp, u, v);
}

```

En la siguiente figura se muestra la construcción de una superficie NURBS utilizando el algoritmo de interpolación global de superficies.

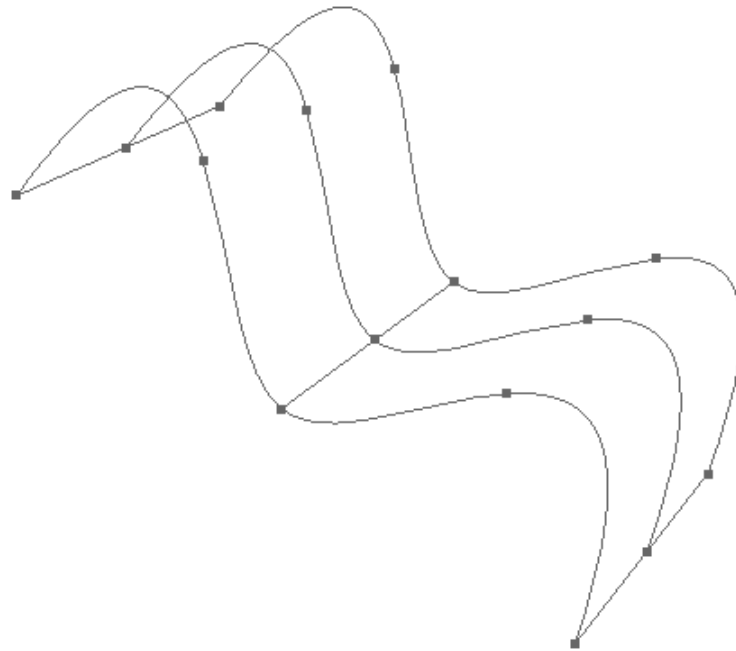


Figura 2.44: Interpolación Global de Superficies NURBS

2.9.2 Extrusión

La operación de extrusión permite construir una superficie NURBS utilizando una curva como perfil (que puede ser cerrada) y un vector de dirección. La superficie de extrusión $S^w(u, v)$ de grado (p, q) se define mediante la siguiente expresión.

$$S^w(u, v) = \sum_{i=0}^n \sum_{j=0}^m N_i^p(u) N_j^q(v) P_{i,j}^w \quad \left| \begin{array}{l} U = \{u_0, \dots, u_r\} \\ P_{i,0}^w = P_i^w \\ V = \{0, 0, 1, 1\} \\ P_{i,1}^w = P_i^w + dW_i^w \end{array} \right. \quad (2.26)$$

El algoritmo se puede modificar para hacer la operación de extrusión a un punto de manera muy sencilla. También es posible realizar la operación de extrusión sobre una superficie, generando un sólido donde las paredes laterales se construyen mediante una extrusión de cada curva de borde de la superficie, y la tapa de arriba es una copia de la superficie original desplazada con el vector de dirección.

En el siguiente algoritmo puede observarse como se crea la superficie de extrusión en el motor gráfico a partir de una curva NURBS y un vector de dirección.

Algoritmo 2.23 Extrusión de una curva NURBS

```

public static NurbsSurface extrudeCurve(NurbsCurve curve, Vector3d extrude) {
    double v[] = new double[]{0.0, 0.0, 1.0, 1.0}
    ControlPoint [][] cpoints = new ControlPoint[curve.getControlPoints().length][2];
    ControlPoint [] curvePoints = curve.getControlPoints();
    for (int i = 0; i < cpoints.length; i++) {
        for (int j=0; j<2; j++) {
            ControlPoint cp = new ControlPoint();
            cp.x = curvePoints[i].x + j * extrude.x;
            cp.y = curvePoints[i].y + j * extrude.y;
            cp.z = curvePoints[i].z + j * extrude.z;
            cp.w = curvePoints[i].w;
            cpoints[i][j] = cp;
        }
    }
    return new NurbsSurface(cpoints, curve.getKnots(), v, curve.getDegree(), 1);
}

```

La siguiente figura muestra la operación de extrusión en funcionamiento, se ha generado una curva NURBS con forma de espiral en el plano XY y se ha realizado la operación con el vector $v = (0, 0, 1)$ en la dirección Z.

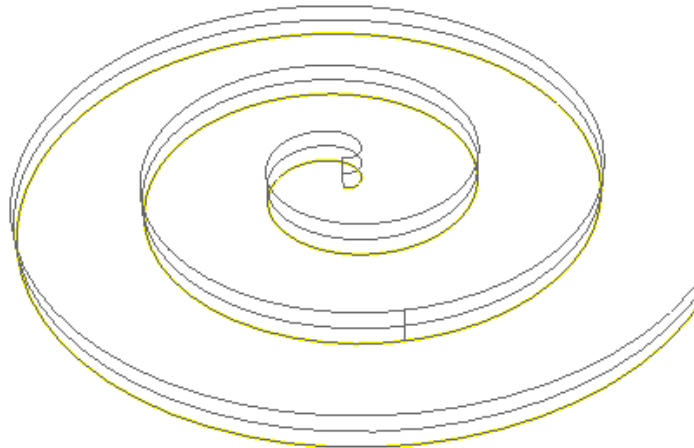


Figura 2.45: Extrusión de una curva NURBS

2.9.3 Revolución

Dada una curva NURBS $C(u)$, conocida como generatriz, la operación de revolución consiste en construir la superficie que se forma al ir rotando la curva generatriz sobre un eje con un ángulo $\alpha = [0, 2\pi)$.

La operación de revolución dados los ángulos $\theta_{ini}, \theta_{fin}$ se realiza de la misma manera, sacando el ángulo $\alpha = \theta_{fin} - \theta_{ini}$ y aplicando al resultado una operación de rotación usando el eje de revolución y el ángulo θ_{ini} .

En el siguiente algoritmo se puede ver en detalle la operación de revolución, nótese que el algoritmo va añadiendo arcos (según *narcs*) dependiendo del ángulo de revolución.

Algoritmo 2.24 Revolución de una curva NURBS

```

public static NurbsSurface revolveCurve(NurbsCurve curve, Point3d orig,
                                         Vector3d dir, double theta) {
    int narcs = getRevolvedArcs(theta); //Devuelve 1, 2, 3 o 4 cada 90 grados
    double uKnot[] = getRevolvedKnots(narcs)
    double dtheta = theta / narcs;
    int j = 3 + 2 * (narcs - 1);
    double angle = 0;
    double cos[] = new double[narcs + 1];
    double sin[] = new double[narcs + 1];
    for (int i = 0; i <= narcs; i++) {
        cos[i] = Math.cos(angle);
        sin[i] = Math.sin(angle);
        angle += dtheta;
    }
    ControlPoint pj[] = curve.getControlPoints();
    double wm = Math.cos(dtheta / 2);
    Point3d O, P2, P0, tmp;
    Vector3d T2, T0, X, Y;
    ControlPoint pij[][] = new ControlPoint[2 * narcs + 1][pj.length];
    for (j=0; j<pj.length; j++) {
        CurveCreator.pointToLine3D(orig, dir, pj[j].getPoint3d(), O);
        X = pj[j] - O
        double r = X.length();
        X.normalize();
        Y = vectorialProduct(dir, X);
        pij[0][j] = new ControlPoint(pj[j]);
        P0 = new Point3d(pj[j]);
        T0 = new Vector3d(Y)
        int index = 0;
        for (int i=1; i<=narcs; i++) {
            P2 = O + r*cos[i]*X + r*sin[i]*Y
            pij[index + 2][j] = new ControlPoint(P2, pj[j].w);
            T2 = -sin[i]*X + cos[i]*Y
            CurveCreator.lineIntersect3D(P0, T0, P2, T2, tmp);
            pij[index + 1][j] = new ControlPoint(tmp, wm*pj[j].w);
            index += 2;
            if (i < narcs) {
                P0 = new Point3d(P2);
                T0 = new Point3d(T2);
            }
        }
    }
    return new NurbsSurface(pij, uKnot, curve.getKnots(), 2, curve.getDegree());
}
  
```

El algoritmo anterior se ha simplificado para que sea más fácil de entender, lógicamente las sumas o multiplicaciones de puntos en 3D o vectores hay que sustituirlas por las llamadas a sus funciones correspondientes.

En la figura 2.46 se muestra la operación de revolución de una curva NURBS alrededor del eje Z, utilizando el algoritmo anterior.

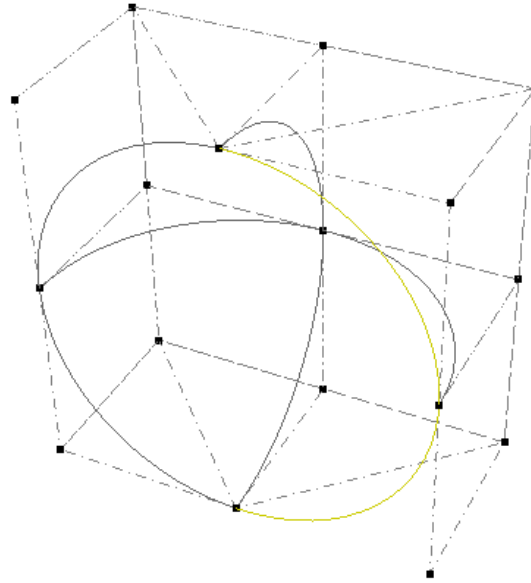


Figura 2.46: Revolución de una curva NURBS

2.9.4 Skinned

La operación 'skinned' permite crear una superficie NURBS que pasa por un conjunto de curvas $C_1(u), C_2(u), \dots, C_n(u)$. El primer paso es hacer que las curvas de entrada sean compatibles, para ello hay que igualar el grado de las curvas y los nodos.

Algoritmo 2.25 Generar curvas compatibles

```

public static NurbsCurve[] generateCompatibleCurves(NurbsCurve[] crvs) {
    int maxDeg = 0;
    int numCrvs = crvs.length;
    for (int i=0; i<numCrvs; ++i) {
        if (crvs[i].getDegree() > maxDeg) {
            maxDeg = crvs[i].getDegree();
        }
    }
    //Iguala todas las curvas al grado máximo
    ArrayList<NurbsCurve> nCrvs = new ArrayList();
    for (int i=0; i<numCrvs; ++i) {
        nCrvs.add((NurbsCurve)CurveUtils.elevateToDegree(crvs[i], maxDeg));
    }
    //Combina todos los vectores de nodos en un único vector
    double[] U = nCrvs.get(0).getKnots();
    for (int i=1; i<numCrvs; ++i) {
        U = CurveUtils.knotVectorUnion(U, nCrvs.get(i).getKnots());
    }
    //Combina el vector anterior con el vector de nodos de cada curva
    NurbsCurve[] outCrvs = new NurbsCurve[numCrvs];
    for (int i=0; i<numCrvs; ++i) {
        NurbsCurve ncrv = CurveUtils.mergeKnotVector(nCrvs.get(i), U);
        outCrvs[i] = ncrv;
    }
    return outCrvs;
}

```

Una vez que se ha obtenido un conjunto de curvas compatibles, el siguiente algoritmo se encarga de generar la superficie NURBS que pasa por todas las curvas. El procedimiento es el siguiente:

1. Se llama al algoritmo anterior para generar un conjunto de curvas compatibles, se crea una matriz con los puntos de control de cada curva colocada en la dirección u .
2. Como se han igualado los vectores de nodos, se coge el de la primera curva como vector de nodos de la superficie en la dirección u .
3. Se generan los nodos en la dirección v calculando la distancia entre un punto de control de cada curva, se hace la media con el resto (Chord-length averaging).
4. Se interpolan los puntos de control en la dirección v con el nuevo vector de nodos.
5. Se genera la superficie con la matriz de puntos de control interpolados y los vectores de nodos u y v .

Algoritmo 2.26 Algoritmo para generar la superficie 'Skinned'

```

public static NurbsSurface createSkinnedSurface(NurbsCurve[] crvs, int tDegree) {
    int numCrvs = crvs.length;
    //Primero hay que crear una lista de curvas NURBS compatibles
    NurbsCurve[] newCrvs = CurveUtils.generateCompatibleCurves(crvs);

    //Se extrae el vector de nodos S de una curva (es común para todas)
    KnotVector sKnots = newCrvs[0].getKnotVector();

    //Se crea la matriz de puntos de control inicial
    ControlPoint[][] crvCParr = new ControlPoint[numCrvs][];
    for (int i = 0; i < numCrvs; ++i) {
        NurbsCurve crv = newCrvs[i];
        crvCParr[i] = crv.getControlPoints();
    }

    //Hay que crear el vector de nodos en la dirección T usando el algoritmo chord-length
    KnotVector tKnots = CurveUtils.buildKnotVector(newCrvs, tDegree);

    //Se crea la matriz de puntos de control de la superficie
    int numCPs = crvCParr[0].length;
    ControlPoint[][] newCPLst = new ControlPoint[numCPs][];
    for (int i = 0; i < numCPs; ++i) {
        //Se crea un vector de puntos de control en la dirección T
        ControlPoint[] pnts = new ControlPoint[numCrvs];
        for (int k = 0; k < numCrvs; ++k) {
            pnts[k] = crvCParr[k][i];
        }
        //Se crea una curva interpolada con esos puntos
        NurbsCurve iCrv = CurveCreator.fitControlPoints(tDegree, pnts, tParams, tKnots);
        ControlPoint[] newCPs = iCrv.getControlPoints();
        newCPLst[i] = newCPs;
    }
    //Se crea la superficie
    return new NurbsSurface(newCPLst, sKnots, tKnots);
}

```

En la figura 2.47 se puede observar el resultado de la operación 'skinned', la superficie NURBS resultante pasa por todas las curvas de entrada.

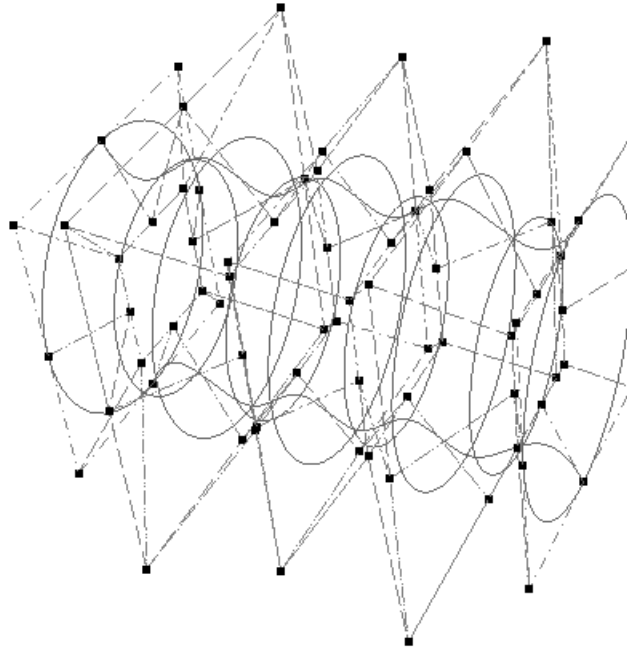


Figura 2.47: Superficie 'skinned' formada por círculos con distinto radio

2.9.5 Coons

La operación 'coons' consiste en generar una superficie a partir del borde, definido por cuatro curvas NURBS conectadas $C_1^k(u)$, $C_2^k(u)$, $C_1^l(v)$, $C_2^l(v)$. Las curvas que forman la dirección u de la superficie y las que forman la dirección v tienen que ser compatibles entre sí. El parche se genera mediante interpolación bilinear.

$$S(u, v) = S_1(u, v) + S_2(u, v) - T(u, v) \quad (2.27)$$

donde $S_1(u, v)$ y $S_2(u, v)$ son las superficies generadas mediante la operación 'skinned' entre las curvas C^k y C^l respectivamente, y $T(u, v)$ es el parche bilinear formado por las cuatro esquinas de la superficie.

$$T(u, v) = \begin{bmatrix} 1 & u \end{bmatrix} \begin{bmatrix} S_{0,0} & S_{0,1} \\ S_{1,0} & S_{1,1} \end{bmatrix} \begin{bmatrix} 1 \\ v \end{bmatrix} \quad (2.28)$$

La operación 'skinned' implementada permite que las curvas de entrada sean diferentes (con orden, nº de puntos de control o vectores de nodos distintos), ya que las adapta utilizando el algoritmo 2.25. Esto permite hacer la operación 'coons' más general que usando 'ruled surfaces' para crear S_1 y S_2 . En el algoritmo 2.27 se muestra un resumen del proceso en detalle.

Algoritmo 2.27 Algoritmo para generar la superficie 'Coons'

```

public static NurbsSurface createCoonsPatch(NurbsCurve s0, NurbsCurve t0,
                                           NurbsCurve s1, NurbsCurve t1) {
    //Superficie skinned intermedia en la dirección S
    NurbsCurve[] sCrvs = new NurbsCurve[]{t0, t1};
    NurbsSurface Ps = createSkinnedSurface(sCrvs);

    //Superficie skinned intermedia en la dirección T
    NurbsCurve[] tCrvs = new NurbsCurve[]{s0, s1};
    NurbsSurface Pt = createSkinnedSurface(tCrvs);

    //Parche bilinear con las cuatro esquinas
    NurbsSurface PsPt = (NurbsSurface) NurbsCreator.createFacet(t00, t01, t10, t11);

    //Elevamos todas las superficies al mismo grado en las direcciones S y T
    PsPt = (NurbsSurface) SurfaceUtils.elevateDegreeTo(PsPt, maxDegU, maxDegV);
    Ps = (NurbsSurface) SurfaceUtils.elevateDegreeTo(Ps, maxDegU, maxDegV);
    Pt = (NurbsSurface) SurfaceUtils.elevateDegreeTo(Pt, maxDegU, maxDegV);

    //Combina todos los vectores de nodos en un vector para cada dirección Us y Ut
    double[] Us = PsPt.getUKnots();
    double[] Ui = Ps.getUKnots();
    Us = CurveUtils.knotVectorUnion(Us, Ui);
    Ui = Pt.getUKnots();
    Us = CurveUtils.knotVectorUnion(Us, Ui);
    double[] Ut = PsPt.getVKnots();
    Ui = Ps.getVKnots();
    Ut = CurveUtils.knotVectorUnion(Ut, Ui);
    Ui = Pt.getVKnots();
    Ut = CurveUtils.knotVectorUnion(Ut, Ui);

    //Combina los vectores anteriores con los de cada superficie
    PsPt = SurfaceUtils.mergeUKnotVector(PsPt, Us);
    PsPt = SurfaceUtils.mergeVKnotVector(PsPt, Ut);
    Ps = SurfaceUtils.mergeUKnotVector(Ps, Us);
    Ps = SurfaceUtils.mergeVKnotVector(Ps, Ut);
    Pt = SurfaceUtils.mergeUKnotVector(Pt, Us);
    Pt = SurfaceUtils.mergeVKnotVector(Pt, Ut);

