Universidad Nacional de Córdoba

Facultad de Matemática, Astronomía, Física y Computación

Extracción de características geométricas para rendering no fotorrealista de mallas 3D en Python

Tesis realizada por Daniel Bauer Santana
para la Licenciatura en Ciencias de la Computación

Dirigida por:
Doctor Emmanuel Iarussi
Doctor Nicolás Wolovick

2021

Resumen

Español

La pregunta de cómo las personas interpretamos y creamos dibujos y bocetos ha sido discutida por décadas. Si bien el dibujo no es un fenómeno que ocurre naturalmente, los humanos tenemos la extraordinaria capacidad de crearlos y distinguir objetos y formas en estos diseños. Existen numerosos algoritmos para crear bocetos de forma automática o semiautomática en una computadora, varios de los cuales se basan en modelos tridimensionales. Hasta ahora, estos dibujos creados por computadoras casi siempre resultan distinguibles de aquellos creados por humanos, y dependen fuertemente de parámetros que deben ser configurados a mano. Muchos de estos algoritmos necesitan procesar una gran cantidad de vértices y caras de triángulos, los cuales se procesan independientemente. Por este motivo, generalmente se utilizan procesadores gráficos (GPUs) para ejecutarlos, ya que estos trabajos independientes son altamente paralelizables. Dentro de los motores de renderizado existentes para sintetizar bocetos a partir de modelos tridimensionales poligonales, *rtsc*[1]es uno de los más utilizados, ya que contiene implementaciones eficientes de numerosas variantes de estos algoritmos.

Por otro lado, el avance reciente sobre el aprendizaje automático, especialmente en lo que concierne a las redes neuronales convolucionales profundas, representa una oportunidad para mejorar la calidad de los dibujos de líneas hechos por computadora, y también la de los algoritmos que trabajan sobre estos dibujos. Recientemente, varios trabajos han explorado las capacidades de las redes neuronales en el ámbito de los dibujos de líneas para su generación a partir de fotografías o de modelos 3D. En particular, estos últimos toman como punto de partida los algoritmos implementados en *rtsc* y utilizan grandes cantidades de dibujos realizados por artistas para aprender la configuración de parámetros óptima que permite generar bocetos a partir de un modelo 3D con un determinado estilo.

[1]https://gfx.cs.princeton.edu/proj/sugcon/

Sin embargo, debido a la tecnología sobre la que está implementado, la funcionalidad de *rtsc* no puede ser fácilmente incorporada dentro de los nuevos pipelines de rendering. En general, los autores de estos trabajos se han visto forzados a preprocesar los modelos y generar dibujos con *rtsc* de manera offline, para luego utilizarlos en sus pipelines de aprendizaje profundo. En esta tesis nos centramos en una reimplementación de los algoritmos que componen el paquete *rtsc*, dentro de las tecnologías y frameworks disponibles para realizar aprendizaje automático. El principal desafío consiste en implementar numerosos algoritmos de procesamiento geométrico dentro de entornos basados en librerías de aprendizaje profundo y diferenciación automática. Contar con estas implementaciones permitirá, principalmente, el desarrollo de nuevas arquitecturas de red neuronal capaces de generar dibujos directamente a partir de modelos tridimensionales, sin recurrir a pasos intermedios de preprocesado. Además, las funciones de cálculo geométrico implementadas podrán ser utilizadas en otros contextos relacionados con el procesamiento de mallas 3D.

English

The question of how humans interpret and create drawings and sketches has been discussed for decades. Even though drawings are not a naturally occurring phenomenon, humans have the extraordinary capability to create them and to distinguish objects and shapes in those designs. There exist numerous algorithms for automatically or semi-automatically creating sketches using a computer, several of which are based on three-dimensional models. So far, these drawings generated using computers are nearly always distinguishable from those created by humans, and depend strongly on parameters which must be configured manually. Many of these algorithms need to process a large amount of vertices and triangle faces, and these are processed independently from one another. For this reason, graphics processing units (GPUs) are generally used to execute them, since this independence means they are very apt for parallel processing. Of the existing rendering engines for synthesizing sketches from three-dimensional polygonal models, *rtsc*[1] is perhaps the most used, since it contains efficient implementations of numerous versions of these algorithms.

On the other hand, recent advances in the field of machine learning, especially in what concerns deep convolutional neural networks (DCNNs), represent an opportunity to improve the quality of computer-generated line drawings, and also to improve the quality of the algorithms that take these drawings as inputs. Recently, several investigations have

explored the capabilities of neural networks to generate line drawings using pictures or 3D models. In particular, those CNNs that use 3D models take as a starting point the algorithms implemented in *rtsc* and utilize large quantities of drawings produced by artists to learn the optimal configuration of parameters which allows the creation of sketches with a particular style from a 3D model.

However, due to the technology with which it is implemented, the functionality in *rtsc* is not easily incorporated into new rendering pipelines. In general, the authors of these works have been forced to pre-process models and generate drawings with *rtsc* offline, to then use them in their deep learning pipelines. In this thesis we focus on a re-implementation of the algorithms that make up the *rtsc* library, using the technology and frameworks available for machine learning. The main challenge consists in implementing numerous geometry-processing algorithms within environments based on deep learning and automatic differentiation. Having these implementations will allow for the development of new neural network architectures capable of generating drawings directly from three-dimensional models, without having to resort to intermediate pre-processing steps. Additionally, the functions for geometric calculations could be utilized in other contexts related to the processing of 3D meshes.

Agradecimientos

- A mi familia y a Mechi por el constante amor y apoyo, y por impulsarme a superar todos los retos a los que me enfrenté.
- A Emmanuel por marcarme el camino y por toda la ayuda y trabajo que me brindó en el transcurso de este proyecto, y a Nicolás por alentarme a completar esta tesis y hacer que sea posible elaborarla.
- A mis amigos por acompañarme siempre que lo he necesitado.

Índice general

Capítulo 1

Introducción

El presente trabajo se enfoca en el área de síntesis de imágenes no fotorealistas de bocetos. A continuación describimos la motivación y los principales objetivos que se desarrollarán en esta tesis.

1.1. Motivación

Los dibujos de líneas se utilizan comúnmente en una amplia gama de contextos, desde el diseño de productos y de interiores a tutoriales de diversos campos. Los dibujos son una potente herramienta para el aprendizaje y la organización, y son a menudo el punto de inicio para nuevas ideas. Frecuentemente son usados para representar la realidad visualmente. La generación automática de estos dibujos ha avanzado a la par del rendering fotorealista y actualmente tenemos acceso a algoritmos que nos permiten crear bocetos razonables a partir de geometría tridimensional, como los bocetos en la Figura 1.1. A través de estos dibujos podemos obtener un entendimiento del arte y del diseño. También podemos crear herramientas que nos permitan tomar ventaja de la expresividad de los dibujos. Por ejemplo, aplicaciones que facilitan el diseño de productos, o que nos permitan hacer búsquedas de productos en base a dibujos hechos por el usuario (en SketchZooms [17] se propone una aplicación de este estilo).

Recientemente han surgido numerosos trabajos de síntesis de imágenes de dibujos de líneas que utilizan aprendizaje automático, específicamente redes neuronales convolucionales (CNN). Generalmente, estos trabajos utilizan modelos que requieren grandes cantidades de de dibujos de líneas para ser entrenados. Como compilar y curar grandes bases de datos de este tipo de imágenes requiere un gran esfuerzo, generalmente se utilizan técnicas de

(a) Modelo original usando Lambertian shading.

(b) Boceto creado utilizando Occluding Contours y Apparent Ridges.

(c) Boceto creado utilizando Occluding Contours y Suggestive Contours.

Figura 1.1: Modelo "Lucy" y dibujos de línea creados a partir del modelo utilizando *rtsc*. Modelo creado por el Stanford Computer Graphics Laboratory

réndering que permiten simular dibujos a partir de modelos 3D. Un método comúnmente usado actualmente se basa en generar estas imágenes mediante *rtsc* [19]. Sin embargo, la tecnología con la que este software está implementado hace que se deban renderizar todos los modelos a priori, para luego utilizar las imágenes producidas en el entrenamiento de CNNs. En otras palabras, *rtsc* no se presta para ser fácilmente integrado en el flujo de trabajo de un modelo de aprendizaje automático. Resultaría deseable contar con una herramienta que genere estas imágenes que esté escrita dentro de los frameworks que se utilizan para el aprendizaje automático y que sea fácil de usar.

Por ejemplo, en Neural Contours [11] se utilizó *rtsc* para el desarrollo de una red neuronal que sintetiza bocetos automáticamente. Los autores de este trabajo utilizaron *rtsc* para obtener una primer versión de los dibujos de línea y los mapas de propiedades geométricas de cada objeto 3D, para luego entrenar un modelo de red neuronal que mejora el realismo de estos renders utilizando imágenes dibujadas por humanos. Sin embargo, la integración de estos dos componentes resulta muy complicada, dado que las tecnologías

sobre las cuales se encuentran implementadas son muy diferentes. En esta tesis nos proponemos realizar una reimplementación de buena parte de los algoritmos incorporados en *rtsc* en un entorno más apropiado para su utilización en contextos que involucren redes neuronales convolucionales. Llamaremos a esta implementación *pyrtsc*.

1.2. Objetivo

El objetivo general del presente trabajo es comprender y reimplementar los algoritmos de dibujo de línea existentes dentro del framework *rtsc*. Dada la poca documentación existente, esta taréa requiere de un fuerte trabajo de análisis y reingeniería de los módulos que componen *rtsc*. Específicamente, se trabajará sobre una reimplementación en PyTorch [18] y Kaolin [16], lo que permitirá la reutilización de algoritmos como *Suggestive Contours* [5] y *Apparent Ridges* [9] dentro de pipelines diferenciables de rendering u otras tareas asociadas a redes neuronales.

1.3. Estructura de la Tesis

- **Marco Teórico.** Breve introducción a la computación gráfica, el rendering no fotorealista y la geometría diferencial. Generación de dibujos de manera sintética.
- **Suggestive contours en rtsc.** Una descripción del algoritmo de *suggestive contours* tal y como está implementado en en C++ en el software original.
- **Implementación de pyrtsc.** Descripción de nuestra implementación de algoritmos de *rtsc* utilizando Python + PyTorch.
- **Resultados.** Evaluación del desempeño de los algoritmos y comparación con *rtsc*.
- **Conclusiones y direcciones futuras.**

1.4. Vinculación con los contenidos de la Carrera

A lo largo de esta tesis fueron esenciales los temas de varias de las materias cursadas a lo largo de la carrera. En primer lugar, gran parte de este trabajo se basa en utilizar vectores y matrices, por lo que los conocimientos adquiridos en la cursada de *Álgebra* resultaron esenciales. Además, empleamos algoritmos de minimización de cuadrados, cuya base

teórica fundamental se apoya en los contenidos vistos en *Análisis Numérico*. Adicionalmente, resultaron útiles los conocimientos de las materias *Aprendizaje Automático*, *Visión por Computadora* y *Computación Paralela* al momento de utilizar el framework PyTorch para implementar, medir y graficar los tiempos de cómputo de distintas versiones de los algoritmos de procesamiento geométrico.

Capítulo 2

Marco Teórico

2.1. Rendering y representación de modelos 3D

La computación gráfica es el área de la computación encargada de generar imágenes para mostrar en una pantalla. Interactuamos diariamente con la computación gráfica en la vida cotidiana al utilizar las interfaces de las aplicaciones, al ver una película con efectos especiales, al editar fotos o vídeo, y fundamentalmente al jugar videojuegos.

Un concepto importante en la computación gráfica es el de *rendering*. La renderización es el proceso de generar imágenes a partir de un modelo (generalmente en dos o tres dimensiones) usando una computadora. Una de las estructuras de datos más comúnmente utilizadas para representar objetos en 3D son las denominadas *mallas poligonales*. En estas estructuras, los objetos se definen en términos de los polígonos que lo componen (por lo general triángulos). La representación de mallas en memoria más común es la de *mallas triangulares*. En esta representación se almacenan en un arreglo las posiciones 3D de todos los vértices y en otro arreglo los índices de los vértices que componen cada triángulo. Podemos observar un ejemplo en la Figura 2.1.

La suposición de que los modelos están discretizados utilizando triángulos (como en la figura 2.2) es esencial para poder trabajar con los algoritmos que describiremos a continuación. En la Sección 2.2 veremos ciertas propiedades geométricas que se utilizan en el dibujo de líneas. Los algoritmos deben encontrar una forma de estimar las propiedades, discretizando conceptos que en teoría trabajan sobre superficies abstractas y continuas.

Shared vertices:

Triangles	
#	*Vertices*
0	(0, 1, 2)
1	(1, 3, 2)
2	(0, 3, 1)

Vertices	
#	*Position*
0	(a_x, a_y, a_z)
1	(b_x, b_y, b_z)
2	(c_x, c_y, c_z)
3	(d_x, d_y, d_z)

Figura 2.1: Un ejemplo de la representación en memoria de una malla triangular. Imagen de *Fundamentals of Computer Graphics, Fourth Edition* [12]. En el capítulo 12 del libro se pueden encontrar más detalles de este tipo de representación.

Aunque en la mayoría de los casos los trabajos de computación gráfica se centran en crear imágenes similares a lo capturado por una cámara fotográfica, existe también un interés en generar imágenes que no parezcan reales. A diferencia del *rendering fotorealísta*, el *rendering no fotorealísta* (NPR por las siglas en inglés de Non-Photorealistic Rendering) no se propone crear imágenes indistinguibles de fotografías. Dentro del NPR existen diversos objetivos, desde crear imágenes caricaturescas a generar imágenes sintetizadas de resonancias magnéticas. En las Figuras 2.3a y 2.3b vemos ejemplos de imágenes creadas con diferentes técnicas de NPR. En este trabajo nos concentramos en la síntesis de imágenes no foto-realistas de bocetos o dibujos de líneas. Estos algoritmos se centran en generar a partir de modelos tridimensionales un boceto que se aproxime a lo que generaría un artista a mano. Un ejemplo de este tipo de algoritmo de rendering puede encontrarse en la Figura 2.3c.

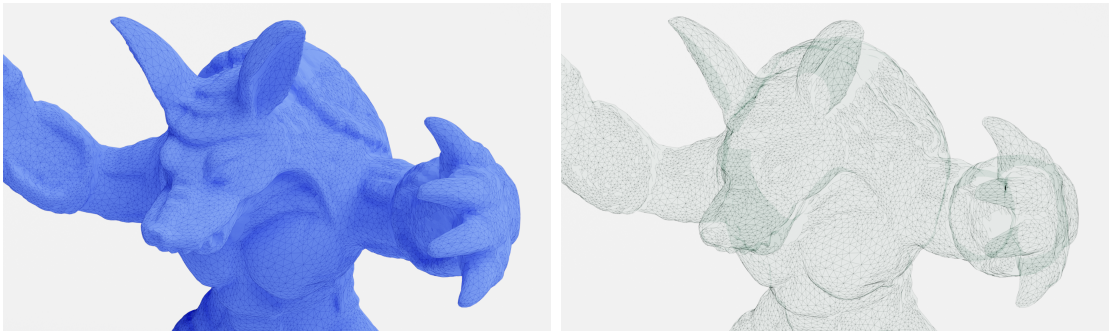
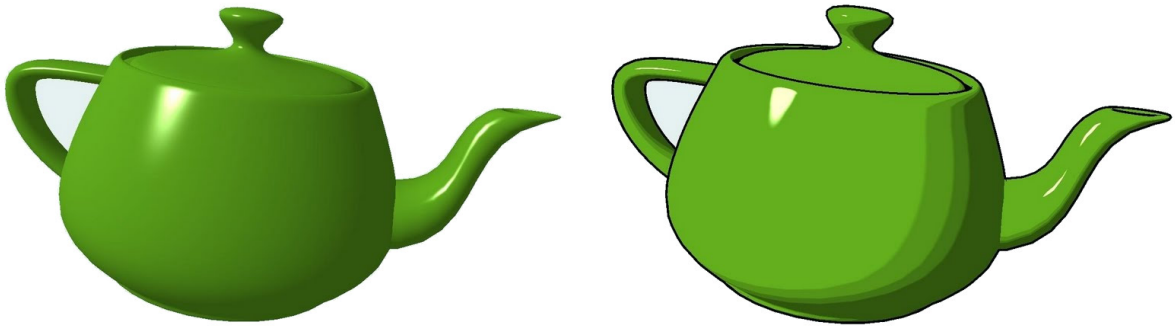
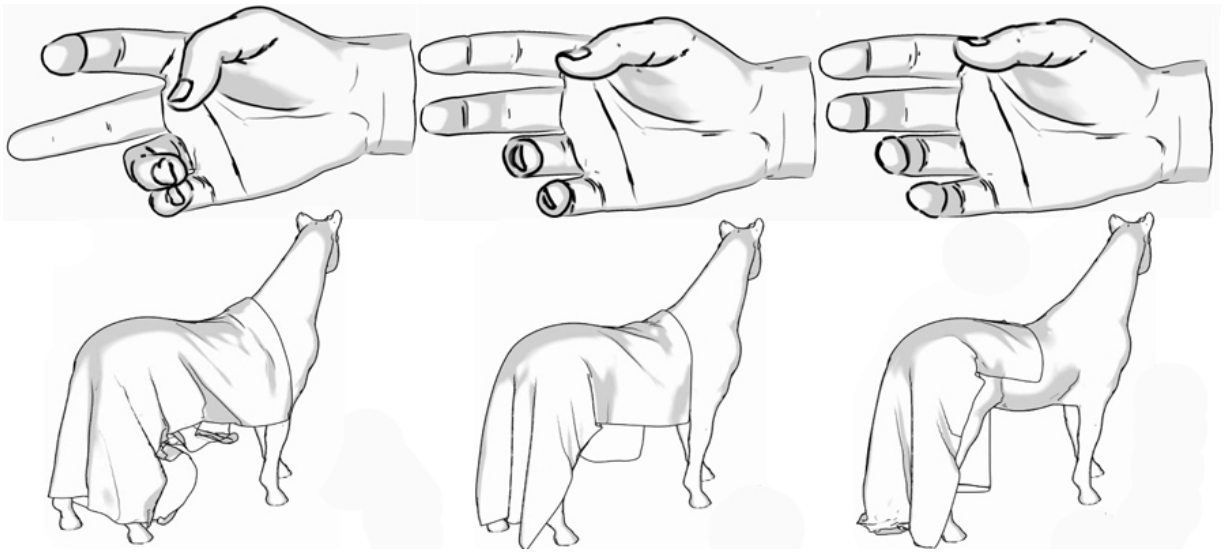


Figura 2.2: Un armadillo modelado con una malla de triángulos.

