Escuela Politécnica Superior de Jaén
Grado en Ingeniería Informática

Universidad de Jaén
Escuela Politécnica Superior
de Jaén

TRABAJO FIN DE GRADO

SIMULACIÓN DE FRACTURAS EN MODELOS DE VÓXEL MEDIANTE ACELERACIÓN GPU

Alumno

José María Cruz Lorite

Tutor

Antonio Jesús Rueda Ruiz
(Departamento de Informática)

Septiembre, 2020

Universidad de Jaén

Departamento de Informática

Don Antonio Jesús Rueda Ruiz, tutor del Trabajo Fin de Grado titulado: '**Simulación de fracturas en modelos de vóxel mediante aceleración gpu**', que presenta Don José María Cruz Lorite, otorga el visto bueno para su entrega y defensa en la Escuela Politécnica Superior de Jaén.

Jaén, Septiembre de 2020

El alumno:

CRUZ LORITE JOSE MARIA - 26515840Z

Firmado digitalmente por CRUZ LORITE JOSE MARIA - 26515840Z
Fecha: 2020.09.09 16:01:10 +02'00'

José María Cruz Lorite

El tutor:

Firmado digitalmente por RUEDA RUIZ ANTONIO JESUS - 33395227D
Fecha: 2020.09.09 14:38:56 +02'00'

Antonio Jesús Rueda Ruiz

Agradecimientos

Al Dr. Antonio Jesús Rueda Ruiz por hacer este año de pandemia algo más llevadero: en la asignatura de Programación Hardware, con su conocimiento y en el desarrollo de este trabajo, con su inestimable ayuda.

A todos los compañeros que han sufrido y disfrutado la Universidad junto conmigo en los últimos cinco años.

Y por último, a mi familia, sin ellos este Trabajo Fin de Grado nunca se hubiese presentado.

FICHA DEL TRABAJO FIN DE GRADO	
Titulación	Grado en Ingeniería Informática
Modalidad	Trabajo Teórico/Experimental
Especialidad	Tecnologías de la Información
Mención	Sistemas Gráficos
Idioma	Español
Tipo	Específico
TFT en equipo	No
Autor/a	José María Cruz Lorite
Fecha de asignación	13/11/2019
Descripción corta	Existen métodos sencillos para simular fracturas en modelos de vóxel basados en el cálculo del diagrama de Voronoi discreto. Con este método pueden simularse roturas fácilmente de objetos cerámicos o de cristal. El problema de este método es que tiene un coste computacional importante, lo que las invalida para aplicaciones de tiempo real. Se propone realizar una implementación GPU que permita el cálculo de roturas en pocos milisegundos.

NORMAS APLICADAS EN ESTE DOCUMENTO	
LOCALES	
TFT-UJA:2017	Normativa de Trabajos Fin de Grado, Fin de Máster y otros Trabajos Fin de Título de la Universidad de Jaén (Normativa marco UJA aprobada en Consejo de Gobierno)
TFT-EPSJ:2017	Normativa sobre Trabajos Fin de Grado y Fin de Máster en la Escuela Politécnica Superior de Jaén (Normativa EPSJ aprobada en Junta de Escuela)
TFT-EPSJ	Criterios de evaluación y normas de estilo para TFG y TFM de la Escuela Politécnica Superior de Jaén
NACIONALES E INTERNACIONALES	
ISO 2145:1978	Documentación - Numeración de divisiones y subdivisiones en documentos escritos
UNE 50132:1994	Traducción de la ISO 2145
APA 6ª edición	Estilo de referencias y citas de APA (American Psychological Association)

Contenido

Índice de ilustraciones

Índice de tablas

1 INTRODUCCIÓN

1.1 Marco del trabajo

El proyecto Kalathos del Instituto de Investigación en Arqueología Ibérica (IAAI) de la Universidad de Jaén tiene dos objetivos fundamentales. El primero es ayudar a los arqueólogos a manejar los fragmentos de los objetos cerámicos encontrados. El segundo es tratar de clasificar los distintos tipos de vasijas existentes.

Desarrollar una herramienta que sugiriera la vasija de donde proviene un fragmento determinado sería un gran hito dentro el proyecto Kalathos. Para ello, están utilizando técnicas de aprendizaje profundo para implementar un clasificador de fragmentos.

Dado que entrenar una red neuronal con fragmentos escaneados reales no sería viable, su idea es utilizar fragmentos artificiales resultado de roturas simuladas de los modelos 3D (ilustración 1.1).

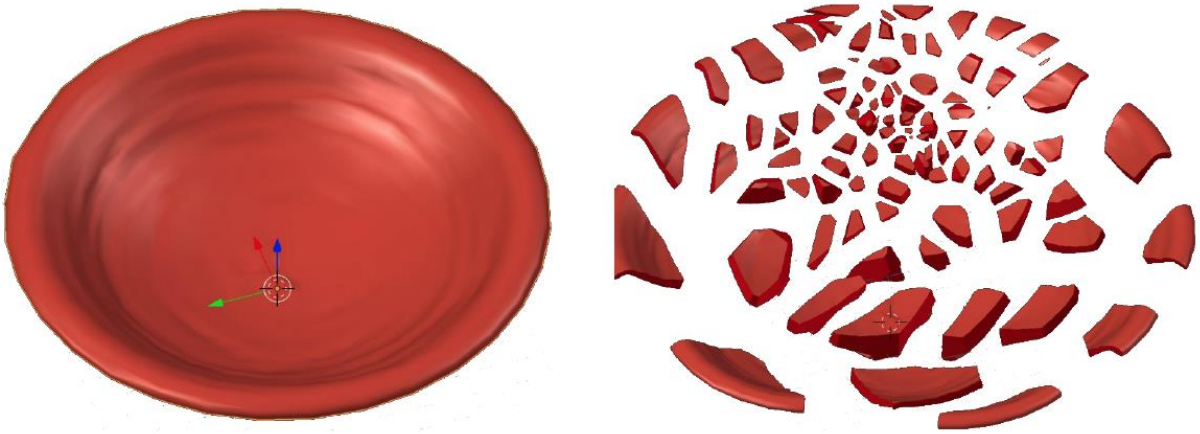

Ilustración 1.1. Simulación de una rotura para generar fragmentos con los que entrenar una red neuronal. ***Fuente:*** IAAI.

Se parte de un algoritmo basado en el diagrama de Voronoi discreto 3D (DVD) para simular roturas implementado en Python. Sin embargo, el cálculo del DVD tiene un coste computacional elevado, lo que lo invalida para aplicaciones en tiempo real. El principal objetivo de este Trabajo Fin de Grado es optimizar el algoritmo original vía GPU para calcular el DVD en tan solo unos pocos milisegundos.

1.2 Objetivos del trabajo

El objetivo principal de este Trabajo Fin de Grado consiste en mejorar el rendimiento de la solución ya implementada en Python, acelerando el algoritmo original vía GPU. En consecuencia, será necesario familiarizarse con la programación en GPUs estudiando las distintas plataformas/tecnologías disponibles.

Los objetivos a cumplir en el planteamiento inicial son los siguientes:

1. Estudiar la programación en GPU mediante OpenGL/ComputeShaders.
2. Implementar la rotura de objetos de vóxel calculando el DVD.
3. Desarrollar métodos de rotura adicionales.
4. Implementar un pequeño visualizador, importador y exportador de fragmentos.

1.3 Metodología a desarrollar

Debemos adecuar los objetivos planteados en la sección anterior con la metodología especificada en la propuesta inicial del Trabajo Fin de Grado. La metodología a desarrollar es la siguiente:

1. Estudiar la programación de GPUs mediante OpenGL/ComputeShaders.
2. Estudiar la implementación de rotura de modelos de vóxel mediante Voronoi discreto tradicional (existe una implementación simple en Python).
3. Realizar una implementación en GPU del método.
4. Integrar la implementación en un pequeño entorno que permita visualizar el modelo antes y después de la rotura, permitiendo cierto nivel de parametrización. Incluir exportación de los fragmentos obtenidos.
5. Implementar formas adicionales de rotura.
6. Realizar pruebas con diferentes modelos y tomar tiempos de rotura.
7. Realizar la memoria del trabajo realizado.

1.4 Tecnologías utilizadas

Software, tecnologías o librerías que se han utilizado:

- **CMake:** herramienta utilizada para controlar el proceso de compilación. Se posee experiencia previa y es multiplataforma.
- **GLFW:** librería *Open Source* para el desarrollo en OpenGL, OpenGL ES y Vulkan. Se posee experiencia previa y es multiplataforma.
- **Dear Imgui:** librería c++ sin dependencias externas e integrable con GLFW utilizada para crear interfaces gráficas de usuario.
- **C++:** lenguaje de programación utilizado para el desarollo de la aplicación.
- **OpenGL:** API multiplataforma para la generación de gráficos por computador. Se posee experiencia previa y es multiplataforma.
- **GLSL:** lenguaje de programación de *shaders* de OpenGL.

2 ANTECEDENTES

En la siguiente sección se realiza una breve revisión de la bibliografía existente sobre simulación de fracturas. Los distintos modelos se han agrupado entre basados en físicas y procedimentales, como hizo Ajees (2009) en su tesis.

2.1 El término *fracture*

Como afirman Hahn y Wojtan (2016), agregar efectos de fractura en simulaciones por ordenador ha recibido mucha atención por parte de la comunidad de la informática gráfica, tanto en investigaciones recientes como en aplicaciones industriales. Pero, ¿a qué se refieren con efectos de fracturas? A veces en la literatura en lugar de *fracture* se utilizan términos como *cracking*, *breaking*, *shattering* o *cutting*, los cuales pueden resultar confusos para un neófito. Acerca de esta cuestión Glondu *et al*. (2012) explicaron lo siguiente:

> "El término *fracture* posee muchos significados en la literatura, como: *tearing* (progresiva separación de un material debido a las tensiones de tracción), *breaking* o *shattering* (separación explosiva de un cuerpo en múltiples fragmentos), *cutting* (separación controlada de un material), *cracking* (formación de roturas dentro de un cuerpo, pero sin separarlo) y *crumbling* (separación del objeto en fragmentos pequeños normalmente resultado de fricciones o del deterioro)".

En este trabajo se utilizará el término fracturación/fracturar para referirnos genéricamente a cualquiera de los fenómenos antes descritos.

2.2 Modelos basados en físicas

Los modelos basados en físicas producen fragmentos físicamente realistas mientras que los modelos procedimentales se valen de otras técnicas para producir fragmentos estéticamente aceptables, normalmente consumiendo menos recursos.

Las simulaciones basadas en físicas han ganado una importancia cada vez mayor en muchos campos de la informática gráfica: motores de videojuegos, animación o realidad virtual (Crawford, 2012). Téngase en cuenta que este enfoque suele requerir *discretizar* los objetos en *voxels* o en mallas de tetraedros (Martinet *et al*., 2004), esto suele conllevar mayores requisitos de memoria y capacidad de computo.

2.2.1 Método de elementos finitos

El método clásico MEF (método de elementos finitos), método muy extendido para resolver problemas de ingeniería, es muy utilizado para generar deformaciones y fracturas basadas en físicas, sin embargo, la precisión del cálculo de MEF exige el mantenimiento de una malla con mucha precisión, esto puede resultar complicado y consumir demasiados recursos (Liu *et al*., 2011).

2.2.2 Método de elementos de frontera

El método de elementos de frontera enfoca todos los cálculos en la superficie de los objetos 3D. Las fracturas se representan como superficies adicionales en la malla BEM (del inglés *boundary element method*) (Hahn y Wojtan, 2016). El método de elementos de frontera puede ser más eficiente que otros métodos, MEF incluido,

en recursos computacionales para problemas donde hay una proporción superficie/volumen baja (Katsikadelis, 2002).

En 2016 Hahn y Wojtan publicaron un método basado en BEM para la simulación de fracturas de objetos rígidos. Como se puede apreciar en la ilustración 2.2 consiguieron animar una fractura por completo en una escena con una esfera, una columna y el conejo de Stanford, en aproximadamente 16 minutos.

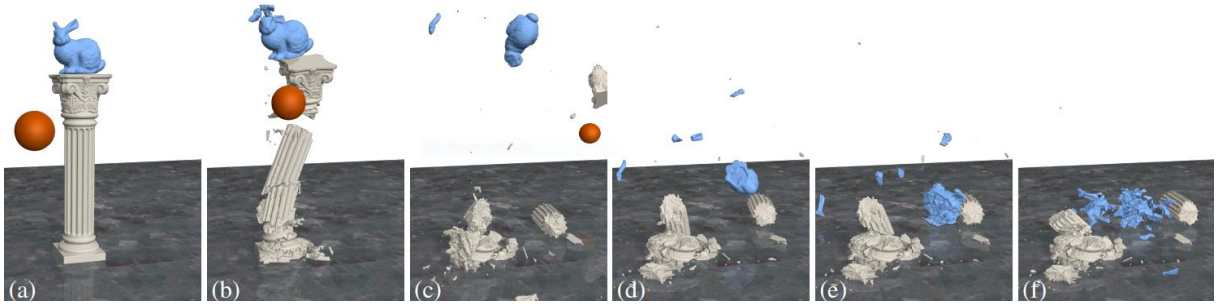

Ilustración 2.1. Destrucción de varios objetos utilizando un método basado en BEM. ***Fuente:*** Hahn y Wojtan (2016).

2.2.3 Métodos sin malla

En los últimos años se han desarrollado métodos que no utilizan mallas (*meshless methods*) con el objetivo de evitar parte de las dificultades de otros métodos y se han aplicado en el ámbito de la informática gráfica con gran éxito (Liu *et al*., 2011). Los métodos sin malla son excepcionalmente simples, pero proporcionan soluciones para ciertas clases de ecuaciones lo suficientemente precisas como para suponer una alternativa al método de elementos finitos y al método de elementos de frontera (Crawford, 2012).

En 2011 Liu *et al*. presentaron un método sin malla al que ellos calificaron como “más ajustado a fracturas de objetos frágiles, comparado con otros métodos sin malla anteriores” debido a que no necesitaba restricciones de *timestep* (lapso de tiempo) para garantizar que no hubiese aberraciones visuales. En la ilustración 2.2 se muestra un ejemplo de los resultados que obtuvieron.

