Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

UNIVERSIDAD DE JAÉN
Nombre del Centro

Trabajo Fin de Grado

DESARROLLO DE UN PROTOTIPO DE APLICACIÓN PARA LA OBTENCIÓN DEL PERFIL DE REVOLUCIÓN DE PIEZAS DE CERÁMICA A PARTIR DE IMÁGENES.

Alumno: David Cruz Rubio

Tutor: Manuel José Lucena López
José Manuel Fuertes García

Dpto: Ingeniería Informática

Septiembre, 2017

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Informática

Don Manuel José Lucena López y José Manuel Fuertes García , tutores del Proyecto Fin de Carrera titulado: Desarrollo de un prototipo de aplicación para la obtención del perfil de revolución de piezas de cerámica a partir de imágenes, que presenta David Cruz Rubio, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, Septiembre de 2017

El alumno:

Los tutores:

Fdo: David Cruz Rubio

Fdo: Don Manuel José Lucena López

Fdo: Don José Manuel Fuertes García

Índice

1. INTRODUCCIÓN

La fotogrametría digital es un concepto que hoy en día convive con todos nosotros; agronomía, arquitectura, arqueología, estos son algunos de los campos en los que esta técnica tiene mayor repercusión y utilidad.

En este caso, partimos de la idea básica de obtener un modelo en tres dimensiones a partir de una sola imagen de una pieza de cerámica, la cual tiene que cumplir una serie de características básicas de simetría y geometría.

Para todo este proceso serán usadas una serie de técnicas, tanto para el análisis de las imágenes, como para la generación de los modelos en tres dimensiones, las cuales se detallarán a continuación.

A continuación, se muestran unas piezas de cerámica obtenidas en la excavación arqueológica de Atzúbia:

Ilustración 1.1

1.1. Motivación

El mundo avanza a pasos agigantados; constantemente se están generando necesidades en los diversos campos que el ser humano está encargado de estudiar, ya sea en la arquitectura, en la arqueología o en la medicina. Es por esto por lo que serán necesarias una serie de herramientas, físicas, hardware o software, para cubrir dichas necesidades.

El propósito de este trabajo es servir de ayuda al personal dedicado a la arqueología a la hora de generar imágenes en tres dimensiones de piezas de cerámica de manera sencilla rápida y eficiente.

Las técnicas, que a continuación se expondrán, tienen también un mínimo inconveniente, y es que abogan por simpleza y rapidez a cambio de fiabilidad sobre el modelo generado tridimensionalmente.

1.2. Propósito

Uno de los grandes propósitos, a parte del aprendizaje y la comprobación del conocimiento aprendido a lo largo de estos años de estudio, es el de crear una aplicación usable y accesible para cualquier usuario.

Y en cuanto a especificidad del proyecto, hablaríamos de la capacidad de poder generar modelos en tres dimensiones a partir de cualquier sólido de revolución seleccionado por un usuario cualquiera, mediante un proceso interactivo.

Se adjunta a continuación una imagen representativa del proceso interactivo para poder obtener el perfil del sólido en cuestión, la cual ha sido extraída de un documento en el cual se recoge información de un proceso similar al expuesto en este proyecto.

(Pernici, 2005)

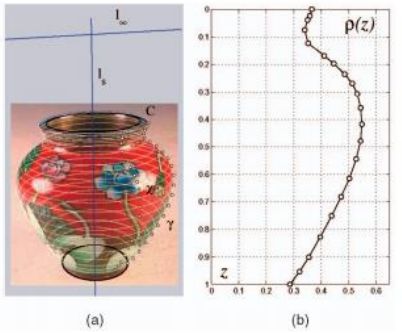


Ilustración 1.2

1.3. Objetivos

En este apartado se pretenden presentar los objetivos a cumplir a lo largo del desarrollo de este proyecto:

1.3.1. Generación de un sólido de revolución a partir de los puntos de su perfil

1.3.1.1. Subdivisión de polilíneas

1.3.1.2. Cálculo de vectores normales para un determinado perfil

1.3.1.3. Revolución del perfil alrededor de un eje

1.3.2. Interfaz gráfica de usuario

1.3.2.1. Implantación de los elementos de la interfaz

1.3.2.2. Escalado, espaciado y márgenes

1.3.3. Uso e implementación de técnicas para la obtención de los puntos de un perfil mediante el análisis de una imagen

1.3.3.1. Análisis de los componentes principales

1.3.3.2. Ajuste de las elipses principales

1.3.3.3. Generación de las elipses secundarias y puntos del perfil

1.4. Resultados esperados

Se espera poder recabar la información necesaria para poder realizar los procesos de análisis necesarios y así implementar el programa que se encargará de realizar todo el proceso descrito brevemente anteriormente.

Por lo que finalmente se espera disponer de una aplicación, la cual posea una interfaz gráfica intuitiva y accesible, mediante la cual podamos generar los

modelos en tres dimensiones de cualquier sólido de revolución de manera interactiva.

Se muestra a continuación un ejemplo del resultado esperado a partir de una imagen de un sólido de revolución cualquiera:

Ilustración 1.3

1.5. Planificación

Con respecto a la planificación del mismo, el trabajo estará dividido en tres partes o etapas, cada una correspondiente a uno de los objetivos principales del trabajo.

Una primera parte que estará centrada en la generación de un modelo en tres dimensiones a partir de los puntos de un perfil de un sólido de revolución. Esta se llevará a cabo durante el mes de octubre del año 2016 debido a que en la asignatura “Programación de Aplicaciones Gráficas” se estudiará la geometría de dichas figuras y los algoritmos pertinentes para poder generar dichas imágenes.

La segunda parte del proyecto se corresponderá con la implementación de la interfaz gráfica de la aplicación, lo cual se realizará durante los primeros meses del año 2017, correspondientes con el segundo cuatrimestre del curso académico. Durante estos meses es estudiarán las diversas alternativas disponibles para poder realizar una interfaz de usuario adecuada, así como la implementación de la misma.

Finalmente, la tercera etapa del desarrollo de la aplicación se corresponderá temporalmente con los meses posteriores a marzo, dándose paralelamente con la etapa anterior. Esto se debe a que la implementación de la interfaz gráfica va ligada a la fase de análisis de las imágenes para así obtener los puntos del perfil necesarios para generar los modelos en tres dimensiones.

Además de estas tres etapas bien diferenciadas se contará con otra etapa, la cual se desarrollará paralelamente a las anteriores y cuyo objetivo es generar el actual informe.

Como se puede apreciar en la *ilustración 1.4* se pretende haber implementado una versión “beta” de la aplicación para el mes de junio del año 2017. También podemos observar las etapas, descritas con anterioridad en dicha ilustración, de manera gráfica.

Finalmente, y como adición final al informe, tras haber realizado todo el desarrollo de la aplicación, destacar que la aplicación se perfiló y ajustó para que fuera más precisa durante el mes de Julio del año presente.

1.6. Estimación de costes

El proyecto contará con costes de carácter humano esencialmente, esto es horas de trabajo invertidas en su realización. Con respecto a los costes atribuidos al software se espera que sean nulos. Por otro lado, y hablando del hardware, se requerirá de una computadora, tanto para la implementación de la aplicación como para su uso.

Volviendo a los costes temporales, se espera invertir unas ciento treinta horas en la realización total del proyecto; por un lado, para la aplicación se espera invertir entre setenta y ochenta horas para su completa realización, para la redacción del actual documento se prevé una inversión de unos cincuentas horas aproximadamente.

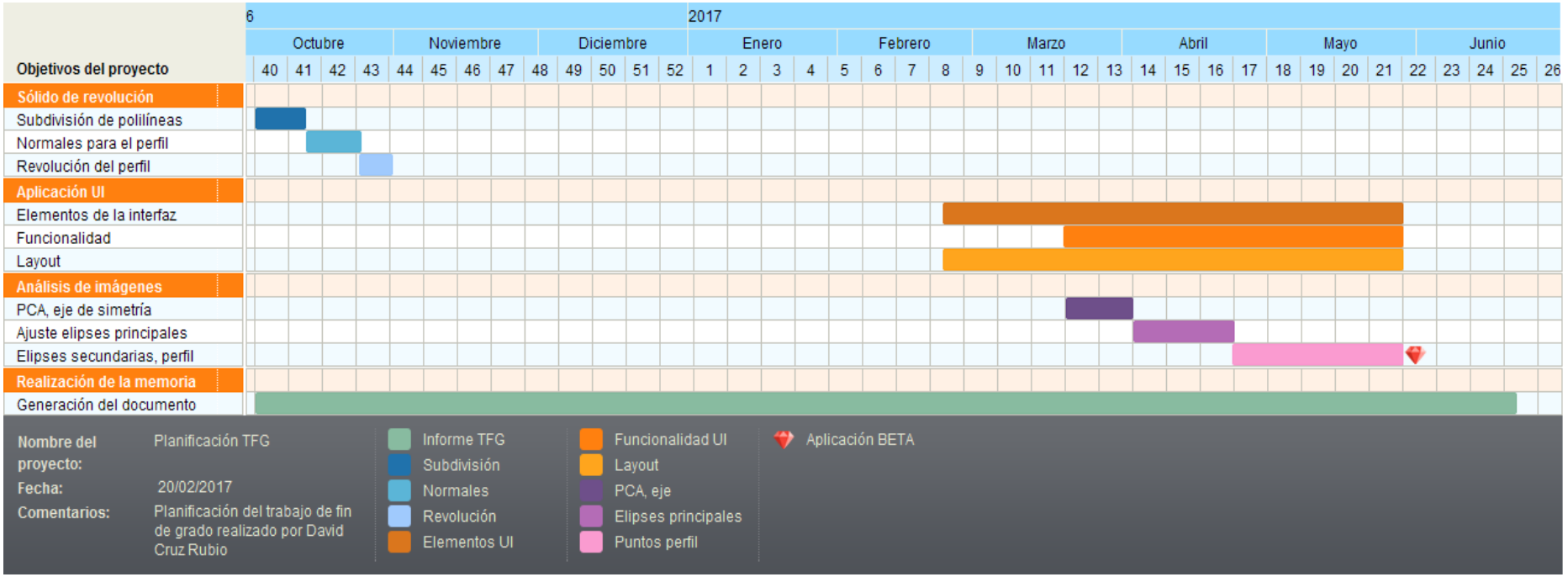


Ilustración 1.4

1.7. Estructura del proyecto

En este apartado se espera detallar un poco la propia estructura que tendrá el documento relacionado al proyecto.

1.7.1. Introducción

En este punto se prevé explicar el contexto del proyecto en cuestión, de esta forma se hará un estudio de los costes esperados del proyecto, así como su planificación y objetivos.

1.7.2. Análisis previo del proyecto

Por otro lado, en este apartado se pretende presentar la información y el análisis de la misma para el desarrollo completo del proyecto, desde la generación del modelo hasta el análisis de las imágenes, de esta forma, se pretende generar una idea previa de lo que se desarrollará posteriormente.

1.7.3. Análisis e implementación

Aquí se expresarán los aspectos de ingeniería del software utilizados para el desarrollo de la aplicación, así como los detalles de la implementación del proyecto.

1.7.4. Manual de usuario y de instalación

En este apartado se expondrá un pequeño manual de uso de la aplicación, así como una guía para poder instalar, compilar y ejecutar la aplicación.

1.7.5. Conclusiones

Para terminar, en este apartado se espera expresar las conclusiones del trabajo realizado, así como una valoración final y personal del mismo.

1.7.6. Bibliografía

Apartado final en el que se enumerarán todos los elementos bibliográficos usados durante el desarrollo del proyecto.

2. ANÁLISIS PREVIO DEL PROYECTO

En este apartado se pretende realizar una serie de reflexiones iniciales antes de dar lugar al propio desarrollo del proyecto; de esta forma se pretenden resolver ciertas dudas sobre qué lenguajes o entornos se van a utilizar, así como aclarar conceptos relacionados con apartados técnicos del proyecto.

2.1. ¿Qué lenguaje de programación utilizar?

De entre todos los lenguajes de programación existentes dudé entre el uso de dos de ellos: Python y C++.

Por una parte, Python presenta muchas facilidades a la hora de usar librerías y es muy útil para el análisis de imágenes, sin embargo, debido a que C++ es un lenguaje polivalente en dichos sentidos, ofreciendo además flexibilidad y potencia a la hora de programar aplicaciones gráficas, se ha optado por el uso de este último para el desarrollo total de la aplicación.

Otros motivos han sido: comodidad y potencia del lenguaje, así como familiaridad con el mismo. Lenguaje usando durante la asignatura “Programación de Aplicaciones Gráficas”.

(Stroustrup, 1980)

Ilustración 2.1

2.2. ¿Qué entorno de desarrollo utilizar?

Llegados a este punto, y sabiendo qué lenguaje de programación se va a utilizar, se duda, también, sobre dos entornos de desarrollo, “Microsoft Visual Studio” o “Qt Creator”.

Teniendo en cuenta la naturaleza gráfica de la aplicación, y del requerimiento de poseer una interfaz gráfica con la que establecer comunicación con el usuario final, me he decantado por el uso de “Qt Creator”. Su facilidad a la hora de desarrollar una interfaz gráfica ha sido el motivo principal de la elección, contando también otros aspectos como la sencillez de la interfaz del propio “IDE” o su accesibilidad.

(The Qt Company, 2017)

2.3. ¿Qué librerías serán necesarias durante el desarrollo?

Este apartado es de vital importancia para el desarrollo de nuestra aplicación. Si analizamos la naturaleza del propio proyecto, podemos observar que será necesaria algún tipo de librería de programación gráfica, debido a sencillez, familiaridad y gratuidad se ha escogido “OpenGL” en su “perfil Legacy”, ya que es también la librería utilizada en las asignaturas de programación gráfica.

(Khronos Group, 2017)

Con respecto al apartado de análisis de imágenes, me he decantado por el uso de “OpenCV”, librería gratuita y muy potente para el análisis de imágenes, uso de filtros, algoritmos de detección de formas, detección de bordes y un largo etcétera de cualidades de esta librería.

(Intel, 1999)

“Qt”, además de hacer uso del entorno de programación con el mismo nombre, se hará uso de la multitud de utilidades, estructuras y clases que

presenta esta librería; principalmente para el manejo de imágenes, textos y elementos de la interfaz gráfica.

Finalmente, se usará la librería “GLM” (OpenGL Mathematics) para el aspecto matemático del proyecto; ya sea manejo de matrices, vectores u operaciones matemáticas entre ellos.

(G-Truc Creation, 2005)

2.4. ¿Qué metodología de desarrollo utilizar?

De entre todas las metodologías tradicionales se ha elegido una en concreto, la metodología o desarrollo en cascada. Esto es debido a que se espera y pretende realizar cada una de las etapas del proyecto en el momento que se termine la anterior.

Esencialmente se realizará una etapa inicial donde se establecerán los requisitos del proyecto, material, costes y recursos. Finalizada esta etapa pasaremos al diseño, donde se realizarán los correspondientes análisis y diseños de clases necesarias para la implementación del proyecto. Durante la implementación se codificará todo lo necesario para el correcto funcionamiento de la aplicación.

Para finalmente, validar la aplicación implementada, tanto por parte del programador, como por parte de los tutores. Se realizará, también, una pequeña etapa de mantenimiento tras todo esto para pulir la aplicación y corregir los problemas que vayan surgiendo con el uso.

(Royce, 1970)

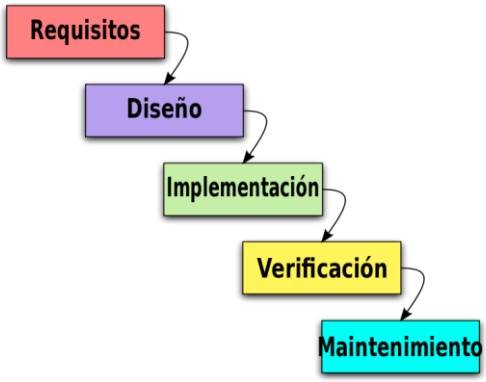


Ilustración 2.2

2.5. ¿Qué es un sólido de revolución?

Entrando en detalles con aspecto matemático, un sólido de revolución es un cuerpo, o volumen, que se puede obtener rotando una superficie plana alrededor de una recta que se contiene en su mismo plano.

(Weisstein, 1999-2017)

Es, debido a la propia naturaleza del proyecto, una de las partes fundamentales que deben ser estudiadas y analizadas detenidamente, para su posterior desarrollo e implementación.

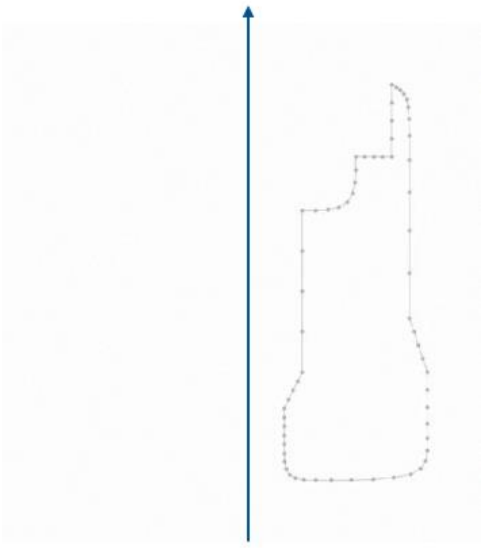


Ilustración 2.3

El proceso consistirá en, a partir de unos puntos iniciales, los cuales conforman el perfil de revolución, obtener los vectores normales a dichos puntos, y una vez obtengamos toda la información necesaria sobre los vértices, rotarlos alrededor de una recta o eje, de esta forma se obtendrá el volumen de revolución asociado al perfil.

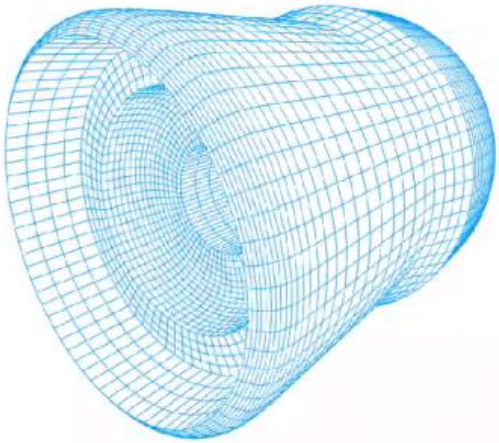


Ilustración 2.4

El estudio, análisis y desarrollo de los aspectos matemáticos sobre este concepto se adquirirán durante el curso de la asignatura “Programación de Aplicaciones Gráficas”, asignatura optativa del cuarto curso de la actual titulación.

(Conde, 2017)

2.6. ¿En qué consiste el proceso de análisis de la imagen?

El proceso de análisis de las imágenes se establecerá justo antes de todo el proceso de generación del sólido de revolución. De esta forma, y grosso modo, se obtendrán, a partir de la imagen, una serie de puntos los cuales conformarán el perfil de revolución de nuestro sólido.

En un principio, se usarán las librerías necesarias, y los métodos requeridos, para obtener el conjunto de píxeles los cuales conforman el jarrón en

una imagen determinada, para ello habrá que establecer una serie de requisitos previos, como que el fondo sea blanco, o, que se intenten evitar las sombras arrojadas por el sólido, a toda costa.

Una vez encontrado el jarrón y los píxeles que lo conforman, se encontrará el eje de simetría del contorno, siendo este, el eje a partir del cual se rotarán los puntos para así obtener la figura en una última instancia.

Tras esto, se ajustarán, de manera interactiva y por parte del usuario, dos elipses en la imagen, una en la circunferencia, o tapa superior, y otra en la inferior. Estas elipses tendrán tres parámetros ajustables, su posición en el eje, su radio horizontal y el ángulo o inclinación de las mismas.

Será, mediante el ajuste de dichas elipses, como se obtendrán y ajustarán una serie de elipses intermedias cuyo radio horizontal determinará los puntos del perfil de revolución. Este proceso se explicará con más detalle posteriormente en este mismo informe.

(Pernici, 2005)

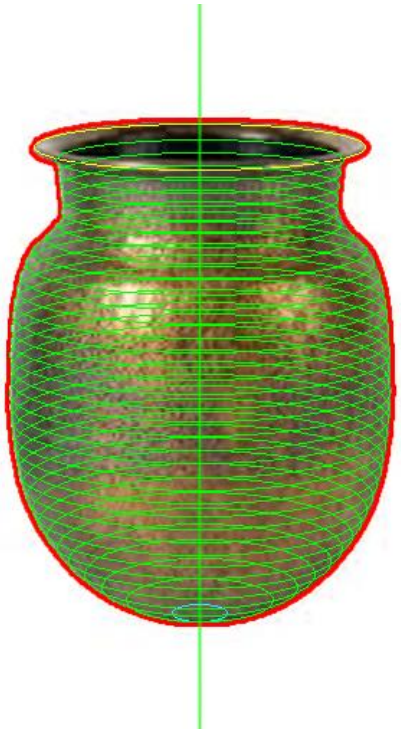


Ilustración 2.5

2.7. ¿Cómo será la interfaz gráfica de la aplicación?

Paralelamente al estudio y desarrollo algorítmico de lo anteriormente nombrado, se implementará una interfaz gráfica mediante la cual el usuario podrá realizar todo el proceso; generando finalmente una imagen en tres dimensiones de un sólido de revolución dado.

