Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Jaén

Trabajo Fin de Grado

Simulación de rotura de objetos frágiles con Blender y Python

Alumno: Pablo Pérez Peña

Tutor: Rafael Jesús Segura Sánchez

Antonio Jesús Rueda Ruiz

Dpto: Informática

UNIVERSIDAD DE JAÉN

D./Dª Rafael Jesús Segura Sánchez y D./Dª Antonio Jesús Rueda Ruiz, tutor(es) del Trabajo Fin de Grado titulado: **Simulación de rotura de objetos frágiles con Blender y Python**, que presenta Pablo Pérez Peña, autoriza(n) su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, septiembre de 2022

El estudiante

PEREZ PEÑA PABLO - 77689557L
Firmado digitalmente por PEREZ PEÑA PABLO - 77689557L
Fecha: 2023.01.19 22:35:14 +01'00'

Pablo Pérez Peña

Los tutores

SEGURA SANCHEZ RAFAEL JESUS - 26003853F
Firmado digitalmente por SEGURA SANCHEZ RAFAEL JESUS - 26003853F
Fecha: 2023.01.16 11:09:24 +01'00'

Rafael Jesús Segura Sánchez

Firmado digitalmente por RUEDA RUIZ ANTONIO JESUS - 33395227D
Fecha: 2023.01.19 22:04:25 +01'00'

Antonio Jesús Rueda Ruiz

Agradecimientos

A mis padres por apoyarme en mis estudios y sueños, tanto moral como económicamente, a enseñarme que nunca hay que doblegarse sin importar la dificultad.

A mis hermanos por darme tan buenos momentos en el mundo real y en el multiverso de los videojuegos.

A Marvel, a Pixar y a incontables estudios de animación, VFX y/o CGI por dar forma a dimensiones, personajes, universos y situaciones que no se podrían dar en el mundo real.

A la existencia de la animación japonesa o anime por sus historias únicas y sus estilos artísticos que tanto me gustan, a cuyo multiverso quisiera contribuir algún día.

A mis DJs y productores de EDM favoritos, cuya música me ha inspirado incontables escenas de acción alucinantes en mi cabeza, que algún día quisiera hacer realidad.

Tabla de contenidos

Lista de figuras

Lista de tablas

Lista de algoritmos

Lista de listados de código

Capítulo 1

INTRODUCCIÓN

1.1. Motivación

En los campos de generación de CGI (*Computer Generated Imagery, literalmente Generación de Imágenes por Ordenador*) o VFX (*Visual Effects, literalmente Efectos Visuales*) para medios de entretenimiento como películas, anuncios o series de animación 2D y 3D, un problema habitual en la simulación visual mediante computación de destrucción de objetos de distintos materiales (cristales, cerámicas, metales, hormigón, etc.), sobretodo para escenas de acción o combate. Este es también un problema habitual en el mundo de los videojuegos para el mismo tipo de escenas, ya que a día de hoy trabajar con gráficos 3D está a la orden del día (incluso se combinan ambas en algunos juegos con gráficos 2D).

Sin embargo también es interesante en otros ámbitos, como es el de la arqueología, ya que a día de hoy se siguen descubriendo restos de civilizaciones antiguas, incluyendo uno de los objetos frágiles más habituales pertenecientes a estas: las vasijas. Otrora recipientes para contener líquidos como agua, vino o aceite, ahora estas vasijas son objeto de estudio, reminiscencias sobre la forma de vida del ser humano en allendes tiempos lejanos. ¿Qué pudo romperlas antes de ser enterradas con el paso de cientos o miles de años? ¿Se cayeron de un almacén? ¿Fueron aplastadas por escombros? ¿O fue por acto de una piedra lanzada?

Este TFG (*Trabajo de Fin de Grado*) tratará de realizar una simulación de roturas de vasijas realista en distintas situaciones. Se han escogido vasijas ya que se contará con la supervisión de expertos en arqueología que juzgarán el realismo de las roturas virtuales.

1.2. Terminología

Definición	Descripción
VFX	Abreviación de *Visual Effects*, literalmente ”Efectos Visuales”. Técnicas de captura de imágenes que permiten recrear en formato de vídeo efectos o fenómenos que en el mundo real no existen, son difíciles de simular o su simulación podría provocar algún tipo de daño. No necesariamente requieren el uso de informática.
CGI	Abreviación de *Computer Generated Imagery*, literalmente ”Generación de Imágenes por Ordenador”. Técnicas de informática gráfica que generan imágenes para insertar objetos, personajes o fenómenos en medios de vídeo que de otra forma no sería posible o requeriría de más trabajo del necesario.
Stop-motion	Técnica de animación consistente en fotografiar objetos reales en posiciones sucesivas que, una vez combinadas, crean ilusión de movimiento.
Ident	Secuencia, generalmente de animación, en la que se muestra el logo de una compañía de televisión.
Motion Capture	Técnica para grabar movimientos reales mediante uso de medios computerizados para aplicarlos a modelos 3D.
Add-on	En *Blender*, término usado para designar a los *plug-ins* o funciones añadidas al software para realizar tareas personalizadas, que no se ofrecen de serie, o para facilitar una secuencia de trabajo.
Cursor 3D	En *Blender*, cursor que aparece en la ventana de vista 3D, representado como un círculo rojo y blanco con aspas. Este cursor se coloca de forma predeterminada en las coordenadas `(0, 0, 0)` de la escena, pero con la combinación de teclas `Shift + click derecho` se puede ajustar a la posición del ratón, y para devolverlo a la posición original se usa `Shift + C`. Los objetos 3D nuevos que se añadan a la escena se generarán en la posición 3D de este cursor.
Desarrollo incremental	Metodología de trabajo usada en desarrollo de software consistente en dividir en etapas la creación del código, empezando por una versión más básica y añadiendo en cada incremento implementaciones adicionales hasta tener el producto final.
Outliner	En *Blender*, panel que muestra una lista de todos los objetos de la escena, los cuales se pueden organizar en carpetas llamadas colecciones. Normalmente se encuentra la esquina superior derecha de la pantalla.

Tabla. 1.1: Tabla de definiciones

Capítulo 2

ANTECEDENTES

2.1. Estado del arte

2.1.1. Breve historia de los VFX

Los VFX están íntimamente relacionados con el cine, y el cine, a su vez (como indica Máxima Uriarte [2020]), nace de la fotografía, la cual nació en el siglo XIX. El cine, gracias a los avances de impresión de movimiento por la persistencia en la retina, nació poco después, siendo la primera proyección pública en 1895 en París por parte de los hermanos *Lumière*.

En aquellos primeros años la edición de vídeo no existía, así que los primeros intentos de edición se limitaban al uso de *Stop Motion* o de cortes, siendo uno de los trucos más fáciles el hacer ”desaparecer” objetos (como explica Serebrennikova [2020]), o pintar manualmente los fotogramas para crear efectos de luz o maquillaje, como era la cinta original de *El Viaje a la Luna* (1902). Alemania también fue pionera en el desarrollo de técnicas de efectos visuales, tal y como lo demostró *Metropolis* (1920).

A los largo del siglo XX la creación de VFX fue evolucionando constantemente, destacando las técnicas usadas en *Ciudadano Kane* (1941), por el uso de pinturas mate y miniaturas; *A vida o muerte* (1946), por ser una de las primeras película en usar el *Technicolor*; o *Los Pájaros* (1963), por el uso de rotoscopia. Pero la verdadera revolución llegó con *2001: Una odisea en el espacio* (1968) en especial por el uso de maquetas de gran escala para recrear naves espaciales. Dicha técnica fue también empleada y perfeccionada en *Star Wars: Episodio 4, Una Nueva Esperanza* (1977)

por parte de *Industrial Light And Magic* (*ILM*).

El CGI no nacería hasta la década de los 70 con la llegada de *Westworld* (1973), acompañado de los desarrollos tecnológicos en el campo de la Informática. En 1982, como afirma al principio de la tercera parte del artículo de Stout [2013], el canal *Channel 4* estrenó su primer *ident* creado exclusivamente con CGI, algo que a día de hoy se ha normalizado completamente.

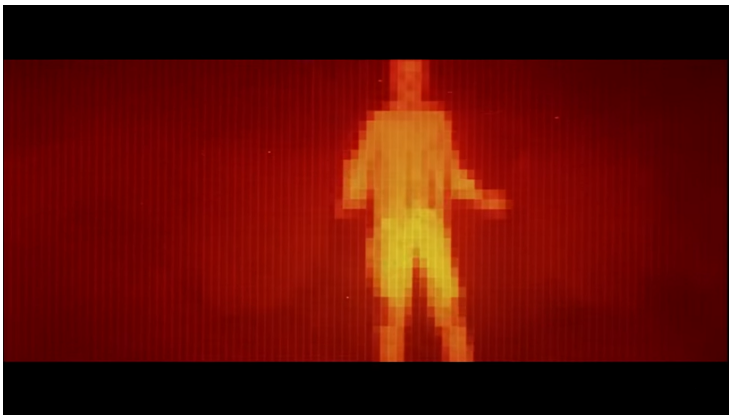

Figura 2.1: Fotograma de la película *Westworld* (1973) que combina imagen real con CGI

En los 80 se fue poco a poco perfeccionando el arte y la tecnología del CGI empezando con *Tron* (1982) y *Star Trek II: La ira de Khan* (1982) con sus técnicas para crear escenas. Poco después se crearía el primer personaje creado por ordenador en *Young Sherlock Holmes: El secreto de la pirámide* (1985), y *Abyss* (1989) generaría por primera vez agua por métodos informáticos.

En los años 90 aparecen los primeros indicios de tecnología de captura de movimiento (*Motion Capture*), con *Desafío Total* (1990), luego mejorado unos 20 años después; así como la creación del *workflow* actual para la generación de efectos especiales usado actualmente, la cual se introdujo en *Contact* (1997). Además, Pixar estrenaría en 1995 la primera película realizada enteramente en ordenador: *Toy Story*.

A partir del siglo XXI se produce la actual proliferación del uso del CGI en la industria del cine. A partir del 2000 se introduce el uso de los famosos cromas verdes y a partir de 2010 el CGI alcanza el realismo impresionante que actualmente vemos con asinuidad en las más recientes películas. Algunos de los ejemplos actuales más destacados por uso o innovación en el campo de VFX y/o CGI son la trilogía del *Señor de los Anillos*, *Avatar* (2009), *El Hobbit: Un viaje inesperado* (2012) o *The Mandalorian* (2019).

2.1.2. Creación de VFX y CGI

Software de creación de VFX

A fecha de realización de este TFG y según Moore [2022], aquí se resume el software más utilizado para VFX.

- **After Effects**: Software de pago por suscripción de la serie Adobe, y uno de los más empleados en la industria profesional. Completo e integrado con el editor de vídeo Premiere, y con posibilidad de añadir *plug-ins*.
- **HitFilm**: Posee dos versiones: Express y Pro. La primera es un editor de vídeo gratuito con expansiones o *plug-ins* de pago. La segunda es una versión mucho más ampliada con muchas más herramientas para creación de VFX, y también es usada en la industria profesional, pero es de pago y no muy asequible económicamente.
- **Blackmagic Fusion**: Software profesional de calidad industrial cuya versión estándar está disponible de forma gratuita. Con posibilidades de combinar secuencias reales con modelos 3D, trabajar con VR y usar estereoscopía 3D. Integrado con el editor Da Vinci Resolve.
- **Apple Motion**: Software exclusivo para dispositivos Mac de la compañía Apple. Precio asequible pero ofrece menos posibilidades que los anteriores y el hecho de ser exclusivo de un sistema operativo que no está al alcance de todo el mundo lo hace bastante limitado.

Software de creación de CGI

También tenemos de parte de Haggard [2022] esta otra lista, actualizada a fecha de creación del TFG, de software de creación de CGI, de la cual se citarán solo aquellas para generación de imágenes 3D.

- **Bryce**: Software de pago muy usado para CGI, con varias versiones y de precio asequible.
- **Lightwave 3D**: Software de pago mucho más caro que Bryce pero incluye muchas más herramientas.

- **Blender**: Software escogido para el desarrollo de este TFG por a su fácil accesibilidad, ya que es software libre de código abierto disponible completamente gratis. Ampliamente usado. Incluye herramientas de modelado, animación, rigging, renderizado, scripting...

- **Houdini**: Software también muy popular para CGI especializado en generación procedural, usado en estudios profesionales. Posibilidades de modelado, animación, simulación de físicas y partículas, etc. Tiene una versión de pago para uso comercial y otra gratuita para uso no comercial.

- **Cinema 4D**: Software muy avanzado y popular pero extremadamente caro. Usa un sistema de tomas único que lo diferencia de otras aplicaciones, y además de lo esperado (más ampliado que las anteriores) permite integración con Houdini.

- **Autodesk Maya**: Aunque no viene en la citada lista, es una mención honorífica al ser el software de este tipo más empleado en la industria profesional cinematográfica y de animación al estar orientado al trabajo realizado en estas[1]. Cuenta con multitud de herramientas de animación y modelado; sin embargo, su subscripción anual es muy cara [2].

- **Autodesk 3D Studio MAX**: También de la misma compañía que Maya, Autodesk, es otro de los software de generación de gráficos 3D más conocidos y empleados. También cuenta con muchas herramientas, pero aunque se usa también para cine y animación, está más orientado a la generación 3D de arquitectura y al desarrollo de videojuegos. También, desafortunadamente, solo se puede adquirir mediante una subscripción cara [3].

- **Unreal Engine**: Mención especial, ya que es un motor de desarrollo de videojuegos, pero su potencia es tal que se ha usado mucho en los más recientes proyectos cinematográficos, sobretodo para generación de escenarios en tiempo real, como se hizo en el filme de *The Mandalorian* (2019). Por si fuera poco está disponible gratuitamente, y permite programar tanto en C++ como con un sistema visual basado en nodos conocido como *Blueprint*[4].

[1] https://www.sculpteo.com/en/3d-learning-hub/3d-printing-software/3dsmax-vs-maya/
[2] https://www.autodesk.es/products/maya/overview
[3] https://www.autodesk.es/products/3ds-max/overview
[4] https://www.unrealengine.com/es-ES

2.2. Algoritmos de fragmentación

Un algoritmo de fragmentación descompone un objeto, generalmente en 3D, en fragmentos más pequeños con los cuales se puede reconstruir el objeto original. Algunos permiten crear roturas en tiempo real, lo cual es muy útil para videojuegos en los que se rompen objetos frágiles y/o necesitan un toque de realismo convincente. Son interesantes porque aunque son algoritmos de carácter aleatorio, el animador poseerá un cierto grado de control sobre los fragmentos, ideal para *VFX* en series o películas (tanto de imagen real como completamente animadas por ordenador), ya que las secuencias deben ajustarse a las necesidades del director y/o del storyboard.

Existen previos estudios de físicas de materiales y mecánica para abordar el problema de la rotura de objetos frágiles, dando lugar a los siguientes tipos de algoritmos:

- **Algoritmos que calculan a la perfección la rotura de estos materiales a través de fórmulas dadas por la física**. Sin embargo, a cambio de su fidelidad, estos algoritmos son de una complejidad computacional tal que tardan horas o incluso días en realizar los cálculos, dejándolos impracticables.

- **Algoritmos basados en sistemas dinámicos complejos**, que tratan de dispersar las roturas en vez de calcular de forma realista los fragmentos. Se centran solamente en que visualmente las roturas sean convincentes.

- **Algoritmos semejantes a los anteriormente mencionados que sacrifican la corrección física y la dispersión en pos de un menor consumo computacional sin perder calidad de realismo**.