    //Se construye la nueva matriz de puntos de control donde:
    // Puntos:  $P = Ps + Pt - PsPt$ 
    // Pesos:  $w = Ws * Wt$ 
    int cols = Ps.uLength();
    int rows = Ps.vLength();
    ControlPoint4d[][] cpNet = new ControlPoint4d[cols][rows];
    ControlPoint[][] cpNetPs = Ps.getControlPoints();
    ControlPoint[][] cpNetPt = Pt.getControlPoints();
    ControlPoint[][] cpNetPsPt = PsPt.getControlPoints();
    for (int t=0; t<cols; t++) {
        for (int s=0; s<rows; s++) {
            ControlPoint cpPs = cpNetPs[t][s];
            ControlPoint cpPt = cpNetPt[t][s];
            ControlPoint cpPsPt = cpNetPsPt[t][s];
            ControlPoint out = new ControlPoint();
            out.add(cpPs);
            out.add(cpPt);
            out.sub(cpPsPt);
            out.w = (cpPs.w * cpPt.w);
            cpNet[t][s] = out;
        }
    }
    //Se crea la superficie
    return new NurbsSurface(cpNet, Ps.getUKnots(), Ps.getVKnots());
}

```

En la siguiente figura se puede observar el resultado de la operación 'coons', la superficie NURBS resultante tiene como borde las cuatro curvas de entrada.

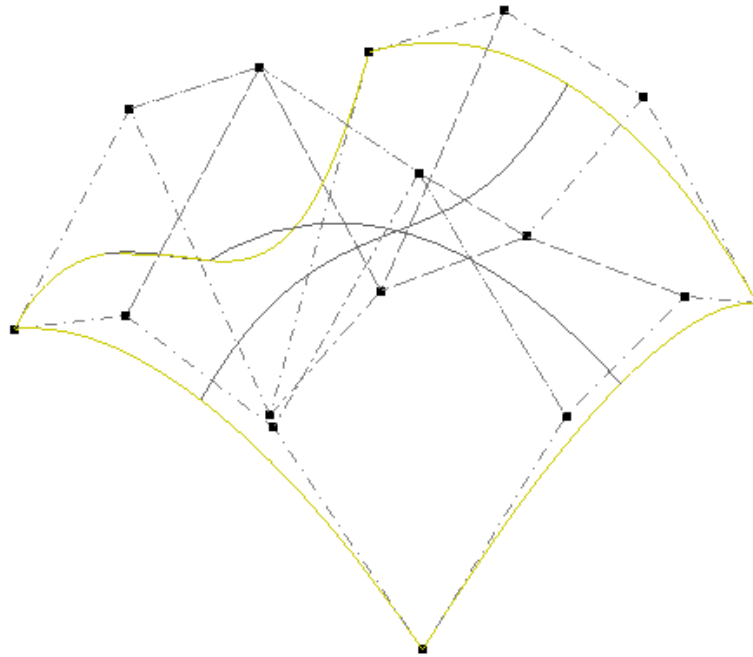


Figura 2.48: Superficie 'coons' formada por cuatro curvas de borde

Bibliografía

- [1] “Java 3D API.” [Online]. Disponible en: <http://www.java3d.org/index.html>
- [2] “Java 3D API Documentation.” [Online]. Disponible en: <http://download.java.net/media/java3d/javadoc/1.5.0/>
- [3] A. Davison, *Killer Game Programming in Java*. O’Reilly Media, 2009.
- [4] “JavaFX 3D Graphics.” [Online]. Disponible en: <https://docs.oracle.com/javase/8/javafx/graphics-tutorial/overview-3d.htm>
- [5] “JOGL Project.” [Online]. Disponible en: <https://jogamp.org/jogl/www/>
- [6] D. Selman, *Java 3D Programming*. Manning Publications, 2002.
- [7] J. J. Pratdepadua, *Programación en 3D con Java 3D*. RA-MA, 2003.
- [8] G. E. Farin, *Curves and Surfaces for CAGD: A Practical Guide*. Morgan Kaufmann, 2002.
- [9] W. T. Les A. Piegl, *The NURBS Book (Monographs in Visual Communication)*. Springer, 2013.
- [10] C. de Boor, “Subroutine package for calculating with B-splines,” Los Alamos Scientific Lab., N. Mex., Tech. Rep., 1970.

-
- [11] “JGeom - The Java geometry graphics library using NURBS.” [Online]. Disponible en: <https://sourceforge.net/projects/jgeom/>
- [12] “iGeo - Java 3D Modeling Library.” [Online]. Disponible en: <https://github.com/sghr/iGeo>
- [13] “GeomSS - Geometry modeling and scripting system in Java.” [Online]. Disponible en: <https://sourceforge.net/projects/geomss/>
- [14] J. Foley, *Computer graphics: principles and practice*. Addison-Wesley, 1996.
- [15] M. W. Dave Shreiner, *OpenGL Programming Guide: the official guide to learning OpenGL, 6th Edition*. Addison-Wesley, 2007.
- [16] S. A. T. William H. Press, *Numerical Recipes: The Art of Scientific Computing, Third Edition*. Cambridge University Press, 2007.

Capítulo 3

Interfaz de Usuario: Diseño 3D

3.1 Introducción

En el siguiente capítulo se describe la parte de diseño 3D de la interfaz de usuario que trabaja con el motor gráfico propuesto en el capítulo anterior. Se ha pretendido que sea sencillo crear o editar un modelo geométrico complejo y que el usuario se sienta cómodo utilizando el programa. El código de la interfaz de usuario ha sido diseñado de forma que sea sencillo de mantener y ampliar con futuras características.

Como en la mayoría de programas de diseño 3D, la creación de superficies NURBS se realiza de forma incremental, permitiendo utilizar puntos y curvas auxiliares para construir superficies complejas. Además, el usuario puede utilizar el plano de referencia como un sistema de coordenadas local, capturar puntos de la geometría, ocultar parte del modelo mediante capas, deshacer y rehacer las operaciones, definir geometría paramétrica y muchas más características que se describen a continuación.

En la primera parte se muestra la ventana principal de la interfaz de usuario y el panel 3D donde se dibuja la geometría. Aquí se verá la integración de las curvas y superficies NURBS del motor gráfico con Java 3D.

En la segunda parte se describe cómo se han implementado los mecanismos de interacción del usuario con la geometría: panel de referencia, geometría auxiliar, selección de objetos, consola de comandos, pick, etc.

En la última parte se describen los ficheros CAD más importantes que se han implementado para permitir importar y exportar la geometría a otros programas.

Una vez descrita la parte de diseño 3D, en el siguiente capítulo se describirá la parte de simulación electromagnética y visualización de resultados de la interfaz de usuario.

3.2 Características Principales

La interfaz de usuario, a partir de ahora GUI, es un programa de modelado geométrico para el análisis electromagnético. El programa se compone de dos grandes bloques:

- **Modelado Geométrico:** Es un programa de diseño 3D que contiene todas las operaciones necesarias para trabajar con curvas y superficies NURBS.
- **Simulación Electromagnética:** Contiene distintos módulos que permiten simular diversos tipos de problemas electromagnéticos. Aunque la simulación se realiza por una serie de núcleos externos, es transparente al usuario ya que cada módulo permite configurar los parámetros, lanzar la simulación y visualizar los resultados.

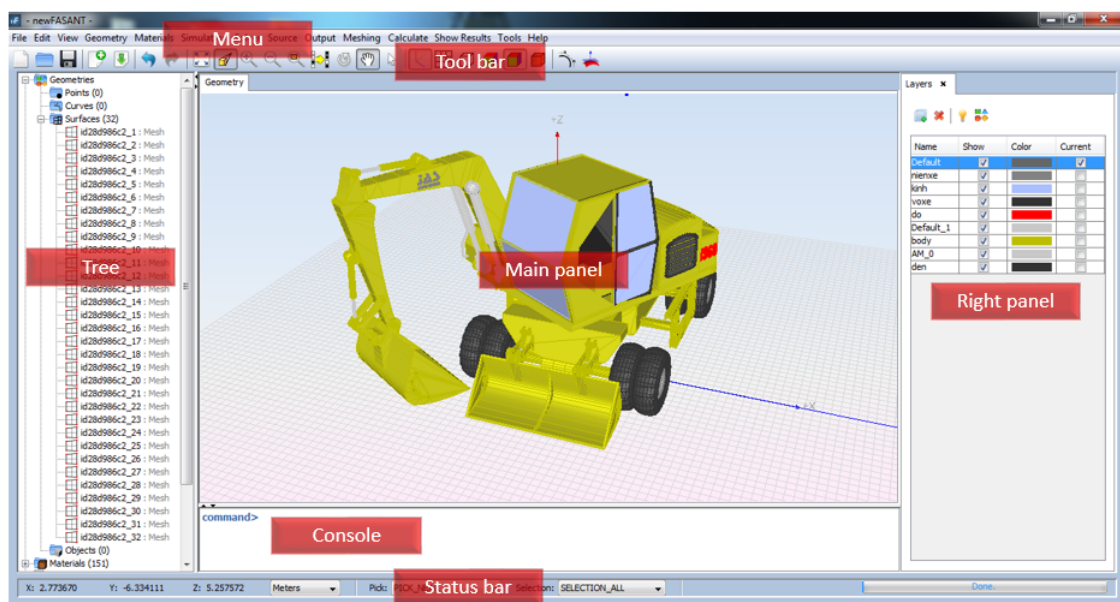


Figura 3.1: Interfaz de Usuario

En la figura 3.1 se muestra la ventana principal de la interfaz de usuario, se compone de los siguientes elementos:

- **Panel principal.** La pestaña 'Geometry' contiene el Canvas de Java3D que utiliza el motor gráfico del capítulo anterior para dibujar la geometría. En este panel también se muestran otras pestañas como el mallado de la geometría o los resultados de simulación.
- **Consola.** En la consola se escriben los distintos comandos para crear y modificar la geometría. Se ha decidido utilizar una consola porque proporciona varias ventajas: cuando se conocen los comandos es muy rápido de usar, permite utilizar el ratón en el panel principal para seleccionar puntos o construir geometría y se pueden importar y exportar *scripts*. Para que sea más sencilla de utilizar todos los comandos están accesibles desde el menú de geometría.

- **Panel derecho.** Aquí se muestran las distintas pestañas con las que interacciona el usuario. Con esto se evita, en la manera de lo posible, que aparezcan nuevas ventanas con lo que todo es accesible desde la ventana principal. Las distintas pestañas se abren desde el menú principal y permiten configurar los parámetros de simulación y otras opciones del programa.
- **Menú.** Desde el menú se acceden a las distintas opciones del programa. Contiene una serie de elementos fijos que son comunes a todos los módulos y otros elementos que cambian según el módulo de simulación.
- **Árbol.** Permite visualizar rápidamente la geometría y los parámetros de simulación del proyecto actual. Además de visualizar, permite seleccionar objetos y modificar varios parámetros de simulación.
- **Barra de herramientas.** Proporciona acceso rápido a varios elementos del menú que se utilizan habitualmente.
- **Barra de estado.** Proporciona información de las coordenadas del ratón sobre el panel de geometría y contiene una barra de progreso para ver el estado de operaciones que requieren mayor tiempo de ejecución. También permite seleccionar las unidades de medida, el modo de pick y de selección activos.

3.3 Estructura del programa

La interfaz de usuario es un programa en continuo crecimiento, actualmente contiene más de 2000 ficheros de código. La estructura del programa es muy importante para su posterior mantenimiento y para que su crecimiento sea ordenado.

La organización de un programa de este tamaño requiere una planificación exhaustiva y el conocimiento de todos los requisitos para dar con una solución que realmente funcione.

Por suerte, antes de comenzar el desarrollo de la interfaz de usuario propuesta en esta tesis, trabajé durante dos años en el mantenimiento de la interfaz anterior lo que me permitió conocer todos los requisitos del programa y los problemas asociados a un crecimiento sin una planificación definida.

En esta interfaz, se ha intentado dar solución a los problemas anteriores de mantenimiento y ampliación del programa, creando una estructura formada por componentes sencillos, a esta técnica se la conoce popularmente como *divide y vencerás*¹.

En la figura 3.2 se puede observar la estructura principal de los distintos paquetes software del programa.

¹Divide y vencerás hace referencia a un refrán que implica resolver un problema difícil, dividiéndolo en partes más simples tantas veces como sea necesario, hasta que la resolución de las partes se torna obvia.

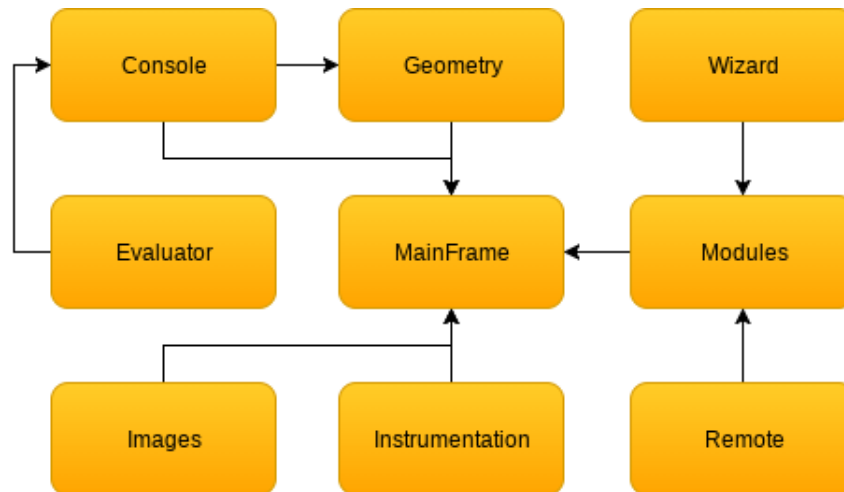


Figura 3.2: Estructura del programa

A continuación se describe el propósito de cada elemento, que será analizado en detalle en las sucesivas secciones de este capítulo.

- **MainFrame.** En este paquete se encuentra la clase principal del programa llamada 'MainFrame', se encarga de dibujar la ventana principal (figura 3.1). Desde esta clase se puede acceder a todos los componentes de la aplicación. En el paquete también se encuentran las clases para crear los distintos componentes: menús, pestañas, árbol, etc.
- **Geometry.** En este paquete se encuentra el motor gráfico y todo lo relacionado con geometría: objetos, primitivas, ficheros geométricos, selección, movimiento de la cámara...
- **Console.** Este paquete contiene la consola, que es una interfaz que permite ejecutar comandos para construir geometría. La consola también guarda el historial y permite realizar construcciones geométricas utilizando parámetros variables.
- **Evaluator.** Contiene un evaluador de expresiones matemáticas. Se utiliza en la consola ya que en algunos comandos y en construcciones paramétricas se pueden utilizar funciones.
- **Instrumentation.** Es un módulo que se utiliza para recolectar datos de los errores producidos en el programa y enviarlos a un servidor, si el usuario lo autoriza.
- **Images.** Contiene todas las imágenes que se utilizan en la aplicación como iconos y primitivas.
- **Modules.** En este paquete se encuentran los distintos módulos de simulación electromagnética. Todos los módulos tienen una estructura similar. También hay un conjunto de utilidades de cosas comunes a los distintos módulos como pueden ser las ventanas de resultados.

- **Remote.** Permite la comunicación entre un cliente y un servidor de la interfaz. Es posible arrancar la aplicación sin mostrar la ventana principal en modo servidor y que otra aplicación en modo cliente se conecte a ella para realizar simulaciones.
- **Wizard.** Contiene un asistente de proyectos con distintos ejemplos de cada módulo. Sirve para mostrar al usuario distintos tipos de simulaciones o para configurar automáticamente los parámetros de simulación que mejor se ajustan a cada proyecto.

3.4 Java Swing

El paquete Swing [1] es parte de la JFC (Java Foundation Classes) en la plataforma Java. La JFC provee facilidades para ayudar a la gente a construir GUIs. Swing abarca componentes como botones, tablas, marcos, etc... Los componentes Swing se identifican porque pertenecen al paquete `javax.swing`.

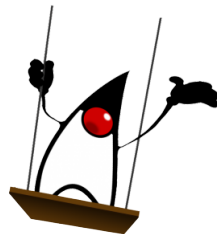


Figura 3.3: Java Swing

Swing existe desde la versión JDK 1.1. Antes de la existencia de Swing, las interfaces gráficas de usuario se realizaban a través de AWT [2] (Abstract Window Toolkit), de quien Swing hereda todo el manejo de eventos. Usualmente, para todo componente AWT existe un componente Swing que lo reemplaza, por ejemplo, la clase 'Button' de AWT es reemplazada por la clase 'JButton' de Swing (el nombre de todos los componentes Swing comienza con "J").

Los componentes de Swing utilizan la infraestructura de AWT, incluyendo el modelo de eventos, el cual controla cómo un componente reacciona a distintos eventos como, eventos de teclado, mouse, etc... Es por esto, que la mayoría de las aplicaciones con Swing necesitan importar dos paquetes AWT: `java.awt.*` y `java.awt.event.*`.

Actualmente las dos bibliotecas que se utilizan para diseñar interfaces gráficas en Java son Swing y SWT [3]. Para el desarrollo de la interfaz de usuario se ha elegido Swing por las siguientes razones:

- Independencia de plataforma. Swing es Java puro, por lo cual es realmente portable.
- El desarrollo de componentes Swing es más activo y hay más documentación.
- La integración con Java 3D y el motor gráfico es sencilla.

- Permite adaptarse a los diferentes sistemas operativos cambiando la apariencia “Look and Feel”.
- El trabajo con SWT esta muy amarrado al IDE Eclipse [4].

3.4.1 Características de una aplicación Swing

Una aplicación Swing se construye mezclando componentes con las siguientes reglas.

1. Debe existir, al menos, un contenedor de alto nivel (Top-Level Container), que provee el soporte que los componentes Swing necesitan para el pintado y el manejo de eventos.
2. El resto de componentes cuelgan del contenedor de alto nivel (estos pueden ser contenedores o componentes simples).

En la interfaz, el contenedor de alto nivel es la clase 'MainFrame' que extiende de JFrame. Un JFrame es una ventana independiente con marco y barra de título y es el componente principal de la aplicación.

Una vez que se ha creado un contenedor, se pueden agregar a este otros contenedores o componentes simples (etiquetas, botones, cuadros de texto, etc.) creando una estructura en forma de árbol.

Para determinar cómo se organizan visualmente los componentes dentro de un contenedor existen los *Layouts*. Los más utilizados son: GroupLayout, BorderLayout, GridLayout, FlowLayout y BoxLayout.

Cada vez que el usuario interactúa con la aplicación se ejecuta un evento. Para que un componente determinado reaccione frente a un evento, debe poseer un "lanzador" con, al menos, un método determinado que se ejecutará al escuchar un evento en particular, como por ejemplo al hacer click con el ratón en un JButton. En la siguiente tabla se muestran los más comunes.

<i>Acción que dispara un evento</i>	<i>Tipo de evento</i>
El usuario hace un click, presiona Return o selecciona un menú.	ActionListener
El usuario escoge un frame (ventana principal).	WindowListener
El usuario hace un click sobre un componente.	MouseListener
El usuario pasa el ratón sobre un componente.	MouseMotionListener
Un componente se hace visible.	ComponentListener
Un componente adquiere el foco del teclado.	FocusListener
Cambia la selección en una lista o tabla.	ListSelectionListener

Tabla 3.1: Tipos de eventos en Swing

3.4.2 Componentes principales de Swing

A continuación se describen los componentes principales de Swing que se han utilizado para construir la interfaz.

Contenedores:

- **JFrame:** Es el contenedor principal de la aplicación.
- **JDialog:** Es una ventana independiente que se muestra sobre la aplicación. En el desarrollo de esta interfaz se ha intentado minimizar su uso.
- **JPanel:** Permite la creación de paneles independientes donde se almacenan otros componentes. La mayoría de ventanas de la interfaz se crean utilizando un panel que se integra en la ventana principal con una pestaña.
- **JSplitPane:** Es un contenedor dividido en dos secciones que se pueden redimensionar. En la figura 3.1 se puede observar que el árbol, la consola y el panel derecho se separan mediante este contenedor.
- **JTabbedPane:** Permite la creación de pestañas, cada pestaña representa un contenedor independiente. Para las pestañas de la interfaz se ha creado un componente que extiende de este contenedor.
- **JScrollPane:** Se utiliza para crear barras de desplazamiento en un panel.
- **JToolBar:** Es un panel especial para crear barras de herramientas.

Componentes simples:

- **JLabel:** Es una etiqueta, permite insertar texto o una imagen.
- **JButton:** Es un botón, normalmente se utiliza un ActionListener para ejecutar una acción cuando el usuario hace click en él.
- **JTextField:** Cuadro de texto, permite recoger datos del usuario.
- **JCheckBox:** Casilla de verificación, se utiliza para activar y desactivar opciones.
- **JRadioButton:** Botones de radio, se utilizan para seleccionar una opción entre varias.
- **JComboBox:** Es una lista desplegable, se utiliza igual que JRadioButton pero ahorra espacio cuando hay muchas opciones.
- **JTextArea:** Es un área de texto, se utiliza para escribir o mostrar varias líneas, como en la consola de la aplicación.

Componentes avanzados:

- **JList:** Permite cargar una lista de elementos, dependiendo de las propiedades puede tenerse una lista de selección múltiple.
- **JTable:** Permite vincular una tabla de datos con sus respectivas filas y columnas.
- **JTree:** Carga un árbol donde se establece cierta jerarquía visual, tipo directorio.
- **JFileChooser:** Permite la búsqueda de un fichero en el sistema.

3.4.3 Netbeans IDE y Swing

Netbeans IDE [5] es un entorno de desarrollo gratuito y de código abierto. Permite el uso de un amplio rango de tecnologías de desarrollo tanto para escritorio, como aplicaciones web, o para dispositivos móviles. Da soporte a las siguientes tecnologías, entre otras: Java, PHP, Groovy, C/C++, HTML5,... Además puede instalarse en la mayoría de sistemas operativos.

El uso de un entorno de desarrollo como Netbeans simplifica la gestión de grandes proyectos con el uso de diferentes vistas, asistentes de ayuda, y estructurando la visualización de manera ordenada, lo que ayuda en el trabajo diario. Por ejemplo, cuando se abre una clase Java, se mostrarán distintas ventanas con el código, su localización en el proyecto, una lista de los métodos y propiedades, las jerarquías que tiene nuestra clase y otras muchas opciones.

Una de las principales características que ofrece NetBeans es la creación de ventanas utilizando la librería Swing por medio de una interfaz gráfica.

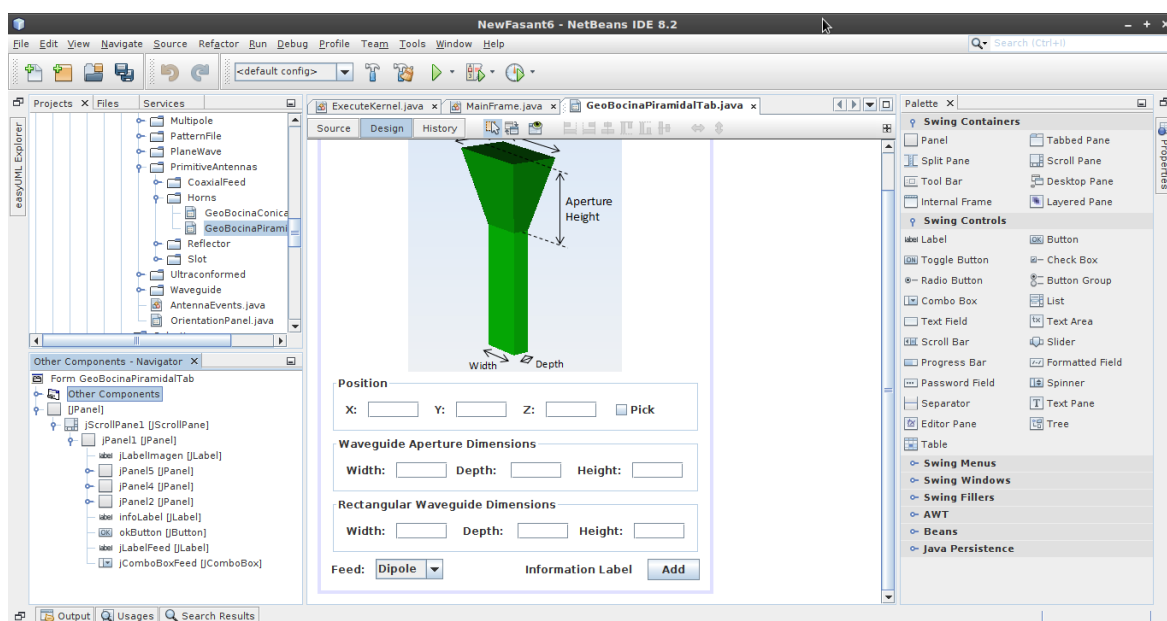


Figura 3.4: Netbeans IDE: Diseñador de ventanas con Swing

En el diseñador de ventanas, se pueden arrastrar los distintos componentes de Swing y asignar sus propiedades. Netbeans utiliza una parte del fichero Java para generar automáticamente el código Swing necesario para dibujar la ventana.

La mayoría de paneles de la interfaz de usuario se han creado utilizando el diseñador de ventanas de Netbeans, lo que ahorra mucho esfuerzo.

3.5 La ventana principal: MainFrame

En la sección anterior se han visto los componentes principales de la librería Swing y la posibilidad de utilizar el diseñador de ventanas de Netbeans para crear los distintos paneles de la interfaz.

Para crear la ventana principal de la aplicación 'MainFrame' se utilizó el diseñador de ventanas para definir la estructura inicial, posteriormente se copió el código generado a una clase distinta y se adaptó para incluir todos los componentes y contenedores necesarios; ya que debido a su complejidad, era muy difícil de generar y mantener utilizando el diseñador.

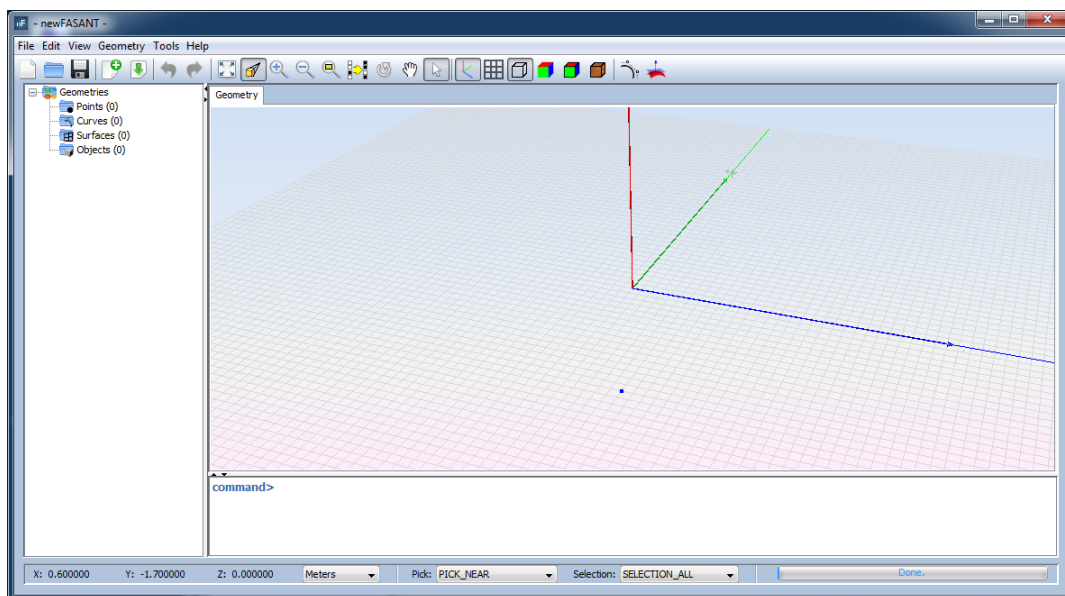


Figura 3.5: Ventana principal: MainFrame

La clase principal 'MainFrame' extiende de la clase Swing 'JFrame'. La mayoría de componentes que incluye también son clases que extienden de las clases básicas de Swing. Esta manera de trabajar es muy importante ya que permite encapsular el código del componente dentro de su clase en vez de definir sus propiedades en la clase que crea la ventana. Así, la clase 'MainFrame' solo se encarga de crear los componentes y distribuirlos dentro de la ventana y el código es mucho más sencillo. En nuestro caso, todo el código necesario para crear la ventana de la figura 3.5 ocupa 150 líneas (la clase principal de la versión anterior tenía 28.021 líneas).

En el esquema de la figura 3.6 se puede ver como se distribuyen los distintos componentes Swing dentro de la ventana principal.

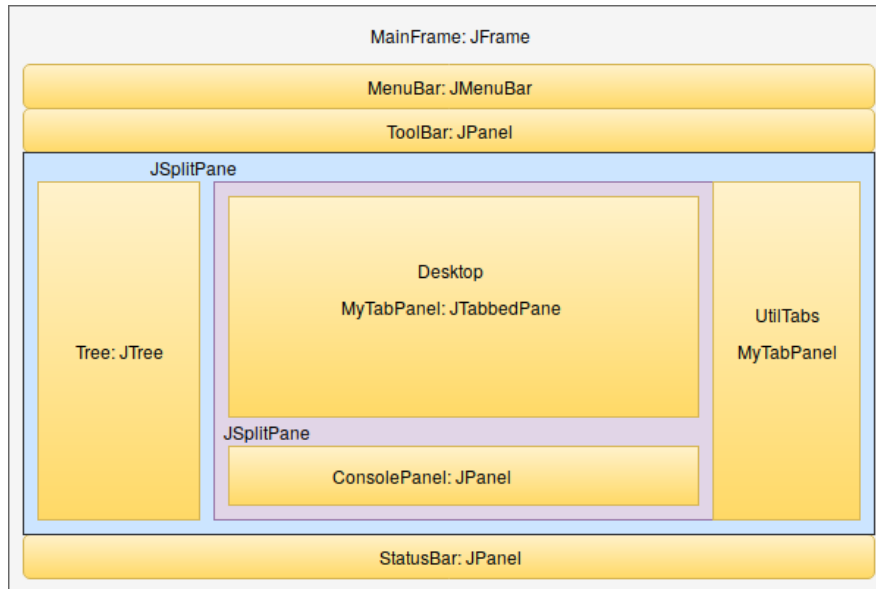


Figura 3.6: Componentes Swing de la ventana principal

En el esquema se observa el panel 'UtilTabs' que no aparece en la figura 3.5; esto es debido a que el panel está oculto cuando no hay ninguna pestaña abierta. Se puede ver el panel con una pestaña abierta en la figura 3.1, corresponde al panel derecho de la interfaz.

La clase 'MainFrame' tiene como atributo un objeto de la clase 'Project', que contiene los datos del módulo de simulación abierto. La pestaña de Geometría es un objeto de la clase 'Panel3D'. La mayoría de los componentes guardan una referencia del objeto 'MainFrame', esto permite la interacción entre los distintos componentes; por ejemplo, si al ejecutar un comando en la consola se dibuja algo en el panel de geometría, la consola accederá al objeto de la clase 'Panel3D' a través de la clase 'MainFrame'.

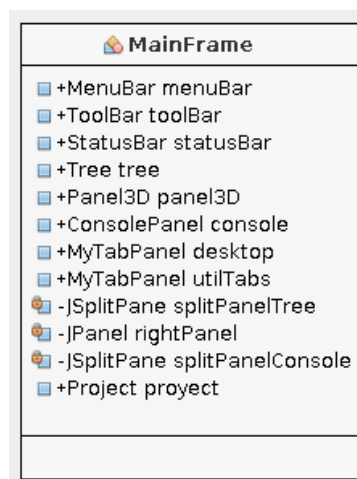


Figura 3.7: Atributos de la clase 'MainFrame'

3.6 Panel de Geometría: Panel3D

En el capítulo 2 se ha descrito el desarrollo del motor gráfico de la aplicación. En esta sección se muestra la integración con el resto de la interfaz para crear el panel de geometría que se ve en la figura 3.5.

El propósito de esta sección es mostrar el diseño de los distintos componentes del panel de geometría, como los tipos de objetos, las capas, el movimiento de la cámara, selección, etc. En la siguiente sección, se describe como interactúa el usuario con el panel de geometría a través de la consola de comandos.

La clase 'Panel3D' se encarga de dibujar el panel de la pestaña *Geometry* de la interfaz. Esta clase extiende de JPanel de Swing y contiene el Canvas3D de Java3D descrito en el apartado 2.2.3.

Para dibujar un escenario en Java3D [6] se necesitan principalmente tres elementos:

- **Geometría:** Son los objetos que se quieren visualizar, generalmente son objetos de la clase Shape3D que cuelgan de un BranchGroup de Java3D.
- **Iluminación:** Para que los objetos geométricos sean visibles necesitan ser iluminados. La apariencia de los objetos depende del material del objeto y de cómo es iluminado.
- **Vista:** La vista es la que define que parte del escenario 3D se muestra en la pantalla. Se crea a partir del objeto SimpleUniverse de Java3D y es controlada por la cámara.

En la sección 2.2.2 se puede ver que un universo virtual de Java3D es definido por un grafo, conocido como Scene Graph. Un esquema del árbol de Java3D generado por la clase 'Panel3D' sería el siguiente.

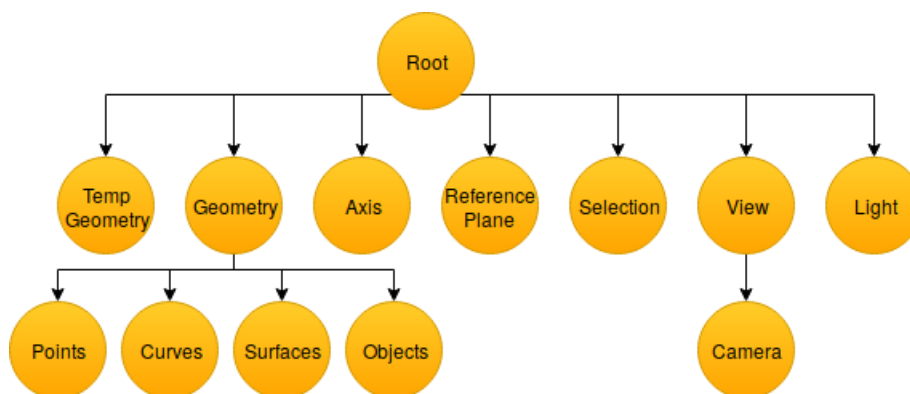


Figura 3.8: Estructura del árbol de Java3D en la interfaz

Una vez que el árbol es compilado y añadido al *Universo* de Java3D, aparecerán dibujados los objetos dentro del *Canvas* [7] que contiene 'Panel3D'. Posteriormente, es posible añadir o modificar elementos del árbol.

3.6.1 Objetos Geométricos

Los objetos que se han creado para mostrar la geometría del motor gráfico en el panel 3D comparten la misma interfaz. Esto permite tener una serie de métodos comunes a todos ellos y simplificar el código, la interfaz también permite saber de qué tipo de objeto se trata para acceder a los atributos y funciones específicas de cada objeto.

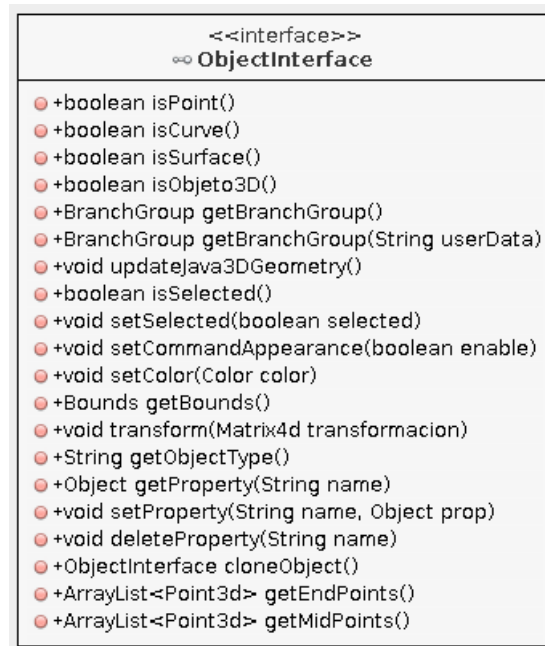


Figura 3.9: Interfaz 'ObjectInterface'

La aplicación permite dibujar cuatro tipos de objetos geométricos: puntos, curvas, superficies y objetos. Los objetos sirven para agrupar un conjunto de superficies y realizar operaciones de manera más sencilla.

Los cuatro tipos de objetos comparten una estructura similar, lo más importante es que extienden de la clase Shape3D de Java3D e implementan 'ObjectInterface'. Esto permite almacenarlos directamente en los nodos correspondientes del árbol de Java3D y se puede trabajar con ellos de una forma sencilla.

Los objetos también almacenan la apariencia, que es un objeto de tipo Appearance de Java3D que indica la forma en la que se dibuja dentro del panel, y un HashMap que se utiliza para guardar las distintas propiedades del objeto como, por ejemplo, el nombre.

Otro elemento común a los cuatro objetos geométricos es que implementan la interfaz Externalizable de Java. Esto permite ser almacenados en un fichero binario y recuperarlos posteriormente. Los objetos geométricos serán almacenados junto a los datos del proyecto de simulación, como se verá más adelante.

3.6.1.1 PointShape

La clase 'PointShape' permite dibujar un punto en el panel 3D. Los puntos se usan como geometría auxiliar, sirven para marcar coordenadas (x, y, z) dentro del espacio. Posteriormente, esas coordenadas pueden ser seleccionadas para realizar operaciones o construir geometrías más complejas.

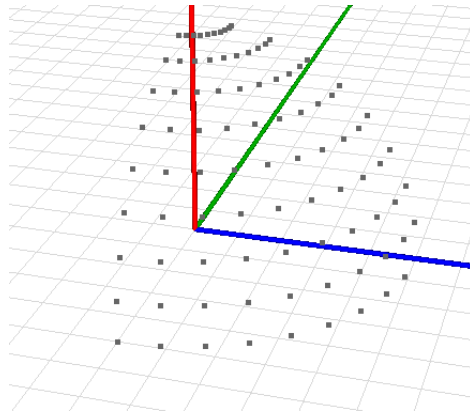


Figura 3.10: Objetos de tipo 'PointShape'

El objeto 'PointShape' se encarga de generar la apariencia y la geometría necesaria para construir el Shape3D. Al ser un punto, se almacenan las coordenadas dentro del objeto.

El método *updateJava3DGeometry* se encarga de construir la geometría, en este caso se utiliza un objeto de tipo *PointArray* de Java 3D (Sección 2.5.1).

Es importante destacar que el objeto *PointShape* no cuelga directamente del nodo *Points* de la figura 3.8, sino que se inserta primero en un *BranchGroup* que contiene el propio objeto; esto es necesario si se quiere eliminar el objeto posteriormente del árbol.

Otra característica es que un objeto, como 'PointShape', puede ser seleccionado. Para que esto sea posible, el objeto guarda la apariencia que adopta el punto al ser seleccionado, en nuestro caso sería una apariencia de color amarillo y un grosor mayor. El método *setSelected* se encarga de cambiar la apariencia del Shape3D y será invocado en el proceso de selección.

3.6.1.2 CurveShape

La clase 'CurveShape' permite dibujar una curva NURBS en el panel 3D. Las curvas NURBS tampoco se usan en las simulaciones y sirven como geometría auxiliar para construir superficies complejas.

La clase 'CurveShape' contiene un objeto de tipo 'NurbsCurve', sección 2.4.3, que define una curva NURBS en el motor gráfico. Utilizando el renderizado de curvas, algoritmo 2.2, crea la geometría del Shape3D necesaria para que pueda ser visualizada en el panel.

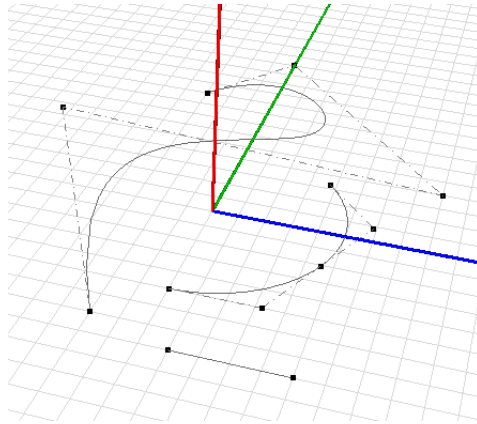


Figura 3.11: Objetos de tipo 'CurveShape'

El funcionamiento es similar al objeto anterior, la clase 'CurveShape' se encarga de generar la apariencia del Shape3D, el color de la apariencia puede ser modificado mediante el método *setColor* de 'ObjectInterface'. También existe una apariencia común a todas las curvas para que puedan ser seleccionadas y un BranchGroup donde se añade el objeto 'CurveShape' antes de añadirlo al nodo Curves del árbol del panel 3D.

3.6.1.3 SurfaceShape

La clase 'SurfaceShape' es abstracta debido a que la interfaz trabaja con dos tipos de superficies. Existen dos clases que extienden de 'SurfaceShape':

- NurbsSurfaceShape. Se encarga de dibujar superficies NURBS.
- MeshSurfaceShape. Se encarga de dibujar mallas.

La clase 'SurfaceShape' extiende de la clase 'Shape3D' de Java3D y se encarga de generar su geometría dependiendo del modo de renderizado y el modo de visualización seleccionado. También almacena los atributos comunes de una superficie, como son su apariencia, el material de la superficie o sus propiedades.

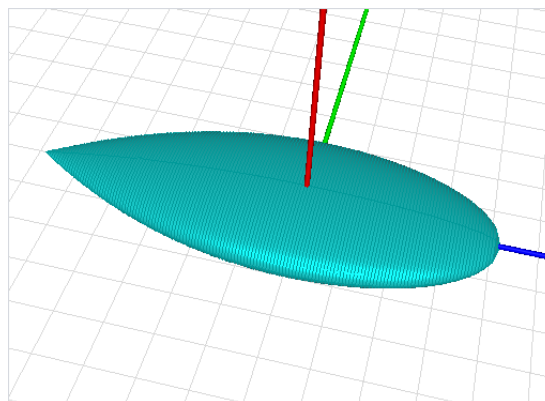


Figura 3.12: Objetos de tipo 'SurfaceShape'

La clase 'SurfaceShape' almacena dos tipos de geometría distintos, *líneas* y *render*. La geometría es generada dentro de las clases hijas. Esta clase se encarga de construir el BranchGroup y el Shape3D dependiendo del modo de renderizado que puede ser:

- LINES: La superficie se representa mediante líneas. Para ello se añade la geometría *líneas* al Shape3D y este al BranchGroup principal.
- SHADED: La superficie se representa mediante un polígono. Igual que el anterior pero usando la geometría *render*.
- LINE_SHADED: La superficie se representa mediante un polígono y líneas. Se usa el modo SHADED pero se crea un BranchGroup auxiliar que contiene otro Shape3D con las líneas que también cuelga del principal.

De la misma forma que el modo de renderizado, es posible ver la superficie con el color de la capa o el color del material seleccionado. La superficie guarda el modo de visualización seleccionado y el método *setColor* se encarga de actualizar la apariencia.

Debido al coste computacional de calcular el renderizado de superficies, cuando se transforma una superficie con una matriz de transformación mediante el método *transform* de 'ObjectInterface', se transforman los puntos del renderizado en vez de volver a calcularlos. De la misma manera al copiar un objeto existe una función de Java3D llamada *cloneNodeComponent* que permite realizar una copia de la geometría directamente.

NurbsSurfaceShape

La clase 'NurbsSurfaceShape' extiende de 'SurfaceShape' y permite dibujar una superficie NURBS en el panel 3D. Para ello, contiene un objeto de tipo 'NurbsSurface', sección 2.4.4, que define una superficie NURBS trimada [8].

La función *updateJava3DGeometry* se encarga de generar los dos tipos de geometría de la clase 'SurfaceShape' utilizando los algoritmos de renderizado del motor gráfico, véase la sección 2.5.3.

Si no se ha seleccionado un nivel de detalle específico, la interfaz calcula un nivel de detalle automático mediante la siguiente fórmula:

$$\text{Precisión} = N \cdot \text{grado} \tag{3.1}$$

Donde N es la suma de los puntos de control de la superficie y sus curvas trimmed y $N < 50$ (nivel de detalle máximo). El nivel de detalle puede ser diferente en cada dirección u, v de la superficie NURBS.

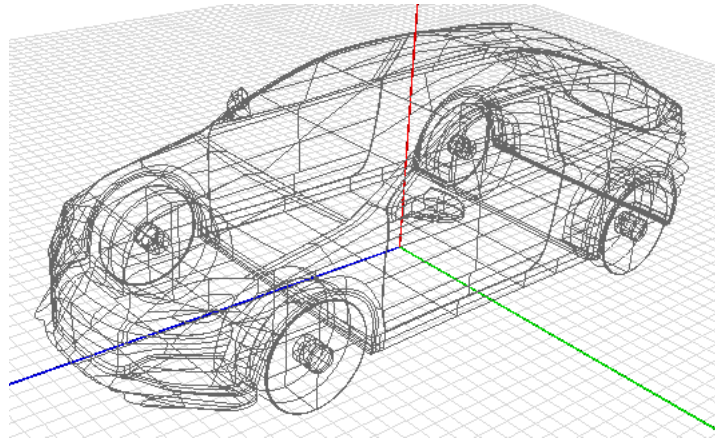


Figura 3.13: Ejemplo de objetos 'NurbsSurfaceShape'

MeshSurfaceShape

La clase 'MeshSurfaceShape' también extiende de 'SurfaceShape' y permite dibujar una malla en el panel 3D. Al tratar una malla como un objeto de tipo superficie, se pueden combinar varias mallas y superficies NURBS. También se pueden utilizar operaciones como mover, rotar, copiar, etc. sobre una malla.

Un objeto de tipo malla está compuesto por un conjunto de puntos y un conjunto de índices que indican cómo se conectan esos puntos. En la interfaz, los índices forman parches cuadrangulares, aunque es posible cargar mallas de triángulos repitiendo los índices 3 y 4 de cada parche.

Al igual que en el resto de objetos, el método *updateJava3DGeometry* se encarga de generar la geometría de la clase 'SurfaceShape'. Para generar las líneas se utiliza un objeto de tipo *LineArray* y para el renderizado un *IndexedQuadArray* de Java3D.

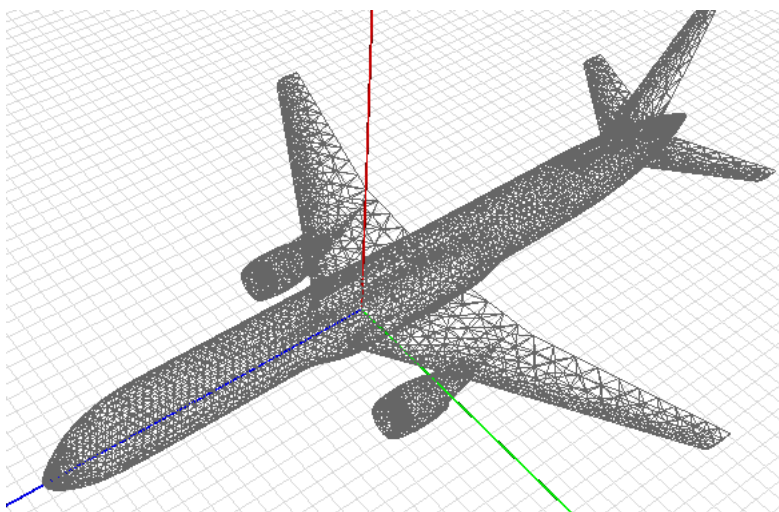


Figura 3.14: Ejemplo de 'MeshSurfaceShape'

3.6.1.4 Objeto3D

La clase 'Objeto3D' sirve para agrupar un conjunto de superficies en un único objeto. Para ello, extiende de la clase BranchGroup de Java3D (2.2.3). Dentro del 'BranchGroup' se almacenan los objetos de tipo 'SurfaceShape' que son las superficies que contiene el objeto, pueden ser mallas o superficies NURBS.

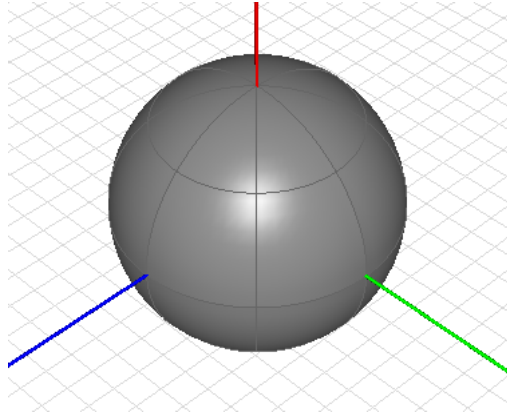


Figura 3.15: Ejemplo de 'Objeto3D' formado por 8 superficies NURBS

Para obtener la lista de superficies de un objeto se utiliza el método *getSurfaceList* que recorre todos los hijos del BranchGroup. Recordar que para poder añadir y eliminar nodos en el árbol de Java3D en tiempo de ejecución, estos deben ser de tipo BranchGroup y que en la clase 'SurfaceShape' existe un objeto BranchGroup principal que contiene el Shape3D. Esto quiere decir que el BranchGroup principal de la superficie es el que cuelga de 'Objeto3D' y su primer hijo es el Shape3D. En nuestro caso ese Shape3D es un objeto de tipo 'SurfaceShape'.²

Una vez que se tiene el método para extraer las superficies de un 'Objeto3D' es muy sencillo implementar los métodos *transform* y *updateJava3DGeometry* de un objeto. Para ello hay que recorrer la lista de superficies y llamar a los métodos correspondientes de la clase 'SurfaceShape'.

3.6.2 Layers

En esta sección se muestra como funciona el sistema de capas para dibujar los objetos geométricos dentro del 'Panel3D'.

Al arrancar la interfaz se crea automáticamente una capa por defecto y se selecciona. El usuario puede crear nuevas capas y asignarles un nombre y un color. Los nuevos objetos que crea el usuario o el resultado de una operación se almacena en la capa seleccionada (current). El usuario también puede mostrar y ocultar todos los objetos de una capa para trabajar cómodamente con modelos que tienen muchas superficies.

²Recordar que la clase 'SurfaceShape' extiende de la clase Shape3D de Java3D.

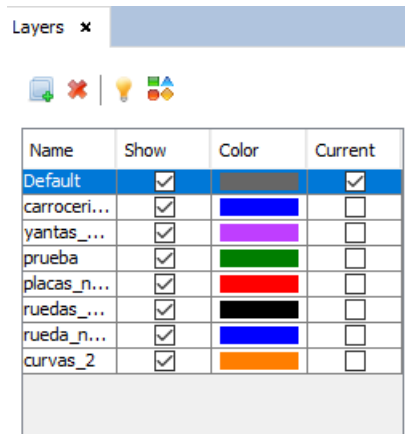


Figura 3.16: Ventana de Layers en la interfaz de usuario

En la figura 3.8 se muestran los nodos Points, Curves, Surfaces y Objects. En estos nodos se insertan los objetos geométricos para visualizarlos.

Los nodos donde se insertan los objetos geométricos son de tipo Switch. Un nodo Switch de Java3D funciona de la misma forma que un BranchGroup pero permite cambiar qué hijos se muestran en cada momento mediante un mapa de bits. Para manejar los mapas de bits se utiliza la clase BitSet de Java.

El funcionamiento es el siguiente:

- Existe un array de capas donde cada capa contiene el nombre, el color, la propiedad visible y cuatro mapas de bits (puntos, curvas, superficies y objetos).
- Si un objeto del nodo Objects está en una capa, esa capa tendrá '1' en el mapa de bits de objetos en la posición que ocupa el objeto dentro del nodo Objects. La capa tiene '0' en las posiciones de los objetos que no pertenecen a esa capa.
- Para calcular la máscara del Switch del nodo Objects se hace la operación OR entre las máscaras de todas las capas visibles. Esto muestra en el panel solo los objetos de las capas visibles.
- Al insertar un objeto en una capa se le asigna el color de la capa a su Apariencia con el método *setColor* de *ObjectInterface*.

3.6.3 Selección

Para realizar operaciones geométricas en un entorno de diseño 3D el usuario necesita un método para seleccionar los distintos objetos que hay dibujados dentro del Panel3D.

La interfaz de usuario permite seleccionar los objetos geométricos de varias maneras: en primer lugar, se puede seleccionar un objeto haciendo click sobre él; en segundo lugar, el usuario puede seleccionar varios objetos a la vez mediante un rectángulo de selección.

Para añadir un objeto a la selección se puede utilizar cualquier método de selección manteniendo pulsada la tecla 'Ctrl'. También existe una lista desplegable en la barra inferior que permite elegir los tipos de objetos que se pueden seleccionar.

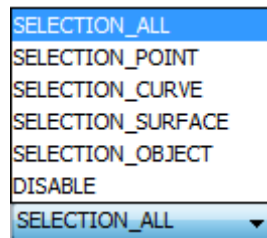


Figura 3.17: Barra de estado: Tipos de selección

Si existen muchos objetos en pantalla se puede seleccionar `SELECTION_CURVE` para que la selección solo funcione con curvas. A veces puede resultar útil deshabilitar por completo la selección cuando se quiere mover la cámara o cazar puntos en la geometría.

Cuando se hace click para seleccionar un objeto geométrico, se puede dar el caso de que varias superficies coincidan en la posición actual del ratón. En este caso, se muestra una lista con las superficies que coinciden en ese punto para que el usuario pueda elegir la que quiere seleccionar.

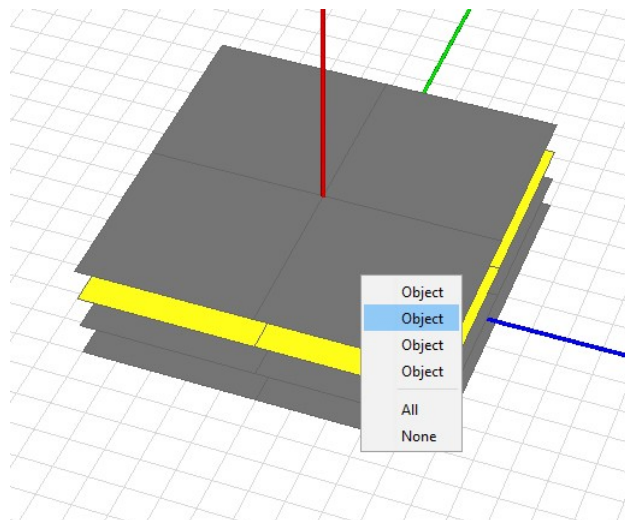


Figura 3.18: Lista para seleccionar varias superficies que coinciden

La selección en la interfaz se ha implementado en la clase `PickObjectSelector` que extiende de `PickMouseBehavior` de Java3D. Esta clase cuelga del nodo `Root` en la interfaz, esto permite que al hacer click en el panel 3D se ejecute la función `updateScene` con la posición x, y del ratón en el plato de imagen (Sección 2.2.2). A partir de ahí se puede crear un objeto `PickCanvas` que devuelva los objetos geométricos que se encuentran en esa posición.

Java3D no implementa ningún método para hacer la selección por rectángulo así que se ha implementado de forma manual utilizando los eventos del Canvas.

Cuando se lanza el evento *mouseDragged* por primera vez se crea un objeto *RectangleSelection* con la posición x, y del ratón inicial en la pantalla. Si ya existe un objeto *RectangleSelection*, según el usuario va arrastrando el ratón se actualizan las posiciones x, y finales del rectángulo. Cuando el usuario suelta el ratón se lanza el evento *mouseReleased*, entonces según los objetos visibles y el tipo de selección seleccionado se mira que posición de la pantalla ocupa el centro del objeto utilizando los métodos *getVworldToImagePlate* y *getPixelLocationFromImagePlate*. Con esta información ya se puede comprobar si cae dentro del rectángulo y seleccionar los objetos correspondientes.

3.6.4 Cámara

La cámara de Java 3D permite visualizar la geometría en el panel 3D desde diferentes posiciones. En la interfaz de usuario se controla mediante el ratón:

- Si se mantiene pulsado el botón derecho del ratón y se arrastra sobre el Panel3D la cámara rota sobre su centro.
- Según el modo seleccionado, con el botón izquierdo del ratón se pueden seleccionar objetos o mover la cámara.
- El zoom se controla mediante la rueda del ratón.

También hay diferentes botones en el menú view que permiten centrar la cámara, poner una vista predeterminada, cambiar el tipo de proyección o hacer zoom a una selección.

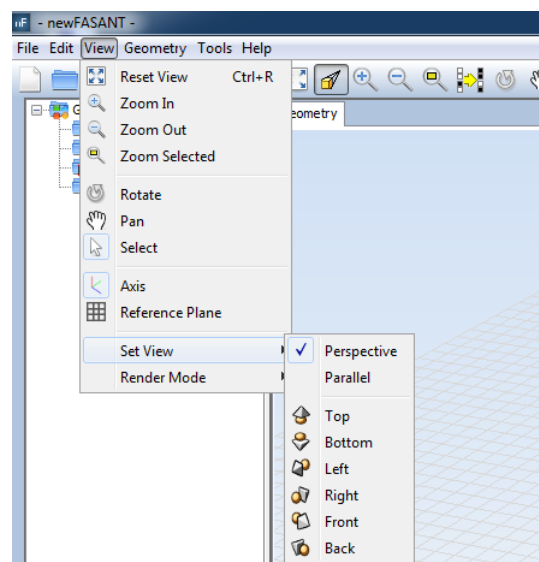


Figura 3.19: Funciones para controlar la cámara

La clase que controla el movimiento y la posición de la cámara se llama *MyOrbitBehavior* y está basada en la clase *OrbitBehavior* de Java3D.

Las funciones que controlan la cámara se definen en 'CameraFunctions'; las más importantes son *setCameraDistance* y *setCameraRotation* que se encargan de establecer la posición y orientación de la cámara modificando su matriz de transformación. En Java3D esta matriz se obtiene con la función: *universe.getViewingPlatform().getViewPlatformTransform()*.

3.6.5 Ejes

Los ejes cartesianos sirven para ver la orientación del sistema de coordenadas absoluto de la interfaz. La clase 'Axis' se encarga de dibujarlos mediante el nodo Axis (de tipo BranchGroup) que se añade al nodo raíz de la figura 3.8.

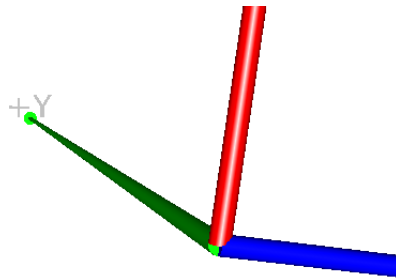


Figura 3.20: Ejes cartesianos

Los ejes se han creado utilizando las primitivas 'Cone' y 'Cylinder' de Java3D. Cada eje es generado en la misma posición inicial y colocado en su posición mediante un TransformGroup. El texto de cada eje se ha creado mediante las clases OrientedShape3D y Text3D.

3.6.6 Plano de Referencia

El plano de referencia es la rejilla de apoyo que se utiliza para construir geometría en la interfaz de usuario. Se utiliza para definir un sistema de coordenadas local ya que es posible variar su posición y orientación respecto al sistema de coordenadas global (definido por los ejes cartesianos) y su tamaño.

Cuando se ejecuta un comando para construir geometría, por ejemplo un cubo, el usuario puede dibujarlo sobre el plano de referencia utilizando el ratón, así como capturar sus coordenadas utilizando el 'pick'.

Al modificar la posición u orientación del plano de referencia, se añade a una lista para que el usuario pueda moverse entre los distintos planos de una manera sencilla. En la siguiente figura se muestra la ventana que controla el plano de referencia.

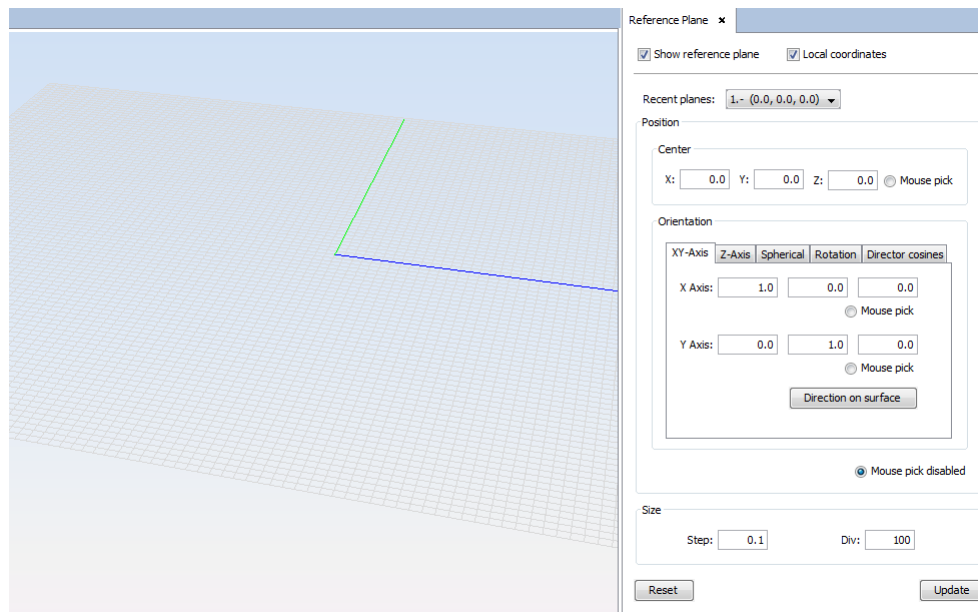


Figura 3.21: Plano de Referencia

Internamente el funcionamiento del plano de referencia es muy sencillo. Se almacena una matriz de transformación con la posición y orientación actual del plano de referencia que se aplica al resultado de cada comando si está seleccionada la opción 'Local coordinates'.

La clase 'ReferencePlane' se encarga de generar la geometría del plano de referencia mediante un IndexedQuadArray y añadirla al nodo raíz. Las líneas de color azul y verde indican los eje X e Y del plano de referencia y se dibujan mediante un LineStripArray.

3.6.7 Pick

La función 'pick' se utiliza para capturar puntos de la geometría en el panel 3D utilizando el ratón. Según el método seleccionado, al pasar el ratón sobre los objetos geométricos o el plano de referencia, el pick devuelve las coordenadas (x, y, z) correspondientes.

Para que el pick funcione correctamente es necesario utilizar un método muy rápido para calcular las coordenadas del ratón, ya que se actualizan en tiempo real según se va moviendo el ratón.

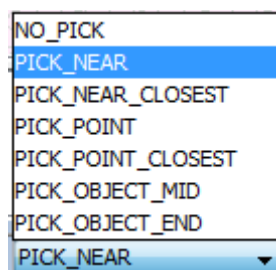


Figura 3.22: Barra de estado: Tipos de pick

La clase 'PickPoint' se encarga de devolver el punto del ratón que se está capturando en el momento según el modo seleccionado y de actualizar la barra de estado con las coordenadas.

El método para obtener el punto mas cercano al ratón es el siguiente:

- A partir de las coordenadas (x, y) del ratón en la pantalla (MouseEvent) se obtienen sus coordenadas en 3D utilizando las funciones de transformación de Canvas3D *getPixelLocationInImagePlate* y *getImagePlateToWorld*.
- De forma análoga se obtienen las coordenadas en 3D de la posición de la cámara o eye (figura 2.3) mediante las funciones *getCenterEyeInImagePlate* y *getImagePlateToWorld*.
- Se obtiene la lista de puntos que se quieren comparar dependiendo del modo seleccionado, por ejemplo, en el modo 'PICK_OBJECT_MID' se obtienen los puntos medios de los objetos utilizando el método *getMidPoints* de 'ObjectInterface'.
- Se busca el punto mas cercano calculando el ángulo que forman los vectores entre el punto eye y el ratón y entre el punto eye y cada punto de búsqueda. El punto que tenga un ángulo menor es el más cercano al ratón.

En la interfaz se utiliza un método para acelerar el proceso en el modo PICK_NEAR, ya que en este modo los puntos de búsqueda son todos los puntos del plano de referencia más todos los puntos de los objetos dibujados. Para ello se utiliza el objeto PickCanvas³ para calcular el punto de intersección del ratón con la geometría utilizando el método *getClosestIntersection* de 'PickResult'.

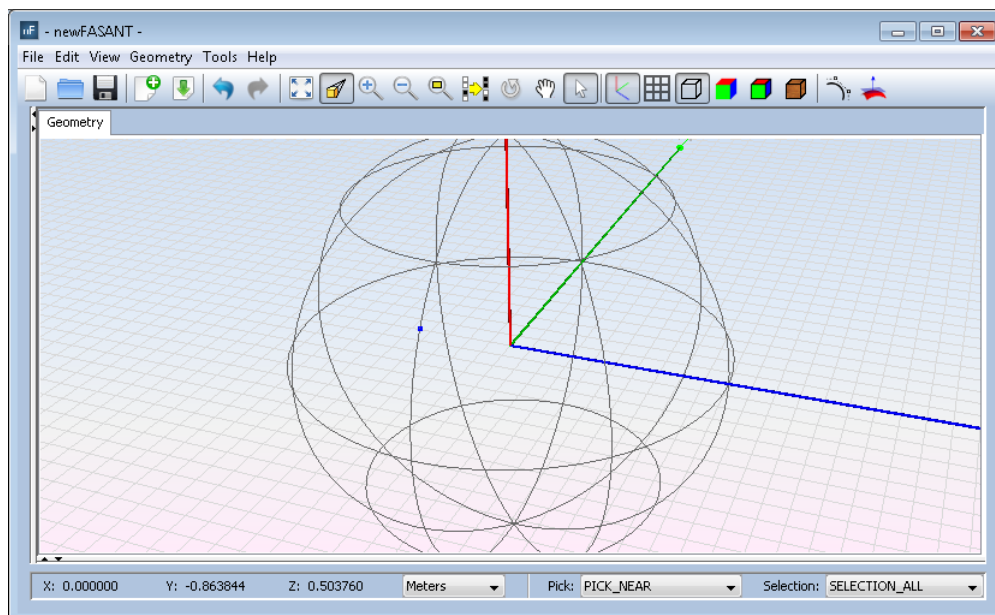


Figura 3.23: Ejemplo de 'Pick' en una esfera

³PickCanvas es el mismo objeto de Java 3D que se utiliza para seleccionar la geometría.

3.7 Consola de comandos

La consola de comandos se utiliza para crear y editar la geometría del panel de geometría visto en la sección anterior. En principio, aunque una consola de comandos pueda parecer una opción complicada para el usuario, ofrece varias ventajas en un programa de diseño gráfico que se enumeran a continuación:

- La estructura de todos los comandos de geometría es similar y no es necesario aprender el nombre de cada comando ya que son accesibles desde el menú de la aplicación.
- Una vez que se aprenden los comandos principales es muy rápido de usar. Los comandos se pueden editar y reutilizar fácilmente. Se pueden guardar e importar scripts de geometría.
- Se puede dibujar la geometría utilizando el ratón y capturar puntos directamente en la consola.
- El historial de comandos es la base para las simulaciones paramétricas.

En la siguiente figura se muestra la consola de comandos ejecutando el comando 'help':

