UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE INGENIERÍA

Departamento de Ingeniería Informática

Herramienta de diseño de objetos y visualización de resultados para simulación de campo magnético

Juan Pablo Verdejo Jorquera

Profesor guía: Fernando Rannou, Ph.D.

Trabajo de titulación presentado
en conformidad a los requisitos para
obtener el título de Ingeniero de
Ejecución en Computación e Informática

Santiago – Chile
2015

RESUMEN

Escribir acá el resumen en español.

Palabras Claves: escribirKeyword1; escribirKeyword2; escribirKeyword3; etc

ABSTRACT

Write the abstract here.

Keywords: writeKeyword1; writeKeyword2; writeKeyword3; etc

DEDICATORIA

BLa bla bla. . .

AGRADECIMIENTOS

Agradezco a ...blablabla

Tabla de Contenido

Índice de Tablas

Tablas del Anexo A

Índice de Ilustraciones

CAPÍTULO 1. INTRODUCCIÓN

1.1 ANTECEDENTES Y MOTIVACIÓN

Investigadores del Departamento de Física de la Universidad de Santiago de Chile pertenecientes al Centro para el Desarrollo de la Nanociencia y la Nanotecnología (CEDENNA) trabajan en la simulación de los efectos del campo magnético en los átomos de distintos objetos, tomando en cuenta su forma, su material y su distribución atómica, entre otras características.

La simulación consiste en ejecutar eun programa computacional que implementa el Método Monte Carlo Cadenas de Markov (MCMC, *Monte Carlo Markov Chain*) sobre una grilla tridimensional de átomos. Uno de los grandes problemas de los usuarios de este software es que el proceso previo y posterior a la simulación es “manual”. Para definir el objeto a simular deben hacer un dibujo en Microsoft Paint®, el cual es analizado por un *script* de Matlab® que debe ser modificado para reflejar las características del objeto específico que se quiere simular. Para generar imágenes para, por ejemplo, una publicación, también deben ejecutar ciertos *scripts*, sin embargo esto es aún más complicado, ya que deben hacer ensayo y error hasta conseguir que la imagen sea representativa del resultado, puesto que muchas veces tienen errores de visualización que no reflejan el estado real del sistema. Estos dos sub-procesos hacen que el proceso de simulación sea tedioso, quitando mucho tiempo que podría ser usado en analizar los resultados.

Es aquí donde la informática puede contribuir, creando aplicaciones que mejoren estos procesos, que valoricen el tiempo de los científicos. Por consiguiente esta es una oportunidad única de ayudar a la obtención de conocimientos que permitan entender el entorno y de mejorar los procesos que permiten avanzar como sociedad hacia la comprensión del universo.

1.2 DESCRIPCIÓN DEL PROBLEMA

El proceso actual de diseño de estructuras atómicas a simular y su visualización ha sido creado por distintos investigadores que han sido parte del CEDENNA, los cuales en general solo tienen conocimientos básicos de diseño y desarrollo de software, esto ha llevado a que este proceso se base mayoritariamente en *scripts* de *MatLab*, los que deben ser modificados y

ejecutados manualmente para ir obteniendo los resultados deseados, esto resulta en la posibilidad de que los usuarios introduzcan errores fácilmente, los que pueden ser muy difíciles de detectar debido a la mala calidad de las pre-visualizaciones que se logran. Debido a la naturaleza de esta solución se requiere de un prolongado periodo de tiempo para poder generar datos para las simulaciones.

Una vez ejecutada la simulación es necesario poder visualizar los resultados de esta, para lo cual también se basan en *scripts* de *MatLab*, los que después de un largo proceso, debido a las distintas configuraciones manuales que deben hacer, genera imágenes de calidad pobre para publicaciones. Esto genera una gran pérdida de tiempo para los investigadores.

A continuación se describe con más detalle el proces actual que los investigadores realizan.

1.3 PROCESO ACTUAL

1.3.1 Diseño de objetos para simular

Actualmente los científicos no tienen un proceso definido y formal de creación de objetos para simulación, ya que existen diversas soluciones implementadas a través del tiempo, desde *scripts* escritos en *MatLab* que fueron creados especialmente para diseñar un objeto en específico hasta unos *scripts* más genéricos, pero igualmente engorrosos, usando como base un mapa de bits.

1.3.1.1 Diseño programático de objetos

La primera opción para diseñar objetos es de manera programática, es decir, escribiendo un *script* que defina todas las partículas de un objeto y sus vecindades. Para alguien con experiencia en esta área puede ser relativamente sencillo diseñar algunos objetos regulares, como por ejemplo un cilindro, no obstante el proceso se complica si se quieren diseñar estructuras irregulares. Dado que la visualización del objeto diseñado es de muy baja calidad con este sistema

es fácil cometer errores que eventualmente pueden invalidar los resultados de una simulación.

La figura 1.1 muestra un ejemplo de la visualización de un *script* de MatLab que genera un cilindro con radio $R = 60nm$ y altura $H = 20nm$ escalado a un sistema con solo 4 capas de átomos.

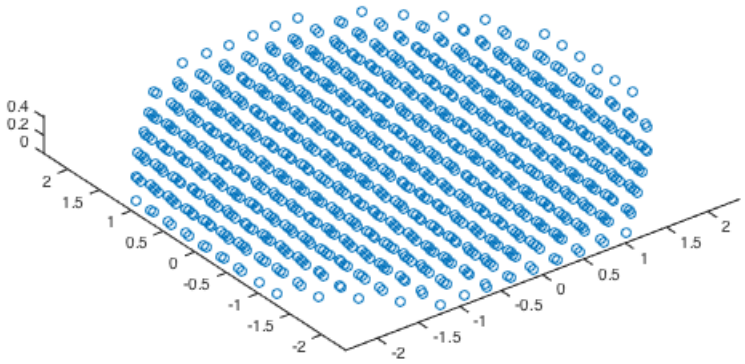

Figura 1.1: *Ejemplo de imagen salida de un* script *de generación de objeto*

1.3.1.2 Diseño en base a mapa de bits

Para flexibilizar el diseño de objetos y bajar la posibilidad de introducir errores en los datos de entrada se creó un sistema que permite especificar la geometría de un objeto en base a un mapa de bits construido en *Microsoft Paint*. De esta forma no es necesario escribir un progama cada vez qe se cambiase la geometría del objeto, si no que basta con crear una imagen que representa la primera capa del objeto y usar el *script* de *MatLab* para el tipo de estructura cristalina específica que se desea definir.

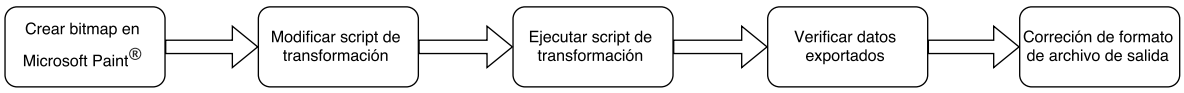

Figura 1.2: *Flujo para creación de diseño de objetos en base a mapa de bits*

Como se puede ver en la figura 1.2, para iniciar este proceso es necesario tener un editor de imágenes que permita crear archivos .bmp, como por ejemplo Microsoft Paint®. Con este editor se crea una imagen del tamaño y forma deseada, la cual usualmente no excede los 50px x 50px. Luego se debe pintar, con color negro, los pixeles que representen la primera capa de átomos del objeto. Estos pixeles luego serán identificados por un *script* de *Matlab*, como posiciones relativas de átomos del objeto.

En la figura 1.3 siguiente imagen se muestra un ejemplo de estos mapas, usando una imagen de 10px x 10px. Esta figura solo representa la forma y posiciones relativas de los átomos y no considera el tipo de átomo ni la estructura cristalina que ellos forman.

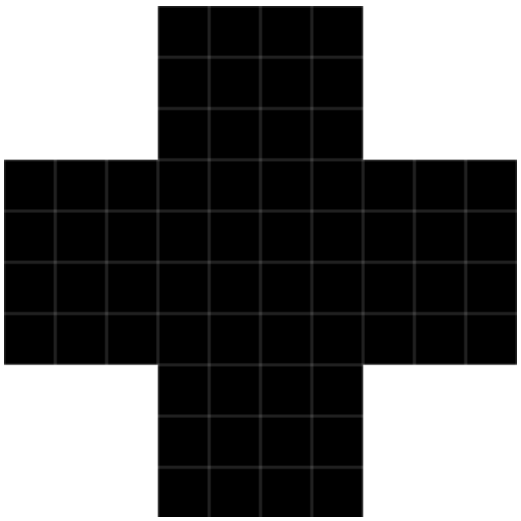

Figura 1.3: *Mapa de bits de 10px x 10px*

Para introducir esta información se debe modificar otro *script* de *Matlab*, el cual tiene múltiples versiones según el tipo de estructura cristalina que se quiera diseñar, que agrega los parámetros físicos del objeto. Algunas de las opciones que se deben configurar antes de iniciar la ejecución son:

Número de capas

Cuántas veces se repite la primera capa para formar un objeto en 3D.

Coeficiente de escalamiento

Es un coeficiente usado para dar al objeto las medidas deseadas para ejecutar la simulación.

Constante de red

Esta es la dimensión real (en nanómetros) de la arista de una celda unitaria

Todos estos parámetros deben ser modificados directamente en el código, y en

distintas partes de este, por lo que es muy fácil cometer errores, los que por supuesto invalidan cualquier simulación hecha en base a estos.

Una vez configurado el script, este debe ser ejecutado. Se lee el mapa de bits ingresado como parámetro analizando cada uno de los pixeles, creando un arreglo binario de 2 dimensiones, donde un pixel negro es representado por un 1 y un pixel blanco es representado por un 0, lo que creará la primera capa de átomos; luego, según las configuraciones ingresadas, generará el resto de átomos buscados con sus correspondientes vecinos. El resultado es un archivo con las posiciones de los átomos y sus vecinos, además de dos imágenes que representan las distintas partículas generadas en base a los archivos de entrada. Estas imágenes no son lo suficientemente claras como para poder identificar errores en el diseño esperado, por lo que la posibilidad de equivocarse y simular con una premisa inválida es algo mucho más común que lo esperado.

La figura 1.4 muestra la visualización 3D del objeto de la figura 1.3 generado con *MatLab*. Como se puede apreciar esta visualización no permite distinguir claramente la forma de cruz del objeto. Solo una vista 2D, como la de la figura 1.5, permite corroborar la forma adecuada.

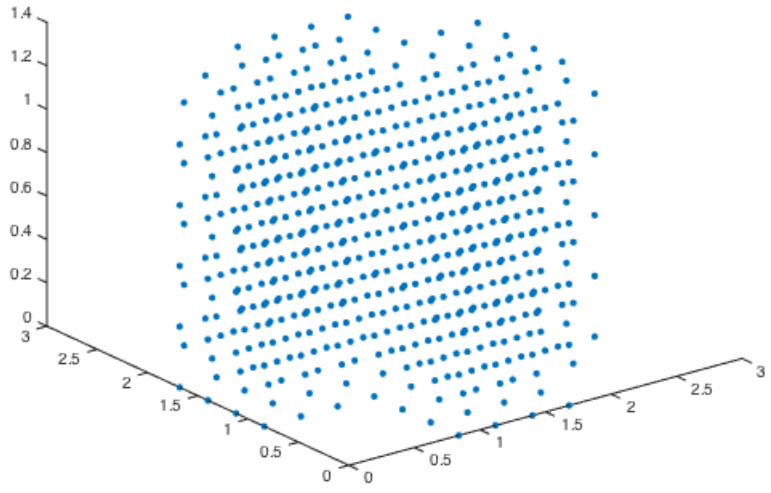

Figura 1.4: *Vista 3D de los átomos encontrados*

Una vez decidido que el objeto diseñado se usará para una simulación es necesario ejecutar un nuevo software, escrito en C, que se encarga de modificar el archivo de salida del script de *Matlab* de tal forma de que pueda ser usado como entrada para la simulación Monte Carlo. Cabe notar que este software está compilado y no se cuenta con el código fuente de este,

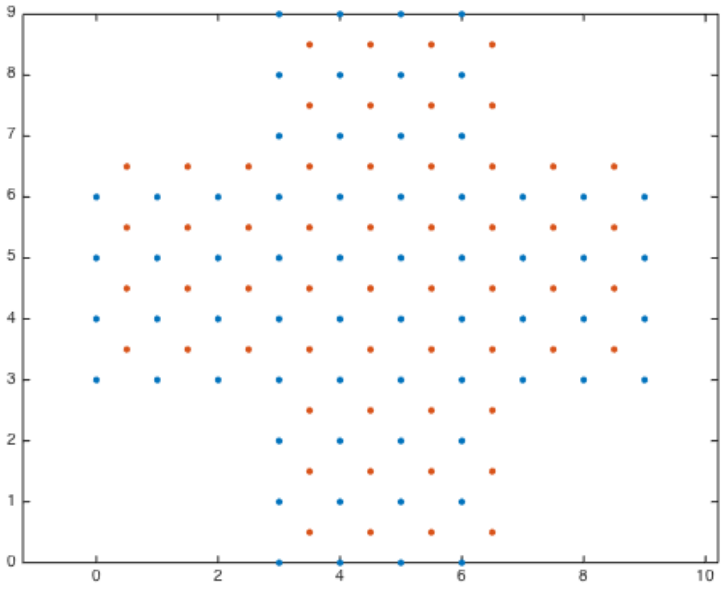

Figura 1.5: *Vista 2D de la primera capa de átomos*

de tal forma que si en algún momento se necesitara modificar el formato de la entrada de la simulación este software se debería re-escribir.