Se adjunta a continuación un dibujo donde se plasma la forma e idea de la interfaz gráfica:

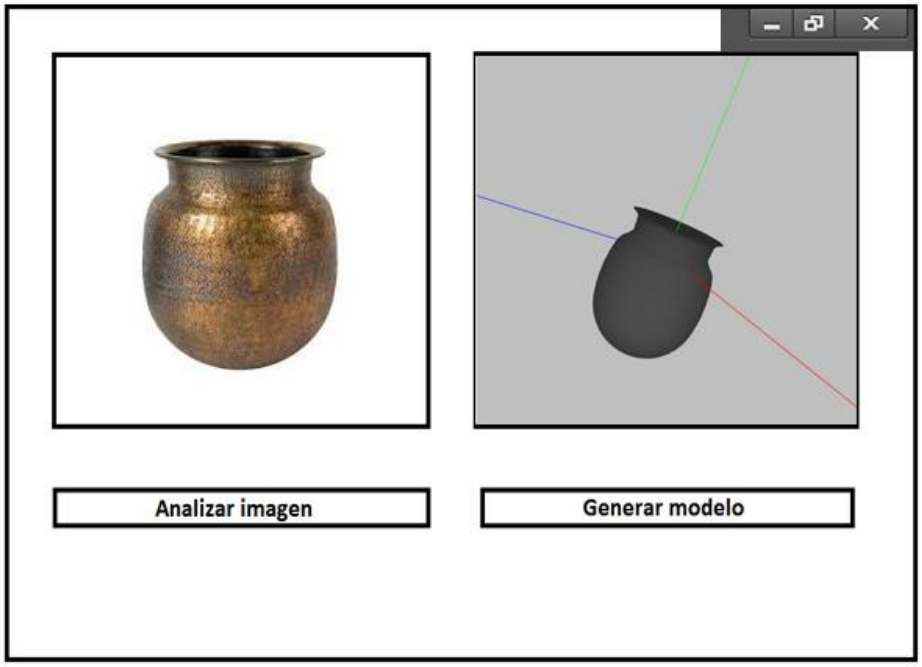


Ilustración 2.6

Como se puede observar en el boceto que se acaba de adjuntar, la aplicación gráfica tendrá dos partes bien diferenciadas, la parte de análisis de la imagen, situada a la izquierda, en la cual se realizará todo el proceso necesario para la obtención de los puntos del perfil necesarios para generar el modelo, el cual aparecerá en la parte derecha de la aplicación una vez finalizado todo el proceso.

3. ANÁLISIS E IMPLEMENTACIÓN

En este apartado se procederá a explicar todo el proceso de desarrollo del proyecto. Como se comentó con anterioridad, se usará una metodología tradicional, basada en cascada, correspondiendo las distintas etapas de esta con las partes de este apartado. Empezando por la especificación de los requisitos, pasando por el análisis y la implementación, hasta acabar con la verificación y mantenimiento del mismo.

3.1. Requisitos del proyecto

Al comienzo de cualquier proyecto es obligatorio establecer una serie de requisitos que se verán realizados a lo largo de la vida del mismo. En este primer apartado se pretenden especificar todos los requisitos que este proyecto pone de manifiesto.

Uno de los requisitos fundamentales, siendo parte del propio título del proyecto, es el de realizar una aplicación. Esto conlleva una serie de requisitos implícitamente; la aplicación debe de ser usable y accesible para cualquier usuario, aunque se centrará en usuarios de la comunidad científica, y al ser el lenguaje establecido el castellano, se centrará en usuarios de habla hispana. También requerirá de una interfaz gráfica con la cual el usuario se comunicará e interactuará con la propia aplicación.

Para satisfacer estos requisitos o necesidades se usará un entorno de programación que permita acompañar a la aplicación con una interfaz gráfica. Es por esto que será usado “Qt Creator” para el desarrollo de la aplicación, entorno de programación muy útil y potente en cuanto al desarrollo de interfaces gráficas.

(The Qt Company, 2017)

Ilustración 3.1

Por otra parte, hay un requisito muy importante y fundamental para el correcto funcionamiento de la aplicación, y es el de ser capaces de generar un modelo en tres dimensiones a partir de los puntos de un perfil de revolución, lo cual conlleva también estudiar modelos de iluminación correctos para estos modelos; así como ser capaces de almacenar, estructurar y generar toda la geometría de una manera eficiente.

Para estos requisitos se estudiará de manera exhaustiva todo lo relacionado con sólidos de revolución: geometría, iluminación y algoritmos; así como su implementación con la famosa librería gráfica “OpenGL”. Esto se llevará a cabo durante el curso de la asignatura “Programación de Aplicaciones Gráficas”, como bien se nombra con anterioridad en este mismo informe.

(Khronos Group, 2017)

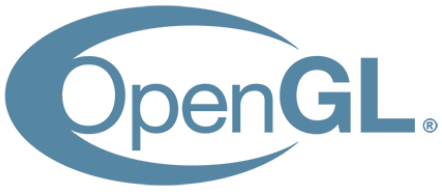


Ilustración 3.2

Finalmente, y no por ello menos importante, hay un conjunto de requisitos relacionados con la parte de análisis de cualquier imagen, tales como ser capaces de obtener y almacenar los píxeles representantes de un sólido de revolución en cualquier imagen dada. Así como ser capaz de analizar dichos píxeles y procesarlos para así obtener los puntos del perfil, los cuales se usarán posteriormente para generar el modelo, gracias a los algoritmos nombrados con anterioridad.

Para este apartado se hará uso del conocimiento y experiencia de los tutores en la materia. Por tanto, se hará uso de la librería “OpenCV”, librería muy útil y potente, capaz de aplicar distintos filtros y algoritmos de detección de formas a cualquier imagen.

(Intel, 1999)

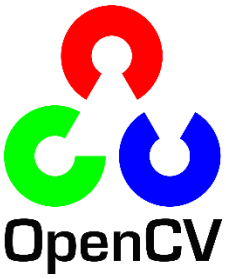


Ilustración 3.3

3.2. Análisis e Implementación de los objetivos

Por otra parte, en este segundo apartado, se pretende especificar toda la etapa de análisis del proyecto, desde la generación de los diagramas de clases pertinentes, los cuales se realizarán con la herramienta gratuita y online “Creately”, hasta su correspondiente implementación, usando como lenguaje para codificar C++.

(Cinergix Pty. Ltd, 2017)

3.2.1. Generación de un sólido de revolución a partir de los puntos de su perfil

Este es el primer objetivo que se marcaba a la hora de realizar este proyecto, como bien se ha nombrado con anterioridad, el estudio, análisis e implementación de técnicas relacionadas con él se dieron durante el mes de octubre de 2016, durante la asignatura “Programación de Aplicaciones Gráficas”.

Será el nombre de la asignatura el que dará lugar a una librería compuesta por varias clases encargadas de implementar todo el proceso de renderizado de

un sólido de revolución, desde algoritmos de generación de geometría hasta iluminación, “PAG”, correspondiéndose con las siglas de la asignatura.

(Conde, 2017)

3.2.1.1. Subdivisión de polilíneas

En esta primera parte se analizará y estudiará como generar curvas suaves a partir de una serie de puntos, los cuales representar el perfil de revolución. De esta forma obtendremos mayor cantidad de elementos geométricos, vértices en este caso, dotando además de un aspecto suave a la curva.

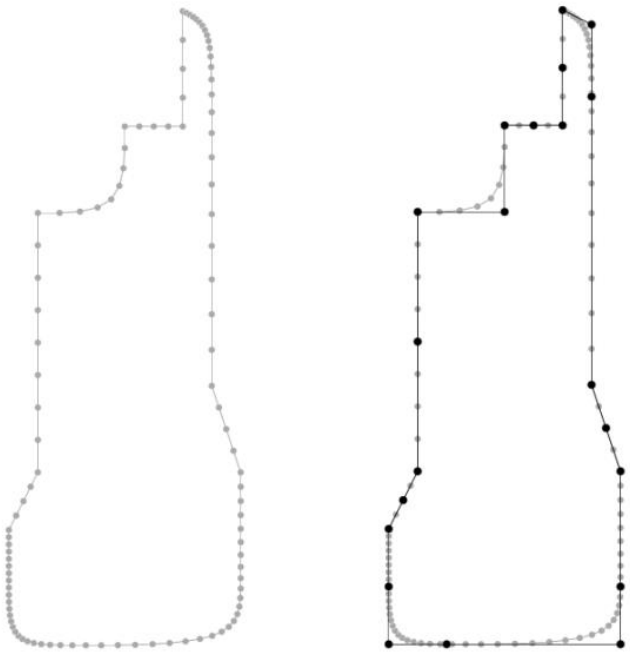


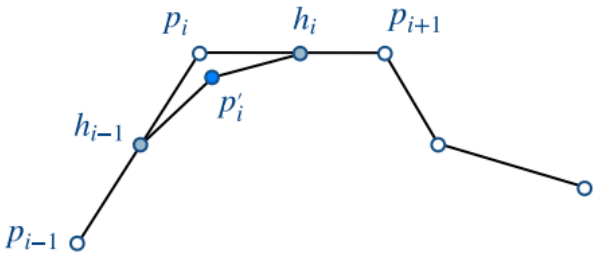
Ilustración 3.4

Como se puede observar en la *ilustración 3.4*, partiremos de un perfil con una cantidad limitada de puntos, “**n**”, y tras realizar un proceso algoritmo obtendremos una geometría más fiable y suave del perfil.

En cuanto al proceso, matemáticamente, se recorrerán los puntos del perfil en orden, empezando por el segundo de ellos, de tal forma que en cada iteración se usará la información referente al punto anterior del perfil, el actual y el

siguiente, llamados a partir de ahora $\mathbf{P_{i-1}}$, $\mathbf{P_i}$ y $\mathbf{P_{i+1}}$, donde **i**, representa el índice de cualquier punto del perfil, con el objetivo de generar otro punto que sustituya a $\mathbf{P_i}$ y que proporcione mayor suavidad en las curvas, $\mathbf{P'_i}$ Se comienza en el segundo punto del perfil debido a que si se empezara en el primero no habría $\mathbf{P_{i-1}}$ válido.

Usando dichos puntos, obtendremos otros dos puntos clave, $\mathbf{h_{i-1}}$ y $\mathbf{h_i}$, dichos puntos representarán la mitad de los segmentos o vectores [$\mathbf{P_{i-1}}$, $\mathbf{P_i}$] y [$\mathbf{P_i}$, $\mathbf{P_{i+1}}$] respectivamente; estos puntos, $\mathbf{h_{i-1}}$ y $\mathbf{h_i}$, al igual que $\mathbf{P'_i}$, se añadirán al nuevo perfil generado tras este proceso. Finalmente, gracias a estos dos puntos, y usando las fórmulas matemáticas que podemos observar en la *ilustración 3.5*, obtendremos los puntos deseados fácilmente.



$$h_{i-1} = \frac{p_{i-1} + p_i}{2} \qquad h_i = \frac{p_i + p_{i+1}}{2} \qquad p'_i = \frac{p_i}{2} + \frac{h_{i-1}}{4} + \frac{h_i}{4}$$

$$p'_i = \frac{3p_i}{4} + \frac{p_{i-1}}{8} + \frac{p_{i+1}}{8}$$

Ilustración 3.5

Como se puede intuir, este proceso se puede repetir "**k**" veces, para así obtener más y más vértices, y por tanto obtener una geometría más fiel. No es aconsejable repetir el proceso más de tres veces, ya que, a partir de dicho número, los resultados son cada vez menos apreciables y el coste de cómputo es elevado, esto es apreciable en la *ilustración 3.6*.

Ilustración 3.6

Como conclusión, podemos elaborar una pequeña regla. Y es que el número final de puntos del perfil será igual al doble de la cantidad de puntos que había originalmente, menos una unidad.

```
expectedNumberOfPoints = (2 * profile.count) - 1)
```

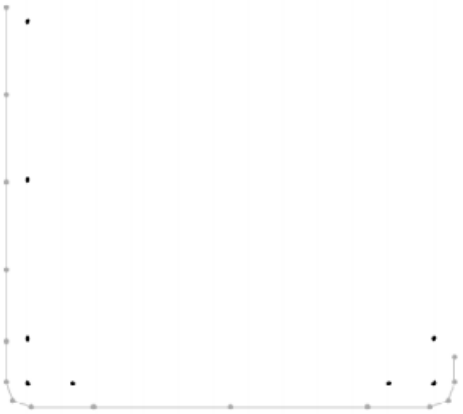


Ilustración 3.7

3.2.1.2. Cálculo de vectores normales para un determinado perfil

En este segundo punto se especifica cómo obtener para cada vértice, que se ha generado mediante la subdivisión de polilíneas, su propio vector normal. Este vector, propio de cada vértice, es necesario para realizar los cálculos de iluminación y visibilidad posteriormente; para ello será necesario que se recorran todos los puntos del perfil de manera ordenada y secuencial.

Como podemos ver en la siguiente *ilustración 3.8*, y a partir de ahora, llamaremos al vector normal al perfil de revolución en un punto **P_i**, **n_i**.

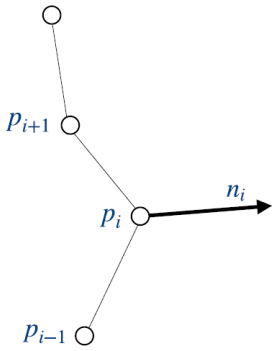


Ilustración 3.8

Dicho vector **n_i**, es perpendicular al perfil de revolución en dicho punto, además de ser unitario, esto es, su módulo tiene por valor una unidad. Realizar el cálculo de dicho vector se puede lograr de manera sencilla si existieran dos vectores perpendiculares a los segmentos, o vectores, [**P_{i-1}**, **P_i**] y [**P_i**, **P_{i+1}**], **ns_{i-1}** y **ns_i**, respectivamente; de tal forma que el cálculo del vector **n_i**, sería simplemente hacer una media entre dichos vectores. Esto se puede observar con mayor claridad en la imagen que se encuentra a continuación, *ilustración 3.9*.

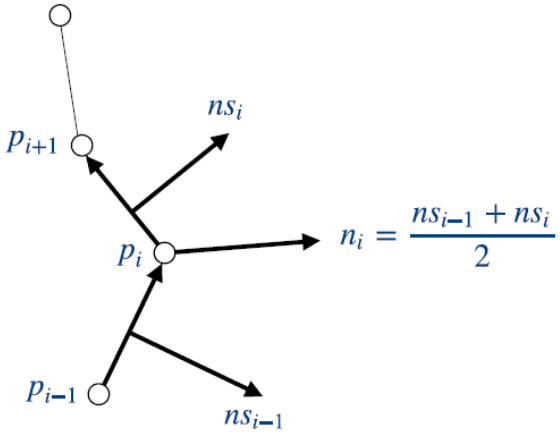


Ilustración 3.9

Sin embargo, realizar el cálculo de dichos vectores no es algo trivial, por ello, y con el objetivo de obtener el vector deseado, $\mathbf{n_i}$, se usará otro método. Este consiste en usar los segmentos, o vectores, anterior y posterior a dicho punto; una vez obtengamos el valor de dichos vectores, anteriormente nombrados como [$\mathbf{P_{i-1}}$, $\mathbf{P_i}$] y [$\mathbf{P_i}$, $\mathbf{P_{i+1}}$], tendremos que normalizarlos para que su módulo sea uno, obteniendo así los vectores $\mathbf{v_i}$ y $\mathbf{v_{i-1}}$. Como bien se conoce, el proceso de normalización se puede implementar con una simple operación; además, multitud de librerías de propósito matemático poseen su propia implementación de la función que se encarga de normalizar un vector.

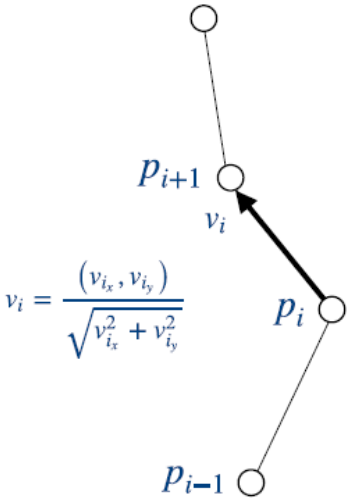


Ilustración 3.10

Finalmente, para el cálculo del vector $\mathbf{n_i}$, simplemente tendremos que rotar noventa grados en sentido horario los vectores que acabamos de calcular, $\mathbf{v_i}$ y $\mathbf{v_{i-1}}$, y proceder a realizar una media aritmética entre ellos para obtener el vector normal a la superficie en dicho punto. La operación que se encarga de realizar la rotación es trivial ya que los cálculos de seno y coseno del ángulo noventa son directos. Con lo que, en un vector de coordenadas ($\mathbf{V_{ix}}$, $\mathbf{V_{iy}}$), simplemente se intercambiarán de posición los valores de sus coordenadas y se cambiará el signo de la coordenada **y**, dando lugar al vector: ($\mathbf{V_{iy}}$, -$\mathbf{V_{ix}}$).

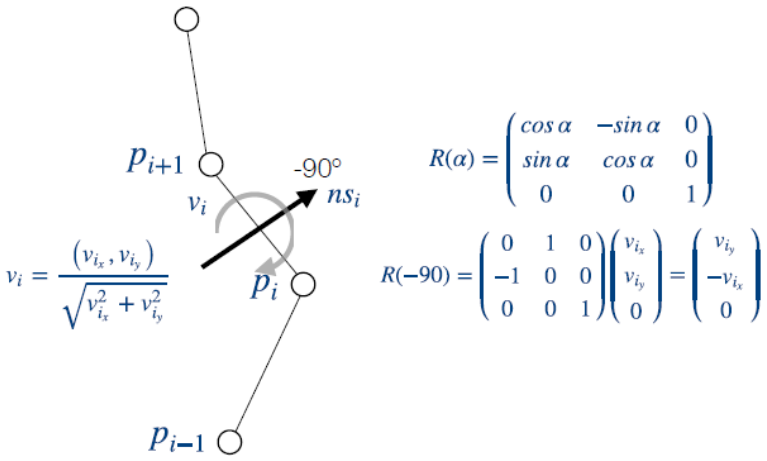


Ilustración 3.11

(Ortega, 1989-2006)

3.2.1.3. Revolución del perfil alrededor de un eje

Finalmente, aquí se pretende explicar la forma de generar toda la geometría necesaria para que la GPU de cualquier ordenador pueda interpretarla y generar las imágenes del sólido de forma tridimensional. Para ello partiremos de los puntos que se generan tras subdividir el perfil, como bien se explica en el *apartado 3.2.1.1*.

Para llevar a cabo este apartado, primero hay que determinar el número de porciones o rodajas con las que contará el sólido de revolución, "**n**", es decir, el número de veces que habrá que rotar un punto alrededor del eje. Una vez comprendido esto, habrá que recorrer todos los puntos pertenecientes al perfil e ir rotándolos de manera incremental, tantas veces como determine el número de porciones. Partiremos por tanto de suponer que nuestro perfil se encuentra contenido en el plano vertical "*XY*". Por lo que la rotación, para obtener un determinado **P'** a partir de **P**, se producirá de manera trivial y siguiendo las fórmulas que se pueden observar en la *ilustración 3.12*.

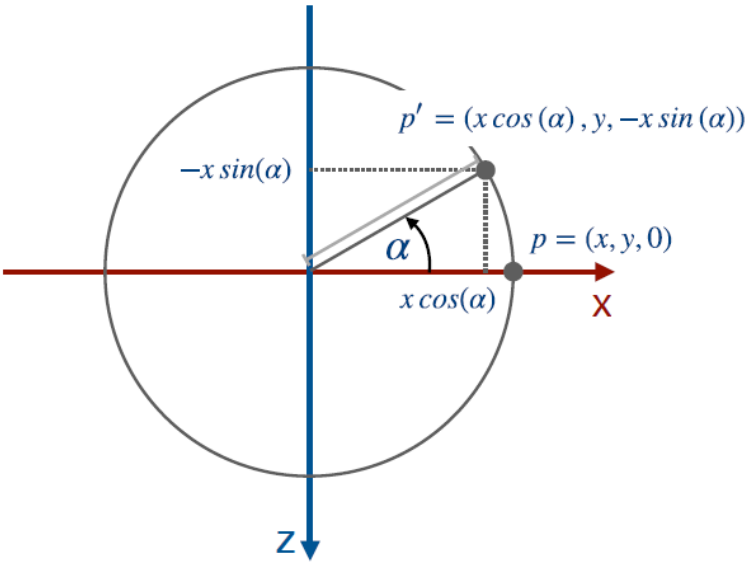


Ilustración 3.12

Como he comentado anteriormente, este proceso se hará de manera incremental, hasta haber rotado un determinado punto tantas veces como porciones haya. Lo cual nos lleva a determinar el incremento angular añadido en cada iteración del bucle, el cual será fácilmente calculable: "**360/n**", para obtener una notación en grados, o, "**2π/n**", en radianes.

De esta forma, el ángulo "**α**", se obtendrá sumando "**i**" número de veces el valor del incremento, siendo "**i**" el número de porción actual, en la siguiente *ilustración 3.13* se esclarece el proceso.

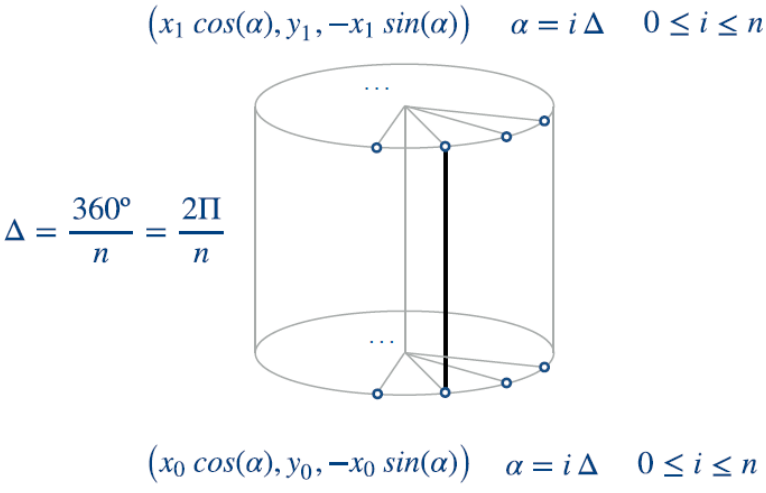


Ilustración 3.13

Como podemos observar en la anterior imagen, el recorrido empezaría desde la porción cero, hasta la porción "**n**", estando esta, incluida; esta inclusión de la última porción puede parecer redundante, ya que coincide con la primera de todas, sin embargo, es una medida simple que servirá para solucionar un pequeño conflicto a la hora de establecer una textura sobre el sólido.

