Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

UNIVERSIDAD DE JAÉN
Nombre del Centro

Trabajo Fin de Grado

PROTOTIPO PARA LA GENERACIÓN DE FRACTURAS EN MODELOS GEOMÉTRICOS 3D UTILIZANDO LA TÉCNICA AUTÓMATA CELULAR

Alumno: Juan Lendínez Sánchez

Tutor: Juan Roberto Jiménez Pérez
Dpto: Departamento de Informática

Septiembre, 2018

Índice

1. Introducción 5
1.1. Motivación 5
1.2. Objetivos y alcance del proyecto 5
1.3. Metodología de desarrollo 6
1.4. Estructura del documento 6
2. Autómata celular 9
2.1. Componentes de un autómata celular 10
2.2. Aplicaciones del autómata celular 12
3. Metodología 13
3.1. Prototipado iterativo 13
3.2. Planificación 15
3.3. Iteraciones 16
4. Autómata celular versión grafo 21
4.1. Requisitos para la versión grafo 22
4.2. Componentes definidas del diseño grafo 22
4.3. Diseño de clases 23
4.4. Creación autómata celular 24
4.5. Conclusiones 25
5. Autómata celular versión cuadrícula *(Grid)* 27
5.1. Requisitos para la versión cuadrícula (Grid) 27
5.2. Diseño de clases 28
5.3. División individual por triángulo 29
5.4. Conexiones entre cuadrículas 32
5.5. Triangular punto en cuadrícula 34
5.5.1. Mediante sistema de ecuaciones con la fórmula de la distancia 35
5.5.2. Mediante porcentajes inversos 36
5.5.3. Mediante coordenadas baricéntricas 37
5.6. Componentes definidas del diseño grafo 40
5.7. Conclusiones 40
6. Autómata celular versión cuadrícula completa (Full Grid) 41
6.1. Requisitos para la versión cuadrícula completa (Full Grid) 41

6.2. Diseño de clases 42
6.3. Componentes definidas del grafo 43
6.4. Objeto de revolución 43
6.4.1. Subdivisión perfil 44
6.4.2. Revolución 45
6.5. Creación del autómata 47
6.5.1. Función punto en triángulo 48
6.6. Conclusiones 49
7. Algoritmos empleados para la fractura 49
7.1. Algoritmo aleatorio con elementos añadidos 50
7.2. Algoritmo por dirección 52
7.3. Caso frontera en diseño de cuadrícula 56
7.4. Caso frontera en diseño de cuadrícula única/total 56
8. Tiempos 57
9. Presupuesto 59
10. Arquitectura base del prototipo 61
10.1. Capa OpenGL 61
10.1.1. OpenGL 3.3 Core Profile 61
10.1.2. Glew y glfw 62
10.1.3. Glm 63
10.1.4. Diseño de clases 64
10.1.5. Shaders utilizados y modelo de iluminación 66
10.2. Cambio de tecnología a Qt 68
10.2.1. Cambios realizados 68
10.3. Interfaz 70
10.3.1. Diseño interfaz 70
10.3.2. ImGui 71
10.3.3. Inclusión de ImGui 72
10.3.4. Ejemplo de uso 72
10.3.5. Qt 73
11. Evaluación y aspectos de iteraciones 77
12. Conclusiones y trabajo futuro 79
Bibliografía 80

Índice ilustraciones

Ilustración 2.1 Ejemplo de un autómata celular 9
Ilustración 2.2 Vecindad del autómata celular 11
Ilustración 3.1 Fases del desarrollo iterativo 14
Ilustración 3.2 Gráfico funcionalidad-tiempo con iteraciones 14
Ilustración 3.3 Diagrama Gantt para la planificación 15
Ilustración 4.1 Estructura autómata celular versión grafo 20
Ilustración 4.2 Diseño UML de las clases para el diseño grafo 22
Ilustración 5.1 Diseño UML de las clases para el diseño cuadrícula 26
Ilustración 5.2 Demarcación del triángulo en cuadrículas 29
Ilustración 5.3 Formación de conexiones entre cuadrículas/triángulos 30
Ilustración 5.4 Ejemplo de triangulación 34
Ilustración 5.5 Autómata celular para modelo roca con menos precisión (versión interpolación por porcentajes inversos) 34
Ilustración 5.6 Autómata celular para modelo roca con más precisión (versión interpolación por porcentajes inversos) 33
Ilustración 5.7 Coordenadas baricéntricas 36
Ilustración 5.8 Autómata celular para modelo roca con menos precisión (versión interpolación por coordenadas baricéntricas) 37
Ilustración 5.9 Autómata celular para modelo roca con más precisión (versión interpolación por coordenadas baricéntricas) 37
Ilustración 6.1 Diseño UML de clases para la versión cuadrícula completa 40
Ilustración 6.2 Subdivisión de un perfil 2D 42
Ilustración 6.3 Formación de triángulos en la revolución del perfil 44
Ilustración 6.4 Generación de normales en los puntos revolucionados 44
Ilustración 6.5 Creación del autómata celular para la versión cuadrícula completa 46
Ilustración 6.6 Punto en triángulo 47
Ilustración 7.1 Fractura aleatoria con la versión grafo del autómata 50
Ilustración 7.2 Fractura por dirección en versión cuadrícula (Grid) 53
Ilustración 7.3 Fractura por dirección con la versión cuadrícula completa(Full Grid) 53
Ilustración 10.1 Flujo de OpenGL 60
Ilustración 10.2 Ejemplo de uso biblioteca glm 61
Ilustración 10.3 Diseño UML de clases para la aplicación base de OpenGL 62
Ilustración 10.4 Vectores para la iluminación 65

Ilustración 10.5 Fórmula de la iluminación 65
Ilustración 10.6 Ejemplo uso ImGui(1) 69
Ilustración 10.7 Ejemplo uso ImGui(2) 70
Ilustración 10.8 Ejemplo uso ImGui(3) 70
Ilustración 10.9 Distribución de layout en la interfaz.....71
Ilustración 10.10 Resultado final de la interfaz.....72

Índice tablas

Tabla 8.1 Tabla tiempos de ejecución en distintas versiones del autómata 55
Tabla 8.2 Tabla tiempos de ejecución en distintas versiones del autómata 56
Tabla 9.1 Tabla del presupuesto del proyecto.....57

1. Introducción

En este documento se presenta la memoria de mi trabajo de fin de grado en Ingeniería Informática sobre fracturas en modelos geométricos 3D usando la técnica del autómata celular, a lo largo del documento se explicara todo lo referente a la creación del proyecto desde la motivación hasta las conclusiones finales pasando por todas las técnicas, diseños e implementaciones realizadas. Todo esto cubierto en una metodología que marcaba el ritmo de trabajo como se verá más adelante.

1.1. Motivación

Este proyecto surge como una propuesta de mi tutor, Juan Roberto Jiménez Pérez, hacia mi persona. A lo largo del grado de Ingeniería Informática mi interés siempre ha recaído en la computación de gráficos 3D y los usos de la misma en diferentes campos, de esta manera, este TFG intenta investigar como esta rama de la informática se puede fusionar con el campo de la salud para crear algo no solo interesante pero también con cierta repercusión. Con ese propósito surge este TFG centrándose en crear una representación 2D de la superficie de un modelo 3D para ser capaces de aplicar algoritmos sobre esa representación (en este caso de fractura), con la base de la técnica del autómata celular.

1.2. Objetivos y alcance del proyecto

Desde un primer momento este proyecto quedo definido como prototipo puesto que tiene una fuerte componente de investigación, crear una versión completamente funcional podría ser algo realmente complicado hasta que no se profundizase lo suficiente en el tema. Sin embargo si había una serie de objetivos propuestos para asegurar cierto éxito del trabajo, entre los objetivos propuestos se encuentran:

Diseñar una estructura para subdividir un modelo geométrico 3D, esta estructura debería ser hasta cierto punto customizable proporcionando una subdivisión homogénea en células para posteriormente utilizarlas con ciertos algoritmos.

Proporcionar una base gráfica donde trabajar, como objetivo lógico era necesario alguna plataforma donde poder renderizar ciertos modelos y mostrar la fractura además, adicionalmente se podría mostrar la propia subdivisión.

Proporcionar una salida de los datos obtenidos, una vez realizada la fractura es necesario mostrar de alguna manera esos datos calculados.

1.3. Metodología de desarrollo

Desde un primer momento y como el propio nombre del proyecto indica se buscaba un prototipo para fracturas 3D utilizando el autómata celular como técnica. No se podía asegurar una versión completamente funcional puesto que había una fuerte componente de investigación, por eso mismo, la metodología de desarrollo esta a su vez ligada con este planteamiento.

La metodología a seguir ha sido **Prototipado Iterativo (o Desarrollo Iterativo)**, la cual será en detalle explicada junto a todas las iteraciones de la misma en la sección correspondiente (sección 3).

1.4. Estructura del documento

Este documento estará separado en distintas secciones fácilmente diferenciables gracias al índice donde se explicarán las bases del mismo, a partir de ahí, ira escalando hasta ver esas bases fácilmente reconocibles en el resultado final.

El capítulo 1 es la introducción donde se proporcionara una breve descripción del proyecto (alcance, metodología, descripción) además de la motivación del mismo.

En el capítulo 2 es introducido el concepto de autómata celular y las características del mismo para, más tarde, desarrollarlo en diseños específicos.

En el capítulo 3 la metodología es explicada en detalle junto a todas las iteraciones del proyecto.

En el capítulo 4, 5 y 6 han sido detallados los tres diseños del autómata celular (diseño grafo, diseño cuadrícula y diseño cuadrícula completa).

En el capítulo 7 se han comentado los algoritmos utilizados para fracturar el modelo.

En el capítulo 8 se habla sobre la aplicación base para dibujar en OpenGL.

En el capítulo 9 se comenta el presupuesto requerido en total para el proyecto y todos los elementos que lo conforman.

En el capítulo 10 se explica la arquitectura base, los cimientos del proyecto en el que se monta todo. Este capítulo esta de los últimos precisamente porque esta arquitectura principal no es el primer objetivo del TFG sino la consecuencia del mismo.

Finalmente en el capítulo 11 se finaliza con las conclusiones y el trabajo futuro que este proyecto puede tener en caso de una continuación del mismo.

2. Autómata celular

El autómata celular es una técnica descubierta entre los años 1940 y 1950 por Konrad Zuse y Stanislaw Ulam pero desarrollado por John Von Neumann, John Horton y Stephen Wolfram. Siguiendo como referencia el comportamiento de las células se creó un sistema para simular dicho comportamiento donde las distintas células de ese sistema crecerían, se reproducirían y finalmente morirían siguiendo un modelo matemático **[Fernando, 2016]**.

Un autómata celular se puede definir como un sistema dinámico que evoluciona discretamente al cambiar el estado de sus componentes (las células) siguiendo un modelo matemático. Este sistema está compuesto por un conjunto de células (también llamadas celdas) que al poder tomar una serie de estados finitos simulan un sistema vivo que va evolucionando de forma discreta a través del tiempo para un determinado fin.

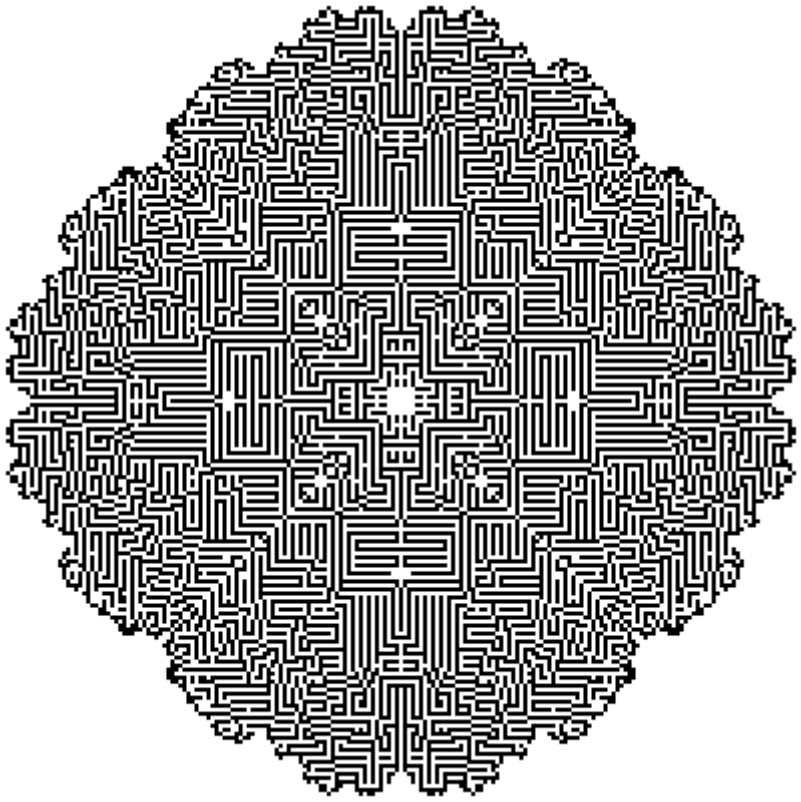


Ilustración 2.1 Ejemplo de automata celular

2.1. Componentes de un autómata celular

Los componentes que forman un autómata celular son:

Espacio: Determina que representación va a ser utilizada para el autómata, esto indica la dimensión del mismo y por ende la división que se ha de realizar para cada célula. Puede ser un plano de 2 dimensiones o cualquier espacio n-dimensional.

Estados: Determina el número finito de estados que cada célula es capaz de adoptar.

Vecindad: Determina el número de vecinos que cada célula tiene, esto se traduce en las células con las que puede comunicarse cada célula. Siguiendo un espacio 2D, el cual es el más usado, tenemos dos estrategias, por un lado está la **vecindad de Von Neumann** que comprendería los vecinos de la derecha, izquierda, arriba y abajo, es decir, suponiendo x(i,j) como la célula sus vecinos serían x(i + 1,j), x(i - 1,j), x(i,j + 1) y x(i,j - 1) resultando en 4 vecinos. La segunda estrategia sería la **vecindad de Moore** que englobaría los mismos vecinos que la de Von Neumann pero añadiría además los vecinos de las diagonales de la siguiente manera, x(i - 1,j - 1), x(i + 1,j - 1), x(i + 1,j + 1) y x(i - 1,j + 1) resultando en 8 vecinos.

Algoritmo de transición: Determina como van a evolucionar las distintas células, este algoritmo normalmente esta definido por una serie de ecuaciones dependiendo de la complejidad del mismo.

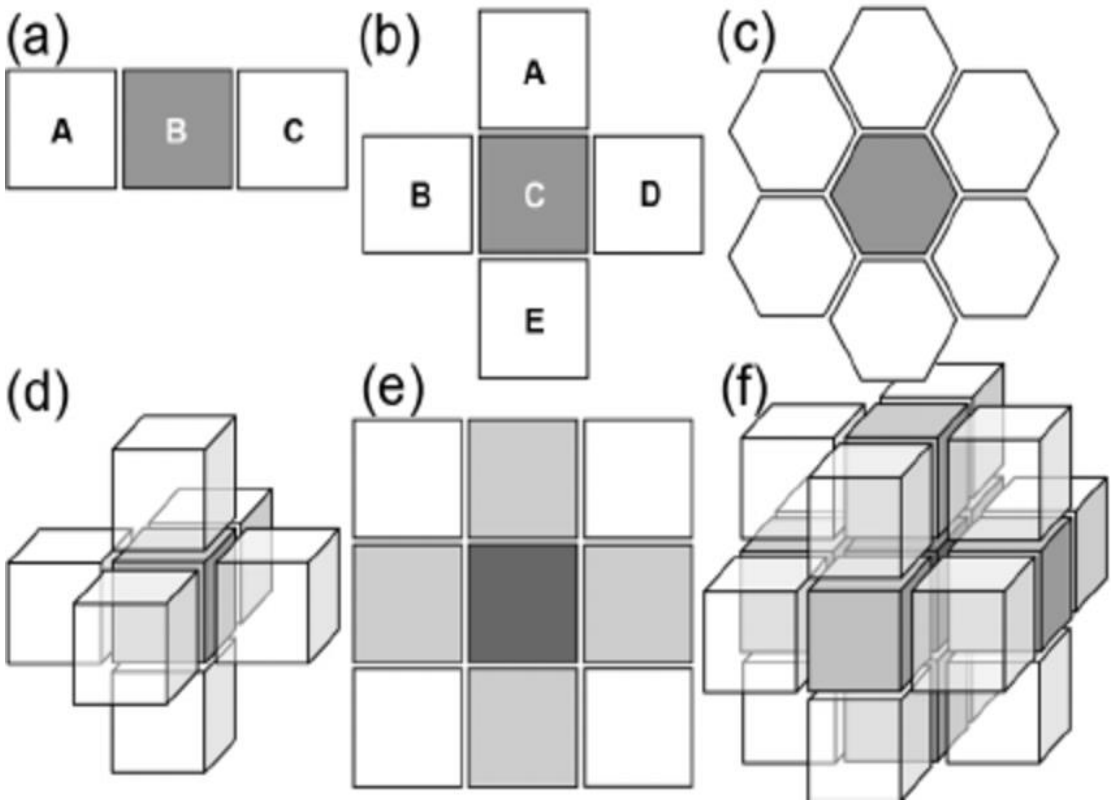


Ilustración 2.2 Vecindad del automata celular

Además de los componentes que un autómata debe tener, la frontera del autómata debe ser definida, dicha frontera corresponde a todas las células que se salen del dominio y dependiendo de la naturaleza del problema a resolver puede ser crucial definir un tipo u otro **[Sage, 2011]**. Hay varias maneras de definir como se comportará las células de la frontera:

Abierta: En este caso todas las células que queden fuera de la frontera tendrán siempre un valor preestablecido.

Reflectora: En este caso todas las células que se queden fuera de la frontera tendrán el mismo valor que las células de dentro simulando un efecto espejo.

Nula: En este caso la frontera es nula pues el autómata es infinito y no tiene fin.

Circular: En este caso no hay células que se salgan de la frontera pues el autómata al cruzar la frontera vuelve al inicio similar a una superficie esférica o un cilindro.

Finalmente también es necesario definir el estado inicial en el que cada célula se encontrará al inicio del autómata celular siendo posible varias soluciones, todas las células pueden adoptar un valor aleatorio en un rango específico, pueden adoptar el mismo valor o pueden adoptar una serie de valores relativos a su posterior uso. La manera de abordar esta decisión dependerá de la naturaleza del problema y el objetivo que se desee alcanzar.

2.2. Aplicaciones del autómata celular

Gracias a la complejidad que puede alcanzar el autómata al ser capaz de simular comportamientos complejos, las áreas donde es utilizado son numerosas **[Ada-Robin]**:

En **Geografía** es usado para simular el comportamiento de terremotos, avalanchas y diversos fenómenos naturales o en áreas más específicas como urbanismo se puede usar para modelar áreas urbanas.

En **Medicina** y campos relativos es usado para simular el comportamiento de enfermedades y distintas patologías tales como predecir el crecimiento de un tumor cerebral.

En **Biología** es usado para simular ecosistemas.

En **Informática Gráfica** para representar comportamientos complejos como el comportamiento de fluidos, corrosión en objetos 3D ect.. Relacionado además con ciertas áreas de fisioterapia o medicina y siendo el protagonista de este TFG también es usado para **fracturas** o para grietas en modelos de huesos.

Estos son solo unos de sus muchos usos ya que al ser una tecnología aún en desarrollo faltan muchas áreas por explorar.