La transformación consiste en modificar cada línea del archivo, con el siguiente formato:

```
5.4700000e+02   1.4000000e-01   1.5400000e+00   1.2600000e+00   4.0000000e+00   4.8300000e+02
      4.8400000e+02   4.8700000e+02   4.8800000e+02   0.0000000e+00   0.0000000e+00
    0.0000000e+00   0.0000000e+00
```

y eliminar todas las potencias, reemplazándolas, de ser posible, con números enteros. Además es necesario agregar una línea en el inicio del archivo que contiene distintos parámetros usados por el software de simulación, como el número de partículas y la constante de escalamiento.

Un ejemplo de las primeras 5 líneas de un archivo correcto sería:

```
104    1   8   0.00245
1  -0.19799    -1.26   0   3   3   4   54  0   0   0   0   0
2  0.19799 -1.26   0   3   4   5   55  0   0   0   0   0
3  -0.39598    -1.12   0   4   1   6   7   58  0   0   0   0
4  0   -1.12   0   6   1   2   7   8   52  59  0   0
```

En la primera línea los datos necesitados son:

- Número de átomos descritos en el archivo
- 1. Este valor no se utiliza y se mantiene por razones de compatibilidad con antiguas versiones del software de simulación
- Número máximo de vecinos
- Constante de escalamiento

Desde la segunda línea en adelante el formato es:

- ID del átomo
- Componente $\hat{i}$ de la ubicación
- Componente $\hat{j}$ de la ubicación
- Componente $\hat{k}$ de la ubicación
- Cantidad de vecinos del átomo
- ID del vecino 1
- ID del vecino 2
- ...
- ID del vecino N

En caso de que un átomo tenga un número vecinos inferior al máximo posible se deben rellenar la línea con ceros, de tal forma que cada línea tenga la misma cantidad de valores.

1.3.2 Visualización de resultados

Si bien el proceso de diseño de objetos es precario, este no se compara con el de visualización y exportación de resultados, tanto en forma de imágenes como videos. Dentro de las limitaciones actuales está la imposibilidad de generar videos con colores que permitan distinguir una tercera dimensión.

Actualmente se usa la funcionalidad *Quiver plot* de *MatLab*, que permite dibujar rápidamente vectores en base a archivos de entrada con cierto formato, no obstante este proceso

requiere de tiempo para obtener representaciones que realmente reflejen lo deseado, siendo un proceso de ensayo y error que utiliza tiempo de manera innecesaria para los científicos.

1.4 PROPÓSITOS DE LA SOLUCIÓN

1. Automatizar el proceso de diseño de configuraciones atómicas usadas para la simulación, de forma de disminuir tanto el tiempo requerido para esto como la probabilidad de introducir errores humanos, los que pueden inválidar una simulación
2. Permitir el manejo interactivo durante el diseño de una configuración atómica mediante rotaciones, traslaciones y *zoom*
3. Exportar configuraciones atómicas que puedan ser simuladas inmediatamente, sin necesidad de transformar los datos
4. Generar imágenes de alta calidad para publicaciones, tanto de una configuración atómica diseñada como de un estado de la simulación en un tiempo t
5. Generar videos de cambios de estados de una simulación para presentación en conferencias

1.5 ALCANCES O LIMITACIONES DE LA SOLUCIÓN

- El software se encargará del diseño de objetos para la simulación entregando la entrada para ésta y posteriormente de la visualización de los resultados, y de la exportación de estos para publicaciones, mas no se encargará de la simulación en sí, la cual queda fuera del alcance de la solución.
- La aplicación estará disponible para sistema operativo MAC OS X.
- El diseño de objeto será por capas, es decir, se define la “vista superior” y la cantidad de veces que se repetirá hacia abajo.

1.6 OBJETIVOS Y ALCANCE DEL PROYECTO

1.6.1 Objetivo general

Diseñar e implementar una aplicación para el diseño interactivo de configuraciones atómicas 3D, basado en capas, y para la visualización dinámisa y estática de la evolución de dicha configuración durante una simulación de campo magnético.

1.6.2 Objetivos específicos

Para la consecución del objetivo general, se plantean las siguientes metas intermedias para el proyecto:

1. Definir los requerimientos funcionales, en base a la interiorización en el proceso actual de los investigadores
2. Diseñar los componentes de diseño de configuraciones atómicas y visualización de resultados de la simulación
3. Construir los componentes utilizando las herramientas de desarrollo mencionadas en el punto 1.8.2
4. Probar la aplicación

1.7 CARACTERÍSTICAS DE LA SOLUCIÓN

Como solución se propone la creación de un software que facilite el trabajo de los científicos. Esta aplicación se divide en dos funcionalidades:

1.7.1 Diseño de configuraciones atómicas

El software debe permitir la creación visual e interactica de un objeto en 3D, con ciertas características físicas como el tipo de estructura cristalina. Luego de la creación y configuración del objeto, la aplicación debe exportar un archivo de texto que sirva como entrada para el software que realiza la simulación. Este archivo tiene un formato expecífico, el cual fue definido en el punto 1.3.1.2.

1.7.2 Visualización dinámica de la evolución de las configuraciones atómicas

El software debe tomar la totalidad de archivos de salida de la simulación como entrada y debe ser capaz de mostrar visualmente el estado magnético de cada átomo en un tiempo t. También debe ser posible ver la simulación animada a través del tiempo, como un video.

La salida de la visualización serán imágenes en 2D del estado de la simulación en un tiempo t. Estas imágenes deben tener colores que permitan al lector entender el resultado a pesar de la dimensión faltante, por ejemplo, usando la proyección del vector en uno de los ejes y asignando un color según la intensidad de éste.

1.8 METODOLOGÍA Y HERRAMIENTAS UTILIZADAS

1.8.1 Metodología

Dado que se tiene conocimiento de los requerimientos mayores, pero pueden existir detalles al trabajar en un ámbito tan específico como simulaciones físicas, se decidió usar una modificación de la metodología Scrum (Beedle et al., 1999).

Scrum está pensado para trabajar en equipos con varios desarrolladores, además de los cargos de gestión, para lo cual se tienen 3 roles (Scrum Alliance, 2015):

Product Owner

Es el encargado de maximizar el valor del trabajo del equipo. Tiene un alto conocimiento del producto mediante un contacto directo con los *stakeholders* y facilita la comunicación de estos con el equipo de desarrollo. Pot lo tanto es el responsable de decidir qué se va a construir, pero no el cómo. Para este proyecto el *Product Owner* será el profesor Fernando Rannou, quién tiene contacto constante con los *stakeholders* por proyectos paralelos que se están desarrollando.

Scrum Master

Es el líder del equipo de desarrollo y debe tener un buen conocimiento de la metodología *Scrum*, el cual debe traspasar al equipo de desarrolladores. Sus 3 principales tareas son: Guíar al equipo teniendo un conocimiento tanto del producto como de las tecnologías a utilizar; mantener al equipo avanzando eliminando toda dificultad que puedan tener durante el desarrollo (ya sea interna como externa), y enseñar la metodología *Scrum* al equipo. En este caso, como el equipo de desarrollo está compuesto por sólo un desarrollador y este tiene un gran conocimiento de la metodología gracias a sus años de experiencia laboral usándola, el rol de *Scrum Master* no se usará.

Desarrollador

El desarrollador es el encargado de entregar los incrementales del producto, para lo cual se basa en la lista de tareas definidas por el *Product Owner* al inicio de un *sprint*. En este proyecto solo trabajará un desarrollador.

Otras adaptaciones hechas a la metodología fue la modificación alguna de sus ceremonias, cambiando la reunión diaria (*Daily Scrum*) por una semanal, entre el profesor y el desarrollador. Las reuniones retrospectivas al finalizar cada *sprint* se unieron con la de planificación del siguiente periodo de desarrollo. Además, como es recomendado, se tuvo una reunión con los *stakeholders* luego de cada *sprint*.

Estas modificaciones fueron necesarias para poder usar la metodología en un proyecto con solo un desarrollador y optimizando al máximo el tiempo usado en reuniones, debido al poco espacio en las agendas tanto del desarrollador como del *Product Owner*.

1.8.2 Herramientas de desarrollo

1.8.3 Modelado 3D: OpenGL

Se usa OpenGL como biblioteca de modelado 3D, por ser el estado del arte en este ámbito. Entre sus ventajas está el ser multi-plataforma, lo que eventualmente permitiría una rápida portación a otro sistema operativo, y el ser la más usada actualmente, lo que permite que tenga una amplia comunidad de usuarios que la soportan y documentan.

1.8.4 Lenguaje de programación: Python

Para el desarrollo se usará el lenguaje de programación Python con la biblioteca wxPython para Intefaz de Usuario. Esta biblioteca tiene soporte para la API OpenGL. Python, al ser un lenguaje multiplataforma, permitiría una rápida portación a otro sistema operativo en el futuro.

1.8.5 Control de versiones: GIT

Para el versionamiento del código se usará GIT, manteniendo un respaldo del repositorio con el código y la documentación en una máquina virtual con Linux ubicada en Estados Unidos.

1.9 AMBIENTE DE DESARROLLO

Para el desarrollo se usará el siguiente ambiente de desarrollo:

- Computador marca Apple, con una tarjeta gráfica que soporte OpenGL 3.2+ y sistema

operativo Mac OS X para el desarrollo.

- Una máquina virtual con Linux, ubicada en Estados Unidos, para mantener un respaldo del código y de la documentación.

CAPÍTULO 2. DOMINIO DE PROBLEMA

2.1 PROBLEMA

Investigadores del Departamento de Física de la Universidad de Santiago de Chile pertenecientes al CEDENNA se encuentran estudiando los efectos de aplicar ciertos campos magnéticos a nano-estructuras cristalinas, con propiedades físicas definidas. Los resultados de estas simulaciones pueden ser aplicados a la producción de nuevos dispositivos magnéticos o para el grabado de datos magnético de alta densidad (Vargas et al., 2011).

2.2 ESTRUCTURAS CRISTALINAS

Las nano-estructuras estudiadas son de tipo cristalinas, las que pueden ser divididas en celdas unitarias de distintos tipos según la distribución de sus átomos. Las 3 estructuras más usadas son el cubo simple (*SC* o *Simple Cubic*), cubo centrado en su cuerpo (*BCC* o *Body-centered Cubic*) y cubo centrado en sus caras (*FCC* o *Face-centered cubic*) (Hahn, 2005).

En las estructuras cristalinas la dimensión física de las aristas está dada por el parámetro llamado *Lattice Parameter* o Parámetro de Red. Como en este caso solo se trabajará con estructuras cúbicas todas las aristas tendrán la misma dimensión, por lo que nos referiremos a este parámetro como *Lattice Constant* o Constante de Red.

Para cada átomo de una estructura cristalina se puede identificar un conjunto de átomos que componen la vecindad, es decir, son los átomos que están a menor distancia de este. Dependiendo del tipo de celdas unitarias se define un algoritmo para encontrar los vecinos e identificar el número máximo de vecinos que pueden ser encontrados.

Dependiendo del tipo de estructura se puede encontrar patrones de periodicidad entre ellas, es decir, que cada cierto número de capas se repita la misma distribución de átomos.

2.2.1 Cubo simple (SC)

Esta es la estructura cristalina más simple, donde en cada vértice podemos encontrar una partícula. Luego la distancia entre dos átomos vecinos es siempre la constante de red (ver figura 2.1). Como en cada vértice de la estructura se unen 8 celdas unitarias, cada cubo tiene $1/8$ de partícula, luego cada celda tiene $1/8 \cdot 8 = 1$ átomo en total.

En el caso del cubo simple la vecindad de cada átomo está compuesta a lo más por otros 6 átomos, los que están en dirección a cada una de las aristas que componen el vértice donde se encuentra la partícula.

El cubo simple tiene una periodicidad de 1, es decir, todas las capas tienen la misma distribución de átomos.

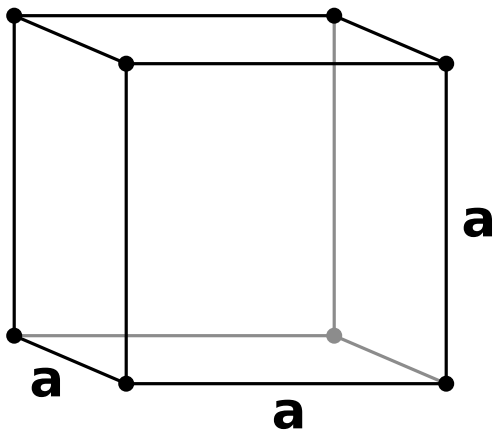

Figura 2.1: *Cubo simple, con constante de red* a

2.2.2 Cubo centrado en su cuerpo (BCC)

Una estructura cristalina de cubos centrados en su cuerpo se caracteriza porque en cada una de sus celdas unitarias se puede encontrar, además de los átomos en sus vértices, un átomo en su centro (ver figura 2.2). De esta forma cada una de sus celdas tiene $1/8$ de átomo en cada uno de sus vértices, más un átomo en su centro, resultando en $1/8 \cdot 8 + 1 = 2$ átomos en total.

A diferencia de un cubo simple, la vecindad de un átomo en BCC está compuesta

por:

- Para el caso de un átomo ubicado en un vértice, las partículas centrales de cada uno de los cubos que componen dicho vértice.
- Para el caso de un átomo ubicado en el centro de una celda unitaria, las partículas ubicadas en cada uno de los vértices de dicha celda.

En ambos casos el número máximo de partículas en una vecindad es de 8.

La periodicidad de capas de una estructura compuesta por cubos centrados en su cuerpo es de 2, es decir, cada 2 capas se repetirá la distribución de átomos.