- **Algoritmos basados en técnicas procedurales y descomposición de Voronoi**. Renuncian a los fundamentos físicos pero simulan fracturas reales, consiguiendo simplicidad, velocidad de ejecución (casi en tiempo real) y un grado de control sobre la estructura y la fragmentación. Los algoritmos realizados para este TFG se corresponden a esta categoría, por lo que expandiremos su funcionamiento más abajo.

Existen otros algoritmos que emplean otros métodos, como el de Glondu et al. [2012], basado en análisis modal (definir la fragmentación en base a la geometría del objeto y realizar una aproximación medianamente realista de la creación de la fractura) y propagación de fragmentos, eficiente y apto para aplicaciones interactivas.

2.2.1. Algoritmos de fragmentación basados en Voronoi

La fragmentación basada en Voronoi, tal y como la describe Raghavachary [2002], consiste en la creación de un diagrama de Voronoi en la que se distribuirán semillas entorno al volumen de la malla con las que luego se triangula.

Un **diagrama de Voronoi**[5], conocido también como telesación de Voronoi o polígonos de Thiessen, es una partición geométrica de un plano euclídeo dada a partir de una serie de puntos que generan regiones. Dada la distancia entre dos puntos, denotada como:

$$||p - q|| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2} \tag{2.1}$$

Sea $P = p_1, p_2, ..., p_n$ el conjunto de n puntos en el plano, que denominaremos *sitios*. Se define el diagrama de Voronoi de P como la subdivisión del plano en n regiones, una para $p_i \in P$, en la que un punto q solo pertenece a una región p_i si y solo si $||q - p_i|| < ||q - p_j||, \forall p_j \in P, j \neq i$.

Existen varios algoritmos de generación de estos diagramas, pero para nuestro caso necesitamos un método que no requiera un coste computacional muy prominente. Los algoritmos de este tipo primero realizan una triangulación y luego realizando un ajuste (*clipping*) a los bordes del plano. El resultado en una división del plano en regiones con forma de polígonos convexos tales que un vértice cualquiera de la división pasa exactamente por 3 puntos generadores. Esta propiedad hace el uso de esta teselación interesante para conseguir un resultado realista.

Una vez obtenida esta primera división, se calculan roturas a partir de estos polígonos, tarea que se puede realizar con varios métodos, por ejemplo: crear segmentos a partir de las aristas que forman las áreas. Las roturas pueden ser controladas por un animador o también se pueden hacer fragmentaciones recursivas siguiendo este método.

2.2.2. Llevando la fragmentación a entornos 3D

Aunque el diagrama de Voronoi sea un diagrama 2D, es perfectamente aplicable al 3D. Algoritmos como la solución proporcionada por Müller et al. [2013] parten del *Volumetric Approximate Convex Decomposition* (VACD) (literalmente *Aproximación Volumétrica de Descomposición Convexa*), permiten realizar roturas de objetos 3D incluso

[5] Polígonos de Thiessen, en *Wikipedia*. 8 de octubre de 2022. https://es.wikipedia.org/w/index.php?title=Pol%C3%ADgonos_de_Thiessen&oldid=146460032

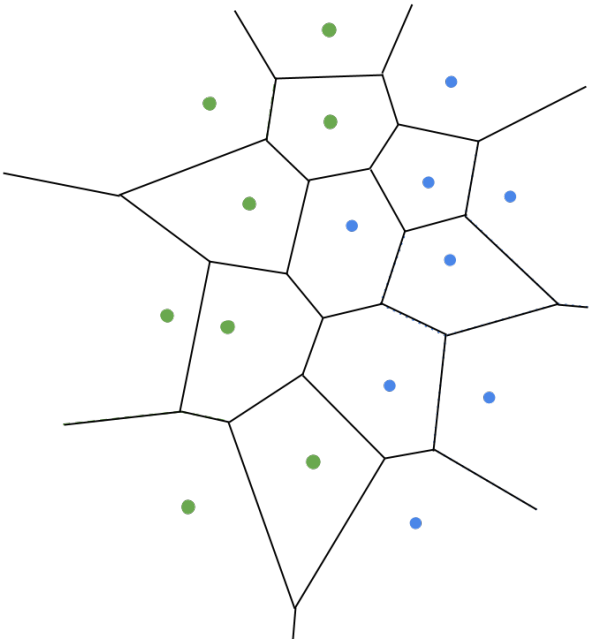

Figura 2.2: Ejemplo de Diagrama de Voronoi (*imagen cedida bajo licencia Creative Commons*)

en tiempo real a partir de los denominados *patrones de fractura*. Los modelos 3D aplicables deben de cumplir las siguientes propiedades:

- Deben de estar compuestos de piezas convexas.
- Las piezas no se solapan entre sí.
- Dos piezas son conexas si al menos un lado de una pieza se solapa parcialmente con la de otra y dichos lados son paralelos a un mismo plano y tienen normales opuestas.

El algoritmo de Müller et al. [2013] sigue el siguiente procedimiento:

1. **Alineación *(Alignment)***: Partiendo de una malla divisoria como la del Diagrama de Voronoi y de un objeto 3D también dividido en piezas convexas que no se solapan, la malla se coloca en una posición dada y se ajusta un punto de fragmentación. Se pueden hacer transformaciones adicionales para un mejor ajuste.
2. **Intersección *(Intersection)***: Se calculan las intersecciones entre el objeto 3D y la división. Recomendable el uso de estructuras de datos que aceleren este proceso.
3. **Soldadura *(Welding)***: Paso de optimización que une entre sí las piezas rodeadas completamente por otras.

4. **Formación del compuesto *(Compound Formation)***: Si un conjunto de piezas se situaran en la misma celda de la malla divisora, se fusionarán todas ellas. Se puede usar un algoritmo de coloración, como los que se usan para colorear grafos.

5. **Detección de islas *(Island detection)***: Último paso que arregla posibles errores del anterior. Fusiona piezas fragmentadas que están alejadas del punto de fragmentación y forman parte de las división original del modelo 3D (es decir, convierte islas inconexas en islas conexas).

La metodología de Blender para fracturas, así como el algoritmo de Müller et al. [2013], emplean el uso de la **envolvente convexa** como apoyo para posicionar la malla de separación. Para estos algoritmos nos conviene emplear un método eficiente, por lo que emplean una **envolvente convexa aproximada** que tiene una menor carga de memoria que una envolvente convexa habitual y se construye más eficientemente. Kavan et al. [2006] realiza la siguiente definición formal: sea $CH(A)$ como la envolvente convexa del objeto A, tal que contiene el origen de coordenadas, es la intersección de todos los *espacios entre medias* del objeto. A tendrá k espacios que dividirán el plano en sectores centrados en el origen tal que:

$$\begin{gathered} S_i = \{x \in R^2 \ \arctan 2(x) \in (\alpha i, \alpha(i+1))\} \\ s_i = S_i \cap A \end{gathered} \tag{2.2}$$

Tal que $\alpha = \frac{2\pi}{k}$ e $i = 0, ..., k-1$. Los *espacios entre medias*, que denominaremos como H_i, se definen de la siguiente forma:

$$\begin{gathered} H_i = \{x \in R^2, (n_i, x) \leq o_i\} \\ n_i = (\cos(\alpha i + \alpha/2), \sin(\alpha i + \alpha/2)) \\ o_i = max(n_i, x) \forall x \in A \end{gathered} \tag{2.3}$$

Tal que n_i son los vectores normales del origen de cada H_i y o_i la distancia de separación de cada H_i con A en la dirección de n_i. Con esto, llamaremos a la envolvente convexa aproximada de k espacios tal que:

$$ACH_k(A) = \bigcap_{i=0,...,k-1} H_i \tag{2.4}$$

La envolvente convexa aproximada será convexa por definición debido a que es la intersección de todos los *espacios entre medias* y a que una envolvente convexa

habitual estará contenida en esta:

$$CH(A) \subseteq ACH_k(A) \tag{2.5}$$

En resumen, partiendo de un punto interior del objeto al que denominaremos origen, crearemos k vectores en distintas direcciones y mediante intersecciones entre estos vectores y los segmentos que forman los puntos del objeto, se obtendrá la envolvente convexa aproximada. Kavan et al. [2006] explica también que el algoritmo será $O(NK)$ asumiendo que $|A| = N$, y que puede ser mejor o peor que el algoritmo clásico $O(NlogN)$ dependiendo del parámetro k. Dicho trabajo detalla también una versión mejorada del algoritmo en tiempo lineal, pero no entraremos en detalle.

2.3. Distribuciones de probabilidad

Una vez explicado el estado del arte y los fundamentos de los algoritmos, pasemos al ejemplo concreto de recrear una rotura de material con software de VFX. Se observa que la rotura de dos objetos frágiles en el mundo real, incluso bajo las mismas condiciones, nunca serán iguales, esto quiere decir que la fragmentación es un **suceso aleatorio**. Este suceso varía en función de parámetros tales como:

- **Las propiedades físicas del material**: fragilidad, dureza... Si imaginamos la destrucción de la pared de un edificio con una ventana, los materiales hechos de hormigón generarán fragmentos más grandes y poligonales, mientras que los fragmentos de cristal de la ventana serán más pequeños y abundantes, y de forma triangular.
- El **tipo de impacto** que ha recibido y su **origen**. Por regla general, se generarán fragmentos más pequeños y abundantes en las zonas del impacto que en las zonas restantes.

La **probabilidad** es una variable de rango [0, 1] que es asignada a un conjunto (eventos). Matemáticamente se expresa como:

$$P(A) = x, 0 \leq x \leq 1 \tag{2.6}$$

Denominaremos **evento** a un suceso en el que el resultado no se decida de forma

determinista. Para nuestro caso, denominaremos así al suceso de qué fragmentos generará un objeto frágil al romperse.

Tenemos un tipo concreto de evento que son las **variables aleatorias**. Como indica la sección Variables Aleatorias[6] de Gil Bellosta [2021], se denotan con letras mayúsculas por razones históricas, y pueden tomar valores cuantitativos o cualitativos.

Es posible, a veces de forma implícita, razonar una probabilidad mediante el uso de argumentos epistemológicos, como hace el **principio de indiferencia**, consistente en que si un suceso tiene n posibilidades, cada una de esas alternativas tiene una probabilidad $1/n$ de suceder. Este principio se corrobora en la teoría de la probabilidad con la **ley de los grandes números**, que indica que repetir un experimento infinitamente, las probabilidades obtenidas se irán aproximando a una constante, que para casos como los de lanzar un dado o una moneda, se aproximan a $1/6 = 0,16$ y $1/2 = 0,5$ respectivamente.

Sin embargo, no todos los sucesos tienen una probabilidad $p(A) = 1/n$ de que se produzcan, tal y como se produce al tirar un dado trucado cuyo peso interior hace que la cara contraria a la que se apoya la mayor parte de ese peso sea la cara más probable.

Las **distribuciones aleatorias** (también conocidas como funciones aleatorias) estudian eventos reales con respecto a los eventos generados por variables aleatorias. Consisten en funciones en base a una variable aleatoria X en la que la variable real x describe el comportamiento de la variable. Formalmente hablando, la función aleatoria F es una función tal que para la variable aleatoria discreta X, $P(X = x) = F(x)$. Cada variable tiene su correspondiente función aleatoria.

Históricamente se ha observado que numerosos eventos aleatorios no relacionados entre sí ofrecen comportamientos o patrones similares. Por ello surge el **Principio de universalidad**: se demuestra que aquellas variables aleatorias que cumplan determinadas propiedades puedan aproximarse mediante una distribución de probabilidad concreta. Es por ello que podemos generar una fragmentación realista si elegimos la función adecuada para la situación. Sin embargo, Gil Bellosta [2021] nos recuerda que no todos los eventos aleatorios siguen siempre una distribución, y que esta distribución será una aproximación, no una ley determinista.

Ahora pasaremos a explicar las distribuciones individualmente, agrupadas en dos categorías: discretas y continuas. También mencionaremos sus funciones de *Python*

[6] https://datanalytics.com/libro_estadistica/variables-aleatorias.html

correspondientes, todas extraídas de la biblioteca *Numpy*[7], que se agrupan en la categoría `Numpy.random`. Los histogramas, usados para distribuciones discretas, han sido generados con la biblioteca de *PythonMatplotlib*[8], mientras que para las funciones continuas se ha ayudado del módulo de funciones estadísticas *stat* de la biblioteca matemática *Scipy*[9].

2.4. Distribuciones de probabilidad discretas

En las distribuciones discretas la variable aleatoria solo puede tomar unos valores determinados. A continuación se detallarán algunas de las distribuciones a usar para la resolución de este problema.

2.4.1. Distribución de Poisson

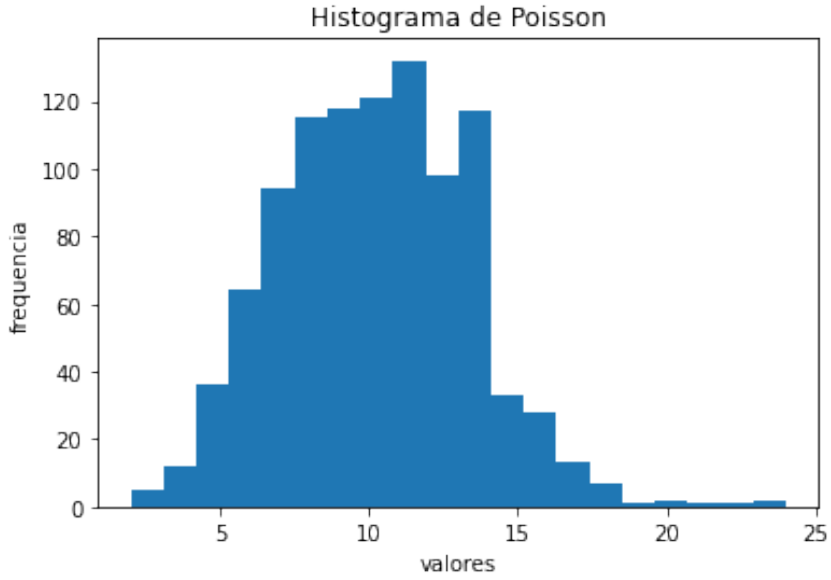

Figura 2.3: Histograma de distribución de Poisson para un valor $\mu = 10$

Una de las distribuciones más empleadas en Estadística, creada por Siméon-Denis Poisson (1781-1840), matemático y filósofo que contribuyó a la creación de la ley de los grandes números. Su uso general es el de modelar las pérdidas operacionales en un evento. Sigue la fórmula:

[7] https://numpy.org/doc/stable/index.html
[8] https://matplotlib.org/2.0.2/index.html
[9] https://docs.scipy.org/doc/scipy/reference/stats.html

$$p(r, \mu) = \frac{\mu^r e^{-\mu}}{r!} \tag{2.7}$$

- μ: También conocido como intensidad. Es un número positivo que representa la media del número de eventos a ocurrir. Para la función del paquete numpy de *Python*, solo necesita conocer este valor.
- r: Número natural positivo que representa los valores discretos posibles ($r \leq 0$).

Como indica Rodó [2020], para que una variable aleatoria siga esta distribución tiene que cumplirse que:

- La variable r debe representar el número de ocurrencias de un suceso durante un intervalo.
- Las ocurrencias han de ser aleatorias y de probabilidad equitativa.
- Las ocurrencias deben estar distribuidas uniformemente durante el intervalo.

Con esta distribución los puntos se agrupan en una zona concreta, por lo que sería útil para calcular roturas producidas por un golpe en un punto de la vasija.

