

Explore Documentation

(Preliminary Draft for v.2 Product Release)

I. User manual

Explore constructs hex-grid terrain maps from a textual specification and produces a graphical display of the result (Figure 1.) Version 2 adds a new shortest path capability.

Explore v.2 can draw any size rectangular map your computer hardware can accommodate. The display engine will automatically supply scroll bars for maps which are too big for your display.

Explore v.2 maps are composed of terrain cells and cell to cell “connectors” which describe traffic way and water ways. Connectors are “weighted” by their traffic capacity. Version 2 also includes provision for marking start and end cells and for computing the shortest path between those cells. The viewer includes new support for displaying the shortest path. (See Figure 2.)



Figure 1 - Example terrain map in Explore viewer.

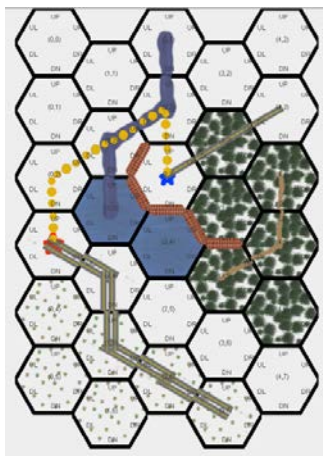


Figure 2 – A map with shortest path displayed (the dotted yellow line.)

Explore Terrain Features

Terrain maps are composed of up to four different terrain types, four different connector types, barriers, and some special markers. Each cell’s “background” graphic corresponds to the terrain type in that cell. Table 1 details the different terrain types. Traffic can cross from cell to cell, even without a connector. When there is no connector at a cell edge, the traffic capacity is that of the lower capacity cell.

Connectors cross cell boundaries, going from center to center on adjacent cells. Each Connector has a traffic capacity independent of the underlying terrain. Table 2 gives connector details. Traffic capacity goes up as edge weight (cost of travel) goes down.

Explore automatically computes the effective capacity of each cell wall, using the following policy:

- if the has a barrier, the weight for that edge is 900.
- if there is no connector between two cells, the weight is the maximum of the terrain weight in the two cells. For example, an edge between forest and default would have an effective weight of 195.
- if there is a connector between two cells the edge weight will be the connector weight. (It is possible for the edge weight to be less than the connector weight, but only if the capacity of the underlying terrain exceeded the capacity of the connector.

Table 1 – Terrain details











Name	Graphic	Represents	Weight
default	none	Relatively flat, easily traversed open land.	165
brush		Like default, but with low obstacles.	180
forest		Dense trees. Difficult for vehicular traffic	195
water		Open, relatively still water, such as a lake or pond. Not necessarily suitable for high speed craft.	190
flag1		The start cell	0
flag2		The finish point.	0

Table 2 – Connector Details

Names	Graphic	Represents	Weight
dividedhwy, Hw4		An interstate or turnpike quality high speed trafficway.	110
hwy, Hw2		A two-lane, two-way (e.g., state or county) highway.	130
unpaved, dirt		An unpaved, dirt or gravel road or path	143
river		Moving water, but may be navigable only by certain craft.	185
barrierwall, wall		A barrier.	900

Installation

Explore is supplied only as source code. Thus, there is no installer. Explore is distributed as a single zip archive file. The name of this file may differ between providers and versions. If you cannot determine the appropriate file to download, please contact your provider.

If you are a developer, you may prefer to follow the installation advice in the Developer Documentation and add the source to a project in your IDE.

To install the application as a user:

1. Download the zip file into a folder (directory) of your choice.
2. Use your local system zip utility to unpack the file. Depending on your operating, this utility may be named unzip, gunzip, zip, 7-zip, or something similar. On many systems you can simply right click on the file in your file browser and select “unzip” or “extract here”.
3. Once you have unpacked the archive, you should browse into the new file hierarchy. In the first two or three levels you should find a folder named “src”.
4. When you have found this “src” directory, open a command line shell and cd into the src directory. (If this operation is unfamiliar, ask a developer friend for help.
5. You now need to compile the code. From the command-line type:

```
javac -classpath . edu/iastate/cs228/hw5/Explore.java
```

NOTE: your system may require “\” in place of “/”.

6. If the compile operation completes without any messages indicating an error, then you should now have an executable version of Explore. To test your new installation, type the command:

```
java edu/iastate/cs228/hw5/Explore.
```

You should see the a terrain map like Figure 2

Note: Explore can be easier to use from the command line if you find the file “Explore.java” at the location referenced in the earlier “javac” command and copy it to the “src” directory and then recompile with:

```
javac -classpath . Explore.java
```

Once you have done this, you will be able to invoke maze from the src directory using only

```
Explore <filename>
```

where <filename> points to a terrain file, via either a local filename (in the same directory with Explore.class) or a relative or absolute file path.

