Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Generación procedural de modelos tridimensionales de naves espaciales

Trabajo fin de Máster

Máster Universitario en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Autor: Joaquim Àngel Montell Serrano
Tutor: Francisco José Abad Cerdá

Curso 2016-2017

Resum

El creixement desmesurat de la indústria gràfica, ha provocat que cada vegada es necessiti més contingut i de millor qualitat. A més, la generació d'aquest contingut és molt laboriosa i costosa, donant lloc a empreses dedicades enterament a aquesta tasca. En aquest treball, s'ha proposat el investigar com fer una eina de disseny 3D utilitzant tècniques de generació per procediments per ajudar els creadors d'aquest contingut

Paraules clau: Generació per procediments, OpenGL, Modelatge 3D, Qt

Resumen

El crecimiento desmesurado de la industria gráfica, ha provocado que cada vez se necesite más contenido y de mejor calidad. Además, la generación de este contenido es muy laboriosa y costosa, dando lugar a empresas dedicadas enteramente a esta tarea. En este trabajo, se ha propuesto el investigar cómo hacer una herramienta de diseño 3D utilizando técnicas de generación por procedimientos para ayudar a los creadores de este contenido

Palabras clave: Generación por procedimientos, OpenGL, Modelado 3D, Qt

Abstract

The excessive growth of the graphic industry has led to the need for more content with better quality. In addition, the generation of this content is very laborious and expensive, giving rise to companies dedicated entirely to this task. Here, we will investigate how to make a 3D design tool using techniques of procedural generation to help the creators of this content

Key words: Procedural generation, OpenGL, 3D Modeling, Qt

Índice general

CAPÍTULO 1
Introducción

1.1 Motivación

Uno de los grandes costes de la industria creativa que usa herramientas digitales consiste en la generación de modelos. Debido a la mejora en las técnicas de *renderizado*, cada vez se piden unos modelos con más nivel de detalle, llegando al punto en el que no se pueda distinguir entre gráficos de ordenador y mundo real.

De esta misma forma, sobre todo en la industria de los videojuegos, se están pidiendo mundos cada vez más grandes llegando a tener escala real. Algunos ejemplos podrían ser juegos como las sagas de *Grand Theft Auto* o *The Elder Scrolls*. Los mundos de estos juegos llegan a tener una gran cantidad de modelos para que exista algo de variedad y que no cree una experiencia repetitiva.

Estos dos factores hacen que trabajar en este sector requiera de un gran número de empleados dedicándose a la tarea exclusiva de generar contenido gráfico reduciendo el presupuesto en otras áreas como, por ejemplo, el desarrollo de mecánicas complejas o de una trama más elaborada.

Por otro lado, tenemos las llamadas empresas *indie* que suelen constar de un grupo muy reducido de personas y con poco presupuesto. Este colectivo se suele caracterizar como el opuesto de las grandes compañías, dicho de otro modo, son empresas que invierten más en las mecánicas o en el desarrollo de la historia aunque terminen con unos gráficos más pobres.

La tecnología investigada en este trabajo es de especial interés para que estas empresas con menos personal puedan generar más y mejores modelos tridimensionales.

1.2 Objetivos

En este trabajo fin de máster se plantean dos objetivos principales:

- Investigar sobre la generación de modelos de naves mediante algoritmos
- Creación de una plataforma para el diseño asistido de modelos de naves espaciales

1.3 Estructura de la memoria

Esta memoria se divide en 7 partes:

- Introducción: Se declaran la motivación, los objetivos del proyecto y se realiza esta descripción de la memoria.
- Estado del arte: Se describen las aplicaciones y algoritmos existentes en este campo y algunos de otros campos relacionados.
- Tecnología utilizada: Donde se describe que tecnologías se han utilizado y la funcionalidad que ofrecen.
- Metodología: Definiremos el trabajo que se ha realizado y como se ha estructurado
- Evaluación de resultados: Se realiza una comparación entre los resultados obtenidos y los objetivos propuestos
- Conclusiones: Se hablará de los mecanismos que se han utilizado para comparar nuestro programa con otros existentes y se hará una reflexión sobre los resultados obtenidos
- Propuestas de futuro: Se describirán distintas mejoras e implementaciones interesantes para incluir.
- Anexos: Se han incluido aquellos aspectos relevantes de las aportaciones realizadas en este trabajo de fin de máster. Estos anexos son:
  - Anexo A: La gramática completa descrita en el apartado 4.2.
  - Anexo B: La librería que se ha desarrollado para el manejo de las curvas de Bézier. Se entrará en más detalle en el apartado 4.4.
  - Anexo C: Código relacionado con la aportación realizada para la teselación de la malla. La explicación de este algoritmo la podemos encontrar en el apartado 4.4.3.
  - Anexo D: Código desarrollado para la subdivisión de una cinta en las distintas formas seleccionadas. Se ha explicado este algoritmo en el apartado 4.4.3.

CAPÍTULO 2
Estado del arte

La generación de contenido por procedimientos (PCG, del inglés *Procedural Content Generation*) hace referencia a la generación de contenido de forma autónoma o con una intervención limitada por parte del usuario. Generalmente, se han utilizado estas técnicas para generar terrenos o niveles pero también se han dado casos en otras áreas. A continuación, describiremos brevemente la historia de la PCG.

2.1 Breve historia de la PCG

Uno de los primeros usos de la PCG fue por la década de los 70. En este periodo de tiempo se crearon dos juegos similares. El primero se llama *Beneath the apple manor* y se creó en 1978 mientras que el segundo es *Rogue* que tuvo tanta fama que dio lugar a todo un género de juegos conocidos como *Roguelike*. La característica que los hace similares es, sin duda, la mecánica del juego. Como jugador, debes recorrer una mazmorra que se genera cada vez que empiezas la partida. Esto solucionaba uno de los grandes problemas de la época que consistía en la falta de espacio puesto que no necesitaba almacenar los distintos niveles. De este mismo modo, se conseguía que cada partida fuese única y diferente, lo que causó, en gran medida, su éxito.

Mas adelante, se crearon juegos con mecánicas para generar los objetos dándoles estadísticas aleatorias o, incluso, generando el modelo de tal forma que distintas partes aportan distintos modificadores. Con esto se consigue dar al jugador un equipo que pueda estar totalmente personalizado a su forma de jugar. Como posibles juegos de esta categoría tenemos los de la saga *Diablo* (años 1996, 2000 y 2012) o saga *Borderlands* (años 2009, 2012 y 2014).

Otro tipo de juegos que han tenido una gran expansión en los últimos años donde se hace un gran uso de la PCG son los *Survival sandbox*. Estos juegos permiten al jugador modificar un mundo generado aleatoriamente, generalmente compuesto de una unidad mínima llamada *voxels*, con total libertad, al mismo tiempo que presenta un reto para la supervivencia (el jugador tiene que buscar comida, defenderse de ciertos enemigos o, incluso, contra otros jugadores). Quizás, el juego más famoso de esta categoría sea el *Minecraft* (2011) pero se han creado muchos mas juegos de este estilo, algunos, como el *Terraria* (2011) que presenta el mismo concepto pero más orientado al combate, o juegos que presentan la misma temática pero cambiando la ambientación, como el *Rust* (2013). Hay un juego en especial, llamado *No Man's Sky* (2016), de esta categoría que ha intentado ir más allá añadiendo, a parte del terreno, modelos de fauna y flora generados por procedimientos.

A parte de estos dos grandes grupos, la PCG se ha utilizado en otros campos de forma dispersa como en el *Left 4 dead* (2008) donde se controlaba la generación de recursos y enemigos para mantener el interés del jugador o *Spore* donde, al ser el usuario el que crea el modelo, se necesita generar las animaciones al vuelo.

Además de su uso en los videojuegos, la PCG también ha tenido su influencia en las películas con una técnica nombrada "fábrica imperfecta"(*imperfect factory* en ingles). Con esta técnica, el usuario puede generar un modelo y aplicarle modificaciones para crear un sinfín de objetos similares para dar una gran profundidad a las escenas.

2.1.1. Fractales, ruido y *L-systems*

Las técnicas más utilizadas en la PCG son el uso de fractales, ruido o los llamados L-systems[1]. A continuación, explicaremos brevemente en qué consiste cada uno de ellos.

En las matemáticas, los fractales son objetos geométricos con una estructura que se repite a distintas escalas. Esta propiedad es llamada autosimilitud que, dependiendo del grado de similitud de los fragmentos al todo, se clasifican en tres tipos: autosimilitud exacta, cuasiautosimilitud, autosimilitud estadística.

La idea básica que surgió para aplicar los fractales a la PCG se encuentra en la pregunta de cómo generar montañas. Si observamos una piedra suelta de una montaña y la comparamos con la misma, podemos observar que comparten una estructura similar. A partir de esta idea, han surgido una serie de algoritmos como son:

- Desplazamiento del punto medio: algoritmo para dos dimensiones, que se puede adaptar para tres y que consiste en desplazar el punto medio de un segmento en una cantidad aleatoria dentro de un rango, que se reducirá en cada iteración. Este proceso lo repetiremos hasta que la longitud de los segmentos sea suficientemente pequeña

- *Diamond-Square*: Algoritmo donde se parte de un cuadrado y que, cada iteración se divide en dos fases. La primera fase es la llamada *diamond* y consiste en generar la altura del punto medio de las diagonales de cada cuadrado. Esto lo hacemos calculando la media de las cuatro esquinas y sumándole un numero aleatorio como en el anterior algoritmo.

  Para la siguiente fase, *Square*, calculamos la altura de los puntos que se encuentran en el punto medio de las aristas. Para ello, utilizamos la misma técnica que en el paso anterior pero utilizando, además, los puntos generados. Para terminar, repetimos reduciendo el rango aleatorio en cada iteración.

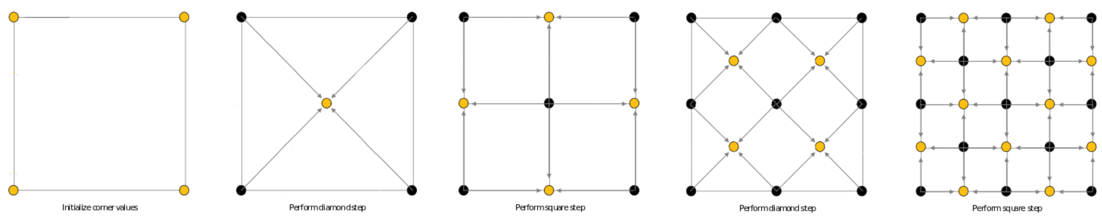

Figura 2.1: Ejemplo de las fases del algoritmo Diamond-Square para una rejilla de 5x5

La siguiente técnica consiste en generar un ruido pseudoaleatorio utilizando una función matemática. El primero en utilizar esta técnica fue Ken Perlin que creo la función de *Perlin noise*[2] para la generación de texturas. Con esta función se generan valores continuos pero con una apariencia aleatoria. El éxito que ha obtenido se debe a su gran versatilidad dando la posibilidad de trabajar con tantas dimensiones como se quiera. Algunos ejemplos de su uso podrían ser la película *Tron* para la cual fue creada o el juego Minecraft donde se utiliza tanto para generar el terreno como las nubes[8].

Mas adelante, Perlin creó una función llamada *Simplex noise* que eliminaba los artefactos que generaba su versión anterior. Además, tenia un coste computacional más bajo y escala mucho mejor con un número mayor de dimensiones $O(n^2)$ frente a $O(2^n)$. Perlin, patentó esta función con lo que surgió una versión libre llamada *OpenSimplex noise*.

La última técnica mencionada, L-systems o un sistema de Lindenmayer, es un tipo de gramática formal consistente de un alfabeto de símbolos, una serie de reglas de producción, el axioma inicial y un mecanismo para traducir las cadenas que se generan a estructuras geométricas. Lindenmayer utilizó este sistema como forma de describir el crecimiento de organismos multicelurares.

Estas gramáticas al tener una naturaleza recursiva, dan lugar a la autosimilitud con lo que facilitan la descripción de fractales. Por otro lado, la plantas, se pueden definir fácilmente con estas estructuras puesto que, al añadir más niveles de recursión, se genera el efecto de que la planta crece. Más usos que se le han dado a estas gramáticas han consistido en el modelado por procedimientos de ciudades[7] y en el de edificios[3]

2.2 Ciudades

La creación de una red de carreteras para una zona urbana es una tarea compleja dado que presenta una estructura artificial pero, a la vez, tiene una serie de restricciones dadas por el terreno, el tipo de distrito (comercial, residencial ...), etc.

Uno de los primeros en proponer un sistema para su generación fueron Parish y Müller [7]. Ellos proponen una solución basada en un *L-System*. Empezando con un único segmento de calle, se van añadiendo más segmentos hasta construir el mapa completo, de forma similar a la creación de un árbol.

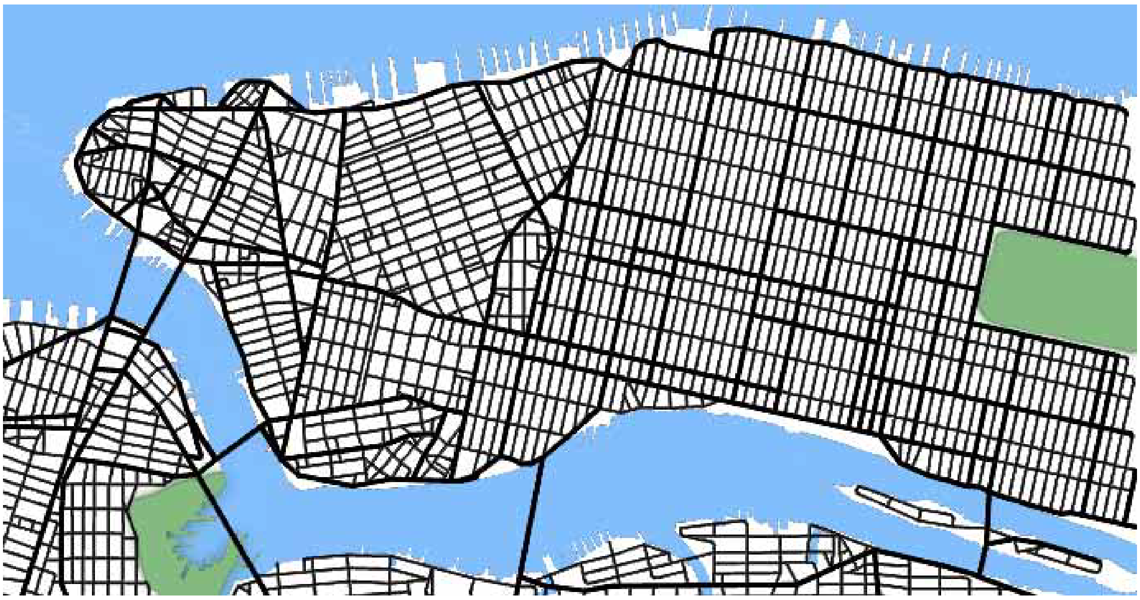

Figura 2.2: Ejemplo del algoritmo de Parish y Müller. Imagen extraída de [7]

El siguiente método consiste en utilizar una serie de agentes para esta tarea por parte de Thomas Lechner et al. Para esta tarea, se utilizan dos tipos de agentes: extensores y conectores. Los primeros se encargan de crear nuevos segmentos de carreteras. Los segundos tienen como tarea crear calles secundarias. Para ello, elije un punto aleatorio en el camino y, en caso de no poder alcanzarlo en una distancia máxima, crea un nuevo camino.

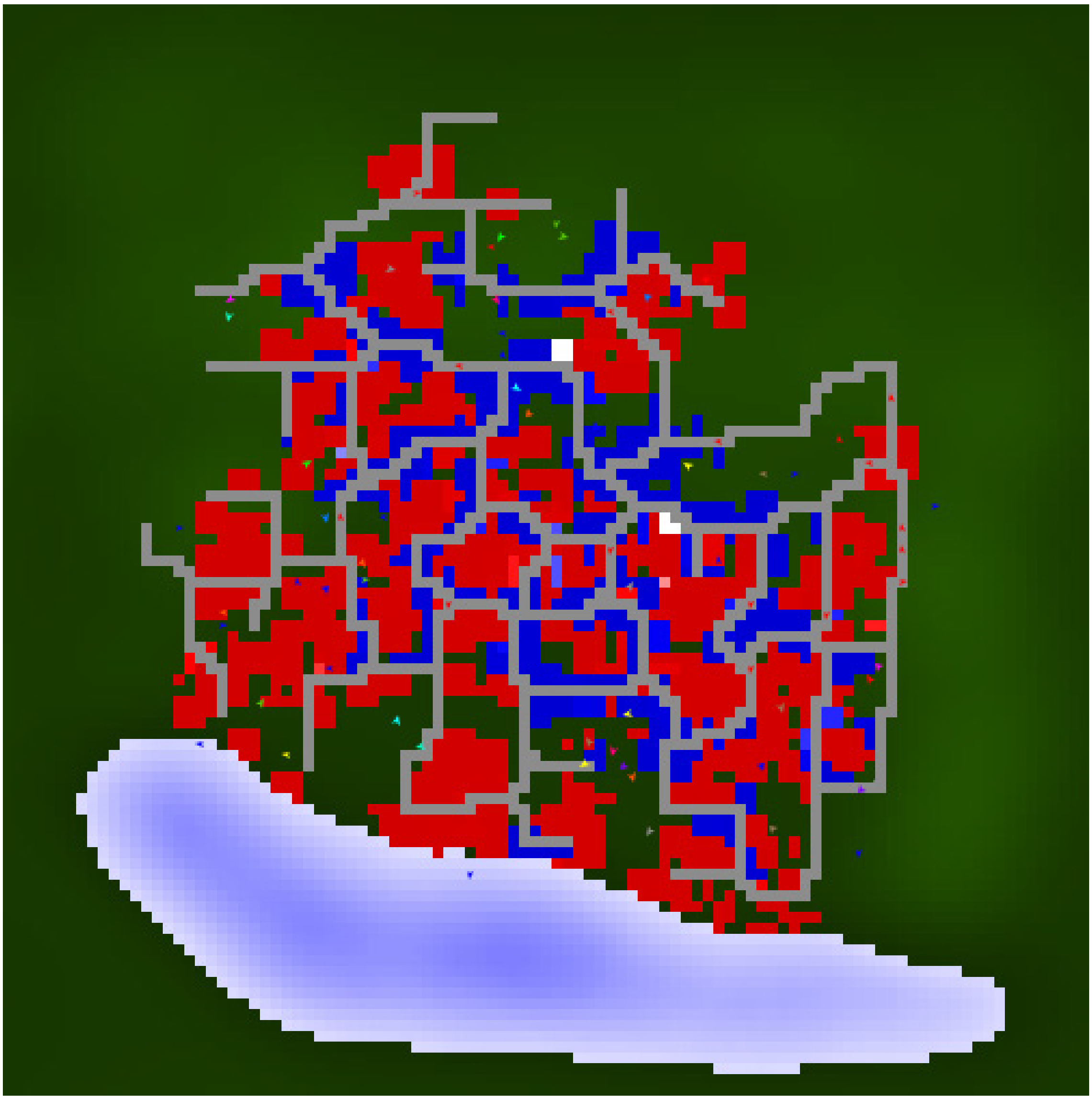

Figura 2.3: Ejemplo del algoritmo de Thomas Lechner et al. Imagen extraída de [5]

2.3 Edificios

Mientras que Parish y Müller trataban de generar una ciudad con edificios utilizando operaciones booleanas entre formas simples, Wonka et al. intentaban dar detalle a las fachadas de los edificios[4]. Para ello, crearon un nuevo tipo de gramáticas llamado *split grammars* basadas en las *shape grammars*. De esta forma, se conseguía dar coherencia vertical en, por ejemplo, balcones y material o coherencia horizontal en el estilo de las ventanas.

Sin embargo, este algoritmo solo es práctico para modelos sencillos. En el caso de un modelo más complejo, esto no funcionaría debido a la posibilidad de solapes entre los distintos volúmenes del edificio como se puede observar en 2.5. Para estos casos Müller et al. [3], proponen un algoritmo donde solucionan dos problemas. Primero, la creación

Figura 2.4: Ejemplo del algoritmo de Wonka et al. Imagen extraída de [4]

del volumen de este edificio y, segundo, las limitaciones del anterior algoritmo. Por un lado, utilizan una gramática de formas con una librería para generar las estructuras y, una vez generada la fachada, utilizan un sistema de control donde modifican este resultado para hacer encajar las distintas estructuras generadas al espacio disponible.

