Genericity as a Solution for Image Processing

Applications to Mathematical Morphology and Digital Topology

Roland Levillain, Thierry Géraud

EPITA Research and Development Laboratory (LRDE)

8/6/2012

Genericity as a Solution for Image Processing

A First Image Processing (IP) Algorithm

Abstract Definition

```
Data:
  I (an image), V (a value)

let D be the domain of I
for each x ∈ D do
  I(x) ← V
```

Specific Implementation

```
void fill(image& ima, unsigned char val)
{
  for (int r = 0; r < ima.nrows(); ++r)
    for (int c = 0; c < ima.ncols(); ++c)
      ima(r,c) = val;
}
```

→ General definition but non-reusable code.

A First Image Processing (IP) Algorithm

Abstract Definition

```
Data:
  I (an image), V (a value)

let D be the domain of I
for each x ∈ D do
  I(x) ← V
```

Specific Implementation

```
void fill(image& ima, unsigned char val)
{
  for (int r = 0; r < ima.nrows(); ++r)
    for (int c = 0; c < ima.ncols(); ++c)
      ima(r,c) = val;
}
```

→ General definition but non-reusable code.

A First Image Processing (IP) Algorithm

Abstract Definition

```
Data:
  I (an image), V (a value)

let D be the domain of I
for each x ∈ D do
  I(x) ← V
```

Specific Implementation

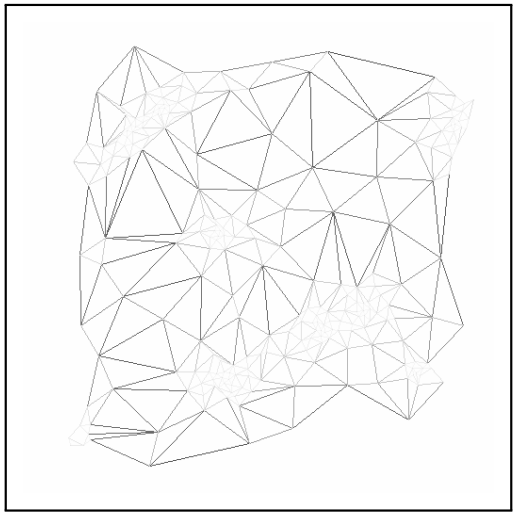
```
void fill(image& ima, unsigned char val)
{
  for (int r = 0; r < ima.nrows(); ++r)
    for (int c = 0; c < ima.ncols(); ++c)
      ima(r,c) = val;
}
```

→ General definition but non-reusable code.

Applying `fill` to Various Inputs





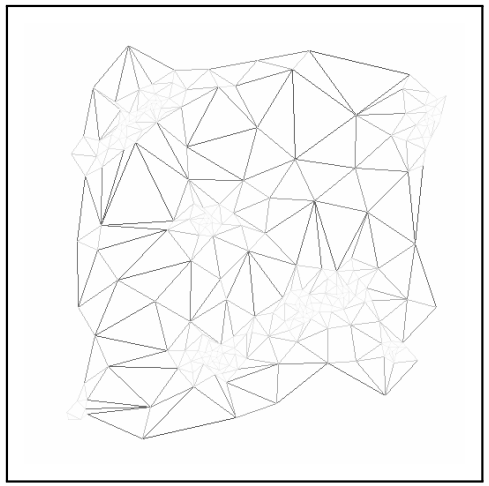
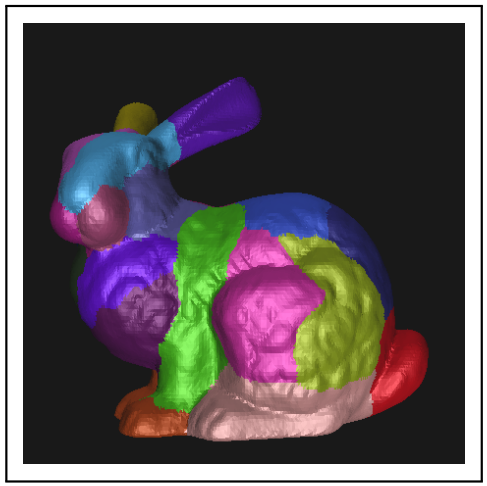



Genericity as a Solution for Image Processing

Context

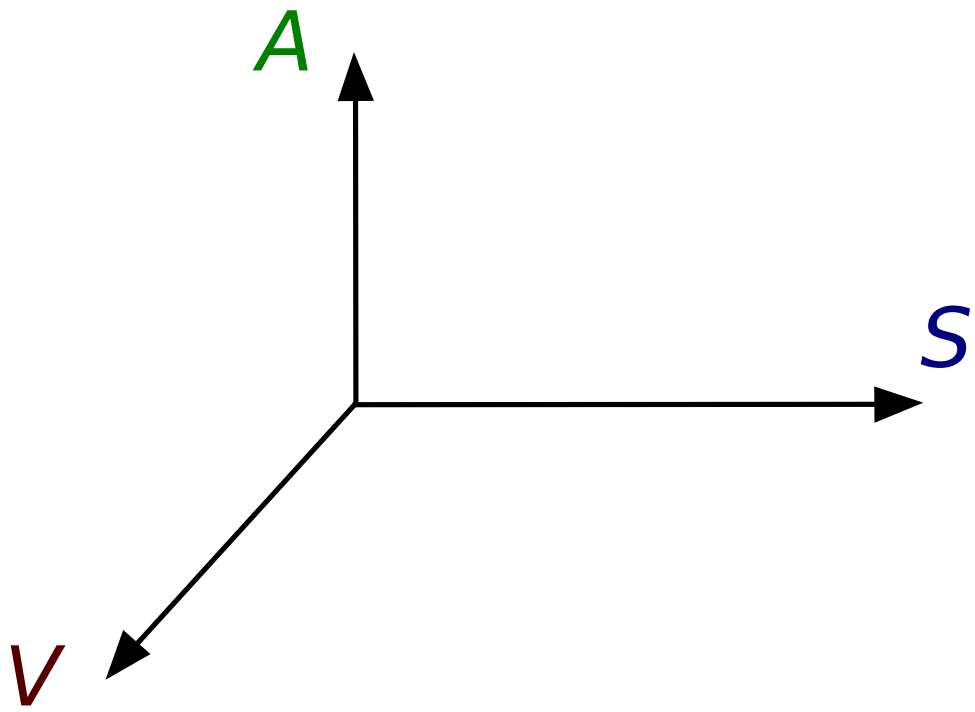
Diversity of Image Types I

- Classic decomposition of IP software :
  - *S* types of data structures.

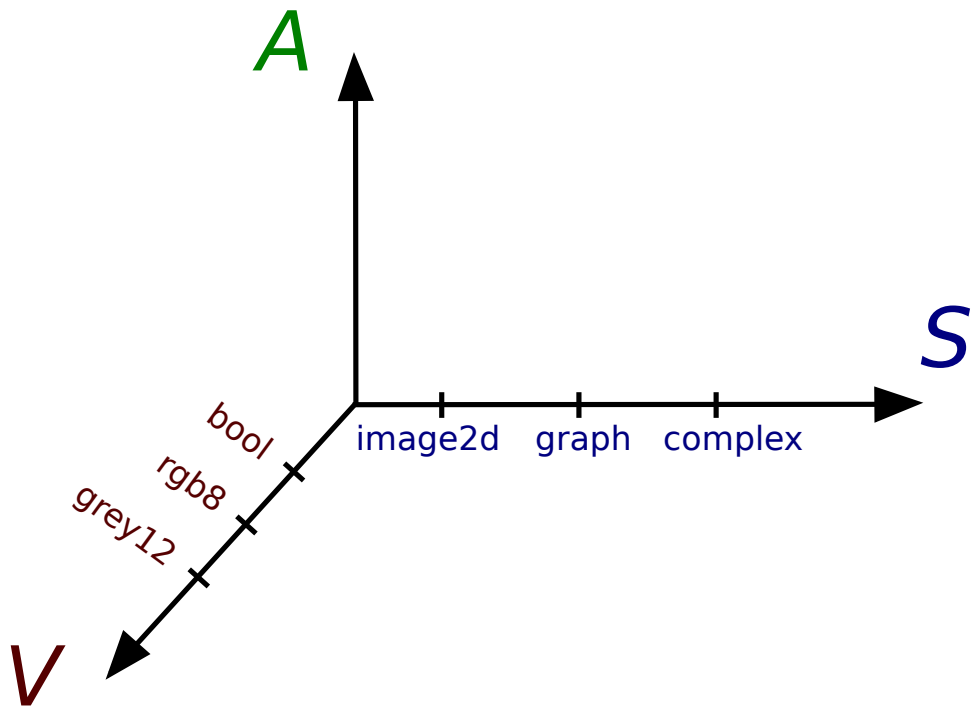
  - *V* types of “pixel” values.
  - *A* algorithms.