3. Metodología

Como se ha mencionado en la Introducción la metodología utilizada en el proyecto es el prototipado iterativo, debido a la naturaleza del proyecto esta metodología se le ajusta perfectamente puesto que se crean varias versiones del autómata mejorando cada vez fallos del anterior hasta llegar al resultado deseado.

3.1. Prototipado iterativo

El prototipado o desarrollo iterativo consiste en una serie de iteraciones las cuales comienzan con una versión del producto que tras un número de fases acaban con una versión mejorada del mismo. Para cada iteración se proponen una serie de objetivos y tras la finalización de la misma se revisa si se han cumplido los objetivos propuestos, además se pasa a la siguiente iteración con nuevos objetivos o viejos objetivos si no se han cumplido en la anterior iteración.

Este proceso de desarrollo suele ser utilizado en interfaces donde al final de cada iteración hay un grupo de personas encargadas de probar la interfaz resultante y comentar su experiencia, a partir de esa experiencia se propondrán nuevos objetivos para mejorar la interfaz en la siguiente iteración. En el caso de este TFG las personas encargadas de probar el producto serán el tutor y yo mismo que a su vez propondrán nuevos objetivos para futuras iteraciones.

Ilustración 3.1 Fases del desarrollo iterativo

Como se muestra en la imagen para cada iteración se pasan por una serie de fases, primero en la fase de requerimientos estarían los objetivos para la iteración que pasaran por un análisis y diseño y posteriormente a la implementación de esos objetivos. En la siguiente fase se harán pruebas en el producto tras la actualización para ver si existe algún problema que no fuese esperado, tras esas pruebas se pasara a la última fase donde se evaluara si se ha completado los objetivos y se crearan los nuevos (en caso de que no fuese la última iteración).

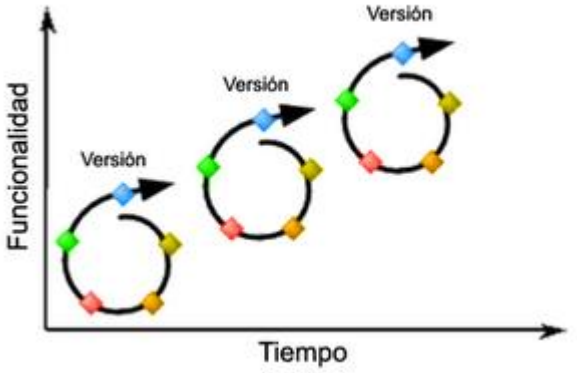


Ilustración 3.2 Gráfico funcionalidad-tiempo con iteraciones

El número de iteraciones dependerá de la complejidad del proyecto y del tiempo disponible por lo que no hay un número determinado. De la misma manera una iteración no tiene porque salir de forma exitosa incluso aun cumpliendo todos los objetivos propuestos por fallos inesperados o resultados raros.

3.2. Planificación

Para la planificación de este proyecto se propuso una serie de tareas, primero deberá existir una fase de recopilación de la información con distintos artículos y otro material necesario. Seguido a esta fase, primero se deberá diseñar e implementar la arquitectura principal en la que se montara el proyecto, a partir de este momento ya será posible empezar a probar representaciones distintas para el autómata celular que culminaran con algoritmos de propagación para fracturar. Finalmente una fase para final para testear toda la implementación y corregir errores.

Para ilustrarlo de mejor manera se muestra un diagrama de Grantt con todas las tareas repartidas en los meses.

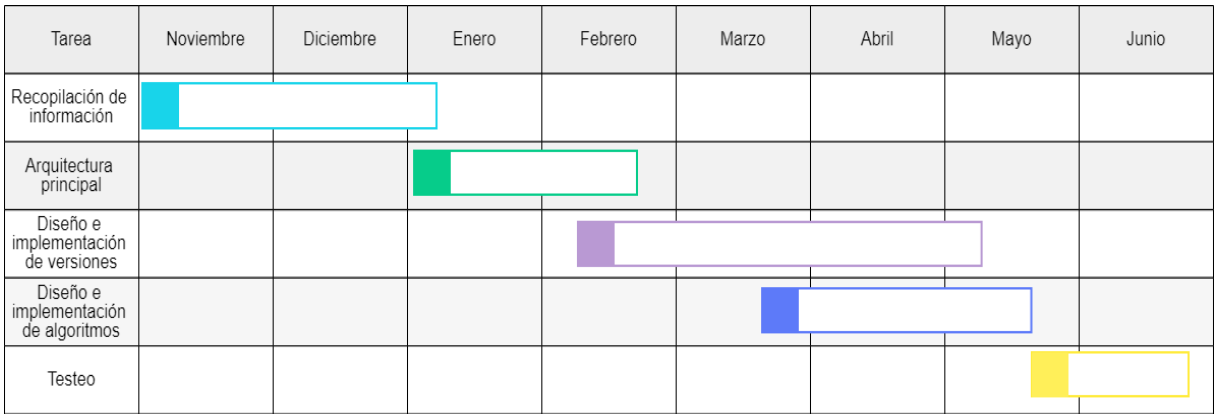


Ilustración 3.3 Diagrama Grantt para la planificación

Como se observa en la ilustración 3.1 cada tarea no necesita necesariamente que acabe su predecesora, es posible que sin llegar a acabar una tarea se pueda empezar la siguiente dando margen para ambas.

3.3. Iteraciones

Cada una de las iteraciones será explicada y razonada incluyendo el tiempo necesario, objetivos de la misma y una conclusión.

- **Iteración 1: Recopilación de información**

  Duración de la iteración: 9 de octubre a 7 de enero.

  Esta iteración es la más larga puesto que fue el comienzo de todo y el objetivo principal era investigar, a su vez, el proyecto aún no había comenzado realmente.

  Objetivos

  1. Investigar autómata celulares previamente creados en artículos de investigación.

  2. Investigar base de la aplicación para empezar el diseño próximamente.

- **Iteración 2: Implementación Arquitectura base de OpenGL**

  Duración de la iteración: 7 de enero a 12 de febrero.

  Una vez completada la fase de investigación, es turno de diseñar e implementar la base de la aplicación para renderizar modelos. Además también es necesario documentar todo el proceso.

  Objetivos

  1. Diseñar e implementar la base de aplicación para dibujar.

  2. Documentar todo el proceso.

  3. Empezar a diseñar la primera versión del autómata.

- **Iteración 3: Diseño e Implementación Versión Grafo e inclusión interfaz**

  Duración de la iteración: 12 de febrero a 5 de marzo.

  Para esta iteración el primer diseño del autómata celular ira tomando forma, adicionalmente se empezará a buscar una interfaz.

  Objetivos

  1. Diseñar e implementar la versión del autómata celular grafo.
  2. Documentar todo el proceso.
  3. Buscar librerías para incluir una interfaz.
  4. Diseñar e implementar fractura aleatoria.

- **Iteración 4: Diseño versión cuadrícula**

  Duración de la iteración: 5 de marzo a 15 de abril.

  Siguiendo la propuesta de la iteración anterior se empieza a diseñar la siguiente versión del autómata celular estilo cuadrícula para solucionar problemas de la primera implementación.

  Objetivos

  1. Diseñar e implementar la versión del autómata celular cuadrícula.
  2. Documentar todo el proceso.

- **Iteración 5: Implementación diseño cuadrícula y cambio a Qt**

  Duración de la iteración: 15 de abril a 10 de junio.

  Se continuo la implementación de la versión cuadrícula resolviendo fallos además se completo el cambio de tecnología a Qt manteniendo OpenGL. Esta iteración fue un poco más larga que las demás debido a un periodo de inactividad.

  Objetivos

  1. Acabar la implementación de la versión cuadrícula.
  2. Documentar todo el proceso.
  3. Realizar el cambio de tecnología a Qt.

- **Iteración 6: Implementación algoritmo de fractura e interfaz**

  Duración de la iteración: 10 de junio a 15 de julio.

  Para esta iteración se empezó el diseño e implementación del algoritmo de fractura por dirección, además el diseño e implementación de la interfaz en qt también fue realizado.

  Objetivos

  1. Diseñar e implementar el algoritmo de fractura por dirección.
  2. Documentar todo el proceso.
  3. Diseñar e implementar la interfaz.

- **Iteración 7: Diseño e implementación de versión cuadrícula única**

  Duración de la iteración: 15 de julio a 10 de agosto.

  Para esta iteración se empezó la versión cuadrícula única del autómata celular además de unir la funcionalidad de las clases a la interfaz previamente creada.

  Objetivos

  1. Diseñar e implementar la versión cuadrícula completa.

  2. Documentar todo el proceso.

  3. Unir la funcionalidad a la interfaz.

  4. Adaptar el algoritmo de fractura por dirección a la versión de la cuadrícula.

- **Iteración final: Documentación atrasada y corrección de fallos**

  Duración de la iteración: 10 de agosto a 3 de septiembre.

  Para la iteración final simplemente se finalizó toda la documentación sobrante que no se había completado y se perfilo detalles y pequeños fallos de la aplicación.

  Objetivos

  1. Acabar la documentación.

  2. Sacar versión funcional de la aplicación (deploy).

  3. Resolver pequeños fallos además de documentar el código.

4. Autómata celular versión grafo

El autómata celular se puede diseñar de varias maneras dependiendo del uso, para este TFG han sido creadas tres posibles diseños cada uno con sus ventajas e inconvenientes. Ambos diseños han sido testeados para determinar cuál se adecua mejor a la tarea propuesta que sería fracturar el modelo. Este primer diseño está fuertemente ligado a la estructura de un grafo puesto que el funcionamiento es similar, habrá nodos y conexiones entre los mismos, cada nodo contendrá la célula perteneciente al autómata celular que simulara la estructura ósea de un hueso.

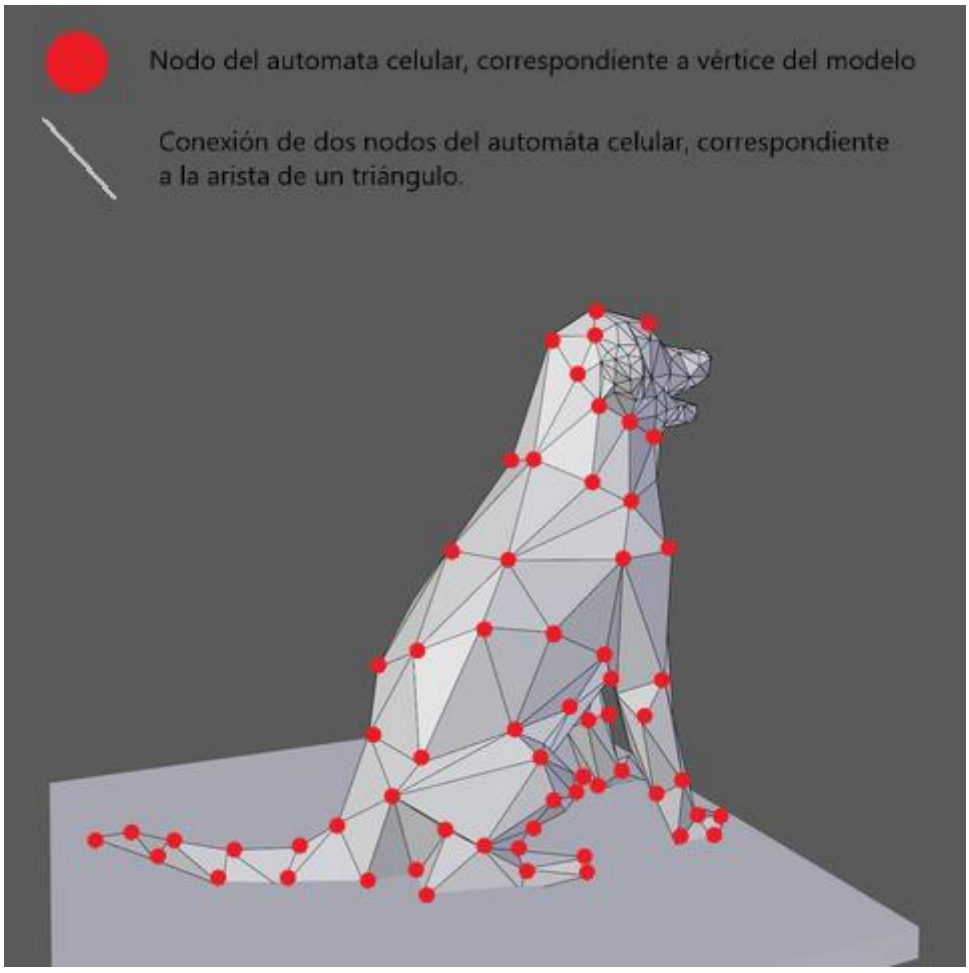


Ilustración 4.1 Estructura automata celular version grafo

Al igual que un grafo tendríamos una serie de nodos, cada nodo será el equivalente a un vértice del modelo así que el número de nodos será el número de vértices mientras que los vecinos de un nodo serán los vértices de los triángulos cuyo nodo original tenga. Por cada vértice que pertenezca a un triángulo habrá otro dos vértices que tendrán que ser vecinos de el vértice original.

4.1. Requisitos para la versión grafo

Para esta versión del autómata celular los requisitos son simples puesto que esta versión no es más que una primera toma de contacto con la técnica del autómata celular.

- **Representar una capa 2D que envuelva al modelo,** este requisito se repetirá en las siguientes versiones debido a que es el principal objetivo. Sin embargo para esta versión no será requerido una representación extensa al ser una primera toma de contacto.
- **Formar conexiones entre los elementos de la capa 2D,** otra vez este requisito se repetirá en las siguientes versiones de la misma manera que el anterior puesto que es necesario una vecindad en la que poder apoyarse.

4.2. Componentes definidas del diseño grafo

Las componentes del autómata celular, previamente explicadas, ahora son definidas en profundidad para este diseño particular.

Espacio: Para este diseño el espacio es 3D puesto que cada punto del grafo está en el espacio 3D, no obstante, siempre se trabaja con la superficie del modelo así que se simula un espacio 2D.

Estados: Los estados que la célula puede adquirir son solo dos, puede estar rota o intacta. El estado inicial de cada célula es intacta.

Vecindad: El número de vecinos que cada nodo puede variar dado que depende de la geometría del modelo, al ser cada nodo un vértice sus vecinos serán los vértices correspondientes a los triángulos que pertenezca.

Algoritmo de transición: Para este diseño el algoritmo será sencillo, simplemente se seleccionará aleatoriamente la célula que se rompe, sin embargo, habrá ciertas restricciones en el algoritmo para mejorar el resultado de las fracturas que se comentará posteriormente.

Frontera: La frontera del autómata será circular debido a que cada célula se comunica con sus vecinos dando la vuelta al modelo.

4.3. Diseño de clases

Una vez establecido las características para el autómata celular, se pasa a un diagrama de clases para su posterior implementación.

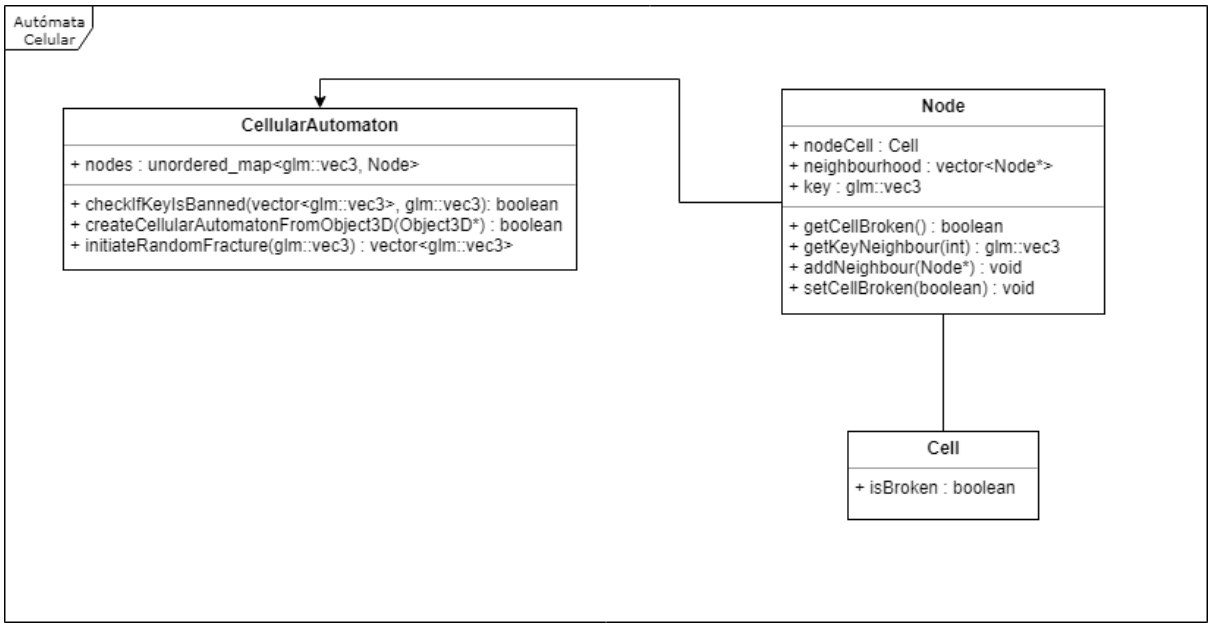


Ilustración 4.2 Diseño UML de las clases para el diseño grafo

La estructura del autómata está separada en tres clases, cada una con una función. Se tendrá por un lado los nodos que corresponden a la clase **"Node"** que deberá almacenar su vecindario y la célula, que será la clase **"Cell"**. Ambas ideas deben estar separadas a la hora de la implementación porque pese a que se podrían fusionar ambas en una sola conteniendo el nodo los atributos necesarios para considerarse a su vez célula, es recomendable separar la estructura del autómata de la capa más abstracta la cual sería las células. Además y puesto que hay otro diseño para la estructura del autómata se reciclará la clase célula "Cell" añadiendo más atributos a la misma. La clase **"CellularAutomaton"** será la que contenga todos los nodos para el posterior procesamiento de la fractura mediante el algoritmo aleatorio.

CellularAutomaton: Clase que contendrá el autómata celular en sí, en esta clase se implementará el algoritmo y se cargará el autómata a partir del modelo 3D. Se almacenarán todos los nodos en un mapa desordenado para acceso O(1), la llave del mapa será el punto 3D (glm::vec3).

Node: Clase para encapsular cada nodo del autómata celular, guardará a sus vecinos a través de un vector de punteros para evitar tener duplicados y tener la información exacta. Además también contendrá la célula (clase “Cell”) que será la que proporcione la información necesaria para el algoritmo.

Cell: Clase encargada de representar la célula para determinar el comportamiento en el algoritmo, en este caso, al ser el algoritmo aleatorio lo único que se almacenará será si la célula se ha roto.

4.4. Creación autómata celular

Para cargar el autómata celular es necesario un objeto de la clase Object3D que además debe ser válido. Para que sea válido basta con que el objeto contenga los vértices y los índices de los triángulos puesto que son la base del autómata, en caso de no ser así el autómata no se creará dando lugar a un error. Al estar muy relacionada la estructura del objeto 3D con la estructura del autómata celular su conversión es sencilla, se tratará de una serie de pasos:

1. Se recorrerá el vector de índices de triángulos de 3 en 3 índices para coger cada triángulo.

2. Se comprobará uno a uno si esos índices (que corresponden a un vértice) existen ya como nodo en la estructura del autómata.

   a. En caso negativo, se creará un nodo que tendrá como clave el vértice 3D.

   b. En caso afirmativo no se hará nada.