Figura 2.5: Ejemplo del algoritmo de Müller et al. Derecha, problemas con los algoritmos tradicionales, izquierda, resultado de su aproximación. Imagen extraída de [3]

2.4 Naves

Para terminar con el estado del arte, se pasa a hablar sobre las aplicaciones existentes acerca de la generación de modelos de naves. Mayormente, estas aplicaciones utilizan algoritmos de síntesis para generar el modelo. Mientras que esto consigue un resultado bastante vistoso con poco esfuerzo, suelen tener el problema de ser muy repetitivos o necesitan tener una base muy grande para tener suficiente variedad. Las aplicaciones encontradas son tres:

- *ShapeWright*: Es una página web creada utilizando *WebGL* que permite la generación de una nave a partir de una cadena de texto. No se ha encontrado información sobre el algoritmo usado, pero una aproximación puede ser como sigue:

  1. Seleccionar una pieza central de forma aleatoria.
  2. Seleccionar una pieza de cuerpo o de complemento y ponerla en uno de los puntos de anclaje.
  3. Repetir el paso anterior un numero de veces determinado (puede que hasta terminar los puntos de anclaje).

  Como notas adicionales, hay que tener en cuenta que:

  - La pieza central es simétrica.
  - Las piezas se ponen por duplicado y de forma simétrica.

  Este algoritmo tiene una gran simplicidad pero, como mayor desventaja, necesita crear una gran cantidad de partes para poder generar modelos de formas variadas. Además, es necesario mantener una lista de puntos de anclaje lo que aumenta considerablemente el trabajo. De la misma forma, no dispone de ningún sistema para evitar intersecciones entre las distintas piezas.

- *Ship-not-even-wrong*: en este caso nos encontramos con un programa de código abierto (aunque parece que ha desaparecido el repositorio) escrito en c. Este está inspirado en el anterior y sigue un sistema similar. La gran diferencia consiste en que cada pieza es generada en tiempo de ejecución utilizando figuras geométricas simples. Esto le da un aspecto más rudimentario aunque, en realidad, la complejidad ha aumentado bastante.

- *Spaceship Generator*: en este caso nos encontramos con un *script* en python para blender. Su funcionamiento es totalmente distinto de los anteriores. En lugar de ensamblar la nave, la construye utilizando extrusiones. Los pasos que realiza son:

  1. Empieza con una caja.
  2. Construye el cuerpo haciendo extrusiones en la parte delantera y trasera mientras aplica transformaciones aleatorias.
  3. Añade asimetría al cuerpo aplicando extrusiones a algunas caras.
  4. Añade detalles como motores, antenas o armas dependiendo de la orientación de cada cara.
  5. Añade un modificador de bisel para la forma.
  6. Añadir materiales.

  Con este método consigues eliminar el problema de depender de una biblioteca de partes pero, al empezar siempre con un cubo, la forma general que nos resultará será siempre similar. Por otra parte, al utilizar detalles ya generados, no se consigue eliminar totalmente la biblioteca de partes.

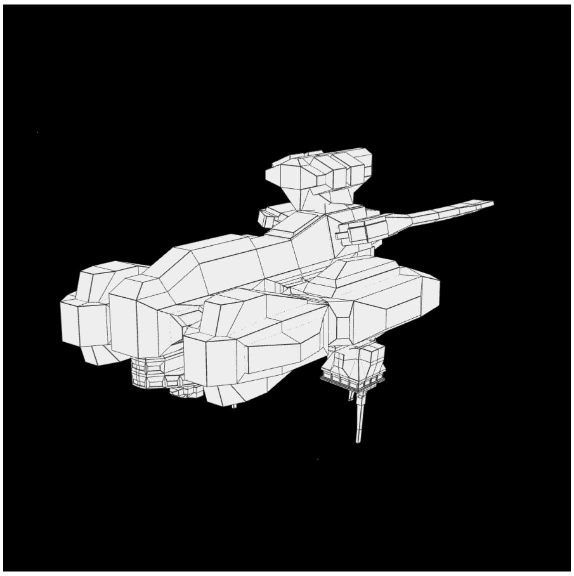

Figura 2.6: ShapeWright example.

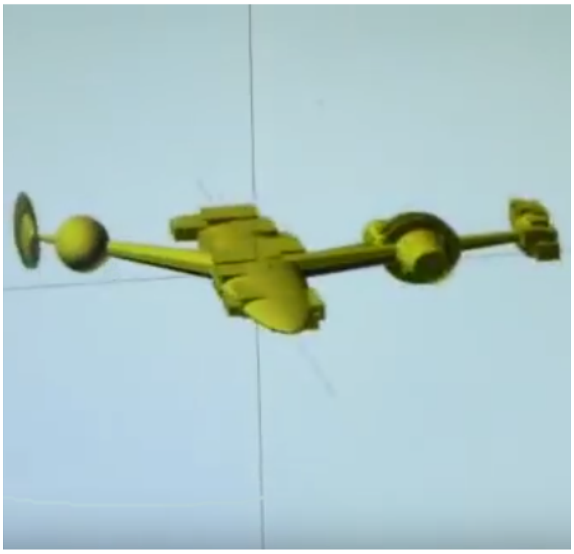

Figura 2.7: Ship-not-even-wrong example.

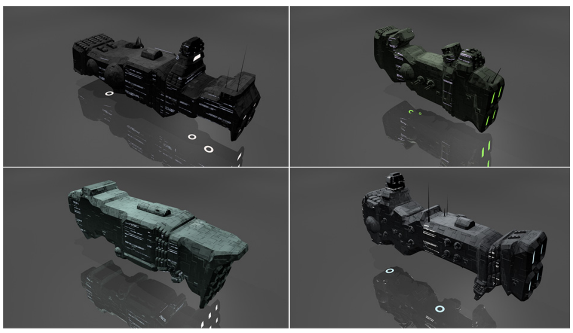

Figura 2.8: Spaceship Generator example.

CAPÍTULO 3
Tecnología utilizada

3.1 OpenGL

OpenGL define una API (Interfaz de programación de aplicaciones, en inglés *Application Programming Interface*) multilenguaje y multiplataforma para aplicaciones gráficas 3D. Esta API fue creada por Silicon Graphics Inc en 1992 para ocultar la complejidad de la interfaz con las distintas tarjetas gráficas y sus diferentes capacidades.

Con esto dicho, OpenGL no presenta una implementación de las distintas funciones que define pues son los fabricantes del hardware quien se encarga de ello. OpenGL solo define un grupo de funciones y cómo se han de comportar.

Esta tecnología se ha elegido, principalmente, por dos razones:

- Es multiplataforma.
- Se estudia durante el máster reduciendo el tiempo de aprendizaje.

3.2 Qt

Qt es un framework multiplataforma para extender el lenguaje c++ con *signals* y *slots*. Para conseguir esto, Qt proporciona un pre-procesador que analiza los archivos de código fuente y genera código c++ estándar antes del compilado. De esta forma, se hace posible compilar las aplicaciones desarrolladas con cualquier compilador.

De la misma forma, Qt proporciona un librería para trabajar con interfaces gráficas, así como un contenedor de bibliotecas como OpenGL. También dispone de clases y funciones como las que se pueden encontrar en la *C++ Standard Library*

Por otro lado, es una librería con un uso muy extendido, por lo que tiene un gran soporte y una gran comunidad expande su funcionalidad creando más contenido que se pueda incluir en un proyecto.

Además dispone de un IDE (entorno de desarrollo integrado) que proporciona una funcionalidad similar a los IDEs más usados (*e.g.*, Visual Studio) incluyendo funcionalidad para trabajar con las características añadidas de Qt. Además, es muy liviano dando una respuesta en tiempo real en funciones, por ejemplo, el auto-completado de código.

Sin embargo, el puente que nos ofrece con OpenGL no nos da toda la funcionalidad que necesitamos. Como ejemplo de esto, podemos encontrar que no ofrece funcionalidad para la lectura de los buffers que se utilizan para mostrar la pantalla. Debido a estos inconvenientes, se ha decidido a cambiar a PGUPV

3.3 PGUPV

PGUPV es una librería desarrollada dentro de la UPV, por el profesor Francisco José Abad, para la asignatura de programación gráfica.Esta librería nos proporciona un enmascaramiento de la funcionalidad de OpenGL para poder utilizarla trabajando con clases de C++. Además nos ofrece la posibilidad de incluir una interfaz gráfica simple pero suficiente para nuestra aplicación.

El mayor inconveniente que nos presenta esta librería es la falta de soporte para múltiples ventanas. Por esta razón, se ha utilizado un método típico en el desarrollo de videojuegos. Esto es, para evitar abrir una nueva ventana, se utilizan una serie de escenas y solo se muestra una de ellas. Cada una de estas, puede ser, por ejemplo, la pantalla de juego, el menú principal o la ventana de opciones. Por esta razón se ha tenido que implementar un gestor de escenas que se encargará de mostrar la escena activa y cambiarla cuando sea necesario.

Además de esto, se ha tenido que implementar la funcionalidad para una serie de widgets que no ofrece esta librería como puede ser el cuadro de texto o los llamados controles de tipo radio que sirven para seleccionar una única opción de una lista.

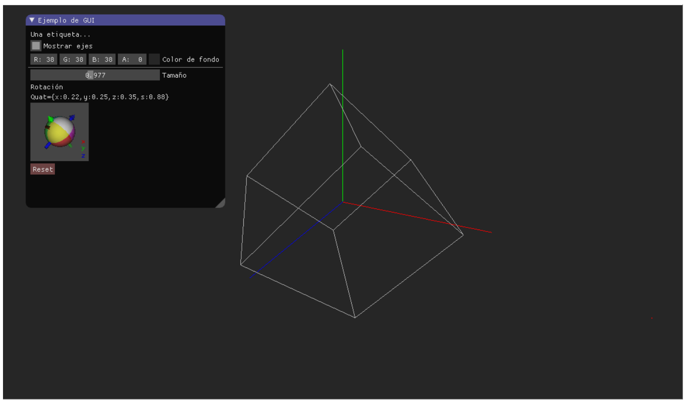

Figura 3.1: Ejemplo de una ventana PGUPV con una GUI simple

CAPÍTULO 4
Metodología

4.1 Estructura de la nave

Se ha decidido generar naves compuestas por un cuerpo simple y una serie de adornos cómo pueden ser la cabina, alas, etc. El cuerpo está compuesto por:

- Esqueleto: línea que define la parte central del cuerpo.
- Secciones: una serie de formas cerradas que formarán el exoesqueleto del cuerpo. Estas secciones son las que se utilizarán para la generación de la malla.

En el caso de los adornos, se sitúan en un punto del cuerpo y siguen una estructura similar a este.

4.2 Prototipo semiautomático

El algoritmo propuesto para la generación del modelo sigue las siguientes etapas:

1. Generar una "siluetaütilizando una gramática.
2. Generar anillos por cada punto de la silueta.
3. Unir los puntos de cada anillo para generar la malla.

4.2.1. Gramáticas

En este caso, una gramática consiste en una estructura para generar o aceptar una serie de cadenas. Los mecanismos que disponen para este fin son:

- Vocabulario: definición de los distintos símbolos que componen la gramática.
- Axioma: Cadena inicial de la gramática.
- Producciones: Serie de reglas para transformar una serie de símbolos en otros.

En nuestro caso, hemos utilizado un tipo especial de gramáticas llamado *L-systems*. Estas incluyen un sistema para convertir las distintas cadenas generadas en una representación gráfica. Estas gramáticas se crearon para dar una descripción formal del desarrollo de organismos pluricelulares pero ha tenido un gran éxito en la generación de flora.

Figura 4.1: Imágen de un L-system extraída de HTML5 L-systems

4.2.2. Generación de la silueta

La primera parte del algoritmo consiste en generar una silueta. Esto lo usaremos para ajustar el radio de cada anillo y determinar su posición. Para este fin, nos hemos definido una gramática.

El vocabulario lo podemos separar en dos grande grupos donde tenemos los símbolos que no afectan al resultado (al convertir el estado de la gramática no generan ningún punto):

- START
- FRONT
- BACK
- CENTER
- RAMP
- FLAT
- WALL

Como ya hemos dicho, estos símbolos son no terminales que sirven para generar nuevas producciones. Sin embargo, tenemos un símbolo terminal que consiste en el punto. Este es el único símbolo que nos proporciona la información para generar la silueta. Estos puntos llevan una serie de características asociadas para definir una posición. Los parámetros más importantes son el rango en el que pueden generar los valores y si el rango es absoluto o no, esto es, si los valores generados tendrán una posición determinada o dependerán de la elevación actual.

Con el vocabulario definido, podemos hablar del axioma:

```
P(0,0), START, P(1,0)
```

Las producciones se han definido por código y, cómo ejemplo, podemos observar una sección de estas:

START →FRONT Pv([0.3,0.4],[0.2,0.3]) CENTER Pv([0.6,0.7],[0.2,0.3]) BACK (33 %)
| FRONT Pv([0.4,0.6],[0.2,0.3]) CENTER Pv(1,[0.2-0.3]) (33 %)
| FRONT P([0.25,0.75],[0.2,0.3]) BACK (34 %)

(Vease Apéndice A para la gramática completa)

En estas producciones, podemos encontrarnos una serie de puntos que mantienen una componente (i.e., todos los puntos generarán el mismo valor para la coordenada guardada). Esto estará marcado por 'v' o 'h' después de la 'P' dependiendo si es la coordenada y o x. En caso de haber más de una serie, se añadirá un numero. Además, para indicar que un punto utiliza rango absoluto, se le ha añadido una 'a'.

Una vez definida la gramática, necesitamos una forma de convertir la cadena generada en algo que podamos utilizar. Esto nos resulta sencillo puesto que sólo nos hace falta saber la posición de los puntos. Sabiendo esto, descartamos cualquier cosa que no sean puntos y se calcula su posición.

4.2.3. Generación de anillos

Para la generación de los anillos, se ha elegido utilizar polígonos regulares puesto que esto facilita mucho su generación y crea formas simétricas. Esta propiedad es interesante ya que se obtienen con más facilidad unos resultados atractivos.

El siguiente paso es determinar el numero de puntos en cada anillo. Para esta razón, hemos definido un par de parámetros que pedimos al usuario: el rango de puntos y la variabilidad entre los anillos. Mientras que el rango viene dado por los valores extremos, la variabilidad es un valor entre 0 y 1 que determinará el porcentaje del rango que se aplicará cuando se genere el siguiente anillo.

De esta forma, empezamos con una cantidad aleatoria y, cuando generamos un nuevo anillo, elegimos un valor aleatorio que tendrá una rugosidad dada por el componente de variabilidad. Cuanto más cerca de 1, más abruptos serán los cambios mientras que, cuanto más cerca de 0, las diferencias entre un anillo y el siguiente serán menores.

4.2.4. Generación de malla

Llegados a este punto, ya solo nos queda unir los puntos del modelo que hemos generado en el paso anterior pero esto es un problema complejo. Por suerte, a causa de la estructura del problema, podemos simplificarlo si lo dividimos en varios subproblemas. En vez de intentar crear un modelo a partir de una nube de puntos, podemos intentar crear mallas para las cintas formadas por cada par de anillos.

En estos momentos, tenemos que unir los puntos del anillo A al anillo B formando triángulos. Con tal fin, podemos observar las siguientes propiedades:

1. Todos los puntos deben formar parte de, al menos, un triangulo.
2. No se puede crear un triangulo que solape con otro.
3. En caso de que las cintas tengan el mismo número de puntos, cada punto formará parte de, exactamente, dos triángulos.
4. Las aristas resultantes uniendo los puntos de distintos anillos, siempre tendrán la longitud mínima posible.

A partir de estos pasos, se puede hacer una algoritmo voraz. Sin embargo, aun tenemos que buscar los puntos de corte para desplegar las cintas. Para ello, buscamos dos puntos, uno de cada cinta, que tengan la distancia mínima. De esta forma, tenemos nuestra primera y última arista y, además, el punto por el que desplegar la cinta.

Después de obtener esta cinta, se va a utilizar una ventana deslizante para recorrer todos los puntos pertenecientes a los anillos. Esta ventana estará formada por dos puntos de cada anillo y nos servirá para determinar cual es la arista del siguiente triángulo. Para ello, es necesario observar qué diagonal es la más corta. Este proceso se puede observar en la figura 4.2.

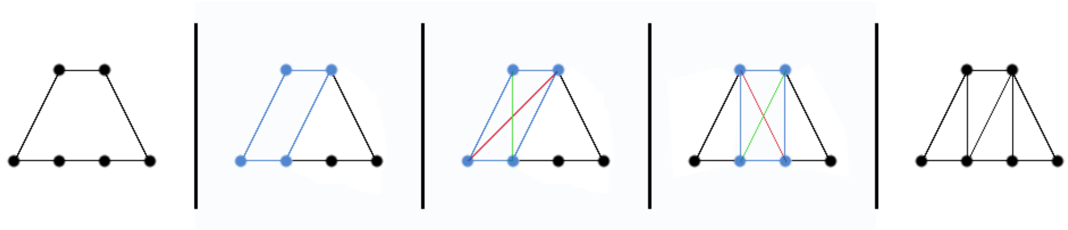

Figura 4.2: Ejemplo de triangulación

Una vez generado el modelo, se han aplicado una serie de transformaciones para alterar el modelo final achatándolo en los distintos ejes. Con estas alteraciones, se intenta eliminar un exceso de ejes de simetría.

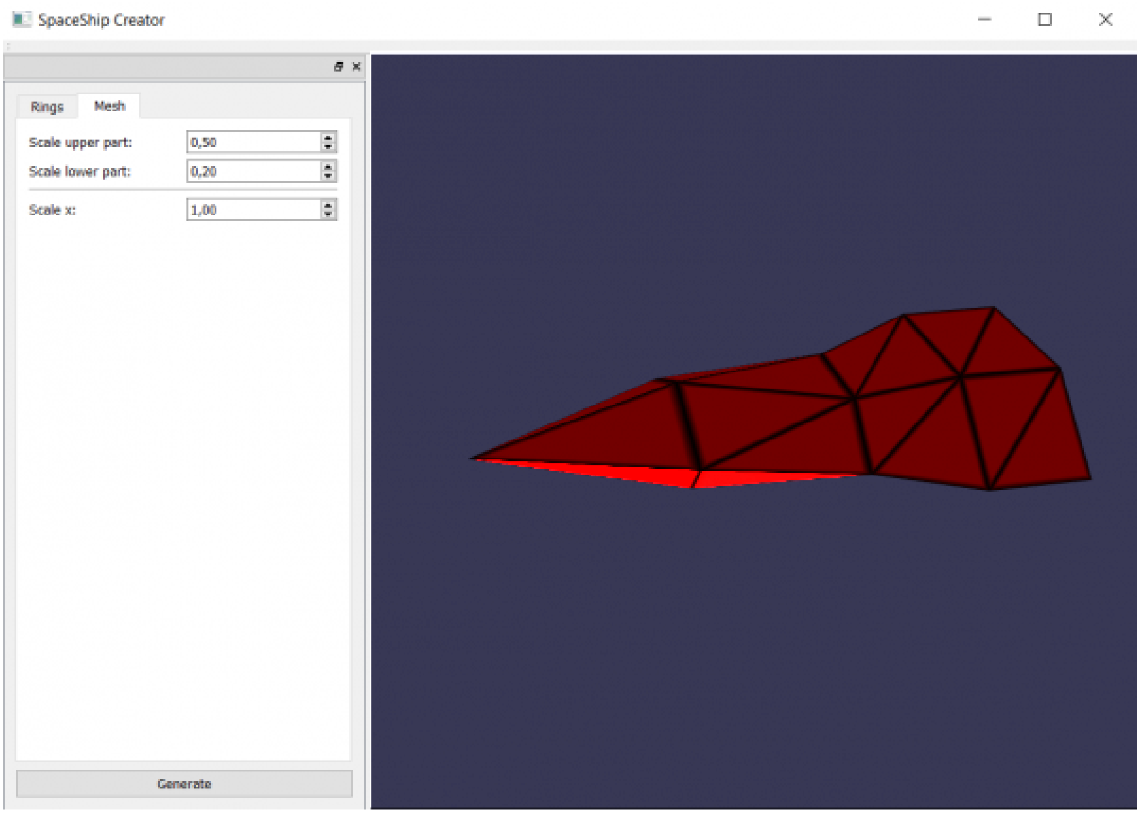

Figura 4.3: Resultado de la primera aproximación

4.2.5. Resultados

Esta aproximación presenta una serie de limitaciones. La primera de ellas, consiste en que la variedad de modelos generados, depende de la complejidad de la gramática.

