Mapas Hexagonales (III) Representando el terreno

Jan 4, 2014

Mapas hexagonales (III): Representando el terreno

En el artículo anterior de la serie vimos las propiedades geométricas del hexágono y describimos un método para trabajar con mapas hexagonales utilizando esas propiedades. Pero a la hora de representar el mapa gráficamente nos limitamos a dibujar los hexágonos, sin representar el terreno.

En este artículo se introducen los elementos más importantes para representar gráficamente las características del terreno sobre un mapa hexagonal. En concreto, se describe un método que permite, por un lado, representar elementos lineales como ríos y carreteras, y por otro lado, permite representar los diferentes tipos de terreno usando transiciones entre hexágonos. En general, el método propuesto permite representar gráficamente cualquier elemento que requiera considerar las direcciones o lados del hexágono.

A nivel de programación, el enfoque descrito se basa en el uso de *enum* como alternativa eficaz de alto nivel a las codificaciones binarias basadas en banderas (*bit flags*). Por eso, para comprender el artículo adecuadamente es conveniente conocer el funcionamiento de los tipos enumerados de Java. Si no dominas esta característica de Java te sugiero que busques algún tutorial, o al menos leas el artículo incluido en este mismo blog sobre la utilización de los enum de Java. La técnica de codificación basada en banderas se describe en el artículo, por lo que no es necesario un conocimiento previo de la misma.

Modelo del terreno

En general una casilla del terreno de juego se puede caracterizar por atributos tanto cualitativos (por ejemplo para indicar el tipo de vegetación y terreno predominante) como cuantitativos (por ejemplo para indicar el nivel de fortificación del terreno o sus valores de ataque y defensa).

Por otro lado, hay que considerar la posibilidad de que algunas características del terreno sean direccionales, es decir, que sólo tengan efecto en alguna de las direcciones o lados de las celdas. El uso de direcciones en la caracterización del terreno permite representar más

adecuadamente elementos del terreno longitudinales, como ríos y acantilados, así como vías de comunicación, como carreteras y vías de ferrocarril.

Un tercer aspecto a considerar, relacionado también con el uso de direcciones, es el uso de una técnica conocida como “transiciones”, la cual se basa en la consideración de los terrenos adyacentes a la hora de pintar el tipo de terreno de una celda. Si queremos representar gráficamente un tipo de terreno, como bosque o colinas, la opción más sencilla es representarlo ocupando toda la casilla. Sin embargo, esta forma de representación resulta estéticamente poco atractiva, pues resalta mucho los hexágonos y le resta realismo al mapa, que se ve muy artificial, tal y como mostramos en el primer artículo de la serie.

Para obtener un efecto mucho más natural se pueden utilizar transiciones entre las celdas del mapa, de manera que no se pinta todo el hexágono, sino que se decora en función del tipo de terreno presente en las celdas adyacentes. Por ejemplo, si una celda tiene bosque y su vecina del norte también tiene bosque, entonces se pintarán de bosque las zonas de transición entre esas dos celdas (es decir, el norte de la primera y el sur de la segunda), pero no las zonas adyacentes a otras celdas que no tengan bosque.

Para simplificar, a lo largo de este artículo abordaremos la representación gráfica del terreno asumiendo que cualquier tipo de terreno es **direccional**, es decir, que no se refiere a una celda del mapa en su totalidad, sino sólo a una combinación de los lados o direcciones de la celda. En nuestro caso, dado que trabajamos con celdas hexagonales tenemos 6 direcciones, tal y como se describe en el artículo previo. Eso nos da un total de 2^6 = 64 combinaciones posibles de direcciones.

Dado que el número de elementos utilizados para modelar el terreno es finito, los elementos del terreno se pueden representar con estructuras de datos muy eficientes y compactas (bajo coste computacional y de memoria) basadas en objetos de tipo *enum*, como son los *EnumSet* y *EnumMap*.