Creating Your Own Terrain Files

Explore terrain descriptions are simple text files. Thus you can create your own mazes in any simple text editor. (Explore graph descriptions *must be* .txt files. Explore does not understand word processor or document files, such as .doc or .pdf.)

Locations

Locations in the terrain file are specified with Hex coordinates, corresponding to a vertical column number and a diagonal row number, as highlighted in Figure 3. Note that cell(0,0) is at the upper left corner, and rows and columns indexes grow larger as you move right and down. The left number in cell(x,y) corresponds to the cell's column; the right number to its diagonal row. The red star in Figure 3 is in cell(1,2).

(Tip: You can make a screen capture of a terrain map with no features, and print it to create a nicely labeled design layout grid. Creating the text file is much easier if you draw your terrain design on the grid and then transcribe the cell appropriate cell numbers.)

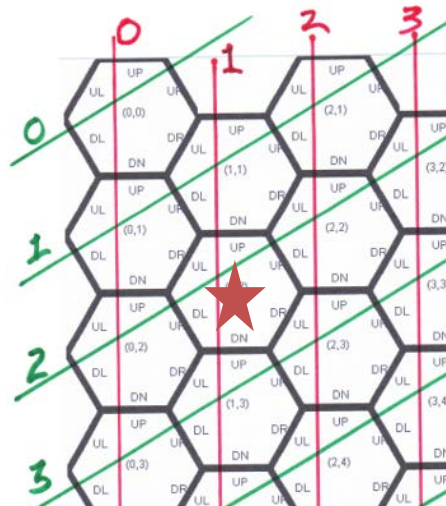


Figure 3—Hex Coordinate system.

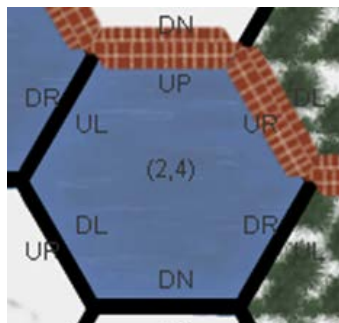


Figure 4 – Cell Walls are named by their direction from the cell center.

Directions

Cell walls are named by their direction from the center of the cell. In terrain files, these wall names (in clockwise order from the top of the cell) are abbreviated “up”, “ur”, “dr”, “dn”, “dl”, and “ul”.

Case is important! While Explore will ignore case in some instances, it is best practice to use only lower case characters in a terrain description file.

File Structure

A legal Explore terrain description consists of a geometry section (which defines the dimensions of the map) followed by sections that describe terrain, connectors, and flag locations. With the exception of the geometry section (which *must* appear only once, and *must be first* in the file) and the flag section (which may appear only once in a file), map description sections may be listed in any order and may be repeated (with different location information) as many times as you need.

```
geometry 5 6

flags
  2 3
  5 5

water 1 3 2 4

brush 0 4 0 5 1 5 1 6 2 6 3 7

forest 3 3 3 4 3 5 4 4 4 5 4 6

hwy 2 3 3 3 4 3

unpaved 3 5 4 5 4 4

dividedhwy 0 3 1 4 1 5 2 6 3 7

river 2 1 2 2 1 2 1 3

barrierwall
  2 3 ul dl dn
  3 4 dl dn
```

Figure 8 shows an example terrain description file which includes an instance of all available features and connectors.

With the exception of the geometry and barrierwall section (detailed later), each of the terrain description sections are structured as a keyword (like “water” or “river”) followed by a list of hex coordinates.

Terrain Coordinates. For terrain keywords, the list of coordinates represents individual cells of the same terrain. Thus, the list can be one or more coordinates and they can be contiguous, isolated, or broken into several clusters. Some designers find it useful to define each cluster of like terrain in a different section. Explore doesn’t care.

Connector Paths. The coordinates supplied with a connector, though, must meet constraints that assure they describe a continuous path through the centers of adjacent cells. Thus the coordinate list accompanying each connector keyword (a river, for example) must include at least two coordinates, and the coordinates must, in order, pass through centers of adjacent cells.

Flag Coordinates. The flag keyword must be followed by exactly two coordinates: one for the blue start flag and one for the red goal (or end) flag.

Geometry. The geometry keyword must be followed by exactly two integers. The first represents the number of columns in the board; the second the number of horizontal (not hex diagonal) rows. The resulting board is always rectangular.