Este problema está debido a que al aplicar una textura sobre una determinada geometría son necesarias las coordenadas de textura, y debido a que un vértice no puede poseer dos coordenadas de textura distintas, se ha optado por una solución algo redundante y que duplica la geometría, haciendo coincidir al último y primer punto durante la rotación del perfil.

Sin embargo, a la hora de trabajar con sólidos de revolución, nos encontramos con un conflicto aún mayor, y es que cuando el perfil de revolución posee alguno de sus puntos en el eje de revolución se dará lugar a una figura con superficies planas, funcionando estas como tapaderas del sólido. Si procesamos este determinado punto como los demás, ya sea para el cálculo de geometría, como el de la iluminación o visibilidad, estaremos cometiendo un gran error, ya que estaremos rotando un punto sobre sí mismo, repitiendo geometría de manera excesiva y errando a la hora de calcular el vector normal para los posteriores cálculos.

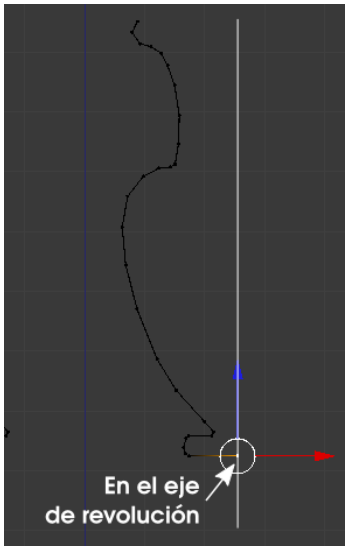


Ilustración 3.14

Para ello, habrá que tratar dichas “tapaderas” como perfiles de revolución independientes, procesando la información de manera distinta a lo anteriormente explicado, y estando estos formados solamente por el punto contenido en el eje y el siguiente directamente en el perfil de revolución.

Por una parte, el punto contenido en el eje de revolución, tendrá un vector normal completamente perpendicular a la superficie horizontal, esto es, un vector totalmente vertical, siendo el sentido positivo si nos encontramos en la “tapadera” superior o negativo, si estamos en la inferior; con respecto al otro punto, este se tratará con normalidad y siguiendo lo estudiado con anterioridad.

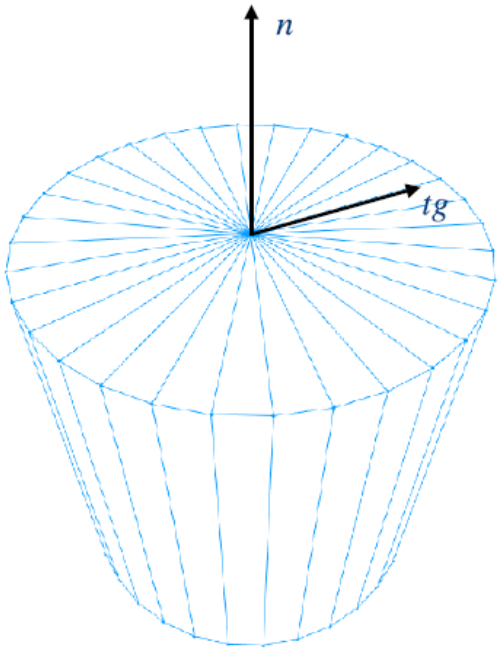


Ilustración 3.15

Otra de las partes más importantes a la hora de generar una geometría interpretable por nuestra GPU es la topología, es decir, la relación que se produce entre los distintos elementos de la geometría. El objetivo de este apartado es generar una geometría simple en términos de interpretación por una tarjeta gráfica, para ello se usarán las “tiras de triángulos”, para el perfil, y los “abanicos de triángulos”, para las posibles superficies planas que posea un determinado sólido.

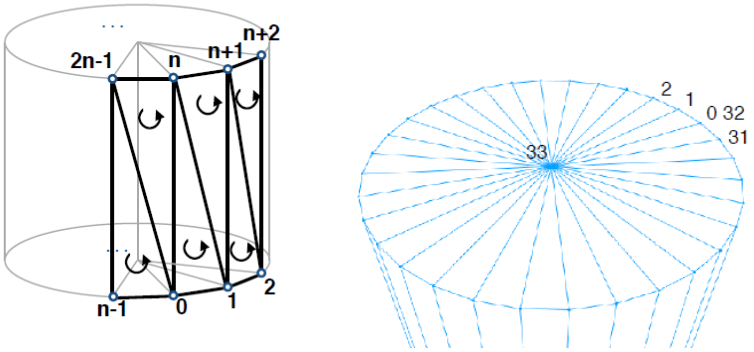


Ilustración 3.16

Todas imágenes expuestas en este apartado son propiedad de:

(Conde, 2017)

En los *apartados anteriores*, *3.2.1.1*, *3.2.1.2* y *3.2.1.3* se han analizado y explicado los conceptos matemáticos necesarios para posteriormente poder realizar la codificación de todo el proceso, pero antes de esto habrá que realizar un análisis para determinar cuáles son las clases necesarias y las relaciones entre ellas. Es por esto por lo que se presenta el siguiente diagrama de clases (*ilustración 3.17*).

Como podemos observar en dicho diagrama, la clase "**Renderer**", será la encargada de unificar las demás y realizar el proceso de renderizado de las imágenes tridimensionales.

Se ha optado por usar una luz de tipo focal, por ello, la clase "**PagSpot**", hereda de la clase "**PagLight**", aportando los atributos necesarios para dotar de la iluminación deseada. Por otro lado, tendremos la clase "**PagCamera**", la cual será la encargada de implementar una cámara virtual, encargada de "capturar" las imágenes.

Entrando en la parte matemática y que se ha estudiado con anterioridad, tendremos que disponer de una clase que represente al perfil de revolución, "**PagSubdivisionProfile**", y otra que represente al propio sólido de revolución "**PagRevolutionObject**".

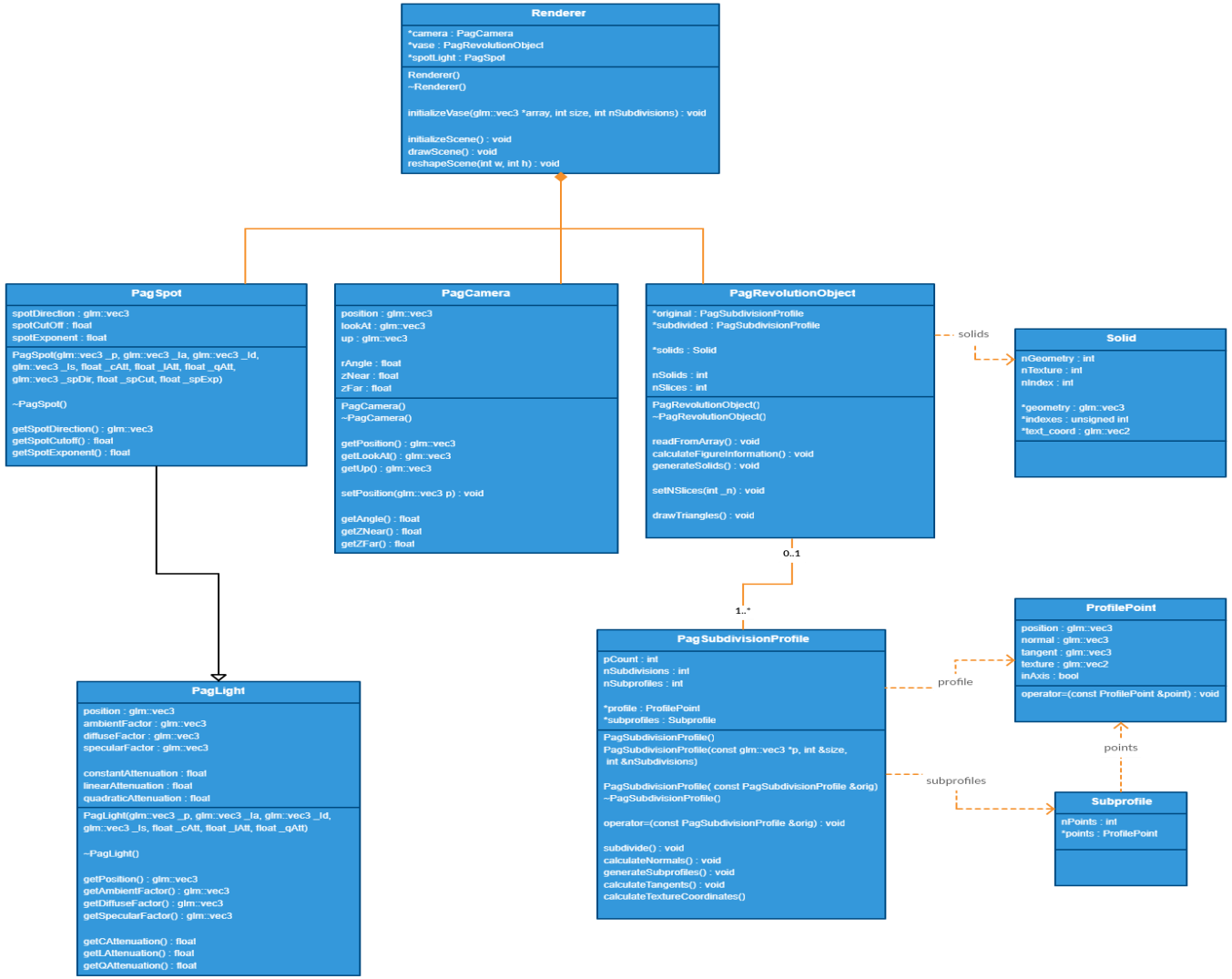


Ilustración 3.17

Se han usado también una serie de estructuras auxiliares para el mejor manejo de toda la información generada en el proceso, estas son: "**ProfilePoint**", "**SubProfile**" y "**Solid**".

Hasta este momento solo se ha hecho hincapié en apartados técnicos, matemáticos y de análisis de los diversos problemas, sin embargo, a partir de aquí se explicarán los procesos realizados para la implementación de dichos conceptos. Como bien se puede observar en la *ilustración 3.17*, se cuenta con una serie de clases que conforman la librería "**PAG**", sin embargo, y con motivos de especificidad, se hará especial atención en las clases "**PagSubdivisionProfile**", "**PagRevolutionObject**" y "**Renderer**", puesto que son las clases donde hay una mayor complejidad algorítmica.

Las clases restantes, como por ejemplo "**PagCamera**" o "**PagSpot**", son simples implementaciones de lo que se especifica en el diagrama de clases, sin mucho interés algorítmico. Para todo esto, y como se comenta en el *apartado 2.1*, se ha utilizado el lenguaje de programación C++.

Con respecto a lo explicado en el *apartado 3.2.1.1*, la subdivisión de polilíneas se ha implementado siguiendo las explicaciones dadas con anterioridad, dando lugar al algoritmo o método "subdivide" de la clase "**PagSubdivisionProfile**".

Esta clase tendrá dentro dos atributos muy importantes, por un lado "profile" representará los puntos del perfil, usando para ello la estructura auxiliar "**ProfilePoint**", detallada en el diagrama de clases anterior, y donde guardaremos todos los parámetros propios de un punto del perfil, como su posición, vector normal, tangente o coordenada de textura; por el otro lado tendremos el atributo "subprofiles", el cual será el encargado de solucionar el problema de las superficies horizontales superior e inferior del sólido, en caso de haberlas. Esto último es llevado a cabo gracias a la estructura auxiliar "**Subprofile**", la cual encapsula simplemente a un conjunto de "**ProfilePoint**".

Como podemos observar en la *ilustración 3.18*, el color rojo representa la parte del código donde se reservan los recursos necesarios para realizar el algoritmo, en la parte azul se generan los puntos deseados del nuevo perfil, que se introducen en el vector auxiliar “divided_aux”, dotándolo de suavidad y continuidad, mientras que en la parte verde se liberan los recursos y se guarda el nuevo perfil generado.

```
/**
 * Function that allows us to subdivide the profile
 */
void PagSubdivisionProfile::subdivide() {
    int size = pCount;
    glm::vec3 *original_aux = new glm::vec3[size];
    glm::vec3 *divided_aux = new glm::vec3[size];
    for (int i = 0; i < size; i++) {
        original_aux[i] = profile[i].position;
        divided_aux[i] = profile[i].position;
    }
    for (int i = 0; i < nSubdivisions; i++) {
        if (divided_aux != 0) {
            delete[] divided_aux;
        }
        divided_aux = new glm::vec3[(2 * size) - 1];
        divided_aux[0] = original_aux[0];
        int counter = 1;
        for (int j = 1; j < (size - 1); j++) {
            // - Calculating the two needed points
            glm::vec3 half_01 = (original_aux[j - 1] + original_aux[j]) / (float)2.0;

            glm::vec3 p_prime = (float)(3.0 / 4.0)*original_aux[j] + original_aux[j - 1] / (float)(8.0) + original_aux[j + 1] / (float)(8.0);

            divided_aux[counter] = half_01;
            divided_aux[++counter] = p_prime;
            ++counter;
        }
        // - Introducing the last points
        divided_aux[(2 * size) - 2] = original_aux[size - 1];

        glm::vec3 last_half = (original_aux[size - 1] + original_aux[size - 2]) / (float)(2.0);
        divided_aux[(2 * size) - 3] = last_half;
        // - Updating temporal information
        size = (2 * size) - 1;
        delete[] original_aux;
        original_aux = new glm::vec3[size];
        for (int h = 0; h < size; h++) {
            original_aux[h] = divided_aux[h];
        }
    }
    // - Saving the profile information, copying the new information from the auxiliar vector
    pCount = size;
    delete[] profile;
    profile = new ProfilePoint[pCount];
    for (int i = 0; i < size; i++) {
        profile[i].position = divided_aux[i];
    }
    delete[] original_aux;
    delete[] divided_aux;
}
```

Ilustración 3.18

Por otro lado, se han implementado una serie de funciones que, aparte de obtener los vectores normales a cada punto del perfil, necesarios e indispensables para la correcta realización del proyecto, calculan otros datos necesarios: ya sea para aplicar una textura sobre el objeto de revolución o para realizar mapas de normales y usar técnicas de “Bump Mapping”. Sin embargo, para evitar cargar el informe con información no referente al proyecto, se evitará explicar estas funciones.

Volviendo al algoritmo para generar los vectores normales, en la *ilustración 3.19*, la cual presenta un fragmento del código implementado, podemos apreciar dos partes bien diferenciadas, la parte de color rojo, en la cual se procesa si los

puntos primero y último están en el eje de revolución o no, y una segunda parte en color azul, en la cual se implementa el caso genérico para cualquier otro punto del perfil. En ambas partes se puede observar cómo se guardan los vectores normales correspondientes a cada punto del perfil en la estructura especificada, mediante la instrucción: "*profile[i].normal = x*".

```
/**
 * This function will use the information obtained in the previous one,
 * and it will calculate all the normals related to our points
 */
void PagSubdivisionProfile::calculateNormals() {
    for (int i = 0; i < pCount; i++) {
        if (i == 0) {
            // - If the first point is in the "Y" axis
            if (profile[i].position.x == 0) {
                glm::vec3 normal(0.0,-1.0,0.0);
                profile[i].normal = normal;
            } else {
                // - We calculate the vector
                glm::vec3 vector = profile[i + 1].position - profile[i].position;
                // - Normalization
                glm::vec3 normalized = glm::normalize(vector);
                // - Rotation
                glm::vec3 axis(0.0, 0.0, 1.0);
                glm::vec3 rotated = glm::rotate(normalized, -(float)M_PI / (float)2.0, axis);
                profile[i].normal = rotated;
            }
        } else if (i == (pCount - 1)) {
            // - If the last point is in the "Y" axis
            if (profile[i].position.x == 0) {
                glm::vec3 normal(0.0,1.0,0.0);
                profile[i].normal = normal;
            } else {
                // - We calculate the vector
                glm::vec3 vector = profile[pCount-1].position - profile[pCount-2].position;
                // - Normalization
                glm::vec3 normalized = glm::normalize(vector);
                // - Rotation
                glm::vec3 axis(0.0, 0.0, 1.0);
                glm::vec3 rotated = glm::rotate(normalized, -(float)M_PI/(float)2.0, axis);
                profile[i].normal = rotated;
            }
        } else {
            // - General case
            glm::vec3 prev_vector = profile[i].position - profile[i - 1].position;
            glm::vec3 next_vector = profile[i + 1].position - profile[i].position;

            glm::vec3 prev_normalized = glm::normalize(prev_vector);
            glm::vec3 next_normalized = glm::normalize(next_vector);

            glm::vec3 axis(0.0, 0.0, 1.0);

            glm::vec3 prev_rotated = glm::rotate(prev_normalized, -(float)M_PI / (float)2.0, axis);
            glm::vec3 next_rotated = glm::rotate(next_normalized, -(float)M_PI / (float)2.0, axis);

            glm::vec3 final_normal = (prev_rotated + next_rotated) / (float)(2.0);
            profile[i].normal = final_normal;
        }
    }
}
```

Ilustración 3.19

Se procede ahora a explicar lo llevado a cabo con la clase "**PagRevolutionObject**", clase que tendrá tres atributos de gran importancia. Dos de ellos serán los perfiles de revolución; por un lado, el perfil original, y por otro el perfil tras haber realizado el proceso de subdivisión sobre el original.

También tendremos el atributo " * solids", perteneciente a la estructura auxiliar "**Solid**", y en el cual guardaremos tanto la geometría como la topología que se obtendrán al revolucionar el perfil sobre el eje. El motivo de que este atributo sea un array es que se tratarán como independientes los perfiles de revolución de los "subperfiles" nombrados con anterioridad, los cuales representaban las posibles tapaderas del sólido de revolución.

Para todo este proceso algorítmico se dispondrá de dos funciones fundamentalmente. La función "generateSolids", la cual será de carácter privado en la clase del objeto de revolución, y la función "calculateFigureInformation", en la cual se hará uso de la primera y se llevará a cabo todo el proceso de generación de geometría y topología.

A parte de estas, tendremos una función encargada de pasar a nuestra tarjeta gráfica toda la información generada para poder generar las imágenes en tres dimensiones, "drawTriangles".

Habiendo también un método inicial encargado de instanciar los dos perfiles de revolución a partir de un trozo de memoria, el número de datos que lo componen y el número de subdivisiones que se llevarán a cabo en dicho perfil: "readFromArray".

Empezando por "generateSolids", esta función tiene una misión crucial, generar toda la información geométrica y topológica de la figura, así como almacenarla de manera ordenada en la estructura auxiliar "**Solid**". El proceso es complejo y tedioso, ya que se tienen que contemplar los casos en los cuales un punto del perfil pertenece al eje de revolución, es decir, el sólido tiene una tapadera, ya sea en la parte superior o en la inferior. Algorítmicamente, en la siguiente *ilustración 3.20*, se observa un fragmento de código en el que se hará la reserva de memoria para guardar toda la información que va a ser generada, tanto geometría, topología como coordenadas de textura.

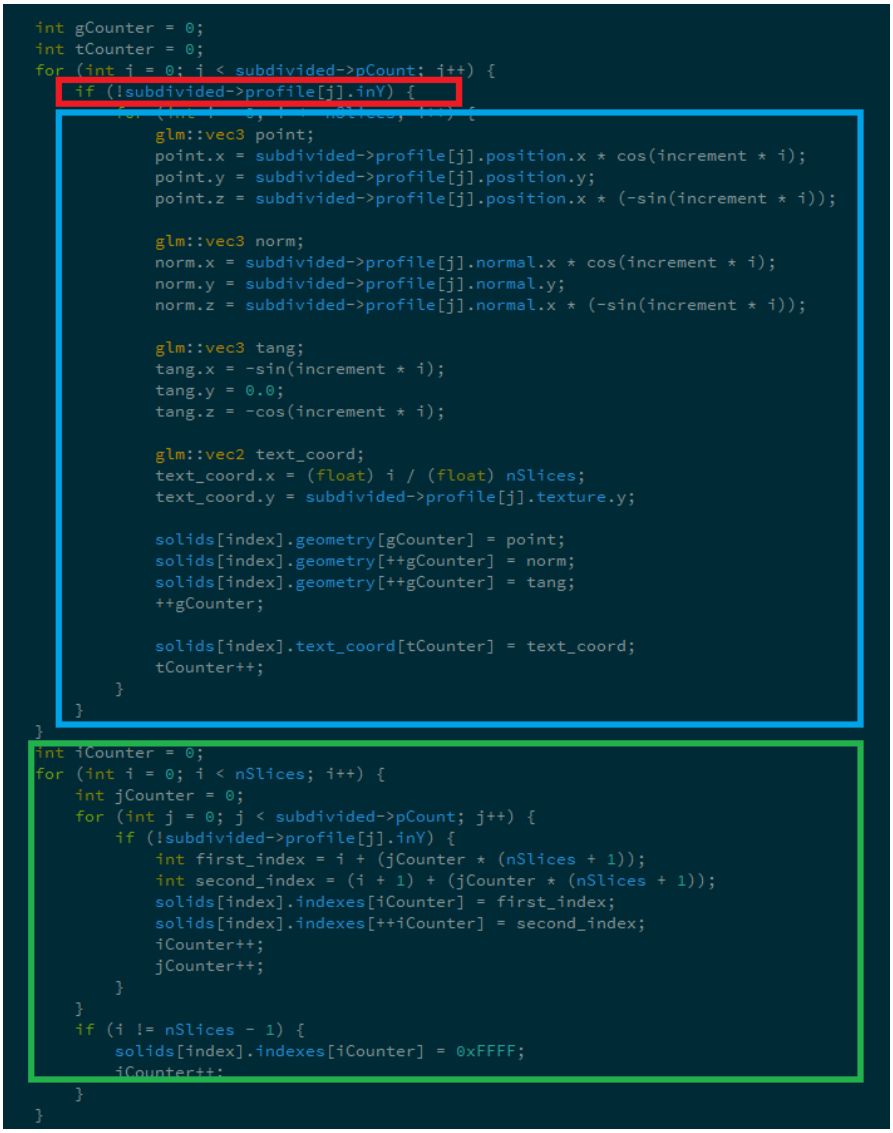
```
nSolids = subdivided->nSubprofiles + 1;
solids = new Solid[nSolids];
// - This index will help us to dynamically generate all the solids
int index = 0;
float increment = (float) (2 * M_PI) / (float) nSlices;
glm::vec3 axis(0.0, 1.0, 0.0);
int tam = subdivided->pCount - subdivided->nSubprofiles;

solids[index].nGeometry = (3 * tam)*(nSlices + 1);
solids[index].nTexture = tam * (nSlices + 1);
solids[index].nIndex = (2 * tam * nSlices) + (nSlices - 1);


solids[index].geometry = new glm::vec3[solids[index].nGeometry];
solids[index].text_coord = new glm::vec2[solids[index].nTexture];
solids[index].indexes = new unsigned int[solids[index].nIndex];
```

Ilustración 3.20

Tras esto, se comenzará con el proceso para aquellos puntos que no se encuentran situados en el eje de rotación, haciendo las pertinentes transformaciones y almacenando la información de manera continua en la estructura "**Solid**", la cual guarda dentro de "geometry" toda la geometría generada, tanto posición de los vértices como normales y tangentes; dentro de "text_coord" las coordenadas de textura y dentro de "indexes", los índices que indican la topología del objeto.