Para concretar, se van a considerar los tipos de terreno incluidos en la *enum* **TerrainType**. Se trata de una modificación de la clase del mismo nombre usada en [1].

```
public enum TerrainType implements MovementEffects, ImageProviderFacto
   OPEN(0),
   SAND(1),
   HILLS(2),
   MOUNTAINS(3),
   ALPINE(IMPASSABLE),
   MARSH(3),
   WATER(IMPASSABLE),
   CROPLANDS(1),
   URBAN(0),
   ROCKY(2),
   ESCARPMENT(3),
   RIVER(2),
```

```
    SUPER_RIVER(IMPASSABLE),
    FOREST(2),
    LIGHT_WOODS(1),
    ROAD(0);

    private final int movementCost;
    private final String filename;

    private TerrainType(final int movementCost) {
        this.movementCost = movementCost;
        filename = "m_terrain_" + name().toLowerCase() + ".png";
    }

    public int getMovementCost() {
        return movementCost;
    }

    public static int getImageIndex(int bitMask) {
        return bitMask - 1;
    }

    @Override
    public String getFilename() {
        return filename;
    }

    @Override
    public ImageProvider createImageProvider() {
        return new MatrixImageProvider("", getFilename(), 8, 8, 408, 35
    }
}
```

TerrainType implementa dos interfaces: *MovementEffects* y *ImageProviderFactory*. La interfaz *MovementEffects* la debe implementar cualquier clase que describa efectos sobre el movimiento. En concreto, esta interfaz declara el método_ int getMovementCost()_, así como la constante `final static int IMPASSABLE = Integer.MAX_VALUE`, la cual se puede usar para indicar que un movimiento es imposible o no está permitido. El modelo subyacente es muy simple, y se resume en 2 ideas: (a) el coste de movimiento se expresa mediante un número entero positivo, y (b) cuanto más alto es ese número mayor es el coste de movimiento.

Por otro lado, la interfaz *ImageProviderFactory* declara 2 métodos: *String getFilename()* y *ImageProvider createImageProvider()*. El método *getFilename* devuelve el nombre del archivo que contiene los gráficos de un determinado proveedor. Si observamos la clase *TerrainType* vemos que cada tipo de terreno devuelve un nombre de archivo distinto, es

decir, que cada tipo de terreno está codificado en un archivo de imagen diferente (en este caso hemos usado imágenes en formato *png*). El método *createImageProvider* devuelve un objeto de tipo *ImageProvider*. Se trata de una interfaz que declara métodos para trabajar con imágenes almacenadas en un formato matricial. Es decir, que cada imagen se ubica en una determinada fila y columna. Así pues, encontramos métodos para obtener el número de filas y columnas, obtener las dimensiones de la imagen, recuperar la imagen completa, o recuperar una imagen situada en unas coordenadas concretas (*BufferedImage getImage(int column, int row)*). Como implementación de esta interfaz se proporciona una implementación particular denominada *MatrixImageProviderFactory*.

No se incluye aquí el código de ImageProvider y MatrixImageProviderFactory porque describen aspectos no esenciales para los fines de este tutorial. Sin embargo, se pueden consultar en el proyecto completo compartido en GitHub.

Continuamos con la clase TerrainType. Además de los métodos ya comentados, se incluye un método estático denominado *getImageIndex(int bitMask)*, el cual dado una máscara de bits nos devuelve el índice correspondiente a una cierta imagen. Como se puede ver, la obtención de este índice es algo tan sencillo como restar 1 a la máscara de bits. Este índice se puede transformar después en unas coordenadas 2D que indican en qué posición se encuentra esa imagen dentro de una matriz de imágenes. Este esquema nos permite almacenar todos los gráficos correspondientes a un cierto tipo de terreno en un único archivo de imagen. Se entenderá mejor mostrando un archivo de ejemplo con los gráficos de un determinado tipo de terreno, en este caso correspondiente al tipo FOREST.

Como se puede observar, las diferentes variaciones de un cierto tipo de terreno que podemos encontrar se representan en una matriz bidimensional. Nótese que la matriz consta de 8 * 8 = 64 imágenes distintas, que es justo el número de combinaciones de direcciones posible. Dada esa matriz y un índice de 0 a 64 es trivial el cálculo de las coordenadas (fila y columna) correspondientes a ese índice. A lo largo de este artículo y del proyecto de software que lo acompaña, asumimos el orden mostrado en este ejemplo, en el cual la columna es más significativa que la fila, lo que se traduce en las siguientes relaciones:

```
columna = índice / num-filas

fila = indíce % num-filas
```

Codificación de las direcciones

Introducción

Para entender como encajan todas las piezas del puzle, es necesario entender el mecanismo utilizado para representar el terreno de forma direccional.