Barrier Walls. Barrier walls are described in short sections (called border segments), where each section consists of all the cell walls (by direction) in a particular cell (by coordinate) that are blocked by a barrier. For example, the following border segment describes two full barriers in Figure 4:

2 4 up ur

Each such border segment consists of a coordinate pair followed by a list of cell walls, named by direction.

A barrierwall keyword can be followed by as many border segment descriptions as convenient. Some designers like to describe each connected run of barrier in a separate barrierwall section.

Displaying A Terrain Map

To display a particular terrain map, you should use the same command you used to display the demonstration map, but follow “.../Explore” with the path to your terrain file. If you moved Explore.java to your working directory and recompiled there (as suggested at the end of the

Tip

Many people find the terrain file easier to read and validate if each coordinate pair is bracketed with punctuation, for example as

water (1 3) (2 4)

or water 1,3 / 2,4

Explore will ignore all punctuation, giving you the freedom to use whatever convention you prefer.

installation instructions), then you should be able to display any description file located in your working directory by entering the command

```
Explore <terrain file name>
```

You can also use a relative path or an absolute path in place of the simple filename. For example if you have created a “terrains” directory in your Explore working directory, then you could load a file (say “myTerrain.txt”) from the terrains directory with

```
Explore terrains/myTerrain.txt
```

If neither of these options is convenient for you, there are other installation options that might be. Contact the discussion board or a developer friend for details.

If You Need Help

Check our discussion board. Other users have probably encountered similar problems, so there is a good chance you will find the answer there. If you don’t find an answer to your question, you can always pose a new question on the discussion board.

If you need additional support, please contact the provider from whom you acquired the zip.

II. Developer Notes

Overview, Design, and Architecture

Figure 5 shows the key inheritance and uses relationships in Explore. While there are many small classes that encapsulate certain key relationships or act as data transfer objects, the classes in this drawing are the application's primary actors.

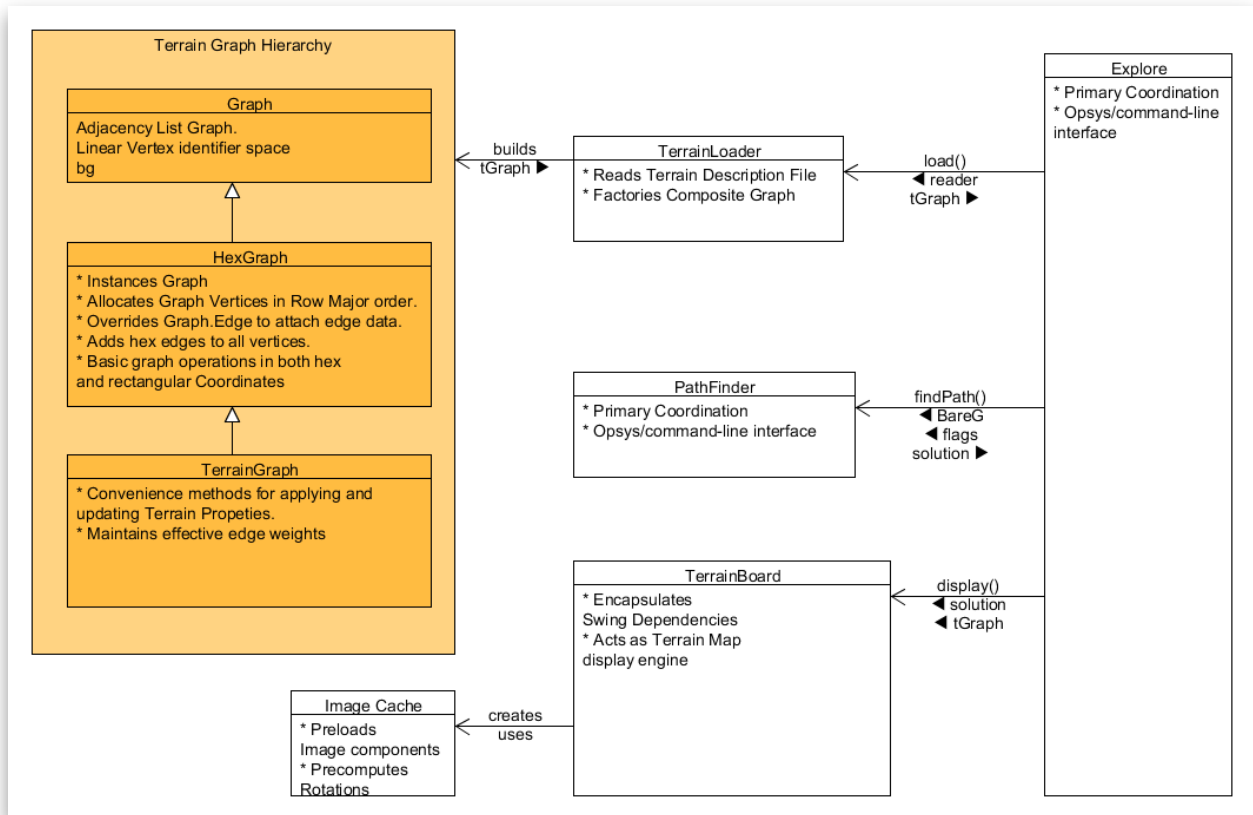


Figure 5 – Combined class hierarchy and interaction drawing.