(a) Un ejemplo de NPR utilizando una caricatura de puercoespín. De izquierda a derecha: la malla original, el modelo renderizado de forma realista, renderizado simulando una pintura de acuarela y renderizado simulando un dibujo de artista. Imagen de "Art-directed watercolor stylization of 3D animations in real-time" [15]



(b) Ejemplo de NPR. A la izquierda el modelo es dibujado de forma realista, y a la derecha es dibujado de forma estilizada.



(c) Bocetos automáticos generados con el trabajo de Kalogerakis et al. [10]

Figura 2.3: Ejemplos de NPR

2.2. Definiciones de geometría diferencial

La geometría diferencial es una rama de la matemática que se encarga del estudio de las propiedades geométricas de variedades (una generalización de curvas y superficies a n dimensiones). Distintas propiedades de la geometría diferencial son usadas intensivamente en los motores de rendering para generar imágenes bidimensionales a partir de representaciones 3D. En particular, el tipo de renderer que creamos a lo largo de este trabajo se basa principalmente en nociones de geometría diferencial. Nos limitaremos a expresar los conceptos en términos de curvas o superficies. Una de las preguntas fundamentales que se busca responder utilizando geometría diferencial es: ¿cómo medimos la curvatura de una superficie? Aunque el concepto de curva no es el único que se maneja en la disciplina, es el concepto en el cual se basan gran parte de los algoritmos de dibujo de líneas. Utilizaremos métodos algebraicos y del cálculo diferencial para definir y estudiar la curvatura de una superficie, y en el capítulo 3 veremos cómo se utiliza la curvatura para crear bocetos automáticamente. La notación y las definiciones que veremos en esta sección provienen principalmente del trabajo de Keenan Crane [3]; si el lector está interesado en profundizar en los conceptos presentados, recomendamos ese trabajo como punto de partida.

Primero, consideremos una superficie en $\mathbb{R}^3$. Podemos describir la forma de esta superficie utilizando un mapa $f : M \mapsto \mathbb{R}^3$, donde M es una región de $\mathbb{R}^2$. $f(M)$ será entonces nuestra superficie. Usaremos el *diferencial* de f, denotado df; este nos dirá cómo pasar de un vector $\mathbf{X}$ en M a el vector correspondiente en la superficie $\mathbf{df}(\mathbf{X})$. En la Figura 2.4 vemos la idea del mapa f.

El vector $\mathbf{df}(\mathbf{X})$ es tangente a la superficie. Otro vector importante es el vector $\mathbf{N}$, el cual llamaremos la normal unitaria (o simplemente la normal). Un vector $\mathbf{n}$ es normal a una superficie en un punto $\mathbf{p}$ si:

$$\mathbf{df}(\mathbf{X}) \cdot \mathbf{n} = 0 \tag{2.1}$$

para todo vector tangente $\mathbf{df}(\mathbf{X})$ en el punto $\mathbf{p}$. El vector $\mathbf{N}(\mathbf{p})$ será un vector normal a la superficie con longitud 1. Generalmente omitiremos a $\mathbf{p}$, y hablaremos solamente de $\mathbf{N}$. En cualquier punto de la superficie tendremos dos normales unitarias, $+N$ y $-N$. Generalmente elegimos utilizar la normal que apunta hacia "afuera" de la superficie (si es una superficie cerrada). Podemos pensar en $\mathbf{N}$ como un mapa que asocia cada punto de M con un vector en la esfera unitaria (en cuyo caso $\mathbf{p}$ será un punto en M).

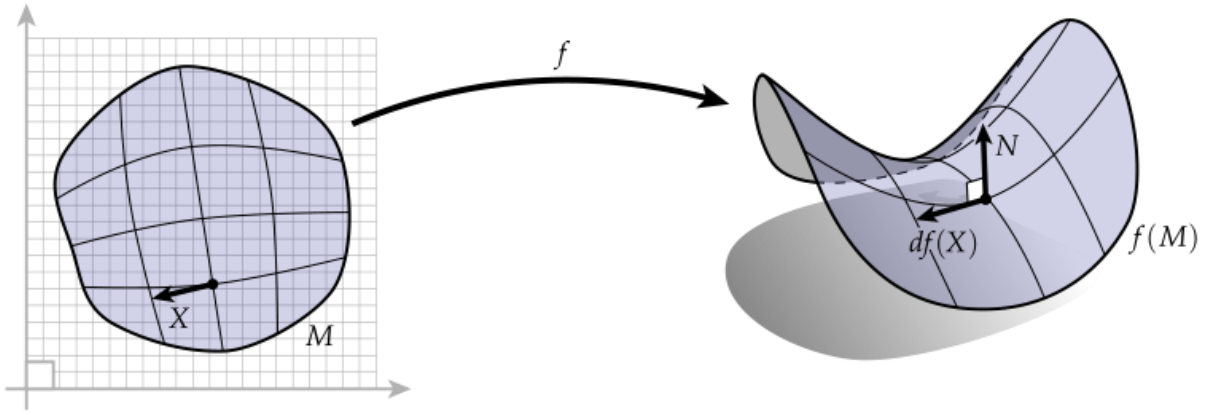


Figura 2.4: Un mapa f que describe la geometría de una superficie. Vemos también un vector $\mathbf{X}$ en M y su vector correspondiente $\mathbf{df}(\mathbf{X})$ tangente a la superficie, junto con la normal $\mathbf{N}$ en ese punto. Imágen de *Discrete differential geometry* [3].

Otro concepto que resulta útil es el del diferencial $\mathbf{dN}$, llamado el *Weingarten map*. Este diferencial representa el cambio en la normal mientras nos movemos de un punto a otro. Por ejemplo, podemos evaluar $\mathbf{dN}(\mathbf{X})$ (donde $\mathbf{X}$ es un vector en M): este representa el cambio en $\mathbf{N}$ en la dirección de $\mathbf{X}$. Veremos su utilidad más adelante, cuando hablemos de la curvatura de una superficie.

A continuación, empezaremos a definir y trabajar con curvatura. Primero, definiremos curvatura sobre una curva. Pensaremos las curvas como otro mapa $\gamma : I \mapsto \mathbb{R}^3$, donde I es un intervalo de $\mathbb{R}$. Podemos ver este mapa en la Figura 2.5. Restringiremos los mapas con los cuales trabajaremos: γ debe cumplir

$$|d\gamma(X)| = |X| \tag{2.2}$$

Estas parametrizaciones se suelen llaman *isométricas* o de *velocidad unitaria*. La idea principal que debemos tener en mente es que los vectores de I no se estiran cuando pasamos a $\gamma(I)$. En el caso discreto (con el cual trabajamos en computación gráfica), no tenemos noción de un dominio I; asumimos que las parametrizaciones sobre las cuales trabajamos son isométricas. Cuando trabajamos con superficies no podemos suponer que la parametrización es isométrica, pero si suponemos que los factores por los cuales se estiran los vectores son positivos.

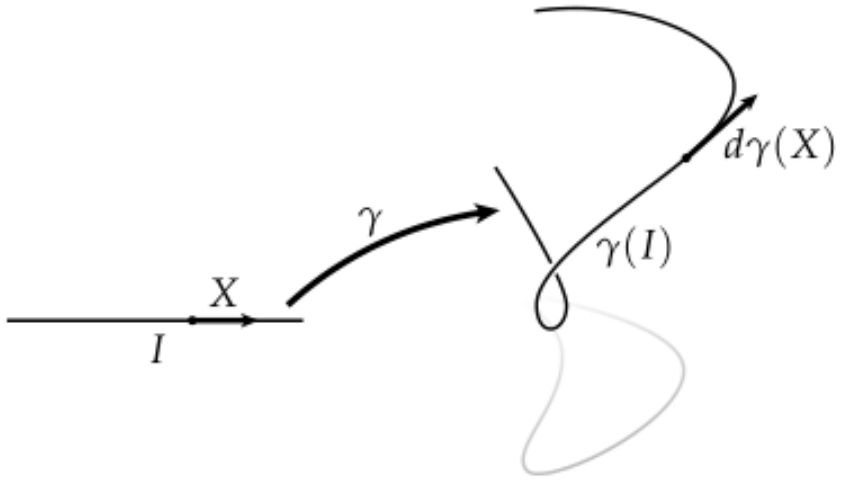


Figura 2.5: Un mapa γ que describe la geometría de una curva. Al igual que en la imágen 2.4, vemos cómo se transforma un vector $\mathbf{X}$ en I a un vector correspondiente $\mathbf{d}\gamma(\mathbf{X})$ tangente a $\gamma(I)$. Imágen de Keenan Crane [3].

Ahora, suponiendo que γ es isométrico, dado un vector unitario y positivo $\mathbf{X}$ en I, $\mathbf{T} = \mathbf{d}\gamma(\mathbf{X})$ es un vector en $\mathbb{R}^3$ tangente a la curva $\gamma(I)$. Podemos estudiar el cambio de $\mathbf{T}$ cuando nos movemos en $\gamma(I)$. Expresaremos este cambio usando dos componentes: la dirección de cambio y la magnitud del cambio. A la dirección de cambio la podemos estudiar usando un vector unitario $\mathbf{N}$, el cual llamaremos la *normal principal*. A la magnitud la expresaremos utilizando $\kappa \in \mathbb{R}$ y la llamaremos la curvatura.

$$\mathbf{dT}(\mathbf{X}) = -\kappa\mathbf{N} \tag{2.3}$$

$\mathbf{T}$ y $\mathbf{N}$ siempre serán ortogonales, ya que si no lo fuesen $\mathbf{T}$ se estiraría y la curva no sería isométrica. $\mathbf{T}$, $\mathbf{N}$ y el vector $\mathbf{B} = \mathbf{T} \times \mathbf{N}$ (llamado la *binormal* en el caso de las curvas y la *bitangente* en el caso de las superficies) juntos definen un marco de coordenadas llamado el marco de *Frenet*. Este marco cambia mientras nos movemos en la curva. La respuesta a cómo cambia está dada por la fórmula de *Frenet-Serret*:

$$\begin{bmatrix} \mathbf{T}' \\ \mathbf{N}' \\ \mathbf{B}' \end{bmatrix} = \begin{bmatrix} 0 & -\kappa & 0 \\ \kappa & 0 & -\tau \\ 0 & \tau & 0 \end{bmatrix} \begin{bmatrix} \mathbf{T} \\ \mathbf{N} \\ \mathbf{B} \end{bmatrix} \tag{2.4}$$

donde $\mathbf{T}'$, $\mathbf{N}'$ y $\mathbf{B}'$ representan el cambio de $\mathbf{T}$, $\mathbf{N}$ y $\mathbf{B}$ cuando nos movemos en la

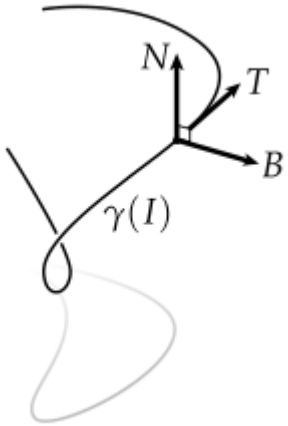


Figura 2.6: El marco de *Frenet* en un punto de una curva. Imagen extraída del trabajo de Crane [3].

curva a velocidad unitaria. τ se llama la *torsión*, y describe cómo $\mathbf{N}$ y $\mathbf{B}$ giran mientras avanzamos en la curva. En la Figura 2.6 vemos un ejemplo ilustrativo del marco de *Frenet*.

Pasemos ahora a estudiar la geometría de superficies. A diferencia de las curvas, podemos movernos en cualquier dirección sobre una superficie. Por ende, para estudiar la curvatura en un punto debemos definir una dirección de curvatura. Supongamos que tenemos una superficie L con un mapa asociado f. Tomemos un vector unitario $\mathbf{df}(\mathbf{X})$ tangente a la superficie en un punto y examinemos el plano que contiene a $\mathbf{df}(\mathbf{X})$ y a $\mathbf{N}$. La intersección de este plano con la superficie es una curva. Llamaremos a la curvatura κ_n de esta curva la *curvatura normal en la dirección de* $\mathbf{X}$. En la Figura 2.7 vemos cómo se forma la intersección mencionada anteriormente, y la curva resultante. Utilizando la fórmula de Frenet-Serret podemos calcular la curvatura normal en la dirección de $\mathbf{X}$:

$$\kappa_n(\mathbf{X}) = \frac{\mathbf{df}(\mathbf{X}) \cdot \mathbf{dN}(\mathbf{X})}{|\mathbf{df}(\mathbf{X})|^2} \tag{2.5}$$

Esta ecuación se obtiene extrayendo la componente tangencial de la parte de la fórmula que define $\mathbf{dN}$.

Cuando hablamos de la curvatura de curvas, no importa si expresamos la curvatura utilizando el cambio en $\mathbf{T}$ o en $\mathbf{N}$. Sin embargo, cuando estudiamos la curvatura de superficies solemos expresar la curvatura en términos de $\mathbf{N}$. Esto es porque para cualquier

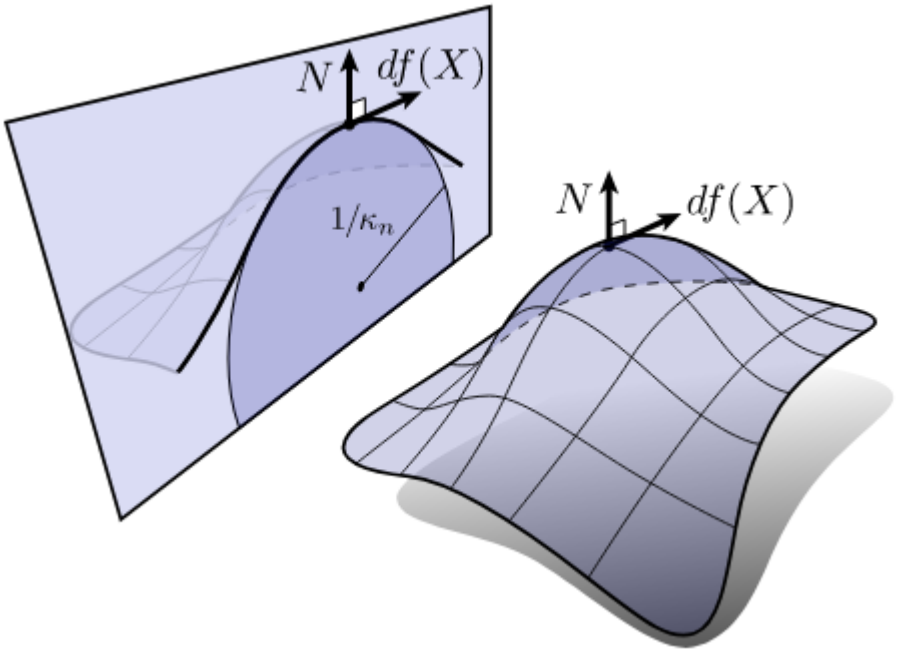


Figura 2.7: Curva resultante de la intersección de una superficie con el plano que contiene a **N** y **df**(**X**). También vemos el llamado *círculo osculador*, el cual tiene la misma curvatura que la curva en ese punto. Imágen de Keenan Crane [3].

punto de una superficie tenemos una normal principal, pero no tenemos un solo vector tangente.

Ahora que tenemos una definición para la curvatura de una superficie, definiremos algunos conceptos relacionados. Dado un punto en una superficie, las *direcciones principales (de curvatura)* $\mathbf{e_1}$ y $\mathbf{e_2}$ son los vectores unitarios tangentes a la curva en ese punto en cuyas direcciones las curvaturas son máximas y mínimas, respectivamente. Las *curvaturas principales* κ_1 y κ_2 son las curvaturas máximas y mínimas. Entonces, para todo punto la curvatura $k(\mathbf{p})$ en la dirección de algún vector $\mathbf{p}$ tendrá que cumplir que $k_2 \leq k(\mathbf{p}) \leq k_1$. Además, para calcular la curvatura $k(\mathbf{p})$ podemos utilizar la fórmula de Euler para la curvatura normal [6]:

$$\kappa(\mathbf{p}) = \kappa_1(\mathbf{p})\, cos^2\phi + \kappa_2(\mathbf{p})\, sin^2\phi \tag{2.6}$$

donde ϕ es el ángulo entre $\mathbf{p}$ y $\mathbf{e_1}$.

Otras dos definiciones importante son las de *curvatura Gaussiana* y *curvatura promedio*. La curvatura promedio H es igual a $\frac{\kappa_1+\kappa_2}{2}$, y la curvatura Gaussiana K es igual a $\kappa_1 * \kappa_2$. La curvatura Gaussiana en particular sirve mucho para desarrollar algunos algoritmos de dibujo de líneas.