Para el almacenamiento de la geometría generada en las tapaderas superior o inferior se ha usado el propio trozo de memoria referente a la estructura nombrada anteriormente, pero en diferentes índices. De esta forma, toda la información generada referente al perfil se encontrará en el trozo de memoria "solids[0]"; la referente a la tapadera superior en "solids[1]" y a la inferior en "solids[2]".

Como podemos observar, en la *ilustración 3.21* se muestra todo el proceso de rotación de los puntos del perfil, solamente si estos no se encuentran contenidos en el eje de revolución:



```
int gCounter = 0;
int tCounter = 0;
for (int j = 0; j < subdivided->pCount; j++) {
    if (!subdivided->profile[j].inY) {
        for (int i = 0; i < nSlices; i++) {
            glm::vec3 point;
            point.x = subdivided->profile[j].position.x * cos(increment * i);
            point.y = subdivided->profile[j].position.y;
            point.z = subdivided->profile[j].position.x * (-sin(increment * i));

            glm::vec3 norm;
            norm.x = subdivided->profile[j].normal.x * cos(increment * i);
            norm.y = subdivided->profile[j].normal.y;
            norm.z = subdivided->profile[j].normal.x * (-sin(increment * i));

            glm::vec3 tang;
            tang.x = -sin(increment * i);
            tang.y = 0.0;
            tang.z = -cos(increment * i);

            glm::vec2 text_coord;
            text_coord.x = (float) i / (float) nSlices;
            text_coord.y = subdivided->profile[j].texture.y;

            solids[index].geometry[gCounter] = point;
            solids[index].geometry[++gCounter] = norm;
            solids[index].geometry[++gCounter] = tang;
            ++gCounter;

            solids[index].text_coord[tCounter] = text_coord;
            tCounter++;
        }
    }
}
int iCounter = 0;
for (int i = 0; i < nSlices; i++) {
    int jCounter = 0;
    for (int j = 0; j < subdivided->pCount; j++) {
        if (!subdivided->profile[j].inY) {
            int first_index = i + (jCounter * (nSlices + 1));
            int second_index = (i + 1) + (jCounter * (nSlices + 1));
            solids[index].indexes[iCounter] = first_index;
            solids[index].indexes[++iCounter] = second_index;
            iCounter++;
            jCounter++;
        }
    }
    if (i != nSlices - 1) {
        solids[index].indexes[iCounter] = 0xFFFF;
        iCounter++;
    }
}
```

Ilustración 3.21

Como podemos observar, en la parte marcada con un rectángulo rojo, se comprobará que los puntos que se procesan no se encuentran en el eje de revolución. Por otro lado, en la parte de color azul se realiza todo el proceso de generación de geometría, almacenando de manera continua en “geometry”, la posición, normal y tangente de un vértice; y en “text_coord” las coordenadas de textura de dicho vértice. Finalmente, en la parte de color verde se genera la topología de todos los puntos del perfil, de tal forma que al recorrer dicho vector “indexes”, la secuencia de tres índices determine el orden de los vértices en “geometry”, capaces de generar un triángulo correctamente orientado. Esta relación se puede observar en la *ilustración 3.16*.

Por otra parte, para poder generar correctamente toda la información referente a una superficie plana superior, habrá que variar los algoritmos usados levemente. Mientras que la geometría se genera de igual manera que para el caso anterior, la topología se generará de manera mucho más sencilla, guardando simplemente los índices de manera ordenada de los vértices una vez rotados, esto también se puede apreciar en la *ilustración 3.16.*

```
if (subdivided->profile[0].inY) {
    index++;

    solids[index].nGeometry = (3 * (nSlices + 2));
    solids[index].nTexture = nSlices + 2;
    solids[index].nIndex = nSlices + 2;

    solids[index].geometry = new glm::vec3[solids[index].nGeometry];
    solids[index].text_coord = new glm::vec2[solids[index].nTexture];
    solids[index].indexes = new unsigned int[solids[index].nIndex];

    int gCounter = 0;
    int tCounter = 0;

    for (int i = 0; i <= nSlices; i++) {
        glm::vec3 point;
        point.x = subdivided->subprofiles[0].points[1].position.x * cos(increment * i);
        point.y = subdivided->subprofiles[0].points[1].position.y;
        point.z = subdivided->subprofiles[0].points[1].position.x * (-sin(increment * i));

        glm::vec3 norm;
        norm.x = subdivided->subprofiles[0].points[1].normal.x * cos(increment * i);
        norm.y = subdivided->subprofiles[0].points[1].normal.y;
        norm.z = subdivided->subprofiles[0].points[1].normal.x * (-sin(increment * i));

        glm::vec3 tang;
        tang = subdivided->subprofiles[0].points[1].tangent;

        glm::vec2 text_coord;
        text_coord.x = (cos(increment * (float) i) / (float) 2.0) + (float) 0.5;
        text_coord.y = (sin(increment * (float) i) / (float) 2.0) + (float) 0.5;

        solids[index].geometry[gCounter] = point;
        solids[index].geometry[++gCounter] = norm;
        solids[index].geometry[++gCounter] = tang;
        ++gCounter;

        solids[index].text_coord[tCounter] = text_coord;
        tCounter++;
    }
    solids[index].geometry[gCounter] = subdivided->subprofiles[0].points[0].position;
    solids[index].geometry[++gCounter] = subdivided->subprofiles[0].points[0].normal;
    solids[index].geometry[++gCounter] = subdivided->subprofiles[0].points[0].tangent;
    glm::vec2 center;
    center.x = 0.5;
    center.y = 0.5;
    solids[index].text_coord[tCounter] = center;

    int counter = 0;
    for (int i = nSlices; i >= 0; i--) {
        solids[index].indexes[counter] = i;
        ++counter;
    }
    solids[index].indexes[(nSlices + 1)] = (nSlices + 1);
}
```

Ilustración 3.22

Como podemos observar en la anterior *ilustración 3.22*, se sigue una estructura similar a la presentada en la *ilustración 3.21*, en la parte roja se reservarán los recursos necesarios para almacenar toda la información, en la parte azul se generará y almacenará toda la información referente a la geometría de todos los vértices generados, y en la parte marcada de color verde se

generará la topología siguiendo el proceso anteriormente explicado. Como podemos observar, una vez rotados todos los puntos, con sus respectivas normales, tangentes y coordenadas de textura, se almacenará un punto más, el punto central, es decir, aquel que se encontraba contenido en el eje de revolución.

Para generar ahora la información de la tapadera inferior se sigue un proceso exactamente igual al que se acaba de exponer, sin embargo, habrá que cambiar la orientación del vector normal para el punto central, ya que en la tapadera superior este tendrá sentido positivo en el eje de revolución, que en nuestro caso es el eje "*Y*"; sin embargo, en la tapadera inferior tendrá sentido negativo. Además, habrá que invertir el orden de los índices topológicos generados, ya que una vez generados los triángulos en la parte superior estos tendrán una orientación contraria a los que se encuentran en la parte inferior.

```
int counter = 0;
for (int i = nSlices; i >= 0; i--) {
    solids[index].indexes[counter] = i;
    ++counter;
}
solids[index].indexes[(nSlices + 1)] = (nSlices + 1);
```

```
for (int i = 0; i <= (nSlices + 1); i++) {
    solids[index].indexes[i] = i;
}
```

Ilustración 3.23

A la izquierda, en la *ilustración 3.23*, el algoritmo para generar los índices en una superficie plana situada en la parte superior, a la derecha para una situada en la parte inferior. Esto está debido a que, en Informática Gráfica, concretamente a la hora de eliminar superficies que no son visibles para una cámara virtual, se tienen en cuenta los vectores normales a dichas superficies, por eso es muy importante dotar de un correcto orden a los índices, y por consiguiente a los vértices.

Se procede ahora a explicar la función "calculateFigureInformation", la cual es de carácter público y se encargará de encapsular tanto el proceso anterior para revolucionar el perfil como el proceso de subdivisión, generación de vectores normales, tangentes y coordenadas de textura; todo esto, de manera ordenada como se puede observar en la *ilustración 3.24*. Así, tras haber subdividido el perfil como se detalla en la función "subdivide", se calcularán las

normales, se generarán los “subperfiles” necesarios, se harán los cálculos de vectores tangentes y coordenadas de textura, para finalmente generar toda la geometría y topología como se detalla anteriormente en este mismo apartado.

```
/**
 * This function will allow us to encapsulate all the other methods aimed to
 * generate the needed information
 */
void PagRevolutionObject::calculateFigureInformation() {

    subdivided->subdivide();
    subdivided->calculateNormals();

    subdivided->generateSubprofiles();

    subdivided->calculateTangents();
    subdivided->calculateTextureCoordinates();

    this->generateSolids();
}
```

Ilustración 3.24

Por otra parte, en la función “drawTriangles”, simplemente se recorrerá la información almacenada en la estructura “ * solids”, y haciendo uso de las funciones y métodos que aporta la librería OpenGL se dibujará en pantalla el sólido de revolución. El funcionamiento de la función es simple, recorrer todos los índices generados anteriormente y almacenados en la parte “indexes” de la estructura, para poder determinar el orden de los vértices y así realizar el pintado de todos los triángulos necesarios. Esta función se llamará en cada pasada del bucle de pintado, el cual se especificará en apartados posteriores.

Finalmente, la función “readFromArray” posee una simple implementación: será la encargada de instanciar los perfiles de revolución, tanto el original como el subdividido.

```
void PagRevolutionObject::readFromArray(glm::vec3 *array, int size, int nSubdivisions){
    this->original = new PagSubdivisionProfile(array, size, nSubdivisions);
    this->subdivided = new PagSubdivisionProfile(*original);
}
```

Ilustración 3.25

Para acabar con lo referente a este *apartado 3.2.1*, se explicará a continuación lo implementado en la clase encargada de renderizar toda la información y generar las imágenes: “Renderer”.

Como bien se aprecia en el diagrama de clases, la clase “**Renderer**” está compuesta por un elemento de la clase “**PagRevolutionObject**”, un objeto de la clase “**PagCamera**” y un objeto de la clase “**PagSpot**”. Dentro de ella habrá una función “initializeVase” que se encargará de inicializar todos los parámetros del objeto de revolución. Por otra parte, esta clase dispondrá de tres funciones fundamentales para generar el contexto de “OpenGL” y realizar el bucle de dibujado. La primera de ellas será: “initializeScene”, función encargada de generar todo el contexto de “OpenGL” necesario para poder dibujar cualquier escena.

Ilustración 3.26

Por otro lado, tendremos la función “drawScene”, que se encargará de pintar la escena en cada pasada del bucle dibujado. Esto es: aplicar las transformaciones necesarias sobre el objeto de revolución para que sea de un tamaño considerable, pintar los ejes de coordenadas del mundo y realizar las rotaciones de cámara pertinentes cuando el usuario interactúe con la ventana. Todo esto se refleja en la *ilustración 3.27*.

Para terminar, la función “reshapeScene” se llamará cada vez que la ventana de “OpenGL” se redimensione, y se encarga de volver a instanciar las matrices de modelado, visión y proyección, necesarias durante el proceso de renderizado; así como establecer el nuevo tamaño de la ventana de visión, también llamada “viewport”.

```
void Renderer::drawScene() {
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(leftCamera->getPosition().x,leftCamera->getPosition().y,leftCamera->getPosition().z, leftCamera->getLookAt().x, leftCamera->getLookAt().y, leftCa

    m_rotation.setToIdentity();
    m_rotation.rotate(180.0f - (m_xRot / 16.0f), 1, 0, 0);
    m_rotation.rotate(m_yRot / 16.0f, 0, 1, 0);
    m_rotation.rotate(m_zRot / 16.0f, 0, 0, 1);
    float* data = m_rotation.data();

    glMultMatrixf(data);
    // - Draw Axis.

    glDisable(GL_LIGHTING);

    glPushMatrix();
        glColor3f(0.0f,1.0f,0.0f);
        glBegin(GL_LINES);
            glVertex3f(0.0f, 0.0f, 0.0f);
            glVertex3f(0.0f, 1000.0f, 0.0f);
        glEnd();

        glColor3f(1.0f,0.0f,0.0f);
        glBegin(GL_LINES);
            glVertex3f(0.0f, 0.0f, 0.0f);
            glVertex3f(1000.0f, 0.0f, 0.0f);
        glEnd();

        glColor3f(0.0f,0.0f,1.0f);
        glBegin(GL_LINES);
            glVertex3f(0.0f, 0.0f, 0.0f);
            glVertex3f(0.0f, 0.0f, 1000.0f);
        glEnd();
    glPopMatrix();

    // - Draw Our Vase.

    glEnable(GL_LIGHTING);

    glPushMatrix();
        glScalef(5.0f, 5.0f, 5.0f);
        glRotatef(180.0f, 0.0, 0.0, 1.0);
        glTranslatef(0.0f, -2.0f, 0.0f);
        glColor3f(0.8,0.6,0.3);
        if (vase != 0) {
            vase->drawTriangles();
        }
    glPopMatrix();
}
```

Ilustración 3.27

3.2.2. Uso e implementación de técnicas para la obtención de los puntos de un perfil mediante el análisis de una imagen

En este apartado se pretenden explicar las técnicas y algoritmos implementados para hacer posible la detección de un perfil de revolución a partir de una simple imagen de un sólido de revolución.

Una vez realizado este proceso se obtendrán una serie de puntos pertenecientes al perfil del objeto, los cuales se usarán en el recientemente explicado, *apartado 3.2.1*.

Para todo esto, y debido a la complejidad del propio proceso, se impondrán dos restricciones para simplificar el proceso. En primer lugar, el fondo de la imagen tendrá que ser de color blanco, esto hará que sea mucho más simple detectar qué píxeles de la imagen son los pertenecientes al objeto en cuestión. Y, en segundo lugar, los objetos no deberán arrojar sombras, ya que de esta forma no habrá errores a la hora de detectar las formas buscadas en cualquier imagen.

3.2.2.1. Análisis de los componentes principales

En esta ocasión se analizarán y explicarán los conceptos referentes al análisis de los componentes principales en una imagen, más concretamente, para el actual proyecto, la detección de ejes de simetría en un determinado contorno.

Ilustración 3.28

Este proceso se realizará de manera interactiva con el usuario; así, el usuario final podrá variar un valor de un umbral con el que ajustar el contorno de manera correcta.

Con respecto al proceso matemático y algoritmo que se lleva a cabo: en primer lugar, la imagen que se carga en la aplicación, será inmediatamente transformada a escala de grises y posteriormente umbralada con el valor que el usuario determina. De esta forma, se pretenden mantener en una imagen, de manera paralela, solamente aquellos píxeles que realmente aportan información importante del actual proceso. Tras esto, se recorrerá el contorno que se acaba de encontrar para rellenar los posibles huecos que haya en el centro de la figura; estos huecos se deben a los cambios de brillo, color o textura en el jarrón, los cuales se pueden perder al aplicar el umbral.

Finalmente, se aplicará un algoritmo, propio de la librería utilizada para el uso y análisis de la imagen, el cual se encarga de obtener el eje y orientación de cualquier contorno que se haya generado mediante los pasos explicados anteriormente. Este algoritmo consiste en analizar un conjunto de puntos, en este

caso, un conjunto de píxeles, los cuales se distribuyen por el espacio en dos dimensiones; variando tanto en sus componentes “x” como en “y”, como bien se puede apreciar en la *ilustración 3.29*.

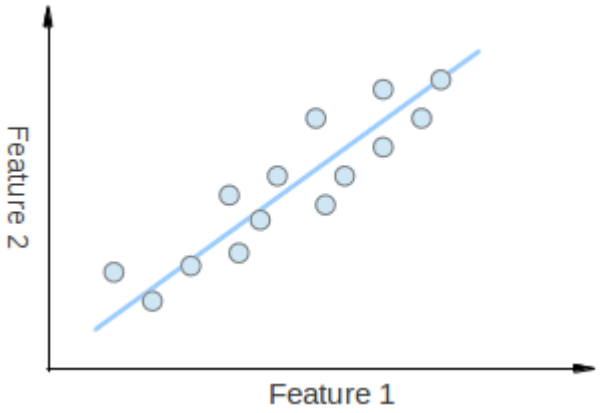


Ilustración 3.29

De esta forma, encontrar el eje de simetría de un determinado contorno, o conjunto de píxeles, se reducirá a encontrar la dirección a través de la cual los puntos varían en mayor magnitud. Esto se puede apreciar en la imagen que se acaba de exponer, ya que los puntos se distribuyen alrededor de la línea azul.

(svpenkov, 2013)

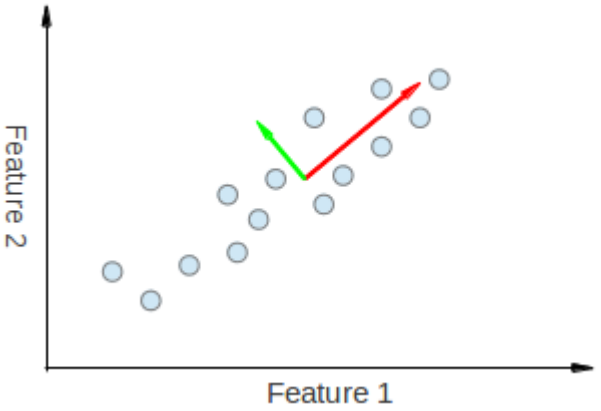


Ilustración 3.30

Por tanto, a la hora de dibujar el eje en la imagen bastará con dibujar una línea infinita que tenga la misma dirección que el eje de simetría encontrado por el anterior algoritmo, dando lugar a algo parecido a lo mostrado en la *ilustración 3.28*.

3.2.2.2. Ajuste de las elipses principales

Otra de las partes fundamentales durante el proceso de obtención de los puntos del perfil de revolución a partir de una imagen es la de ajustar dos elipses en la imagen, una que coincida con la tapadera superior del sólido de revolución en cuestión, y otra que coincida con la inferior. Este paso es bastante simple de ejecutar y se llevará a cabo por parte del usuario final, siendo por tanto un proceso interactivo.

Dichas elipses principales tendrán tres grados de libertad: por un lado, podremos ajustar su centro a lo largo del eje, variando por tanto su posición vertical en la imagen; por otro, el radio horizontal de la propia elipse; y finalmente, su radio vertical.

Una vez se ajusten dichas elipses, se usarán sus parámetros para interpolar la relación de aspecto, entre el radio vertical y el horizontal, entre todo el eje que se obtiene en el *apartado 3.2.2.1*, así se generarán las elipses intermedias necesarias para obtener los puntos del perfil, esto se especificará más detalladamente en el siguiente *apartado 3.2.2.3*.

Como se puede intuir, este apartado se basa esencialmente en usar la librería que nos permite trabajar con imágenes, “OpenCV”, para así dibujar en estas las dos elipses con los parámetros deseados por el usuario.

Ilustración 3.31

3.2.2.3. Generación de las elipses secundarias y puntos del perfil

Finalmente, una vez se han ajustado las elipses superior e inferior, se realizará un proceso para obtener una serie de elipses intermedias, las cuales nos aportarán, de manera directa y haciendo uso del radio horizontal de estas, los puntos del perfil buscado para poder generar el modelo tridimensional.

Dicho proceso consistirá en, primer lugar, obtener el número de elipses intermedias a generar; ya que, si se generan pocas elipses entre la tapadera superior y la inferior, el modelo que se obtendrá no será fiel a la realidad y contará con poco detalle.

Una vez ajustado el número de elipses a generar, se procede entonces a realizar la interpolación lineal entre la relación de aspecto de la elipse superior y la inferior, relación que se obtiene dividiendo el radio vertical entre el radio horizontal. Por lo que, para cada elipse que se genere en el proceso habrá que establecer un valor para el radio horizontal y vertical que se ajuste a la relación de aspecto interpolada en dicho punto del eje de revolución.

Finalmente, y para deshacer la transformación de perspectiva y así obtener los puntos del perfil deseados, habrá que buscar el valor que haga que el radio horizontal quede perfectamente ajustado dentro del contorno del jarrón; esto es, la elipse intermedia generada tendrá que tocar el contorno, quedando totalmente dentro de este sin que ninguna parte de la elipse quede fuera. Esto se puede apreciar de una manera mucho más clara en la *ilustración 3.32*, donde las elipses de color verde quedan perfectamente ajustadas al contorno del jarrón.

La perspectiva no es más que el efecto que se produce en nuestro cerebro al interpretar las imágenes que reciben nuestros ojos; de esta forma, al ver un objeto de frente, en este caso un jarrón, no percibimos sus tapaderas como circunferencias, aunque estas lo sean, sino que tenemos la sensación de que son elipses.

Finalmente, una vez se hayan generado y ajustado todas las elipses, bastará con almacenar el valor del radio horizontal, de manera que, el conjunto de todos los radios horizontales, incluidos los de las elipses superior e inferior, dará lugar al perfil de revolución del sólido en cuestión.