2.4.2. Distribución binomial

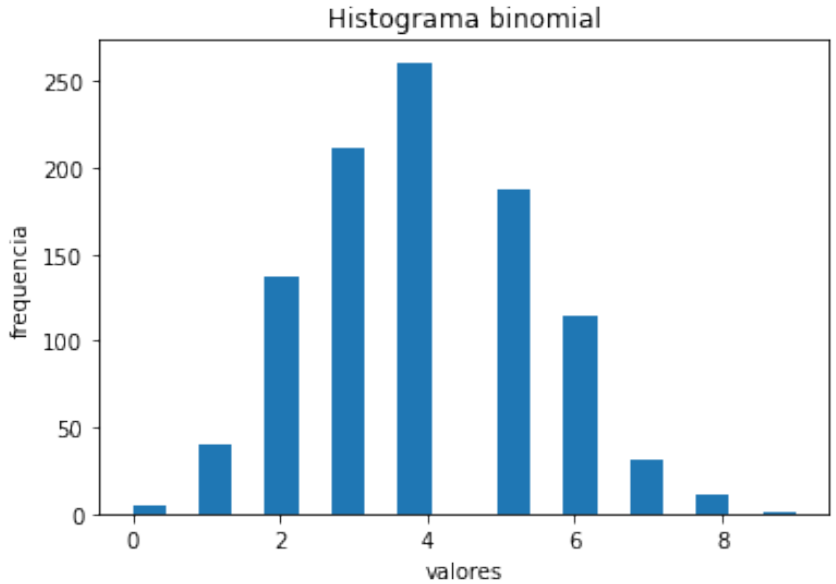

Figura 2.4: Histograma de distribución binomial para valores $n = 10, p = 0{,}4$

La distribución binomial consiste en la suma de variables aleatorias de Bernoulli, variables que pueden tomar dos valores: éxito o fracaso (semejantes a las variables booleanas usadas ampliamente en Informática). Esta distribución es usada para modelar experimentos en los que solo existen dos posibles resultados: lanzar una moneda varias veces, acertar que se cumple o no una característica en un número dado de individuos, etc.

Una **variable aleatoria de Bernoulli** X la define formalmente Mendoza [2002] como:

$$\begin{aligned} X : \mathcal{S} &\rightarrow \mathcal{R} \\ Exito &\rightarrow 1 \\ Fracaso &\rightarrow 0 \end{aligned} \tag{2.8}$$

La función aleatoria de una variable de Bernoulli se denota como:

$$p(x) = p^x(1-p)^{1-x}, x = \{0, 1\} \tag{2.9}$$

Y la fórmula de la distribución binomial es:

$$p(r; N, p) = \binom{N}{r} p^r(1-p)^{N-r} \tag{2.10}$$

Las variables usadas son:

- N: Número de ensayos o experimentos. $N \geq 0$ (Variable a introducir en funciones de *Python*)
- r: Número de éxitos. $0 \leq r \leq N$ (Variable a introducir en funciones de *Python*)
- p: Probabilidad de éxito.

Según Sanjuán [2017], esta distribución funciona bien siempre y cuando se cumplan las siguientes propiedades:

- Los posibles resultados han de ser éxito o fracaso (como se ha explicado con las variables aleatorias de Bernoulli); es decir, deben de ser mutuamente excluyentes y colectivamente exhaustivos: cada experimento tendrá de resultado uno de los dos posibles valores pero no ambos o ninguno.
- La probabilidad de éxito y fracaso han de ser constantes.

- Los resultados de cada experimento son independientes: el resultado anterior no influirá en el siguiente o en los posibles sucesivos.
- Se debe cumplir $p \cdot n > 5$ para que los resultados sean fiables.[10]

Probablemente con esta distribución los fragmentos generados tengan una distribución relativamente regular debido a que concentra las frecuencias en unos cuantos determinados valores.

2.4.3. Distribución geométrica

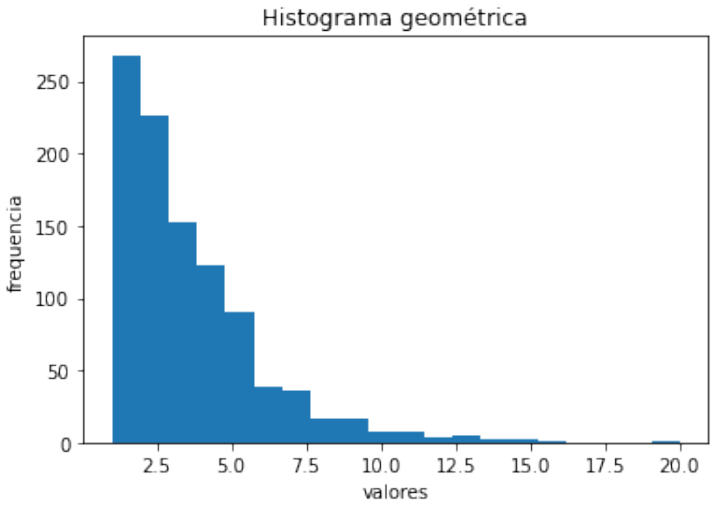

Figura 2.5: Histograma de distribución geométrica para un valor $p = 0{,}3$

La distribución geométrica define la probabilidad de que un número de muestras se consiga un acierto en base a una probabilidad de éxito. Sigue la fórmula:

$$p(r; p) = p(1 - p)^{r-1} \tag{2.11}$$

Sus variables son:

- r: Define el número de pruebas a evaluar. Número real donde $r \leq 1$.
- p: Probabilidad de acierto. Número real donde $0 \leq p \leq 1$. (La función de *Python* solo necesita esta variable)

[10] *numpy.random.binomial. Numpy API reference Ver. 1.23 (Stable)* https://numpy.org/doc/stable/reference/random/generated/numpy.random.binomial.html#numpy-random-binomial

Holmes et al. [2022] define las propiedades que cumple un experimento geométrico:

- Se realiza el experimento, produciendo en el proceso uno o más fallos, hasta que se halla un acierto.
- La probabilidad de acertar (p) o fallar (q) es siempre la misma para todo el experimento: $p + q = 1$ y $q = 1 - p$.
- $\exists X =$ número de fallos independientes hasta producir un acierto.

Las frecuencias se agrupan más cerca de los menores valores, por lo que se teoriza que agruparía los fragmentos en la base de la vasija.

2.5. Distribuciones de probabilidad continuas

Estas distribuciones de probabilidad toman valores decimales en vez de enteros; en otras palabras (como dice Lopez Briega [2016]), puede tomar cualquier valor dentro de un valor específico. Se representan mejor gráficamente mediante funciones de densidad de probabilidad. La mayoría de las distribuciones usadas para este TFG pertenecen a esta categoría.

2.5.1. Distribución uniforme

Afirma Gil Bellosta [2021] que esta es la más sencilla de las distribuciones de esta categoría. Se trata de una distribución simétrica y equiprobable en la que la probabilidad será 0 en todos los valores salvo los del rango $[a, b]$, que se puede expresar como:

$$p(x; a, b) = \frac{1}{b-a}, \textit{ para } a \leq x \leq b \tag{2.12}$$

Su función de probabilidad será entonces.

$$F(x; a, b) = \begin{cases} 0 & , \textit{ para } x \leq a \\ \frac{1}{b-a} & , \textit{ para } x \in [a, b] \\ 1 & , \textit{ para } x \leq b \end{cases} \tag{2.13}$$

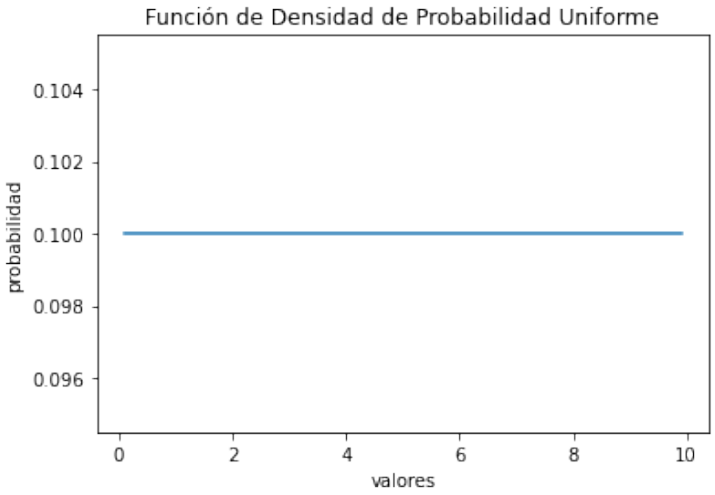

Figura 2.6: Función de densidad de probabilidad de distribución uniforme para el rango $[0, 10]$

Posee también una versión discreta[11], de función de probabilidad:

$$F(k; a, b) = \frac{\lfloor k \rfloor - a + 1}{b - a + 1} \tag{2.14}$$

Se trata de una distribución interesante, ya que la mayoría de generadores de número pseudoaleatorios intentan muestrear un valor entre $[0, 1]$. Y en efecto, algunas distribuciones muestrean valores de ese rango, como la Distribución logarítmica. Para el caso de las vasijas, muy seguramente distribuya las roturas de forma equilibrada por toda la superficie del objeto.

Para la función correspondiente en *Python* solo se necesitan los valores de a y b.

2.5.2. Distribución normal

También conocida como Distribución de Gauss, es la distribución más usada en Estadística. Su fórmula viene dada por:

$$p(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-1}{2}(\frac{x-\mu}{\sigma})^2} \tag{2.15}$$

Variables:

- μ: Media, o según otros autores, parámetro de ubicación o valor central. Modifica

[11] Distribución uniforme discreta, en *Wikipedia*. 21 de octubre de 2021. https://es.wikipedia.org/w/index.php?title=Distribuci%C3%B3n_uniforme_discreta&oldid=139187791

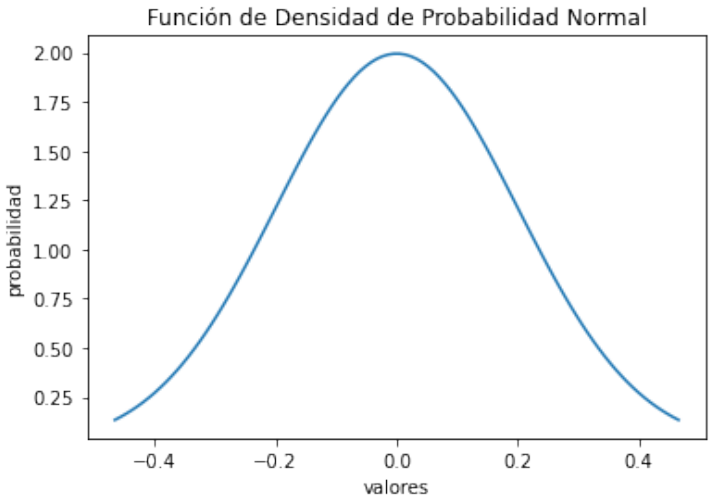

Figura 2.7: Función de densidad de probabilidad de distribución normal para los valores $\mu = 0, \sigma = 0{,}2$

el valor máximo que alcanza.

- σ^2: Desviación típica o estándar. Modifica el rango de valores que toma.

Rodó [2019a] enumera las dos propiedades de esta distribución:

- Es una **distribución simétrica**. Sus valores de media, moda y mediana son equivalentes.
- Es una **distribución equimodal**. Como se puede inferir de la función de probabilidad, la mayor probabilidad se concentra en la media, y disminuye al alejarse de esta.

Su función de probabilidad genera la denominada *Campana de Gauss*. En otras palabras, concentra los valores en el centro, por lo que esta función concentraría las fracturas en un punto central.

2.5.3. Distribución exponencial

Según Holmes et al. [2022], esta distribución normalmente se asocia al tiempo que tarda en producirse un evento. En una variable aleatoria exponencial, hay menos valores grandes y más valores pequeños (pongamos un ejemplo: muchas personas harán una pequeña contribución a un proyecto y pocas harán una gran contribución). Esta variable aleatoria es continua y, en efecto, mide el paso del tiempo, por lo que se

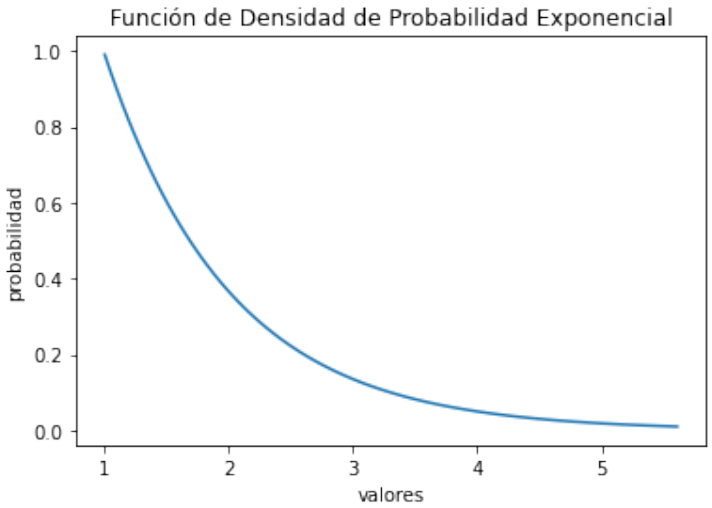

Figura 2.8: Función de densidad de probabilidad de distribución exponencial para el valor $\alpha = 1$

usa para estudiar acciones a lo largo del tiempo, como el tiempo en el que un tema en redes sociales es popular. Su función viene dada por una sencilla ecuación:

$$p(x;\alpha) = \frac{1}{\alpha} e^{\frac{-x}{\alpha}} \tag{2.16}$$

La única variable necesaria es α o tiempo promedio de espera. La media y varianza corresponden a $1/\alpha$.

Al igual que la distribución geométrica, su función concentra los puntos en los valores más pequeños, resultando probablemente en una distribución concentrada en la base de la vasija.

2.5.4. Distribución beta

Generalización de la distribución uniforme, que también toma valores en el rango $[0, 1]$. Emplea el uso de dos valores: α y β, que se emplean en la **Función Beta** para modelar la forma de la probabilidad en un rango finito de dos valores en presencia de incertidumbre. Su función de probabilidad es:

$$p(x;\alpha,\beta) = \frac{1}{B(\alpha,\beta)} x^{\alpha-1}(1-x)^{\beta-1} \tag{2.17}$$

La función beta $B(\alpha, \beta)$ equivale a:

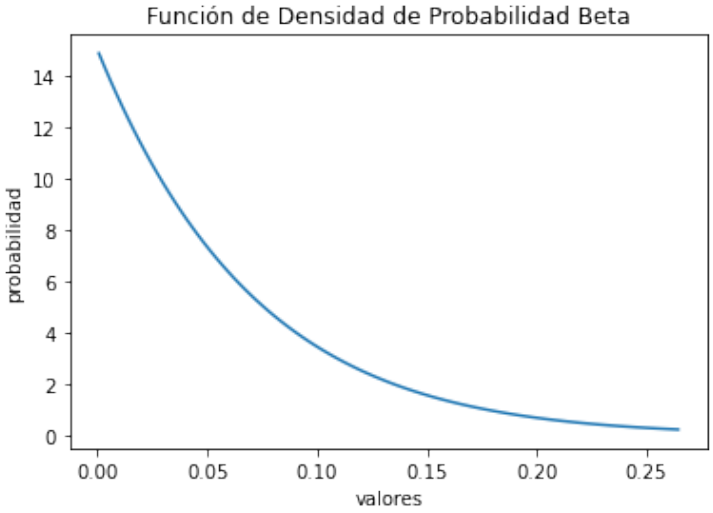

Figura 2.9: Función de densidad de probabilidad de distribución beta para los valores $\alpha = 1, \beta = 15$

$$B(\alpha, \beta) = \int_0^1 x^{\alpha-1}(1-x)^{\beta-1}dx \tag{2.18}$$

Para $p(x; 1, 1)$, la probabilidad tendrá un valor constante. Para $(\alpha, \beta) \in [2, \infty]$ tendrá forma de curva. Si fijamos uno de los dos valores y variamos el otro, la curva de su respectiva función de densidad se irá moviendo a la derecha, pero dependiendo de cuál escojamos su "aparente movimiento" será distinto dependiendo si variamos o α o β y dejamos constante el restante.