3. Una vez que es seguro que los nodos de esos índices existen accederemos a ellos para guardarlos en variables locales.

4. Finalmente para cada nodo de esos índices le añadimos dos vecinos los cuales serían los otros dos nodos (correspondientes a los vértices de los índices de ese triángulo) y se lo pasamos como referencia para no tener duplicados.

5. Se vuelve al punto 2 para cada triángulo del modelo hasta finalizar.

4.5. Conclusiones

Esta representación del modelo conlleva una serie de ventajas e inconvenientes, por un lado debido a la simplicidad del mismo no requiere mucho tiempo de computación lo cual se traduce en que sin importar lo grande que sea el modelo no se requiere de tiempos de espera a la hora de crear el autómata (más información será proporcionada en lo referente a tiempos en la sección [poner]). En adición, la propia implementación del algoritmo para el autómata es también sencilla de implementar por la misma razón que anteriormente.

Por otra parte como inconvenientes el primero y único que se presenta sería la representación demasiado ligada a la geometría del modelo que fuerza a que dicha geometría sea fiable. Si en la propia geometría se encuentran partes más densas (con más vértices debido al detalle en cierta parte) condiciona que el autómata esté mejor representado mientras que en partes más simples que quizás no requieran de dicha complejidad se forma una versión más simple del autómata; al final el resultado sería un modelo para nada heterogéneo que no es representado adecuadamente.

5. Autómata celular versión cuadrícula *(Grid)*

Debido en gran parte al problema de la heterogeneidad en la anterior representación del autómata celular se buscó otro diseño con el fin de solventar dicho problema. La solución a la que se llegó fue representar el modelo a un diseño estilo cuadrícula (Grid), en esta solución se propone que la superficie del modelo entero se representa como una cuadrícula con mayor o menor detalle permitiendo así una heterogeneidad casi perfecta y customizable. Con ese fin el proceso se divide en dos partes claramente diferenciadas, primero dividir individualmente cada triángulo en una cuadrícula (Grid) teniendo en cuenta que el tamaño de cada triángulo es distinto, y segundo, establecer las conexiones entre todos las cuadrículas previamente formadas para formar el autómata celular. A partir de esa estructura inicializada se podrá implementar cualquier tipo de algoritmo para completar el objetivo.

5.1. Requisitos para la versión cuadrícula (Grid)

Para esta versión del autómata celular los requisitos cambian ligeramente con respecto al anterior diseño añadiendo uno nuevo.

- **Representar una capa 2D que envuelva al modelo con detalle personalizable,** de forma parecida a la anterior versión es necesario crear esa capa 2D para representar al modelo por la superficie. Para esta versión en particular será necesaria un detalle personalizable que permita aumentar el detalle de la representación.
- **Formar conexiones entre los elementos de la capa 2D,** de la misma manera que en la anterior versión es necesario formar una vecindad en la representación elegida.
- **Apoyarse de la representación propuesta por Stephane Gobron y Norishige Chiba [Stephane y Norishige, 1998-1999].**

5.2. Diseño de clases

El diagrama de clases correspondiente no será muy extenso, por un lado tendremos la clase principal que guardara la estructura del autómata celular (*CellularAutomatonV2*) en esta clase se almacenará todas las cuadrículas existentes (tantas como triángulos) bajo la estructura de un mapa desordenado cuya clave será un entero marcando el índice del triángulo al que corresponde dicha cuadrícula (Grid). Para la clase cuadrícula (Grid en el diagrama) guardaremos una matriz de casillas (SquareGrid), la matriz siempre será cuadrada por eso solo es necesario guardar una variable para el tamaño de la misma (sizeM). Además también será necesario guardar las coordenadas de los tres puntos del triángulo para tener acceso rápido al mismo, al ser cada cuadrícula la representación de un triángulo se establecen tres puntos y se trasladan a la cuadrícula para una representación fiel, dicho cambio se comentará más adelante.

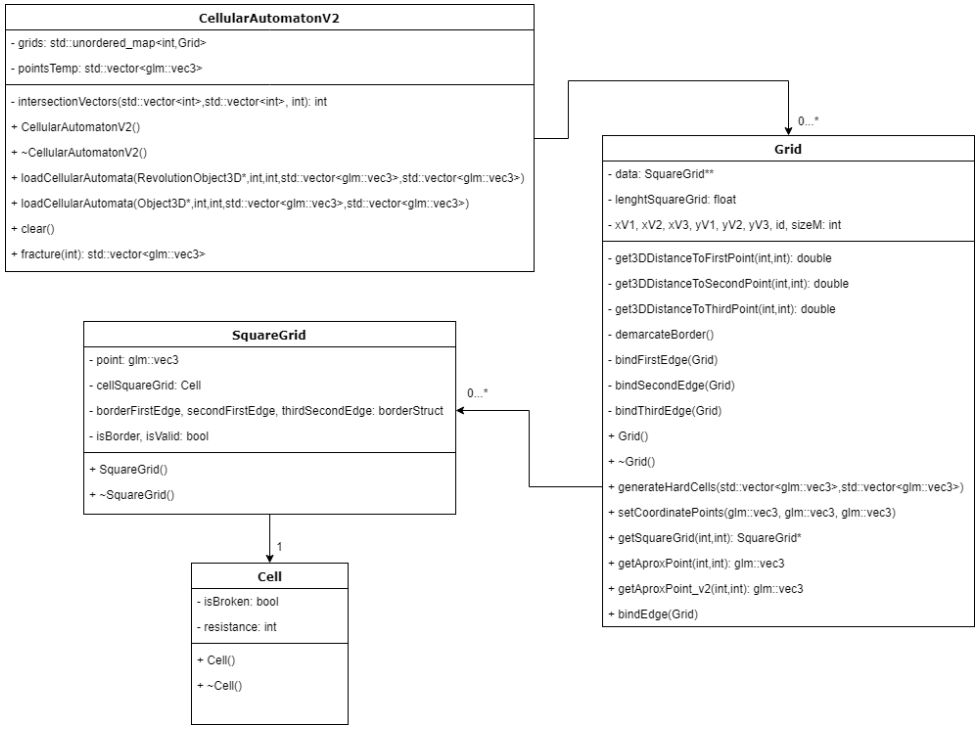


Ilustración 5.1 Diseño UML de las clases para el diseño cuadrícula

En la ilustración 5.1 se han omitido las funciones de obtención y modificación de variables de clase puesto que son irrelevantes.

5.3. División individual por triángulo

Como se ha comentado anteriormente la primera etapa para crear este tipo de autómata sería dividir cada triángulo en una cuadrícula (Grid), el primer problema que surge al intentar hacerlo es bajo que criterio es creado un grid más grande o más pequeño puesto que cada triángulo va a tener un tamaño distinto. La solución consiste en establecer de antemano dos valores que determinarán el tamaño mínimo y máximo de cada cuadrícula (Grid), además de estos valores se calculará la mínima y máxima arista de todos los triángulos. De esta manera tenemos dos rangos, uno de ellos para el tamaño de las cuadrículas (Grids) y otro para determinar la mínima y máxima arista de los triángulos; para cada triángulo se seleccionará la mayor arista para calcular su distancia y posteriormente mediante la siguiente fórmula pasaremos del rango de distancias mínimas y máximas de las aristas a el tamaño necesario de la cuadrícula.

```
valorNuevo = (((valorAntiguo - minimoRangoAntiguo) * (maximoRangoNuevo - minimoRangoNuevo)) / (maximoRangoAntiguo - minimoRangoAntiguo)) + minimoRangoNuevo
```

Tanto el *minimoRangoAntiguo* como *maximoRangoAntiguo* hacen referencia a la mínima y máxima arista (en distancia) de todos los triángulos mientras que el *maximoRangoNuevo* y *minimoRangoNuevo* hacen referencia al tamaño mínimo y máximo de la cuadrícula que se pasa como datos de entrada. Gracias a este método es posible controlar el detalle del propio autómata además de ser una representación bastante realista y homogénea.

Tras crear la cuadrícula con el tamaño calculado en el paso anterior la siguiente tarea será pasar los 3 puntos correspondientes del triángulo a la cuadrícula, para ello será necesario hacer uso de las distancias 3D. En el anterior paso se calculaba la mayor distancia de las 3 aristas pertenecientes al triángulo, esa distancia se convertía a un entero que sería el tamaño de la cuadrícula. Con esas dos variables es posible averiguar la distancia 3D que atraviesa cada casilla, esta distancia es pasada a la clase cuadrícula para futuras operaciones. Ahora, para pasar los tres puntos a casillas en la cuadrícula es necesario que primero la arista mayor conforme la base del triángulo, después los puntos se pasan de la siguiente manera:

El primer punto es fijo, es decir, siempre toma las mismas coordenadas dentro de la cuadrícula, **(0,0)**.

El segundo punto es calculado a partir de la distancia 3D entre el primero y el segundo, esta distancia junto a la distancia por casilla calculada anteriormente resultan en un punto en el eje X, **(x,0)**.

Finalmente para el tercer punto es necesario calcular la altura del triángulo para posteriormente aplicar la misma operación que en el segundo punto con la distancia 3D por casilla. Para calcular la altura de un triángulo es necesaria el área:

$$\text{alturaTriangulo} = \frac{Area*2}{base}$$

Y para calcular el área sin saber la altura se recurre a la fórmula de Herón que nos permite calcular el área de un triángulo mediante las longitudes de las aristas de la siguiente manera:

$$\text{Area} = \frac{1}{4}\sqrt{(a^2+b^2+c^2)^2 - 2(a^4+b^4+c^4)}$$

Siendo a, b y c la longitud de las tres aristas respectivamente.

También es necesario considerar delimitar los límites del triángulo en la cuadrícula (Grid) puesto que al crear la cuadrícula se crea cuadrada con unas dimensiones específicas pero el triángulo que la va a contener no ocupa la totalidad de la misma. Al mismo tiempo de delimitar los límites se marcaran aquellas casillas dentro de la cuadrícula que estén fuera de los límites puesto que no serán de utilidad quedando como resultado algo así.

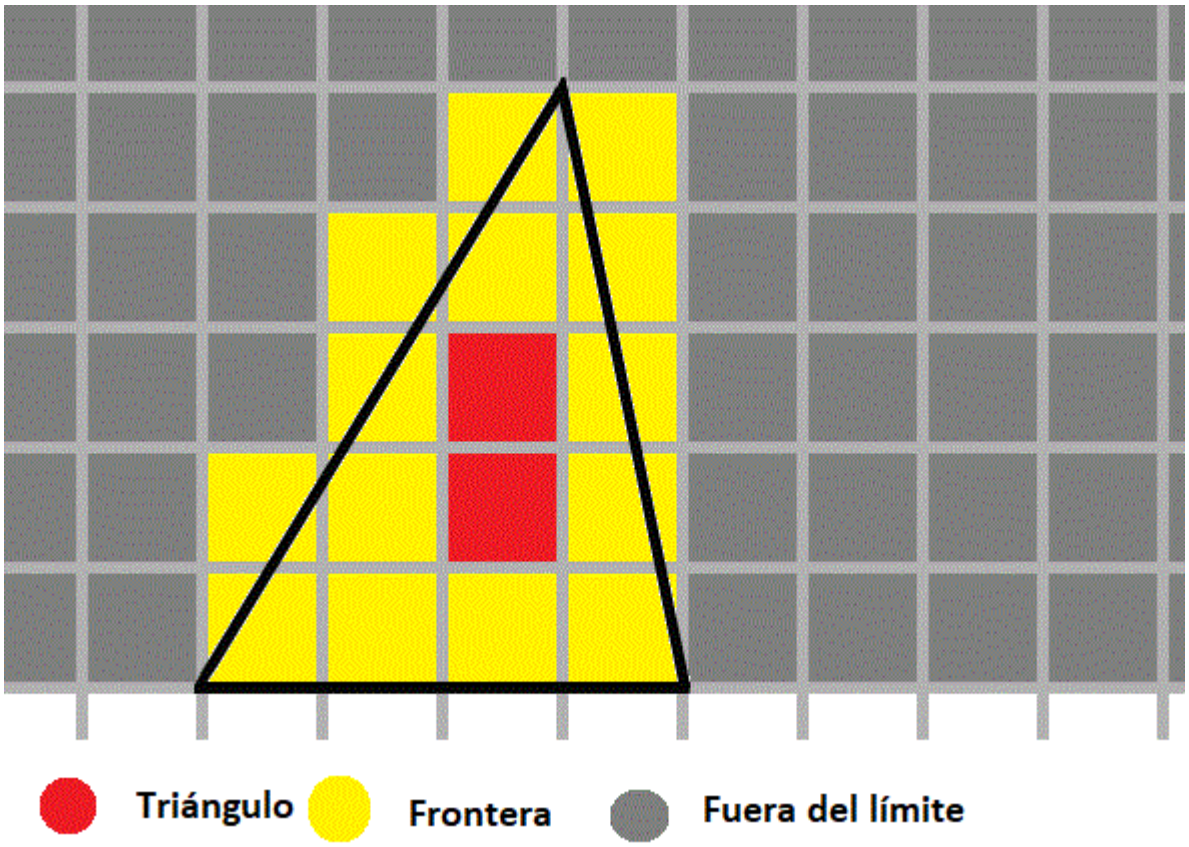


Ilustración 5.2 Demarcación del triángulo en cuadrículas

Marcaremos las casillas de 3 posibles maneras, puede estar completamente dentro del triángulo que con respecto a la imagen sería el color rojo, estas casillas se aseguran que todos sus vecinos sean válidos. Por otra parte y siguiendo la imagen el color amarillo indica aquellas casillas en las que alguno de sus vecinos tiene alguna casilla no válida que ya se sale del triángulo, para marcar la frontera se trabaja ahora en 2D. Al haber ya pasado los tres puntos relativos al triángulo a cada una de las casillas correspondientes tenemos 3 coordenadas 2D que forman 3 posibles líneas, para cada una de esas líneas se comprobara si hay intersección con cada casilla. Cada casilla al final no es mas que un cuadrado con el que se puede comprobar si hay intersección o no, por la naturaleza de la primera arista que conforma la base de la cuadrícula no es necesario hacer intersección con el resto de casillas puesto que es evidente por cuales casillas se cruza.

5.4. Conexiones entre cuadrículas

Una vez cada triángulo tiene asociado su respectiva cuadrícula (Grid) el siguiente paso será asociar cada triángulo con sus vecinos, cada triángulo tendrá 3 vecinos uno por cada arista. El primer paso será encontrarlos, para posteriormente unir esas dos aristas; unirlas significa recorrer el borde de la arista correspondiente que previamente se ha calculado para añadir a cada una de las casillas del borde un puntero con la otra casilla de la otra cuadrícula. Para acelerar la búsqueda se hará uso de una estructura intermedia que contendrá por cada punto los triángulos en los que está contenido, esta estructura será un mapa desordenado que tendrá como clave el valor del punto 3D y como valor un vector de enteros con los índices a los triángulos que pertenece. Con el apoyo de esta estructura para cada arista de un triángulo se comprobaran ambos puntos en dicha estructura, se obtendrán dos vectores con los triángulos, posteriormente se hará una intersección en esos vectores para ver que triángulos comparten y poder asignar la arista de ese triángulo al actual. En total por triángulo se comprobaran todas las combinaciones con los 3 puntos del mismo para poder asignar cada arista de forma correcta.

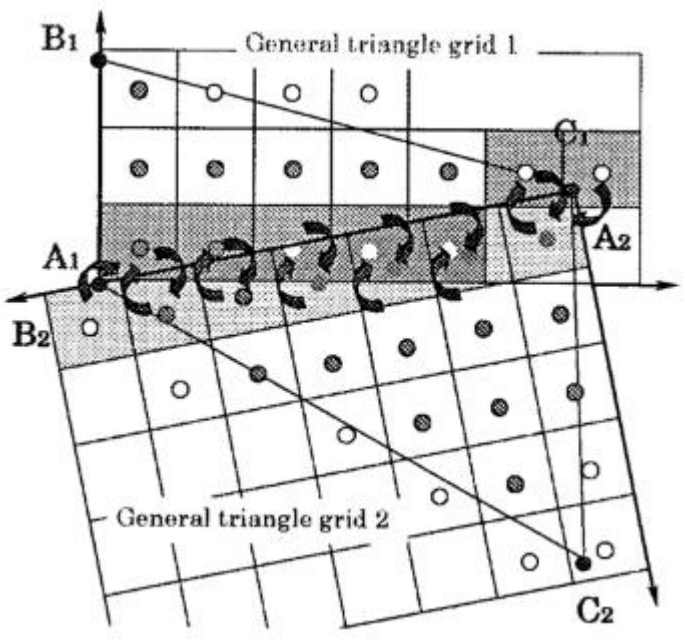


Ilustración 5.3 Formación de conexiones entre cuadrículas/triángulos

Tras haber localizado para determinada arista su respectivo vecino se procede a diferenciar a que caso corresponde para tratarla de diferente manera, se pueden diferenciar tres casos para cada arista del triángulo.

Independientemente de que arista escojamos siempre tenemos 3 casos a diferenciar:

Unión con la primera arista del otro triángulo, esta arista correspondería a la base de la propia cuadrícula y a su vez, la mas larga del triángulo.

Unión con la segunda arista del otro triángulo, esta arista corresponde con la arista derecha del triángulo.

Unión con la tercera arista del otro triángulo, esta arista corresponde con la arista izquierda que vuelve al inicio del triángulo.

Es importante denotar que dentro de cada posibilidad se encuentra la opción de que la arista esté invertida ya que al final se trabaja con segmentos con dirección y en caso de estar invertidos el procesamiento es ligeramente distinto como se verá más adelante.

Una vez identificado que triángulos y aristas vamos a unir se procede a la unión, para unirlos simplemente se recorre ambas aristas asegurándose de que la arista objetivo se recorra completamente, esto es, para cada unión de dos aristas siempre hay una de ellas la cual es la que se va a cambiar mientras que la otra no se modifica. Mientras se recorren para cada posición se une la casilla de la arista a pegar con la arista objetivo resultando al final en la arista ya comunicada con su vecina. En la unión como se puede apreciar en el diseño de clases existe tres variables fruto de un struct en las que se guarda la posición 2D de la frontera y el identificador de la cuadrícula vecina para tener toda la información necesaria, además existen tres variables debido a que puede haber tres tipos de frontera (una por cada arista, en ciertas casillas puede coincidir las fronteras por lo que es necesario tenerlas diferenciadas). Para recorrer las aristas se hace uso de la delimitación de la frontera echa anteriormente para pasar por las casillas frontera en una dirección dada.

5.5. Triangular punto en cuadrícula

Otra de las funcionalidades necesarias para este diseño es la de calcular puntos intermedios en la propia cuadrícula, cada cuadrícula esta compuesta por 3 puntos 3D pertenecientes al triángulo pero para calcular los puntos dentro del triángulo es necesario hacer un procesamiento añadido.