```

command> help
Help Usage: Type 'help <command>' or '<command> -h' to show help about a specific command.
Commands:
- Type a command and press ENTER to execute it.
- Press ESC to abort a command.
- Press TAB to complete the written pattern name with the coincident commands.
- Use the mouse to pick points or draw the geometry directly.
- Press Ctrl+Left in a command step for coming back to the previous one.
History:
- Use Up/Down arrows to show the commands used recently.
- Use Ctrl+Z or Ctrl+Y for Undo/Redo.
- Go to Edit->History to modify previous operations or delete specific commands.
Ctrl+L: Clear the console panel.

```

Figura 3.24: Consola de comandos

La consola de comandos imprime la palabra *command* (en color azul) y se queda a la espera de que el usuario escriba un comando y presione 'Enter' o utilice el menú Geometry para seleccionar una operación automáticamente. El comando puede escribir mensajes en la consola (en color gris) para informar o pedir al usuario los distintos parámetros. El usuario va introduciendo los distintos parámetros y pulsando 'Enter'. Cuando el proceso acaba el comando se encarga de realizar la operación seleccionada, o si hay algún error, mostrarlo en la consola con un mensaje (en color rojo).

El paquete 'Console' de la interfaz se encarga de implementar toda la funcionalidad de la consola de comandos, la clase principal es 'ConsolePanel'.

3.7.1 Clase ConsolePanel

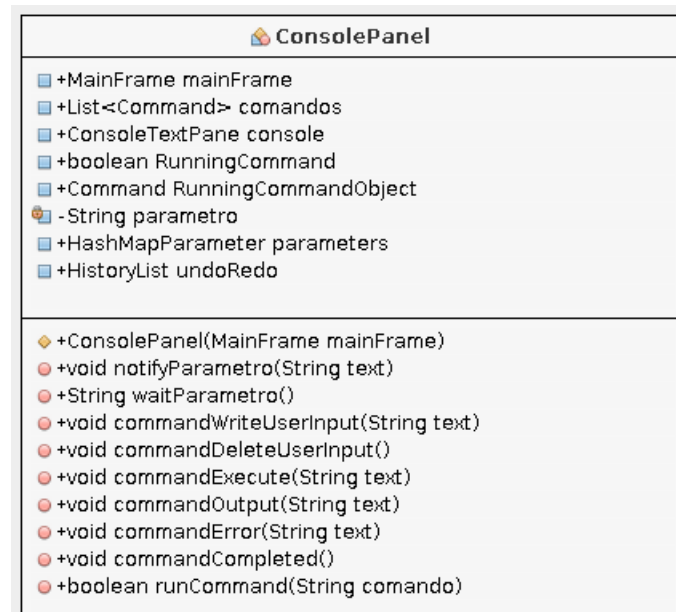


Figura 3.25: Clase 'ConsolePanel'

La clase 'ConsolePanel' extiende de JPanel y contiene el objeto console de la clase 'ConsoleTextPane' que es un JTextPane modificado para que el usuario solo pueda escribir a continuación del cursor y con los distintos estilos para los mensajes. Además contiene los siguientes atributos:

- **MainFrame.** Es el puntero a la clase principal 'MainFrame', se pasa a los comandos para que tengan acceso al Panel 3D.
- **Comandos.** Contiene la lista de comandos que se pueden ejecutar. Cada comando extiende de la clase 'Command'.
- **RunningCommand.** Indica si se está ejecutando un comando en este momento.
- **RunningCommandObject.** Contiene el comando que se está ejecutando.
- **Parametro.** Si se está ejecutando un comando, en parámetro se guarda lo que escribe el usuario.
- **Parameters.** Contiene la lista de variables que se pueden utilizar para construir geometría paramétrica.
- **UndoRedo.** Contiene el historial de comandos que se han ejecutado para deshacer y rehacer las operaciones.

El funcionamiento es el siguiente:

La clase está escuchando el evento keyPressed de Swing. Cuando se presiona la tecla 'Enter', si no hay ningún comando ejecutándose, se llama al método *commandExecute* que

busca en la lista de comandos si existe un comando con el mismo nombre. Si lo encuentra, ejecuta el comando pasándole los argumentos que ha introducido el usuario en la misma línea. El comando utiliza los métodos *commandOutput* y *commandError* para escribir su salida en la consola. Si el comando es interactivo el proceso se ejecuta en un hilo. El comando puede utilizar el método *commandWriteUserInput* para escribir en el espacio del usuario y quedarse a la espera de un parámetro llamando a la función *waitParametro*. Si se pulsa la tecla 'Enter' en la consola mientras un comando se está ejecutando, la consola despierta el hilo del comando con *notifyParametro* y guarda el parámetro en la variable 'parametro', que es recogida por el comando para continuar su ejecución.

3.7.2 Clase Command

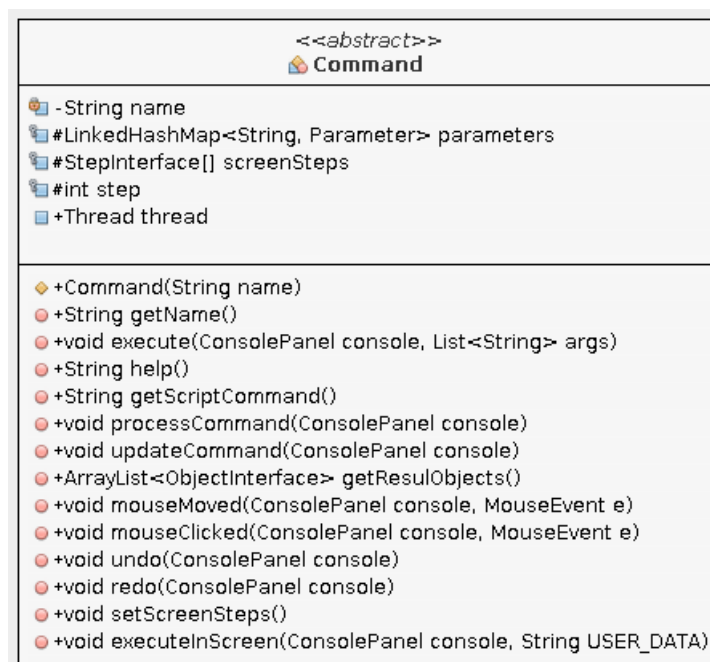


Figura 3.26: Clase 'Command'

La clase 'Command' es una clase abstracta de la que extienden todos los comandos de la consola. Esto le permite a cada comando responder a los distintos eventos de la interfaz y, como a través de la clase 'ConsolePanel' pueden acceder a 'MainFrame' que contiene todos los datos de la aplicación, se pueden hacer comandos para realizar cualquier tarea de la interfaz.

Cuando se ejecuta un comando la consola llamará a la función *execute* y el comando realizará sus tareas. Mientras un comando está en ejecución, el comando puede atender a los eventos de ratón del panel 3D, para dibujar geometría temporal o capturar puntos mediante el pick. El comando también se encarga de deshacer y rehacer sus operaciones, además de actualizar su geometría en el caso de hacer una simulación paramétrica, como se verá en las siguientes secciones.

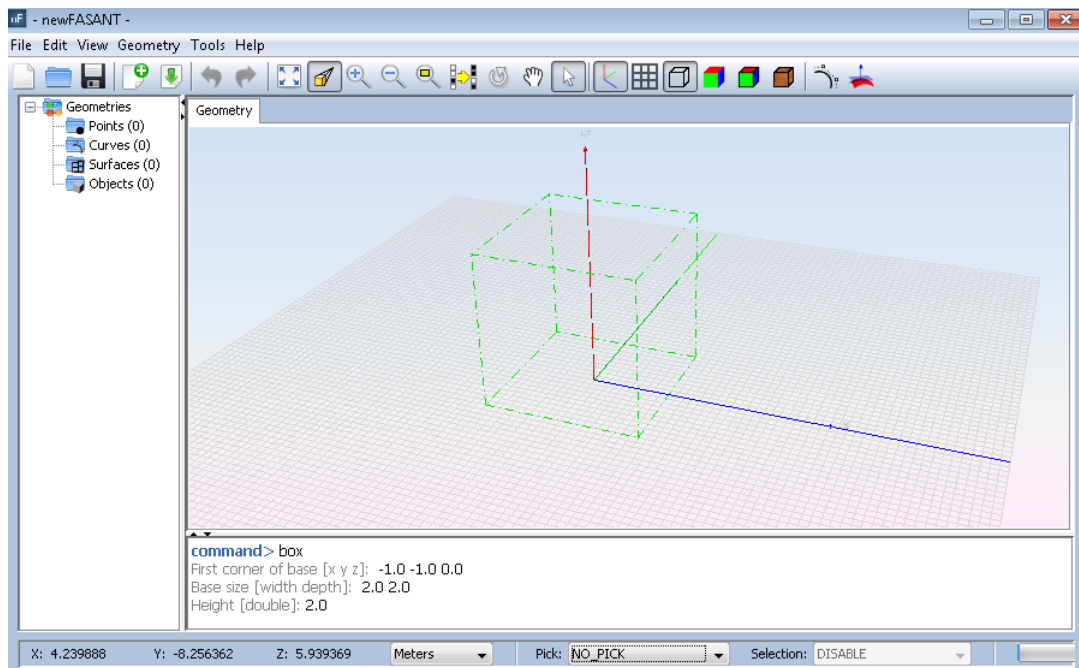


Figura 3.27: Ejemplo de comando: box

En la figura 3.27 se puede observar el comando 'box' mientras se está ejecutando. Las líneas en verde son líneas de geometría temporal que se crean al dibujar el cubo con el ratón. Al hacer click con el ratón en la pantalla el comando escribe en la consola las coordenadas del punto capturado para ayudar al usuario a construir la geometría.

3.7.3 Historial de Comandos

El historial de comandos contiene una lista con todas las operaciones geométricas que se han hecho en la interfaz. Para ello, almacena una lista de comandos y un puntero que indica la posición actual en la lista.

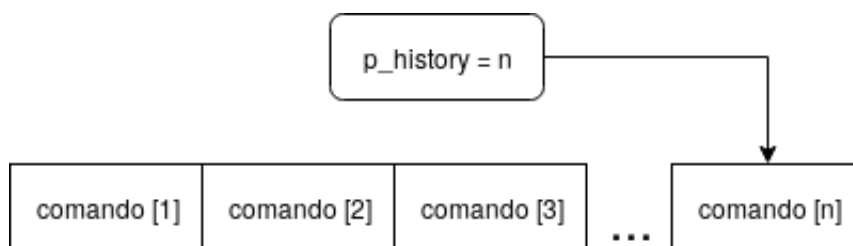


Figura 3.28: Historial de comandos

El funcionamiento es el siguiente:

- La consola cuando ejecuta un comando clona el objeto 'Command' de su lista de comandos y llama a su función *execute*, cuando este termina contiene los datos de ese comando, entonces se borran todos los comandos que hay a la derecha del puntero, se añade al historial y se incrementa el puntero en 1.

- Si se quiere deshacer una operación, se llama al método *undo* del comando apuntado por el puntero y se decrementa.
- Si se quiere rehacer una operación, se incrementa el puntero y se llama al método *redo* del comando.

La siguiente pestaña permite visualizar y editar el historial de comandos.

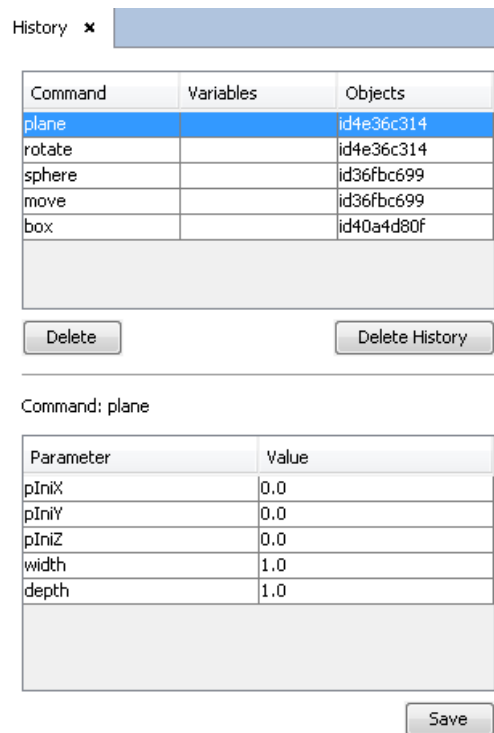


Figura 3.29: Pestaña para editar el historial de comandos

En la figura 3.26 se puede observar que cada comando contiene una lista de parámetros y un método *updateCommand* que se llama al editar un comando. Estos parámetros son los que se muestran en la ventana y permiten ser modificados.

La consola y el historial de comandos son la base para la creación de geometría paramétrica, como se muestra a continuación.

3.8 Parametrización de la geometría

La mayoría de comandos de la interfaz pueden ser parametrizados. Esto quiere decir que pueden definirse parámetros que tomen diferentes valores y utilizarlos en los comandos para crear la geometría o realizar operaciones.

Los parámetros variables pueden ser de tres tipos:

- Un conjunto de valores, que se define como $p = \{v_1, v_2, v_3, v_4, \dots, v_n\}$.

- Un conjunto de valores lineales $p = [v_{ini}, v_{fin}] n$, donde n es el numero de muestras.
- Una función que depende de otros parámetros $p = f(p_1, p_2, \dots, p_n)$

En la figura 3.30 se puede ver un ejemplo de parámetros definidos en la interfaz utilizando los distintos tipos.

Define Parameters ✕

Parameter	Values
t	0.3
a	[5.4,5.7] 31
b	{6.9,7.0,7.1,7.2}
\$1	-t
\$2	-t/2
\$3	t/2
\$4	a/2-\$3
\$5	-a/2-\$2
\$6	b/2-\$3
\$7	-b/2-\$2

Add parameter Delete parameter

Saved Save

Figura 3.30: Pestaña para definir parámetros variables

Definir la geometría de esta manera permite realizar simulaciones paramétricas, en las que la interfaz irá variando la geometría automáticamente y realizando las simulaciones. Posteriormente, se pueden visualizar y comparar los resultados de los distintos pasos.

Para ver los distintos pasos que se van a simular se utiliza la pestaña de la figura 3.31, en la que además de cambiar el paso se puede elegir cómo se van a variar los parámetros. Como es de suponer, en la variación solo se van a tener en cuenta los parámetros que se están utilizando en algún comando.

Actualmente, existen dos maneras de variar los parámetros para realizar simulaciones paramétricas:

- **Cross join.** Realiza todas las combinaciones posibles de todos los parámetros en orden, lo que se conoce matemáticamente como permutaciones⁴.
- **Linear.** Incrementa el valor de todos los parámetros en cada simulación. Si un parámetro ya no tiene más valores realizará las siguientes simulaciones con su valor final.

⁴Sea un conjunto de n elementos. A las ordenaciones que se pueden hacer con estos n elementos sin repetir ningún elemento y utilizándolos todos se las denomina permutaciones. El número de permutaciones que se pueden realizar coincide con el factorial de n .

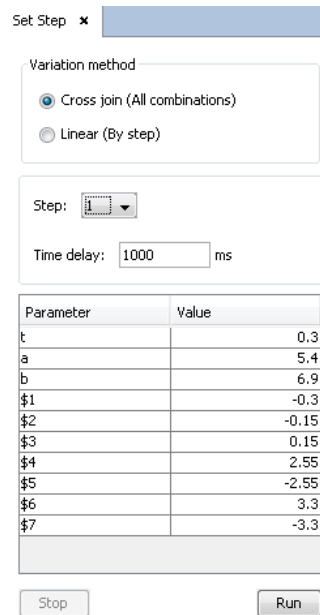


Figura 3.31: Pestaña para seleccionar la variación paramétrica y el paso

La ventana también permite visualizar automáticamente el valor de todos los pasos eligiendo el tiempo de espera entre cada paso y pulsando el botón 'Run'. La animación se puede abortar en cualquier momento pulsando el botón 'Stop'.

En el siguiente ejemplo se construye un cubo paramétrico en el que se varía la altura utilizando el comando 'box', en la figura 3.27 se puede ver el mismo ejemplo sin utilizar parámetros.

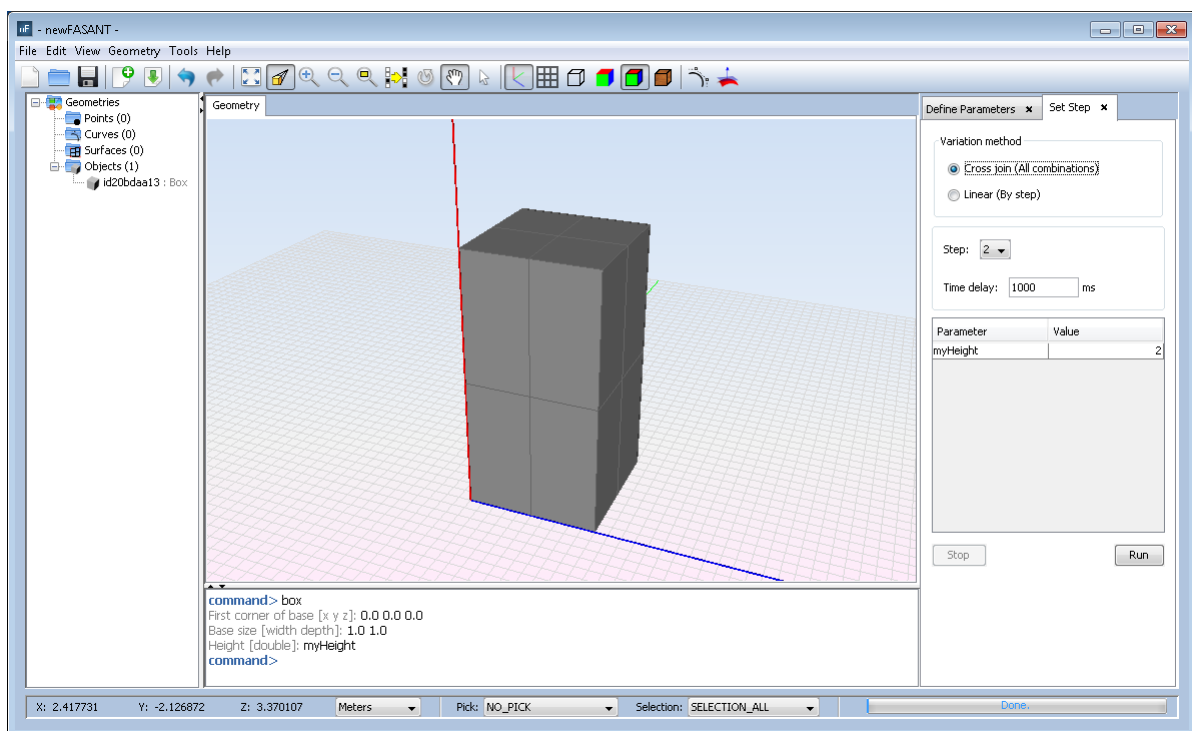


Figura 3.32: Comando 'box' usando el parámetro $myHeight = \{1, 2, 3\}$ en el segundo paso

Al cambiar de paso, la interfaz actualiza todos los comandos. Para ello, deshace todas las operaciones del historial y las va rehaciendo con los nuevos valores de los parámetros. En la siguiente figura se muestra el resultado del ejemplo anterior:

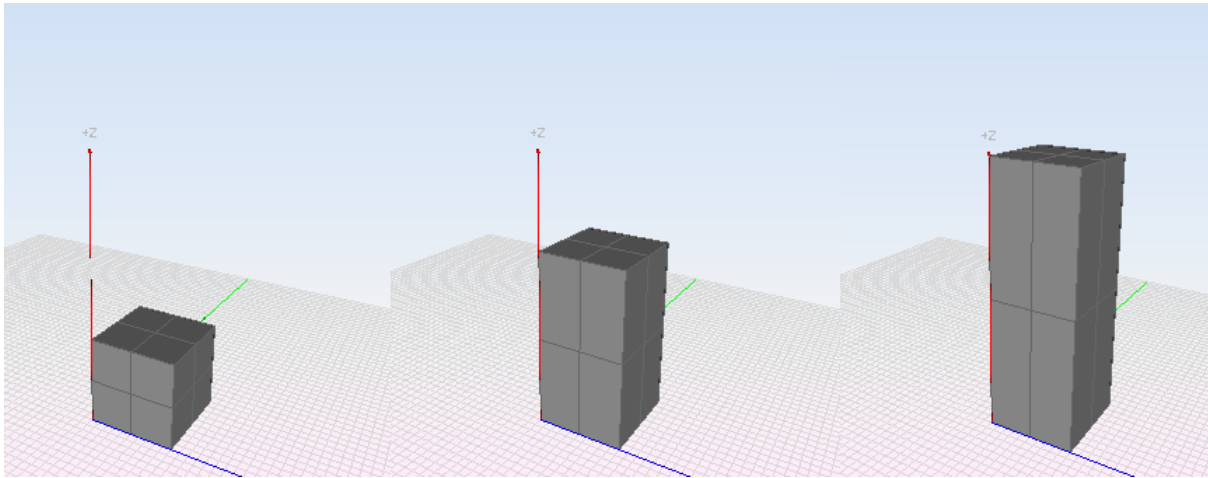


Figura 3.33: Resultado de la variación paramétrica

3.8.1 Clase HashMapParameter

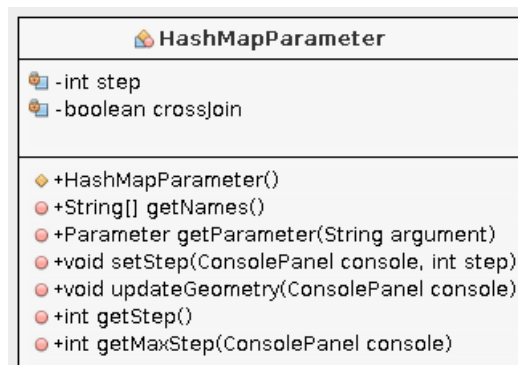


Figura 3.34: Clase 'HashMapParameter'

La clase 'HashMapParameter' extiende de 'LinkedHashMap<String,Parameter>' y se encuentra dentro de la consola en la clase 'ConsolePanel'. Esta clase almacena una tabla hash cuya clave es el nombre del parámetro y su valor es un objeto de tipo 'Parameter'.

La clase 'Parameter' es el tipo más sencillo y contiene el nombre y el valor actual del parámetro. El resto de tipos de parámetros extienden de esta clase. Cada comando almacena una tabla con sus parámetros que pueden ser variables, de esta manera, cuando un comando le pide un parámetro al usuario se llama a la función *getParameter* de esta clase. La función comprueba si es un valor, en cuyo caso se crea un nuevo parámetro, o es un parámetro variable de su tabla, el cual se envía directamente al comando para que lo almacene.

De esta manera, cuando se quiere cambiar el paso, primero se actualizan los valores

de los parámetros de esta tabla llamando a la función *setStep* de cada parámetro. Cada comando tiene el valor actualizado ya que ha guardado una referencia al objeto 'Parameter' de la tabla. A continuación, la interfaz deshace todo el historial y llama a la función *updateCommand* de cada comando. La función vuelve a ejecutar el comando con los nuevos parámetros y solo queda rehacer las operaciones del historial para obtener toda la geometría actualizada.

3.8.2 Evaluación de Funciones

Existe un tipo de parámetros especiales que pueden ser funciones de otros parámetros. En la interfaz se han implementado mediante la clase 'FunctionParameter' que extiende de 'Parameter'.

Esto obliga a la interfaz a evaluar los parámetros en orden, para ello es necesario extraer las variables de cada función y evaluarlas antes. En la interfaz la función se guarda como una cadena de texto, por lo que se mira si esa cadena contiene algún otro parámetro definido en la tabla.

Para evaluar funciones en la interfaz se utiliza el paquete BeanShell [9]. La clase 'Interpreter' de BeanShell permite ejecutar código en Java directamente en la aplicación. Una vez creado el interprete se introducen los valores de los parámetros que se han extraído de la función con el método *set* y se llama al método *eval* que devuelve el valor de la función para este paso.

Los parámetros funcionales pueden ser definidos directamente en la consola cuando se está ejecutando un comando. La función *getParameter* de la clase 'HashMapParameter' comprueba si lo que ha introducido el usuario es una función evaluable, en cuyo caso crea automáticamente un nuevo parámetro que almacena en la tabla y envía al comando. Estos parámetros aparecerán nombrados como \$1, \$2, \$3, ..., \$n.

```
command> box
First corner of base [x y z]: 0.0 0.0 0.0
Base size [width depth]: 1.0 (a+b)/2
Height [double]: 1.0
command>
```

Figura 3.35: Parámetros funcionales definidos en la consola

3.9 Ficheros Geométricos

Los ficheros de geometría se utilizan para cargar y guardar la geometría del panel 3D. También permiten la comunicación entre la interfaz, el mallador y los programas de simulación.

Actualmente, la interfaz de usuario soporta los formatos de geometría más usados, lo que permite al usuario importar y exportar geometrías desde otros programas de diseño.

En este apartado se explican con detalle los formatos que han sido implementados en la interfaz de usuario, estos son los formatos NUR, IGES y MSH.

3.9.1 Formato NUR

El formato NUR es un formato propio para el intercambio de la geometría entre la interfaz y los programas de simulación. Al principio se diseñó como un formato para pasar las superficies NURBS al mallador, pero se ha ampliado para almacenar todos los tipos de objetos geométricos de la interfaz junto con sus propiedades.

Tener un formato propio permite guardar más información que utilizando un formato estándar, por ello, es el formato recomendado para exportar e importar geometría entre distintos proyectos diseñados con la interfaz de usuario.

La estructura del fichero es la siguiente:

```
LAYERS
  2                !Número de capas
  capa_negra      !Nombre de la capa
  0 0 0           !Color (rgb)
  capa_roja
  255 0 0
END_LAYERS
GEOMETRY
  1                !Número de objetos geométricos
  POINT           !Tipo de objeto
  PROPERTIES      !Propiedades del objeto
    name = punto
    layer = 0
  END_PROPERTIES
  0.0 0.0 0.0     !Coordenadas del punto (x,y,z)
END_POINT
END_GEOMETRY
```

Tabla 3.2: Formato de fichero .nur

Dentro del apartado GEOMETRY se definen los objetos geométricos y sus propiedades. En este caso el fichero contiene un punto en el origen de coordenadas. El apartado PROPERTIES se ha creado para almacenar la tabla de propiedades de cada objeto (Sección 3.6.1). En esta versión solo se almacena el nombre del objeto y el índice de la capa al que pertenece. Como puede verse, cada apartado termina con la etiqueta END_ seguida del nombre del apartado.

A continuación se describe la forma de almacenar cada objeto de la interfaz de usuario utilizando el formato NUR. El objeto de tipo POINT que representa un punto puede verse en el ejemplo anterior.

El objeto 'NurbsCurveShape' que representa una curva NURBS se almacena en el fichero .nur de la siguiente manera.

```

NURBS_CURVE          !Tipo de objeto
  PROPERTIES         !Propiedades del objeto
    name = curva_1
    layer = 0
  END_PROPERTIES
  1                  !Grado de la curva NURBS
  2                  !Número de puntos de control
  0.0 0.0 0.0 1.0   !Punto 1: (x,y,z,w)
  2.0 0.0 0.0 1.0   !Punto 2: (x,y,z,w)
  4                  !Número de knots de la curva NURBS
  0.0 0.0 1.0 1.0   !Vector de knots
END_NURBS_CURVE

```

Tabla 3.3: Formato de fichero .nur: Curva NURBS

El objeto NURBS_CURVE anterior define una curva NURBS de grado uno, con dos puntos de control y un vector con cuatro nodos. Esta curva representa una línea recta entre los puntos (0,0,0) y (2,0,0).

El siguiente objeto geométrico es 'NurbsSurfaceShape' y representa una superficie NURBS. Puede verse como una extensión de la curva en dos dimensiones.

```

NURBS_SURFACE          !Tipo de objeto
  PROPERTIES         !Propiedades del objeto
    name = superficie_1
    layer = 0
  END_PROPERTIES
  1                  !Grado en U de la superficie NURBS
  1                  !Grado en V
  2                  !Número de puntos de control en U
  2                  !Número de puntos de control en V
  -1.0 -1.0 0.0 1.0 !Punto 1,1: (x,y,z,w)
  1.0 -1.0 0.0 1.0 !Punto 1,2
  -1.0 1.0 0.0 1.0 !Punto 2,1
  1.0 1.0 0.0 1.0 !Punto 2,2
  4                  !Número de knots en U
  0.0 0.0 1.0 1.0 !Valores
  4                  !Número de knots en V
  0.0 0.0 1.0 1.0 !Valores
END_NURBS_SURFACE

```

Tabla 3.4: Formato de fichero .nur: Superficie NURBS

Si la superficie NURBS es trimada, la interfaz almacena la información de las curvas recortadoras en coordenadas paramétricas y cartesianas en la clase 'NurbsSurface', descrita en la sección [2.4.4](#).

En este caso antes de la etiqueta END_NURBS_SURFACE se almacena la información de los lazos recortadores, la estructura es similar para los dos tipos de curvas.

```

NURBS_SURFACE      !Tipo de objeto
...               !Información de la superficie NURBS (Figura anterior)
TRIMMED           !Curvas recortadoras en coordenadas paramétricas
  2               !Número de lazos
  1               !Índice del lazo actual
  1               !Sentido del lazo 0-agujero 1-borde
  1               !Número de curvas del lazo
  1               !Índice de la curva actual
  1               !Grado de la curva
  2               !Número de puntos de control
  0.0 0.0 1.0    !Punto 1: (u,v,w)
  2.0 0.0 1.0    !Punto 2: (u,v,w)
  4               !Número de knots
  0.0 0.0 1.0 1.0 !Valores de los knots
  2               !Índice del lazo actual
  0               !Sentido del lazo 0-agujero 1-borde
...
END_TRIMMED
TRIMMED_3D       !Curvas recortadoras en coordenadas cartesianas
  2               !Número de lazos
  1               !Índice del lazo actual
  1               !Sentido del lazo 0-agujero 1-borde
  1               !Número de curvas del lazo
  1               !Índice de la curva actual
  1               !Grado de la curva
  2               !Número de puntos de control
  0.0 0.0 0.0 1.0 !Punto 1: (x,y,z,w)
  2.0 0.0 0.0 1.0 !Punto 2: (x,y,z,w)
  4               !Número de knots
  0.0 0.0 1.0 1.0 !Valores de los knots
  2               !Índice del lazo actual
...
END_TRIMMED_3D
END_NURBS_SURFACE

```

Tabla 3.5: Formato de fichero .nur: Superficies NURBS trimadas

El objeto 'MeshSurfaceShape' representa una superficie de tipo malla que se define de la siguiente forma.

```

MESH_SURFACE      !Tipo de objeto
PROPERTIES       !Propiedades del objeto
  name = malla_1
  layer = 0
  mesh = false    !Propiedad que indica si hay que volver a mallar
END_PROPERTIES
  4               !Tipo de malla - Número de índices por elemento
  100            !Número de puntos
  10             !Número de elementos
  0.0 0.0 0.0    !Punto 1: (x,y,z)
...
  1 2 3 4        !Elemento 1: (p1,p2,p3,p4)
...
END_MESH_SURFACE

```

Tabla 3.6: Formato de fichero .nur: Malla

Como se muestra en la sección 3.6.1, los objetos 'NurbsSurfaceShape' y 'MeshSurfaceShape' pueden agruparse dentro de 'Objeto3D'; el formato NUR mantiene esa agrupación.

```

OBJECT                !Tipo de objeto
  PROPERTIES          !Propiedades del objeto
    name = objeto_1
    layer = 0
  END_PROPERTIES
  3                   !Número de superficies del objeto
  NURBS_SURFACE
  ...                !Información de la superficie 1
  END_NURBS_SURFACE
  NURBS_SURFACE
  ...                !Información de la superficie 2
  END_NURBS_SURFACE
  MESH_SURFACE
  ...                !Información de la superficie 3
  END_MESH_SURFACE
END_OBJECT

```

Tabla 3.7: Formato de fichero .nur: Objeto 3D

Al estar basado en los objetos de la interfaz de usuario, la implementación de la lectura y escritura del formato NUR es muy sencilla y básicamente es una transcripción a texto de cada objeto. Para ello se ha creado la clase 'ENTITY' que contiene un objeto de tipo 'ObjectInterface' y los métodos *read* y *write*. Cada tipo de objeto tiene una clase que extiende de 'ENTITY', por ejemplo 'NURBS_CURVE' que recibe el objeto 'NurbsCurveShape' e implementa las funciones de lectura y escritura de ese objeto.

La clase que implementa la lectura, 'FileNurbsReader', se encarga de abrir el fichero e ir leyendo líneas y cuando encuentra un objeto conocido crea un nuevo objeto de esa entidad y llama a su función *read*, que lee sus líneas y crea el objeto geométrico correspondiente.

Para escribir el fichero se utiliza la clase 'FileNurbsWriter' que recorre la lista de geometría y según el tipo de objeto, se crea la entidad correspondiente y se llama a su función *write*, que escribe las líneas correspondientes a esa entidad en el fichero.

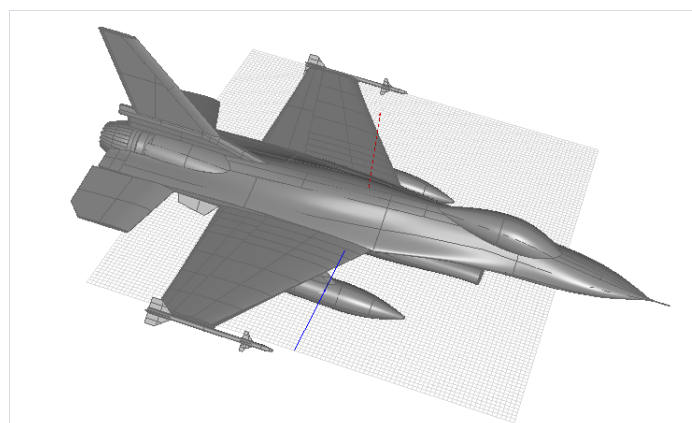


Figura 3.36: Aircraft.nur: 491 superficies NURBS

3.9.2 Formato IGES

IGES [10] o Initial Graphics Exchange Specification (Especificación de Intercambio Inicial de Gráficos) es un formato muy utilizado entre sistemas de diseño asistido por computadora (CAD). El formato IGES se creó en 1980 y sigue siendo el formato estándar de intercambio de gráficos más extendido, ya que está implementado en la mayoría de programas de diseño.

Un archivo IGES se compone de un juego de 80 caracteres ASCII, el tamaño del juego de caracteres viene de la era de las tarjetas perforadas. Las cadenas de texto estaban representadas en formato "Hollerith": el número de caracteres en la cadena, seguido por la letra "H", seguido por el texto, p.ej., "4HSLOT" (es la cadena de texto utilizada en las primeras versiones del lenguaje Fortran).

A continuación se muestra un pequeño archivo IGES generado con la interfaz de usuario que contiene tres entidades: una superficie NURBS (tipo 128) y la información de la capa en la que se encuentra que se define con dos entidades (tipos 406 y 314).

newFASANT v6.0 IGES File								S	1
1H, ,1H; , ,								G	1
9Hplane.igs ,								G	2
9HnewFASANT,11HVersion 6.0,32,38,6,308,15, ,								G	3
1.000,6,1HM,1,0.001,14H2018521.111023 ,								G	4
0.001,0.0, , ,11,0,14H2018521.111023;								G	5
314 1 0 0 0 0 0 000000200D									1
314 0 1 1 0 0 0 COLOR 0D									2
406 2 0 0 0 0 0 000000300D									3
406 0 -1 1 3 0 0LEVELDEF 0D									4
128 3 0 0 0 0 0 000000000D									5
128 0 -1 3 0 0 0 TrimSrf 0D									6
314,40.0,40.0,40.0,;									0000001P 1
406,2,1,7HDefault;									0000003P 2
128,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1.0,1.0,0.0,0.0,1.0,1.0,1.0,1.0,;									0000005P 3
1.0,1.0,-1.0,-1.0,0.0,1.0,-1.0,0.0,-1.0,1.0,0.0,1.0,1.0,0.0,0.0,;									0000005P 4
1.0,0.0,1.0;									0000005P 5
S0000001G0000005D0000006P0000005								T	1

Tabla 3.8: Formato de fichero IGES

El archivo está dividido en 5 secciones: Inicio, Global, Directorio de entrada, Parámetros de datos y Terminación, indicados por un carácter (S, G, D, P o T) en la columna 73. Las características y la información geométrica de una entidad se divide en dos secciones; una en dos registros de longitud fija (el Directorio Entrada o D), la otra en múltiples registros delimitados por comas (los Parámetros de datos o P).

En un archivo IGES las entidades se conectan a través de punteros. El valor del puntero de una entidad es el que aparece delante de la letra P, que corresponde con el número de línea de su entrada en la sección D. En el ejemplo anterior, en la entidad 406 (3P), el valor 1 es el puntero a la entidad 314 (1P) que define el color de la capa.

Aunque el formato IGES contiene más de 150 entidades en la práctica se usan unas pocas dependiendo del tipo de superficies con las que se trabaja. En la interfaz se ha implementado la lectura y escritura del formato IGES de forma que sea sencillo añadir nuevas entidades. Actualmente hay implementadas 23 entidades que permiten importar el 99 % de ficheros que contienen curvas y superficies NURBS. Estas entidades son:

- Tipo 100: CIRCULAR ARC ENTITY
- Tipo 102: COMPOSITE CURVE ENTITY
- Tipo 108: UNBOUNDED PLANE
- Tipo 110: LINE ENTITY
- Tipo 118: RULED SURFACE ENTITY
- Tipo 120: SURFACE OF REVOLUTION ENTITY
- Tipo 122: TABULATED CYLINDER ENTITY
- Tipo 124: TRANSFORMATION MATRIX ENTITY
- Tipo 126: RATIONAL B-SPLINE CURVE ENTITY
- Tipo 128: RATIONAL B-SPLINE SURFACE ENTITY
- Tipo 141: BOUNDARY ENTITY
- Tipo 142: CURVE ON A PARAMETRIC SURFACE ENTITY
- Tipo 143: BOUNDED SURFACE ENTITY
- Tipo 144: TRIMMED (PARAMETRIC) SURFACE ENTITY
- Tipo 186: MANIFOLD SOLID B-REP OBJECT ENTITY
- Tipo 308: GROUP ENTITIES
- Tipo 314: COLOR DEFINITION ENTITY
- Tipo 406: PROPERTY ENTITY
- Tipo 408: INSTANCE GROUP ENTITY
- Tipo 504: EDGE ENTITY
- Tipo 508: LOOP ENTITY
- Tipo 510: FACE ENTITY
- Tipo 514: SHELL ENTITY

En la siguiente figura se muestra un fichero IGES renderizado en la interfaz de usuario. El fichero ocupa 67MB y contiene 7590 superficies NURBS, el tiempo de procesado en la interfaz ha sido inferior a 20 segundos.

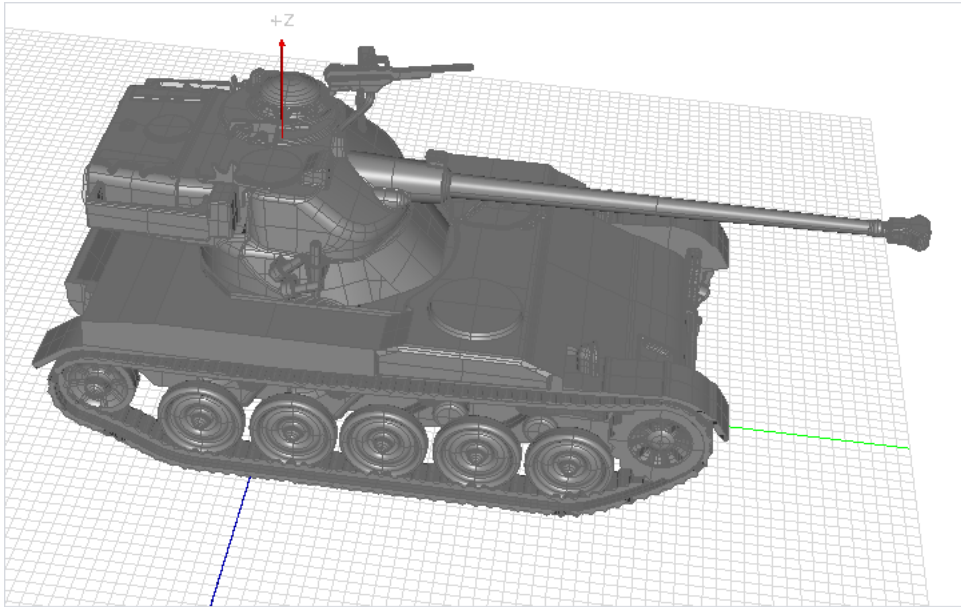


Figura 3.37: Tank.igs: 7590 superficies NURBS

3.9.3 Formato MESH

El formato MESH (.msh) se utiliza para importar y exportar mallas en la interfaz de usuario. La herramienta GID [11] y nuestro mallador generan ficheros con este tipo de formato. En la interfaz de usuario una malla se define con el objeto 'MeshSurfaceShape' con lo que es considerada una superficie y se puede operar con ellas y combinarlas con el resto de la geometría. Esto permite a la interfaz de usuario reutilizar una malla como parte de otra simulación.

La estructura del fichero es muy sencilla, en primer lugar se define la sección de coordenadas de puntos y en segundo lugar se establece la descripción de los elementos a partir de los puntos indicados en la primera sección. La cabecera incluye información sobre el número de puntos que contiene cada elemento (nnode) y sobre el número de coordenadas que tiene cada punto (dimension).

El mallador de la interfaz de usuario genera elementos curvos de 3x3 puntos, esto le permite a los núcleos de simulación conservar la curvatura de las superficies NURBS; por lo tanto el número de nodos del fichero MSH en las mallas generadas con la interfaz de usuario es 9.

A continuación se muestra un ejemplo de este tipo de ficheros en el que se describe el mallado de una placa plana.