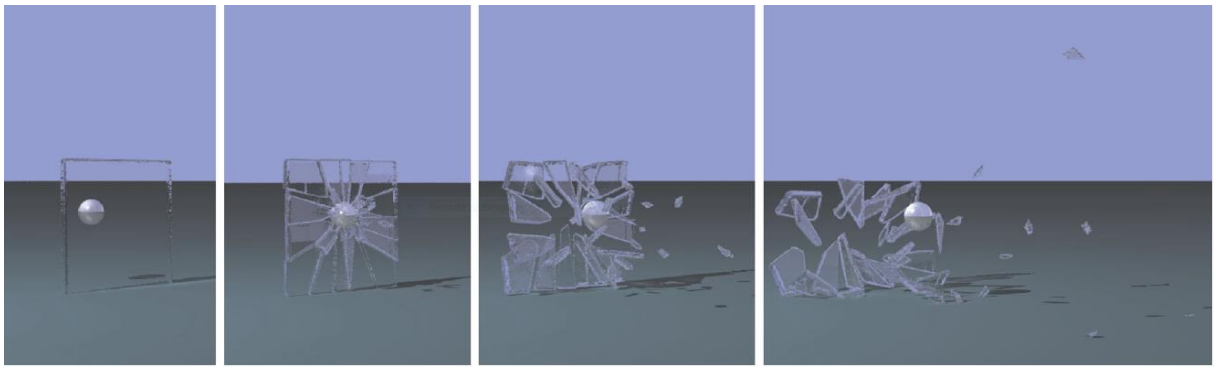

Ilustración 2.2. Fracturación de un cristal golpeado por una esfera. ***Fuente:*** Liu *et al.* (2011).

2.2.4 Modelo punto-masa

En los comienzos de la animación de fracturas se utilizaron conexiones punto-masa para modelar fracturas, el proceso de fractura se realizaba eliminando algunas de las conexiones de los modelos (Domaradzki y Martyn, 2018). En la ilustración 2.3 se representan dos tetraedros como puntos-masa y sus conexiones (caras compartidas) como restricciones (Smith *et al.* 2001).

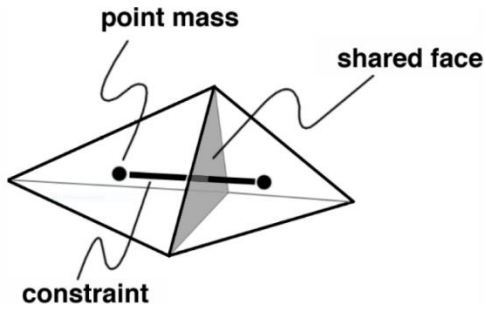

Ilustración 2.3. Restricción entre dos tetraedros (punto-masa) que comparten una cara. ***Fuente:*** Smith *et al.* (2001).

Smith *et al*. (2001) consiguieron modelar fracturas de cuerpos rígidos de forma rápida y controlable. El objeto que se va a fracturar es representado como un conjunto de puntos masa (del inglés *point masses*) conectados con restricciones de preservación de la distancia. En los resultados expusieron que “el tiempo total que se necesita para romper y reconstruir modelos de tamaño moderado (varios miles de restricciones) es solo de unos minutos”. Esto fue en 2001, probablemente con el hardware actual se podrían conseguir resultados mucho mejores. En ilustración 2.4 se muestra el resultado obtenido fracturando un cuenco.

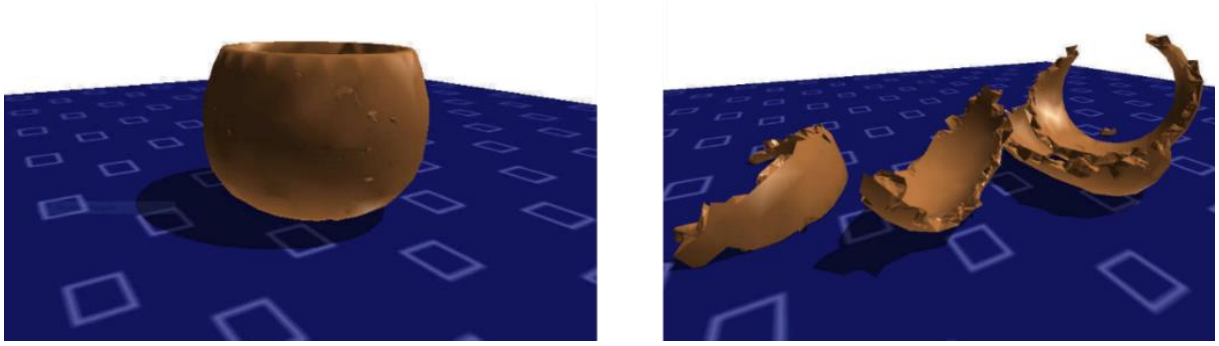

Ilustración 2.4. Fracturación de un cuenco utilizando un modelo punto-masa. ***Fuente:*** Smith *et al.* (2001).

2.2.5 Modelo resorte-masa

Un modelo resorte-masa puede entenderse como un grafo (como el de la ilustración 2.5) donde cada vértice (masa) es un nodo y cada arista un resorte. Los modelos resorte-masa proporcionan un método simple pero práctico para modelar una amplia variedad de fenómenos. Terzopoulos y Fleischer (1988) introdujeron un método para animar tres comportamientos inelásticos: viscoelasticidad, plasticidad y fracturación. Hirota *et al*. (1998) consiguieron emular el proceso de secado del barro empleando una malla resorte-masa.

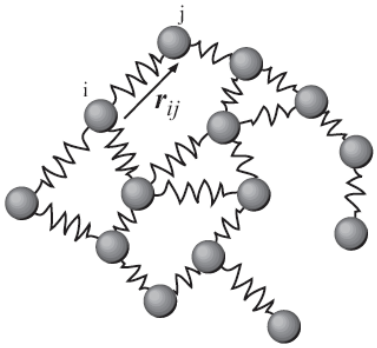

Ilustración 2.5. Grafo de un sistema resorte-masa. ***Fuente:*** https://en.wikipedia.org/wiki/Spring_system (recuperado en Julio de 2020)

2.3 Modelos procedimentales

Los modelos procedimentales son una aproximación a la fracturación de objetos sin basarse en físicas. Estos modelos pueden ser más eficientes a cambio de ser físicamente más imprecisos (Ajees, 2009). Los modelos basados en físicas suelen permitir poco control sobre la propagación de las fracturas (Martinet *et al*., 2004), en

cambio los modelos procedimentales suelen proveer de algún mecanismo para controlar las características de los fragmentos generados.

Martinet *et al.* (2004) presentaron un método procedimental que otorgaba pleno control sobre el tamaño y la forma de los fragmentos generados. Dados unos pocos parámetros se podía crear una enorme variedad de tipos de roturas y generar fragmentos de diferentes formas.

Ajees en su tesis de 2009, desarrolló un método que permitía generar fracturas de objetos rígidos para videojuegos en tiempo real. Su método consigue generar fragmentos realistas utilizando texturas de probabilidad (ilustración 2.6) que definen las áreas frágiles por donde las fracturas se irán propagando. Se pueden crear texturas de probabilidad para simular varios materiales y deben ser diseñadas con anterioridad por el usuario.

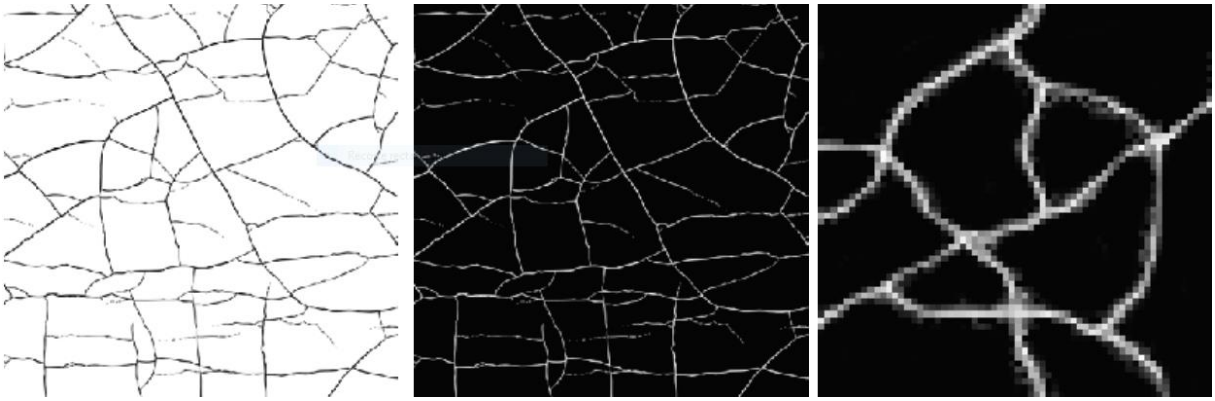

Ilustración 2.6. Foto de una fractura en cemento y las texturas de probabilidad asociadas. ***Fuente:*** Ajees (2009)

Más recientemente Domaradzki y Martyn (2018) propusieron un algoritmo novedoso para fracturar objetos carcasa (del inglés *shell object; s*on objetos frágiles, huecos y con una superficie fina, como jarrones, vasos o cerámicas). Su método mejora los existentes basados en patrones de fracturas y utiliza SVOs (del inglés *sparse voxel octrees*) para representar los objetos con un gran nivel de detalle y de forma eficiente. Utilizan diagramas de Voronoi para generar patrones de fractura que pueden ser calculados en tiempo real. En la ilustración 2.7 se presenta una simulación de la fracturación de un florero.

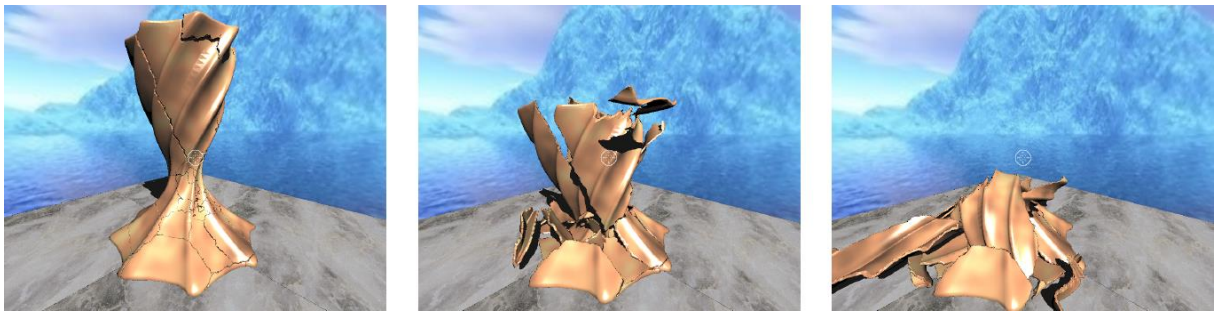

Ilustración 2.7. Fracturación de un objeto basado en diagramas de Voronoi. ***Fuente:*** Domaradzki y Martyn (2018)

2.4 Resumen

En la tabla 2.1 se ofrece un resumen de los de los modelos desarrollados por los autores que han servido como referencia para la realización de este apartado.

Autores	Tipo	Método	Tiempo real
Ajees (2009)	Procedimental	Textura de probabilidad	Sí
Domaradzki y Martyn (2018)	Procedimental	Diagramas de Voronoi	Sí (50 ms)
Glondu *et al.* (2012)	Basado en físicas	Análisis modal y propagación de fracturas basado en energía	Sí
Hahn y Wojtan (2016)	Basado en físicas	BEM	No
Liu *et al*. (2011)	Basado en físicas	MLPG 5	No
Martinet *et al.* (2004)	Procedimental	*Hybrid Tree*	No
O'Brien y Hodgins (1999)	Basado en físicas	FEM	No
Smith *et al*. (2001)	Basado en físicas	Modelo punto-masa	No

Tabla 2.1. Comparación de la bibliografía

En general los modelos procedimentales producen resultados físicamente poco realistas requiriendo pocos recursos. De forma homóloga, los modelos basados en físicas producen resultados físicamente muy realistas requiriendo muchos recursos. Sin embargo, algunos modelos basados en físicas pueden ser empleados en

aplicaciones que requieren tiempo real (p. ej. Glondu *et al*., 2012); normalmente lo consiguen realizando algún tipo simplificación o empleando alguna técnica híbrida.

3 MODELOS DE VÓXEL

Un modelo de vóxel es una cuadrícula cartesiana en un espacio tridimensional con dimensiones fijas. A cada una de las cuadrículas se le denomina vóxel (*volumetric element*). Conceptualmente podemos entender un modelo de vóxel como una matriz tridimensional de cubos que fragmentan el espacio de forma similar a como lo hace un *octree*.

Agrupando conjuntos de vóxeles con el mismo valor podemos representar objetos de forma aproximada. Como se muestra en la ilustración 3.1, este concepto se puede extender para representar regiones de Voronoi como grupos de vóxeles equivalentes.

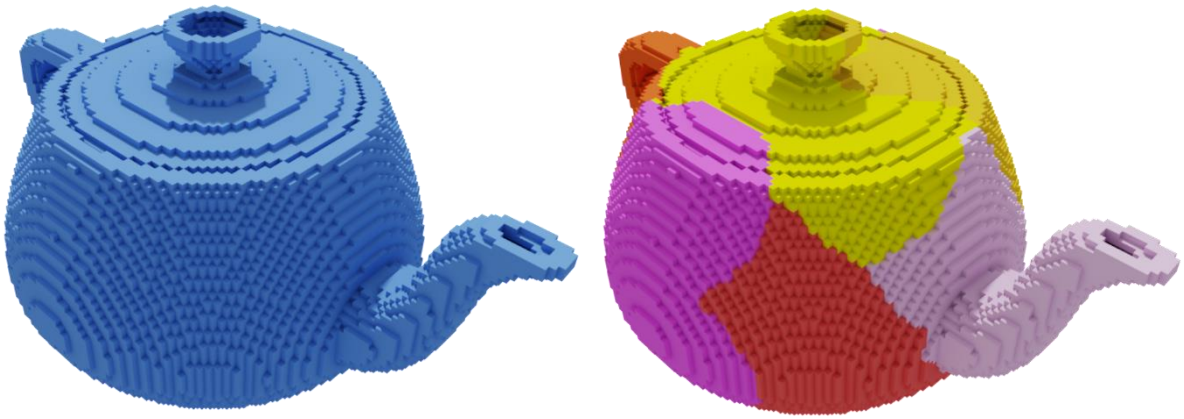

Ilustración 3.1. Tetera de Utah como modelo de vóxel con dimensiones $\mathbf{126} \times \mathbf{81} \times \mathbf{61}$. A la izquierda el modelo original y a la derecha fragmentado.

3.1 Estados de un vóxel en el cálculo del DVD

Un vóxel puede estar en tres estados distintos:

- **Vacío**: se representa con un 0, significa que el vóxel no pertenece al modelo. Por lo tanto, se puede excluir del proceso de fragmentación.

- **Libre**: se representa con un 1, significa que el vóxel pertenece al modelo y que aún no está asignado a una partición concreta.
- **Ocupado**: un valor mayor que 1. Este estado se utiliza para identificar cada partición cuando se calcula el DVD.

Inicialmente todos lo vóxeles que componen un modelo están libres. Asignar colores a los vóxeles será la forma de producir fragmentar

3.2 Vecindad de un vóxel

La topología de un modelo de vóxel es explicita. Dado un vóxel cualquiera, conocemos todos los vóxeles que conforman su vecindad. Existen varias maneras de definir el conjunto de vecindad. En la ilustración 3.2 se muestran distintas formas de definir la vecindad de un píxel en una cuadrícula cartesiana bidimensional. Para un vóxel se pueden definir de forma análoga, pero en tres dimensiones.