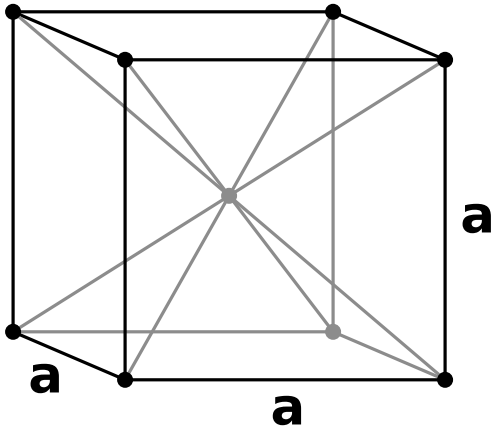

Figura 2.2: *Cubo centrado en su cuerpo, con constante de red* a

2.2.3 Cubo centrado en sus caras (FCC)

En el caso de las estructuras cristalinas compuesta por cubos centrados en sus caras se puede distinguir átomos en cada uno de sus vértices y átomos en cada una de las caras de cada cubo (ver figura 2.3). Como el átomo de una cara es compartido por 2 cubos, cada uno de ellos contiene $1/2$ de éste, y por lo tanto cada celda unitaria contiene $1/8 \cdot 8 + 1/2 \cdot 6 = 4$ átomos en total.

La vecindad de un átomo FCC está compuesta por:

- Las partículas de cada cara que esté compuesta por una de las aristas que conforman el vértice, para el caso de un átomo ubicado en ese punto

- En el caso de las partículas que se encuentran en una cara, todos los átomos ubicados en las caras formados por las aristas que componen la cara del átomo inicial (2 caras por arista), además de los átomos que están en los vértices que componen la cara inicial

De las reglas anteriores se puede inferir que el tamaño máximo de una vecindad es de 12 átomos.

Para las estructuras formadas por cubos centrados en sus caras se identifica una periodicidad de 2, es decir, al igual que en el caso de los cubos centrados en su cuerpo, cada 2 capas se repite la distribución atómica.

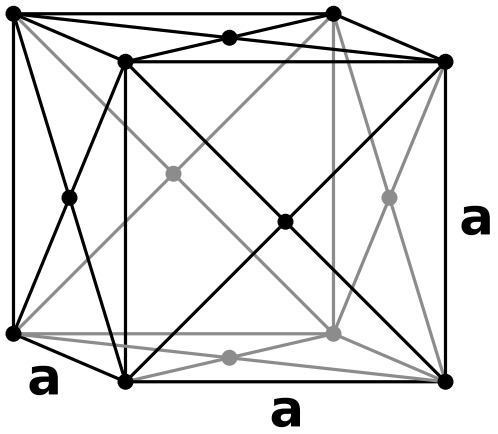

Figura 2.3: *Cubo centrado en sus caras*

2.3 ESCALAMIENTO

A pesar de la gran capacidad de procesamiento de los computadores actuales aún resulta inconveniente simular los sistemas con sus características reales, debido a la gran cantidad de átomos que deberían ser simulados. Para esto los científicos han propuesto un método de escalamiento de las configuraciones que les permita reducir significativamente el número de átomos y obtener soluciones válidas (Landeros et al., 2005). El método fue validado mediante un diagrama de fase magnético para un cilindro con ciertas características materiales y físicas, en función de su diámetro y altura. En este diagrama notaron que un diagrama modificado (o escalado) $J' = J \cdot X$, con la constante de escalamiento $X < 1$ se puede obtener modificando las dimensiones del cilindro en base a la siguiente relación:

$$Longitud' = Longitud \cdot X^{\eta} \text{ , con } \eta = 0.55$$

donde Longitud puede ser cualquiera de las 3 dimensiones, alto, ancho o profundidad.

Con esta relación de escalamiento las relaciones magnéticas entre los átomos se mantienen intactas, y las simulaciones entregan resultados válidos.

EL diseño de configuraciones escaladas en base a las dimensiones reales se realiza de la siguiente dorma. Dada la constante de red, el tipo de estructura cristalina y el número de capas deseado, se calcula la altura escalada del objeto mediante:

$$Altura' = \frac{Capas * Constante\ de\ Red}{Periodicidad\ de\ la\ estructura\ cristalina}$$

Luego, y conociendo la altura real, se calcula la constante de escalamiento con la siguiente ecuación:

$$X = \left(\frac{Altura'}{Altura}\right)^{1/\eta}$$

Finalmente, las otras dos dimensiones escaladas pueden obtenerse mediante:

$$Ancho' = X^{\eta} \cdot Ancho$$

$$Profundidad' = X^{\eta} \cdot Profundidad$$

2.4 MÉTODO MONTE CARLO

El Método Monte Carlo es un método estadístico (no determinista) propuesto por Nicholas Metropolis et al en 1949, mientras trabajaba en el desarrollo de la Bomba Atómica en el Laboratorio nacional de Los Alamos en EEUU, que permite obtener soluciones aproximadas a problemas muy difíciles de solucionar de forma matemática (determinista) debido a su magnitud o complejidad, para esto se basa en la aleatoriedad. Primero se debe determinar que condiciones se deben cumplir para que cierto caso a probar sea válido, estos pueden ser, por ejemplo, un

set de ecuaciones con múltiples parámetros o resultados de interacciones entre un cierto grupo de partículas, luego genera sets de prueba aleatorios que pueden cumplir o no las condiciones definidas, luego de una gran cantidad de iteraciones se pueden usar los resultados válidos para obtener conclusiones. De cierta forma se experimenta teóricamente las distintas combinaciones de factores que afectan un problema definido.

Dependiendo de donde se desea aplicar el método es posible que no sea necesario que los números generados para las pruebas no sean realmente aleatorios, si no que, en general, solo se necesita que sean lo suficientemente aleatorios, para esto existen diversas pruebas estadísticas con las que se puede validar, como que con una gran cantidad de elementos generados no exista un patrón claro de generación y que estos estén distribuidos de forma uniforme.

2.5 APLICACIÓN DEL MÉTODO MONTE CARLO AL PROBLEMA

Para esta investigación los científicos aplicarán el Método Monte Carlo usando las siguientes condiciones (Allende, 2008):

- Se selecciona una partícula i al azar y se calcula su energía $E_1 = E(\vec{m}_i)$
- Se modifica la dirección del momento magnético de la partícula i al azar y se calcula la nueva energía $E_2 = E(\vec{m}_i')$
- Se considera válido el cambio de $\vec{m}_i$ a $\vec{m}_i'$ si se cumple alguna de las siguientes condiciones
  - E_2 es menor que E_1
  - E_2 es mayor que E_1 y se cumple con la relación $\exp(-\beta(E_2 - E_1)) > \varepsilon$, donde $\beta = (kT)^{-}1$ siendo k la constante de Boltzmann, T la temperatura (Newman & Barkema, 1999) y ε un número aleatorio entre 0 y 1

En caso de que las condiciones no se satisfagan se considera inválido el cambio y se rechaza, es decir, se conserva $\vec{m}_i$.

Analizaremos los parámetros de simulación de uno de los casos estudiado por los científicos usando el Método Monte Carlo (Vargas et al., 2011), para esto trabajaremos con un *dot* circular, con diámetro d = 80 nm y altura h = 20 nm. Como fue explicado anteriormente resulta

prácticamente imposible analizar estos objetos en tamaño real, por lo que deben ser escalados lo suficiente para poder ejecutar los cálculos usando la tecnología disponible actualmente sin perder sus características magnéticas como el desarrollo de *vortex* magnéticos, para esto se usará un factor de escalamiento $X = 0.01 - 0.001$, esto se logra usando $\eta \approx 0.55 - 0.57$ y por supuesto escalando las dimensiones iniciales de forma $d' = dx^{\eta}$ y $h' = hx^{\eta}$.

Para elegir la nueva orientación del campo magnético se usará un generador aleatorio con una probabilidad $p = min[1, \exp(-\Delta E/k_B T')]$, donde ΔE es el cambio de energía debido a la reorientación del spin, k_B es la constante de Boltzmann y $T' = Tx$, con $T = 10K$.

Inicialmente se aplica un campo hacia el eje X de magnitud $H = 5.5kOe$ y se analizaran pasos de $\Delta H = 0.1kOe$, es decir, para completar una curva de histéresis son necesarios 110 pasos de ΔH. En cada uno de estos pasos se analizará el campo magnético del *dot*, ejecutando 3.500 pasos Monte Carlo por cada ΔH, por lo que la cantidad de pasos para esta simulación es de $110 \cdot 3500 = 385000$ de forma de completar la curva de histéresis.

En cada uno de estos pasos es necesario hacer un cálculo de energía del *dot*.

Este proceso completo se repite varias veces con distintas semillas para el generador aleatorio de números, para así validar el resultado.

2.6 CÁLCULO DE ENERGÍA

Para calcular la energía total E_{tot} de un *dot* se usa la siguiente ecuación:

$$E_{tot} = \frac{1}{2} \sum_{i \neq j} (E_{ij} - J_{ij} \hat{\mu}_i \cdot \hat{\mu}_j) + E_H$$

donde E_{ij} es la energía dipolar dada por

$$E_{ij} = [\vec{\mu}_i \cdot \hat{\mu}_j - 3(\vec{\mu}_i \cdot \hat{n}_{ij})(\vec{\mu}_j \cdot \hat{n}_{ij})]/r_{ij}^3$$

con r_{ij} la distancia entre los momentos magnéticos $\vec{\mu}_i$ y $\vec{\mu}_j$ y $\hat{n}_{ij}$ el vector unitario en la dirección que conecta los dos momentos magnéticos. $\hat{\mu}_i$ es un vector unitario en la dirección de $\vec{\mu}_i$ y $E_H = -\sum_i \vec{\mu}_i \cdot \vec{H}$ representa la energía Zeeman para un campo $\vec{H}$ aplicado hacia el eje x.

J_{ij} es la interacción de canje magnético entre i y j, es decir, como sus campos magnéticos internos se afectan entre ellos, para este caso se asumirá que $J_{ij} = 0$ para dos átomos que no son vecinos. Como en el caso de dos partículas vecinas es necesario calcular esta interacción, al momento de ejecutar la simulación es necesario saber, para cada átomo, quienes componen su vecindad.

CAPÍTULO 3. DISEÑO DE LA SOLUCIÓN

La aplicación, llamada Lattice Designer, forma parte de una estructura de capas de todos los componentes del sistema. La figura 3.1 muestr como Lattice Designer se encuentra entre los componentes de más bajo nivel, como librerías, y el componente de más alto nivel, el programa de simulación.

A continuación se describe cada uno de estos niveles.

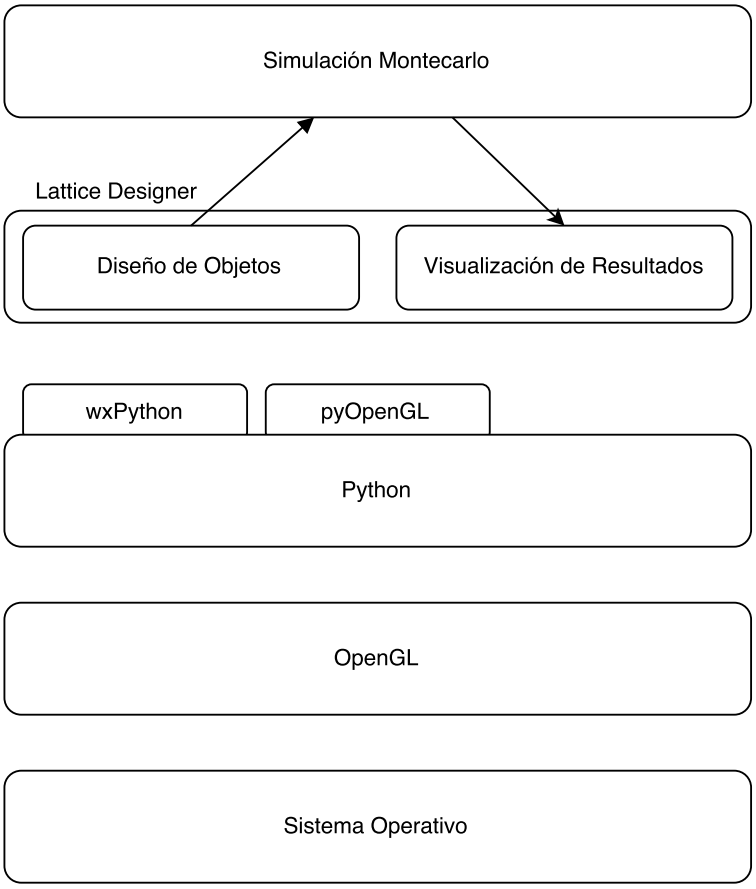

Figura 3.1: *Diagrama de la arquitectura de la solución.*

3.1 SISTEMA OPERATIVO

Si bien la aplicación fue desarrollada íntegramente en Mac OS X, y es por lo tanto el sistema operativo oficial de la presente memoria, todo el sistema fue desarrollado pensando en ser eventualmente multi-plataforma, sin que esta portación a distintos Sistemas Operativos tenga mayores complicaciones, más allá de los problemas que uno pueda encontrar por la distinta estructura de estos. Para esto tanto la elección del lenguaje de programación como de las distintas bibliotecas usadas fueron hechas teniendo el cuenta que deben poder ser ejecutados en los 3 sistemas operativos más usados, OS X, Windows y Linux.

3.2 OPENGL

Como parte importante del software es su capacidad de representar gráficamente en 3D tanto los átomos de un objeto diseñado como los resultados de las simulaciones, es necesario buscar una biblioteca de procesamiento gráfico 3D que fuera realmente multi-plataforma, tanto a nivel de sistema operativo como de tarjetas gráficas. Debido a esto se elige el estándar OpenGL, el cuál tiene implementaciones tanto para Windows, OS X y Linux, además de ser el estándar de la industria para gráficos 2D y 3D (The Khronos Group, 2015c). Esto permite tener una gran comunidad activa lo que a su vez se traduce en una gran cantidad de documentación al respecto.