(Pernici, 2005)

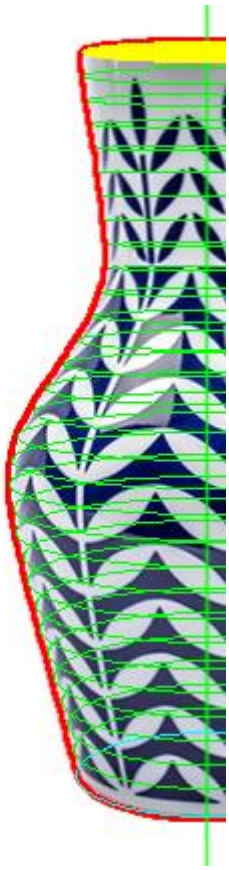


Ilustración 3.32

Una vez explicado todo esto sobre los procesos algorítmicos y matemáticos detrás del análisis de las imágenes, se procede a analizar, con el correspondiente diagrama de clases (*ilustración 3.33*), las clases que intervendrán en el proceso. Esto servirá para realizar correctamente la posterior implementación de todos los conceptos expuestos con anterioridad.

Por un lado, tendremos la clase "**VaseDetector**", la cual se encargará de encapsular todo el proceso algorítmico y de análisis de las imágenes que se ha explicado en este mismo apartado.

Por otro, la clase "**Elipse**", clase que implementará las características propias de una elipse.

Finalmente, se hará uso de la clase "**MainWindow**", la cual tiene que ver con la interfaz gráfica, y por tanto servirá de mensajera entre el usuario y todo el proceso. Esta última clase se especificará detalladamente en el *apartado 3.2.3*.

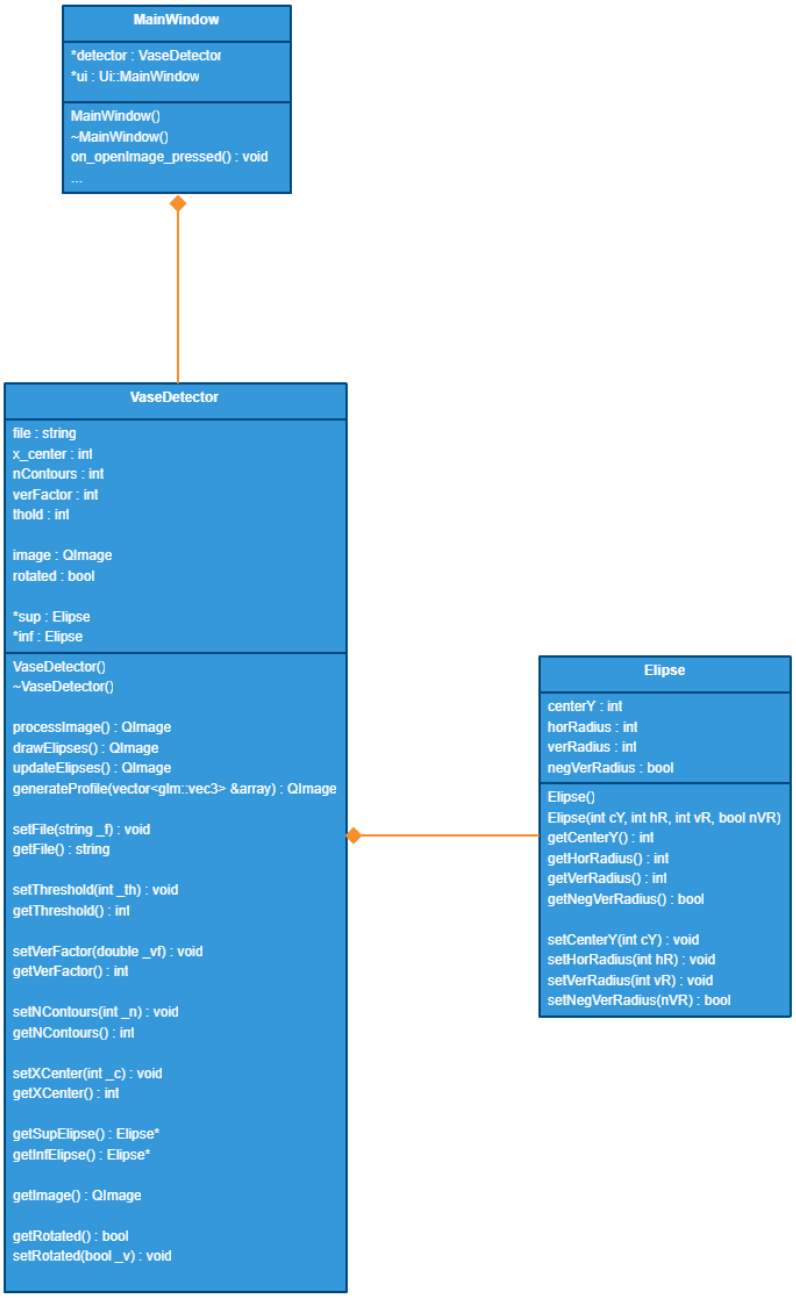


Ilustración 3.33

Como podemos observar, la clase "**Elipse**" posee una relación de composición con la clase "**VaseDetector**", y a su misma vez, "**VaseDetector**" posee una relación de composición con "**MainWindow**", lo que delega, de manera directa, a esta clase la responsabilidad de crear y gestionar todo lo referente al proceso de análisis de las imágenes.

Una vez realizado todo el proceso de análisis, tanto de un punto de vista algorítmico y matemático, como de un punto de vista estructural, se procede a explicar todo lo realizado durante el proceso de implementación. Como bien se ha comentado hasta ahora en repetidas ocasiones, todo se ha llevado a cabo mediante "C++" y la librería matemática "OpenCV".

Cabe destacar que la clase realmente importante en cuanto a algoritmia, es la clase "**VaseDetector**", ya que la clase "**Elipse**" simplemente encapsula información y se encarga de mantenerla accesible, y la clase "**MainWindow**" representa la interfaz gráfica del programa.

Antes de proceder a explicar lo referente a las funciones más destacables de dicha clase, se van a explicar los distintos atributos que esta posee. En primer lugar, el atributo "file", se va a encargar de almacenar la cadena de texto que representa el archivo de la imagen en el computador.

Por otro lado, disponemos del atributo "x_center", el cual es de tipo entero, y que almacena la información referente al píxel donde se encuentra el eje de rotación, esto es: su valor de abscisa.

Además, se dispone de un atributo que se encargará de numerar la cantidad de contornos encontrados en la imagen, se usará como atributo de control a la hora de interactuar con la interfaz gráfica, "nContours".

"verFactor" almacenará el valor de la altura de la ventana donde se encuentra la imagen, se usa para redimensionar las distintas imágenes y que todas ellas posean un tamaño similar.

El atributo “thold” es el encargado de almacenar el valor del umbral dado por el usuario, este valor está sujeto a cambios. Esto se puede ver expresado en el *apartado 3.2.2.1*.

También se cuenta con el atributo “image” que representa la imagen que se usará a lo largo de todo el proceso. Además de contar con el atributo “rotated” de tipo booleano, que determinará cuando habrá que rotar la imagen para que esta coincida con la orientación del eje de rotación.

Finalmente se hará uso de dos atributos de tipo “Elipse”, los cuales representan a la elipse superior y a la inferior, “*sup” y “*inf”, respectivamente.

Una vez explicados los atributos necesarios para llevar a cabo el proceso, podemos explicar el funcionamiento del método “processImage()”, el cual devuelve una imagen como última instancia, y es llamado cada vez que se lee una imagen o que se ajusta el valor del parámetro “thold” o umbral.

El objetivo primordial de esta función es el de encontrar el número de contornos que hay en una imagen, así como determinar el eje de rotación para cada uno de ellos. Como bien se explicaba con anterioridad, el proceso consistirá en primer lugar, en leer la imagen, aplicar la pertinente transformación a escala de grises necesaria y aplicar el correspondiente umbral determinado por el usuario, y ajustable mediante la interfaz; así como cambiar el tamaño de la imagen para que esta quede perfectamente encajada en la interfaz diseñada. Todo esto se puede observar en la *ilustración 3.34*, donde se presenta un pequeño fragmento del código de dicha función en el que se realiza la operación descrita con anterioridad, todo ello haciendo uso de la librería anteriormente descrita, “OpenCV”.

```
nContours = 0;

Mat im = imread(file,1);
Mat bw;

cv::resize(im, im, Size(), (double)verFactor/(double)im.rows, (double)verFactor/(double)im.rows, INTER_CUBIC);

cvtColor(im, bw, COLOR_BGR2GRAY);
threshold(bw, bw, thold, 255, CV_THRESH_BINARY);
```

Ilustración 3.34

Tras esto, y tratando la imagen que se acaba de leer como una matriz, se recorrerán sus filas una a una, de tal manera que al encontrar un píxel con valor distinto al blanco se rellenarán los posteriores píxeles a este con el color negro, hasta llegar al final del contorno. Esto se lleva acabo para eliminar los posibles huecos de color blanco que queden al umbralar la imagen debidos a brillos, rugosidades o a la propia iluminación de la imagen, dando lugar a una imagen como la que se observa en la *ilustración 3.35*.

Ilustración 3.35

En la siguiente *ilustración 3.36*, se puede observar el ajuste de un valor apropiado para el umbral, ajustable por el propio usuario final, cambiando en su totalidad el resultado a esperar, y por tanto la fiabilidad y detalle del modelo generado tridimensionalmente.

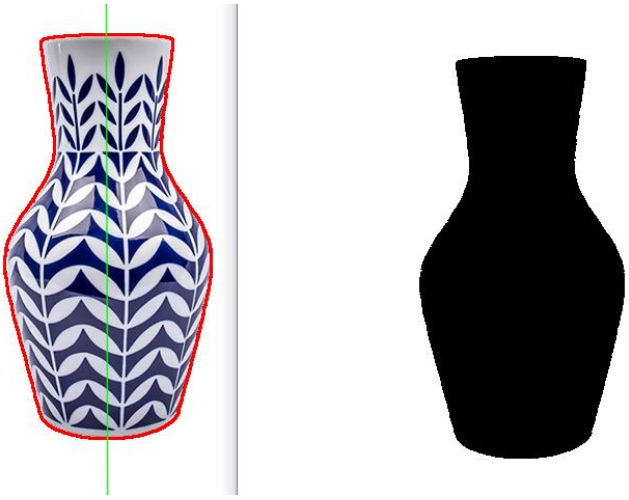


Ilustración 3.36

Tras todo esto, y para finalizar la función, se llamarán a las propias funciones de la librería "OpenCV" para encontrar los contornos y dibujarlos en la imagen, así como encontrar para cada uno de ellos su pertinente eje de revolución. Por cada contorno encontrado se aumentará el valor del atributo "nContours", y tras encontrar el contorno apropiado se guardará el valor de su posición en la imagen en el atributo "x_center". Finalmente se guardará la imagen obtenida, con todos los contornos dibujados y sus ejes, en el atributo descrito con anterioridad, "image". Toda esta parte final se puede encontrar codificada en la *ilustración 3.37*.

```
std::vector<std::vector<Point> > contours;
std::vector<Vec4i> hierarchy;
findContours(bw, contours, hierarchy, CV_RETR_LIST, CV_CHAIN_APPROX_NONE);

for (size_t i = 0; i < contours.size(); ++i)
{
    // Calculate the area of each contour
    double area = contourArea(contours[i]);
    // Ignore contours that are too small or too large
    if (area < 1e2 || 1e5 < area) continue;
    nContours++;
    // Draw each contour only for visualisation purposes
    drawContours(im, contours, i, CV_RGB(255, 0, 0), 2, 8, hierarchy, 0);
    // Find the orientation of each shape
    //Construct a buffer used by the pca analysis
    x_center = (getOrientation(contours[i], im).x);
}

image = Mat2QImage(im);
return image;
```

Ilustración 3.37

Otras de las funciones importantes de esta clase es la encargada de dibujar las elipses principales, "drawElipses"; sin ajustar correctamente en las tapaderas del sólido de revolución, ya que esta tarea le corresponde al usuario final de la aplicación; si no que estas tendrán unos parámetros por defecto independientes de la imagen cargada.

Para llevar a cabo esta tarea, primero habrá que comprobar si la imagen se ha rotado, ya que como explicábamos con anterioridad, esta tiene que encontrarse alineada verticalmente con el eje de rotación para que el resultado sea el mejor posible; por lo que simplemente, y haciendo uso de la librería "OpenCV", rotaremos la imagen tantos grados como grados de inclinación tenga

el eje de revolución. Esto se realizará una vez por cada imagen cargada, es por esto que se utiliza el parámetro “rotated”, controlando el estado de la imagen. Finalmente, se dibujarán y crearán los atributos “* inf” y “* sup”, devolviendo en última instancia la imagen sobre la que se ha realizado todo el proceso de dibujado.

```
QImage VaseDetector::drawElipses() {
    Mat im = QImage2Mat(image);

    if (!rotated){
        Mat rot_mat( 2, 3, CV_32FC1 );

        rot_mat = getRotationMatrix2D( Point(im.cols/2, im.rows/2), 270.0 - ang, 1.0 );
        warpAffine(im, im, rot_mat, Size(im.cols, im.rows), INTER_LINEAR, cv::BORDER_CONSTANT, Scalar(255, 255, 255));
        warpAffine(blackwhite, blackwhite, rot_mat, Size(blackwhite.cols, blackwhite.rows), INTER_LINEAR, cv::BORDER_CONSTANT,

        image = Mat2QImage(im);
        rotated = true;
    }

    ellipse( im, Point( x_center, image.height()/2 + 50 ), Size( 10, 60 ), 90, 0, 360, Scalar( 255, 255, 0 ), 1, 8 );
    inf = new Elipse(im.rows/2 + 50, 60, 10, false);

    ellipse( im, Point( x_center, image.height()/2 - 50 ), Size( 10, 60 ), 90, 0, 360, Scalar( 0, 255, 255 ), 1, 8 );
    sup = new Elipse(im.rows/2 - 50, 60, 10, false);

    QImage final = Mat2QImage(im);
    return final;
}
```

Ilustración 3.38

Esta clase posee también la función encargada de actualizar las elipses gráficamente en la imagen. Esta función será utilizada cada vez que el usuario varíe algún parámetro de las elipses, ya sea su posición en el eje, su radio horizontal o el radio vertical.

Lo que se realizará será muy simple: cuando en la interfaz gráfica se actualice algún parámetro se modificarán las elipses generadas en la función anterior, “* inf” y “* sup”; para finalmente, en la función “updateElipses”, leer la imagen y dibujar sobre esta las elipses con sus nuevos parámetros. La función devuelve, al igual que la anterior, una imagen al finalizar el proceso.

```
QImage VaseDetector::updateElipses() {
    Mat im = QImage2Mat(image);
    if (inf->getNegVerRadius()) {
        ellipse( im, Point( x_center, inf->getCenterY() ), Size( inf->getVerRadius(), inf->getHorRadius() ),
    } else {
        ellipse( im, Point( x_center, inf->getCenterY() ), Size( inf->getVerRadius(), inf->getHorRadius() ),
    }

    if (sup->getNegVerRadius()) {
        ellipse( im, Point( x_center, sup->getCenterY() ), Size( sup->getVerRadius(), sup->getHorRadius() ),
    } else {
        ellipse( im, Point( x_center, sup->getCenterY() ), Size( sup->getVerRadius(), sup->getHorRadius() ),
    }
    QImage final = Mat2QImage(im);
    return final;
}
```

Ilustración 3.39

Para finalizar con las funciones destacadas, desde un punto de vista algorítmico, de la clase "**VaseDetector**", se procede a hablar de la función "generateProfile", la cual recibe como parámetro un contenedor vacío de vectores en el espacio tridimensional, el cual se llenará a lo largo del transcurso de la función. Cabe destacar que esta función se llamará única y exclusivamente cuando el usuario lo determine, siendo necesario haber ajustado correctamente las elipses superior e inferior para que así el resultado sea el deseado. En ella se llevará a cabo el proceso de interpolación por el cual se generarán todas las elipses intermedias, y obteniendo por tanto todos los puntos del perfil de revolución.

Para esto, en primer lugar, habrá que obtener los valores de las relaciones de aspecto de las elipses superior e inferior, así como todos los parámetros necesarios para poder realizar la interpolación lineal entre el centro de la elipse superior y el centro de la elipse inferior.

```
double supAspect = (double)sup->getVerRadius() / (double)sup->getHorRadius();
double infAspect = (double)inf->getVerRadius() / (double)inf->getHorRadius();

if (sup->getNegVerRadius()) supAspect = -supAspect;
if (inf->getNegVerRadius()) infAspect = -infAspect;

int centerYSup = sup->getCenterY();
int centerYInf = inf->getCenterY();
```

```
double aspectIterator = supAspect;
double incrementAspect = (supAspect - infAspect) / (double)size;
int increment = diff / size;

int pos = centerYSup;
```

Ilustración 3.40

Otro atributo importante a calcular será el número de elipses a interpolar por todo el eje de revolución; para ello se parte de un valor arbitrario, siendo a su misma vez este la cantidad mínima de estas elipses que habrá en el eje, intentándose encontrar un valor superior y que sea divisible entre la cantidad de píxeles, o distancia, entre el centro de la elipse superior e inferior.

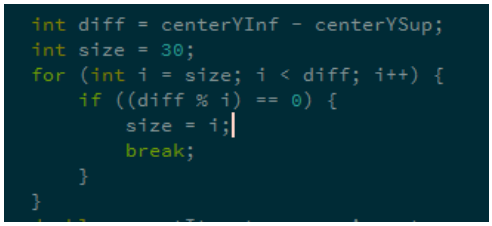


Ilustración 3.41

Tras esto se cargará la imagen original, en la cual se dibujarán de nuevo las elipses con sus parámetros correctamente ajustados, será entonces cuando se introduzca en el contenedor, provisto a la función, la primera de las elipses, la elipse superior.

Es el momento de realizar un bucle que se repetirá tantas veces como elipses haya que interpolar. En este bucle, se interpolará la relación de aspecto para todas las elipses intermedias, y en cada pasada, se buscará el radio horizontal que, respetando la relación de aspecto, haga que la elipse quede ajustada de la mejor forma posible con el contorno del sólido de revolución, es decir, el mayor radio horizontal posible sin que la elipse salga del contorno.

Para esta comprobación se usará un contenedor auxiliar, llamado “points”, el cual guardará todos los puntos, o píxeles, que conforman la elipse, y se comprobará que estos puntos quedan dentro del contorno, lo cual es bastante simple de comprobar si se tiene en cuenta que estamos trabajando sobre una imagen binaria: es decir, si los puntos quedan dentro del contorno tendrán el color negro, y si están fuera el color blanco.

En cada pasada de este bucle se añadirá otro punto al contenedor encargado de almacenar los puntos del perfil de revolución, para finalmente, trans haber acabado el bucle, añadir el punto final, el de la elipse inferior.

Para finalmente acabar devolviendo la imagen resultante obtenida durante el proceso, la cual tendrá dibujada todas las elipses interpoladas. Esta parte final de la función se puede observar codificada en la *ilustración 3.42*.

Las demás funciones detalladas en el diagrama de clases, *ilustración 3.33*, de la clase “**VaseDetector**”, serán las funciones encargadas de devolver y

establecer los parámetros privados de la clase, los mismos parámetros que se han descrito con anterioridad en este mismo apartado.

```
array.push_back(glm::vec3((double)sup->getHorRadius()/100.0, (double)centerYSup/100.0, 0.0));

for (int i = 1; i < size; i++) {
    aspectIterator -= incrementAspect;
    pos += increment;
    int xPos = 0;

    bool found = false;

    for (int j = 1; j < im.cols/2 ; j++) {
        int verRadius;

        if (aspectIterator < 0.0001 && aspectIterator > -0.0001) verRadius = 0;
        else verRadius = j * aspectIterator;

        vector<Point> points;

        ellipse2Poly(Point(x_center, pos), Size(verRadius, j), 90, 0, 360, 1, points);

        for (uint h = 0; h < points.size(); h++) {
            if (blackwhite.at<uchar>(points[h]) != 0) {
                found = true;
                xPos = j - 1;
                break;
            }
        }
        if (found) break;
    }
    int vertical;
    if (aspectIterator < 0.0001 && aspectIterator > -0.0001) vertical = 0;
    else vertical = xPos * aspectIterator;

    ellipse( im, Point( x_center, pos ), Size( std::abs(vertical), xPos ), 90, 0, 360, Scalar( 0, 255, 0 ), 1, 8 );
    glm::vec3 point((double)xPos/100.0, (double)pos/100.0, 0.0);
    array.push_back(point);
}
array.push_back(glm::vec3(0.0, array[size-1].y, 0.0));

QImage final = Mat2QImage(im);
return final;
```

Ilustración 3.42

Como bien se ha mencionado anteriormente, y aunque la clase “**Elipse**” no presenta un gran interés algorítmico, sí que cabe destacar, y resolver, el significado de uno de sus parámetros, “negVerRadius”, el cual hace referencia a si el radio vertical de una elipse es negativo o no, atributo que por defecto tendrá por valor “falso”. Y es que, para el presente proyecto, esto solo dependerá de la perspectiva y del punto de vista que se tenga del propio sólido de revolución.

El radio vertical será positivo cuando la elipse en cuestión quede por debajo de la línea de visión, línea que será paralela al mundo y saldrá desde los ojos del observador, por otro lado, este radio será negativo si la elipse queda dispuesta por encima de la línea de visión, esto se puede observar en la *ilustración 3.43*, en la cual, la parte de color amarillo se corresponde con la parte negativa, y la parte de color azul con la positiva.

El uso, y sentido, de tener en cuenta este factor viene a la hora de interpolar la relación de aspecto entre la elipse superior e inferior. Es debido a esto por lo que una elipse con radio vertical, o ángulo, negativo será completamente distinta a una que tenga los mismos valores en sus parámetros y su radio sea positivo.