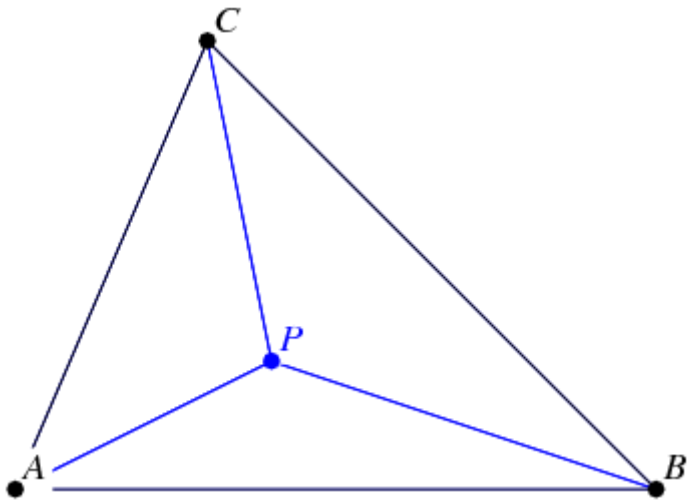


Ilustración 5.4 Ejemplo de triangulación

Con ese fin se han propuesto tres métodos para este objetivo. De estos métodos 2 precisan de unos datos adicionales para poder triangular de forma correcta, se dispondrá de 3 distancias, una a cada vértice del triángulo. Para calcular esta distancia será necesario hacer uso de la variable previamente calculada de la distancia 3D por casilla, para calcular cada distancia al punto se formara un triángulo rectángulo con el nuevo punto en el que la hipotenusa será la distancia. Para calcular la hipotenusa será necesario usar el teorema de Pitágoras.

$$a^2 = b^2 + c^2$$

Siendo “a” la hipotenusa y “b”, “c” los catetos del triángulo rectángulo que se forma.

5.5.1. Mediante sistema de ecuaciones con la fórmula de la distancia

Utilizando la fórmula de la distancia es posible crear un sistema de ecuaciones para calcular el punto a triangular **[Lewis, 2016]**.

$$d(p1, p2) = \sqrt{(x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2}$$

Para cada ecuación tenemos 3 incógnitas que serían las 3 coordenadas del punto que queremos triangular, dado que tenemos 3 distancias relativas a cada punto del triángulo eso proporciona tres ecuaciones de la siguiente manera.

$$d(p1, p) = \sqrt{(x - x1)^2 + (y - y1)^2 + (z - z1)^2}$$

$$d(p2, p) = \sqrt{(x - x2)^2 + (y - y2)^2 + (z - z2)^2}$$

$$d(p3, p) = \sqrt{(x - x3)^2 + (y - y3)^2 + (z - z3)^2}$$

Siendo “p” el punto que queremos calcular y por tanto x, y, z las coordenadas de dicho punto desconocidas, el resto de variables las conocemos y por tanto es posible resolver dicho sistema de ecuaciones resultante para triangular el punto.

No obstante este método resulto no ser válido puesto que los sistemas de ecuaciones resultantes no se podían resolver o daban soluciones muy alejadas de las deseadas. La explicación a este fallo se encuentra en el paso de coordenadas 3D a la cuadrícula y en el cálculo de la variable de la distancia por casilla, al no ser milimétricamente exacta la ecuación resultante era imprecisa.

5.5.2. Mediante porcentajes inversos

Uno de los métodos más intuitivos es crear un peso para cada vértice dependiendo de la distancia para después aplicar ese peso como un porcentaje para calcular el punto deseado **[Lewis, 2016]**. El peso de cada vértice se calcularía de la siguiente manera:

$$PesoVertice1 = \frac{1}{distanciaAVertice1}$$

$$PesoVertice2 = \frac{1}{distanciaAVertice2}$$

$$PesoVertice3 = \frac{1}{distanciaAVertice3}$$

Tras calcular los pesos de cada vértice el paso final sería aplicarlo en base a los puntos del triángulo para obtener el punto nuevo, para aplicarlo seguimos la siguiente fórmula:

$$puntoInterpolado = \frac{pesoVertice1 * vertice1 + pesoVertice2 * vertice2 + pesoVertice3 * vertice3}{pesoVertice1 + pesoVertice2 + pesoVertice3}$$

Con este método los resultados fueron generalmente buenos pero tras representar el modelo 3D con todos los puntos pertenecientes a todas las cuadrículas se observó un problema al tratar de aumentar el detalle del autómata.

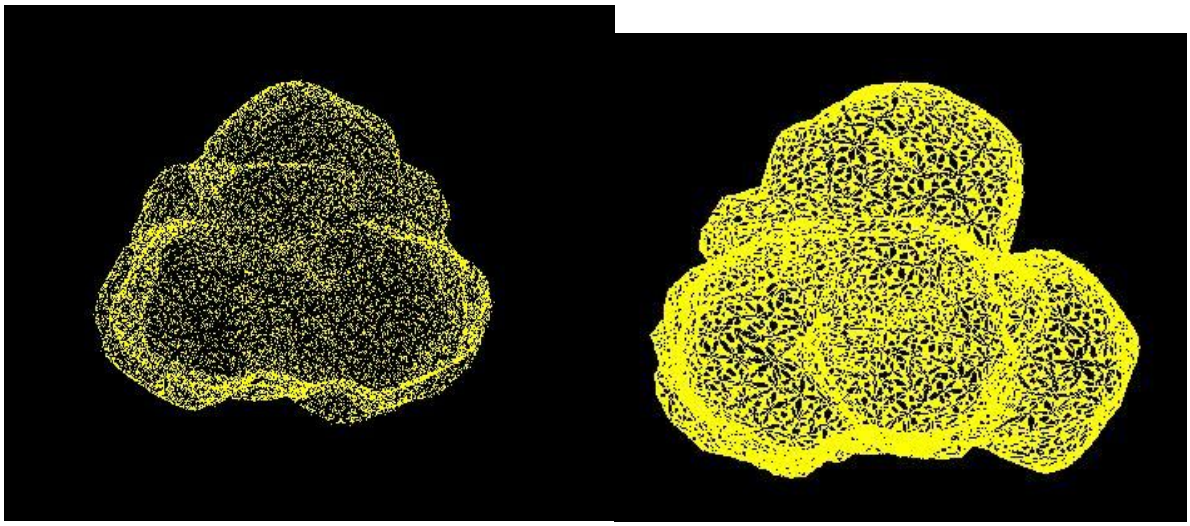


Ilustración Izquierda 5.5 Autómata celular para modelo con menos precision (version interpolación con porcentajes inversos)

Ilustración Derecha 5.6 Autómata celular para modelo con mas precision (version interpolación con porcentajes inversos)

Como se comenta anteriormente en el diseño, a la hora de cargar el autómata celular le pasamos una serie de parámetros para el nivel de detalle deseado (mínimo y máximo número de casillas por cuadrícula), en la primera imagen se ha usado 3-7 como número mínimo y máximo de casillas mientras que en la segunda imagen se ha utilizado 10-15. Como se puede observar en la segunda imagen se han formado huecos entre los triángulos dejando los puntos de los extremos de los triángulos vacíos. Este fenómeno ocurre debido al cálculo de porcentaje inversos que no controla de manera adecuada cuando el triángulo tiene formas mas enrevesadas o se aleja mas de la forma de un triángulo equilátero. Generalmente este método es válido para gran parte de los casos pero es muy dependiente del modelo y como se verá a continuación existe otro método mas efectivo que soluciona este problema.

5.5.3. Mediante coordenadas baricéntricas

El último método propuesto es a su vez el que mejor resultados ha proporcionado además de ser relativamente el más simple puesto que no necesita de las distancias anteriormente mencionadas **[Lewis, 2016]**. Este método consiste en utilizar las coordenadas baricéntricas de un triángulo para obtener unos pesos (porcentajes al final) que se puedan aplicar para interpolar el punto deseado. Pero ¿Qué son las coordenadas baricéntricas? Consisten en representar un triángulo del modo que sus vértices serían (0,0,1), (0,1,0) y (1,0,0), estas coordenadas buscan parametrizar los puntos contenidos en el triángulo para después extrapolarlos al uso requerido.

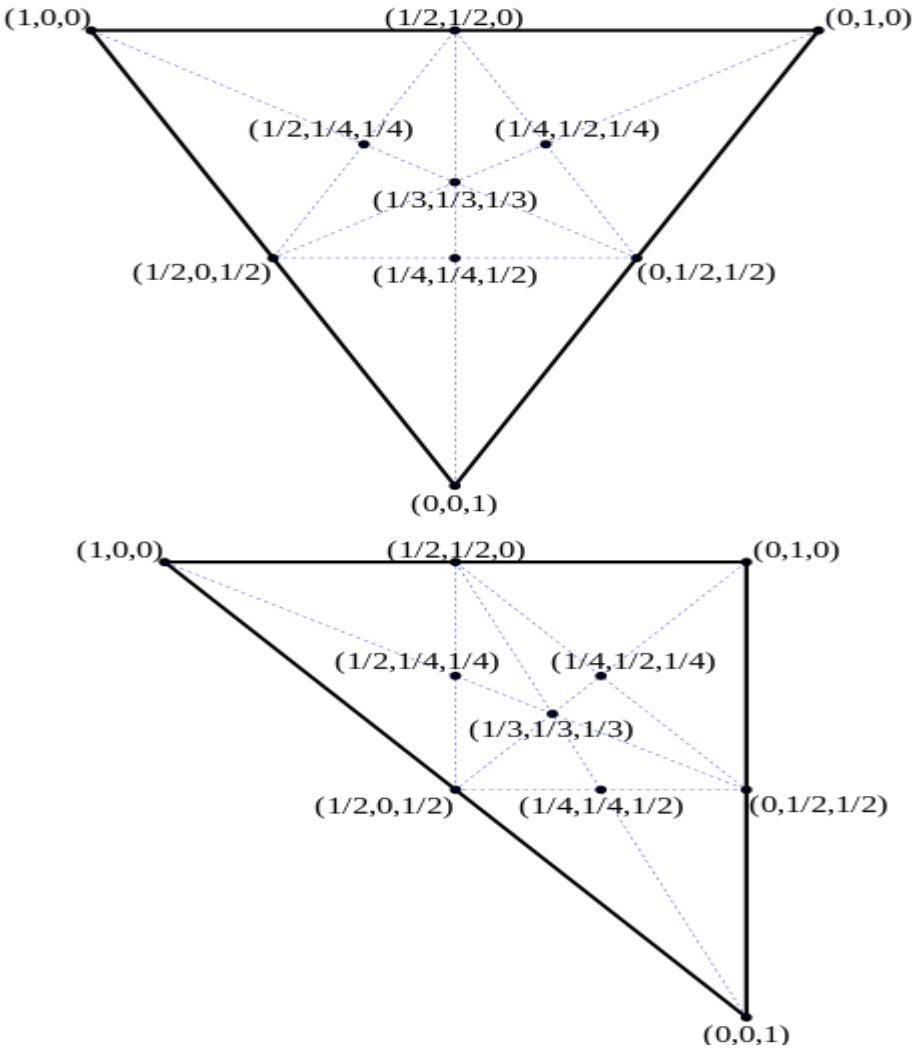


Ilustración 5.7 Coordenadas baricéntricas

Esta parametrización es la que utilizaremos para interpolar un punto determinado dentro del propio triángulo, a diferencia del resto de métodos en este trabajamos en 2D todo el tiempo con las coordenadas (x,y) de la cuadrícula. Utilizando las coordenadas dentro del triángulo que queremos interpolar además de saber las coordenadas 2D donde los tres vértices del triángulo se encuentran podemos calcular el peso de cada vértice en las coordenadas pasadas como parámetro.

$$PesoVertice1 = \frac{(Yv2 - Yv3)(Px - Xv3) + (Xv3 - Xv2)(Py - Yv3)}{(Yv2 - Yv3)(Xv1 - Xv3) + (Xv3 - Xv2)(Yv1 - Yv3)}$$

$$PesoVertice2 = \frac{(Yv3 - Yv1)(Px - Xv3) + (Xv1 - Xv3)(Py - Yv3)}{(Yv2 - Yv3)(Xv1 - Xv3) + (Xv3 - Xv2)(Yv1 - Yv3)}$$

$$PesoVertice3 = 1 - PesoVertice1 - PesoVertice2$$

P(Px,Py) sería el punto a calcular en coordenadas de cuadrícula, mientras V1(Xv1,Yv1), V2(Xv2,Yv2), V3(Xv3,Yv3) corresponderían a las coordenadas en cuadrícula de los tres vértices del triángulo. Aplicando los pesos calculados a cada uno de los vértices y sumando los vértices obtendríamos el punto interpolado.

$$PuntoInterpolado = Vertice1 * PesoVertice1 + Vertice2 * PesoVertice2 + Vertice3 * PesoVertice3$$

Tras probar los resultados con los mismos valores del autómata que para el método anterior, se observa que el problema ya ha desaparecido y los puntos están mucho mejor repartidos alrededor del modelo, seguir añadiendo nivel de detalle significa representar el modelo a base de puntos sin un hueco libre.

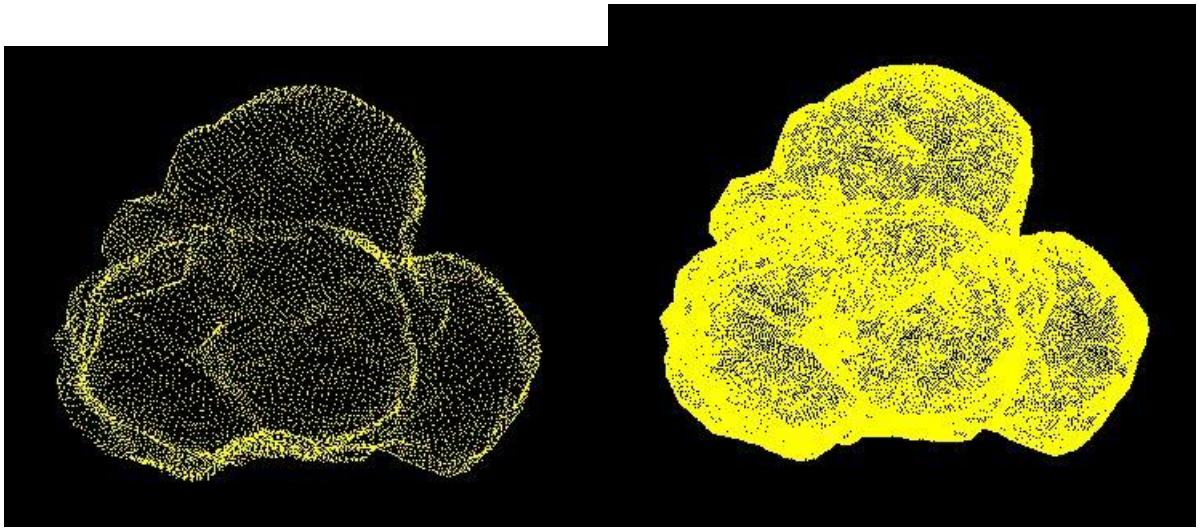


Ilustración Izquierda 5.8 Autómata celular para modelo con menos precision (version interpolación por coordenadas baricéntricas)

Ilustración Derecha 5.9 Autómata celular para modelo con más precision (version interpolación por coordenadas baricéntricas)

Como última mención, alguno de los pesos calculados anteriormente pueden ser negativos lo que significaría que el punto esta fuera del triángulo, si bien es cierto que no debería pasar puesto que delimitamos la frontera la realidad es que puede haber alguna casilla que se considere frontera estando técnicamente casi fuera del triángulo por lo tanto es necesario controlar este suceso.

5.6. Componentes definidas del diseño grafo

Las componentes del autómata celular, previamente explicadas, ahora son definidas en profundidad para este diseño en concreto de la misma manera que con el diseño Grafo.

Espacio: Para este diseño el espacio será 2D puesto que se recrea una cuadrícula en forma de matriz.

Estados: Rota o intacta, esta célula tendrá como característica adicional la resistencia de la misma.

Vecindad: La vecindad vendrá dado por 8 vecinos al igual que en una matriz, no obstante en ciertos casos donde se encuentre en el borde el número de vecinos se mantendrá pero con repeticiones.

Algoritmo de transición: Para el algoritmo de transición será utilizado el algoritmo en base a la dirección descrito en el apartado de algoritmos.

Frontera: La frontera del autómata será circular debido a que cada célula se comunica con sus vecinos dando la vuelta al modelo.

5.7. Conclusiones

Este modelo soluciona el problema de la versión del grafo ya que mantiene una distribución justa (además de modificable) en la que se distribuye el autómata y permite un nivel de detalle casi infinito, no obstante, surge un inconveniente que puede afectar al algoritmo de transición. El problema es referente a las conexiones formadas con cada triángulo, dependiendo del algoritmo de transición puede ser un obstáculo ya que al cambiar de cuadrícula puede cambiar la orientación de la cuadrícula, en el caso del algoritmo de transición por dirección al intentar utilizar una dirección fija y cambiar de cuadrícula al cambiar la orientación no se mantiene la misma dirección real.

6. Autómata celular versión cuadrícula completa (Full Grid)

Debido al problema del algoritmo de transición en la versión cuadrícula del autómata celular se propuso una versión mejorada de dicha versión que se basa en una cuadrícula completa dividiendo el modelo entero en una sola cuadrícula en vez de dividir cada triángulo.

Sin embargo esta tarea no es trivial así que será necesario el apoyo de un elemento, las coordenadas de textura del modelo. Una textura es formada entonces como capa abstracta para representar la superficie del modelo **[Stephane y Norishige, 2001]**. Estas coordenadas, al definir como es aplicada una imagen en un modelo son perfectas para definir una cuadrícula única, no obstante surge un requerimiento para este autómata, no solo debe disponer de coordenadas de textura sino que además esas coordenadas deben mantener cierta coherencia. Se observó que ciertos modelos que contenían coordenadas de textura las contenía sin ninguna coherencia espacial lo que llevaba a resultados totalmente absurdos. Para calculas las coordenadas de textura con coherencia programas como blender tienen maneras de calcular dichas coordenadas con un modelo seleccionado.

6.1. Requisitos para la versión cuadrícula completa (Full Grid)

Para esta versión del autómata celular los requisitos vuelven a cambiar ligeramente acorde con las necesidades del proyecto pero manteniendo la base de los mismos.

- **Representar una capa 2D que envuelva al modelo con detalle personalizable y manteniendo una única cuadrícula,** partiendo de la versión anterior ahora el requisito añadido en esta parte es asegurarse de que se forma una sola cuadrícula en vez de una por cada triángulo. Con este fin se espera resolver el problema de la orientación comentando en las conclusiones.
- **Formar conexiones entre los elementos de la capa 2D,** de la misma manera que en la anterior versión es necesario formar una vecindad en la representación elegida.
- **Apoyarse de algún elemento para poder formar esta cuadrícula única.**

6.2. Diseño de clases

El diseño de clases, de la misma manera que el anterior diseño, es bastante simple ya que consiste en la clase encargada de encapsular la cuadrícula (CelullarAutomatonV3) y cada cuadrado del mismo (Box). Se puede apreciar además por el diseño de clases que la complejidad del mismo también es inferior al diseño anterior, en gran parte esto es debido a las coordenadas de textura que reducen la complejidad del autómata celular.