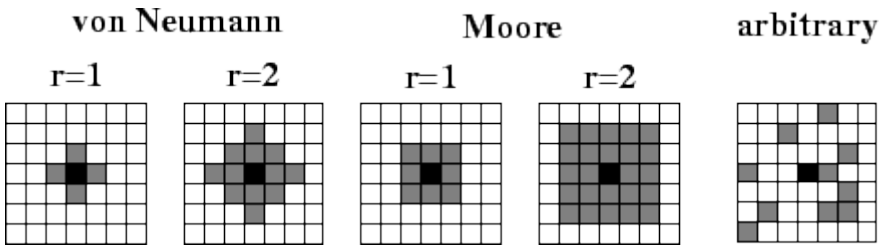

Ilustración 3.2. Ejemplo de distintas vecindades en una cuadrícula cartesiana 2D. ***Fuente: http://www.jweimar.de/jcasim/main/node5.html*** (recuperado en Agosto de 2020)

3.3 Representación en memoria

En la implementación de este trabajo se han utilizado dos representaciones en memoria distintas: matricial y tabular. En la tabla 3.1 se detallan las características de ambas representaciones.

Representación matricial	Representación tabular

• Matriz tridimensional $\mathbf{X} \times \mathbf{Y} \times \mathbf{Z}$. Cada vóxel es un $\boldsymbol{color}$ almacenado en una posición $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ de la matriz. • Alto consumo de memoria • Acceso en $\boldsymbol{O(1)}$ • Acceso a la vecindad en $\boldsymbol{O(1)}$	• Tabla/lista únicamente de los vóxeles no vacios. Cada vóxel es una 4-upla $(x, y, z, color)$. • Bajo consumo de memoria (si hay muchos vóxeles no vacíos, no) • Acceso en $O(n)$ • Acceso a la vecindad en $O(n)$

Tabla 3.1. Comparativa entre la representación matricial y tabular.

La representación matricial resulta ser óptima cuando se requieren muchos accesos a la estructura de datos. Los algoritmos que necesiten acceder a los vóxeles y sus vecindades de forma eficiente, optarán por la representación matricial. En contraposición, la representación tabular será eficiente cuando no se necesite acceder a la información del modelo, por ejemplo, para almacenamiento en disco.

4 DIAGRAMA VORONOI DISCRETO

En general, si se da un conjunto finito S (conjunto de semillas o generadores) de objetos p_i en un espacio M, calcular el diagrama de Voronoi (DV) de S significa dividir M en regiones, $R(p_i, S)$ (región de Voronoi), de tal manera que $R(p_i, S)$ contiene todos los puntos de M que están más cerca de p_i que de cualquier otro objeto $p_j \in S$ (Klein, 1989).

El diagrama de Voronoi discreto 3D (DVD) es una variante natural del diagrama de Voronoi “normal” (Van der Putte y Ledoux, 2011). Como se muestra en la ilustración 4.1 el DV emplea una representación matemáticamente exacta (puntos, líneas, planos, etc…). El DVD, por el contrario, utiliza espacios discretizados (aproximaciones inexactas). Debido a que los modelos de vóxel son espacios discretizados, el DVD se ajusta mucho mejor para trabajar con ellos que el DV “normal”.

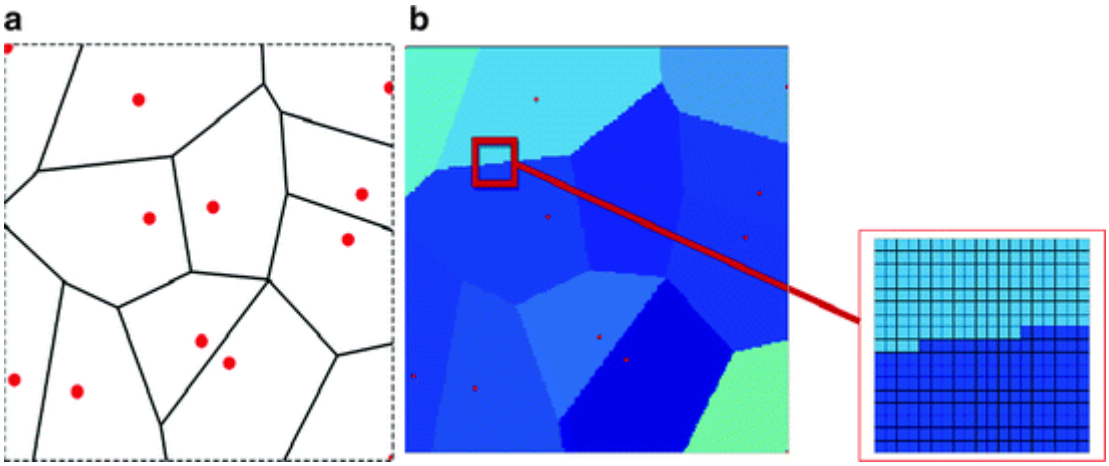

Ilustración 4.1. (a) El DV para un conjunto de puntos en 2D. (b) El DVD en 2D para el mismo conjunto de puntos; al acercar una pequeña sección del DVD se muestra su naturaleza discreta.
Fuente: Van der Putte y Ledoux (2011)

Para el problema particular que se aborda en este trabajo, la generación de fragmentos en modelos de vóxel, el espacio M será el conjunto de todos los vóxeles no vacíos de nuestro modelo y S será un subconjunto de M. Una implicación evidente de la afirmación anterior es que ninguna semilla $p_i \in S$ puede corresponder a algún vóxel vacío.

Sin embargo, quedan dos cuestiones que debemos resolver: ¿cuál es la medida de distancia subyacente en el espacio M? y ¿qué hacer con los puntos de M que están a la misma distancia de varias semillas?

4.1 Medida de distancia

Una medida, función o métrica de distancia define la distancia entre cada par de elementos de un conjunto. Para el problema que nos ocupa, la función de distancia deberá definir la distancia entre cualquier par de vóxeles.

Aunque existen multitud de funciones de distancia, en este trabajo solo se han considerado tres: la distancia euclidiana, la de Manhattan y la de Chebyshev: por considerarse las más ajustadas al problema abordado en este trabajo.

La distancia euclidiana es la distancia en línea recta entre dos puntos en un espacio euclidiano.

$$d(p,q) = \sqrt{(p_x - q_x)^2 + \left(p_y - q_y\right)^2 + (p_z - q_z)^2} \quad (1)$$

La distancia de Manhattan se expresa como la suma del valor absoluto de las diferencias componente por componente (como si midiésemos distancias en el distrito de Manhattan).

$$d(p,q) = |p_x - q_x| + \left|p_y - q_y\right| + |p_z - q_z| \quad (2)$$

La distancia de Chebyshev o distancia de ajedrez se puede entender como la distancia del rey en un tablero de ajedrez. Se expresa como la máxima diferencia en valor absoluto sobre el eje x, y y z:

$$d(p,q) = \max(|p_x - q_x|, \max(\left|p_y - q_y\right|, |p_z - q_z|)) \quad (3)$$

En la ilustración 4.2 se ejemplifica la diferencia entre las tres medidas de distancia en una cuadrícula 2D y en la ilustración 4.3 se muestra el cáculo del diagrama de Voronoi utilizando cada métrica.

√8	√5	2	√5	√8
√5	√2	1	√2	√5
2	1	a)	1	2
√5	√2	1	√2	√5
√8	√5	2	√5	√8

4	3	2	3	4
3	2	1	2	3
2	1	b)	1	2
3	2	1	2	3
4	3	2	3	4

2	2	2	2	2
2	1	1	1	2
2	1	c)	1	2
2	1	1	1	2
2	2	2	2	2

Ilustración 4.2. Distancia del vecindario del pixel rojo: (a) euclidiana, (b) Manhattan y (c) Chebyshev.

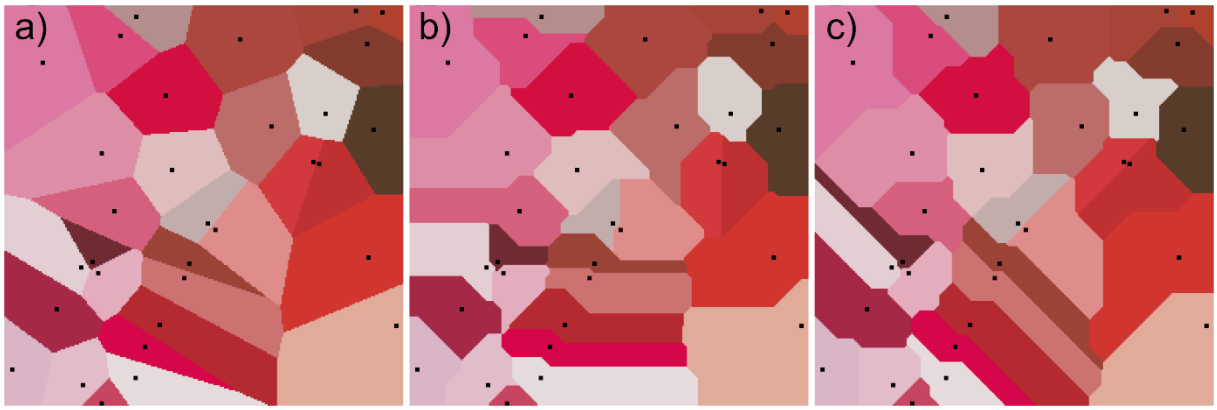

Ilustración 4.3. (a) DV utilizando la distancia euclidiana. (b) DV utilizando la distancia de Manhattan. (c) DV utilizando la distancia de Chebyshev.

4.2 Semillas equidistantes

Cuando un punto $q \in M$ está a la misma distancia de dos puntos, p_i y p_j pertenecientes a S, habrá que establecer algún criterio para decidir si q pertenece a $R(p_i, S)$ o a $R(p_j, S)$. Existen tres alternativas posibles (ilustración 4.4):

- Asignar q siempre a la región de Voronoi de p_i.
- Asignar q siempre a la región de Voronoi de p_j.
- Asignar q en base algún criterio predefinido (p. ej. aleatoriamente).

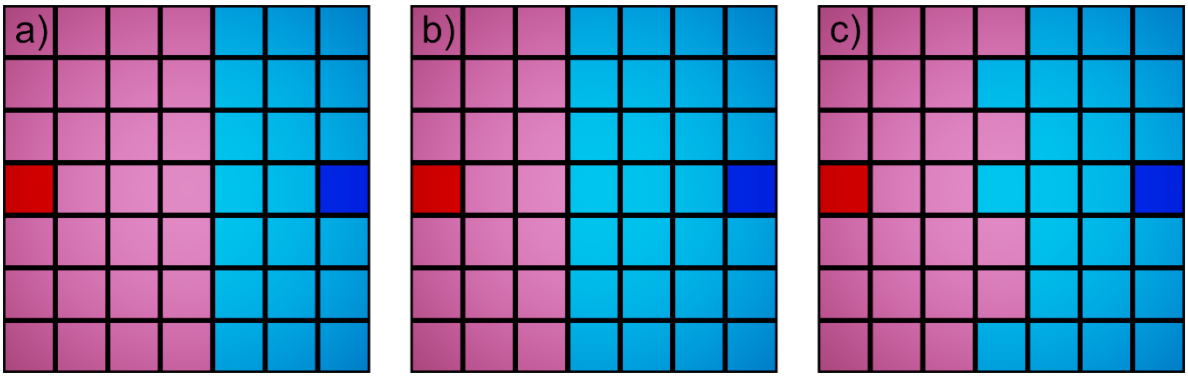

Ilustración 4.4. (a) Asignación de los vóxeles siempre a la semilla roja. (b) Asignación de los vóxeles siempre a la semilla azul. (c) Asignación de los vóxeles equidistantes aleatoriamente.

En la solución desarrollada en este trabajo, se necesita generar fragmentos con fronteras suaves, por lo tanto, los vóxeles equidistantes se asignarán siempre a la misma semilla. En concreto se asignarán a la semilla con menor subíndice. Por ejemplo, dado un punto $q \in M$ y dos puntos, p_i y p_j pertenecientes a S, ambos

equidistantes a q. Si $i < j$ entonces q se asigna a la región de Voronoi de p_i, en caso contrario, a la de p_j.

4.3 Cálculo del Diagrama de Voronoi Discreto

Van der Putte (2009) divide los métodos existentes en dos categorías, los métodos implícitos y los métodos explícitos. En los métodos explícitos, para cada voxel del diagrama se calcula su distancia con todas las semillas y se le asigna el color de la semilla más cercada. En los métodos implícitos, la asignación del valor de un voxel depende de los valores de sus vecinos.

En este trabajo se ha profundizado en dos métodos: el método exacto y el de inundación: explícito e implícito respectivamente.

4.3.1 Método exacto

Este método explicito para calcular el DVD es sencillo y fácil de entender, pero al mismo tiempo, uno de los métodos más ineficientes que existen. Su lógica es muy intuitiva: como a cada vóxel se le asigna el color de la semilla más cercana, bastará con calcular la distancia de cada vóxel con todas las semillas, buscar la semilla a menor distancia, y asignarle el color de esa semilla al vóxel.

Este planteamiento tan “ingenuo” resuelve el problema perfectamente a costa de un gran consumo de recursos computacionales. El algoritmo sería el siguiente:

```
1. FOR cada vóxel libre V:
2.     minDistance = INFINITY
3.     FOR cada semilla S:
4.         IF distance(V, S) < minDistance:
5.             minDistance = distance(V, S)
6.             Asignar al vóxel V el color de la semilla S
```

La función de distancia puede ser cualquiera de las vistas anteriormente (euclidiana, Manhattan o Chebysev) y a los vóxeles con varias semillas equidistantes se les asigna el color de la primera semilla, es decir, el de aquella con menor índice.

4.3.2 Algoritmo de inundación

Este es el algoritmo de partida de este trabajo. El procedimiento es el siguiente: primero se inicializan las semillas; luego se expande la frontera mientras queden vóxeles libres; cuando no queda ningún voxel en la frontera, el algoritmo finaliza.

```
1. FRONT = {todas las semillas}
2. NEW_FRONT = {} // Se expande la frontera como en el algoritmo BFS
3.
4. WHILE size(FRONT) > 0:
5.     FOR cada vóxel F en FRONT:
6.         SEED_F = semilla de F
7.         FOR cada vecino V no vacío de F de distinto color a SEEDS_F:
8.             SEED_V = semilla de V
9.             IF V está libre:
10.                Marcar V con SEED_F; push_back(NEW_FRONT, V)
11.            ELSE IF distance(SEED_F, V) < distance(SEED_V, V):
12.                Marcar V con SEED_F; push_back(NEW_FRONT, V)
13.            ELSE IF distance(SEED_F, V) = distance(SEED_V, V):
14.                Marcar V con el color de min_índice(SEED_F, SEED_V)
15.    FRONT = NEW_FRONT
16.    clear(NEW_FRONT)
```

Una forma de interpretar un modelo de vóxel es como un grafo donde cada vértice es un vóxel y cada arista lo relaciona con un vecino. Si expandimos la frontera del mismo modo que se explora un grafo utilizando el algoritmo BFS (Breadth-first search) se garantiza que se procesan todos los vóxeles a una determinada profundidad antes de moverse a la siguiente. De esta forma conseguimos expandir todas las semillas al mismo tiempo utilizando una única estructura de datos.

