DGtal: Topology module
http://liris.cnrs.fr/dgtal

Jacques-Olivier Lachaud

DGtal Meeting, September 2011

UMR 5127

Package description

Should contain

- classical digital topology *à la* Rosenfeld
- cartesian cellular topology
- digital surface topology *à la* Herman
- must be the base block of geometric algorithms

Examples

- adjacencies, connected components, simple points, thinning
- cells, boundary operators, incidence, opening, closing
- contours, surfel adjacency, surface tracking
- topological invariants

Location

- `{DGtal}/src/DGtal/topology`
- `{DGtal}/src/DGtal/helpers`
- `{DGtal}/tests/topology`

Available in DGtal 0.4

1. classical digital topology
   - Arbitrary adjacencies in $\mathbb{Z}^n$, but also in subdomains
   - Digital topology = couple of adjacencies (Rosenfeld)
   - Object = Topology + Set
   - Operations : neighborhoods, border, connectedness and connected components, decomposition into digital layers, simple points
2. cubical cellular topology
   - cells, adjacent and incident cells, faces and cofaces
   - signed cells, signed incidence,
3. digital surface topology
   - surfels, surfel adjacency, surfel neighborhood
   - surface tracking (normal, fast), contour tracking in nD

Adjacency

Genericity ⇒ concept `CAdjacency`

- Inner types : `Space`, `Point`, `Adjacency`
- Methods :
  - `isAdjacentTo`(p1, p2)
  - `isProperlyAdjacentTo`(p1, p2)
  - `writeNeighborhood`(p, output_iterator)
  - `writeProperNeighborhood`(p, output_iterator)
  - `writeNeighborhood`(p, output_iterator, predicate)
  - `writeProperNeighborhood`(p, output_iterator, predicate)
- Models :
  - `MetricAdjacency` : 4-, 8-, 6-, 18-, 26-, $2n$-, $3^n - 1$-adjacencies
  - `DomainAdjacency` : adjacency limited by a specified domain.

Usage

```
typedef SpaceND<2> Z2i;
// Simple definition of metric adjacencies
typedef MetricAdjacency< Zi2, 1 > Adj4;
typedef MetricAdjacency< Zi2, 2 > Adj8;
Adj4 adj4;
Adj8 adj8;
// Adjacencies restricted to some given set.
typedef DigitalSetDomain<DigitalSet>
    RestrictedDomain;
typedef DomainAdjacency< RestrictedDomain, Adj4 >
     RestrictedAdj4;
typedef DomainAdjacency< RestrictedDomain, Adj8 >
     RestrictedAdj8;
DigitalSet mySet ...;
RestrictedDomain myDomain( mySet );
RestrictedAdj4 myAdj4( myDomain, adj4 );
RestrictedAdj8 myAdj8( myDomain, adj8 );
```

Digital topology

Digital topology = couple of instances of adjacencies

- template class `DigitalTopology`

```
typedef SpaceND< 3,int > Z3;
typedef MetricAdjacency< Z3, 1 > Adj6;
typedef MetricAdjacency< Z3, 2 > Adj18;
typedef DigitalTopology< Adj6, Adj18 > DT6_18;

Adj6 adj6;
Adj18 adj18;
DT6_18 dt6_18( adj6, adj18, JORDAN_DT );
```

- Jordan topologies may be specified (for future use)
- instances are necessary (e.g., adj may not be invariant by translation)
- reverse topology is the reversed couple

Digital Object

Digital object = topology + digital set

- template class `Object`

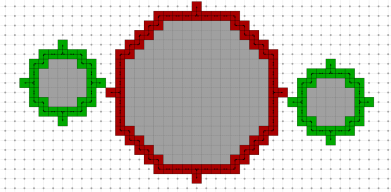
```
typedef HyperRectDomain< Z3 > Domain;
typedef DigitalSetSelector<Domain, BIG_DS+
    HIGH_BEL_DS>::Type DigitalSet;
typedef Object<DT6_18, DigitalSet> ObjectType;
Point p1( -50, -50, -50 );
Point p2( 50, 50, 50 );
Domain domain( p1, p2 );
// ball of radius 30
DigitalSet ball_set( domain );
Shapes<Domain>::addNorm2Ball( ball_set, Point(
    0, 0 ), 30 );
ObjectType ball_object( dt6_18, ball_set );
ObjectType clone( ball_object ); // no cost
```