Probablemente sea una distribución muy interesante para generar fragmentos de diversas formas si se ajustan los valores de α y β, ya que la curva tiene un mayor grado de ajuste que las anteriores distribuciones.

2.5.5. Distribución gamma

Parecida a la anterior distribución, con la diferencia de que esta se vale de la función gamma. Su fórmula es [12]:

$$p(x; k, \theta) = x^{k-1}\frac{e^{\frac{-x}{\theta}}}{\theta^k \Gamma(k)} \tag{2.19}$$

Los parámetros a determinar son:

[12] Se ha escogido la fórmula en *numpy.random.gamma. Numpy API reference Ver. 1.23 (Stable)* https://numpy.org/doc/stable/reference/random/generated/numpy.random.gamma.html#numpy-random-gamma

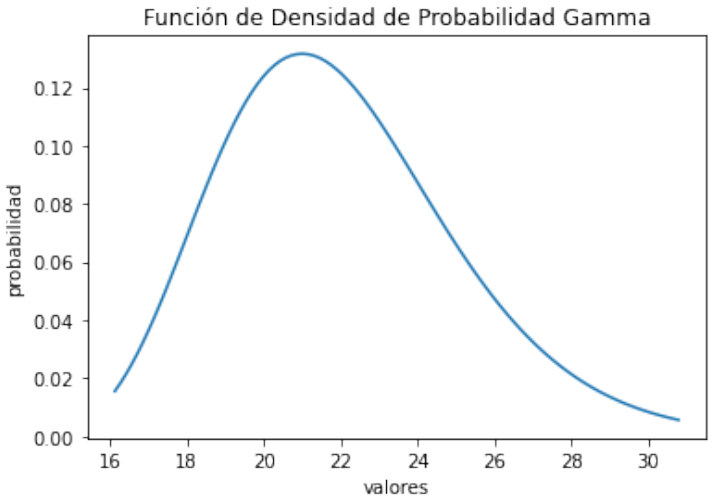

Figura 2.10: Función de densidad de probabilidad de distribución gamma para los valores $k = 10, \theta = 12$

- k: Parámetro de forma.
- θ: Parámetro gamma. Indica la intensidad de la función gamma.

La función gamma $\Gamma(k)$ equivale a:

$$\Gamma(k) = \int_0^\infty t^{k-1} e^{-t} dt \tag{2.20}$$

Tendría bastantes posibilidades de ajuste debido a su funcionamiento, ya que su comportamiento es similar a la de la distribución Beta.

2.5.6. Distribución Log-Normal

Esta distribución continua tiene una forma parecida a la del anterior apartado, aunque tiene una forma ligeramente distinta debido a su fórmula, la cual viene dada por:

$$p(x; \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} e^{\frac{-1}{2}(\frac{lnx-\mu}{\sigma})^2} \tag{2.21}$$

Como indica Gil Bellosta [2021], esta distribución (de asimetría positiva, afirma Lopez Briega [2016]) es la exponencial de una distribución normal, y hace de *atractor de distribuciones* al usar distribuciones que se multiplican (por ejemplo, modelar resultados bursátiles), o para variables aleatorias limitadas por cero que tienen pocos valores

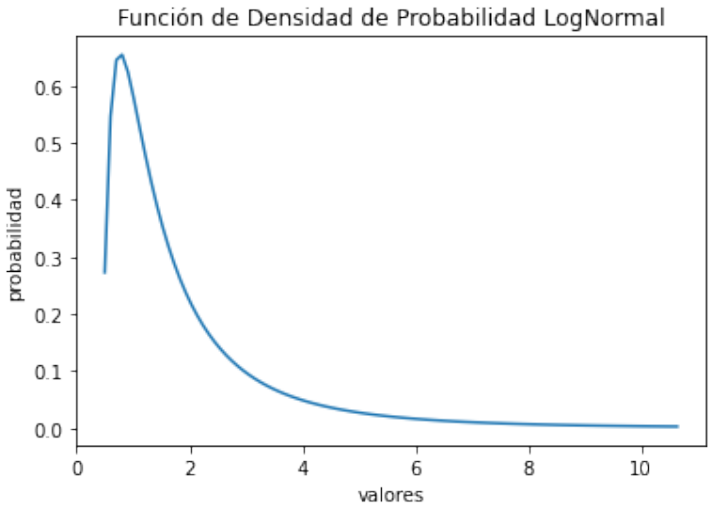

Figura 2.11: Función de densidad de probabilidad de distribución Log-Normal para los valores $\mu = 1, \sigma = 0{,}4$

grandes. Sus variables[13] necesarias para funciones de *Python* son:

- μ: Media para la distribución normal subyacente. $\mu > 0$
- σ: Desviación típica para la distribución normal subyacente. $\sigma > 0$

Parece ser una distribución también ideal para concentrar la fragmentación en un punto de la vasija.

2.5.7. Distribución de Pareto

Esta distribución fue creada por Vilfredo Pareto para explicar la distribución de salarios en la sociedad, aunque se puede aplicar a problemas como la frecuencia de descargas en un sitio web. Sigue el denominado **Principio de Pareto**, que Westreicher [2020] define como:

> *"La ley de Pareto es un principio que establece que el 20 % del esfuerzo destinado a una tarea genera un 80 % de los resultados. Esto es aplicable a distintas áreas, como la actividad empresarial o el ámbito personal."*

Esto significa que mientras que el 80 % de los pesos de distribuyen en el 20 % de los valores, el 20 % de los pesos restantes se distribuyen en el 80 % de los valores

[13] *numpy.random.lognormal. Numpy API reference Ver. 1.23 (Stable)* https://numpy.org/doc/stable/reference/random/generated/numpy.random.lognormal.html#numpy-random-lognormal

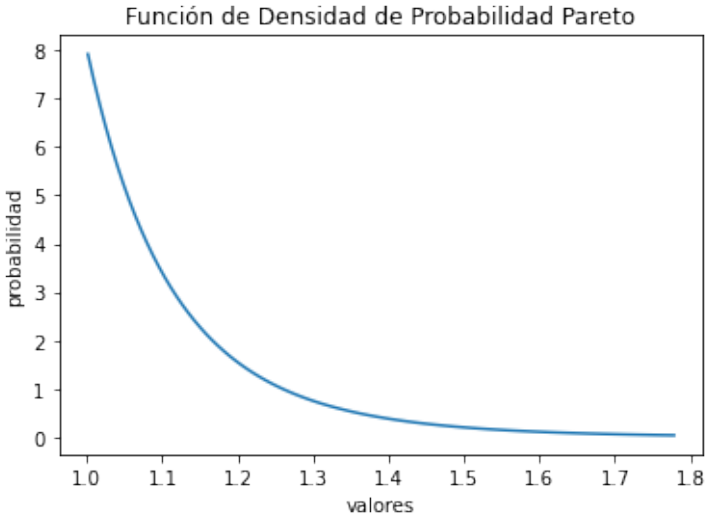

Figura 2.12: Función de densidad de probabilidad de distribución de Pareto para el valor $\alpha = 8$

restantes. Esto se refleja en su función de densidad, en la que la mayor probabilidad se reparte en un extremo, y baja exponencialmente a lo largo del rango. Sigue la función:

$$p(x; \alpha, k) = \frac{\alpha k^{\alpha}}{x^{\alpha+1}} \tag{2.22}$$

Sus variables[14] son:

- α: Forma de la función. Es la única variable que necesita su correspondiente función de *Python*. $\alpha > 0$
- k: Escala. $x \geq k$

Esta distribución también parece concentrar todos los fragmentos en la base de la vasija debido a su función.

2.5.8. Distribución Chi Cuadrado

Conocida también como distribución de Pearson o distribución ji al cuadrado. Para esta distribución primero definiremos **chi cuadrado**[15] (χ^2). Dadas $Z_1, Z_2, ...Z_k$ variables aleatorias independientes tales que $Z_i \in (0, 1)$ para $i = 1, 2..., k$, entonces tenemos la

[14] *numpy.random.pareto. Numpy API reference Ver. 1.23 (Stable)* https://numpy.org/doc/stable/reference/random/generated/numpy.random.pareto.html#numpy-random-pareto

[15] Distribución χ^2, en *Wikipedia*. 5 de junio de 2022.https://es.wikipedia.org/w/index.php?title=Distribuci%C3%B3n_%CF%87%C2%B2&oldid=144011286

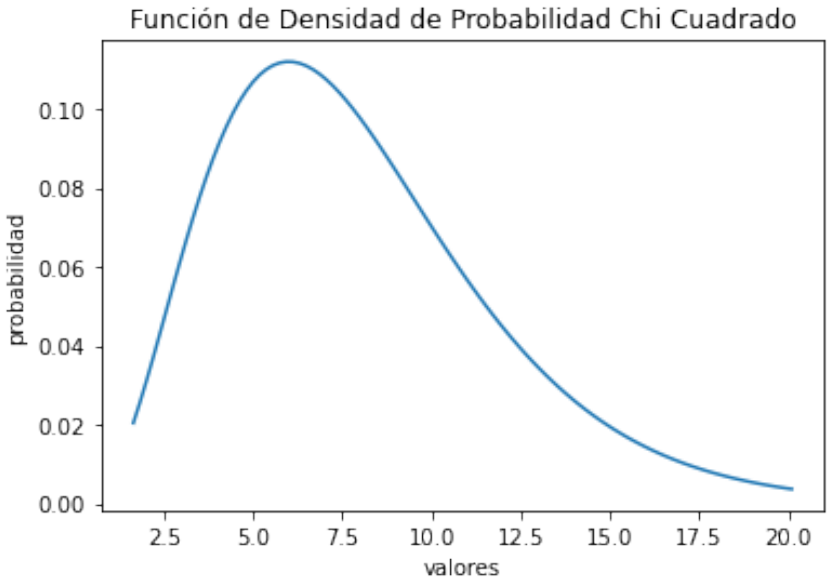

Figura 2.13: Función de densidad de probabilidad de distribución Chi Cuadrado para el valor $\alpha = 8$

variable aleatoria X consistente en la sumatoria de dichas variables aleatorias elevadas al cuadrado, denotado como:

$$X = Z_1^2 + Z_2^2 + ...Z_k^2 = \sum_{k=1}^{i} Z_i^2 \tag{2.23}$$

Afirmaremos entonces que X es una distribución chi cuadrada con k grados de libertad.

$$X \sim \chi_k^2 \tag{2.24}$$

Por tanto la función de distribución será:

$$p(x;n) = \frac{(\frac{x}{2})^{\frac{n}{2}-1} e^{\frac{-x}{2}}}{2\Gamma(\frac{n}{2})} \tag{2.25}$$

La única variable necesaria es n, conocida como grados de libertad. Genera una curva semejante a la distribución normal pero desplazada y que además se puede ajustar con la variable antes mencionada. Usa la función Gamma (*véase 2.5.5 Distribución gamma*).

2.5.9. Distribución t de Student

Al igual que la anterior distribución, la distribución t de Student también genera una curva semejante a la que crea la distribución normal (esta vez sin desplazamientos).

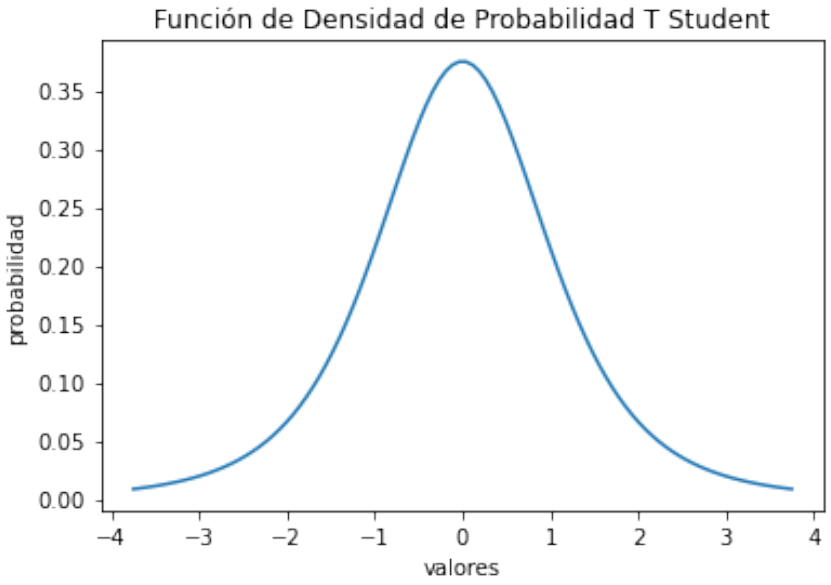

Figura 2.14: Función de densidad de probabilidad de distribución t de Student para el valor $n = 5$

Sin embargo, la curva se modifica en base a la variable t, que representa el grado de libertad (no depende de la media y la varianza a diferencia como ocurre con la normal, solo del grado de libertad). Aumentar dicho grado de libertad reduce el valor máximo que alcanza la curva y también ”suaviza” su pendiente.

Viene dada por la fórmula (que también usa la función gamma):

$$p(t;n) = \frac{\Gamma(\frac{n+1}{2})}{\Gamma(\frac{n}{2})\sqrt{n\pi}}(1 + \frac{r^2}{2})^{-\frac{n+1}{2}} \tag{2.26}$$

También Rodó [2019b] recoge la aplicación ideal de esta distribución:

- Hallar la media de una población normalmente distribuida por una muestra pequeña.
- Emplear una muestra de menos de 30 elementos: $n < 30$.

2.5.10. Distribución triangular

La última distribución continua que explicaremos posee una función de densidad con forma de triángulo, como sugiere su nombre. Los 3 parámetros a definir serán los 3 lados de dicho triángulo (a, b, c). En su correspondiente función de *Python*, hay que asegurarse de que $a \leq b \leq c$, ya que a y c representan el mínimo y el máximo, y la

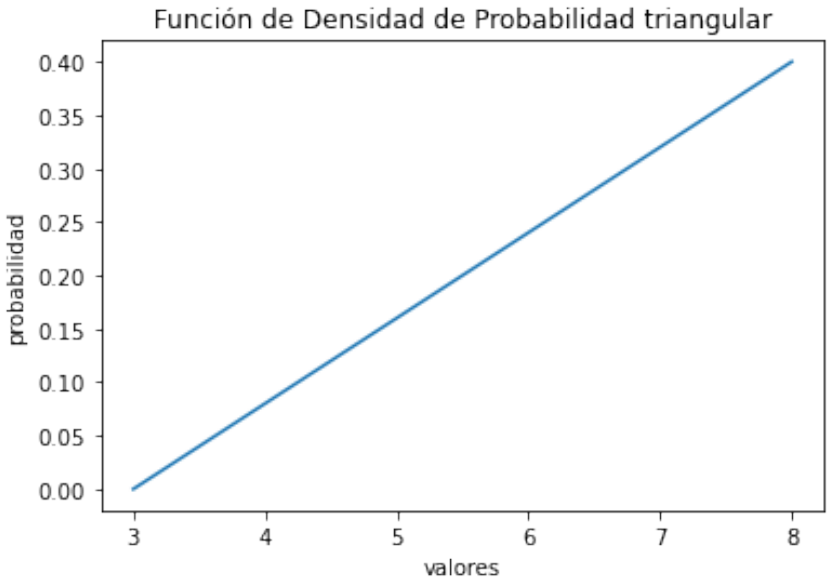

Figura 2.15: Función de densidad de probabilidad de distribución triangular para los valores $a = 1, b = 3, c = 5$

variable restante b el pico del triángulo. *(Debido al funcionamiento de Scipy, no se ha podido obtener la visualización del pico, ya que solo permite visualizar una pendiente.)*

Como afirma Benites [2022], es una distribución muy útil ya que permite estimar una variable aleatoria conociendo su mínimo (a), su máximo (c) y su valor más probable (b). Su función se define como:

$$F(x; a, b, c) = \begin{cases} \frac{2(x-a)}{(c-a)(b-a)} & , \textit{para } a \leq x \leq b \\ \frac{2(c-x)}{(c-a)(c-b)} & , \textit{para } b \leq x \leq c \\ 0 & , \textit{en otro caso} \end{cases} \tag{2.27}$$

Esta última distribución bien podría crear fragmentaciones de forma triangular o bien concentrar en un solo punto como otra previas.

