Diseño y desarrollo de un visualizador de moléculas en formato PDB para Android

Juan González Salinas.
Máster universitario en Bioinformática y Bioestadística.
Trabajo final de Máster.

Brian Jiménez García.
2 de enero de 2018.

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Diseño y desarrollo de un visualizador de moléculas en formato PDB para Android*
Nombre del autor:	*Juan González Salinas*
Nombre del consultor/a:	*Brian Jiménez García*
Nombre del PRA:	*Brian Jiménez García*
Fecha de entrega (mm/aaaa):	01/2018
Titulación:	Máster universitario en Bioinformática y Bioestadística
Área del Trabajo Final:	*M0.177 - TFM-Estadística y Bioinformática*
Idioma del trabajo:	*Castellano*
Palabras clave	*Gráficos moleculares, Renderizado 3D, Android*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

En este proyecto se lleva a cabo el diseño y la creación de una aplicación para el sistema operativo móvil *Android* (desarrollado con *XML*, *Java* y *C/C++*) con la cual se pueden visualizar estructuras tridimensionales de moléculas como proteínas y ácidos nucleicos en el formato de *Protein Data Bank* (*PDB*). El trabajo también incluye una interfaz gráfica de usuario, adaptado a la tecnología táctil que proporcionan los dispositivos móviles inteligentes actuales basados en el sistema operativo *Android,* que permite la interacción con el usuario.

Abstract (in English, 250 words or less):

The project performs the design and creation of an application for the Android mobile operating system (developed with XML, Java and C/C++) with which is possible to visualize three-dimensional structures of molecules like proteins and nucleic acids in the Protein Data Bank (PDB) format. This work also includes a graphical user interface, adapted to the touch technology provided by nowadays smartphone devices that Android offers, with which to allow interaction with the user.

Índice

Lista de figuras

1. Introducción.

1.1. Contexto y justificación del Trabajo.

Actualmente, en la biología y todas sus ramas en las que se trabajan y se realizan estudios con moléculas y sus estructuras compuestas o derivadas, se emplean representaciones de éstas, tanto para visualización y comprensión de su aspecto y funcionamiento como para las actividades y procesos que se realizan con estas, como son alineamientos, predicciones e interacciones. Para profundizar en investigaciones pertinentes o incluso en enseñanza, es importante agilizar el proceso y entorno de trabajo con la descripción de estos componentes microscópicos, ya sea mediante imágenes o definiciones. Es aquí donde la representación toma importancia, siendo necesarias las herramientas para visualización de estructuras moleculares.

Por medio de la portabilidad, conectividad, versatilidad, multitarea, interactividad y ergonomía que brindan, a día de hoy, los teléfonos inteligentes y otros dispositivos similares como las tabletas, se puede mejorar la eficiencia en el entorno de trabajo y agilizar algunos procesos. Gracias a estas ventajas, los dispositivos móviles pueden cubrir algunas insuficiencias o inconvenientes de los ordenadores habituales o incluso llegar a sustituirlos completamente en los trabajos e investigaciones que se realicen.

La idea general que pretende el proyecto es proporcionar una herramienta que asista en el trabajo con moléculas, ofreciendo una interfaz de usuario definida para representación gráfica tridimensional de las mismas. Con este trabajo se intenta agilizar el proceso y mejorar la eficiencia en el lugar de estudio o práctica con moléculas. En definitiva, y más específicamente, se implementa una aplicación móvil para el sistema operativo *Android* con el que visualizar tridimensionalmente moléculas en el formato *PDB*. El trabajo de la aplicación consiste en analizar los ficheros de moléculas, representarlos y ofrecer una interfaz al usuario para que pueda interaccionar.

Este proyecto también se diseña teniendo en mente futuras mejoras y expansiones, siendo flexible en el desarrollo. Teniendo esto en cuenta, el código programado podría ser incorporado en otros proyectos, fuera incluso del sistema operativo Android, o bien se podrían incorporar más funcionalidades que ayuden al trabajo con moléculas.

En cuanto al mercado, si bien *Android* está saturado con una gran oferta de programas, la mayoría son de ámbito ajeno a la ciencia, así que hay un espacio en este sector en el cual aprovechar sus ventajas y lanzar el proyecto en donde otras plataformas y proyectos no han llegado

todavía. También, al ser *PDB* un formato globalizado, hacer uso de éste ofrece portabilidad a los usuarios con otros entornos y herramientas.

1.2. Objetivos del Trabajo.

Con todo lo expuesto hasta el momento, se pueden definir de forma clara los objetivos a realizar en este TFM:

1. La realización e implementación del analizador de ficheros PDB, con el cual poder extraer la información geométrica guardada en estos para su representación en 3D. Esta parte se divide en los sub-objetivos de estudio del formato del archivo para su comprensión y el diseño del algoritmo del analizador sintáctico (*parser*).

2. El programado del renderizado tridimensional de las moléculas. Con los objetivos específicos de preparar funciones y sombreadores (*shaders*) para la tubería de renderizado de OpenGL ES en Android y el uso de la información obtenida del analizador de ficheros para la representación de las moléculas mediante el modelo de representación en 3D de espacio lleno (*space-filling*).

3. La interfaz gráfica de usuario, que ofrecerá a los usuarios la capacidad de interactuar con la aplicación. Ésta tendrá opciones de navegación y selección de ficheros, pudiendo cargar diferentes moléculas en el formato PDB, mediante el analizador desarrollado. También contendrá el renderizador 3D creado para representarlas y ajustar el tamaño, posición y orientación con el que visualizarlas. Y, además, múltiples opciones de configuración para ajustar la representación, a saber: filtrado de átomos que dibujar o no según su elemento o si son estándar, coloreado de átomos según elemento o cadena, selección del color de fondo, mostrar u ocultar un indicador de fotogramas por segundo y rotación automática de la molécula tanto en horizontal como en vertical.

4. Depuración del proyecto. Otro factor significativo es la eficiencia y precisión del software desarrollado o, dicho de otro modo, evitar errores y mal funcionamiento. Puesto que va más allá de la finalidad de este trabajo centrarse en optimizar la aplicación, esto supone un objetivo secundario, pero se tiene en cuenta la mejora sustancial del programa mediante técnicas de depuración o con el uso de patrones de diseño. Se puede considerar que este objetivo se extiende durante toda la duración del proyecto como una idea a tener en cuenta para aplicar estos métodos siempre que se encuentre justificado y sea viable.

5. Redacción de la memoria. Hace falta redactar la memoria para contrastar todos los aspectos del proyecto, la metodología empleada y el resto de factores que han intervenido en el desarrollo del proyecto.

6. Planificación temporal (Diagrama de Gantt). A lo largo del desarrollo se controla la temporización del proyecto para aplicar medidas

de mitigación en caso necesario y comprobar que se lleva a día el progreso.

1.3. Enfoque y método seguido.

Primero, como todo proyecto ha de tener nombre, se elige “MolVie”, acrónimo de *Molecule Viewer* (visor de moléculas en inglés), para el nombre de la App.

Ya se han nombrado los patrones de diseño. Estos consisten en aplicar técnicas en el desarrollo de software que ya se han utilizado para solucionar algún problema general u ofrecer alguna funcionalidad común. Si bien Android ya ofrece muchos patrones de diseño a la hora de programar, esto no es cierto para código nativo, donde se programa en bajo nivel con C. Gracias a los patrones de diseño se puede evitar tener código de mala calidad, reduciendo futuras necesidades de optimización.

Uno de los diseños que se plantea es la programación modular, con lo que se obtienen tres módulos claramente identificables (aun pudiendo tener sub-módulos), que son el análisis del fichero para la extracción de la información geométrica de la molécula a representar, el renderizado 3D a partir de la información anterior y el interfaz de usuario, que actúa como contenedor del renderizado ofreciendo acceso al analizador de ficheros. No hay que confundir módulos con clases u objetos dentro del propio lenguaje, un módulo es capaz de funcionar sin depender de los otros, con lo que ofrece la posibilidad de programar cada uno de forma paralela. Otra ventaja de la programación modular es que, al tener estas funcionalidades claramente separadas, se pueden hacer cambios más efectivos y focalizados a éstas, combinarlas, adaptarlas a otros proyectos e integrar otras funcionalidades.

Otro aspecto significativo es el uso de memoria por la aplicación desarrollada. Aunque ya es cierta la importancia de administrar bien la memoria de acceso aleatorio (RAM), llega a ser todavía más importante en dispositivos móviles, debido a que en estos la memoria disponible es más limitada que en ordenadores convencionales. Por esto, se consideran y diseñan estructuras de datos que reduzcan el consumo de memoria.

Además, si bien se puede programar solamente con Java en Android Studio (con alguna ayuda de XML) sin necesidad de emplear la Interfaz nativa de Java (JNI) y programar en C/C++, para casos de aplicaciones científicas o de alto nivel de cómputo, incluido gráficos, es necesario tener las ventajas de rendimiento que ofrece el sistema de desarrollo nativo de Android (NDK). No es buena idea programar con C en Android solamente para aumentar la compatibilidad y portabilidad del código, ya que añade complejidad al proyecto, pero es un factor del que se beneficia este trabajo ya que se necesitan programar algunas funcionalidades en nativo. Dicho esto, se toma la decisión, de desarrollar gran parte de la aplicación empleando el NDK de Android.

Por último, aparte de usar patrones de diseño, depuración y otros métodos de optimización, se intenta mantener un equilibrio entre eficiencia y comprensibilidad del código, organizándolo, estructurándolo adecuadamente y documentándolo.

1.4. Planificación del Trabajo.

Para poder realizar un proyecto adecuadamente, es necesario realizar un plan de trabajo, organizando las tareas que hacen falta, fijando metas y controlando la temporización de todas las partes. Así que un buen plan de trabajo ayuda a alcanzar todos los objetivos establecidos a tiempo. Seguidamente, se detalla el plan de trabajo de este TFM.

1. Tareas

La implementación del módulo de análisis del fichero se divide en:

· Estudio del formato PDB. Para poder leer el fichero, hace falta comprender el formato PDB para analizar los métodos con que implementar el lector y convertir la información para su representación en tres dimensiones. Se estima una duración de cinco días.

· Definición de la estructura de datos. Después de analizar el fichero, se define la forma en la que se representa y se organiza la información extraída para facilitar el dibujado y tratamiento de estos datos por la *App*. Se estima una duración de tres días.

· Implementación del análisis. Tras definir el modo en que se almacenan estos datos, se pasa a programar la interpretación del fichero. Se estima una duración de cinco días.

· Implementación de la importación. Siguiendo la lectura de datos, se organizan y almacenan según se ha preestablecido para emplear la información útil obtenida. Se estima una duración de cinco días.

La implementación de la parte de renderizado abarca las fases de:

· Definición de las estructuras 3D. Se definen las figuras que serán dibujadas para representar las distintas partes de las moléculas (átomos), en los distintos modelos de representación que sean posibles. Se espera una duración de tres días.

· Compilado de *Shaders*. Una parte importante son los *shaders* (o sombreadores) con los que hacer la renderización de los elementos de las moléculas a representar, aplicando colores y otros efectos como la iluminación y el sombreado. Se espera una duración de cinco días.

· Interpretación de la geometría. Se ha de utilizar la información del análisis de fichero para generar los búferes de vértices que representará el procesador gráfico. Se espera una duración de cinco días.

· Aplicación de transformaciones en el espacio. Antes de ser dibujadas las estructuras, se aplican las transformaciones de posición orientación y tamaño que se adaptan a la interacción con el usuario. Se espera una duración de cinco días.

· Etapa de dibujado. Finalmente se representa gráficamente la información que se ha preparado para dibujar la molécula. Se espera una duración de tres días.

La implementación de la interfaz comprende las tareas de:

· Integración con la lectura de ficheros. La interfaz añade el módulo de lectura. Se espera una duración de un día.

· Integración del renderizado. la interfaz incluye el módulo de dibujado. Se espera una duración de un día.

· Implementación de controles de usuario. Se programa la interfaz para que el usuario pueda seleccionar el fichero que leer y pueda cambiar la rotación, posición y tamaño relativo con el que se muestra la molécula. Se espera una duración de tres días.