Esto no sería tan problemático, si no fuese por la dificultad que presenta, al usuario, el modificar esta gramática.

El otro problema consiste en que, para conseguir mayor variabilidad, hay que crear más producciones dentro de la gramática. Esto hace que mantenerla sea un proceso tedioso y limita mucho los posibles resultados.

Por otro lado, el algoritmo presentado da la posibilidad de generar una gran variedad de modelos (con una gramática compleja) sin necesidad de que el usuario introduzca muchos datos pero también ofrece la posibilidad de configurar las formas de las naves modificando la gramática.

4.3 Prototipo paramétrico

Dada la limitación de trabajar con polígonos regulares, se ha decidido buscar algún otro sistema. Por esta razón se ha buscado una figura geométrica que se pueda crear a partir de una serie de parámetros.

4.3.1. Generación de anillos

La figura que se ha decidido utilizar consiste en un cuadrilátero con las esquinas redondeadas. Esta elección se ha elegido dado que se puede controlar con unos pocos parámetros, como serían la altura y la anchura, mientras que las modificaciones resultantes son las esperadas por parte del usuario. Esto no se da con otras figuras geométricas como podría ser el hexágono donde, con la misma cantidad de parámetros, no resulta posible conseguir un buen control y, en el caso de incrementarlos, estos resultarían engorrosos de controlar.

Por otro lado, al aplicar un redondeado en las esquinas, conseguimos permitir la generación de naves orgánicas utilizando un único parámetro. Éste nos determina el porcentaje de las aristas que constituirá parte de la curva.

4.3.2. Generación de malla

Para la generación de la malla se ha decidido utilizar el mismo número de puntos en cada anillo. Con esto conseguimos una correlación directa entre los distintos anillos facilitando enormemente el proceso de creación de la malla. Sin embargo, esto presenta el riesgo de terminar teniendo una cantidad de puntos inferior o superior a la necesitada en anillos de diferente tamaño.

4.3.3. Resultados

Como resultados, hemos obtenido un algoritmo que le ofrece mucho control al usuario a la hora de generar el modelo a partir de una serie de parámetros. Estos, a su vez, permiten ser generados de forma automática lo que nos proporciona una base para un algoritmo mucho más complejo.

4.4 Programa de diseño asistido por ordenador (CAD)

En este punto, se ha decidido crear una plataforma para la implementación y experimentación de los algoritmos implementados y de cualquiera que queramos implementar

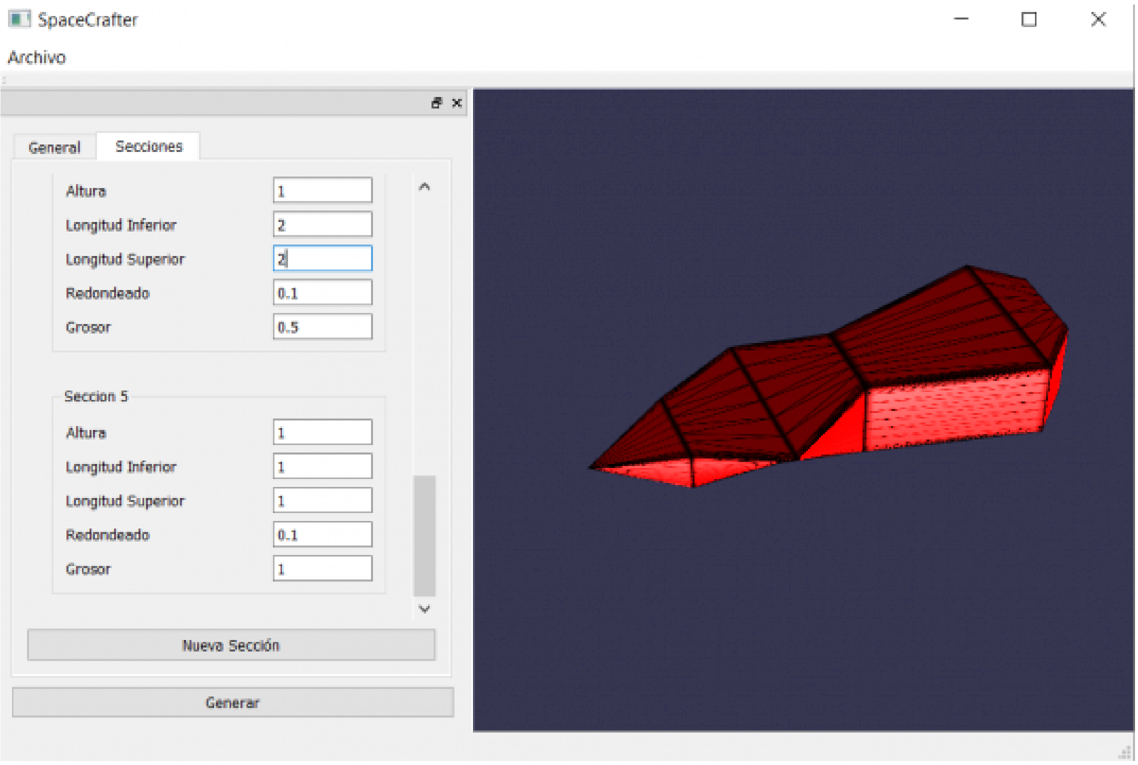

Figura 4.4: Resultado de la segunda aproximación

en un futuro. Para ello, se ha creado un programa de diseño asistido por ordenador que nos permite editar las distintas partes de la nave a partir de una serie de ventanas o escenas. Para ello se han utilizado tres herramientas:

- Curvas de Bézier: nos ofrecen la posibilidad de generar el esqueleto y las secciones mezclando tanto curvas como segmentos rectos
- Superficies de Bézier: nos proporciona la capacidad de crear unas mallas con uniones perfectas entre las distintas superficies.
- Gestor de escenas: Conjunto de clases que se encarga de hacer la transición entre las distintas escenas.

4.4.1. Curvas de Bézier

El primer punto importante de esta aplicación son las curvas de bezier. Estas son el componente básico de la aplicación pues cada segmento que se introduce es una de estas curvas. Esta es una poderosa herramienta pues nos permite dar una gran libertad sobre como construir el modelo ya que permite generar tanto modelos orgánicos como inorgánicos.

Estas curvas vienen definidas por cuatro puntos de control, en el caso de las cúbicas, o por $n+1$ donde n es el grado de la curva para un caso general. Estos puntos no se encontrarán en la curva excepto para el primero y el último. Gracias a esto, podemos conectar varias curvas de forma consecutiva. Para ello, solo necesitamos utilizar el mismo punto como el último y el primero de las dos curvas.

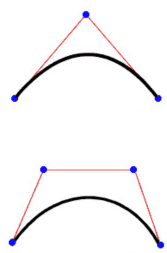

Figura 4.5: Bézier cuadrática y cubica

Para definir cada uno de los puntos que forman estas curvas, se utiliza una serie de interpolaciones entre los cuatro puntos de control y, de forma recursiva, las lineas que se consiguen al unir los puntos del paso anterior hasta terminar con un único punto. Este proceso se puede ver en la figura 4.6

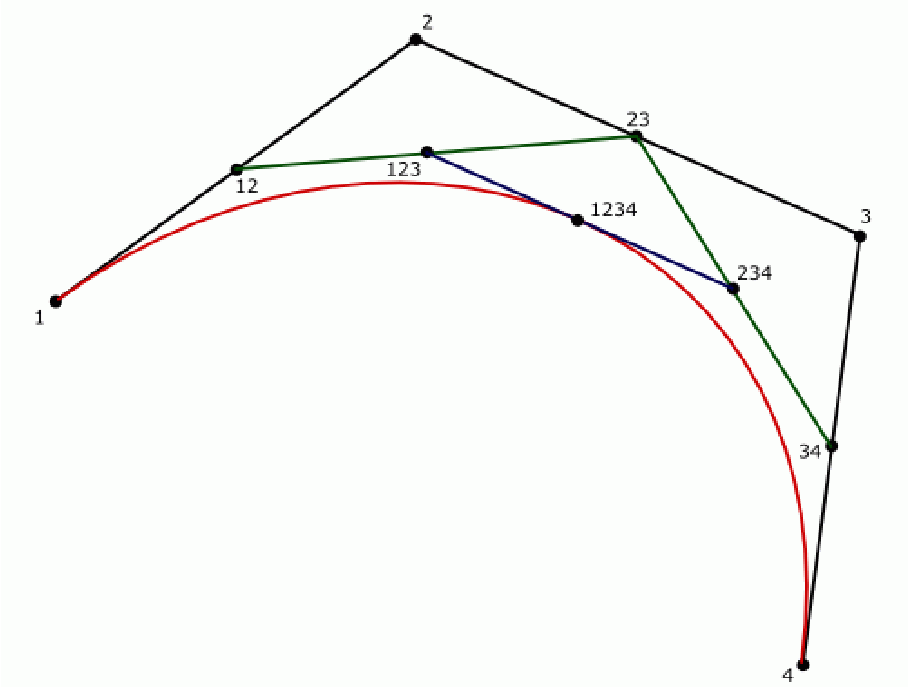

Figura 4.6: Proceso de interpolación para t=0.5

Sin embargo, esta forma de calcularlas no es práctica. Para eso, tenemos la fórmula 4.1. De esta forma, podemos calcular un punto cualquiera dentro de una curva. Además, se puede generalizar para llegar a la fórmula4.2 de las curvas cúbicas que son con las que se trabajará.

$$B(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-1} P_i, t \in [0,1] \tag{4.1}$$

$$B(t) = P_0(1-t)^3 + 3P_1 t(1-t)^2 + 3P_2 t^2(1-t) + P_3 * t^3, t \in [0,1] \tag{4.2}$$

Para trabajar con estas curvas, se ha implementado una librería que nos ofrece toda la funcionalidad necesaria para trabajar con ellas:

- Generar la curva: esto nos genera una serie de puntos que formarán los segmentos de esta curva. Para determinar la cantidad que vamos a generar, podemos definir un número exacto o que se genere a partir de la longitud y un nivel de detalle.
- Generar los puntos de Castlejau: estos puntos son los que nos genera el sistema para calcular un punto de la curva descrito en la figura 4.6. Por regla general, nos interesa trabajar con las ecuaciones excepto en el caso de dividir una curva.
- Caja de inclusión: función para obtener los valores máximos y mínimos de las curvas con las que trabajamos. Esto nos resulta muy práctico para centrar la curva.
- Tangente, normal y binormal: funciones para calcular estas componente para un punto dado. Esto es lo necesario para poder orientar una curva en un segmento.

- Punto en curva: comprobar si un punto dado está en la curva. Esto se debe hacer por aproximación dado que no se puede calcular de forma exacta.
- Proyectar un punto en la curva: busca el punto de la curva más cercano al punto dado.
- Intersección en el eje x: para aplicar un modificador de simetría, es necesario saber en que punto cruza por el eje de simetría. Para simplificar, se ha utilizado el eje x como tal. Esto nos simplifica el problema a tener que resolver una ecuación cúbica. De esta forma, se ha utilizado el algoritmo de Cardano.

4.4.2. Superficies de Bézier

Para generar una malla, tenemos las superficies de bezier. Su funcionamiento es muy similar a sus equivalentes de dos dimensiones. También están formadas por una serie de puntos de control por los que la superficie, generalmente, no pasa.

Estas superficies presentan una propiedad interesante. Si tomamos uno de los cuatro bordes, tenemos una curva de bezier. Esto nos permite dos cosas. La primera, podemos crear una superficie a partir de una serie de curvas. La segunda consiste en, dadas dos superficies que compartan la misma curva como borde, conseguiremos una superficie que no presente agujeros en la unión.

En este caso, hemos utilizado superficies cúbicas de dos formas: parches bicúbicos y triángulos de bezier.

Parches bicúbicos

Este tipo de superficies se han elegido dado a su simplicidad de implementación mientras que dan un gran control sobre la superficie que genera. Además, se puede crear fácilmente a partir las curvas de bezier que tenemos generando los puntos que nos faltan.

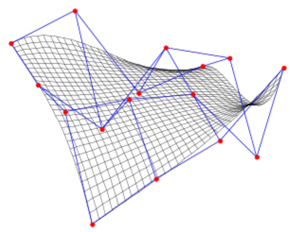

Figura 4.7: parche bicúbico con sus puntos de control

Por otro lado, el cálculo de las normales es directo además de que se pueden obtener de forma exacta. Esta propiedad es muy importante dado que nos permite la correcta iluminación del modelo.

El único inconveniente que nos presenta es su forma. En nuestro caso necesitamos generar tanto triángulos como cuadriláteros. Se puede aplastar uno de los lados en uno de estos parches para conseguir un triangulo pero esto nos deja con dos problemas. Por una parte, nos causa problemas de iluminación mientras que, por la otra, nos deja una parte del triangulo con una mayor densidad de polígonos que la otra.

Triángulos de bezier

Para poder representar estas formas, se ha recurrido al uso de triángulos de bezier. A diferencia de un triangulo normal, esta superficie es bastante más compleja que su versión de cuatro puntos.

Igual que en los casos anteriores, la generación de cada punto de la curva se hace mediante una interpolación recursiva. Sin embargo, dado la forma de la superficie, se deben utilizar coordinadas baricéntricas para esta tarea.

Además, no hay garantía de que, en dos triángulos contiguos, generen una unión suave aunque la unión si lo sea. Este comportamiento es diferente al que se da en el caso de los parches bicúbicos.

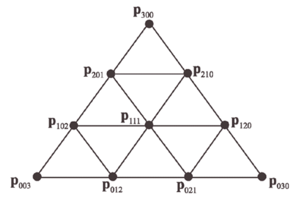

Figura 4.8: puntos de control de un triangulo de grado 3

4.4.3. Generación de la malla

La generación de la malla es uno de los problemas principales de este trabajo. El primer intento ha consistido en aplicar la misma técnica que se ha utilizado al principio. Sin embargo, esto a resultado inviable por dos razones. Al utilizar curvas, se han tenido que generar más puntos para la representación de cada una de las curvas. Sin embargo, si aplicamos el algoritmo que teníamos diseñado, no distingue estos nuevos puntos de los originales. Este es nuestro primer problema.

La solución directa, sería ejecutar el algoritmo del primer prototipo para crear los triángulos entre las distintas curvas. Al hacer esto, nos genera unas superficies curvas mal definidas. Para evitar esto, tenemos que distinguir entre las dos primitivas que vamos a utilizar.

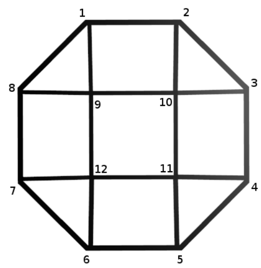

Figura 4.9: Ejemplo de las uniones entre un cuadrado y un octágono

Esto nos ofrece un nuevo problema que podemos ver en la imagen 4.9. Si dividimos el cuadrilátero formado por los puntos 1, 2, 9 y 10 en dos triángulos, terminaríamos con la malla deformada. Por otro lado, no podemos crear un cuadrilátero con los puntos 2, 3, 10 y 11. No podemos considerar el cuadrilátero 2, 3, 4, y 10 puesto que estaríamos consumiendo dos aristas del polígono exterior y ninguna del interior.

Para poder considerar un cuadrilátero como tal, necesitamos que cumpla dos condiciones. La primera, que se puede observar en la figura, implica que no hayan 3 puntos colineales. Esto es bastante obvio puesto que si los tenemos, solo conseguimos un triangulo. La segunda condición consiste en que los cuatro puntos sean coplanares. Esto no se puede ver en la imagen puesto que, al estar en 2D, siempre van a serlo. En un caso en tres dimensiones, como el que se da en la aplicación, será lo que nos diga, mayormente, si tenemos que considerar la forma que estamos observando como un triángulo o un cuadrilátero.

Una vez dividida la cinta en triángulos o cuadriláteros, necesitamos subdividir estas formas. Para ello, se han contemplado dos métodos. En el primero, podemos considerarlo como una cinta. Sin embargo, esto nos genera unos triángulos muy alargados y estrechos que nos dificultarán la tarea del texturado. Aquí tenemos el segundo problema que comentábamos en el primer párrafo.

La segunda opción es trabajar con las superficies de Bézier que se han explicado arriba. Para ello, primero necesitamos obtener los puntos de control que nos faltan:

En los parches bicúbicos esto es una tarea fácil. Dados los puntos de control de las dos curvas, solo tenemos utilizar interpolación lineal en cada par de puntos de control para obtener dos curvas más. De esta forma, conseguimos los 16 puntos para una superficie que, entre anillos, es lisa. Para darle curvatura, tendríamos que desplazar estos puntos hacia fuera o hacia dentro. Esto no se ha implementado debido a la falta de espacio en la interfaz.

Para los triángulos, esto es más complicado. En este caso, necesitamos generar las dos curvas que componen los bordes. Esto lo podemos hacer como en el caso del parche puesto que el caso es similar. Sin embargo, para el punto central necesitamos calcular el correspondiente al baricentro.

Una forma para hacer la teselación es utilizando una malla regular pero esta idea se ha descartado porque nos obliga a generar las curvas utilizando el mismo número de segmentos para todas. Esta es la opción que está implementada en todas las librerías para trabajar con superficies de Bézier.

Otra opción consiste en utilizar la teselación de OpenGL. Esta opción nos presenta una forma rápida y viable pero también inconvenientes. Si utilizamos está opción estamos pidiendo una versión de OpenGL que aun no está soportada por gran parte de las tarjetas gráficas integradas. Además, nuestro resultado dependerá del fabricante puesto que son ellos los que controlan el algoritmo de teselación.

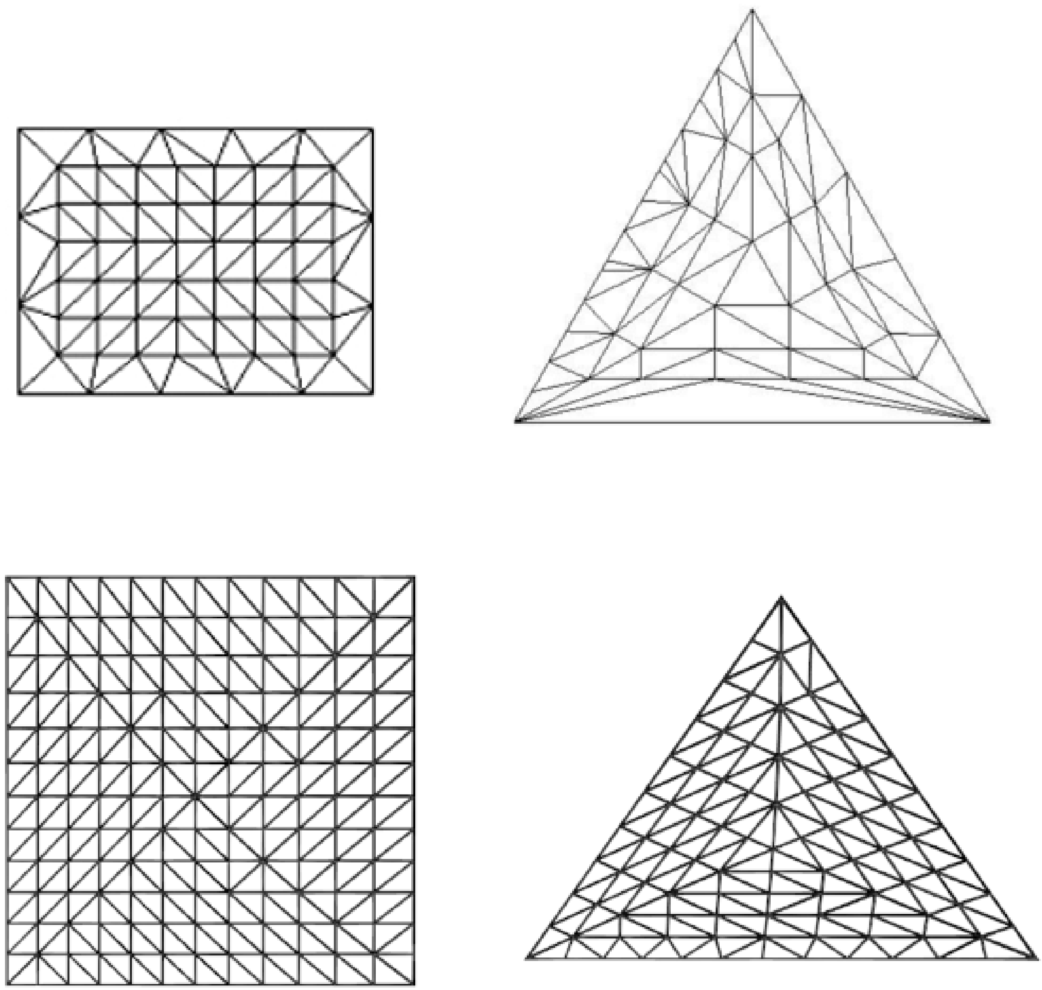

Figura 4.10: Ejemplos de teselación de opengl (arriba) y la generada por nuestro código (abajo)

Por estas razones, se ha decidido implementar nuestro propio algoritmo siguiendo el estándar de OpenGL[10]. La idea detrás de este algoritmo consiste en generar una serie de anillos concéntricos con una cantidad distinta de puntos y tamaños diferentes. De ésta forma, vamos a generar un primer anillo que tendrá una cantidad de puntos en cada lado, no necesariamente la misma, que nos servirá de puente entre el exterior y los anillos interiores. Estos, irán decreciendo en tamaño y numero de puntos en una cantidad equitativa.