Diversity of Image Types II



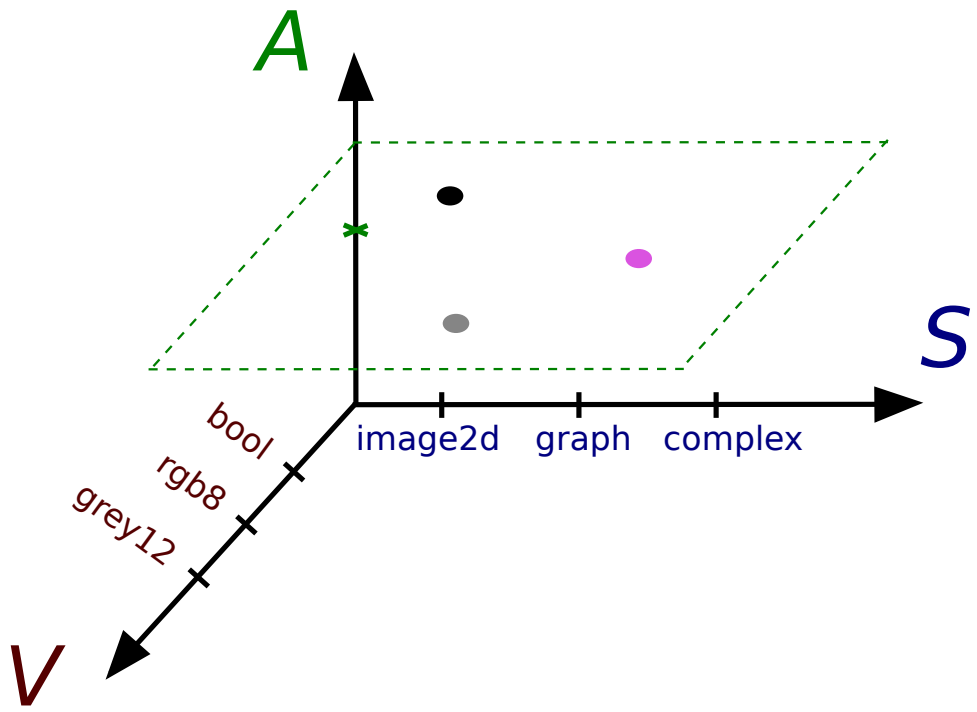
- High combinatorial complexity : $A \times S(\times V)$ implementations.
- How to uncouple axes ?

Diversity of Image Types II



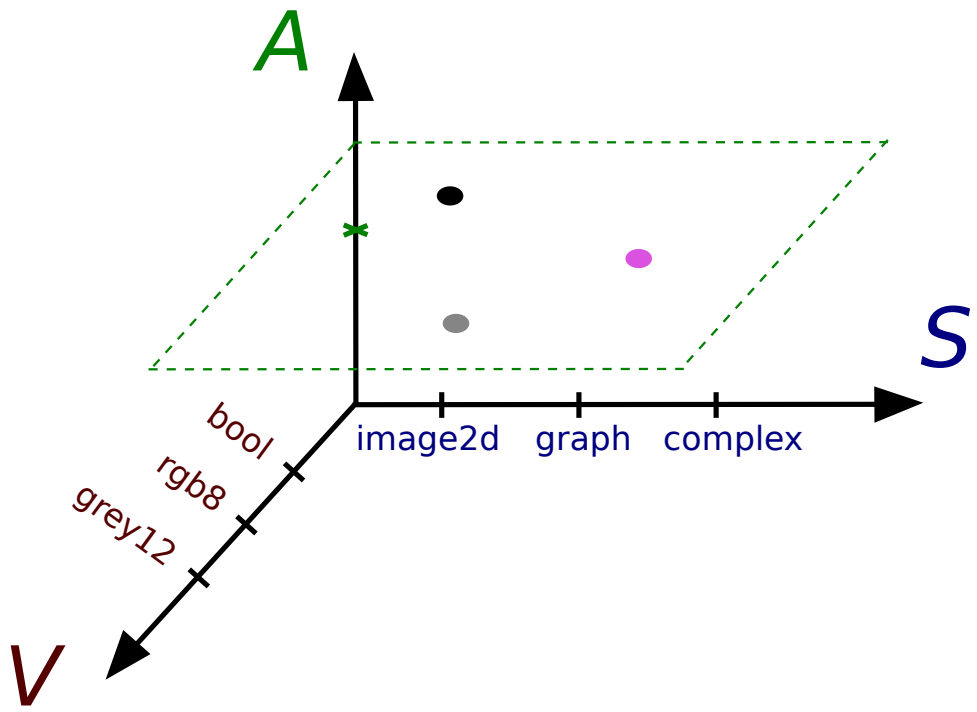
- High combinatorial complexity : $A \times S(\times V)$ implementations.
- How to uncouple axes ?

Diversity of Image Types II



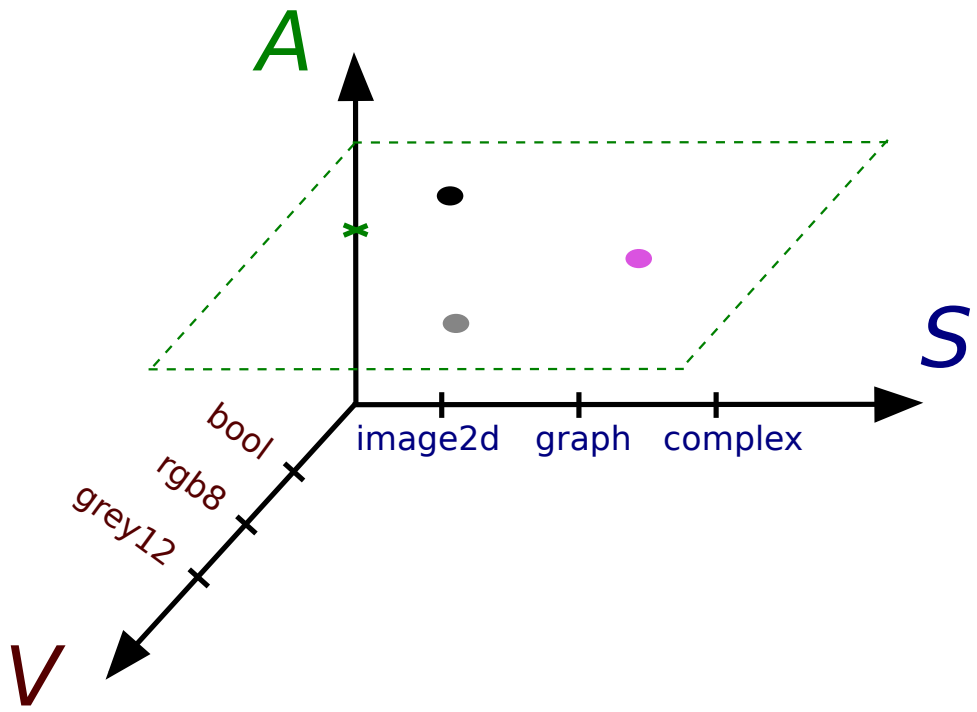
- High combinatorial complexity : $A \times S(\times V)$ implementations.
- How to uncouple axes ?

Diversity of Image Types II



- High combinatorial complexity : $A \times S(\times V)$ implementations.
- How to uncouple axes ?

Diversity of Image Types II



- High combinatorial complexity : $A \times S(\times V)$ implementations.
- How to uncouple axes ?

Our Proposal : The Olena Project

- Available in the C++ library in 2012 :
  - 7 primary image structures.
  - 17 image structures modifications (morphers).
  - Much more in beta versions.
- In practice :
  - > 100 concrete image types commonly used in Olena.
    vs
  - < 10 image types in most classic libraries.
- Objectives :
  - → Scalability.
  - → No a priori limitations.

→ Only possible with a new programming paradigm.

Our Proposal : The Olena Project

- Available in the C++ library in 2012 :
  - 7 primary image structures.
  - 17 image structures modifications (morphers).
  - Much more in beta versions.
- In practice :
  - > 100 concrete image types commonly used in Olena.
    vs
  - < 10 image types in most classic libraries.
- Objectives :
  - → Scalability.
  - → No a priori limitations.

→ Only possible with a new programming paradigm.

Our Proposal : The Olena Project

- Available in the C++ library in 2012 :
  - 7 primary image structures.
  - 17 image structures modifications (morphers).
  - Much more in beta versions.
- In practice :
  - $>$ 100 concrete image types commonly used in Olena.
    vs
  - $<$ 10 image types in most classic libraries.
- Objectives :
  - $\rightarrow$ Scalability.
  - $\rightarrow$ No a priori limitations.

$\rightarrow$ Only possible with a new programming paradigm.

Our Proposal : The Olena Project

- Available in the C++ library in 2012 :
  - 7 primary image structures.
  - 17 image structures modifications (morphers).
  - Much more in beta versions.
- In practice :
  - $>$ 100 concrete image types commonly used in Olena.
    vs
  - $<$ 10 image types in most classic libraries.
- Objectives :
  - $\rightarrow$ Scalability.
  - $\rightarrow$ No a priori limitations.

$\rightarrow$ Only possible with a new programming paradigm.

Handling Multiple Input Types

- Existing solutions :

Paradigms	Type Checking	Code Complexity	Efficiency	Explicit Abstractions	One impl per Algorithm
Code Duplication	✓	✗	✓	✗	✗
Generalization	✗	=	=	✗	✓
Object-Oriented Programming (OOP)	=	✓	✗	✓	✓
Generic Programming (GP)	✓	✓	✓	=	✓

- Generic Programming (GP) and IP : VIGRA (1998), Olena (1998), ITK (1999), ImLib3D, GIL (2006), Yayi (2009), Morph-M, DGtal (2010), . . .

→ Idea : mixing OOP and GP.

Mixing Object-Oriented Programming and Generic Programming

- Benefits from their respective advantages :

  reusability + abstraction + efficiency

- Getting rid of their drawbacks :

  run-time overhead, lack of abstraction mechanism

→ A Static C++ Object Oriented Programming (SCOOP) paradigm.

Nicolas Burrus, Alexandre Duret-Lutz, Thierry Géraud, David Lesage, and Raphaël Poss.
A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming.
In *Proceedings of the Workshop on Multiple Paradigm with Object-Oriented Languages (MPOOL)*, 2003.

Mixing Object-Oriented Programming and Generic Programming

- Benefits from their respective advantages :

  reusability + abstraction + efficiency

- Getting rid of their drawbacks :

  run-time overhead, lack of abstraction mechanism

→ A Static C++ Object Oriented Programming (SCOOP) paradigm.

Nicolas Burrus, Alexandre Duret-Lutz, Thierry Géraud, David Lesage, and Raphaël Poss.
A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming.
In *Proceedings of the Workshop on Multiple Paradigm with Object-Oriented Languages (MPOOL)*, 2003.