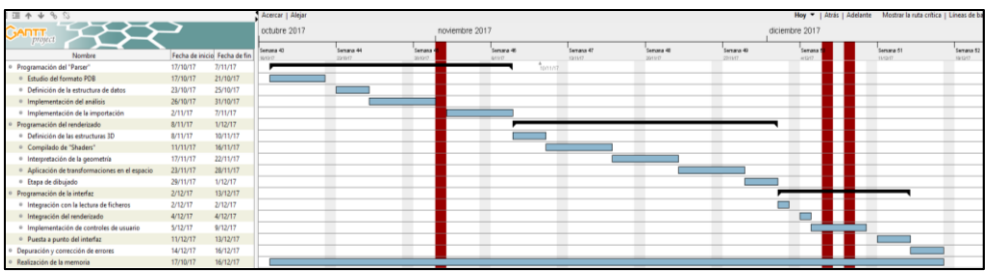
· Puesta a punto del interfaz. Se ajusta el control de la interfaz y se hacen cambios de diseño garantizando una buena experiencia de usuario. Se espera una duración de tres días.

Y finalmente, la etapa de depuración consta de:

· Depuración y corrección de errores. Se corrigen algunos errores de diseño que se hayan podido encontrar. Se espera una duración de tres días.

· Realización de la memoria. a la que se le puede ir añadiendo información durante toda la realización del proyecto conforme se obtienen cambios y resultados.

2. Calendario



1. Diagrama de Gantt.

Las duraciones que pasan por encima de días no hábiles ya se tienen en cuenta. Aunque se muestra el desarrollo de los tres módulos del programa de forma secuencial, se intenta paralelizar lo posible para garantizar la máxima compatibilidad entre módulos a la vez que algunos días se realiza alguna prueba según se avanza para garantizar el progreso del desarrollo.

3. Hitos

Los hitos máximos son los módulos del *parser* y del renderizado, si no se hacen, obviamente no existe proyecto. A estos le sigue el módulo de la interfaz, que solo necesita ofrecer unos mínimos de interacción con el usuario necesarios, a pesar de que se intenta ofrecer la mejor experiencia.

Estos hitos serían desde el punto de vista secuencial ya presentado, como la idea es paralelizar tanto como sea viable, las metas parciales son tener desarrollada la parte en la que se lee un fichero y se prepara la estructura de datos que se representará tridimensionalmente, seguido de la representación gráfica en tres dimensiones de estos datos y la interactividad con el usuario.

4. Análisis de riesgos

Aunque la idea también es programar el renderizado, en caso de no disponer de más tiempo debido a que la implementación del analizador de ficheros se alargue o tener otros problemas con esta implementación, se optaría por utilizar alguna librería, adaptándola e integrándola. Este es el único objetivo que habría que cambiar, afectando a la planificación en gran medida y dependiendo del tiempo empleado en la parte del análisis del fichero.

A priori, todas las demás tareas son realizables dadas las capacidades, las herramientas y el tiempo del que se dispone. Así y todo, se ha incluido días de más en las tareas más importantes para tener en cuenta errores y problemas que puedan surgir durante el desarrollo de la aplicación.

1.5. Breve sumario de productos obtenidos.

La aplicación para Android.

Un fichero con extensión "*.apk*" para instalar la App en dispositivos Android con versión 4.4 o superior. Compilado en modo depuración (*debug*), para poder ser depurado y analizado, usando las herramientas de Android Studio y leyendo el registro de salida (*log*) con información adicional.

El código de la aplicación.

El proyecto de Android Studio que genera la App “MolVie.apk”, con todo el código, configuración, recursos y ficheros de prueba empleados para desarrollar la aplicación y testearla.

Una memoria.

Donde se justifican y dan a entender todas las actividades llevadas a cabo, el trabajo de investigación realizado y, por último, se redactan y analizan los resultados obtenidos.

1.6. Breve descripción de los otros capítulos de la memoria.

Capítulo 2. Estado del arte.

Hace falta conocer algunas plataformas con las que el proyecto que se desarrolla tiene alguna relación y comparar funcionalidades y otros aspectos.

Capítulo 3. Herramientas empleadas.

También es necesario dar a conocer el hardware y software empleado, su importancia, alternativas, si las hay, y motivos por los que se han elegido. Con este capítulo se ayuda al lector a entender mejor el trabajo realizado.

Capítulo 4. Desarrollo del programa.

Una vez dadas las herramientas con las que se trabajan, se pasa explicar el desarrollo y funcionamiento de la aplicación, comentando otras características necesarias.

Además de lo anterior, también se incluyen los siguientes anexos:

Anexo I: Manual de uso de la aplicación.

Anexo II: Sombreadores (*Shaders*) de OpenGL ES, usados en la App.

2. Estado del arte.

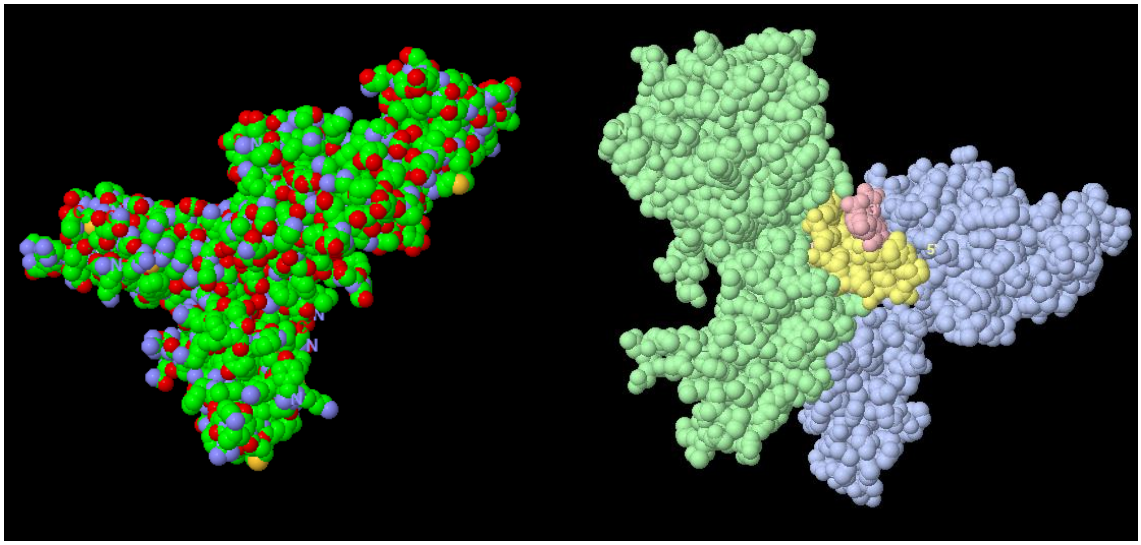
En cuanto a plataformas que, de un modo u otro, puedan estar relacionadas con el proyecto que se desarrolla en este TFM, se destacan las siguientes.

2.1. Jena3D.

Se trata de un visor molecular interactivo disponible en la Web, que representa los compuestos mediante *Jmol*, usando un applet Java, o *JSmol* usando *JavaScript* con *HTML5*, según elija el usuario.

Admite PDB entre otros formatos (*mmCif*, *CIF*, *CML*, *MOL*, *XYZ*). Acepta tanto ficheros locales como búsqueda en bases de datos como *PDB*, *NDB* y *UniProt*.

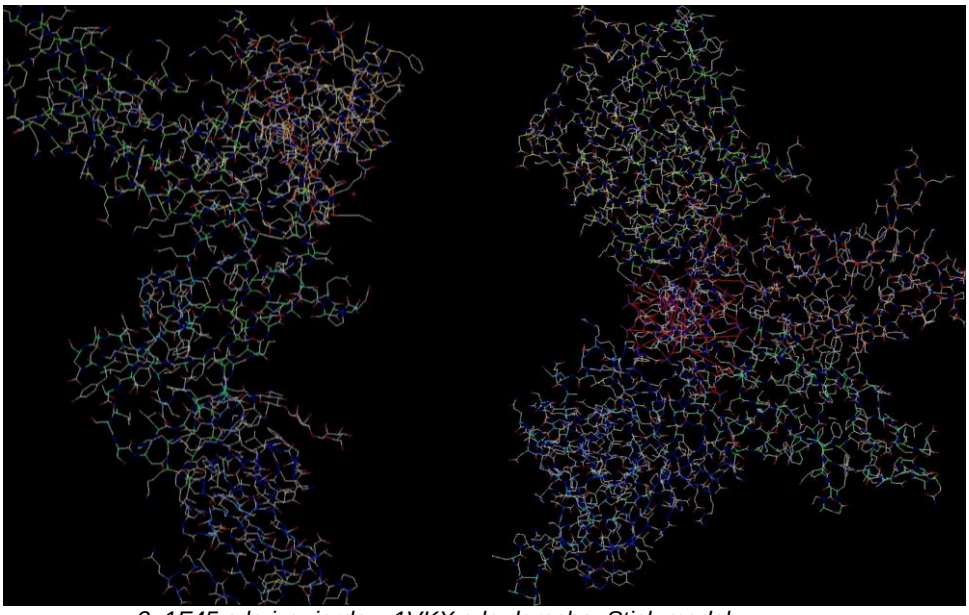
Como características a mencionar, permite renderizar en varios modelos de representación molecular (Space-filling, Ribbon diagram, *Bonds*) y colorear según cadena o átomo, entre otros.



2. 1F45 a la izquierda, usando space-filling y coloreado por átomos. 1VKX a la derecha, esta vez coloreado por cadena.

2.2. NDKmol.

Disponible en Android para la versión 2.2 y superior. Última vez actualizado en 2015. Ofrece varios modelos de representación divididos entre polímeros y ligandos (no tiene los mismos modelos para ambos tipos). El interfaz es manejable, pero no contiene explorador de archivos y la opción de búsqueda online no funciona.

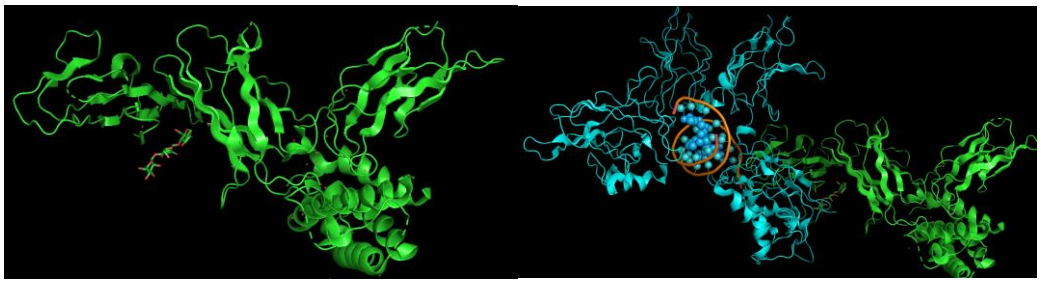


3. 1F45 a la izquierda y 1VKX a la derecha. Stick model.

2.3. PyMol.

Herramienta de gráficos moleculares de código abierto (aunque los binarios son privativos) creado en *Python* que permite tanto visualización como edición de proteínas y otras moléculas.