La especificación del estándar OpenGL era dirigido por el consorcio independiente *OpenGL Architecture Review Board* hasta el año 2006, cuando se decidió transferir esta responsabilidad al *Khronos Group* (The Khronos Group, 2015b), un consorcio formado por distintas organizaciones, tanto empresariales como académicas, quienes manejan múltiples estándares de la industria como *OpenGL ES*, *OpenCL* y *WebGL* (The Khronos Group, 2015a).

3.3 PYTHON

Para el lenguaje de programación se barajó inicialmente la opción de C++ por las ventajas que conlleva trabajar a bajo nivel. No obstante debido a su inclinada curva de aprendizaje y complejidad he decidido usar Python, ya que es un lenguage que también cumple con las

características necesarias para este desarrollo, como el ser multi-plataforma, y tener a disposición bibliotecas de manejo gráfico como su compatibilidad con la API OpenGL, de tal forma de centrar la complejidad del proyecto en las representaciones 3D.

Python es un lenguaje de programación de alto nivel, interpretado y orientado a objetos, desarrollado durante el año 1990 por Guido van Rossum, aunque actualmente es de código abierto y mantenido por su comunidad, la que es liderada por la *Python Software Foundation*, quienes además resguardan los derechos de este.

Algunas de las características más conocidas de Python es su fácil sintaxis y su gran biblioteca estándar, la que cubre áreas como Protocolos de comunicación, Ingeniería de software e Interfaces de Sistema Operativo (Python Software Foundation, 2015).

Durante el desarrollo se la aplicación se usaron distintas bibliotecas, siendo las dos más importantes wxPython y pyOpenGL.

3.3.1 wxPython

wxPython es una biblioteca de python que permite usar de forma nativa wxWidgets, un set de herramientas de código abierto, escrita en C++ y inicialmente escrito por Julian Smart y actualmente mantenida por la comunidad que permiten crear interfaces gráficas en distintas plataformas como Windows, OS X, iOS, Linux, entre otros. En este trabajo wxPython maneja todas las interacciones de los usuarios, además de casi todas las interfaces gráficas con excepción de los *canvas* de OpenGL.

3.3.2 pyOpenGL

pyOpenGL es una biblioteca que permite la programación en OpenGL directamente desde python. Es multiplataforma y totalmente compatible con la biblioteca de interfaz gráfica usada (wxPython). Uno de los sub-paquetes de pyOpenGL usado en esta memoria es GLUT, la que permite crear fácilmente ciertos objetos en OpenGL, como esferas o conos, de tal forma de no tener que programar estos a partir de triángulos, como sería si se usara OpenGL puro.

3.4 LATTICE DESIGNER

Lattice Designer es el software desarrollado en esta memoria, el cual permite a científicos generar configuraciones atómicas sobre los cuales se simula la aplicación de campos electromagnéticos y la visualización de resultados generados por la simulación. Aunque esta herramienta está pensada para ser usada por investigadores del Departamento de Física de la Universidad de Santiago de Chile, la aplicación que ejecuta la simulación es usada por diversas organizaciones académicas, por lo que este software pretende ser un aporte a la comunidad científica en general.

Durante el desarrollo se puso especial énfasis en la experiencia de usuario, de tal forma que cualquier científico que tenga acceso a esta aplicación sea capaz de usarlo sin la necesidad de un entrenamiento; por este motivo el software tiene todos sus textos en inglés.

Como lo muestra la figura 3.1, la aplicación se compone de dos módulos claramente definidos: el diseño de configuraciones atómicas y la visualización de resultados de la simulación Monte Carlo.

3.4.1 Diseño de configuraciones atómicas

Para el desarrollo de esta funcionalidad se usó como base el sistema de diseño de objetos actual, basado en un mapa de bits, explicado en la sección 1.3.1.2 de esta memoria, pero con la finalidad de que el proceso completo sea ejecutado en la misma aplicación disminuyendo con esto la probabilidad de errores que se puedan producir.

Dentro de las entradas de esta funcionalidad están tanto el mapa de bits, mediante una grilla binaria, como los distintos parámetros físicos asociados al objeto, como el tipo de estructura cristalina, la constante de red y el número de capas, entre otros (ver figura 3.2). Además se le ofrece al usuario mucha información dinámica, es decir, que va cambiando en base a los parámetros de entrada, como por ejemplo la constante de escalamiento.

Una de las características del diseño es la posibilidad de crear imágenes predefinidas en base a dimensiones entregadas por el usuario. De esta forma se pueden crear cuadriláteros definiendo su ancho y alto o circunferencias con un radio específico. Una vez ingresados los parámetros solo es necesario hacer click en la posición de la grilla donde se quiere

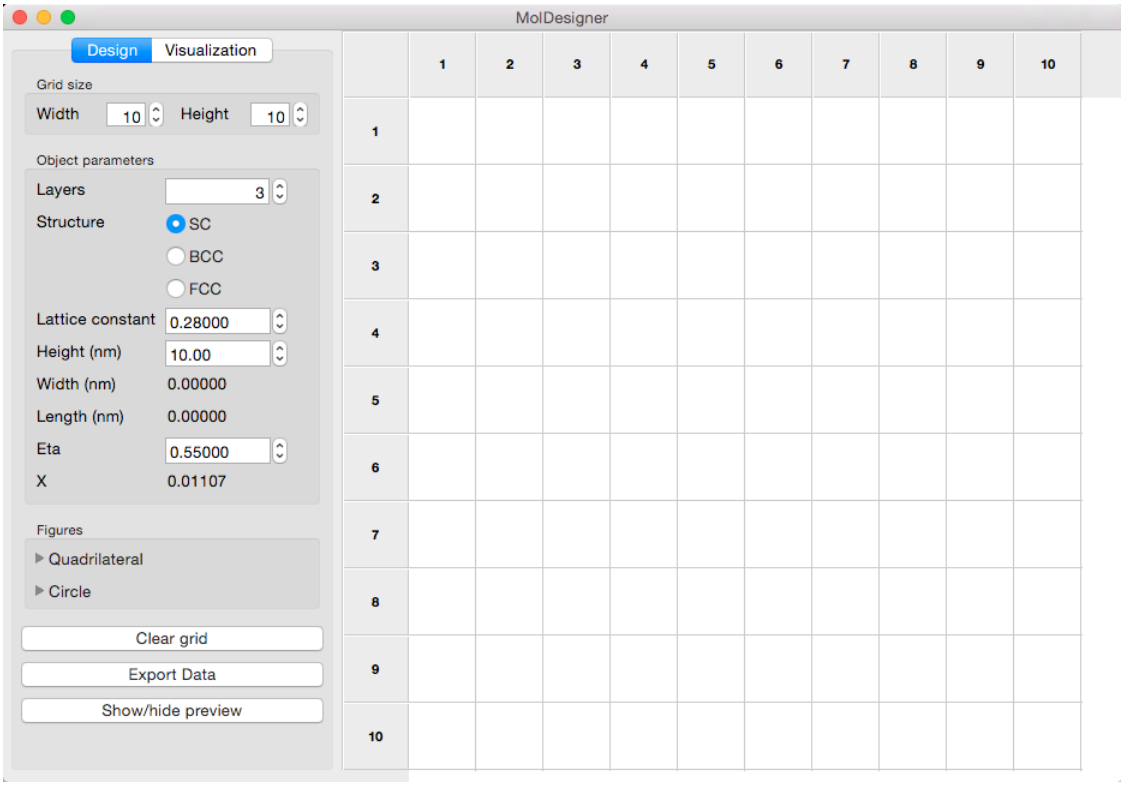

Figura 3.2: *Vista de diseño de configuraciones atómicas del software*

dibujar.

Como muestra la figura 3.3, el software permite también al usuario ver de forma inmediata la visualización 3D de la configuración atómica que se está creando con colores identificando sus distintos tipos de partículas, permitiendo así comprobar de manera rápida y fácil si efectivamente el diseño es el deseado.

Una vez confirmado que el objeto diseñado es el deseado es posible exportar directamente las imágenes de la configuración atómica y los ejes coordenados para referencia; así como también el archivo que describe la configuración atómica, el que sirve como entrada para el software de simulación. A diferencia del proceso actual, donde el archivo exportado debe ser modificado para poder ser usado, en este caso los datos exportados puedes ser usados inmediatamente para ejecutar una simulación.

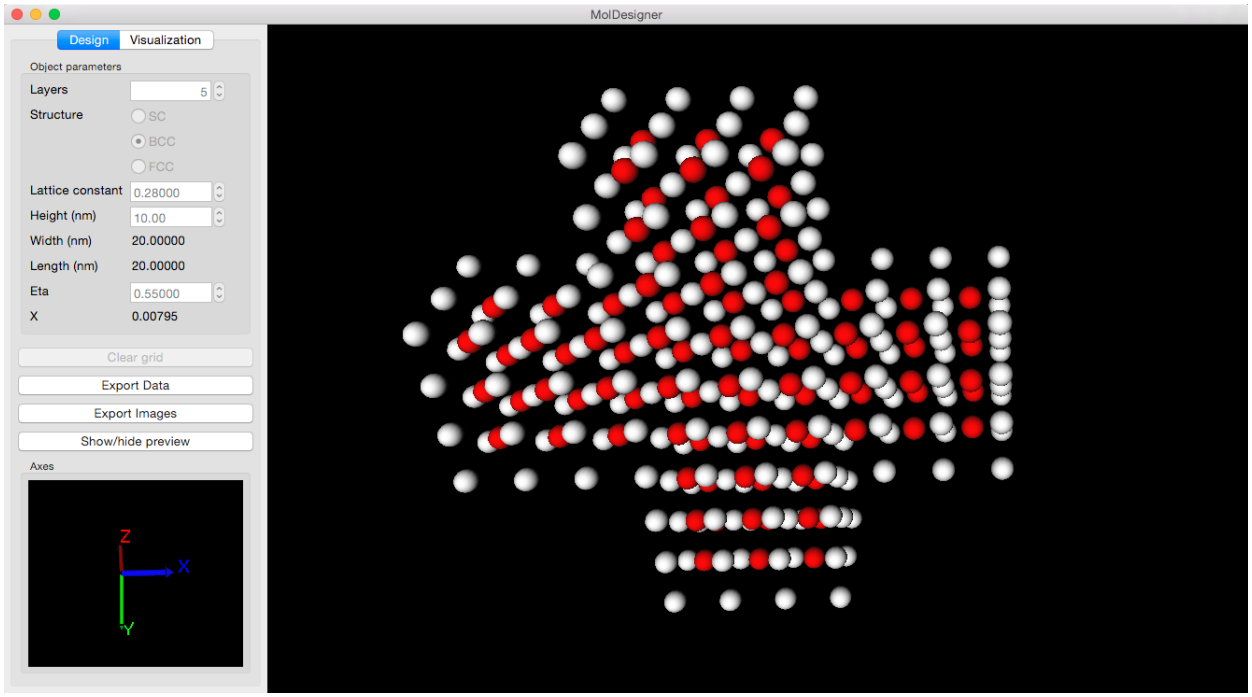

Figura 3.3: *Previsualización de un objeto diseñado*

3.4.2 Visualización de resultados

El segundo módulo se basa específicamente en los requerimientos de los *stakeholders*, sin tener un proceso actual como base, ya que esta es la gran debilidad que tienen actualmente.

Dentro de las funcionalidades de este módulo se encuentra la visualización y exportación del estado de la simulación en un tiempo *t*, incluyendo los vectores de campo magnético, la curva de histéresis del campo magnético y los ejes coordenados para referencia. Estos son elementos que los científicos necesitan en sus publicaciones.

Las figuras 3.5, 3.6 y 3.7 son las tres imágenes que se pueden exportar del estado de la simulación representado en la figura 3.4.

Otras características disponibles es la asignación de colores para cada vector según uno de sus componentes cartesianos, usando una escala de colores de azul a rojo. En el caso de la figura 3.5 el máximo valor de $\hat{i}$ será rojo y el mínimo será azul. Esto permite identificar rápidamente una tercera dimensión en una imagen 2D y también los vortex que se generan.

También es posible ver la variación de los vectores a través del tiempo como video

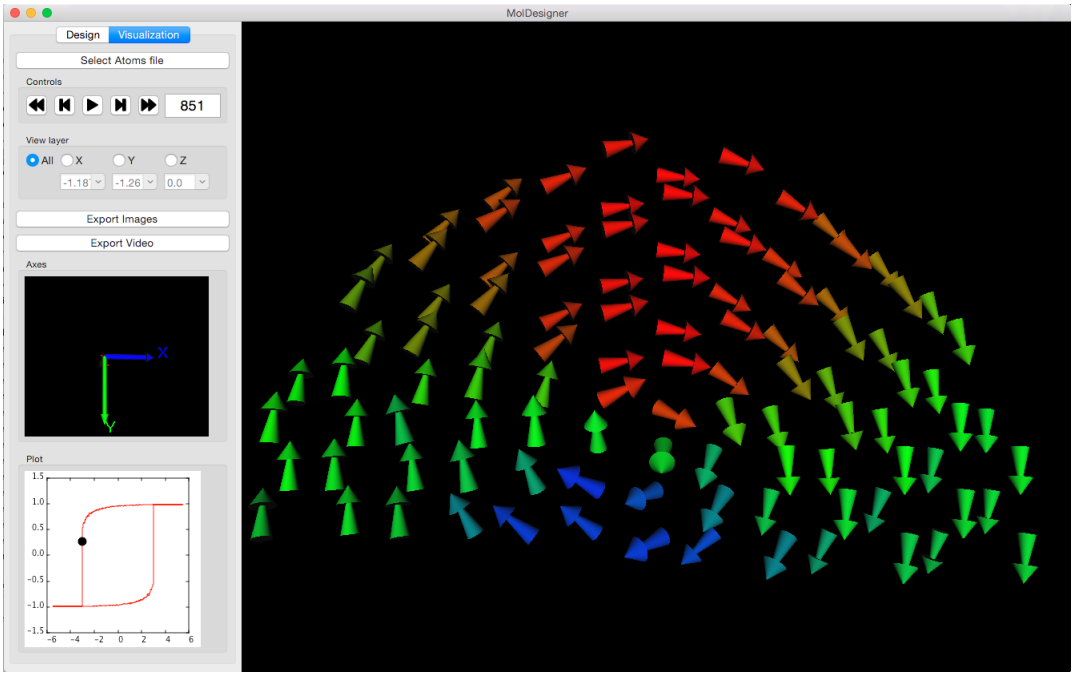

Figura 3.4: *Pestaña de visualización de resultados en* $t = 851$

y luego exportarlo de tal forma que este pueda ser usado fácilmente en conferencias donde se divulguen los resultados.