Mixing Object-Oriented Programming and Generic Programming

- Benefits from their respective advantages :

  reusability + abstraction + efficiency

- Getting rid of their drawbacks :

  run-time overhead, lack of abstraction mechanism

→ A Static C++ Object Oriented Programming (SCOOP) paradigm.

Nicolas Burrus, Alexandre Duret-Lutz, Thierry Géraud, David Lesage, and Raphaël Poss.
A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming.
In *Proceedings of the Workshop on Multiple Paradigm with Object-Oriented Languages (MPOOL)*, 2003.

Mixing Object-Oriented Programming and Generic Programming

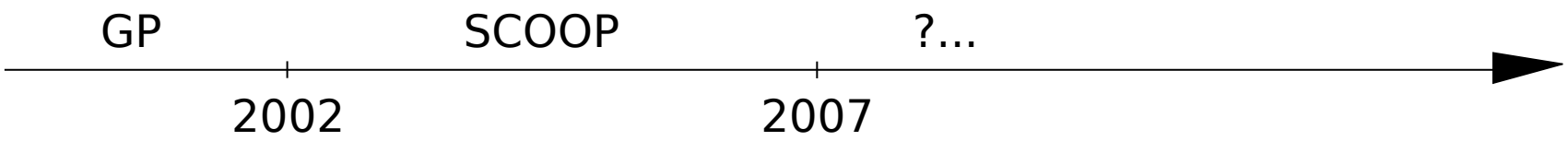
- Benefits from their respective advantages :

  reusability + abstraction + efficiency

- Getting rid of their drawbacks :

  run-time overhead, lack of abstraction mechanism

→ A Static C++ Object Oriented Programming (SCOOP) paradigm.



Nicolas Burrus, Alexandre Duret-Lutz, Thierry Géraud, David Lesage, and Raphaël Poss.
A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming.
In *Proceedings of the Workshop on Multiple Paradigm with Object-Oriented Languages (MPOOL)*, 2003.

Thanks to SCOOP

A general and reusable algorithm ⇒ a single generic routine.

Abstract Definition

```
Data:
  I (an image),
  B (a structuring element)
Result: O (an image)

initialize O so that it has
  the same structure as I
let D be the domain of I
let V be the value type of I
for each x ∈ D do
  O(x) ← inf(V)
  for each h in B do
    O(x) ← sup(O(x), I(x + h))
```

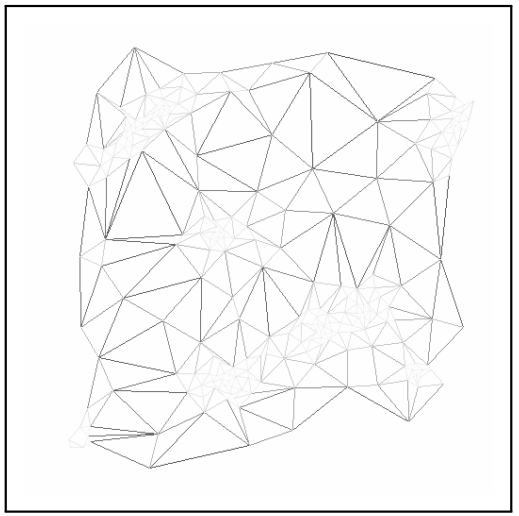
Generic Implementation

```
template <typename I, typename W>
I dilation(const Image<I>& input,
           const Window<W>& win)
{
  // Output image.
  I output; initialize(output, input);
  // Iterator on input's domain.
  typedef I::iter input.domain());
  // Iterator on window.
  typedef W::iter q(win, p);
  for_all(p)
    {
      output(p) = I::value::inf;
      for_all(q) if (input.has(q))
        output(p) = sup(output(p), input(q));
    }
  return output;
}
```

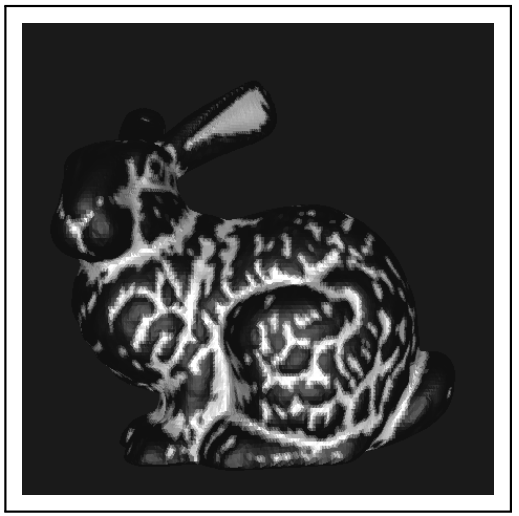
SCOOP Applied : Mathematical Morphology

Segmentation based on a Watershed Transform

OK



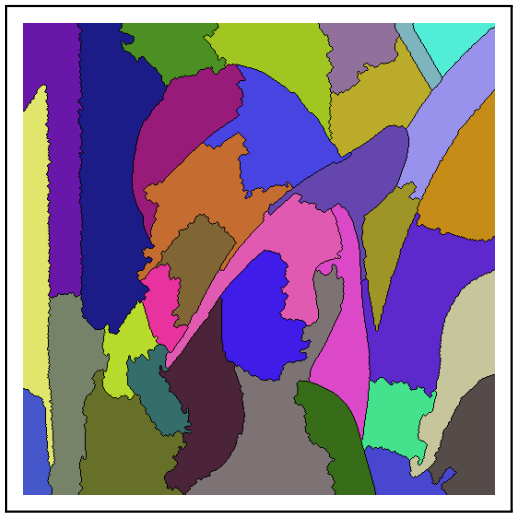
≈ OK



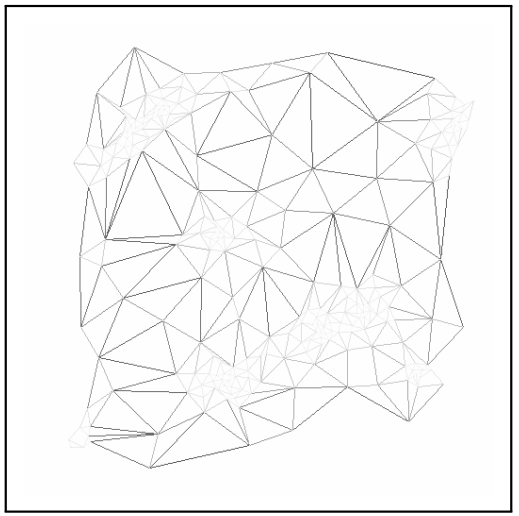
KO !

SCOOP Applied : Mathematical Morphology

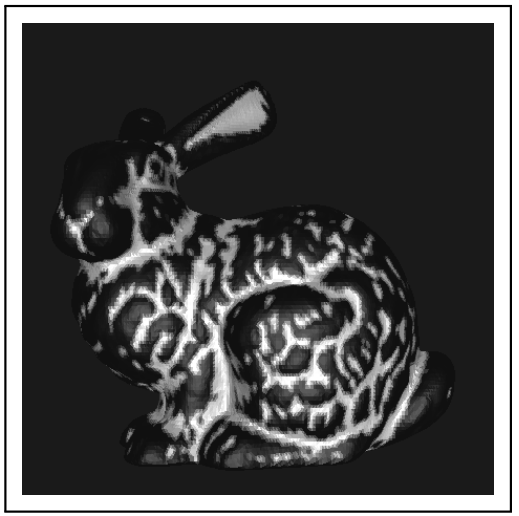
Segmentation based on a Watershed Transform



OK



≈ OK

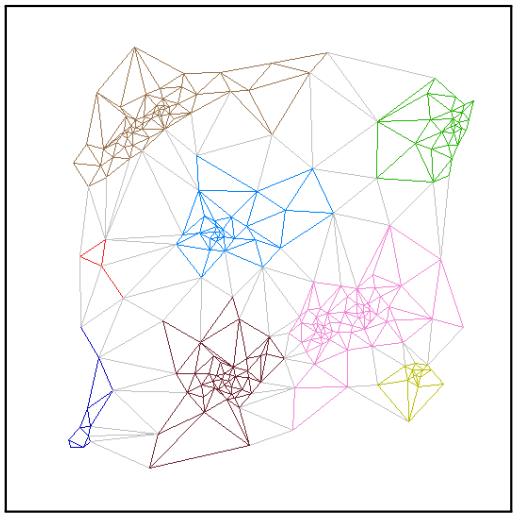


KO !

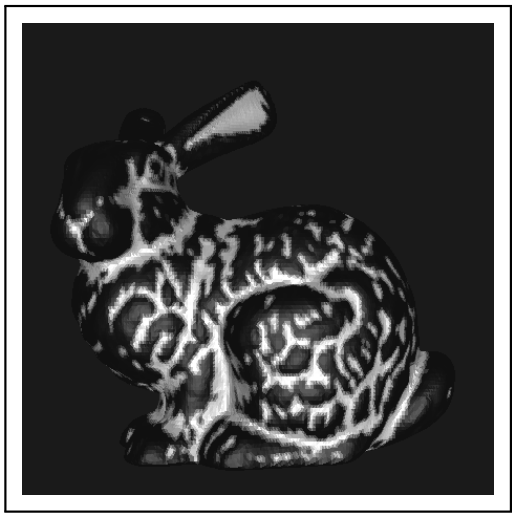
SCOOP Applied : Mathematical Morphology

Segmentation based on a Watershed Transform

OK



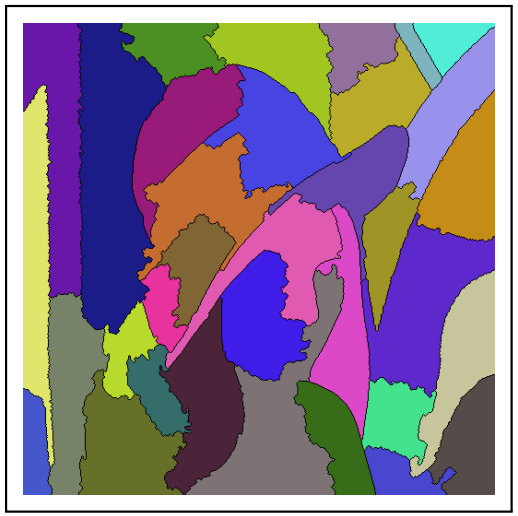
≈ OK



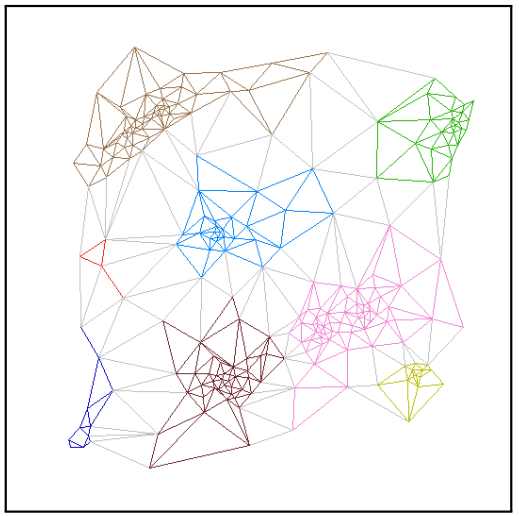
KO !