4.4.4. Gestor de escenas

Para controlar el estado de la aplicación, se ha creado un conjunto de clases para gestionar las distintas transiciones. Estas son:

- Gestor de escena: en esta clase se incluyen las funciones para cambiar de escena. Además, se incluye un mecanismo para poder hacer una serie de pasos y volver a una escena en concreto al estilo de un dialogo emergente.
- Escena: clase abstracta que nos proporciona el esqueleto de las distintas funciones que se ejecutaran al cambiar entre escenas.

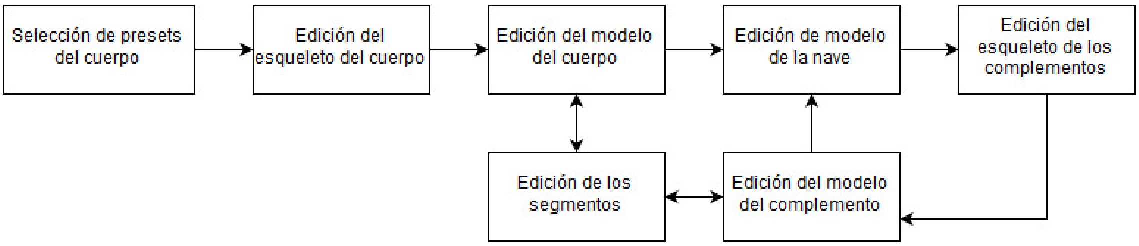

Figura 4.11: Diagrama de flujo de la aplicación

4.4.5. Interfaz

Selección de presets

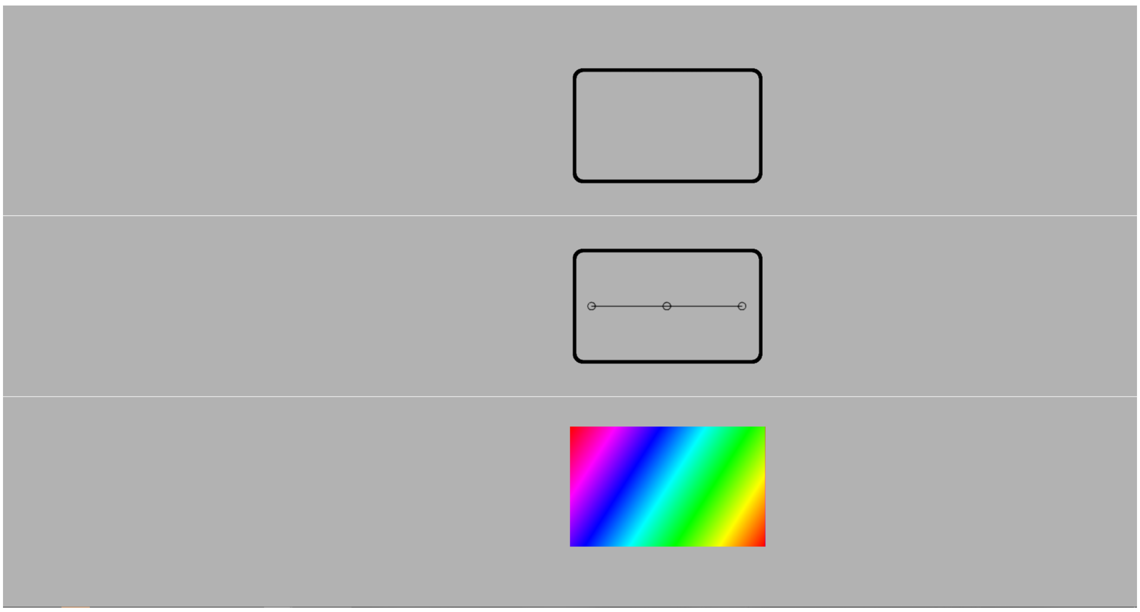

Figura 4.12: ventana de selección de presets del cuerpo

Esta es la ventana de introducción a la aplicación. Esta ventana nos da a elegir entre una serie de *presets*. Esto son modelos predefinidos, enteros o solo del esqueleto. Se dividen en tres secciones, la primera es para los especiales como es empezar en blanco o, si se hubiese añadido, empezar con un modelo aleatorio. La segunda sección, es para los modelos que solo tienen el esqueleto y, la tercera, para los que tienen todo definido. Como no se dispone de un modelo ya hecho, se ha utilizado uno de pruebas para mostrar la funcionalidad.

Estos modelos, están guardados en un fichero obj que contiene, los puntos que forman el esqueleto y, si lo tiene, las distintas secciones. Como no se ha tenido tiempo de generar un sistema para guardar estos modelos, no se ha podido crear una biblioteca con una cantidad decente de modelos.

Las tres bibliotecas que se han creado, cargan los modelos de sus carpetas correspondientes. Primero buscan el archivo del modelo y, luego, cargan la imagen con el mismo nombre. Estas imágenes actúan como botones para pasar a la siguiente ventana.

Cada uno de los botones está compuesto por un rectángulo con una textura. Para detectar si se ha seleccionado alguno, se busca un botón que contenga el punto que hay debajo del ratón.

Edición de esqueleto

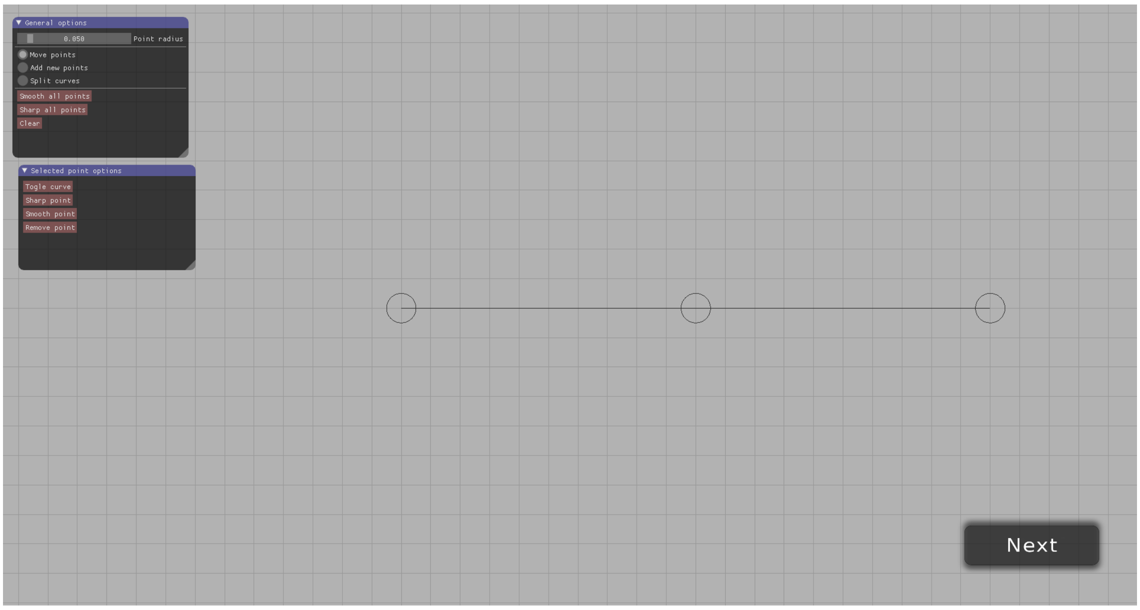

Figura 4.13: ventana de edición del esqueleto perteneciente al cuerpo de la nave

En esta ventana empezamos a tener contenido. Como podemos observar, disponemos de un menú en la parte superior izquierda para controlar algunas opciones y un botón en la inferior derecha para pasar a la siguiente escena.En la parte central, tenemos el área de dibujo con una rejilla y nuestros dos primeros segmentos.

Como ya se ha comentado con anterioridad, estos segmentos son, en realidad, curvas de Bézier y, para controlarlas, tenemos sus puntos de control. Hay dos pasos que necesitamos para poder verlos: hacer clic en ellos y habilitar la curva. Para seleccionar el punto que hay debajo del ratón, al estar en 2D, se puede calcular el punto del ratón en el espacio del mundo. Esto consiste en aplicar las operaciones de transformación que afectan a la escena (cámara y proyección) en orden inverso. Sin embargo, como estamos tratando con la posición del ratón (2D), necesitamos una coordenada más para aplicar las posiciones. Todo esto es en un caso general aunque, al ser el problema en 2 dimensiones, no nos interesa la tercera componente, con lo que la podemos descartar. Una vez tenemos las coordenadas, tan solo tenemos que mirar si está a menos de cierta distancia de cada punto.

También podemos observar una serie de botones para aplicar operaciones a nuestras curvas. Las que tienen algo de interés son las de convertir las uniones en ángulos rectos o suavizadas. Para el primero, solo tenemos que mover los puntos de control de ese vértice a la linea formada por él y el siguiente o anterior. Para crear una unión suave, necesitamos que los puntos de control de alrededor del vértice formen una línea perpendicular a la bisectriz.

El otro punto de interés, es la herramienta para partir la curva. Para ello, se ha de obtener el punto de la curva que tenemos debajo del ratón y conseguir los nuevos puntos de control. Para ello, podemos utilizar las dos funciones implementadas en la librería de curvas de Bézier.

Edición del cuerpo

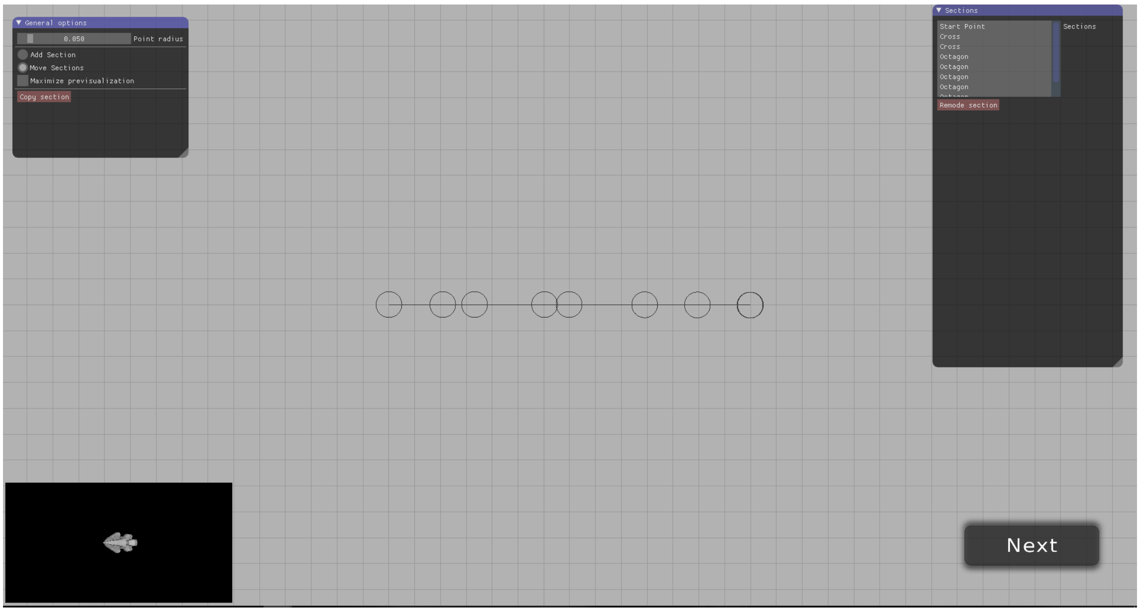

Figura 4.14: Ventana de edición del modelo perteneciente al cuerpo de la nave

En esta ventana, se nos vuelve a presentar algo similar que en la anterior. Sin embargo, ha habido una transformación antes de llegar a esta ventana. Se ha centrado la curva en el centro de coordenadas. Si no lo hacemos, la nave tampoco estaría centrada con lo que, a la hora de importar el modelo en alguna otra aplicación, no se podría incluir de forma correcta.

Para realizar esta tarea, necesitamos los máximos y mínimos de la curva. Este problema, a su vez, se resuelve buscando los puntos de inflexión, esto es, resolviendo las ecuaciones de $B'(t) = 0$ y $B''(t) = 0$. De esta forma, obtenemos las ecuaciones 4.3 y 4.4 respectivamente.

$$B(t)' = 2(1-t)(P_1 - P_0) + 2t(P_2 - P_1) \tag{4.3}$$

$$B(t)'' = 2(P_2 - 2P_1 + P_0) \tag{4.4}$$

Después de centrar el modelo, podemos observar que se ha añadido un panel a la derecha para mostrar los distintos segmentos y un recuadro en la esquina inferior izquierda, donde se puede ver una previsualización del cuerpo. También disponemos de un botón para que esta ocupe toda la ventana.

Los controles en esta ventana te permiten añadir una nueva sección o mover una existente. En el primer caso, nos mostrará la siguiente escena y, al volver, se necesita generar una matriz de transformación para posicionar los puntos de la nueva sección en el sitio final que ocuparán dentro del modelo. Para ello es necesario calcular la normal, tangente y binormal en el punto de la curva donde se quiera añadir esta sección.

Para calcular la tangente, solo necesitamos la ecuación 4.3 pero, en la normal, resulta más complicado. La forma para obtenerla pasa por generar dos tangentes muy cercanas, calcular el vector perpendicular entre ellas y, utilizando este nuevo vector y la tangente que teníamos, conseguimos la normal. Esto funciona puesto que, en una curva de Bézier, las tangentes cambian incluso en distancias pequeñas. Sin embargo, hay un caso que es necesario tratar. En nuestra aplicación pueden aparecer casos de curvas degeneradas, es decir, líneas rectas. Estas, al ser una línea, tienen una cantidad infinita de normales, por lo que es necesario seleccionar una de ellas.

Edición de segmentos

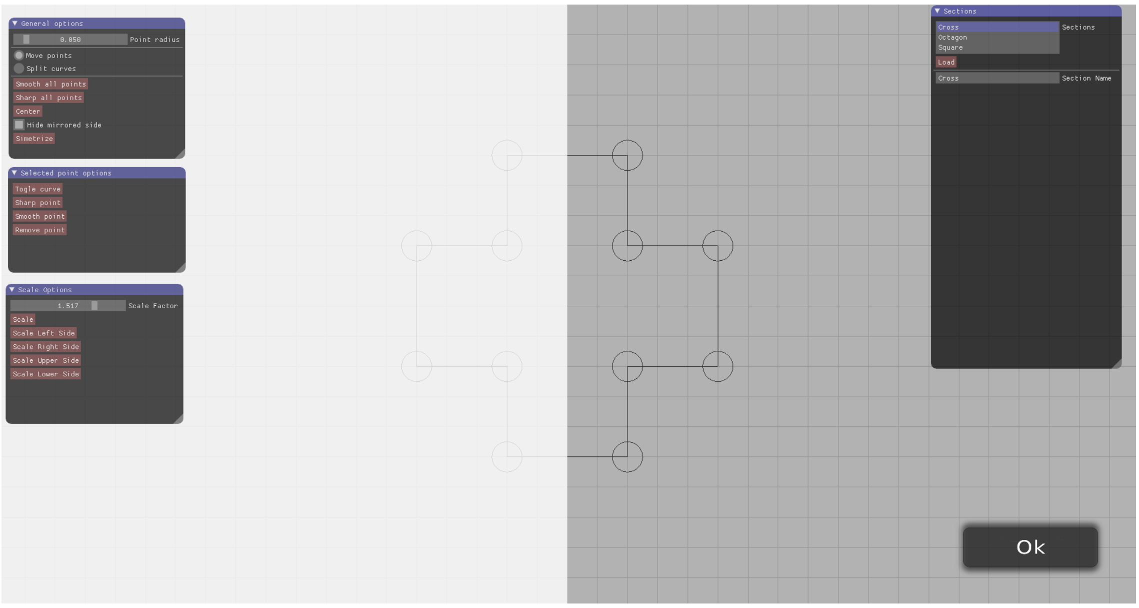

Figura 4.15: Ventana de edición secciones

En esta ventana se nos muestra otro editor de curvas de bezier al que le hemos añadido más funcionalidad. Una de estas, es la zona blanqueada que podemos ver a la derecha. Esta máscara sirve para marcar el área que se eliminará si se utiliza el operador de simetría. También se ha incorporado la opción de ocultar la máscara en caso de no querer trabajar con segmentos simétricos.

Otro cambio, es el panel que podemos ver a la derecha que nos permite cargar algunas formas predeterminadas y asignarle un nombre personalizado a nuestra forma para una mejor identificación en la ventana anterior. Esta biblioteca funciona de la misma forma que la presentada en la primera ventana, aunque con una presentación distinta.

De esta misma forma, se han añadido algunas operaciones más para el escalado y una para generar una forma simétrica utilizando un modificador de espejo en la mitad del anillo. Para ello, primero necesitamos partir el segmento en dos mitades. Para facilitar esta tarea, se ha decidido utilizar el eje 'y' como eje de simetría. De esta forma, solo se necesita resolver la ecuación $B(t) = 0$ para obtener los puntos de corte. Esto es una ecuación de grado 4 que, para resolverla, hemos aplicado el algoritmo de Cardano.

Después de obtener los puntos de corte para todas las curvas, necesitamos comprobar que no haya más de 2 puesto que, de darse el caso, tendríamos formas aisladas con las que no sabemos tratar. A continuación, se elimina los puntos que están en la zona reflejada y se substituyen por los puntos de la otra zona con la componente x de la coordenada invertida y en orden inverso.

Edición de modelo

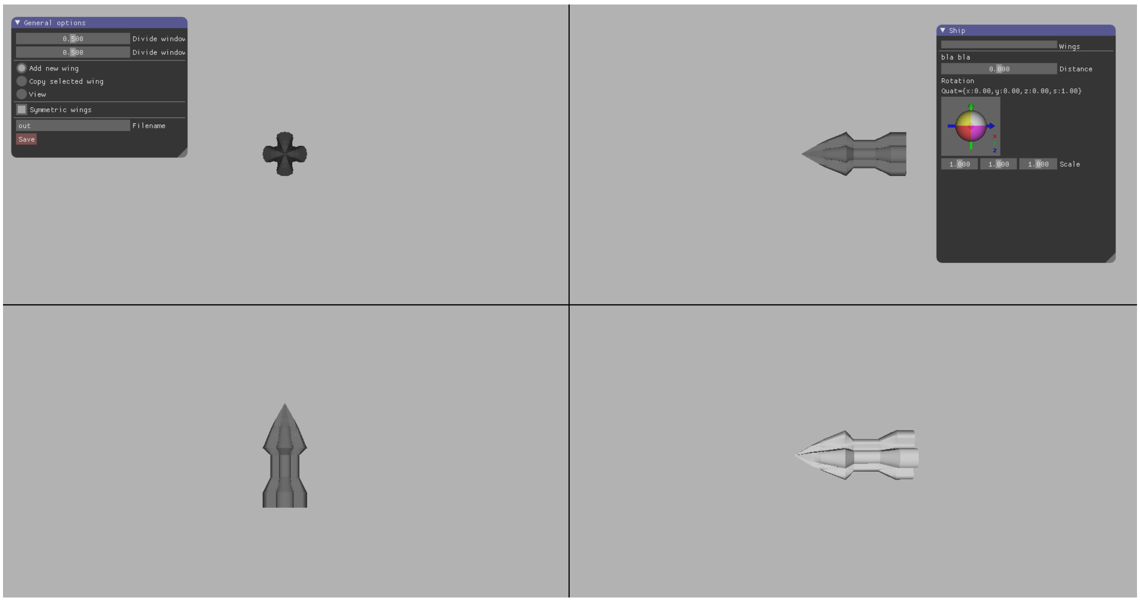

Figura 4.16: Ventana de edición del modelo

Llegados a este punto, tenemos la oportunidad de guardar nuestro modelo aunque podemos empezar a añadir detalles como alas u otros complementos. Esta ventana tiene dos puntos interesantes. El primero es la función de guardar el modelo que ya se ha nombrado. Para ello, se ha decidido utilizar el formato *wabefront* (obj).