Cabe notar que todas las características anteriores pueden ser ejecutadas mientras se visualiza solo una de las múltiples capas en la que se puede dividir el objeto, en base a cualquier eje coordenado, como se puede ver en las figuras 3.8 y 3.9.

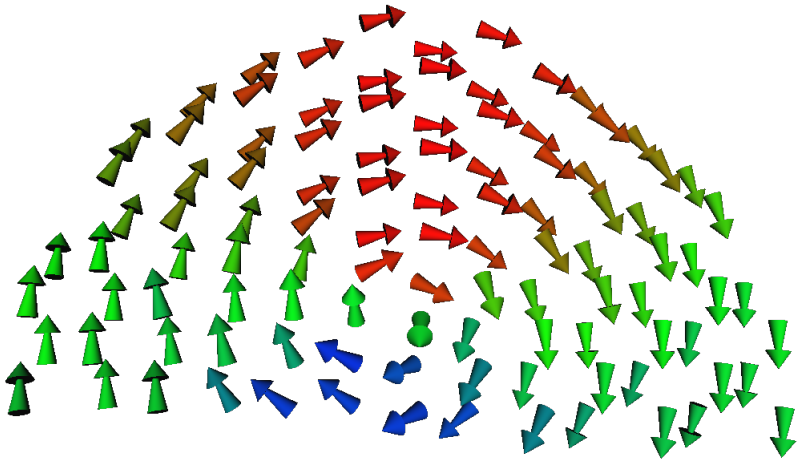

Figura 3.5: *Imagen de vectores exportada para publicación*

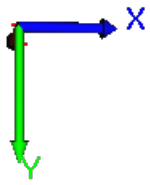

Figura 3.6: *Imagen de ejes de referencia exportada para publicación*

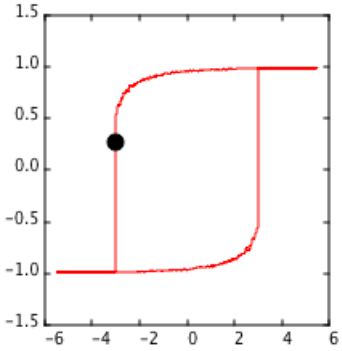

Figura 3.7: *Imagen de curva de histéresis exportada para publicación*

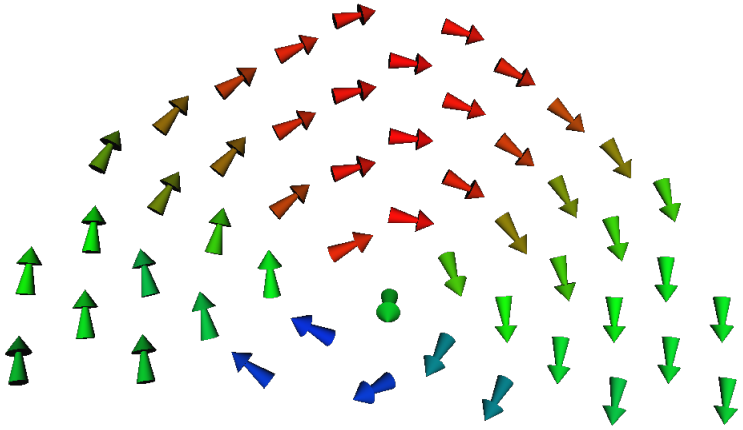

Figura 3.8: *Estado de la simulación para t = 851 solo para la capa Z = 0*

Figura 3.9: *Estado de la simulación para t = 851 solo para la capa X = -0.395*

CAPÍTULO 4. IMPLEMENTACIÓN DE LA SOLUCIÓN

4.1 CLASES

Para el desarrollo de la aplicación se usó el lenguaje Python, dividiendo la aplicación en clases cuya relación puede ser vista en la figura 4.1. Estas clases se categorizan en: Clases visuales; clases 3D y clases de cubos.

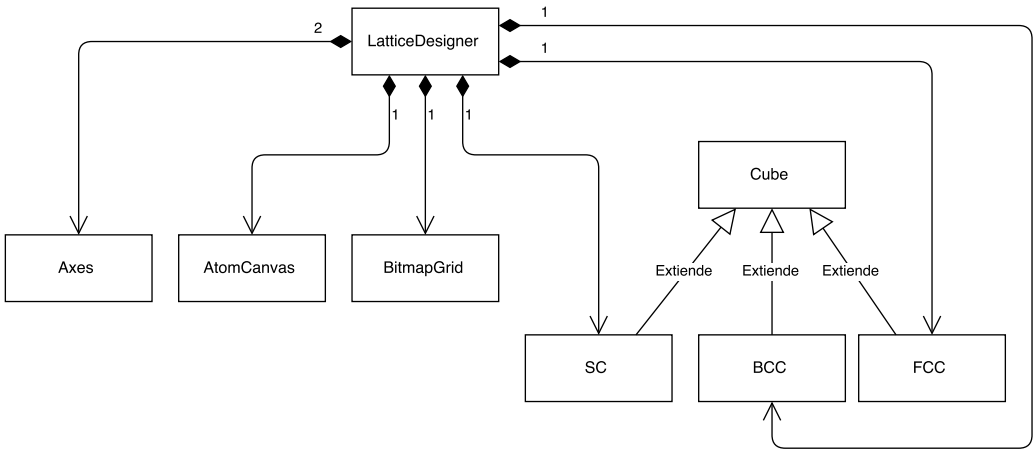

Figura 4.1: *Diagrama de clases de la aplicación Lattice Designer.*

4.1.1 Clases visuales

Estas clases están relacionadas directamente con la interfaz gráfica de usuario. Permiten crear las distintas ventanas, objetos e interacciones de los usuarios con estos.

4.1.1.1 LatticeDesigner

Esta es la clase principal de la aplicación, ya que es la encargada de iniciar el programa, cargando todas las dependencias necesarias para su ejecución, pero principalmente

implementa la lógica de la parte visual del *software*, es decir, de la creación y distribución en pantalla de los distintos elementos visuales, como ventanas, botones, campos de texto, etc., además de la interacción del usuario con estos. Cada reacción a una acción ejecutada sobre estos elementos es orquestada por esta clase.

4.1.1.2 BitmapGrid

La clase BitmapGrid es la encargada de manejar la grilla con la cuales los científicos diseñarán las configuraciones atómicas que se simularán. Para esto se basa en la biblioteca *wx.grid* de *wxPython*, una implementación de una planilla de cálculos tipo *excel*, la cual es modificada para no poder ser editada y que las distintas acciones del *mouse* sobre esta (click, seleccionar fila o columna, seleccionar un rango de celdas, etc.) generen cambios en el color de fondo de cada celda, pudiendo este ser negro o blanco, transformando esta planilla de cálculos en un mapa de bits binario.

Entre las características de este mapa de bits se encuentra la capacidad de crear figuras predefinidas rápidamente. Por ejemplo, se puede dibujar un cuadrilátero indicando el ancho, el alto y seleccionando la esquina superior derecha de esta figura. También es posible crear un círculo indicando el radio que tendrá este y su centro. Estas dos figuras predefinidas pueden combinarse para crear configuraciones más complejas, como un elipsoide, que son de mucha utilidad para los científicos (ver figura 4.2).

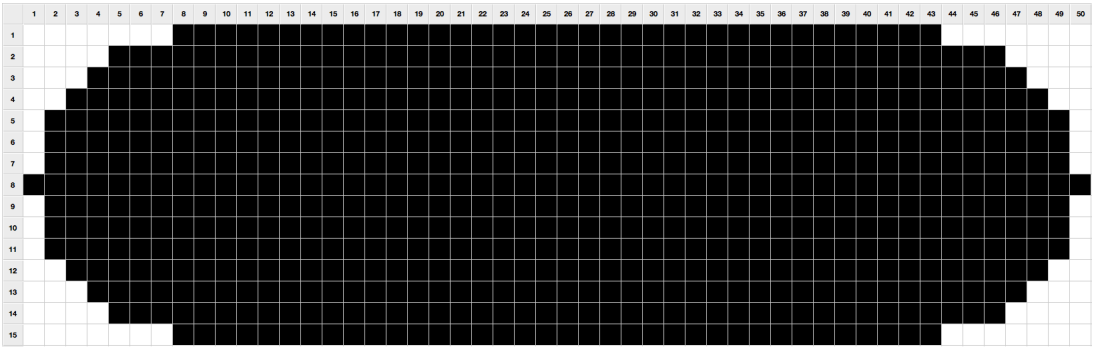

Figura 4.2: *Mapa de bits binario con 2 figuras pre-diseñadas.*

4.1.2 Clases 3D

Estas clases son las encargadas de manejar los distintos *canvas* que se usan en el *software*, tanto para la visualización del diseño y del resultado de la simulación, como para ayudas referenciales para los científicos.

4.1.2.1 AtomCanvas

La clase AtomCanvas es la más importante con respecto a la visualización 3D, ya que es la encargada de mostrar en pantalla tanto el diseño de los objetos sobre los cuales se correrá la simulación como los resultados de estas usando OpenGL. En el desarrollo de esta clase se puso énfasis en la optimización, pudiendo mostrar sin mayores demoras más de 20.000 átomos. Para tener una idea un sistema promedio escalado analizado por los científicos usa 3.000 átomos (TODO: CONFIRMAR).

Otra de las tareas de esta clase es manejar las distintas interacciones del usuario, tanto con el teclado como con el *mouse* con las representaciones en 3D, como rotaciones, movimientos y *zoom*.

Para la visualización del diseño se usan esferas de distintos colores, representando cada uno de los distintos tipos de átomos que pueden haber según la estructura cúbica elegida. En las figuras 4.3, 4.4 y 4.5 se representan una estructura de 5x5 átomos, con 3 capas, siendo diferente solo la estructura cristalina elegida.

En el caso de la visualización de resultados se usan flechas de colores como representación del momento magnético. Para asignar un color a una flecha se parte de la premisa que en tiempo t=0 el campo magnético externo está siendo aplicado hacia un eje, y por lo tanto inicialmente todos los vectores tendrán la misma magnitud, sentido y dirección, paralelos al campo externo, como se puede notar en la figura 4.6. Además se sabe que en ese momento su magnitud es máxima. Si inicialmente todos los vectores son paralelos al eje A, se usa la componente â de cada vector para definir el color. Si la componente es 0 entonces será de color verde; si la magnitud es máxima en sentido contrario a los vectores iniciales el color será azul; si la magnitud es máxima en el mismo sentido de los vectores iniciales el color será rojo. Como se puede inferir la escala va desde azul a rojo, siendo este último color el inicial para todos los vectores. En la

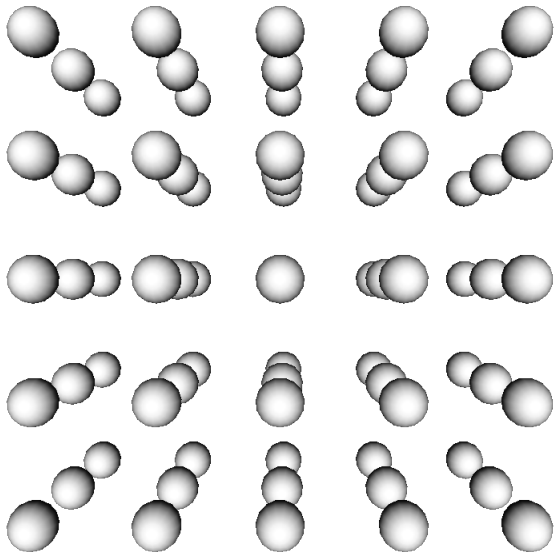

Figura 4.3: *En un SC todos los átomos son blancos.*

figura 4.7 se puede ver los distintos colores según la componente $\hat{i}$ de los vectores, dado que el campo externo inicial tiene sentido $[1, 0, 0]$.

4.1.2.2 Axes

La clase Axes es la encargada de mostrar los ejes coordenados de los distintos canvas, tanto del de diseño de objetos como el de visualización de resultados, usando Open GL. Cada eje se representa con su propio color, usando azul, rojo y verde para los ejes X, Y y Z respectivamente, y una etiqueta con el mismo color, de forma de hacerlo fácil de visualizar para el usuario. Debido al diseño del *software*, donde las distintas funciones del programa (diseño y visualización) se seleccionan mediante pestañas, esta clase debe ser instanciada 2 veces, ya que no es posible usar la misma instancia en ambas secciones. Estas se comunican directamente con AtomCanvas para obtener los distintos parámetros de rotación de forma que los ejes sean coherentes a la imagen mostrada.