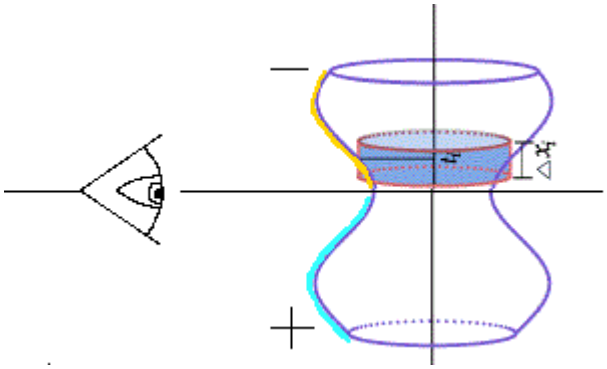


Ilustración 3.43

En la aplicación a implementar, esto se diferenciará de manera visual para el usuario, siendo las elipses con radio vertical positivo representadas simplemente por la envolvente propia que posee una elipse, mientras que las que posean el radio vertical negativo estarán completamente coloreadas de su color propio.

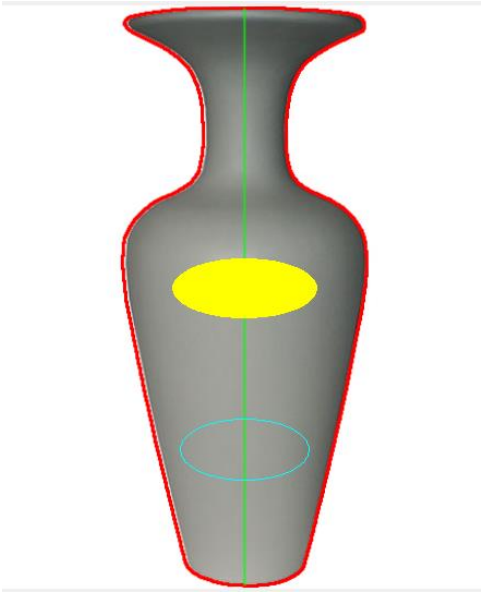


Ilustración 3.44

3.2.3. Interfaz gráfica de usuario

Todo lo descrito anteriormente no tiene sentido por sí solo, es necesario el uso de algún tipo de mensajero entre los usuarios y la propia aplicación, para ello se implementará una interfaz de usuario sencilla e intuitiva, supliendo además la necesidad de obtener una respuesta visual al llevar a cabo el proceso.

Esta parte de la aplicación, como bien se nombra en apartados anteriores, se llevará a cabo gracias al propio entorno de desarrollo “Qt Creator” y a la librería con el mismo nombre “Qt”. Se adjunta, a continuación, una ilustración donde se aprecia el entorno de desarrollo propio para todo lo relacionado con interfaces de usuario.

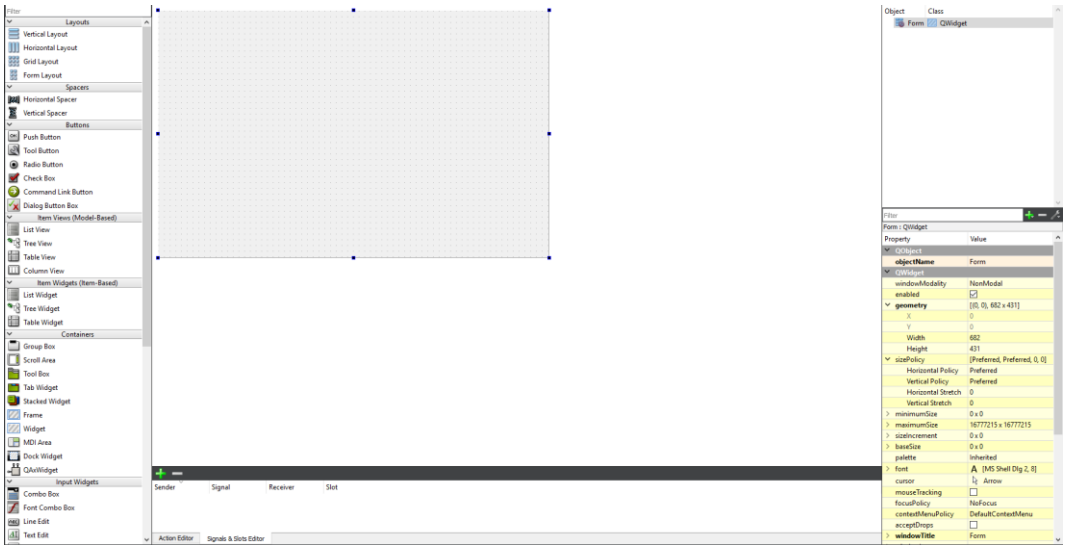


Ilustración 3.45

Como podemos observar, en la parte de la izquierda nos encontramos con todos los elementos que el usuario puede necesitar, desde botones, desplegables y menús, hasta tablas, calendarios y gráficos.

En la parte central nos encontramos con la propia interfaz de usuario que se irá desarrollando gracias al mecanismo “drag and drop” del que se dispone en el entorno de desarrollo “Qt Creator”; mientras que en la parte de la derecha tenemos dos partes diferenciadas. La parte derecha superior, reservada para los estilos y ajustes de “layout”, y la parte derecha inferior, donde se muestra un menú con todos los atributos del elemento seleccionado actualmente.

Poseemos, por tanto, todos los elementos requeridos para poder llevar a cabo la tarea de implementar la interfaz de usuario, necesaria para unificar todos los conceptos expuestos con anterioridad.

3.2.3.1. Implantación de los elementos de la interfaz

Con respecto a los elementos que poseerá la interfaz, en principio caben destacar dos de ellos, aquellos encargados de expresar visualmente lo referente al análisis de la imagen y al modelo en tres dimensiones. Para expresar continuidad y dotar de sentido al diseño de la propia interfaz, y ya que la etapa de análisis de la imagen se realiza de manera anterior a todas las demás, esta parte de la interfaz se encontrará en la zona izquierda, mientras que la encargada de representar el modelo en tres dimensiones se encontrará en la zona derecha, siendo esta el resultado de todo el proceso.

(Smith, 2003)

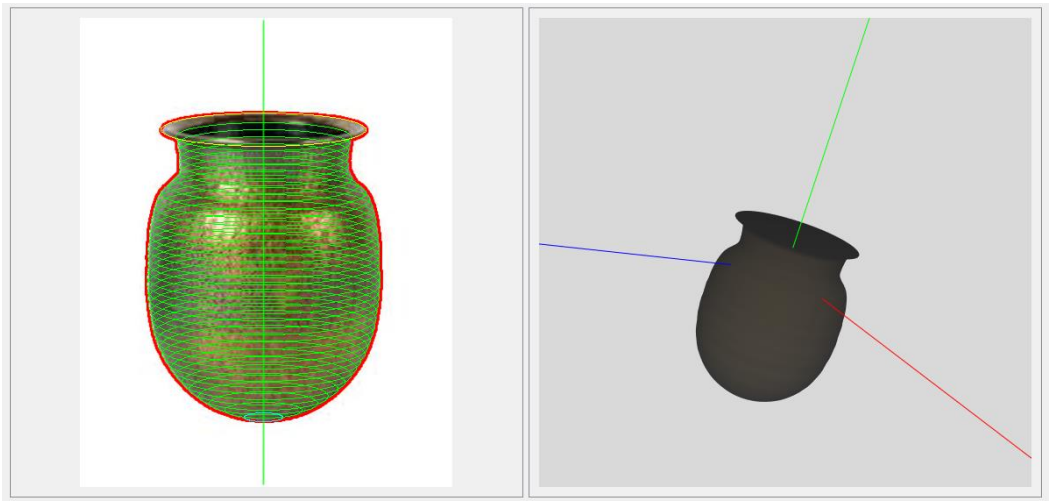


Ilustración 3.46

Pero, además de estos dos elementos de la interfaz de usuario, habrá que disponer, en siguiente lugar, de algún mecanismo que permita seleccionar alguna imagen del propio sistema, por lo que se ha optado por usar un botón cuyo accionamiento abra un navegador de archivos, el cual acepte imágenes de varios formatos, “.jpg”, “.tif”, “.png” o “.bmp”.

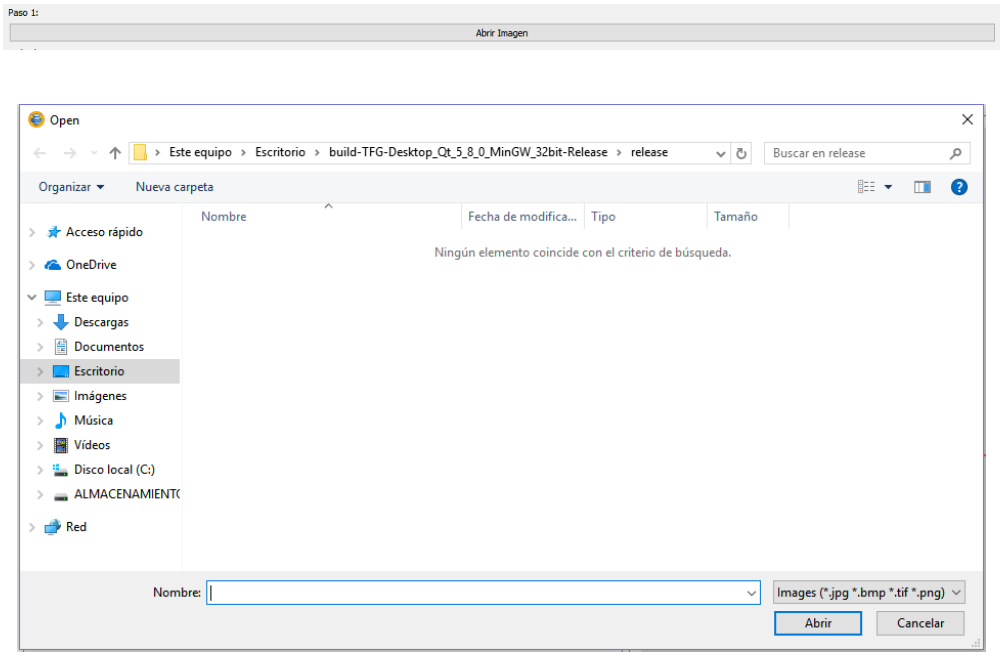


Ilustración 3.47

Como bien se expresaba durante el apartado 3.2.2, el proceso es interactivo, dependiendo este en gran medida del usuario final, y teniendo este que ajustar diversos parámetros referentes al propio proceso. En primer lugar, se hablaba del umbral que se aplicaría a la imagen tras transformarla a escala de grises; esto se conseguirá usando una barra deslizadora, variando el valor del parámetro manejado entre cero y doscientos cincuenta y cinco, valores mínimo y máximo que podría darse a la hora de umbralar una imagen.

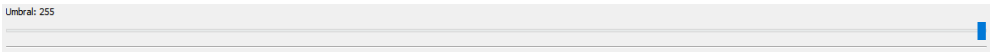


Ilustración 3.48

Tras este paso habrá que poder mostrar las elipses superior e inferior en la propia imagen, así como permitir variar sus parámetros y mostrar una respuesta en tiempo real al usuario, para ello se han usado botones, deslizadores horizontales y cajas de comprobación, o “check-boxes”.

Ilustración 3.49

Finalmente, y tras haber realizado todo el proceso interactivo descrito en anteriores apartados, se precisará de otro mecanismo que se encargue de accionar el proceso mediante el cual se genera el modelo tridimensional del sólido de revolución analizado, en este caso se ha optado por un botón, todo esto se puede observar en la siguiente ilustración 3.48.

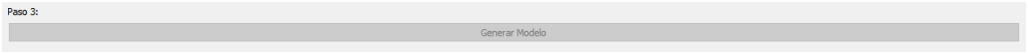


Ilustración 3.50

Así mismo, toda la aplicación se irá dividiendo en distintos “pasos”, de tal forma que para avanzar entre todos ellos habrá que ir realizando todo el proceso interactivo de manera correcta. Cabe decir que es imposible cerciorarse de que el usuario, que se encarga de manejar la aplicación, ajuste de manera correcta las elipses en la imagen, y aunque esto no provocará ningún fallo en la ejecución de la aplicación, el modelo resultante no será el esperado ni deseado por el usuario.

3.2.3.2. Escalado, espaciado y márgenes

Por otro lado, en el presente apartado se pretende explicar como todos los elementos se ajustarán dentro de la ventana que representará la aplicación.

En primer lugar, tendremos el conjunto de todos los elementos de la ventana, los cuales estarán separados unos de otros mediante un espaciado vertical, o “vertical layout”. Esto hará que los diversos elementos que conforman la aplicación se distribuyan verticalmente a lo largo de toda la ventana, y a su vez estos se ajusten a la forma que esta posea, si esta es redimensionada.

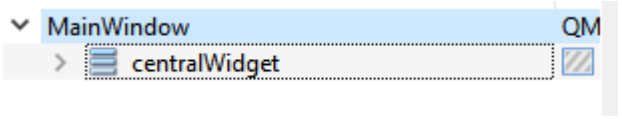
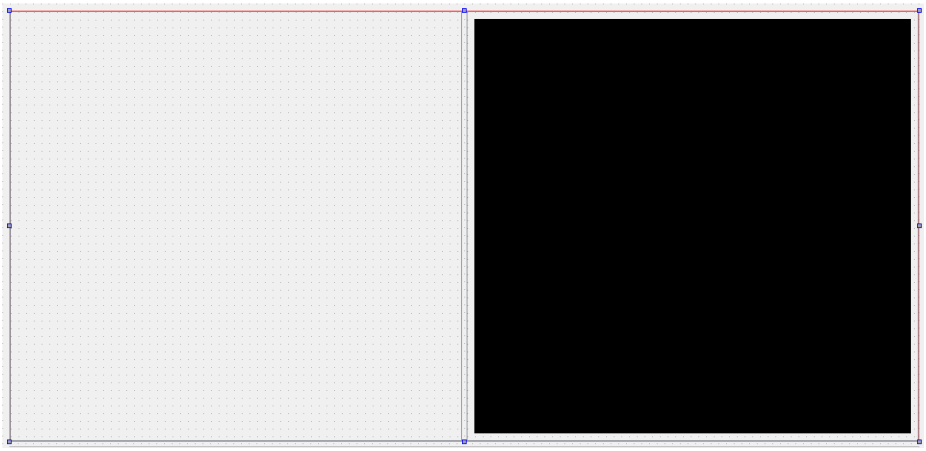


Ilustración 3.51

En segundo lugar, y contenido dentro del anterior, tendremos el conjunto de los dos elementos principales, la zona de análisis de la imagen y la zona de presentación de las imágenes tridimensionales, los cuales estarán separados mediante un espaciado horizontal, lo que hará que ambos elementos formen un solo cuerpo uno al lado del otro.



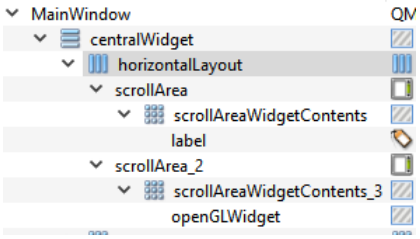


Ilustración 3.52

En tercer lugar, disponemos de los botones, etiquetas y demás elementos que guiarán al usuario a lo largo del proceso, estos se encontrarán dispuestos de manera vertical a lo largo de toda la ventana de la aplicación, siguiendo la disposición de la ventana principal o “MainWindow”.

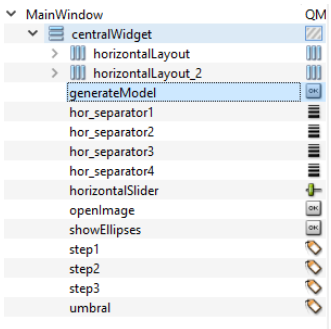


Ilustración 3.53

Y en último lugar, dispondremos de los elementos encargados de modificar los parámetros de las elipses principales; deslizadores horizontales, etiquetas y botones, los cuales se agruparán dentro de un contenedor vertical para cada una de las elipses. A su misma vez, estos dos contenedores espaciados verticalmente se agruparán en uno que se dispondrá de manera horizontal a lo largo de la ventana, dando como resultado lo expuesto en la *ilustración 3.52*.

Ilustración 3.54

Sin embargo, todo lo anterior tampoco dotará de sentido a la aplicación si no se conecta con las demás clases implementadas de la manera correcta, es por eso por lo que se presenta el diagrama de clases expuesto en la *ilustración 3.55*.

En este diagrama podemos observar como toda la aplicación cobra sentido, ya que la clase analizada e implementada anteriormente, "**Renderer**", queda enlazada con la interfaz de usuario, haciendo posible el mostrar las imágenes renderizadas en la ventana de la aplicación; y a su misma vez, la clase "**VaseDetector**", encargada por otro lado de realizar todo el proceso sobre las imágenes, queda también unida a la interfaz, haciendo posible al usuario recibir información visual de las etapas del proceso.

Todo queda unido mediante la clase "**MainWindow**", la cual representa a la ventana principal de la interfaz gráfica, y a su misma vez, esta poseerá un elemento de la clase "**glwidget**", clase que hereda de "**QOpenGLWidget**", y representa a un elemento de la interfaz, propio de la librería "Qt"; haciendo posible que esta quede enlazada a la interfaz de usuario.

Debido a que anteriormente se explicó todo lo relacionado con las clases "**Renderer**" y "**VaseDetector**", en este apartado se hará mayor hincapié en la

clase “**MainWindow**” y en todo lo relacionado con la interfaz de usuario, aclarando algunos de los aspectos más importantes de las demás clases.

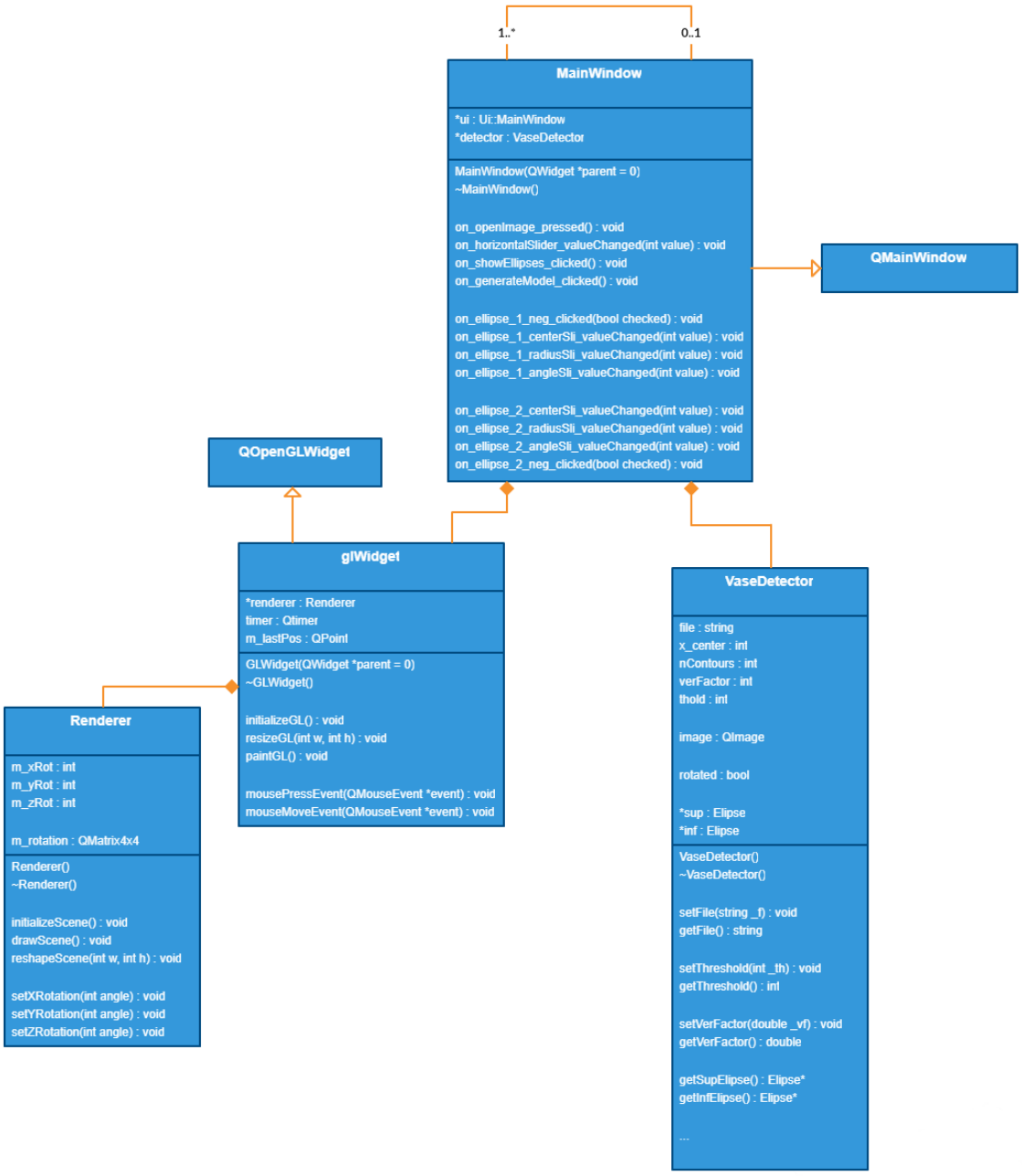


Ilustración 3.55

En el anterior diagrama de clases podemos observar como “**MainWindow**” posee un atributo de tipo “**VaseDetector**”, pero no posee uno de tipo “**glWidget**”; esto está debido a que “**MainWindow**” usa una relación reflexiva consigo misma para crear el objeto “ui”, el cual tiene asociado todos los objetos de la interfaz: botones, deslizadores, etiquetas e incluso los demás “widgets”, todo mediante el

uso de un archivo con extensión ".ui" el cual simplemente codifica la posición y funcionalidad de todos los elementos de la interfaz mediante líneas de código "*xml*". Es por todo esto por lo que esta clase hereda de "**QMainWindow**".

Aclarando algunos conceptos, los "widgets" no son más que elementos que "Qt" aporta a los desarrolladores para que la implementación de sus aplicaciones mediante una interfaz de usuario sea más simple e intuitiva.