Finalmente, debemos definir la *segunda forma fundamental*. Las dos formas fundamentales se denotan con **I** y **II**, pero nos basaremos en la segunda. A **II** también se la llama la matriz de Weingarten.

$$\mathbf{II}(\mathbf{X}, \mathbf{Y}) = -\mathbf{dN}(\mathbf{X}) \cdot \mathbf{df}(\mathbf{Y}) \tag{2.7}$$

II se puede definir también como una matriz utilizando las derivadas direccionales de la curvatura normal:

$$\mathbf{II} = \begin{bmatrix} e & f \\ f & g \end{bmatrix} = \begin{bmatrix} D_{\mathbf{u}}\,\mathbf{N} & D_{\mathbf{v}}\,\mathbf{N} \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathbf{N}}{\partial \mathbf{u}} \cdot \mathbf{u} & \frac{\partial \mathbf{N}}{\partial \mathbf{v}} \cdot \mathbf{u} \\ \frac{\partial \mathbf{N}}{\partial \mathbf{u}} \cdot \mathbf{v} & \frac{\partial \mathbf{N}}{\partial \mathbf{v}} \cdot \mathbf{v} \end{bmatrix} \tag{2.8}$$

Aquí e, f y g son las letras generalmente usadas para definir la matriz de Weingarten. Los vectores **u** y **v** definen un sistema de coordenadas ortonormal en el plano tangente al punto en el que estamos. $D_{\mathbf{r}}\,\mathbf{N}$ es la derivada direccional de **N** en la dirección de **r**. Utilizamos esta definición dada por Rusinkiewicz [20], ya que es la base de gran parte de los algoritmos que implementamos.

2.3. Occluding Contours

En esta sección describiremos a los *occluding contours*, a veces llamados simplemente *contours* en la literatura. La mayor parte de las definiciones y los conceptos descriptos a continuación provienen del trabajo de Pierre Bénard y Aaron Hertzmann [1]. Nos limitaremos a explicar los fundamentos que sostienen los conceptos detrás de occluding contours, para dar lugar en las siguientes secciones a los detalles de los distintos algoritmos que se utilizamos para calcularlos. Supongamos que tenemos un objeto en 3D que estamos observando desde un punto de vista específico. Una definición coloquial consiste en decir que los occluding contours son las curvas sobre el objeto que separan las regiones que podemos ver de las que no podemos ver. En la Figura 2.8 vemos un ejemplo de un modelo 3D junto con el dibujo de las curvas que componen los occluding contours.

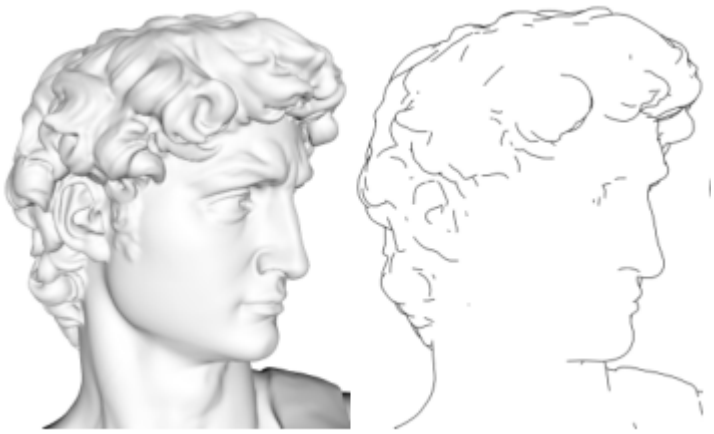


Figura 2.8: A la izquierda una renderización del modelo 3D de David de "Scan The World" (http://mmf.io/o/2052) y a la derecha los occluding contours del modelo dibujados desde el mismo punto de vista. Imágen extraída de Line Drawing From 3D Models: A Tutorial [1].

Dado un modelo 3D compuesto de triángulos y una posición de cámara $\mathbf{c}$, podemos calcular para cada cara de triángulo si está mirando a la cámara o no. Una cara de un triángulo estará mirando la cámara si su normal se encuentra del mismo lado que el vector que apunta de la cara a la cámara. Más formalmente, $(\mathbf{c} - \mathbf{p}) \cdot \mathbf{N} > 0$ para todo punto $\mathbf{p}$ en el triángulo. En este caso, diremos que la cara está *orientada hacia adelante*. En caso contrario, estará *orientada hacia atrás*. Utilizando estos conceptos de orientación, podemos mejorar nuestra definición anterior:

Definición 2.1 *El* ***occluding contour generator*** *es, dado un punto de vista específico, la curva sobre la superficie de un objeto que marca la frontera entre regiones de caras orientadas hacia adelante y caras orientadas hacia atrás.*

El lector notará que aquí hemos definido occluding contour *generators*. Utilizamos la palabra *generator* para diferenciar entre las curvas sobre el objeto (en tres dimensiones) y las curvas que dibujaremos, que son bidimensionales.

Definición 2.2 *Para un punto de vista, el* occluding contour *es la proyección a dos dimensiones del occluding contour generator.*

Ahora que tenemos un marco teórico que forma la base del dibujo automático de bocetos, en el siguiente capítulo veremos cómo se aplican estos conceptos en *rtsc*, primero en la teoría de *suggestive contours* y luego en la implementación en si. Luego, en los capítulos que siguen podremos explorar la implementación hecha a lo largo de este proyecto.

Capítulo 3

Suggestive contours en *rtsc*

En este capítulo desarrollaremos los conceptos principales alrededor de uno de los algoritmos más conocidos implementados en *rtsc*: *suggestive contours* [5]. Esta tesis se centra en la implementación de este algoritmo en otro lenguaje y bajo un nuevo framework. Para esto se ha realizado un trabajo de reingeniería de algoritmos encontrados en *rtsc*. Gran parte de los algoritmos que re-implementamos se encuentran en una librería auxiliar que es utilizada en *rtsc*: *trimesh*. Esta librería contiene los algoritmos que veremos en la siguiente sección, los cuales permiten extraer información de una malla necesaria para implementar *suggestive contours*.

La teoría detrás de los algoritmos que presentaremos en este capítulo está formulada extensamente en diversos artículos, pero su implementación (en C++) no ha sido bien documentada. Por ende, aquí desarrollaremos las ideas principales detrás de las curvas que se renderizan con *suggestive contours*, las definiciones (en términos de geometría diferencial), y los lineamientos generales de los algoritmos. Luego, seguiremos el flujo de *rtsc* que va desde la malla 3D, hasta llegar al dibujo de las lineas en dos dimensiones. La implementación encontrada en *rtsc* es distinta a la que se propuso en el trabajo original de DeCarlo et al. [5]. En esta tesis se introducen algunos cambios basados trabajos posteriores [4] [20] y también se hacen algunas simplificaciones que explicaremos oportunamente. En el Capítulo 4 exploraremos estos cambios y su impacto en los resultados.

3.1. Cómputo de curvaturas y normales en mallas de triángulos

Como primer paso, debemos calcular varias propiedades de la malla. Específicamente, es necesario tener acceso a las direcciones principales de curvatura (e_1 y e_2), las curvaturas principales (κ_1 y κ_2) y al tensor de derivadas de curvatura ($\mathbf{C}$) en cada vértice de la malla. También es necesario obtener las normales en cada vértice. Para obtener estas propiedades se puede aplicar el algoritmo para la estimación de curvaturas sobre una malla detallado por Rusinkiewicz en *Estimating Curvatures and Their Derivatives on Triangle Meshes* [20].

Este algoritmo está compuesto de varios pasos. En primer lugar, se debe calcular el valor de los vectores normales a la malla en cada vértice. Para esto, se aplica un algoritmo que calcula la normal de cada triángulo de la malla y le asigna una normal a cada vértice en base a las normales de los triángulos que lo contienen. Además, este vector se escala utilizando factores proporcionales a las áreas de los triángulos que comparten cada normal. Para implementar este algoritmo de forma eficiente, se itera sobre las caras de los triángulos de la malla, calculando el producto cruz de los bordes. El vector resultante tendrá la misma dirección que el vector normal del triángulo, pero la magnitud será proporcional al área del mismo. En cada vértice se acumula una suma de estos vectores de cada cara a la que pertenece el vértice. Finalmente, el vector resultante en cada vértice se normaliza para obtener una estimación del vector normal. A continuación presentamos el pseudocódigo de este algoritmo:

```
def calcularNormales(mesh):
    for face in mesh.faces:
        normal_pesada_cara = calcularProductoCruz(face)

        for vertice in face:
            normales[vertice] += normal_pesada_cara

    for normal in normales:
        normalizar(normal)

    return normales
```

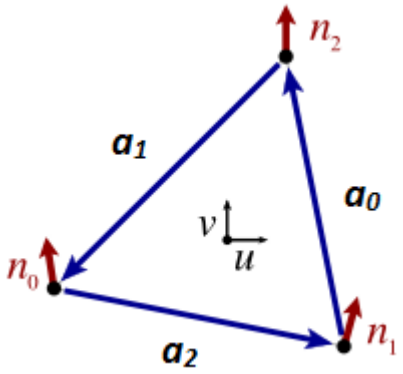


Figura 3.1: Sistema de coordenadas en base al cual se calcularán las curvaturas principales y sus direcciones. Obtenido de Rusinkiewicz [20].

Adicionalmente, en *rtsc* se implementó una variación de este algoritmo, en el cual se pesan las normales de cada triángulo asociado a un vértice v_i utilizando $\frac{n_f}{|a_1|^2|a_2|^2}$, dónde n_f es la normal del triángulo y a_1 y a_2 son las aristas del triángulo que contienen a v_i. Esta variación fue propuesta por Max [13] como una mejor aproximación al cálculo de normales.

Por otro lado, y como veremos más adelante, para el cómputo de la curvatura por vértice se debe comenzar por calcular la curvatura sobre los triángulos de la malla. Luego, se asigna una porción de la curvatura de cada triángulo a cada vértice que lo compone. Hay distintos métodos propuestos para determinar la porción de la curvatura que se asignará a distintos vértices. Siguiendo las recomendaciones de Meyer et al. [14], en *rtsc* se generan tres pesos por triángulo en base a la proporción del área de ese triángulo que está más cerca de sus tres vértices, y estos se utilizan al momento de acumular la curvatura en los vértices.

Las curvaturas principales y sus direcciones en cada triángulo se calculan en un sistema de coordenadas definido por el mismo triángulo. Este sistema de coordenadas lo podemos ver en la Figura 3.1, donde n_i indica el vector normal en el vértice i. Para obtener e_1 y e_2 en cada vértice, se debe considerar cada uno de los triángulos a los cuales pertenece el vértice. Esto requiere que se trabaje con varios sistemas de coordenadas, lo cual significa que es necesario convertir los vectores obtenidos en cada triángulo a un mismo sistema de coordenadas. Llamemos e_1^f y e_2^f a los vectores de direcciones principales de curvatura obtenidos en el triángulo f. Para obtener e_{1p} y e_{2p} (las direcciones principales de curvatura en el vértice p), se define un sistema de coordenadas por cada vértice, y se convierten e_1^f y e_2^f a estas coordenadas. Luego, se calcula un promedio pesado utilizando los vectores

correspondientes y los pesos $w_{f,p}$.

Volvamos a la matriz de la segunda forma fundamental definida en el Capítulo 2, denotada con **II**. Al multiplicar **II** por cualquier vector en el plano tangente a un punto, obtenemos la derivada de la normal en esa dirección, en ese punto:

$$\mathbf{II}\,\mathbf{s} = D_s n \tag{3.1}$$

En la ecuación 2.8 vemos además que **II** se puede definir en términos de derivadas direccionales del vector normal a la superficie. Este último hecho es importante, ya que significa que si obtenemos una aproximación de las derivadas direccionales del vector normal, podemos utilizar dicha aproximación para aproximar **II**. Luego, utilizando **II** podemos obtener curvaturas y derivadas de curvatura en cada vértice.

Para nuestro caso discreto, se pueden aproximar las derivadas direccionales de la normal utilizando las diferencias entre las normales en los vértices, como proponen DeCarlo et al [5]. A su vez, si observamos la Figura 3.1, vemos que **II** debe cumplir:

$$\mathbf{II}\begin{bmatrix} a_i \cdot u \\ a_i \cdot v \end{bmatrix} = \begin{bmatrix} (n_j - n_k) \cdot u \\ (n_j - n_k) \cdot v \end{bmatrix} \tag{3.2}$$

donde $j = (i+1)$ mód 3 y $k = (i-1)$ mód 3. Esto resulta en un sistema de ecuaciones con las cuales se pueden utilizar el método de cuadrados mínimos para determinar los coeficientes de la matriz **II** en términos de las derivadas direccionales de n en las direcciones u y v. Una vez que se encuentra una solución, se hace un cambio de coordenadas de las derivadas direccionales al sistema de coordenadas de cada vértice del triángulo. Con estas propiedades calculadas para cada triángulo y con los pesos que mencionamos anteriormente, finalmente se obtiene una aproximación de **II** en el triángulo. Usando esta aproximación, se calculan e_1, e_2, κ_1 y κ_2 como los autovectores y autovalores, respectivamente, de la matriz. De esta forma se obtienen las curvaturas principales y sus direcciones en cada vértice.

3.2. Derivadas de Curvatura

Otra propiedad de la malla necesaria para los algoritmos de dibujo de líneas es la derivada de curvatura. Este es un tensor de $2 * 2 * 2$ el cual provee una forma de calcular derivadas direccionales de curvatura en direcciones tangentes a la superficie. Llamaremos $\mathbf{C}$ a este tensor. $\mathbf{C}$ tiene varias propiedades interesantes, algunas explicadas por Rusinkiewicz [20]. Las más importantes para este trabajo son:

Propiedad 3.1 *Multiplicar* $\mathbf{C}$ *por un vector* $\boldsymbol{w}$ *(definido en el plano tangente a la superficie en un punto) tres veces nos da la derivada de la curvatura en esa dirección. A esta multiplicación la denotaremos* $\mathbf{C}(\mathbf{w}, \mathbf{w}, \mathbf{w})$.

Propiedad 3.2

$$D_{\mathbf{w}}\, \kappa_r = \frac{\mathbf{C}(\mathbf{w}, \mathbf{w}, \mathbf{w})}{||\mathbf{w}||^2} + 2K\cot(\theta)\, ||\mathbf{w}||, \; si\; \kappa_r = 0 \tag{3.3}$$

Donde θ *es el ángulo entre* $\boldsymbol{n}$ *y* $\boldsymbol{v}$*, y* K *es la curvatura gaussiana.*

La propiedad 3.2 se encuentra en el trabajo de DeCarlo, Finkelstein y Rusinkiewicz [4].

$\mathbf{C}$ tiene la forma:

$$\mathbf{C} = (\, D_u\, \mathbf{II} \quad D_v\, \mathbf{II}\,) = \left(\begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} b & c \\ c & d \end{bmatrix} \right) \tag{3.4}$$

Lo cual significa que solo debemos calcular cuatro valores.

Una vez conocido el algoritmo para calcular κ_i y e_i (las curvaturas principales y direcciones de curvatura principales, respectivamente) en cada vértice de una malla, el algoritmo para calcular $\mathbf{C}$ se puede implementar de forma relativamente directa. La idea es la misma que la del algoritmo de curvatura, pero en vez de aproximar usando diferencias entre normales, aproximamos la derivada de curvatura utilizando diferencias entre la segunda forma fundamental. Esto significa que para este algoritmo debemos haber calculado $\mathbf{II}$ en cada vértice, utilizando el algoritmo anterior.

Al igual que cuando calculamos $\mathbf{II}$, debemos minimizar cuadrados para obtener una aproximación de $\mathbf{C}$. Sin embargo, a diferencia del algoritmo de curvatura, una vez obteni-

da esta aproximación no necesitamos calcular autovalores ni autovectores, ya que lo que queremos calcular es la matriz en si.

3.3. Suggestive Contours

DeCarlo et al [5] presentan varias definiciones de lo que es un *suggestive contour generator*. Para dar una idea intuitiva, lo definimos como el conjunto de puntos de una superficie que casi forman un *occluding contour generator* (definición 2.1) desde la posición del observador. Se puede pensar el *suggestive contour generator* como una continuación de un generador de contorno, como se puede ver en la Figura 3.2. Un *suggestive contour* es la proyección del *suggestive contour generator* al plano: es lo que dibujamos en la pantalla. De ahora en más usaremos *suggestive contour* para referirnos a ambos conceptos; de cuál hablamos dependerá del contexto. Los *suggestive contours* nos proporcionan más detalles en un dibujo de una superficie que el contorno solo y, junto con el contorno, generan bocetos más cercanos a los que haría un artista. En la Figura 3.3 vemos un ejemplo de dos dibujos de una superficie: uno del contorno y otro del contorno junto con el *suggestive contour*.

Para definir un *occluding contour generator*, es necesario contar con una dirección desde la cual se observa el modelo. Definimos a continuación esta dirección:

Definición 3.1 *El vector de vista respecto a* $\mathbf{p}$ *(denotado generalmente con* $\mathbf{v}(\mathbf{p})$*) es el resultado de normalizar el vector desde el punto* $\mathbf{p}$ *hasta la cámara (el punto desde el cual observamos la escena).*

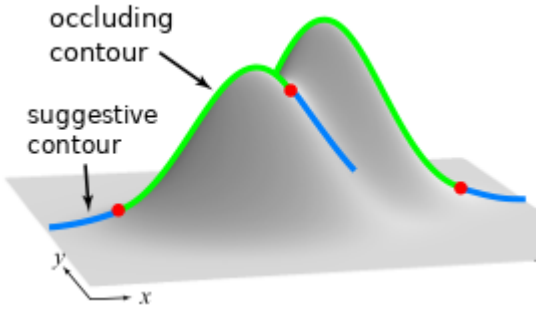


Figura 3.2: *Suggestive contour* como continuación de un generador de *occluding contour*. Imagen de *Suggestive Contours for Conveying Shape* [5]

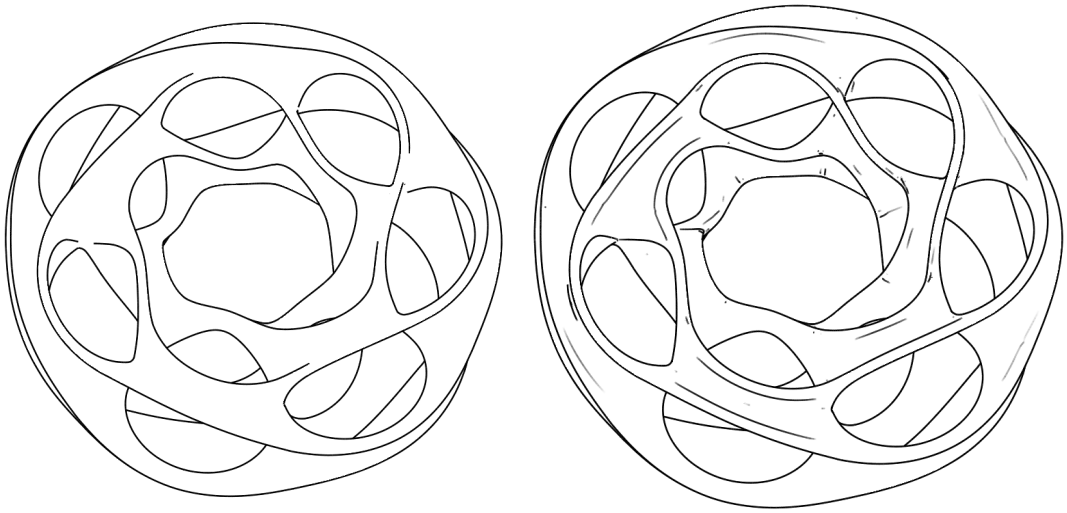


Figura 3.3: A la izquierda, un boceto de un "heptoroid" solo con contorno. A la derecha, con contorno y *suggestive contour*. Notar que con *suggestive contour* se puede apreciar más la curvatura de las caras y los detalles más finos. Modelo cortesía del "Princeton Graphics Group".