Capítulo 3

OBJETIVOS

3.1. Objetivos generales

Los principales problemas a resolver en este TFG son:

- Realizar distintas simulaciones de rotura de objetos y determinar ante qué condiciones, como golpes, caídas o disparos de proyectiles, son las más acertadas.
- Comprender los procedimientos de ingeniería del software en el contexto de creación de efectos visuales.
- Integrar distintos componentes y lenguajes de programación en una sola plataforma de desarrollo.

3.2. Objetivos específicos

- Estudiar y comprender algoritmos existentes de fragmentación para VFX y aplicar sus principios a nuestro software.
- Aprender el funcionamiento de scripts de distribución libre para adaptarlo a nuestras necesidades.
- Emplear lenguaje de programación *Python* adaptado a su uso en *Blender*.
- Crear *scripts* para cada una de las distribuciones de probabilidad mencionadas en *2.3 Distribuciones de probabilidad* que modificarán el resultado final de la generación de fragmentos.

- Realizar experimentación con los ajustes para cada tipo de fragmentación.
- Generar una serie de roturas para cada distribución y determinar para qué tipo de situación es cada distribución más idónea.

3.3. Metodologías

En esta sección analizaremos los pasos y materiales necesarios para realizar este proyecto, asumiendo que es un proyecto organizado por una empresa (de informática).

Se seguirá un **desarrollo incremental** de los *scripts* necesarios para obtener la versión funcional. Los pasos a seguir son:

1. **Aprendizaje**: Habituarse al entorno de trabajo en *Blender* y aprender a usar su *add-on Fracture Cell* con *Python*.
2. **Primera versión**: Crear una versión preliminar y funcional de los scripts con todo lo necesario para funcionar independientemente, incluyendo posibles modificaciones en el código de *Fracture Cell* en caso de encontrar incompatibilidades.
3. **Versión modificada**: Crear una versión modificada con el código común encapsulado en un *script* de uso común para reducir la carga de los *scripts* y facilitar el cambio de variables.
4. **Versión final**: Crear una versión final con un archivo de configuración de uso común para cada *script*.
5. **Experimentación**: Obtener fragmentaciones con vasijas de ejemplo para corroborar su funcionamiento y extraer conclusiones de cada distribución de probabilidad.
6. **Desarrollo de esta memoria** de forma paralela a los demás pasos, incluyendo la generación de histogramas para explicar las distribuciones empleadas.

3.3.1. Recursos

- **Recursos humanos**: Debido a la simplicidad del software, solo ha sido necesario un solo programador, el cual es un empleado al que le pagan un sueldo en bruto al mes.

- **Recursos materiales**: En principio solo necesitaríamos un ordenador conectado a Internet ofrecido por la empresa para poder desarrollar el proyecto. Sin embargo debemos factorizar el precio de la luz, del uso de Internet, las horas empleadas y también costes adicionales del domicilio del programador para poder terminarlo.

- **Recursos software**: Se ha usado principalmente *Blender*, que como se ha dicho anteriormente, es un programa gratis de código libre. Los recursos necesarios también se han obtenido de sitios web de acceso gratuito, incluyendo principalmente la plataforma de vídeos YouTube. Adicionalmente, se ha usado el entorno de programación *Anaconda*, concretamente *Jupiter Notebook*, para emplear la biblioteca *Matplotlib* y generar los histogramas, también disponible gratuitamente.

3.3.2. Cronograma

El desarrollo comenzó en Febrero de 2022 y terminó en Diciembre de 2022 (11 meses de trabajo), distribuyendo el trabajo aproximadamente de esta manera:

Fase	Mes	Total horas
Fase de aprendizaje	Febrero	20 horas
	Marzo	40 horas
Primera versión	Abril	50 horas
	Mayo	50 horas
	Junio	55 horas
Versión modificada + memoria	Agosto	65 horas
	Septiembre	30 horas
Versión final + memoria	Octubre	40 horas
	Noviembre	40 horas
Experimentación + memoria	Diciembre	35 horas
-	**Total**	**425 horas**

Tabla. 3.1: Horas trabajadas por mes

Desde Agosto se comenzó el desarrollo de la memoria de forma paralela a las demás actividades.

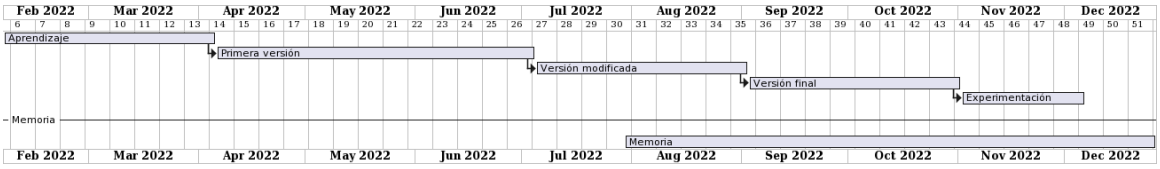

Figura 3.1: Diagrama de Gantt

3.3.3. Presupuesto

Recursos humanos

Definiremos los parámetros económicos necesarios para el programador (licenciado en Grado de Informática) que ha realizado el proyecto:

- **Sueldo**: Sueldo tanto mensual como por hora del trabajador.
- **Incremento adicional** del 33 % por costes de seguros sociales y otros costes de empresa.
- **Jornada laboral**: Horas que trabaja por semana.
- **Desplazamientos**: El trabajador casualmente no necesita hacer inversiones en sus desplazamiento, ya que se permite hacerlos a pie.

Atributo	Valor
Trabajadores	1 (Graduado en ingeniería informática)
Incremento	33 %
Jornada Laboral	40 h / semana
Desplazamientos	0 €
Sueldo mensual	1900 € / mes (brutos)
Sueldo por hora	11,875 € / hora
Total sin incremento	20137,5 €
Total	2842,875 €

Tabla. 3.2: Presupuesto de Recursos Humanos

Recursos materiales

En esta sección se definirá una lista de todos los materiales necesarios.

- **Ordenador**: En nuestro caso se ha trabajado con un ordenador de sobremesa.
- **Software**: *Blender* y *Anaconda 3*. Ambos de descarga y uso gratuitos.

Hay que amortizar el precio del ordenador a lo largo de la duración del proyecto.

Atributo	Valor
Ordenador	1200 €
Software	0 €
Periodo de uso del ordenador	6 años * 24 horas * 365 días = 52560 horas
Amortización del ordenador	(425 * 1200) / 52560 = 9,7 €
Total	9,7 €

Tabla. 3.3: Presupuesto de Recursos Materiales

Costes indirectos

- **Precio de la luz**: Asumiremos un coste total para la iluminación y calcularemos por otro lado la luz consumida por el ordenador.
- **Precio de internet**: El trabajador en este caso posee Internet por fibra óptica.
- **Vivienda**: En este caso el trabajador vive en un hogar que ya estaba totalmente pagado mucho antes del comienzo del proyecto, por lo que se descarta.
- **Administración**: También se descarta porque no se han requerido acciones administrativas para crear el proyecto.

Presupuesto total

A los precios totales le añadiremos un beneficio.

Atributo	Valor
Coste kWh	0,15 €
Internet (fibra)	20 € / mes
Cuota de línea	18,15 € / mes
Internet + Teléfono	38,15 € / mes
Vivienda	0 €
Administración	0 €
Total luz	5,40 €
Total Internet	42,92 €
Total	48,32 €

Tabla. 3.4: Tabla de costes indirectos

Atributo	Valor
Presupuesto de RRHH	2848,875 €
Presupuesto de Recursos materiales	9,7 €
Presupuesto de costes indirectos	48,32 €
Beneficios	60 %
Total sin beneficios	2906,895 €
Beneficio	1744,137
Total	4651,032 €

Tabla. 3.5: Tabla de presupuesto total

Capítulo 4

MATERIALES Y MÉTODOS

4.1. Materiales

4.1.1. Introducción a Blender

Blender es un software libre de gráficos en 3D de código abierto, aunque también puede trabajar con gráficos 2D, tanto imágenes como vídeos. Ya se introdujo en la sección *2.1.2 Software de creación de CGI*, pero aquí haremos una ampliación al ser el software principal con el que se ha trabajado.

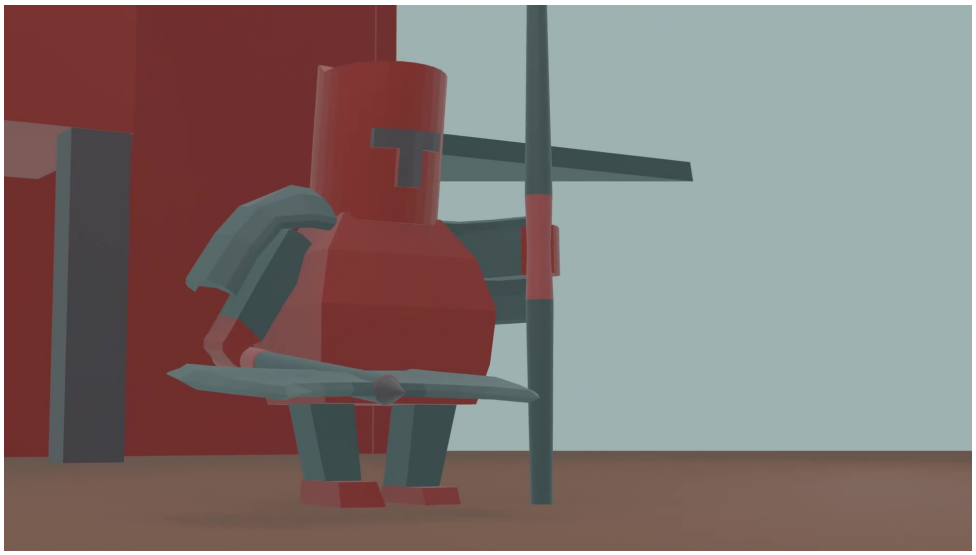

Figura 4.1: *Render* generado en Blender, con modelos creados por el autor.

La versión trabajada ha sido la 2.9. Nada más iniciar el programa, veremos una ventana con varias pestañas en la parte superior. Estas pestañas son los distintos

entornos de trabajo disponibles (aunque es posible editarlos o añadir más). Los que vienen por defecto para un proyecto en 3D son:

- ***Layout***: Entorno predeterminado.
- ***Modelling***: Muy parecido al primero pero está orientado al modelado 3D.
- ***Sculpting***: Contiene herramientas para manipular objetos como si de arcilla se tratase.
- ***UV Editing***: Permite asignar y aplicar texturas al volumen de un modelo 3D.
- ***Texture paint***: Parecido al anterior pero permite también pintar texturas como en un software de pintura.
- ***Shading***: Posee un editor de *shaders* basado en conexión de nodos.
- ***Animation***: Editor de animación 3D.
- ***Rendering***: Muestra el *render* generado.
- ***Compositing***: En esencia un editor de imágenes.
- ***Scripting***: Entorno en el que se ha trabajado principalmente. Se expondrá más a continuación.

4.1.2. Entorno de programación

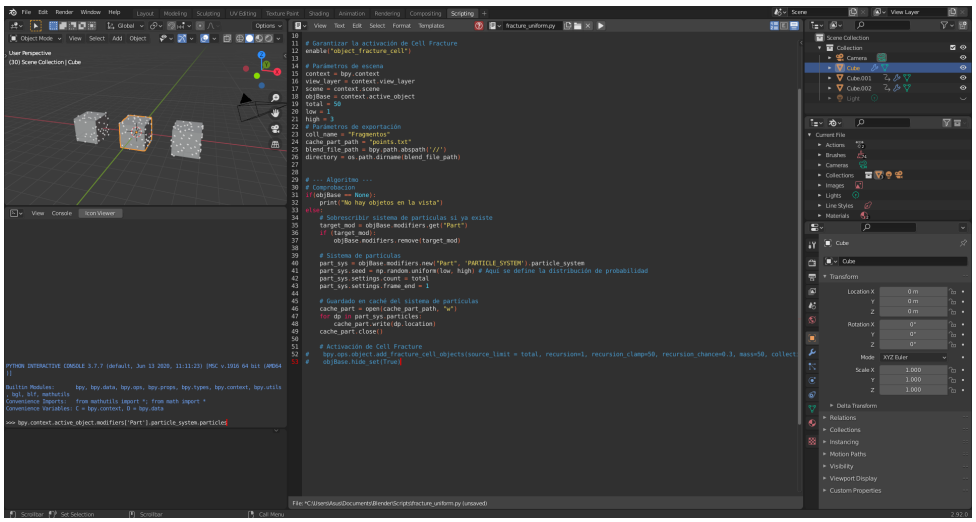

Figura 4.2: Entorno de programación Blender con el que se ha realizado esta sección del TFG.

Ya hemos mencionado que Blender permite la ejecución de *scripts* en lenguaje de programación *Python* para multitud de posibilidades. Mediante el uso de *scripts*, podremos modificar el comportamiento del *add-on* que realiza la fragmentación automáticamente: ***Cell Fragment***. Se ha trabajado en la pestaña *scripting*, diseñada específicamente para esta tarea.

4.1.3. El *add-on Cell Fragment*

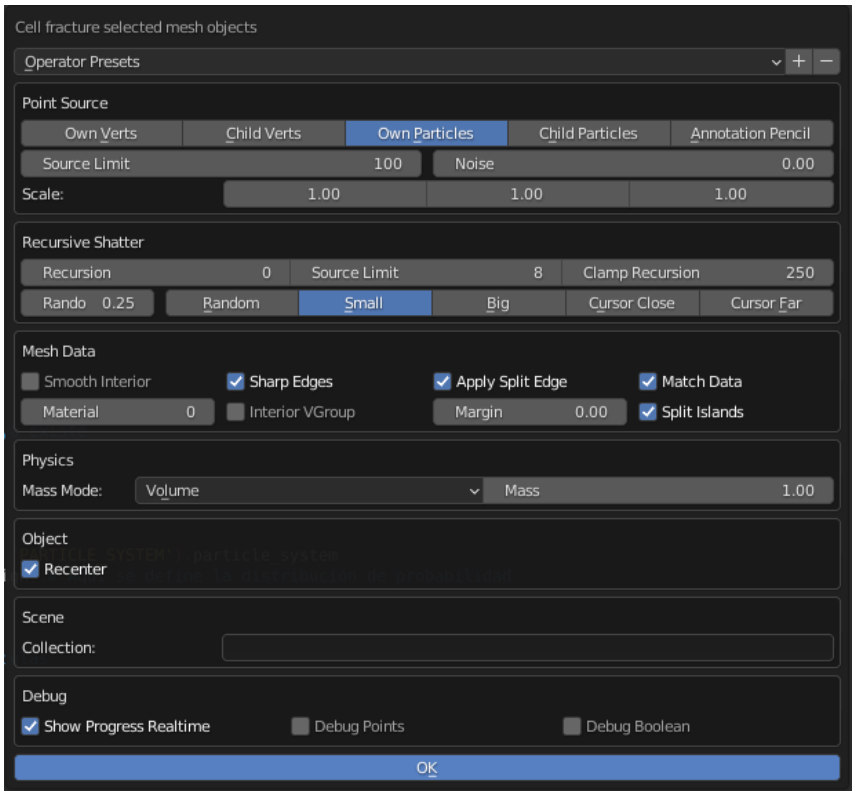

Figura 4.3: Captura de pantalla de la ventana de Cell Fracture

Esta función de Blender permite crear fragmentos de un objeto dado en función de diferentes parámetros. Para que se ejecute correctamente, se debe seleccionar un objeto 3D de la escena antes de abrir la ventana y, tras pulsar en el botón *OK*, se crearán los fragmentos del objeto seleccionado. Una forma rápida de acceder a este *plug-in* es pulsar `F3`, atajo de teclado que abre un buscador, y escribir "`Cell Fracture`". Los parámetros se organizan en las subsiguientes categorías:

- ***Point source***: Indica el modo de realizar la fragmentación. Posee las siguientes opciones:
  - ***Own verts***: El objeto se fragmenta en función de sus propios vértices. Así, un cubo se subdividirá en 8 cubos de forma predeterminada.