Ilustración 3.56

A la hora de implementar todo lo expuesto anteriormente, podemos observar como dicha clase, "**MainWindow**", posee una gran cantidad de funciones con nombres similares; dichas funciones serán llamadas cuando se realice la acción indicada, por ejemplo: la función "on_openImage_pressed" será llamada cada vez que se pulse el botón de la interfaz de usuario llamado "openImage"; la función "on_horizontalSlider_valueChanged" será llamada cada vez que se varié el valor del deslizador horizontal, encargada de variar el valor del umbral; la función "on_ellipse_1_neg_clicked", será llamada cuando la caja de comprobación o "checkbox" sea pulsada con el ratón. Como podemos observar es relativamente fácil intuir cuando serán utilizadas las funciones implementadas.

```
void MainWindow::on_openImage_pressed()
{
    detector->setVerFactor(this->ui->label->height());

    if (this->ui->ellipse_1_neg->isChecked()) this->ui->ellipse_1_neg->click();
    this->ui->ellipse_1_neg->setEnabled(false);

    if (this->ui->ellipse_2_neg->isChecked()) this->ui->ellipse_2_neg->click();
    this->ui->ellipse_2_neg->setEnabled(false);

    this->ui->ellipse_1_angleSli->setEnabled(false);
    this->ui->ellipse_1_radiusSli->setEnabled(false);
    this->ui->ellipse_1_centerSli->setEnabled(false);

    this->ui->ellipse_2_angleSli->setEnabled(false);
    this->ui->ellipse_2_radiusSli->setEnabled(false);
    this->ui->ellipse_2_centerSli->setEnabled(false);

    this->ui->generateModel->setEnabled(false);

    // - We open and process an image from our system.
    QString fname = QFileDialog::getOpenFileName(this, "Open", qApp->applicationDirPath(), "Images (*.j
    std::string aux = fname.toLocal8Bit().constData();
    if (aux != ""){
        detector->setRotated(false);
        detector->setFile(aux);
        ui->label->setPixmap(QPixmap::fromImage(detector->processImage()));
    }
}
```

Ilustración 3.57

Como podemos observar en la *ilustración 3.57*, la cual presenta el código de la función llamada cada vez que se requiere cargar una imagen a la aplicación, primero se deshabilitarán todos los elementos de la interfaz que no pueden ser accedidos en dicho momento, como, por ejemplo, aquellos que permiten variar los atributos de las elipses superior e inferior, ya que estas aún no existen tan siquiera. Tras todo esto se obtendrá la cadena de texto que representa la ruta hasta la imagen dentro de nuestro sistema, para ello se usará un objeto del tipo "**QFileDialog**", lo que nos permite acceder a todos los archivos de nuestro sistema en busca de cualquier imagen que posea alguno de los formatos permitidos. Finalmente se pasará dicha ruta al objeto "detector" de la clase "**VaseDetector**" y se realizará un primer procesamiento de la imagen, como podemos observar en las últimas líneas, mostrando el resultado en el objeto de la interfaz, "label".

Por otro lado, esta clase posee otra función de vital importancia. Se habla de la función encargada de actualizar el valor del parámetro umbral o "thold" de la clase "**VaseDetector**". En primer lugar, se actualizará el texto que indica el valor del umbral actualmente, de esta forma el usuario puede saber el valor del umbral simplemente leyendo la información que se presenta en la ventana. Una vez actualizada la cadena de texto, se actualizará el valor en la clase encargada de analizar la imagen, y se bloqueará el acceso a los elementos de la interfaz cuyos objetos asociados no están instanciados, o que pueden dar lugar a un

error en la aplicación. Finalmente se actualizará la imagen que se presente en la interfaz, para ello se llamará de nuevo a la función encargada de procesar la imagen, aunque esta vez el valor del umbral aplicado sobre esta será diferente. El fragmento de código asociado a dicha función queda expuesto en la *ilustración 3.58*.

```
void MainWindow::on_horizontalSlider_valueChanged(int value)
{
    QString umb("Umbral: ");
    // - Updating the label and the value.
    QString val = QString::number(value);
    umb.append(val);
    ui->umbral->setText(umb);

    detector->setThreshold(value);

    if (this->ui->ellipse_1_neg->isChecked()) this->ui->ellipse_1_neg->click();
    this->ui->ellipse_1_neg->setEnabled(false);

    if (this->ui->ellipse_2_neg->isChecked()) this->ui->ellipse_2_neg->click();
    this->ui->ellipse_2_neg->setEnabled(false);

    this->ui->generateModel->setEnabled(false);

    this->ui->ellipse_1_angleSli->setEnabled(false);
    this->ui->ellipse_1_radiusSli->setEnabled(false);
    this->ui->ellipse_1_centerSli->setEnabled(false);

    this->ui->ellipse_2_angleSli->setEnabled(false);
    this->ui->ellipse_2_radiusSli->setEnabled(false);
    this->ui->ellipse_2_centerSli->setEnabled(false);

    if (ui->label->pixmap() != NULL){
        detector->setRotated(false);
        ui->label->setPixmap(QPixmap::fromImage(detector->processImage()));
    }
}
```

Ilustración 3.58

Otros de los pasos más importantes para el usuario es el de ajustar las elipses superior e inferior, para ello se deberá pulsar un botón el cual hará que estas elipses aparezcan dibujadas sobre la imagen cargada en la aplicación. La función que se encuentra oculta detrás de este proceso se encargará de, en primer lugar, comprobar que hay una imagen cargada en la aplicación, y que se ha encontrado el contorno adecuado para poder llevar a cabo el proceso. Una vez realizada dicha comprobación, se dibujarán las elipses en la imagen que se presenta en la interfaz de usuario, y además se activarán todos los elementos de la interfaz, permitiendo al usuario en este mismo momento modificar los parámetros de las elipses a voluntad. Para ello, se establecerá un valor máximo y otro mínimo a la hora de ajustar dichos elementos, evitando así que el usuario pueda establecer un parámetro fuera de los límites lógicos de la propia aplicación.

Esto se puede observar en la *ilustración 3.59*, en la sección de código marcada con color azul podemos observar como el valor máximo de dicho deslizador se iguala con el de la altura de la propia imagen, y el mínimo, con la mitad de dicho valor; además de establecer un valor por defecto que en este caso se corresponde con cincuenta unidades más el valor mínimo.

```
void MainWindow::on_showEllipses_clicked()
{
    if (ui->label->pixmap() != NULL && detector->getNContours() != 0) {
        // - Show the actual image.
        ui->label->setPixmap(QPixmap::fromImage(detector->drawElipses()));

        // - Update parameters in our user interface.
        if (this->ui->ellipse_1_neg->isChecked()) this->ui->ellipse_1_neg->click();
        this->ui->ellipse_1_neg->setEnabled(true);
        if (this->ui->ellipse_2_neg->isChecked()) this->ui->ellipse_2_neg->click();
        this->ui->ellipse_2_neg->setEnabled(true);

        this->ui->generateModel->setEnabled(true);

        this->ui->ellipse_1_angleSli->setEnabled(true);
        this->ui->ellipse_1_radiusSli->setEnabled(true);
        this->ui->ellipse_1_centerSli->setEnabled(true);

        this->ui->ellipse_2_angleSli->setEnabled(true);
        this->ui->ellipse_2_radiusSli->setEnabled(true);
        this->ui->ellipse_2_centerSli->setEnabled(true);

        this->ui->ellipse_1_centerSli->setMaximum((detector->getImage().height()/2) - 2);
        this->ui->ellipse_1_centerSli->setMinimum(0);
        this->ui->ellipse_1_centerSli->setValue((detector->getImage().height()/2) - 50);

        this->ui->ellipse_2_centerSli->setMaximum(detector->getImage().height());
        this->ui->ellipse_2_centerSli->setMinimum(detector->getImage().height()/2);
        this->ui->ellipse_2_centerSli->setValue((detector->getImage().height()/2) + 50);

        this->ui->ellipse_1_radiusSli->setMaximum(detector->getImage().width()/2);
        this->ui->ellipse_1_radiusSli->setMinimum(0);
        this->ui->ellipse_1_radiusSli->setValue(detector->getSupElipse()->getHorRadius());

        this->ui->ellipse_2_radiusSli->setMaximum(detector->getImage().width()/2);
        this->ui->ellipse_2_radiusSli->setMinimum(0);
        this->ui->ellipse_2_radiusSli->setValue(detector->getInfElipse()->getHorRadius());

        this->ui->ellipse_1_angleSli->setMaximum(50);
        this->ui->ellipse_1_angleSli->setMinimum(0);
        this->ui->ellipse_1_angleSli->setValue(detector->getSupElipse()->getVerRadius());

        this->ui->ellipse_2_angleSli->setMaximum(50);
        this->ui->ellipse_2_angleSli->setMinimum(0);
        this->ui->ellipse_2_angleSli->setValue(detector->getInfElipse()->getVerRadius());
    }
}
```

Ilustración 3.59

Pero de todos los elementos y funciones de las que dispone esta interfaz, sin duda, el elemento, o funcionalidad, más importante es el encargado de generar el modelo en tres dimensiones. Esto se llevará a cabo con un botón, cuyo accionamiento hará que se realice todo el proceso para generar las elipses intermedias, gracias a las dos elipses que se acaban de ajustar en la interfaz. Este proceso devolverá una imagen la cual se presentará de nuevo al usuario, dando a conocer a este que el proceso se ha llevado a cabo correctamente, además, se habrán almacenado en un vector los puntos que conformarán el perfil de revolución del sólido; por lo que simplemente se enviarán dichos puntos al objeto encargado de generar las imágenes tridimensionales, el cual se encuentra

dentro del “widget” propio de “OpenGL”, “glWidget”. Todo este proceso final se puede observar en la *ilustración 3.60*.

```
void MainWindow::on_generateModel_clicked()
{
    std::vector<glm::vec3> points;
    ui->label->setPixmap(QPixmap::fromImage(detector->generateProfile(points)));

    glm::vec3 * array = new glm::vec3[points.size()];
    for (uint i = 0; i < points.size(); i++) {
        array[i] = points[i];
    }
    ui->openGLWidget->renderer->initializeVase(array, points.size(), 3);
}
```

Ilustración 3.60

Además, y como aclaración, se dispondrán de otras funciones, las cuales simplemente se encargarán de actualizar los parámetros de las elipses principales y la imagen que se presenta al usuario; para así realizar una retroalimentación con este, y poder llevar a cabo el proceso de ajuste correctamente.

```
void on_ellipse_1_centerSli_valueChanged(int value);

void on_ellipse_2_centerSli_valueChanged(int value);

void on_ellipse_1_radiusSli_valueChanged(int value);

void on_ellipse_2_radiusSli_valueChanged(int value);

void on_ellipse_1_angleSli_valueChanged(int value);

void on_ellipse_2_angleSli_valueChanged(int value);
```

```
void MainWindow::on_ellipse_1_radiusSli_valueChanged(int value)
{
    detector->getSupElipse()->setHorRadius(value);
    if (ui->label->pixmap() != NULL) {
        ui->label->setPixmap(QPixmap::fromImage(detector->updateElipses()));
    }
}

void MainWindow::on_ellipse_2_radiusSli_valueChanged(int value)
{
    detector->getInfElipse()->setHorRadius(value);
    if (ui->label->pixmap() != NULL) {
        ui->label->setPixmap(QPixmap::fromImage(detector->updateElipses()));
    }
}
```

Ilustración 3.61

Pasamos a hablar ahora de la clase “**glWidget**”, la cual posee algunos atributos interesantes; como un temporizador, la última posición del ratón en la pantalla y el propio objeto de la clase “**Renderer**”. Por un lado, el temporizador servirá para determinar la cantidad de veces en un segundo que se llama a la función de pintado, pudiéndose variar la cantidad de fotogramas por segundo que se generarán.

```
GLWidget::GLWidget(QWidget *parent) : QOpenGLWidget(parent) {
    renderer = new Renderer();

    QSurfaceFormat format;
    format.setSamples(16);
    format.setStereo(true);
    this->setFormat(format);

    connect(&timer, SIGNAL(timeout()), this, SLOT(update()));
    timer.start(17);
}
```

Ilustración 3.62

Como podemos observar en la *ilustración 3.58*, expuesta anteriormente, el temporizador se inicia con un valor entero de diecisiete unidades, esto significa que la función encargada de dibujar la escena será llamada una vez cada diecisiete milisegundos, dando un equivalente de cincuenta y ocho fotogramas, o imágenes, por segundo.

Por otro, la última posición del ratón en la pantalla se almacenará para poder generar interacción entre el usuario y las imágenes renderizadas; de esta forma el usuario podrá rotar la figura en tres dimensiones pulsando y arrastrando el ratón a lo largo de la ventana.

Esto se puede observar en los eventos de ratón implementados por la propia clase, “mousePressEvent” y “mouseMoveEvent”, en las cuales simplemente se actualizará el valor de dicho atributo y se enviará la información necesaria al objeto de la clase “**Renderer**” para que se realicen las transformaciones requeridas, respectivamente.

```
void GLWidget::mousePressEvent(QMouseEvent *event)
{
    m_lastPos = event->pos();
}

void GLWidget::mouseMoveEvent(QMouseEvent *event)
{
    int dx = event->x() - m_lastPos.x();
    int dy = event->y() - m_lastPos.y();

    if (event->buttons() & Qt::LeftButton) {
        renderer->setXRotation(8 * dy);
        renderer->setYRotation(8 * dx);
    } else if (event->buttons() & Qt::RightButton) {
        renderer->setXRotation(8 * dy);
        renderer->setZRotation(8 * dx);
    }
    m_lastPos = event->pos();
}
```

Ilustración 3.63

Finalmente, la propia clase "**Renderer**" ha sufrido algunas variaciones para poder llevar a cabo lo que se acaba de exponer, es decir, se han añadido los atributos necesarios para poder realizar las rotaciones deseadas por el usuario (*m_xRot, m_yRot, m_zRot*), así como una matriz sobre la cual se almacenará dicha transformación, "m_rotation". Además, se han añadido los pertinentes métodos para poder normalizar e ir acumulando las sucesivas rotaciones que pueda sufrir el objeto.

```
void Renderer::setXRotation(int angle)
{
    qNormalizeAngle(angle);
    if (angle != m_xRot) {
        m_xRot += angle;
    }
}

void Renderer::setYRotation(int angle)
{
    qNormalizeAngle(angle);
    if (angle != m_yRot) {
        m_yRot += angle;
    }
}

void Renderer::setZRotation(int angle)
{
    qNormalizeAngle(angle);
    if (angle != m_zRot) {
        m_zRot += angle;
    }
}
```

Ilustración 3.64

Por otro lado, la clase "**VaseDetector**" no ha sufrido ningún cambio remarcable con respecto a lo expuesto en el *apartado 3.2.2*, a partir de ahora, las funciones que dan acceso a sus atributos privados serán usadas por las funciones de la clase "**MainWindow**", las cuales serán llamadas cuando el usuario final interactúe con la interfaz.

3.3. Verificación

Una vez terminada la implementación de la aplicación, habrá que llevar a cabo un proceso de verificación por parte del propio desarrollador y de los supervisores del proyecto, siendo en este caso los tutores del mismo. Este proceso consistirá en comprobar que la aplicación cumple con los requisitos

iniciales del proyecto, detallados en el *apartado 3.1*, además de cerciorarse de que la implementación del mismo se ha llevado a cabo correctamente, para así llevar a cabo el mantenimiento pertinente.

Para esto, y durante el mes de junio del año presente, se tuvo una reunión con los profesores encargados de tutorizar el proyecto, en la cual se encontraron varios errores y mejorías posibles para la propia aplicación, las cuales se expondrán en la siguiente etapa del desarrollo: el mantenimiento.

Ilustración 3.65

3.4. Mantenimiento de la aplicación

Tras haber llevado a cabo la etapa de verificación, durante el mes de julio y agosto fue llevado a cabo el mantenimiento de la aplicación, con el que se pretendía mejorar y pulir algunos detalles de la implementación, así como corregir los errores encontrados durante la anterior etapa, los cuales se detallan a continuación.

En primer lugar, cuando el usuario se encontraba modificando los parámetros de las elipses superior e inferior, y marcaba la casilla que indicaba: “radio vertical negativo”, no se producía ninguna respuesta visual, simplemente dicho parámetro se almacenaba con el valor correspondiente, sin embargo, la modificación de cualquier otro parámetro provocaba un cambio en la imagen que se mostraba en la interfaz.

Ilustración 3.66

En la anterior *ilustración 3.66*, podemos observar como el radio horizontal de la elipse amarilla, o superior, se ha modificado por el usuario, ofreciendo a este una respuesta en tiempo real. Para corregir este inconveniente, se adoptó una política muy simple, y es que cuando la casilla que se encarga de determinar el ángulo, o radio vertical, negativo, es marcada, la elipse pasará de estar hueca a estar rellena del color correspondiente.

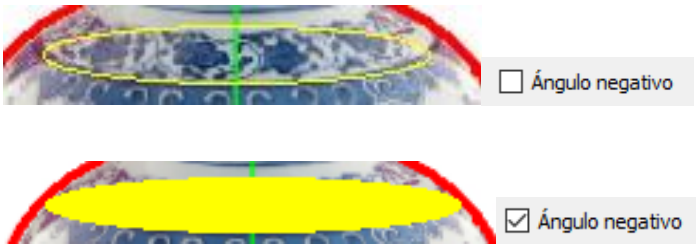


Ilustración 3.67

De esta forma, será mucho más simple para el usuario final realizar el ajuste de las elipses principales.

Por otra parte, el proceso de análisis de la imagen ha sufrido grandes cambios, ya que cuando se cargaba una imagen, y se pasaba de seleccionar el valor del umbral a mostrar las elipses, el eje de revolución aparecía siempre de manera vertical en la imagen, independientemente de la orientación que poseyera el objeto que se encontraba en ella. Por esto, simplemente se corrigió dicho problema, haciendo que el eje variase su orientación dependiendo de la del jarrón.

Además, una vez ajustado el umbral, al pulsar el botón encargado de mostrar las elipses principales en la imagen, esta se rotará para que la orientación sea completamente vertical, haciendo que el proceso de ajuste de las elipses principales, y más tarde, la generación de las elipses secundarias, sea mucho más preciso.

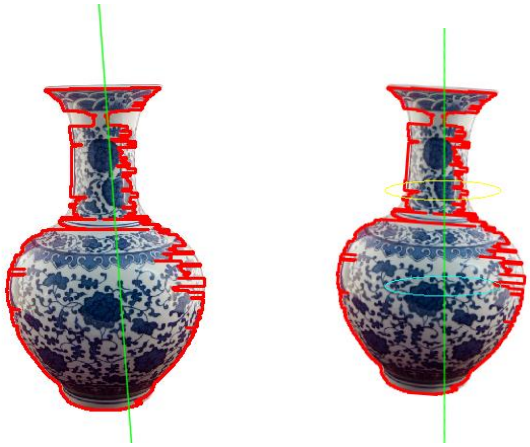


Ilustración 3.68

En la *ilustración 3.68* se aprecia como el eje de revolución de la figura de la izquierda está levemente inclinado, sin embargo, en la figura de la derecha se encuentra completamente vertical.

Otro de los grandes cambios que ha sufrido el proceso de análisis de la imagen tiene que ver con el propio proceso para obtener los puntos del perfil de revolución. En un principio estos puntos se obtenían de una manera errónea y poco precisa; el método seguido consistía en lanzar líneas horizontales por toda la imagen e ir iterando sobre estas, de tal forma que al encontrar el borde del contorno el punto se guardaba en un vector con los demás puntos del perfil de revolución. El problema de seguir este proceso es que la figura generada es errónea, aunque parecida, ya que la transformación de perspectiva no se deshace correctamente. Esto fue corregido siguiendo el proceso que se explica en el *apartado 3.2.2.3*.

Ilustración 3.69

Como bien se puede apreciar en la *ilustración 3.69*, la elipse ajustada, usando el método erróneo, queda por fuera del propio contorno del jarrón, lo que provocará que el modelo obtenido tridimensionalmente, usando dicho punto, sea pobremente fiel al objeto real.

Ilustración 3.70

Sin embargo, y como se puede apreciar en la *ilustración 3.70*, siguiendo el método correcto, la elipse queda perfectamente ajustada al contorno del jarrón, con lo que el perfil de revolución obtenido será mucho más fiel al que posee el objeto real.

Finalmente, otro de los aspectos a corregir de la aplicación estaba relacionado con la variedad de resoluciones de las distintas imágenes. Y es que el problema venía a la hora de mostrar las imágenes en la interfaz, dando lugar por un lado a imágenes de un tamaño muy pequeño, y por otro a unas de un tamaño exagerado. Para corregir este inconveniente se aplicó una transformación al tamaño de las imágenes gracias a la librería “OpenCV”, de tal forma que todas mantienen un tamaño similar y su propia relación de aspecto.

```
cv::resize(im, im, Size(), (double)verFactor/(double)im.rows, (double)verFactor/(double)im.rows, INTER_CUBIC);
```

Ilustración 3.71

Mientras que la imagen de la figura de la izquierda posee una resolución de 1200x1200 píxeles, la de la derecha posee una resolución de 200x252 píxeles, sin embargo, ambas aparecen con tamaños similares dentro de la aplicación. Esta corrección aporta comodidad al usuario, ya que anteriormente las imágenes se cargaban a la aplicación con la resolución que estas poseían originalmente, y si la imagen cargada era muy grande había que desplazarse por ella al igual que en un documento o en una página web; lo cual se convertía en un verdadero inconveniente a la hora de realizar todo el proceso.

De esta forma, y tras haber llevado a cabo las anteriores correcciones, la aplicación estaría lista para ser desplegada. Cumpliendo los requisitos y objetivos propuestos al inicio del proyecto.

4. MANUAL DE USUARIO Y GUÍA DE INSTALACIÓN

En este apartado número cuatro, se pretende explicar qué es necesario para poder compilar y ejecutar el proyecto que se ha detallado con anterioridad, así como redactar un pequeño manual de usuario de forma que sea más simple el uso de la aplicación por parte de cualquier usuario.