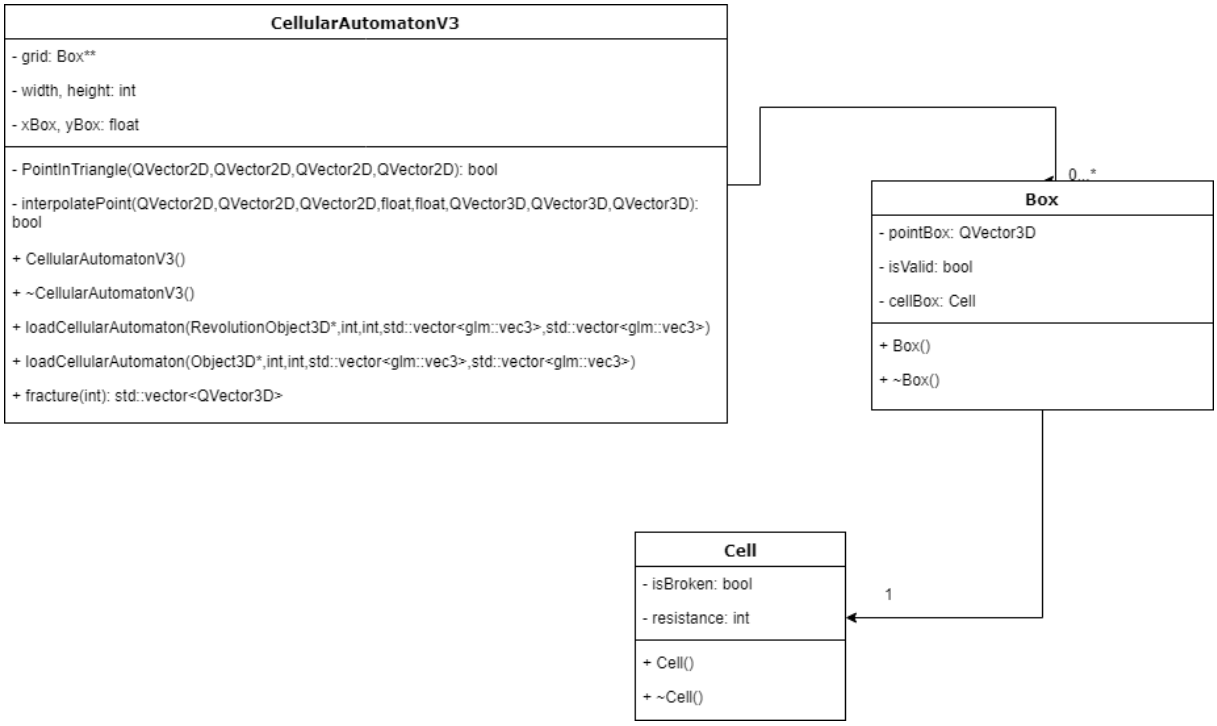


Ilustración 6.1 Diseño UML de clases para la version cuadrícula completa

De la misma manera que el UML del diseño anterior se omiten las funciones de sacar y cambiar variables de clase por ser redundantes.

6.3. Componentes definidas del grafo

Las componentes del autómata celular, previamente explicadas, ahora son definidas en profundidad para este diseño en concreto de la misma manera que con el resto. Este diseño es muy parecido al anterior puesto que se basa en la misma idea pero con distinta implementación.

Espacio: Para este diseño el espacio será 2D puesto que se recrea una cuadrícula en forma de matriz.

Estados: Rota o intacta, esta célula tendrá como característica adicional la resistencia de la misma.

Vecindad: La vecindad vendrá dado por 8 vecinos al igual que en una matriz, incluso en casos de “frontera” cuando se encuentre en el limite del ancho/alto se adaptará a una nueva posición.

Algoritmo de transición: Para el algoritmo de transición será utilizado el algoritmo en base a la dirección descrito en el apartado de algoritmos.

Frontera: En este caso la frontera del autómata viene dada por el ancho y alto de la cuadrícula, no obstante, para hacerla circular cada vez que llegue a la frontera las coordenadas se ajustan para que puedan seguir el camino.

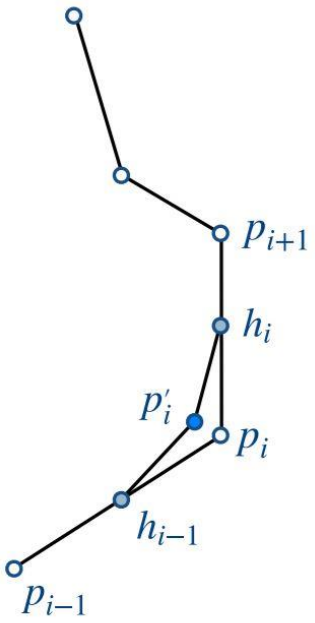
6.4. Objeto de revolución

Hasta cierto momento se han estado utilizando modelos sacados de archivos “.obj” pero para poder testear este diseño es necesario un modelo con coordenadas de textura que mantenga coherencia espacial. Con ese fin entra el objeto revolucionado, este modelo se crea a partir de un perfil 2D que se revoluciona alrededor del eje Y creando los vértices, triángulos y coordenadas de textura (con coherencia espacial), siendo esta división dinámica para cada modelo a gusto del usuario. Con la ayuda de este tipo de modelo este diseño podrá ser testeado a todo su potencial.

La creación del objeto revolucionado consta de dos pasos, primero con el perfil dado se subdivide (es opcional pero recomendable para dar más suavidad al contorno) y posteriormente la revolución alrededor del eje Y con toda la creación de la geometría.

6.4.1. Subdivisión perfil

Esta subdivisión consiste en encontrar para cada par de puntos los puntos intermedios e interpolaros entre sí para obtener un punto más ajustado a la curva como indica la imagen.



$$h_i = \frac{p_i + p_{i+1}}{2}$$

$$h_{i-1} = \frac{p_{i-1} + p_i}{2}$$

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

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

Ilustración 6.2 Subdivisión de un perfil 2D

Posteriormente, esos puntos interpolados se añaden como nuevos al perfil aumentando el número de puntos en el perfil en el doble de puntos menos 1. Aumentando el número de iteraciones a este proceso hará las curvas más suaves y al mismo tiempo aumentara el número de puntos lo que supondrá un número mayor de geometría a la hora de revolucionar.

6.4.2. Revolución

Tras haber subdividido el perfil (o no) se procede a revolucionar ese perfil alrededor del eje Y, esas revoluciones vendrán dadas por un parámetro que indicara el número de particiones alrededor del eje Y.

Para revolucionar un punto dado del perfil alrededor del eje Y utilizamos la siguiente fórmula:

$$puntoRevolucionado.x = \cos(\text{angleRadian}) * \text{puntoPerfil.x}$$

$$puntoRevolucionado.y = \text{puntoPerfil.y}$$

$$puntoRevolucionado.z = -\sin(\text{angleRadian}) * \text{puntoPerfil.x}$$

Para cada punto del perfil se utilizará esta fórmula “n” veces, siendo n el número de revoluciones dadas. Además en cada una de esas iteraciones “angleRadian” debe incrementarse de manera que complete el rango de acorde a las revoluciones, el rango además está en radianes para completar la vuelta entera alrededor del eje Y. La componente “y” se mantiene en todas las iteraciones puesto que la altura no cambia al revolucionar los puntos.

Para crear los triángulos es necesario utilizar los índices de los vértices de manera que formen triángulos en la dirección opuesto al reloj para que sean válidos. Se recorrerán todos los puntos, para cada punto se seleccionará el actual, el siguiente y el de por encima formando el primer triángulo. Posteriormente con el punto de arriba, el siguiente y el de abajo se formará el segundo triángulo como se muestra en la imagen.

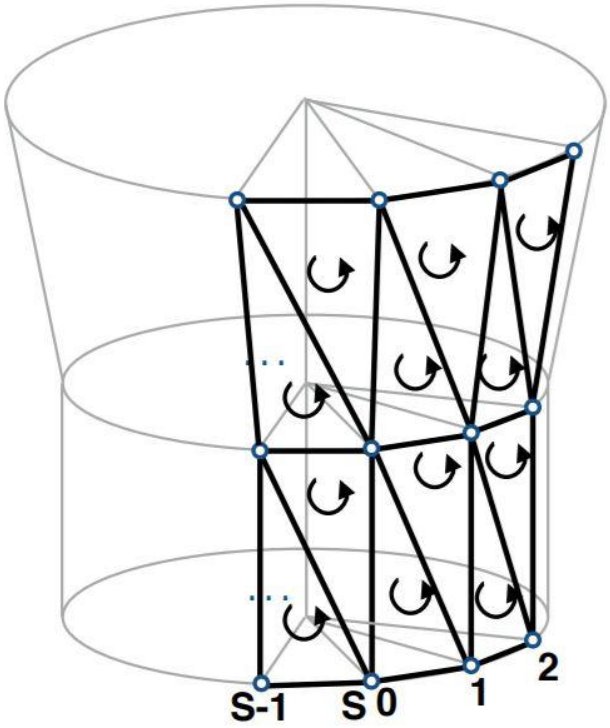


Ilustración 6.3 Formación de triángulos en la revolución del perfil

Las normales se calculan en base a los vértices revolucionados, para cada vértice se procede a realizar una serie de pasos como se muestra en la imagen.

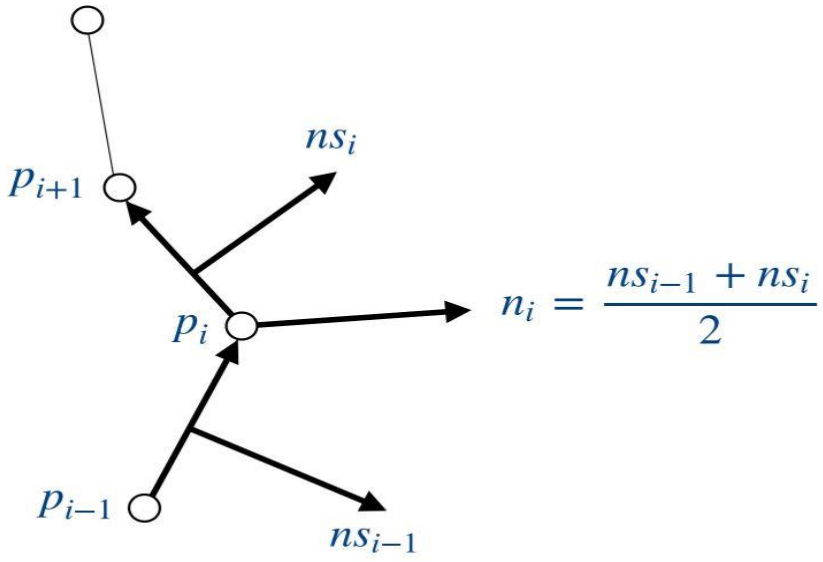


Ilustración 6.4 Generación de normales en los puntos revolucionados

Para cada vértice se calculan el vector que forma con el punto de arriba y con el punto de abajo, con esos dos vectores se calcula el vector perpendicular. Finalmente con esos dos vectores se calcula el vector intermedio resultando en el vector normal del vértice elegido. Este proceso se completa para cada vértice de la geometría, siendo un caso aparte los que están justo en el límite y no tienen vector de arriba o vector de abajo en cuyo caso la normal perpendicular del vector resultante se convierte en la normal.

Finalmente las coordenadas de textura se calculan en base a la altura del modelo para repartir el peso de cada vértice en la componente “u” en el rango [0,1], para la componente “v” se calculaba en base al número de revoluciones en el mismo rango [0,1].

6.5. Creación del autómata

La creación de este autómata es algo más sencilla que la de la versión anterior, no obstante, en términos de tiempos es mucho más cara que el anterior. La cuadrícula consiste en una matriz dinámica de objetos “Box” que se inicializa con un ancho y alto, esta matriz utilizara las coordenadas de textura para unirla al modelo real. Para ello existirán dos variables, “xBox” e “yBox” que indicaran la distancia en cada cuadrado en el eje X y en el eje Y, esta distancia se calculará en base al ancho (1/width) y el alto (1/height). De esta manera para cada cuadrado en la matriz existe una componente “x” e “y” en el rango [0,1] con el que se puede hacer una conexión en el modelo 3D.

Una vez creada la matriz se rellena cada posición de la matriz con un punto del modelo 3D, para hacer esta tarea se hace uso de las coordenadas de textura del objeto. Para cada posición de la matriz se calcula una “x” e “y” con las variables “xBox” e “yBox”, con esa posición se comprueba en que triángulo del modelo esta contenida para poder interpolarla creando un nuevo punto 3D y asociarle a la posición de la matriz. El triángulo se forma con las coordenadas de textura resultando en un triángulo 2D, posteriormente calcular un punto en cada uno de los triángulos del modelo se hace con una sencilla función. Adicionalmente también hay que interpolar el punto de la misma manera que en el diseño anterior en la sección 5.4.3 utilizando coordenadas baricéntricas.

Los resultados muestran una buena distribución de los puntos y un nivel de detalle tan alto como el diseño anterior.

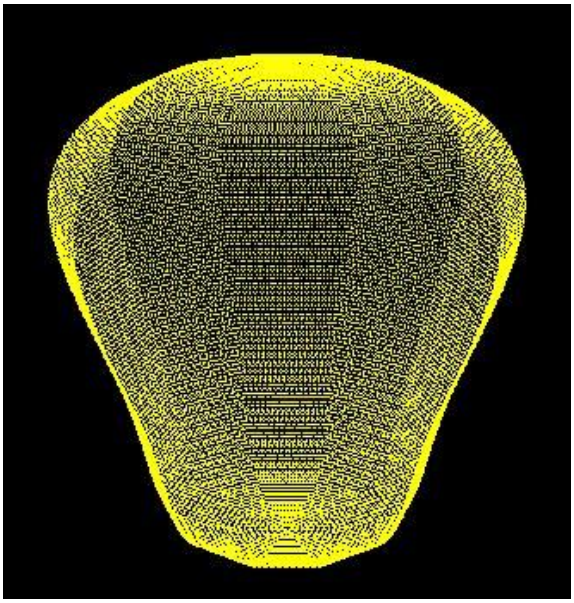


Ilustración 6.5 Creación del automata celular para la versión cuadrícula completa

Este objeto de revolución se basa en un perfil de 6 puntos que ha sido subdividido 3 veces para crear la suavidad que se puede apreciar en la imagen. Posteriormente se ha revolucionado 10 veces alrededor del eje Y, se puede apreciar las revoluciones como los cortes en los grupos de puntos. Una vez el objeto revolucionado se ha creado el autómata celular con una matriz de 300x150 dando un total de 45000 puntos, como se puede observar los puntos se distribuyen a lo largo del modelo formando en casi su totalidad el modelo en sí.

6.5.1. Función punto en triángulo

Una de las funciones necesarias para la creación del autómata es calcular si un punto 2D esta contenido en un triángulo 2D. Para ello se comprobará si el punto a calcular se encuentra a la izquierda de cada arista del triángulo, si se cumple esas tres comprobaciones el punto estará contenido dentro del triángulo. Un punto a remarcar de esta función es que el triángulo debe estar en dirección opuesta a las agujas del reloj para que la orientación sea correcta con respecto al punto, esto no es problema dado que todos los triángulos del modelo están siempre con esa orientación.

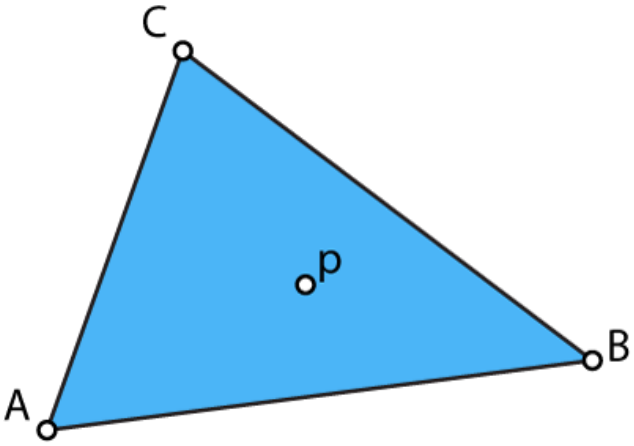


Ilustración 6.6 Punto en triángulo

6.6. Conclusiones

Este diseño al evitar unir fronteras entre varias cuadrículas mantiene la orientación y el algoritmo de dirección puede utilizarse en todo su potencial. Además el diseño lo hace más sencillo que la versión de Cuadrícula por cada triángulo por lo que su implementación es mas asequible. En el aspecto negativo el algoritmo es significativamente más costoso que cualquiera de los diseños anteriores (para información más detallada ver la sección …….) lo que hace que aumentar el detalle del autómata sea considerablemente más complejo con respecto a los demás. Otro de los inconvenientes es la necesidad de las coordenadas de textura con coherencia espacial del modelo, lo cuál restringe el número de modelos en los que se puede utilizar.

7. Algoritmos empleados para la fractura

Principalmente han sido utilizados dos algoritmos para generar una fractura en los modelos, un algoritmo aleatorio con elementos añadidos para asegurar la continuidad y un algoritmo que funciona en base a una dirección para generar la fractura. El algoritmo aleatorio solo fue utilizado en el primer diseño del autómata celular para hacer una pequeña prueba, en los dos diseños posteriores el algoritmo de dirección fue el elegido debido a sus mejores resultados.

7.1. Algoritmo aleatorio con elementos añadidos

Este primer algoritmo fue creado como prueba exclusivamente para el diseño del autómata celular grafo aunque puede ser fácilmente extensible a los demás diseños.

Como algoritmo inicial primero se probó a crear la fractura de forma aleatoria, esto sería eligiendo un nodo al azar y de la misma manera eligiendo un vecino al azar para crear la fractura. Por cada nodo que se elija se establecería la célula como rota de manera que una vez la fractura llegue a otro nodo con una célula rota el algoritmo terminaría, dejando un rastro de nodos por los cuales ha pasado. En la práctica este algoritmo deja mucho que desear dando resultados bastante pobres por lo que se le han añadido un par de modificaciones para asegurar que la fractura llega a cierto punto. Estos elementos añadidos son:

Un vector de posiciones prohibidas para asegurar que la fractura no pueda volver por donde ha venido hasta cierta longitud de la misma, en este vector irán entrando todas las posiciones por donde pase la fractura y no irán saliendo hasta que pasen ciertas iteraciones del algoritmo. Este vector de prohibidos hace que el algoritmo derive a un caso particular de error, puede llegar el momento en el que se encuentre en una posición y todos los vecinos estén prohibidos no teniendo opción de continuar con la fractura, la solución está ligada al siguiente elemento.

Backtracking (Vuelta atrás), debido al problema comentado anteriormente este algoritmo aleatorio tiene también la posibilidad de hacer “backtracking” si un nodo se encuentra en una posición donde todos sus vecinos están prohibidos, si esto pasa volverá a la posición anterior conocida y seguirá el algoritmo por esa posición sin posibilidad de volver a escoger el nodo “malo”. No obstante el backtracking tiene limitaciones, solo se podrá utilizar una vez de forma seguida, esto quiere decir, que si al volver al nodo anterior y en caso de que este también tenga todos sus vecinos prohibidos el algoritmo terminará ahí marcando la fractura como incompleta.

Los pasos relativos al algoritmo son sencillos puesto que se basa en la aleatoriedad con ciertos matices:

1 - Aleatoriamente se escogerá uno de los nodos que será el punto de inicio de la fractura, adicionalmente en la función también se podrá pasar como argumento el punto inicial siempre que pertenezca al modelo.

2 - Una vez escogido el punto se iniciará el bucle y dicho punto se añadirá a un vector para ir recogiendo el rastro.

3 - Se comprobará si la célula ha sido ya quebrada anteriormente.

a - Si ha sido quebrada significa el final del algoritmo.

b - Si no ha sido quebrada la fractura seguirá expandiéndose.

4. Se establece la célula como rota.

5. Se consideran todas las posibles opciones (los vecinos) para meterlas en un vector aparte, a su vez se comprueba que ninguno esté en la estructura de baneados para no incluirla.

6. Se comprueba si hay alguna opción tras la anterior comprobación.

a - Si no hay ninguna opción se procede a hacer backtracking (Punto 7).

b - Si se encuentra alguna opción se sigue con el algoritmo (Punto 8).