En la mayoría de los casos omitiremos nombrar a $\mathbf{p}$, y se entenderá que estamos trabajando sobre algún punto particular de la malla. Para dar la definición formal de *suggestive contour*, también debemos definir la curvatura radial (κ_r) y el vector $\mathbf{w}$. Dado un vector de vista $\mathbf{v}$ y un punto de la superficie $\mathbf{p}$, $\mathbf{w}$ es el la proyección de $\mathbf{v}$ al plano tangente en $\mathbf{p}$. Es importante notar que $\mathbf{w}$ no es necesariamente un vector unitario. κ_r es la curvatura normal de la superficie en la dirección de $\mathbf{w}$. A partir de estas aclaraciones definimos formalmente a un *suggestive contour* de la siguiente manera:

Definición 3.2 *El* suggestive contour *es el conjunto de puntos en la superficie en los cuales la curvatura radial* κ_r *es igual a* 0 *y* $D_{\mathbf{w}}\,\kappa_r > 0$.

Bajo esta definición, un punto de un *suggestive contour* representa un punto de inflexión en la curvatura en la dirección de la cámara. Como se puede ver en la Figura 3.4, la curvatura pasa de ser cóncava a ser convexa mientras nos movemos hacia la cámara. Por lo tanto, si la cámara se moviese hacia la superficie, el punto $\mathbf{p}$ pasaría a formar parte de un generador de contorno.

Existe un problema de estabilidad a la hora de calcular el *suggestive contour*: si el vector de vista hacia la cámara es aproximadamente paralelo al vector normal en un punto, pequeñas variaciones en el vector de vista (por movimientos de la cámara) pueden cambiar

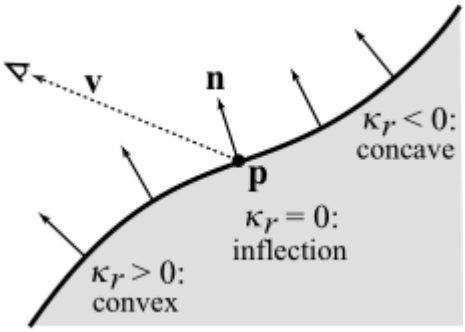


Figura 3.4: Puntos de un *suggestive contour* como puntos de inflexión. Imagen de *Suggestive Contours for Conveying Shape* [5]

drásticamente el generador. Bajo estas condiciones, el *suggestive contour* es demasiado inestable para proveer información útil. Por esta razón, se utiliza un hiper-parámetro para controlar el ángulo del vector de vista con respecto al vector normal:

$$0 < \theta_c < \cos^{-1}\left(\frac{\mathbf{n}(\mathbf{p}) \cdot \mathbf{v}(\mathbf{p})}{\|\mathbf{v}(\mathbf{p})\|}\right) \tag{3.5}$$

Por otro lado, surgen problemas adicionales causados por el ruido y los errores de estimación de curvatura. Estos problemas pueden resultar en generadores incorrectos en algunos casos. Para evitar estos casos, se utiliza un hiper-parámetro adicional para acotar el valor mínimo de $D_{\mathbf{w}}\,\kappa_r$:

$$t_d < \frac{D_{\mathbf{w}}\,\kappa_r}{\|\mathbf{w}\|} \tag{3.6}$$

El algoritmo para calcular *suggestive contours* se basa en el trabajo de Hertzmann y Zorin [8]. Primero, se calculan las curvaturas y las derivadas de curvatura de la malla utilizando los algoritmos explicados anteriormente en este capítulo. Opcionalmente, antes de calcular dichas propiedades se puede aplicar un algoritmo de suavizado, o "smoothing". Estos algoritmos suavizan la malla, moviendo vértices en base a la ubicación promedio de vértices cercanos, o aplicando un filtro Gaussiano a los vectores normales.

Una vez que se tienen las curvaturas y derivadas de curvatura, se buscan todos los vértices en los que $\kappa_r = 0$. Para calcular κ_r, se aplica la propiedad 2.6. Aplicando el algoritmo de Hertzmann y Zorin, se obtiene un conjunto de curvas sobre la superficie, el cual

se filtra utilizando los parámetros t_d y θ_c provistos por el usuario.

El algoritmo de filtrado comienza con el cómputo de $D_{\mathbf{w}}\,\kappa_r$ usando $\mathbf{C}$ y κ_r. En base a este valor se aplica la umbralización con histéresis de Canny [2]. Se filtran primero usando t_d y θ_c y luego se agregan los vecinos de segmentos de curva aplicando umbrales t'_d y θ'_c. Estos últimos umbrales son menos estrictos y se calculan a partir de t_d y θ_c como parte del algoritmo. Finalmente, se descartan aquellos segmentos de curva más cortas que un umbral t_s.

En la Figura 3.5 vemos un diagrama del flujo de datos y la estructura general de los algoritmos que se utilizan para dibujar *suggestive contours*.

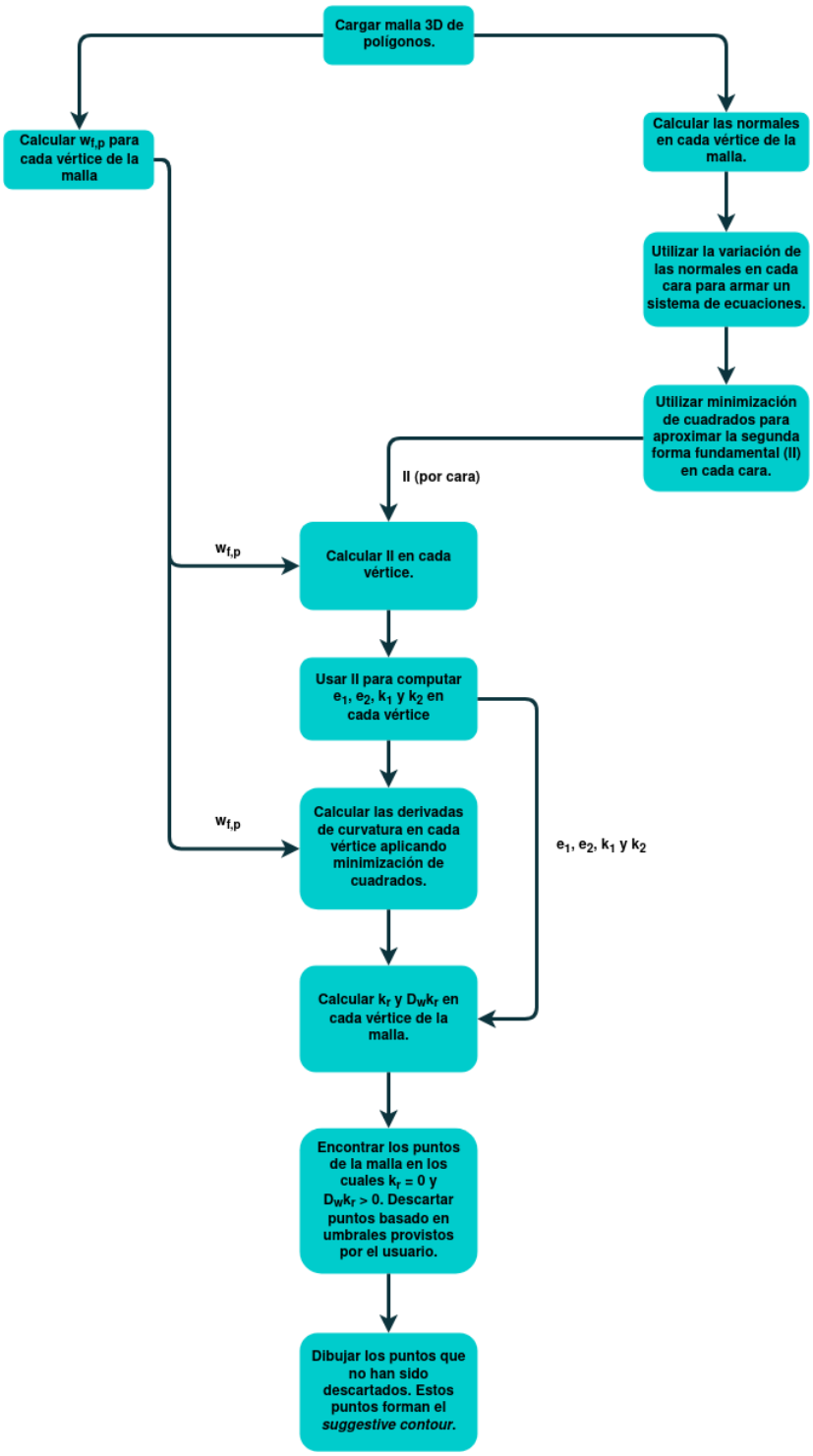


Figura 3.5: Pipeline de extracción de características geométricas y rendering de *suggestive contours*.

3.4. Detalles de implementación de *rtsc*

En las secciones anteriores exploramos el algoritmo de *suggestive contours* y los algoritmos para el cómputo de propiedades de una malla desde el punto de vista teórico. A continuación, nos enfocaremos en el análisis particular de su implementación en *rtsc*, centrándonos en los cambios que realizaron sus autores respecto a la teoría inicial. En *rtsc*, el cómputo de las normales, curvaturas y derivadas de curvatura se realiza con la librería *trimesh*, respetando las descripciones realizadas en las secciones anteriores. Sin embargo, los algoritmos para computar $D_{\mathbf{w}}\,\kappa_r$ y κ_r difieren en algunos puntos de lo descripto hasta el momento. La razón principal dada por DeCarlo, Finkelstein y Rusinkiewicz [4] para introducir estas modificaciones es la de aumentar la velocidad de cómputo y la estabilidad numérica.

Dos cambios importantes que se desarrollaron en publicaciones posteriores de los autores de *rtsc* son:

1. Combinar los dos hiper-parámetros explicados en la Sección 3.3 en uno solo.
2. Mantener la coherencia temporal de los *suggestive contours*.

Como en esta tesis no nos interesa trabajar con secuencias de imágenes (sino renders estáticos), nos concentramos en la descripción del punto 1. Para lograrlo, solamente se acepta un punto en la malla como parte de un *suggestive contour* si:

$$sin^2(\theta)\frac{D_{\mathbf{w}}\,\kappa_r}{||\mathbf{w}||} > t_d \tag{3.7}$$

Aquí t_d es el único hiper-parámetro que se le pide al usuario (umbral). Para utilizar este hiper-parámetro como cota, primero se normalizan las curvaturas y las derivadas de curvatura en base al tamaño del modelo. Esto se hace para que el usuario no tenga que cambiar la cota al cambiar de un modelo a otro. Se define s como la mediana de las longitudes de un borde en la malla. Todas las curvaturas se multiplican por s, y todas las derivadas de curvatura por s^2. Equivalentemente, se pueden dividir las cotas por s o por s^2 cuando se van a realizar pruebas sobre curvaturas o derivadas de curvatura, respectivamente.

Al utilizar la ecuación 3.7, ya se asegura que se cumple la ecuación 3.6. La cota sobre θ está contenida en el término $sin^2(\theta)$, ya que:

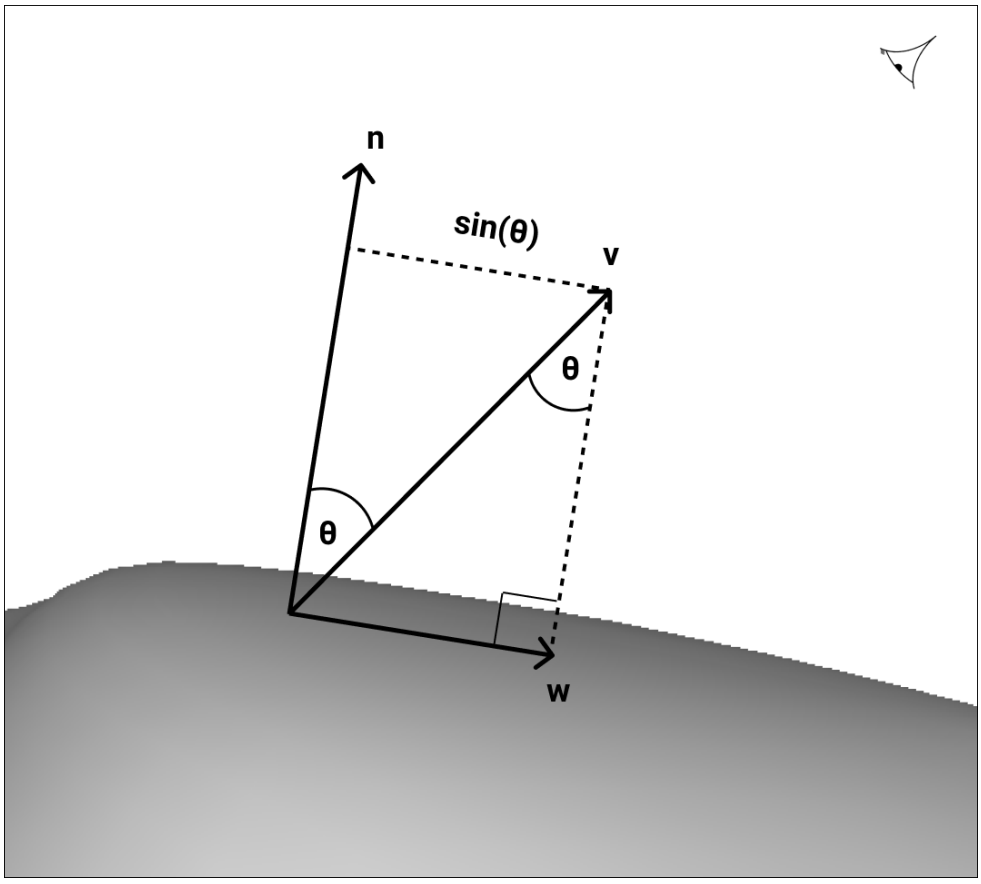


Figura 3.6: Demostración visual de que $sin(\theta) = ||w||$. Es importante para esta prueba recordar que $||v|| = 1$.

$$sin^2(\theta) = \frac{\mathbf{w} \cdot \mathbf{w}}{\mathbf{v} \cdot \mathbf{v}} = 1 - \left(\frac{\mathbf{n} \cdot \mathbf{v}}{||\mathbf{v}||}\right)^2 \tag{3.8}$$

En la Figura 3.6 se puede observar que $sin^2(\theta) = w \cdot w$ utilizando semejanza de triángulos. A partir de esta igualdad y mediante propiedades trigonométricas y del producto escalar deducimos la ecuación anterior.

DeCarlo, Finkelstein y Rusinkiewicz [4] encontraron que reducir la cantidad de parámetros de esta forma no afecta la calidad de control ofrecida al usuario. Sin embargo, en la forma de aplicar el umbral, la implementación de *rtsc* también difiere de esta descripción. En lugar de utilizar la desigualdad 3.7, *rtsc* verifica que:

$$\frac{D_{\mathbf{w}}\, \kappa_r}{||\mathbf{w}||} \tan(\theta) > t_d \tag{3.9}$$

Debido a que $sin^2(\theta) = ||\mathbf{w}||^2$ (ver la ecuación 3.8) y a la propiedad 3.2, esto es lo mismo que:

$$\frac{\mathbf{C}(\mathbf{w}, \mathbf{w}, \mathbf{w})}{\cos(\theta) * ||\mathbf{w}||^2} + 2K > t_d \tag{3.10}$$

Aunque en *rtsc* no se provee una explicación de por qué se calcula el producto con $\tan(\theta)$ en lugar de $\sin^2(\theta)$, computar el lado izquierdo de la desigualdad anterior es más eficiente y más estable, ya que se evita calcular $\cot(\theta)$. Llamaremos al resultado de la ecuación anterior *sctest*.

Capítulo 4

Implementación de los Algoritmos

En este capítulo describiremos cómo se implementaron los algoritmos para el dibujo de líneas en el lenguaje Python, utilizando principalmente la librería PyTorch. Esta librería hace uso de tensores, los cuales permiten hacer operaciones en simultáneo sobre grandes cantidades de datos. Aunque no nos enfocamos en la velocidad de cómputo, estos tensores se utilizaron en casi todas las operaciones para poder acelerar el programa. Esto es sumamente importante, ya que las implementaciones utilizando Python son en general mucho más lentas que en C++ (el lenguaje utilizado para *rtsc*).

Los problemas de estabilidad jugaron un rol importante en nuestra implementación, lo que motivó el diseño de algoritmos alternativos para explorar distintos métodos que mejoren la estabilidad numérica. En este capítulo detallaremos estos algoritmos y en el Capítulo 5 mostraremos los resultados de las distintas implementaciones.