See the code or Javadoc for API details.

Terrain File Syntax

The augmented grammar of the file is

```
<file>:: geometry <pair> <feature>+  
<feature>:: <flags>?|<terrain>*<connector>*<barrier>*  
<flags>:: <coord><coord>  
<terrain> :: (water|brush|forest) <coord>+  
<connector>::(hwy|dividedhwy|unpaved|river) <path>  
<path> :: <start coord><adj coord>*<end coord>  
<barrier>:: barrierwall <border segment>+  
<border segment> :: <coord> <dir list>
```

Additional information:

- The file *must* start with a geometry section, so that the Coordinate geometry can be set before processing any terrain features.
- Geometry and flags may only appear once. Geometry is required. The flags section is optional. If it is not present, you should not attempt to run your solver.
- The cells listed in terrain sections do not need to be contiguous. However it might make it easier for you to check the file against your terrain plan if you use a separate section for each contiguous region. Two regions of forest would be described in two different “forest” sections.
- Each connector section must list at least two coordinates (a start and an end). Each pair of connectors in a section list must correspond to adjacent cells. Thus each continuous run of highway, for example, should be described in its own section and the coordinates should name the cells in the order they are crossed. Unconnected (and crossing) highways should each have their own section. The same path-related restrictions apply to all connector types except barrierwall.

Identified Technical Debt

- Class Delta.
- Resource handling.
- Default Terrain directory
- Self-painting cells, connectors, and other graphic components.
- Independent JPanels for each drawing layer.
- Logging system
- Automated Tests
- Subclassing Graph.Edge and Graph.Vertex (vs. edge data lookup)
- Richer Path Component
- Pattern fragility in NoiseFilterReader
- Open instead of packaged library

- The viewer should have a diagnostic mode which displays linear index and effective edge weights on each vertex.

Version and Other History Notes

Unlike Maze, the 2.0 version of Explore is currently broken. We believe that, given an appropriate implementation of PathFinder, the new shortest path feature should “just work.” All of the display functionality and related wiring is there and has been tested separately.

However, a rewrite of the main parsing code in TerrainLoader has not been finished. Several important helper and data transfer classes have been completed and tested, but the code which uses these to invoke the construction methods in TerrainGraph has not been written. See the details elsewhere in the developer notes.

Feature Description – terrain file parser

TBD

Feature Description – shortest path finder

TBD

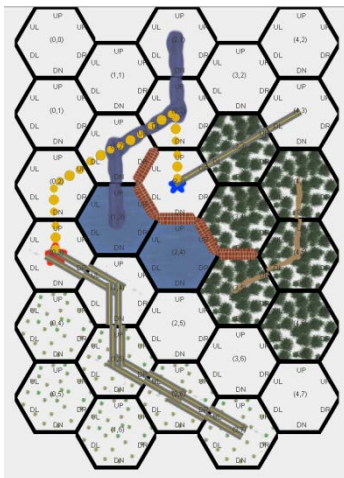


Figure 6 – The map with solution (the dotted yellow line.)

Code Details

Warnings

You MUST set the geometry!

Coordinate is also the static authority for board geometry. `Coordinate.setGeometry(cols, rows)` ***must be call before any coordinate operation are possible*** and before any vertices are created.

Coding Status in TerrainLoader

All of the work related to finding the terrain file, opening it, and connecting the appropriate chain of readers is completed, mostly outside of TerrainLoader. Within TerrainLoader, the initial parsing of

sections (identifying each type of section by its lead keyword) is complete. Additionally, both the parsing and the proper configuration of Coordinate/graph Geometry is complete.

The unfinished portions of TerrainLoader all involve parsing coordinates, paths, and border segments. There is special support in for recognizing these structures in the new TerrainScanner (which is described elsewhere in these notes). Use it.