```

MESH      dimension 3 ElemType Quadrilateral Nnode 9
Coordinates
 1  -0.50000000E+00 -0.50000000E+00  0.00000000E+00
 2   0.00000000E+00 -0.50000000E+00  0.00000000E+00
 3   0.50000000E+00 -0.50000000E+00  0.00000000E+00
 4   0.50000000E+00  0.00000000E+00  0.00000000E+00
 5   0.50000000E+00  0.50000000E+00  0.00000000E+00
 6   0.00000000E+00  0.50000000E+00  0.00000000E+00
 7  -0.50000000E+00  0.50000000E+00  0.00000000E+00
 8  -0.50000000E+00  0.00000000E+00  0.00000000E+00
 9  -0.25000000E+00 -0.50000000E+00  0.00000000E+00
10   0.25000000E+00 -0.50000000E+00  0.00000000E+00
11   0.50000000E+00 -0.25000000E+00  0.00000000E+00
12   0.50000000E+00  0.25000000E+00  0.00000000E+00
13   0.25000000E+00  0.50000000E+00  0.00000000E+00
14  -0.25000000E+00  0.50000000E+00  0.00000000E+00
15  -0.50000000E+00  0.25000000E+00  0.00000000E+00
16  -0.50000000E+00 -0.25000000E+00  0.00000000E+00
17   0.00000000E+00  0.00000000E+00  0.00000000E+00
18   0.00000000E+00 -0.25000000E+00  0.00000000E+00
19  -0.25000000E+00  0.00000000E+00  0.00000000E+00
20  -0.25000000E+00 -0.25000000E+00  0.00000000E+00
21   0.25000000E+00  0.00000000E+00  0.00000000E+00
22   0.25000000E+00 -0.25000000E+00  0.00000000E+00
23   0.00000000E+00  0.25000000E+00  0.00000000E+00
24  -0.25000000E+00  0.25000000E+00  0.00000000E+00
25   0.25000000E+00  0.25000000E+00  0.00000000E+00
end coordinates

Elements
 1   1   2  17   8   9  18  19  16  20
 2   2   3   4  17  10  11  21  18  22
 3   6   7   8  17  14  15  19  23  24
 4  17   4   5   6  21  12  13  23  25
end elements

```

Tabla 3.9: Formato de fichero MESH

En la primera parte del fichero, sección 'Coordinates', se encuentra el índice de cada punto con sus coordenadas en (x, y, z) . Posteriormente en la sección 'Elements' se encuentran los índices de los puntos que forman cada elemento de la malla.

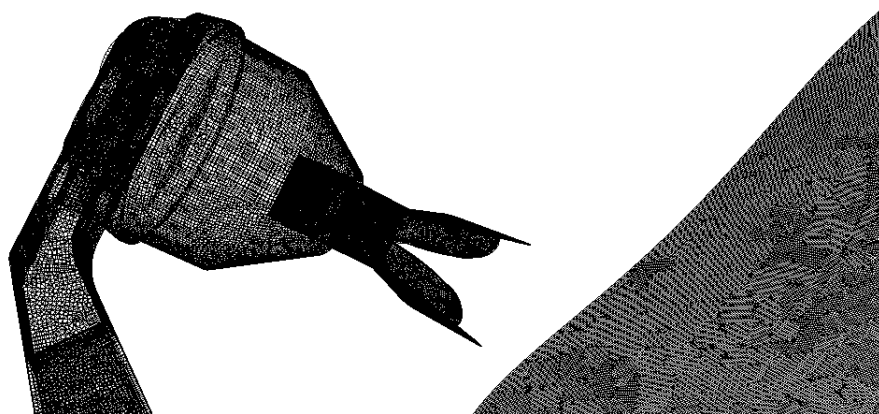


Figura 3.38: Reflector.msh: 304.729 elementos

Bibliografía

- [1] “Swing - Creating a GUI With JFC/Swing.” [Online]. Disponible en: <https://docs.oracle.com/javase/tutorial/uiswing/>
- [2] J. Zukowski, *Java AWT Reference*. O’Reilly Media Inc., 1997.
- [3] “SWT: The Standard Widget Toolkit.” [Online]. Disponible en: <https://www.eclipse.org/swt/>
- [4] “Eclipse IDE.” [Online]. Disponible en: <https://www.eclipse.org/>
- [5] “Netbeans IDE.” [Online]. Disponible en: <https://netbeans.org/>
- [6] A. Davison, *Killer Game Programming in Java*. O’Reilly Media, 2009.
- [7] D. Selman, *Java 3D Programming*. Manning Publications, 2002.
- [8] W. T. Les A. Piegl, *The NURBS Book (Monographs in Visual Communication)*. Springer, 2013.
- [9] “BeanShell - Lightweight Scripting for Java.” [Online]. Disponible en: <http://www.beanshell.org/>
- [10] “The Initial Graphics Exchange Specification (IGES) Version 6.0.” [Online]. Disponible en: <https://filemonger.com/specs/igs/devdept.com/version6.pdf>
- [11] “GiD: a universal, adaptive and user-friendly pre and post processor.” [Online]. Disponible en: <https://www.gidhome.com>

Capítulo 4

Interfaz de Usuario: Simulación Electromagnética

4.1 Introducción

En este capítulo se muestra el desarrollo de la parte de simulación electromagnética de la interfaz de usuario. Para ello, se describirá en detalle el desarrollo del módulo MOM, el resto de módulos se han implementado siguiendo la misma estructura.

El módulo MOM utiliza el núcleo electromagnético MONURBS [1] para la simulación. Esta es una herramienta informática para el análisis complejo de antenas, antenas a bordo de plataformas, sección radar, estructuras periódicas, dispositivos de microondas y circuitos.

La interfaz de usuario se encarga de recoger todos los parámetros de simulación. Una vez definida la geometría y los parámetros de simulación, la interfaz se encarga de llamar a los distintos programas de simulación escribiendo sus ficheros de entrada y ejecutándolos de manera transparente al usuario. Cuando la simulación ha terminado, la interfaz puede leer los ficheros de salida y mostrarlos al usuario de manera amigable.

Una vez concluida la parte de simulación electromagnética, en el siguiente capítulo se mostrará un caso de validación de la interfaz de usuario, describiendo todo el proceso de construcción y diseño de un reflectarray novedoso.

4.2 Módulos de Simulación Electromagnética

La interfaz de usuario está dividida en varios módulos de simulación, cada módulo se utiliza para resolver distintos problemas de simulación electromagnética. A continuación se muestra una breve descripción de cada uno de los módulos que componen la interfaz de usuario:

- **GTD.** Es un módulo muy eficiente para el análisis de antenas embarcadas en satélites, aviones y otros modelos complejos. El kernel del módulo GTD está basado en Óptica Geométrica (GO), Óptica Física (PO) y en la Teoría Uniforme de la Difracción (UTD) [2].
- **MOM.** El módulo MOM está enfocado al análisis y diseño de antenas. Utiliza el kernel MONURBS que está basado en el Método de los Momentos [3] (MoM) con MLFMM (multilevel fast multipole method), proporcionando resultados excelentes.
- **PO.** El módulo PO se utiliza para el análisis de la Sección Radar (RCS) de grandes objetos eléctricamente complejos. Su kernel se basa en la Óptica Física (PO) [4].
- **MONCROS.** Este módulo se utiliza para la obtención de la Sección Radar (RCS) mediante el Método de los Momentos (MoM). Incorpora además la posibilidad de utilizar Funciones de Base Características (CBFs) y la técnicas de Descomposición de Dominio, con objeto de mejorar la eficiencia [5].
- **PERIODICAL STRUCTURES.** El módulo de estructuras periódicas se utiliza para calcular los coeficientes de reflexión y transmisión de una estructura periódica con una o varias capas de material y metalizaciones aplicando un método muy eficiente para el cálculo de las funciones de Green y utilizando MoM [6].
- **ISAR.** El módulo ISAR (Inverse Synthetic Aperture Radar) está enfocado a la obtención de imágenes ISAR, perfiles y mapas de radiación de objetivos eléctricamente grandes y complejos.
- **IR.** El módulo IR contiene una herramienta rápida y precisa para el análisis térmico de grandes objetivos en escenarios con tierra o mar, así como la generación de imágenes de infrarrojos y el análisis de los parámetros térmicos y de radiación.
- **GTD-PO.** El módulo GTD-PO permite el análisis de la RCS de cuerpos complejos que son eléctricamente grandes y el estudio de la propagación en entornos urbanos y comunicación V2V [7]. Es una hibridación de los módulos de GTD y PO.
- **CHAFF.** El módulo CHAFF sirve para el análisis 3D de la Sección Radar (RCS) en nubes de dipolos rectangulares, esféricos o arbitrarios.

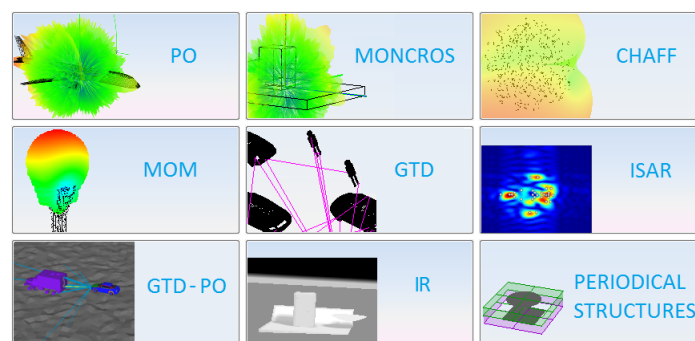


Figura 4.1: Módulos de simulación electromagnética

4.2.1 Estructura del programa

La implementación de todos los módulos de la interfaz se encuentra dentro del paquete Modules, que se puede ver en la figura 3.2.

Las funciones comunes a distintos módulos, como pueden ser las ventanas de resultados, se encuentran en el paquete 'Util'. Como se verá posteriormente, la clase más importante de cada módulo extiende de 'Project' y contiene todos los datos de la interfaz relacionados con ese módulo. Para que el mantenimiento sea sencillo todos los módulos comparten la misma estructura: Data, Frames, Menus y Tree.

A continuación se muestra un esquema de la estructura del programa y posteriormente se explica el propósito de cada una de las partes.

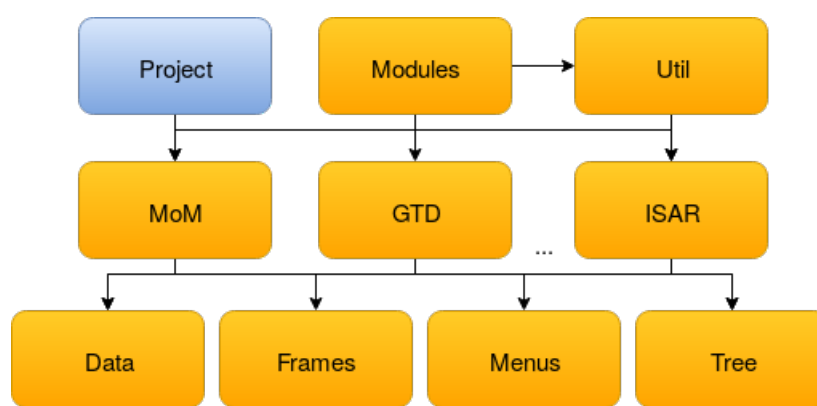


Figura 4.2: Estructura del paquete Modules

- **Data.** Contiene clases que almacenan todos los datos del módulo que son recogidos por las distintas ventanas y que hay que guardar en el proyecto. Es donde se encuentra la clase principal de cada módulo que extiende de 'Project'.
- **Frames.** Es donde se encuentran todas las ventanas propias del módulo, separadas por su menú correspondiente. También contiene el proceso de mallado y simulación, dentro de los paquetes Meshing y Calculate respectivamente.
- **Menus.** En este paquete se encuentra el menú de cada módulo que se añade a la barra de menús de la interfaz de usuario cuando se abre o se crea un proyecto nuevo.
- **Tree.** Contiene los datos y la funcionalidad de cada módulo que se añade al árbol de la interfaz de usuario.

4.2.2 El Proyecto de Simulación

En el capítulo anterior dentro de la clase 'MainFrame', en la figura 3.7, se encuentra el atributo 'Project' que contiene los datos de un proyecto de simulación.

La clase 'Project' es una clase abstracta y contiene los datos comunes a todos los módulos, como por ejemplo el tipo de proyecto, la geometría y el historial de comandos.

Esta clase implementa la interfaz 'Externalizable', lo que quiere decir que puede almacenar y recuperar sus datos de un fichero binario.

Para cada módulo existe una clase que se encarga de almacenar todos los datos de simulación y extiende de 'Project'. En el caso del modulo MoM la clase es 'MomData'. Esta clase tiene que sobrescribir los métodos abstractos *initModule* y *closeModule* que se llaman al abrir y cerrar un proyecto e implementar la interfaz 'Externalizable'.

Además de los datos del módulo y la geometría, un proyecto tiene que almacenar los resultados que ha producido el núcleo al ejecutar la simulación. Para ello, en la interfaz de usuario cada proyecto tiene asociado un identificador único. Al abrir un nuevo proyecto se crea un directorio en la carpeta *projects* del directorio de instalación que tiene como nombre ese identificador. En ese directorio se almacena tanto el fichero binario con los datos de la clase 'Project' (*project.dat*) como los ficheros que han generado las distintas simulaciones.

Para el usuario un proyecto se guarda como un fichero con extensión *.nfp* (newFASANT Project). Para generar ese fichero la interfaz comprime la carpeta del proyecto en formato ZIP y renombra su extensión.

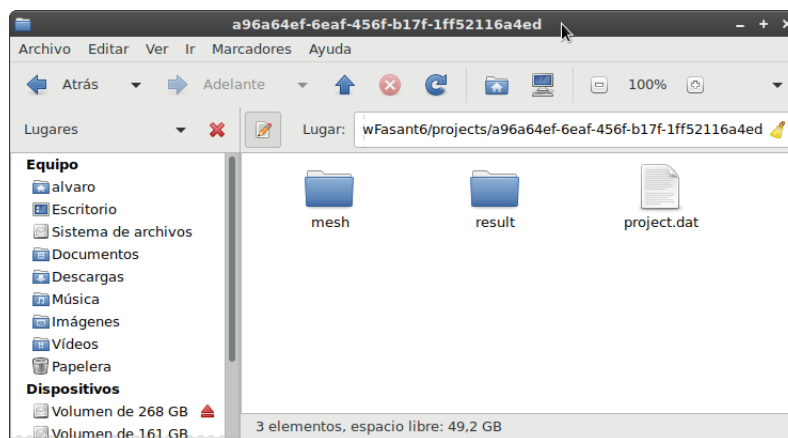


Figura 4.3: Directorio de un proyecto de simulación

Para abrir un proyecto de simulación, se descomprime el archivo *.nfp* en el directorio *projects* y se carga el fichero binario *project.dat* con los datos de la interfaz. Cuando se cierra un proyecto se borra su directorio de la carpeta *projects*.

4.3 El núcleo electromagnético MONURBS

MONURBS es una herramienta muy potente capaz de resolver problemas eléctricamente muy grandes. Proporciona análisis rápidos y precisos de estructuras arbitrarias tanto metálicas como dieléctricas.

El código del núcleo de MONURBS, escrito en FORTRAN, es un código de onda completa basado en el Método de los Momentos (MoM). El MoM es una de las técnicas más

populares para analizar problemas de radiación y de dispersión; sin embargo, este método tiene un gran coste computacional cuando el tamaño eléctrico de la estructura bajo análisis es muy grande. Para tratar este tipo de estructuras se aplica el método rápido de multipolos multinivel (MLFMM) [8]. MONURBS incluye una técnica híbrida basada en el MoM y en la técnica de física óptica (PO) propuesta para analizar antenas muy grandes. PO es un método numérico de alta frecuencia basado en corrientes. Esta combinación híbrida es la solución ideal cuando un cuerpo es demasiado costoso de analizar con MLFMM [9]. La idea de utilizar PO es reducir el tiempo de cálculo y los requerimientos de memoria con respecto al MoM tradicional cuando se analizan estructuras grandes a altas frecuencias [10]. Respecto a las técnicas numéricas, MONURBS puede resolver ecuaciones integrales de campo eléctrico (EFIE), ecuaciones integrales de campo magnético (MFIE) y ecuaciones integrales de campo combinado (CFIE). A pesar de la complejidad matemática de las técnicas incluidas en el núcleo, todas las fases del proceso de resolución han sido paralelizadas utilizando los paradigmas MPI [11] y OpenMP [12].

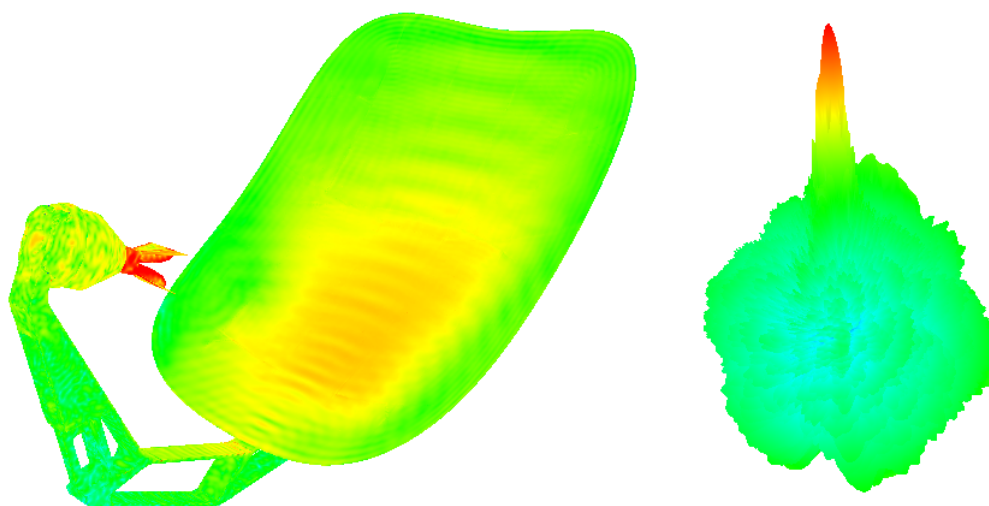


Figura 4.4: Reflector analizado con MONURBS

4.4 Módulo MOM

El módulo MOM se utiliza para el análisis preciso de antenas mediante el método de los momentos, utiliza el núcleo de simulación electromagnética MONURBS descrito en la sección anterior. Para realizar la simulación, se necesitan recoger una serie de parámetros de entrada y realizar un mallado de la geometría, llamando al programa de mallado.

Todo el proceso empieza con la creación de un nuevo proyecto. Al hacer click en el botón 'New Project' de la interfaz se muestra la pestaña de la figura 4.1. Al seleccionar MOM, la interfaz crea un nuevo objeto de tipo 'MomData' que guarda en la clase principal 'MainFrame' y llama a la función *initModule*.

El método *initModule* se encarga de añadir los nuevos menús para el manejo del módulo y de inicializar el árbol; en el caso del módulo MOM se inicializan también las antenas y se añaden comandos que solo se pueden utilizar dentro de este módulo para crear geometría específica. En la siguiente figura se muestra la interfaz con el módulo MOM abierto, puede compararse con la figura 3.5 donde se muestra la interfaz sin ningún módulo abierto.

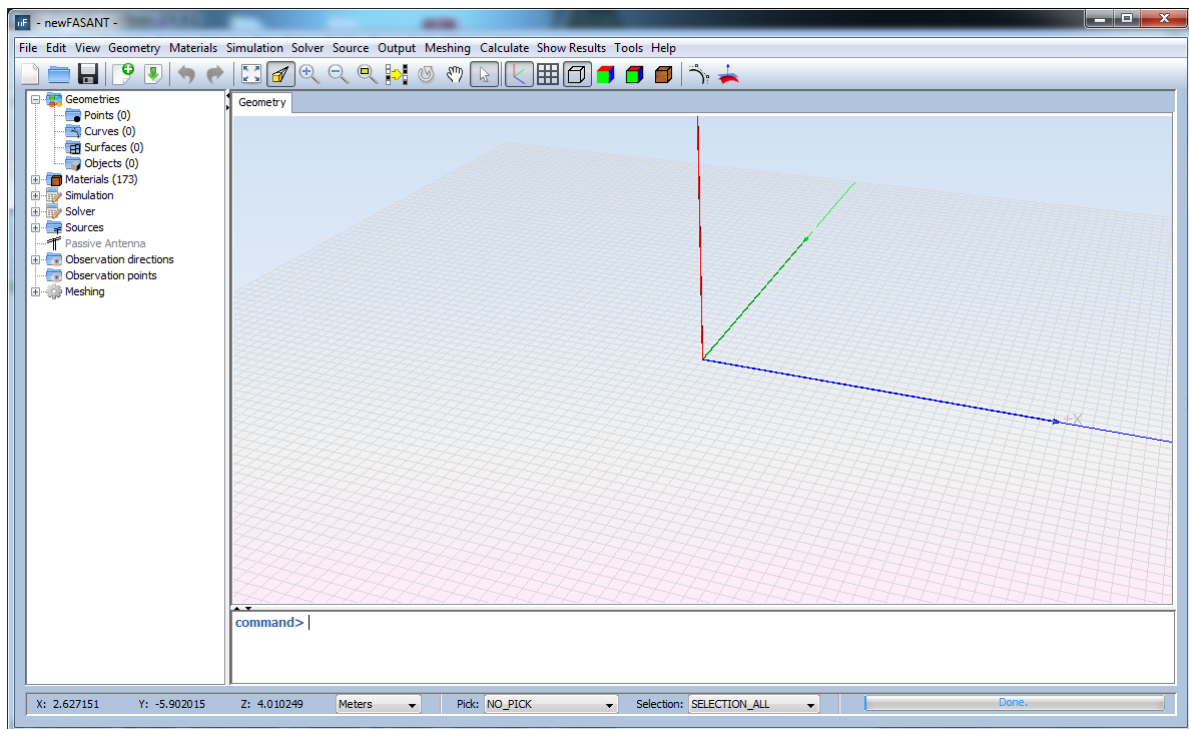


Figura 4.5: Interfaz de usuario con el módulo MOM

Como puede observarse, se han añadido los menús Materials, Simulation, Solver, Source, Output, Meshing, Calculate y Show Results a la barra de menús de la interfaz. En el árbol se han añadido elementos para que el usuario pueda ver y editar los datos principales de simulación de una manera más cómoda que accediendo a cada ventana de parámetros.

Para que el aprendizaje del programa sea sencillo para un nuevo usuario, en el constructor de la clase 'MomData' se guardan todos los parámetros por defecto, así no hace falta pasar por todas las ventanas de introducción de parámetros para realizar una simulación.

4.4.1 Ejemplo: Simulación de una bocina

En esta sección se describe un ejemplo sencillo de uso del módulo MOM donde se pueden ver las ventanas principales del programa. Posteriormente se describirá en detalle las secciones más importantes del módulo.

Una vez que el usuario ha creado un nuevo proyecto de MOM, como se muestra en la sección anterior, hay que definir la geometría que se va a simular. Para ello el usuario utilizará el programa de diseño 3D que se ha desarrollado en la interfaz de usuario.

En el módulo MOM se han creado una serie de antenas predefinidas que se pueden utilizar para construir el caso de simulación. Estas antenas se encuentran dentro del menú Source y el usuario solo tiene que añadirlas al panel 3D. Para ello puede utilizar su comando correspondiente (solo disponible en este módulo) o la pestaña donde se describen sus parámetros. En este ejemplo se define una bocina piramidal utilizando su pestaña correspondiente.

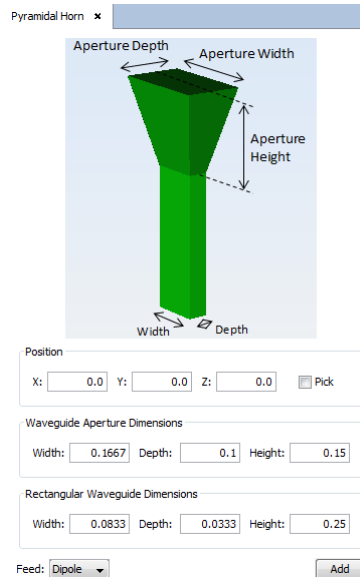


Figura 4.6: Pestaña para introducir los parámetros de una bocina piramidal

En este caso, el tamaño de la bocina se ajusta automáticamente a la frecuencia de simulación y se puede alimentar de manera automática. Si se activa el Pick, cuando se hace click con el ratón en el panel 3D se actualiza la posición de la bocina. En la interfaz de usuario todas las primitivas se crean en el plano XY por defecto, para variar su orientación se utiliza el plano de referencia del Panel 3D.

La interfaz muestra una previsualización de como va a quedar la geometría en el Panel 3D que se va actualizando según se cambian los parámetros de la pestaña. Cuando se hace click en el botón Add, la geometría definitiva se añade al panel. En la siguiente figura se muestra la bocina correspondiente al ejemplo añadida al panel y la previsualización de una segunda bocina.

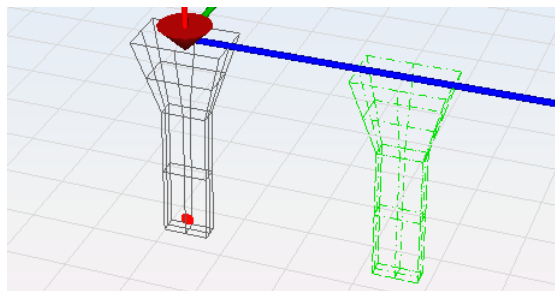


Figura 4.7: Bocina añadida al panel junto a la previsualización de otra bocina

En la figura 4.7 se puede observar que se ha añadido la alimentación a la bocina, en este caso es un dipolo. Posteriormente se describen los distintos tipos de alimentación del módulo MOM.

Una vez definida la geometría se definen los distintos parámetros de entrada del núcleo de simulación. Para ello se han creado una serie de pestañas donde se van introduciendo los parámetros y guardando en el proyecto dentro de la clase 'MomData'. Los menús se han dispuesto de izquierda a derecha según el orden natural de introducción de los parámetros del módulo pero no es obligatorio hacerlo así.

Los parámetros más importantes son los parámetros de simulación, en el menú Simulation, donde se especifica la frecuencia de simulación. En la siguiente figura se muestra la pestaña.

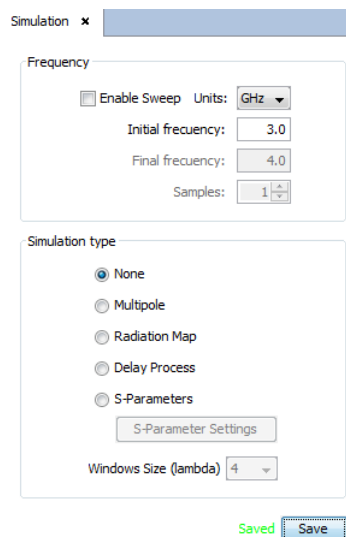


Figura 4.8: Pestaña de 'Simulation Parameters'

En el ejemplo se va a realizar una simulación de la bocina anterior a 3GHz. Al pulsar en el botón Save se almacenan los datos correspondientes a la pestaña dentro del proyecto, en su clase correspondiente. Al abrir la ventana, se ha realizado el proceso contrario, es decir, se han cogido los datos correspondientes almacenados en 'MomData' y se han cargado en la pestaña, en este caso los que se han definido en el constructor por defecto.

El siguiente menú es Solver donde se especifican los parámetros de configuración del núcleo de simulación como el modo de paralelización, el número máximo de iteraciones, etc. Aunque en este caso se han utilizado todos los parámetros por defecto, en la figura 4.9 se muestra la pestaña para dar una idea de las opciones disponibles. El funcionamiento de todas las pestañas de introducción de parámetros es similar y, en este caso, sus datos serán almacenados dentro de la clase 'DataSolver' en 'MomData'.

Figura 4.9: Pestaña de 'Solver Parameters'

El menú Source se utilizó para definir la antena al principio del ejemplo, por lo que el siguiente paso es definir la salida de datos mediante el menú Output.

En este ejemplo se va a realizar una observación en campo lejano definiendo en coordenadas esféricas el corte $\varphi = 0$ grados con una variación en $\theta = [-90, 90]$ grados de 181 puntos.

Theta cut	Initial phi	Increment	Samples	Final phi

Phi cut	Initial theta	Increment	Samples	Final theta
0.0	-90.0	1.0	181	90.0

Figura 4.10: Pestaña de 'Observation Directions'

El usuario puede seleccionar en esta pestaña los cortes de salida que desee, posteriormente se verá que también es posible seleccionar puntos de observación en campo cercano.

Por defecto, además de los cortes definidos por el usuario, se calcula el diagrama de radiación y una serie de resultados que se muestran a continuación.

En este momento, el usuario ha introducido los parámetros necesarios para empezar el proceso de simulación. El proceso empieza con el mallado de la geometría en el menú Meshing. La interfaz, a partir de los parámetros de mallado que se configuran en la siguiente ventana escribirá los ficheros necesarios y llamará al mallador.

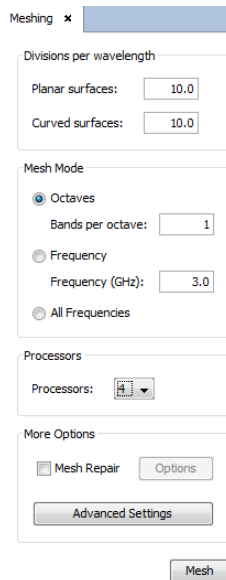


Figura 4.11: Pestaña de 'Mesh Parameters'

Una vez comienza el proceso de mallado se muestra un *log* en la parte derecha de la interfaz junto con un porcentaje en la barra de progreso, estos datos se cogen de la salida estándar del programa mallador.

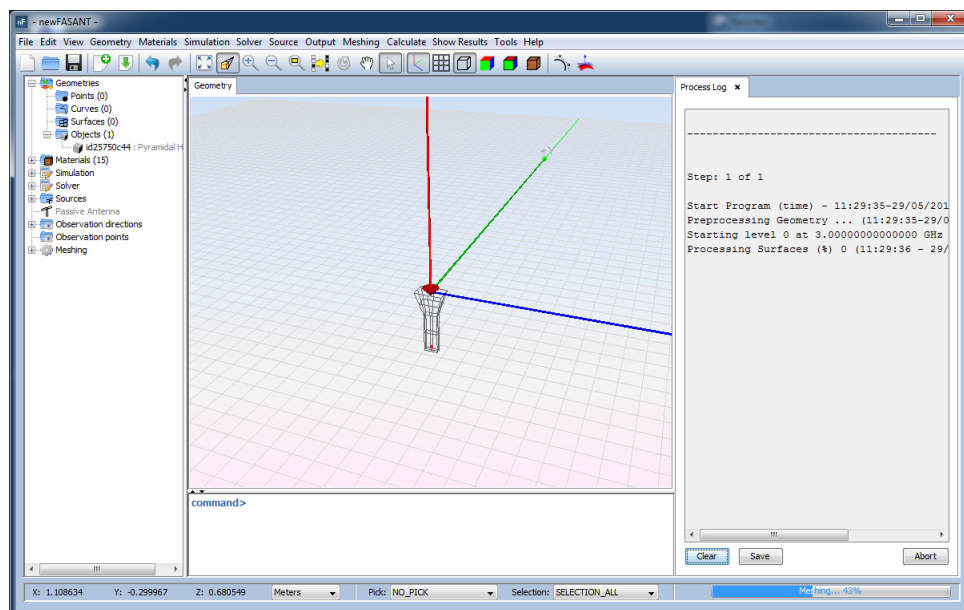


Figura 4.12: Proceso de mallado

Si la geometría fuera paramétrica o se hiciera un barrido en frecuencia, la interfaz se encarga de llamar al mallador tantas veces como sea necesario, los resultados del mallador se almacenan en la carpeta *projects* ordenados por paso y frecuencia para utilizarlos posteriormente en la simulación.

En la siguiente figura se muestra el resultado del proceso de mallado, que puede verse también a través del menú Meshing. Recordar que las ventanas de resultados se muestran en el panel central de pestañas de la interfaz de usuario.

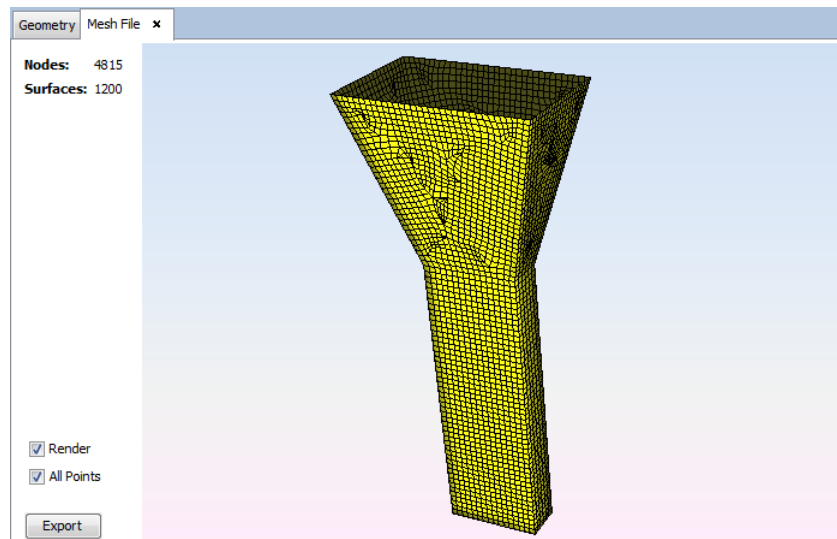


Figura 4.13: Resultados del proceso de mallado

En la mayoría de programas de simulación electromagnética el proceso de mallado y simulación se realiza de forma conjunta. Sin embargo, en la interfaz de usuario propuesta en esta tesis, se realiza de forma separada, ya que es posible cambiar varios parámetros de la simulación sin tener que volver a mallar. El usuario puede ver en qué casos es necesario volver a mallar la geometría antes de realizar la simulación, como se muestra en la siguiente figura.

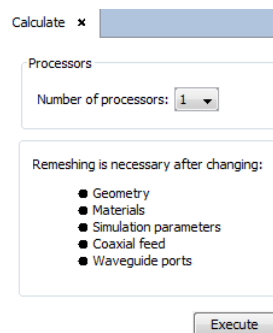


Figura 4.14: Pestaña de 'Calculate'

El funcionamiento interno del proceso de simulación es similar al proceso de mallado, la interfaz se encarga de copiar los ficheros de mallado correspondientes al directorio de

simulación y de escribir los ficheros de entrada del núcleo electromagnético MONURBS. Si se realiza una simulación paramétrica o un barrido en frecuencia, repetirá el proceso las veces que sea necesario almacenando todos los resultados dentro del directorio *result* del proyecto, que se vio en la figura 4.3.

De la misma forma, la salida estándar del núcleo se muestra en la pestaña Process Log de la interfaz de usuario y su porcentaje en la barra de tareas. Si el proceso escribe algo en la salida de errores la interfaz escribe el error y abortará la simulación. También se puede abortar la simulación de manera manual pulsando el botón Abort, en este caso, la interfaz comunica al proceso que debe terminar mediante una señal.

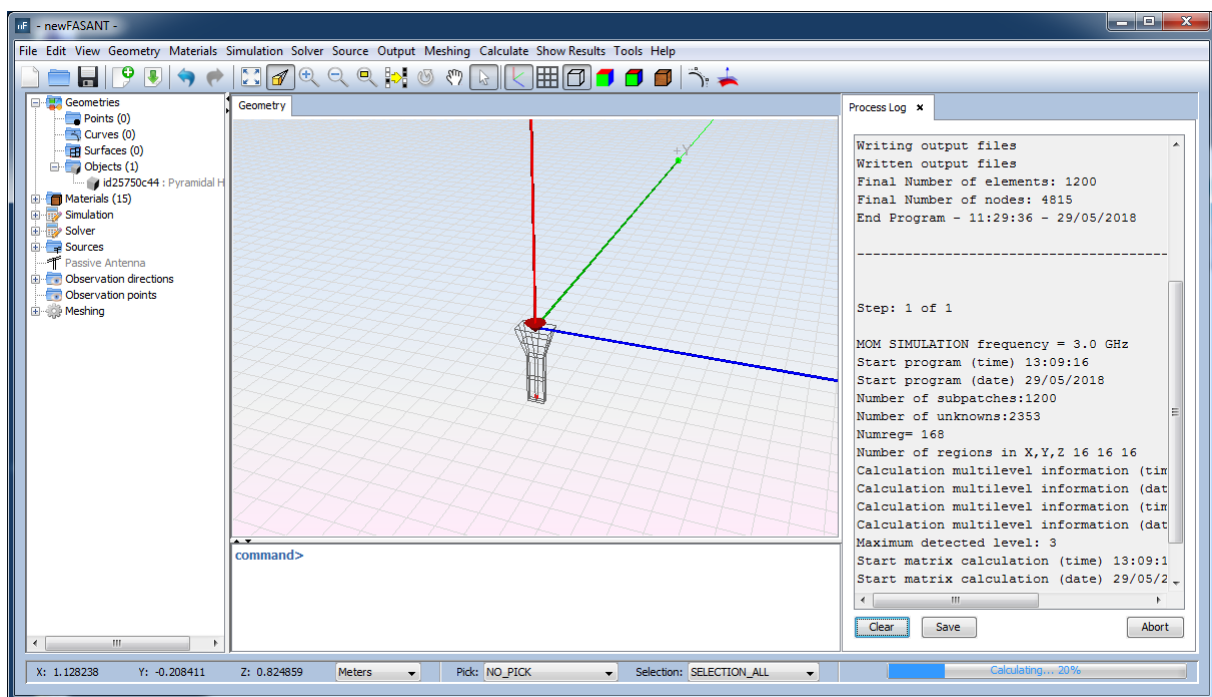


Figura 4.15: Proceso de simulación

Una vez concluye el proceso de simulación, se habilitan los resultados correspondientes en el menú Show Results. Las ventanas de resultados se encargan de leer los distintos ficheros de salida del núcleo electromagnético y con los datos del proyecto, muestran los resultados de manera ordenada. Como se ha visto al principio de este capítulo, el proyecto puede almacenarse y abrirse de nuevo en otra interfaz y contiene todos los datos del mallado y la simulación.

Los resultados principales que muestran la interfaz son:

- **Ficheros de texto.** Muestra los ficheros de salida del núcleo electromagnético lo que puede ser útil para postprocesarlos externamente.
- **Gráficas.** Permite visualizar y comparar resultados mediante gráficas lineales o polares. Se pueden importar y exportar datos.

- **Resultados en 3D.** Muestra otro panel 3D donde se puede dibujar el diagrama de radiación o la distribución de corrientes sobre el modelo geométrico.

En la siguiente figura se muestra el fichero de texto para el corte $\varphi = 0$ seleccionado en el ejemplo:

THETA (deg)	PHI (deg)	E_V (V/m)	E_H (V/m)
-90.000000	0.000000	-.9300E+00 0.1797E+02	0.2313E+03 0.4035E+02
-89.000000	0.000000	0.2273E+01 0.1894E+02	0.2534E+03 0.3938E+02
-88.000000	0.000000	0.5501E+01 0.1904E+02	0.2743E+03 0.3179E+02
-87.000000	0.000000	0.8509E+01 0.1834E+02	0.2916E+03 0.1830E+02
-86.000000	0.000000	0.1110E+02 0.1697E+02	0.3094E+03 0.5200E+00
-85.000000	0.000000	0.1313E+02 0.1515E+02	0.3087E+03 -.1930E+02
-84.000000	0.000000	0.1454E+02 0.1301E+02	0.3072E+03 -.3869E+02
-83.000000	0.000000	0.1595E+02 0.1081E+02	0.2999E+03 -.5533E+02
-82.000000	0.000000	0.1561E+02 0.8687E+01	0.2885E+03 -.6748E+02
-81.000000	0.000000	0.1544E+02 0.6763E+01	0.2753E+03 -.7425E+02
-80.000000	0.000000	0.1495E+02 0.5095E+01	0.2628E+03 -.7576E+02
-79.000000	0.000000	0.1426E+02 0.3694E+01	0.2531E+03 -.7306E+02
-78.000000	0.000000	0.1347E+02 0.2528E+01	0.2479E+03 -.6805E+02
-77.000000	0.000000	0.1263E+02 0.1546E+01	0.2479E+03 -.6304E+02
-76.000000	0.000000	0.1175E+02 0.6890E+00	0.2528E+03 -.6045E+02
-75.000000	0.000000	0.1082E+02 -.8738E-01	0.2615E+03 -.6243E+02
-74.000000	0.000000	0.9804E+01 -.8022E+00	0.2724E+03 -.7049E+02
-73.000000	0.000000	0.8665E+01 -.1444E+01	0.2831E+03 -.8536E+02
-72.000000	0.000000	0.7369E+01 -.1974E+01	0.2915E+03 -.1069E+03
-71.000000	0.000000	0.5909E+01 -.2328E+01	0.2957E+03 -.1343E+03
-70.000000	0.000000	0.4308E+01 -.2438E+01	0.2943E+03 -.1659E+03
-69.000000	0.000000	0.2622E+01 -.2241E+01	0.2869E+03 -.2002E+03
-68.000000	0.000000	0.9363E+00 -.1693E+01	0.2736E+03 -.2352E+03
-67.000000	0.000000	-.6467E+00 -.7847E+00	0.2552E+03 -.2697E+03
-66.000000	0.000000	-.2019E+01 0.4588E+00	0.2331E+03 -.3028E+03
-65.000000	0.000000	-.3083E+01 0.1973E-01	0.2069E+03 -.3341E+03
-64.000000	0.000000	-.3762E+01 0.3663E+01	0.1840E+03 -.3644E+03
-63.000000	0.000000	-.4018E+01 0.5410E+01	0.1597E+03 -.3945E+03
-62.000000	0.000000	-.3847E+01 0.7090E+01	0.1368E+03 -.4259E+03
-61.000000	0.000000	-.3285E+01 0.8580E+01	0.1155E+03 -.4606E+03
-60.000000	0.000000	-.2407E+01 0.9780E+01	0.9547E+02 -.5001E+03
-59.000000	0.000000	-.1310E+01 0.1062E+02	0.7569E+02 -.5459E+03
-58.000000	0.000000	-.1064E+00 0.1104E+02	0.5501E+02 -.5994E+03
-57.000000	0.000000	0.1086E+01 0.1107E+02	0.3175E+02 -.6611E+03
-56.000000	0.000000	0.2159E+01 0.1071E+02	0.4415E+01 -.7313E+03
-55.000000	0.000000	0.3024E+01 0.1004E+02	-.2342E+02 -.8099E+03
-54.000000	0.000000	0.3614E+01 0.9121E+01	-.6811E+02 -.8962E+03
-53.000000	0.000000	0.3893E+01 0.8059E+01	-.1155E+03 -.9894E+03

Figura 4.16: Resultados de la simulación: Ficheros de texto

Si se ha realizado una simulación paramétrica o un barrido en frecuencia, la interfaz muestra una ventana para seleccionar el fichero correspondiente a una frecuencia y un paso determinado; como todos los ficheros se han guardado dentro del directorio del proyecto en la carpeta Resul de forma ordenada por paso y por frecuencia, la interfaz selecciona el fichero correspondiente y lo muestra al usuario.

Para el resto de resultados, al abrir la ventana correspondiente la interfaz lee los ficheros de todos los pasos y todas las frecuencias y los almacena en clases de datos de tipo 'DataResult'. A partir de estos ficheros la interfaz realiza un postprocesado donde calcula los distintos resultados que pueden seleccionarse en la ventana.

También existe un módulo de postprocesado donde el usuario puede aplicar sus propias fórmulas a los resultados utilizando la evaluación de funciones mediante el paquete BeanShell que se vio en la Sección 3.8.2.

En la figura 4.17 se muestra la gráfica correspondiente al corte que se seleccionó en el ejemplo, en este caso se muestra el valor del campo en componentes Lineales (dB) de las dos componentes ETheta y EPhi. Estos datos han sido calculados a partir de los valores del fichero de la figura anterior.

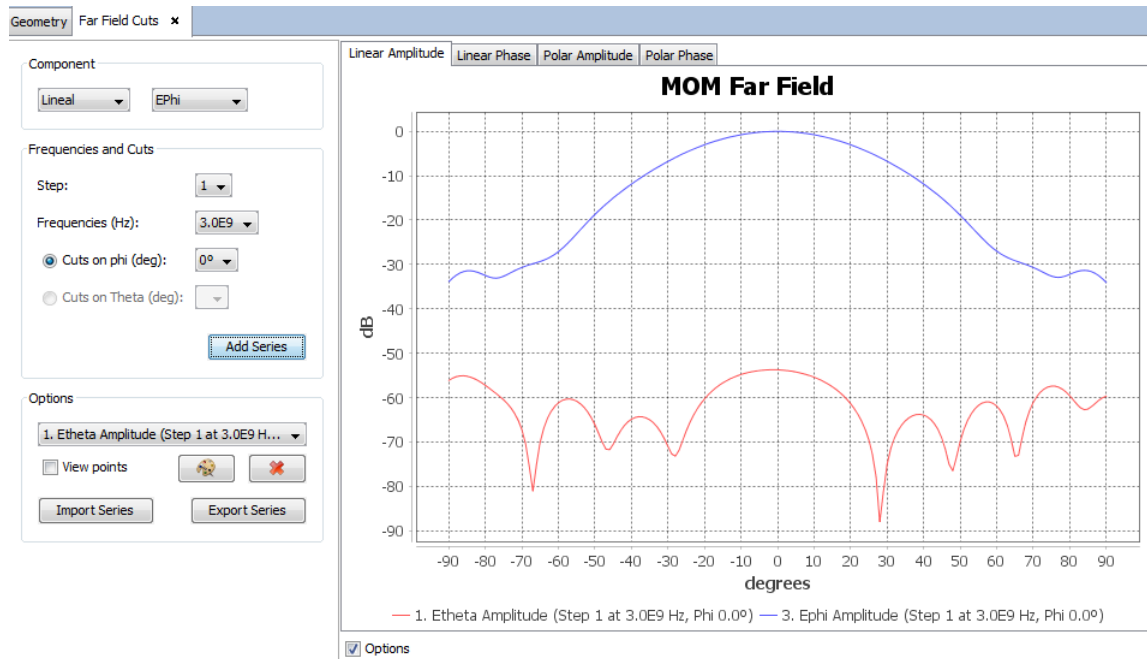


Figura 4.17: Resultados de la simulación: Gráficas

En la ventana de gráficas el usuario puede seleccionar el paso, la frecuencia o el corte que quiere representar y los distintos tipos de resultados que se han calculado. Además, puede importar y exportar datos de otras gráficas para realizar comparaciones. El usuario puede cambiar de gráfica con las pestañas de la parte de arriba para visualizar la amplitud o la fase o las gráficas en forma polar.

A continuación se muestra el diagrama de radiación en 3D, que es calculado por defecto por el núcleo electromagnético si no se indica lo contrario en el menú Solver, en la figura 4.9.

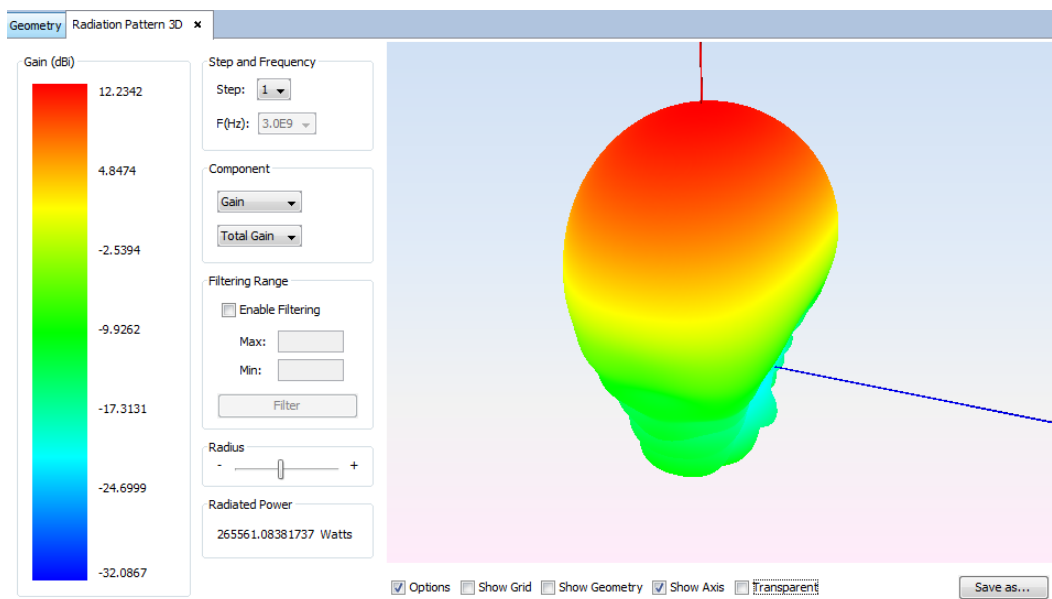


Figura 4.18: Resultados de la simulación: Diagrama de radiación

Para dibujar el diagrama de radiación el núcleo electromagnético ha calculado todos los cortes de $\varphi = [0, 360]$ grados con una variación en $\theta = [0, 180]$ cada 0,5 grados. La interfaz busca el valor máximo y el mínimo para crear la escala de colores y crea un IndexedQuadArray de Java 3D donde asigna el radio y el color correspondiente a cada punto según su valor. También se pueden visualizar los cortes del diagrama en forma de gráfica.

Por último se puede ver la distribución de la carga y la corriente sobre la geometría. Para dibujarlo se utiliza la malla y el valor de la corriente o carga en cada uno de sus puntos según la escala de colores.

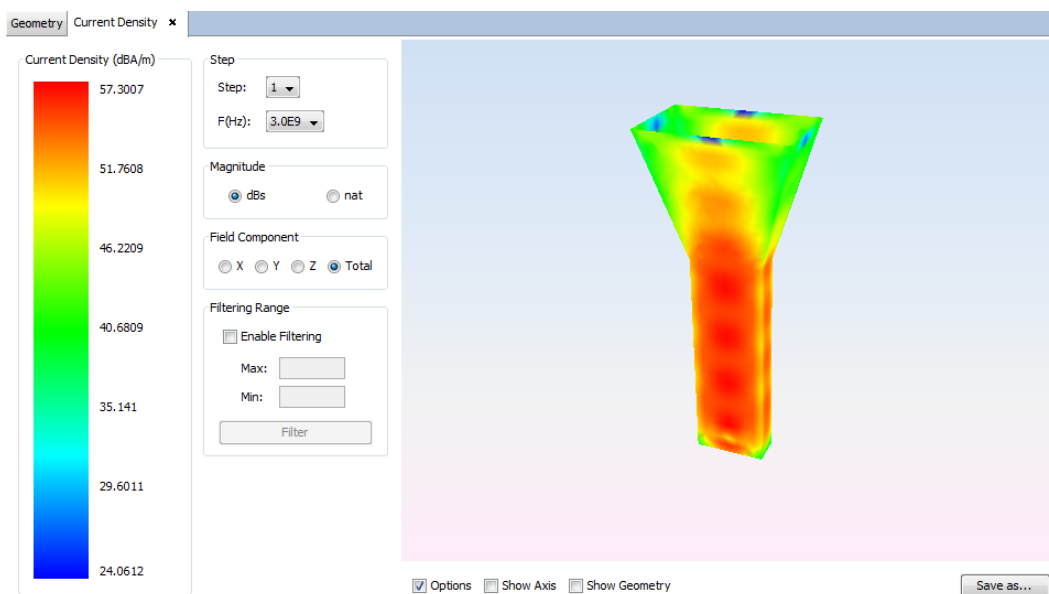


Figura 4.19: Resultados de la simulación: Corrientes

4.4.2 Materiales

Por defecto, la geometría analizada con el núcleo electromagnético MONURBS es de tipo PEC (Perfect Electric Conductor). En la interfaz de usuario es posible definir distintos materiales dieléctricos y asignarlos a superficies. Cuando se va a realizar la simulación, la interfaz escribe el fichero de entrada materiales.dat con las propiedades de los materiales de cada superficie.

En la interfaz de usuario se han agrupado los materiales de todos los módulos y se almacenan en una lista dentro de la clase 'Project'. Al abrir o cerrar un proyecto, o crear un proyecto nuevo, la lista de materiales se lee y almacena en un fichero serializado llamado Materiales.db que se encuentra en el directorio de instalación de la interfaz. Al ser un atributo de la clase 'Project', la lista de materiales también se almacena con los datos del proyecto de simulación; esto permite compartir proyectos con nuevos materiales entre distintas interfaces.

Todos los materiales extienden de la clase abstracta 'MaterialDielectrico' que contiene

el nombre y el color del material. La lista de materiales es un HashMap que tiene como clave el nombre del material.

Dentro de cada superficie se encuentra un atributo de tipo 'MaterialObjeto' que contiene el nombre del material asignado a esa superficie y el grosor.

El núcleo electromagnético MONURBS es compatible con dos tipos de materiales:

- **MaterialEpsilonMu.** Se definen las propiedades electromagnéticas del material mediante su permitividad eléctrica relativa y su permitividad magnética relativa.
- **MaterialCoefficient.** El material se define mediante su coeficiente de reflexión.

Para añadir un nuevo material a la base de datos de la interfaz en el módulo MOM se utiliza la pestaña de la siguiente figura.

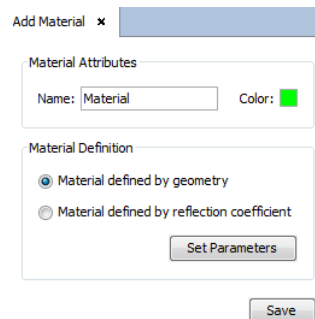


Figura 4.20: Definición de materiales

Una vez seleccionado el nombre y el color se elige el tipo de material y se introducen sus parámetros en la ventana que aparece al pulsar el botón Set Parameters.

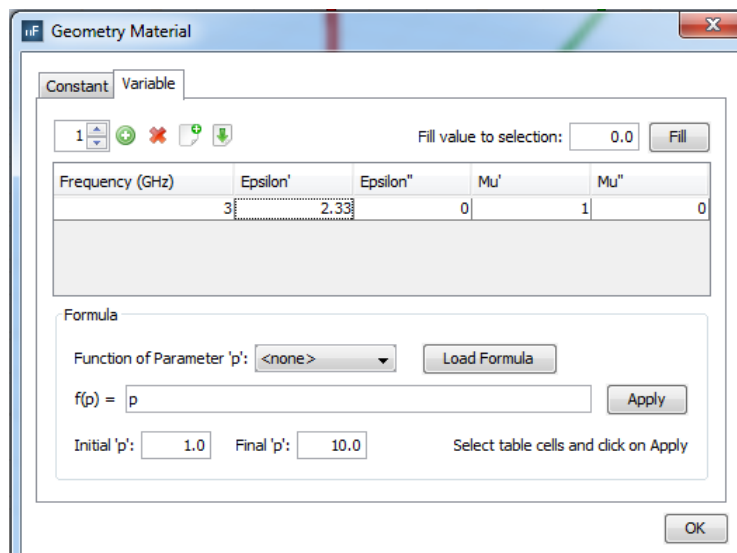


Figura 4.21: Parámetros de los materiales

El material definido por geometría puede ser variable en frecuencia, como se muestra en la ventana. Dependiendo de la frecuencia de simulación, el núcleo electromagnético

seleccionará las propiedades del material interpolando entre los valores de la tabla. Los valores se pueden importar y exportar en un fichero de texto. La parte de abajo de la ventana se utiliza para poder utilizar una fórmula para definir los valores de la tabla; se puede crear una función $f(p)$ que se evaluará utilizando BeanShell, igual que en los parámetros, y se aplicará a las celdas que se han seleccionado en la tabla.

Para editar un material se utiliza la misma ventana, solo que se cargan los valores del material seleccionado en una lista desplegable cuando se pulsa el botón Set Parameters. Los materiales también se pueden borrar de la base de datos.

Para asignar un material a una superficie o a todas las superficies de un objeto se utiliza la pestaña de la siguiente figura.

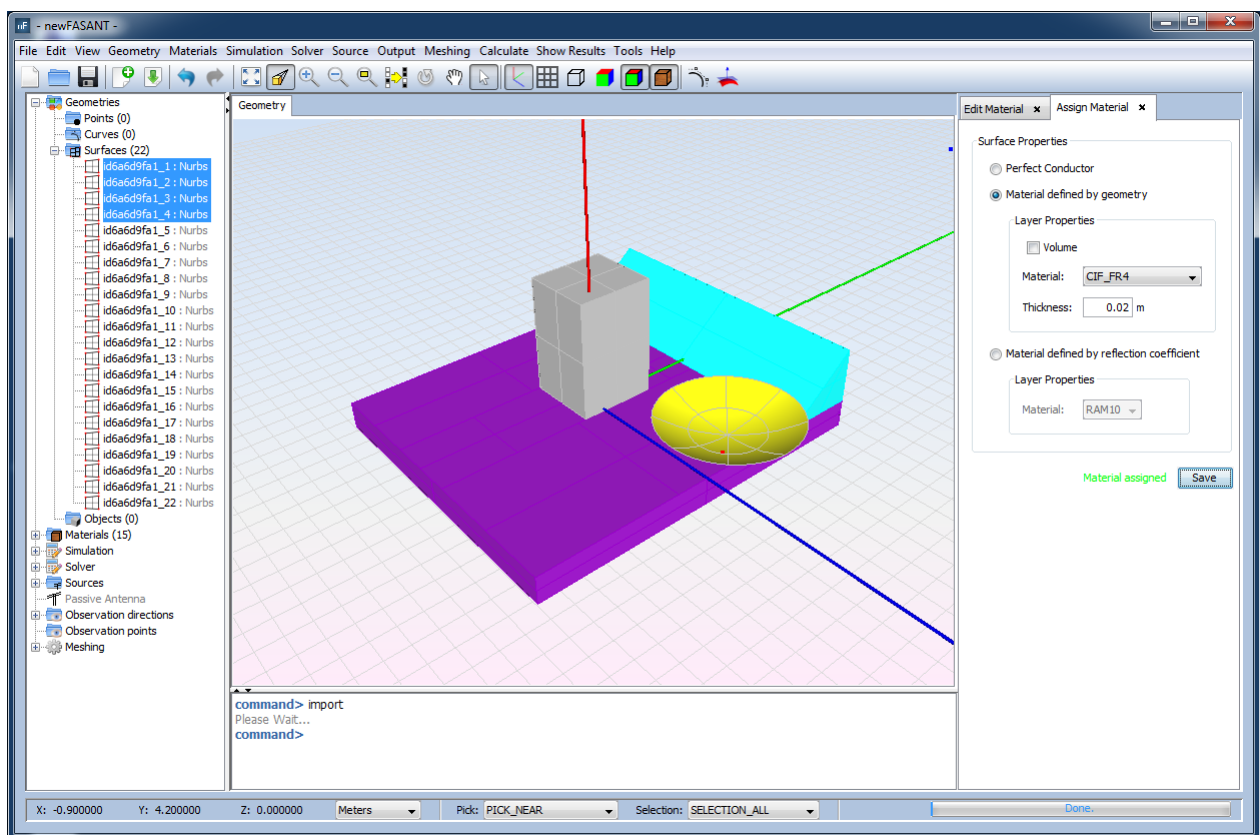


Figura 4.22: Pestaña para asignar material a una superficie

Primero se seleccionan los objetos utilizando el método de selección de la interfaz de usuario y después el material que se quiere asignar en la pestaña. Al pulsar el botón Save se asigna el material seleccionado al objeto. Para ver el material que se ha asignado a cada objeto se puede seleccionar y pulsar el botón Properties del menú Materials. También se pueden visualizar las superficies por el color del material asignado en el menú View.

Para visualizar los objetos con el color del material, se cambia la apariencia de todos los Shape3D, que por defecto tienen el color de la capa en la que se encuentran, al color del material que tienen asignado; esto también se controla al seleccionar y deseleccionar un objeto.

4.4.3 Parámetros de Simulación

En los parámetros de simulación se configuran los parámetros generales del núcleo electromagnético, como el tipo de simulación o la frecuencia. Estos datos se almacenan en la clase del proyecto 'MomData' y se utilizarán para escribir los ficheros de entrada de los programas externos que realizan la simulación. En el módulo MOM existen dos ventanas principales de parámetros de simulación que corresponden al menú Simulation y Solver respectivamente.

4.4.3.1 Configuración de la simulación

Los parámetros de simulación principales se definen en pestaña que se abre al pulsar en el menú Simulation. Lo más importante es la frecuencia o frecuencias de simulación; es conveniente que el usuario la defina después de crear el proyecto ya que muchas antenas se crean con dimensiones por defecto para que funcionen correctamente a esa frecuencia.

En la siguiente figura se muestra la pestaña de parámetros de simulación.

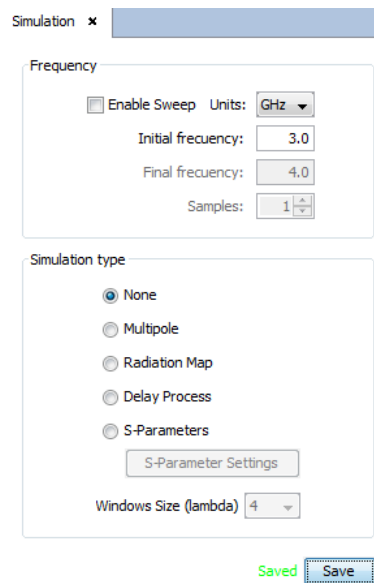


Figura 4.23: Pestaña de parámetros de simulación

En la pestaña se establece la frecuencia de simulación. En el núcleo electromagnético MONURBS se necesita una simulación independiente para cada frecuencia, esto será controlado por la interfaz de usuario en el proceso de mallado y simulación, que se explicará posteriormente.

El otro parámetro que se puede establecer es el tipo de simulación, que sirve para calcular algún tipo de postproceso. Según el tipo de simulación elegido se habilitará en el menú Show Results un submenú correspondiente para ver los resultados del postproceso que se ha seleccionado.

4.4.3.2 Configuración del núcleo electromagnético

En la pestaña del menú Solver se configuran las opciones de simulación del núcleo electromagnético MONURBS. En la siguiente figura se muestra la pestaña correspondiente.

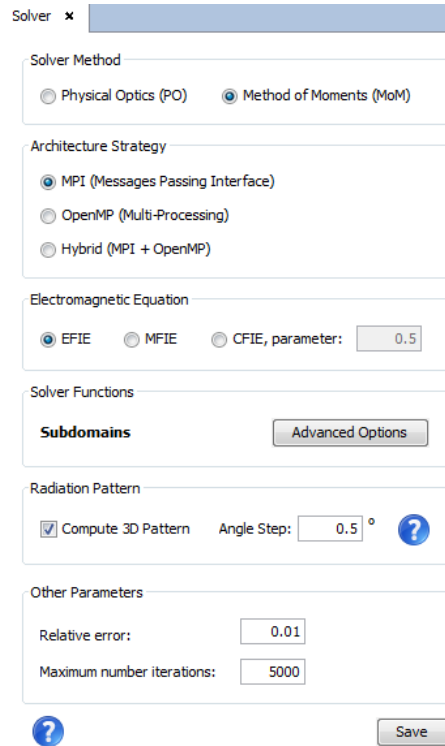


Figura 4.24: Pestaña de configuración del núcleo electromagnético

El método de simulación permite seleccionar entre Óptica Física (PO) y Método de los Momentos (MoM). En las primeras fases de diseño dependiendo del caso se puede utilizar PO ya que es un método mucho más rápido aunque menos preciso.

La arquitectura permite seleccionar el tipo de paralelización que se va a usar en el cálculo, conviene seleccionar la correcta según el tipo de máquina en el que se va a realizar la simulación. Generalmente se escogerá OpenMP (hilos con memoria compartida) si la simulación se realiza en un ordenador personal o workstation y MPI (procesos con memoria distribuida) si se simula en un HPC. Existe un método híbrido que combina las ventajas de ambos mundos, permitiendo ejecutar en cada nodo de un Cluster una tarea MPI que se paraleliza por OpenMP dentro del nodo.

Los siguientes parámetros son la ecuación electromagnética que puede resolver el núcleo MONURBS y los parámetros avanzados del algoritmo de simulación, como por ejemplo si se escoge un preconditionador que puede ayudar a resolver el proceso iterativo de la simulación.

Los últimos parámetros corresponden a si se quiere calcular el diagrama de radiación completo o solo los cortes que se seleccionen en los parámetros de observación y los parámetros de control del proceso iterativo.

4.4.4 Tipos de Alimentación

El núcleo electromagnético MONURBS permite definir varios tipos de alimentación para realizar las simulaciones electromagnéticas. En la interfaz de usuario se han implementado todos los tipos de alimentación disponibles, dentro del menú Source. El usuario debe seleccionar un tipo de alimentación, ya que es posible definir varias antenas del mismo tipo de alimentación, pero no es posible combinar dos tipos distintos. Cuando el usuario selecciona un tipo de alimentación, aparece marcado dentro del menú Source como puede observarse en la figura.

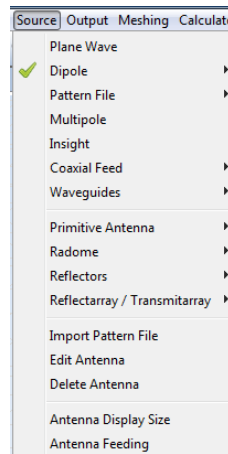


Figura 4.25: Menú Source

Los distintos tipos de alimentación aparecen en la parte de arriba del menú. Si el usuario quiere cambiar de tipo de alimentación y hay otro en uso se le pide una confirmación. Cada tipo de alimentación aparece representado en el panel 3D de una forma diferente para que sea más sencillo de reconocer. Como se verá posteriormente, se pueden asignar varios tipos de alimentación a una geometría para formar un objeto de tipo Antena.

4.4.4.1 Onda Plana

La alimentación se define con una onda plana que incide sobre la geometría. El campo eléctrico incidente viene dado por:

$$\vec{E}_i(\vec{r}) = \vec{E}_0 \cdot e^{-j\vec{k}_{inc}\vec{r}} \quad (4.1)$$

donde \vec{E}_0 es el vector de polarización, \vec{k}_{inc} es la constante de propagación que tiene en cuenta la dirección de incidencia¹ y \vec{r} es el vector de posición.

En la alimentación por onda plana no se dibuja nada en el panel 3D, solo se indica que está seleccionada en el menú Source. Los parámetros se definen en la figura 4.26.

¹ $\vec{k}_{inc} = k_0[\sin(\theta_{inc})\cos(\phi_{inc})\hat{x} + \sin(\theta_{inc})\sin(\phi_{inc})\hat{y} - \cos(\theta_{inc})\hat{z}]$ donde k_0 es la constante de propagación en el vacío, θ_{inc}, ϕ_{inc} las coordenadas esféricas angulares de la dirección de incidencia de la onda plana incidente y $\hat{x}, \hat{y}, \hat{z}$ son los vectores cartesianos unitarios referidos al sistema de referencia absoluto.

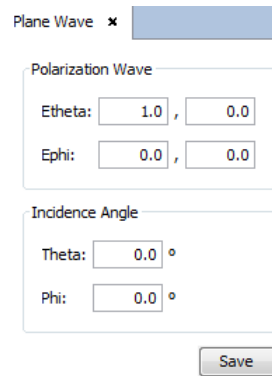


Figura 4.26: Alimentación por onda plana

4.4.4.2 Dipolos

Es el caso de alimentación más sencillo, en MOM se pueden definir varios dipolos, eléctricos y magnéticos, para realizar la simulación. Para definir un dipolo en la interfaz de usuario se define un sistema de coordenadas relativo llamado sistema antena mediante su posición y orientación y los dipolos que contiene. Para cada dipolo se define su amplitud y fase y su posición y orientación dentro del sistema antena. La orientación del dipolo se puede definir de varias maneras y el sistema antena puede ser relativo al plano de referencia.

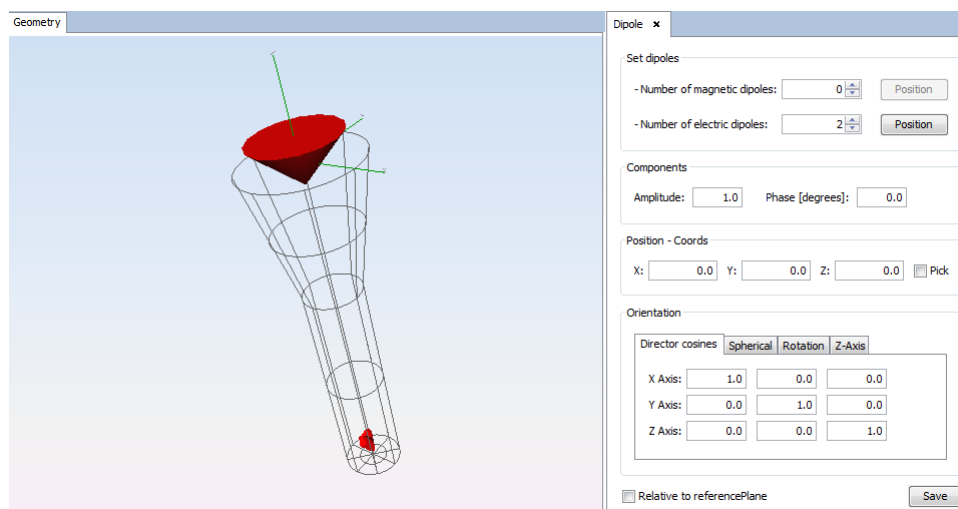


Figura 4.27: Alimentación por dipolo

En la figura anterior puede verse el sistema antena en la boca de la bocina circular y los dos dipolos eléctricos asociados a él en la base para formar una alimentación con polarización circular.

4.4.4.3 Diagramas de Radiación

Es posible establecer la iluminación de la antena mediante un fichero de diagrama de radiación externo. En ese caso, se debe especificar su posición y la orientación.

El fichero que contiene el diagrama de radiación (con extensión .dia) puede tener distintos formatos: simetría de revolución REV, simetría de semi-revolución RV2 o diagrama tridimensional 3DE; y distintas polarizaciones: lineal o circular.

La opción de análisis mediante un diagrama de radiación de antena es muy útil, por ejemplo, cuando se desea analizar una antena reflectora. Es posible analizar la bocina por separado, utilizando el módulo MOM, y exportar su diagrama de radiación para utilizarlo en el análisis del reflector. También es posible añadir varios diagramas de radiación en una simulación. A continuación se muestra un ejemplo del fichero.