7. Al no tener vecino al que ir, volvemos hacia atrás borrando el nodo actual del vector que contiene la línea de la fractura y cogemos sus posibles vecinos siempre que no estén prohibidos.

a. Si vuelve a no tener ningún vecino disponible porque están todos prohibidos el algoritmo acabaría.

b. Si tiene algún vecino posible el algoritmo continúa.

8. Teniendo ya todas las opciones disponibles elegimos una cogiendo aleatoriamente utilizando una semilla con la fecha y hora actual.

9. Una vez elegido el nuevo vecino se volvería al punto 3 hasta que finalice.

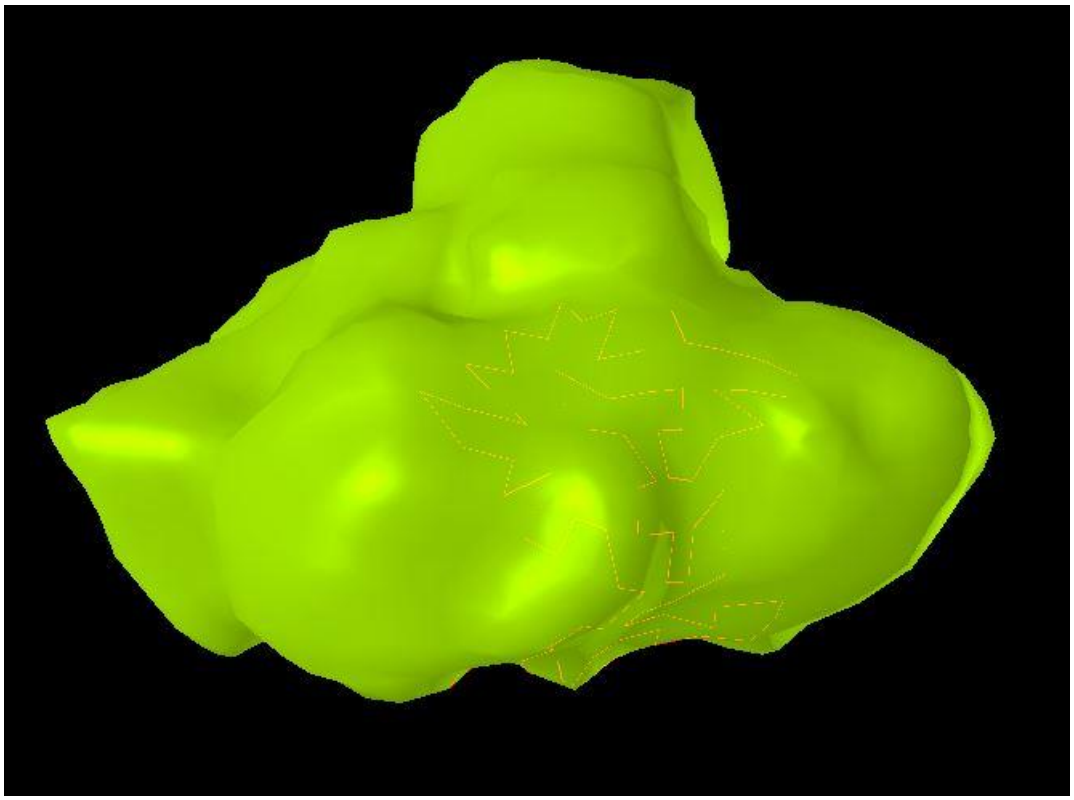


Ilustración 7.1 Fractura aleatoria con la versión grafo del autómata

Como se puede observar en la ilustración 7.1 la fractura al estar condicionada por la representación más simplista de la versión grafo da giros bruscos, adicionalmente al seguir un curso aleatorio el resultado es peor.

7.2. Algoritmo por dirección

En este algoritmo se hará uso de la estructura 2D resultante de cargar el autómata para las versiones cuadrícula y cuadrícula completa. Adicionalmente se introducirá el concepto de fuerza de fractura en el que para llegar a romper una célula se requerirá de una fuerza necesaria, de otro modo, la fractura podría resultar incompleta. Para la dirección se utilizará un vector 2D que indicará la dirección que se intentará seguir en todo momento, en adición a la operación del producto escalar que nos permitirá ser capaz de continuar la fractura.

Este algoritmo intenta ser una aproximación realista de una fractura real, generalmente las fracturas una vez creadas tienen una dirección predeterminada que a no ser que pare (fractura incompleta) se sigue hasta que se finalice. Esta finalización

depende de si se encuentra otra parte rota, se puede dar también el caso de que la fractura sea masiva lo que haría que el hueso entero se fraccione en varias partes, no obstante, este caso no esta cubierto por el algoritmo.

Como se ha comentado anteriormente, para ser capaz de seguir una dirección se hará uso del producto escalar. Este tipo de operación permite comparar dos vectores dando un resultado flotante, dependiendo de ese resultado es posible saber como “de cerca” están situados por lo tanto para utilizarlo en el algoritmo basta con hacer esta operación por cada vecino (creando su respectivo vector restando el punto del vecino con el punto actual donde se encuentra el algoritmo) y buscando siempre el mayor resultado a esta operación lo cual significaría que es la dirección correcta a seguir. Es importante también denotar que los vectores deben estar normalizados para un rango de respuestas efectivo de [-1,1], se nos puede presentar cuatro tipos de casos:

- El resultado del producto escalar es 1, en este caso ambos vectores son iguales por lo que esta es la dirección que hay que seguir.
- El resultado del producto escalar es mayor que 0, en este caso el ángulo entre los dos vectores no supera los 180º por lo que son direcciones plausibles.
- El resultado del producto escalar es menor que 0, en este caso el ángulo entre los dos vectores supera los 180º por lo que no son direcciones plausibles.
- El resultado del producto escalar es -1, en este caso los vectores son opuestos por lo que es la peor opción posible.

Se podría argumentar que basta siempre con seguir aquella dirección en la que el producto escalar es 1 pero esto no es siempre posible puesto que puede ser una célula dura de una gran resistencia o simplemente una fuerza de fractura baja lo que cambiaría ligeramente la dirección de la fractura.

Sabiendo ya todo esto, el curso del algoritmo será sencillo.

1. Primero se elegirá una casilla en la que empezar, esta elección es aleatoria.

2. Se comprueba si la casilla actual esta rota.

    a. Si esta rota el algoritmo acaba.

b. Si no esta rota se sigue al punto 3.

3. Se recorrerán todos los vecinos comprobando el mayor producto escalar con la dirección dada como parámetro inicial. Se elegirá aquella con mayor resultado del producto escalar comprobando además que la fuerza de la fractura es suficiente para romperla.

4. Se establece la célula contenida en la casilla como rota.

5. Se guarda el punto 3D de la casilla en el vector a devolver.

6. Se cambia la casilla actual como la vecina actual y se vuelve al paso 2.

A la hora de cambiar de casilla se diferencia un caso particular, el caso frontera. Para este caso además se hace una distinción entre ambos diseños por la forma en la que lo tratan. Además debido al cambio de orientación del diseño cuadrícula el vector no podía ser 2D porque al cambiar de cuadrícula la orientación cambiaba también, por eso mismo se utilizo vectores 3D para la dirección. Por utilizar vectores 3D el resultado de la fractura era siempre incompleta, llegaba un momento en el que el producto escalar no resultaba fiable por estar en tres dimensiones.

A continuación se mostraran ambas fracturas mediante ilustraciones, a su vez la fractura será representada encima de la representación del autómata donde los puntos azules corresponden a células normales y los puntos amarillos a células duras los cuales la fractura no puede atravesar.

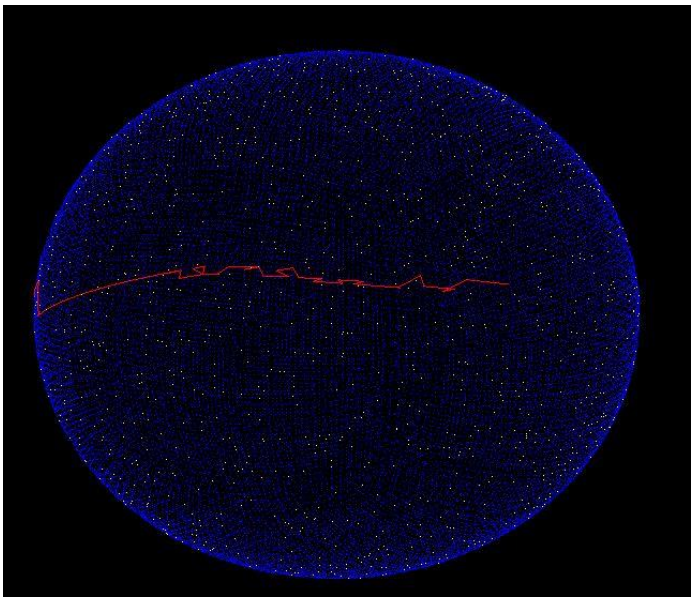


Ilustración 7.2 Fractura por dirección en versión cuadrícula (Grid)

En la versión cuadrícula y como se puede observar las fracturas por dirección ocurre el problema comentado, no llegan a ser completas debido a la dirección utilizada como referencia y la estructura del autómata celular. No obstante se puede observar como hasta el punto límite la fractura sigue su dirección intentando desviarse lo mínimo.

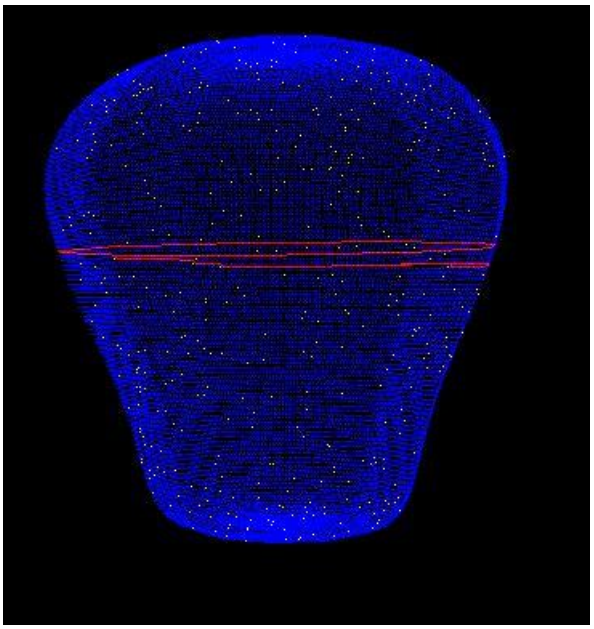


Ilustración 7.3 Fractura por dirección con la versión cuadrícula complete (Full Grid)

A diferencia del anterior ejemplo de fractura esta consigue acabar completándose la misma, en la ilustración 7.3 se observa como no solo da una vuelta si no un par debido a las células duras.

7.3. Caso frontera en diseño de cuadrícula

Para el caso del diseño cada casilla puede ser o no ser frontera dependiendo de su posición con respecto al triángulo, de ser frontera se presentan tres opciones. Puede ser frontera de cada una de las aristas, por esta razón hay que incluir siempre que exista alguna de las fronteras como vecino y comprobar la dirección de la misma manera que con los demás vecinos. Para calcular la dirección de ese vecino basta con acceder al struct de la frontera de la casilla que contiene también la dirección (para más información ver diseño de cuadrícula).

7.4. Caso frontera en diseño de cuadrícula única/total

Para el caso del diseño de cuadrícula única, la frontera consistirá en los bordes de la matriz. Se consideraran dos casos particulares, salirse por el eje X (el ancho de la matriz) y salirse por el eje Y (el alto de la matriz).

Por una parte salirse por el eje X consiste básicamente en ajustar la componente X en el rango del ancho (si es 0 pasa al ancho máximo y si es el ancho máximo pasa a 0).

Por otra parte salirse por el eje Y es ligeramente más complicado, la componente X debe ajustarse de manera que sea proporcional por el otro extremo del ancho.

8. Tiempos

Crear el autómata es una tarea con cierto peso para la CPU especialmente con modelos complejos, en esta sección se comparara entre dos modelos y un objeto de revolución con distinta cantidad de geometría para todas las versiones del autómata celular.

Modelo roca: 1386 vértices con 2768 triángulos.

Modelo buda: 26763 vértices con 53556 triángulos.

Modelo revolucionado: 8346 vértices con 16000 triángulos. Seis subdivisiones al perfil y veinte y cinco revoluciones alrededor del eje Y.

Además todos los tiempos de ejecución se harán sobre una máquina con las siguientes características:

Procesador: i7-7700HQ 2.8 GHz

Ram: 16 GB

GPU: Nvidia GTX 1070

Los datos estarán distribuidos en dos tablas (una para ejecución en debug y otra para ejecución en release) de la siguiente manera:

Debug	**Diseño grafo**	**Diseño cuadrícula**	**Diseño cuadrícula completo**
Roca	0 seg	0 seg	31 seg
Buda	0 seg	17 seg	X
Objeto revolucionado	0 seg	0 seg	129 seg

Tabla 8.1 Tabla tiempos de ejecución en distintas versiones del autómata (Debug)

Como se puede ver por los resultados el tiempo del diseño grafo es totalmente imperceptible para la CPU por su sencillez. De la misma manera los tiempos del diseño cuadrícula no son especialmente relevantes a no ser que modelos grandes como el de buda tomen lugar e incluso de esa manera no es un tiempo demasiado

preocupante. Es en el diseño cuadrícula donde se puede observar resultados mucho mas extensos por el proceso de creación del diseño.

Release	**Diseño grafo**	**Diseño cuadrícula**	**Diseño cuadrícula completo**
Roca	0 seg	0 seg	3 seg
Buda	0 seg	3 seg	X
Objeto revolucionado	0 seg	1 seg	6 seg

Tabla 8.2 Tabla tiempos de ejecución en distintas versiones del autómata (Release)

En modo “release” todo cambia bastante siendo los tiempos de CPU bastante despreciables, no obstante, esto es en parte por el modelo de la CPU. Generalmente no se puede esperar los últimos modelos en distintos ordenadores por lo que los resultandos pueden ser ligeramente distintos. La X marcada en ambas tablas muestra que el modelo no es válido para el autómata versión cuadrícula completa debido a la ausencia de texturas del mismo necesarias.

9. Presupuesto

A fin de cuentas este proyecto se considera un producto y como tal debe tener un presupuesto acorde que cubra los gastos y costes del mismo. En este proyecto en particular el presupuesto es relativamente básico debido a que no hay una necesidad de elementos de pago o externos adicionales.

Elemento	**Unidades**	**Precio por unidad**	**Total**
Horas de trabajo	300 horas aprox	19€/hora	5700€
Licencia Visual Studio 2017	1	0 €	0 €
Lápices	1	0,10 €	0,10 €
Folios	20	0,02	0,40 €
Portátil	1	1500 €	1500 €
			Total: 7200,50 €

Tabla 9.1 Tabla del presupuesto del proyecto

El coste mayor es relativo a las horas de trabajo empleadas, el precio por hora esta calculado en base a una página web que hace una estimación, ver link:

[https://www.payscale.com/research/ES/Job=Software_Engineer/Salary].

Varios elementos intermedios no tienen ningún coste reseñable tales como lápices o folios. Adicionalmente la licencia de Visual Studio 2017 sale gratis si usas la versión community o si eres estudiante de la universidad de Jaén como es el caso. Finalmente un portátil donde desarrollar el proyecto es necesario y se incluirá en el presupuesto, para este caso el portátil es de última generación (ver especificaciones en sección 8 tiempos) en caso de que fuese necesario potencia gráfica.

10. Arquitectura base del prototipo

En esta sección serán comentados toda la base necesaria en la que montar el proyecto, esto incluye de que manera van a ser dibujados los modelos, la interfaz y los cambios que afecten todos estos elementos.

10.1. Capa OpenGL

Debido a la naturaleza del TFG es necesario tener una base gráfica para poder renderizar todos los elementos, se tendrán que mostrar los modelos además de ofrecer una cierta interacción al usuario a la hora de crear la fractura, posteriormente tras fracturar el modelo se deberá visualizar dicha fractura de alguna manera, para todo esto es necesario una aplicación base sobre la que montar el autómata. Con ese fin llega OpenGL, que será una parte esencial para el TFG, más específicamente OpenGL 3.3 Core Profile.

10.1.1. OpenGL 3.3 Core Profile

OpenGL será el intermediario entre la GPU y la aplicación que permitirá renderizar todo lo que se le solicite, sin embargo OpenGL tiene muchas versiones y un par de “variantes” por lo que la elección resultante debe argumentarse. Pero primero ¿qué es openGL? OpenGL es simplemente una API de una especificación multiplataforma que proporciona una capa intermedia para utilizar la GPU a través de una serie de determinados comandos resultando en gráficos 2D o 3D.

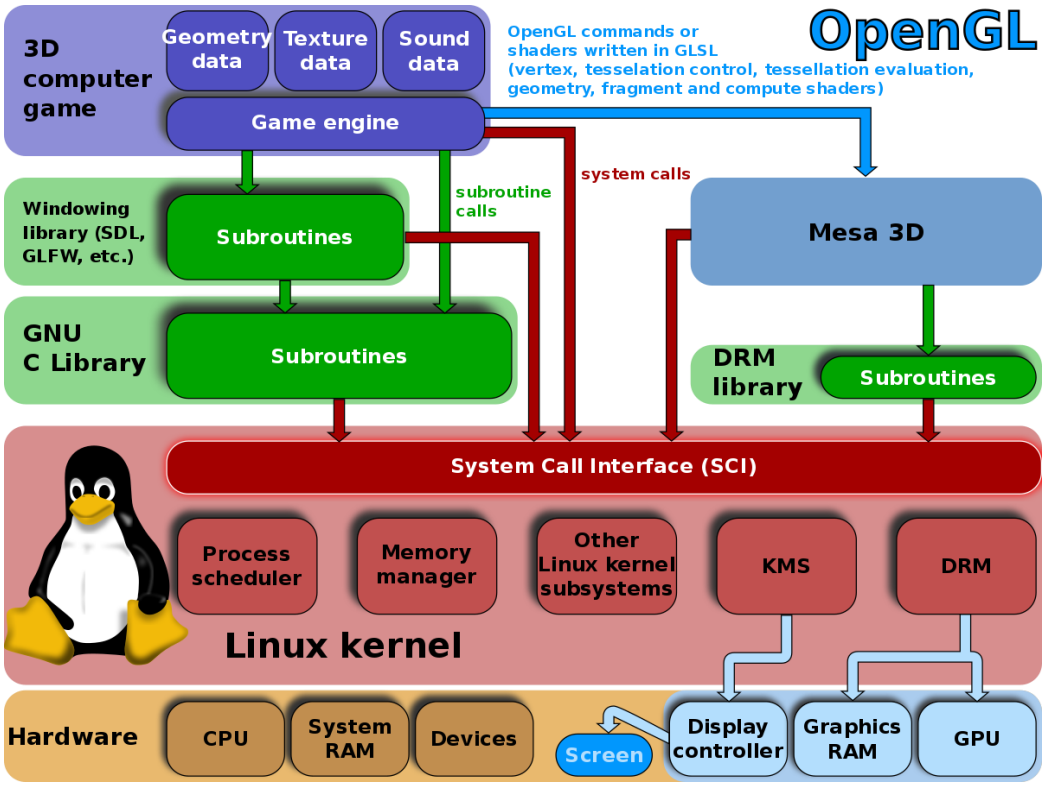


Ilustración 10.1 Flujo de OpenGL

Dentro de OpenGL se puede diferenciar dos contextos, esas dos “variantes” relativas al uso de las versiones. Por un lado está Core Context/Core Profile y por otro lado Compatibility Context/Compatibility Profile; la diferencia es muy simple, en “Core Profile” no se puede utilizar ninguna función que este deprecada, además la versión mínima en la que se puede utilizar es 3.2 puesto que por debajo se considera que ya tiene funciones esenciales deprecadas. Adicionalmente Apple o Intel solo soporta OpenGL “Core Profile” mientras que “Compatibility Profile” no es soportado. Como se presupone en el modo “Compatibility Profile” todas las funciones se pueden usar estén deprecadas o no y por tanto se puede usar cualquier versión. Considerando ambas opciones para este TFG al final “Core Profile” fue la elegida debido a la compatibilidad en todos los aspectos y a la alta customización que nos permite con shaders, funciones geométricas etc…

10.1.2. Glew y glfw

En orden para utilizar OpenGL es requerido dos librerías que nos proporcionen la funcionalidad necesaria, por un lado está “**Glew**” que es una librería multiplataforma que declara y ejecuta funciones OpenGL eliminando cierta complejidad al usuario, al

mismo tiempo comprueba que tipo de funciones OpenGL soporta la máquina que la está ejecutando. “Glew” no es estrictamente necesaria para usar OpenGL pero al eliminar cierta complejidad y estar testeado en varios sistemas operativos es muy recomendable utilizarlo, además de Glew hay otras opciones tales como GLAD, glLoadGen o GLXW **[Nigel, 2017]**.