4.1. Cómputo de Normales

Como explicamos en el Capítulo 3, los algoritmos de dibujo de línea requieren que se tenga acceso a las normales en cada vértice de la malla. Por lo tanto, lo primero que se debe hacer es calcular las normales a partir de la malla. A continuación mostramos el pseudocódigo de la implementación utilizando tensores del algoritmo que utiliza las áreas de los triángulos como pesos. Es importante notar que solamente iteramos sobre las caras para sumar las normales pesadas; el resto del algoritmo se hace con tensores y de forma paralela, sin necesidad de iterar. Esta última iteración es necesaria para evitar condiciones de carrera, por las cuales no es posible realizar las operaciones de forma paralela.

Antes de continuar con el pseudocódigo de los algoritmos, debemos aclarar la notación utilizada. Al trabajar con tensores, que son matrices multidimensionales, debemos poder acceder a columnas, filas, a todos los elementos en alguna dimensión, a un elemento particular, etc. Por esto, dada una matriz A de $n * m$, utilizamos la notación $A[:, j]$ para acceder a la j-ésima columna, $A[i, :]$ para acceder a la i-ésima fila y $A[i, j]$ para acceder al elemento $A_{i,j}$. Esto se extiende a cualquier cantidad de dimensiones.

```
def computeSimpleVertexNormals(mesh):
    # Las caras de la malla contienen los 3 índices de
    # los vértices que las componen.
    faces = mesh.faces
    # La ubicación de los vértices en el espacio 3D.
    verts = mesh.vertices

    indices_vert1 = verts[faces[:,0]]
    indices_vert2 = verts[faces[:,1]]
    indices_vert3 = verts[faces[:,2]]

    aristas_1 = verts[indices_vert2] - verts[indices_vert1]
    aristas_2 = verts[indices_vert3] - verts[indices_vert1]

    # Vectores normales a las caras, proporcionales a las
    # áreas de las caras correspondientes.
    normales_pesadas = productoCruz(aristas_1, aristas_2)

    # Acumulamos en el índice de cada vértice la suma de
    # las normales pesadas de las caras de las cuales
    # forman parte.
    for face in faces:
        normales[cara[0]] += normales_pesadas[indice(face)]
        normales[cara[1]] += normales_pesadas[indice(face)]
        normales[cara[2]] += normales_pesadas[indice(face)]

    normalizar(normales)

    return normales
```

El algoritmo alternativo, utilizando lo que propone Max [13] es casi idéntico. Solamente se agrega una multiplicación de las normales pesadas antes de sumarlas a las normales.

Para verificar que nuestro cálculo de normales sea correcto, visualizamos las normales

Figura 4.1: Visualización de normales sobre una malla de una vaca.

sobre la malla y comparamos el resultado con aquel generado por *rtsc*. Las tres componentes de las normales fueron convertidas a una escala RGB, para facilitar su visualización como coloreo sobre la malla (ver Figura 4.1)

4.2. Cómputo de Curvaturas

Una vez obtenidos los vectores normales en cada vértice, lo único que nos falta para calcular las curvaturas de la malla son los pesos, $w_{f,p}$. Como nuestra implementación es muy similar a la de *trimesh* y a lo descripto en la formulación teórica, omitimos la descripción detallada del algoritmo.

Una vez obtenidos los pesos, podemos proceder a calcular las curvaturas en cada vértice de la malla. En primer lugar, implementamos el algoritmo de la misma forma que está implementado en *trimesh*, y luego procedimos a optimizarlo utilizando tensores. Esta optimización presentó varios retos. El primero fue que, de forma similar al cálculo de normales, se pueden producir condiciones de carrera al acumular las curvaturas calculadas por cada cara en los vértices. Por esta razón, algunas partes del código debieron ser implementadas para su ejecución secuencial.

El segundo reto tuvo que ver con encontrar la solución de cuadrados mínimos. Usando PyTorch y Numpy, tenemos acceso a varios métodos para la resolución de este problema. Ya que conocemos ciertas propiedades de la matriz **II**, existe la posibilidad de usar descomposiciones específicas que tomen ventaja de la estructura de la matriz para conseguir una solución de forma más eficiente. Evaluamos dos posibles alternativas:

1. Utilizar el solver de least squares implementado en PyTorch (*torch.lstsq*).

2. Descomponiendo la matriz utilizando la descomposición de Cholesky (*torch.cholesky*) y resolviendo el sistema de ecuaciones resultante mediante *torch.cholesky_solve*.

Al implementar los algoritmos completos para el cómputo de las curvaturas, se observó que los resultados no eran idénticos a aquellos generados por *trimesh* dentro de *rtsc*. Esta diferencia se debe fundamentalmente a una diferencia en las soluciones generadas por *torch.lstsq* y *torch.cholesky_solve* con respecto a la solución provista por el solver de *trimesh*. Esta solución se obtiene descomponiendo la matriz utilizando una descomposición LDL^T, una variación de la descomposición de Cholesky que es libre de raíces cuadradas. El algoritmo utilizado es el de Golub y van Loan [7]. No calcular raíces cuadradas puede ser importante cuando las dimensiones de las matrices son pequeñas pero, como veremos, en nuestro caso este aspecto no impactó de manera significativa.

En el código que sigue mostramos cómo calculamos las curvaturas de una malla en PyTorch. Utilizamos tensores siempre que era posible, y trabajamos en el GPU en todo momento. En la Figura 4.2 hay un recordatorio de la parte del flujo de *suggestive contours* que corresponde al cálculo de curvatura.

```
def computeCurvatures(mesh, method="lstsq"):
    verts = mesh.vertices
    faces = mesh.faces
    normals = computeNormals(mesh)
    pointareas, cornerareas = computePointareas(mesh)

    # Creo un sistema de coordenadas inicial por cada vertice
    e1 = crearSistemaCoordenadas(mesh)
```

```
# Producto cruz fila a fila para que e1 y e2 estén en el
# plano de la cara y sean ortogonales.
e1 = productoCruz(e1, normals)
# Normalización de cada fila
normalizar(e1)
e2 = productoCruz(normals, e1)

# Computar aristas por cara
edges = calcularAristas(faces, verts)

# Uso el marco de Frenet por cada cara, y
# uso la primer arista de cada cara como vector tangente.
t = edges[:,0]
normalizar(t)
n = productoCruz(edges[:,0], edges[:,1])
b = productoCruz(n, t)
b = normalizar(b)

# Estimo la curvatura basado en la variación de las normales
# m y w son tensores que contienen una matriz de dimensión
# 3x1 y 3x3, respectivamente, por cada cara.
m, w = estimarMatricesVariaciónNormal(t, n, b)

# Resuelvo mínimos cuadrados por cada cara
# Esto no es paralelizable, es necesario iterar
for i in [0, ..., count(faces)]:
    m[i] = resolveLeastSquares(m[i], w[i])

# Sumar los valores computados en los vertices
for j in [0, 1, 2]:
    indices_vert = faces[:,j]
    c1, c12, c2 = proyectarCurv(t, b, m, e1[indices_vert],
                                e2[indices_vert])

    wt = cornerareas[:,j] / pointareas[indices_vert]
    # Sumo la curvatura en cada vertice con los pesos w[i]
```

```
        for i in [0, ..., count(faces)]:
            curv1[indices_vert[i]] += wt[i] * c1[i]
            curv12[indices_vert[i]] += wt[i] * c12[i]
            curv2[indices_vert[i]] += wt[i] * c2[i]

    # Computo direcciones y curvaturas principales en cada vértice
    return diagonalizarCurvatura(e1, e2,curv1, curv12,
                                 curv2, normals)
```

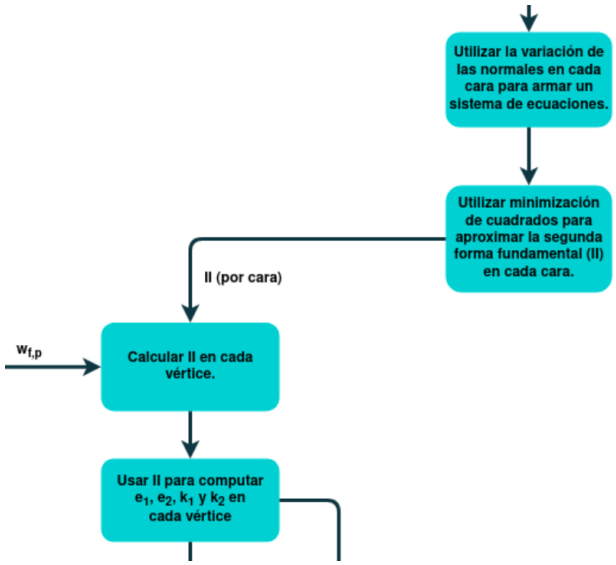


Figura 4.2: La porción del flujo del algoritmo que corresponde al cálculo de curvatura.

En este cálculo, cornerareas[i, j] es el área de la cara i-ésima que está más cerca del j-ésimo vértice de esa cara, y pointareas[k] es la sumatoria de todas las cornerareas que involucran al k-ésimo vértice. La función *projCurv(u, v, e, f, g, new_u, new_v)* convierte una curvatura dada por e, f y g de una base (dada por u y v) a una nueva base (dada por new_u y new_v). La función diagonializarCurvatura devuelve las direcciones de curvatura principales y las curvaturas asociadas a ellas.

Las matrices w y m en el algoritmo las utilizamos para la minimización de cuadrados necesaria para calcular las propiedades de curvatura. Contienen la información de las ecuaciones 3.2, y proveen restricciones que deben cumplir e, f y g.

4.3. Cómputo de Derivadas de Curvaturas

Nuestro algoritmo para calcular las derivadas de curvatura es igual al que implementamos para las propiedades de curvatura hasta el punto en el que creamos las matrices m y w:

```
def computeDcurvs(mesh):
    ...

    # Cambiar las base de las direcciones y curvaturas
    # principales a las de las caras.
    fcurv = proyectarCurv(e1, e2, k1, 0, k2, t, b)

    # Estimo las derivadas de curvatura en base a cómo
    # varía la curvatura en los vértices.
    m, w = estimarMatricesVariaciónCurvatura(fcurv, t, n, b)

    # Resuelvo mínimos cuadrados por cada cara
    for i in [0, ..., count(faces)]:
        m[i] = resolveLeastSquares(m[i], w[i])

    # Propago los valores a los vértices
    for j in [0, 1, 2]:
        indices_vert = faces[:,j]
        vert_dcurv = proyectarDcurv(t, b, m, e1[indices_vert],
                                    e2[indices_vert])
        wt = cornerareas[:,j] / pointareas[indices_vert]
        for i in range(0, faces.shape[0]):
            dcurv[indices_vert[i]] += wt[i] * vert_dcurv[i]

    return dcurv
```

Es importante notar que utilizamos los mismos pesos para calcular las derivadas de curvatura y para calcular las anteriores propiedades de curvatura. En este algoritmo también utilizamos los métodos para minimizar cuadrados que aplicamos anteriormente.

4.4. Cómputo de $D_{\mathbf{w}}\kappa_r$

En el cálculo de $D_{\mathbf{w}}\kappa_r$ nuestra implementación difiere de la implementación de *rtsc*. Como hemos visto, *rtsc* calcula *sctest* en lugar de $D_{\mathbf{w}}\kappa_r$ y κ_r. En *pyrtsc* calculamos *sctest*, pero también implementamos un cálculo directo de $D_{\mathbf{w}}\kappa_r$. En la Sección 5.2 compararemos la performance de ambas alternativas. Comenzaremos esta sección describiendo el cómputo de $D_{\mathbf{w}}\kappa_r$.

Primero, debemos definir una posición de la cámara, $\mathbf{c}$. Una vez establecida esta posición, calculamos por cada vértice el vector $\mathbf{v}$ correspondiente. También calculamos $n \cdot v$. Con estos dos valores calculados en cada vértice, podemos proceder a calcular $D_{\mathbf{w}}\kappa_r$.

Nuestra implementación inicial fue la siguiente:

```
def computeDwKr(mesh):
    # Calculamos todas las propiedades necesarias
    ...
    # viewdirs contiene los vectores v de cada vértice
    w = viewdirs - normals * ndotv

    # Igual que cuando calculo dcurv, uso e1[i] y e2[i] como
    # base del sistema de coordenadas del vértice i.
    u = dot(w, e1)
    v = dot(w, e2)
    u2 = u^2
    v2 = v^2

    # phi es el ángulo entre w y e1
    cosphi = u
    cos2phi = cosphi^2
    sin2phi = 1.0 - cos2phi
    # Raiz cuadrada de cada elemento
    sinphi = sqrt(sin2phi)
    kr = k1 * cos2phi + k2 * sin2phi

    # DwKr = C(w,w,w) + 2Kcot(theta)
```

```
    DwKr = u2 * (u*dcurv[:,0] + 3.0*v*dcurv[:,1])
         + v2 * (3.0*u*dcurv[:,2] + v*dcurv[:,3])

    K = k1 * k2

    cos2theta = ndotv^2
    sin2theta = 1 - cos2theta
    sintheta = sqrt(sin2theta)
    cot = ndotv / sintheta

    DwKr += 2.0 * K * cot

    return DwKr, kr
```

Notese que no calculamos realmente $cos(\phi)$ en esta implementación: para obtener $cos(\phi)$ deberíamos dividir por $||\mathbf{w}||$ ($\mathbf{w} \cdot \mathbf{e_1} = ||\mathbf{e_1}||\,||\mathbf{w}||\cos(\phi)$ y $||\mathbf{w}||$ no necesariamente es 1). Esta división resulta en valores teóricamente correctos, pero puede conducir a errores numéricos debido a cómo se representan los números en la computadora y a que $||\mathbf{w}||$ es en general chico.

Esta implementación causó errores en las imágenes generadas. Encontramos que la razón de estos errores era que $cos(\phi)$ en algunos vértices era ligeramente mayor a 1. Esto es causado por pequeños errores numéricos al momento de calcular $\mathbf{u}$, el producto escalar entre $\mathbf{w}$ y $\mathbf{e_1}$, y resultaba en que al calcular $sin(\phi)$ se obtuviesen errores por aplicar una raíz cuadrada a un número negativo. Como veremos más adelante, este error fue evitado en la implementación de *rtsc*, ya que no se calculan $cos(\phi)$ ni $sin(\phi)$. Para resolver este problema en nuestra implementación, evaluamos dos soluciones:

```
def computeDwKr(mesh):
    ...
    cosphi = u

    for elem in cosphi:
        if elem > 1.0:
            elem = 1.0
```

```
    cos2phi = cosphi^2
    sin2phi = 1.0 - cos2phi
    # Raiz cuadrada de cada elemento
    sinphi = sqrt(sin2phi)

    ...
```

```
def computeDwKr(mesh):
    ...

    # Norma de cada elemento
    cosphi = u / norm(w)
    cos2phi = cosphi^2
    sin2phi = 1.0 - cos2phi
    # Raiz cuadrada de cada elemento
    sinphi = sqrt(sin2phi)

    ...
```

En la primer solución acotamos los valores de $cos(\phi)$ al rango $[0, 1]$, lo cual nos permite ignorar artefactos y problemas numéricos. En la segunda solución, dividimos $\mathbf{u}$ por $||\mathbf{w}||$, lo cual involucra una operación adicional pero resulta en valores correctos de $cos(\phi)$ y del resto de cálculos que dependen de este. Optamos por combinar ambas soluciones, lo cual nos asegura que implementamos correctamente el algoritmo y evita los problemas numéricos mencionados.

```
def computeDwKr(mesh):
    ...

    # Norma de cada elemento
    cosphi = u / norm(w)
    cos2phi = cosphi^2
    sin2phi = 1.0 - cos2phi
```

```
    for elem in sin2phi:
        if elem < 0.0:
            elem = 0.0

    # Raiz cuadrada de cada elemento
    sinphi = sqrt(sin2phi)

    ...
```

Además de nuestras implementaciones basadas en *suggestive contours*, también implementamos la versión original del algoritmo de *rtsc* en Python. Esto fue útil para observar posibles diferencias en los resultados obtenidos sin tener diferencias en los lenguajes utilizados para implementar los algoritmos.

```
def rtscDwKr(mesh):
    # Calculo las propiedades necesarias
    ...

    # Computar n.v
    viewdir = viewpos - verts
    normalizar(viewdir)
    ndotv = dot(viewdir, normals)

    u = dot(viewdir, pdir1)
    v = dot(viewdir, pdir2)
    u2 = u^2
    v2 = v^2

    # Esto equivale a Kr * sin^2(theta)
    kr = k1 * u2 + k2 * v2

    # Calculo DwKr/||w|| * tan(theta)
    sctest = ( u2 * (u*dcurv[:,0] + 3.0*v*dcurv[:,1])
             + v2 * (3.0*u*dcurv[:,2] + v*dcurv[:,3]))
    csc2theta = 1/(u2 + v2)
    sctest *= csc2theta
```

```
    # A lo anterior le sumo 2 * K * cos(theta)
    tr = (k2 - k1) * u * v * csc2theta
    sctest -= 2.0 * ndotv * tr^2
    sctest /= ndotv

    return ndotv, kr, viewdir, sctest
```

Capítulo 5

Resultados

En este capítulo presentaremos los resultados de los algoritmos detallados en el capítulo anterior. Observaremos las distribuciones de valores y visualizaciones gráficas de los mismos sobre las mallas de los cuales fueron extraídos. Los modelos que utilizamos para generar estos resultados se pueden encontrar en https://gfx.cs.princeton.edu/proj/sugcon/models/.

5.1. Curvatura

Primero, compararemos los valores de curvatura principales (κ_1 y κ_2) resultantes de nuestra implementación *pyrtsc* con los obtenidos mediante *rtsc*. Para todas las visualizaciones de curvaturas se utilizó el mapa de colores "jet_r", el inverso del popular mapa "jet". Al utilizar un lenguaje y solver distinto se introducen pequeñas diferencias en la aproximación de las curvaturas entre *pyrtsc* y *rtsc*. Optamos por el solver más eficiente, que resultó ser *cholesky*, ya que los dos métodos no mostraron diferencias significativas en las métricas analizadas.