```
newFASANT DIA File: bocina.dia
3DE
LIN
Eth , Eph

0.0 361 0.5    0.0 721 0.5

0.0    0.0    -5.52810E01    120.154    -2.22017E-04    50.986
0.5    0.0    -5.52969E01    120.375    -2.33708E-03    50.988
1.0    0.0    -5.53182E01    120.592    -8.68534E-03    50.992
1.5    0.0    -5.53477E01    120.775    -1.82163E-02    50.999
2.0    0.0    -5.53772E01    120.959    -3.09404E-02    51.008
2.5    0.0    -5.54175E01    121.084    -4.79346E-02    51.019
3.0    0.0    -5.54578E01    121.211    -6.81580E-02    51.033
3.5    0.0    -5.55083E01    121.344    -9.27014E-02    51.051
4.0    0.0    -5.55637E01    121.428    -1.21601E-01    51.071
...
```

Tabla 4.1: Formato de fichero .dia: Tipo 3DE

Para añadir un diagrama de radiación a la simulación es necesario importar el fichero. Este fichero se guarda en la carpeta del proyecto dentro del directorio externalFiles, para que el proyecto pueda ser simulado en otra interfaz. Después se añade el diagrama de radiación de forma similar a un dipolo.

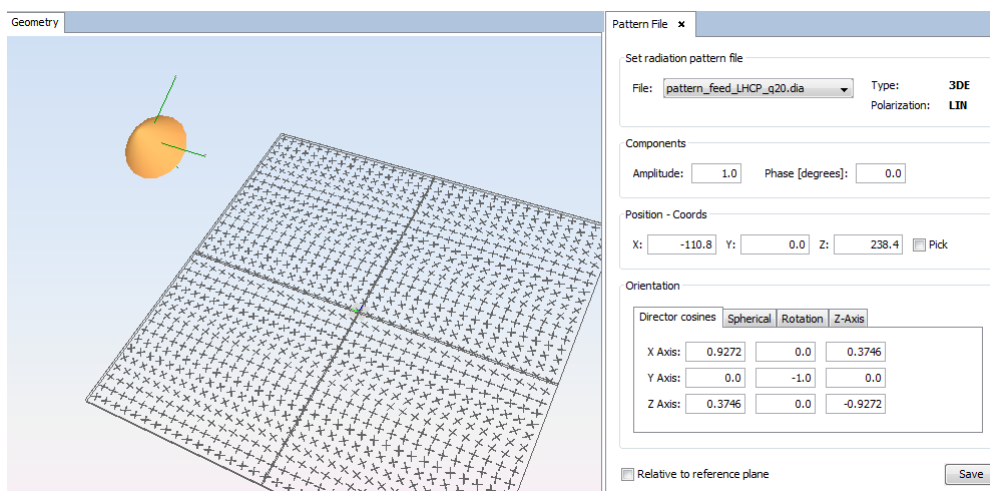


Figura 4.28: Alimentación por diagrama de radiación

4.4.4.4 Campo Impreso

La alimentación por campo impreso consiste en seleccionar puntos de alimentación para aplicarles un voltaje y una impedancia. Esta opción es ampliamente utilizada, por ejemplo para alimentar antenas de hilo o bocinas alimentadas por cable coaxial. Al igual que ocurre con los dipolos o diagramas de radiación, el sistema permite establecer un número arbitrario de puntos de alimentación.

Para alimentar por campo impreso se seleccionan dos superficies que compartan un borde, el núcleo electromagnético MONURBS aplicará el voltaje y la impedancia seleccionados a todos los subdominios de la malla que se encuentran en ese borde. En la siguiente figura se muestra la geometría mientras se añade una alimentación por campo impreso.

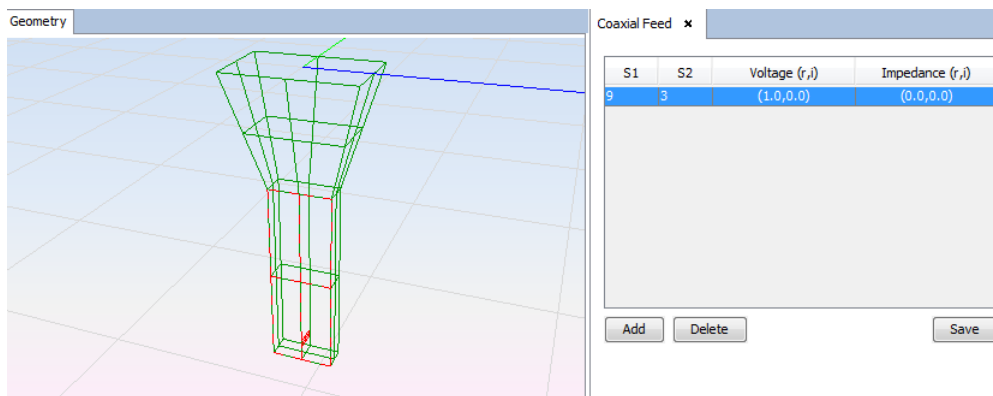


Figura 4.29: Alimentación por campo impreso

4.4.4.5 Guía onda

En la interfaz de usuario también se puede establecer un puerto para simular la colocación de una guía de onda como alimentación de una estructura. Para ello se deben seleccionar las curvas de contorno del borde donde se quiere colocar el puerto, que debe ser plano.

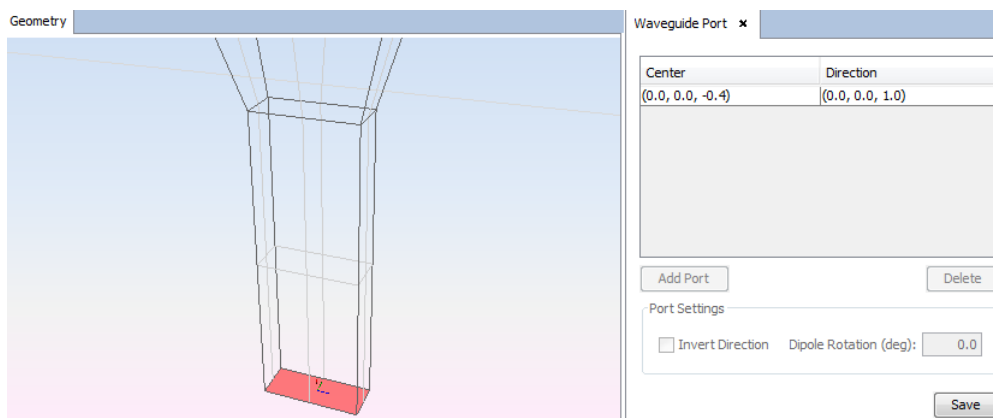


Figura 4.30: Alimentación por guía onda

4.4.4.6 Multipolo

El multipolo es un archivo que crea el núcleo electromagnético MONURBS a partir de una simulación para que pueda ser utilizado como alimentación en otro proyecto. Es un método mucho más preciso que el diagrama de radiación, ya que este solo contiene la información del comportamiento de la antena en campo lejano.

Para generar el fichero se tiene que activar la opción de multipolo en la ventana de los parámetros de simulación del módulo MOM, como se muestra en la figura 4.8. Una vez realizada la simulación, se habilita una opción en el menú Show Results para exportar el fichero, que se guardará con extensión '.suj'. Este fichero puede seleccionarse como alimentación en el menú Source -> Multipole.

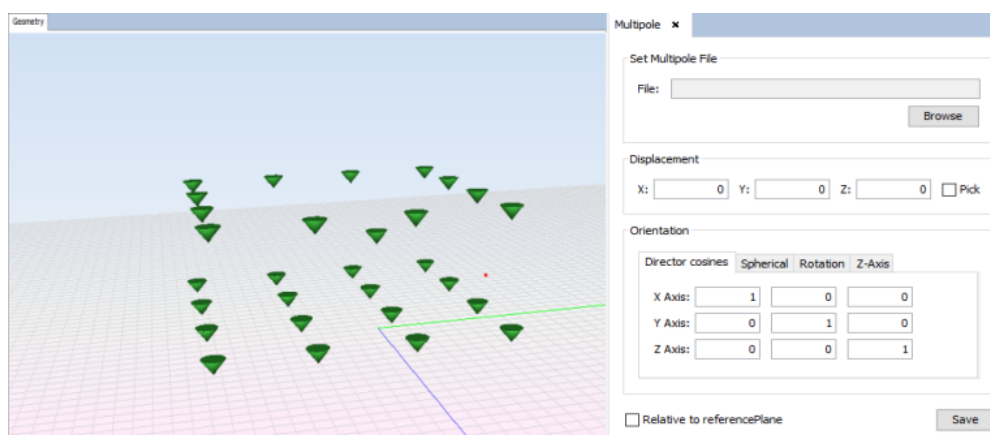


Figura 4.31: Alimentación por multipolo

4.4.5 Parámetros de Observación

Además de definir la geometría y la alimentación, en el módulo MOM hay que definir los parámetros de observación, que le indican al núcleo electromagnético MONURBS en que puntos se van a calcular los resultados.

En el módulo MOM existen dos tipos de observaciones, campo cercano y campo lejano. El campo cercano se encuentra en la región próxima a la antena transmisora, mientras que el campo lejano está más lejos. El núcleo electromagnético MONURBS permite realizar los cálculos en campo cercano y lejano de una antena en la misma simulación.

4.4.5.1 Campo Cercano

El campo cercano es la región del espacio donde la onda electromagnética posee un campo conservativo y es predominantemente de campo magnético (H) o de campo eléctrico (E). En esta región, a diferencia del campo lejano, la forma del diagrama de radiación pueden variar considerablemente con la distancia.

El límite de la región de campo cercano se expresa comúnmente en función de la dimensión lineal de la antena D y de la longitud de onda λ .

$$R < \frac{2D^2}{\lambda} \quad (4.2)$$

En MONURBS, el campo cercano se define mediante un fichero 'puntos.dat' que contiene las coordenadas (x, y, z) de los puntos donde se va a calcular el valor del campo. La interfaz de usuario permite agrupar estos puntos en distintas formas geométricas como pueden ser: líneas, planos, cilindros, esferas o curvas y superficies NURBS.

Los puntos de observación que se han añadido en el módulo MOM se muestran en el panel 3D. En la siguiente figura se pueden ver varios tipos de puntos de observación y su representación dentro del panel de geometría.

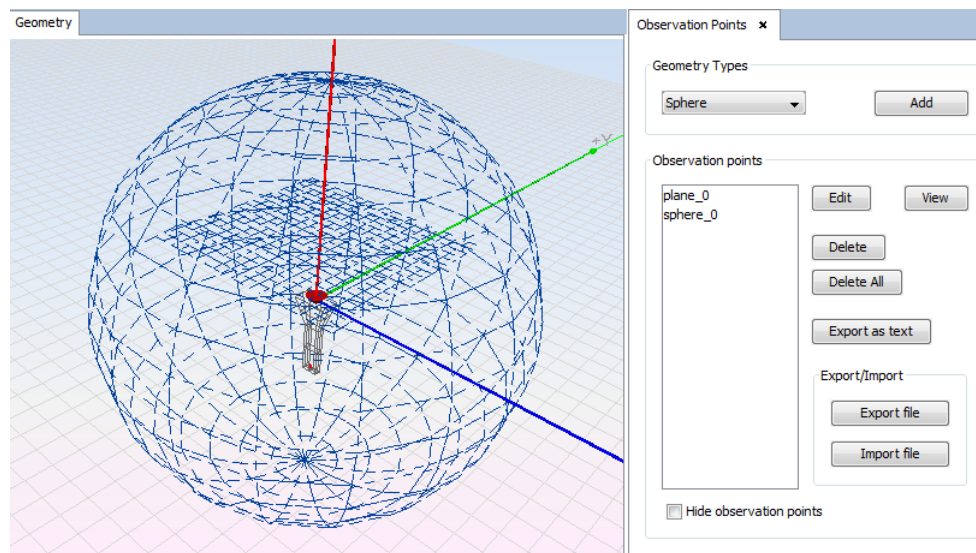


Figura 4.32: Puntos de observación en campo cercano

El núcleo electromagnético MONURBS escribe los resultados en el fichero 'Near-Field.out' con el índice del punto de entrada del fichero 'puntos.dat' y los valores del campo electromagnético en cada punto.

Los resultados de campo cercano pueden verse como fichero de texto, gráficas o dibujando los puntos en un panel 3D auxiliar. En el fichero de texto y las gráficas se pueden ver los resultados de manera similar a las figuras 4.16 y 4.17.

En 3D, al abrir los resultados de campo cercano en el menú Show Results, la interfaz permite seleccionar los puntos de observación que se quieren visualizar; como la interfaz conoce la forma geométrica de los puntos de observación que ha introducido el usuario, se puede construir un IndexedQuadArray y asignar un color a cada punto a partir de la escala que se ha calculado con el valor máximo y mínimo de todos los puntos. En la figura 4.33 se pueden ver los resultados del plano de observación que se introdujo en la figura 4.32 después de realizar la simulación.

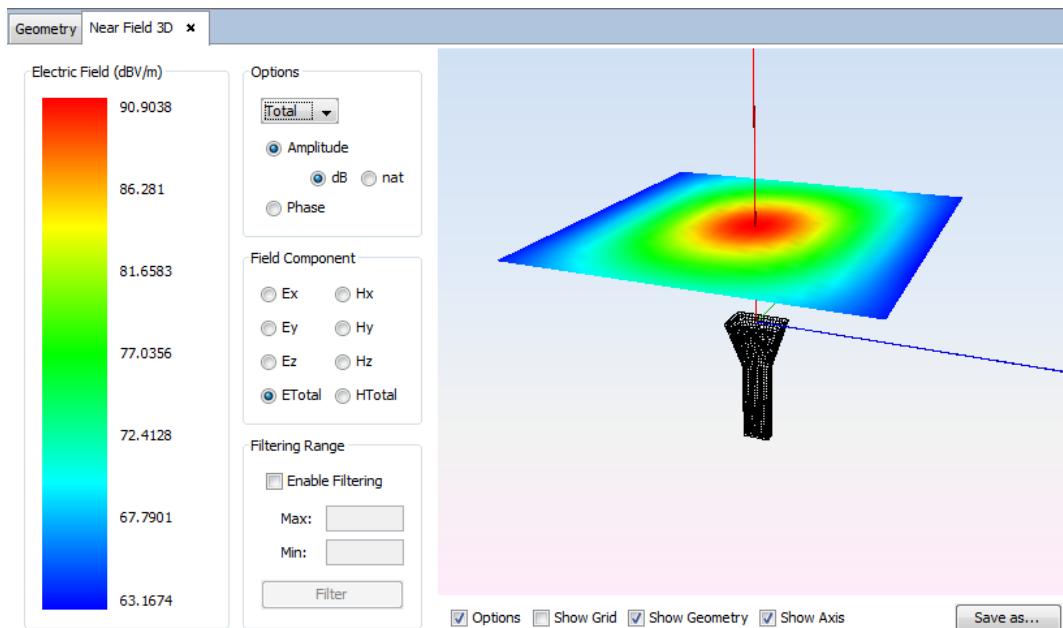


Figura 4.33: Resultados de los puntos de observación en campo cercano

4.4.5.2 Campo Lejano

El campo lejano es la región del espacio donde la onda electromagnética es radiante y está lejos del generador en términos de longitud de onda. En esta región, $E/H = 377 \Omega$. Esto significa que E es 377 veces mayor que el campo H y ambas intensidades de campo, E y H se atenúan proporcionalmente a $1/d$. Ambos campos están a 90° entre ellos. Aquí el conocimiento de uno de los campos, por ejemplo, E, permite la determinación del otro (H), utilizando la relación en la que estas dos cantidades se relacionan entre sí por medio de la impedancia del espacio libre (377Ω).

En la interfaz de usuario el campo lejano se define mediante un barrido de direcciones de observación en coordenadas esféricas Theta (θ) y Phi (φ).

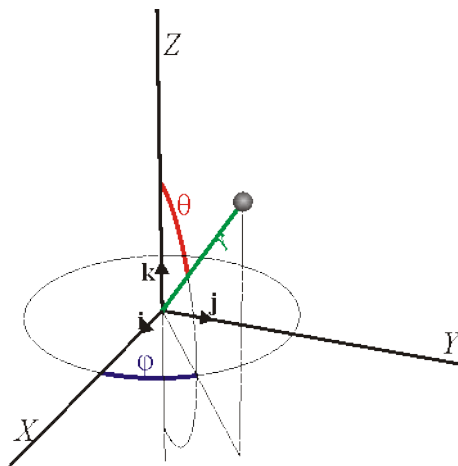


Figura 4.34: Coordenadas esféricas

Las coordenadas esféricas se definen de la siguiente manera:

- La coordenada radial r : distancia al origen.
- La coordenada polar θ : ángulo que el vector de posición forma con el eje Z .
- La coordenada acimutal φ : ángulo que la proyección sobre el plano XY , forma con el eje X .

Los rangos de variación de estas coordenadas son:

$$r \in [0, \infty) \quad \theta \in [0, \pi] \quad \varphi \in [0, 2\pi) \quad (4.3)$$

En la interfaz de usuario se definen barridos en θ o φ , la coordenada r no es necesaria. En la siguiente figura se muestra la ventana de definición del campo lejano.

The screenshot shows a software window titled "Observation Directions" with a close button (x). It contains two main sections for defining observation cuts:

Theta cuts

Theta cut	Initial phi	Increment	Samples	Final phi
90.0	0.0	1.0	361	360.0
45.0	0.0	1.0	361	360.0

Buttons: Delete, Clear, Add

Phi cuts

Phi cut	Initial theta	Increment	Samples	Final theta
0.0	0.0	1.0	181	180.0
45	0.0	1.0	91	90.0
0	-10	0.1	201	10.0

Buttons: Delete, Clear, Add

At the bottom of the window, there are buttons for "Import File", "Saved" (in green), and "Save".

Figura 4.35: Puntos de observación en campo cercano

El núcleo electromagnético calcula todos los cortes en la simulación y los escribe en orden dentro del fichero 'FarField.out' que se muestra en la figura 4.16. Los resultados de los cortes también se pueden ver en forma de gráfica, según se muestra en la figura 4.17. La gráfica permite añadir la serie del corte que se desee seleccionándolo en los desplegados.

4.4.6 Mallado

Después de establecer todos los parámetros de configuración necesarios para poder iniciar la simulación, se ejecuta el algoritmo de mallado [13], de modo que se obtenga un modelo geométrico discretizado sobre el que poder aplicar el Método de los Momentos.

Al pulsar sobre el menú Create Mesh la interfaz muestra una pestaña donde se pueden ajustar los parámetros de mallado.

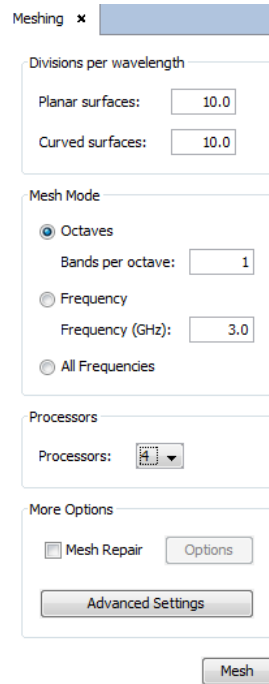


Figura 4.36: Pestaña de parámetros de mallado

En la ventana se puede seleccionar el número de divisiones por longitud de onda, que según la frecuencia seleccionada nos permite definir el tamaño de los parches. Por defecto, para un buen funcionamiento del núcleo electromagnético, 10 divisiones por longitud de onda suele ser un valor adecuado para la mayoría de los casos.

El siguiente parámetro a establecer es el modo de mallado; como el núcleo electromagnético necesita realizar una simulación por cada frecuencia, se puede elegir qué malla utilizar para cada frecuencia seleccionando uno de los tres modos.

Posteriormente se selecciona el número de procesadores con los que se va a ejecutar el proceso 'mallador.exe' y se pueden seleccionar opciones avanzadas que permiten corregir defectos de la malla y cambiar algunos parámetros del algoritmo de mallado.

4.4.6.1 Características de la implementación

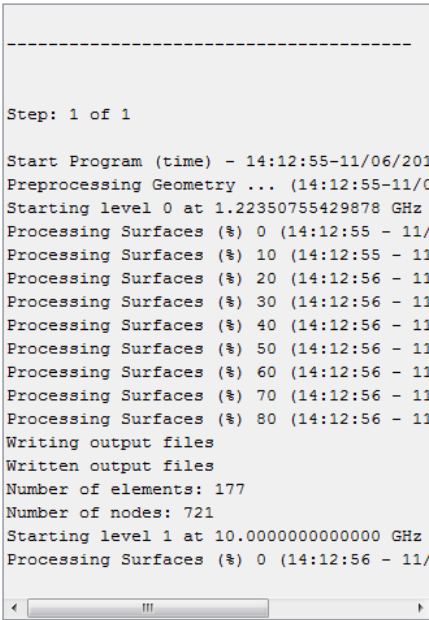
Cuando se pulsa el botón Mesh empieza el proceso de mallado, la interfaz muestra la ventana 'ProcessingCommand' y llama al método *execute* de esa ventana pasándole un objeto de la clase 'ProcessingThread', en este caso es un objeto de la clase 'ExecuteMeshing' del módulo MOM, este método ejecuta el hilo.

Tanto la ventana 'ProcessingCommand' como la barra de progreso de la interfaz son accesibles a través de 'MainFrame' para que el hilo 'ExecuteMeshing' pueda ir escribiendo la salida del proceso.

El hilo se encarga de borrar el directorio Mesh del proyecto de simulación y el directorio de datos de entrada de la carpeta donde se encuentra el mallador (./mesh). La interfaz hace un bucle al número de pasos por si se realiza una simulación paramétrica y al número de frecuencias (dependiendo del modo seleccionado). Dentro de los bucles, la interfaz escribe los ficheros de entrada del programa y ejecuta el proceso 'mallador.exe'.

Entonces, se recoge el PID del proceso y se va leyendo su salida y escribiéndola en la ventana. Cuando el proceso termina se copia la malla a la carpeta correspondiente según el paso y la frecuencia dentro del proyecto de simulación.

Si en algún momento se produce un error, se lanza una excepción de tipo 'ProcessingThreadException' que aborta el proceso.