Por otra parte está “**Glfw**” la cual es otra librería multiplataforma encargada de crear el contexto OpenGL, la ventana (en la que se mostrará todo) y manejar los eventos de entrada y salida con respecto al contexto. Dado que OpenGL no da mecanismos para hacer todas estas tareas, esta librería es necesaria para usar eficazmente OpenGL **[GLFW-Team, 2016]**.

10.1.3. Glm

Trabajar con OpenGL significa trabajar con matrices, vectores y operaciones con los mismos, por ello, es recomendable disponer de una biblioteca en la que venga predefinido todo este tipo de estructuras además de ciertas operaciones más utilizadas **[G-Truc, 2017]**. Con ese fin “**Glm**” es incluido en este TFG, “Glm” es simplemente una serie de encabezados escritos en c++ basado en el lenguaje de los shaders (GLSL) por lo cual su uso combinado con los shaders es posible. Soporta varios tipos de estructuras desde vectores de 2 posiciones hasta matrices de 4x4, además de ciertas operaciones como traslaciones, cálculo de la perspectiva (para la cámara) etc…

```
#include <glm/vec3.hpp> // glm::vec3
#include <glm/vec4.hpp> // glm::vec4
#include <glm/mat4x4.hpp> // glm::mat4
#include <glm/gtc/matrix_transform.hpp> // glm::translate, glm::rotate, glm::scale, glm::perspective
#include <glm/gtc/constants.hpp> // glm::pi

glm::mat4 camera(float Translate, glm::vec2 const& Rotate)
{
        glm::mat4 Projection = glm::perspective(glm::pi<float>() * 0.25f, 4.0f / 3.0f, 0.1f, 100.f);
        glm::mat4 View = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, -Translate));
        View = glm::rotate(View, Rotate.y, glm::vec3(-1.0f, 0.0f, 0.0f));
        View = glm::rotate(View, Rotate.x, glm::vec3(0.0f, 1.0f, 0.0f));
        glm::mat4 Model = glm::scale(glm::mat4(1.0f), glm::vec3(0.5f));
        return Projection * View * Model;
}
```

Ilustración 10.2 Ejemplo de uso biblioteca glm

10.1.4. Diseño de clases

Para la estructura del prototipo serán necesarios todos los elementos previamente mencionados fusionados junto a un diseño de clases.

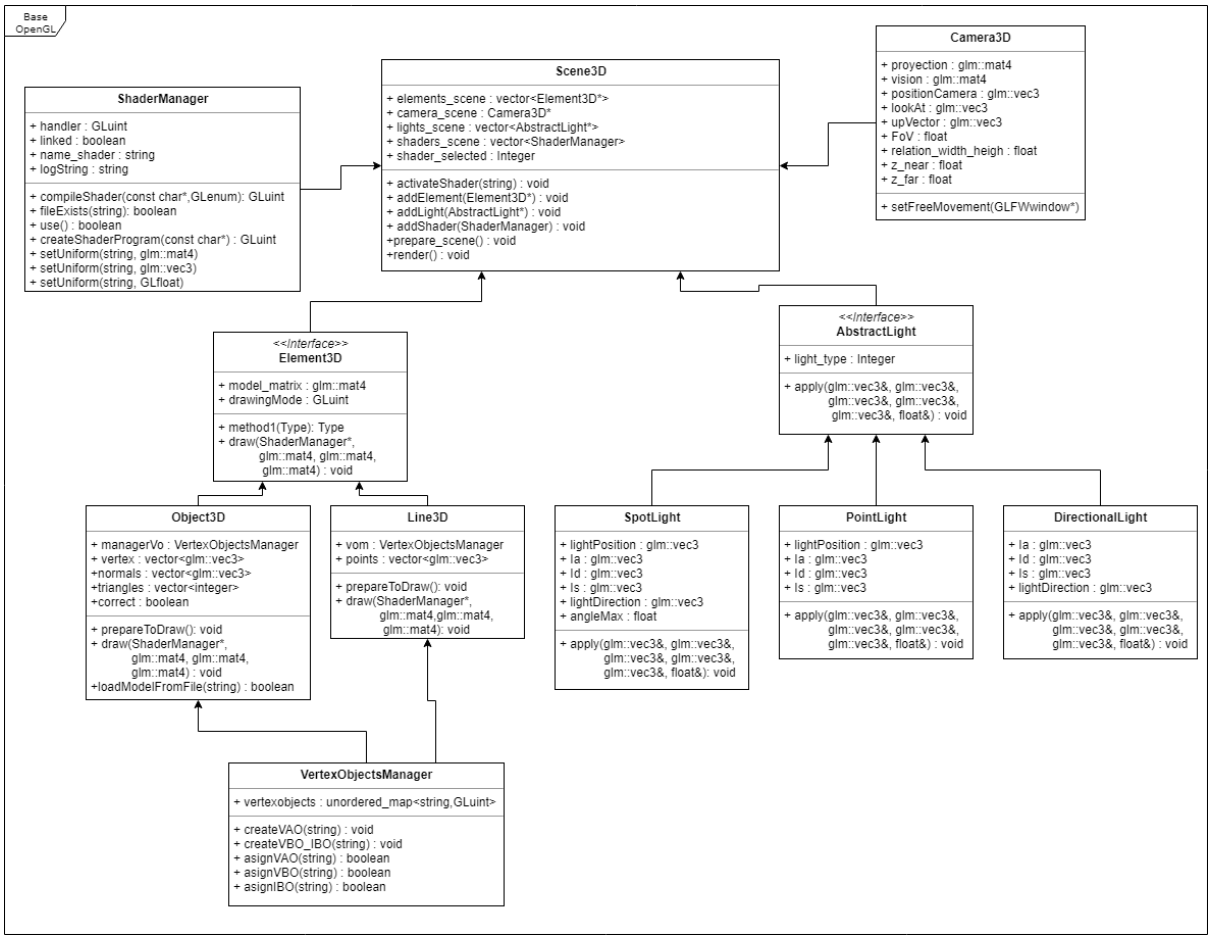


Ilustración 10.3 Diseño UML de clases para la aplicación base de OpenGL

ShaderManager: Clase encargada de cargar, compilar y usar un shader, cada shader debe tener dos archivos “.frag” y “.vert” que deben tener el mismo nombre, de otra manera la carga fallará. Si al compilarlo se encuentra algún error un log aparecerá por consola con detalles del error.

Camera3D: Clase encargada del comportamiento de la cámara 3D, es decir, la “lente” a través de la cual se visualiza el proyecto. Cuenta con una cámara con parámetros por defecto además de una más customizable.

Element3D: Clase abstracta que representa cualquier objeto que vaya a ser renderizable. Las clases derivadas serán capaces de renderizar cualquier cosa que se implemente, desde modelos hasta simples líneas. Para ello dos funciones (“prepareToDraw()” y “Draw(...)”) se deberán implementar en las clases derivadas, “**prepareToDraw()**” que se encarga de pasar la geometría pertinente a una memoria intermedia que OpenGL usará posteriormente para renderizar mientras que “**Draw(...)**” utilizara dicha memoria para dibujar. Por ende “prepareToDraw” solo se ejecutará una vez mientras que “Draw(...)” estará ejecutándose en el ciclo de eventos.

Object3D: Clase derivada de Element3D encargada de cargar y dibujar modelos 3D a partir de archivos con un rango extenso de formatos como “.3ds”, “.obj” o “.blend”. En esta clase el modelo se guarda con sus vértices, normales (necesarias para una visión más realista del modelo) e índices de triángulos.

Line3D: Clase derivada de Element3D encargada de almacenar la geometría necesaria para dibujar una línea 3D, está será utilizada para indicar el curso de la línea de la fractura creada por el algoritmo.

VertexObjectsManager: Clase encargada de proporcionar “Vertex Objects” necesarios para guardar la geometría en la estructura intermedia de OpenGL, los “Vertex Objects” son identificadores para acceder a la estructura intermedia (“prepareToDraw”) que deben definirse y almacenarse para su posterior uso(“Draw”). Cualquier clase derivada de Element3D necesitará de esta clase para guardar y utilizar sus “Vertex Objects”. Los “Vertex Objects” se guardan en un mapa desordenado para acceso eficiente siendo la key una cadena de caracteres (string).

AbstractLight: Clase abstracta que representa cualquier tipo de luz, las clases derivadas implementaran la función “apply(...)” para pasar sus datos por referencia. Al ser necesarios las variables de las luces, la función “apply(...)” funciona como puente para sacar las variables de la luz independientemente de que tipo de luz sea.

SpotLight: Clase derivada de AbstractLight que representa el tipo de luz focal donde un foco se encuentra en una posición apuntando la luz en una dirección y con un ángulo para la apertura del foco.

DirectionalLight: Clase derivada de AbstractLight que representa el tipo de luz direccional donde la luz viene de todos lados en una determinada dirección.

PointLight: Clase derivada de AbstractLight que representa el tipo de luz puntual donde la luz va en todas direcciones desde un punto en concreto.

Scene3D: Clase encargada de manejar todos los elementos necesarios para la renderización, esta clase guarda varios objetos “Element3D” los cuales pueden ser “Object3D” o “Line3D” además de las luces guardadas como “AbstractLight” pero pueden ser cualquiera de las derivadas. También se guardan en esta clase los shaders que se cargan automáticamente y el objeto de “Camera3D” para la escena. Si todos los elementos están añadidos entonces se puede renderizar la escena.

Como se puede ver, la clase en la cual convergen todas las demás sería “Scene3D” que se encargaría de manejar todos los elementos y renderizar el resultado. El ciclo de eventos es simple, primero es necesario crear un objeto “Scene3D” que tendrá que tener una serie de elementos mínimos para renderizar algo. Ese objeto “Scene3D” deberá almacenar como mínimo un objeto “Element3D” que será el modelo a renderizar, también será necesario como mínimo un objeto “AbstractLight” para iluminar la escena. Los shaders no será necesario cargarlos puesto que se cargan automáticamente y son utilizados dependiendo del modo de renderizado y el objeto a renderizar. Finalmente se deberá añadir una sola cámara a la escena, con todos los elementos añadidos y preparados el objeto “Scene3D” ya será capaz de renderizar los objetos mediantes los shaders aplicando las luces correspondientes en un modo de visión determinado por la cámara.

10.1.5. Shaders utilizados y modelo de iluminación

Los shaders utilizados en este proyecto son lo suficientemente simples como para representar un modelo de forma bastante realista pero sin profundizar demasiado para no tener que centrarse en exceso. Nos encontramos con 4 shaders distintos con una finalidad parecida pero distintos propósitos, todos ellos implementan el **modelo de iluminación de Phong**. Este modelo de iluminación y sombreado creado por Bui Tuong Phong asigna un color a cada pixel basándose en tres distintas visualizaciones, la ambiente que correspondería al color natural del propio objeto, la difusa que correspondería al efecto de la luz pertinente en ese

determinado objeto y la especular que aplicaría el brillo en ese correspondiente objeto. Para ello este modelo de iluminación toma como entrada una serie de parámetros (posición de la luz, dirección de la luz, intensidad de brillo, color ambiente, difuso y especular de la luz etc...)

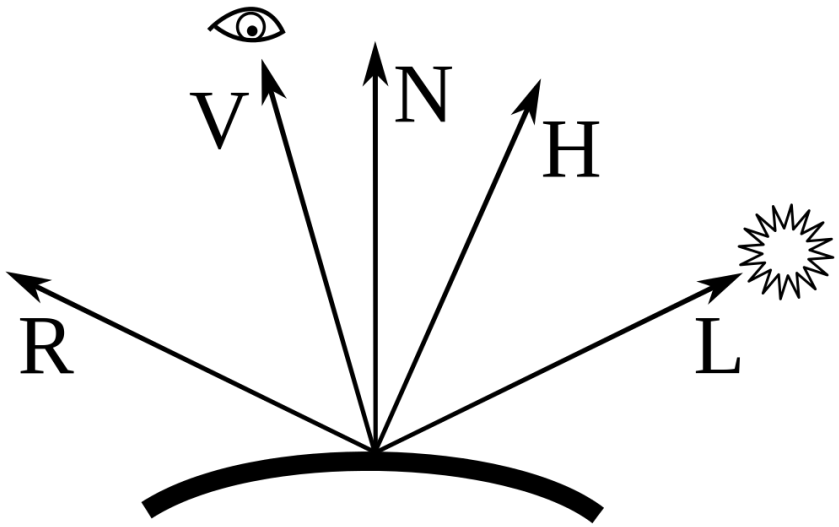


Ilustración 10.4 Vectores para la iluminación

Todos los shaders creados excepto uno han sido diseñados para seguir este modelo, la fórmula matemática que lo define es la siguiente:

$$\mathbf{I} = \mathbf{I}_a \mathbf{k}_a + \mathbf{I}_d \mathbf{k}_d (\mathbf{l} \cdot \mathbf{n}) + \mathbf{I}_s \mathbf{k}_s (\mathbf{r} \cdot \mathbf{v})^{n_s}$$

Ilustración 10.5 Fórmula de la iluminación

adsPointLight, shader creado para luces puntuales que tienen una posición e iluminan en todas direcciones.

adsDirectionalLight, shader creado para luces direccionales que no tienen una posición concreta pero iluminan en una dirección.

adsSpotLight, shader creado para luces con ángulo que tienen una posición, una dirección y además un ángulo de apertura.

line, shader creado para dibujar líneas o puntos en un color específico. Sera utilizado para representar la fractura además de la estructura de los autómatas celulares.

10.2. Cambio de tecnología a Qt

En una de las iteraciones del proyecto se denoto la imposibilidad de incluir una interfaz gráfica completa debido a la falta de las mismas para la configuración actual, de esta manera, se propuso un cambio de tecnología a Qt. En particular este cambio de tecnología no tenía demasiado impacto puesto que todo el código relativo a los diseños del autómata celular no era necesario cambiarlo gracias a que Qt utiliza c++ de base, además Qt para el apartado gráfico también utiliza OpenGL por lo que los cambios son reducidos.

Qt es un framework multiplataforma orientada a objetos enfocado a desarrollar programas que requieran de una interfaz gráfica dando soporte además para otras muchas funcionalidades tales como dibujado 3D/2D, acceso a base de datos SQL etc... **[Qt, 2018]** Nativamente utiliza c++ como lenguaje de programación pero es posible usarlo con otros lenguajes tales como Python. Adicionalmente Qt tiene una extensión para Visual Studio (**Qt Visual Studio Tools)** en la cual integra todas las bibliotecas de Qt para crear proyectos con interfaz en Visual Studio, esta extensión es la que será utilizado en nuestro port para evitar más cambios de los necesarios, además la extensión maneja archivos adicionales necesarios para el proyecto Qt. Como se ha mencionado antes Qt proporciona la posibilidad de añadir una interfaz, disponiendo así de varios elementos para ese fin, que se combina fácilmente con el resto de procesamiento necesario. Esos elementos reciben el nombre de Widgets que se van incorporando en layouts, a su vez la ventana se le añade una serie de layout (siempre en uno principal) para distribuir los elementos como sea necesario.

10.2.1. Cambios realizados

El mayor cambio al pasar el proyecto a Qt es relativo a las clases de dibujado puesto que aunque Qt utiliza OpenGL como base para dibujar, en la anterior implementación eran utilizadas las bibliotecas glew and glfw y ya no son necesarias con la integración de Qt. Para dibujar en Qt necesitamos usar “QOpenGLWidget”,

siguiendo el mismo diseño para la aplicación base de OpenGL previamente creado, la clase Scene3D deberá heredar ahora de QOpenGLWidget para convertirse en un elemento de Qt insertable en una ventana. Ahora en esta clase será necesario sobrescribir una serie de funciones para ser capaz de renderizar un entorno 3D.

paintGL(), función principal para dibujar la escena completa, a diferencia de la estructura anterior aquí no habrá un bucle infinito que se ejecutara de forma infinita hasta el fin de la ejecución del programa. En esta configuración esta función solo se ejecutara cuando sea llamada ya sea por Qt o por el usuario a través de la función "update()" siguiendo la estrategia ejecución perezosa.

initializeGL(), función para inicializar el contexto de OpenGL y hacer el procesamiento necesario. Una vez llamada esta función ya será posible utilizar funciones de OpenGL en el hilo actual, es importante mencionar que esta función no se ejecuta a la misma vez que el constructor de la clase si no después una vez la escena esta ya activa.

resizeGL(), función que se ejecutara automáticamente tras cambiar el tamaño de la pantalla de dibujo para redimensionar el objeto.

keyPressEvent(), función que se ejecutara automáticamente tras pulsar cualquier tecla del teclado pasando como parámetros la tecla pulsada, Qt dispone de un sistema de atención que determina que clases van a tener el "control" del teclado. Por eso mismo es necesario establecer en el constructor que esta clase necesitara atender las peticiones del teclado para procesarlas.

mouseMoveEvent(), función que se ejecuta cada vez que el ratón se mueve pasando como parámetros la posición "x" e "y" del ratón tanto global (en toda la pantalla) como local (solo en la ventana de dibujado).

Además de estas funciones será necesario cambiar la forma de tratar los shaders, Qt ya proporciona una clase para gestionar los shaders, de esta manera, ya no es necesario la clase que se utilizaba anteriormente "ShaderProgram".

Qt también dispone de su propia librería de estructuras de matrices, vectores de distintas dimensiones con operaciones matemáticas necesarias para la computación

gráfica. Por esta razón estas estructuras reemplazaran a glm en gran parte del proyecto (especialmente en la parte de dibujado), no obstante, glm no podrá ser totalmente eliminado puesto que sigue siendo necesario como clave en mapas desordenados. Las estructuras de Qt no soportan ser claves en estructuras como mapas.

Finalmente el último cambio necesario será todas las clases derivadas de Element3D, las cuales pueden ser renderizadas y pasan la geometría a la GPU. Este paso de geometría será necesario cambiarlo ligeramente para que sea operativo en Qt.