La Figura 5.1 muestra una comparación cualitativa entre los valores de κ_1 generados con *pyrtsc* y *rtsc*. Una comparación similar se muestra en la Figura 5.2 para κ_2. El análisis cuantitativo de los principales parámetros de las distribuciones en los datos pueden observarse en la Figura 5.3. Ambas implementaciones mostraron resultados casi idénticos.

En la Figura 5.3 podemos ver una comparación de las distribuciones de e_1 generadas utilizando *pyrtsc* y aquellas generadas por *rtsc*. Comparamos los parámetros θ y ϕ de las

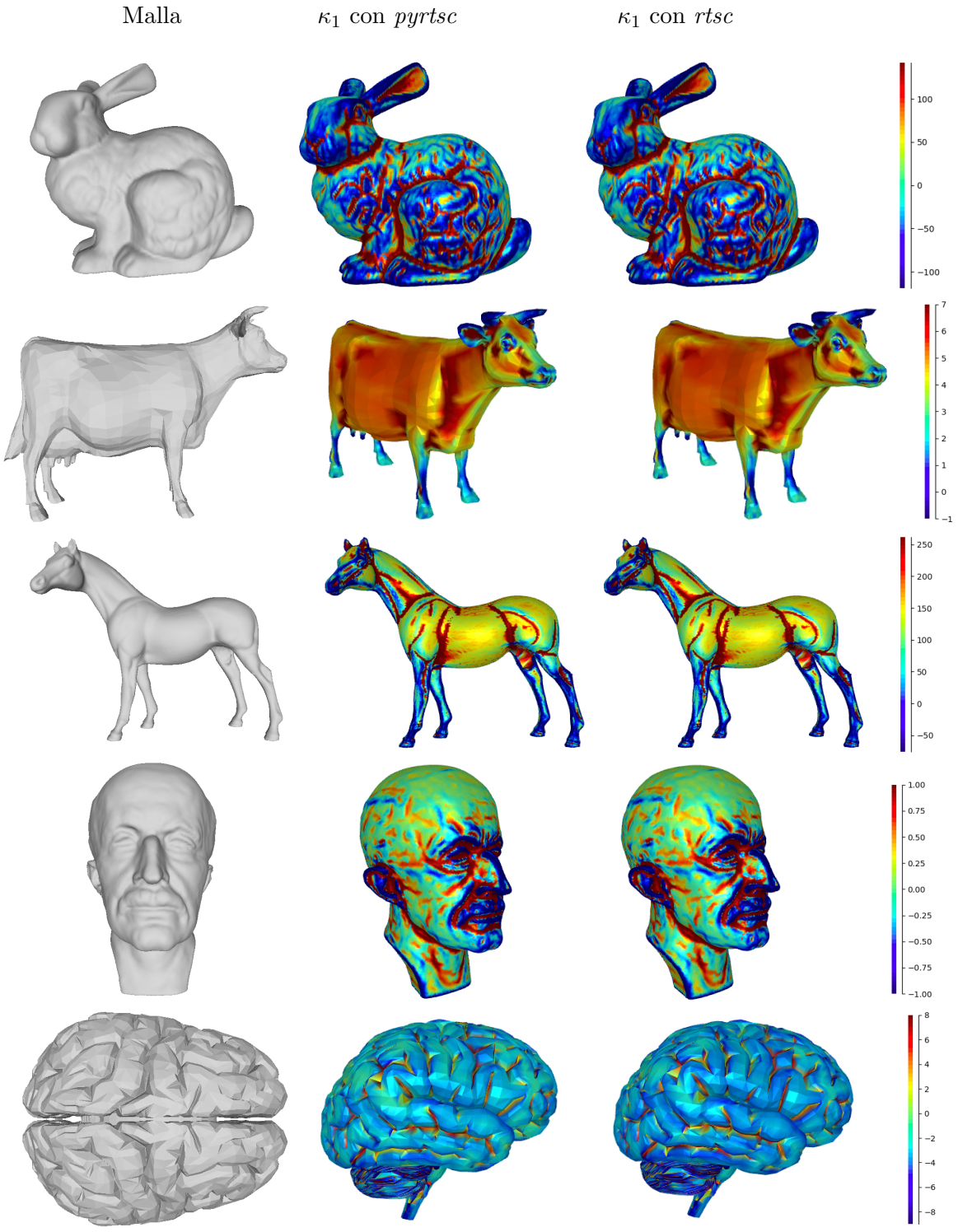


Figura 5.1: Visualización de κ_1 sobre las mallas. De izquierda a derecha: malla original con sombreado Lambertiano, κ_1 con *pyrtsc* y κ_1 con *rtsc*.

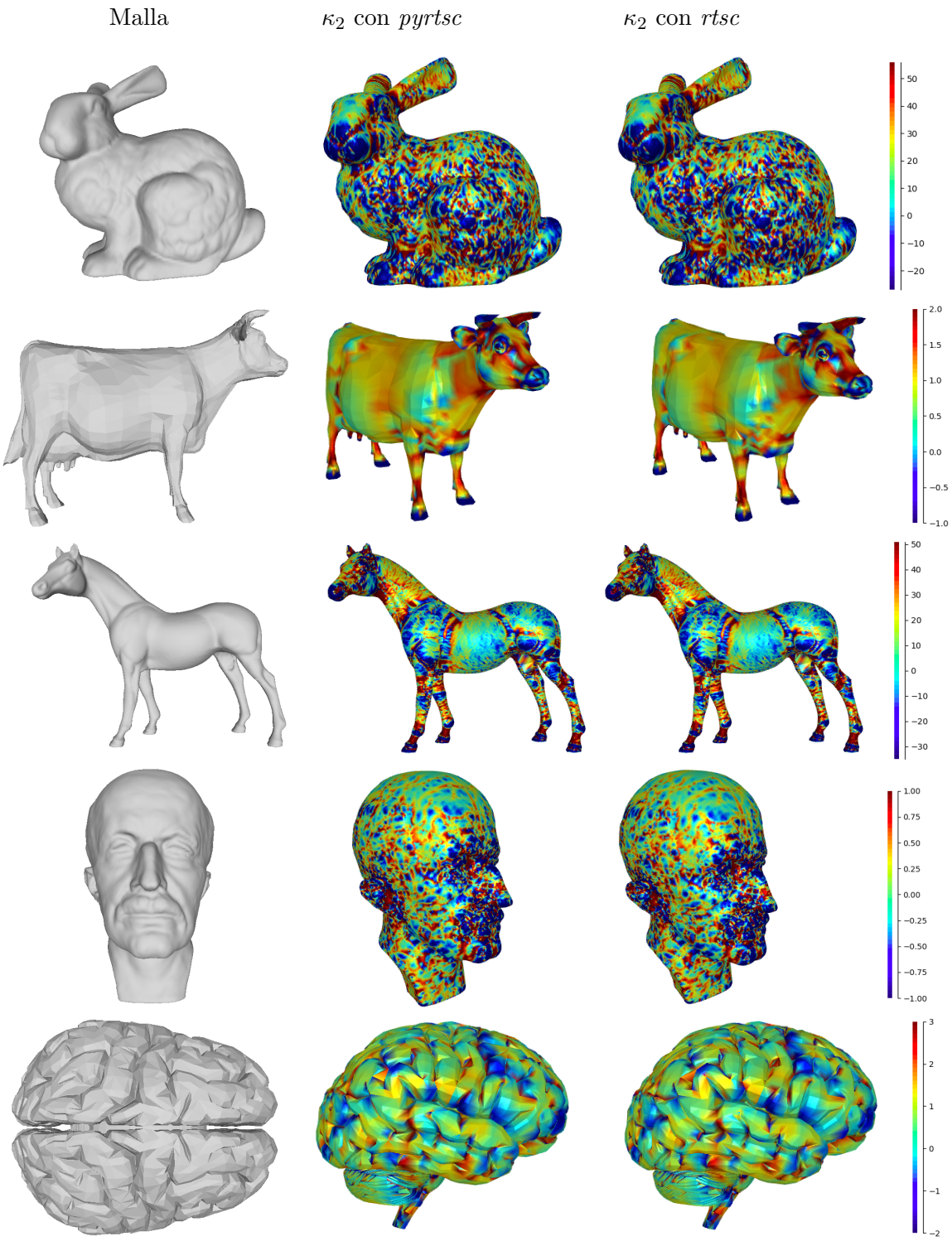


Figura 5.2: Visualización de κ_2 sobre las mallas. De izquierda a derecha: malla original con sombreado Lambertiano, κ_2 con *pyrtsc* y κ_2 con *rtsc*.

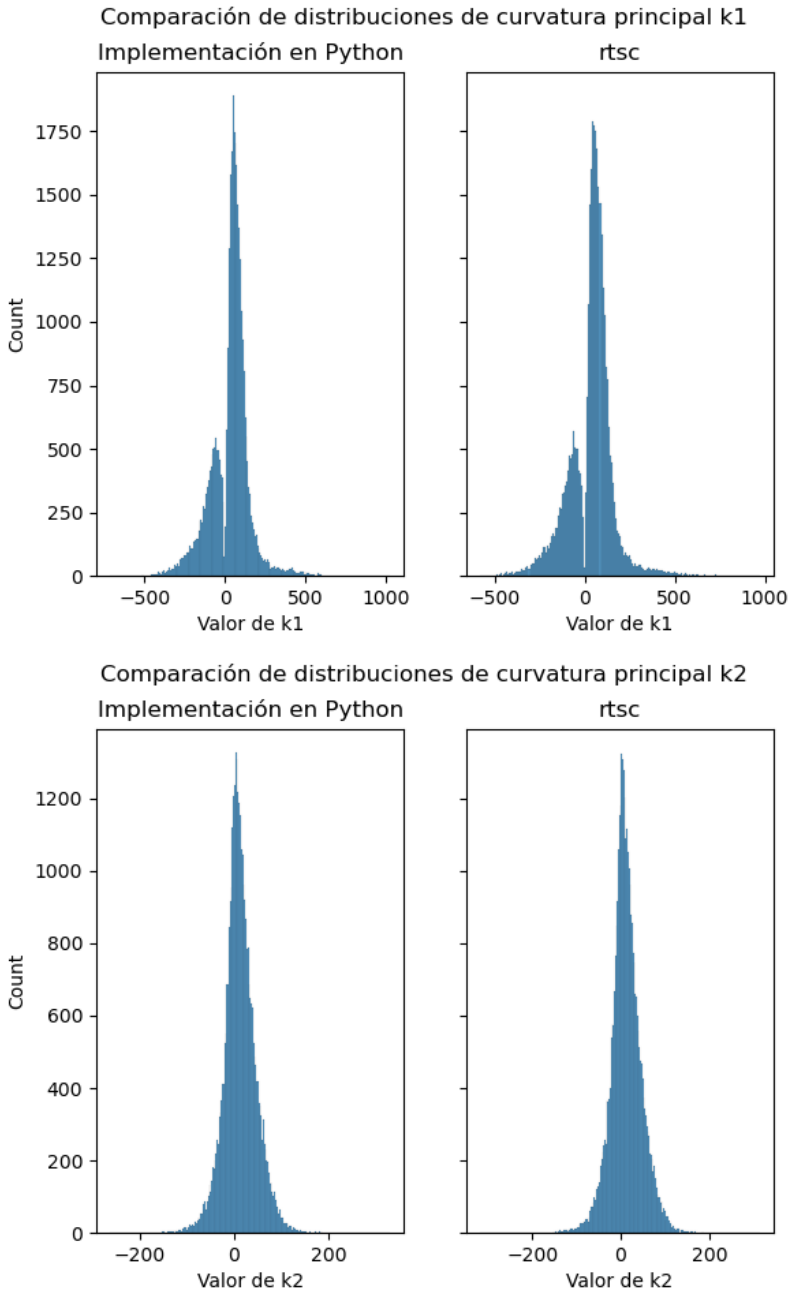


Figura 5.3: Distribuciones de κ_1 y κ_2 utilizando histogramas. Generadas desde la malla de conejo de la primer fila de la Figura 5.1

coordenadas esféricas de los vectores. No comparamos ρ porque e_1 y e_2 están normalizados. Es interesante notar que aunque las distribuciones de los ángulos de e_1 difieren entre las implementaciones, las derivadas de curvatura que se obtienen a partir de ellos no parecen diferir notablemente, como veremos en la siguiente sección.

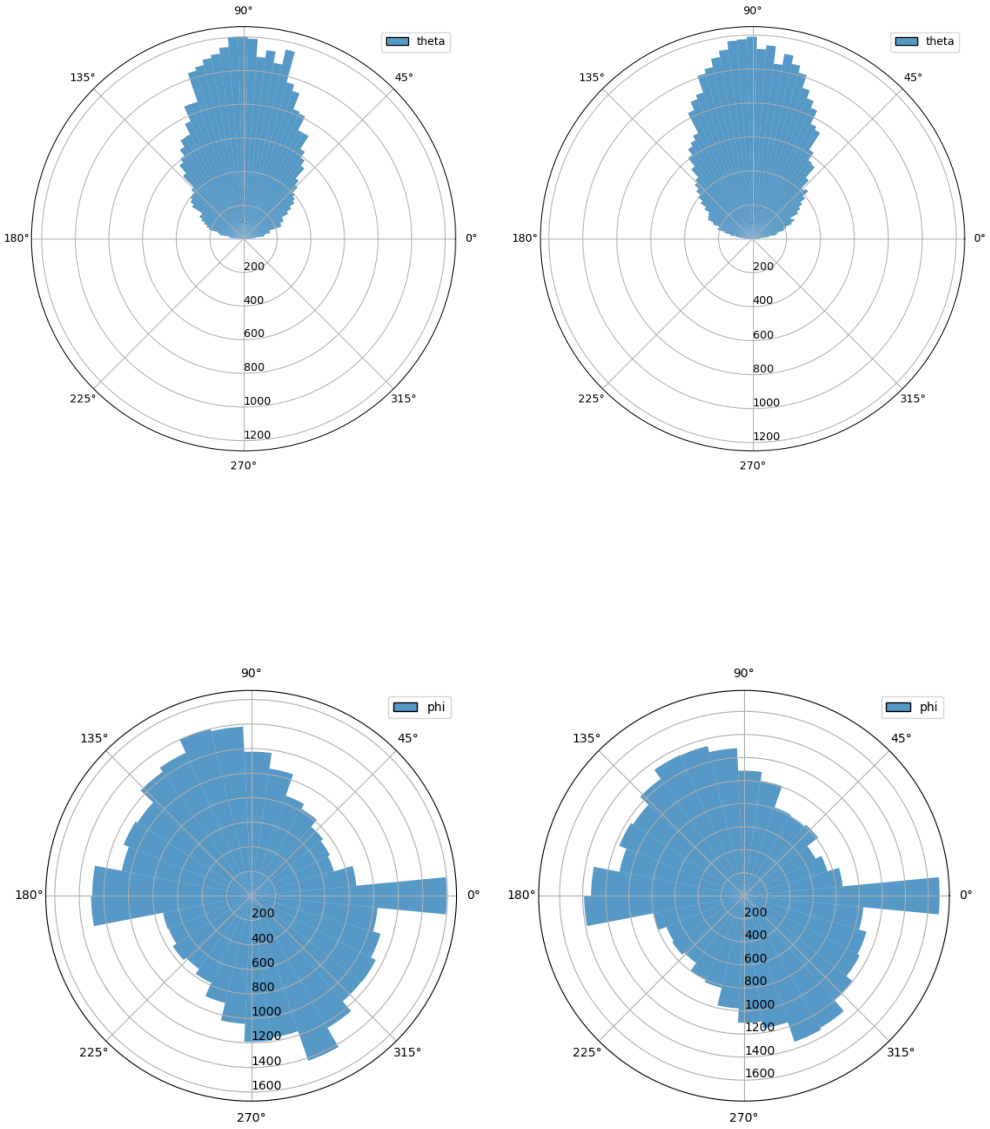


Figura 5.4: Distribuciones de e_1 utilizando histogramas polares. Calculamos las coordenadas de los vectores en coordenadas polares y graficamos θ y φ. Generadas desde la malla de conejo de la primer fila de la Figura 5.1.

5.2. κ_r y $D_{\mathbf{w}}\kappa_r$

Dado que la derivada de curvatura contiene 4 valores distintos, la comparación de resultados es compleja. Por consecuencia, decidimos comparar los resultados de κ_r y $D_{\mathbf{w}}\kappa_r$ para asegurar que el algoritmo esté generando valores correctos. En estos casos hacemos varias comparaciones: entre los valores generados con *pyrtsc* de principio a fin, los valores generados utilizando *pyrtsc* para generar κ_r y $D_{\mathbf{w}}\kappa_r$ en base a las derivadas de curvatura generadas con *rtsc*, y los valores generados por *rtsc* de principio a fin. Es importante recordar que *rtsc* no genera realmente κ_r y $D_{\mathbf{w}}\kappa_r$: ver el Capítulo 4. Por esta razón es importante hacer estas comparaciones, ya que los valores generados por *rtsc* contienen información adicional para descartar puntos de los contornos. Además, si empleamos nuestro algoritmo para calcular $D_{\mathbf{w}}\kappa_r$ con distintas derivadas de curvatura como parámetro, podemos observar las diferencias generadas por estas derivadas de curvatura. Esto nos permite detectar si nuestro algoritmo para generar derivadas de curvatura es incorrecto.

Las diferencias entre *pyrtsc* y *trimesh* debido a las diferencias entre los lenguajes y los solvers se vuelven más grandes al haber aplicado dos veces la minimización de cuadrados. Igual que antes, optamos por *cholesky* por ser el solver más eficiente.

Basándonos en el método utilizando en *Neural Contours* [11], acotamos los valores afuera del percentil 95º. Hacemos esto porque los cálculos de estos valores siempre resultan en algunos valores atípicos que no son estadísticamente relevantes. Debido a la forma en que se deben procesar los datos para visualizarlos con colores sobre la malla, estos valores extremos pueden causar distorsiones en la escala que imposibilitan ver las características más relevantes.