```
-----  
Step: 1 of 1  
  
Start Program (time) - 14:12:55-11/06/201  
Preprocessing Geometry ... (14:12:55-11/0  
Starting level 0 at 1.22350755429878 GHz  
Processing Surfaces (%) 0 (14:12:55 - 11/  
Processing Surfaces (%) 10 (14:12:55 - 11/  
Processing Surfaces (%) 20 (14:12:56 - 11/  
Processing Surfaces (%) 30 (14:12:56 - 11/  
Processing Surfaces (%) 40 (14:12:56 - 11/  
Processing Surfaces (%) 50 (14:12:56 - 11/  
Processing Surfaces (%) 60 (14:12:56 - 11/  
Processing Surfaces (%) 70 (14:12:56 - 11/  
Processing Surfaces (%) 80 (14:12:56 - 11/  
Writing output files  
Written output files  
Number of elements: 177  
Number of nodes: 721  
Starting level 1 at 10.0000000000000 GHz  
Processing Surfaces (%) 0 (14:12:56 - 11/
```

Figura 4.37: Proceso de mallado

4.4.6.2 Visualización de la malla

En el menú Visualize Existing Mesh se pueden visualizar las mallas que han sido generadas en el proceso de mallado. Para ello se abre el directorio Mesh de la carpeta del proyecto y el usuario selecciona el fichero de malla que quiere visualizar según el número de paso o frecuencia.

La clase 'FileMsh' se encarga de leer el fichero (.msh) generado por el programa externo 'mallador.exe' y devuelve un objeto de tipo 'GeoMsh' con la información de los puntos y los índices de la malla.

La clase 'GeoMsh' se encarga de construir la geometría de la malla en Java 3D usando el objeto IndexedQuadArray. La ventana 'MeshResultsTab' crea un panel 3D auxiliar

donde se dibuja la geometría junto con la información de la malla.

En la siguiente figura se muestra la ventana de visualización del mallado de una geometría en el módulo MOM.

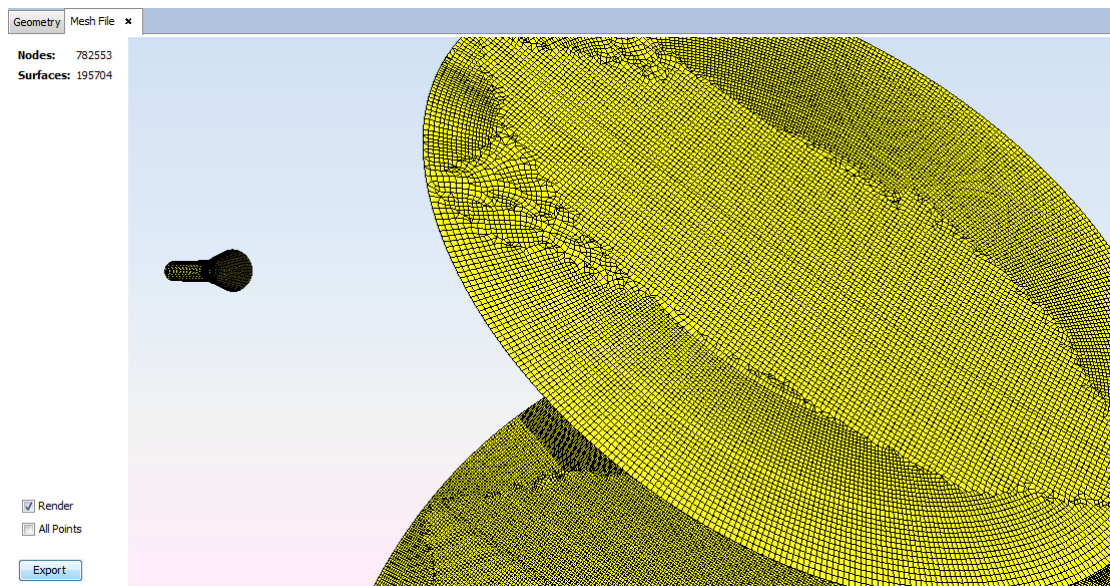


Figura 4.38: Visualización de la malla

4.4.7 Simulación

Una vez que se ha realizado el proceso de mallado en el módulo MOM, se puede ejecutar la simulación del sistema antena. Para ello se pulsa en el botón Execute del menú Calculate donde se abre la siguiente pestaña.

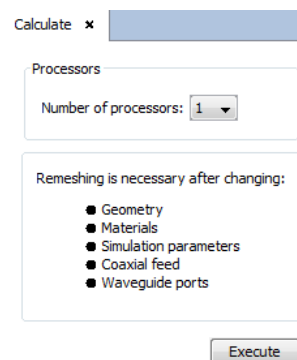


Figura 4.39: Pestaña de parámetros de ejecución

Como se puede observar en la figura, separar el mallado de la ejecución de la simulación permite cambiar parámetros como la posición de un dipolo y volver a ejecutar la simulación sin tener que mallar, ya que la geometría no se ha modificado.

El proceso de simulación es similar al proceso de mallado y se utiliza la misma pestaña 'ProcessingCommand' de la sección 4.4.6.1. En este caso se llama al método *execute* con el objeto 'ExecuteKernel' que extiende de 'ProcessingThread'.

La interfaz hace un bucle al número de pasos por si se realiza una simulación paramétrica y al número de frecuencias. Dentro del bucle copia la malla correspondiente de la carpeta del proyecto al directorio donde se encuentra el núcleo electromagnético MONURBS (`./kernel_monurbs`) y escribe los ficheros de entrada necesarios según los parámetros de simulación. Después se copia el ejecutable correspondiente según el método de paralelización escogido en la pestaña Solver y se ejecuta el proceso 'kernel_monurbs.exe'. Cuando la simulación termina, se copian los resultados a la carpeta del proyecto que corresponde con el paso y la frecuencia que se han simulado.

4.4.8 Resultados

Dependiendo del tipo de simulación seleccionado en las ventanas de parámetros de simulación, en la sección 4.4.3, se habilitarán los resultados correspondientes en el menú Show Results. En las siguientes secciones se muestran los distintos tipos de resultados que se pueden obtener con el módulo MOM.

4.4.8.1 Campo Lejano

Si en los parámetros de observación se ha definido algún corte en campo lejano (figura 4.35), se habilita este menú con las siguientes opciones:

- **View Cuts.** Muestra en una gráfica el valor del campo lejano para todos los puntos de un corte.
- **View Cuts By Frequency.** Muestra en una gráfica el valor del campo lejano en todas las frecuencias de simulación para un punto de un corte cualquiera.
- **View Cuts By Step.** Muestra en una gráfica el valor del campo lejano en todos los pasos de una simulación paramétrica para un punto de un corte.
- **View Text Files.** Después de seleccionar el paso y la frecuencia correspondientes en una pestaña, muestra el fichero de salida 'FarField.out' del núcleo electromagnético que contiene todos los cortes que se han definido.

Los resultados de las gráficas se calculan a partir del fichero que se muestra en el menú View Text Files. Este fichero corresponde al fichero de resultados 'FarField.out' del núcleo electromagnético donde se ha calculado el valor del campo $E_V(V/m)$ y $E_H(V/m)$ para cada dirección de observación. A partir de estos valores y la potencia radiada de la antena, la interfaz calcula los resultados para mostrar el valor del campo en componentes Lineales, Circulares, +/- 45, 3rd Ludwig y Ganancia.

La interfaz lee los ficheros correspondientes a todos los pasos y frecuencias y los almacena en una estructura dentro de la clase 'DataFF', la clase 'DataFFPoint' contiene los

datos de una dirección de observación y es donde se realizan los cálculos para mostrar los distintos resultados.

En las figuras 4.40 y 4.41 se muestra el contenido del fichero de resultados de campo lejano y los mismos resultados transformados a componentes circulares y visualizados en una gráfica.

THETA (deg)	PHI (deg)	E_V (V/m)		E_H (V/m)	
0.000000	0.000000	0.2671E-04	-.6692E-04	-.4951E-04	-.4807E-04
1.000000	0.000000	0.1078E-05	0.1491E-06	-.9107E-06	-.1314E-05
2.000000	0.000000	-.1581E-04	0.5549E-05	-.7599E-06	0.1635E-04
3.000000	0.000000	-.6568E-05	-.4427E-05	-.5912E-05	0.3188E-05
4.000000	0.000000	0.7835E-05	-.5520E-05	-.2262E-05	-.9223E-05
5.000000	0.000000	-.1696E-05	-.3055E-06	-.1823E-05	0.9377E-06
6.000000	0.000000	-.4032E-05	-.3888E-05	-.4957E-05	0.2160E-05
7.000000	0.000000	-.1526E-05	-.4972E-05	-.5117E-05	0.4363E-06
8.000000	0.000000	-.5250E-05	-.2362E-06	-.1350E-05	0.2438E-05
9.000000	0.000000	-.4465E-06	-.1858E-06	-.1362E-05	0.5805E-06
10.000000	0.000000	-.2811E-05	0.2874E-07	-.1471E-05	0.3367E-05
11.000000	0.000000	-.1072E-05	-.9811E-06	-.1116E-05	0.1158E-05
12.000000	0.000000	-.5077E-05	0.4099E-06	-.4987E-06	0.3259E-05
13.000000	0.000000	-.1033E-05	0.3292E-05	0.1621E-05	0.1136E-05
14.000000	0.000000	-.4517E-06	0.1028E-05	0.1115E-06	0.2177E-05
15.000000	0.000000	0.6027E-06	0.2598E-05	0.3574E-05	0.1416E-05
16.000000	0.000000	-.8667E-06	0.1400E-05	0.9920E-06	-.6158E-06
17.000000	0.000000	0.1361E-05	0.3798E-05	0.1150E-05	0.5693E-07
18.000000	0.000000	0.2734E-05	-.1208E-05	0.1208E-05	-.1481E-05
19.000000	0.000000	0.5887E-06	-.8406E-06	0.6898E-06	-.2408E-05
20.000000	0.000000	0.2342E-05	-.2141E-05	-.2070E-05	-.2130E-05
21.000000	0.000000	0.1864E-05	-.1867E-05	-.7715E-06	-.2871E-06
22.000000	0.000000	-.4035E-06	-.2500E-05	-.8707E-06	-.2497E-06
23.000000	0.000000	-.2309E-05	0.1671E-05	-.1182E-05	0.2423E-05
24.000000	0.000000	-.2055E-05	0.1837E-05	0.1094E-05	0.4884E-05
25.000000	0.000000	-.4620E-05	0.4562E-05	0.3323E-05	0.3803E-05

Figura 4.40: Fichero de campo lejano

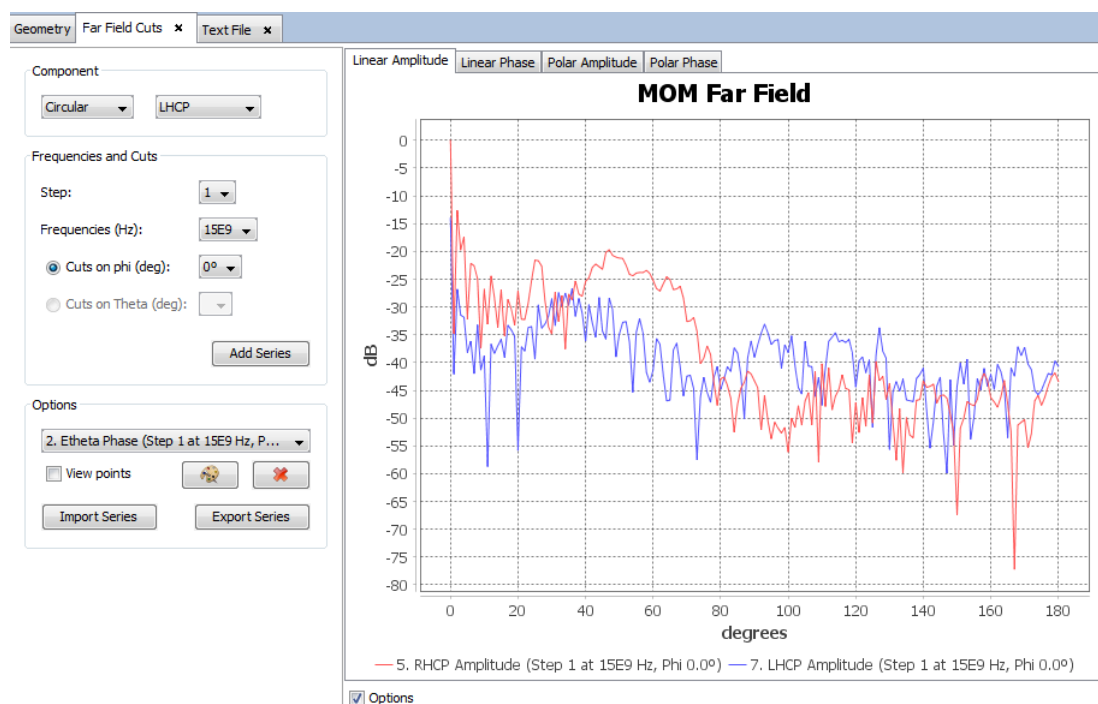


Figura 4.41: Gráfica de campo lejano

Los ficheros de texto que se muestran en la pestaña anterior se pueden exportar mediante el botón Save As por si se quiere realizar un postproceso en MATLAB o mediante cualquier programa externo, la interfaz también tiene su propio módulo de postproceso.

En las gráficas, después de seleccionar los datos que se quieren visualizar en los desplegables correspondientes, se pulsa el botón Add Series que automáticamente insertará la amplitud y la fase en las gráficas correspondientes. En el panel de opciones se pueden borrar y cambiar de apariencia, además se pueden editar pulsando con el botón derecho encima de la gráfica. Es posible exportar una serie como fichero de texto e importarla en cualquier otra gráfica del programa para realizar comparaciones.

4.4.8.2 Diagrama de Radiación

Si se ha seleccionado la opción de calcular el diagrama de radiación 3D en los parámetros del núcleo electromagnético de la sección 4.4.3 se habilitará este menú.

En el menú se encuentran las mismas opciones que la sección anterior, pero para todas las direcciones de observación del diagrama de radiación, según el paso seleccionado. Los datos se leen del fichero '3DFarField.out' y el formato es el mismo que en la figura 4.40.

En este menú también aparecen las siguientes opciones:

- **View 3D Pattern.** Muestra un panel 3D con el diagrama de radiación. Se pueden mostrar los mismos resultados que en las gráficas (lineales, circulares, ganancia...). El diagrama se calcula a partir de una escala de colores y asignando un radio a cada punto de la esfera dependiendo del valor del campo en esa dirección.
- **Export as DIA File.** Permite exportar el diagrama de radiación como un fichero en formato DIA que puede utilizarse como alimentación en otro proyecto, como se muestra en la sección 4.4.4.3.

Si se han realizado varios pasos o frecuencias, antes de exportar el fichero DIA hay que seleccionar el paso y la frecuencia deseadas, después aparecerá una pestaña similar a la figura 4.40 con el fichero de diagrama de radiación creado.

En la figura 4.42 se muestra el diagrama de radiación de un sistema de antena Cassegrain simulado con la interfaz de usuario, la ventana permite seleccionar el paso o la frecuencia que se quieren mostrar y el tipo de resultado. También se puede filtrar la escala entre los valores que desee el usuario lo que afecta únicamente a los colores del diagrama.

En la parte de abajo del panel 3D se muestran opciones de visualización del diagrama y se puede ocultar el panel de la izquierda. El botón Save As permite guardar una imagen en formato PNG formada por el panel 3D y la escala de colores.

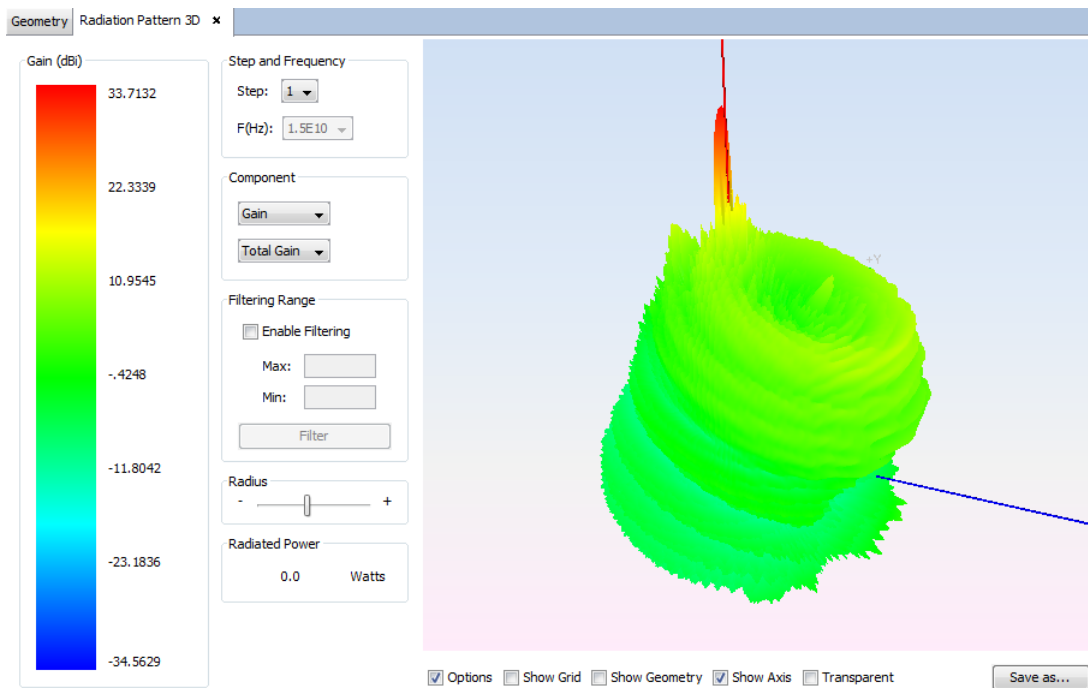


Figura 4.42: Diagrama de radiación

4.4.8.3 Campo Cercano

Si en los parámetros de observación se definen puntos en campo cercano, como puede verse en la figura 4.32, se habilita el menú de resultados de campo cercano con las siguientes opciones:

- **View Near Field Diagram.** Una vez seleccionados los puntos de observación que se quieren visualizar muestra un panel 3D con la escala y los puntos de observación coloreados según su valor del campo.
- **View Observation Points.** Seleccionando el paso y la frecuencia muestra una gráfica con todos los puntos de observación.
- **View Observation Points By Frequency.** Se selecciona un punto de observación y muestra una gráfica con su valor para todas las frecuencias de simulación.
- **View Observation Points By Step.** Se selecciona un punto de observación y una frecuencia y muestra una gráfica con su valor para todos los pasos de una simulación paramétrica.
- **View Text Files.** Muestra el fichero de texto 'NearField.out' que genera el núcleo electromagnético MONURBS con el valor del campo para cada punto.

En la figura 4.43 se muestra el fichero de salida del núcleo electromagnético y en la figura posterior el fichero representado en la interfaz en el panel 3D de resultados de campo cercano.

PTO	Ex (V/m)	Ey (V/m)	Ez (V/m)	Hx (A/m)
1	-0.4394E+03	0.1604E+03	0.6007E+03	-0.4742E+03
2	-0.1884E+03	-0.4656E+03	0.8709E+03	0.9060E+03
3	0.4887E+03	-0.2627E+03	-0.1299E+04	0.1320E+04
4	0.3761E+03	0.4632E+03	-0.1905E+04	-0.1522E+04
5	-0.3337E+03	0.5012E+03	0.1235E+04	-0.2660E+04
6	-0.5539E+03	-0.7658E+02	0.3277E+04	0.7314E+02
7	-0.2224E+03	-0.4146E+03	0.1921E+04	0.2874E+04
8	0.8382E+02	-0.3390E+03	-0.5221E+03	0.3459E+04
9	0.1443E+03	-0.1544E+03	-0.2101E+04	0.2756E+04
10	0.5914E+02	-0.3406E+02	-0.2692E+04	0.2118E+04
11	-0.6046E+02	0.4549E+02	-0.2688E+04	0.2119E+04
12	-0.1412E+03	0.1660E+03	-0.2088E+04	0.2757E+04
13	-0.7327E+02	0.3465E+03	-0.5051E+03	0.3450E+04
14	0.2359E+03	0.4103E+03	0.1926E+04	0.2852E+04
15	0.5563E+03	0.6166E+02	0.3261E+04	0.5663E+02
16	0.3197E+03	-0.5080E+03	0.1216E+04	-0.2650E+04
17	-0.3848E+03	-0.4504E+03	-0.1899E+04	-0.1505E+04
18	-0.4762E+03	0.2713E+03	-0.1285E+04	0.1315E+04
19	0.1950E+03	0.4522E+03	0.8651E+03	0.8957E+03
20	0.4246E+03	-0.1628E+03	0.5940E+03	-0.4680E+03
21	-0.1187E+03	-0.4986E+03	0.8172E+03	0.7258E+03
22	0.5376E+03	-0.9759E+02	-0.1359E+04	0.1102E+04
23	0.1210E+03	0.5812E+03	-0.1291E+04	-0.2066E+04
24	-0.5941E+03	0.1934E+03	0.2594E+04	-0.1596E+04
25	-0.3145E+03	-0.5269E+03	0.2241E+04	0.2664E+04

Figura 4.43: Fichero de campo cercano

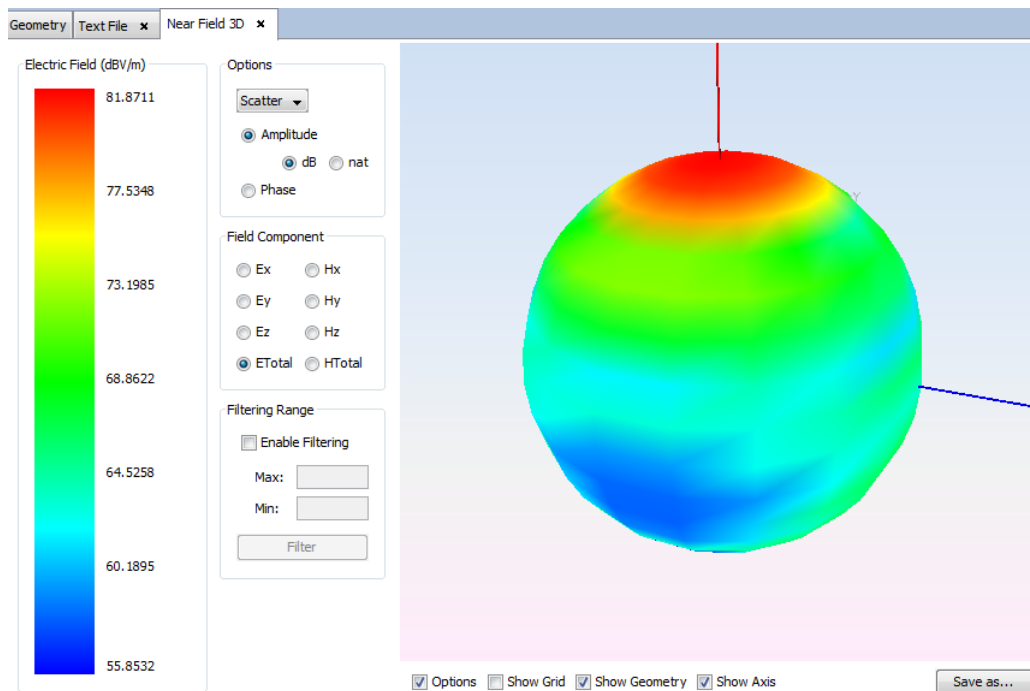


Figura 4.44: Resultados de campo cercano

4.4.8.4 Visualización de Corrientes y Cargas sobre la Geometría

Debido a que para el cálculo del Método de los Momentos que utiliza el núcleo electromagnético, es necesario calcular la distribución de las corrientes y las cargas en la geometría, es posible representarla en un panel 3D de resultados.

Los ficheros de cargas y corrientes tienen un formato similar. El núcleo escribe el valor de la carga o la corriente para cada punto de la malla.

La interfaz lee el fichero de malla utilizando las clases de la sección 4.4.6.2 y el fichero de corrientes o cargas generado por el núcleo electromagnético. Al igual que en el resto de resultados genera una escala con los valores mínimo y máximo y asigna un color a cada punto del IndexedQuadArray de la malla, que se dibuja en el panel 3D de la ventana de resultados.

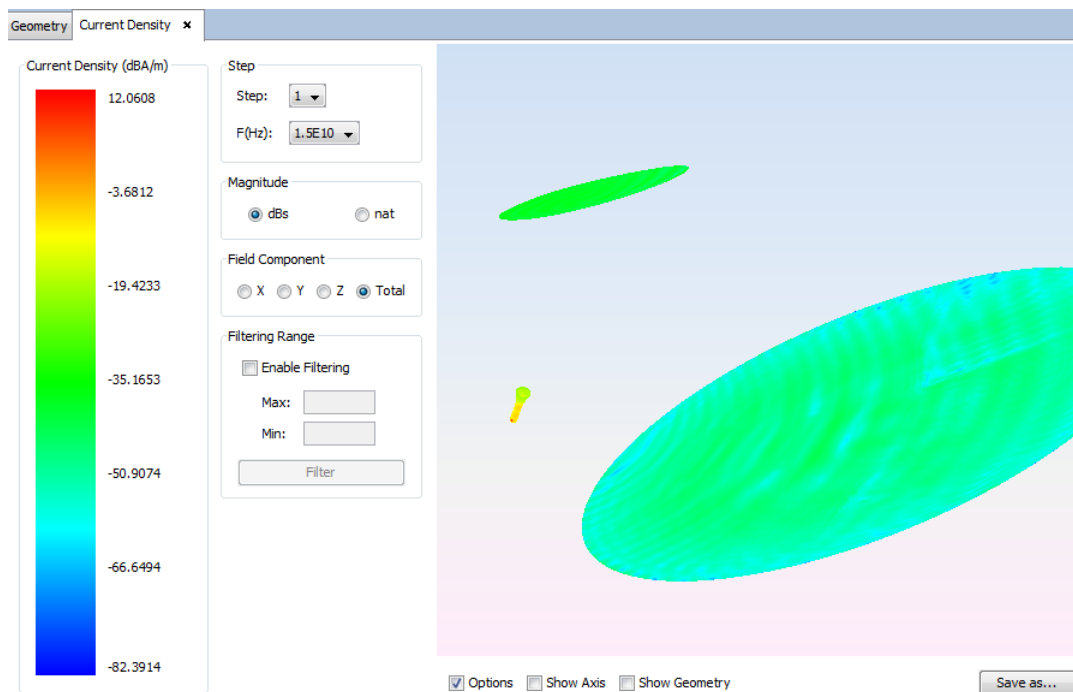


Figura 4.45: Visualización de corrientes sobre la geometría

Bibliografía

- [1] I. González, E. Garcia, F. Sáez de Adana, y F. Cátedra, “Monurbs: a parallelized fast multipole multilevel code for analyzing complex bodies modeled by NURBS surfaces,” *Applied Computational Electromagnetics Society Journal*, 2008.
- [2] F. M. Sáez de Adana Herrero, “Aplicación de GTD al Análisis de Radiación y Propagación Electromagnética en Entornos Complejos,” Ph.D. dissertation, Universidad de Cantabria, 2000.
- [3] R. Harrington, *Field Computation by Moment Methods*. Macmillan, 1968.
- [4] J. P. Arriaga, “Aplicación de la Optica Física al Cálculo de la RCS de Cuerpos de Geometría Compleja Modelados Mediante Parches NURBS,” Ph.D. dissertation, Universidad de Cantabria, 1999.

- [5] C. Delgado, M. F. Cátedra, y R. Mittra, "Application of the Characteristics Basis Functions Method Utilizing a Class of Basis and Testing Functions Defined on NURBS Patches," *IEEE Transactions on Antennas and Propagation*, 2008.
- [6] I. Stevanovic, P. Crespo Valero, K. Blagovic, F. Bongard, y J. R. Mosig, "Integral-Equation Analysis of 3-D Metallic Objects Arranged in 2-D Lattices Using the Ewald Transformation," *IEEE Transactions on Microwave Theory and Techniques*, 2006.
- [7] A. Somolinos, J. L. García, L. Lozano, F. Cátedra, y I. Gonzalez, "Computer Tool for Simulating Frequency Modulated Continuous Wave Radar Systems in Urban Traffic Scenes," in *EUCAP*, 2018.
- [8] I. González, F. Sáez de Adana, y F. Cátedra, "Application of the Multilevel Fast Multipole Method to the Analysis of Conformed Multilayered Periodic Structures," *IEEE AP-S International Symposium on Antennas and Propagation*, 2007.
- [9] E. Garcia, C. Delgado, I. González, y F. Cátedra, "An Iterative Solution for Electrically Large Problems Combining the Characteristic Basis Function Method and the Multilevel Fast Multipole Algorithm," *IEEE Transactions on Antennas and Propagation*, 2008.
- [10] E. García García, "Contribución al análisis de problemas electromagnéticos mediante el Método de los Momentos con bajo coste computacional," Ph.D. dissertation, Universidad de Alcalá, 2005.
- [11] "MPI: Message Parsing Interface." [Online]. Disponible en: <https://www.mpi-forum.org/docs/>
- [12] "OpenMP API specification for parallel programming." [Online]. Disponible en: <https://www.openmp.org/>
- [13] J. Moreno Garrido, "Desarrollo y Optimización de Un Generador de Mallas Superficiales y/o Volumétricas para Aplicaciones de Simulación Electromagnética," Ph.D. dissertation, Universidad de Alcalá, 2013.

Capítulo 5

Resultados: Diseño de un Reflectarray Parabólico

5.1 Introducción

En este capítulo se muestra el diseño completo de una antena reflectarray [1], utilizando la interfaz de usuario propuesta en esta tesis. El reflectarray presenta un diseño bastante novedoso, ha sido conformado en una superficie parabólica y utiliza un elemento de tipo cruz para generar una discriminación en polarización circular. El capítulo también incluye el diseño y fabricación de un demostrador más pequeño para comprobar el funcionamiento del reflectarray.

En la primera parte se describen las características de la antena y los conceptos teóricos que se van a aplicar en el diseño del reflectarray y del demostrador. Después, se diseñará la celda utilizando el módulo de estructuras periódicas de la interfaz de usuario. Para ello, se realizarán varias simulaciones paramétricas para generar una base de datos que posteriormente se usará para generar el reflectarray.

En el módulo MOM que se ha descrito en el capítulo anterior, se generará el reflectarray con una herramienta de la interfaz de usuario y se realizarán las simulaciones con el núcleo electromagnético MONURBS. Por último se describe el diseño y la fabricación de un demostrador y se realiza la comparativa de las simulaciones con las medidas obtenidas.

El diseño y la fabricación de esta antena reflectarray se ha llevado a cabo gracias a D. José Antonio Encinar Garcinuño, profesor en la Escuela Superior de Ingenieros de Telecomunicación de la Universidad Politécnica de Madrid, como parte de mi estancia incluida en el programa de doctorado de la Universidad de Alcalá.

5.2 Descripción de la Antena

En los últimos años las técnicas para diseñar y analizar antenas reflectarrays han tenido un gran impulso, como puede verse en [2–4]. Los reflectarrays ofrecen una gran flexibilidad de diseño, debido a su construcción mediante capas microstrip y a que pueden colocarse sobre superficies planas o arbitrarias. Recientemente, se ha demostrado que es posible diseñar y construir reflectarrays con un grado de precisión similar al de las antenas reflectoras. Se pueden encontrar diseños de antenas reflectarray que proporcionen buenas características de ganancia, eficiencia, pureza de polarización y conformación del haz con especificaciones muy estrictas para ser utilizadas en satélites de comunicación [5–7].

En el trabajo presentado en [8] se muestra una antena reflectarray de doble banda trabajando a 19.7GHz y 29.5GHz que utiliza dos capas de dipolos paralelos para obtener un haz en polarización lineal para ambas frecuencias. La antena diseñada en este capítulo utiliza como elemento una cruz y la técnica de rotación variable (VRT) [9] que permite separar el haz principal en dos haces simétricos según se alimente con polarización circular a izquierdas LHCP o a derechas RHCP. Estas antenas de doble banda y doble polarización tienen mucho interés hoy en día para las antenas de satélite multi-beam, ya que permiten reducir considerablemente el número de aperturas en el satélite.

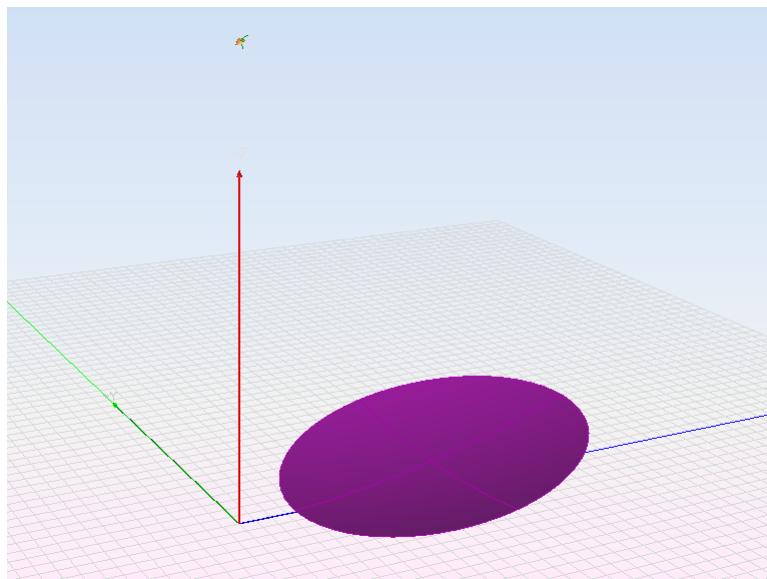


Figura 5.1: Superficie parabólica de la antena reflectarray

En la figura 5.1 se puede ver la superficie donde va a ir conformado el reflectarray y su alimentación. La superficie es un reflector parabólico con offset que tiene su vértice en el origen de coordenadas.

El foco de la parábola, que es el punto de la figura anterior donde se ha colocado la alimentación, se encuentra a 2.718m de altura sobre el eje Z. El diámetro del disco de la parábola son 1.812m y tiene un offset de 0.35m. La relación entre la distancia focal y el diámetro de la parábola es de 1.5m, con un ángulo subtendido de $\pm 18.09^\circ$.

Alimentación

El reflectarray propuesto en este capítulo se ha diseñado a una frecuencia de 19.7GHz. Es posible obtener una iluminación en el borde de -12dB utilizando un alimentador con un radio exterior de 54mm.

En la interfaz de usuario la alimentación se ha definido mediante un diagrama de radiación, visto en la sección 4.4.4.3. Para obtener esas características, se ha generado el patrón de radiación utilizando la siguiente función con un valor de $q = 24$:

$$f(\theta) = \cos^q(\theta)$$

La interfaz de usuario tiene una utilidad para generar diagramas de radiación mediante una función. El primer paso ha sido generar el diagrama del campo EH en lineales y posteriormente, utilizando el núcleo electromagnético MONURBS, componer el campo en circulares para obtener dos diagramas de radiación, uno con polarización circular a izquierdas LHCP y otro a derechas RHCP, que se utilizarán para alimentar el reflectarray. En la siguiente figura se muestra el diagrama de radiación LHCP simulado utilizando el módulo MOM del capítulo anterior.

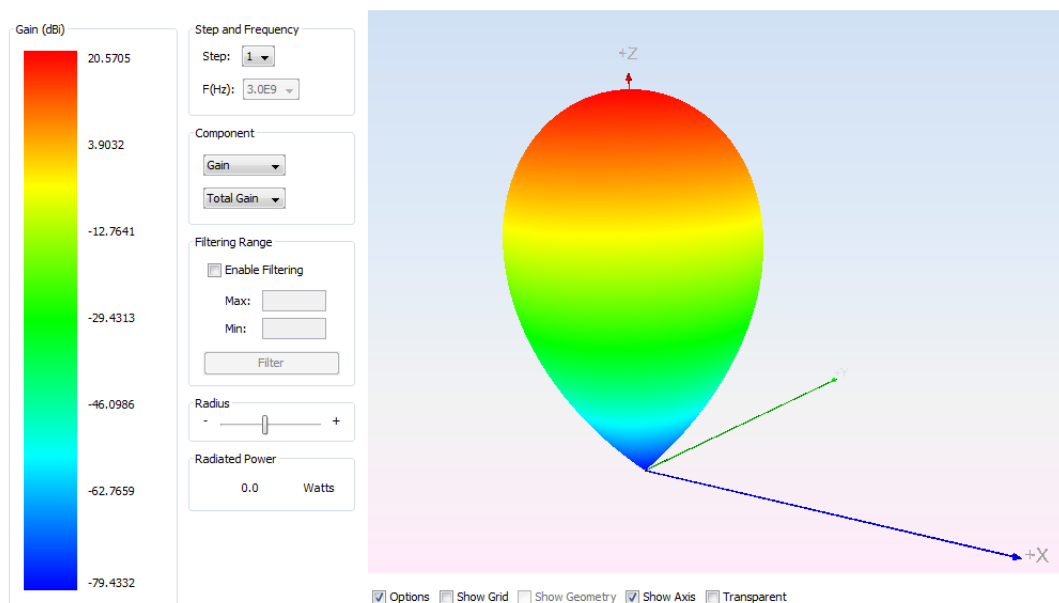


Figura 5.2: Diagrama de radiación de la bocina con polarización LHCP

Posteriormente se utilizará en las medidas una bocina similar al diagrama de radiación utilizado en las simulaciones.

La bocina ha sido medida a 19.7GHz y se han comparado sus resultados con el diagrama de radiación generado en la interfaz. En la figura 5.3 se muestra la comparación del diagrama generado en la interfaz de usuario con las medidas de la bocina real en la cámara anecoica. La gráfica corresponde al corte $\varphi = 0^\circ$ con una variación en $\theta = [-90^\circ, 90^\circ]$.

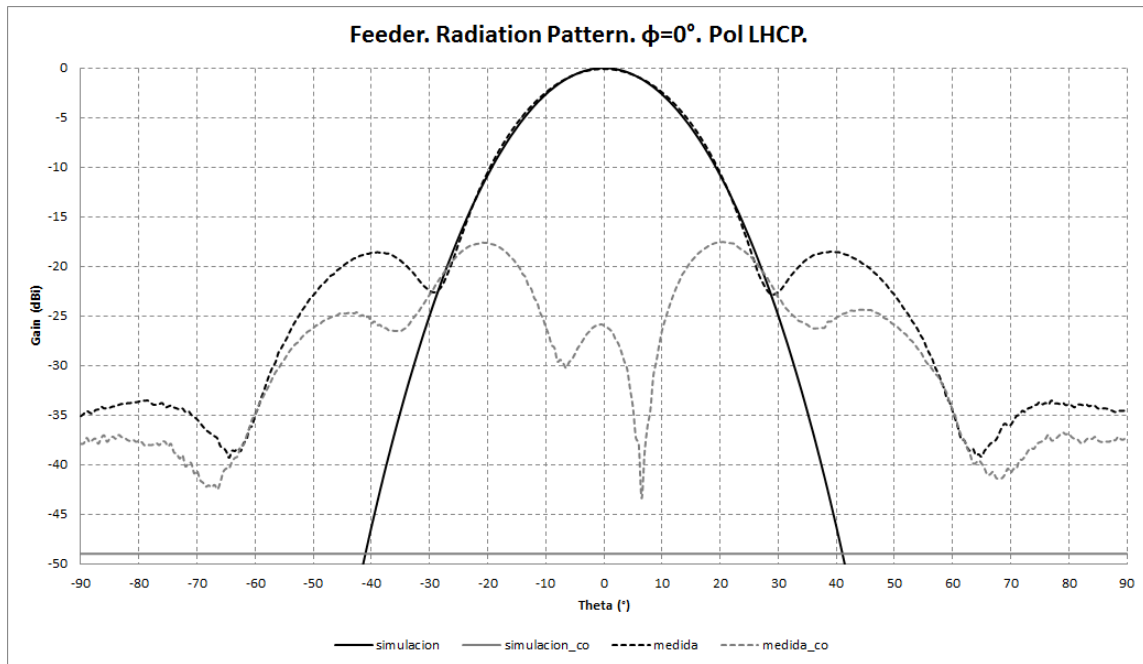


Figura 5.3: Comparativa entre la bocina real y el diagrama de radiación

Resultados del Reflector

Se ha simulado el reflector de la figura 5.1 utilizando el módulo MOM del capítulo anterior. En la figura también se puede observar la posición de la alimentación que se encuentra en el foco de la parábola apuntando hacia el centro geométrico del reflector.

Para ello, en la interfaz de usuario se ha creado un nuevo proyecto, configurando en los parámetros de simulación la frecuencia a 19.7GHz y se ha alimentado con el diagrama de radiación del apartado anterior.

En la siguiente figura se muestra el mallado de la geometría realizado a la frecuencia de simulación con 10 divisiones por longitud de onda.

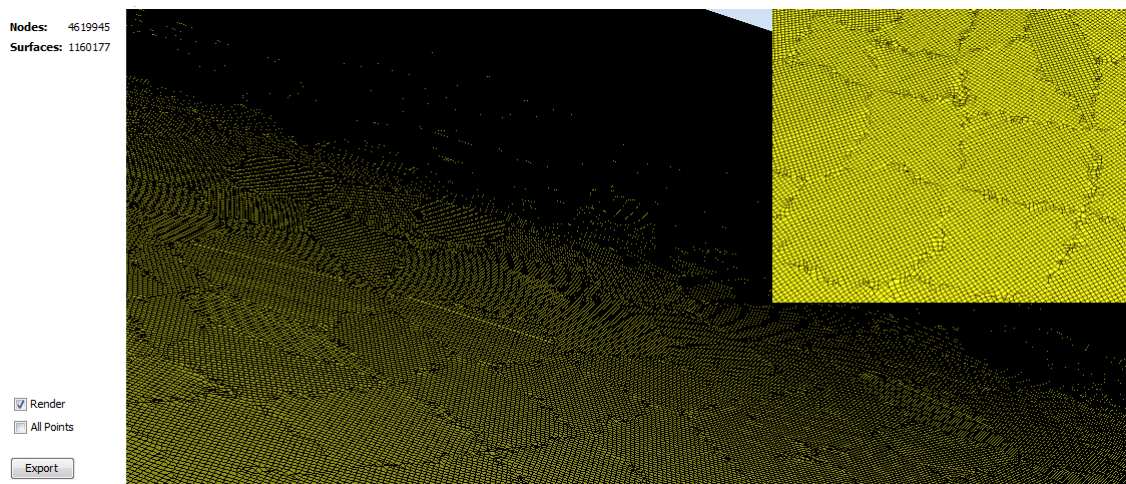


Figura 5.4: Mallado del reflector parabólico

Después del mallado, se ha realizado la simulación utilizando la interfaz con el núcleo electromagnético MONURBS. En la siguiente figura se muestra el diagrama de radiación del reflector alimentado por la bocina con polarización LHCP. El diagrama representa la ganancia total.

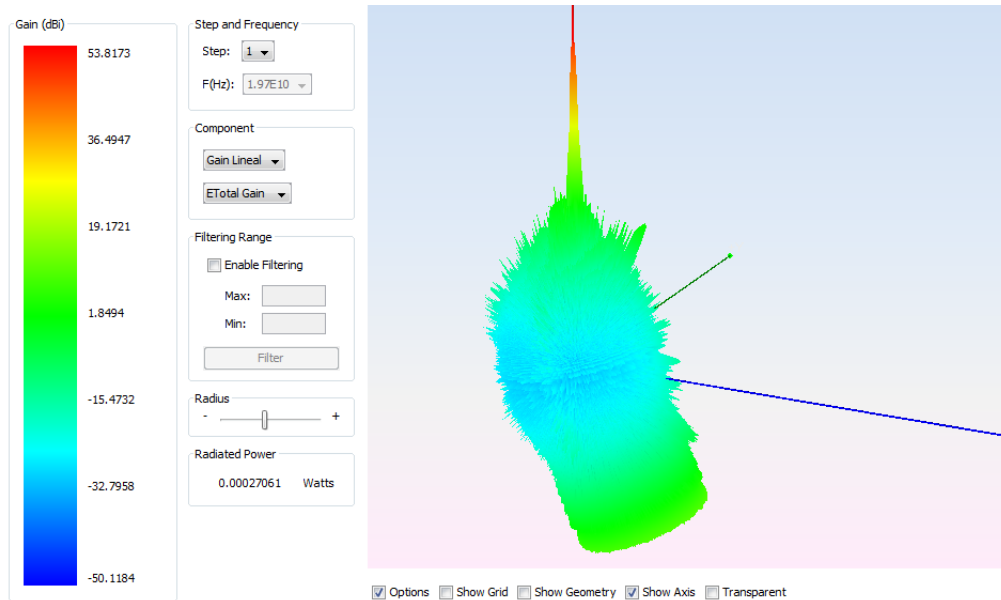


Figura 5.5: Diagrama de radiación del reflector parabólico

En la siguiente gráfica se muestra la ganancia en circulares del reflector parabólico para el corte $\varphi = 0^\circ$ y $\theta = [-20^\circ, 20^\circ]$.

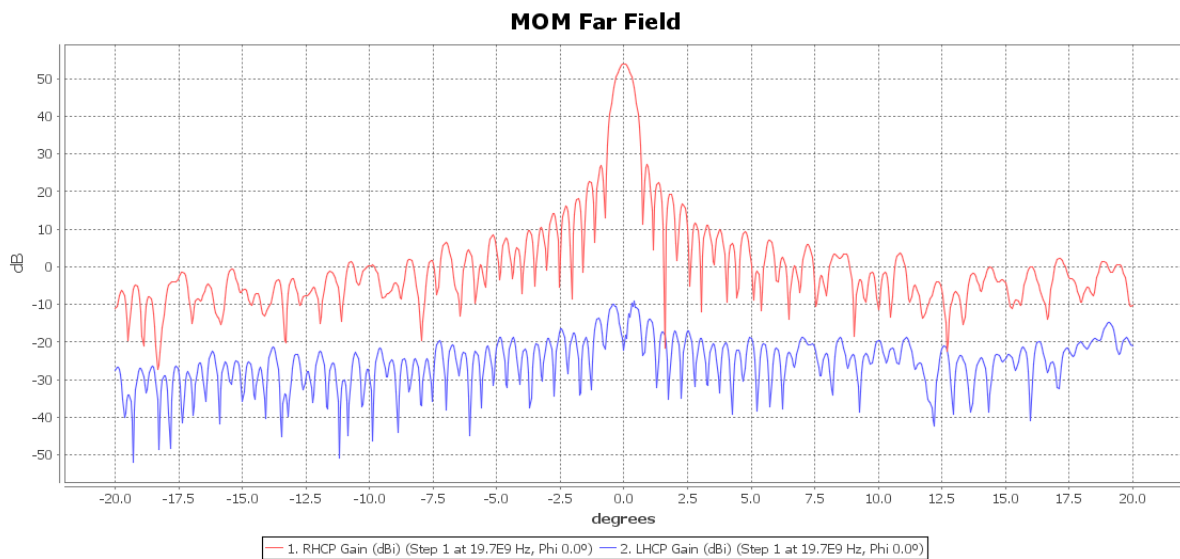


Figura 5.6: Resultados del reflector parabólico alimentado con LHCP

Por último, en la siguiente figura se puede ver la distribución de corrientes sobre la superficie del reflector parabólico para comprobar que la bocina apunta correctamente al centro de la parábola.

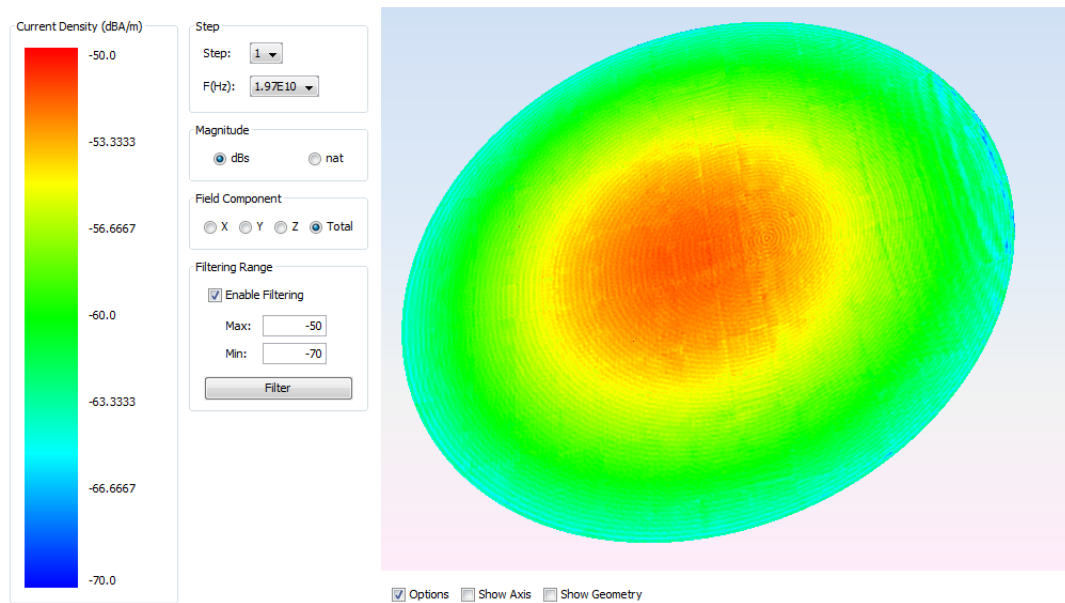


Figura 5.7: Distribución de corrientes del reflector parabólico

5.3 Diseño de la Celda del Reflectarray

Para diseñar un reflectarray utilizando el módulo MOM de la interfaz de usuario, primero hay que generar una base de datos que contiene las celdas que van a ser colocadas en el reflectarray.

El primer paso es utilizar el módulo de estructuras periódicas para diseñar la celda inicial del reflectarray. Una vez definida la celda inicial, se realizan pequeñas variaciones geométricas mediante simulaciones paramétricas para obtener la curva de fase deseada y se genera la base de datos.

Posteriormente, la base de datos se importa en el módulo MOM y según la función definida por el usuario, se coloca la celda que corresponda en cada posición sobre la superficie del reflectarray.

Para el diseño de la celda del reflectarray a una frecuencia de 19.7GHz se escogió un tamaño aproximado de $\lambda/2 \approx 7,5\text{mm}$ y la siguiente estructura:

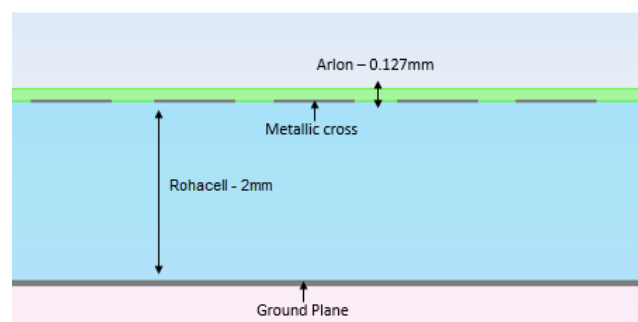


Figura 5.8: Estructura de la celda del reflectarray

En la siguiente tabla se muestran las propiedades de los materiales que se han utilizado en el diseño y simulación de la estructura periódica y en la fabricación del demostrador.

<i>Material</i>	<i>Er</i>	<i>tan(p)</i>	<i>Espesor</i>
Arlon	2.3	0.002	0.127mm
Rohacell	1.12	0.002	2mm

Tabla 5.1: Materiales del reflectarray

Para utilizar la técnica de rotación variable (VRT) que permite separar el haz principal en dos haces simétricos según se incida con LHCP o RHCP es necesario encontrar un elemento que tenga una diferencia de fase de 180° entre ambas polarizaciones lineales (VV y HH).

El elemento escogido es una cruz metálica, que se ha optimizado mediante simulaciones paramétricas, para conseguir la longitud de los brazos necesaria para tener una diferencia de fase de 180° entre las polarizaciones lineales.

El primer paso ha sido crear la celda en el módulo de estructuras periódicas de la interfaz de usuario y simular un dipolo con distintas longitudes mediante una simulación paramétrica (Sección 3.8). En la siguiente figura se muestra el dipolo y la definición de los parámetros en la interfaz.

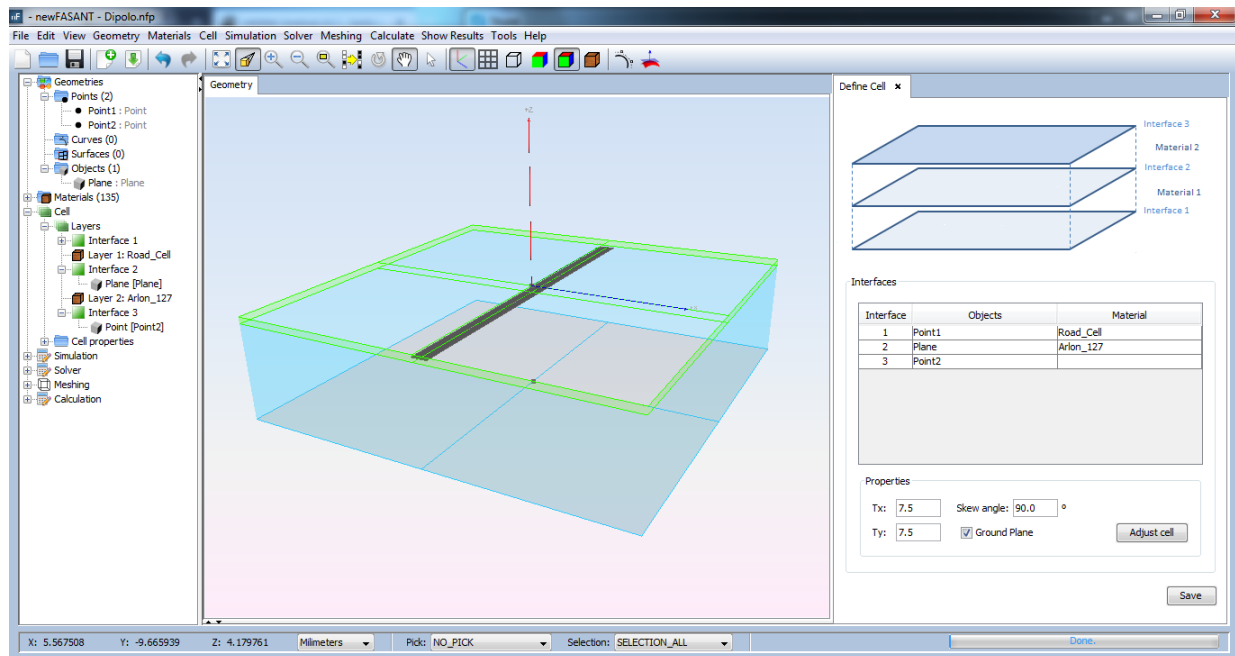


Figura 5.9: Simulación de un dipolo paramétrico

La anchura del dipolo es de 0.3mm y se han simulado 17 pasos variando la longitud del dipolo de 4mm a 7.2mm.

En la siguiente figura se muestra la ventana de definición de parámetros que se han

utilizado en el dipolo. La geometría se ha definido con un plano utilizando el parámetro \$1 para definir el punto inicial y \$2 para definir las dimensiones.

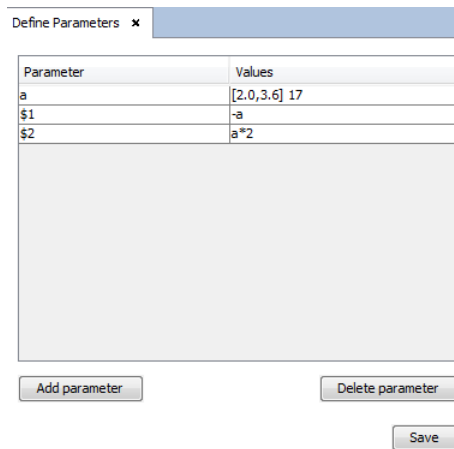


Figura 5.10: Definición de parámetros del dipolo

Una vez realizadas las simulaciones, se obtiene la siguiente curva de fase en polarización lineal según el paso que se ha simulado en la variación paramétrica.

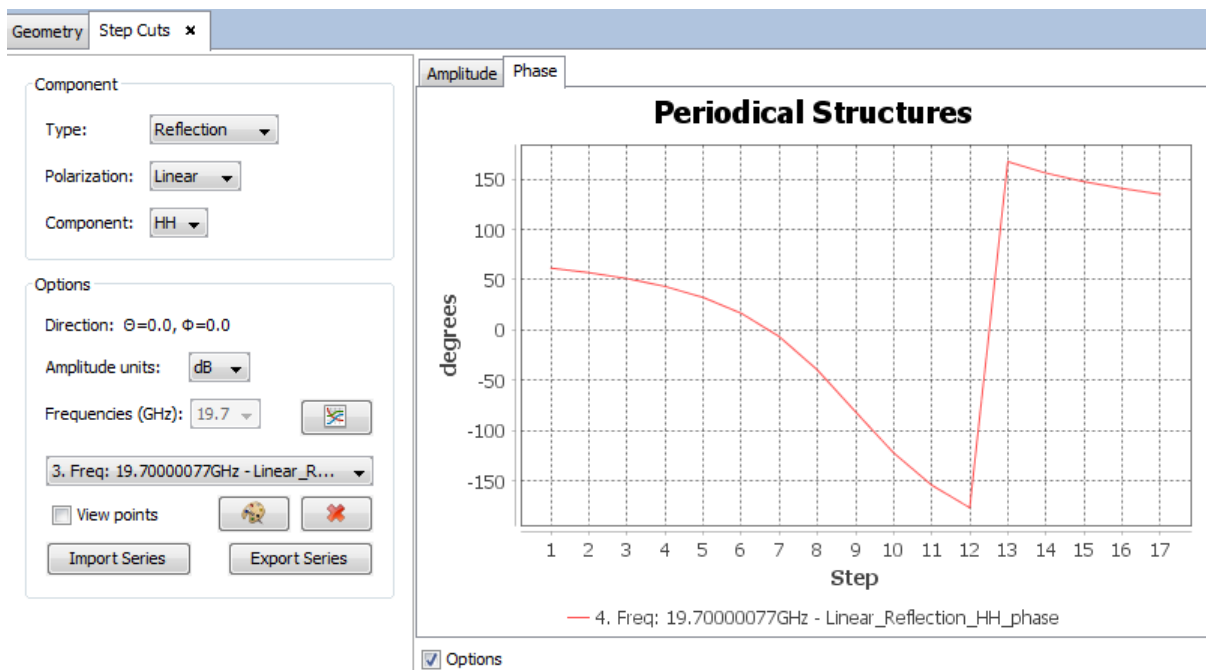


Figura 5.11: Curva de fases del dipolo paramétrico

Ahora, para formar la cruz se han escogido como longitud del brazo largo la correspondiente al paso 11 y se ajustará longitud del brazo corto entre los pasos 5 y 6 para conseguir exactamente la diferencia de fase de 180° .

Utilizando la misma técnica, se ha creado la geometría de la cruz paramétrica con un brazo fijo de 6mm, parámetro 'a' que corresponde al paso 11, y un brazo variable con un parámetro 'b' que va de 4.8mm a 5mm.

La geometría de la cruz paramétrica se ha definido mediante un *script* ya que la interfaz de usuario permite importar archivos de texto con comandos que son ejecutados en orden por la consola (Sección 3.7).