El otro punto interesante consiste en añadir los distintos complementos. Para ello, vamos a buscar el punto del mundo donde se va a situar. Por otro lado, para conseguir la dirección que va a tener el complemento, hemos dibujado la escena utilizando una conversión de las normales del modelo a colores. De esta forma, vamos a utilizar la normal que vemos y no una que podamos aproximar nosotros que puede ser distinta.

Edición de complementos

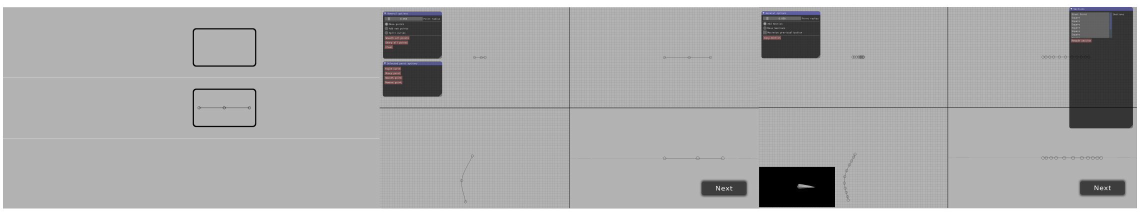

Figura 4.17: Ventanas de edición de los complementos

Para crear los complementos, se ha seguido el mismo proceso que en la creación del cuerpo de la nave pero permitiendo modificar sin restringir una coordenada. No ha habido que hacer muchos cambios complejos dado de que, desde un primer momento, ya se había implementado la librería para trabajar con las curvas de Bézier en tres dimensiones. Sin embargo, sí surgió un problema a la hora de detectar si un punto está sobre la curva o no.

Para resolverlo, se ha utilizado la función de proyección. Sin embargo, esto no es suficiente con lo que hemos tenido que buscar el punto de la curva que más cerca está con respecto al segmento que abarca todos los puntos del mundo que existen debajo del ratón.

CAPÍTULO 5

Evaluación de resultados

Durante la realización de este trabajo, se han cumplido los siguientes objetivos:

- Se ha investigado y creado dos métodos para la generación de modelos de naves por parámetros.
- Se ha creado un programa de diseño asistido por ordenador para la creación de modelos de naves espaciales.

Por otro lado, se han realizado las siguientes aportaciones:

- Creación de una librería para el procesado de curvas de Bézier en c++
- Diseño de un algoritmo para la triangulación de cintas
- Implementación de un sistema de teselación siguiendo las especificaciones de OpenGL
- Creación de procedimiento para la subdivisión de una cinta en triángulos y cuadriláteros.

Comparando los prototipos implementados, podemos observar que, los existentes presentan una serie de inconvenientes. Los primeros, necesitan una base de datos de piezas, modeladas a mano, y una información extra que también hay que introducir manualmente. Mientras que, el último algoritmo, tiene una generación de naves extremadamente similares.

Esto nos diferencia de estos algoritmos puesto que nosotros no dependemos de ninguna base de datos mientras que conseguimos la creación de una gran variedad de naves con una serie de parámetros limitados por parte del usuario.

En el caso del programa de diseño asistido por ordenador, se ha propuesto realizar una prueba para comparar el método tradicional (utilizando la herramienta Blender) con la que se ha desarrollado. Esta prueba ha consistido en hacer un modelo similar con los dos programas y medir los tiempos. En el caso de Blender se ha tardado 24 minutos mientras que, en el nuestro, se ha conseguido hacerlo en 13 minutos.

CAPÍTULO 6
Conclusiones

En este proyecto se ha realizado, en primer lugar, un estudio sobre los métodos existentes sobre la generación de modelos de naves. Al no existir una gran cantidad de material, se ha decidido expandir su búsqueda a dos campos relacionados: Generación de ciudades y edificios. A diferencia de otros campos, estos presentan una estructura similar puesto que son elementos artificiales construidos por el hombre que presentan una estructura.

En segundo lugar, se ha propuesto una nueva forma de modelar basada en una estructura de modelos de naves compuesta por un esqueleto y una serie de anillos.

En tercer lugar, hemos creado una serie de prototipos que nos han permitido estudiar sobre la generación de modelos de naves. Aunque estos modelos no dispongan de una gran cantidad de detalles, podemos observar que la ventaja sobre el diseño tradicional es grande puesto que estamos creando un modelo con solo presionar un botón.

Finalmente, se ha creado un programa de diseño por ordenador centrado en las naves que sigue nuestra propuesta de modelado. Además, se ha realizado una pequeña prueba con resultados positivos para este programa, consiguiendo una mejora de 11 minutos sobre 24 con respecto a un programa tradicional.

Por otro lado, del desarrollo de este proyecto se ha conseguido experiencia en el campo de la generación por procedimientos de modelos y la posibilidad de aplicar unos u otros algoritmos. Así mismo, también se ha conseguido experiencia en algunas herramientas gráficas cómo, por ejemplo, las curvas o superficies de Bézier o la teselación.

Además, se han aportado una serie de mecanismos que nos permiten subdividir una cinta en una serie de formas más básicas y, a su vez, dividir estas con un nivel de detalle dependiente del área que ocupa la superficie. Esto contrasta con lo que se ha realizado que consiste en dar una cantidad de detalle fija para todo el modelo.

Para terminar, se han conseguido cumplir con los objetivos propuestos al principio de la memoria. Primero, se ha conseguido crear un algoritmo que tiene la capacidad de generar un modelo de nave. Segundo, se ha realizado un programa de diseño asistido por ordenador centrado en las naves y que permite la incorporación de cualquier generador automático que siga la estructura del modelo descrita.

CAPÍTULO 7

Propuestas de futuro

7.1 Generación de naves

Algunas mejoras que se podrían incluir en la generación de naves, son:

- Generación de anillos utilizando técnicas de algoritmos genéticos
- Estudiar la posibilidad de utilizar las split grammars descritas en el estado del arte

7.2 Programa CAD

Algunas mejoras que se podrían incluir en el programa CAD, son:

- Inclusión de texturas utilizando algoritmos de síntesis
- Inclusión de una herramienta para la generación de secciones intermedias
- Mostrar la sección seleccionada en la previsualización
- creación de un mecanismo para crear nuevos presets.

Bibliografía

[1] D. M. D. Carli, F. Bevilacqua, C. T. Pozzer and M. C. dOrnellas, "*A Survey of Procedural Content Generation Techniques Suitable to Game Development*"2011 Brazilian Symposium on Games and Digital Entertainment, Salvador, 2011, pp. 26-35. doi: 10.1109/SBGAMES.2011.15

[2] LAGAE A., LEFEBVRE S., COOK R., DEROSE T., DRETTAKIS G., EBERT D., LEWIS J., PERLIN K., ZWICKER M.: A survey of procedural noise functions. Computer Graphics Forum 29, 8 (December 2010), 2579–2600. https://doi.org/10.1111/j.1467-8659.2010.01827.x.

[3] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. 2006. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers* (SIGGRAPH '06). ACM, New York, NY, USA, 614-623. DOI: https://doi.org/10.1145/1179352.1141931

[4] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. 2003. Instant architecture. ACM Trans. Graph. 22, 3 (July 2003), 669-677. DOI: https://doi.org/10.1145/882262.882324

[5] Thomas Lechner, Ben Watson and Uri Wilensky. 2003. Procedural city modeling. In *1st Midwestern Graphics Conference*

[6] Togelius, J., Kastbjerg, E., Schedl, D., and Yannakakis, G. N. What is procedural content generation? Mario on the borderline. In *Proc. Procedural Content Generation in Games 2011*, ACM Press (2011)

[7] Yoav I. H. Parish and Pascal Müller. 2001. *Procedural modeling of cities*. In Proceedings of the 28th annual conference on Computer graphics and interactive techniques (SIGGRAPH '01). ACM, New York, NY, USA, 301-308. DOI: https://doi.org/10.1145/383259.383292

[8] The World of Notch. Consultado el 11-09-2017 en `https://notch.tumblr.com/post/3746989361/terrain-generation-part-1`.

[9] Post de ship-not-even-wrong en reddit.

Consultado el 12-09-2017 en `https://www.reddit.com/r/gamedev/comments/1k6i17/a_3d_procedural_spaceship_model_generator/`.

[10] Página de teselación de la wiki oficial de OpenGL Consultado el 15-09-2017 en `https://www.khronos.org/opengl/wiki/Tessellation`.

APÉNDICE A
Gramática completa

Alfabeto:

- START
- BACK
- RAMP
- WALL
- FRONT
- CENTER
- FLAT
- P

Axioma:

P(0,0), START, P(1,0)

Producciones:

START →FRONT Pv([0.3,0.4],[0.2,0.3]) CENTER Pv([0.6,0.7],[0.2,0.3]) BACK (33 %)
| FRONT Pv([0.4,0.6],[0.2,0.3]) CENTER Pv(1,[0.2-0.3]) (33 %)
| FRONT P([0.25,0.75],[0.2,0.3]) BACK (34 %)

HEAD →WALL P(0,[0.05,0.15]) RAMP P([0.90,1.0],[-0.1,0.05]) WALL (20 %)
| WALL P(0,[0.05,0.15]) RAMP Pva([0.3,0.4],[0.3,0.4])
FLAT Pva(1,[0.3,0.4] WALL (20 %)
| RAMP P([0.90,1.0],[-0.1,0.05]) WALL (20 %)
| WALL P(0,[0.05,0.15]) RAMP (20 %)
| RAMP (20 %)

TAIL →RAMP Pva([0.3,0.5],[0.3,0.4]) FLAT Pva(1,[0.3,0.4]) WALL (100 %)

APÉNDICE B
Curvas de Bézier

Definición de la cabecera

Código B.1: Definición de la cabecera

```

#ifndef CUBIC_BEZIER_SPLINE_GENERATOR_H
#define CUBIC_BEZIER_SPLINE_GENERATOR_H


#include <vector>
#include <tuple>
#include <glm/glm.hpp>

class CubicPoliBezierGenerator
{
  public:
  CubicPoliBezierGenerator();
  ~CubicPoliBezierGenerator();

  CubicPoliBezierGenerator& setNumSlices(unsigned int n);
  CubicPoliBezierGenerator& setProfile(
      const std::vector<glm::vec3> &points
    );
  CubicPoliBezierGenerator& buildNumberOfSegments(unsigned int n);
  CubicPoliBezierGenerator& computeNormalsAtExtremes();

  std::vector<glm::vec3> generate();
  std::vector<glm::vec3> generateCasteljauPoints(int forSection, float t);
  std::vector<glm::vec3> extremities();
  std::vector<glm::vec3> boundingBox();

  glm::vec3 tangent(int section,float t);
  std::tuple<glm::vec3, glm::vec3, glm::vec3> tnb(int section,
                                                   float t,
                                                   float mix=0.0f);

  void center();

  float on(glm::vec3 point, float error);
  float project(glm::vec3 point, float error=0.00001);

  std::vector<float> xRoots();

  glm::vec3 getPoint(int section, float t);
  int getNumSections();
  std::vector<glm::vec3> getProfile();

  protected:
```

```
    const static unsigned int grade;
    unsigned int slices;

    bool useDivisionsPerSide;

    std::vector<glm::vec3> profile;
    std::vector<glm::vec3> lut;
    std::vector<int> sectionStartAt;
    std::vector<int> slicesPerSide;

    std::vector<float> angleOfNormalsAtExtremes;

    bool normalsPrecalculated;


  };


  #endif
```

Código B.2: Definición de los métodos

```

#include "CubicPoliBezierGenerator.h"

#define M_PI 3.14159265358979323846
#include <cmath>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/epsilon.hpp>

#define CUBEROOT(x) (x<0) ? -pow(-x, 1.0 / 3.0) : pow(x, 1.0 / 3.0)

const unsigned int CubicPoliBezierGenerator::grade = 4;


CubicPoliBezierGenerator::CubicPoliBezierGenerator()
{
  slices = 10;
  useDivisionsPerSide = false;
  normalsPrecalculated = false;
}


CubicPoliBezierGenerator::~CubicPoliBezierGenerator()
{
}

CubicPoliBezierGenerator & CubicPoliBezierGenerator::setNumSlices(unsigned int
    n)
{
  slices = n;
  useDivisionsPerSide = false;

  return *this;
}

CubicPoliBezierGenerator & CubicPoliBezierGenerator::setProfile(const std::
    vector<glm::vec3>& points)
{
  profile = points;
  useDivisionsPerSide = false;

  normalsPrecalculated = false;

```

```
  return *this;
}

CubicPoliBezierGenerator & CubicPoliBezierGenerator::buildNumberOfSegments(unsigned int n)
{
  if (profile.size() >= grade) {
    slicesPerSide.clear();
    for (unsigned int i = 0; i < profile.size() - (grade - 1); i += grade - 1) {
      float length = 0;
      for (unsigned int j = 0; j < grade - 1; j++) {
        length += glm::distance(profile[i + j], profile[i + j + 1]);
      }
      int nslices = length*n;
      if (nslices == 0) nslices++;
      slicesPerSide.push_back(nslices);
    }
    slices = n;
  }

  return *this;
}

CubicPoliBezierGenerator & CubicPoliBezierGenerator::computeNormalsAtExtremes()
{
  unsigned int sections = (profile.size() - 1) / (grade - 1);
  angleOfNormalsAtExtremes = std::vector<float>(sections * 2);

  std::vector<glm::vec3> tangentsAtExtremes(sections * 2);
  std::vector<glm::vec3> normalsAtExtremes(sections * 2);

  for (int i = 0; i < sections; i++) {

    glm::vec3 tang = tangent(i, 0);
    glm::vec3 tangEnd = tangent(i, 1);
    tangentsAtExtremes[i * 2] = tang;
    tangentsAtExtremes[i * 2 + 1] = tangEnd;

    //calcule normal at start
    glm::vec3 tangAux = tangent(i, 0.01);
    tang = glm::normalize(tang);
    tangAux = glm::normalize(tangAux);
    if(glm::all(glm::epsilonEqual(tang,tangAux, glm::vec3(0.00001f, 0.00001f, 0.00001f)))){
      normalsAtExtremes[i * 2] = glm::vec3(0, 0, 0);
      normalsAtExtremes[i * 2 + 1] = glm::vec3(0, 0, 0);

      angleOfNormalsAtExtremes[i * 2]=0;
      angleOfNormalsAtExtremes[i * 2 + 1] = 0;

      continue;
    }

    glm::vec3 c = glm::cross(tangAux, tang);

    glm::mat3 r = glm::mat3(glm::vec3(c.x*c.x, c.x*c.y + c.z, c.x*c.z - c.y),
      glm::vec3(c.x*c.y - c.z, c.y*c.y, c.y*c.z + c.x),
      glm::vec3(c.x*c.z + c.y, c.y*c.z - c.x, c.z*c.z));

    glm::vec3 normal = r*tang;
    normal = glm::normalize(normal);
    normalsAtExtremes[i * 2] = normal;

```

```

    //calcule normal at end
    glm::vec3 tangAuxEnd = tangent(i, 1.01);
    tangEnd = glm::normalize(tangEnd);
    tangAuxEnd = glm::normalize(tangAuxEnd);

    glm::vec3 cEnd = glm::cross(tangAuxEnd, tangEnd);

    glm::mat3 rEnd = glm::mat3(glm::vec3(cEnd.x*cEnd.x, cEnd.x*cEnd.y + cEnd.z,
        cEnd.x*cEnd.z - cEnd.y),
    glm::vec3(cEnd.x*cEnd.y - cEnd.z, cEnd.y*cEnd.y, cEnd.y*cEnd.z + cEnd.x),
    glm::vec3(cEnd.x*cEnd.z + cEnd.y, cEnd.y*cEnd.z - cEnd.x, cEnd.z*cEnd.z));

    glm::vec3 normalEnd = rEnd*tangEnd;
    normalEnd = glm::normalize(normalEnd);

    normalsAtExtremes[i * 2 + 1] = normalEnd;

  }

  int i = 0;
  while (i<sections) {
    if (normalsAtExtremes[i * 2] != glm::vec3(0, 0, 0)) {
      i++;
      continue;
    }

    float lastNonZero = 0;
    if (i != 0) {
      glm::vec3 normAux = glm::vec3(tangentsAtExtremes[(i - 1) * 2].y, -
          tangentsAtExtremes[(i-1)*2].x, 0);
      float dotProduct = glm::dot(normAux, normalsAtExtremes[(i - 1) * 2]);
      if (dotProduct > 1.0f) dotProduct = 1.0f;
      else if (dotProduct < -1.0f) dotProduct = -1.0f;
      lastNonZero = acos(dotProduct);
    }
    int nextNonZero = 1;
    while (i + nextNonZero < sections && normalsAtExtremes[nextNonZero * 2] ==
        glm::vec3(0, 0, 0) ) {
      nextNonZero++;
    }
    float nextAngleNonZero = 0;
    if (nextNonZero != sections) {

      glm::vec3 normAux = glm::vec3(tangentsAtExtremes[nextNonZero * 2].y, -
          tangentsAtExtremes[nextNonZero * 2].x, 0);
      float dotProduct = glm::dot(normAux, normalsAtExtremes[nextNonZero * 2]);
      if (dotProduct > 1.0f) dotProduct = 1.0f;
      else if (dotProduct < -1.0f) dotProduct = -1.0f;
      nextAngleNonZero = acos(dotProduct);
    }

    angleOfNormalsAtExtremes[i * 2] = lastNonZero;
    angleOfNormalsAtExtremes[(i + nextNonZero - 1) * 2 + 1] = nextAngleNonZero;

    for (int j = 1; j < nextNonZero; j++) {
      float t = (float)j / nextNonZero;
      float angle = (1 - t)*lastNonZero + t*nextAngleNonZero;
      angleOfNormalsAtExtremes[(i + j - 1) * 2 + 1] = angle;
      angleOfNormalsAtExtremes[(i + j) * 2 ] = angle;

    }
    i += nextNonZero;
  }
```

```

  normalsPrecalculated = true;
  return *this;
}

std::vector<glm::vec3> CubicPoliBezierGenerator::generate()
{
  lut.clear();
  sectionStartAt.clear();
  if (profile.size() >= grade) {
    unsigned int sections = (profile.size() - 4) / grade;
    int section = 0;
    for (unsigned int i = 0; i < profile.size() - (grade - 1); i += grade - 1,
        section++) {
      sectionStartAt.push_back(lut.size());
      int s = useDivisionsPerSide ? slicesPerSide[section] : slices;
      double temp;
      for (temp = 0.0; temp + 1.0 / s < 1.0 && glm::epsilonNotEqual(temp + 1.0
          / s, 1.0, 1.0 / (s * 2)); temp += 1.0 / s) {
        lut.push_back(getPoint(section, temp));
      }
      lut.push_back(getPoint(section, 1.0));
    }
    sectionStartAt.push_back(lut.size());

  }
  return lut;
}

std::vector<glm::vec3> CubicPoliBezierGenerator::generateCasteljauPoints(int
    forSection, float t)
{
  if (forSection >= 0) {
    std::vector<glm::vec3> res;
    for (int i = 0; i < grade-1; i++) {
      int pointIndex = forSection * 3 + i;
      glm::vec3 p = profile[pointIndex] + t*(profile[pointIndex + 1] - profile[
          pointIndex]);
      res.push_back(p);
    }
    for (int i = 0; i < grade - 2; i++) {
      glm::vec3 p = res[i] + t*(res[i + 1] - res[i]);
      res.push_back(p);
    }
    glm::vec3 p = res[res.size()-2] + t*(res[res.size() - 1] - res[res.size() -
        2]);
    res.push_back(p);
    return res;
  }
  return std::vector<glm::vec3>();
}

std::vector<glm::vec3> CubicPoliBezierGenerator::extremities()
{

  return std::vector<glm::vec3>();
}

std::vector<glm::vec3> CubicPoliBezierGenerator::boundingBox()
{

  unsigned int sections = (profile.size() - 1) / (grade-1);

  glm::vec3 min = profile[0];
```

```
  glm::vec3 max = profile[0];

  for (int j = 0; j < sections; j++) {
    int i = j * 3;
    glm::vec3 a = 3.0f * profile[i + 3] - 9.0f * profile[i + 2] + 9.0f *
        profile[i + 1] - 3.0f * profile[i];
    glm::vec3 b = 6.0f * profile[i] - 12.0f * profile[i + 1] + 6.0f * profile[i
         + 2];
    glm::vec3 c = 3.0f * profile[i + 1] - 3.0f * profile[i];

    glm::vec3 rootContent = b * b - 4.0f * a * c;

    if (profile[i].x < min.x) min.x = profile[i].x;
    if (profile[i].y < min.y) min.y = profile[i].y;
    if (profile[i].z < min.z) min.z = profile[i].z;

    if (profile[i].x > max.x) max.x = profile[i].x;
    if (profile[i].y > max.y) max.y = profile[i].y;
    if (profile[i].z > max.z) max.z = profile[i].z;


    //x component
    if (a.x != 0) {
      if (rootContent.x >= 0) {
        float t = (-b.x + sqrt(rootContent.x)) / (2 * a.x);

        if (t > 0 && t < 1) {
          glm::vec3 p = getPoint(j, t);
          if (p.x < min.x) min.x = p.x;
          if (p.x > max.x) max.x = p.x;
        }

        t = (-b.x - sqrt(rootContent.x)) / (2 * a.x);

        if (t > 0 && t < 1) {
          glm::vec3 p = getPoint(j, t);
          if (p.x < min.x) min.x = p.x;
          if (p.x > max.x) max.x = p.x;
        }
      }
    }
    else {
      float t = -c.x / b.x;
      if (t > 0 && t < 1) {
        glm::vec3 p = getPoint(j, t);
        if (p.x < min.x) min.x = p.x;
        if (p.x > max.x) max.x = p.x;
      }
    }
    //y component
    if (a.y != 0) {
      if (rootContent.y >= 0) {
        float t = (-b.y + sqrt(rootContent.y)) / (2 * a.y);

        if (t > 0 && t < 1) {
          glm::vec3 p = getPoint(j, t);
          if (p.y < min.y) min.y = p.y;
          if (p.y > max.y) max.y = p.y;
        }

        t = (-b.y - sqrt(rootContent.y)) / (2 * a.y);

        if (t > 0 && t < 1) {
          glm::vec3 p = getPoint(j, t);
```

```
          if (p.y < min.y) min.y = p.y;
          if (p.y > max.y) max.y = p.y;
        }
      }
    }
    else {
      float t = -c.y / b.y;
      if (t > 0 && t < 1) {
        glm::vec3 p = getPoint(j, t);
        if (p.y < min.y) min.y = p.y;
        if (p.y > max.y) max.y = p.y;
      }
    }
    //z component
    if (a.z != 0) {
      if (rootContent.z >= 0) {
        float t = (-b.z + sqrt(rootContent.z)) / (2 * a.z);

        if (t > 0 && t < 1) {
          glm::vec3 p = getPoint(j, t);
          if (p.z < min.z) min.x = p.z;
          if (p.z > max.z) max.x = p.z;
        }

        t = (-b.z - sqrt(rootContent.z)) / (2 * a.z);

        if (t > 0 && t < 1) {
          glm::vec3 p = getPoint(j, t);
          if (p.z < min.z) min.z = p.z;
          if (p.z > max.z) max.z = p.z;
        }
      }
    }
    else {
      float t = -c.z / b.z;
      if (t > 0 && t < 1) {
        glm::vec3 p = getPoint(j, t);
        if (p.z < min.z) min.z = p.z;
        if (p.z > max.z) max.z = p.z;
      }
    }
  }

  if (profile[sections * 3].x < min.x) min.x = profile[sections * 3].x;
  if (profile[sections * 3].y < min.y) min.y = profile[sections * 3].y;
  if (profile[sections * 3].z < min.z) min.z = profile[sections * 3].z;

  if (profile[sections * 3].x > max.x) max.x = profile[sections * 3].x;
  if (profile[sections * 3].y > max.y) max.y = profile[sections * 3].y;
  if (profile[sections * 3].z > max.z) max.z = profile[sections * 3].z;

  std::vector<glm::vec3> res(2);
  res[0] = min; res[1] = max;
  return res;

}

glm::vec3 CubicPoliBezierGenerator::tangent(int section, float t)
{
  float mt = 1 - t,
  a = 3*mt*mt, b = 6*mt * t , c = 3*t*t;

  int startPoint = section * 3;