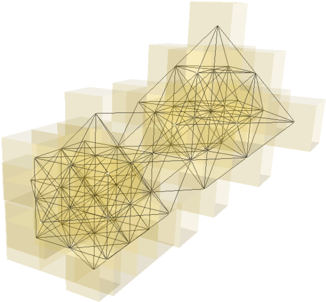
- Objects use smart pointers : they may be passed by value and copied without cost

Digital Object : main services

- `neighborhood`(Point), `properNeighborhood`(Point) return an `Object`
- border : set of point λ-adjacent to background. `border`() return an `Object`
- geodesic neighborhoods [Bertrand93]. `geodesicNeighborhood<TAdj>`(TAdj, Point, uint) return an `Object`
- (lazy) connectedness : `connectedness`, `computeConnectedness` ; connected components : `writeComponents`
- simple points (valid in Z2 and Z3). `isSimple`(Point) return a `bool`
- and Objects are drawable in 2D and in 3D (with adjacencies or not).

Digital Object : main services

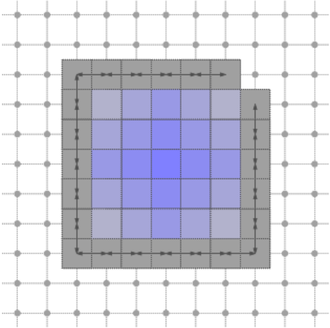




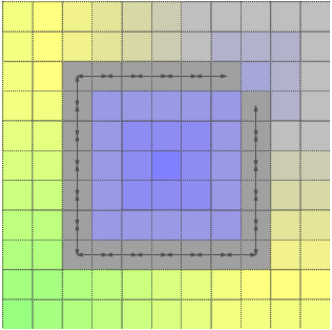
Expander : digital layers in an object

- Expansion layer by layer within an object, starting from an initial core
- core = a point or a pointset specified by iterators
- each new layer = the set of points of the object adjacent to the preceding layer
- each layer is iterable, has a digital distance to core
- finished when no more neighbor expansion is possible
- useful for connectedness, geodesic neighborhoods and thus simpleness

Expander : digital layers in an object



background in 4-adj



background in 8-adj

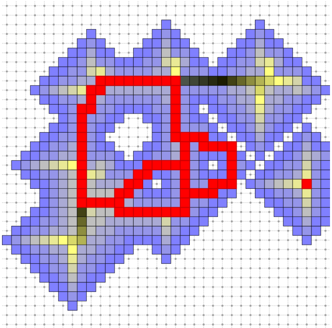
`tests/topology/testSimpleExpander.cpp`

Example : greedy homotopic thinning

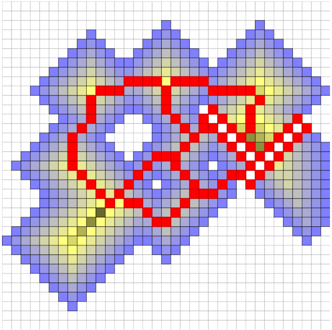
```
int layer = 0;
do {
    DigitalSet & S = shape.pointSet();
    std::queue<DigitalSet::Iterator> Q;
    for ( DigitalSet::Iterator it = S.begin(); it != S.end(); ++it )
      if ( shape.\alertred{isSimple}( *it ) )
        Q.push( it );
    nb_simple = 0;
    while ( ! Q.empty() ) {
      DigitalSet::Iterator it = Q.front();
      Q.pop();
      if ( shape.isSimple( *it ) ) {
        S.erase( *it );
        ++nb_simple;
      }
    }
    ++layer;
} while ( nb_simple != 0 );
```

See `testObject.cpp`

Example : greedy homotopic thinning



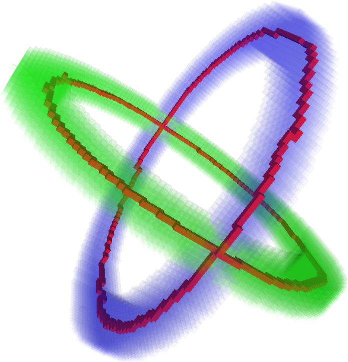
thinning in (4,8)



thinning in (8,4)