En un mapa hexagonal, la información sobre el terreno se asocia a un hexágono en particular. Así pues, vamos a necesitar una clase de objetos que se encargue de representar toda la información relativa a cada celda del mapa. A continuación se muestra el código de esta clase, denominada Tile.

```
public class Tile {
    private java.util.Map<TerrainType, Directions> terrain;

    private Tile() {
        this.terrain = new EnumMap<>(TerrainType.class);
    }

    public Map<TerrainType, Directions> getTerrain() {
        return terrain;
    }

    private void setTerrain(Map<TerrainType, Directions> terrain) {
        this.terrain = terrain;
    }

    public static Tile createSingleTerrainRandomTile(TerrainType terrai
        if (probability > 1 || probability < 0)
            throw new IllegalArgumentException("Wrong probability param
        Tile tile = new Tile();
        EnumMap<TerrainType, Directions> terrain = new EnumMap<>(Terrai
        if (RandomGenerator.probabilityCheck(probability)) {
            int directions = RandomGenerator.getInstance().nextInt(Dire
            terrain.put(terrainType, Directions.ALL_COMBINED_DIRECTIONS
        }
        tile.setTerrain(terrain);
        return tile;
    }
}
```

La clase Tile debe indicar, para cada tipo de terreno posible, las direcciones presentes de ese tipo de terreno. Una primera aproximación sería el uso de una estructura de tipo Map<TerrainType, Set>, el cual asociaría a cada tipo de terreno un conjunto finito de elementos de tipo Direction. A nivel de implementación, como tanto TerrainType como Direction son de tipo enum, se podrían usar un EnumMap y un EnumSet para representar, respectivamente, el Map y el Set.

Codificación basada en máscaras de bits

Sin embargo, hay una forma aún más eficiente de representar el terreno usando enum, inspirada en la forma tradicional de codificar este tipo de problemas a bajo nivel, mediante máscaras de bits. En efecto, un conjunto de elementos con un dominio finito (por ejemplo las direcciones de una celda) se puede representar como una serie de bits que indican si un elemento particular del conjunto está presente o no. A cada uno de estos bits se le denomina bandera (bitflag), y son necesarios tantos como elementos tenga el dominio del conjunto. Por ejemplo, si tenemos 6 direcciones posibles, entonces cualquier conjunto de direcciones se puede representar con 6 bits, uno por cada dirección, lo que nos da un total 2^6 conjuntos posibles.

Al número resultante de considerar cada uno de esos bits bandera como parte de un único número binario, se le denomina máscara binaria. Para calcular una máscara necesitamos decidir el nivel de significación de cada bandera. En particular, en este proyecto empezamos asignando el bit menos significativo (el 0) a la dirección norte (Direction.N), y asignamos las demás direcciones de forma incremental, recorriendo las direcciones de la celda en sentido horario: el 1 para el NE, el 2 para el SE, el 3 para el S, el 4 para el SO, y el 5 para el NO. Se muestran a continuación algunos ejemplos de conjuntos de direcciones y las máscaras asociadas a cada uno:

```
    {N} -> 000001
    {N, NE} -> 000011
    {S, SE} -> 001100
```

Esta forma de representación binaria ocupa muy poca memoria (para cada celda se requiere un byte por cada tipo de terreno) y además permite realizar operaciones de conjunto de forma muy eficiente, mediante lógica binaria.

En Java disponemos de los enum como alternativa al enfoque basado en máscaras de bits. Es una solución de más alto nivel, más flexible, más legible, type safe y con una implementación eficiente y compacta. Además, como cada elemento enumerado lleva asociado un ordinal, es muy fácil pasar de una a otra representación según convenga.

Por ejemplo se podrían utilizar máscaras de bits para su almacenamiento en archivos y EnumSet para su manipulación. La clave para poder pasar de una representación a otra es considerar el ordinal de cada elemento enumerado como el nivel de significación del bit bandera correspondiente.