SCOOP Applied : Mathematical Morphology

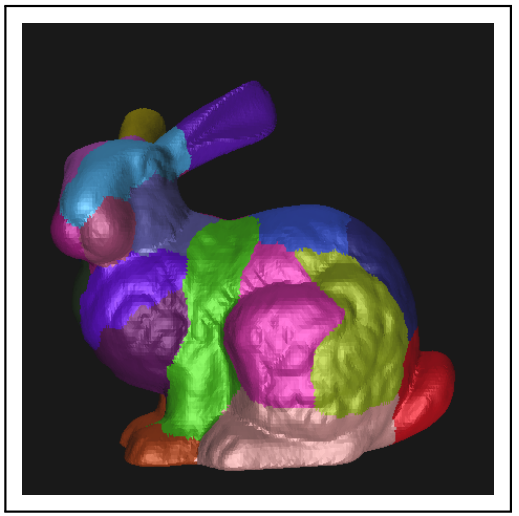
Segmentation based on a Watershed Transform



OK



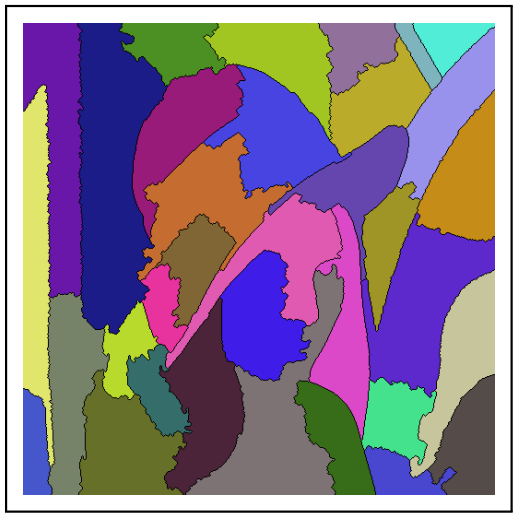
≈ OK



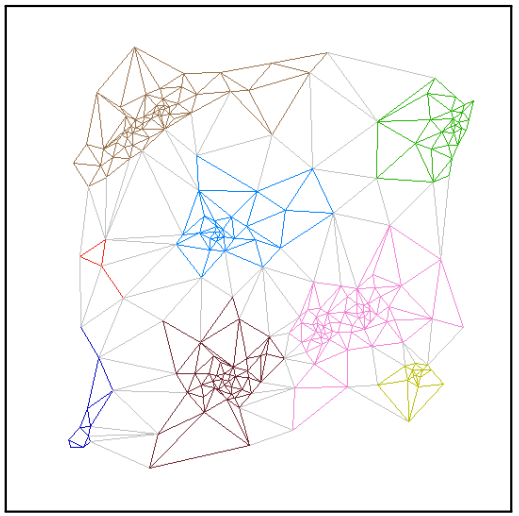
KO !

SCOOP Applied : Mathematical Morphology

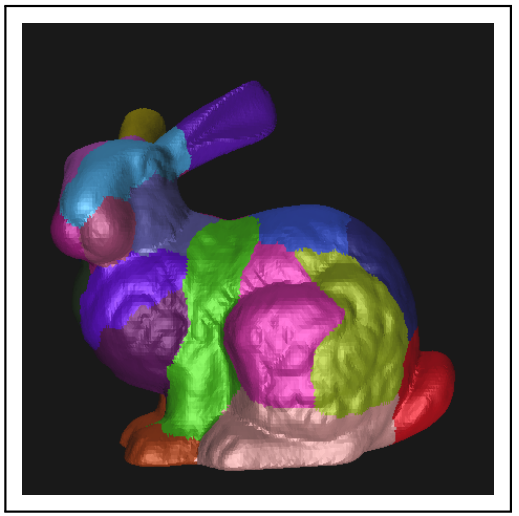
Segmentation based on a Watershed Transform



OK



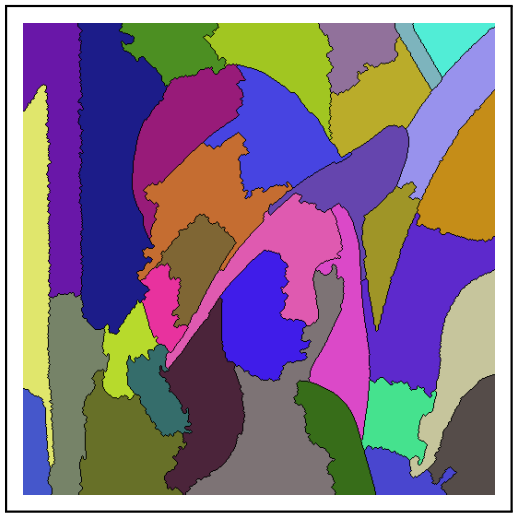
≈ OK



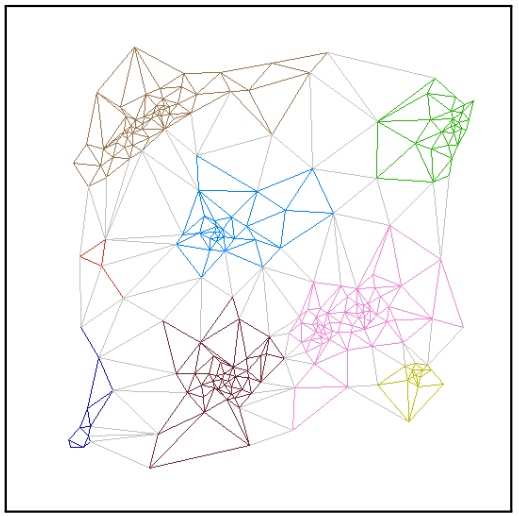
KO !

SCOOP Applied : Mathematical Morphology

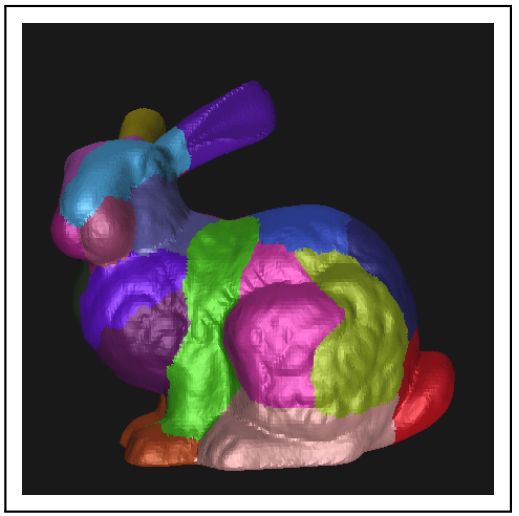
Segmentation based on a Watershed Transform



OK



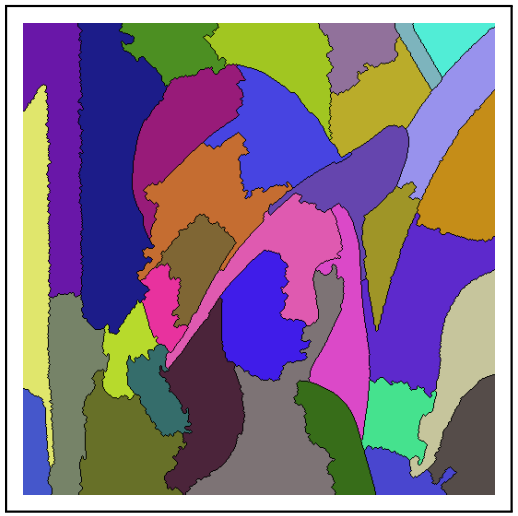
≈ OK



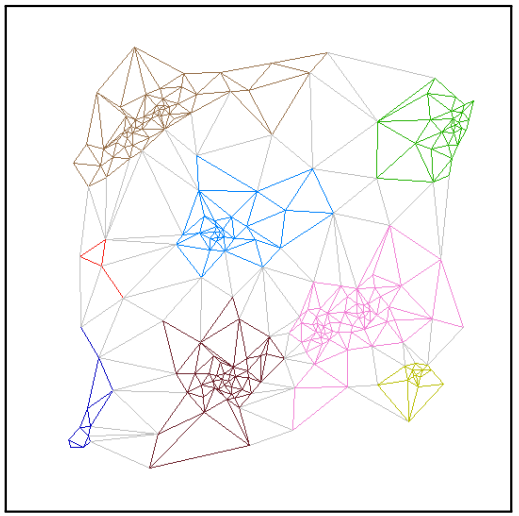
KO !