4.1. Guía de instalación

Antes de comenzar con la guía, cabe destacar que la aplicación ha sido desarrollada con el sistema operativo “Windows 10” de 64 bits, por lo que no se asegura el correcto funcionamiento en cualquier otro sistema operativo.

En primer lugar habrá que instalar el entorno de desarrollo, en este caso se trata de “Qt Creator 4.2.1”, bastará con descargar el instalador desde la página web oficial. En un determinado momento, a lo largo de la instalación, podremos elegir que elementos instalar, habrá que seleccionar, como mínimo, “Qt 5.8” y

dentro de “Tools”, seleccionar el compilador “MinGW 32 bits”. Tras este proceso el entorno de programación se habrá configurado correctamente.

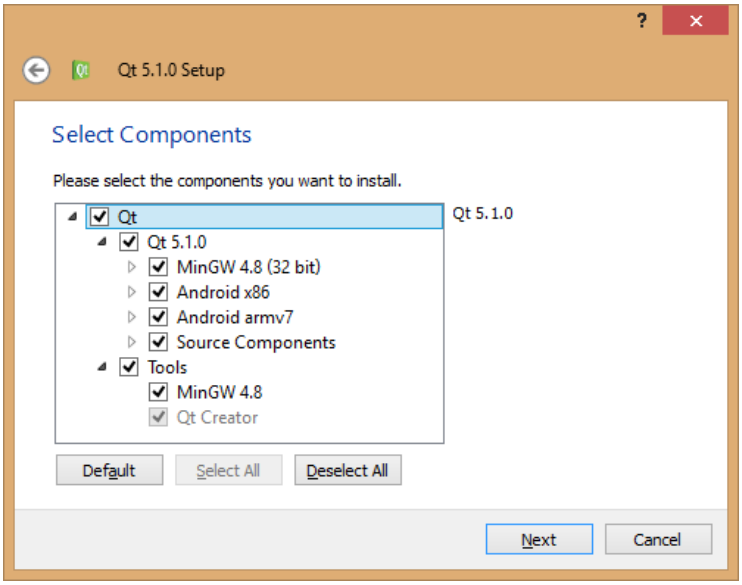


Ilustración 4.1

(The Qt Company, 2017)

Después, simplemente habrá que abrir el proyecto desde el IDE, haciendo uso del archivo “TFG.pro”.

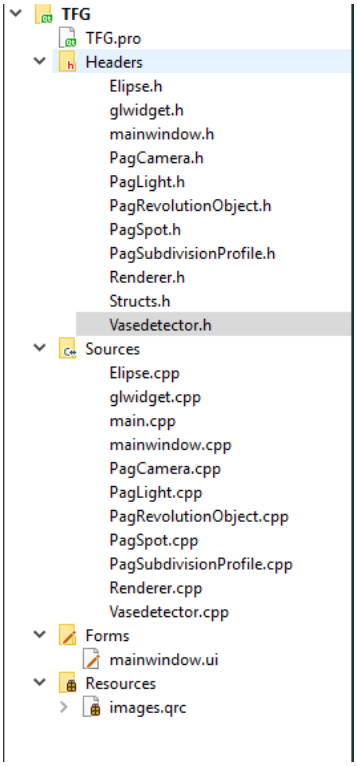


Ilustración 4.2

Como paso final, antes de compilar todo el proyecto, habrá que abrir el archivo “TFG.pro” desde el navegador de archivos propio del IDE, como se puede observar en la *ilustración 4.2*. En este archivo, habrá que establecer correctamente las rutas pertenecientes a la librería “OpenCV”, las cuales dependen del directorio en el que se encuentre alojado el proyecto, ya que la librería está contenida dentro del propio proyecto. Esto está indicado en la *ilustración 4.3*, en mi caso, el proyecto se encuentra en el “escritorio”.

```
INCLUDEPATH += C:\Users\David\Desktop\TFG\opencv\include
LIBS += -LC:\Users\David\Desktop\TFG\opencv\x86\mingw\lib \
    -lopencv_core320.dll \
    -lopencv_highgui320.dll \
    -lopencv_imgcodecs320.dll \
    -lopencv_imgproc320.dll \
    -lopencv_features2d320.dll \
    -lopencv_calib3d320.dll
```

Ilustración 4.3

Finalmente, simplemente habrá que compilar el proyecto, y ejecutar la aplicación.

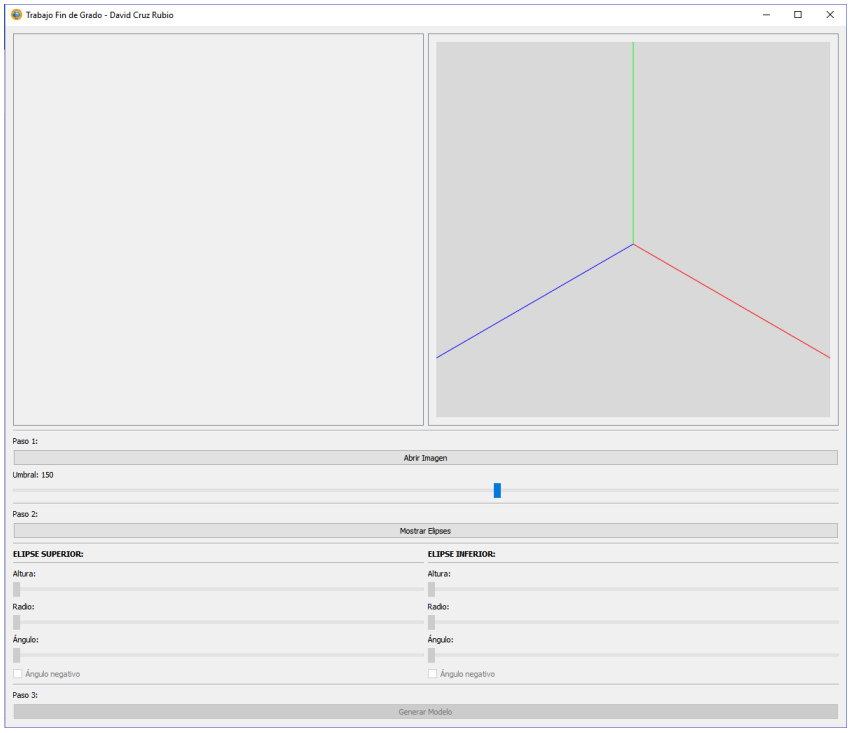


Ilustración 4.4

4.2. Manual de usuario

Además de poder realizar la guía de instalación, se adjunta con el propio proyecto una versión ejecutable de la aplicación desarrollada para así poder ejecutarla sin problemas, tan solo habrá que ejecutar el archivo ejecutable “application” que se encuentra dentro de la carpeta “TFG_application”.

En primer lugar, habrá que cargar alguna imagen, para ello se tendrá que pulsar el botón “Abrir Imagen” en la interfaz de usuario, lo que abrirá un explorador de archivos. Tras seleccionar la imagen deseada, esta aparecerá en la parte izquierda de la ventana principal de la interfaz de usuario.

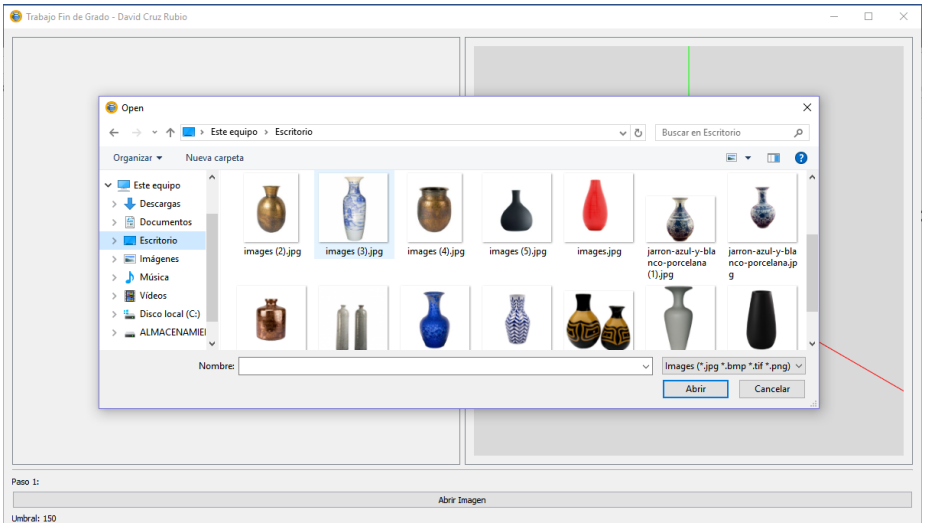


Ilustración 4.5

Una vez se haya seleccionado la imagen, habrá que ajustar el valor del umbral que se va a aplicar a la misma, para ello se dispone de un deslizador justo debajo del botón anterior.

Si el umbral posee un valor muy pequeño, es muy probable que el contorno que se encuentre al procesar la imagen no sea correcto, esto se puede observar en la *ilustración 4.6*.

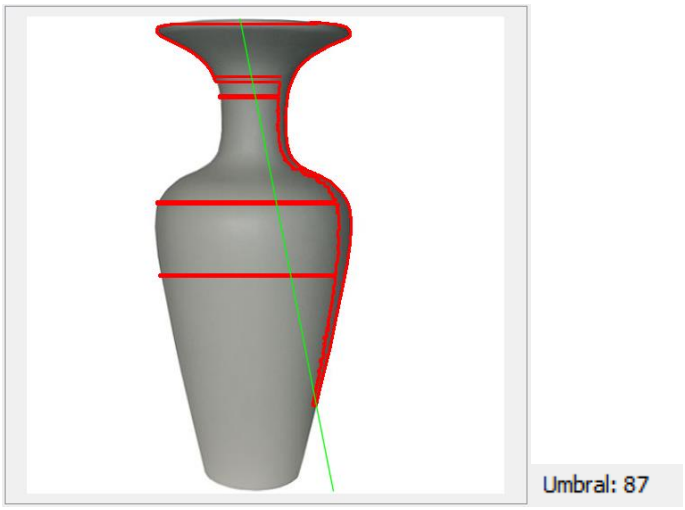


Ilustración 4.6

Por lo que habrá que ajustar el umbral hasta que el contorno encontrado en la imagen sea el deseado por el usuario, esto se aprecia en la *ilustración 4.7*, en la que el contorno encontrado se corresponde enteramente con el jarrón.

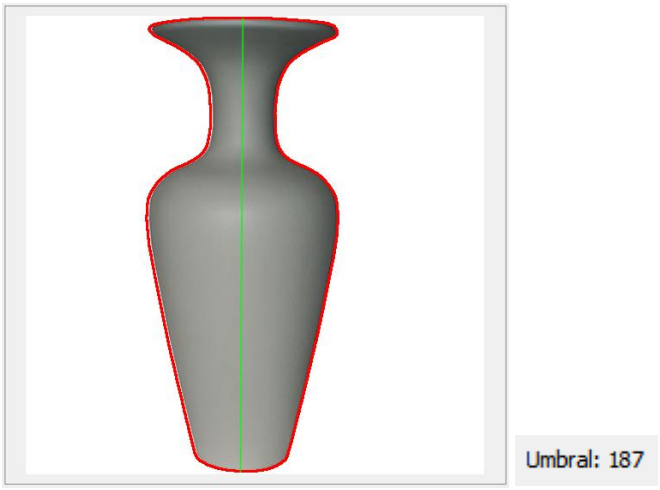


Ilustración 4.7

Cuando se haya encontrado un valor para el umbral correcto, habrá que proceder a ajustar las elipses principales, superior e inferior, haciéndolas coincidir con las tapaderas superior e inferior del jarrón. Pero primero habrá que hacer que estas aparezcan dibujadas en la imagen pulsando el botón “Mostrar Elipses”, lo cual hará aparecer dos elipses ajustadas por defecto en la imagen.

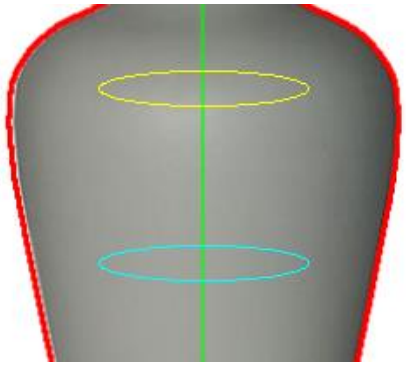


Ilustración 4.8

Con las elipses expuestas explícitamente en la imagen, los demás elementos de la interfaz estarán desbloqueados para que el usuario pueda modificar los parámetros de dichas elipses, y así hacerlas coincidir con las tapaderas superior e inferior.

Ilustración 4.9

Finalmente, solo habrá que pulsar el botón “Generar Modelo” para obtener la representación tridimensional del jarrón en cuestión, la cual se mostrará en la parte derecha de la ventana principal de la interfaz de usuario. El usuario podrá usar el ratón para rotar el modelo obtenido, pudiendo así establecer distintos puntos de vista sobre la figura.

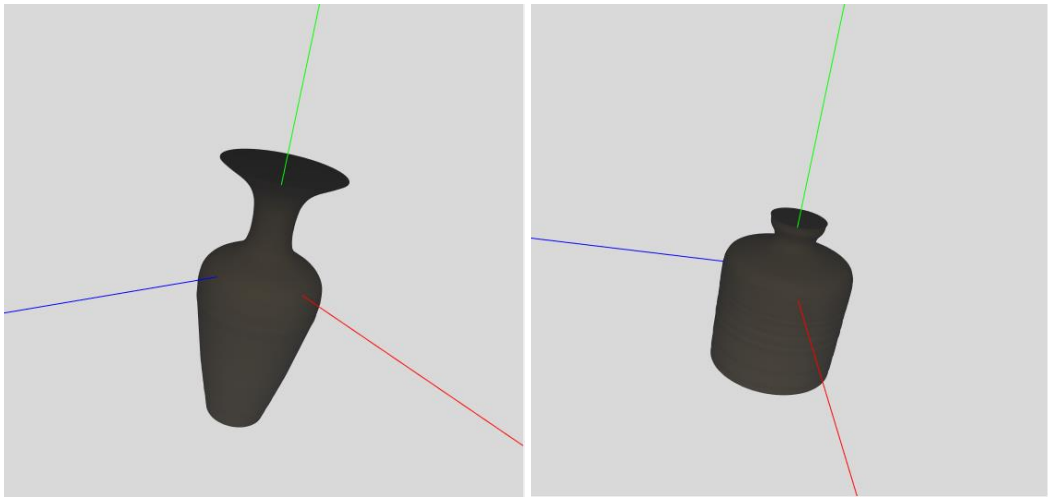


Ilustración 4.10

5. CONCLUSIONES Y TRABAJO FUTURO

Durante la etapa de verificación también se habló sobre muchas de las posibles mejoras que se podrían llevar a cabo sobre este proyecto, algunas simplemente se dirigían a mejorar aspectos visuales de la misma, mientras que otros suponían grandes variaciones sobre la aplicación.

En primer lugar, y siendo desde mi punto de vista el aspecto más importante sobre el que trabajar futuramente, están las restricciones del entorno; como bien se dijo en apartados anteriores, es necesario que el fondo de la fotografía, o imagen, sea blanco; así como que los objetos expuestos en las fotografías no arrojen sombras, evitando así que estas influyan en el análisis que se debe llevar a cabo. Estos requisitos son muy exigentes, ya que habría que disponer de una perfecta iluminación para evitar las sombras, o llevar a cabo retoques fotográficos para eliminarlas.

Es por esto por lo que se podría llevar a cabo un preprocesamiento de la imagen, mediante el cual se eliminen las sombras que arrojan los objetos, además de establecer el color del fondo a blanco.

Sin embargo, este proceso no es trivial ni simple desde un punto de vista algorítmico y analítico, ya que no habría certeza a la hora de diferenciar el fondo de la imagen, o la sombra, y el propio jarrón que se usará para llevar a cabo todo el procedimiento. Por lo que, por motivos de simpleza, se ha evitado tratar este asunto durante el desarrollo del proyecto, teniendo siempre en cuenta que es un aspecto sujeto a grandes cambios y mejoras.

Gracias a este posible cambio en la aplicación, se pretendería mejorar la aplicación, de tal forma que los usuarios pudiesen realizar fotografías a objetos de su día a día, sin importar las condiciones del entorno, y rápidamente, tras realizar unos pequeños ajustes, obtener el modelo en tres dimensiones del objeto en cuestión.

Ilustración 5.1

Además, la interfaz de la aplicación podría albergar más funcionalidad encargada de modificar parámetros propios de la escena en tres dimensiones, como la iluminación o el propio material del objeto presentado; lo cual sería bastante simple de implementar, aunque se ha obviado por motivos de tiempo y funcionalidad, debido a que es algo meramente visual y que no repercute en la fiabilidad del modelo obtenido.

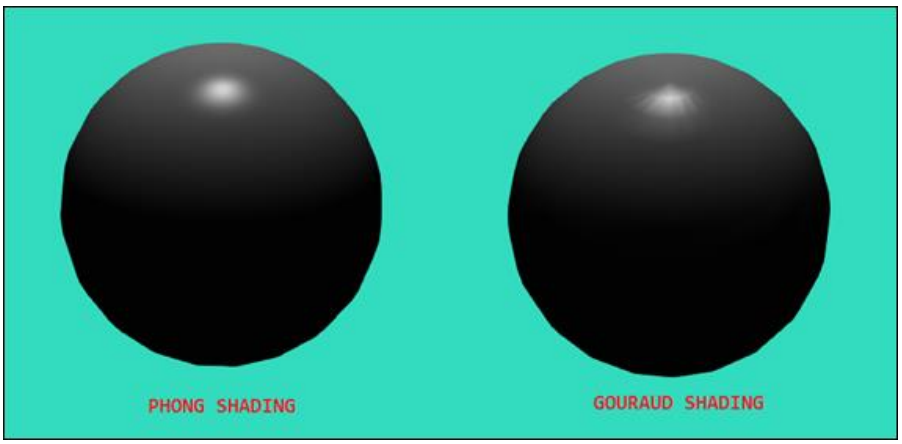


Ilustración 5.2

En la anterior *ilustración 5.2*, se presentan las diferencias entre usar el modelo de iluminación de “Gouraud” y el de “Phong”, siendo este uno de los aspectos que el propio usuario final podría modificar a voluntad.

Por otro lado, un aspecto muy interesante a implementar sería el de texturizar el objeto representado tridimensionalmente gracias a la propia imagen analizada. Este proceso no es extremadamente complejo, ya que simplemente habría que usar los píxeles del contorno encontrado y almacenarlos como una imagen independiente; tras esto, simplemente habría que aplicarla sobre el objeto representado, gracias a que este posee también la información de las coordenadas de textura necesarias para dicha operación.

Ilustración 5.3

Finalmente, como conclusión sobre el trabajo, decir que es un proyecto que engloba muchos aspectos matemáticos y físicos, como bien se ha expresado en los anteriores apartados, lo que lo convierte en un verdadero reto a la hora de realizar una adecuada, y eficiente, implementación del mismo.

Por otro lado, uno de los objetivos principales, expuesto al inicio del proyecto por parte de los tutores, era el de llevar a cabo una aplicación atractiva y usable, sobre todo por aquellas personas relacionadas con la arqueología, lo cual se ha cumplido en cierta medida. Sin embargo, esto se habría cumplido en su totalidad, si se hubieran implementado algunas de las mejoras expuestas en este mismo apartado, aunque algunas de ellas hubieran sido aptas como temática para un proyecto independiente.

Como resultado del trabajo, se ha obtenido una aplicación atractiva visualmente, además de sencilla, aunque con un gran margen de mejoría hasta que esta pueda tener algún interés científico o económico, más allá del meramente intelectual.

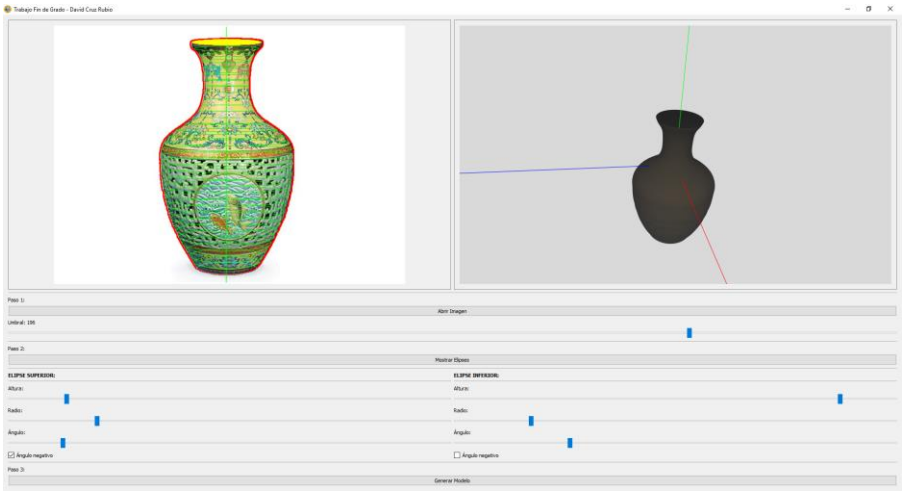


Ilustración 5.3

Bibliografía

Cinergix Pty. Ltd. (2017). Obtenido de Creately: https://creately.com/

Conde, F. R. (2017). *PAG 17-02-Geometria.*

G-Truc Creation. (2005).

Intel. (1999).

Khronos Group. (2017).

Ortega, M. (1989-2006). *Lecciones de física.* Monytex.

Pernici, F. (2005). Metric 3D Reconstruction and Texture Acquisition of Surfaces of Revolution from Single Uncalibrated View. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Royce, W. W. (1970).

Smith, M. (2003). *Human-Computer Interaction: Theory and Practice.*

Stroustrup, B. (1980).

svpenkov. (2013). *OBJECT ORIENTATION, PRINCIPAL COMPONENT ANALYSIS & OPENCV.* Obtenido de robospace: https://robospace.wordpress.com/2013/10/09/object-orientation-principal-component-analysis-opencv/

The Qt Company. (2017).

Weisstein, E. W. (1999-2017). *Solid of Revolution.* Obtenido de MathWorld: http://mathworld.wolfram.com/SolidofRevolution.html