```

```
  glm::vec3 res = a*(profile[startPoint + 1] - profile[startPoint]) +
  b*(profile[startPoint + 2] - profile[startPoint + 1]) +
  c*(profile[startPoint + 3] - profile[startPoint + 2]);
  if (res == glm::vec3(0, 0, 0)) {
    res = profile[startPoint + 2] - profile[startPoint + 1];
  }
  res = glm::normalize(res);
  return res;
}

std::tuple<glm::vec3, glm::vec3, glm::vec3> CubicPoliBezierGenerator::tnb(int
    section, float t, float mix)
{
  glm::vec3 tang;
  glm::vec3 tangAux;
  if (t >= 1.0f) {
    tang = tangent(section, t-0.01);
    tangAux = tangent(section, t);
  }
  else {
    tang = tangent(section, t);
    tangAux = tangent(section, t + 0.01);
  }
  tang = glm::normalize(tang);
  tangAux = glm::normalize(tangAux);

  glm::vec3 normal = glm::vec3(tang.y, -tang.x, tang.z);
  glm::vec3 binormal = glm::cross(tang,normal);
  binormal = glm::normalize(binormal);

  return std::make_tuple(tang,normal,binormal);
}

void CubicPoliBezierGenerator::center()
{
  std::vector<glm::vec3> bb = boundingBox();
  glm::vec3 despl = -(bb[0] + (bb[1] - bb[0])/2.0f);
  for (unsigned int i = 0; i < profile.size(); i++) {
    profile[i] += despl;
  }
}

float CubicPoliBezierGenerator::on(glm::vec3 point, float error)
{
  float minDistance = INFINITY;
  float minSection = -1;
  std::vector<float> t;
  for (int i = 0; i < sectionStartAt.size()-1; i++) {
    t.push_back(0);
    int hits = 0;
    for (int j = sectionStartAt[i]; j < sectionStartAt[i + 1]; j++) {
      float dist = glm::distance(point, lut[j]);
      if (dist < error) {
        t.back() += (float)(j - sectionStartAt[i]) / (sectionStartAt[i + 1] -
            sectionStartAt[i]);
        hits++;
        if (minDistance > dist) {
          minDistance = dist;
          minSection = i;
        }
      }
    }
    t.back() /= hits;
  }
```

```
  if (minSection == -1) return -1;
  return minSection + t[minSection];

}

float CubicPoliBezierGenerator::project(glm::vec3 point, float error)
{
  float minDistance = INFINITY;
  int minSection = -1;
  int minLUT;
  for (int i = 0; i < sectionStartAt.size() - 1; i++) {
    for (int j = sectionStartAt[i]; j < sectionStartAt[i + 1]; j++) {
      float dist = glm::distance(point, lut[j]);
      if (minDistance > dist) {
        minDistance = dist;
        minSection = i;
        minLUT = j;
      }

    }
  }

  if (minLUT == sectionStartAt[minSection ] || minLUT == sectionStartAt[minSection + 1]) {
    return minSection + (float)(minLUT - sectionStartAt[minSection]) / (sectionStartAt[minSection + 1] - sectionStartAt[minSection]);
  }

  // step 2: fine check
  float t = (float)(minLUT - sectionStartAt[minSection]) / (sectionStartAt[minSection + 1] - sectionStartAt[minSection]);
  float incrementT = 1 / (sectionStartAt[minSection + 1] - sectionStartAt[minSection]);
  while (true) {
    glm::vec3 p = getPoint(minSection, t);
    glm::vec3 tPlusInter = getPoint(minSection, t + incrementT / 2);
    glm::vec3 tMinusInter = getPoint(minSection, t - incrementT / 2);
    float distPPlus = glm::distance(point,tPlusInter);
    float distPMinus = glm::distance(point, tMinusInter);
    float distPP = glm::distance(point, p);

    float diffPPlus = distPPlus > distPP ? distPPlus - distPP : distPP - distPPlus;
    float diffPMinus = distPMinus > distPP ? distPMinus - distPP : distPP - distPMinus;

    if (diffPPlus <= error || diffPMinus <= error) break;
    else if (distPPlus < distPP) {
      t = t + incrementT / 2;
    }
    else if (distPMinus < distPP) {
      t = t - incrementT / 2;
    }
    else {
      incrementT /= 2;
    }
  }

  return minSection + t;
}

std::vector<float> CubicPoliBezierGenerator::xRoots()
{
  std::vector<float> result;
```

```
  for (int i = 0, section = 0; i + 3 < profile.size(); i += 3, section++) {
    double pa = profile[i].x, pb = profile[i + 1].x, pc = profile[i + 2].x, pd = profile[i + 3].x;

    double d = (-pa + 3 * pb - 3 * pc + pd),
    a = (3 * pa - 6 * pb + 3 * pc) / d,
    b = (-3 * pa + 3 * pb) / d,
    c = pa / d;
    double p = (3 * b - a*a) / 3,
    p3 = p / 3,
    q = (2 * a*a*a - 9 * a*b + 27 * c) / 27,
    q2 = q / 2,
    discriminant = q2*q2 + p3*p3*p3;

    if (glm::epsilonEqual(discriminant, 0.0000, 0.0000001)) {
      double u1 = q2 < 0 ? CUBEROOT(-q2) : -CUBEROOT(q2);
      double root1 = 2 * u1 - a / 3;
      double root2 = -u1 - a / 3;

      if (root1 >= 0 && root1 < 1) result.push_back(section + root1);
      if (root2 >= 0 && root2 < 1) result.push_back(section + root2);

    }
    else if (discriminant < 0) {
      double mp3 = -p / 3,
      mp33 = mp3*mp3*mp3,
      r = sqrt(mp33),
      t = -q / (2 * r),
      cosphi = t<-1 ? -1 : t>1 ? 1 : t,
      phi = acos(cosphi),
      crtr = CUBEROOT(r),
      t1 = 2 * crtr;
      double root1 = t1 * cos(phi / 3) - a / 3;
      double root2 = t1 * cos((phi + 2 * M_PI) / 3) - a / 3;
      double root3 = t1 * cos((phi + 4 * M_PI) / 3) - a / 3;

      if (root1 >= 0 && root1 < 1) result.push_back(section + root1);
      if (root2 >= 0 && root2 < 1) result.push_back(section + root2);
      if (root3 >= 0 && root3 < 1) result.push_back(section + root3);

    }
    else {
      double sd = sqrt(discriminant);
      double u1 = CUBEROOT(sd - q2);
      double v1 = CUBEROOT(sd + q2);
      double root1 = u1 - v1 - a / 3;

      if (root1 >= 0 && root1 < 1) result.push_back(section + root1);

    }
  }
  return result;
}

glm::vec3 CubicPoliBezierGenerator::getPoint(int section, float t)
{
  int nsect = (profile.size() - 1) / (grade - 1);
  if (nsect > section) {
    int k = grade - 1;

    int startPoint = section * k;

    float invertedT = 1 - t;
    float squaredit = invertedT * invertedT;
```

```
    float cubicit = squaredit  * invertedT;

    float squaredT = t*t;
    float cubicT = squaredT*t;

    glm::vec3 Pt = cubicit*profile[startPoint] + 3 * squaredit*t*profile[
        startPoint + 1] + 3 * invertedT*squaredT*profile[startPoint + 2] +
        cubicT*profile[startPoint + 3];
    return Pt;
  }

  return glm::vec3();
}

int CubicPoliBezierGenerator::getNumSections()
{
  return (profile.size() - 1) / (grade - 1);
}

std::vector<glm::vec3> CubicPoliBezierGenerator::getProfile()
{
  return profile;
}
```

APÉNDICE C

Teselación de un parche de Bézier

Código C.1: cuadrilátero

```
void ShipBodyModel::triangulateQuadrilateral(const std::vector<glm::vec3> &
    curveA, const std::vector<glm::vec3> &curveB, float subdivisions, float
    scale, std::vector<glm::vec3>* meshVertex, std::vector<GLuint>* meshIndex,
    std::vector<glm::vec3>* meshNormals, std::vector<glm::vec2>* meshRS)
{
  const int vertexStart = meshVertex->size();
  const int indexStart = meshIndex->size();
  const int normalsStart = meshNormals->size();
  const int uvStart = meshRS->size();



  const int outerSub[4] = {
    glm::length(curveA[3] - curveA[0])*subdivisions + 2,
    glm::length(curveB[3] - curveA[3])*subdivisions + 2,
    glm::length(curveB[3] - curveB[0])*subdivisions + 2,
    glm::length(curveA[0] - curveB[0])*subdivisions + 2
  };

  const int minXSubs = glm::min(outerSub[0], outerSub[2]);
  const int maxXSubs = glm::max(outerSub[0], outerSub[2]);

  const int minYSubs = glm::min(outerSub[1], outerSub[3]);
  const int maxYSubs = glm::max(outerSub[1], outerSub[3]);

  const int innerSub[2] = { (minXSubs + maxXSubs) / 2,(minYSubs + maxYSubs) / 2
      };

  std::vector<glm::vec2> UV;


  std::vector<glm::vec2> outerRing, innerRing;

  //--------
  //First
  //Ring
  //--------

  //A Segment
  for (int i = 0; i < outerSub[0]; i++) {
    glm::vec2 point(i * 1.0f / outerSub[0], 0.0f);
    UV.push_back(point);
    outerRing.push_back(point);
  }