10.3. Interfaz

Al inicio de este proyecto la interfaz no era uno de los objetivos del mismo pero tras ver las posibilidades del proyecto una interfaz ayudaría a manejarlo todo para dejar el producto a escasos pasos de una versión completamente funcional. Con ese objetivo fueron diseñadas varias interfaces que se mostrarán a continuación, no obstante, debido al cambio de tecnología para la interfaz tenemos dos vertientes muy distintas. En cierta manera el cambio de tecnología se debió en gran parte a la interfaz puesto que trabajando en c++ con Visual Studio sin nada de base las posibilidades de interfaces quedan reducidas.

10.3.1. Diseño interfaz

La interfaz no es excesivamente compleja puesto que sirve para un propósito específico además de no ser uno de los pilares principales de este proyecto, por ello su diseño se mantendrá en todo momento simple. Por un lado la atención deberá estar en la escena de OpenGL donde el modelo y la fractura se muestra, de esta manera, la escena deberá ser una gran parte de la interfaz, alrededor de ¾ de la pantalla. En el ¼ restante tendremos el resto de la interfaz para cambiar aspectos de la ejecución, estos serán los siguientes:

Cargar distintos modelos, uno de los objetivos de la interfaz será ser capaces de cambiar el modelo en tiempo de ejecución.

Cambiar parámetros de la cámara, con el fin de mejorar la experiencia visual ciertos parámetros de la cámara serán modificables.

Generar fracturas con las técnicas propuestas, con el modelo activo otro de los objetivos será crear los autómatas celulares y generar la fractura.

Teniendo estos requisitos en mente el siguiente paso será crear un diseño para obtener una primera vista de como se vería la interfaz.

10.3.2. ImGui

Primero se opto por buscar entre bibliotecas gratuitas para comprobar si alguna se adecuaba a las necesidades del proyecto, pese a que el número de estas era bajo una de ellas parecía cumplir los requisitos. ImGui es una librería para interfaces gráficas para c++, es totalmente portable, esta optimizada y no depende de otra librería para funcionar por lo que es completamente independiente.

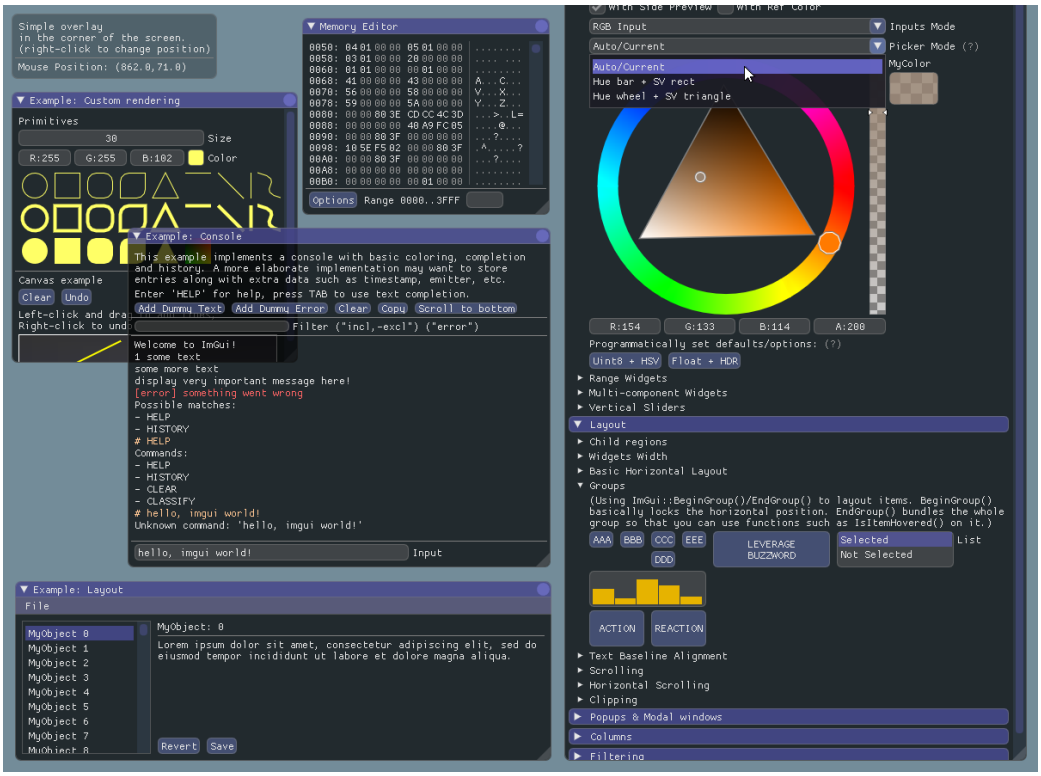


Ilustración 10.6 Ejemplo de uso ImGui(1)

ImGui requiere de alguna plataforma para computación gráfica, OpenGL, DirectX por lo que para este proyecto era perfecta puesto que ya se disponía de una configurada.

Los elementos de la interfaz se pasan como elementos a modelar, a su vez, los eventos del ratón o del teclado son capturados por ImGui para ser procesados en botones, entradas de texto y demás. Todo esto hace que ImGui este especialmente diseñada para videojuegos (menús, pantallas de inicio), no obstante, su uso en otras aplicaciones es plausible y hay varios ejemplos de ello.

10.3.3. Inclusión de ImGui

La inclusión de ImGui en el proyecto es relativamente simple, sin embargo no se incluye como una librería normal puesto que para incluirla se debe coger todas las clases y añadirlas al proyecto para ser compiladas a la vez. Una vez incluidas todas las clases es necesario crear el nexo de unión entre tu plataforma de dibujo y ImGui, afortunadamente hay varios ejemplos tanto para OpenGL como para las demás. Ese nexo de unión es el que utilizaremos para hacer las llamadas a las funciones y crear los elementos, a partir de ese momento en el bucle principal de la aplicación será muy fácil crear una ventana de la interfaz con distintos elementos.

10.3.4. Ejemplo de uso

```
ImGui::CreateContext();
ImGuiIO& io = ImGui::GetIO(); (void)io;
ImGui_ImplGlfwGL3_Init(window, true);
```

Ilustración 10.7 Ejemplo de uso ImGui(2)

Primero se crea el contexto y se inicializa pasándole la variable **“glfwWindow”** (window), una vez inicializado ya se puede empezar a crear ventanas de la siguiente manera.

```
ImGui_ImplGlfwGL3_NewFrame();
{
    ImGui::Begin("Panel de control");
    ImGui::TextWrapped("Carga del objeto.");
}
```

Ilustración 10.8 Ejemplo de uso ImGui(3)

Primero se crea la nueva ventana con “ImGui_ImplGLfwGL3_NewFrame()” para después ir poniendo los elementos requeridos. En este pequeño ejemplo se crearía una ventana pequeña con el título “Panel de control” y un texto con el contenido de “Carga del objeto”.

10.3.5. Qt

Tras el cambio de tecnología a Qt se desecho la posibilidad de utilizar ImGui debido a que Qt proporciona herramientas para la interfaz de forma intuitiva y sencilla sin tener que usar OpenGL para mostrar los elementos (más información sobre Qt en la sección [poner]). El mayor cambio con respecto a ImGui para pasar el diseño de la interfaz previamente desarrollada fue la incorporación de layout para distribuir los distintos elementos, para la interfaz se utilizaron dos tipos de layout, el horizontal y el vertical.

Layout horizontal: Distribuye los elementos de la interfaz de forma horizontal manteniendo siempre el ratio.

Layout vertical: Distribuye los elementos de la interfaz de forma vertical manteniendo siempre el ratio.

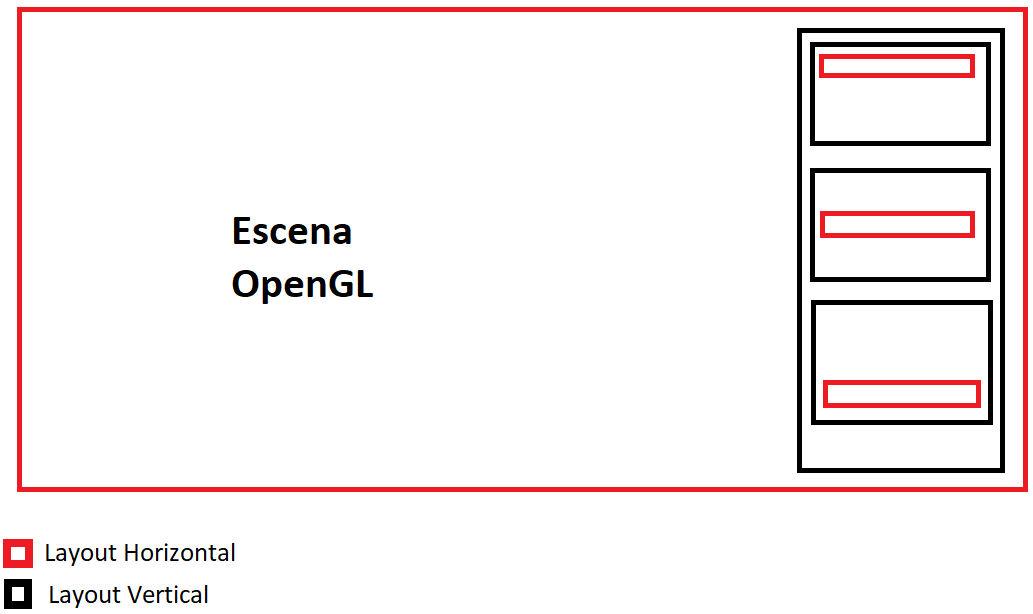


Ilustración 10.9 Distribución de los layout en la interfaz

Como se puede observar en la imagen hay distintos layout interpuestos unos en otros para cubrir las necesidades. Primero, la ventana entera tiene como layout principal uno horizontal donde irán todas las aglomeraciones de los demás, dentro de ese layout se encontrará otro vertical que contendrá todos los elementos de la interfaz tales como botones, textos etc… Para cada parte de la interfaz habrá un layout vertical diferente de tal manera que tendremos un total de 4 partes distintas (modelo, cámara, autómata celular versión grafo y autómata celular versión grid). Finalmente cada parte de la interfaz es posible que tenga a su vez otro layout horizontal para juntar una serie de elementos juntos en la misma posición.

El resultado final de la interfaz se queda de la siguiente manera:

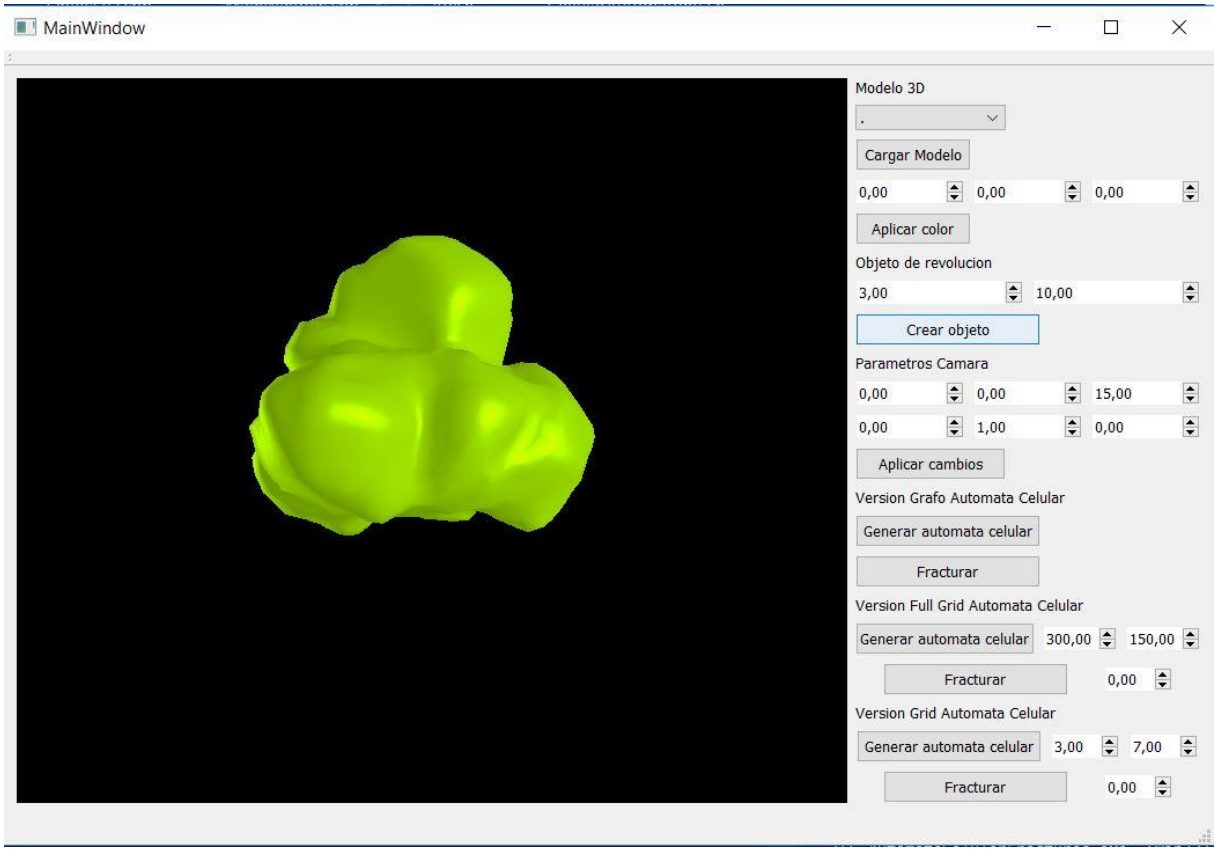


Ilustración 10.10 Resultado final de la interfaz

De manera similar a la ilustración se distribuyen los elementos, cada versión del autómata se debe primero generar (alguna con parámetros) para después fracturar con una fuerza determinada que siempre para asegurar resultados debe ser de 60 como mínimo sin llegar a 100.

11. Evaluación y aspectos de iteraciones

Las iteraciones de la metodología en general se han cubierto satisfactoriamente, no obstante, en algunas de ellas algunos objetivos fueron aplazados o dieron resultados inesperados. Esta sección abarcara esos casos:

- Para la iteración 1 y 2 (Recopilación de información e implementación de arquitectura base) no hubo problemas reseñables, la investigación fue correctamente dentro de los plazos acordados con respeto a la planificación.
- La iteración 3 (diseño e implementación de versión grafo) se completo adecuadamente exceptuando parte de la documentación que se quedo atrasada. Adicionalmente el algoritmo aleatorio creado dio resultados desfavorables por lo que se incluye en una futura iteración mejorar algoritmo.
- La iteración 4 y 5 (Diseño de versión cuadrícula e implementación de versión cuadrícula con cambio de tecnología a Qt) fueron problemáticas en el aspecto de que la implementación de la versión cuadrícula resulto ser mas complicada por lo que se tuvo que dejar como tarea para futuras iteraciones. En esta iteración (5) además se traslado el proyecto a Qt sin incidencias reseñables.
- Para la iteración 6 (Diseño e implementación algoritmo fractura e interfaz) el algoritmo de fractura fue diseñado e implementado con relativo éxito dando resultados bastante favorables, adicionalmente la interfaz también fue creada en los plazos acordados.
- En la iteración 7 se diseño la última versión del autómata celular para resolver el fallo de orientación en la anterior, aplicando el mismo algoritmo de dirección los resultados fueron buenos produciéndose fracturas completas.
- En la iteración 8 se completo toda la documentación atrasada además de hacer el deploy del prototipo. En adición fallos menores fueron resueltos.

Como se puede observar con respecto a la planificación no se llegaron a cumplir los objetivos en los plazos acordados teniendo que ampliar hasta agosto para

completar con éxito el proyecto, esto se debe a la implementación de la versión cuadrícula que resulto dar fallos inesperados (no relativos al diseño) la cual tuvo que ser revisada.

12. Conclusiones y trabajo futuro

En general todos los objetivos han sido cumplidos dejando como detalle el algoritmo de fractura que puede ser mejorado, la visión de los algoritmos utilizados es demasiado simple lo que conforma en fracturas menos realistas. Por otra parte el nivel de subdivisión y detalle conseguido con los diseños del autómata celular combinado con la coherencia espacial conseguida hace que se forme un sistema celular de infinitos usos que simula un organismo, donde las células pueden morir, nacer y realizar cualquier tarea que les haya sido programada. Esta técnica, por tanto, resulta extremadamente útil para ciertas tareas (corrosión de objetos, propagación de grietas, crecimiento de tumores, etc..) pero puede llegar a ser compleja de implementar dependiendo del uso que se le quiera dar.

Curiosamente pese a haber sido utilizada (como muestran ciertos artículos de investigación) no es especialmente popular lo que hace que aún se considere una tecnología en desarrollo. Es posible que en un futuro gane cierta presencia especialmente en áreas combinadas de medicina/fisioterapia e informática, que es en la que más usos reales tienes. En adición, áreas de la informática gráfica puede necesitar de esta técnica para representar ciertos comportamientos de objetos o organismos vivos.

Como trabajo futuro quedan muchas cosas por perfilar puesto que esto es simplemente un prototipo, una pequeña demostración de la fuerza de esta técnica. Parte del trabajo futuro consiste en partir el modelo y mostrar las dos partes del propio modelo separadas, para ese fin no basta con crear dos modelos separados porque es necesario dividir los triángulos por los cuales la fractura pasa **[Aurelien-Eric-Brett-Samir-Artis, 2004]**. Adicionalmente no todas las fracturas acaban completas por lo que no todas son válidas para dividir el modelo.

Por otra parte hay otras vertientes para el trabajo futuro respectivas a mejorar el diseño actual, en este proyecto solo se han diseñado tres versiones pero las posibilidades son infinitas. Incluso se podría considerar aumentar las dimensiones del autómata celular a uno 3D para crear una representación aún más realista.

Finalmente ligeros cambios pueden hacerse para la interfaz (haciéndola más intuitiva) y mejorar el sistema de la cámara para visualizar el modelo de una forma más clara.

Bibliografía

Artículos de investigación

[Stephane y Norishige, 1998-1999] Stephane Gobron and Norishige Chiba, 3D Surface Cellular Automata and Their Applications1998-1999

[Stephane y Norishige, 2001] Stephane Gobron and Norishige Chiba 2001, Crack pattern simulation based on 3D surface cellular automata

[Aurelien-Eric-Brett-Samir-Artis, 2004] Aurelien Martinet, Eric Galin, Brett Desbenoit, Samir Akkouche and Artis-GRAVIR INRIA, Procedural Modeling of Cracks and Fractures

Páginas web para información

[Fernando, 2016] Fernando Sancho Caparrini, 2016, http://www.cs.us.es/~fsancho/?e=66

[Lewis, 2016] Lewis Van Winkle, 2016, https://codeplea.com/triangular-interpolation

[David, 2011] David Alejandro Reyes Gómez, 2011, http://delta.cs.cinvestav.mx/~mcintosh/cellularautomata/Summer_Research_files/Arti_Ver_Inv_2011_DARG.pdf

[Sage, 2011] The Sage Developgment Team, 2011 http://verso.mat.uam.es/~pablo.angulo/doc/laboratorio/celulares.html

[Ada-Robin] Ada Yuen y Robin Kay https://pdfs.semanticscholar.org/1809/5be55b156ecfd95ea55717e9aa412906e2b7.pdf

[Qt, 2018] Qt, 2018, http://doc.qt.io/qt-5/qtopengl-hellogl2-example.html

Páginas web de bibliotecas utilizadas

[G-Truc, 2017] G-Truc Creation, 2017, https://glm.g-truc.net/0.9.8/index.html

[Nigel, 2017] Nigel Stewart, 2017, http://glew.sourceforge.net/

[GLFW-Team, 2016] GLFW Development Team, 2016, http://www.glfw.org/