Veamos una nueva versión de la clase Direction que incorpora métodos para pasar de máscaras a conjuntos de direcciones (EnumSet).

```
public enum Direction {
    N(0, -1, -1),
    NE(1, 0, -1),
    SE(1, 1, 0),
    S(0, 1, 1),
```

```
    SW(-1, 1, 0),
    NW(-1, 0, -1),
    C(0, 0, 0);
    private final int incColumn;
    private final int incRowEven;
    private final int incRowOdd;

    private Direction(final int incI, final int incJEven, final int inc
        this.incColumn = incI;
        this.incRowEven = incJEven;
        this.incRowOdd = incJOdd;
    }

    public int getIncColumn() {
        return incColumn;
    }

    public int getIncRowEven() {
        return incRowEven;
    }

    public int getIncRowOdd() {
        return incRowOdd;
    }

    public Point getNeighborCoordinates(Point coordinates) {
        int column = coordinates.x + getIncColumn();
        int row = coordinates.y + (Util.isEven(coordinates.x) ? getIncR
        return new Point(column, row);
    }

    public static int getBitmask(Set<Direction> directions) {
        int mask = 0;
        for (Direction dir : directions) {
            int bit = 1 << dir.ordinal();
            mask |= bit;
        }
        return mask;
    }

    public static Set<Direction> getDirections(int bitmask) {
        Set<Direction> directions = EnumSet.noneOf(Direction.class);
        for (int bit = 0; bit < ALL_DIRECTIONS.length; bit++) {
            if (testBitFlag(bitmask, bit)) {
                directions.add(ALL_DIRECTIONS[bit]);
            }
        }
```

```
        return directions;
    }

    private static boolean testBitFlag(int bitmask, int bit) {
        int flag = 1 << bit;
        return (bitmask & flag) != 0;
    }
}
```

Una máscara no es más que un entero acotado. En nuestro caso tenemos 6 direcciones y por tanto sólo necesitamos 6 bits. Aunque Java proporciona la clase byte en el código proporcionado se utiliza int, que es el formato utilizado por Java para realizar operaciones con enteros, y así nos ahorramos conversiones de tipo implícitas. En realidad, y para ser más precisos, necesitamos 7 bits, pues además de las 6 direcciones reales, usamos también la dirección C, asignada al bit 6.

El método getBitmask(Set directions) nos devuelve la máscara correspondiente a un conjunto de direcciones, y el método getDirections(int bitmask) hace justo lo contrario, dada una máscara nos devuelve el conjunto de direcciones.

Dado un conjunto de direcciones, para pintar el terreno sólo tendríamos que comprobar para cada tipo de terreno las direcciones presentes, obtener su máscara, con la máscara obtener el índice, usando el método getImageIndex(int bitMask). Finalmente usaríamos el índice obtenido para obtener la imagen adecuada del ImageProvider correspondiente.

Nótese que algunos de estos métodos tienen bucles que iteran sobre las direcciones posibles. Sin embargo, es posible optimizar aún más el proceso de la siguiente forma y evitar por completo muchos de esos cálculos usando un enfoque que se describe a continuación