`tests/topology/testObject.cpp`

Example : greedy homotopic thinning 3D

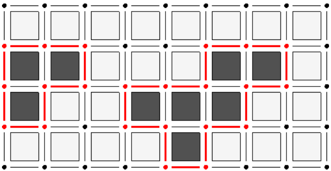


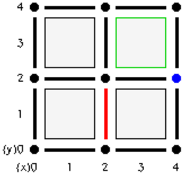
thinning in (6,26)

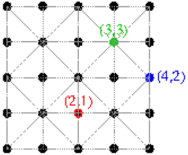
The thinning algorithm is the same as in 2d.

Digital space as a regular cubical cell complex

- classical combinatorial topology : cellular decomposition of $\mathbb{R}^n$ into the regular grid topology [Khalimsky,Kovalevsky]
- cellular complex whose cells are points, unit edges, unit squares, etc
- Khalimsky view as a cartesian product of $\mathbb{Z}^n$ with alternate topologies.







- even coordinate = closed, odd coordinate = open

Model of cubical cellular space I

Genericity $\Rightarrow$ concept `CCellularGridSpaceND`

Model `KhalimskySpaceND<dim,Integer>`

- Inner types : `Space`, `Point`, `Vector`, ...
  `Cell`, `SCell`, `Cells`, `SCells`
- the user provide a bounding box at space creation
  `init`(Point, Point, bool) returns bool
- cells may be signed (algebraic manipulation)
- cells are black boxes : managed through methods of space

Model of cubical cellular space II

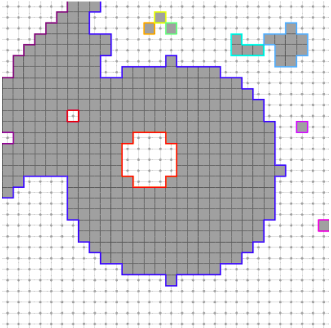
- cells are black boxes : managed through methods of space
  - creation : `uCell`, `sCell`, ...
  - read/write access : `uCoord`, ...
  - sign services : `signs`, `unsigns`, `sOpp`,
  - topology services : `uDim`, `uIsSurfel`, ...
  - direction iterators : `uDirs`, `uOrthDirs`, ...
  - geometric services : `uFirst`, `uLast`, `uTranslation`, `uProjection`, ...
  - neighborhood services : `uNeighborhood`, `uAdjacent`, ...
  - incidence services : `uIncident`, `uFaces`, ...
  - direct orientation service : `sDirect`, ...

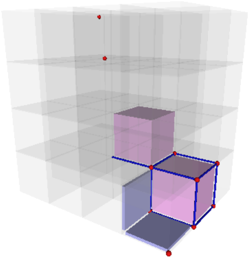
Example : cell creation and view

```
Viewer3D viewer;
...
KSpace K;
Point plow(0,0,0);
Point pup(3,3,2);
// should return true
K.init( plow, pup, true );
// Drawing cell of dimension 3
Cell voxelA = K.uCell(Point(1,1,1));
SCell voxelB = K.sCell(Point(1,1,3));
viewer << voxelB << voxelA;
// drawing cells of dimension 2
SCell surfelA = K.sCell( Point( 2, 1, 3 ) );
SCell surfelB = K.sCell( Point( 1, 0, 1 ), false );
Cell surfelC = K.uCell( Point( 1, 2, 1 ) );
SCell surfelD = K.sCell( Point( 1, 1, 0 ) );
Cell surfelE = K.uCell( Point( 1, 1, 2 ) );
viewer << surfelA << surfelB << surfelC << surfelD
    << surfelE;
```

Visualization of cells in 2D/3D

You can put cells in a 2D/3D visualization stream.