```

```
//AB Segment
for (int i = 0; i < outerSub[1]; i++) {
  glm::vec2 point(1.0f, i * 1.0f / outerSub[1]);
  UV.push_back(point);
  outerRing.push_back(point);
}


//B Segment
//inverse
for (int i = 0; i < outerSub[2]; i++) {
  glm::vec2 point((outerSub[2] - i) * 1.0f / outerSub[2], 1.0f);
  UV.push_back(point);
  outerRing.push_back(point);
}


//BA Segment
//inverse
for (int i = 0; i < outerSub[3]; i++) {
  glm::vec2 point(0.0f, (outerSub[3] - i) * 1.0f / outerSub[3]);
  UV.push_back(point);
  outerRing.push_back(point);
}


int AStart = vertexStart, BStart = AStart + outerRing.size();



//Base cases
if (innerSub[0] == 1) {
  meshIndex->push_back(AStart + 0);
  meshIndex->push_back(AStart + 1);
  meshIndex->push_back(AStart + outerRing.size() - 1);

  meshIndex->push_back(AStart + 1);
  meshIndex->push_back(AStart + 2);
  meshIndex->push_back(AStart + outerRing.size() - 1);

  for (int i = 2; i <= outerRing.size() / 2; i += 1) {
    meshIndex->push_back(AStart + outerRing.size() - i + 1);
    meshIndex->push_back(AStart + i);
    meshIndex->push_back(AStart + outerRing.size() - i);

    meshIndex->push_back(AStart + i);
    meshIndex->push_back(AStart + i + 1);
    meshIndex->push_back(AStart + outerRing.size() - i);

  }

}
else if (innerSub[1] == 1) {
  for (int i = 0; i + 1 < outerRing.size() / 2; i += 1) {
    meshIndex->push_back(AStart + i);
    meshIndex->push_back(AStart + outerRing.size() - i - 2);
    meshIndex->push_back(AStart + outerRing.size() - i - 1);

    meshIndex->push_back(AStart + i);
    meshIndex->push_back(AStart + i + 1);
    meshIndex->push_back(AStart + outerRing.size() - i - 2);

  }
}
```

```
else if (innerSub[0] == 2) {
  int rings = 1;
  for (int i = rings; i + rings <= innerSub[1]; i++) {

    glm::vec2 point(rings * 1.0f / innerSub[0], i * 1.0f / innerSub[1]);
    UV.push_back(point);
    innerRing.push_back(point);
  }

  int despA = 0;


  for (int i = 0; i < outerSub[0]; i++) {
    meshIndex->push_back(AStart + 1 + i + despA);
    meshIndex->push_back(BStart + 0);
    meshIndex->push_back(AStart + 0 + i + despA);

  }

  despA += outerSub[0];
  int i = 0, j = 0;
  while (i < outerSub[1] && j + 1 < innerRing.size()) {
    glm::vec2 distItoJ = outerRing[despA + i] - innerRing[j + 1];
    glm::vec2 distJtoI = outerRing[despA + i + 1] - innerRing[j];
    float lengthIToJ = glm::length(distItoJ), lengthJToI = glm::length(
        distJtoI);
    if (glm::length(distItoJ) < glm::length(distJtoI)) {
      meshIndex->push_back(BStart + j);
      meshIndex->push_back(AStart + i + despA);
      meshIndex->push_back(BStart + j + 1);
      j++;
    }
    else {
      meshIndex->push_back(BStart + j);
      meshIndex->push_back(AStart + i + despA);
      meshIndex->push_back(AStart + i + 1 + despA);
      i++;
    }
  }

  while (i < outerSub[1]) {
    meshIndex->push_back(BStart + j);
    meshIndex->push_back(AStart + i + despA);
    meshIndex->push_back(AStart + i + 1 + despA);
    i = i + 1;
  }


  despA += outerSub[1];
  for (int i = 0; i < outerSub[2]; i++) {
    meshIndex->push_back(AStart + 1 + i + despA);
    meshIndex->push_back(BStart + innerRing.size() - 1);
    meshIndex->push_back(AStart + 0 + i + despA);
  }

  i = 0, j = innerRing.size() -1;
  despA += outerSub[2];
  while (i < outerSub[3] && j > 0) {
    glm::vec2 distItoJ = outerRing[despA + i] - innerRing[j - 1];
    glm::vec2 distJtoI = outerRing[(despA + i + 1)%outerRing.size()] -
        innerRing[j];
    float lengthIToJ = glm::length(distItoJ), lengthJToI = glm::length(
        distJtoI);
    if (glm::length(distItoJ) < glm::length(distJtoI)) {
```

```
        meshIndex->push_back(BStart + j);
        meshIndex->push_back(AStart + i + despA);
        meshIndex->push_back(BStart + j - 1);
        j--;
      }
      else {
        meshIndex->push_back(BStart + j);
        meshIndex->push_back(AStart + i + despA);
        meshIndex->push_back(AStart + i + 1 + despA);
        i++;
      }
    }
    while (i < outerSub[3]) {
      meshIndex->push_back(BStart + j);
      meshIndex->push_back(AStart + i + despA);
      meshIndex->push_back(AStart + (i + 1 + despA)%outerRing.size());
      i = i + 1;
    }


  }
  else if (innerSub[1] == 2) {
    int rings = 1;
    for (int i = rings; i + rings <= innerSub[0]; i++) {
      glm::vec2 point(i * 1.0f / innerSub[0], rings * 1.0f / innerSub[1]);
      UV.push_back(point);
      innerRing.push_back(point);
    }

    int despA = 0;

    int i = 0, j = 0;
    while (i < outerSub[0] && j + 1 < innerRing.size()) {
      glm::vec2 distItoJ = outerRing[despA + i] - innerRing[j + 1];
      glm::vec2 distJtoI = outerRing[despA + i + 1] - innerRing[j];
      float lengthIToJ = glm::length(distItoJ), lengthJToI = glm::length(
          distJtoI);
      if (glm::length(distItoJ) < glm::length(distJtoI)) {
        meshIndex->push_back(BStart + j);
        meshIndex->push_back(AStart + i + despA);
        meshIndex->push_back(BStart + j + 1);
        j++;
      }
      else {
        meshIndex->push_back(BStart + j);
        meshIndex->push_back(AStart + i + despA);
        meshIndex->push_back(AStart + i + 1 + despA);
        i++;
      }
    }

    while (i < outerSub[0]) {
      meshIndex->push_back(BStart + j);
      meshIndex->push_back(AStart + i + despA);
      meshIndex->push_back(AStart + i + 1 + despA);
      i = i + 1;
    }

    despA += outerSub[0];

    for (int i = 0; i < outerSub[1]; i++) {
      meshIndex->push_back(AStart + 1 + i + despA);
      meshIndex->push_back(BStart + innerRing.size() - 1);
      meshIndex->push_back(AStart + 0 + i + despA);
```

```
    }

    i = 0, j = innerRing.size() - 1;
    despA += outerSub[1];

    while (i < outerSub[2] && j > 0) {
      glm::vec2 distItoJ = outerRing[despA + i] - innerRing[j - 1];
      glm::vec2 distJtoI = outerRing[(despA + i + 1) % outerRing.size()] -
          innerRing[j];
      float lengthIToJ = glm::length(distItoJ), lengthJToI = glm::length(
          distJtoI);
      if (glm::length(distItoJ) < glm::length(distJtoI)) {
        meshIndex->push_back(BStart + j);
        meshIndex->push_back(AStart + i + despA);
        meshIndex->push_back(BStart + j - 1);
        j--;
      }
      else {
        meshIndex->push_back(BStart + j);
        meshIndex->push_back(AStart + i + despA);
        meshIndex->push_back(AStart + i + 1 + despA);
        i++;
      }
    }
    while (i < outerSub[2]) {
      meshIndex->push_back(BStart + j);
      meshIndex->push_back(AStart + i + despA);
      meshIndex->push_back(AStart + (i + 1 + despA));
      i = i + 1;
    }

    despA += outerSub[2];
    for (int i = 0; i <= outerSub[3]; i++) {
      meshIndex->push_back(AStart + (1 + i + despA) % outerRing.size());
      meshIndex->push_back(BStart + 0);
      meshIndex->push_back(AStart + 0 + i + despA);

    }
  }
  else {



    //--------
    //Second
    //Ring
    //--------

    //inner A Segment (x)
    for (int i = 1; i + 1 < innerSub[0]; i++) {
      glm::vec2 point(i * 1.0f / innerSub[0], 1.0f / innerSub[1]);
      UV.push_back(point);
      innerRing.push_back(point);
    }


    //inner AB Segment (y)
    for (int i = 1; i + 1 < innerSub[1]; i++) {
      glm::vec2 point(1.0f - 1.0f / innerSub[0], i * 1.0f / innerSub[1]);
      UV.push_back(point);
      innerRing.push_back(point);
    }

    //inner B Segment (x)
```

```
    //inverse
    for (int i = 1; i + 1 < innerSub[0]; i++) {
      glm::vec2 point((innerSub[0] - i) * 1.0f / innerSub[0], 1.0f - 1.0f / innerSub[1]);
      UV.push_back(point);
      innerRing.push_back(point);
    }

    //inner BA Segment (y)
    //inverse
    for (int i = 1; i + 1 < innerSub[1]; i++) {
      glm::vec2 point(0.0f + 1.0f / innerSub[0], (innerSub[1] - i) * 1.0f / innerSub[1]);
      UV.push_back(point);
      innerRing.push_back(point);
    }

    //make triangle index for first and second ring (diferent number of points)
    int i = 0, j = 0;
    int segment = 0;
    int maxI = 0, maxJ = 0;
    while (segment < 4) {
      maxI += outerSub[segment];
      maxJ += innerSub[segment % 2] - 2;
      while (i < maxI && j < maxJ) {
        glm::vec2 distItoJ = outerRing[i] - innerRing[(j + 1) % innerRing.size()];
        glm::vec2 distJtoI = outerRing[(i + 1) % outerRing.size()] - innerRing[j];
        float lengthIToJ = glm::length(distItoJ), lengthJToI = glm::length(distJtoI);
        if (glm::length(distItoJ) < glm::length(distJtoI)) {
          meshIndex->push_back(BStart + j % innerRing.size());
          meshIndex->push_back(AStart + i % outerRing.size());
          meshIndex->push_back(BStart + (j + 1) % innerRing.size());
          j++;
        }
        else {
          meshIndex->push_back(BStart + j % innerRing.size());
          meshIndex->push_back(AStart + i % outerRing.size());
          meshIndex->push_back(AStart + (i + 1) % outerRing.size());
          i++;
        }
      }
      while (i < maxI && i < outerRing.size()) {
        meshIndex->push_back(BStart + j % innerRing.size());
        meshIndex->push_back(AStart + i % outerRing.size());
        meshIndex->push_back(AStart + (i + 1) % outerRing.size());
        i = i + 1;
      }
      while (j < maxJ && j < innerRing.size()) {
        meshIndex->push_back(BStart + j % innerRing.size());
        meshIndex->push_back(AStart + i % outerRing.size());
        meshIndex->push_back(BStart + (j + 1) % innerRing.size());

        j = j + 1;
      }
      segment++;
    }

    //update starting point for indexing points
    AStart = BStart;
    BStart += innerRing.size();
    //pass nxt ring
```

```
    outerRing = innerRing;
    innerRing.clear();



    //do the same with all grid rings
    int rings = 1;
    while (innerSub[0] - rings * 2 > 2 && innerSub[1] - rings * 2 > 2) {
      rings++;


      //inner A Segment (x)
      for (int i = rings; i + rings < innerSub[0]; i++) {
        glm::vec2 point(i * 1.0f / innerSub[0], rings * 1.0f / innerSub[1]);
        UV.push_back(point);
        innerRing.push_back(point);
      }


      //inner AB Segment (y)
      for (int i = rings; i + rings < innerSub[1]; i++) {
        glm::vec2 point(1.0f - rings * 1.0f / innerSub[0], i * 1.0f / innerSub[1]);
        UV.push_back(point);
        innerRing.push_back(point);
      }

      //inner B Segment (x)
      //inverse
      for (int i = rings; i + rings < innerSub[0]; i++) {
        glm::vec2 point((innerSub[0] - i) * 1.0f / innerSub[0], 1.0f - rings * 1.0f / innerSub[1]);
        UV.push_back(point);
        innerRing.push_back(point);
      }

      //inner BA Segment (y)
      //inverse
      for (int i = rings; i + rings < innerSub[1]; i++) {
        glm::vec2 point(rings * 1.0f / innerSub[0], (innerSub[1] - i) * 1.0f / innerSub[1]);
        UV.push_back(point);
        innerRing.push_back(point);
      }


      //make triangle index TODO: Make a proper way to do this

      int i = 0, j = 0;
      int segment = 0;
      int maxI = 0, maxJ = 0;
      while (segment < 4) {
        maxI += innerSub[segment %2] - 2 * (rings - 1);
        maxJ += innerSub[segment %2] - 2 * rings;
        while (i < maxI && j < maxJ) {
          glm::vec2 distItoJ = outerRing[i] - innerRing[(j + 1) %innerRing.size()];
          glm::vec2 distJtoI = outerRing[(i + 1) %outerRing.size()] - innerRing[j];
          float lengthIToJ = glm::length(distItoJ), lengthJToI = glm::length(distJtoI);
          if (glm::length(distItoJ) < glm::length(distJtoI)) {
            meshIndex->push_back(BStart + j %innerRing.size());
            meshIndex->push_back(AStart + i %outerRing.size());
```

```
            meshIndex->push_back(BStart + (j + 1) % innerRing.size());
            j++;
          }
          else {
            meshIndex->push_back(BStart + j % innerRing.size());
            meshIndex->push_back(AStart + i % outerRing.size());
            meshIndex->push_back(AStart + (i + 1) % outerRing.size());
            i++;
          }
        }
        while (i < maxI && i < outerRing.size()) {
          meshIndex->push_back(BStart + j % innerRing.size());
          meshIndex->push_back(AStart + i % outerRing.size());
          meshIndex->push_back(AStart + (i + 1) % outerRing.size());
          i = i + 1;
        }
        while (j < maxJ && j < innerRing.size()) {
          meshIndex->push_back(BStart + j % innerRing.size());
          meshIndex->push_back(AStart + i % outerRing.size());
          meshIndex->push_back(BStart + (j + 1) % innerRing.size());

          j = j + 1;
        }
        segment++;
      }

      //update starting point for indexing points
      AStart = BStart;
      BStart += innerRing.size();


      outerRing = innerRing;
      innerRing.clear();

    }

    int xLeft = innerSub[0] - rings * 2;
    int yLeft = innerSub[1] - rings * 2;

    if (xLeft == 1) {
      meshIndex->push_back(AStart + 0);
      meshIndex->push_back(AStart + 1);
      meshIndex->push_back(AStart + outerRing.size() - 1);

      meshIndex->push_back(AStart + 1);
      meshIndex->push_back(AStart + 2);
      meshIndex->push_back(AStart + outerRing.size() - 1);

      for (int i = 2; i <= outerRing.size() / 2; i += 1) {
        meshIndex->push_back(AStart + outerRing.size() - i + 1);
        meshIndex->push_back(AStart + i);
        meshIndex->push_back(AStart + outerRing.size() - i);

        meshIndex->push_back(AStart + i);
        meshIndex->push_back(AStart + i + 1);
        meshIndex->push_back(AStart + outerRing.size() - i);

      }

    }
    else if (yLeft == 1) {
      for (int i = 0; i + 1 < outerRing.size() / 2; i += 1) {
        meshIndex->push_back(AStart + i);
        meshIndex->push_back(AStart + outerRing.size() - i - 2);
```

```
        meshIndex->push_back(AStart + outerRing.size() - i - 1);

        meshIndex->push_back(AStart + i);
        meshIndex->push_back(AStart + i + 1);
        meshIndex->push_back(AStart + outerRing.size() - i - 2);

      }
    }
    else if (xLeft == 2) {
      rings++;
      for (int i = rings; i + rings <= innerSub[1]; i++) {

        glm::vec2 point(rings * 1.0f / innerSub[0], i * 1.0f / innerSub[1]);
        UV.push_back(point);
        innerRing.push_back(point);
      }

      meshIndex->push_back(AStart + 0);
      meshIndex->push_back(BStart + 0);
      meshIndex->push_back(AStart + outerRing.size() - 1);

      meshIndex->push_back(AStart + 1);
      meshIndex->push_back(BStart + 0);
      meshIndex->push_back(AStart + 0);

      meshIndex->push_back(AStart + 2);
      meshIndex->push_back(BStart + 0);
      meshIndex->push_back(AStart + 1);

      meshIndex->push_back(AStart + 3);
      meshIndex->push_back(BStart + 0);
      meshIndex->push_back(AStart + 2);


      const int ALast = outerRing.size() - 1;

      for (int i = 0; i + 1 < innerRing.size(); i++) {
        meshIndex->push_back(AStart + 3 + i);
        meshIndex->push_back(AStart + 4 + i);
        meshIndex->push_back(BStart + i);

        meshIndex->push_back(AStart + 4 + i);
        meshIndex->push_back(BStart + 1 + i);
        meshIndex->push_back(BStart + 0 + i);

        meshIndex->push_back(AStart + ALast - i - 1);
        meshIndex->push_back(BStart + 0 + i);
        meshIndex->push_back(BStart + 1 + i);

        meshIndex->push_back(AStart + ALast - i - 1);
        meshIndex->push_back(AStart + ALast - i);
        meshIndex->push_back(BStart + i);
      }

      const int AHalf = outerRing.size() / 2;
      const int BLast = innerRing.size() - 1;

      meshIndex->push_back(AStart + AHalf - 1);
      meshIndex->push_back(AStart + AHalf);
      meshIndex->push_back(BStart + BLast);

      meshIndex->push_back(AStart + AHalf);
      meshIndex->push_back(AStart + AHalf + 1);
      meshIndex->push_back(BStart + BLast);
```

```

      meshIndex->push_back(AStart + AHalf + 1);
      meshIndex->push_back(AStart + AHalf + 2);
      meshIndex->push_back(BStart + BLast);

      meshIndex->push_back(AStart + AHalf + 2);
      meshIndex->push_back(AStart + AHalf + 3);
      meshIndex->push_back(BStart + BLast);
    }
    else if (yLeft == 2) {
      rings++;
      for (int i = rings; i + rings <= innerSub[0]; i++) {
        glm::vec2 point(i * 1.0f / innerSub[0], rings * 1.0f / innerSub[1]);
        UV.push_back(point);
        innerRing.push_back(point);
      }


      const int ALast = outerRing.size() - 1;

      meshIndex->push_back(AStart + 1);
      meshIndex->push_back(BStart + 0);
      meshIndex->push_back(AStart + 0);

      meshIndex->push_back(AStart + 0);
      meshIndex->push_back(BStart + 0);
      meshIndex->push_back(AStart + ALast);

      meshIndex->push_back(AStart + ALast - 1);
      meshIndex->push_back(AStart + ALast);
      meshIndex->push_back(BStart + 0);

      meshIndex->push_back(AStart + ALast - 2);
      meshIndex->push_back(AStart + ALast - 1);
      meshIndex->push_back(BStart + 0);




      for (int i = 0; i + 1 < innerRing.size(); i++) {
        meshIndex->push_back(AStart + 1 + i);
        meshIndex->push_back(AStart + 2 + i);
        meshIndex->push_back(BStart + i);

        meshIndex->push_back(AStart + 2 + i);
        meshIndex->push_back(BStart + 1 + i);
        meshIndex->push_back(BStart + 0 + i);

        meshIndex->push_back(AStart + ALast - i - 3);
        meshIndex->push_back(BStart + 0 + i);
        meshIndex->push_back(BStart + 1 + i);

        meshIndex->push_back(AStart + ALast - i - 3);
        meshIndex->push_back(AStart + ALast - i - 2);
        meshIndex->push_back(BStart + i);
      }

      const int AHalf = outerRing.size() / 2;
      const int BLast = innerRing.size() - 1;

      meshIndex->push_back(AStart + AHalf - 3);
      meshIndex->push_back(AStart + AHalf - 2);
      meshIndex->push_back(BStart + BLast);

```

```
      meshIndex->push_back(AStart + AHalf - 2);
      meshIndex->push_back(AStart + AHalf - 1);
      meshIndex->push_back(BStart + BLast);

      meshIndex->push_back(AStart + AHalf - 1);
      meshIndex->push_back(AStart + AHalf);
      meshIndex->push_back(BStart + BLast);

      meshIndex->push_back(AStart + AHalf);
      meshIndex->push_back(AStart + AHalf + 1);
      meshIndex->push_back(BStart + BLast);
    }

  }
  const glm::mat4 M(glm::vec4(-1, 3, -3, 1), glm::vec4(3, -6, 3, 0), glm::vec4
      (-3, 3, 0, 0), glm::vec4(1, 0, 0, 0));




  glm::vec4 pAx, pAy, pAz, pBx, pBy, pBz;
  if (glm::all(glm::equal(curveA[0], curveA[1])) && glm::all(glm::equal(curveA
      [2], curveA[3])) ) {
    glm::vec3 a03(glm::mix(curveA[0],curveA[3],0.3f)), a06(glm::mix(curveA[0],
        curveA[3], 0.6f));

    pAx = glm::vec4(curveA[0].x, a03.x, a06.x, curveA[3].x);
    pAy = glm::vec4(curveA[0].y, a03.y, a06.y, curveA[3].y);
    pAz = glm::vec4(curveA[0].z, a03.z, a06.z, curveA[3].z);

  }
  else {
    pAx = glm::vec4(curveA[0].x, curveA[1].x, curveA[2].x, curveA[3].x);
    pAy = glm::vec4(curveA[0].y, curveA[1].y, curveA[2].y, curveA[3].y);
    pAz = glm::vec4(curveA[0].z, curveA[1].z, curveA[2].z, curveA[3].z);

  }

  if (glm::all(glm::equal(curveB[0], curveB[1])) && glm::all(glm::equal(curveB
      [2], curveB[3]))) {
    glm::vec3 b03(glm::mix(curveB[0], curveB[3], 0.3f)), b06(glm::mix(curveB
        [0], curveB[3], 0.6f));

    pBx = glm::vec4(curveB[0].x, b03.x, b06.x, curveB[3].x);
    pBy = glm::vec4(curveB[0].y, b03.y, b06.y, curveB[3].y);
    pBz = glm::vec4(curveB[0].z, b03.z, b06.z, curveB[3].z);
  }
  else {
    pBx = glm::vec4(curveB[0].x, curveB[1].x, curveB[2].x, curveB[3].x);
    pBy = glm::vec4(curveB[0].y, curveB[1].y, curveB[2].y, curveB[3].y);
    pBz = glm::vec4(curveB[0].z, curveB[1].z, curveB[2].z, curveB[3].z);
  }

  const glm::mat4 MGMx = M* glm::mat4(pAx,glm::mix(pAx,pBx,0.3f), glm::mix(pAx,
       pBx, 0.6f),pBx) * glm::transpose(M);
  const glm::mat4 MGMy = M* glm::mat4(pAy, glm::mix(pAy, pBy, 0.3f), glm::mix(
      pAy, pBy, 0.6f), pBy) * glm::transpose(M);
  const glm::mat4 MGMz = M* glm::mat4(pAz, glm::mix(pAz, pBz, 0.3f), glm::mix(
      pAz, pBz, 0.6f), pBz) * glm::transpose(M);

  for (int i = 0; i < UV.size(); i++) {
    glm::vec2 uv = UV[i];
    glm::vec4 U(uv.x*uv.x*uv.x, uv.x*uv.x, uv.x, 1), V(uv.y*uv.y*uv.y, uv.y*uv.
        y, uv.y, 1);
```

```
    float px = glm::dot(U,MGMx*V);
    float py = glm::dot(U, MGMy*V);
    float pz = glm::dot(U, MGMz*V);

    meshVertex->push_back(glm::vec3(px,py,pz));

    glm::vec4 dU(3*uv.x*uv.x, 2*uv.x, 1, 0), dV(3*uv.y*uv.y, 2*uv.y, 1, 0);

    float ux = glm::dot(dU, MGMx*V);
    float uy = glm::dot(dU, MGMy*V);
    float uz = glm::dot(dU, MGMz*V);

    float vx = glm::dot(U, MGMx*dV);
    float vy = glm::dot(U, MGMy*dV);
    float vz = glm::dot(U, MGMz*dV);

    glm::vec3 du= glm::normalize(glm::vec3(ux, uy, uz));
    glm::vec3 dv = glm::normalize(glm::vec3(vx, vy, vz));

    glm::vec3 normal = glm::normalize(glm::cross(dv, du));


    meshNormals->push_back(normal);

  }

}
```