Codificación optimizada

Dado que el número de combinaciones de direcciones posibles es finito, podemos representar cada una de las 64 combinaciones posibles de direcciones como una constante en sí misma. Esa constante, representada como una enum podrá incluir toda la información que necesitamos precomputada. En concreto, la máscara de bits, el índice de una imagen, e incluso las coordenadas de cada imagen se pueden precomputar para todas y cada una de las 64 combinaciones de direcciones posibles. La clase Directions se encarga precisamente de hacer eso.

```
public enum Directions {

    N,
    NE,
    N_NE,
```

```
    SE,
    N_SE,
    NE_SE,
    N_NE_SE,
    S,
    N_S,
    NE_S,
    N_NE_S,
    SE_S,
    N_SE_S,
    NE_SE_S,
    N_NE_SE_S,
    SW,
    N_SW,
    NE_SW,
    N_NE_SW,
    SE_SW,
    N_SE_SW,
    NE_SE_SW,
    N_NE_SE_SW,
    S_SW,
    N_S_SW,
    NE_S_SW,
    N_NE_S_SW,
    SE_S_SW,
    N_SE_S_SW,
    NE_SE_S_SW,
    N_NE_SE_S_SW,
    NW,
    N_NW,
    NE_NW,
    N_NE_NW,
    SE_NW,
    N_SE_NW,
    NE_SE_NW,
    N_NE_SE_NW,
    S_NW,
    N_S_NW,
    NE_S_NW,
    N_NE_S_NW,
    SE_S_NW,
    N_SE_S_NW,
    NE_SE_S_NW,
    N_NE_SE_S_NW,
    SW_NW,
    N_SW_NW,
    NE_SW_NW,
```

```
    N_NE_SW_NW,
    SE_SW_NW,
    N_SE_SW_NW,
    NE_SE_SW_NW,
    N_NE_SE_SW_NW,
    S_SW_NW,
    N_S_SW_NW,
    NE_S_SW_NW,
    N_NE_S_SW_NW,
    SE_S_SW_NW,
    N_SE_S_SW_NW,
    NE_SE_S_SW_NW,
    N_NE_SE_S_SW_NW,
    C;
    private final int bitmask;
    private final Point coordinates;
    private final Set<Direction> directions;
    public static final Directions[] ALL_COMBINED_DIRECTIONS = Directio

    private Directions() {
        this.bitmask = ordinal() + 1;
        int column = ordinal() / 8;
        int row = ordinal() % 8;
        coordinates = new Point(column, row);
        directions = Direction.getDirections(bitmask);
    }

    public int getBitmask() {
        return bitmask;
    }

    public Point getCoordinates() {
        return coordinates;
    }

    public Set<Direction> getDirections() {
        return directions;
    }

    public boolean contains(Direction direction) {
        return directions.contains(direction);
    }

    public static Directions getDirections(int bitmask) {
        return ALL_COMBINED_DIRECTIONS[bitmask - 1];
    }
```

```
    public boolean containsSome(Set<Direction> directions) {
        Set<Direction> result = EnumSet.copyOf(directions);
        result.retainAll(this.directions);
        return !result.isEmpty();
    }
}
```

Lo primero que hay que destacar en esta solución es que aunque se usa la dirección C, en realidad no hacen falta 2^7, sino tan sólo 2^6 elementos. Esto se debe a que cuando aparece C lo hace siempre sola, es decir, sólo se necesita una combinación extra para incluir C, y por otro lado no es necesario representar la ausencia de terreno como valor (si un tipo de terreno no está presente en una celda se sabe porque no habrá una entrada en el atributo terrain ((Map<TerrainType, Directions>) de la clase Tile). Así pues, se necesitan 2^6 -1 + 1 = 64 constantes o elementos del tipo enumerado *Directions*. Ponemos el elemento C en último lugar, y el resto ordenados según el valor de máscara que les correponde. De esta manera, el valor de la máscara se obtiene simplemente sumando 1 al valor ordinal. Nótese que la máscara puede valer desde 1 a 65. El valor 0 representaría el caso en que un tipo de terreno no está presente, pero en la práctica no es codificarlo explícitamente como acabamos de ver. De forma complementaria, la obtención de las direcciones a partir de una máscara es tan sencilla como devolver el elemento de tipo Directions cuyo ordinal vale bitmask – 1 .

Para optimizar un poco más si cabe, las coordenadas correspondientes a cada combinación de direcciones están precomputadas (en el atributo coordinates), lo que evita tener que calcularlas cada vez que se pinta una celda.

Notas:

- La mayoría de estos métodos en realidad no son necesarios para la completar la aplicación descrita con el enfoque adoptado; su inclusión obedece más a motivos didácticos, para apoyar con código las cosas explicadas por escrito y ofrecer una visión más completa de la relación entre nuestro enfoque y el enfoque basado en máscaras de bits.
- El hecho de necesitar justo 64 elementos para codificar conjuntos de direcciones es óptimo, ya que maximiza la eficiencia de los conjuntos (y mapas) basados en enum. Esto es así por la forma en que están implementadas las colecciones de tipo EnumSet. En efecto, EnumSet es en realidad una clase abstracta con 2 especializaciones: RegularEnumSet y JumboEnumSet. La clase RegularEnumSet se utiliza cuando el número de elementos enumerados no supera 64; en ese caso sólo se necesita un número de tipo long (64 bits) para representar el conjunto en memoria. En contrapartida, si hay más de 64 elementos se emplea la clase JumboEnumSet, la cual utiliza un array de long. Así pues, un enum de 64 elementos es el óptimo en cuanto a uso de memoria (no se desperdician bits) y eficiencia (el acceso a un long es más eficiente que el acceso a un long[]).