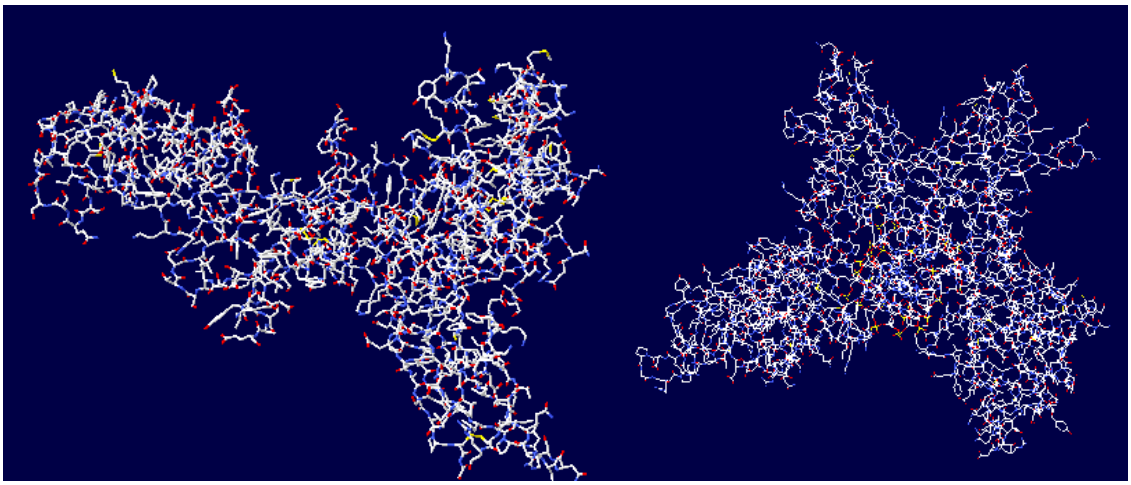
Acepta ficheros en diversos formatos como *PDB*, *MOL*, *XPLOR* entre otros. Su interfaz es bastante complejo, tanto los controles como los métodos de representación.



4. 1F45 a la izquierda y 1VKX a la derecha. Ribbon diagram.

2.4. DeepView (Swiss-PdbViewer).

Aplicación de escritorio, disponible en Windows, Mac y Linux. Acepta ficheros *PDB*, *MOL* y *mmCif*. Permite analizar varias moléculas al mismo tiempo, aunque su interfaz es poco amigable. Su última actualización fue en 2012. Y, por último, no ofrece muchas opciones de coloreado ni modelos de representación.



5. 1F45 a la izquierda y 1VKX a la derecha. Stick model.

Comparación de las plataformas.

Se añade una tabla comparando algunas de las funcionalidades de las demás aplicaciones y la desarrollada en este trabajo, las cuales se consideran más importantes en el ámbito de este proyecto.

	Uso fácil	Modelos	Coloreado	Android	Actualizado
Jena3D	Sí	Múltiples	Múltiples	Navegador	Sí
NDKmol	Sí	Múltiples	Múltiples	Sí	No
PyMol	No	Múltiples	Múltiples	No	Sí
DeepView	No	Múltiples	Múltiples	No	No
MolVie	Sí	Space-fill	Múltiples	Sí	Sí

6. Tabla comparativa de plataformas relacionadas.

3. Herramientas empleadas.

Para poder empezar con el desarrollo de la plataforma, es necesario saber todas las herramientas, componentes y utilidades que se emplean en el proyecto, la importancia que tienen y por qué se han elegido. Al mismo tiempo, este capítulo también ayuda al lector a entender mejor el trabajo realizado en el TFM.

Banco de Datos de Proteínas.

El *Protein Data Bank* fue creado en 1971 por los doctores Edgar Meyer y Walter Hamilton con el propósito de almacenar datos estructurales de moléculas en tres dimensiones, como son proteínas y ácidos nucleicos. Aunque en aquellos años empezó con pocas proteínas registradas como para ser útil, en 2014 superó las cien mil entradas. El *PDB* se mantiene a día de hoy como la base de datos central para estas estructuras, gracias al *Worldwide Protein Data Bank*, que permite el almacenamiento y la distribución de forma gratuita y universal. A esto hay que añadirle que todas las proteínas y ácidos nucleicos subidos se mantienen bajo el dominio público, por lo que pueden ser usados libremente.

El Worldwide Protein Data Bank (wwPDB) está formado por diferentes organizaciones que se hacen cargo del almacenamiento, validación y oferta de los archivos PDB. Actualmente, los miembros que forman esta asociación son PDBj (Japón), PDBe (Europa), BMRB (EE. UU.) y RCSB PDB (EE. UU.). Además, el wwPDB también realiza conferencias, talleres y estudios sobre biología estructural.

Los datos de las proteínas y ácidos nucleicos de este repositorio se obtienen mediante diversas técnicas, como son la cristalografía de rayos X, la espectroscopia mediante resonancia magnética nuclear (RMN), criomicroscopía electrónica y otros métodos. Estas estructuras son entregadas por biólogos y bioquímicos de todo el mundo y supervisadas por el *Worldwide Protein Data Bank*.

Su formato homónimo, con extensión de archivo "*.pdb*", consiste en un fichero de texto con descripción de las estructuras de proteínas y ácidos nucleicos como son las coordenadas atómicas, secuencias, anotaciones o conectividad. Estos ficheros reciben un identificador alfanumérico único de 4 caracteres. El formato se creó para ser fácilmente legible, con un límite de 80 caracteres por línea debido a la limitación de las tarjetas perforadas que se usaban por aquella década. Un defecto de este formato limitado es que algunas estructuras grandes, como ribosomas, tienen que ser divididos en múltiples ficheros.

Los archivos de este formato están formados por líneas de texto que empiezan por una palabra clave de 6 caracteres (con espacios si es necesario) seguida de información dependiendo de esta palabra. Un ejemplo de una porción de un fichero sería:

```
HEADER    OXYGEN STORAGE                          27-AUG-81   1MBO
TITLE     STRUCTURE AND REFINEMENT OF OXYMYOGLOBIN AT 1.6 ANGSTROMS RESOLUTION
[...]
ATOM      1  N   VAL A   1      -0.686  14.852  16.090  1.00 42.31           N
ATOM      2  CA  VAL A   1       0.621  15.543  16.081  1.00 53.86           C
[...]
CONECT  746 1276
CONECT 1229 1230 1231 1232 1233
[...]
MASTER      295    0    3    8    0    0    9    6 1601    1   52   12
END
```

7. Ejemplo de fichero PDB.

Como se puede observar, las líneas HEADER y TITLE contienen información sobre el tipo de estructura, fecha y otros datos. Las líneas ATOM, por otro lado, contienen información sobre los átomos que forman la macromolécula registrada, como son las coordenadas espaciales y el elemento químico. Más información sobre el formato PDB puede ser encontrada en la referencia número 8 de la bibliografía.

Se elige este formato como el inicial para este proyecto debido a ser el estándar de facto en formatos de datos de estructuras macromoleculares, por lo que es necesario estar incluido en cualquier entorno. Esto beneficia a la aplicación en que consigue un alcance de usuarios más amplio que si se usara algún otro formato menos conocido o poco empleado.

Más concretamente, las líneas que se explican a continuación son las elegidas inicialmente para el lector de ficheros, dejando el resto para futuras expansiones, teniendo en cuenta el límite de tiempo que se tiene para el desarrollo:

· NUMMDL: En caso de que exista, viene a indicar el número de modelos para una misma estructura que contiene el fichero.

· MODEL: Indica el inicio de información pertinente a un modelo. Las entradas seguidas a esta línea contienen información sobre un modelo.

· ENDMDL: Marca el fin de un modelo precedido por MODEL.

· ATOM: Contiene información sobre un átomo estándar del compuesto. Se puede obtener coordenadas, elemento, carga, una identificación de átomo, una identificación de cadena, entre otros parámetros.

· HETATM: Guarda información sobre átomos no estándar, con la misma información que la entrada ATOM.

· CONECT: Nombra conexiones entre los átomos del fichero. Cada entrada contiene un identificador de un átomo, seguido de hasta cuatro identificadores más, indicando otros átomos con los que enlace el primero. En caso de tener más de cuatro enlaces el mencionado, existirán más líneas para añadir los restantes. También, cada uno de estos átomos que se enlazan,

tendrán su propia entrada enlazando al primero, por lo que el algoritmo que se implementa en este proyecto ha de verificar y evitar conexiones redundantes.

· MASTER: Tiene recuento de entradas del fichero, como átomos totales o conexiones. Puede ser usado para comprobaciones y recuentos. En caso de haber múltiples modelos en el fichero, los números indicados son para el primer modelo y no para la totalidad del fichero.

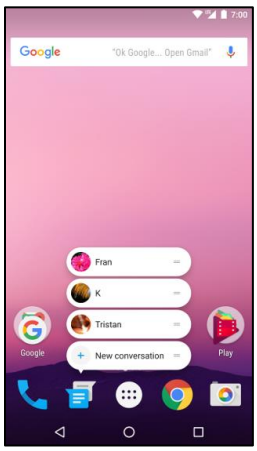
· END: Indica el final del fichero PDB.

Android.

El software libre Android, que incluye tanto el sistema operativo basado en Linux como aplicaciones que se ejecutan en este, se inició en 2003 por Andy Rubin, Rich Miner, Nick Sears, y Chris White, con el lanzamiento de la primera versión de su sistema operativo en 2008. Fue adquirido por la empresa Google, que lo ha convertido en el sistema operativo de teléfonos inteligentes más vendido en todo el mundo.

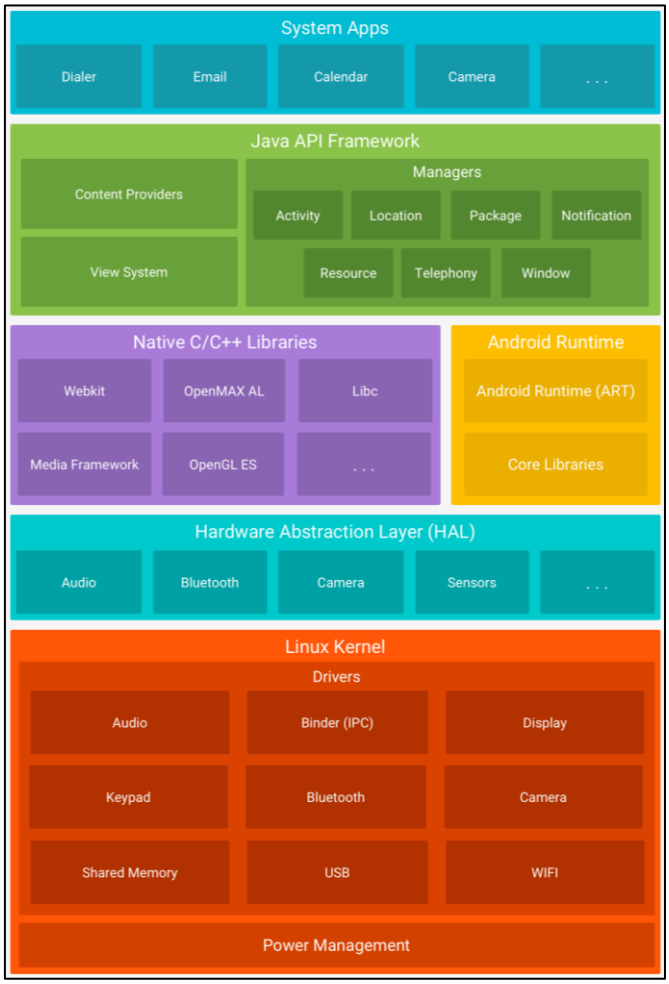
El sistema operativo cuenta con un diseño que aprovecha todas las características que ofrecen los dispositivos móviles inteligentes de hoy en día, como son las pantallas táctiles, cámara y micrófono integrados, además de una extensa lista de sensores (acelerómetro, sensor de luz, barómetro...) y diversas tecnologías de comunicación inalámbrica (GSM, Bluetooth, NFC…). De este modo, presenta una interfaz rápida y de fácil uso, que puede adaptarse a cualquier tipo de interacción con el usuario de forma intuitiva, ya sea tocando la pantalla del dispositivo o rotándolo entre otras acciones.

Al encenderse un dispositivo con Android, se muestra un escritorio o pantalla de inicio, que contiene iconos de aplicaciones y widgets (pequeños componentes interactivos), junto con una barra de estado y unos botones virtuales de navegación. También dispone de los botones físicos incluidos en el propio dispositivo, normalmente controles de volumen y apagado, y se pueden extender sus controles, amén de sus funcionalidades, conectando el dispositivo a otros periféricos mediante Bluetooth, USB o Wifi, como por ejemplo teclados físicos, auriculares, altavoces, joysticks, televisores, impresoras, etc.



8. Dispositivo con Android Nougat.

Debido a que los dispositivos que usan Android suelen tener limitaciones en cuanto a la alimentación y memoria, Android está diseñado para reducir el consumo de energía y memoria, suspendiendo y cerrando aplicaciones y procesos inactivos. Aunque los dispositivos Android también se han servido de la arquitectura ARM para reducir costes y consumo energético, en las últimas versiones, Android también soporta x86.