Incomplete methods are marked with //TODO: The following wiring diagram shows how the incomplete methods should connect (use) TerrainGraph. . Most of these methods readXXX where “XXX” is some structure (like a path or border segment) in the description file. Each read method constructs an appropriate set of data transfer objects and “pushes” them to the relevant construction method in TerrainGraph, listed below:

- *setFlags(Coordinate flag1, Coordinate flag2)*. Records the cells selected for the start and end points, respectively, of the shortest path search function. Attempts to select the same cell for both flags should result in a parse error with appropriate diagnostic message.
- *setConnectedPath(Path path, ConnType type)*. Lays down a connected section of highway, or other connector.
- *setTerrain(List<Coordinate> coords, TerrainType type)*. Marks the cells listed in coords as being *type* terrain. Used for water, forrest, and brush. Note that default terrain is never marked nor indicated explicitly in the terrain description.
- *setBorderEdges(List<BorderSegment>, ConnType)*. Updates the many graph edges and weights involved in a setting up a continuous chain of barriers. Presently this method is used only for barrierwall, but it is designed to be general enough to support other types of cell perimeter attributes. Thus, the parsing code should *not* hard-wire the connector type.
- *SetSolution(Path)*. Normally, terrain files will not include this element This section is supported only to enable us to test the display engine’s support for the new shortest path feature independently of the shortest path computation. The path specified to this call is not incorporated into the main graph structure, so it has no effect on edge weights. Furthermore, the display engine will image this element on a layer above the normal connectors, so they will not be lost. Any attempt to use the same Coordinate for start and end should result in a Parse Error with appropriate diagnostic information.

The following “wiring diagram” shows how the parsing routines in TerrainLoader connect to graph operations in TerrainGraph.

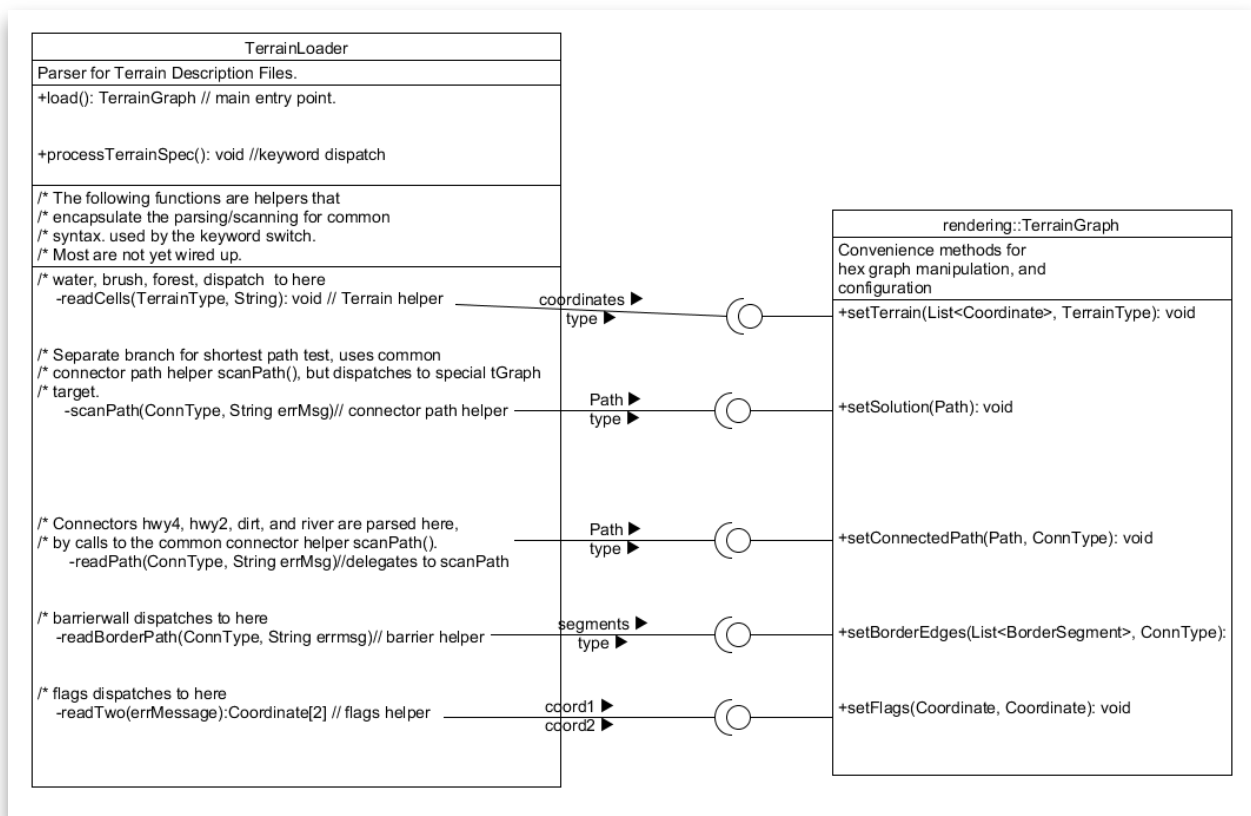


Figure 7 – Details of the intended connection between TerrainLoader and TerrainGraph. See the source or the Javadoc for individual method details.