Representación de un mapa

Ya hemos visto como codificar la información de cada celda de nuestro mapa. Finalmente, vamos a ver como obtener una representación gráfica de toda esa información contenida en celdas.

Para ello, se describe una extensión de la clase JPanel que proporciona todos los métodos necesarios para dibujar un mapa completo. En primer lugar, y antes de describir esa extensión, se presenta la descripción a nivel lógico del mapa o tablero de juego, a la cual denominados Board.

```
public class Board {
    private Tile[][] tiles;
    private int width;
    private int height;

    private Board(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public int getWidth() {
        return width;
    }

    public int getHeight() {
        return height;
    }

    public Tile[][] getTiles() {
        return tiles;
    }

    public static Board createRandomMap(int width, int height, TerrainTy
        Board board = new Board(width, height);
        board.tiles = new Tile[width][height];
        for (int i = 0; i < width; i++) {
            Tile[] tileColumn = board.tiles[i];
            for (int j = 0; j < height; j++) {
                tileColumn[j] = Tile.createSingleTerrainRandomTile(terra
            }
        }
        return board;
    }

}
```

Esta clase se utiliza como mero contenedor de celdas, organizadas en forma de un array bidimensional de tipo Tile y dimensiones width * height. Como utilidad se ha incluido un método para crear mapas aleatorios usando un tipo de terreno en particular (método createRandomMap).