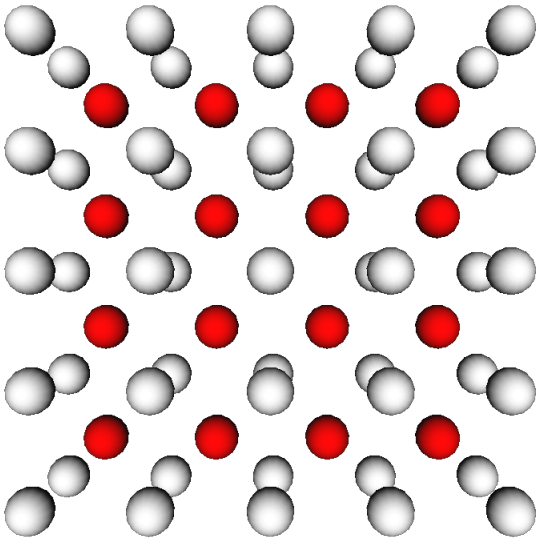

Figura 4.4: *En un BCC los átomos centrales son rojos.*

4.1.3 Clases de cubos

Las clases de cubos son 3 clases (*SC*, *BCC* y *FCC*) que heredan de un mismo padre, *Cube*. Estas crean los átomos de un objeto que se está diseñando. Las 3 clases hermanas tienen solo 2 métodos: *calculate* y *find_neighborhood*. El primero se encarga de identificar todos los átomos que conforman la configuración atómica, dados los parámetros físicos como la estructura de la primera capa. El segundo define cuales son los parámetros para llamar al método *find_neighborhood* de la súper clase, el que busca todos los vecinos inmediatos para cada átomo. Estos parametros son necesarios para exportar el archivo que luego servirá de entrada para la simulación.

4.1.3.1 Cube

TBD

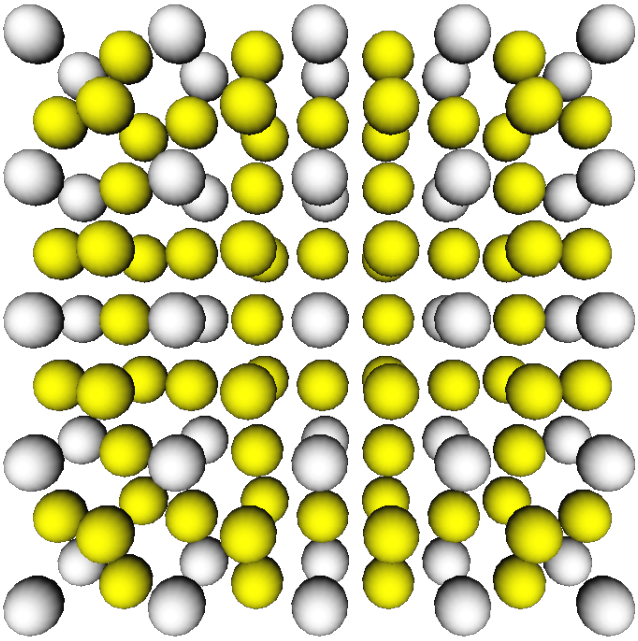

Figura 4.5: *En un FCC los átomos de las caras son amarillos.*

4.1.3.2 SC

SC es la clase que maneja los cubos simples (*Simple Cubic* o *SC*), estas estructuras cúbicas se caracterizan por tener un átomo en cada uno de sus vértices, por lo que la identificación de sus átomos se reduce a simplemente repetir la capa superior tantas veces como sea indicado en la entrada de propiedades físicas. Para encontrar el vecindario es necesario buscar todos los átomos que estén en las siguientes posiciones relativas [-1,0,0], [1,0,0], [0,-1,0], [0,1,0], [0,0,-1] y [0,0,1], por lo que el tamaño máximo de su vecindad es de 6 átomos.

4.1.3.3 BCC

BCC es la clase que maneja los cubos centrados en el cuerpo (*Body Centered Cubic* o *BCC*), que son las estructuras cúbicas que además de tener un átomo en cada vértice de los cubos tienen uno en el centro de cada uno de estos. Esto implica que en la identificación de átomos se debe trabajar con una capa intermedia que contiene los centros de cada cubo. por ejemplo, para una estructura de 5 capas quedaría así:

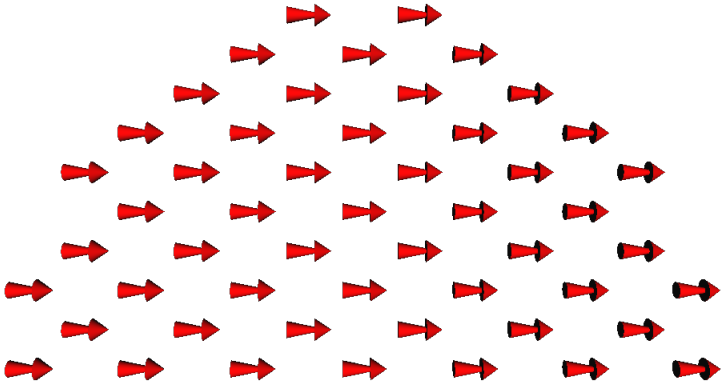

Figura 4.6: *Estado inicial de la visualización, con todos los vectores rojos paralelos al campo magnético externo inicial.*

#	Capa
1	Primaria
2	Intermedia
3	Primaria
4	Intermedia
5	Primaria

La regla para agregar un átomo central es que debe tener un cubo de átomos a su alrededor, y en caso que el cubo no esté completo simplemente se usan las capas primarias, como se puede notar en las figuras 4.9 y 4.10.

En el caso de los BCC los átomos que conforman la vecindad siempre estarán en las posiciones relativas $[\pm 0.5, \pm 0.5, \pm 0.5]$, es decir, cada átomo puede tener una vecindad compuesta por hasta 8 átomos.

4.1.3.4 FCC

FCC es la clase que maneja los cubos centrados en las caras (*Face Centered Cubic* o *FCC*), los cuales se caracterizan por tener un átomo extra por cara además de uno en cada

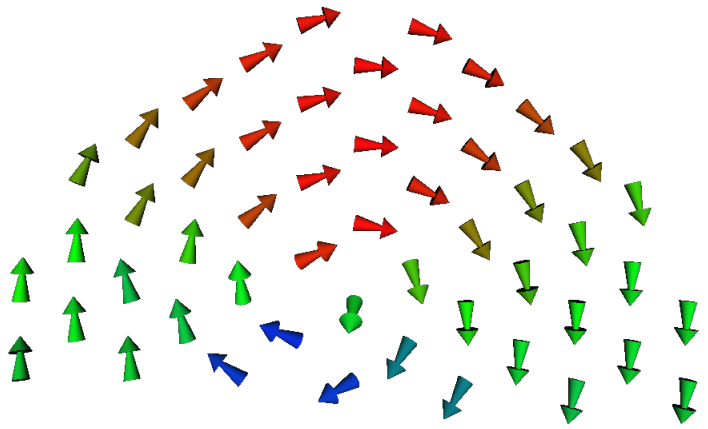

Figura 4.7: *Vectores de distintos colores según la componente* $\hat{i}$.

uno de sus vértices, por lo que además de tener que crear una capa intermedia es necesario modificar la capa primaria, es decir, la que crea el usuario usando el mapa de bits. La regla para agregar estos átomos en las caras es que ellos estén en la diagonal creada por otros 2 átomos, en cualquier dirección. En la figura 4.11 se ve como en una estructura cúbica de 1x2, como en una de sus caras se forma una diagonal entre 2 átomos y por lo tanto es necesario agregar un átomo extra en una capa intermedia.

La vecindad de estas estructuras cúbicas está dada por la posición relativa dada por $[\alpha, \beta, \gamma]$, donde:

$$\begin{cases} (\alpha = \pm 0.5; \beta = \pm 0.5; \gamma = 0) \\ (\alpha = \pm 0.5; \beta = 0; \gamma = \pm 0.5) \\ (\alpha = 0; \beta = \pm 0.5; \gamma = \pm 0.5) \end{cases}$$

Lo que resulta en 12 posiciones posibles, siendo este el número máximo de átomos en una vecindad.

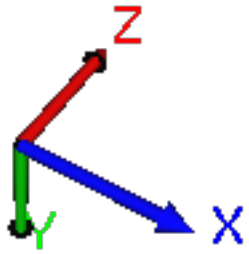

Figura 4.8: *Representación visual de los ejes coordenados.*

4.2 DISEÑO DE CONFIGURACIONES ATÓMICAS

El diagrama presentado en la figura 4.12 representa el proceso de diseño de configuraciones atómicas. Para comenzar es necesario definir los parámetros físicos de la configuración, como el tipo de estructura cristalina o la constante de red, y la primera capa de la configuración, en base a un mapa binario de bits. Estas dos acciones no deben ser ejecutadas en un orden en específico, pero ambas son necesarias para el resto de las acciones.

Una vez configurada la configuración a diseñar es posible previsualizarla en 3D, para esto el usuario debe ejecutar una acción (presionar un botón), la que es manejada por un objeto de *LatticeDesigner*, para luego decidir si se debe crear una instancia de *SC*; *BCC* o *FCC*, dependiendo del tipo de estructura cristalina elegida. En la instancia de cubo correspondiente se llama al método *calculate*, el que identifica todos los átomos que pertenecen a la configuración atómica que se está diseñando. Esta a su vez invoca al método *find_neighborhood* de la súper clase *Cube* con los parámetros correspondientes al tipo de estructura cristalina, para que identifique la vecindad de cada átomo. Luego se pasan los datos de la configuración atómica a la instancia de la clase *AtomCanvas*, la que crea una visualización en 3D de dicha configuración. Para finalizar, *LatticeDesigner* oculta el mapa de bits binario y muestra el *canvas* 3D con la representación de la configuración.

Dentro de la previsualización de la configuración atómica es posible interactuar con esta, haciendo *zoom* o rotándola. Como es necesario que los ejes coordenados representados

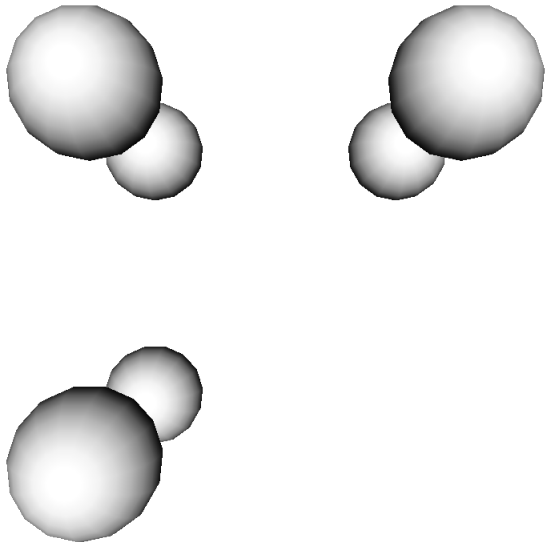

Figura 4.9: *Cubo BCC incompleto, sin átomo central*

por una instancia de la clase *Axes* concuerden con la visualización de la instancia de *AtomCanvas*, cada vez que esta última se actualiza es necesario forzar la actualización de los ejes, a través de la referencia en la instancia de *LatticeDesigner*.

Cuando el usuario desea exportar las imágenes de la configuración atómica diseñada, debe invocar el método *export* presente en las clases *Axes* y *AtomCanvas*, pasando como parámetro el nombre de archivo que debe tener el archivo de salida para cada uno de ellos. Este método funciona prácticamente de la misma manera para ambas clases: obtiene el estado actual de los *canvas* 3D, respetando el acercamiento y las rotaciones, y crea un archivo de imagen con el nombre indicado.

Para la exportación del archivo que define la configuración atómica, y que sirve como entrada para la aplicación que efectúa la simulación Monte Carlo, se identifican los átomos pertenecientes a la configuración atómica diseñada invocando al método *calculate* en la clase de cubo correspondiente (*SC*, *BCC* o *FCC*). Una vez identificados todos los átomos se identifica la vecindad para cada uno de ellos, invocando al método *find_neighborhood* de la súper clase *Cube*. Luego la instancia de *LatticeDesigner* guarda todos los datos un archivo con el formato que exige la aplicación que ejecuta la simulación Monte Carlo.

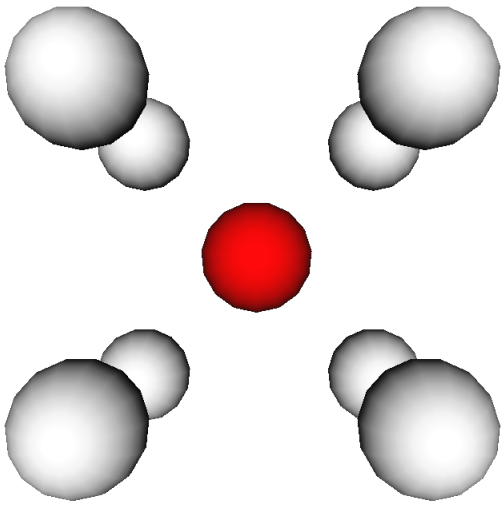

Figura 4.10: *Cubo BCC completo, con átomo central*

4.3 VISUALIZACIÓN DE RESULTADOS DE SIMULACIÓN MONTE CARLO

En la figura 4.13 se puede ver el diagrama de secuencia del proceso de visualización de resultados de la simulación Monte Carlo. La aplicación que efectúa la simulación genera como resultados un archivo que describe todos los átomos que conforman la configuración atómica. Además por cada ΔH se genera un archivo con la descripción del vector de momento magnético (TO DO: confirmar nombre del vector) para cada átomo. Para tener una idea, una simulación puede tener más de 2.200 ΔH. Para visualizar los resultados de la simulación es necesario que todos los archivos de salida de la aplicación que la ejecuta se encuentren en el mismo directorio. Además los archivos que describen los momentos magnéticos deben mantener el mismo nombre con que fueron creados.