Developer Infrastructure

Drivers and Automated tests

TestLoader is designed to make it easier to test and develop the parsing code in TerrainLoader. Using it, you can easily test specifications with just one type of section, or specific combinations of sections without needing to deal with data files and command-line invocation. While it has been used as a driver to date, it should be fairly easy to construct automated verification for specific small tests.

DTOs and Domain Abstractions

Some of the most important “developer infrastructure” in this project consists of the collection of data transfer objects and small abstractions we’ve developed to simplify operations and improve the expressibility of concepts in the hex grid space.

The key objects in this group are

Coordinate, Coordinate.Pair, Path, BorderSegment, ConnType, and TerrainType.

Some of these classes are described in more detail here, but even if the class is not described here, you should take time to find the definition and familiarize yourself with its API. You will need to use virtually all of these objects if you parse a terrain description.

Class Coordinate

Because of the need for convenient interactions among the underlying, linear Graph object, the rectangular coordinate display system, and the hex board components, convenient and reliable coordinate translation is a must. Thus we have encapsulated almost all knowledge of coordinate systems and “world geometry” in the Coordinate class. We have also encapsulated information about grid directions in a complex HexDir enum.

Class Coordinate represents both the “identity” of the cell and its associated vertex, and knows how to produce, on demand, the right corresponding identity value in any of the coordinate.

A Coordinate object can be constructed from any of the graph coordinate values.

- `Coordinate(int i)` constructs a coordinate starting from a vertex’s linear index.
- `Coordinate(int xHex, int yHex)` starts from a hex coordinate pair.
- `Coordinate(int x, int y, boolean true)` starts from a rectangular x,y pair.

Once created, a Coordinate instance can produce a convenient to use value in any of the systems. Single-valued (linear) coordinates are ints. Two-valued coordinates are returned as a Pair object which includes an external flag (‘B’, ‘R’, ‘S’) indicating whether it belongs to the Board (hex), Rectangular, or Screen coordinate system.

- `public Pair getBoard()` returns a hex pair,
- `public Pair getRect()` returns a rectangular pair, and
- `public int getLinear()` returns the linear value.

Coordinate objects also support various utility and convenience functions. For example:

- `Boolean isValid(Pair coord)` will tell you if the coordinate is within the grid or its graph.
- `Pair getDelta(Coordinate c)` computes the hex-valued difference between two coordinates.
- `Coordinate get(HexDir dir)` returns the coordinate of the board neighbor at direction dir.

Several functions test whether or not Coordinates and Pairs represent neighbors.

HexDir

In Explore, edges between vertices correspond to directions on the board, so it is convenient to have a way to identify, represent, and manipulate these directions. The complex enum HexDir (complex because it supports many mapping and characterization functions) is an important companion to Pair and Coordinate. HexDir not only gives you convenient direction names and convenient iteration through directions, it also knows how to convert Pair instances (e.g., delta’s between two Coordinates) into conveniently tested HexDir values. Of particular use in this project:

- `HexDir reverse()` returns the HexDir for 180 degrees from this direction.
- `HexDir fromAbbrev(String d)` returns the HexDir corresponding to the two letter border segment direction abbreviation (see the discussion of BorderSegments in the section on Explore’s file format).

TerrainScanner

A specialized scanner, *TerrainScanner*, knows how to recognize not just Java types, but also Coordinates, direction abbreviations, and BorderSegments. Thus, once you have recognized the *dividedhwy* keyword, you can just loop over the coordinate elements like this:

```
Path path = new Path(c1, scan.nextCoord());
while (scan.hasNextCoord()){
    path.add(scan.nextCoord());
}
```

The same strategy works for the more complex BorderSegment elements:

```
List<BorderSegment> segs = new ArrayList<BorderSegment>();
while (scan.hasNextBorderSegment()) {
    BorderSegment seg = scan.nextBorderSegment();
    segs.add(seg);
}
```

TerrainScanner will not work reliably unless the input stream has been pre-processed by *NoiseFilterReader*.

NoiseFilterReader

NoiseFilterReader which will replace all punctuation (noise or eye candy) and linefeeds with space characters. You must use this reader if you want to use *Terrain Scanner*. More importantly, though, this allows you to use whatever punctuation you find helpful when authoring a terrain description file. All of that noise, while it certainly helps readability, can greatly complicate parsing. The *NoiseFilterReader* makes it all go away.