Cuando un vóxel es alcanzado se marca con el color de la semilla. De esta forma si se llega de nuevo al mismo vóxel se ignora porque ya tiene el mismo color. Si no tiene el mismo color se debe comparar la distancia para discernir a qué región pertenece.

En el método exacto hay que calcular la distancia con todas las semillas para todos los vóxeles. En este algoritmo solo realizamos el test de distancia una vez por vóxel y al menos en dos ocasiones para aquellos vóxeles que conforman la frontera entre varias regiones de Voronoi (Van der Putte, 2009). Esta optimización supone un

aumento de rendimiento considerable. En la ilustración 4.5 se muestra el número de cálculos de distancia necesarios calculando el DVD sobre una cuadrícula de 10x10.

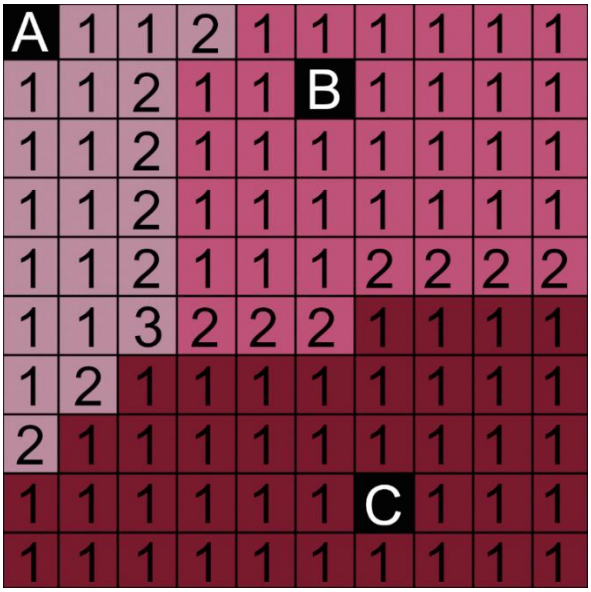

Ilustración 4.5. Cálculo del DVD en una cuadrícula 2D utilizando la distancia de manhattan. En cada píxel se muestra el número de cálculos de distancia realizados.

Van der Putte (2009) argumenta que es una opción excelente utilizar el algoritmo de inundación cuando se hace coincidir la distancia de Chebyshev con el vecindario de Moore y la distancia de Manhattan con el vecindario de Neumann porque se podrían eliminar todos los cálculos de distancias.

Por ejemplo, si utilizamos el vecindario de Von Neumann, implícitamente, se está utilizando como métrica la distancia de Manhattan (puede apreciarse comparando la ilustración 3.2 y la 4.2). Cada vóxel es asignado a la partición de la semilla que “llegue” en menor cantidad de iteraciones. Debido a que se utiliza la vecindad de von Neumann, el número de iteraciones que le “cuesta llegar” a una semilla será igual a la distancia de Manhattan entre el vóxel y la semilla. Por lo tanto, puede eliminarse el cálculo de la distancia debido a que la primera semilla en “llegar” ya es la semilla con menor distancia. Lo mismo ocurre con el vecindario de Moore y la distancia de chebyshev.

Sin embargo, esto sólo cierto en modelos macizos. En nuestros modelos de vóxel, debido a que existen vóxeles vacíos, el número de iteraciones que se tarda “en llegar” no siempre coincide con la medida distancia asociada. Por lo tanto, se obtienen resultados diferentes.

4.4 Generación de fragmentos

Se pueden generar fragmentos de forma inmediata en un modelo de vóxel utilizado el DVD. Una vez calculado, basta con considerar que cada región de Voronoi es un fragmento. Sin embargo, no todas las regiones de Voronoi calculadas poseerán una apariencia realista. A menudo se producirán fragmentos con regiones aisladas y formas antinaturales.

4.4.1 Algoritmo de inundación modificado

En el algoritmo de inundación, se pueden producir reasignaciones de vóxeles. Estas ocurren cuando un vóxel ya ocupado es reclamado por otra región de Voronoi de cuya semilla se encuentra más cerca.

El tutor me explicó que en el proyecto Kalathos descubrieron que los fragmentos generados sin comprobar distancias poseían un aspecto mucho más realista. Por este motivo, tanto en la solución original como en la desarrollada en este trabajo, se ha implementado el algoritmo sin comprobaciones de distancia. El algoritmo modificado quedaría así:

```
1. FRONT = {todas las semillas}
2. NEW_FRONT = {} // Se expande la frontera como en el algoritmo BFS
3.
4. WHILE size(FRONT) > 0:
5.     FOR cada vóxel F en FRONT:
6.         SEED = semilla de F
7.         FOR cada vecino libre V de F:
8.             Marcar V con el color de SEED
9.             push_back(NEW_FRONT, V)
10.     FRONT = NEW_FRONT
11.     clear(NEW_FRONT)
```

En la ilustración 4.6 se muestran dos fragmentos generados utilizando el algoritmo de inundación; en el fragmento (a) se comprueban las distancias (euclidiana), la (b) y (c) no.

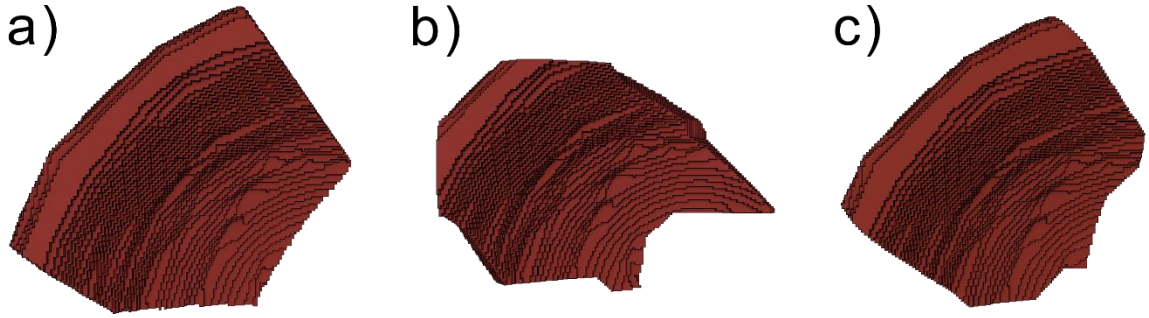

Ilustración 4.6. (a) Algoritmo de inundación "normal" con distancia euclidian. (b) Algoritmo de inundación modificado (vecindario de Von Neumann). (c) Algoritmo de inundación modificado (vecindario de Moore).

4.4.2 Diagrama de Voronoi recursivo

Los fragmentos generados utilizando el DV a menudo presentan un aspecto artificial y poco realista. Dependiendo de la métrica de distancia que se emplee, los fragmentos pueden poseer un mejor aspecto, pero ninguna métrica es la panacea, y todas tienen ventajas e inconvenientes.

En la ilustración 4.7 se presentan tres fragmentos calculados utilizando tres métricas distintas.

Ilustración 4.7. Mismo fragmento 3D generado utilizando tres métricas distintas. Respectivamente: euclidiana, manhattan y chebyshev.

Utilizando la distancia euclidiana se obtienen fragmentos con fronteras suaves, pero que siempre son convexos: característica extraña en fragmentos reales. Con la distancia de Manhattan y la de Chebyshev, se pueden producir fragmentos cóncavos y convexos, pero generan fronteras y formas artificiales.

El diagrama de Voronoi recursivo (DVR) nos permite superar las limitaciones propias de cada métrica de distancia utilizando semillas adicionales. El método consiste en calcular el DV en dos ocasiones.

Imaginemos que queremos generar 8 fragmentos y queremos aplicar el DVR con 32 semillas adicionales. Generaremos dos conjuntos de semillas: f_seeds (semillas de fragmento) y e_seeds (semillas extra). Ahora se calcula el DV sobre las e_seeds utilizando como semillas las f_seeds. El procedimiento es el siguiente:

```
1. e_seeds = {32 semillas aleatorias}
2. // Al utilizar como f_seeds elementos del conjunto e_seeds se
   garantiza que al menos a cada semilla de fragmento le corresponde una
   semilla extra.
3. f_seeds = {las ocho primeras semillas de e_seeds}
4.
5. FOR cada semilla E en e_seeds:
6.     NEAREST = semilla más cercana a E en f_seeds
7.     Asignar a E el color de NEAREST
```

Por último, una vez calculadas las semillas extra, se calcula el DVD con el método exacto o el algoritmo inundación utilizando como semillas las e_seeds. La función de distancia usada en el DVR debe coincidir con la utilizada en el DVD. La ilustración 4.8 es una demostración del cálculo del DVR en un plano 2D.

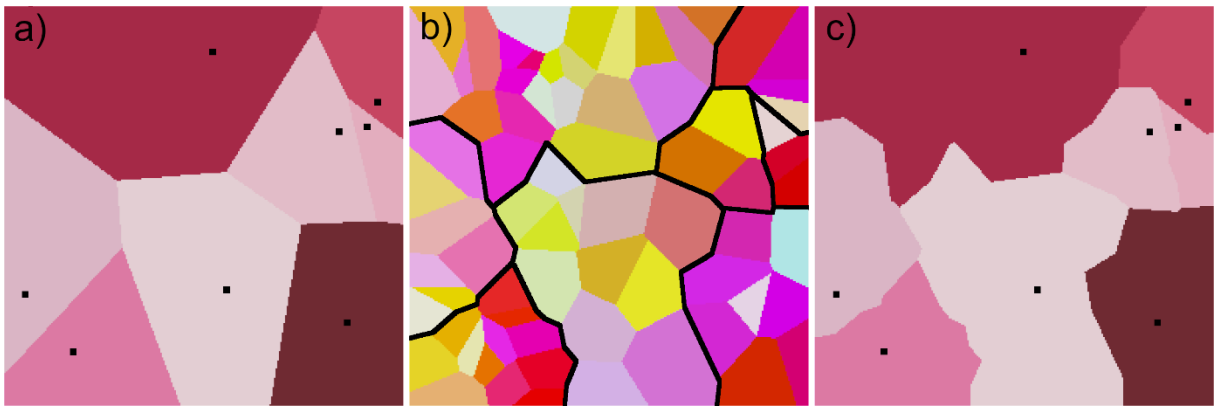

Ilustración 4.8. (a) Semillas para generar fragmentos y sus regiones de Voronoi . (b) Regiones de Voronoi utilizadas en el DVR silueteados con los fragmentos finales. (c) Resultado final tras aplicar el DVR.

4.4.3 Regiones aisladas

Cuando utilizamos el método exacto o aplicamos el DVR se pueden producir regiones aisladas como la mostrada en la ilustración 4.9. Una región aislada produce fragmentos mal formados.

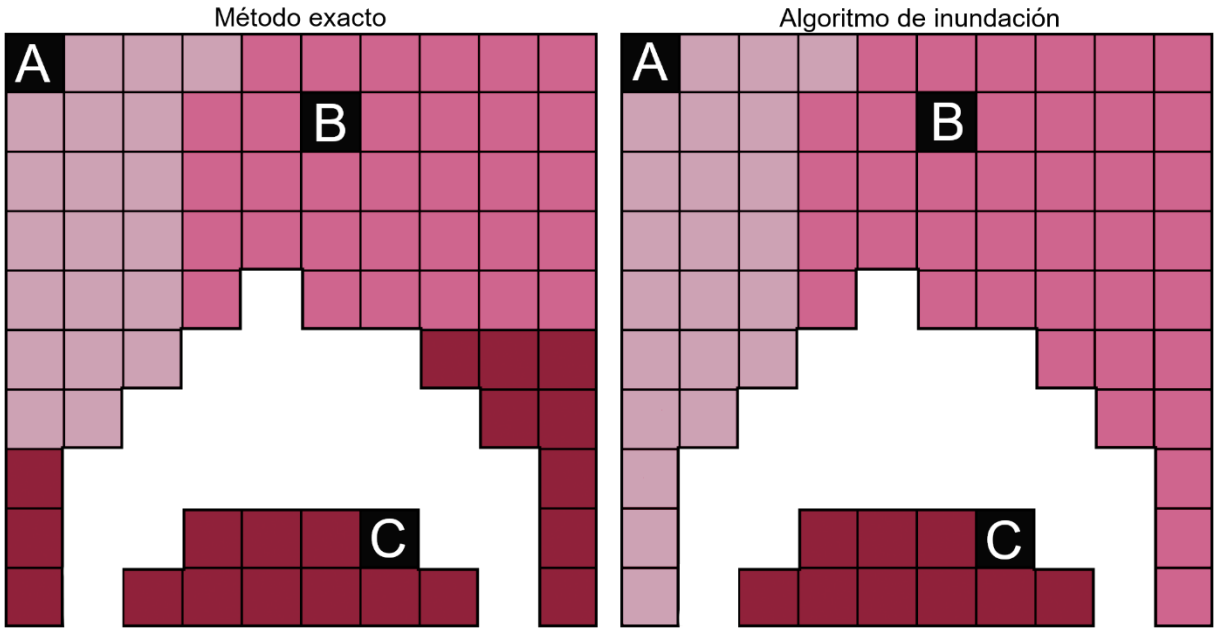

Ilustración 4.9. Utilizando el método exacto se le asignan a la semilla C vóxeles que no son alcanzables. Con el algoritmo de inundación esos vóxeles se los queda la semilla que llegue primero.

Para eliminar las regiones aisladas (sin utilizar estructuras de datos adicionales) se puede aplicar un algoritmo de inundación muy sencillo como posprocesamiento de la asignación inicial de vóxeles a cada región. Se añaden a la frontera las semillas y se va expandiendo a la vecindad mientras sean del mismo color. Aquellos vóxeles que no son alcanzados por el algoritmo de inundación, es decir, que nunca forman parte de la frontera conforman una región aislada. Finalmente, bastará con marcar esos vóxeles como vacíos. El algoritmo es el siguiente:

```
1. FRONTERA = {todas las semillas}
2.
3. WHILE size(FRONTERA) > 0:
4.     F = pop_first(FRONTERA)
5.     FOR cada vecino V con el mismo color que F:
6.         push_last(FRONTERA, V)
7.
8. Marcar los vóxeles que no han pasado por la frontera como vácios
```

En la ilustración 4.10 se muestra el antes y el después de eliminar las regiones aisladas en un fragmento mal formado.