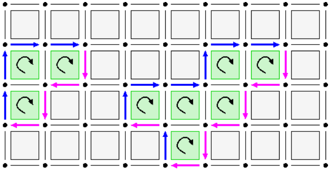




Why signing cells : algebraic view

- *r-chain* : formal sum of r-cells
  - Example : $\sum_i +o_i^n$, with o_i^n n-cells, is a digital object
  - Example : $\sum_i a_j s_j^{n-1}$, with s_j^{n-1} $n-1$-cells, is a digital surface

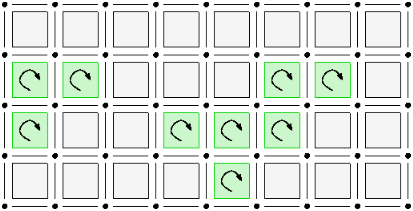
spels $+o_i^n$
surfels $+s_j^{n-1}$
and $-s_j^{n-1}$



Why signing cells : algebraic view

- *r-chain* : formal sum of r-cells
- Linear operators boundary Δ and co-boundary ∇
  - Δ : r-chain $\mapsto r-1$-chain ($\equiv$ (low) incidence)
  - ∇ : r-chain $\mapsto r+1$-chain ($\equiv$ (up) incidence)
  - $\Delta\Delta = 0$ and $\nabla\nabla = 0$ (Homology)

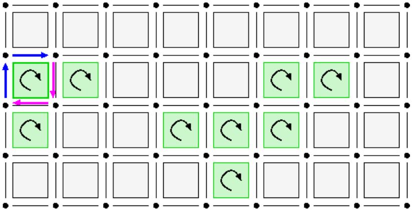
$\Delta \sum +o_i^n$?



Why signing cells : algebraic view

- *r-chain* : formal sum of r-cells
- Linear operators boundary Δ and co-boundary ∇
  - Δ : r-chain $\mapsto r-1$-chain ($\equiv$ (low) incidence)
  - ∇ : r-chain $\mapsto r+1$-chain ($\equiv$ (up) incidence)
  - $\Delta\Delta = 0$ and $\nabla\nabla = 0$ (Homology)

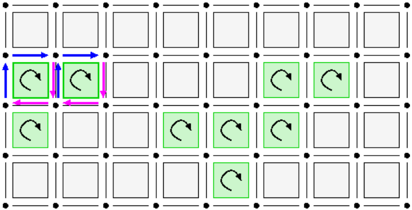
$\Delta \sum +o_i^n$?



Why signing cells : algebraic view

- *r-chain* : formal sum of r-cells
- Linear operators boundary Δ and co-boundary ∇
  - Δ : r-chain $\mapsto r-1$-chain ($\equiv$ (low) incidence)
  - ∇ : r-chain $\mapsto r+1$-chain ($\equiv$ (up) incidence)
  - $\Delta\Delta = 0$ and $\nabla\nabla = 0$ (Homology)

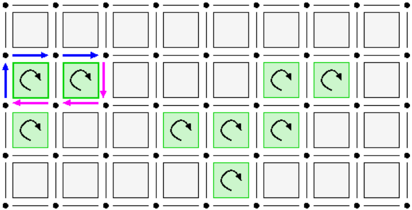
$\Delta \sum +o_i^n$?



Why signing cells : algebraic view

- *r-chain* : formal sum of r-cells
- Linear operators boundary Δ and co-boundary ∇
  - Δ : r-chain $\mapsto r-1$-chain ($\equiv$ (low) incidence)
  - ∇ : r-chain $\mapsto r+1$-chain ($\equiv$ (up) incidence)
  - $\Delta\Delta = 0$ and $\nabla\nabla = 0$ (Homology)

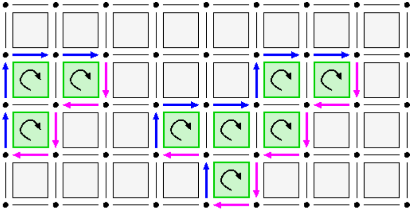
$\Delta \sum +o_i^n$?



Why signing cells : algebraic view

- *r-chain* : formal sum of r-cells
- Linear operators boundary Δ and co-boundary ∇
  - Δ : r-chain $\mapsto r-1$-chain ($\equiv$ (low) incidence)
  - ∇ : r-chain $\mapsto r+1$-chain ($\equiv$ (up) incidence)
  - $\Delta\Delta = 0$ and $\nabla\nabla = 0$ (Homology)

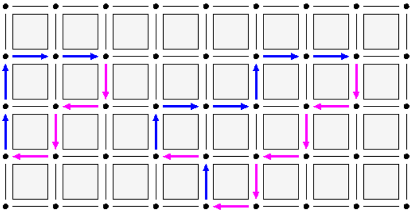
$\Delta \sum +o_i^n$?