En la Figura 5.5 se pueden observar comparaciones entre las distribuciones de κ_r y $D_{\mathbf{w}}\kappa_r$ utilizando las derivadas de curvatura generadas *pyrtsc* y las derivadas de curvatura generadas por *rtsc*. En la Figura 5.6 vemos una comparación de los diagramas de caja de estos valores. Como podemos observar, aún acotando los valores extremos, tenemos muchos valores afuera de 1.5 veces el rango intercuartil. La presencia de estos valores atípicos es lo que causa los picos en los extremos de los diagramas de la Figura 5.5. En los algoritmos que trabajan sobre mallas de triángulos es común tener valores atípicos, ya que estos se pueden generar por situaciones comunes (como por ejemplo, que el vector de

vista esté perpendicular a la normal de un vértice).

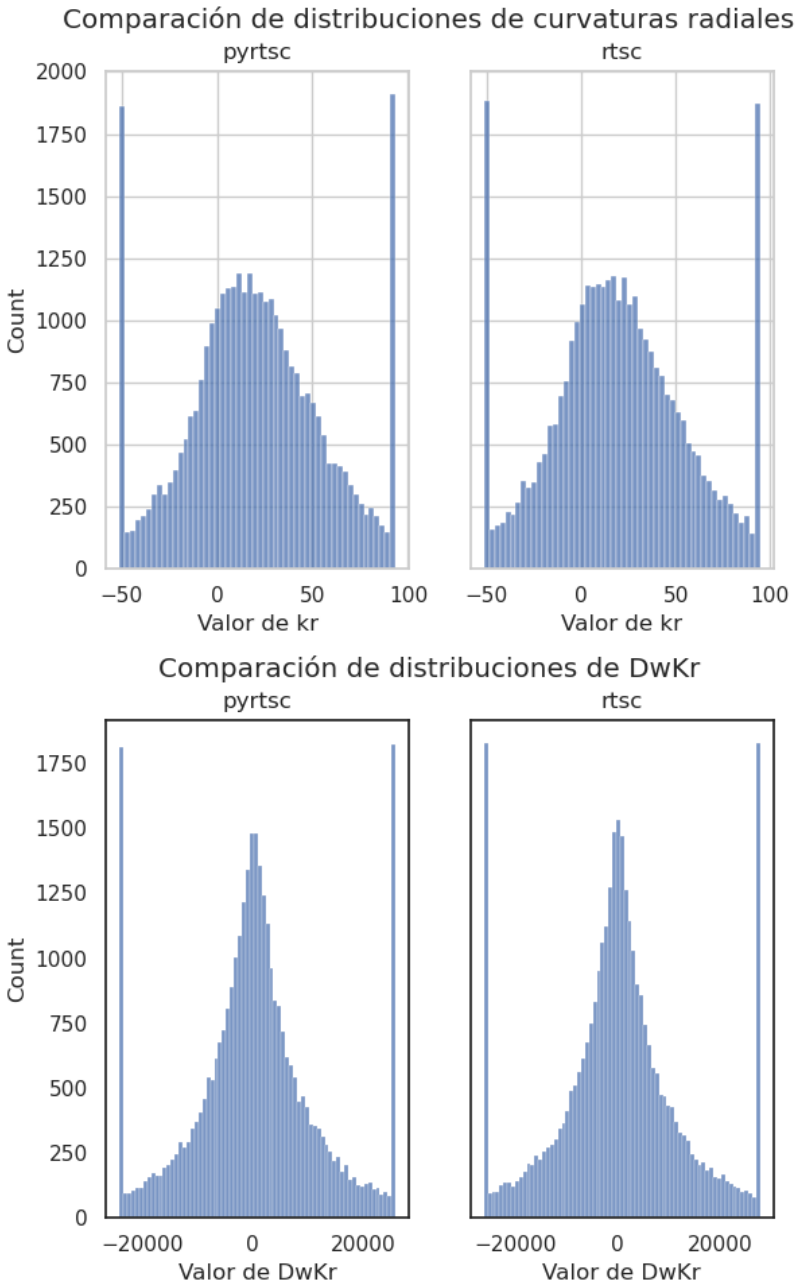


Figura 5.5: Distribuciones de κ_r y $D_{\mathbf{w}}\kappa_r$ utilizando histogramas. Generadas desde la malla del conejo de la primera fila de la Figura 5.1. Los picos de valores en los extremos se deben a cómo acotamos los valores al percentil 95º.

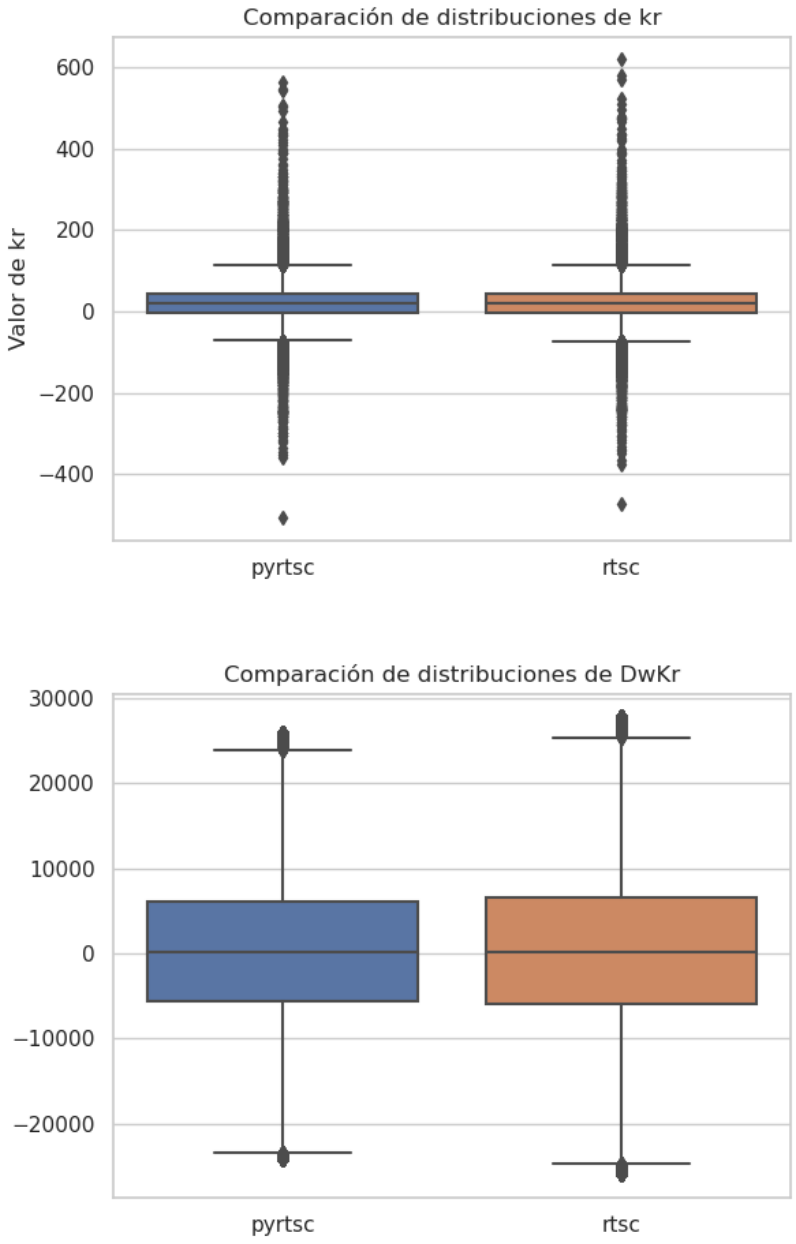


Figura 5.6: Distribuciones de κ_r y $D_{\mathbf{w}}\kappa_r$ utilizando diagramas de caja. Generadas desde la malla del conejo de la primera fila de la Figura 5.1. Al igual que en la Figura 5.5, podemos notar que las distribuciones son similares, aunque difieren ligeramente en el caso de $D_{\mathbf{w}}\kappa_r$. En el gráfico de $D_{\mathbf{w}}\kappa_r$ acotamos los valores al percentil 95º debido a que habían valores muy atípicos que dificultaban la comparación de las distribuciones.

Observando estas figuras, es claro que seguimos obteniendo distribuciones casi idénticas a las de *rtsc*. También podemos visualizar los valores en las mallas, igual que con κ_1 y κ_2 (Figuras 5.7 y 5.8). Todas estas visualizaciones y las comparaciones de distribuciones se generaron desde el punto de vista que se ve en las mallas con sombreado Lambertiano. Los modelos con colores se han rotado para facilitar la visualización.

En la Figura 5.9 encontramos las comparaciones de $D_{\mathbf{w}}\kappa_r$ que obtuvimos con los valores de *sctest*. En la Figura 5.10 podemos ver una comparación de las distribuciones de los valores de *sctest* utilizando un histograma. Como se puede apreciar en las visualizaciones de *sctest*, son mucho más claros los detalles de los modelos que con $D_{\mathbf{w}}\kappa_r$, y se pueden ver las regiones en donde se dibujarían los *suggestive contours*.

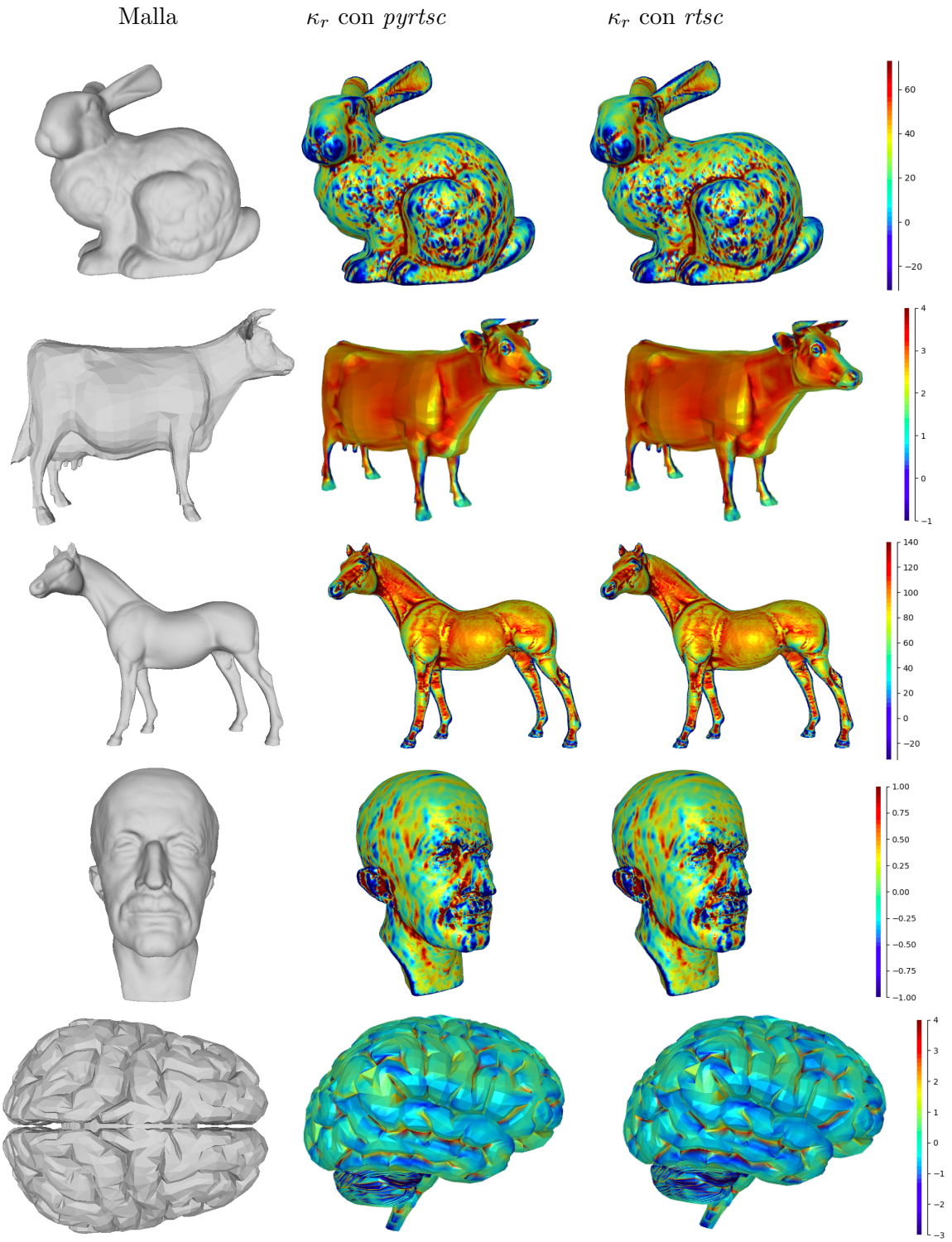


Figura 5.7: Visualización de curvaturas radiales sobre las mallas. De izquierda a derecha: malla original con sombreado Lambertiano, κ_r con *pyrtsc* y κ_r con *rtsc*.

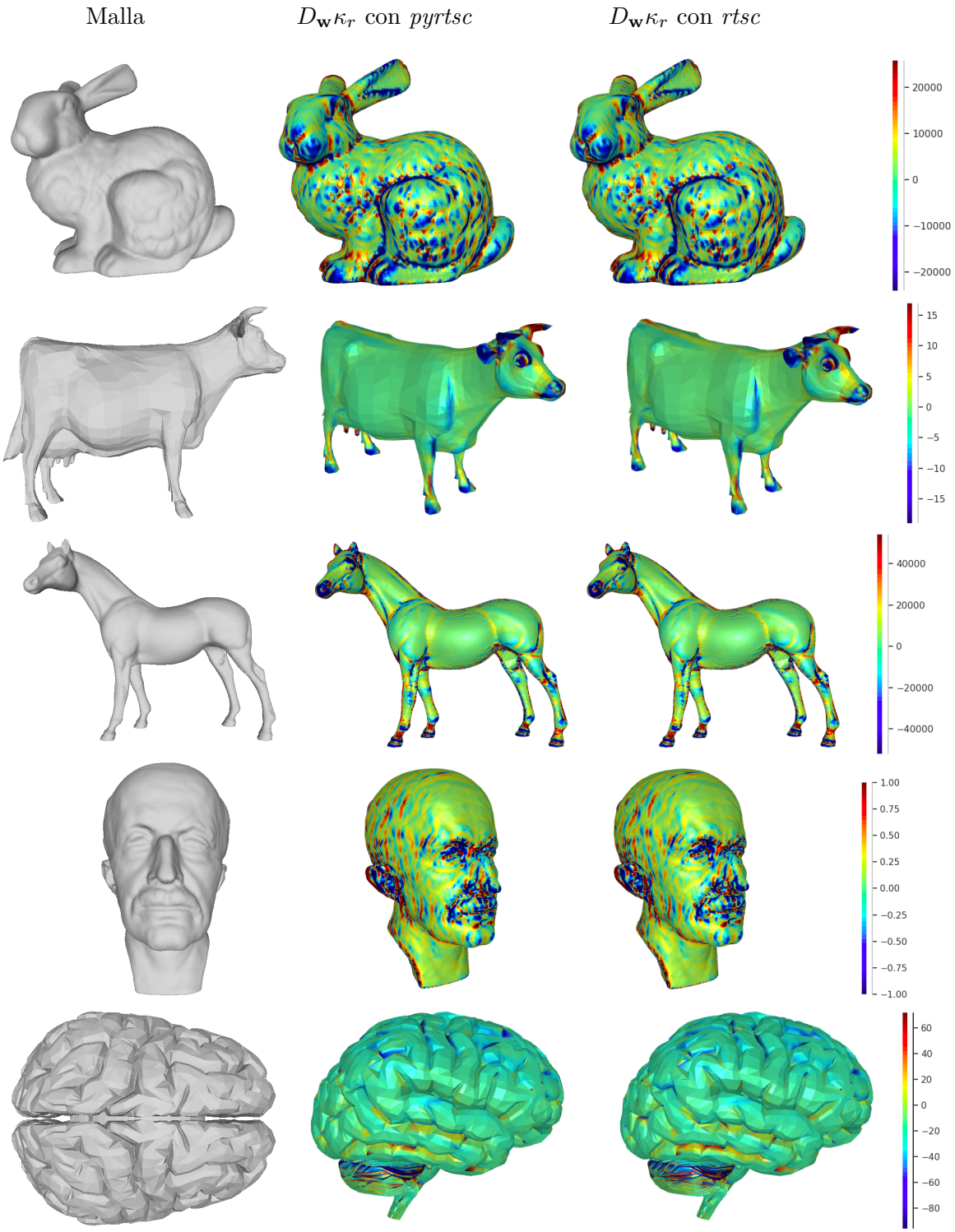


Figura 5.8: Visualización de $D_{\mathbf{w}}\kappa_r$ sobre las mallas. De izquierda a derecha: malla original con sombreado Lambertiano, $D_{\mathbf{w}}\kappa_r$ con *pyrtsc* y $D_{\mathbf{w}}\kappa_r$ con *rtsc*.