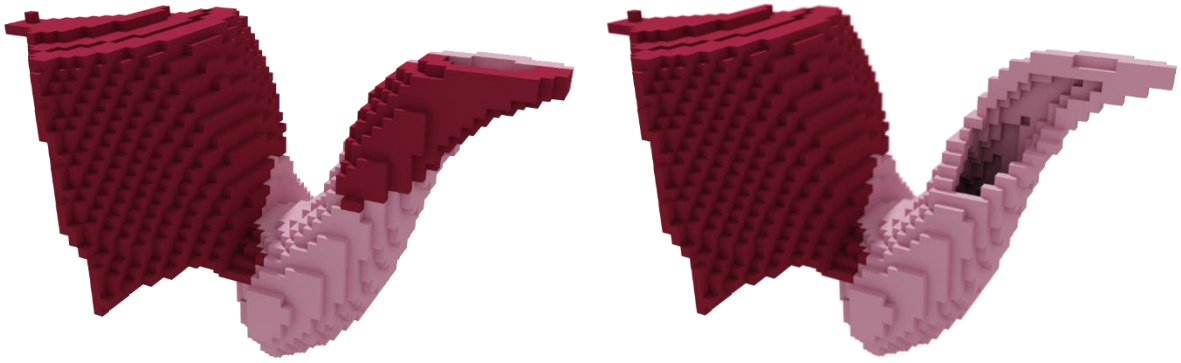

Ilustración 4.10. (a) Fragmento de la tetera de Utah mal formado por que tiene una región aislada. (b) El fragmento tras haber eliminado la región aislada.

Una alternativa a eliminar las regiones aisladas sería generar nuevos fragmentos con ellas o anexionarlas a otros fragmentos ya existentes. Sin embargo, ambas opciones han sido descartadas porque obligarían a elaborar una etapa de posprocesamiento más compleja.

5 PROGRAMACIÓN EN GPU

En 2001 con el lanzamiento de la NVIDIA GeForce 3 y las etapas programables en GPU nace la GPGPU (*General-purpose Computing on Graphics Processing Units*) que consiste en el uso de las GPUs para ejecutar código de propósito general (Harris, 2014).

En 2007 CUDA (*Compute Unified Device Architecture*) aparece junto con las nuevas generaciones de GPUs con arquitectura unificada de NVIDIA. Con CUDA, NVIDIA revolucionó el mundo de la programación en GPU permitiendo extraer el máximo rendimiento para computación de propósito general de sus tarjetas gráficas (Dehal *et al.,* 2018).

A finales de 2008, OpenCL fue lanzado como una alternativa a las tecnologías propietarias existentes. A diferencia de CUDA, OpenCL se define como un estándar para sistemas heterogéneos como CPUs, GPUs o FPGAs (Khronos Group, 2020). La ventaja principal de OpenCL, respecto a CUDA, es la portabilidad. CUDA solo

funciona con dispositivos de NVIDIA, OpenCL funciona en una gama mucho más amplia de dispositivos.

En la versión 4.3 de OpenGL se introdujeron los *Compute Shaders*. A diferencia de otros *shaders* tradicionales, estos no tienen un propósito en específico. Le corresponde a cada *Compute Shader* determinar cuál es su función (Khronos Group, 2020).

A diferencia de CUDA y OpenCL, OpenGL está soportado de forma nativa en una gran variedad de sistemas, por lo tanto, no necesita la instalación de *toolkits* o bibliotecas adicionales.

5.1 ¿Por qué OpenGL Compute Shaders?

Para la realización de este Trabajo Fin de Grado se ha utilizado OpenGL. Estas son las razones que motivaron su elección:

- Se posee experiencia previa (ganada en dos asignaturas del Grado en Ingeniería Informática: Programación de Aplicaciones Gráficas y Programación Hardware).
- Compatible con toda clase de GPUs (CUDA solo funciona con dispositivos de NVIDIA).
- No requiere instalación de *toolkits* o bibliotecas adicionales.
- Desde el planteamiento de este Trabajo Fin de Grado se partía con la intención de profundizar en el uso de los *Compute Shader* añadidos en la versión 4.3 de OpenGL.

5.2 OpenGL Compute Shaders

Un *Compute Shader* es un tipo de *Shader* que se puede utilizar para computación de propósito general. Si bien puede utilizarse para tareas de visualización, normalmente se utiliza con otros propósitos.

5.2.1 Modelo de ejecución

Normalmente en la ejecución de un *Compute Shader* pueden necesitarse miles de hebras, de la misma forma que en el *pipeline* de visualización clásico, un *Fragment Shader* puede ejecutarse para calcular la iluminación en millones de píxeles. El problema es que ni siquiera una GPU puede mantener tal cantidad de hebras al unísono. Por lo tanto, la solución pasa por dividir el trabajo total en *Work Groups* (Grupos de Trabajo) que están formados por subconjuntos de hebras. Todas las hebras dentro de un *Work Group* se ejecutan en paralelo. La ejecución de cada *Work Group* es independiente a la de los demás. En la ilustración 5.1 se representan los distintos *Work Groups* en la ejecución de un *Compute Shader*.

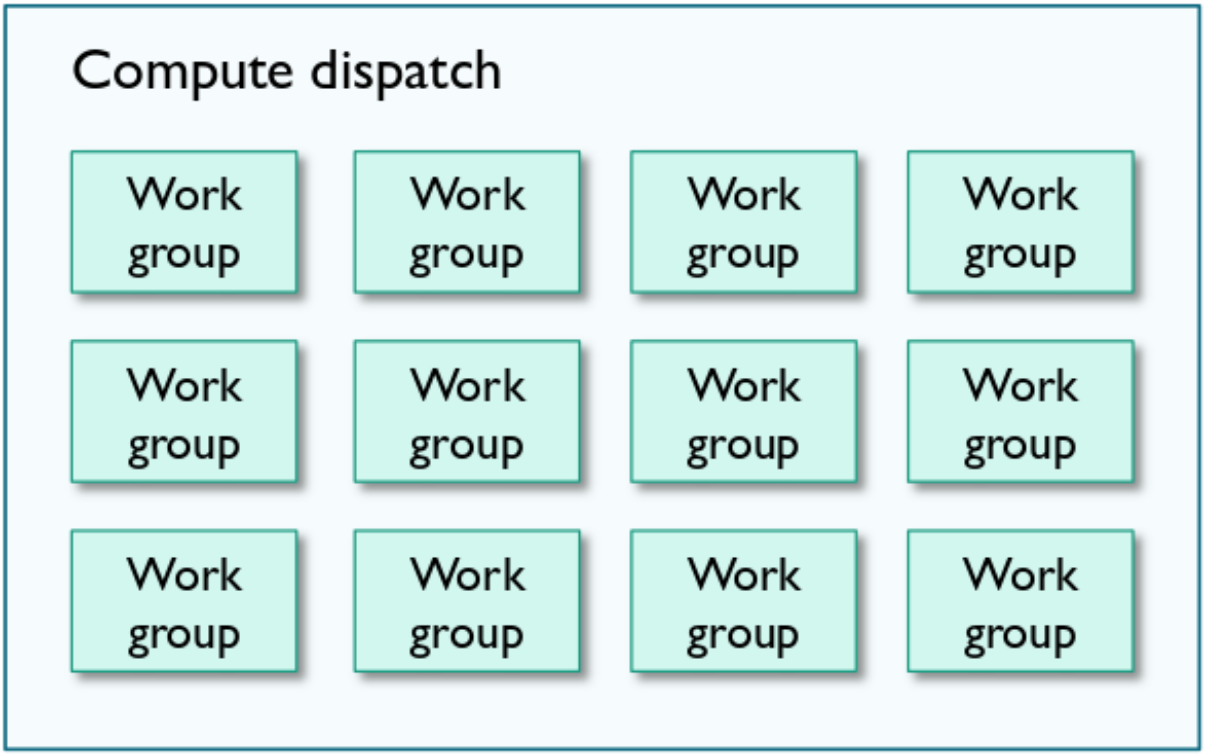

Ilustración 5.1. Grupos de trabajo en una ejecución de un Compute Shader. ***Fuente: https://arm-software.github.io/opengl-es-sdk-for-android/compute_intro.html*** (recuperado en Julio de 2020)

Al igual que una ejecución de un *Compute shader* se divide en varios *Work Groups*, un *Work Group* está formado por un conjunto de hebras y una memoria compartida (del inglés *Shared Memory*). En la ilustración 5.2 se representa la estructura de un *Work Group*. Por un lado, están las hebras de ejecución, por el otro la memoria compartida. La memoria compartida de cada *Work Group* es accesible solo por las hebras del mismo *Work Group*.

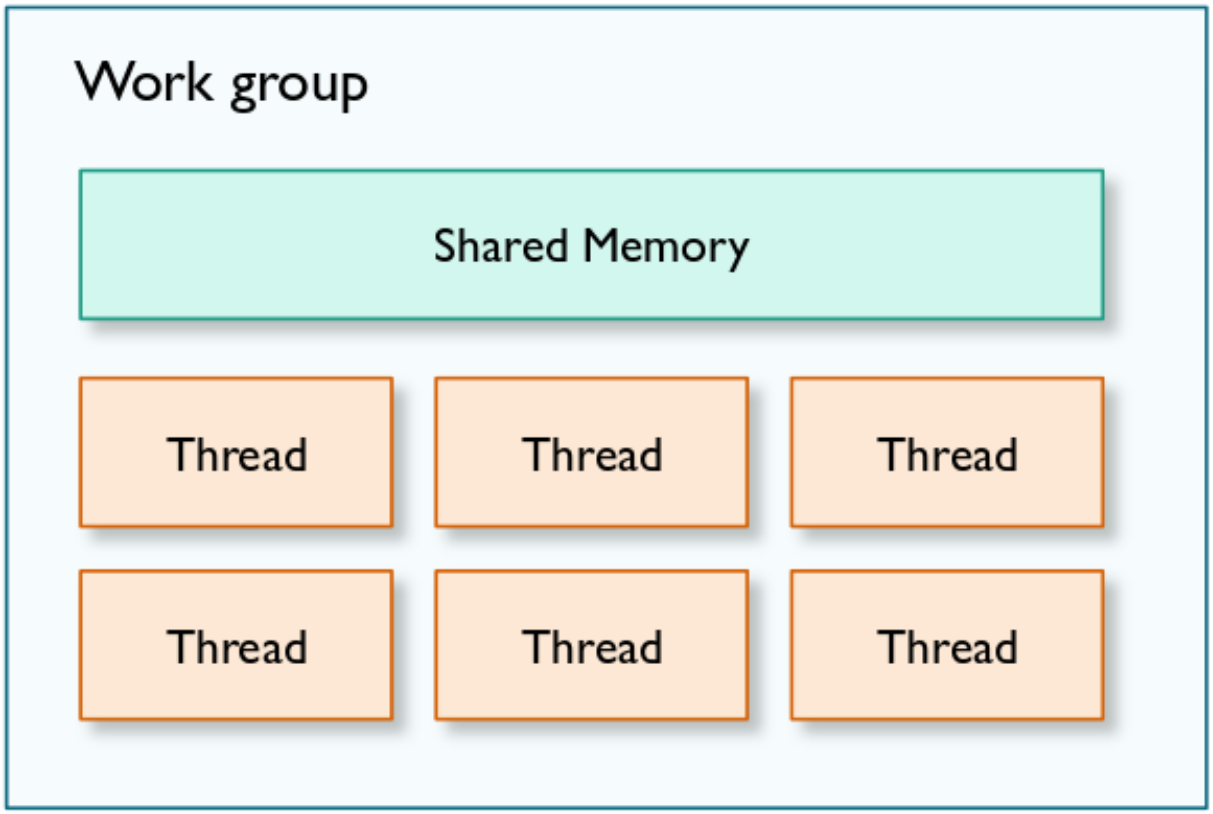

Ilustración 5.2. Hilos y memoria compartida en un grupo de trabajo. ***Fuente:* https://arm-software.github.io/opengl-es-sdk-for-android/compute_intro.html** (recuperado en Julio de 2020)

5.2.2 Tipos de memoria

Emplear adecuadamente cada tipo dependiendo de la naturaleza del problema puede ser la diferencia entre el éxito y el fracaso. En programación en GPU tenemos disponibles tres tipos de memoria:

- **Memoria local:** exclusiva de cada hebra de ejecución. Muy limitada aunque muy rápida.
- **Memoria compartida:** compartida por todos los hilos de cada *Work Group*. Mucho más rápida que la memoria global.
- **Memoria global:** accesible desde cualquier hebra de ejecución. Masiva aunque muy lenta.

Tanto en OpenGL, como en CUDA u OpenCL existen operaciones atómicas. En muchas ocasiones son la única forma de resolver algún problema, aun así, se debe

tener cuidado con ellas: pueden reducir el rendimiento considerablemente. Además, existe el inconveniente añadido de que sólo pueden aplicarse sobre tipos enteros.

5.3 Sincronización

En OpenGL sólo se puede lograr sincronización a nivel de *Work Group*. Si se necesita sincronizar todas las hebras globalmente, habrá que dividir el trabajo en dos *Compute Shader* diferentes e invocar la barrera de memoria conveniente entre la ejecución de ambos.

6 Diagrama de Voronoi Discreto en GPU

En este apartado se discute cómo adaptar a GPU el método exacto y el algoritmo de inundación. Aunque se emplee la terminología propia de OpenGL, los problemas y soluciones, planteados en este apartado, son aplicables a cualquier entorno de programación GPU: como CUDA u OpenCL.

6.1 Paralelismo en GPU

La computación paralela es una forma de computación en la que se realizan muchos cálculos simultáneamente, operando sobre el principio de que los problemas grandes a menudo se pueden dividir en pequeños, que luego se resuelven al mismo tiempo. El grado de paralelismo que se puede lograr depende de la naturaleza inherente del problema (Kaeli *et al*., 2015).

Las GPUs siguen el paradigma SIMD (*single instruction, multiple data*), esto es, cientos de unidades de procesamiento que aplican las mismas operaciones sobre distintos datos. En consecuencia, las GPUs son especialmente buenas explotando el paralelismo de datos; para que una tarea sea optimizable en GPU debe permitirlo, es decir, debe poder dividirse en muchas subtareas más ligeras e independientes entre sí.

6.2 Método exacto

El método exacto requiere calcular la distancia de cada vóxel contra todas las semillas, eso lo convierte en un algoritmo muy ineficiente (Van der Putte, 2009). Esto es cierto en CPU, donde existen alternativas muy eficientes; en GPU, donde se explota el paralelismo de datos, el método exacto produce resultados excelentes sin apenas esfuerzo de programación.

El cálculo del color de cada vóxel depende únicamente de las semillas (que no varían) y de su posición en el modelo. En consecuencia, en el método exacto, el cálculo del color de cada vóxel es independiente, lo cual convierte a este método en un caso perfecto donde explotar el paralelismo de datos.

La adaptación a GPU del algoritmo no requiere de ninguna modificación. Sin embargo, se puede conseguir un aumento marginal del rendimiento si se utiliza memoria compartida para precargar las semillas dentro de cada *Work Group*. De esta forma se reduce el número de accesos a memoria global sustancialmente.