- ***Child verts***: Requiere que el objeto a fragmentar tenga al menos un objeto agrupado[1], de forma que el "hijo" será la guía para generar fragmentos. Nota: La posición del hijo determinará la fragmentación final.
- ***Own particles***: Requiere que el objeto a fragmentar posea un modificador de tipo Sistema de partículas.
- ***Child particles***: Parecido al anterior pero el sistema de partículas lo tendrá el objeto emparentado con el que se va a fragmentar.
- ***Annotation pencil***: Se realiza la fragmentación en base a los trazos con el lápiz de anotación. Al usar este modo, es recomendable seleccionar la opción *surface* en el menú desplegable que se mostrará al seleccionar la herramienta.

  Este apartado incluye opciones adicionales:

  - *Source limit*: Indica el máximo número de fragmentos que se generarán.
  - *Noise*: Aleatoriza los fragmentos a crear en función del grado escogido.
  - *Scale*: Modifica el tamaño de los fragmentos en los ejes X, Y o Z.

- ***Recursive shattering***: Si está activado, permite fragmentar de forma recursiva fragmentos ya generados.

  - ***Recursion***: Con 0 se desactiva.
  - ***Source limit***: Define el máximo número de fragmentos que pueden ser fragmentados recursivamente. Con 0 no hay límite.
  - ***Clamp recursion***: Define el máximo número de fragmentos que se generarán a partir de un fragmento que se vaya a subdividir.
  - ***Random***: Define la probabilidad de que un fragmento se fragmente recursivamente. Debe de ser mayor que 0 o no se producirá fragmentación recursiva.
  - ***Recursion chance select***: Una serie de botones que indican qué fragmentos podrán ser fragmentados recursivamente.
    - *Random*: Cualquier fragmento
    - *Small*: Solo fragmentos pequeños.
    - *Big*: Solo fragmentos grandes.
    - *Cursor close*: Solo fragmentos cerca del cursor 3D (*véase Tabla de definiciones*).
    - *Cursor far*: Solo fragmentos alejados del cursor 3D.

[1] Para agrupar, primero se selecciona el objeto que será el hijo y luego el objeto padre. A continuación se pulsa `Ctrl + P` y se selecciona agrupar objetos (*group objects*).

- ***Mesh data***: Opciones relacionadas con las mallas. Según el tutorial para usar este *add-on* del canal de *YouTube* de *Mr. Cheebs* [2020], es recomendable dejar las casillas ya marcadas como están para que funcione correctamente, además de que la mayoría no realizan cambios notorios. Sin embargo, nos interesan las opciones que no son de casilla:
  - *Material*: Indica qué material asignado al objeto a fragmentar se usará para los fragmentos.
  - *Margin*: Define un márgen entre los fragmentos. Esto permite mejores resultados para usar físicas.
- ***Physics***: Opciones de físicas. Su única opción es ***Mass mode***: que define el método para generar las masas de los fragmentos.
  - *Volume*: La masa se determina en función del volumen.
  - *Uniform*: La masa de todos los fragmentos será la especificada en Mass.
- ***Objects***: Solo posee una casilla de verificación: ***Recenter***, que, una vez seleccionada, permite equilibrar el centro de los fragmentos. Recomendable dejarlo marcado para uso de físicas.
- ***Scene***: Solo posee una casilla de introducción de texto: ***Collection***. Si se escribe un nombre en la casilla, los fragmentos se generarán en una nueva colección con dicho nombre de no existir, de lo contrario, se colocarán en la colección especificada. Si no se especifica nada, se generarán en la misma colección en la que se sitúa el objeto.
- ***Debug***: Opciones para hacer *debugging* (no modifican el resultado final).
  - ***Show progress realtime***: Visualiza de forma iterativa la creación de fragmentos.
  - ***Debug points***: Genera los puntos con los que se generan los fragmentos.
  - ***Debug booleans***: Los fragmentos tendrán un modificador *Boolean*.

4.1.4. *Script* base

Para la primera versión, se han creado un *script* que genera una fragmentación en función de una de las distribuciones ya presentadas. Para su correcto funcionamiento de todas las versiones de los scripts, es necesario una modificación del archivo fuente

`fracture_cell_setup.py`, en la carpeta de *add-ons* en la versión de *Blender* que estamos usando, ya que la única posibilidad es usar las partículas asignadas al objeto y la estructura interna del *setup* solo toma como base las posiciones de partículas en el momento de su creación. Por ello, para poder usar las posiciones modificadas, debemos guardar en un fichero aparte las posiciones y luego cargarlas dentro del *setup*. *(Ver 4.1.5 Modificación del add-on Cell Fragment)*

Su esquema general viene dado por:

Algoritmo 1: Algoritmo de fragmentación en base a una distribución

```
Result: Objeto 3D fragmentado
Definir variables de escena;
Definir variables de distribución;
Definir variables de exportación;
Definir opciones de comportamiento;
if hay exactamente un objeto seleccionado then
    Encontrar objeto de recipiente;
    Crear sistema de partículas en el recipiente;
    Ajustar partículas del sistema a la función de distribución;
    Guardar posiciones de nuevas partículas en fichero caché;
    Ejecutar cell_fracture modificado;
    if se activa la separación then
        Separar fragmentos;
    end
    if se activa la exportación then
        if se exportan todos then
            Exportar todos los fragmentos en un solo fichero .obj.;
        else
            for fragmento generado do
                Exportar fragmento en un fichero .obj;
            end
        end
    end
end
```

```
1 import bpy
2 import os
3 import numpy as np
4 import random
5 import time
6 from addon_utils import enable
7 from addon_utils import disable
```

```
8  from mathutils import Vector
9
10 # IMPORTANTE:
11 # 1- Seleccionar el objeto para que se aplique la fragmentación.
12 # 2- Asegurarse de que el proyecto está guardado en disco.
13 # 3- Para reiniciar, eliminar la colección creada y sus objetos con Delete Hierarchy y mostrar el objeto
       base.
14
15 # Garantizar la activación de Cell Fracture
16 #enable(.object_fracture_cell")
17
18 # Parámetros de escena
19 context = bpy.context
20 view_layer = context.view_layer
21 scene = context.scene
22 objBase = context.active_object
23 # Parámetros del sistema de partículas
24 total = 4
25 partSysName = "ParticleSystem"
26 # Parámetros de la función de distribución
27 media, sigma = 8, 0.9
28 # Parámetros de exportación
29 coll_name = "Fragmentos"
30 cache_name = 'points.txt'
31 file_name = 'Vasija'
32 directory = 'C:\\Users\\Asus\\Documents\\Blender'
33 # Opciones
34 export = True # Realizar la exportación
35 export_all = True # Exportar todos los fragmentos en un solo obj.
36 separate = True # Separa los fragmentos en función de la variable sepMult.
37 sepMult = 0.3
38
39 # Funciones
40 def particleSetter(self):
41         particle_systems = vessel.evaluated_get(depsgraph).particle_systems
42         for m in range(total):
43             x=m/total
44             y=m/total
45             z=m/total
46             particle_systems[0].particles[m].location = (x,y,z)
47             print('ParticleSetter ',m,' (',x,', ',y,',',z,')\n')
48
49         # Toggle back and forth to update viewport
50         bpy.ops.particle.particle_edit_toggle()
51         bpy.ops.particle.particle_edit_toggle()
52
53 def calcCenter(obj):
54     local_center = 0.125 * sum((Vector(b) for b in obj.bound_box), Vector())
55     global_center = obj.matrix_world @ local_center
56     return global_center
57
58
59 # Main
60
61 disable ("object_fracture_cell")
62
63 selected_list = [obj.name for obj in context.selected_objects]
64 if(len(selected_list) != 1):
65     print("Se debe seleccionar exactamente un objeto")
66 else:
```

```
    degp = bpy.context.evaluated_depsgraph_get()

    #Seleccionamos el objeto del que colgará el sistema de particulas
    vessel = bpy.data.objects[selected_list[0]]

    # Buscamos su centro
    centro = calcCenter(vessel)

    # Sobrescribir sistema de particulas si ya existe
    target_mod = vessel.modifiers.get(partSysName)
    if (target_mod):
        bpy.ops.object.particle_system_remove()

    #Le añadimos el sistema de partícula
    psys=vessel.modifiers.new("ParticleSystem", 'PARTICLE_SYSTEM').particle_system
    psys.settings.count = total
    psys.settings.frame_start = 0
    psys.settings.frame_end = 100
    psys.settings.lifetime = 1000

    depsgraph = bpy.context.evaluated_depsgraph_get()
    particle_systems = vessel.evaluated_get(depsgraph).particle_systems

    particleSetter(vessel)
    time.sleep(0.3)
    bpy.ops.particle.particle_edit_toggle()
    bpy.ops.particle.particle_edit_toggle()

    # Modificación de las partículas a la distribución
    for m in range(total):
        x=particle_systems[0].particles[m].location[0]
        y=particle_systems[0].particles[m].location[1]
        z=particle_systems[0].particles[m].location[2]
        print('Particle first pass: ',m,' (',x,', ',y,', ',z,')\n')

        particle_systems[0].particles[m].location[0]=np.random.uniform(media, sigma)
        particle_systems[0].particles[m].location[1]=np.random.uniform(media, sigma)
        particle_systems[0].particles[m].location[2]=np.random.uniform(media, sigma)
        x=particle_systems[0].particles[m].location[0]
        y=particle_systems[0].particles[m].location[1]
        z=particle_systems[0].particles[m].location[2]
        print('Particle second pass: ',m,': (',x,', ',y,', ',z,')\n')

    # Escribir caché de partículas
    particles_data = vessel.evaluated_get(depsgraph).particle_systems[0].particles

    # Crea un bloque de texto específico para guardar coordenadas
    text_list=[txt.name for txt in bpy.data.texts]
    if (not 'Cache' in text_list):
        bpy.ops.text.new()
        bpy.data.texts[-1].name = 'Cache'
    else:
        bpy.data.texts['Cache'].clear()

    for p in particles_data:
        bpy.data.texts['Cache'].write(str(p.location[0])+","+str(p.location[1])+","+str(p
            .location[2])+"\n")

    blenderCipher = open(directory + '\\' + cache_name,'w')
```

```
    textpos_str = bpy.data.texts['Cache'].as_string()

    for line in textpos_str:
        blenderCipher.write(line)

    blenderCipher.close()

    # Activación de Cell Fracture
    enable("object_fracture_cell")
    bpy.ops.object.add\_fracture\_cell\_objects(source={'PARTICLE_OWN'},source_limit =
        total, recursion=2, recursion_clamp=50, recursion_chance=0.9, mass=50,
        collection_name=coll_name)
    #bpy.ops.object.add_fracture_cell_objects(source='PARTICLE_OWN',source_limit=total)
    #bpy.ops.object.add_fracture_cell_objects(source='PARTICLE_CHILD',source_limit=total)
    #bpy.ops.object.add_fracture_cell_objects(source='PARTICLE_OWN')

    # Separar las piezas
    coll = bpy.data.collections['Fragmentos']

    if (separate):
        for frag in coll.all_objects:
            vec_a = Vector(frag.location)
            frag.location = frag.location + sepMult * (vec_a - centro)


    # Exportación
    if (export):
        if (export_all):
            # Exportar todos los fragmentos en un solo archivo
            export_path = directory + '\\' + file_name + '.obj'
            bpy.ops.export_scene.obj(filepath=export_path, use_selection=True)
        else:
            # Exportar los fragmentos en diferentes archivos
            cont = 0

            # Deseleccionar todos los objetos
            obj_selected_list = [obj for obj in bpy.context.selected_objects]
            for frag in obj_selected_list:
                frag.select_set(False)

            # Seleccionar individualmente y exportar
            for frag in coll.all_objects:
                frag.select_set(True)
                export_path = directory + file_name + '_' + str(cont) + '.obj'
                bpy.ops.export_scene.obj(filepath=export_path, use_selection=True)
                cont += 1
                frag.select_set(False)
```

Listado 4.1: *Script* para fragmentación en base a una distribución uniforme

Las variables a definir son:

- **Parámetros de escena**: Variables necesarias para el funcionamiento base que no deben de ser modificadas.
- **Parámetros del sistema de partículas**:

  - `total`: Total de partículas base.
  - `partSysName`: Nombre del sistema de partículas. No es necesario modificarlo pero es recomendable dejar el nombre por defecto para evitar crear sistemas de partículas adicionales.

- **Parámetros de la función de distribución**: Una o dos variables que dependerán de cada distribución de probabilidad.

- **Parámetros de exportación**:

  - `coll_name`: Nombre de la colección en la que se depositarán los fragmentos generados.
  - `cache_name`: Nombre del fichero de extensión `.txt` que usará la modificación de `fracture_cell_setup.py` *(ver abajo)*. **Importante: Debe coincidir con la variable del mismo nombre de dicha modificación**.
  - `file_name`: Nombre a usar en el archivo o archivos `.obj` exportados.
  - `directory`: Directorio donde se exportan los archivos generados, tanto los archivos *.obj* como la caché para los puntos modificados. **Importante: Debe coincidir con la variable del mismo nombre de** `fracture_cell_setup.py`.

- **Opciones**: Variables que modifican el comportamiento.

  - `export`: Si se activa esta variable booleana, el script exportará los fragmentos.
  - `export_all`: Si se va a proceder a exportar y esta variable booleana se activa, todos los fragmentos se exportarán en un solo fichero *.obj*; de lo contrario, se exportará cada fragmentos individualmente.
  - `separate`: Al activarse, los fragmentos se separarán siguiendo su normal en función de la variable a continuación. Útil para visualizar mejor la disparidad entre los fragmentos.
  - `sepMult`: Distancia en la que se separarán los fragmentos de hacerlo. 0.3 es una cantidad óptima.

4.1.5. Modificación del *add-on Cell Fragment*

Uno de los mayores obtáculos en la realización del proyecto ha sido debido al funcionamiento interno de este *add-on*. La opción de sistema de partículas no posee una función para que se ajuste correctamente a una función de distribución concreta, sino

que siempre tiene una distribución uniforme. La solución más evidente para este problema sería simplemente hacer el cambio desde el *script* y luego ejecutar *Cell Fragment*; sin embargo, no funciona así. El código, en lugar de usar el sistema de partículas modificado, emplea el sistema de partículas en su estado original, en el momento que fue creado. Esto se debe al *Dependency Graph*, también conocido como *depsgraph*, responsable de que se actualicen ciertos cambios. El *depsgraph* no registra ese cambio en el sistema de partículas, ya que solo registra los cambios dinámicos, es decir, debido a físicas o animaciones[2]. La única manera de hacer frente a este problema ha sido guardar los cambios en una caché en disco y luego cargarlos desde el interior de *Cell Fracture*, concretamente en el archivo fuente `fracture_cell_setup.py`. Este código añadido a dicho fichero (concretamente a la función `points_from_particles(obj)`) permite cargar las posiciones modificadas de las partículas desde la caché que crean los *scripts*.