9. Arquitectura de la plataforma Android.

Android se fundamenta en usar aplicaciones (*Apps*) que, a partir de las características del sistema, ofrezcan funcionalidades al usuario. Estas aplicaciones vienen en un formato "*.apk"*, el cual se trata de un paquete para instalar la aplicación en los dispositivos. Las Apps se desarrollan usando el kit de desarrollo de software propio de Android (*Android SDK*), las cuales se programan principalmente en el lenguaje de propósito general *Java*. También se puede emplear el meta-lenguaje *XML* para diseños (*layouts*) y otros recursos, creando elementos de interfaz de forma sencilla. A esto hay que sumarle la posibilidad de emplear el kit de desarrollo nativo de Android, pudiendo programar aplicaciones que necesiten el máximo rendimiento posible, gracias a la programación con el lenguaje *C/C++.*

Android Studio es el entorno de desarrollo integrado (IDE) oficial desde 2014, sustituyendo a Eclipse, que se servía de módulos (*plugins*) hasta el

momento. Android Studio se encarga de ofrecer todas las funcionalidades y ayudas necesarias para facilitar el desarrollo de Apps. Incluye las herramientas básicas que suelen tener todos los entornos de desarrollo, como son el completado de código, coloreado de sintaxis, depurado y detección de errores, etc. También cuenta con Gradle (con su lenguaje específico), que se encarga de la compilación de las aplicaciones; un emulador o dispositivo virtual (AVD), para probar aplicaciones; herramientas de control de versión y soporte de almacenamiento en la nube.

Hace falta mencionar que existen una numerosa cantidad de alternativas para el desarrollo en Android aparte de Android Studio. Como, por ejemplo, *Visual Studio*, *IntelliJ IDEA*, *Thunkable*, *QT Creator* y *Bubble*. Algunos de estos requieren más o menos nivel de programación o incluso usan otros lenguajes de programación distintos a los que emplea Android, permitiendo construir aplicaciones para diversas plataformas y sistemas operativos.

Como se ha expuesto, Android es el sistema operativo idóneo con el que desarrollar la aplicación, tanto por sus características como por el hecho de existir un gran mercado donde lanzar la aplicación que se desarrolla en este proyecto. Así pues, se elige Android Studio como el entorno en el que desarrollar la aplicación por ser la oficial, con más soporte, ofrecer libertad a la hora de programar y tener, el alumno, experiencia previa trabajando con éste.

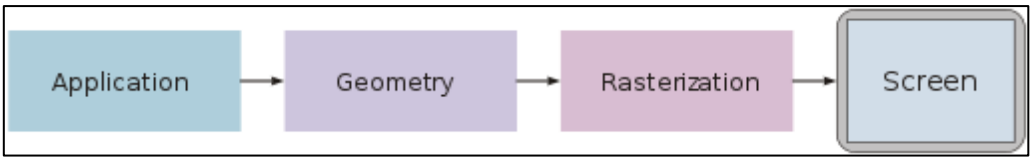
OpenGL ES.

OpenGL ES es una versión de la API gráfica OpenGL del Grupo Khronos diseñada reducida específicamente para dispositivos integrados como por ejemplo móviles inteligentes. Su última versión es la 3.2 que es retrocompatible con la versión 2.0, permitiendo a las aplicaciones incorporar nuevas funcionalidades visuales. Con su lanzamiento inicial en 2003, y la versión 3.0 puesta al público en 2012, ofrece a los dispositivos la capacidad de generar gráficos tanto en 2D como en 3D acelerados por hardware, integrada ampliamente en casi la totalidad de los dispositivos existentes.

Para programar, del mismo modo que con OpenGL, se utiliza su lenguaje propio GLSL, que está basado en C, con el que se obtiene acceso directo a su tubería de renderizado. Gracias a tener este lenguaje específico, se puede desarrollar código multiplataforma que será optimizado para el dispositivo mediante el compilador de GLSL.

La metodología para trabajar con este lenguaje consiste en el programado de sombreadores (*shaders*), consistente en código que se encarga de varias etapas de la tubería de renderizado de OpenGL. Específicamente, en OpenGL, después del inicio de la aplicación, se interpreta la geometría a representar, consistente en coordenadas de vértices y los polígonos que estos forman, aplicando transformaciones (como proyección) y recortes. En OpenGL, esta etapa es administrada por el sombreador de vértices (*vertex shader*). Después de esto le sigue la etapa de rasterización, en la cual se generan los píxeles que

van a ser representados en pantalla. Este proceso es controlado por el sombreador de fragmentos (*fragment shader*) en OpenGL.



10. Tubería de renderizado de OpenGL.

Puesto que es el único estándar disponible para gráficos acelerados en Android, será necesario emplearlo. Aun pudiendo usar otras librerías que estén implementadas sobre la de OpenGL ES, se prefiere emplear directamente ésta para poder optimizar el dibujado de un gran número de átomos en pantalla. Y, como ya se ha mencionado, se emplearán *Vertex Shaders* y *Fragment Shaders* desde nativo para la realización de la parte de renderizado de las estructuras moleculares. Para más información sobre el estándar, léase la referencia 9 de la bibliografía.

Dispositivos de prueba.

Para poder desarrollar y testear la aplicación, hace falta un equipo en el que programar y dispositivos móviles, tanto virtuales como físicos, para probar la aplicación. Para el desarrollo de la App se usaron los siguientes dispositivos:

· PC para programar con Android Studio: Sistema Operativo Windows 10, Procesador Intel i7-4790 (3,6 GHz) y Memoria RAM de 16 GB.

· Smartphone para pruebas: ZTE Axon mini, Android 5.1.1, Procesador Qualcomm Snapdragon 616 (1,5 GHz), Memoria RAM de 3 GB y GPU Adreno 405 (550 MHz, 48 ALUs y OpenGL ES 3.1).

Para probar la aplicación también hacen falta ficheros PDB. Se eligieron las siguientes macromoléculas: Interleucina-12 (1F45), Oximioglobina (1MBO), NF-kappaB p50 / p65 (1VKX) y Receptor adrenérgico beta 2.

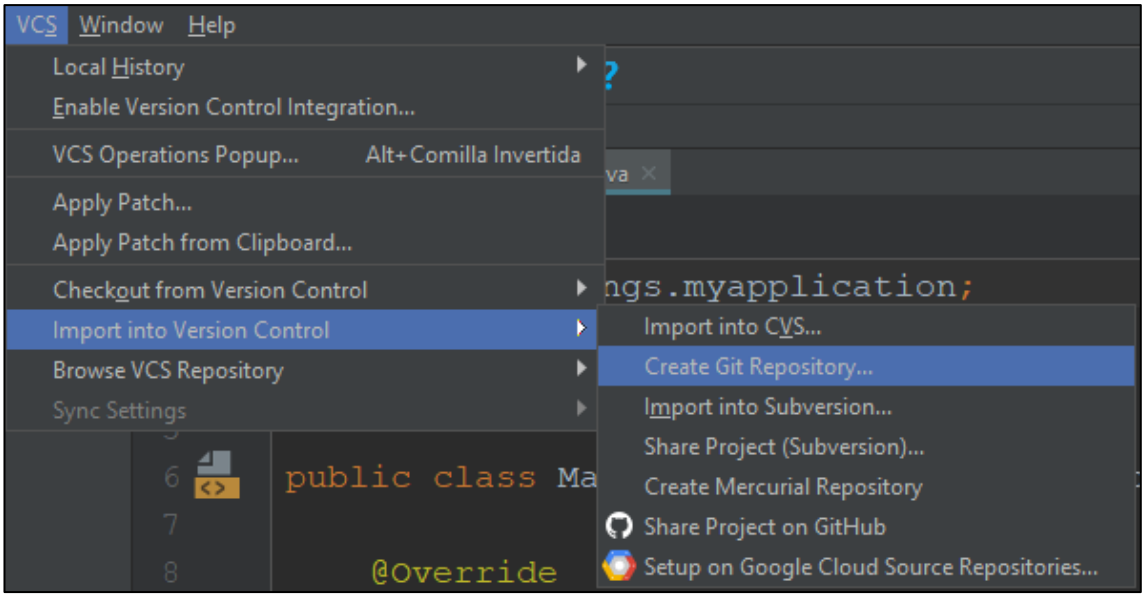
Estos datos son importantes a la hora de lanzar la aplicación, para tener en cuenta como se desenvolverá la aplicación en otros dispositivos. Sobre todo, teniendo en cuenta la gran diversidad de dispositivos que emplean el sistema operativo Android.

Sistema de control de versiones.

Android Studio tiene integrado la herramienta Git, que se trata de un software de control de versiones que permite gestionar, configurar y mantener un historial de cambios en el código fuente de los proyectos que se desarrollen. Se usa el término repositorio para referirse al lugar donde se guardan todos los cambios, junto con la última versión. Cada una de las versiones (o cambios) la cual es añadida junto con sus cambios, es llamada revisión.

El uso de repositorios para mantener el control de versiones es útil para controlar cambios funcionales en el código, teniendo así un historial de todo lo desarrollado, en caso de necesitar acceder a código eliminado anteriormente para restaurarlo.

Para poder usar esta utilidad en Android Studio, tan solo hay que acceder al menú VCS (*Version Control System*) y en el submenú de Importación se elige la opción de crear el repositorio en Git (o importarlo).



11. Creación de un repositorio en Git.

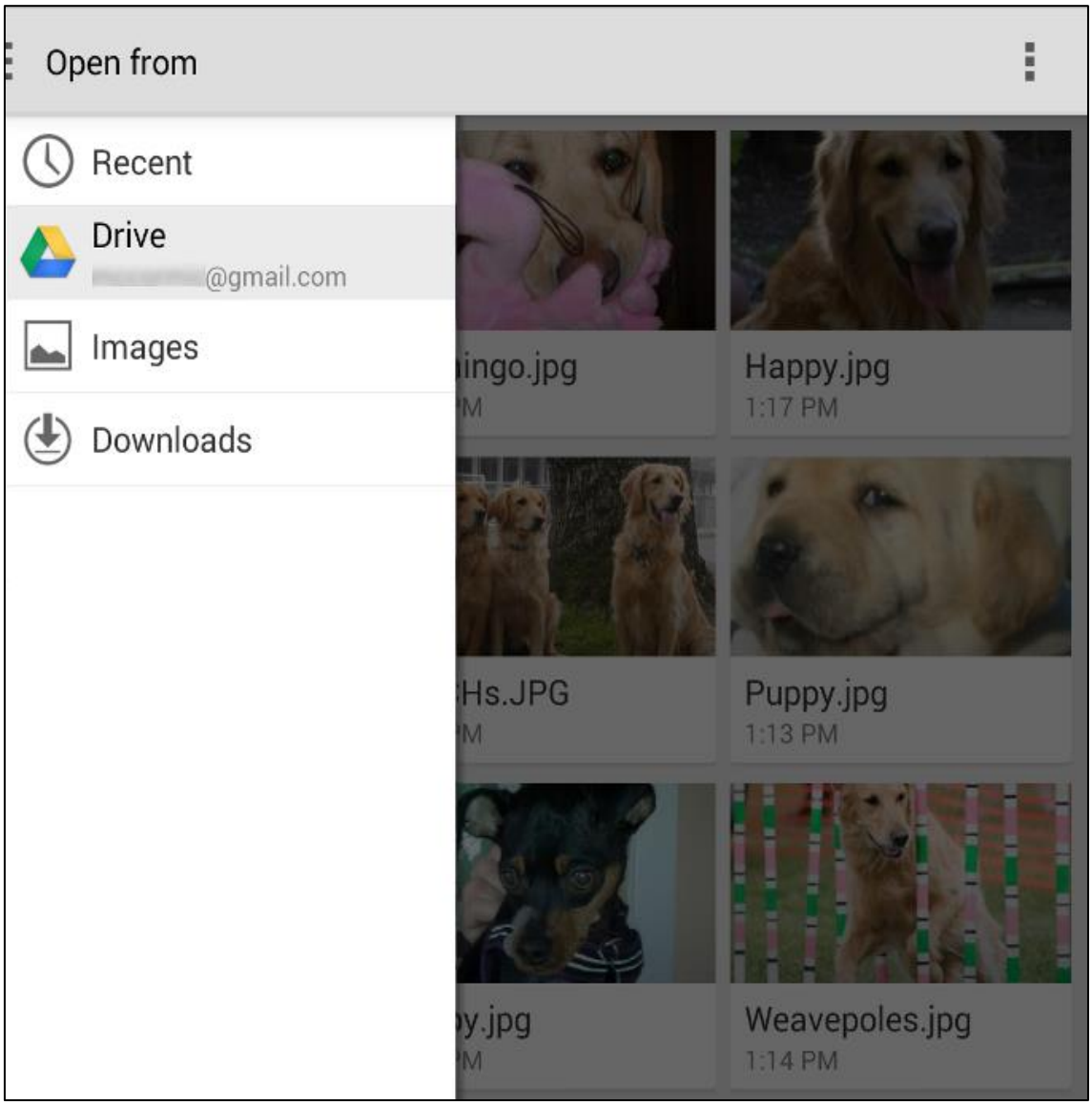
De este modo se incluye el control de versiones en el proyecto de la aplicación a desarrollar.