```
# newFASANT Script File (.nfs)
#
# Creates a cross with two sizes.
#   t: thickness
#   a: size of xArm
#   b: size of yArm
#
set t = 0.3
set a [4.8, 5.0] 21
set b {6.0}

# Auxiliary parameters defined automatically
set $1 = -t
set $2 = -t/2
set $3 = t/2
set $4 = a/2-$3
set $5 = -a/2-$2
set $6 = b/2-$3
set $7 = -b/2-$2

# Creates the cross arms with five rectangular surfaces
plane -n Center -p $2 $2 0.0 t t
plane -n ArmRight -p $3 $2 0.0 $4 t
plane -n ArmLeft -p $2 $3 0.0 $5 $1
plane -n ArmTop -p $2 $3 0.0 t $6
plane -n ArmBottom -p $3 $2 0.0 $1 $7

# Group the cross surfaces in an only object
group -s Center ArmRight ArmLeft ArmTop ArmBottom -n Cross
```

Tabla 5.2: Fichero de *script* para crear la geometría de la cruz paramétrica

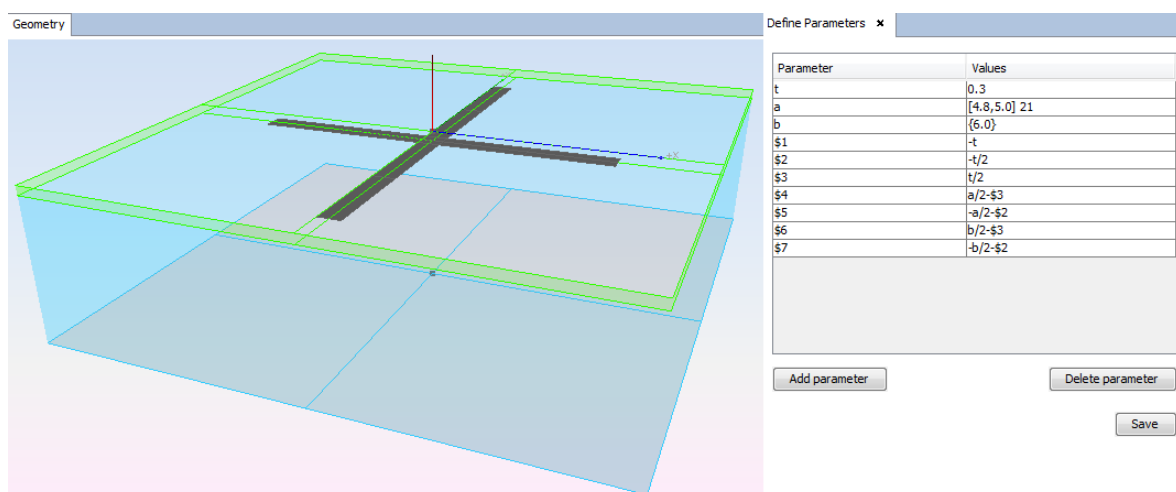


Figura 5.12: Simulación de una cruz paramétrica

En la figura 5.12 puede verse el resultado del *script* importado en el módulo de estructuras periódicas de la interfaz de usuario. A continuación se han realizado las 21

simulaciones correspondientes al parámetro 'b' para obtener la gráfica que se muestra en la siguiente figura.

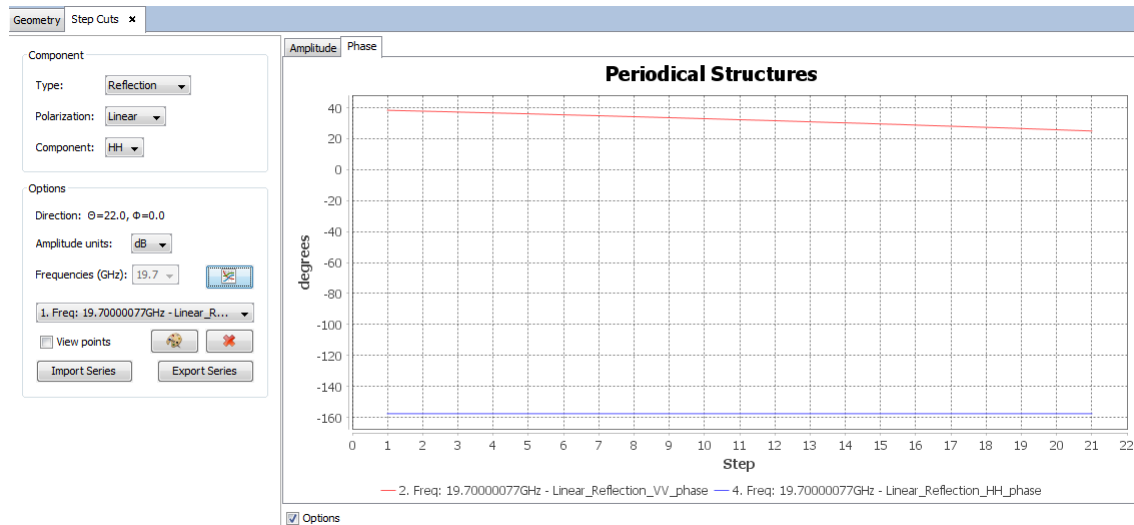


Figura 5.13: Curva de fases de la cruz paramétrica

En la gráfica se compara en polarización lineal la componente vertical (rojo) que varía según aumenta la longitud del brazo corto de la cruz, y la componente horizontal (azul) que no varía ya que el brazo largo siempre tiene la misma longitud.

A partir de esta gráfica se pueden obtener los valores óptimos de las longitudes de los brazos en los que la diferencia de fase entre la componente vertical y horizontal en polarización lineal es 180° , en este caso la cruz tiene un brazo largo de 6.0mm, un brazo corto de 4.9mm y una anchura de 0.3mm.

Creación de la base de datos

Una vez encontrado el elemento apropiado, la técnica de rotación variable (VRT) establece que si se rota el elemento con un ángulo de rotación α , la fase de las componentes circulares tendrá una variación lineal de $\pm 2\alpha$ dependiendo de si es componente circular a izquierdas o a derechas. En la práctica este comportamiento permite que al diseñar el reflectarray para desviar el haz de una de las componentes circulares, automáticamente se desvíe el haz de la componente circular contraria en la dirección opuesta, permitiendo generar dos haces simétricos sin ninguna pérdida de ganancia.

Para generar la base de datos se ha añadido el parámetro '*giro* = [0, 180] 1' al proyecto de la cruz que se utilizará en la operación *rotate* sobre la geometría de la cruz, para realizar simulaciones paramétricas variando el ángulo de rotación. El resto de parámetros de la cruz se han puesto fijos a las dimensiones del elemento optimizado. En figura 5.14 se muestra la geometría de la cruz rotada para el paso 45.

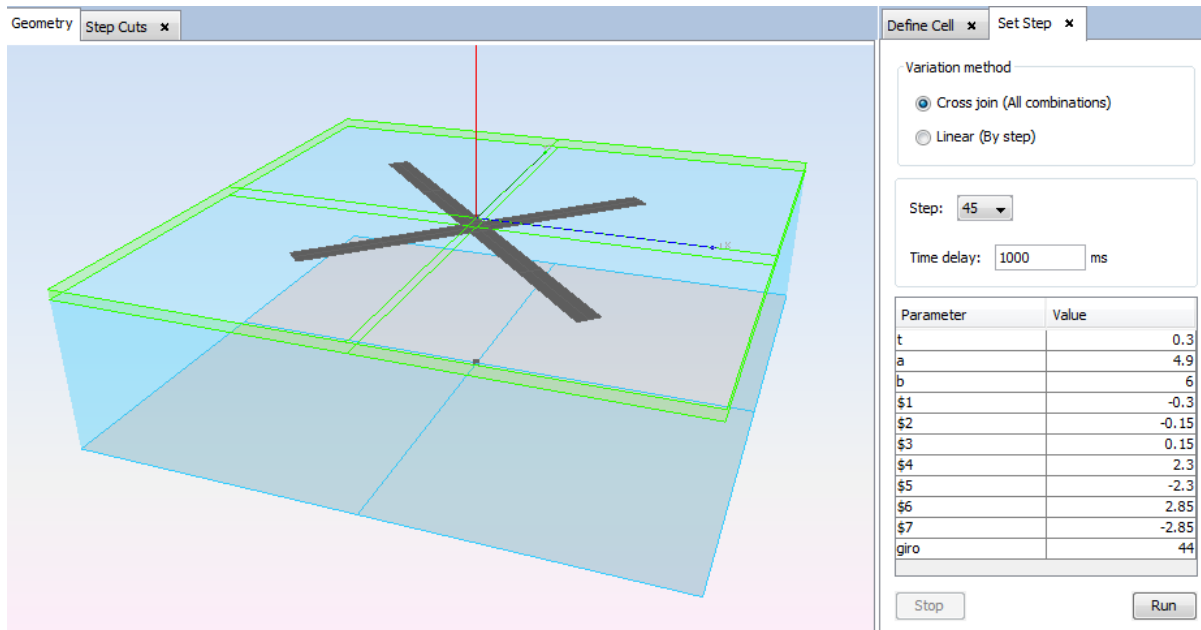


Figura 5.14: Simulación de una cruz paramétrica variando el ángulo de rotación

Ahora se pueden simular las 180 rotaciones de la cruz con la interfaz de usuario de manera automática. En la siguiente figura se muestra la fase en polarización circular de las componentes R_{rr} y R_{ll} .¹ donde se puede ver la pendiente aproximada de $\pm 2\alpha$ para ambas polarizaciones.

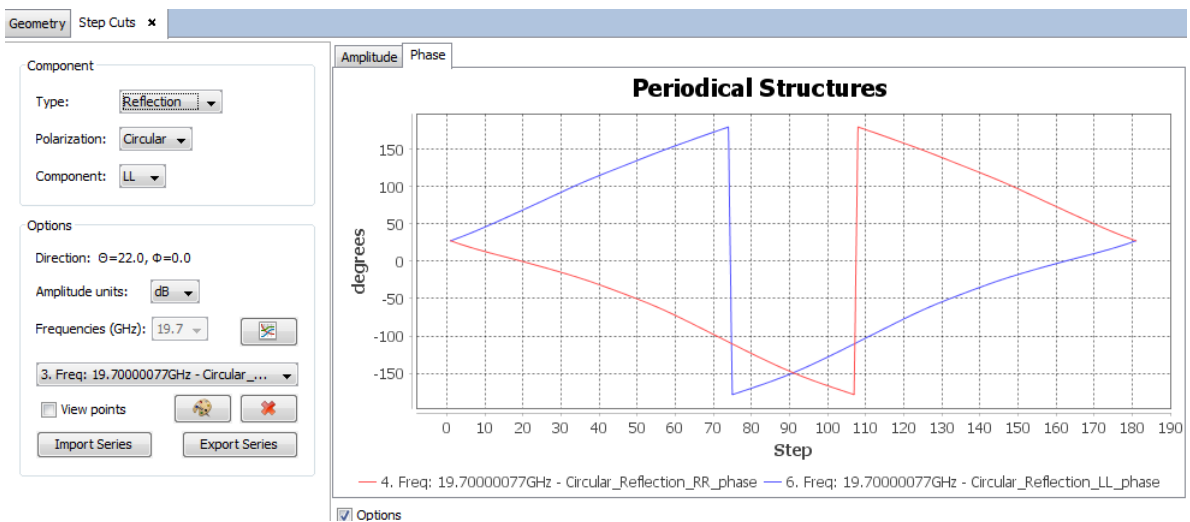


Figura 5.15: Fases en polarización circular de la cruz rotada (VRT)

Optimización de la base de datos

Si se utiliza la base de datos anterior para construir el reflectarray se obtiene una polarización circular cruzada demasiado alta (-15dB). El problema principal de este tipo de reflectarrays es que son muy sensibles a los errores de fase y es muy importante que se

¹ R_{rr} y R_{ll} son los coeficientes de reflexión de las componentes copolares del campo eléctrico reflejado de polarización circular a derechas y a izquierdas.

mantengan los mismos valores de fase de las componentes en polarización lineal para cada elemento rotado, manteniendo la diferencia de 180° entre las polarizaciones VV y HH.

En la siguiente gráfica se pueden ver los datos de la figura 5.15 centrada en la rotación de 0° , se puede apreciar el error de fase en la pendiente de las rectas.

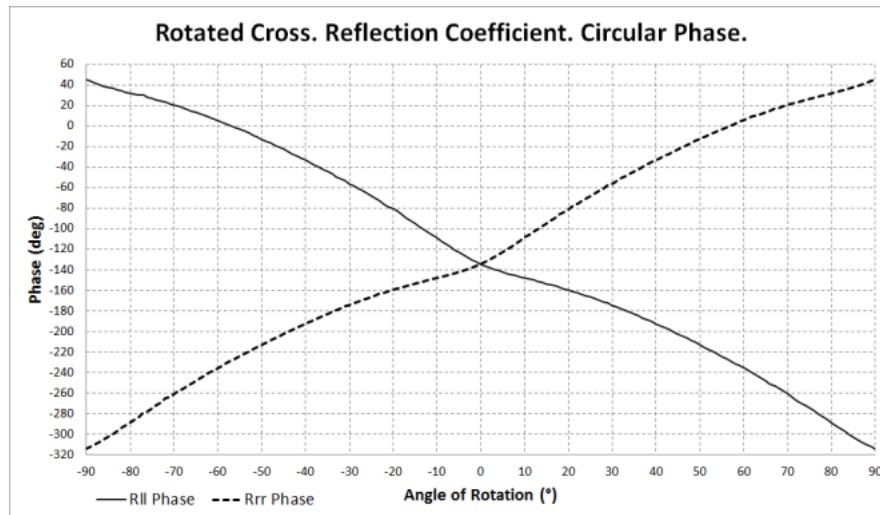


Figura 5.16: Fases en circulares de la base de datos sin optimizar

En la siguiente gráfica se puede ver como varía la componente contrapolar en polarización circular con el ángulo de rotación.

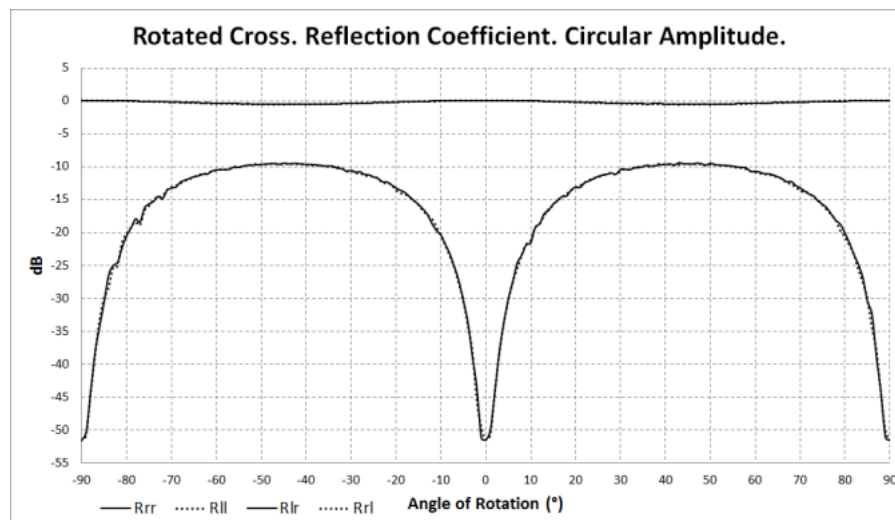


Figura 5.17: Amplitud en circulares de la base de datos sin optimizar

Para ver el error de fase, se simuló la estructura periódica de cruces rotadas a 45° incidiendo con una onda plana paralela a los brazos de la cruz $\varphi = 45^\circ$ y se vio que la diferencia de fase en lineales era de 158° . La explicación es que al rotar los elementos, la distancia entre unas cruces y otras varía, como puede apreciarse en la siguiente imagen.

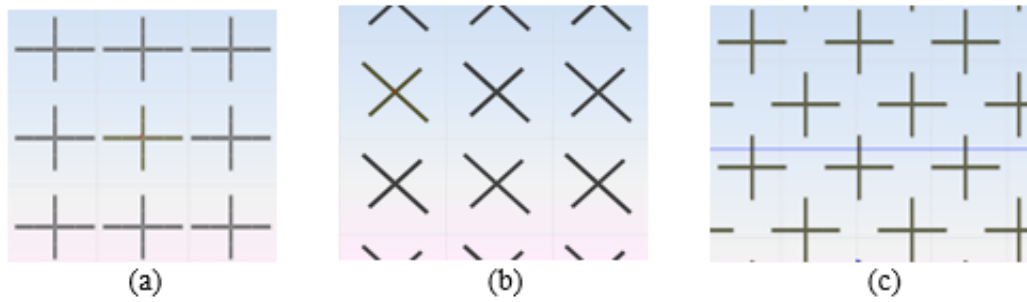


Figura 5.18: Problemas de fase al rotar los elementos

La solución propuesta fue buscar una cruz optimizada a 45° que tuviera exactamente los mismos valores de fase que la cruz optimizada a 0° . Para ello se usó exactamente el mismo proceso de diseño que para la cruz de 0° . Al observar que esto mejoraba mucho la contrapolar de la base de datos a 45° , se hizo lo mismo a 22.5° y después se interpoló el tamaño de las cruces entre estos valores dando lugar a una base de datos con una contrapolar muy baja que va de -60° a 60° de rotación. En la siguiente tabla se pueden ver los tamaños de las cruces optimizadas.

<i>Cruz</i>	<i>Longitud del brazo largo</i>	<i>Longitud del brazo corto</i>
0.0°	6.0mm	4.9mm
22.5°	6.14mm	4.91mm
45.0°	6.24mm	4.92mm

Tabla 5.3: Tamaño de las cruces optimizadas

En la siguiente gráfica se muestra el error de fase en polarización lineal que se comete en la base de datos optimizada para cada cruz, incidiendo con un campo paralelo a los brazos de la misma para mirar la diferencia de fase de 180° .

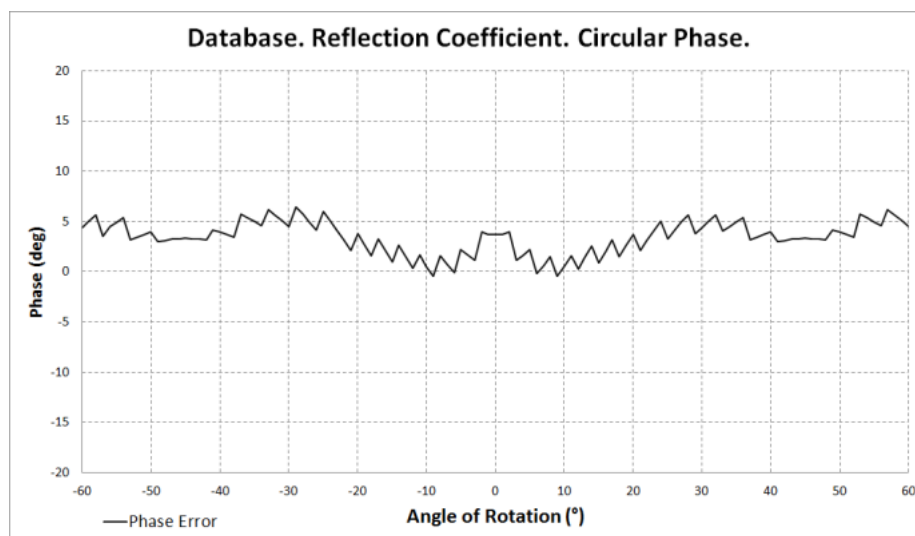


Figura 5.19: Errores de fase de las cruces optimizadas

A continuación se muestran las curvas de fase de las cruces optimizadas; se puede comparar con la figura 5.16 para comprobar que ahora se mantiene mucho mejor la pendiente de $\pm 2\alpha$ teórica de la técnica de rotación variable.

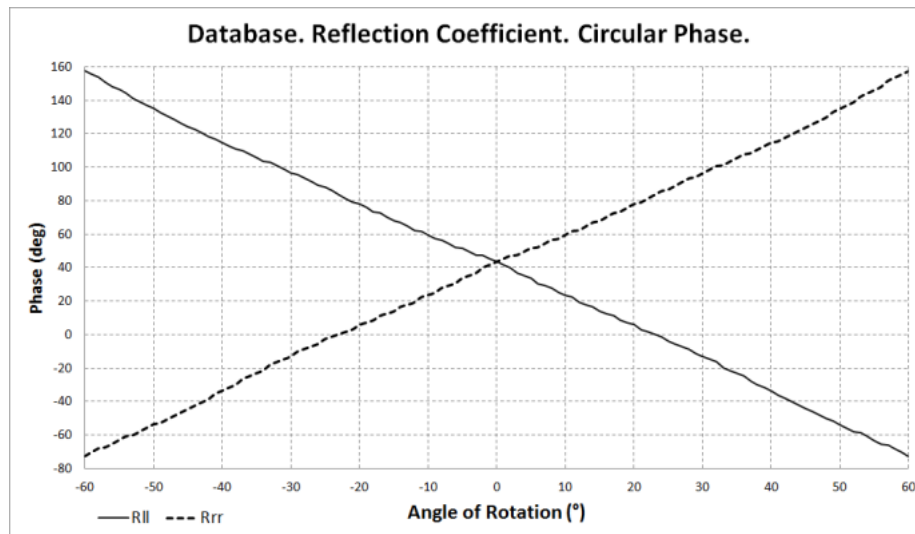


Figura 5.20: Fase en circulares de la base de datos optimizada

También se puede observar en la siguiente figura, comparando con la figura 5.17, que la componente contrapolar en polarización circular ha disminuido.

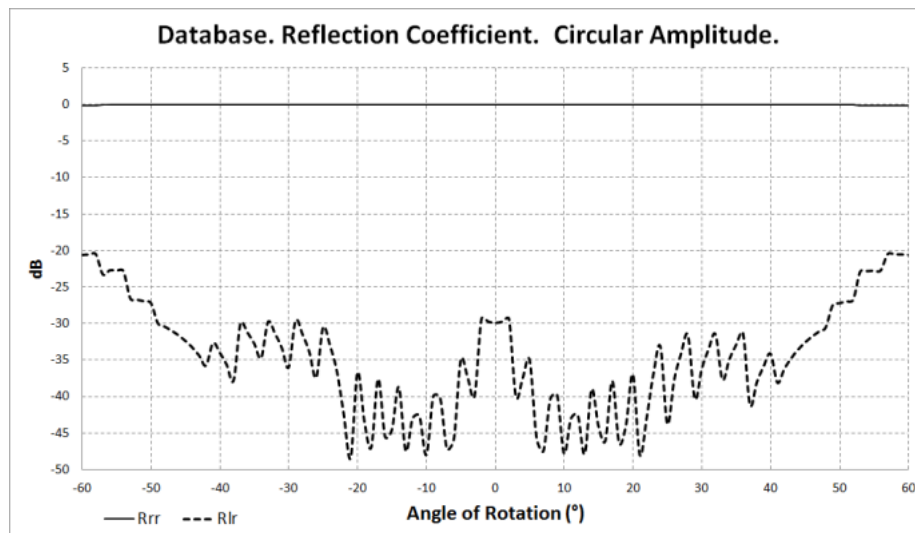


Figura 5.21: Amplitud en circulares de la base de datos optimizada

5.4 Diseño del Reflectarray Parabólico

Una vez que se ha realizado la base de datos en la Sección 5.3, se puede generar el reflectarray en el módulo MOM con la herramienta de la interfaz de usuario que se muestra a continuación.

Para crear el reflectarray parabólico se va a utilizar el proyecto de la sección 5.2, que contiene la superficie parabólica y los parámetros de simulación del reflectarray, solo habrá que modificar la posición y polarización circular de la antenna para realizar las simulaciones.

El objetivo del reflectarray es generar un gran número de haces superpuestos utilizando un array de antenas sobre la superficie parabólica. El diseño propuesto en este capítulo permite reducir el número de antenas a la mitad ya que cada bocina va a generar dos haces simétricos en polarización circular.

Según los resultados de la simulación de la antenna, en la figura 5.6, el ancho del haz principal es de 1.12° por lo que la desviación de cada haz en polarización circular será de 0.56° y cada antenna del array estará separada 55mm de la bocina central en las coordenadas (x, y) del plano paralelo al reflector con centro en el foco.

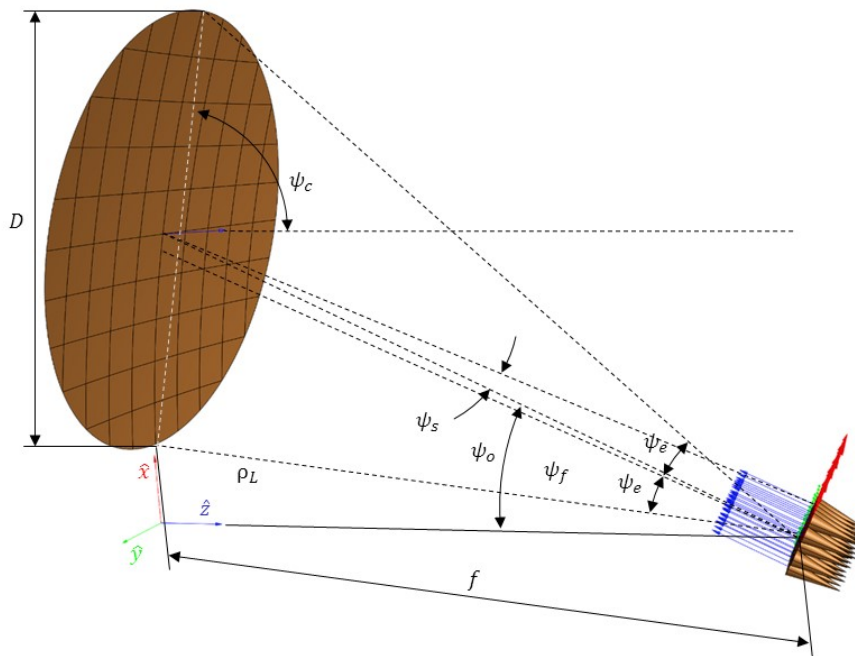


Figura 5.22: Alimentación del reflectarray parabólico

El primer paso es generar el reflectarray y comprobar su funcionamiento con una bocina situada en el foco de la parábola y apuntando hacia el centro del reflector. Al simular el reflectarray con polarización circular RHCP y LHCP se obtendrán dos haces con una desviación de $\theta = \pm 0,28^\circ$ para el corte $\varphi = 0^\circ$, lo que corresponde a una separación de 0.56° entre haces.

Para generar el reflectarray hay que seleccionar los elementos que van a ir en cada posición del reflectarray de la base de datos. En la interfaz de usuario los elementos se seleccionan a partir de la curva de fase de uno de los coeficientes de reflexión R_{vv} o R_{hh} para reflectarrays con polarización lineal y R_{rr} o R_{ll} para polarización circular. En nuestro caso se puede elegir R_{rr} o R_{ll} indistintamente, pero afectará al sentido de rotación de las cruces dentro del reflectarray.

Para elegir el elemento del reflectarray que va en una posición se utilizan las funciones de usuario que son evaluadas mediante el paquete 'BeanShell' (Sección 3.8.2). El usuario introduce los parámetros del reflectarray en la siguiente ventana, selecciona la base de datos y la función de usuario que va a utilizar.

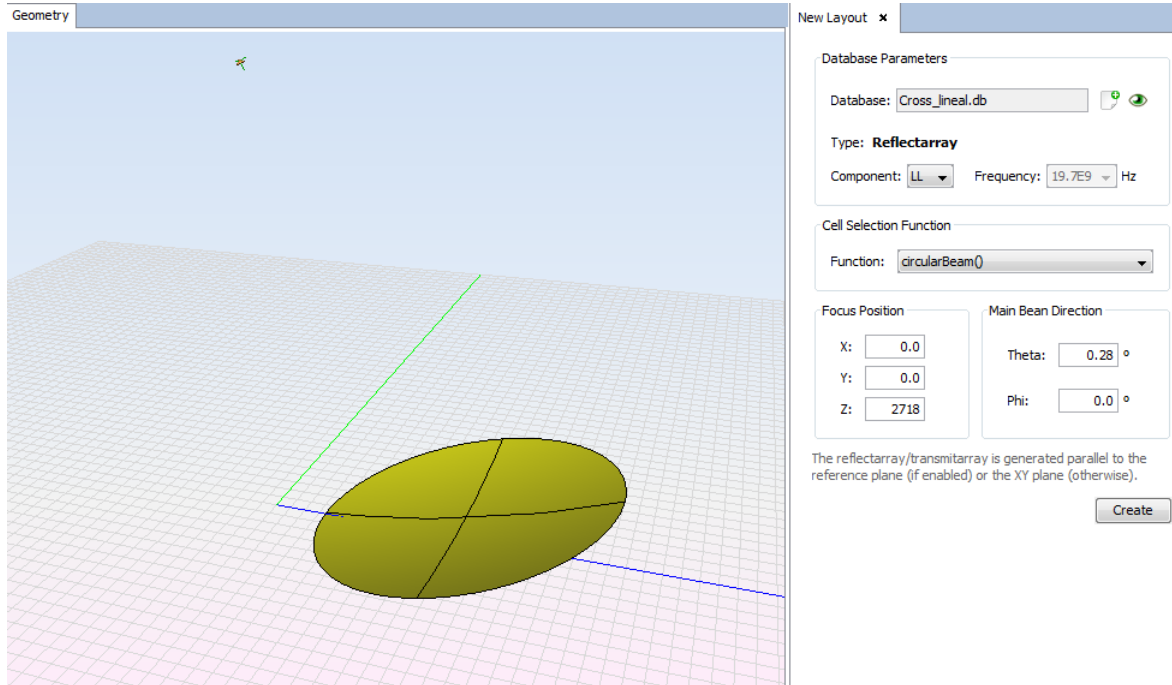


Figura 5.23: Parámetros del reflectarray parabólico

La interfaz de usuario divide la superficie en celdas del tamaño de la celda de la base de datos y llama a la función de usuario con los parámetros de la ventana y el centro de cada celda. La función de usuario calcula el valor de la fase para una posición del reflectarray. Según la componente seleccionada, la interfaz selecciona la celda de la base de datos con esa fase y la coloca sobre la superficie.

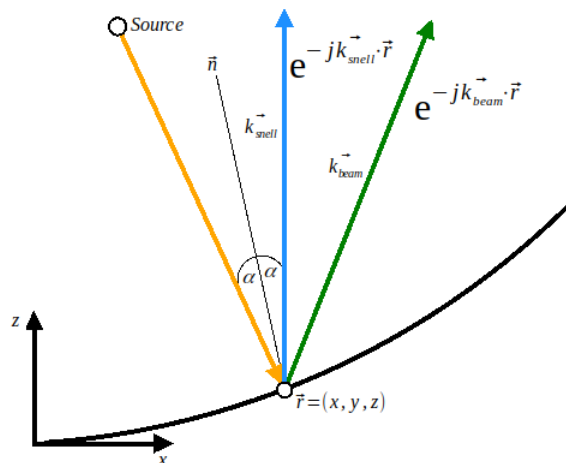


Figura 5.24: Esquema de funcionamiento del reflectarray

En la figura 5.24 se muestra el funcionamiento del reflectarray parabólico. En la parábola original, el campo incidente en el punto \vec{r} se refleja según el vector \vec{k}_{snell} con fase $e^{-j\vec{k}_{snell}\vec{r}}$. Si se quiere desviar el campo reflejado para que salga con la dirección del vector \vec{k}_{beam} se coloca un elemento del reflectarray en esa posición que introduzca la fase adicional determinada por la siguiente expresión.

$$\phi = e^{-j(\vec{k}_{beam} - \vec{k}_{snell})\vec{r}} \quad (5.1)$$

A continuación se muestra la función de usuario que devuelve la fase apropiada para cada elemento del reflectarray. La interfaz evalúa la función para todas las celdas del reflectarray modificando las variables $\$x$, $\$y$, $\$z$. El resto de parámetros que comienzan por '\$' son los que ha introducido el usuario en la ventana de configuración del reflectarray de la figura 5.23. Las funciones de usuario se almacenan en ficheros de texto dentro del directorio 'functions'.

Algoritmo 5.1 Función de usuario para generar el reflectarray 'circularBeam.java'

```

double circularBeam() {
    /*
    Parameters:
    - Frequency (Hz): $freq
    - Cell position: $x, $y, $z
    - Focus position: $focusX, $focusY, $focusZ
    - Main beam direction: $theta, $phi
    */
    double lambda = 3.0E08/$freq;
    double snellTheta = 0;
    double snellPhi = 0;
    double faseInicial = 53.493;

    //Distancia al plano de snell
    double roX1 = Math.sin(Math.toRadians(snellTheta))*Math.cos(Math.toRadians(snellPhi));
    double roY1 = Math.sin(Math.toRadians(snellTheta))*Math.sin(Math.toRadians(snellPhi));
    double roZ1 = Math.cos(Math.toRadians(snellTheta));
    double rixro1 = ($x*roX1)+($y*roY1)+($z*roZ1);

    //Distancia al plano de apuntamiento
    double roX = Math.sin(Math.toRadians($theta))*Math.cos(Math.toRadians($phi));
    double roY = Math.sin(Math.toRadians($theta))*Math.sin(Math.toRadians($phi));
    double roZ = Math.cos(Math.toRadians($theta));
    double rixro = ($x*roX)+($y*roY)+($z*roZ);