Para comenzar la visualización es necesario seleccionar el archivo que describe la configuración atómica. Luego la instancia de *LatticeDesigner* valida que los archivos que describen los momentos magnéticos se encuentren en el mismo directorio y valida que todos los archivos sean válidos. Una vez verificado esto se da la opción al usuario de cargar esta información en la memoria para visualizar los resultados. Este proceso se llama “Carga de archivos de entrada”.

La carga de archivos de entrada consiste en leer cada uno de los archivos que genera la aplicación de simulación Monte Carlo y crear las estructuras de datos necesarias para su visualización. Luego se pasan estos datos a la instancia de *AtomCanvas* para que genere la

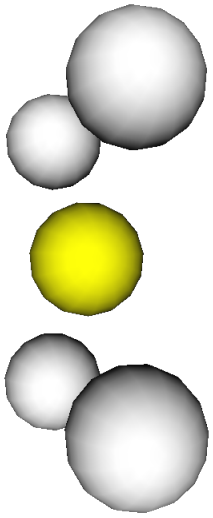

Figura 4.11: *Estructura cúbica FCC, con átomo en una de sus caras*

visualización del 3D de los vectores de momento magnético para $t = t_{min}$. Para finalizar se dibuja la curva de histéresis de la simulación, con un marcador que indica que punto de la curva se está representando (ver figura 3.7).

Dentro del *canvas* 3D es posible la interacción del usuario, tanto de *zoom* como de rotaciones. Estas interacciones son las mismas que en la funcionalidad de "Diseño de configuraciones atómicas" y están explicadas en el punto 4.2.

Dentro de las funcionalidades de la visualización de resultados está el exportar imágenes para publicaciones. Para esto es necesario invocar al método *export* de las instancias de *Axes* y *AtomCanvas*. Esto está explicado en el punto 4.2. Además es necesario exportar una imagen de la curva de histéresis, indicando el estado actual. Esto se ejecuta directamente en la instancia de *LatticeDesigner*.

Otra de las funcionalidades de la visualización de resultados es el exportar un video con el cambio de estado de los vectores de momento magnético a través de toda la simulación, para esto se ejecuta un ciclo para todo t que cumpla con $t_{min} \leq t \leq t_{max}$. Para cada t se actualiza la visualización del resultado en el *canvas* 3D, se obtiene la imagen que representa el estado de los momentos magnéticos y se agrega como un *frame* al video. Una vez obtenidos todos los estados se guarda el archivo de video en la memoria principal.

4.4 IDENTIFICACIÓN DE ÁTOMOS DE UNA CONFIGURACIÓN

Para poder generar un archivo que sirva como entrada para la aplicación de la simulación Monte Carlo es necesario determinar todos los átomos que corresponden a la configuración atómica que es diseñaba por un científico. Este proceso, a pesar de tener similitudes, es distinto para cada tipo de estructura cristalina.

Independiente del tipo de estructura cristalina se deben crear tres diccionarios al momento de identificar los átomos que componen la configuración atómica. Estos son: diccionario de átomos; diccionario inverso de átomos y diccionario de tipos de átomos.

Diccionario de átomos

Este diccionario tiene como llave el ID del átomo y como valor la posicion $[x, y, z]$ del mismo. Se usa para crear la previsualización de la configuración atómica y para crear el archivo que describe la configuración y que es usado para la simulación Monte Carlo.

Diccionario inverso de átomos

Este diccionario tiene como llave las posiciones x, y y z de un átomo, separadas por un guión bajo (_), y como valor el ID del átomo. Por ejemplo para el átomo con ID 15 en la posición $[1, 2, 3]$ la entrada en el diccionario sería: $[1\_2\_3] \Rightarrow 15$. Este diccionario se usa para buscar la vecindad de cada átomo, ya que se conoce la posición relativa para cada átomo según el tipo de estructura cristalina (ver punto 4.5).

Diccionario de tipo de átomos

Este diccionario se usa para seleccionar el color de un átomo en la visualización en 3D de una configuración atómica diseñada. Dependiendo del tipo de estructura cristalina los átomos se pueden categorizar de la siguiente manera:

Átomos de vértice Son los átomos que están en un vértice de una celda unitaria. Están presentes en todos los tipos de estructura cristalina usados en esta memoria. En la previsualización de una configuración atómica se representan con el color blanco.

Átomos centrales Son los átomos que están en el centro de una celda unitaria. Este tipo de átomos se encuentra en las configuraciones atómicas con estructura cristalina BCC. En la previsualización se representan con el color rojo.

Átomos de cara Son los átomos que están en una de las caras de una celda unitaria. Este tipo de átomos se encuentra en las configuraciones atómicas con estructura cristalina FCC. En la previsualización se representan con el color amarillo.

A continuación se explican los algoritmos usados para cada uno de estos tipos.

4.4.1 Cubo simple (SC)

Para el caso de cubo simple la capa principal debe ser simplemente replicada n veces, donde n es la cantidad de capas que tiene la configuración (ver punto 2.2.1). El estado del mapa de bits binario se guarda en una matriz bidimensional, donde cada casilla marcada se representa con un 1. Se recorre esta matriz buscando todas las casillas marcadas, de tal forma de obtener la capa primaria que será replicada. En un arreglo unidimensional se guarda una tupla de tipo (x, y) para cada casilla marcada. Esto se puede ver en el algoritmo 4.1.

Algoritmo 4.1: Obtener capa primaria para estructuras cristalinas

```
for i = 1 to Ancho grilla do
    for j = 1 to Alto grilla do
        if Grilla[i][j] == 1 then
            Agregar a CapaPrimaria = (i, j)
        end if
    end for
end for
```

Luego necesitamos agregar la componente z para cada capa de la configuración atómica, obteniendo la posición final de cada átomo. Para esto se ejecuta un ciclo n veces, donde n es la cantidad de capas que tiene la configuración. Finalmente, cuando se tiene la posición de cada átomo, se agrega a cada uno de los 3 diccionarios necesarios (Ver algoritmo 4.2).

Algoritmo 4.2: Identificar átomos para estructura cristalina de tipo SC

```
ID atomo ← 1
for z = 1 to Numero de capas do
    for cada posición (x, y) en CapaPrimaria do
        atomos[ID atomo] ← (x, y, z)
        atomos_por_posicion[x_y_z] ← ID atomo
        tipo_de_atomo[ID atomo] ← Vertice
        ID atomo ← ID atomo + 1
    end for
end for
```

Cabe notar que para una configuración atómica con estructura cristalina de tipo SC todos los átomos serán de tipo “Vértice”.

4.4.2 Cubo centrado en el cuerpo (BCC)

Para las estructuras cristalinas de tipo BCC es necesario hacer algunas modificaciones al algoritmo usado para el cubo simple. En este caso también debemos obtener la capa primaria usando el algoritmo 4.1. Además debemos obtener la capa intermedia, que contendrá todos los átomos centrales del cubo. Tal como está explicado en el punto 2.2.2 es necesario que existan los 4 átomos de vértice en una celda unitaria para agregar el átomo central. Para esto se usará el algoritmo 4.3.

Algoritmo 4.3: Obtener capa intermedia para estructura cristalina de tipo BCC

1: **for** $i = 1$ **to** $Ancho\ grilla$ **do**
2: **for** $j = 1$ **to** $Alto\ grilla$ **do**
3: **if** $Grilla[i][j] == 1$ y $Grilla[i+1][j] == 1$ y $Grilla[i][j+1] == 1$ y $Grilla[i+1][j+1] == 1$ **then**
4: Agregar a $CapaIntermedia$ = $(i + 0.5, j + 0.5)$
5: **end if**
6: **end for**
7: **end for**

Una vez teniendo la capa primaria y la capa intermedia para la configuración atómica se deben identificar todos los átomos de dicha configuración. Para saber cuantas capas de cada tipo se crearan es necesario dividir el número total de capas en 2. En caso de ser que el número de capas sea impar se usa la aproximación al entero mayor ($ceil()$) para la cantidad de capas primarias y la aproximacion al entero menor ($floor()$) para la cantidad de capas intermedias. Esto se muestra en el algoritmo 4.4.

4.4.3 Cubo centrado en sus caras (FCC)

Para los cubos centrados en sus caras es necesario crear 3 tipos de capas:

Algoritmo 4.4: Identificar átomos para estructura cristalina de tipo BCC

1: $ID\ atomo \leftarrow 1$
2: $capas\_primarias = ceil(Numero\ de\ capas)$
3: $capas\_intermedias = floor(Numero\ de\ capas)$
4: **for** $z = 1$ **to** $capas\_primarias$ **do**
5: **for** cada posición (x, y) en $CapaPrimaria$ **do**
6: $atomos[ID\ atomo] \leftarrow (x, y, z)$
7: $atomos\_por\_posicion[x\_y\_z] \leftarrow ID\ atomo$
8: $tipo\_de\_atomo[ID\ atomo] \leftarrow Vertice$
9: $ID\ atomo \leftarrow ID\ atomo + 1$
10: **end for**
11: **end for**
12: **for** $z = 1$ **to** $capas\_intermedias$ **do**
13: **for** cada posición (x, y) en $CapaIntermedia$ **do**
14: $zReal \leftarrow z + 0.5$
15: $atomos[ID\ atomo] \leftarrow (x, y, zReal)$
16: $atomos\_por\_posicion[x\_y\_zReal] \leftarrow ID\ atomo$
17: $tipo\_de\_atomo[ID\ atomo] \leftarrow Central$
18: $ID\ atomo \leftarrow ID\ atomo + 1$
19: **end for**
20: **end for**

Capa primaria Esta capa define los átomos en los vértices de las capas primarias. Es la misma capa que se usa para las estructuras cristalinas SC y BCC. Para obtener esta capa se usa el algoritmo 4.1.

Capa de cara primaria Esta capa define los átomos de cara que están a la misma altura que los de la Capa Primaria.

Capa de cara intermedia Esta capa define los átomos de cara que están entre 2 capas primarias.

Tal como se describe en el punto 2.2.3, un átomo de cara se crea cuando este está sobre una diagonal formada por 2 átomos de vértice. Esto se obtiene usando el algoritmo 4.5. Como se puede notar, en algunos casos este algoritmo puede definir un átomo de cara más de una vez, por lo que luego de ser ejecutado es necesario buscar todas las tuplas repetidas para cada una de las matrices.

Algoritmo 4.5: Obtener capas de cara para estructura cristalina de tipo FCC

1: **for** $i = 1$ **to** $Ancho\ grilla$ **do**
2: **for** $j = 1$ **to** $Alto\ grilla$ **do**
3: **if** $Grilla[i+1][j+1] == 1$ **then**
4: Agregar a $CapaCaraPrimaria$ = $(i + 0.5, j + 0.5)$
5: **end if**
6: **if** $Grilla[i+1][j-1] == 1$ **then**
7: Agregar a $CapaCaraPrimaria$ = $(ii0.5, j - 0.5)$
8: **end if**
9: **if** $Grilla[i][j+1] == 1$ **then**
10: Agregar a $CapaCaraIntermedia$ = $(i, j + 0.5)$
11: **end if**
12: **if** $Grilla[i+1][j] == 1$ **then**
13: Agregar a $CapaCaraIntermedia$ = $(i + 0.5, j)$
14: **end if**
15: **end for**
16: **end for**

Una vez que se tienen las 3 capas necesarias se usa el algoritmo 4.6 para identificar los átomos de la configuración cristalina. Este algoritmo es muy similar al usado para identificar los átomos de unan estructura cristalina de tipo BCC (ver algoritmo 4.4). Ambos difieren en que al definir los átomos de las capas primarias es necesario tomar en cuenta tanto los átomos de la matriz $CapaPrimaria$ como los de la matriz $CapaCaraPrimaria$ para las estructuras cristalinas de tipo FCC.

4.5 IDENTIFICACIÓN DE VECINDAD PARA UN ÁTOMO

Cuando se diseña una configuración atómica es necesario identificar los vecinos más cercanos para cada átomo. Debido a que el algoritmo de búsqueda es similar para los tres tipos de estructuras cristalinas, este se efectúa en la clase *Cube*, padre de las clases *SC*, *BCC* y *FCC*.

Para identificar la vecindad de cada átomo se usa el diccionario inverso de átomos creado en la etapa de identificación de átomos (ver punto 4.4.1). Según el tipo de estructura cristalina se pueden identificar las distintas posiciones relativas de los vecinos en un átomo.

Por ejemplo, para un átomo en una estructura cristalina de tipo SC se sabe que los vecinos se encuentran en las posiciones relativas $[-1,0,0]$, $[1,0,0]$, $[0,-1,0]$, $[0,1,0]$, $[0,0,-1]$ y $[0,0,1]$ (más información en los puntos 2.2.1 y 4.1.3.2). Por lo tanto para un átomo en la posición $[1,2,3]$ sus vecinos pueden estar en las siguientes ubicaciones:

- $[1,2,3]+[-1,0,0]=[0,2,3]$
- $[1,2,3]+[1,0,0]=[2,2,3]$
- $[1,2,3]+[0,-1,0]=[1,-1,3]$
- $[1,2,3]+[0,1,0]=[1,3,3]$
- $[1,2,3]+[0,0,-1]=[1,2,2]$
- $[1,2,3]+[0,0,1]=[1,2,4]$

Luego se deben buscar, en el diccionario inverso de átomos, las siguientes llaves: “0_2_3”, “2_2_3”, “1_-1_3”, “1_3_3”, “1_2_2” y “1_2_4”.