To process a string (e.g., to make test construction more convenient), the setup would be:

```
String testData = " ..... ";
NoiseFilterReader readr = new NoiseFilterReader(new StringReader(testData));
TerrainScanner scan = new TerrainScanner(readr);
```

A similar sequence, but chaining *File* and *FileReader* in place of *StringReader* can be used to strip punctuation from your terrain descriptions.

The *NoiseFilterReader* is an example of specialization through delegation (instead of through inheritance). If you look at the definition, you will see a lot of code, but with the exception of three or four methods, all of that code was generated using the eclipse source>generate>delegate.

Diagnostic Aids

The *toString()* override on *Graph* (inherited by both *HexGraph* and *TerrainGraph*) will print the following style dump of the graph with edge weights. If called after *TerrainLoader* has finished constructing *TerrainGraph*, you will get a dump like the following, which includes the effective weight associated with each vertex. Currently this is the most convenient way to check that edge weights are being computed properly.

```

0: (0, 1: 165) (0, 5: 165)
1: (1, 2: 165) (1, 0: 165) (1, 7: 165) (1, 5: 165) (1, 6: 165)
2: (2, 7: 165) (2, 3: 165) (2, 1: 165)
3: (3, 7: 165) (3, 9: 165) (3, 2: 165) (3, 4: 165) (3, 8: 195)
4: (4, 9: 165) (4, 3: 165)
5: (5, 0: 165) (5, 6: 165) (5, 1: 165) (5, 10: 165)
6: (6, 10: 165) (6, 7: 165) (6, 11: 185) (6, 12: 900) (6, 1: 165) (6, 5: 165)
7: (7, 8: 195) (7, 12: 165) (7, 2: 165) (7, 3: 165) (7, 1: 165) (7, 6: 165)
8: (8, 7: 195) (8, 9: 130) (8, 12: 130) (8, 13: 195) (8, 3: 195) (8, 14: 195)
9: (9, 8: 130) (9, 4: 165) (9, 14: 195) (9, 3: 165)
10: (10, 15: 165) (10, 6: 165) (10, 11: 190) (10, 5: 165)
11: (11, 16: 190) (11, 15: 190) (11, 17: 190) (11, 10: 190) (11, 12: 900) (11, 6: 185)
12: (12, 11: 900) (12, 17: 900) (12, 13: 195) (12, 6: 900) (12, 7: 165) (12, 8: 130)
13: (13, 8: 195) (13, 17: 900) (13, 14: 195) (13, 19: 195) (13, 12: 195) (13, 18: 900)
14: (14, 13: 195) (14, 9: 195) (14, 8: 195) (14, 19: 143)
15: (15, 20: 180) (15, 10: 165) (15, 11: 190) (15, 16: 110)
16: (16, 20: 180) (16, 22: 165) (16, 15: 110) (16, 11: 190) (16, 17: 190) (16, 21: 110)
17: (17, 16: 190) (17, 13: 900) (17, 12: 900) (17, 18: 195) (17, 22: 190) (17, 11: 190)
18: (18, 19: 143) (18, 24: 195) (18, 17: 195) (18, 13: 900) (18, 23: 195) (18, 22: 195)
19: (19, 13: 195) (19, 24: 195) (19, 18: 143) (19, 14: 143)
20: (20, 15: 180) (20, 21: 180) (20, 25: 180) (20, 16: 180)
21: (21, 20: 180) (21, 22: 180) (21, 27: 110) (21, 16: 110) (21, 25: 180) (21, 26: 180)
22: (22, 27: 180) (22, 17: 190) (22, 18: 195) (22, 23: 165) (22, 16: 165) (22, 21: 180)
23: (23, 29: 165) (23, 24: 195) (23, 27: 180) (23, 18: 195) (23, 22: 165) (23, 28: 180)
24: (24, 18: 195) (24, 19: 195) (24, 29: 195) (24, 23: 195)
25: (25, 21: 180) (25, 26: 180) (25, 20: 180)
26: (26, 25: 180) (26, 27: 180) (26, 21: 180)
27: (27, 22: 180) (27, 28: 110) (27, 21: 110) (27, 23: 180) (27, 26: 180)
28: (28, 23: 180) (28, 29: 180) (28, 27: 110)
29: (29, 24: 195) (29, 23: 165) (29, 28: 180)

```

This is generated on stdout once the terrain map is constructed. The format is vertexId: (edge (edge) ... where each edge is (fromId, toId: weight).