Importante: Para usar esta modificación, cambiar la dirección de `dir_config` por el del fichero `Frag_config.txt` (como veremos más abajo, este *script* posee variables universales a todos los *scripts*).

```
def points_from_particles(obj):
    obj_eval = obj.evaluated_get(depsgraph)
    #Added by Pablo Pérez Peña

    # Cambiar por direccion y archivo adecuados
    dir_config = 'C:\\Users\\Asus\\Documents\\Blender\\Scripts\\frag_config.txt'
    configCipher = open(dir_config,'r')
    directory = configCipher.readline().rstrip()
    cache_name = configCipher.readline().rstrip()
    configCipher.close()

    blenderCipher = open(directory + '\\' + cache_name,'r')

    #Added by Rafael Segura
    for psys in obj_eval.particle_systems:
        for p in psys.particles:
            line = blenderCipher.readline()
            line_div = line.split(",")
            x=float(line_div[0])
            y=float(line_div[1])
            z=float(line_div[2])
            p.location= [x,y,z]
            print('Inside Addon : (' , x ,', ' , y , ', ' , z ,')\n')

    blenderCipher.close()

    points.extend([p.location.copy()
                   for psys in obj_eval.particle_systems
                   for p in psys.particles])
```

[2] *Blender 2.8: Dependency Graph, Blender Wiki.* https://wiki.blender.org/wiki/Source/Depsgraph

Listado 4.2: *Script* modificado para fragmentación en base a una distribución uniforme

4.1.6. *Scripts* definitivos

Para la versión final de los scripts se han encapsulado los métodos comunes en un solo archivo que se usará como módulo compartido: `fragcore.py`, de tal modo que las modificaciones del código en común se realicen en un solo lugar en vez de aplicarlas a todos los *scripts* de forma individual. De este modo en cada *script* solo se realiza la modificación del sistema de partículas en base a la distribución de probabilidad correspondiente. También unas cuantas variables comunes se almacenan en un archivo de texto que usan todos los *scripts (ver abajo)*.

```
import bpy
import numpy as np
import random
from addon_utils import enable
from addon_utils import disable

# IMPORTANTE:
# 1- Seleccionar el objeto para que se aplique la fragmentación.
# 2- Asegurarse de que el proyecto está guardado en disco.
# 3- Para reiniciar, eliminar la colección creada y sus objetos con Delete Hierarchy y mostrar el objeto base.
# 4- Para que la colección se guarde en una colección determinada, seleccionarla y seguidamente seleccionar el objeto.

# Garantizar la activación de Cell Fracture
#enable(.ºbject_fracture_cell")

# Parámetros de importacion
fragcore =  bpy.data.texts['fragcore.py'].as\_module()
fragconfig =  bpy.data.texts['frag\_config.py'].as\_module()
# Parámetros de escena
context = bpy.context
view_layer = context.view_layer
scene = context.scene
objBase = context.active\_object
# Parámetros del sistema de partículas
total = 15
partSysName = "ParticleSystem"
# Parámetros de la función de distribución
n, r = 15, 0.2
# Parámetros de exportación
dir\_config = 'C:\\Users\\Asus\\Documents\\Blender\\Scripts\\frag\_config.txt'
coll\_name\_sub = "␣(Binomial␣"+ str(n) + ",␣" + str(r) + ";␣" + str(total) + ")"
file\_name\_sub = '\_Binomial\_' + str(n) + '\_' + str(r) + ";" +  str(total)
# Opciones
export = True # Realizar la exportación
export\_all = True # Exportar todos los fragmentos en un solo obj.
separate = True # Separa los fragmentos en función de la variable sepMult.
sepMult = 0.3
```

```
38
39 def param\_ext():
40     configCipher = open(dir\_config,'r')
41     directory = configCipher.readline().rstrip()
42     cache\_name = configCipher.readline().rstrip()
43     name = configCipher.readline().rstrip()
44     configCipher.close()
45     config = fragconfig.export\_config(name, cache\_name, directory)
46
47     coll\_name = config.get\_file\_name() + coll\_name\_sub
48     file\_name = config.get\_file\_name() + file\_name\_sub
49     directory = config.get\_directory()
50     return coll\_name, cache\_name, file\_name, directory
51
52
53
54 # Main
55 disable ("object\_fracture\_cell")
56 coll\_name, cache\_name, file\_name, directory = param\_ext()
57 selected\_list = [obj.name for obj in context.selected\_objects]
58 if(len(selected\_list) != 1):
59     print("Se␣debe␣seleccionar␣exactamente␣un␣objeto")
60 else:
61     particle\_systems, depsgraph, vessel = fragcore.phase\_1(selected\_list, partSysName,
          total)
62
63     # Modificación de las partículas a la distribución
64     for m in range(total):
65         x=particle\_systems[0].particles[m].location[0]
66         y=particle\_systems[0].particles[m].location[1]
67         z=particle\_systems[0].particles[m].location[2]
68         print('Particle␣first␣pass:␣',m,'␣(',x,'␣,',y,',',z,')\n')
69
70         particle\_systems[0].particles[m].location[0]=np.random.binomial(n, r)
71         particle\_systems[0].particles[m].location[1]=np.random.binomial(n, r)
72         particle\_systems[0].particles[m].location[2]=np.random.binomial(n, r)
73         x=particle\_systems[0].particles[m].location[0]
74         y=particle\_systems[0].particles[m].location[1]
75         z=particle\_systems[0].particles[m].location[2]
76         print('Particle␣second␣pass:␣',m,':␣(',x,'␣,␣',y,',␣',z,')\n')
77
78     fragcore.phase\_2(particle\_systems, depsgraph, vessel, directory, cache\_name, file\
         _name, coll\_name, total, separate, sepMult, export, export\_all)
```

Listado 4.3: *Script* definitivo para una fragmentación en base a una distribución binomial

Esencialmente se realiza el mismo proceso explicado en *4.1.4 Script base*, solamente se ha agrupado el código común en un solo *script*.

4.1.7. *Fragcore*

`Fragcore.py` es el *script* en el que se han agrupado las funciones comunes a todos los *scripts* basados en una distribución. Accede a sus funciones encapsuladas como un módulo.

- La fase uno (*phase 1*) es el preprocesamiento de datos. Corresponde a los pasos previos a la modificación del sistema de partículas.
- La fase dos (*phase 2*) realiza el resto de pasos (activación de *Cell Fracture*, separación, exportación).

4.1.8. *Frag Exporter*

`Frag_exporter.py` es un *script* de apoyo que permite exportar directamente una colección de fragmentos a posteriori, sin tener que crear una nueva fragmentación. Funciona seleccionando una colección de fragmentos visible y puede exportar o bien todos los fragmentos en el mismo archivo o bien de forma separada. Como medida de exportación masiva de archivos, también se ha creado `frag_exporter_all.py`, el cual usando los fundamentos del *script* anteriormente mencionado, exporta todo el contenido para una colección de distribución, tanto el modelo completo como cada fragmento individualmente, organizado en carpetas. Hay que seleccionar una de las colecciones de distribuciones para que funcione, y hay que asegurarse de que todas las colecciones contenidas son visibles (activar botón de ojo en el *Outliner*). Esta operación manual, pese a que es poco elegante, es debido a la complejidad para hacer que Python oculte una colección y a que el algoritmo de exportación excluye las colecciones invisibles.

4.1.9. *Frag Config*

`Frag_config.txt` es un archivo de configuración que contiene 3 variables usadas en todos los scripts, siendo las dos primeras usadas también por la modificación de `fracture_cell_setup.py` para uso universal. Hay que definir su localización en la variable `dir_config` de los ficheros que lo usan. Las tres variables son:

- **Primera línea**: Directorio en el que se exportará. Será también el mismo directorio en el que se almacenará el fichero caché que usa la modificación. Entre

directorio y directorio hay que usar doble barra (//) para que funcione correctamente.

- **Segunda línea**: Nombre del archivo caché. Es recomendable dejar el que tiene por defecto para evitar crear archivos innecesarios.
- **Tercera línea**: Nombre del archivo a exportar. Se usará tanto para la colección como para los archivos exportados).

Se apoya de una clase contenida en `frag_config.py` para organizar los datos momentáneamente, que únicamente almacena estas variables.

4.2. Métodos

Explicados los materiales requeridos para experimentar, ya podemos proceder a experimentar. Para cada distribución de probabilidad generaremos varias fragmentaciones y haremos un análisis cualitativo a través de una hipótesis del tipo de situación que podría haber llegado a romper la vasija.

Normalmente es suficiente con ejecutar el script con las variables deseables, pero algunos de los modelos 3D con los que se ha trabajado tenían las piezas separadas y para fusionarlas en un solo objeto existen dos métodos:

- **Operación *Join*** con la combinación de teclas `Ctrl + J`.
- **Modificador *boolean***: Al hacer una operación booleana de unión entre dos objetos, se fusionan en una sola.

Para que las fracturas se realicen correctamente, los modelos deben de tener un tamaño considerable, por lo que la mayoría de los modelos usados han tenido que ser escalados para obtener resultados deseables. Se puede usar como tamaño de referencia el del modelo base (`VasijaClasica`).

Capítulo 5

RESULTADOS

5.1. Fragmentaciones obtenidas

Se han realizado varias fragmentaciones con cada una de las distribuciones de probabilidad variando sus atributos. Analizaremos cualitativamente los fragmentos generados por tamaño o forma.

5.1.1. Roturas por Distribución de *Poisson*

A mayor intensidad, más pequeñas y más numerosas son las roturas que genera, con una irregularidad convincente. Puede haber casos excepcionales, como una rotura con intensidad 3 con menor superficie fracturada que otra de intensidad 2.

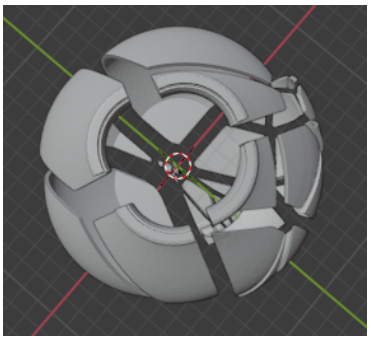

Figura 5.1: Rotura por *Poisson* de intensidad = 2 y 10 partículas

5.1.2. Roturas por Distribución Uniforme

Esta distribución ha dado más problemas que las demás. Estaba realizando exactamente las mismas fragmentaciones independientemente de las variables, hasta que he hecho varios cambios en la organización del código. Las fragmentaciones generadas por lo general son un poco imprevisibles, sin cambios notorios cuando los valores de las variables se modifican, con fragmentos relativamente numerosos. Pero por lo general a mayor media y sigma, mayor es la superficie fracturada.

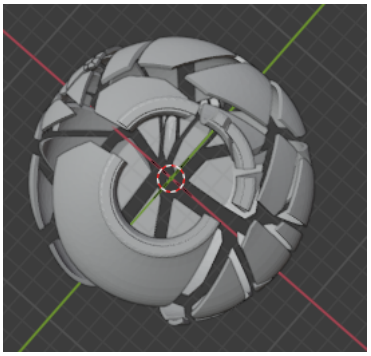

Figura 5.2: Rotura por uniforme de media = 2, sigma = 0.1 y 10 partículas

5.1.3. Roturas por Distribución *T Student*

A diferencia de la distribución exponencial *(5.1.9)*, una menor superficie fracturada que se concentra en la base y los fragmentos son más grandes y poligonales, solo generando trozos más pequeños si se activa la recursividad en algún fragmento. Para fragmentaciones debido a una fuerza en la base, Chi Cuadrado *(5.1.8)* es una alternativa más recomendable.

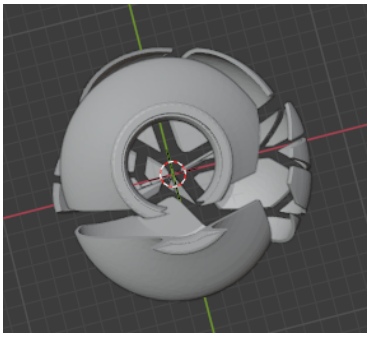

Figura 5.3: Rotura por T de *Student* de n = 8 y 15 partículas

5.1.4. Roturas por Distribución Triangular

Como sugiere el nombre, esta distribución genera roturas de forma triangular. No fragmentos de dicha forma, sino que las grietas van formando triángulos en la vasija. Estas roturas se encuentran especialmente en la base del recipiente. Los fragmentos tienen una irregularidad convincente. Usar combinaciones secuenciales de números (1, 2, 3) para la distribución genera triángulos más equiláteros, mientras que usar combinaciones no secuenciales dan lugar a triángulos escalenos.

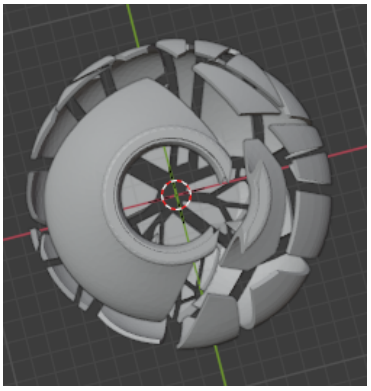

Figura 5.4: Rotura por triangular de a = 1, b = 2, c = 3 y 12 partículas

5.1.5. Roturas por Distribución Normal

Con esta distribución, los fragmentos son relativamente pequeños (se hacen más pequeños a mayor valor de media) y la superficie fracturada es relativamente grande. El valor de sigma parece ser inversamente proporcional al área de superficie con más fragmentación, por lo que si se quiere mucha superficie, es recomendable usar un valor por debajo de 0.5.

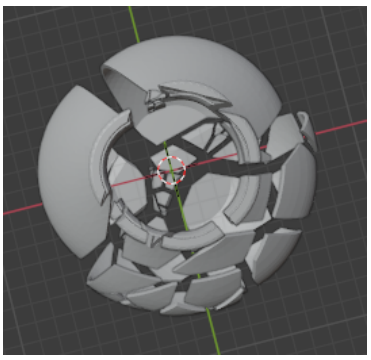

Figura 5.5: Rotura por normal de media = 5, sigma = 0.8 y 10 partículas

5.1.6. Roturas por Distribución de Pareto

Con un valor de n pequeño, la vasija se fragmenta casi completamente o bien solo afecta a una sección completa. Tiene un comportamiento bastante impredecible. Sin embargo, al aumentar n, la sección se hará mucho más pequeña, centrando las fracturas en un área concreta, como si un plano lo dividiera.

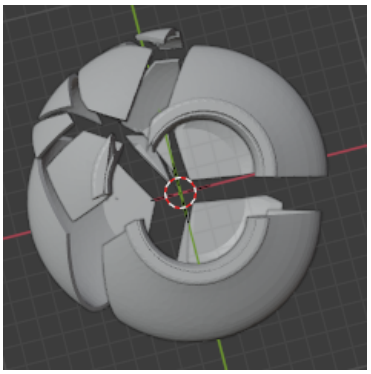

Figura 5.6: Rotura por Pareto de n = 8 y 10 partículas

5.1.7. Roturas por Distribución Beta

Lo más característico de esta distribución es que divide en tres grandes partes la vasija, de las cuales uno de los tres tercios no se fragmenta y el resto se dividen. Con un valor de sigma de 0.5, se hace una partición en 3 casi perfecta, y el tercio sin fragmentar se hace más grande al aumentar dicho valor, y con un menor valor no varía demasiado la partición principal, pero aún así hay valores de sigma que no cumplen esta observación. A mayor media, más geométrica y ordenada es la fragmentación resultante de las zonas agrietadas.