6.3 Algoritmo de inundación

El principal reto de este Trabajo Fin de Grado es la adaptación a GPU del algoritmo de inundación. A diferencia del método exacto, donde se puede explotar el paralelismo de datos de forma inmediata, en el algoritmo de inundación el cálculo del color de cada vóxel depende de su vecindario. Para poder adaptar el algoritmo de inundación a GPU será necesario acelerar aquellos cálculos que sean paralelizables.

En el algoritmo de inundación, la frontera no es más que un conjunto de vóxeles. Expandir la frontera, por lo tanto, significará calcular la unión de los vecinos libres de todos los vóxeles que conforman la frontera. Como el cálculo de la vecindad de cada vóxel es independiente del resto: la expansión de la frontera es un proceso que podemos paralelizar.

En una implementación no paralela cuando un vóxel de la frontera es procesado se marca para que deje de estar libre. Cuando otra semilla reclame el mismo vóxel ya no estará libre, y por tando, no podrá quedarselo. Así se evita añadir a la frontera vóxeles duplicados.

Cuando se expande la frontera en paralelo sí se pueden producir duplicados. En la imagen (a) de la ilustración 6.1 se muestra como varias semillas se expanden al

mismo vóxel. Esto ocurre porque ambos vóxeles se expanden en paralelo y ambos creen que el vóxel rojo está libre. Una forma simple de resolver el problema es expandir la frontera de forma ordenada, como ocurre en la imagen b, c, d y e de la ilustración 6.1.

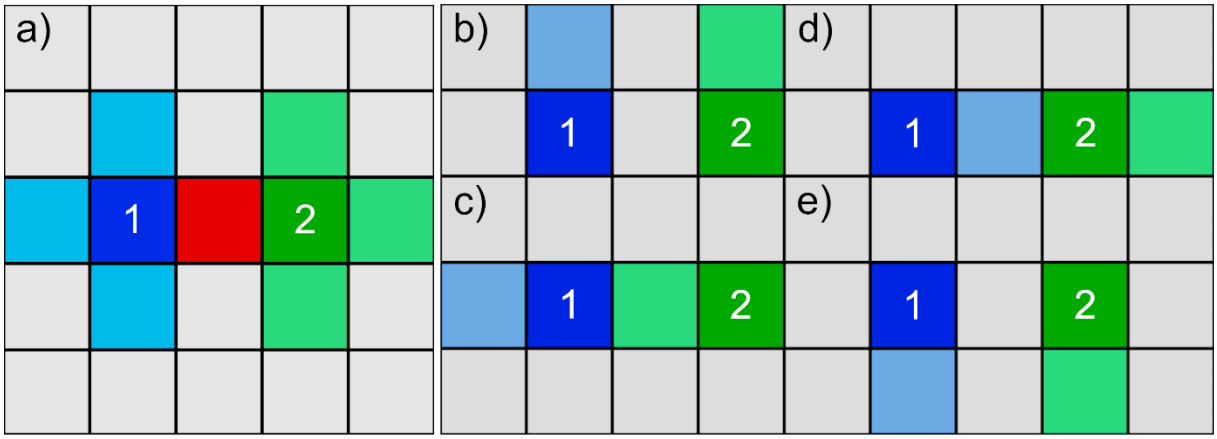

Ilustración 6.1. (a) Dos vóxeles, 1 y 2, intentando expandirse a su vecindario, al hacerlo en paralelo, ambos reclaman el vóxel rojo. En b, c, d y e se muestra cómo se pueden evitar las colisiones si todos los vóxeles se expanden de forma ordenada.

En la ilustración 6.2 se muestra la expansión ordenada en cuatro tiempos de la frontera en 2D (extrapolable a 3D). Ahora los vóxeles con varias semillas equidistantes no se asignan a la semilla de menor indice sino a la semilla que llegue antes. De igual manera, no supone ningún problema para generar fragmentos: las fronteras entre distintas regiones siguén siendo homogéneas.

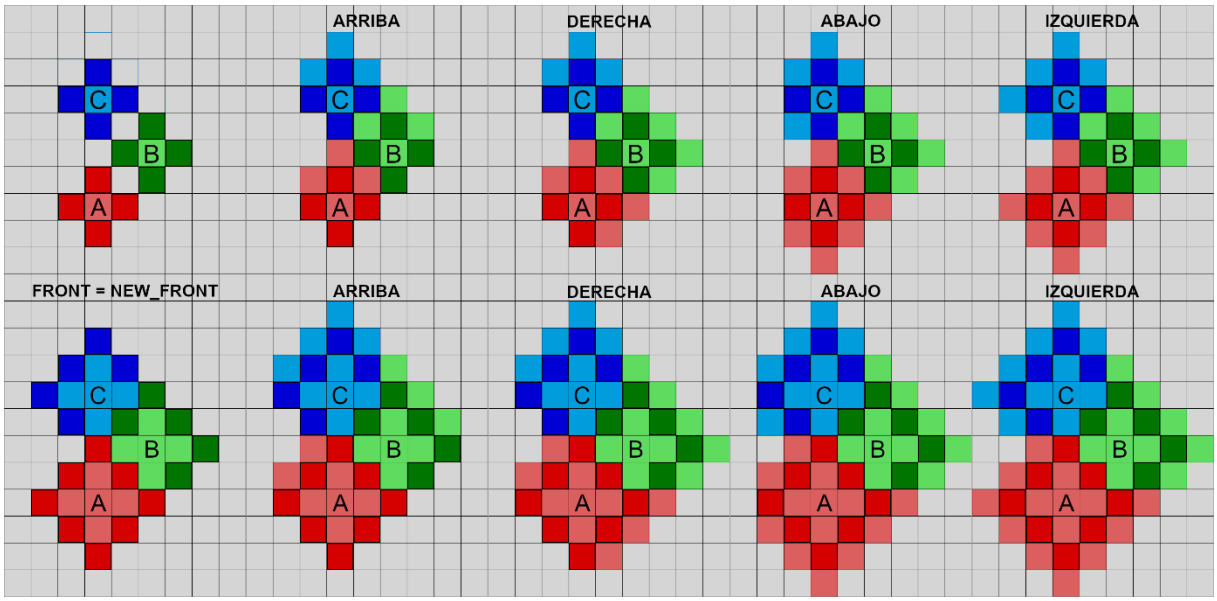

Ilustración 6.2. Ejemplo de expansión de frontera ordenadamente en cuatro tiempos. Los vóxeles más oscuros conforman la frontera.

La expansión de toda la frontera sobre una misma dirección puede realizarse en paralelo sin riesgo de duplicar vóxeles:

```
1. PROC expandir_hacia(DIR):
2.     PARALLEL FOR cada vóxel V en FRONT:
3.         SEED = semilla de V
4.         NEIGHBOUR = vecino de V en la dirección DIR
5.         IF NEIGHBOUR está libre:
6.             Marcar NEIGHBOUR con el color de SEED
7.             push_back(NEW_FRONT, NEIGHBOUR)
```

La expansión completa de la frontera, en las seis direcciones posibles, se realizaría así:

```
1. PROC expandir_frontera():
2.     expandir_hacia(+X); expandir_hacia(-X);
3.     expandir_hacia(+Y); expandir_hacia(-Y);
4.     expandir_hacia(+Z); expandir_hacia(-Z);
```

Debe tenerse en cuenta que dependiendo del orden al expandir la frontera los vóxeles equidistantes se distribuyen de forma distinta (ilustración 6.3). Utilizando el vecindario de Moore el número de vóxeles equidistantes es muy bajo, por lo tanto, apenas modifica el resultado.

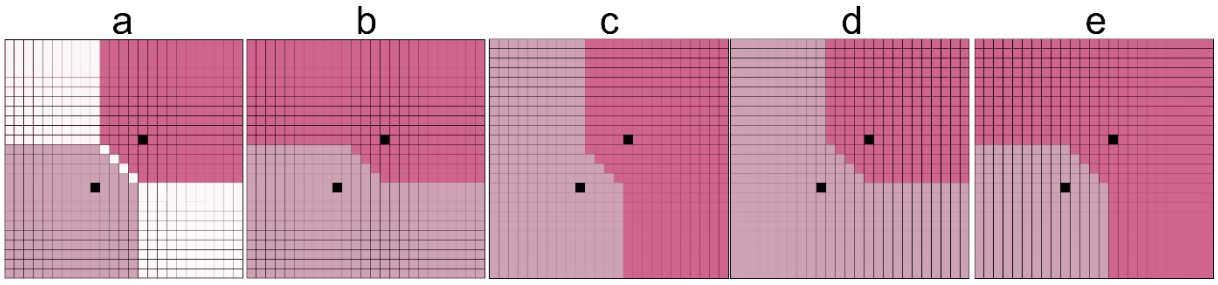

Ilustración 6.3. (a) DVD en 2D, los vóxeles blancos están a la misma distancia (de manhattan) de ambas semillas. La frontera se expande en el siguiente orden: (b) $+\boldsymbol{X}, -\boldsymbol{X}, +\boldsymbol{Y}, -\boldsymbol{Y}$; (c) $-\boldsymbol{Y}, +\boldsymbol{Y}, -\boldsymbol{X}, +\boldsymbol{X}$; (d) sentido antihorario y (e) sentido horario.

En la ilustración 6.4 se presenta el diagrama de flujo del algoritmo de inundación adaptado a GPU. La región en verde representa la parte del algoritmo que se ejecuta en GPU.

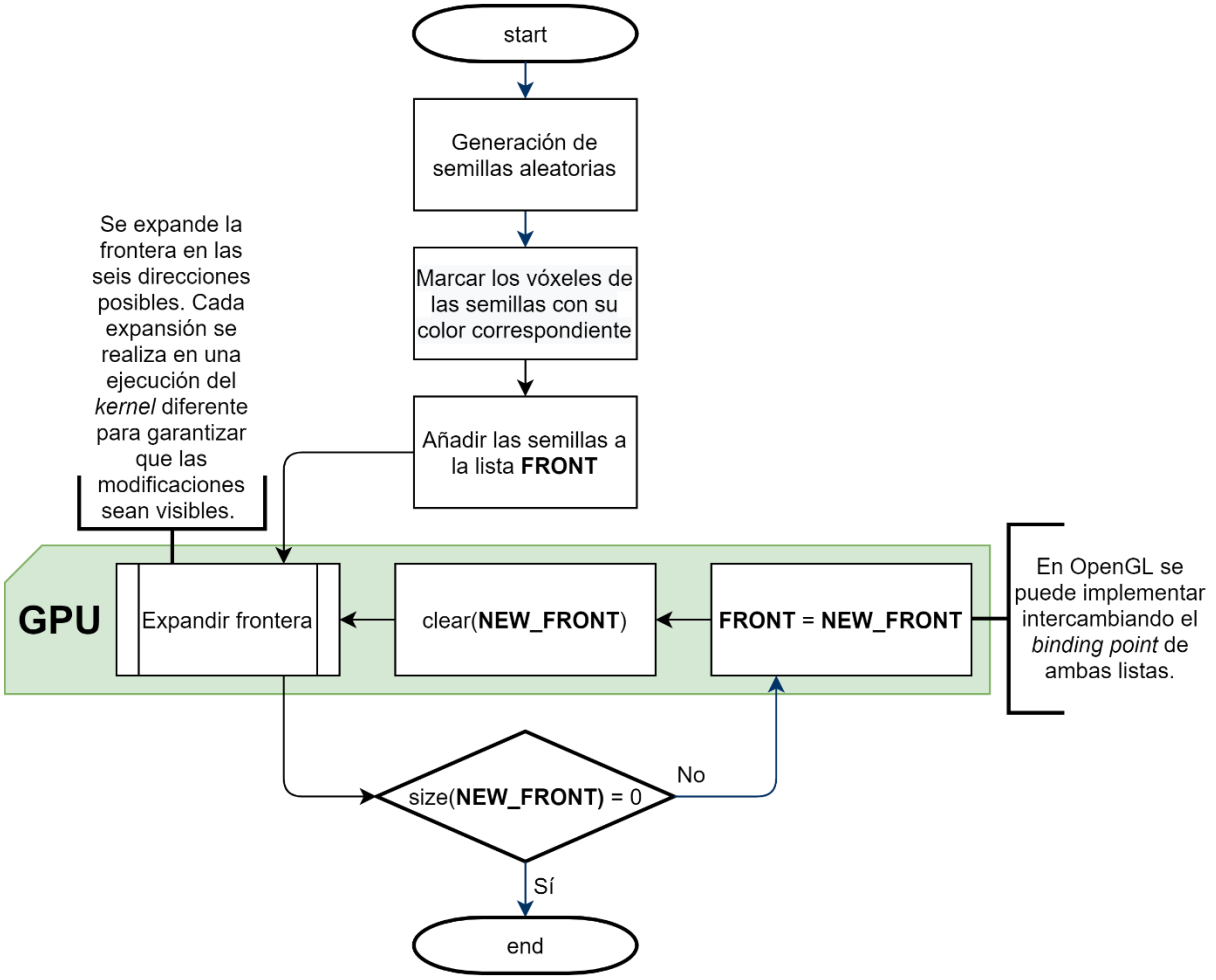

Ilustración 6.4. Diagrama de flujo del algoritmo de inundación para calcular el DVD en GPU.

6.3.1 Implementación de una pila en GPU

En el algoritmo de inundación en GPU, la lista FRONT y NEW_FRONT pueden ser implementadas como una pila. Sin embargo, en GPU no podemos operar con estructuras de datos dinámicas, por lo tanto, la pila que implementemos, deberá tener un tamaño máximo fijo; que no se quede corto, pero que tampoco suponga un uso excesivo de memoria. Además, será necesario dotar a nuestra pila de un comportamiento atómico para garantizar su coherencia.

En OpenGL se puede implementar utilizando un *buffer* de memoria y un contador atómico. Así sería el procedimiento para añadir un elemento:

```
1. uvec4 STACK[MAX_SIZE] /*Array en GPU*/
2. atomic uint SIZE /*Contador atómico con el tamaño de la pila*/
3.
4. PROC push(vec4 ELEMENTO):
5.     INDEX = atomicIncrement(SIZE)
6.     STACK[INDEX] = ELEMENTO /*Solo yo puedo escribir en este índice*/
```

Estableciendo el contador atómico a cero se puede implementar la operación de limpiar la pila.

7 DISCUSIÓN Y EVALUACIÓN DE RESULTADOS

7.1 Comparación de fragmentos

Para probar los algoritmos implementados en este trabajo se han utilizado tres modelos con distintas características. El objetivo es experimentar sobre objetos variados que sean representativos de los que se utilizan en el proyecto Kalathos. En la ilustración 7.1 se muestran los modelos de pruebas sin fragmentar.