Además, se emplea la plataforma GitHub para alojar el proyecto. Android Studio también ofrece la posibilidad de subir los repositorios Git a GitHub, así, de forma sencilla, se puede mantener el código en la nube, pudiendo gestionar y colaborar en línea.

4. Desarrollo de la aplicación.

Lector de ficheros PDB.

A partir de la versión 4.4 de Android (nivel de API 19), se integra un sistema de acceso al almacenamiento para incorporar de forma fácil y rápida un explorador de ficheros en cualquier aplicación que se desarrolle con el cual abrir documentos, imágenes y otros tipos de archivos, tanto de la nube (Google Drive) como del almacenamiento del propio dispositivo.



12. Framework de acceso al almacenamiento (SAF).

Se intentó implementar esta funcionalidad en la aplicación a desarrollar en este proyecto, pero debido a dos problemas no se pudo.

La primera dificultad consiste en que este sistema devuelve un identificador de recursos uniforme (URI), en lugar de una ruta al fichero. Con este identificador, Android puede generar un flujo de lectura a este archivo, pero únicamente desde Java. Como el objetivo del proyecto es realizar la lectura del fichero mediante código nativo para optimizar el análisis, el método de generar un flujo no sirve. Una posible solución sería copiar este flujo a una ruta fija a modo de archivo temporal, pero se añade la sobrecarga de leer el fichero una vez más y escribirlo, que escapa al interés de ofrecer una lectura rápida del archivo.

El otro inconveniente se trata de que, aun pudiendo obtener una ruta de archivo a partir del URI, desde la versión 4.2, Android permite el uso de múltiples cuentas de usuario en un mismo dispositivo, lo cual genera rutas virtuales, que impiden acceder al fichero desde código nativo mediante este método.

Así pues, se implementa un selector de ficheros propio. El cual consiste en un *ScrollView* que se muestra en forma de diálogo a solicitud del usuario. El *ScrollView* permite desplazar el contenido en su interior, verticalmente, en caso de superar el número de elementos que pueda mostrar simultáneamente, con tan solo deslizar el dedo sobre este. Éste contiene una lista de botones generados de forma dinámica, con los nombres de las carpetas y ficheros del directorio que se esté examinando. Inicialmente se forma la lista en la ruta de almacenaje externo de Android (no es necesariamente la memoria extraíble). Al pulsar sobre un botón perteneciente a un directorio, en caso de no estar vacío se genera una nueva lista en este mismo contenedor con los directorios y archivos que contenga, en caso contrario, se muestra un mensaje al usuario indicando que la carpeta seleccionada está vacía. Al seleccionar un archivo, se procede a su lectura, notificando al usuario de que no ha sido leído ningún átomo en caso de no ser un fichero PDB o estar vacío.

Pasando ahora al algoritmo del analizador, una vez se empieza a leer el fichero, línea a línea se comparan los caracteres iniciales y en caso de coincidir con las palabras clave que se aceptan, se pasa a procesar la línea. En esta versión de la aplicación, del fichero PDB se leen las siguientes líneas del modo que se pasa a explicar:

· NUMMDL: Si aparece, se obtiene la cantidad de modelos totales, pudiendo comparar con el número de modelos leídos. Los caracteres del 11 al 14 contienen este número, ofreciendo valores posibles entre 1 y 9999. Para este rango de valores hace falta un entero sin signo de 16 bits. En caso de no existir la línea o estar mal formada, el número de modelos será 0. Este parámetro es útil para comparar el número de modelos que se han procesados y compararlo por si existe algún error.

· MODEL: Para organizar distintos modelos, se mantiene un recuento de modelos y se añade esta información a los átomos que se leen. Los caracteres del 11 al 14 indican el número de modelo que se abre. La App utiliza esto para,

a cada átomo que se esté procesando a partir de este momento, incluirle el número de modelo al que pertenece y así diferenciarlo. Además, se mantiene un recuento de este número, para compararlo con el de NUMMDL.

· ENDMDL: Fin de un modelo precedido por MODEL, se puede comparar si cada modelo indicado por MODEL es cerrado por esta línea. La línea no tiene otra información, pero la aplicación mantiene un recuento, para comparar también si todos los MODEL han sido cerrados por esta línea y, así, encontrar posibles errores en el fichero.

· ATOM y HETATM: De estas líneas se obtiene la información importante para la representación tridimensional. Los caracteres del 7 al 11 contienen el identificativo del átomo, teniendo números del 1 al 99999, por lo que hace falta un entero sin signo de 32 bits. El identificador de cadena se encuentra en el carácter 22, por lo que con 8 bits es suficiente. Del 31 al 38, del 39 al 46 y del 47 al 54 contienen los valores X, Y y Z respectivamente, como en este caso las variables pueden ser negativas, hacen falta enteros con signo de 32 bits para almacenar esta información. Después, en los caracteres 77 y 78 se encuentra el símbolo del elemento químico del átomo (justificado a la derecha). A estas variables que se obtienen de la línea, se les añade también el modelo actual y el tipo (ATOM o HETATM). Una vez procesada esta línea y generado los datos, se añade a un vector (de la librería estándar de C++) que almacena todos los átomos para su posterior representación.

· CONECT: Se guardan conexiones entre átomos en otro vector, comprobando que no estén repetidas. Los caracteres del 7 al 11 indican el serial del átomo origen al cual enlazan los siguientes, que aparecen en los caracteres del 12 al 16, del 17 al 21, del 22 al 26 y del 27 al 31. En caso de estar, alguno de estos, vacío, no se añade (puesto que no existe), lo que significa que hay menos de 4 enlaces en esa entrada. Además, se itera sobre la actual lista de conexiones para comprobar que no se ha añadido anteriormente (uno de los identificadores origen, puede aparecer como enlazado de otro) para evitar duplicados.

· MASTER: Se obtiene el número total de átomos y conexiones que indica el fichero para comprobación de los datos. En los caracteres del 51 al 55 se encuentra el número de átomos (ATOM más HETATM) indicados por el fichero, así se puede comprobar el tamaño del vector de átomos generado con este número y verificar el fichero. También, en los caracteres del 61 al 65 se obtiene cuantas entradas CONECT hay, esperando que este número sea mayor que el procesado debido a que se ignoran duplicados.

· END: Fin del fichero. A partir de esta línea (o al llegar al final del fichero en caso de no aparecer), se deja de procesar el fichero y se puede pasar a representar los datos.

Estructura de los datos.

Los átomos y conexiones se añaden a un vector en C++ de la librería estándar, el cual permite añadir elementos a éste de forma dinámica. Esta clase

administra la memoria en tiempo de ejecución para ir añadiendo nuevos elementos sin tener que fijar un espacio máximo antes de compilar. Al terminar la lectura de un fichero se usa la función "*shrink_to_fit()*" para eliminar espacio no usado.

Siguiendo esta filosofía de optimización de almacenamiento, para asegurar el mínimo uso de memoria sin afectar al rendimiento de la aplicación y mejorando la compatibilidad, los elementos dentro de las estructuras de datos (*struct*) de los átomos y conexiones son enteros de ancho mínimo (*minimum-width integer*), de esta forma el código es adaptable a cualquier sistema. Esto es, *int_leastX_t* y *uint_leastX_t*, dónde *X* es el número mínimo necesario de bits para almacenar la variable en el sistema. Como alternativas existen *int_fastX_t* y *uint_fastX_t*, que aseguran que la variable ofrece un mayor rendimiento, pero no que ocupe el menor tamaño posible, después están *intX_t* y *uintX_t*, que aseguran el tamaño exacto, pero podrían no venir implementadas en alguna arquitectura, o incluso ser reemplazadas por un tipo de mayor tamaño mediante enmascaramiento de bits, el cual reduciría la eficiencia.

Del modo explicado, para almacenar estas variables de diferentes tipos, se emplea el tipo de datos "*struct*" que ofrece el lenguaje C.

```
struct atomo {
        uint_least32_t serial;    // Máximo 99999
        int_least32_t x, y, z;    // Máximo 9999.999; Mínimo -999.999
        char elemento[2];         // Ej. 'Ca', 'Fe', 'Ba', 'Be'
        bool hetero;              // true: HETATM; false: ATOM
        uint_least16_t modelo; // Máximo 9999
        uint_least8_t charId;     // Identificado de cadena
    };
```

13. Estructura de datos interna de los átomos.

Así, los datos se mantienen de forma estructurada y es más sencillo acceder a ellos.

Representación tridimensional.

Para el renderizado 3D se intentó organizar los vértices de los poliedros que simularían las esferas para representar los átomos de las estructuras moleculares. Pero, debido a algún motivo que no se llegó a conocer, algunas caras de los poliedros no se dibujaban.

Afortunadamente, se consiguió implementar una alternativa, la cual consiste en representar planos en lugar de poliedros, dibujando sobre estos planos las esferas, gracias a los *shaders* de fragmentos que ofrece OpenGL ES.

Pasando a explicar esta solución, se incluye el siguiente código dentro del sombreador de fragmentos:

```
"vec2 p = 2.0*((gl_FragCoord.xy) / fDimrat.xy) - 1.0 - fTranslation.xy;"
```

El tipo “vec2” se trata de un vector de 2 valores (*.x* y *.y*).

La variable “gl_FragCoord” forma parte del estándar de OpenGL y contiene las coordenadas del pixel del fragmento que se pasa a procesar por el programa. Estas variables se divided entre “fDimrat”, que es una variable propia que se añade y contiene las dimensiones de la pantalla. Puesto que estos valores que se obtienen son relativos y van de 0 a 1, se multiplican por 2 y restan 1 para ser entre -1 y 1, pasando a las coordenadas en pantalla que usa OpenGL para la ventana de renderizado (viewport), y se resta la traslación que tienen respecto al origen, para centrarlos en este. De este modo, la variable “p” contendrá la posición de cada pixel del fragmento que se procesa en relación a su centro.