I. Background Material

Coordinates: getting from linear to hex

In Explore, Terrain features are described using a particular Hex coordinate system. (There are at least three different systems one could use, so don't assume you already know how this works.)

Explore uses a hex cell with horizontal top and bottom, so the coordinates are described by vertical lines down and (in our case), diagonal lines running right and up, as in Figure 6.

These are the coordinates you will use when describing terrain features to TerrainGraph. However, the vertices in Explore each have multiple identities. When mentioned in a feature description to TerrainGraph, we identify the vertex by its hex coordinate. However, the solver you created for Maze doesn't know anything about hex coordinates, it wants to identify vertices by their "serial number" – i.e., by a linear 1-dimensional coordinate system.

This amounts to the same problem compiler developers encounter when trying to create two (or multiple) dimensional arrays in a linear memory system. Explore uses essentially the same solution. If we know the dimensions of our grid before creating the graph, then we can map sequential positions to the grid in row-major order, as in Figure 7. If we combine the solid red line segments in Figure 7 with the vertical green lines in the other figures, we have the basis for assigning rectangular coordinates to each cell. For example, linear cell 7 is also rectangular cell 2,1.

The rectangular coordinates are actually more convenient for computing screen display coordinates for each cell. Rectangular coordinates are also the most convenient starting point for computing a cell's hex coordinate. Thus, you can visualize what happens as we construct the Explore graph thus: all vertices are created left to right along a single line. That line is then 'folded' (or segmented) (based on the grid width) to create a stack of rows which are linked left and right and up and down.

That however, creates a grid where each cell/vertex has links to four neighbors. In a hex grid, there should be six links (neighbors) per cell.

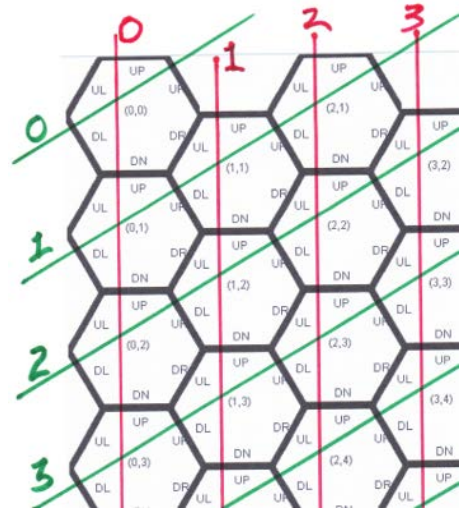


Figure 8 – The hex coordinate system used in Explore. Red lines represent x coordinates and green lines represent y coordinates.

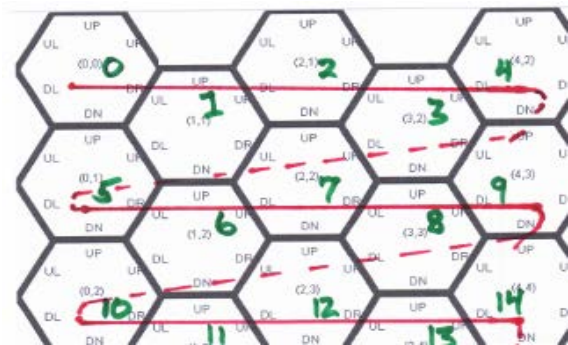


Figure 9 – The red line shows a row-major scan through a 5-wide grid. The green numbers are the linear address associated with the cell.

To get six links, we link neighbor cells up and down along the up-right diagonal (along the green lines in Figure 6). That gives us an appropriate connection pattern, then we just need to figure out compute the appropriate hex coordinate for a cell in that pattern.

The X value, of course, is identical for both Rectangular and Hex Coordinates. However, if you examine the hex coordinates for cells at the same rectangular y value, you'll see that the hex "y" changes at half the rate of the Y coordinate. The second coordinate of the cells in rectangular row 0 form the sequence 0, 1, 1, 2, 2, 3, 3 ... Thus we can transform a rectangular coordinate to a hex coordinate with these relations:

$$x_{\text{Hex}} = x_{\text{Rect}}$$

$$y_{\text{Hex}} = y_{\text{Rect}} + (x_{\text{Rect}} + 1) / 2.$$

Complex Enums?

Enums can do much more than just express an enumerated identity. The big difference between an enum and a typical class is that the enum has some predefined behavior (for example `.name()`, `values()`, and `.valueOf()`) and the java compiler automatically constructs enums with sequential identity values.

Developers, though, can add additional behaviors and override the default behavior. This can make Enums a convenient tool for abstracting certain highly coupled characteristics of types with a limited domain. See for example `HexDir` and `TerrainType`.