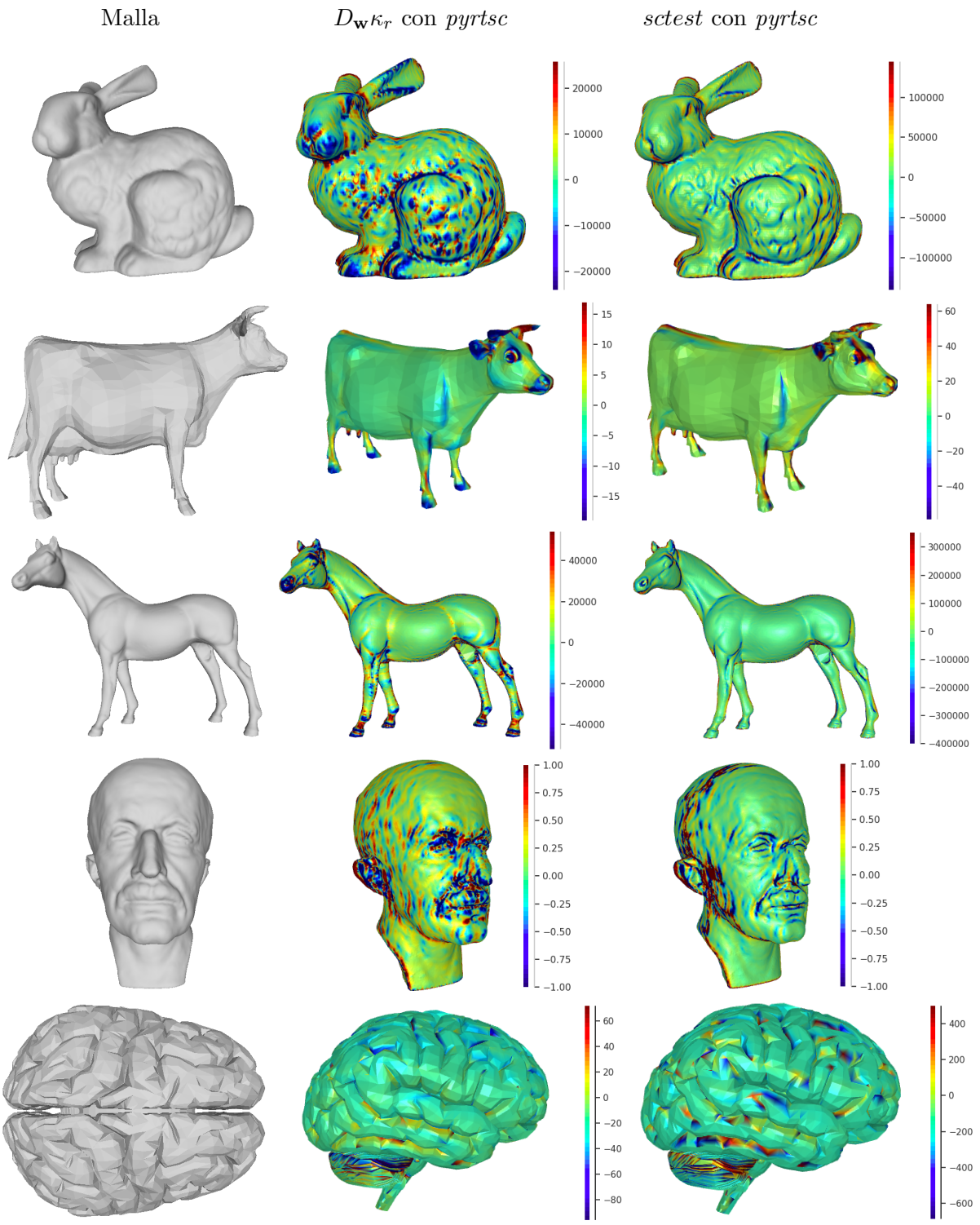


Figura 5.9: Visualización de $D_{\mathbf{w}}\kappa_r$ y *sctest* sobre las mallas. De izquierda a derecha: malla original con sombreado Lambertiano, $D_{\mathbf{w}}\kappa_r$ con *pyrtsc* y *sctest*. Podemos observar que en las visualizaciones de *sctest* se comienzan a notar los contornos de la malla.

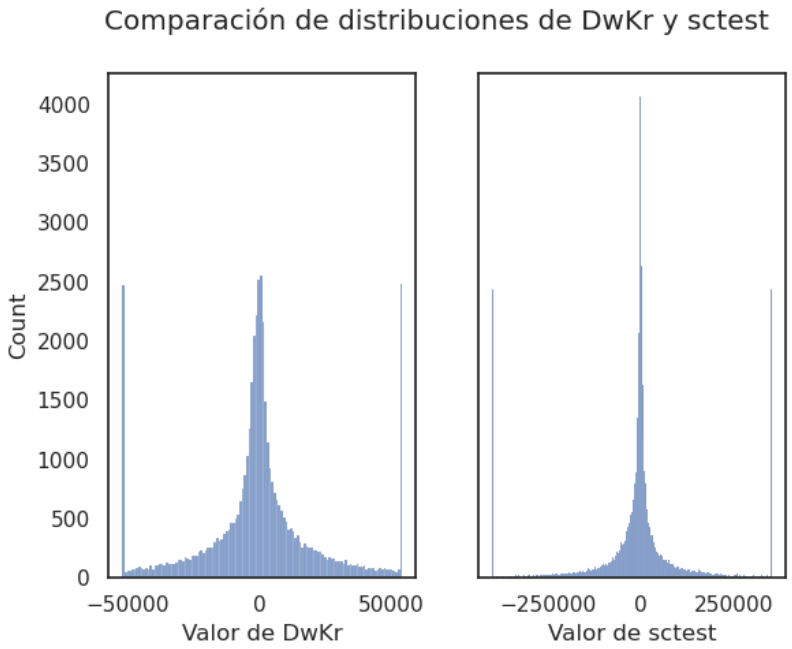


Figura 5.10: Distribuciones de $D_{\mathbf{w}}\kappa_r$ y *sctest* utilizando histogramas. Generadas desde la malla del caballo de la tercer fila de la Figura 5.1. Los valores fueron acotados al percentil 95º para facilitar la comparación. *sctest* es el valor que *rtsc* utiliza en lugar de $D_{\mathbf{w}}\kappa_r$ para descartar posibles puntos del *suggestive contour*, y es esperable que sea distinto de $D_{\mathbf{w}}\kappa_r$.

5.3. Comparaciones entre implementaciones de *pyrtsc*

En esta sección mostraremos comparaciones entre implementaciones de *pyrtsc* utilizando los métodos de resolución *lstsq* y *cholesky* de *PyTorch*. En la Figura 5.11 comparamos las velocidades promedio de los dos métodos utilizando modelos con distintas cantidades de polígonos y en la Figura 5.12 comparamos las distribuciones (sin acotar) de κ_1 y κ_2 generadas utilizando histogramas.

Es esperable que haya diferencias entre las aproximaciones de *rtsc* y *pyrtsc*, ya que se utilizan métodos numéricos distintos en lenguajes de programación distintos. Dado que las aproximaciones obtenidas con *torch.lstsq* y *torch.cholesky_solve* se encontraron suficientemente cercanas a aquellas hechas por *trimesh*, se decidió que no era necesario implementar un algoritmo alternativo de resolución de mínimos cuadrados. Como se puede observar, los dos algoritmos de PyTorch resultaron prácticamente idénticos, la decisión de cual utilizar se basó en los tiempos de cómputo. Debido a los tiempos obtenidos, decidimos utilizar el método de Cholesky.

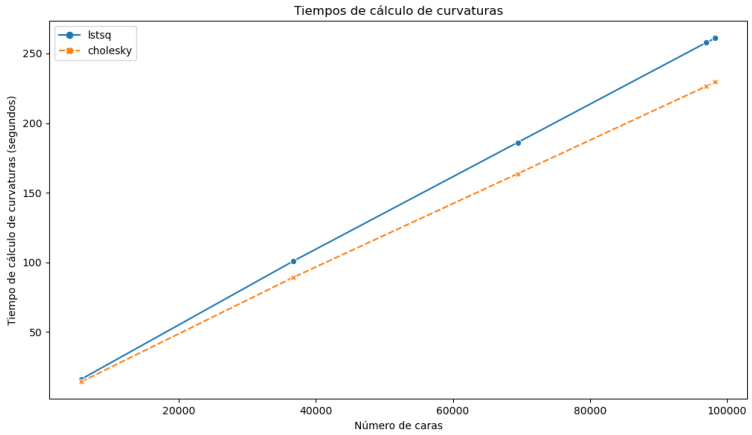


Figura 5.11: Comparación de velocidades del cálculo de curvaturas utilizando *lstsq* y *Cholesky*. Las mediciones fueron generadas utilizando las mallas de la Figura 5.1 y promediando los tiempos de 10 cálculos en cada malla. Como podemos ver, el método de minimización de cuadrados de Cholesky resultó en tiempos de cómputo menores para todo tamaño de malla.

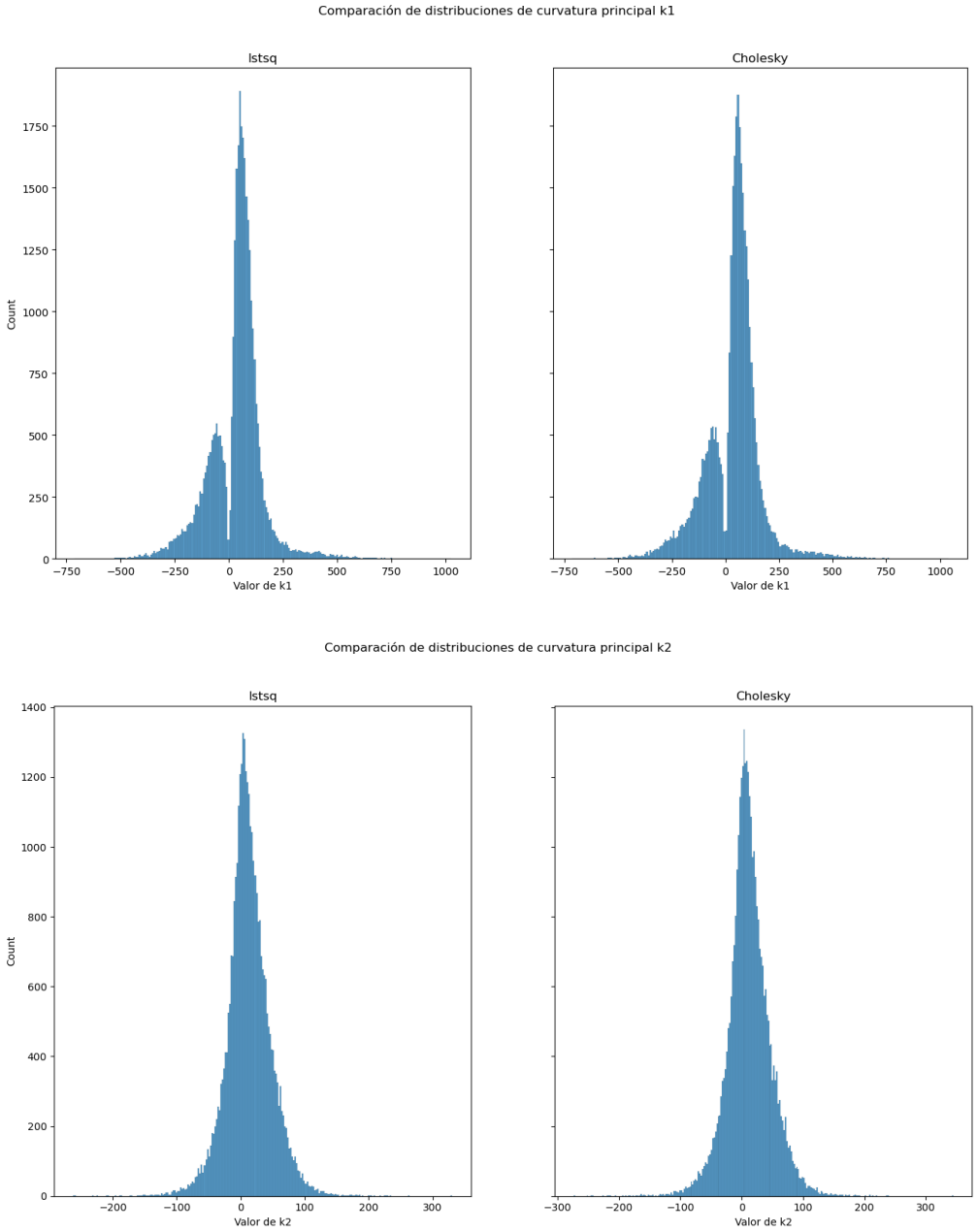


Figura 5.12: Distribuciones de κ_1 y κ_2 utilizando histogramas. Generadas desde la malla del conejo de la primer fila de la Figura 5.1. No notamos diferencias notables en las distribuciones, por lo cual nos basamos en las velocidades para elegir un método.

Capítulo 6

Conclusiones

En este trabajo reimplementamos algoritmos de extracción de propiedades geométricas sobre mallas triangulares. Además, implementamos un algoritmo propio para calcular derivadas direccionales de curvatura radial. Al implementar los algoritmos utilizando PyTorch, estas propiedades geométricas se pueden incorporar con más facilidad en un pipeline de aprendizaje automático. Al utilizar un lenguaje y frameworks distintos, era esperable que los resultados no sean exactamente iguales. Sin embargo, las distribuciones de los datos obtenidos demostraron ser muy similares.

Al comenzar este trabajo, no encontramos una librería actual en Python que implemente el cálculo de propiedades geométricas sobre mallas utilizando álgebra de tensores. Con nuestra implementación se pueden calcular normales, curvaturas, derivadas de curvatura y derivadas direccionales de curvatura radial. Esto posibilita la generación de bocetos utilizando Python.

Al comparar los valores y las implementaciones de $D_{\mathbf{w}}\kappa_r$ y *sctest*, concluimos que al incorporar la umbralización en *sctest* no solamente es más apto para el dibujo de *suggestive contours*, sino que además previene posibles problemas numéricos que surgen al implementar directamente las derivadas direccionales de curvatura. $D_{\mathbf{w}}\kappa_r$ es una propiedad menos especializada y posiblemente se pueda utilizar para otros algoritmos, pero para el dibujo de línea es recomendable emplear *sctest*.

Un posible camino de investigación basado en este trabajo es una comparación más amplia de distintos métodos de minimización de cuadrados y su impacto sobre las pro-

piedades calculadas. Como primer paso, propondríamos la reimplementación del método que utiliza *trimesh*, lo cual permitiría de comparaciones más directas con otros métodos ya existentes en la librería de PyTorch.

En esta tesis utilizamos las estructuras provistas por PyTorch (específicamente los tensores) para almacenar y computar todas las características geométricas de las mallas. También aprovechamos su integración con CUDA para acelerar el cómputo utilizando un GPU. A pesar de esto, el tiempo que tardan en completarse los cálculos sigue siendo sustancial, y esto indica que se pueden optimizar aún más la implementaciones actuales. Este sería un interesante trabajo a futuro basado en esta tesis.

Bibliografía

[1] Pierre Bénard y Aaron Hertzmann. Line drawings from 3d models. *arXiv preprint arXiv:1810.01175*, 2018.

[2] John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, 6:679–698, 1986.

[3] Keenan Crane. Discrete differential geometry: An applied introduction. *Notices of the AMS, Communication*, págs. 1153–1159, 2018.

[4] Doug DeCarlo, Adam Finkelstein, y Szymon Rusinkiewicz. Interactive rendering of suggestive contours with temporal coherence. En *Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*, págs. 15–145. 2004.

[5] Doug DeCarlo, Adam Finkelstein, Szymon Rusinkiewicz, y Anthony Santella. Suggestive contours for conveying shape. En *ACM SIGGRAPH 2003 Papers*, págs. 848–855. 2003.

[6] Manfredo P Do Carmo. *Differential Geometry of Curves and Surfaces*. Prentice-Hall, 1976.

[7] Gene H Golub y Charles F Van Loan. *Matrix computations. Johns Hopkins studies in the mathematical sciences*. Johns Hopkins University Press, Baltimore, MD,, 1996.

[8] Aaron Hertzmann y Denis Zorin. Illustrating smooth surfaces. En *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, págs. 517–526. 2000.

[9] Tilke Judd, Frédo Durand, y Edward Adelson. Apparent ridges for line drawing. *ACM transactions on graphics (TOG)*, 26(3):19–es, 2007.

[10] Evangelos Kalogerakis, Derek Nowrouzezahrai, Patricio Simari, James McCrae, Aaron Hertzmann, y Karan Singh. Data-driven curvature for real-time line drawing of dynamic scenes. *ACM Transactions on Graphics (TOG)*, 28(1):1–13, 2009.

[11] Difan Liu, Mohamed Nabail, Aaron Hertzmann, y Evangelos Kalogerakis. Neural contours: Learning to draw lines from 3d shapes. En *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, págs. 5428–5436. 2020.

[12] Steve Marschner y Peter Shirley. *Fundamentals of computer graphics*. CRC Press, 2018.

[13] Nelson Max. Weights for computing vertex normals from facet normals. *Journal of graphics tools*, 4(2):1–6, 1999.

[14] Mark Meyer, Mathieu Desbrun, Peter Schröder, y Alan H Barr. Discrete differential-geometry operators for triangulated 2-manifolds. En *Visualization and mathematics III*, págs. 35–57. Springer, 2003.

[15] S.E. Montesdeoca, H.S. Seah, H.-M. Rall, y D. Benvenuti. Art-directed watercolor stylization of 3d animations in real-time. *Computers & Graphics*, 65:60 – 72, 2017. ISSN 0097-8493. doi:10.1016/j.cag.2017.03.002. URL http://www.sciencedirect.com/science/article/pii/S0097849317300316.

[16] Krishna Murthy Jatavallabhula, Edward Smith, Jean-Francois Lafleche, Clement Fuji Tsang, Artem Rozantsev, Wenzheng Chen, Tommy Xiang, Rev Lebaredian, y Sanja Fidler. Kaolin: A pytorch library for accelerating 3d deep learning research. *arXiv*, págs. arXiv–1911, 2019.

[17] Pablo Navarro, José Ignacio Orlando, Claudio Delrieux, y Emmanuel Iarussi. Sketchzooms: Deep multi-view descriptors for matching line drawings. *arXiv preprint arXiv:1912.05019*, 2019.

[18] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. En *Advances in neural information processing systems*, págs. 8026–8037. 2019.

[19] Princeton Graphics Group. Suggestive contours. URL https://gfx.cs.princeton.edu/proj/sugcon/. Online; last accessed 4-December-2020.

[20] Szymon Rusinkiewicz. Estimating curvatures and their derivatives on triangle meshes. En *Proceedings. 2nd International Symposium on 3D Data Processing, Visualization and Transmission, 2004. 3DPVT 2004.*, págs. 486–493. IEEE, 2004.