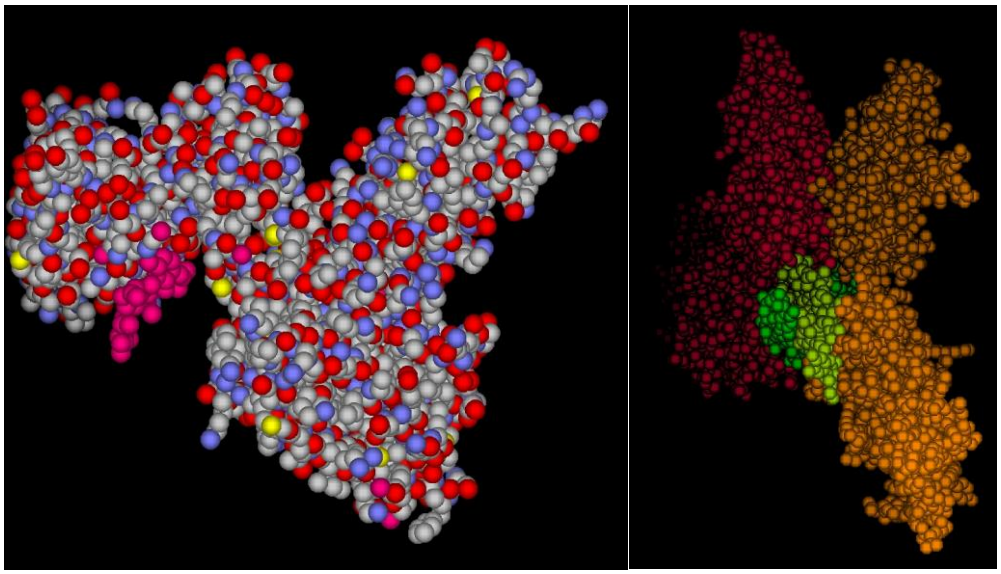
```
“float r = sqrt(dot(p/vec2(1.0, fDimrat.z), p/vec2(1.0, fDimrat.z)))/fScale.x;”
```

El método “dot” de OpenGL, calcula el producto escalar entre dos vectores, el tercer atributo de “fDimrat” (.z), se ha utilizado para almacenar el ratio de aspecto de la pantalla, de este modo se asegura que, indistintamente de las dimensiones de la pantalla, se dibuja una circunferencia correctamente. Si solo se aplicara el producto escalar, se obtendría la distancia lineal respecto al centro y, al usar esta variable para iluminar, se tendría una variación lineal. Para dar el efecto de que tiene más profundidad, más concretamente, el de una esfera, se aplica la raíz cuadrada (función “sqrt”), de este modo ya no varía de forma lineal. Antes de aplicar este valor, se divide entre la escala para ajustar los valores según se requiera dibujar un círculo de mayor o menor tamaño. Finalmente, esta variable se aplica al color (gl_FragColor) para ajustar su iluminación y darle el efecto al átomo.

Ya que la esfera se ve igual tenga la rotación que tenga, más que una solución alternativa esto supone una mejora en cuanto a la calidad visual, pudiendo dibujar esferas lisas de mayor a menor tamaño con la misma precisión visual y simplificando el uso de memoria.

Para aplicar desplazamiento, escalado y rotación a las estructuras moleculares, también se tuvieron problemas con la integración de la librería GLM a Android y se optó por implementar las funcionalidades de escalado, translación y rotación programando manualmente las matrices, gracias a que el lenguaje GLSL permite operar con matrices de forma sencilla y a que la complejidad de trabajar con los vértices había sido reducida.

Como funcionalidades aplicadas al dibujado constan el filtrado por elementos (se ignoran o dibujan los átomos según el elemento), el filtrado por átomos estándar (según la variable registrada de si es ATOM o HETATM), coloreado según el elemento (se mira que elemento es antes de seleccionar el color) y el coloreado por cadena (se mira la cadena antes de dibujar).



14. 1F45 a la izquierda, coloreado por elemento, y 1VKX a la derecha, coloreado por cadena.

Interfaz gráfica de usuario.

La interfaz se ha diseñado para ser amigable, intuitiva, y, en general, fácil de usar. Al iniciar la aplicación, se muestra directamente la pantalla en donde se representan las moléculas (el renderizador) y se enseña un mensaje al usuario para recordar que en el menú de opciones se puede abrir un fichero. Pasando a explicar los componentes de la interfaz, se tiene:

· La superficie de renderizado, que abarca casi la totalidad de la pantalla. Ésta es la parte en la cual se dibujan las moléculas y, a su vez, responde a las interacciones táctiles con el usuario.

· Un indicador de fotogramas por segundo que puede mostrarse u ocultarse, situado en la parte superior izquierda de la pantalla. Útil para observar el rendimiento de la aplicación en el dispositivo y con la actual estructura que se esté representando con sus ajustes.

· En la parte inferior, ocupando lo mínimo posible, unos ajustes rápidos para elegir entre mostrar u ocultar los átomos estándar y no estándar, para aplicar y percibir los cambios de forma rápida.

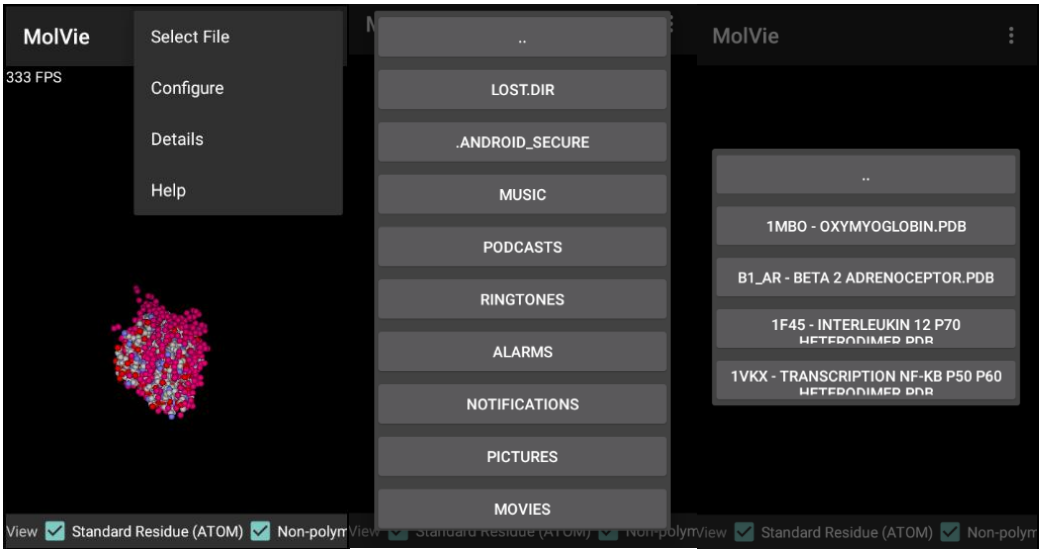
Las aplicaciones Android, por defecto ofrecen un menú de opciones, adaptado a la versión del sistema operativo instalado. Al programar la aplicación, se ha hecho uso de esta utilidad y, al crearse el menú de opciones, se añaden distintos ítems y se controla el que el usuario elija del menú desplegable, para acceder a otras funcionalidades y aspectos de la App. Al pulsar el botón de opciones, se despliega un menú que permite acceder a los siguientes ítems:

· Selector de ficheros. Que utiliza el selector de ficheros ya comentado para elegir y abrir un fichero PDB. Al finalizar la lectura de forma correcta, se representa la estructura en la superficie existente.

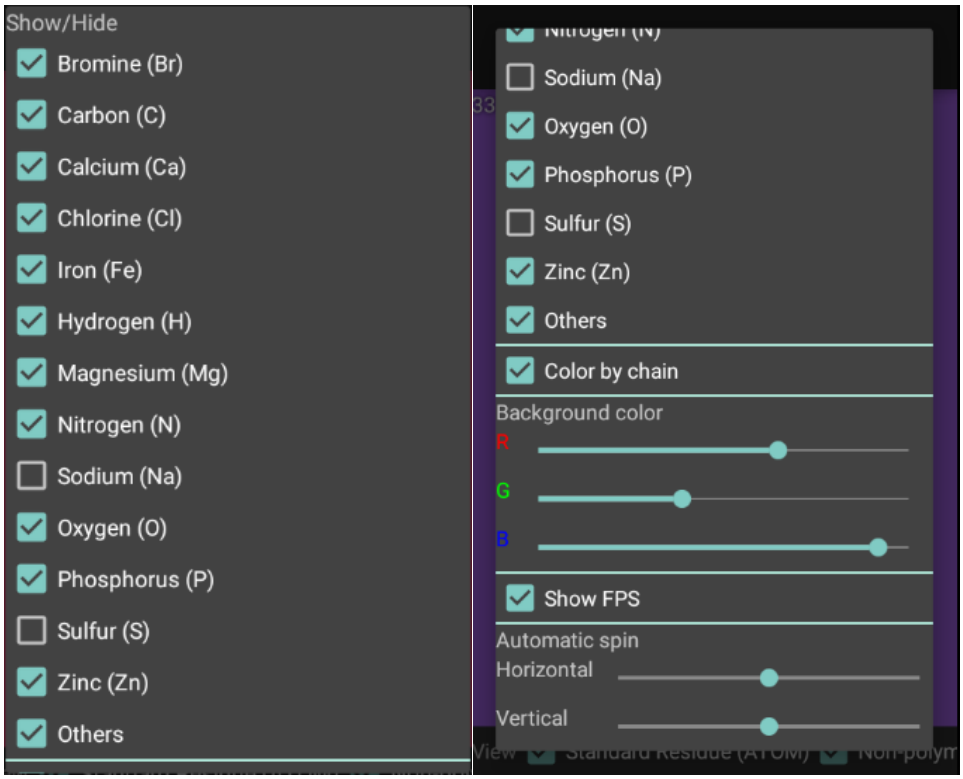
· Menú de configuración. Para seleccionar los elementos que se representen o no, si se colorean los átomos por elemento o por cadena, el color de fondo, mostrar u ocultar el indicador de FPS, y aplicar rotación automática a las moléculas.

· Detalles de la estructura. Que enseña información obtenida al leer el fichero (átomos, modelos y conexiones indicadas y procesadas), pudiendo comprobar alguno de los datos ya mencionados.

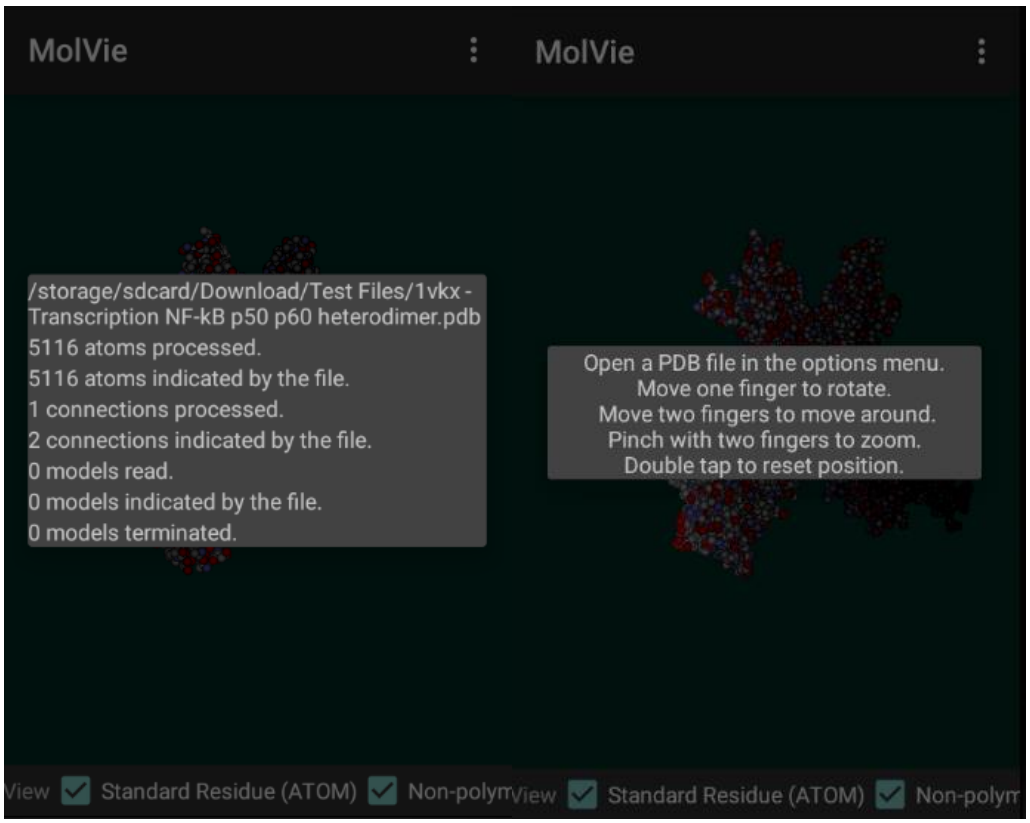
· Diálogo de ayuda. Que muestra los controles de la aplicación. Para recordar cómo se maneja la aplicación.