Sólo queda ver como obtener la representación gráfica de toda la información incluida en un Board. Para ello se ha modificado la clase HexagonalMap. Ahora en vez de limitarse a dibujar hexágono, esta clase de panel es capaz de pintar el mapa usando los gráficos de almacenados en archivos de mapa de bits.

```
public class HexagonalMap extends JPanel {
    private int width; // Number of columns

    private int height; // Number of rows

    private int hexSide; // Side of the hexagon

    private int hexOffset; // Distance from left horizontal vertex to v

    private int hexApotheme; // Apotheme of the hexagon = radius of ins

    private int hexRectWidth; // Width of the circumscribed rectangle

    private int hexRectHeight; // Height of the circumscribed rectangle

    private int hexGridWidth;  // hexOffset + hexSide (b + s)

    private BufferedImage globalImage;
    private Board board;
    private Map<TerrainType, ImageProvider> terrainImageProvider;

    public HexagonalMap(Board board) {
        terrainImageProvider = new EnumMap<>(TerrainType.class);
        for (TerrainType tt : TerrainType.values()) {
            terrainImageProvider.put(tt, tt.createImageProvider());
        }
        this.board = board;
        this.width = board.getWidth();
        this.height = board.getHeight();
        // load some graphics to obtain the measures of the hexagonal t

        ImageProvider someImageProvider = terrainImageProvider.get(Terr
        BufferedImage someTerrainImage = someImageProvider.getImage(0);
        hexRectWidth = someTerrainImage.getWidth();
        hexRectHeight = someTerrainImage.getHeight();
        hexApotheme = hexRectHeight / 2;
```

```
        hexSide = (int) ((double) hexApotheme / Math.cos(Math.PI / 6));
        hexOffset = (int) (hexSide * Math.sin(Math.PI / 6));
        hexGridWidth = hexOffset + hexSide;
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        if (globalImage == null)
            globalImage = new BufferedImage(width * hexGridWidth + hexO
        Graphics2D g2 = globalImage.createGraphics();
        // Paint it black!

        g2.setColor(Color.BLACK);
        g2.fillRect(0, 0, globalImage.getWidth(), globalImage.getHeight
        Tile[][] tiles = board.getTiles();
        for (int i = 0; i < width; i++) {
            Tile[] tileColumn = tiles[i];
            for (int j = 0; j < height; j++) {
//                g.drawPolygon(buildHexagon(i,j));

                paintTile(g2, i, j, tileColumn[j]);
            }
        }
        g2.dispose();
        ((Graphics2D) g).drawImage(globalImage, 0, 0, null);
        g.dispose();
    }

    void paintTile(Graphics2D g2, int column, int row, Tile tile) {
        //Obtain tile position

        Point pos = tileToPixel(column, row);
        // First paint open terrain for the background

        BufferedImage terrainImage = terrainImageProvider.get(TerrainTy
        g2.drawImage(terrainImage, pos.x, pos.y, null);
        Map<TerrainType, Directions> m = tile.getTerrain();
        for (Map.Entry<TerrainType, Directions> entry : m.entrySet()) {
            TerrainType terrainType = entry.getKey();
            Directions directions = entry.getValue();
            // Next paint the actual terrain type

            Point imageCoordinates = directions.getCoordinates();
            terrainImage = terrainImageProvider.get(terrainType).getIma
            // Paint terrain image
```

```
            g2.drawImage(terrainImage, pos.x, pos.y, null);
        }
    }

    @Override
    public Dimension getPreferredSize() {
        int panelWidth = width * hexGridWidth + hexOffset;
        int panelHeight = height * hexRectHeight + hexApotheme + 1;
        return new Dimension(panelWidth, panelHeight);
    }

    private Polygon buildHexagon(int column, int row) {
        Polygon hex = new Polygon();
        Point origin = tileToPixel(column, row);
        hex.addPoint(origin.x + hexOffset, origin.y);
        hex.addPoint(origin.x + hexGridWidth, origin.y);
        hex.addPoint(origin.x + hexRectWidth, origin.y + hexApotheme);
        hex.addPoint(origin.x + hexGridWidth, origin.y + hexRectHeight)
        hex.addPoint(origin.x + hexOffset, origin.y + hexRectHeight);
        hex.addPoint(origin.x, origin.y + hexApotheme);
        return hex;
    }

    public Point tileToPixel(int column, int row) {
        Point pixel = new Point();
        pixel.x = hexGridWidth * column;
        if (Util.isOdd(column)) pixel.y = hexRectHeight * row;
        else pixel.y = hexRectHeight * row + hexApotheme;
        return pixel;
    }

    public Point pixelToTile(int x, int y) {
        double hexRise = (double) hexApotheme / (double) hexOffset;
        Point p = new Point(x / hexGridWidth, y / hexRectHeight);
        Point r = new Point(x % hexGridWidth, y % hexRectHeight);
        Direction direction;
        if (Util.isOdd(p.x)) { //odd column

            if (r.y < -hexRise * r.x + hexApotheme) {
                direction = Direction.NW;
            } else if (r.y > hexRise * r.x + hexApotheme) {
                direction = Direction.SW;
            } else {
                direction = Direction.C;
            }
        } else { //even column
```

```
            if (r.y > hexRise * r.x && r.y < -hexRise * r.x + hexRectHe
                direction = Direction.NW;
            } else if (r.y < hexApotheme) {
                direction = Direction.N;
            } else direction = Direction.C;
        }
        return new Point(direction.getNeighborCoordinates(p));
    }

    public boolean tileIsWithinBoard(Point coordinates) {
        int column = coordinates.x;
        int row = coordinates.y;
        return (column >= 0 && column < width) && (row >= 0 && row < he
    }

    public void update(Board board) {
        this.board = board;
        globalImage = null;
        repaint();
    }
}
```

En el artículo anterior se detallaba el cálculo de los parámetros gráficos necesarios para representar un teselado hexagonal a partir del valor del lado del hexágono, pasado como parámetro en el constructor. En esta nueva versión de la clase HexagonalMap, en vez de pasar el lado del hexágono en el constructor, se pasa un objeto describiendo un tablero completo (Board), y las dimensiones de los hexágonos se calculan automáticamente a partir de un archivo de imagen de ejemplo, como se puede ver en el código del constructor.

Además del constructor, la parte de código más interesante es la encargada de pintar el mapa, y se halla en los métodos paintComponent y paintTile. El primero se encarga de inicializar la imagen global, recorrer todas las celdas e invocar al segundo método para pintar cada una de las celdas.