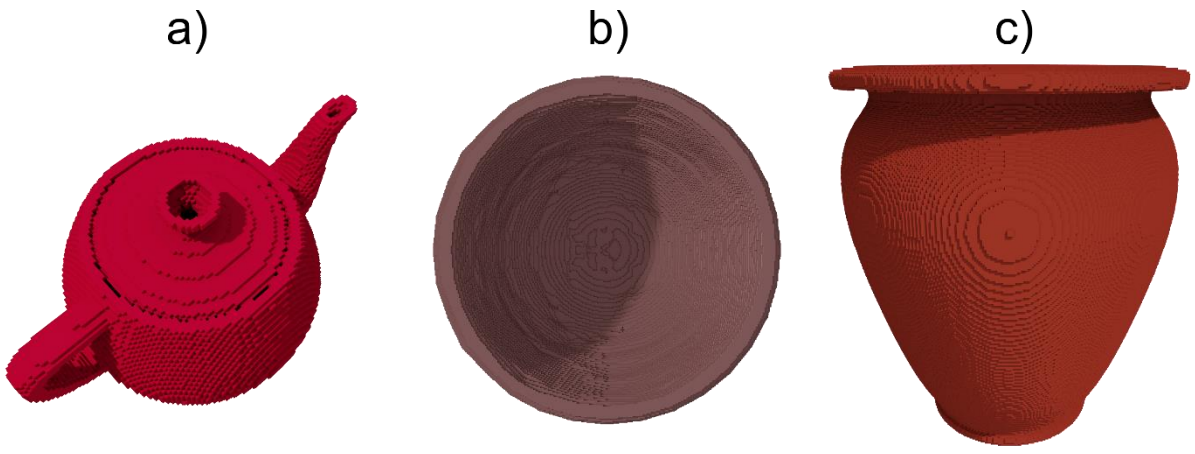

Ilustración 7.1. Modelos de prueba: (a) teapot.vox ($\mathbf{126 \times 81 \times 61}$) , (b) vasija.vox ($\mathbf{256 \times 256 \times 65}$) y (c) BA074_1.vox ($\mathbf{225 \times 225 \times 256}$) .

En lo sucesivo me referiré a los modelos a, b y c de la ilustración 7.1, como modelo A, modelo B y modelo C, respectivamente.

Los modelos B y C fueron proporcionadas por el tutor para la realización de pruebas. El modelo A, la tetera de Utah, se ha seleccionado por ser un modelo cerrado

y presentar zonas sobresalientes (asa y pico): características ausentes en los dos anteriores.

En el proyecto original que me entregó el tutor, los fragmentos generados que suponían menos de un cinco por ciento del volumen total del modelo, eran descartados. Además, aquellos fragmentos que no formaban parte de la base o el tope de los modelos, también se descartaban. Todos los fragmentos considerados en esta sección cumplen ambas condiciones.

Conseguir mejorar el rendimiento de la implementación original en Python es el principal objetivo de este Trabajo Fin de Grado. Además, ambas implementaciones deben producir resultados similares. En la ilustración 7.2 se muestra como los fragmentos generados con el algoritmo de inundación en CPU y en GPU son prácticamente iguales.

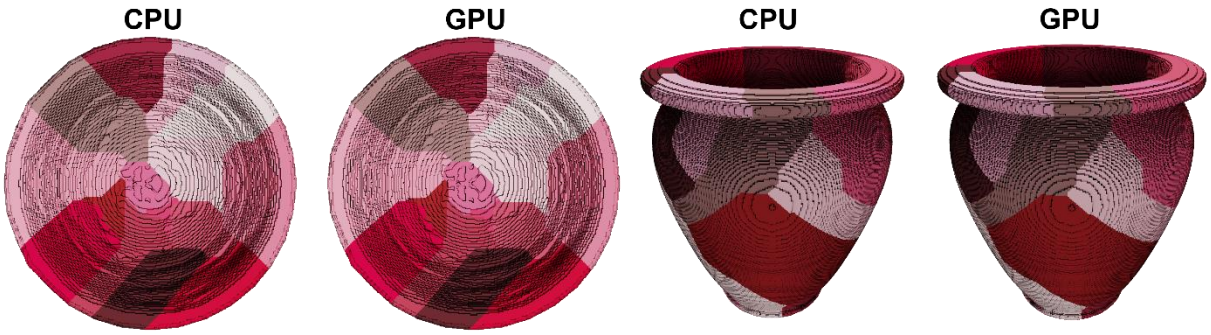

Ilustración 7.2. Modelo B y C fragmentados utilizando la implementación original en Python y la implementación en GPU.

En la ilustración es dificil apreciarlo, pero los fragmentos generados tienen tamaños ligeramente distintos. Esto se debe a que en ambas implementaciones se siguen criterios diferentes al asignar los vóxeles que poseen varias semillas equidistantes. Salvo por este detalle, los fragmentos generados son idénticos.

Como se detalla en el aparado 5.3, dependiendo de qué función de distancia se emplee se obtienen resultados distintos. En la ilustración 7.3 se comparan los fragmentos generados utilizando las tres métricas estudiadas: euclidiana, manhattan y chebyshev.

Ilustración 7.3. Comparación de fragmentos generados utilizando tres métricas de distancia: arriba, euclidiana; en medio, manhattan; y abajo, chebyshev. Los fragmentos han sido generados utilizando las mismas semillas.

Utilizando la distancia euclidiana solo pueden producirse fragmentos convexos (todos los ángulos internos menores a 180º), mientras que con la de Manhattan o la de Chebyshev también se pueden producir fragmentos cóncavos; esta característica puede resultar conveniente, en fenómenos reales es extraño encontrar fragmentos convexos producto de alguna rotura natural.

El defecto de la distancia de manhattan, como se aprecia en el fragmento rosa de la segunda fila de la ilustración 7.3, es que a menudo produce protuberancias y salientes poco realistas.

De las tres métricas estudiadas, la distancia de Chebyshev es la que permite producir fragmentos más equilibrados. Puede generar fragmentos cóncavos, a diferencia de la euclidiana. Además, no produce las irregularidades que sí aparecen con la distancia de manhattan. No obstante, no se debe interpretar que existe una medida de distancia superior al resto. Aprovechando las tres, se aumenta sustancialmente el conjunto de posibles fragmentos que se pueden generar.

El DVR permite incrementar la variabilidad y el realismo de los fragmentos generados. La ilustración 7.4 es una comparativa de la fracturación del modelo C utilizando y sin utilizar el DVR.

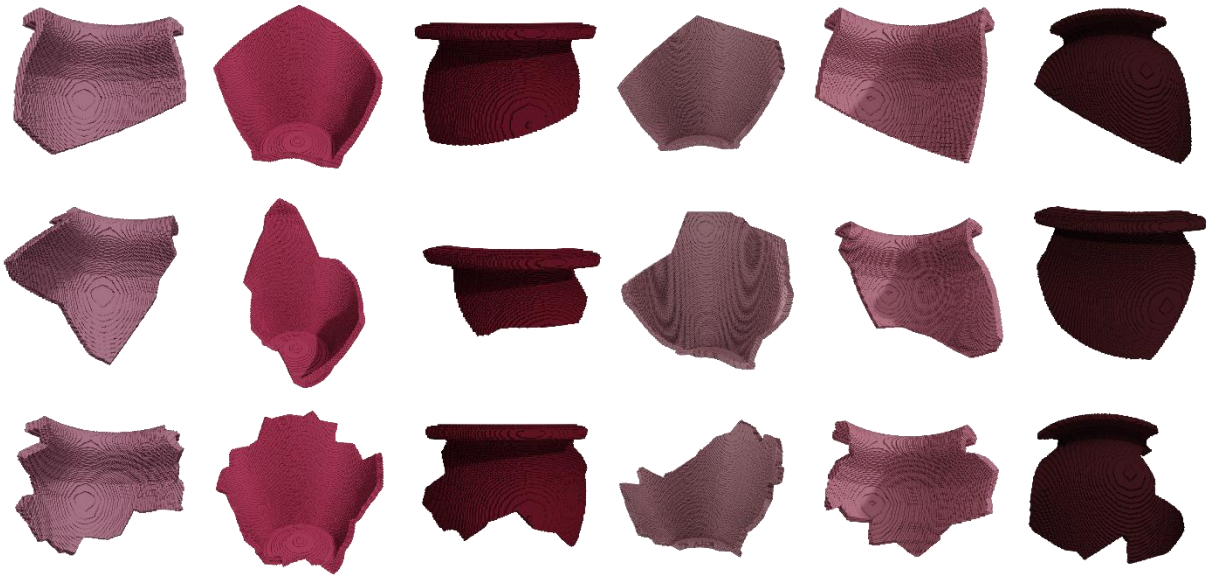

Ilustración 7.4. Fracturación del modelo C en seis fragmentos utilizando el método exacto y la distancia euclidiana. En la fila del medio se ha aplicado el DVR con 32 semillas extra. En la fila de abajo con 248 ($\mathbf{248 + 6 = 254 = max\_fragmentos}$).

La cantidad de semillas que producen buenos resultados aplicando el DVR, depende de la geometría de cada modelo y de la cantidad de fragmentos a generar.

Al calcular el DVR debe tenerse cuidado de no utilizar demasiadas semillas extra. La ilustración 7.5 muestra varios fragmentos generados aplicando el DVR con una apariencia extraña porque poseen una silueta demasiado irregular. En general, basándome en la experimentación realizada, si se generan pocos fragmentos, basta con utilizar entre 8 y 32 semillas extra para producir buenos resultados.

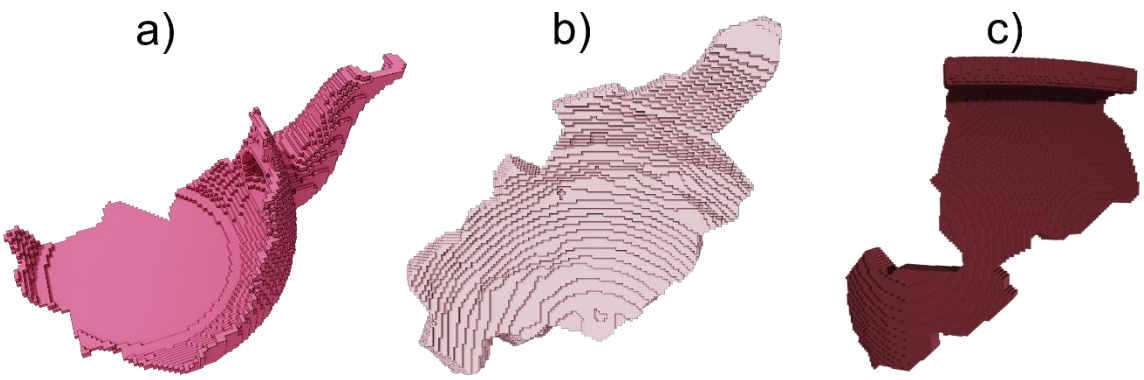

Ilustración 7.5. Fragmentos demasiado irregulares del modelo A, B y C respectivamente. Los tres han sido generados utilizando DVR con 248 semillas extra.

En la imagen c de la ilustración 7.5 se muestra un fragmento con un estrechamiento poco realista en el medio. Este es el principal problema del DVD, incluso con una cantidad moderada de semillas extra puede ocurrir.

7.2 Comparación de rendimiento

Las pruebas de rendimiento han sido realizadas con un portátil Acer Aspire 5 con una CPU Intel Core i7-8565 1.8 GHz, tarjeta gráfica NVIDIA GeForce MX130 con 2GB VRAM y 12 GB de RAM DDR4.

Estudiando los datos obtenidos, la solución desarrollada en este trabajo ofrece mejores resultados que el algoritmo original. Una de las causas que produce esta diferencia es que el algoritmo original utiliza Python, un lenguaje de alto nivel, mientras que los algoritmos desarrollados en este trabajo están implementados en c++/OpenGL.

En esta sección se compara el rendimiento de la implementación original, el método exacto en GPU y el algoritmo de inundación modificado en GPU. Junto con el método exacto se añade el coste de aplicar el algoritmo de eliminación de regiones aisladas. A diferencia del algoritmo de inundación, sin este posprocesado el método exacto puede generar fragmentos mal formados (ilustración 4.9). El coste computacional que añade el DVR es tan bajo que se ha sido ignorado en esta comparativa.

La tabla 7.1 muestra los resultados obtenidos fragmentando los modelos A, B y C. Cuando se fragmentan modelos ligeros, como el modelo A, el algoritmo original muestra un rendimiento superior a los algoritmos en GPU porque el trasiego de información entre memoria principal y memoria gráfica, penaliza a la implementación en GPU. Sin embargo, en modelos más pesados, como el B y el C, el rendimiento del algoritmo original se desploma. Se consiguen alcanzar *speedups* superiores a 400, lo cual es muchísimo.

	Modelo A	Modelo B	Modelo C
Implementación original CPU:			
Tiempo (s)	0,0047	59	146
Método exacto GPU + eliminar regiones aisladas:			
Tiempo (s)	0,038	0,149	0,362
Speedup	0,123	395	403
Algoritmo inundación modificado GPU:			
Tiempo (s)	0,046	0,145	0,313
Speedup	0,102	406	466

Tabla 7.1. Rendimiento obtenido en la implementación original en Python y la implementación en c++/OpenGL, al fracturar el modelo A, B y C, generando 64 fragmentos. Los valores de *speedup* inferior a 1 están marcados en rojo porque .

En la tabla 7.1 se muestra el efecto sobre el rendimiento dependiendo del tamaño del modelo: modelo A pequeño, modelo B mediano y modelo C grande. En la tabla 7.2 se comparan los resultados dependiendo del número de fragmentos generados.

	64 fragmentos	128 fragmentos	254 fragmentos
Implementación original CPU:			
Tiempo (s)	146	150	149
Método exacto GPU + eliminar regiones aisladas:			
Tiempo (s)	0,335	0,381	0,413
Speedup	435	394	360
Algoritmo inundación modificado GPU:			
Tiempo (s)	0,313	0,288	0,287
Speedup	466	520	519

Tabla 7.2. Rendimiento obtenido en la implementación original en Python y la implementación en c++/OpenGL, al fracturar el modelo C, generando 64, 128 y 254 fragmentos.

El número de fragmentos a generar apenas afecta al rendimiento del algoritmo de inundación. El método exacto, en cambio, sí que se ve afectado; cuantas más semillas se utilicen, más test de distancia tiene que realizar por vóxel.

Hay operaciones sobre los modelos de vóxel que se han implementado directamente en CPU, por ejemplo, dividir un modelo por índice de color o la eliminación de regiones aisladas. El cómputo en CPU supone, en algunas ocasiones,

un coste superior al cálculo del propio DVD. En la tabla 7.3 se ofrece un desglose del tiempo entre GPU y CPU de los algoritmos implementados.

	Modelo A	Modelo B	Modelo C
Método exacto GPU:			
Tiempo CPU (ms)	2 (5%)	23 (23%)	63 (30%)
Tiempo GPU (ms)	34 (95%)	77 (77%)	143 (70%)
CPU + GPU (ms)	36	100	206
Método exacto GPU + eliminar regiones aisladas:			
Tiempo CPU (ms)	9 (15%)	86 (51%)	199 (50%)
Tiempo GPU (ms)	46 (85%)	81 (49%)	136 (40%)
CPU + GPU (ms)	57	167	335
Algoritmo inundación modificado GPU:			
Tiempo CPU (ms)	3 (5%)	21 (15%)	59 (18%)
Tiempo GPU (ms)	32 (95%)	116 (85%)	262 (82%)
CPU + GPU (ms)	35	137	321

Tabla 7.3. Desglose del tiempo de los algoritmos implementados entre CPU/GPU, fragmentando el modelo A, B y C. Entre paréntesis se expresa el procentaje de tiempo respecto a la suma CPU + GPU. Los tiempos se han obtenido generando 64 fragmentos.