15. Menú inicial y selección de ficheros.



16. Menú de configuración.



17. Detalles y menú de ayuda.

5. Conclusiones.

En este proyecto de TFM se ha podido llevar a cabo el diseño y desarrollo de una aplicación para el sistema operativo Android para el renderizado tridimensional de estructuras macromoleculares en el formato de *Protein Data Bank*, presentando una interfaz interactiva, intuitiva y, en general, fácil de usar. La aplicación consigue el propósito inicial de leer los ficheros de proteínas y ácidos nucleicos, representarlos generando imágenes 3D con los datos de estos archivos y ofrecer al usuario los controles para manipular el visionado de éstas, garantizando una buena experiencia al usar la App.

Analizando todas las funcionalidades que se han implementado, el aspecto final y el rendimiento que ofrece la aplicación, se puede concluir que se han obtenido unos resultados bastante satisfactorios. Esto permite afirmar que se tiene un producto básico para ayudar en las investigaciones con moléculas en entornos de trabajo y aprendizaje con las mismas.

Este proyecto ha sido multidisciplinar, abarcando diversas disciplinas como son los gráficos 3D, la programación en Android, interacción con dispositivos multitáctiles, biología estructural y el álgebra geométrica. De todo esto, el alumno ha podido tanto aportar todos los conocimientos previos y capacidades de las que disponía, como adquirir nueva experiencia y enriquecerse profesionalmente, por lo que el TFM ha sido de gran utilidad para trabajar y aprender de igual manera.

Aunque una de las ideas principales era emplear la información de los enlaces de átomos en el dibujado, ofreciendo algún modelo de renderizado adicional (p. ej. *ball-and-stick*), no hubo tiempo para incluir la funcionalidad, pero al menos se consiguió desarrollar una aplicación capaz de representar los átomos de las moléculas en el modelo de *space-filling*, dejando esta característica para futuras mejoras.

Algunas de las otras posibles futuras mejoras, por mencionar algunas, incluyen añadir más modelos de representación tridimensional (ball-and-stick, sticks, ribbon diagram, etc), añadir más parámetros de coloreado y filtrado del dibujado, desarrollar una versión de la App para Windows Phone y iPhone OS, ofrecer la posibilidad de representar más de un fichero de moléculas simultáneamente, un buscador en línea de ficheros en las principales bases de datos, aceptar otros formatos de ficheros, edición de las propias estructuras moleculares, mejoras visuales de la interfaz, opción de guardar la representación actual como una imagen en el dispositivo para compartirla y optimización del código para mejorar su rendimiento.

6. Glosario.

· PDB: Protein Data Bank (Banco de Datos de Proteínas), hace referencia tanto al formato de fichero que contiene información estructural de proteínas y ácidos nucleicos, como al repositorio en donde se almacenan y comparten globalmente estos ficheros.

· Parser: Analizador sintáctico o gramatical, encargado de analizad una cadena de caracteres textuales, transformándola, según unas reglas, a una estructura de datos derivada.

· Shader: Sombreador, programa informático que se encarga de una etapa de la renderización. Puede ser compilado independientemente.

· Renderizado: Render, renderizador, renderización, se refiere al proceso automatizado de generar imágenes digitales, tanto en 2D como en 3D, mediante programación informática.

· OpenGL ES: Open Graphics Library for Embedded Systems, API multiplataforma para la programación de aplicaciones gráficas 2D y 3D.

· Space-filling model: Modelo de espacio lleno, también llamado modelo *calotte*, es un modelo de representación tridimensional de moléculas donde los átomos se dibujan como esferas separados unos de otros.

· Depuración: Del inglés *debugging*, se refiere a la búsqueda de errores en programas informáticos y la corrección de éstos. Un depurador, o *debugger*, se encarga de probar un programa con este objetivo.

· Java: Lenguaje de programación de propósito general orientado a objetos. Se ejecuta sobre una máquina virtual para evitar el tener que generar versiones compiladas para distintas arquitecturas.

· XML: Lenguaje de marcado para generar documentos y diseños de forma que tanto humanos como máquinas puedan leerlos fácilmente.

· C/C++: C consiste en un lenguaje de programación de bajo nivel que adapta de forma eficiente el código al lenguaje ensamblador del sistema para ofrecer el máximo rendimiento posible. C++, lenguaje de programación basado en C, pero elevando el nivel de abstracción y añadiendo orientación a objetos.

· JNI: Java Native Interface, interfaz nativa de java que permite que el programa, ejecutándose en la máquina virtual, invoque funciones nativas como las programadas en C o C++.

· NDK: Native Development Kit, kit de desarrollo nativo, que permite a Android incluir código y librerías programadas en C y C++ a las aplicaciones programadas en Java.

7. Bibliografía.

1. www.wwpdb.org – Web oficial del Worldwide PDB.

2. www.wwpdb.org/documentation/file-format – Documentación del formato PDB.

3. https://developer.android.com – Web oficial de Android para desarrollo.

4. www.expasy.org/spdbv – Página del software DeepView.

5. SWISS-MODEL and the Swiss-PdbViewer: An environment for comparative protein modeling. – Guex, N. and Peitsch, M.C. (1997) – Electrophoresis 18, 2714-2723.

6. es.wikipedia.org/wiki/Programación_modular – Programación modular.

7. es.wikipedia.org/wiki/Patrón_de_diseño – Patrón de diseño.

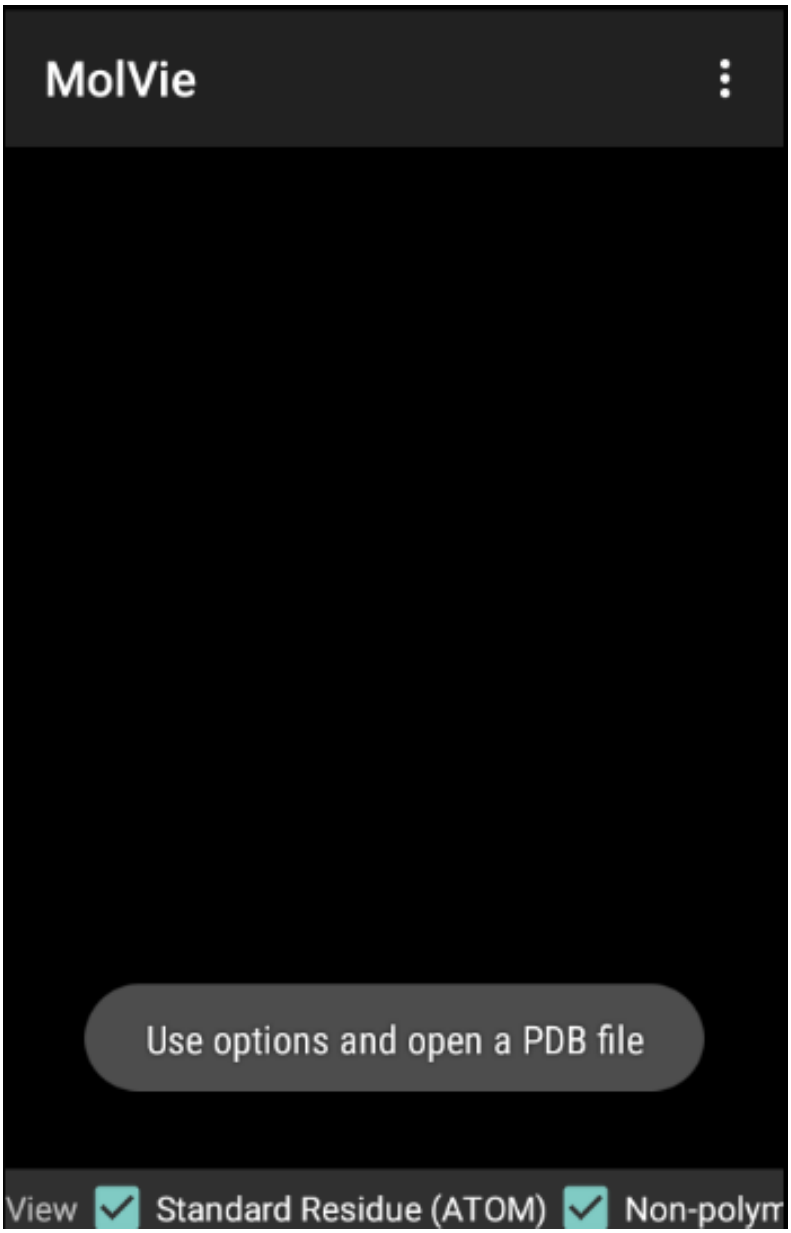
8. ftp.wwpdb.org/pub/pdb/doc/format_descriptions/Format_v33_Letter.pdf – Descripción del formato PDB.

9. www.khronos.org/registry/OpenGL/specs/es/3.0/GLSL_ES_Specification_3.00.pdf – Especificación del lenguaje de OpenGL.

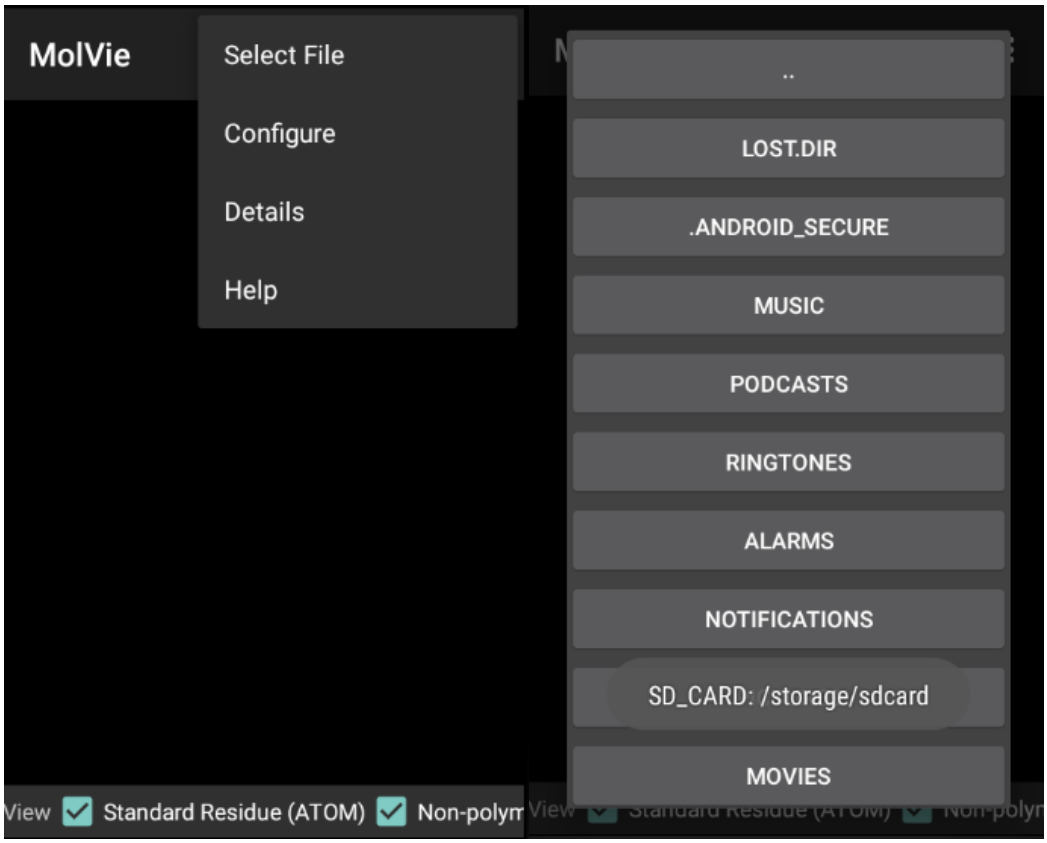
8. Anexos.

Anexo I. Manual de uso de la aplicación.

En la siguiente imagen se puede observar la aplicación en el momento inicial de su ejecución, mostrando el entorno vacío (aun sin ninguna molécula seleccionada para dibujar) y con un mensaje indicando que se acceda al menú de opciones para acceder al selector de ficheros. La aplicación está disponible tanto en inglés, como en español y en catalán. Según el idioma por defecto del dispositivo, la App ofrecerá todos los textos en el idioma adecuado. En caso de tener el dispositivo en otro idioma, la aplicación utiliza por defecto el idioma inglés.