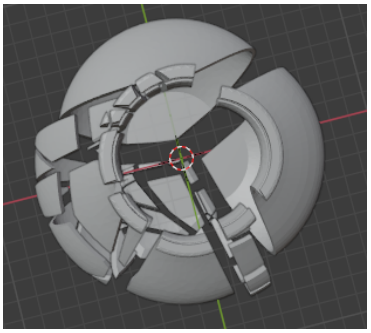

Figura 5.7: Rotura por Beta de media = 5, sigma = 0.5 y 10 partículas

5.1.8. Roturas por Distribución Chi Cuadrado

Como en otras roturas, a mayor valor de n, más pequeños y numerosos son los fragmentos, los cuales tienden a agruparse en la base. El centro de la base siempre acaba agrietado y genera una fragmentación relativamente irregular, por lo que sería una rotura ideal para situaciones como dejar caer una vasija desde una cierta altura.

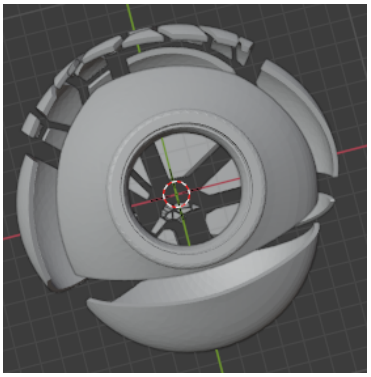

Figura 5.8: Rotura por Chi Cuadrado de n = 4 y 20 partículas

5.1.9. Roturas por Distribución Exponencial

Esta distribución ha generado fragmentos muy regulares y perfectamente distribuidos entorno a la base de la vasija, con cortes relativamente perfectos y sin curvas, por lo que es poco realista. Aparentemente, a mayor valor de alfa, más alargados son los fragmentos. También genera 3 fragmentos grandes exactamente iguales como la distribución Beta.

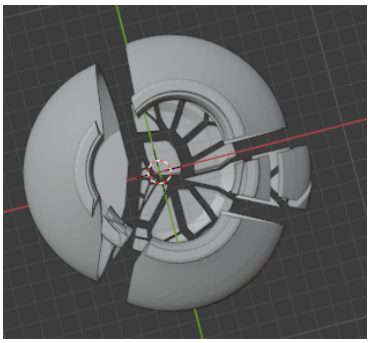

Figura 5.9: Rotura por Exponencial de alfa = 10 y 15 partículas

5.1.10. Roturas por Distribución Binomial

A pesar de que a simple vista pueden parecer roturas convincentes, en ocasiones con un menor valor de media hace la fragmentación se observa una cierta regularidad, dividiendo en 4 partes iguales la vasija entera. Valores más grandes (más de 10) hacen la fragmentación más irregular. El valor de sigma no parece realizar efectos notorios en la fragmentación.

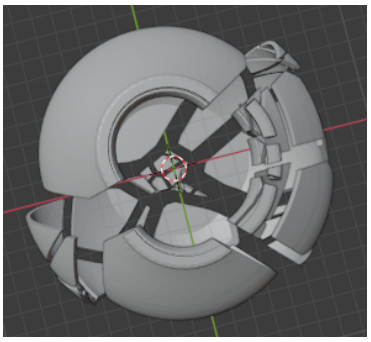

Figura 5.10: Rotura por Binomial de media = 10, sigma = 0.5 y 15 partículas

5.1.11. Roturas por Distribución Gamma

El valor de gamma parece determinar el tamaño y dispersión de los fragmentos. Es recomendable usar un valor pequeño (5 o menos) para una fragmentación razonable, ya que genera una mayor dispersión con un valor más grande, y por tanto menos fragmentos y más grandes. Con un valor de forma pequeño la distribución de las roturas se hace en casi toda la vasija, pero al aumentarlo se concentrar en un lateral.

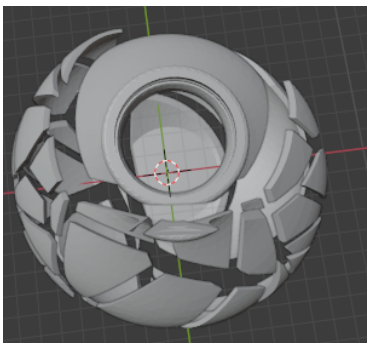

Figura 5.11: Rotura por Gamma de forma = 2, gamma = 6 y 10 partículas

5.1.12. Roturas por Distribución Logaritmo Normal

Dado el tamaño de los modelos 3D usados, para que funcione bien esta distribución el valor de la media debe de ser menor de 5, de lo contrario las partículas no se generarán dentro de la vasija. Aumentar el valor de sigma parece añadir algunas roturas más que cuando tiene un valor pequeño (alrededor de 0.2). La fragmentación y tamaño de los fragmentos también parece ser inversamente proporcional a la media.

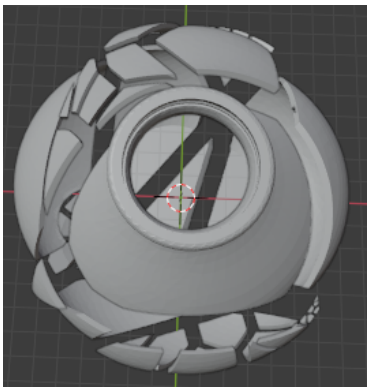

Figura 5.12: Rotura por Logaritmo Normal de media = 2, sigma = 0.99 y 15 partículas

5.1.13. Roturas por Distribución Geométrica

Por último, esta distribución genera fragmentos más ajustados a la base y se distribuye más por el resto del modelo cuanto menor sea el valor de p. Sin embargo, los fragmentos generados tienen un a distribución relativamente regular. Funciona mejor con valores menores a 0.5, ya que con valores mayores se genera una distribución mucho más regular y se crean solo muy cerca de la base, creando por consiguiente una fragmentación poco realista.

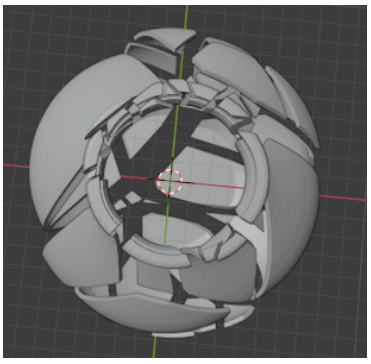

Figura 5.13: Rotura por Geométrica de p = 0.05 y 10 partículas

5.2. Roturas de vasijas con asas

Como ya de dijo en *4.2 Métodos*, hay dos formas de fusionar mallas. **La más recomendable de las dos es mediante la unión booleana**. Al usar *join*, genera problemas tales como que las piezas de las asas se fragmenten de forma separada o en ciertas distribuciones se pierda información, eliminando las piezas correspondientes a un área cercana a las asas. Todo esto es debido a que estas vasijas vienen con las piezas separadas (o regiones) en el modelo *obj* en vez de unidas en una sola malla, lo cual hace que aunque se consideren un solo modelo 3D con la operación *join*, se hagan las roturas como objetos aparte. Sin embargo, con la unión booleana estos problemas se ausentan, generando roturas mucho más razonables, por lo que de esta manera se pueden realizar fragmentaciones de vasijas compuestas de varias piezas.

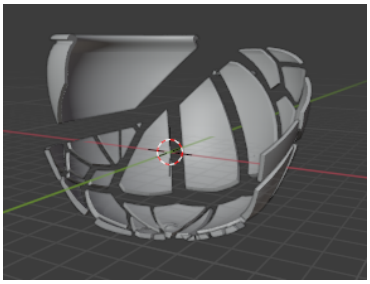

Figura 5.14: Rotura incorrecta mediante Chi cuadrado de una vasija con asas unida mediante *join*

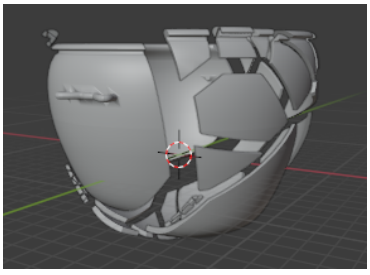

Figura 5.15: Rotura correcta mediante Chi cuadrado de una vasija con asas unida mediante unión booleana

5.3. Otras consideraciones

- Existen otros modelos que poseen una pieza exterior y otra interior. Fusionarlos con la unión booleana da buenos resultados la mayoría de las veces.
- Los modelos muy alargados, mención especial las vasijas con cuello largo, no dan resultados fiables. Fragmentar estos modelos dan lugar a piezas inexistentes provenientes de la envolvente convexa del algoritmo o bien les faltan piezas.

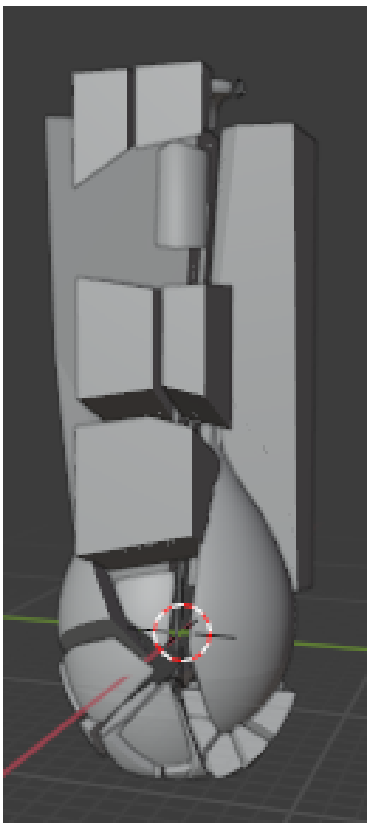

Figura 5.16: Rotura incorrecta mediante distribución triangular de una vasija alargada.

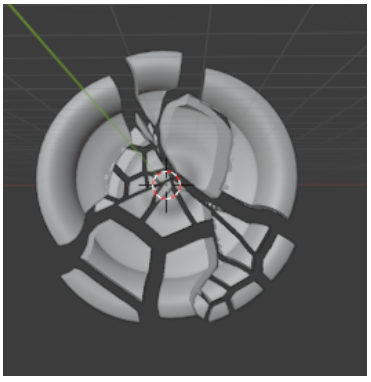

Figura 5.17: Rotura correcta mediante distribución triangular de una vasija compuesta por una pieza interior y exterior.

Capítulo 6

CONCLUSIONES

6.1. Análisis de resultados

- Se han creado un total de 13 scripts en *Python*, y adicionalmente 2 scripts de apoyo.
- Se ha modificado exitosamente un script interno de Blender para un funcionamiento adecuado de los scripts.
- Se han generado las fragmentaciones esperadas usando 6 modelos 3D distintos de vasijas, incluyendo tanto vasijas básicas como vasijas con piezas adicionales (asas).
- Se han completado todos los objetivos marcados en los tiempos previstos.

6.2. Conclusiones adicionales

Este proyecto de 11 meses de duración ha sido un éxito. Se han cumplido la mayoría de objetivos marcados. Los *scripts* son aplicables a cualquier modelo 3D de un tamaño considerable, incluyendo los compuestos por varias piezas ya que la unión booleana permite unirlas y que funcione como un solo modelo sin problemas. Si bien los resultados han sido satisfactorios, existe un margen de mejora en varias categorías que podrían plantearse en trabajos derivados:

- Hacer el algoritmo robusto al tamaño del modelo 3D, ya que la mayoría han tenido que ser escalados para un funcionamiento adecuado.

- Se han usado la mayoría de distribuciones de probabilidad principales, pero existen otras menos comunes que no se han usado, como la distribución de Gauss inversa, que a lo mejor podría generar resultados interesantes.
- Haber comparado roturas reales con las generadas por el programa. Se podría haber hecho pero no se tuvo en cuenta durante el desarrollo.

Bibliografía

Julia Máxima Uriarte. Historia del cine, 2020. URL https://www.caracteristicas.co/historia-del-cine/. Comprobado en 2022-08-07.

Kseniya Serebrennikova. The history of vfx: How the morphing effect evolved, 2020. URL https://www.rebelway.net/history-of-vfx-morphing/. Comprobado en 2022-08-07.

Andy Stout. The history of vfx - part one: From mary queen of scots to citizen kane + part two: The model men + part three: Digital killed the optical star + part iv: Concerning hobbits and other creatures, 2013. URL https://www.redsharknews.com/production/item/530-the-history-of-vfx-part-one-from-mary-queen-of-scots-to-citizen-kane. Comprobado en 2022-08-07. Se han consultado las partes 1-4.

Kieron Moore. Best vfx software in 2022, 2022. URL https://www.techradar.com/best/best-vfx-software. Comprobado en 2022-08-10.

Tammy Haggard. Top 10 ultimate computer-generated imagery software tools, 2022. URL https://www.techyv.com/article/top-10-ultimate-computer-generated-imagery-software-tools/. Comprobado en 2022-08-11.

Loeïz Glondu, Maud Marchal, and Georges Dumont. Real-time simulation of brittle fracture using modal analysis. ieee transactions on visualization and computer graphics. *Institute of Electrical and Electronics Engineers*, 19:201–209, 2012. doi: ff10.1109/TVCG.2012.121ff.ffhal-00752372f.

Saty Raghavachary. Fracture generation on polygonal meshes using voronoi polygons. In *ACM SIGGRAPH 2002 Conference Abstracts and Applications*, SIGGRAPH '02, page 187, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135254. doi: 10.1145/1242073.1242200. URL https://doi.org/10.1145/1242073.1242200.

Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Real time dynamic fracture with volumetric approximate convex decompositions. *ACM Trans. Graph.*, 32(4), jul 2013. ISSN 0730-0301. doi: 10.1145/2461912.2461934. URL https://doi.org/10.1145/2461912.2461934.

Ladislav Kavan, Ivana Kolingerova, and Jiri Zara. Fast approximation of convex hull. volume 8, pages 101–104, 01 2006.

Carlos J. Gil Bellosta. Introducción a la probabilidad y la estadística para científicos de datos, 2021. URL https://datanalytics.com/libro_estadistica/index.html. Comprobado en 2022-07-29.

Paula Rodó. Distribución de poisson, 2020. URL https://economipedia.com/definiciones/distribucion-de-poisson.html. Comprobado en 2022-07-31.

Bautista H Mendoza. Probabilidad y estadística. universidad nacional de colombia, 2002. URL http://www.virtual.unal.edu.co/cursos/ciencias/2001065/. Comprobado en 2022-08-02. Licencia: Creative Commons BY-NC-ND.

Javier Marco Sanjuán. Distribución binomial, 2017. URL https://economipedia.com/definiciones/distribucion-binomial.html. Comprobado en 2022-08-02.

Alexander Holmes, Barbara Illowsky, Susan Dean, and Kevin Hadley. *Introducción a la estadística empresarial*. OpenStax, 2022. ISBN 978-1-951693-49-7.

Raul E. Lopez Briega. Distribuciones de probabilidad con python, 2016. URL https://relopezbriega.github.io/blog/2016/06/29/distribuciones-de-probabilidad-con-python/. Comprobado en 2022-07-31.

Paula Rodó. Distribución normal, 2019a. URL https://economipedia.com/definiciones/distribucion-normal.html. Comprobado en 2022-08-15.

Guillermo Westreicher. Ley de pareto, 2020. URL https://economipedia.com/definiciones/ley-de-pareto.html. Comprobado en 2022-08-18.

Paula Rodó. Distribución t de student, 2019b. URL https://economipedia.com/definiciones/distribucion-t-de-student.html. Comprobado en 2022-08-19.

Luis Benites. Introducción a la distribución triangular, 2022. URL https://statologos.com/distribucion-triangular/. Comprobado en 2022-08-23.

Mr. Cheebs. Ep 6: A full guide to cell fracturing, 2020. URL https://www.youtube.com/watch?v=T2nsntEzlAw. Vídeo de YouTube. Comprobado en 2022-02-23.