Código C.2: Triángulo

```
void ShipBodyModel::triangulateTriangle(const std::vector<glm::vec3>& curveA,
    const glm::vec3 & curveB, float subdivisions, float scale, std::vector<glm
    ::vec3>* meshVertex, std::vector<GLuint>* meshIndex, std::vector<glm::vec3
    >* meshNormals, std::vector<glm::vec2>* meshRS, bool flipTriangleOrder)
{

  const int vertexStart = meshVertex->size();
  const int indexStart = meshIndex->size();
  const int normalsStart = meshNormals->size();
  const int uvStart = meshRS->size();

  //build a equilateral triangle centered at the origin
  const glm::vec2 triangle[3] = { glm::vec2(0, 1), glm::vec2(-0.866025, -0.5),
      glm::vec2(0.866025, -0.5) };


  const int outerSub[3] = {
    glm::length(curveA[3] - curveA[0])*subdivisions + 2,
    glm::length(curveB - curveA[3])*subdivisions + 2,
    glm::length(curveB - curveA[0])*subdivisions + 2,
  };


  const int innerSub = (outerSub[0] + outerSub[1] + outerSub[2]) / 3;

  std::vector<glm::vec3> UV;


  std::vector<glm::vec2> outerRing, innerRing;

  //--------
  //First
  //Ring
  //--------
```

```

//A Segment
for (int i = 0; i < outerSub[0]; i++) {
  glm::vec3 baryPoint(glm::mix(glm::vec3(1.0f,0.0f,0.0f), glm::vec3(0.0f, 1.0
      f, 0.0f), i * 1.0f / outerSub[0]));
  glm::vec2 point(glm::mix(triangle[1], triangle[2], i * 1.0f / outerSub[0]))
      ;

  UV.push_back(baryPoint);
  outerRing.push_back(point);
}


//AB Segment
for (int i = 0; i < outerSub[1]; i++) {
  glm::vec3 baryPoint(glm::mix(glm::vec3(0.0f, 1.0f, 0.0f), glm::vec3(0.0f,
      0.0f, 1.0f), i * 1.0f / outerSub[1]));
  glm::vec2 point(glm::mix(triangle[2], triangle[0], i * 1.0f / outerSub[1]))
      ;

  UV.push_back(baryPoint);
  outerRing.push_back(point);
}


//BA Segment
//inverse
for (int i = 0; i < outerSub[2]; i++) {
  glm::vec3 baryPoint(glm::mix(glm::vec3(0.0f, 0.0f, 1.0f), glm::vec3(1.0f,
      0.0f, 0.0f), i * 1.0f / outerSub[2]));
  glm::vec2 point(glm::mix(triangle[0], triangle[1], i * 1.0f / outerSub[2]))
      ;

  UV.push_back(baryPoint);
  outerRing.push_back(point);
}


int AStart = vertexStart, BStart = AStart + outerRing.size();


//Base cases
if (outerSub[0]<2 && outerSub[1]<2 && outerSub[2]<2) {
  if (flipTriangleOrder) {
    meshIndex->push_back(AStart + 0);
    meshIndex->push_back(AStart + 2);
    meshIndex->push_back(AStart + 1);
  }
  else {
    meshIndex->push_back(AStart + 0);
    meshIndex->push_back(AStart + 1);
    meshIndex->push_back(AStart + 2);
  }
}
else if (innerSub < 3) {
  glm::vec3 baryPoint(1.0f / 3.0f, 1.0f / 3.0f, 1.0f / 3.0f);

  UV.push_back(baryPoint);

  for (int i = 0; i < outerRing.size(); i++) {
    if (flipTriangleOrder) {
      meshIndex->push_back(AStart + (1 + i) %outerRing.size());
      meshIndex->push_back(BStart);
      meshIndex->push_back(AStart + 0 + i);
```

```cpp

      }
      else {
        meshIndex->push_back(AStart + (1 + i) %outerRing.size());
        meshIndex->push_back(AStart + 0 + i);
        meshIndex->push_back(BStart);
      }
    }

  }
  else {
    const float scaleFactor = -(1.0f / innerSub - 0.5) * 2;
    const glm::vec2 reducedTriangle[3] = { triangle[0]*scaleFactor, triangle[1] * scaleFactor, triangle[2] * scaleFactor };
    const glm::vec3 reducedTriangleBarycenter[3] = {
      barycentric(reducedTriangle[0],triangle[1],triangle[2],triangle[0]),
      barycentric(reducedTriangle[1],triangle[1],triangle[2],triangle[0]),
      barycentric(reducedTriangle[2],triangle[1],triangle[2],triangle[0])
    };

    //A Segment
    for (int i = 0; i < innerSub - 2; i++) {

      glm::vec2 point(glm::mix(reducedTriangle[1], reducedTriangle[2], i * 1.0f / (innerSub - 2)));
      glm::vec3 baryPoint(barycentric(point, triangle[1], triangle[2], triangle[0]));



      UV.push_back(baryPoint);
      innerRing.push_back(point);
    }


    //AB Segment
    for (int i = 0; i < innerSub - 2; i++) {
      glm::vec2 point(glm::mix(reducedTriangle[2], reducedTriangle[0], i * 1.0f / (innerSub - 2)));
      glm::vec3 baryPoint(barycentric(point, triangle[1], triangle[2], triangle[0]));


      UV.push_back(baryPoint);
      innerRing.push_back(point);
    }


    //BA Segment
    //inverse
    for (int i = 0; i < innerSub - 2; i++) {
      glm::vec2 point(glm::mix(reducedTriangle[0], reducedTriangle[1], i * 1.0f / (innerSub-2)));
      glm::vec3 baryPoint(barycentric(point, triangle[1], triangle[2], triangle[0]));



      UV.push_back(baryPoint);
      innerRing.push_back(point);
    }


    int i = 0, j = 0;
```

```
    int segment = 0;
    int maxI = 0, maxJ = 0;
    while (segment < 3) {
      maxI += outerSub[segment];
      maxJ += innerSub - 2;
      while (i < maxI && j < maxJ) {
        glm::vec2 distItoJ = outerRing[i] - innerRing[(j + 1) % innerRing.size()];
        glm::vec2 distJtoI = outerRing[(i + 1) % outerRing.size()] - innerRing[j];
        float lengthIToJ = glm::length(distItoJ), lengthJToI = glm::length(distJtoI);
        if (glm::length(distItoJ) < glm::length(distJtoI)) {
          if (flipTriangleOrder) {
            meshIndex->push_back(BStart + j % innerRing.size());
            meshIndex->push_back(AStart + i % outerRing.size());
            meshIndex->push_back(BStart + (j + 1) % innerRing.size());

          }
          else {
            meshIndex->push_back(BStart + j % innerRing.size());
            meshIndex->push_back(BStart + (j + 1) % innerRing.size());
            meshIndex->push_back(AStart + i % outerRing.size());
          }
          j++;
        }
        else {
          if (flipTriangleOrder) {
            meshIndex->push_back(BStart + j % innerRing.size());
            meshIndex->push_back(AStart + i % outerRing.size());
            meshIndex->push_back(AStart + (i + 1) % outerRing.size());

          }
          else {
            meshIndex->push_back(BStart + j % innerRing.size());
            meshIndex->push_back(AStart + (i + 1) % outerRing.size());
            meshIndex->push_back(AStart + i % outerRing.size());
          }
          i++;
        }
      }
      while (i < maxI && i < outerRing.size()) {
        if (flipTriangleOrder) {
          meshIndex->push_back(BStart + j % innerRing.size());
          meshIndex->push_back(AStart + i % outerRing.size());
          meshIndex->push_back(AStart + (i + 1) % outerRing.size());

        }
        else {
          meshIndex->push_back(BStart + j % innerRing.size());
          meshIndex->push_back(AStart + (i + 1) % outerRing.size());
          meshIndex->push_back(AStart + i % outerRing.size());
        }


        i = i + 1;
      }
      while (j < maxJ && j < innerRing.size()) {
        if (flipTriangleOrder) {
          meshIndex->push_back(BStart + j % innerRing.size());
          meshIndex->push_back(AStart + i % outerRing.size());
          meshIndex->push_back(BStart + (j + 1) % innerRing.size());

        }
```

```
        else {
          meshIndex->push_back(BStart + j %innerRing.size());
          meshIndex->push_back(BStart + (j + 1) %innerRing.size());
          meshIndex->push_back(AStart + i %outerRing.size());
        }
        j = j + 1;
      }
      segment++;
    }

    //update starting point for indexing points
    AStart = BStart;
    BStart += innerRing.size();

    outerRing = innerRing;
    innerRing.clear();

    int rings = 1;
    while (innerSub - rings * 2 > 2) {
      rings++;

      const float scaleFactor = -( rings * (1.0f / innerSub) - 0.5) * 2;
      const glm::vec2 reducedTriangle[3] = { triangle[0] * scaleFactor, triangle[1] * scaleFactor, triangle[2] * scaleFactor };
      const glm::vec3 reducedTriangleBarycenter[3] = {
        barycentric(reducedTriangle[0],triangle[0],triangle[1],triangle[2]),
        barycentric(reducedTriangle[1],triangle[0],triangle[1],triangle[2]),
        barycentric(reducedTriangle[2],triangle[0],triangle[1],triangle[2])
      };

      //A Segment
      for (int i = 0; i < innerSub - 2*rings; i++) {

        glm::vec2 point(glm::mix(reducedTriangle[1], reducedTriangle[2], i * 1.0f / (innerSub - 2*rings)));
        glm::vec3 baryPoint(barycentric(point, triangle[1], triangle[2], triangle[0]));

        UV.push_back(baryPoint);
        innerRing.push_back(point);
      }

      //AB Segment
      for (int i = 0; i < innerSub - 2*rings; i++) {
        glm::vec2 point(glm::mix(reducedTriangle[2], reducedTriangle[0], i * 1.0f / (innerSub - 2*rings)));
        glm::vec3 baryPoint(barycentric(point, triangle[1], triangle[2], triangle[0]));

        UV.push_back(baryPoint);
        innerRing.push_back(point);
      }

      //BA Segment
```

```
    //inverse
    for (int i = 0; i < innerSub - 2*rings; i++) {

      glm::vec2 point(glm::mix(reducedTriangle[0], reducedTriangle[1], i * 1.0f / (innerSub - 2*rings)));
      glm::vec3 baryPoint(barycentric(point, triangle[1], triangle[2], triangle[0]));



      UV.push_back(baryPoint);
      innerRing.push_back(point);
    }




    int i = 0, j = 0;
    int segment = 0;
    int maxI = 0, maxJ = 0;
    while (segment < 3) {
      maxI += innerSub - 2 * (rings - 1);
      maxJ += innerSub - 2 * rings;
      while (i < maxI && j < maxJ) {
        glm::vec2 distItoJ = outerRing[i] - innerRing[(j + 1) % innerRing.size()];
        glm::vec2 distJtoI = outerRing[(i + 1) % outerRing.size()] - innerRing[j];
        float lengthIToJ = glm::length(distItoJ), lengthJToI = glm::length(distJtoI);
        if (glm::length(distItoJ) < glm::length(distJtoI)) {
          if (flipTriangleOrder) {
            meshIndex->push_back(BStart + j % innerRing.size());
            meshIndex->push_back(AStart + i % outerRing.size());
            meshIndex->push_back(BStart + (j + 1) % innerRing.size());

          }
          else {
            meshIndex->push_back(BStart + j % innerRing.size());
            meshIndex->push_back(BStart + (j + 1) % innerRing.size());
            meshIndex->push_back(AStart + i % outerRing.size());
          }
          j++;
        }
        else {
          if (flipTriangleOrder) {
            meshIndex->push_back(BStart + j % innerRing.size());
            meshIndex->push_back(AStart + i % outerRing.size());
            meshIndex->push_back(AStart + (i + 1) % outerRing.size());

          }
          else {
            meshIndex->push_back(BStart + j % innerRing.size());
            meshIndex->push_back(AStart + (i + 1) % outerRing.size());
            meshIndex->push_back(AStart + i % outerRing.size());
          }
          i++;
        }
      }
      while (i < maxI && i < outerRing.size()) {
        if (flipTriangleOrder) {
          meshIndex->push_back(BStart + j % innerRing.size());
          meshIndex->push_back(AStart + i % outerRing.size());
          meshIndex->push_back(AStart + (i + 1) % outerRing.size());
```

```
          }
          else {
            meshIndex->push_back(BStart + j % innerRing.size());
            meshIndex->push_back(AStart + (i + 1) % outerRing.size());
            meshIndex->push_back(AStart + i % outerRing.size());
          }


          i = i + 1;
        }
        while (j < maxJ && j < innerRing.size()) {
          if (flipTriangleOrder) {
            meshIndex->push_back(BStart + j % innerRing.size());
            meshIndex->push_back(AStart + i % outerRing.size());
            meshIndex->push_back(BStart + (j + 1) % innerRing.size());

          }
          else {
            meshIndex->push_back(BStart + j % innerRing.size());
            meshIndex->push_back(BStart + (j + 1) % innerRing.size());
            meshIndex->push_back(AStart + i % outerRing.size());
          }
          j = j + 1;
        }
        segment++;
      }


      AStart = BStart;
      BStart += innerRing.size();


      outerRing = innerRing;
      innerRing.clear();
    }





    if (innerSub - rings*2 == 1) {
      if (flipTriangleOrder) {
        meshIndex->push_back(AStart + 0);
        meshIndex->push_back(AStart + 1);
        meshIndex->push_back(AStart + 2);

      }
      else {
        meshIndex->push_back(AStart + 0);
        meshIndex->push_back(AStart + 2);
        meshIndex->push_back(AStart + 1);
      }
    }
    else if (innerSub - rings * 2 == 2) {

      glm::vec3 baryPoint(1.0f/3.0f, 1.0f / 3.0f, 1.0f / 3.0f);

      UV.push_back(baryPoint);

      for (int i = 0; i < outerRing.size(); i++) {
        if (flipTriangleOrder) {
          meshIndex->push_back(AStart + (1 + i) % outerRing.size());
          meshIndex->push_back(BStart);
```

```
          meshIndex->push_back(AStart + 0 + i);

        }
        else {
          meshIndex->push_back(AStart + (1 + i) % outerRing.size());
          meshIndex->push_back(AStart + 0 + i);
          meshIndex->push_back(BStart);
        }
      }
    }


  }




  glm::vec3 B003,B102,B201,B300,B012,B111,B210,B021,B120,B030;

  B300 = curveA[0];
  B030 = curveA[3];
  B003 = curveB;
  if (glm::all(glm::equal(curveA[0], curveA[1])) && glm::all(glm::equal(curveA
      [2], curveA[3]))) {
    glm::vec3 a03(glm::mix(curveA[0], curveA[3], 0.3f)), a06(glm::mix(curveA
        [0], curveA[3], 0.6f));

    B210 = a03;
    B120 = a06;

  }
  else {
    B210 = curveA[1];
    B120 = curveA[2];
  }

  B201 = glm::mix(curveA[0], curveB, 0.3f);
  B102 = glm::mix(curveA[0], curveB, 0.6f);

  B021 = glm::mix(curveA[3], curveB, 0.3f);
  B012 = glm::mix(curveA[3], curveB, 0.6f);

  B111 = glm::mix(glm::mix(B210, B012, 0.5f),glm::mix(B120,B102,0.5f),0.5f);

  for (int i = 0; i < UV.size(); i++) {

    float u = UV[i].x;
    float v = UV[i].y;
    float w = UV[i].z;

    float uu = u*u, uuu = uu*u;
    float vv = v*v, vvv = vv*v;
    float ww = w*w, www = ww*w;

    glm::vec3 point = B030*vvv + 3.0f*B120*u*vv + 3.0f*B021*vv*w +
    3.0f*B210*uu*v + 6.0f*B111*u*v*w + 3.0f *B012*v*ww +
    B300*uuu + 3.0f * B201 * uu*w + 3.0f*B102 * u*ww + B003*www;

    meshVertex->push_back(point);


    glm::vec3 du = B210*uu + B030*vv + B012*ww +
    2.0f*B120*u*v + 2.0f*B111*u*w + 2.0f*B021*w*v -
    (B300*uu + B120*vv + B102*ww +
```

```
    2.0f*B210*u*v + 2.0f*B201*w*u + 2.0f*B111*w*v);
    glm::vec3 dv = B201*uu + B021*vv + B003*ww +
    2.0f*B111*v*u + 2.0f*B102*w*u + 2.0f*B012*w*v -
    (B300*uu + B120*vv + B102*ww +
    2.0f*B210*u*v + 2.0f*B201*w*u + 2.0f*B111*w*v);

    du = glm::normalize(du);
    dv = glm::normalize(dv);

    if(flipTriangleOrder)
    meshNormals->push_back(glm::normalize(glm::cross(dv, du)));
    else
    meshNormals->push_back(glm::normalize(glm::cross(du, dv)));
  }

}
```

APÉNDICE D

Subdivisión de una cinta en parches

```
std::vector<glm::vec3> A;
std::vector<glm::vec3> B;

std::vector<glm::vec3> meshVertex;
std::vector<GLuint> meshInidices;
std::vector<glm::vec3> meshNormals;
std::vector<glm::vec2> meshRS;


A = sections[sectionStart].getControlPoints();
B = sections[sectionStart + 1].getControlPoints();

float angleA = sections[sectionStart].getRotation();
glm::mat4 rotA = glm::rotate(glm::mat4(), angleA, glm::vec3(0,0,1));
for (int i = 0; i < A.size(); i++) {
  A[i] = glm::vec3(rotA*glm::vec4(A[i],1.0f));
}

float angleB = sections[sectionStart+1].getRotation();
glm::mat4 rotB = glm::rotate(glm::mat4(), angleB, glm::vec3(0, 0, 1));
for (int i = 0; i < B.size(); i++) {
  B[i] = glm::vec3(rotB*glm::vec4(B[i], 1.0f));
  B[i].z = 1;
}


std::vector<glm::vec3> AB;

for (int i = 0; i < A.size(); i++) {
  AB.push_back(A[i]);
}

for (int i = 0; i < B.size(); i++) {
  AB.push_back(B[i]);
}

std::vector<GLuint> indices;

if (A.size()==0) {



  auto pcb = sections[sectionStart + 1].getTransformedControlPoints();
  auto pb = pcb.begin();
  for (int i = 0; i+3 < pcb.size(); i += 3) {
```

```
    int jy = i ;

    std::vector<glm::vec3> curveB(pb + jy , pb + jy + 4);
    auto pointA = sections[sectionStart].getCurve();

    triangulateTriangle(curveB, pointA[0], 10.0f, 0.0f, &meshVertex, &
        meshInidices, &meshNormals, &meshRS);
  }
  for (int i = 0; i+1 < B.size();i++) {
    indices.push_back(i + 1);
    indices.push_back(0);
    indices.push_back(i+2);
  }
}
else if (B.size()==0) {

  auto pca = sections[sectionStart].getTransformedControlPoints();
  auto pa = pca.begin();
  for (int i = 0; i+3 < pca.size(); i += 3) {
    int ix = i ;



    std::vector<glm::vec3> curveA(pa + ix , pa + ix + 4);
    auto pointB = sections[sectionStart + 1].getCurve();


    triangulateTriangle(curveA, pointB[0], 10.0f, 0.0f, &meshVertex, &
        meshInidices, &meshNormals, &meshRS, true);
  }

  for (int i = 0; i+1 < A.size(); i++) {
    indices.push_back(i);
    indices.push_back(i + 1);
    indices.push_back(A.size());
  }
}
else {


  float minDist = 1.0e20;
  int minI = -1, minJ = -1;
  for (int i = 0; i < A.size(); i += 3) {
    for (int j = 0; j < B.size(); j += 3) {
      glm::vec2 dist = A[i] - B[j];
      float distf = glm::length(dist);
      if (distf < minDist) {
        minDist = distf;
        minI = i;
        minJ = j;
      }

    }

  }

  int despA = 0, despB = A.size();

  int i = 0, j = 0;
  i = minI, j = minJ;
  int nextI = (minI + 3) %(A.size()-1), nextJ = (minJ + 3) %(B.size()-1);

  int numTriangles = A.size()/3+B.size()/3;
```

```
while (numTriangles > 0)
{

  glm::vec3 AB = A[nextI] - A[i],
  AC = B[j] - A[i],
  AD = B[nextJ] - A[i];
  //check if the quad isn't coplanar

  glm::vec2 distItoJ = A[i] - B[nextJ];
  glm::vec2 distJtoI = A[nextI] - B[j];

  float lengthItoJ = glm::length(distItoJ), lengthJtoI = glm::length(distJtoI);

  if (glm::epsilonNotEqual(glm::dot(AD,glm::cross(AB,AC)),0.0f,0.0001f) || (lengthItoJ>2*lengthJtoI || lengthJtoI>2 * lengthItoJ )) {


    if (lengthItoJ > lengthJtoI) {

      auto pca = sections[sectionStart].getTransformedControlPoints();
      auto pa = pca.begin();
      int ix = i ;

      auto pcb = sections[sectionStart + 1].getTransformedControlPoints();
      auto pb = pcb.begin();
      int jy = j ;

      std::vector<glm::vec3> curveA(pa + ix, pa + ix + 4);


      triangulateTriangle(curveA, *(pb + jy), 10.0f, 0.0f, &meshVertex, &meshInidices, &meshNormals, &meshRS,true);



      i = nextI;
      nextI = (i + 3) %(A.size()-1);
      numTriangles -= 1;
    }
    else {



      auto pca = sections[sectionStart].getTransformedControlPoints();
      auto pa = pca.begin();
      int ix =  i ;

      auto pcb = sections[sectionStart + 1].getTransformedControlPoints();
      auto pb = pcb.begin();
      int jy = j;

      std::vector<glm::vec3> curveB(pb + jy, pb + jy + 4);

      triangulateTriangle(curveB, *(pa + ix), 10.0f, 0.0f, &meshVertex, &meshInidices, &meshNormals, &meshRS);


      j = nextJ;
      nextJ = (j + 3) %(B.size()-1);
      numTriangles -= 1;
    }

```

```
    }
    else {
      //makeCurveIndex(A, B, i, j, ComponentSection::numPointsPerCurve, ComponentSection::numPointsPerCurve, &indices);

      auto pca = sections[sectionStart].getTransformedControlPoints();
      auto pa = pca.begin();
      int ix = i ;

      auto pcb = sections[sectionStart + 1].getTransformedControlPoints();
      auto pb = pcb.begin();
      int jy = j;

      std::vector<glm::vec3> curveA(pa + ix, pa + ix + 4), curveB(pb + jy, pb + jy + 4);

      triangulateQuadrilateral(curveA, curveB, 10.0f, 0.0f, &meshVertex, &meshInidices, &meshNormals, &meshRS);

      j = nextJ;
      nextJ = (j + 3) %(B.size()-1);

      i = nextI;
      nextI = (i + 3) %(A.size()-1);

      numTriangles -= 2;

    }

  }

}
```