Este proceso se puede ver en el algoritmo 4.7.

Desde el método *find_neighborhood* en las clases *SC*, *BCC* y *FCC* se invoca al método de la súper clase *Cubes* con las siguientes posiciones relativas por tipo de estructura cristalina:

Cubo simple $[-1,0,0]$, $[1,0,0]$, $[0,-1,0]$, $[0,1,0]$, $[0,0,-1]$ y $[0,0,1]$.

Cubo centrado en su cuerpo $[0.5,0.5,0.5]$, $[0.5,-0.5,0.5]$, $[-0.5,0.5,0.5]$, $[-0.5,-0.5,0.5]$, $[0.5,0.5,-0.5]$, $[0.5,-0.5,-0.5]$, $[-0.5,0.5,-0.5]$ y $[-0.5,-0.5,-0.5]$.

Cubo centrado en sus caras $[0,0.5,0.5]$, $[0,0.5,-0.5]$, $[0,-0.5,0.5]$, $[0,-0.5,-0.5]$, $[0.5,0,0.5]$, $[0.5,0,-0.5]$, $[-0.5,0,0.5]$, $[-0.5,0,-0.5]$, $[0.5,0.5,0]$, $[0.5,-0.5,0]$, $[-0.5,0.5,0]$ y $[-0.5,-0.5,0]$.

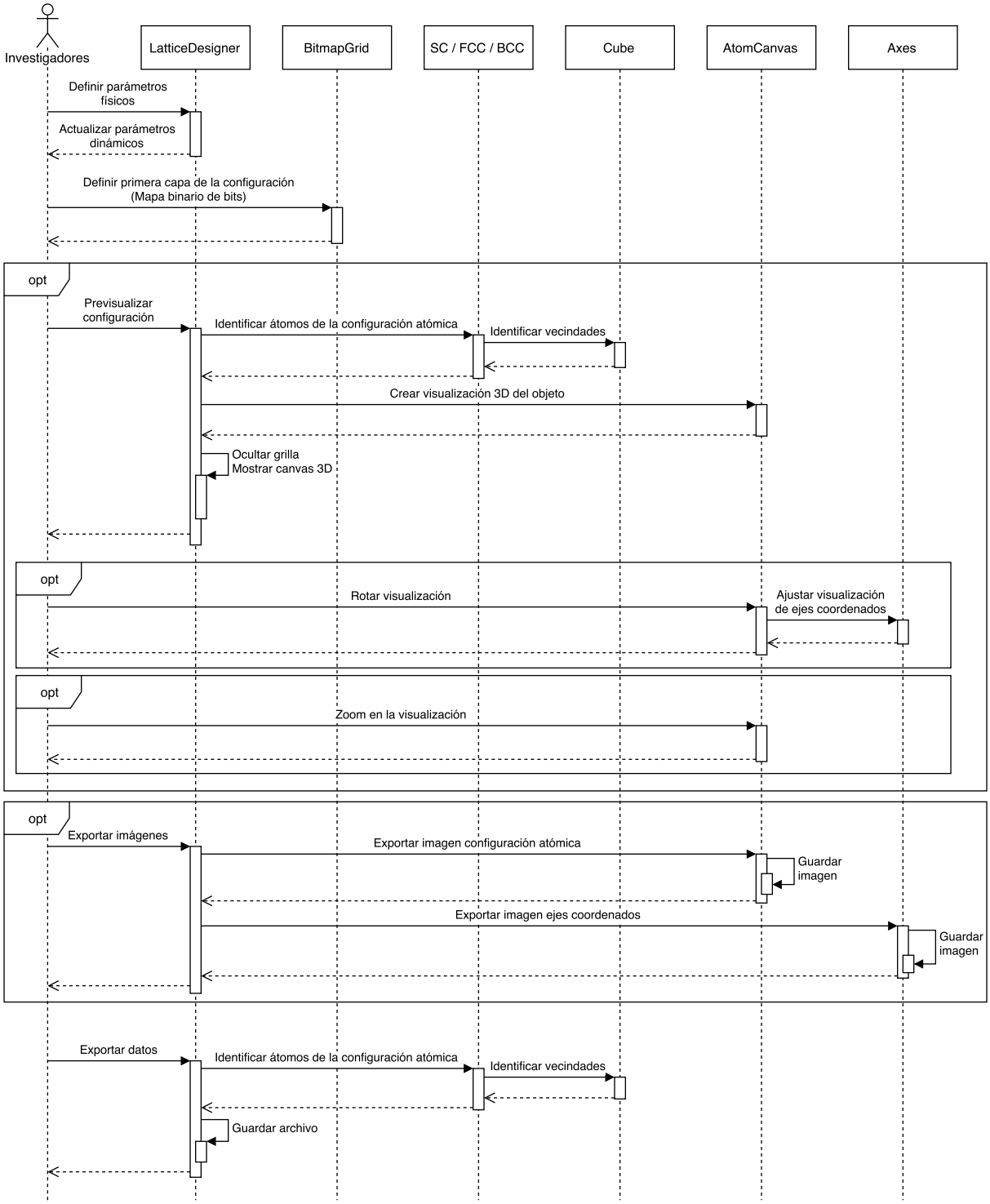

Figura 4.12: *Diagrama de secuencia para el diseño de una configuración atómica*

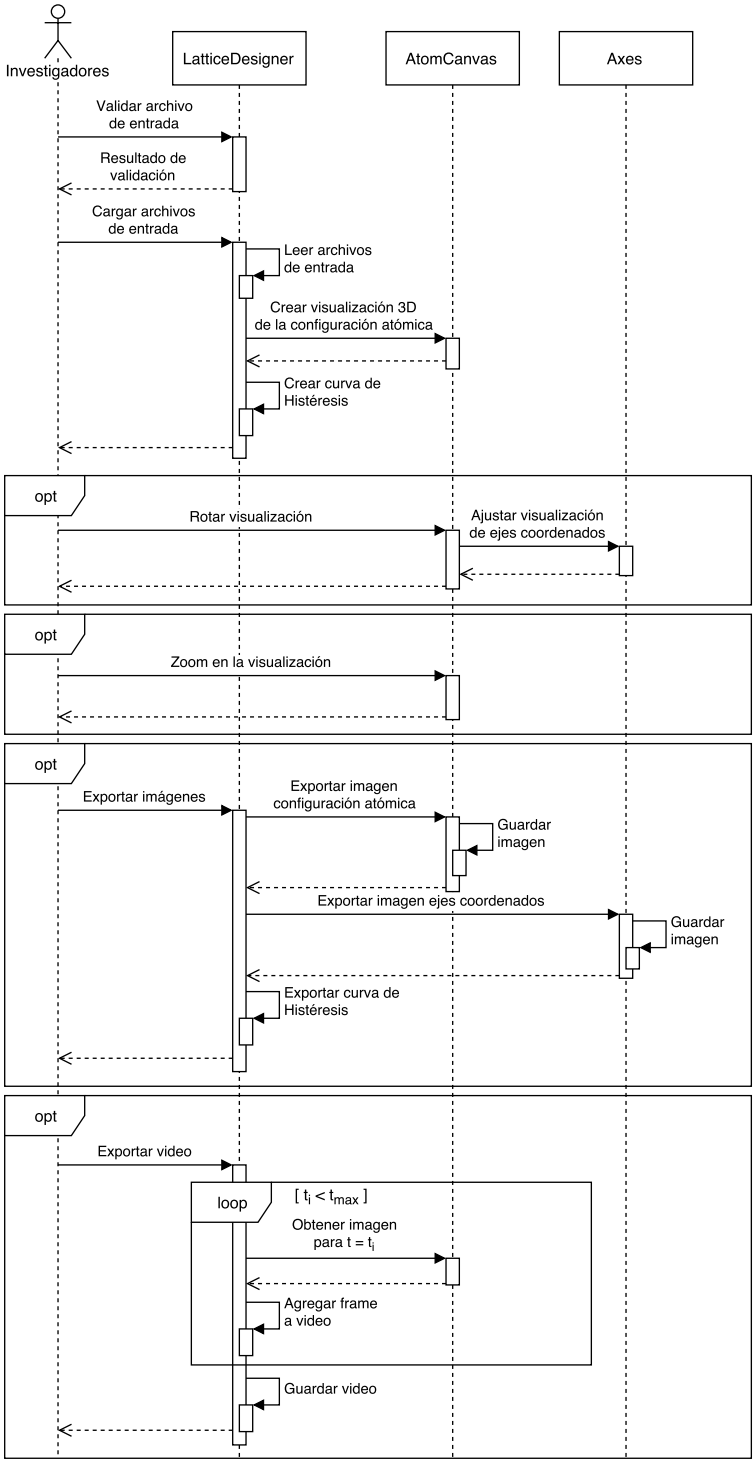

Figura 4.13: *Diagrama de secuencia para la visualización de los resultados de una simulación Monte Carlo*

Algoritmo 4.6: Identificar átomos para estructura cristalina de tipo FCC

1: $ID\ atomo \leftarrow 1$
2: $capas\_primarias = ceil(Numero\ de\ capas)$
3: $capas\_intermedias = floor(Numero\ de\ capas)$
4: **for** $z = 1$ a $capas\_primarias$ **do**
5: **for** cada posición (x, y) en $CapaPrimaria$ **do**
6: $atomos[ID\ atomo] \leftarrow (x, y, z)$
7: $atomos\_por\_posicion[x\_y\_z] \leftarrow ID\ atomo$
8: $tipo\_de\_atomo[ID\ atomo] \leftarrow Vertice$
9: $ID\ atomo \leftarrow ID\ atomo + 1$
10: **end for**
11: **for** cada posición (x, y) en $CapaCaraPrimaria$ **do**
12: $atomos[ID\ atomo] \leftarrow (x, y, z)$
13: $atomos\_por\_posicion[x\_y\_z] \leftarrow ID\ atomo$
14: $tipo\_de\_atomo[ID\ atomo] \leftarrow Vertice$
15: $ID\ atomo \leftarrow ID\ atomo + 1$
16: **end for**
17: **end for**
18: **for** $z = 1$ a $capas\_intermedias$ **do**
19: **for** cada posición (x, y) en $CapaCaraIntermedia$ **do**
20: $zReal \leftarrow z + 0.5$
21: $atomos[ID\ atomo] \leftarrow (x, y, zReal)$
22: $atomos\_por\_posicion[x\_y\_zReal] \leftarrow ID\ atomo$
23: $tipo\_de\_atomo[ID\ atomo] \leftarrow Central$
24: $ID\ atomo \leftarrow ID\ atomo + 1$
25: **end for**
26: **end for**

Algoritmo 4.7: Obtener los vecinos de átomos en base a posiciones relativas

1: $vecinos = \{\ \}$ //Diccionario vacío
2: **for all** $ID\ atomo, posicionAtomo$ en $Atomos$ **do**
3: $vecinosAtomo = [\ ]$ //Matriz vacía
4: **for all** $posicionVecino$ en $posicionesVecinos$ **do**
5: $posicionDondeBuscar \leftarrow posicionAtomo + posicionVecino$
6: **if** existe llave $posicionDondeBuscar$ en $diccionarioInverso$ **then**
7: $ID\ Vecino \leftarrow diccionarioInverso[posicionDondeBuscar]$
8: Agregar a $vecinosAtomo$ = $IDVecino$
9: **end if**
10: **end for**
11: $vecinos[ID\ atomo] \leftarrow vecinos$
12: **end for**

BIBLIOGRAFÍA

Allende, S. (2008). *Propiedades magnéticas de los nanohilos de niquel*. Ph.D. thesis, Universidad de Santiago de Chile.

Beedle, M., Devos, M., Sharon, Y., Schwaber, K., & Sutherland, J. (1999). Scrum - a pattern language for software development. *Pattern Languages of Program Design*, *4*, 637–651.

Hahn, T. (Ed.) (2005). *International Tables For Crystallography*, vol. A, chap. 2.1.3, (pp. 14–16). Springer.

Landeros, P., Escrig, J., Altbir, D., Laroze, D., d'Albuquerque e Castro, J., & Vargas, P. (2005). Scaling relations for magnetic nanoparticles. *Phys. Rev. B*, *71*, 094435.
URL http://link.aps.org/doi/10.1103/PhysRevB.71.094435

Newman, M. E. J., & Barkema, G. T. (1999). *Monte Carlo methods in statistical physics*. Oxford: Clarendon Press.

Python Software Foundation (2015). General python faq.
URL https://docs.python.org/3/faq/general.html#what-is-python-good-for

Scrum Alliance (2015). Core scrum.
URL https://www.scrumalliance.org/why-scrum/core-scrum-values-roles

The Khronos Group (2015a). About the khronos group.
URL https://www.khronos.org/about/

The Khronos Group (2015b). About the opengl arb 'architecture review board'.
URL https://www.opengl.org/archives/about/arb/

The Khronos Group (2015c). Opengl overview.
URL https://www.opengl.org/about/#1

Vargas, N. M., Allende, S., Leighton, B., Escrig, J., Mejía-López, J., Altbir, D., & Schuller, I. K. (2011). Asymmetric magnetic dots: A way to control magnetic properties. *Journal of Applied Physics*, *109*(7), –.
URL http://scitation.aip.org/content/aip/journal/jap/109/7/10.1063/1.3561483

APÉNDICE A. MANUAL DE USUARIO

A.1 REQUERIMIENTOS PREVIOS

A.1.1 Requerimientos para instalación

blablabla

Tabla A.1: Tabla que dice nada

Animal	Description	Price ($)
Gnat	per gram	13.65
	each	0.01
Gnu	stuffed	92.50
Emu	stuffed	33.33
Armadillo	frozen	8.99