    //Fase objetivo
    double phaseRad = (2.0*Math.PI/lambda)*(rixro1-rixro);
    double phaseDeg = (Math.toDegrees(phaseRad) + faseInicial) % 360;
    if (phaseDeg < 0){phaseDeg = phaseDeg + 360;}
    phaseDeg = phaseDeg - 180;
    return Math rint (phaseDeg*Math.pow(10,3))/Math.pow(10,3);
}

```

La variable *faseInicial* se utiliza para sumar un valor de fase a todos los elementos del reflectarray. Esto no afecta a su dirección de apuntamiento pero permite variar el elemento que se coloca en el centro del reflectarray. Para que el reflectarray diseñado tenga una

polarización cruzada óptima, se ha colocado la cruz con rotación de 0° en el centro del reflector; recordar que la base de datos optimizada, en la figura 5.21, está diseñada en el intervalo $(-60^\circ, 60^\circ)$.

Después de configurar los parámetros de la ventana del reflectarray (figura 5.23), hay que seleccionar la superficie objetivo y pulsar en el botón 'Create'. La interfaz crea una rejilla de tamaño suficiente sobre el plano de referencia y coloca cada celda de la base de datos en la posición que corresponde según la función de usuario seleccionada. La superficie original corresponde a la capa superior del reflectarray. Para cada capa de la celda del reflectarray, la interfaz replica la superficie a la distancia que corresponda y si contiene geometría, coloca los elementos sobre la rejilla y los proyecta en la superficie de esa capa.

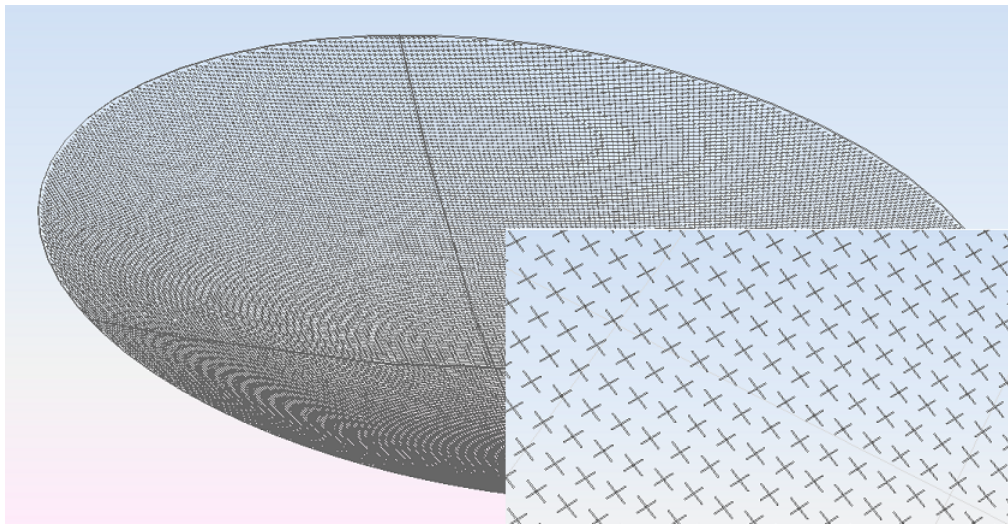


Figura 5.25: Layout del reflectarray parabólico

Una vez generado el *layout* del reflectarray parabólico el siguiente paso es analizarlo con la interfaz de usuario en el módulo MOM para comprobar su funcionamiento. En la siguiente figura se muestra el mallado de la geometría a 19.7GHz y 10 divisiones por lambda.

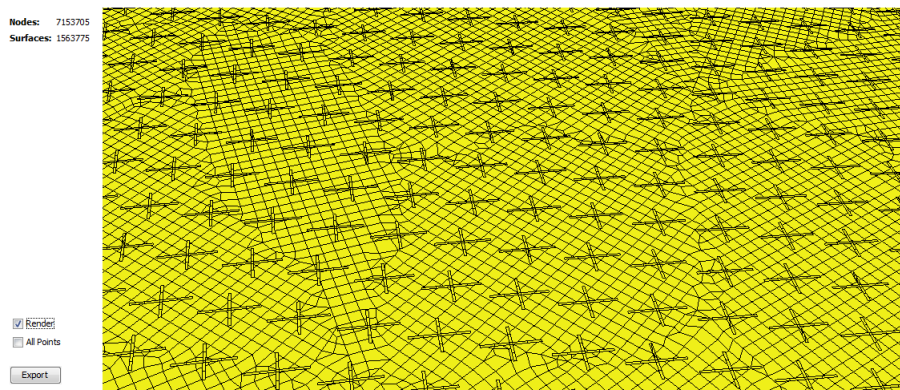


Figura 5.26: Mallado del reflectarray parabólico

Manteniendo la posición de la alimentación de la figura 5.1 y el resto de parámetros, se ha simulado el reflectarray con las dos polarizaciones circulares RHCP y LHCP. En la siguiente gráfica se muestran los resultados para el corte $\varphi = 0^\circ$ y $\theta = [-20^\circ, 20^\circ]$, se puede comprobar que se desvían los haces correctamente sin pérdida de ganancia comparado con los resultados del reflector de la figura 5.6.

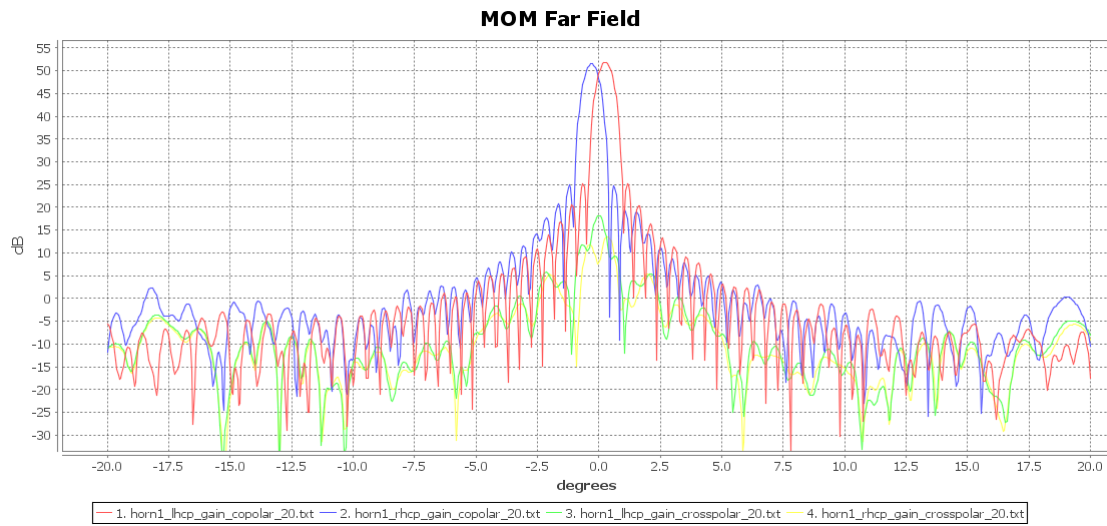


Figura 5.27: Resultados del reflectarray parabólico alimentado con LHCP y RHCP

Se puede apreciar mejor la superposición de los haces en la siguiente gráfica que muestra el corte $\varphi = 0^\circ$ y $\theta = [-10^\circ, 10^\circ]$.

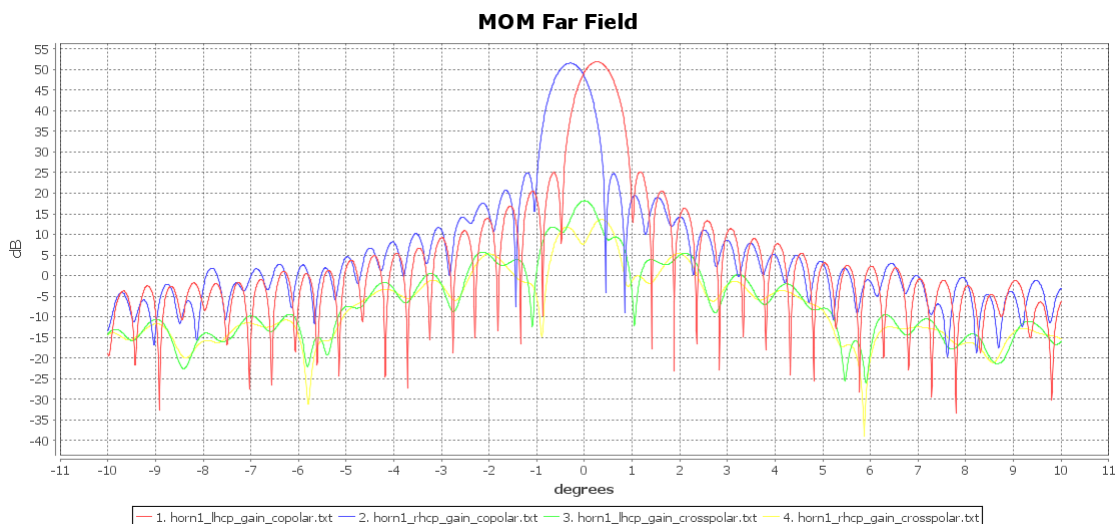


Figura 5.28: Resultados del reflectarray parabólico alimentado con LHCP y RHCP (2)

Alimentación con un array de bocinas

Para comprobar el funcionamiento de la alimentación del reflectarray con el array de bocinas mostrado en la figura 5.22, se han realizado las simulaciones colocando tres bocinas, separadas 55mm en el plano paralelo al reflector con centro en el foco, de manera que

puedan superponerse los resultados para el corte $\varphi = 0^\circ$. En la siguiente figura se muestra el esquema de alimentación.

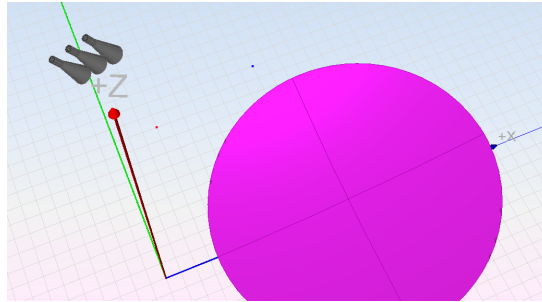


Figura 5.29: Alimentación con 3 bocinas

A continuación se muestran los resultados del reflectarray parabólico alimentado con las tres bocinas con polarización RHCP y LHCP para el corte $\varphi = 0^\circ$ y $\theta = [-20^\circ, 20^\circ]$.

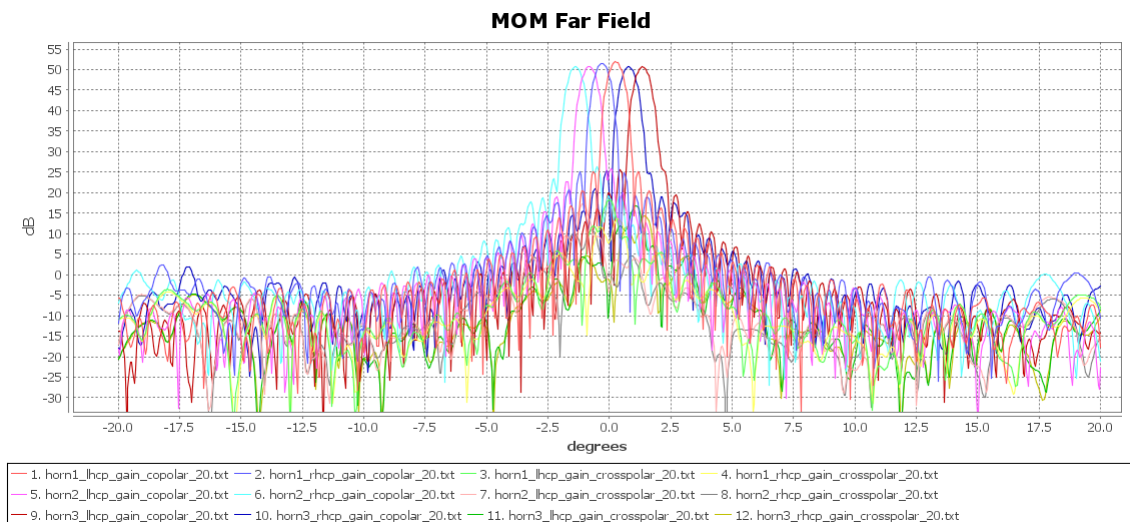


Figura 5.30: Resultados del reflectarray parabólico alimentado con tres bocinas

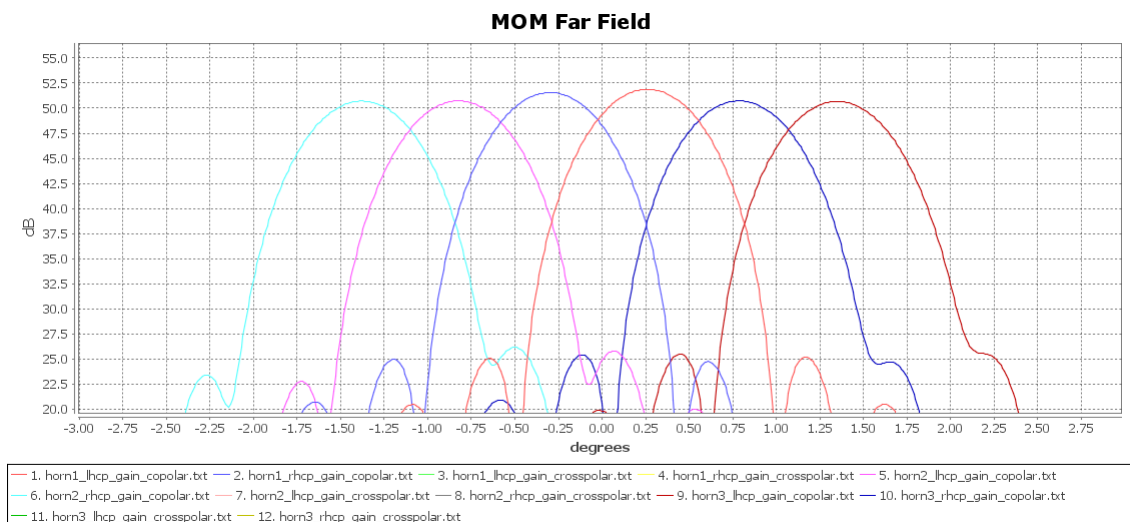


Figura 5.31: Resultados ampliados del reflectarray parabólico alimentado con tres bocinas

5.5 Diseño y Fabricación de un Demostrador

Para comprobar el funcionamiento del reflectarray propuesto se ha diseñado un demostrador más pequeño utilizando el mismo elemento. El demostrador se ha simulado de la misma forma que el reflectarray parabólico y posteriormente ha sido construido y medido.

La superficie del demostrador es una placa plana de 25x25cm centrada en el origen de coordenadas. El centro de fases de la bocina se encuentra en la posición $(-0.1108, 0.0, 0.2384)$ (m) apuntando hacia la posición $(-0.0145, 0, 0)$ (m).

Se ha calculado la diferencia de fase que tienen unas cruces de una fila a otra del reflectarray para conseguir desviar el haz $\pm 10^\circ$ respecto a su ángulo de Snell de 22° . La diferencia de fase es de 28.5455° . Al igual que en el reflectarray parabólico, se ha optimizado la disposición de los elementos colocando la cruz con una rotación de 0° en la fila que se encuentra en la posición apuntada por la bocina.

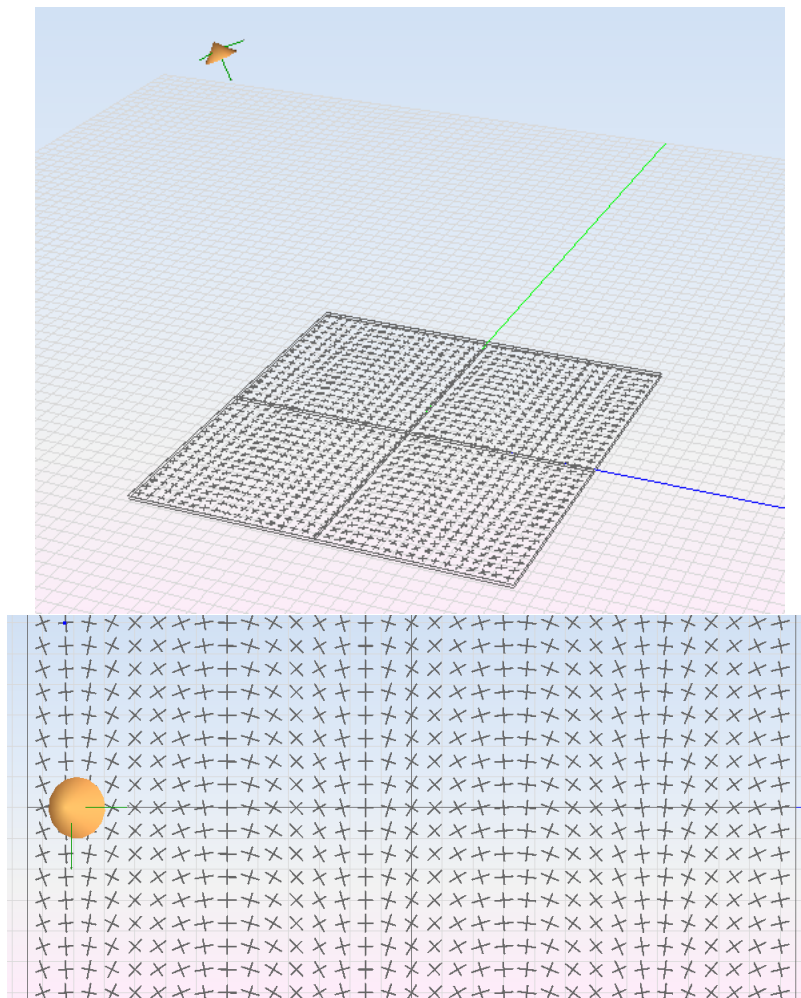


Figura 5.32: Demostrador generado en la interfaz de usuario

El demostrador se ha simulado en el módulo MOM a 19.7GHz con ambas polarizaciones circulares RHCP y LHCP. En las figuras 5.33 y 5.34 se pueden ver los resultados de ganancia en circulares del demostrador simulados con la interfaz de usuario. Se puede

observar que se cumple la desviación del haz de $\pm 10^\circ$ respecto al ángulo de Snell. La gráfica corresponde al corte $\varphi = 0^\circ$ y $\theta = [-90^\circ, 90^\circ]$.

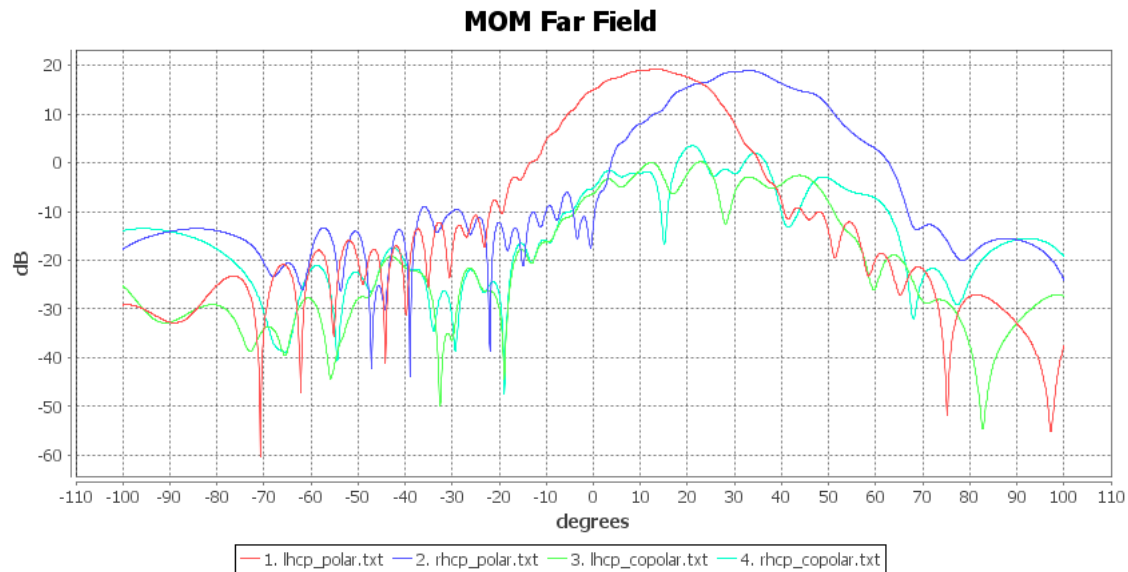


Figura 5.33: Resultados del demostrador en la interfaz de usuario

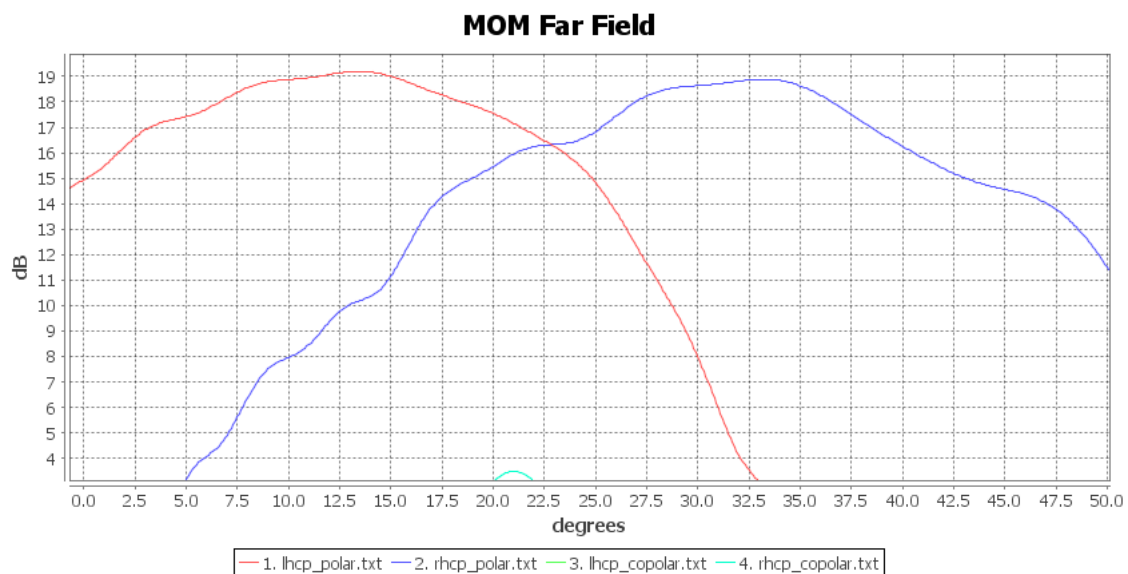


Figura 5.34: Resultados ampliados del demostrador

Para la fabricación del demostrador el primer paso es generar el fichero de geometría con el *layout* del reflectarray. Este fichero contiene la geometría de las cruces así como la colocación de los agujeros para los taladros y unos dipolos de control.

El fichero ha sido generado mediante la herramienta AutoCAD con la colaboración de D. Rafael Florencio Díaz, exportando la geometría de las cruces del demostrador con la interfaz de usuario e importándolas en el programa de diseño externo. Una vez generado el fichero, se ha mandado construir la lámina que contiene los elementos metálicos del demostrador. En la figura 5.35 se puede ver el fichero de geometría.

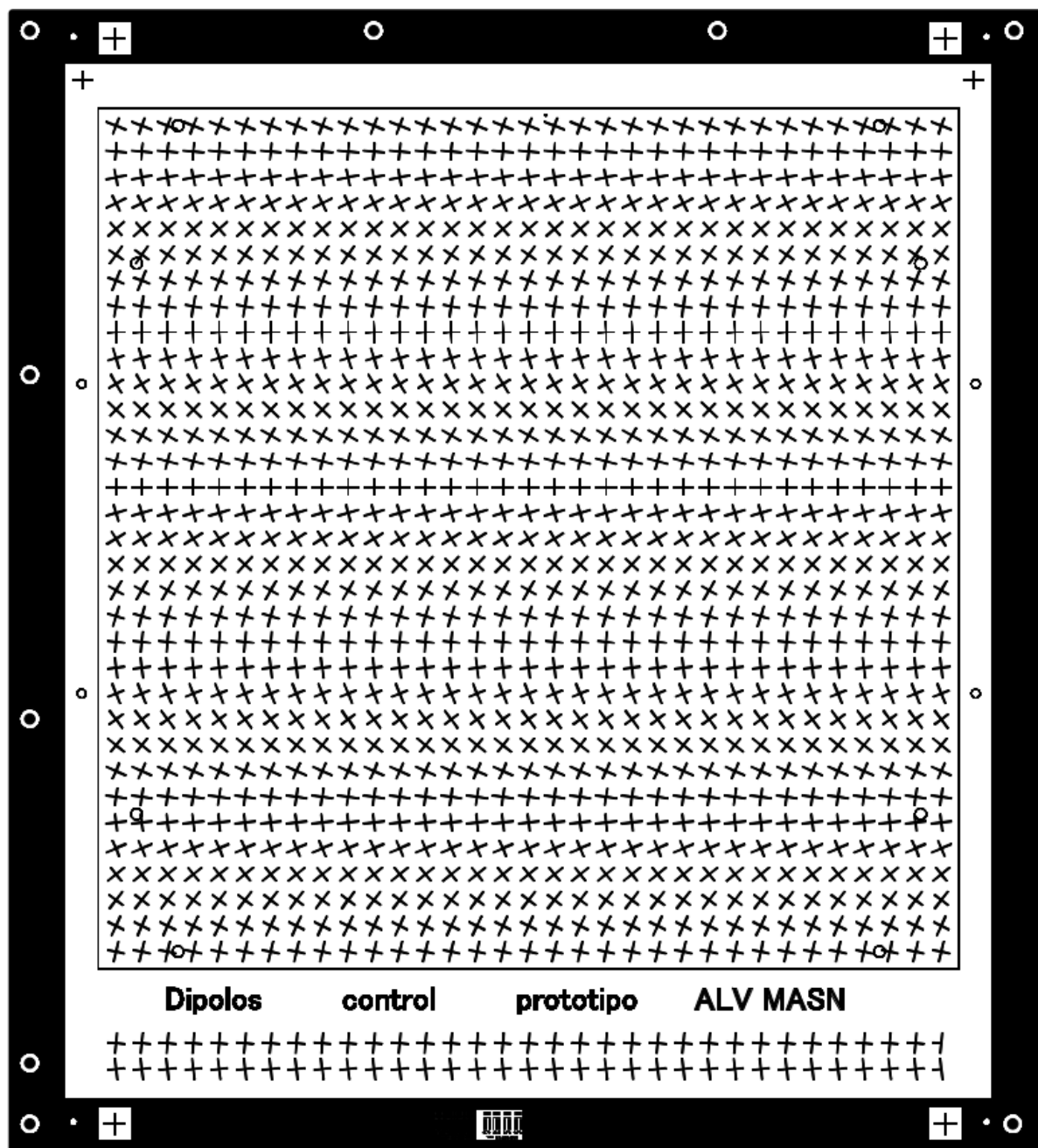


Figura 5.35: Fichero de geometría que contiene el layout del demostrador

El montaje del demostrador y la bocina ha sido realizado gracias a la ayuda de Miguel A. Salas Natera, de la Universidad Politécnica de Madrid, que se ha encargado de ensamblar las distintas capas de materiales del reflectarray y fabricar la estructura donde va colocada la antena.

Una vez fabricado, el demostrador ha sido medido en la cámara anecoica del Laboratorio de Ensayo y Homologación de Antenas en la Universidad Politécnica de Madrid. En primer lugar, se realiza la medida de la bocina aislada y posteriormente de todo el sistema para poder calcular después la ganancia de la antena. En las siguientes figuras se muestra el demostrador fabricado y el proceso de medida.

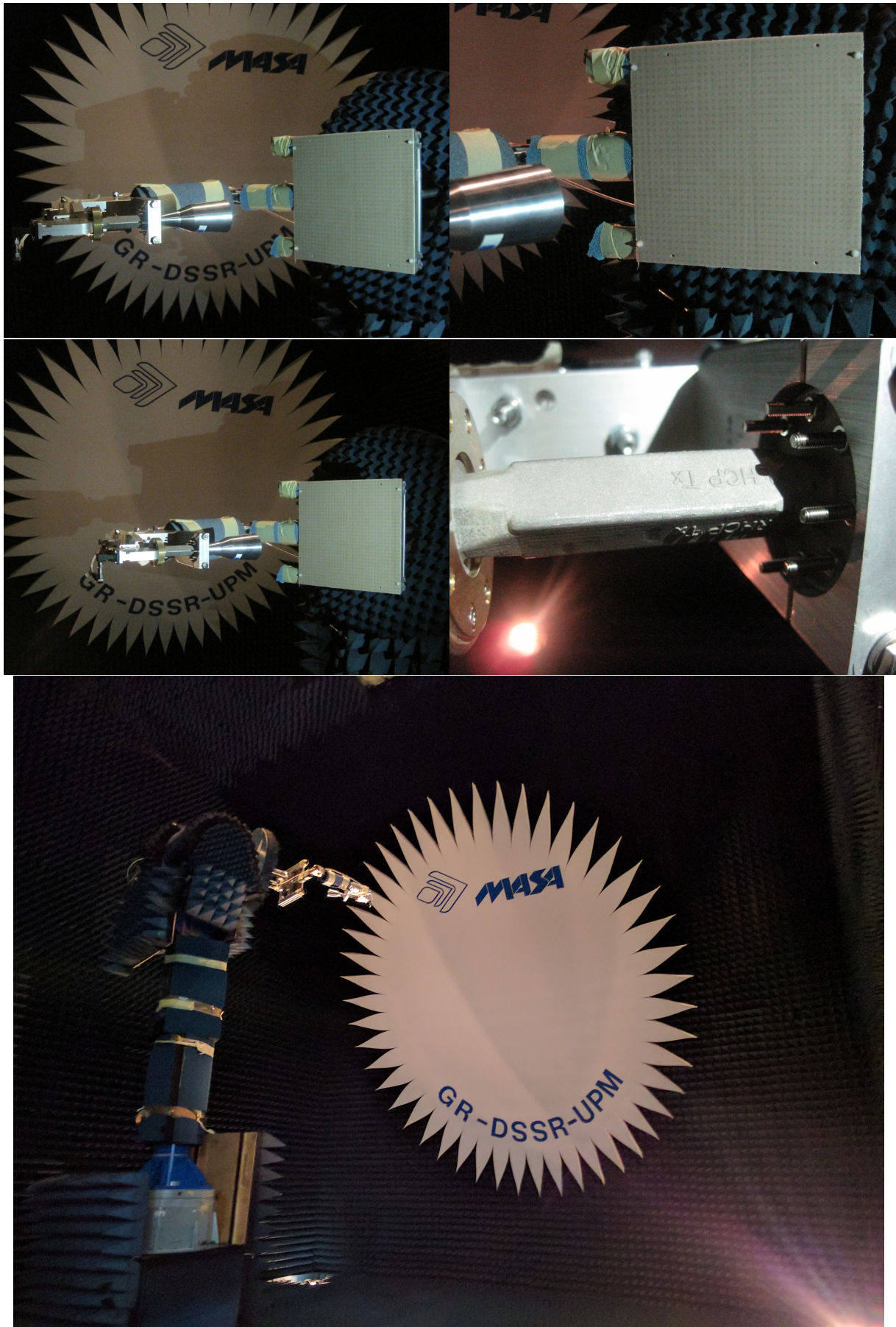


Figura 5.36: Fotografías del demostrador en la cámara anecoica de la UPM

En la siguiente gráfica se muestra el resultado de la ganancia en circulares para el corte $\varphi = 0^\circ$ de las medidas del demostrador para la banda de 19.7GHz.

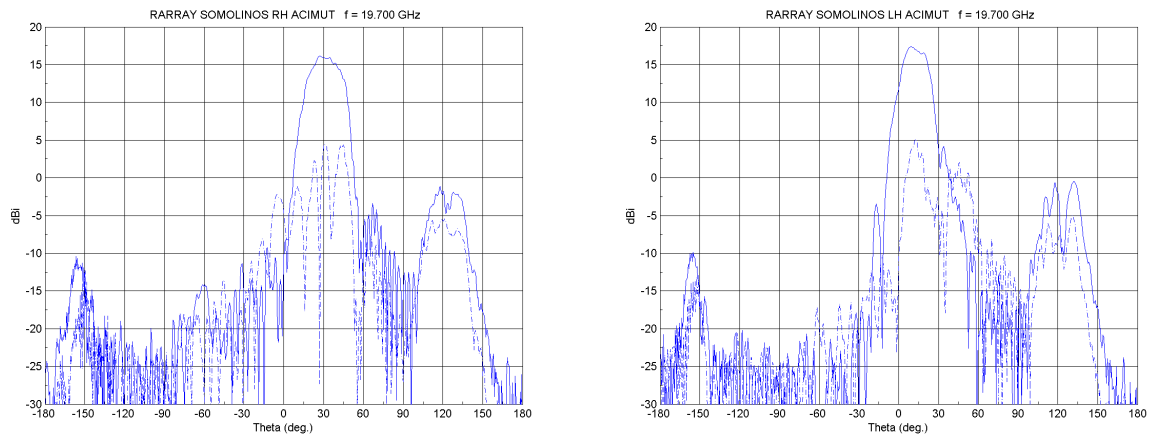


Figura 5.37: Medidas del demostrador RHCP y LHCP

En la figura 5.38 se muestran los resultados de las medidas junto a los de las simulaciones para las dos polarizaciones del demostrador, se puede observar que el haz se desvía en la dirección adecuada para cada polarización, por lo tanto, es posible utilizar la técnica de rotación variable (VRT) en un reflectarray para obtener dos haces superpuestos en polarización circular utilizando una cruz como elemento, independientemente de si la superficie es plana o parabólica.

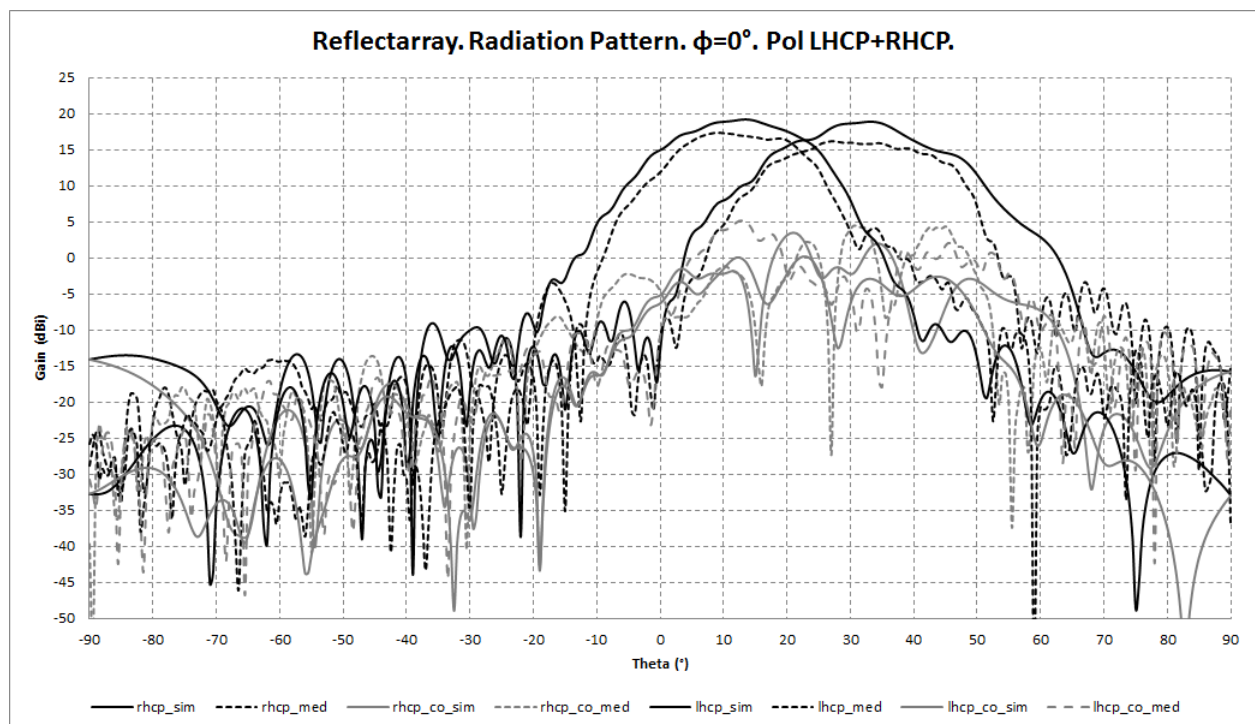


Figura 5.38: Resultados ampliados del demostrador

Bibliografía

- [1] J. Huang y J. A. Encinar, "Reflectarray Antennas," in *IEEE Press*, Piscataway, NJ, USA, 2008.
- [2] D. M. Pozar, S. D. Targonski, y R. Pokuls, "A shaped-beam microstrip patch reflectarray," *IEEE Transactions on Antennas and Propagation*, 1999.
- [3] J. A. Encinar y J. A. Zornoza, "Three-layer printed reflectarrays for contoured beam space applications," *IEEE Transactions on Antennas and Propagation*, 2004.
- [4] J. Shaker, M. R. Chaharmir, y J. Ethier, "Reflectarray Antennas: Analysis, Design, Fabrication, and Measurement," in *Artech House*, Norwood, MA, USA, 2014.
- [5] J. A. Encinar, L. S. Datashvili, J. A. Zornoza, M. Arrebola, M. Sierra-Castaner, J. L. Besada-Sanmartin, H. Baier, y H. Legay, "Dual-polarization dual-coverage reflectarray for space applications," *IEEE Transactions on Antennas and Propagation*, 2006.
- [6] M. Zhou, S. B. Sørensen, O. S. Kim, E. Jørgensen, P. Meincke, y O. Breinbjerg, "Direct optimization of printed reflectarrays for contoured beam satellite antenna applications," *IEEE Transactions on Antennas and Propagation*, 2014.
- [7] M. Zhou, S. B. Sorensen, O. S. Kim, E. Jorgensen, P. Meincke, O. Breinbjerg, y G. Toso, "The generalized direct optimization technique for printed reflectarrays," *IEEE Transactions on Antennas and Propagation*, 2014.
- [8] E. M. de Rioja, J. A. Encinar, R. Florencio, y R. Boix, "Reflectarray in K and Ka Bands with Independent Beams in Each Polarization," in *IEEE International Symposium on Antennas and Propagation*, Fajardo, Puerto Rico, 2016.
- [9] J. Huang y R. J. Pogorzelski, "A Ka-band microstrip reflectarray with elements having variable rotation angles," *IEEE Transactions on Antennas and Propagation*, 1998.

Capítulo 6

Conclusiones y Futuras Líneas de Trabajo

6.1 Conclusiones

En esta tesis se ha presentado el diseño y la implementación de una nueva interfaz de usuario para el conjunto de software de simulación electromagnética newFASANT. El sistema presentado permite integrar el modelado geométrico de escenarios complejos en 3D, la simulación electromagnética y la visualización de resultados.

En la primera parte se han desarrollado las librerías para trabajar con NURBS donde se han estudiado e implementando los algoritmos necesarios para poder crear, visualizar y modificar este tipo de curvas y superficies en Java. Los algoritmos matemáticos son independientes de Java3D y pueden ser fácilmente adaptados a otras librerías gráficas como JOGL o LWJGL. En esta parte también se han diseñado los algoritmos de renderizado, que permiten convertir las curvas y superficies NURBS en objetos que puedan ser dibujados por el motor gráfico, y las operaciones avanzadas con NURBS, que permiten el diseño de curvas y superficies arbitrarias que se adapten lo máximo posible al modelo real.

En la segunda parte se ha diseñado la interfaz de usuario y se ha desarrollado toda la parte de modelado geométrico. Primero se ha diseñado el panel de visualización integrando la librería de NURBS con Java3D y se ha desarrollado toda la funcionalidad necesaria para que el usuario pueda interactuar con la geometría: movimiento de la cámara, selección de objetos, plano de referencia, pick, etc. Posteriormente, se ha diseñado la consola de comandos que agrupa todas las operaciones que puede realizar el usuario sobre la geometría y permite las simulaciones paramétricas utilizando el historial de comandos y parámetros para definir la geometría. Por último, se han implementado en la interfaz varios formatos CAD para poder importar y exportar los modelos geométricos desde otras herramientas. En esta parte se ha creado un programa de diseño CAD completo, que permite a los usuarios diseñar o modificar la geometría que se va a simular sin tener que

recurrir a programas externos.

En la tercera parte se ha diseñado la estructura de los distintos módulos de simulación y se ha mostrado en detalle la implementación del módulo MOM, que utiliza el núcleo electromagnético MONURBS para realizar las simulaciones. En este apartado, se ha definido el proyecto de simulación y se han implementado las ventanas que recogen los datos de simulación. Después, se ha implementado el proceso de mallado y simulación para ejecutar los programas externos de manera transparente al usuario. Por último, se ha desarrollado la visualización de los resultados de simulación, para integrar todo el proceso en la interfaz de usuario.

Para terminar se ha realizado el proceso completo de diseño de un reflectarray, utilizando las herramientas de la nueva interfaz de usuario. La antena está conformada sobre una superficie parabólica y utiliza como elemento una cruz que permite dividir los haces en polarización circular. Para demostrar su funcionamiento se ha diseñado y fabricado un reflectarray más pequeño y se han comparado las medidas reales con las simulaciones. El proceso de diseño de un reflectarray es muy sencillo con la nueva interfaz de usuario. Para diseñar los elementos se utiliza el módulo de estructuras periódicas, mediante simulaciones paramétricas se puede variar la geometría automáticamente para generar una base de datos de elementos. Después, para generar el reflectarray se ha creado una novedosa herramienta en el módulo MOM, que utiliza una función definida por el usuario para seleccionar el elemento que debe ir en cada posición del reflectarray y lo conforma automáticamente en la superficie objetivo.

El resultado de este trabajo es que se ha renovado por completo la interfaz de usuario del software de simulación electromagnética newFASANT v6.x, desarrollando un programa robusto y fácil de mantener, que contiene un editor geométrico muy completo basado en superficies NURBS y estructurado en módulos que permiten resolver problemas electromagnéticos muy variados. La nueva versión del programa ha recibido muy buenas críticas por parte de los usuarios, aumentando el interés por la aplicación y permitiendo a newFASANT posicionarse como una alternativa viable a los grandes programas de simulación electromagnética.

6.2 Futuras Líneas de Trabajo

En los próximos años, la interfaz de usuario propuesta en esta tesis será mejorada y ampliada con nuevas características para seguir compitiendo con el resto de software de simulación electromagnética.

Actualmente, se están desarrollando las siguientes características:

- Es posible arrancar la interfaz en modo servidor en Windows o Linux para que escuche en un puerto. La interfaz cliente puede conectarse a una interfaz servidor para realizar

las simulaciones en un ordenador remoto. De forma transparente al usuario, el cliente envía el proyecto de simulación al servidor, el servidor realiza los cálculos y devuelve el proyecto al cliente con los resultados. El servidor puede instalarse en un HPC (clúster de alto rendimiento) y utilizar los sistemas de planificación SLURM y LSF para ejecutar las simulaciones.

- Si el usuario da su consentimiento, la interfaz guarda un registro de los fallos que se producen y los envía a un servicio web utilizando JSON, este crea un registro de errores para su posterior mantenimiento.
- Se están desarrollando nuevos módulos de simulación para resolver otros problemas, por ejemplo, el módulo de infrarrojos permite realizar análisis térmicos de la geometría. El programa ha suscitado el interés de otras empresas para incluir sus núcleos de simulación electromagnética como módulos de la interfaz de usuario.
- Se ha creado un asistente de proyectos con diversas plantillas con parámetros predefinidos y que permiten mostrar al usuario los distintos problemas que se pueden resolver con la herramienta.

Respecto a la inclusión de nuevas funcionalidades, sería interesante implementar las siguientes funcionalidades:

- Implementar un servidor que pueda aceptar peticiones de varios clientes, para ello habría que cambiar la estructura de los núcleos de simulación para especificar las rutas donde se leen y se escriben los ficheros. El programa debería manejar los datos en la carpeta '/home/user/.newfasant' o en 'AppData/Local/newfasant' en Windows ya que actualmente no es posible usar dos interfaces desde el mismo directorio de instalación simultáneamente.
- Implementar el renderizado de superficies NURBS sobre OpenGL 3.0 o superior (JOGL) utilizando el lenguaje GLSL para realizar los cálculos en la tarjeta gráfica o buscar alguna alternativa. Implementar un sistema de LOD (Level of Detail) para cambiar el nivel de detalle con el que se visualizan las curvas y superficies NURBS dependiendo de la distancia a la que se visualizan.
- Crear y optimizar las operaciones que permiten reconstruir y arreglar partes del modelo geométrico, también mejorar las operaciones booleanas entre superficies NURBS. Es muy importante que en el modelo geométrico exista una continuidad perfecta entre superficies adyacentes para obtener buenos resultados.