Observe el funcionamiento del método paintTile. En primer lugar, se obtienen las coordenadas correspondiente a una celda, usando el método tileToPixel . A continuación se pinta todo el hexágono usando un gráfico de base que representa terreno abierto (TerrainType.OPEN) y se usa de base sobre la que ir pintando el resto de tipos de terreno. Finalmente, se obtiene el Map<TerrainType, Directions> de el Tile pasado como parámetro y pintan todos los terrenos. Para ellos se recorren las claves, que son los tipos de terreno presentes, y para cada clave se obtiene la imagen correspondiente al objeto Directions asociado (se obtienen las coordenadas y se solicita al proveedor de imagen apropiado la imagen correspondiente esas coordenadas). Se incluyen varios métodos que ya hemos visto en artículos anteriores, así como un método void update(Board) para actualizar la información del tablero de juego y repintarlo.

Finalmente, se ha modificado la clase principal, HexagonalMapGUI para que obtenga un mapa aleatorio y lo pase como parámetro al constructor de la clase HexagonalMap.

```
public class HexagonalMapGUI extends JFrame {
    static final int MAP_WIDTH = 10; // number of columns

    static final int MAP_HEIGHT = 10; // number of rows

    static final TerrainType SOME_TERRAIN_TYPE = TerrainType.FOREST;
    private HexagonalMap map;
    private MapInfo info;
    private JPanel mainPanel;


    public HexagonalMapGUI() {
        super("Hexagonal Board Demo");
        info = new MapInfo();
        // Create random map

        Board board = Board.createRandomMap(MAP_WIDTH, MAP_HEIGHT, SOME_
        map = new HexagonalMap(board);
        map.addMouseMotionListener(new BoardMouseMotionListener());
        mainPanel = new JPanel(new BorderLayout());
        mainPanel.add(BorderLayout.CENTER, map);
        mainPanel.add(BorderLayout.EAST, info);
        JToolBar toolBar = new BoardToolBar(map);
        mainPanel.add(BorderLayout.NORTH, toolBar);
        setContentPane(mainPanel);
    }

    public static void main(String[] args) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                HexagonalMapGUI frame = new HexagonalMapGUI();
                frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_
                frame.pack();
                frame.setVisible(true);
            }
        });
    }

    private class BoardMouseMotionListener extends MouseMotionAdapter {
        @Override
        public void mouseMoved(MouseEvent me) {
            info.setMousePosition(me.getX(), me.getY());
            Point tileCoordinates = map.pixelToTile(me.getX(), me.getY(
            if (map.tileIsWithinBoard(tileCoordinates)) {
```

```
                info.setTileCoordinates(tileCoordinates);
            } else info.setTileCoordinates(null);
        }
    }
}
```

A los 2 paneles existentes previamente, se ha añadido una barra de tareas con controles para crear un nuevo mapa aleatorio usando un tipo de terreno específico. La imagen siguiente muestra la nueva interfaz de usuario.

Se sugiere probar con diferentes terrenos, incluyendo al menos los tipos ROAD y RIVER, que son características lineales. De esta forma se pueden ver los 2 usos de las direcciones, para crear transiciones de terreno (FOREST, HILLS, etc.) y para representar características lineales (ROAD, RIVER).

Como el mapa se ha generado aleatoriamente, el resultado no es lógico (carreteras y ríos partidos, sin continuidad, y transiciones incorrectas) pero es suficiente para ilustrar los puntos desarrollados a lo largo de este artículo.

Con esta entrega damos por completados los conceptos básicos necesarios para empezar a representar gráficamente mapas hexagonales usando un modelo rico del terreno. Aunque los métodos incluidos para calcular máscaras de bits u obtener conjuntos de direcciones no son estrictamente necesarios para la solución propuesta finalmente, sí que es conveniente entenderlos para tener la relación entre la técnica propuesta y la técnica tradicional basada en máscaras de bits.

Se puede clonar o descargar el proyecto completo en GitHub. Lo que aquí se describe se corresponde con la versión etiquetada como 0.0.2.

De Ludo Bellico

De Ludo Bellico
magomar@gmail.com

magomar
magomarX

Blog dedicado a la programación de juegos de estrategia en Java. Desde la representación del mapa mediante coordenadas hexagonales hasta la creación de algoritmos de pathfinding e inteligencia artificial.