SCOOP Applied : Mathematical Morphology

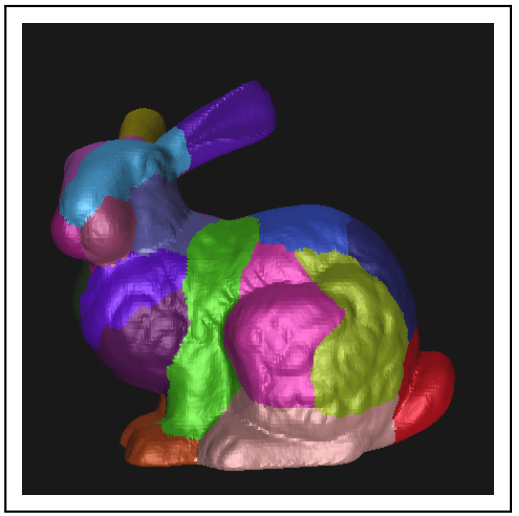
Segmentation based on a Watershed Transform



OK



≈ OK

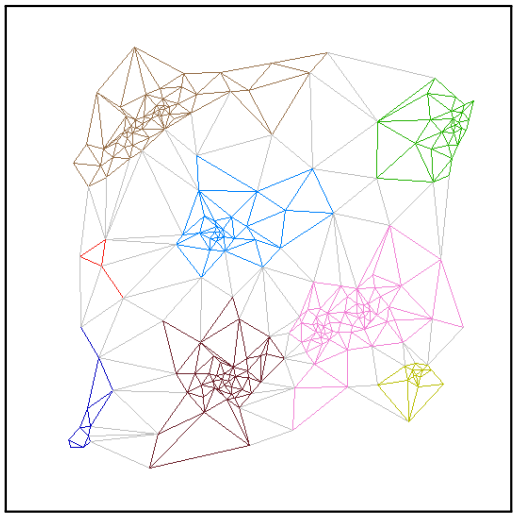


KO !

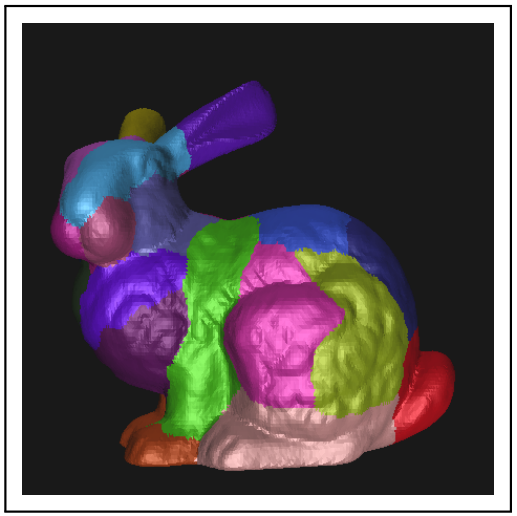
SCOOP Applied : Mathematical Morphology

Segmentation based on a Watershed Transform

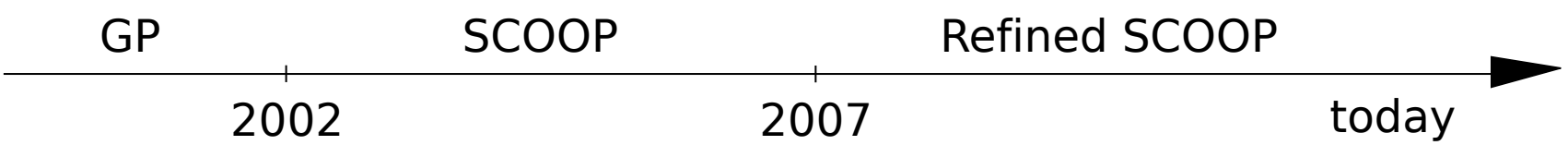
OK



≈ OK



KO !



Genericity Through A New Programming Paradigm

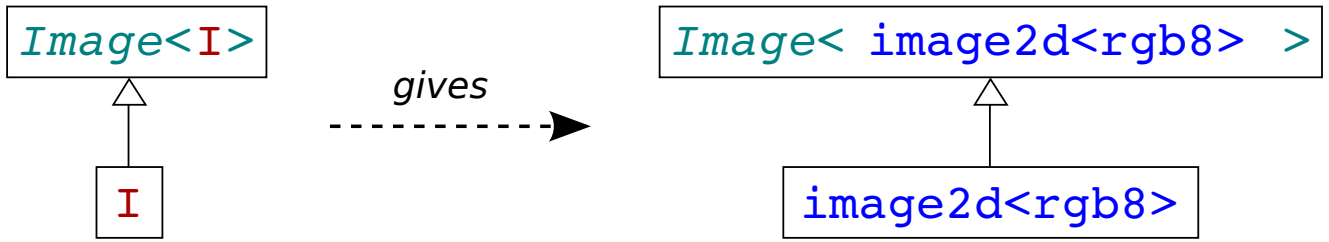
Definition of Image Types

- Class hierarchies : abstractions and concrete classes.
- Static hierarchies : Curiously Recurring Template Pattern (CRTP) :
- Concrete class : with implementation.
- Abstraction (`Image`) : hollow class (no interface).

James O. Coplien.
Curiously recurring template patterns.
In Stanley B. Lippman, editor, *C++ Gems*. Cambridge Press University & Sigs Books, 1996.

Definition of Image Types

- Class hierarchies : abstractions and concrete classes.
- Static hierarchies : Curiously Recurring Template Pattern (CRTP) :

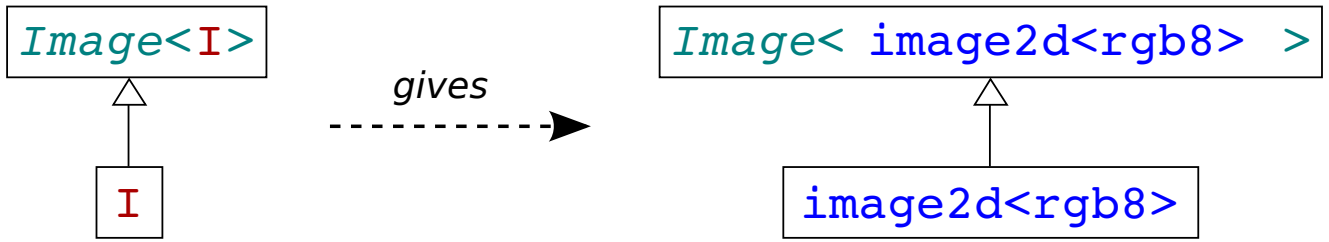


- Concrete class : with implementation.
- Abstraction (`Image`) : hollow class (no interface).

James O. Coplien.
Curiously recurring template patterns.
In Stanley B. Lippman, editor, *C++ Gems*. Cambridge Press University & Sigs Books, 1996.

Definition of Image Types

- Class hierarchies : abstractions and concrete classes.
- Static hierarchies : Curiously Recurring Template Pattern (CRTP) :

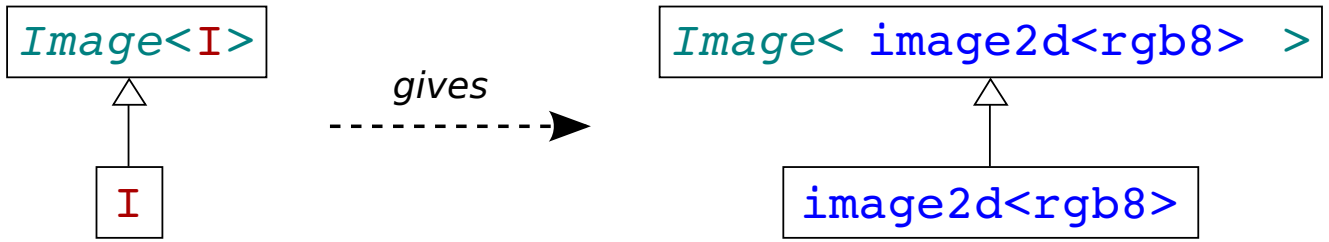


- Concrete class : with implementation.
- Abstraction (`Image`) : hollow class (no interface).

James O. Coplien.
Curiously recurring template patterns.
In Stanley B. Lippman, editor, *C++ Gems*. Cambridge Press University & Sigs Books, 1996.

Definition of Image Types

- Class hierarchies : abstractions and concrete classes.
- Static hierarchies : Curiously Recurring Template Pattern (CRTP) :



- Concrete class : with implementation.
- Abstraction (`Image`) : hollow class (no interface).

James O. Coplien.
Curiously recurring template patterns.
In Stanley B. Lippman, editor, *C++ Gems*. Cambridge Press University & Sigs Books, 1996.

Benefits of the Refined Paradigm

- Features preserved from the initial paradigm :
  - Materialized abstractions (named types).
  - Strong type checking
- New characteristics :
  - Simpler and more concise.
  - Finer and more precise thanks to properties attached to data types.
- Enables a new powerful construction : morphers.

Thierry Géraud and Roland Levillain.

Semantics-driven genericity : A sequel to the static C++ object-oriented programming paradigm (SCOOP 2).
In *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL)*, Paphos, Cyprus, July 2008.

Roland Levillain.

Towards a Software Architecture for Generic Image Processing.
PhD thesis, Université Paris-Est, Marne-la-Vallée, France, November 2011.

Benefits of the Refined Paradigm

- Features preserved from the initial paradigm :
  - Materialized abstractions (named types).
  - Strong type checking
- New characteristics :
  - Simpler and more concise.
  - Finer and more precise thanks to properties attached to data types.
- Enables a new powerful construction : morphers.

Thierry Géraud and Roland Levillain.

Semantics-driven genericity : A sequel to the static C++ object-oriented programming paradigm (SCOOP 2).
In *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL)*, Paphos, Cyprus, July 2008.

Roland Levillain.

Towards a Software Architecture for Generic Image Processing.
PhD thesis, Université Paris-Est, Marne-la-Vallée, France, November 2011.