Why signing cells : algebraic view

- *r-chain* : formal sum of r-cells
- Linear operators boundary Δ and co-boundary ∇
  - Δ : r-chain $\mapsto r-1$-chain ($\equiv$ (low) incidence)
  - ∇ : r-chain $\mapsto r+1$-chain ($\equiv$ (up) incidence)
  - $\Delta\Delta = 0$ and $\nabla\nabla = 0$ (Homology)

$\Delta \sum +o_i^n$?

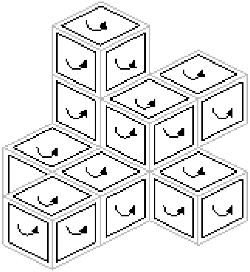


Why signing cells : algebraic view

- *r-chain* : formal sum of r-cells
- Linear operators boundary Δ and co-boundary ∇
  - Δ : r-chain $\mapsto r-1$-chain ($\equiv$ (low) incidence)
  - ∇ : r-chain $\mapsto r+1$-chain ($\equiv$ (up) incidence)
  - $\Delta\Delta = 0$ and $\nabla\nabla = 0$ (Homology)
- coefficients are generally taken ± 1. It is enough to sign cells to design that kind of operators.

Applications

1. Any object boundary is closed.
   Boundary of a digital object $O = \Delta O$

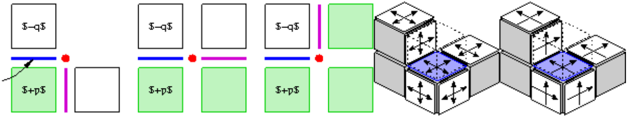


∂O is a closed surface.

Since $\Delta\Delta = 0$, the boundary of a digital object is a surface without boundary.

Applications

1. Any object boundary is closed.
2. Neighborhood and tracking over ∂O
   - Any surfel has $2n - 2$ neighbors
   - Formal definition of the two neighbors of a surfel σ, of orth. dir. i, along direction $j \neq i$.
     $\Delta_i^\epsilon \nabla_j^\mu \sigma$, $\nabla_i^\epsilon \Delta_i^\epsilon \sigma$, $\Delta_i^\epsilon \nabla_j^{-\mu} \sigma$ with $\mu = \pm 1$ and $\epsilon = \pm 1$



   - Neighbors are oriented (direct or indirect orientation)

Adjacency between surfels

- `SurfelAdjacency<dim>` specifies interior toward exterior or the reverse for each direction.

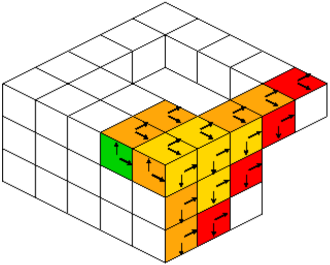
```
SurfelAdjacency<2> sAdj1( true ); // (4,8)
SurfelAdjacency<2> sAdj2( false ); // (8,4)
SurfelAdjacency<3> sAdj3( true ); // (6,18)
SurfelAdjacency<3> sAdj4( false ); // (18,6)
sAdj4.setAdjacency( 0, 1, true ); // hybrid
```

- `SurfelNeighborhood<KSpace>` computes adjacent surfels
  - initialized by `init( KSpace*, SurfelAdjacency<dim>, Cell )`
  - surfel can be changed `setSurfel`
  - get surrounding spels : `innerSpel`(), `innerAdjacentSpel`(Dimension, bool), . . .
  - get following surfels : `follower1`(Dimension, bool)
  - get adjacent surfels : `getAdjacentOnSpelSet`, . . .