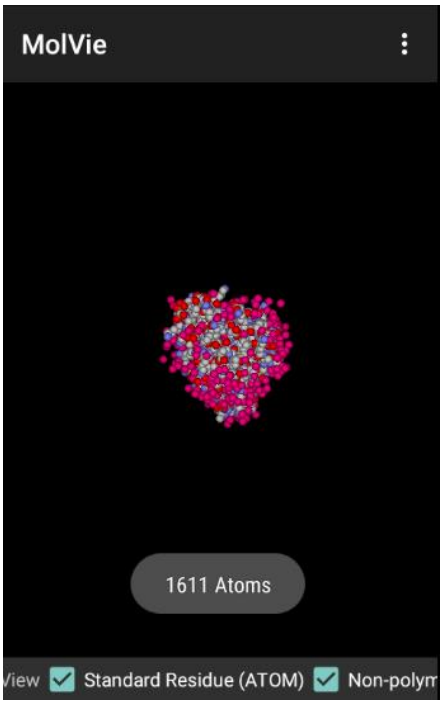
El siguiente paso consiste en abrir este menú y pulsar la opción del selector de ficheros.



Como se puede ver en la imagen anterior, al abrir el selector, éste accede a la raíz del almacenamiento externo del dispositivo y muestra una lista con los directorios y ficheros que se encuentran en esta ruta. A su vez, muestra la ruta de este almacenamiento ofrecido por el dispositivo, para que el usuario lo pueda verificar. Nótese que esta ruta al almacenamiento externo no es necesariamente la memoria extraíble o *SD Card*, podría ser alguna memoria integrada, o en caso de aceptar múltiples tarjetas extraíbles, solo una de estas.

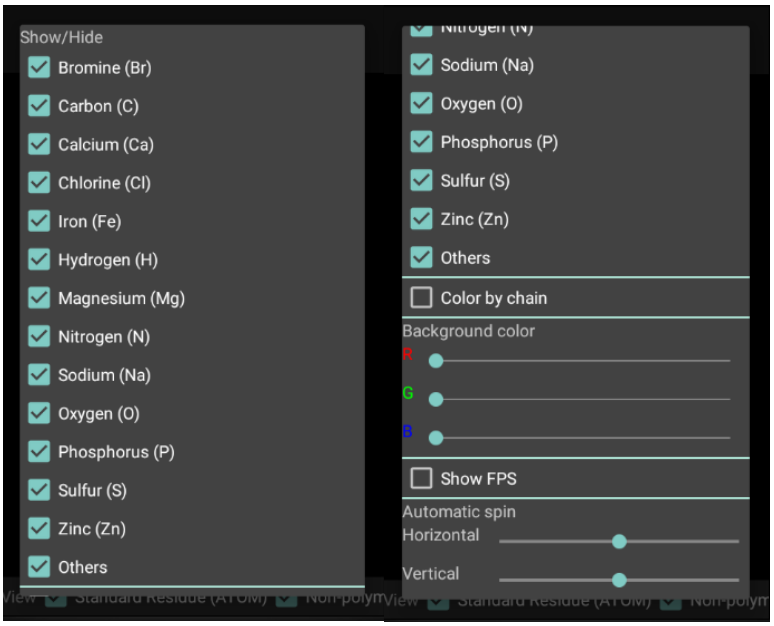
Para encontrar el fichero que se desea abrir, se puede navegar por el selector de ficheros, pulsando sobre las carpetas para abrirlas y mostrar su contenido, o sobre los ficheros para abrirlos por el lector de la App. Si la lista de ficheros y directorios es superior a la cantidad mostrada, se puede deslizar un dedo sobre este menú para desplazar el contenido oculto y acceder al resto de archivos y carpetas. En caso de querer volver al directorio anterior. Se puede pulsar sobre la carpeta nombrada “..” para volver atrás. En caso de que la carpeta esté vacía, no se abrirá y se mostrará una notificación, evitando tener que acceder a esta y volver atrás.

Una vez seleccionado un fichero, se pasa a su lectura. Si este no contiene información estructural atómica o no se trata de un fichero PDB, se mostrará otra notificación. En el caso de ser un fichero PDB válido y con información, la aplicación procederá a leerlo y al terminar mostrará el renderizado 3D inicial y notificando los átomos totales que fueron leídos.



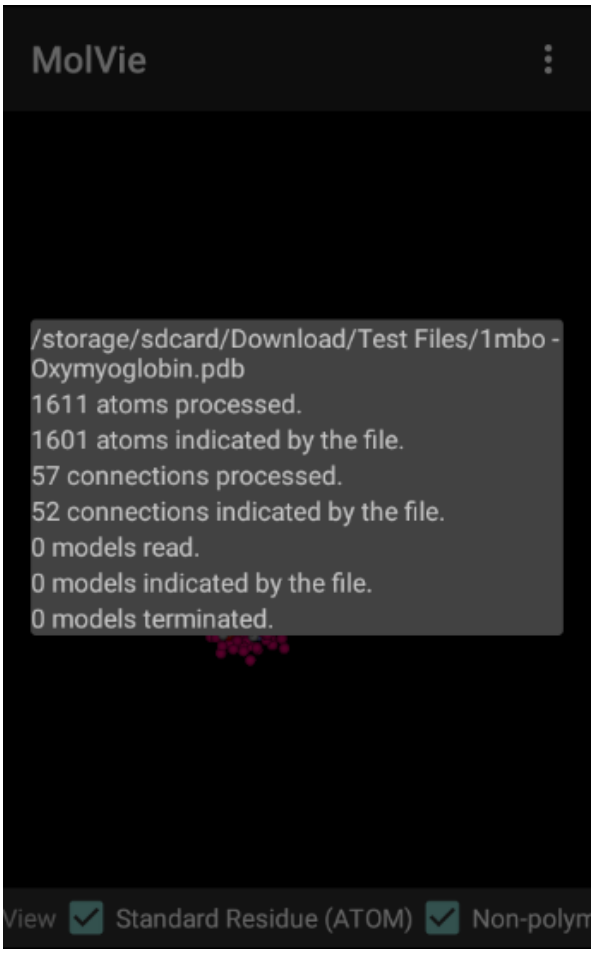
Una vez representando la molécula, se pueden usar los controles que usualmente se emplean en otros entornos de 3D. Al deslizar un solo dedo sobre la pantalla, la molécula rotará en los dos ejes posibles. Si se deslizan dos dedos simultáneamente, se moverá la molécula sobre la pantalla. Si se pellizca, acercando o separando dos dedos sobre la pantalla, se manipulará el tamaño con el que se representa la estructura. En caso de querer volver a la posición, rotación y escala inicial de la representación, con tan solo dar dos toques rápidos en la pantalla, ésta se resetea. Estos controles son recordados en el menú de ayuda por si hace falta.

MolVie también ofrece la posibilidad de cambiar varios aspectos del renderizado de las moléculas, así como del interfaz. Accediendo al menú de configuración se pueden ver los siguientes ajustes:



Como opciones que pueden ajustarse se puede elegir una serie de elementos químicos de los átomos que se mostrarán u ocultarán según se activen o desactiven sus casillas pertinentes en este menú. A esto le sigue la casilla de coloreado por cadena, si se deja desactivada, se colorean los átomos según su elemento y en caso de activarla, se colorean según la cadena a la que pertenecen. La siguiente opción es la de cambiar el color de fondo, pudiendo ajustar con los distintos deslizadores el nivel de rojo, verde y azul del fondo, para cambiarlo al color que se desee. Después, se encuentra la opción de mostrar el indicador de fotogramas por segundo que la aplicación dibuja. Y finalmente, se encuentran los ajustes de rotación automática, con dos deslizadores que permiten elegir la dirección y velocidad en cada eje para que la molécula rote automáticamente sin tener que interactuar con la pantalla.

En último lugar, el diálogo de detalles, accesible desde las opciones. Permite ver información extraída del fichero, para comprobación del fichero.



En la imagen anterior, se ha accedido al panel de detalles. La primera entrada contiene la ruta completa, junto con el nombre del fichero leído que se representa actualmente. A esto le siguen el número de átomos que ha leído la aplicación y el número de átomos que se indicaron en el fichero (línea MASTER). Gracias a esto podemos detectar una errata en el fichero, en este caso, se han procesado 1611 átomos, pero el fichero solo indicaba 1601. Analizar el fichero en cuestión, sí existen 1611 entradas de ATOM más HETATM, por lo que el recuento que provee el fichero es erróneo. Después de la información de átomos,

vienen las de conexiones, en este caso, que ambos números sean distintos no significa error en el fichero, puesto que el número de conexiones procesadas, son todos los enlaces entre átomos sin duplicados, mientras el número que ofrece la línea MASTER solo cuenta el número de líneas CONECT existentes, que en este caso es correcto. Además, también se muestra el número de modelos que se han contado, el número de modelos existentes que indica el fichero, y el número de indicaciones de final de modelo incluidos.

Anexo II. *Shaders* empleados.

Sombreador de Vértices. Usado para la etapa de geometría de OpenGL.

```
attribute vec4 position; //Atributo para la posición de los vértices.
uniform vec4 translation; //Traslación de la estructura en pantalla.
uniform vec4 scale; //Escala del tamaño del átomo en pantalla.
uniform vec4 color; //Color del átomo.
uniform vec3 dimrat; //Dimensiones de la pantalla en x e y, y ratio en z.
uniform float dist; //Distancia desde la pantalla (profundidad).
uniform mat4 rotationX; //Matriz de rotación en el eje X.
uniform mat4 rotationY; //Matriz de rotación en el eje Y.
varying vec4 fColor; //Variable “color” pasada al fragment shader.
varying vec4 fScale; //Variable “scale” pasada al fragment shader.
varying vec4 fTranslation; //”translation” pasado al fragment shader.
varying vec3 fDimrat; //”dimrat” pasado al fragment shader.
varying float fDist; //”dist” pasado al fragment shader.

void main(){ //Función principal para el procesado de vértices.
    gl_Position = position*scale*vec4(1.0, dimrat.z, 1.0, 1.0) +
translation*rotationX*rotationY; //Se aplican las coordenadas.
    fColor = color; //Se pasan los valores al fragment shader:
    fScale = scale;
    fTranslation = translation*rotationX*rotationY; //Posición final
    fDimrat = dimrat;
    fDist = clamp(dist, 0.0, 1.0);
}
```

18. Sombreador de vértices.

Sombreador de Fragmentos. Usado para la etapa de rasterización de OpenGL.

```
precision mediump float; //El Fragment Shader ha de especificar esto.
varying lowp vec4 fColor; //Variable para aplicar el color.
varying lowp vec4 fScale; //Variable para aplicar la escala.
varying lowp vec4 fTranslation; //Variable para aplicar la translación.
varying lowp vec3 fDimrat; //Variable para aplicar dimensión y ratio.
varying lowp float fDist; //Variable para iluminar según la distancia.
void main(){ //Función principal para el procesado de fragmentos.
 vec2 p = 2.0*( (gl_FragCoord.xy) / fDimrat.xy ) - 1.0 - fTranslation.xy;
 float r = sqrt(dot(p/vec2(1.0, fDimrat.z),
                    p/vec2(1.0,fDimrat.z)))/fScale.x;
 if(r < 1.0){ //Radio inferior a 1, se dibuja dentro del fragmento.
  float f = 1.0 - sqrt(1.0 - r*r);
  if(f > 0.125){
   discard; //No dibujar el píxel.
  }
  gl_FragColor = clamp(fDist*fColor - clamp(5.0*f*fColor, 0.0, 1.0), 0.0,
                       1.0);//Se aplica color e iluminación, falsa esfera
  gl_FragColor.a = 1.0; //Píxel opaco
  if(f > 0.75){ //Si el radio es mayor al de un 25% de iluminado
   gl_FragColor.a = 0.0; //Píxel transparente, se dibuja el círculo.
  }
 }else{ //Píxeles fuera del fragmento, no se dibujan
  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
 }
}
```

19. Sombreador de fragmentos.