Benefits of the Refined Paradigm

- Features preserved from the initial paradigm :
  - Materialized abstractions (named types).
  - Strong type checking
- New characteristics :
  - Simpler and more concise.
  - Finer and more precise thanks to properties attached to data types.
- Enables a new powerful construction : morphers.

Thierry Géraud and Roland Levillain.

Semantics-driven genericity : A sequel to the static C++ object-oriented programming paradigm (SCOOP 2).
In *Proceedings of the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL)*, Paphos, Cyprus, July 2008.

Roland Levillain.

Towards a Software Architecture for Generic Image Processing.
PhD thesis, Université Paris-Est, Marne-la-Vallée, France, November 2011.

Genericity Applied to Image Processing

A Generic Breadth-First Thinning Algorithm

Abstract Definition

```
Data : E, X ⊆ E, N (neighborhood),
is_simple (a function saying whether a point is
           simple),
detach (a routine detaching a point from X),
constraint (a function representing a
            constraint)
Result : X
P ← {p ∈ X | is_simple(p, X) }
while P ≠ ∅ do
  S ← ∅
  for each p ∈ P do
    if constraint(p) and is_simple(p, X) then
      X ← detach(X, p)
      for each n ∈ N(p) ∩ X do
        S ← S ∪ {n}
  P ← ∅
  for each p ∈ S do
    if is_simple(p, X) then P ← P ∪ {p}
```

Generic Implementation

```
template <typename I, typename N, typename F,
          typename G, typename H>
I breadth_first_thinning (const Image<I>& input,
   const Neighborhood<N>& nbh, Function_v2b<F>& is_simple,
   G& detach, const Function_v2b<H>& constraint) {
  // ...
  mln_piter(I) p(output.domain());
  for_all(p)
    if (output(p) && constraint(p) && is_simple(p))
      { queue.push(p); in_queue(p) = true; }
  while (!queue.is_empty()) {
    psite p = queue.pop_front(); in_queue(p) = false;
    if (output(p) && constraint(p) && is_simple(p)) {
      detach(p);
      mln_niter(N) n(nbh, p);
      for_all(n)
        if (output.domain().has(n) && !in_queue(n)
            && output(n) && constraint(n) && is_simple(n))
          { queue.push(n); in_queue(n) = true; }
    }
  }
  return output;
}
```

R. Levillain, Th. Géraud, and L. Najman.
Writing reusable digital geometry algorithms in a generic image processing framework.
In *WADGMM*, Istanbul, Turkey, August 2010.

A Generic Breadth-First Thinning Algorithm

Abstract Definition

```
Data : E, X ⊆ E, N (neighborhood),
is_simple (a function saying whether a point is
           simple),
detach (a routine detaching a point from X),
constraint (a function representing a
            constraint)
Result : X
P ← {p ∈ X | is_simple(p, X) }
while P ≠ ∅ do
  S ← ∅
  for each p ∈ P do
    if constraint(p) and is_simple(p, X) then
      X ← detach(X, p)
      for each n ∈ N(p) ∩ X do
        S ← S ∪ {n}
  P ← ∅
  for each p ∈ S do
    if is_simple(p, X) then P ← P ∪ {p}
```

R. Levillain, Th. Géraud, and L. Najman.
Writing reusable digital geometry algorithms in a generic image processing framework.
In *WADGMM*, Istanbul, Turkey, August 2010.

Generic Implementation

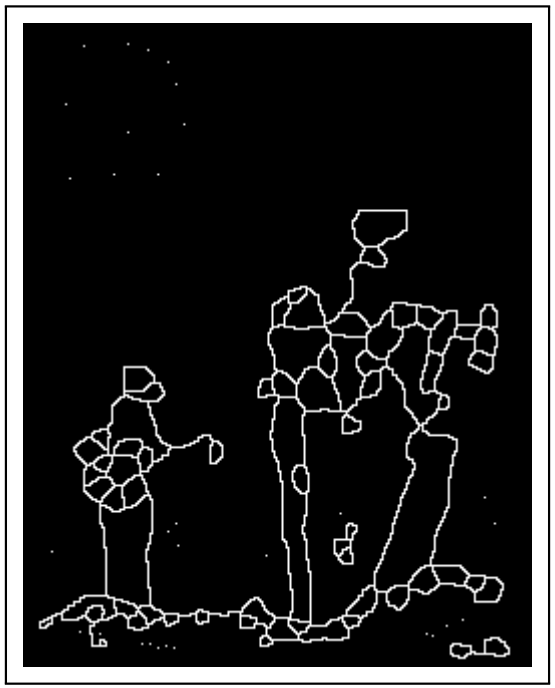
```
template <typename I, typename N, typename F,
          typename G, typename H>
I breadth_first_thinning (const Image<I>& input,
   const Neighborhood<N>& nbh, Function_v2b<F>& is_simple,
   G& detach, const Function_v2b<H>& constraint) {
  // ...
  mln_piter(I) p(output.domain());
  for_all(p)
    if (output(p) && constraint(p) && is_simple(p))
      { queue.push(p); in_queue(p) = true; }
  while (!queue.is_empty()) {
    psite p = queue.pop_front(); in_queue(p) = false;
    if (output(p) && constraint(p) && is_simple(p)) {
      detach(p);
      mln_niter(N) n(nbh, p);
      for_all(n)
        if (output.domain().has(n) && !in_queue(n)
            && output(n) && constraint(n) && is_simple(n))
        { queue.push(n); in_queue(n) = true; }
    }
  }
  return output;
}
```

Skeletons by Generic Thinning I

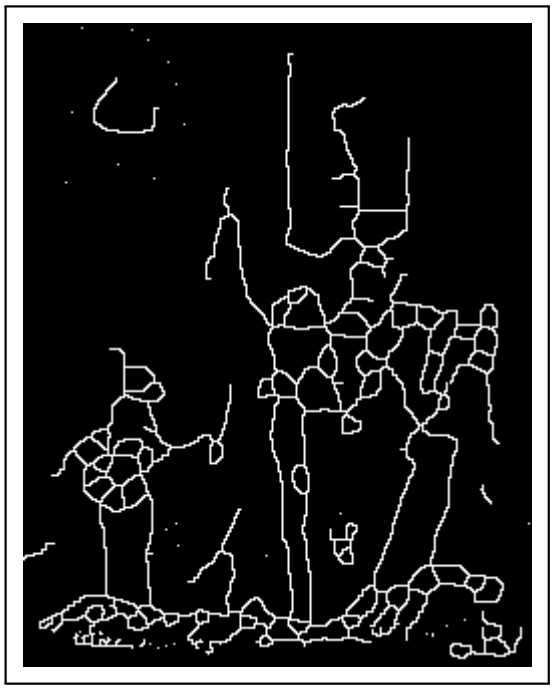
2D image on a square grid

Input.

Ultimate skeleton.



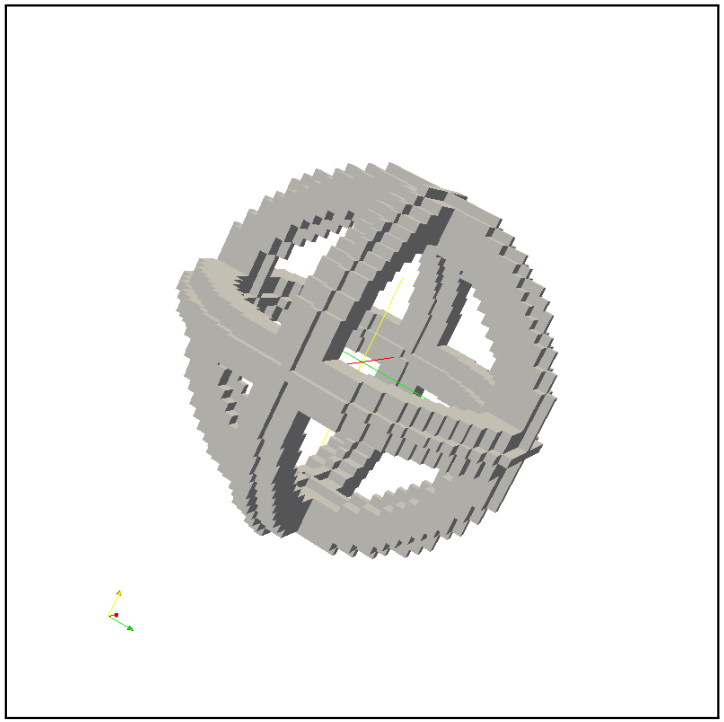
Skeleton preserving end points.



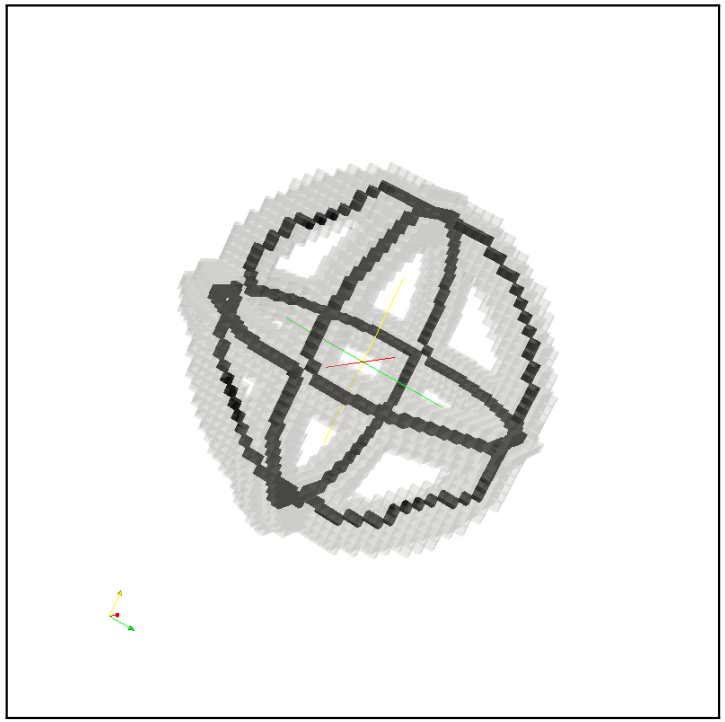
Skeletons by Generic Thinning II

3D volume on a cubic grid

Input.