El método exacto es algo más eficiente que el algoritmo de inundación, sin embargo, al requerir de una etapa de posprocesado para garantizar que los fragmentos generados sean válidos, ambos algoritmos acaban ofreciendo un rendimiento muy similar.

8 CONCLUSIONES Y TRABAJOS FUTUROS

Este Trabajo Fin de Grado se originó por las necesidades de rendimiento de una aplicación que permitiera fragmentar modelos de vóxel en solo unos pocos milisegundos. Como base, se partía de una implementación en Python que lograba fragmentar objetos utilizando un algoritmo de inundación tardando varios minutos. El principal objetivo de este Trabajo Fin de Grado era conseguir adaptar el algoritmo original al modelo de ejecución de la GPU para mejorar el rendimiento de la

implementación original. Este objetivo se ha cumplido con éxito, la implementación en GPU genera fragmentos consumiendo hasta 500 veces menos tiempo que la implementación original.

También se ha desarrollado métodos adicionales de romper objetos. Utilizando distintas métricas de distancia junto con el DVD, la variabilidad y el realismo de los fragmentos generados se puede incrementar considerablemente.

Respecto a las futuras líneas de trabajo, fundamentalmente hay dos. La primera es mejorar la generación de semillas. En este trabajo todas las semillas han sido generadas aleatoriamente. Sin embargo, esto no es muy realista. En una rotura natural los fragmentos no se reparten de forma aleatoria sobre el objeto fracturado; normalmente allá donde se produzca el impacto, habrá más. Una aproximación sencilla sería determinar aleatoriamente un punto de impacto y generar las semillas en torno a ese punto. La segunda es mejorar el DVD para conseguir que no se generen fragmentos irreales como el de la imagen c de la ilustración 7.5. Bien sea modificando el método o incluyendo algún posprocesamiento sobre los fragmentos.

9 APÉNDICES

9.1 Guía original del Trabajo Fin de Título

(Cód.: 19/20-2797) Simulación de fracturas en modelos de vóxel mediante aceleración GPU

Tutor del TFG: **ANTONIO JESÚS RUEDA RUIZ**
Modalidad: Trabajo Teórico/Experimental | Tipo: TFG Específico
Número máximo de estudiantes: 1 (1 asignados)
Idioma: Castellano
Asignado al alumno con DNI: 26515840Z

Existen métodos sencillos para simular fracturas en modelos de vóxel basados en el cálculo del diagrama de Voronoi discreto. Con este método pueden simularse fácilmente roturas de objetos cerámicos o de cristal. El problema de este método es que tiene un coste computacional importante, lo que las invalida para aplicaciones de tiempo real. Se propone realizar una implementación GPU que permita el cálculo de roturas en pocos milisegundos.

Conocimientos Previos

Se recomienda haber cursado (aunque no es imprescindible):

- Asignaturas de la especialidad de gráficos.
- Asignatura de Programación Hardware

Objetivos del TFG

- Familiarizarse con la programación de GPUs mediante OpenCL/compute shaders.
- Implementar la rotura por Voronoi discreto acelerada por GPU.
- Introducir diferentes modos de rotura.
- Implementar un pequeño visualizador para cargar modelos, calcular roturas y exportar fragmentos.

Metodología a Desarrollar

- Estudiar la programación de GPUs mediante OpenCL y la nueva extensión Compute Shaders.
- Estudiar la implementación de rotura de modelos de vóxel mediante Voronoi discreto tradicional (existe una implementación sencilla en Python).
- Realizar una implementación en GPU del método.
- Integrar la implementación en un pequeño entorno que permita visualizar el modelo antes y después de la rotura, permitiendo cierto nivel de parametrización. Incluir exportación de los fragmentos obtenidos.
- Implementar formas adicionales de rotura.
- Realizar pruebas con diferentes modelos y tomar tiempos de rotura.
- Realizar la memoria del trabajo realizado.

Documentos y Formatos de Entrega

- Memoria del trabajo realizado.

- Código fuente.

9.2 Manual de Usuario

Voxfracturer es el nombre con el que se ha bautizado el programa que implementa el método de fracturación desarrollado en este Trabajo Fin de Grado. Voxfracturer está preparado para ser ejecutado solamente en sistemas GNU/Linux y Windows. Por defecto se incluye el ejecutable para ambos sistemas operativos en el directorio **/bin**. En el fichero **README.md** se detalla cómo compilar el proyecto en GNU-Linux y Windows.

9.2.1 Ejecución

En el directorio **voxfracturer/bin/GNU-Linux** se encuentra el ejecutable del proyecto para GNU-Linux. En **voxfracturer/bin/WIN32** está el ejecutable para Windows. Se puede ejecutar desde el intérprete de órdenes (ilustración 8.1).

9.2.2 Interfaz por Línea de Comandos

Aunque VoxFracturer incluye un pequeño visualizador, la forma principal de interactuar con él es pasando argumentos por línea de comandos:

- **filename**: nombre del modelo que se quiere fragmentar.
- **nfragments**: número de fragmentos a generar.
- **-d/--distance**: métrica de distancia utilizada al calcular el DVD.
- **-e/--extra**: número de fragmentos extra para calcular el DVR.
- **-g/--gui**: si se especifica, se muestra la GUI.
- **-m/--mode**: algoritmo utilizado para generar los fragmentos.
- **-o/--out**: directorio al exportar los fragmentos (por defecto **./out)**.
- **-r/--rem-iso**: posprocesado para eliminar posibles regiones aisladas.

En la ilustración 8.1 se muestra el mensaje de ayuda que emite Voxfracturer si se ejecuta sin recibir ningún argumento y en la ilustración 8.2 un ejemplo de uso.

```
PS C:\Users\jmcl\Desktop\voxfracturer> .\bin\WIN32\Debug\voxfracturer.exe
Missing required arguments:
filename
nfragments

voxfracturer version 1.1, by Jos├® Mar├¡a Cruz Lorite, jmcl0028@red.ujaen.es
Break into pieces a voxel object using discrete Voronoi diagram 3D
Compiled on Sep  5 2020

Usage: voxfracturer filename nfragments

All arguments:
 filename       MagicaVoxel .vox model filename
 nfragments     Number of fragments in which to split the object
-d/--distance   Distance metric: euclidean, manhattan or chebyshev
-e/--extra      Number of extra fragments used in Recursive Voronoi
-g/--gui        Enable graphic user interface
-m/--mode       Fracturing algorithm: flood or naive
-o/--out        Folder where to export the fragments
-r/--rem-iso    Enable removing isolated regions postprocessing
```

Ilustración 8.1. Mensaje de ayuda de Voxfracturer cuando se ejecuta sin recibir argumentos.

```
.\bin\WIN32\Debug\voxfracturer.exe .\model\vasija.vox 8 --mode naive --extra 16
```

Ilustración 8.2. Ejemplo de uso de Voxfracturer.

9.2.3 Interfaz Gráfica

Voxfracturer dispone de una pequeña interfaz gráfica donde pueden visualizarse los fragmentos generados. Para activar la interfaz gráfica debe ejecutarse con la opción **--gui**. Como se aprecia en la ilustración 8.3 en la parte superior izquierda de la interfaz gráfica aparecerá una lista donde podremos seleccionar el fragmento que se quiere visualizar.

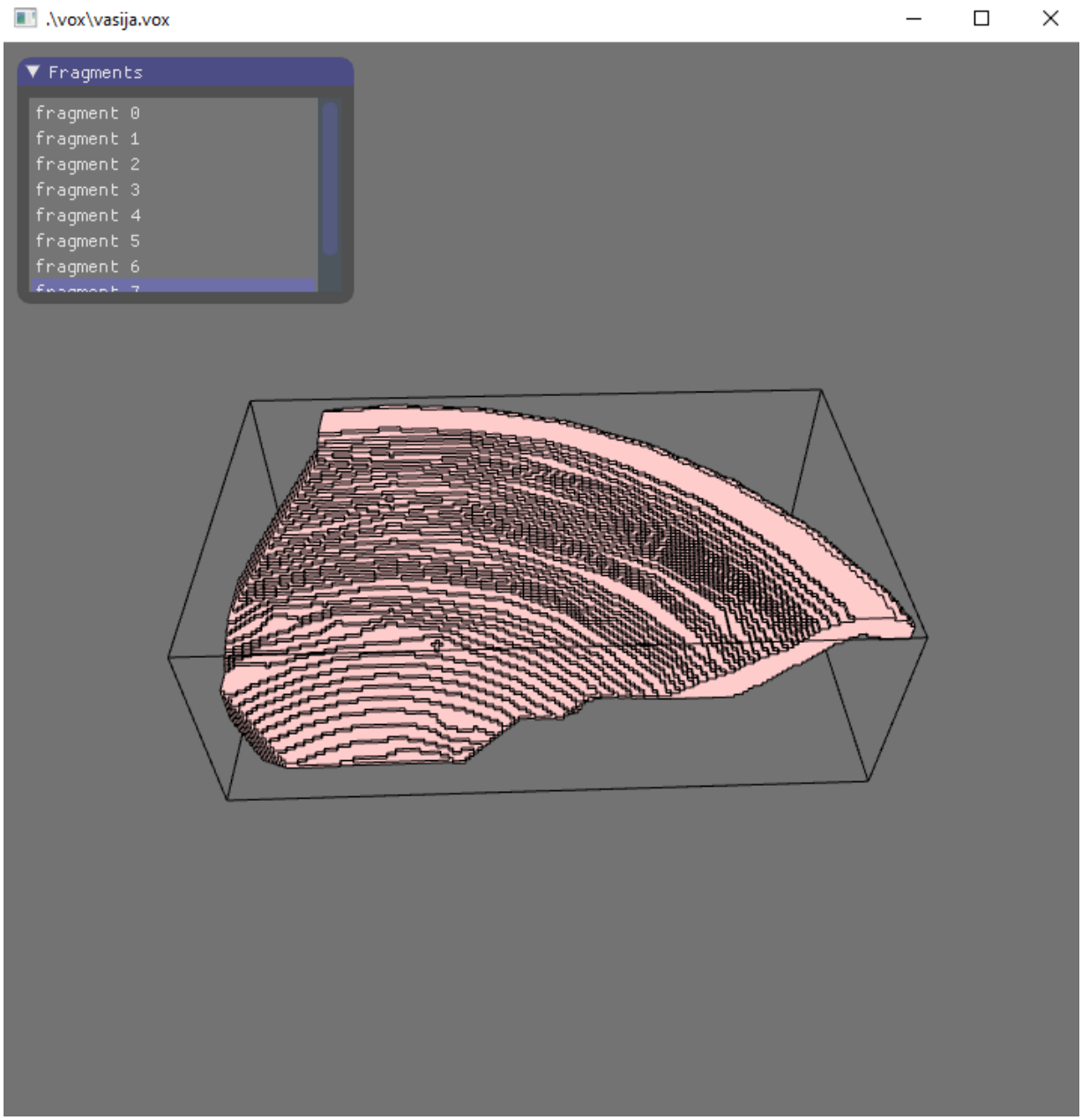

Ilustración 8.3. Visualización de un fragmento en la GUI de Voxfracturer.

Con el ratón puede mover la cámara orbitando alrededor del objeto y con la rueda nos podemos acercar/alejar.

Resumiendo, estas son las formas de interactuar con la interfaz gráfica:

- Seleccionar el fragmento que visualizar en la lista de fragmentos realizando clic con el ratón.
- Orbitar la camara alrededor del fragmento clicando y moviendo el ratón.
- Acercarse/alejarse del objeto realizando scroll up/down con la rueda.

10 BIBLIOGRAFÍA

Ajees, S. D. A. (2009). *Real-time procedural modeling of fracture of brittle materials in games* (Doctoral dissertation).

Cheng, J., Grossman, M., & McKercher, T. (2014). *Professional CUDA C programming*. John Wiley & Sons.

Crawford, M. (2012). *Meshless Computational Methods*. Recuperado en Julio de 2020, desde https://www.asme.org/topics-resources/content/meshless-computational-methods.

Dehal, R. S., Munjal, C., Ansari, A. A., t Kushwaha, A. S. (2018). GPU Computing Revolution: CUDA. In *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)* (pp. 197-201). IEEE.

Domaradzki, J., y Martyn, T. (2018). *Procedural fracture of shell objects*.

Glondu, L., Marchal, M., y Dumont, G. (2012). Real-time simulation of brittle fracture using modal analysis. *IEEE Transactions on Visualization and Computer Graphics*, *19*(2), 201-209.

Hahn, D., y Wojtan, C. (2016). Fast approximations for boundary element based brittle fracture simulation. *ACM Transactions on Graphics (TOG)*, *35*(4), 1-11.

Hirota, K., Tanoue, Y., y Kaneko, T. (1998). Generation of crack patterns with a physical model. *The visual computer*, *3*(14), 126-137.

Kaeli, D. R., Mistry, P., Schaa, D., y Zhang, D. P. (2015). *Heterogeneous computing with OpenCL 2.0*. Morgan Kaufmann.

Katsikadelis, J. T. (2002). *Boundary elements: theory and applications*. Elsevier.

Klein, R. (1989). *Concrete and abstract Voronoi diagrams* (Vol. 400). Springer Science & Business Media.

Khronos Group. (2013). *OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems*. Recuperado en Julio de 2020, desde https://www.khronos.org/opencl/

Liu, N., He, X., Li, S., & Wang, G. (2011). Meshless simulation of brittle fracture. *Computer Animation and Virtual Worlds*, *22*(2-3), 115-124.

Martinet, A., Galin, E., Desbenoit, B., & Akkouche, S. (2004, June). Procedural modeling of cracks and fractures. In *Proceedings Shape Modeling Applications, 2004.* (pp. 346-349). IEEE.

O'brien, J. F., y Hodgins, J. K. (1999). Graphical modeling and animation of brittle fracture. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (pp. 137-146).

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., y Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. In *Computer graphics forum* (Vol. 26, No. 1, pp. 80-113). Oxford, UK: Blackwell Publishing Ltd.

Van der Putte, T. (2009). Using the discrete 3D Voronoi diagram for the modelling of 3D continuous information in geosciences.

Van der Putte, T., y Ledoux, H. (2011). Modelling three-dimensional geoscientific datasets with the discrete voronoi diagram. In *Advances in 3D Geo-Information Sciences* (pp. 227-242). Springer, Berlin, Heidelberg.

Wikiversity. (2016). *Point mass*. Recuperado en Agosto de 2020, desde https://en.wikiversity.org/wiki/Point_mass