Swarm Space Library

The Swarm Space library is the beginnings of a library to assist in building environments for interacting agents. In general, environments can be just as varied as the agents themselves (in one view, the environment itself is simply another agent). However, many simulations have similar types of environments that can be helpfully supported by generic code.

> The current space library only addresses simple kinds of discretized 2d space. Improvement is planned in the future: see the todo list (../todo.html) for ideas. Briefly, coordinates need to be elevated to the status of objects, which should hopefully allow spaces of different scales and boundary conditions to interact through a common reference system. In addition, other types of spaces are desired: continuous coordinates, other dimensions, arbitrary graphs, etc.

Discrete2d

Root class of all 2d discrete spaces. A Discrete2d is basically a 2d array of ids. Subclasses add particular space semantics onto this. Currently Discrete2d grids are accessed by integer pairs of X and Y coordinates.

Creation

- `setSizeX: (int)x Y: (int)y`
  Set the world size.
- `createEnd`
  Create the lattice, precompute the offsets based on Y coordinate.

Querying Discrete2d state

- `(int)getSizeX` and `- (int)getSizeY`
  Get the size of the lattice.

Global Operations

- `fastFillWithValue: (int)aValue`
- `fastFillWithObject: anObject`
- `fillWithValue: (int)aValue`
- `fillWithObject: anObject`
  These methods will fill the entire space with a specified value or object. The difference between the 'fill' and 'fastFill' methods is that the former use `putObject` and `putValue` internally, in other words, they are sensitive to any subclassing and alteration of these methods, whereas the latter use low level access routines directly and are therefore much faster (and remain unaffected by subclassing).
- `copyDiscrete2d: (Discrete2d *)a toDiscrete2d: (Discrete2d *)b;`
  This method copies the data in one Discrete2d object to another Discrete2d object. It assumes that both objects already exist.
- `(int)setDiscrete2d: (Discrete2d *)a toFile: (const char *)filename;`
  This method reads a PGM formatted file and pipes the data into a Discrete2d object.

Accessing elements

The Discrete2d is essentially a 2d array of ids. However, we also allow users to access the array as if it were an array of integers: the integers are cast into ids for storage. The Object methods retrieve values as objects, the Value methods cast things to integers before returning. We believe this is safe on all architectures.

- `getObjectAtX: (int)x Y: (int)y`
  Return the pointer stored at (x,y).
- `getValueAtX: (int)x Y: (int)y`
  Return the integer stored at (x,y).
- `putObject: anObject atX: (int)x Y: (int)y`
  Put the given pointer to (x,y) overwriting whatever was there.
- `putValue: (int)v atX: (int)x Y: (int)y`
  Put the given integer to (x,y) overwriting whatever was there.

Low level access

For speed, sometimes you want to go below the method call layer and access cells quickly. To allow this Discrete2d defines some macros that give you internal access. Use these at your own peril! In particular, subclasses have no way to redefine the macros.

- `(id *)getLattice`
  Returns the lattice pointer - use this for fast access.
- `discrete2dSiteAt(lattice, offsets, x, y)`
  Macro to return a pointer to the value at site x, y. lattice is the return value from getLattice, offsets is the precomputed offsets (stored in self->offsets).

Miscellaneous

These methods are used in `- createEnd` to actually allocate the array. Subclasses might need to use these.

- `makeOffsets`
  Given an array size, compute the offsets array that caches the multiplication by ysize. See the `discrete2dSiteAt` macro.
- `(id *)allocLattice`
  Allocate memory for the lattice.

Grid2d

Grid2d is a generic container class to represent agent position on a 2d lattice. It gets most of its behaviour from Discrete2d, adding extra code to check that you don't overwrite things by accident. Grid2d is pretty primitive: only one object can be stored at a site, no boundary conditions are implied, etc. A fancier Grid2d is in the works.

New Methods

- `setOverwriteWarnings: (BOOL)b`
  If set to true, then if you try to store something at a site that doesn't have 0x0 there, a warning will be generated.

Overridden methods

- `putObject: anObject atX: (int)x Y: (int)y`
  Replaces the Discrete2d method. First check to see if it should do overwrite warnings, and if so if you're going to overwrite: if both conditions are true, print out a warning message. Regardless of the check, it writes the new object in.
- \+ `createBegin: aZone`
  Make overwrite warnings be on by default.

DblBuffer2d

DblBuffer2d augments Discrete2d to provide a form of double buffered space. Two lattices are maintained: lattice (the current state), and newLattice (the future state). All reads take place from lattice, all writes take place to newLattice. newLattice is copied to lattice when `- updateLattice` is called. DblBuffer2d can be used to implement one model of concurrent action, like in Ca2ds. `NOTE`: be very careful if you're using low-level macro access to the world, in particular be sure that you preserve the write semantics on the newLattice.

New Methods

- `updateLattice`
  Copy newLattice to lattice, in effect updating the lattice.
- `(id *)getNewLattice`
  Like `- (id *)getLattice`: return a pointer to the newLattice buffer.

Overridden Methods

- `createEnd`
  Rewrites the method from Discrete2d. Allocate two lattices, makes the offsets.
- `putObject: anObject atX: (int)x Y: (int)y` and `- putValue: (int)value atX: (int)x Y: (int)y` and
  Overridden so writes happen to newLattice.