Tracking surfels through surfel adjacencies I

```
Surfaces<KSpace>.trackClosedBoundary(
```

- SCellSet & surface,
- const KSpace & K,
- const SurfelAdjacency<KSpace : :dimension> & surfel_adj,
- const PointPredicate & pp,
- const SCell & start_surfel)



```
SCell b;  // current surfel
SCell bn; // neighboring surfel
SurfelNeighborhood<KSpace> SN;
SN.init( &K, &surfel_adj, start_surfel );
std::queue<SCell> qbels;
qbels.push( start_surfel );
surface.insert( start_surfel ); // output
while ( ! qbels.empty() ) { // For all pending bels
  b = qbels.front();
  qbels.pop();
  SN.setSurfel( b );
  for ( DirIterator q = K.sDirs( b ); q != 0; ++q ) {
    Dimension track_dir = *q;
    // One pass, look for direct orientation
    if ( SN.getAdjacentOnPointPredicate( bn, pp,
           track_dir, K.sDirect( b, track_dir ) ) )
    {
      if ( surface.find( bn ) == surface.end() )
      {
        surface.insert( bn );
        qbels.push( bn );
      }
    }
  } // end for
} // end while
```

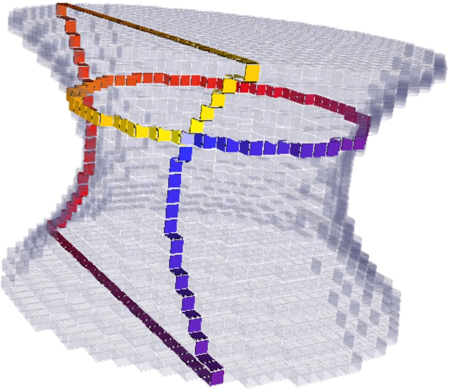
Helper class Surfaces

Provide methods for

- finding a bel (i.e. a surfel between inside/outside of object)
- track boundaries in nD (closed or not)
- track contours of 2D shapes
- track 2D slices of n shapes
- extract all contours of a 2D domain
- extract all boundaries of a nD shape
- computes the whole boundary of a nD shape by scanning

(with B. Kerautret)

Surface tracking example



Surface tracking snippet

```
// Extract an initial boundary cell
Z3i::SCell aCell = Surfaces<Z3i::KSpace>::findABel(
    ks, set3dPredicate);
// Extracting all boundary surfels connected to the
    initial one
Surfaces<Z3i::KSpace>::trackBoundary(
    vectBdrySCellALL, ks,SAdj, set3dPredicate,
    aCell );
// Extract the boundary contour associated to the
    initial surfel in its first direction
Surfaces<Z3i::KSpace>::track2DBoundary(
    vectBdrySCell, ks, *(ks.sDirs( aCell )), SAdj,
    set3dPredicate, aCell );
// Extract the bondary contour associated to the
    initial surfel in its second direction
Surfaces<Z3i::KSpace>::track2DBoundary(
    vectBdrySCell2, ks, *(++(ks.sDirs( aCell ))),
    SAdj, set3dPredicate, aCell );
```

Getting the contour of a digitized shape

```
// Digitizer
GaussDigitizer<Space,Shape> dig;
dig.attach( aShape ); // attaches the shape.
Vector vlow(-1,-1); Vector vup(1,1);
dig.init( aShape.getLowerBound()+vlow, aShape.
    getUpperBound()+vup, h );
Domain domain = dig.getDomain();
// Extracts shape boundary
SurfelAdjacency<KSpace::dimension> SAdj( true );
SCell bel = Surfaces<KSpace>::findABel( K, dig,
    10000 );
// Getting the consecutive surfels of the 2D
    boundary
std::vector<Point> points;
Surfaces<KSpace>::track2DBoundaryPoints( points, K,
     SAdj, dig, bel );
// Create GridCurve
GridCurve<KSpace> gridcurve;
gridcurve.initFromVector( points );
```

To go further

On-line user guide in DGtal documentation

- Topology Package
  - Digital topology and digital objects
  - Cellular grid space and topology, cells, digital surfaces

(nicely illustrated in 3D, thanks to B. Kerautret)

Next objectives

1. classical digital topology
   - other adjacencies
   - Adjacency = unoriented graph, create associated concepts
   - make everything faster with specialization (especially simpleness)
2. cubical cellular topology
   - cubical complexes, interior, closure
   - path, mapping (homotopy)
   - chains, boundary operator, cochains, coboundary
   - (co)homology
3. digital surface topology
   - digital surface concept, digital surface graph and cograph (umbrellas), digital surface map