Ultimate skeleton.



Skeletons by Generic Thinning III

2D image on a square grid

Input.

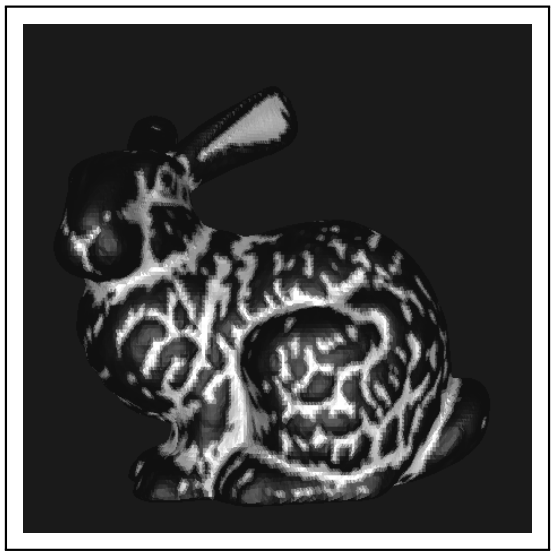
Gray-level skeleton.

Skeletons by Generic Thinning IV (continued)

Surface mesh (triangles only)

Mesh.

Curvature.



Simplicial 2-complex.

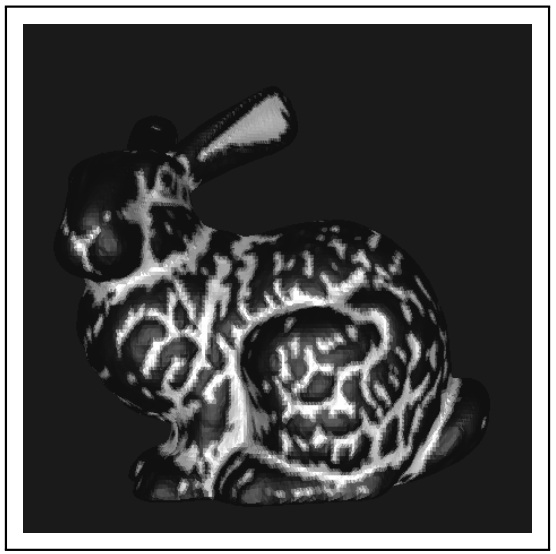
Michel Couprie and Gilles Bertrand.
New characterizations of simple points in 2D, 3D, and 4D discrete spaces.
IEEE Transactions on Pattern Analysis and Machine Intelligence, 31(4) :637–648, April 2009.

Skeletons by Generic Thinning IV (continued)

Surface mesh (triangles only)

Mesh.

Curvature.



Skeleton on 2-faces.

?

Simplicial 2-complex.

How to restrict the domain to triangles ?

Michel Couprie and Gilles Bertrand.
New characterizations of simple points in 2D, 3D, and 4D discrete spaces.
IEEE Transactions on Pattern Analysis and Machine Intelligence, 31(4) :637–648, April 2009.

Morphers

Handling Variations in Algorithms I

How to use a Region of Interest in an algorithm ?

Generic Morphological Dilation

```
template <typename I, typename W>
I dilation(const Image<I>& input_, const Window<W>& win_)
{

  // ...
  for_all(p)                                  // ∀ p ∈ ima's domain.
    {
      accu::supremum<mln_value(I)> sup;
      for_all(q) if (input.has(q))
        sup.take(input(q));
      output(p) = sup.to_result();
    }
  return output;
}
```

Handling Variations in Algorithms II

Duplicate and rewrite the algorithm ?

Generic Morphological Dilation on a Subset

```
template <typename I, typename W, typename J>
I dilation(const Image<I>& input_, const Window<W>& win_,
           const Image<J>& mask_)
{
  // ...
  for_all(p) if (mask.has(p) && mask(p)) // ∀ p ∈ ima's domain ∩ mask.
    {
      accu::supremum<mln_value(I)> sup;
      for_all(q) if (input.has(q) && mask.has(q) && mask(q))
        sup.take(input(q));
      output(p) = sup.to_result();
    }
  return output;
}
```

Morphers I

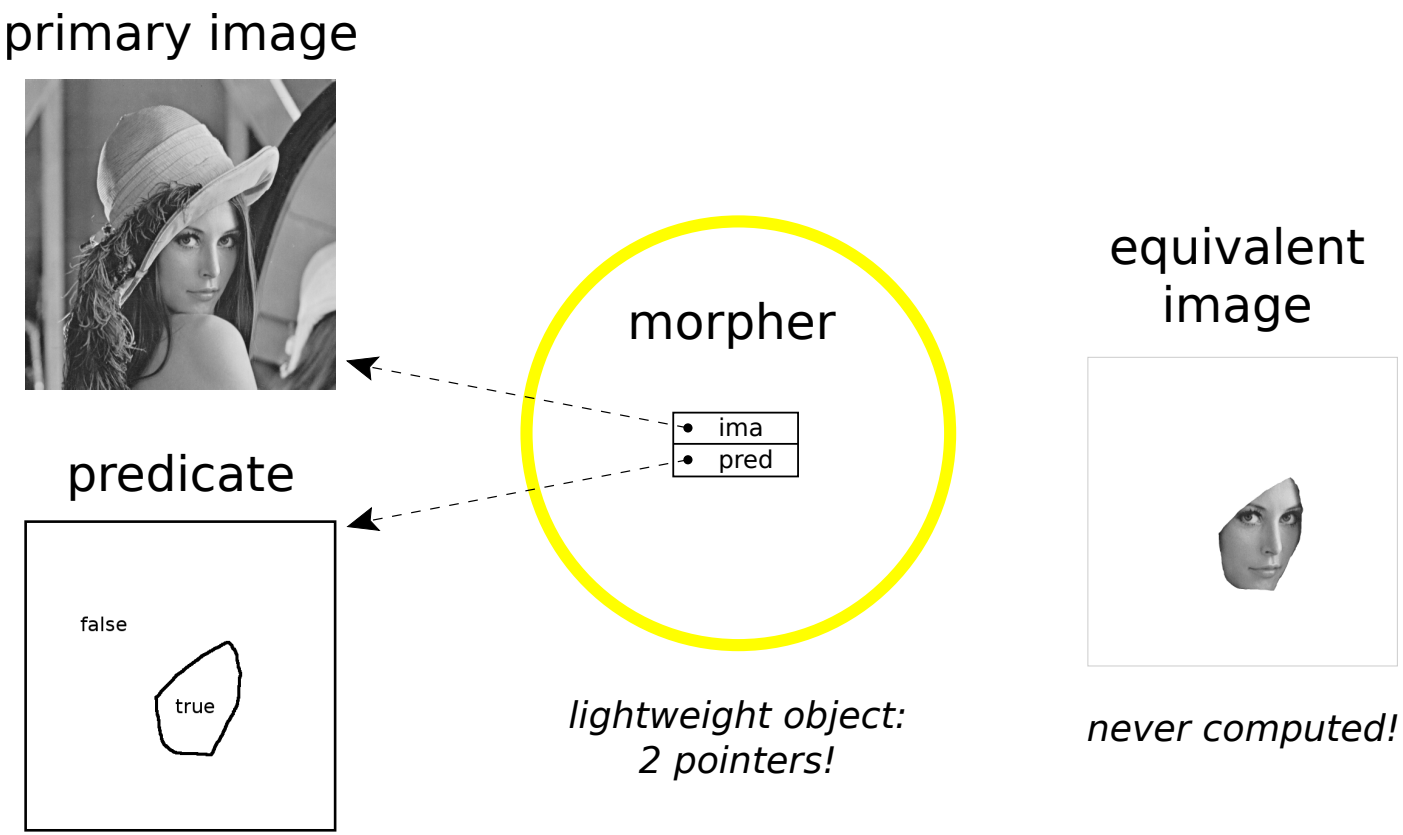
Definition (Morpher)

A morpher is a pseudo-modification of an existing image.

Morphers I

Definition (Morpher)

A morpher is a pseudo-modification of an existing image.



Morphers II

- Light-weight.
- Generic themselves.

Example (Restriction via a predicate)

∀ Image 'ima' of type I ∀ Function 'pred' of type F :

```
image_if<I, F> restricted_ima = ima | pred;
```

Morphers II

- Light-weight.
- Generic themselves.

Example (Restriction via a predicate)

∀ `Image` ‘`ima`’ of type `I` ∀ `Function` ‘`pred`’ of type `F` :

```
image_if<I, F> restricted_ima = ima | pred;
```

Morphers II

- Light-weight.
- Generic themselves.

Example (Restriction via a predicate)

∀ `Image` ‘`ima`’ of type `I` ∀ `Function` ‘`pred`’ of type `F` :

```
image_if<I, F> restricted_ima = ima | pred;
```

Categories of Image Morphers

1. Modifying an image's domain.

Example (Adding an "outer domain" extension)

Providing values outside the initial domain.

2. Modifying an image's values.

Example (Image of vectors seen through the L^2 norm.)

Reading values On-the-fly computation of the L^2 norm.

Writing values Rescaling vectors.

3. Adding an extra behavior to an image.

Example (Addition of an interactive visualization)

Following changes made by an algorithm.

Categories of Image Morphers

1. Modifying an image's domain.

Example (Adding an "outer domain" extension)

Providing values outside the initial domain.

2. Modifying an image's values.

Example (Image of vectors seen through the L^2 norm.)

Reading values On-the-fly computation of the L^2 norm.