Ca2d

Inherits from DblBuffer2d, defines abstract protocol for cellular automata.

New Methods

- `setNumStates: (int)d`
  Record the number of states the CA understands.
- `initializeLattice`
  Use this to set up your CA to a default initial state. Unimplemented in Ca2d.
- `stepRule`
  One iteration of the CA rule. Unimplemented in Ca2d.

Overridden Methods

- `createEnd`
  Check that numStates has been set.

ConwayLife2d

Classic 2d Conway's Life CA.

Overridden Methods

```
+createBegin: aZone
     Set number of states to 2.
- initializeLattice
     Initialize lattice to random 1/3 in state 1.
- stepRule
     Run Conway's Life rule (simpleminded version).
```

Diffuse2d

Discrete 2nd order approximation to 2d diffusion with evaporation. Math is done in integers on the range [0,0x7fff].

New Methods

```
- setDiffusionConstant: (double)d
     Set the diffusion constant. Values over 1.0 might not be valid.
- setEvaporationRate: (double)d
     Set the evaporation rate. Values over 1.0 don't make much sense.
```

Overridden Methods

```
+createBegin: aZone
     Set diffusion constant and evaporation rate to 1.0, numStates to 0x7fff.
- initializeLattice
     Initialize world to 0.
- stepRule
     Run discrete approximation to diffusion. Roughly, it's
     newHeat = evapRate * (self + diffuseConstant*(nbdavg - self))
     where nbdavg is the weighted average of the 8 neighbours.
```

Object2dDisplay

Object2dDisplay helps display 2d arrays of objects. Create a Object2dDisplay, give it a Raster widget to draw on, a Discrete2d, a message to call on each object, and (optionally) a collection of objects and it will dispatch the message to all objects with the Raster widget as an argument. In addition, Object2dDisplay can help you make probees.

Creation Phase

```
- setDisplayWidget: (Raster *)r
     Set the display widget to use for drawing.
- setDiscrete2dToDisplay: (Discrete2d *)c
     Set the 2d array to draw
- setDisplayMessage: (SEL)s
     Set the message to be sent to each object in the grid to make it draw itself.
- setObjectCollection: objects
     (optional) set a collection of objects to be displayed. If this is not given,
     then Object2dDisplay loops through the 2d grid sending draw messages to all
     objects it finds there. Giving an explicit collection of objects to draw
     is more efficient if your grid is sparsely populated.
```

Use

```
- display
     Draw all objects in the array (or optionally, the collection) on the raster
     widget. All that happens here is the display message is sent to each object
     - it is the object's responsibility to render itself.
- makeProbeAtX: (int)x Y: (int)y
     Find an object at the given (x,y) coordinate and build a probe display for
     it. This is a good method to make a button client for a raster widget like
```

```
[aRaster setButtonClient: aObject2dDisplay Message: M(makeProbeAtX:Y:)]
```

Value2dDisplay

Value2dDisplay helps display 2d arrays of values. Value2dDisplay goes through a given Discrete2d array, turn states into colours, and draws them into a Raster widget.

Creation Phase

```
- setDisplayWidget: (Raster *)r Colormap: (XColormap *)c
     Set the display widget and the colourmap to use to draw the value array.
- setDiscrete2dToDisplay: (Discrete2d *)c
     Set which array to draw.
- setDisplayMappingM: (int)m C: (int)c
     Linear transform of states to colours for drawing.
     color = state / m + c
     If not set, assume m == 1 and c == 0.
```

Use

```
- display
     Draw the array on the given widget. Note that you still have to tell the
     widget to draw itself afterwards. The code for display uses the fast macro
     access in Discrete2d on the cached return value from getLattice. It also
     caches the drawPointX:Y: method lookup on the display widget - this is a nice
     trick that you might want to look at.
```

Int2dFiler

The Int2dFiler class is used to save the state of any Discrete2d object (or a subclass thereof) to a specified file.

A Note on Output Formats

The issue of what format(s) the class should know about is still open. For the purposes that I have encountered, simply storing the values in space-delimited line format:

```
value0_0 value0_1 ...
value1_0 value1_1 ...
...
...
```

has been sufficient. In particular, the output of this class has been used, in conjunction with the `fragstats` `(ftp://ftp.fsl.orst.edu/pub/fragstats.2.0/)` package, to apply landscape metrics to the output of a Swarm simulation. Nevertheless, I (`Manor Askenazi` `(mailto: manor@santafe.edu)`) would be very interested in hearing from end-users about any other (presumably more complex formats), as the need arises.

```
- setDiscrete2dToFile: (Discrete2d *)aSpace
     Set the target space to be filled. This message can be used more than once,
     but often it is useful to keep one Int2dFiler per space (e.g. when the space
     is saved multiple times).
- setValueMessage: (SEL)aSelector
     This message is optional. It is used when the target Discrete2d contains
     objects. By sending each object the message specified by the selector, the
     Int2dFiler is able to get from the object an integer representing its state,
     which it then writes to the file.
- setBackground: (int)aValue
     This message is optional. It is used when the target Discrete2d contains
     objects. If a particular location in the space has no resident object, the
     argument of this message is the value which gets writtent to the file. The
     default background value is 0.
- fileTo: (const char *)fileName
     When the Int2dFiler receives this message, it opens a file called fileName,
     stores the state of a pre-specified space into it, and then closes the file.
```

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1997-12-17