Writing values Rescaling vectors.

3. Adding an extra behavior to an image.

Example (Addition of an interactive visualization)

Following changes made by an algorithm.

Categories of Image Morphers

1. Modifying an image's domain.

Example (Adding an "outer domain" extension)

Providing values outside the initial domain.

2. Modifying an image's values.

Example (Image of vectors seen through the L^2 norm.)

Reading values On-the-fly computation of the L^2 norm.

Writing values Rescaling vectors.

3. Adding an extra behavior to an image.

Example (Addition of an interactive visualization)

Following changes made by an algorithm.

Morphers Examples I

Variations on a simple algorithm : `fill`.

Generic Filling Algorithm

```
template <typename I, typename V>
I fill(Image<I>& ima_, const V& val)
{
  I& ima = exact(ima);
  mln_piter(p) p(ima.domain());
  for_all(p)
    ima(p) = val;
}
```

- Algorithm also generic w.r.t. morphed images.

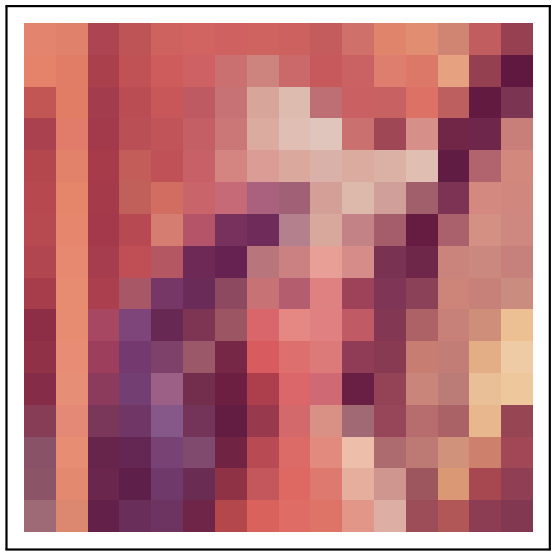
→ A lot of combinations.

Morphers Examples II

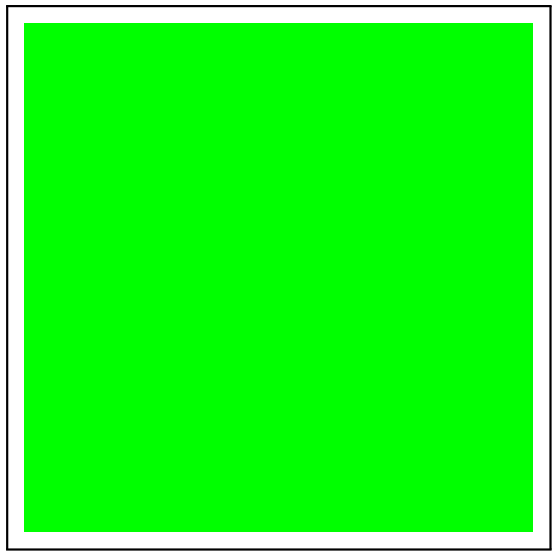
Normal (“unmorphed”) case

```
image2d<rgb8> lena = load("lena.png");
rgb8 green(0, 255, 0);
fill(lena, green);
```

“lena” before.



“lena” after.

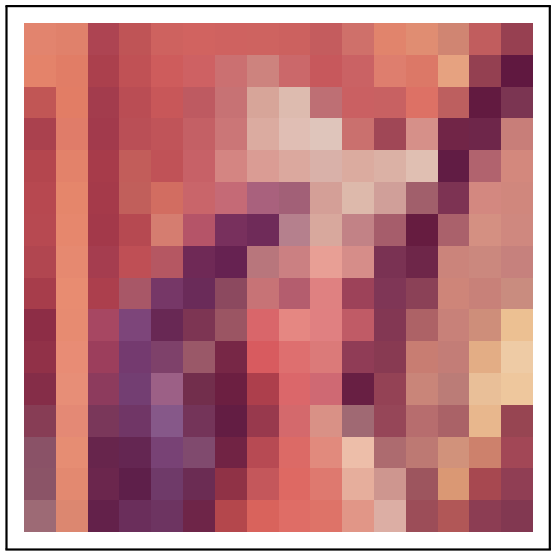


Morphers Examples III

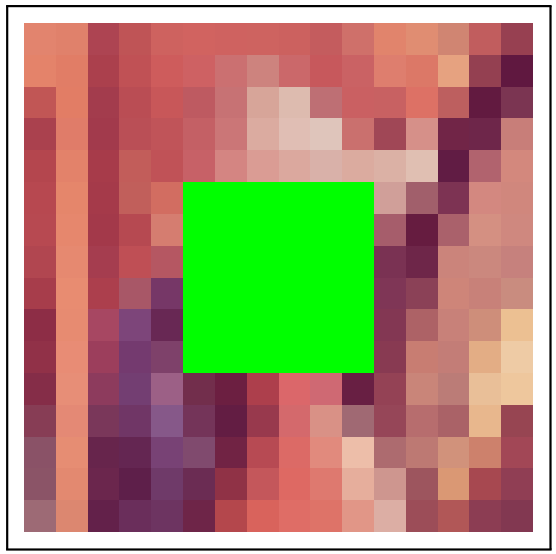
Restriction to a subset

```
box2d roi(5,5, 10,10);          // Region Of Interest (ROI).
fill(lena | roi,                // 'lena' restricted to 'roi'.
     green);
```

"lena" before.



"lena" after.

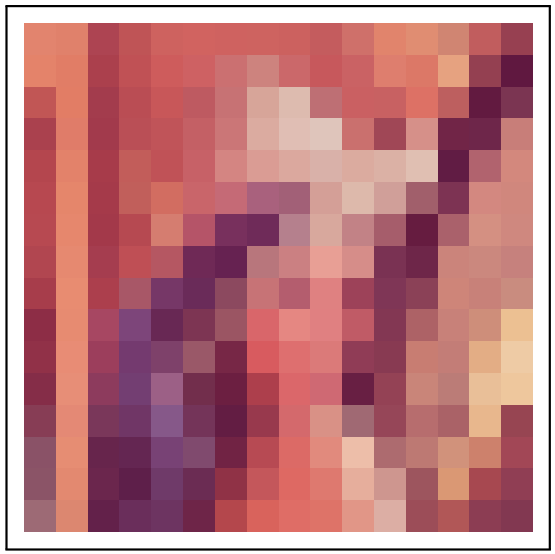


Morphers Examples III

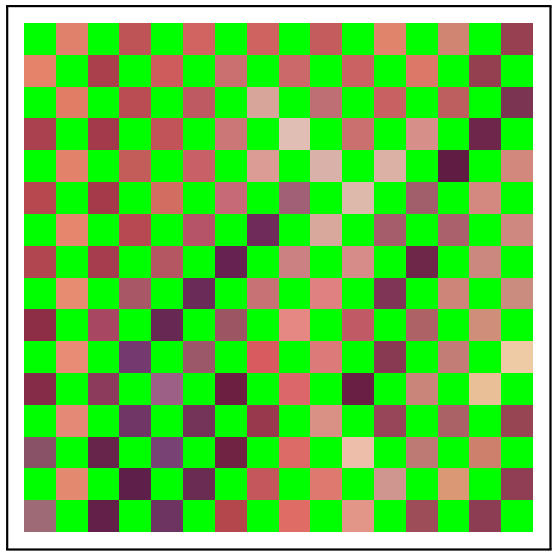
Restriction by a predicate

```
fun::p2b::chess chessboard; // Predicate: (x,y) ↦ (x + y) ≡ 0 (mod 2).
fill(lena | chessboard,     // 'lena' restricted by 'chessboard'.
     green);
```

"lena" before.



"lena" after.

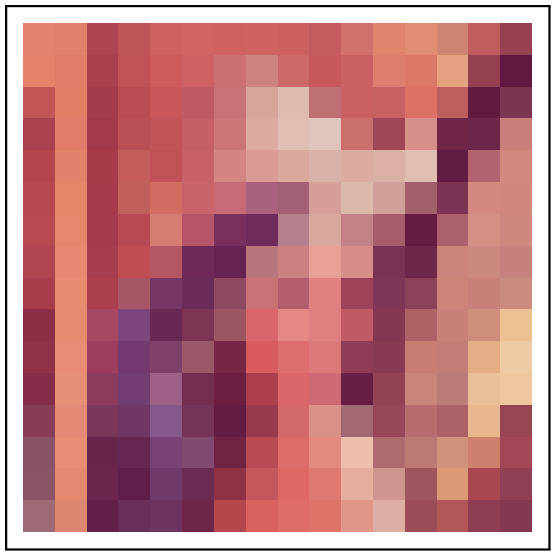


Morphers Examples IV

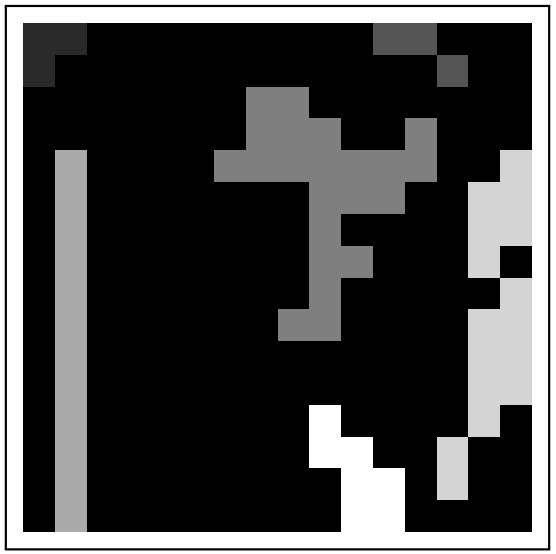
Restriction by a mask

```
image2d<label_8> label = load("label.pgm");
fill(lena | pw::value(label) == 3, // 'lena' limited to label 3.
     green);
```

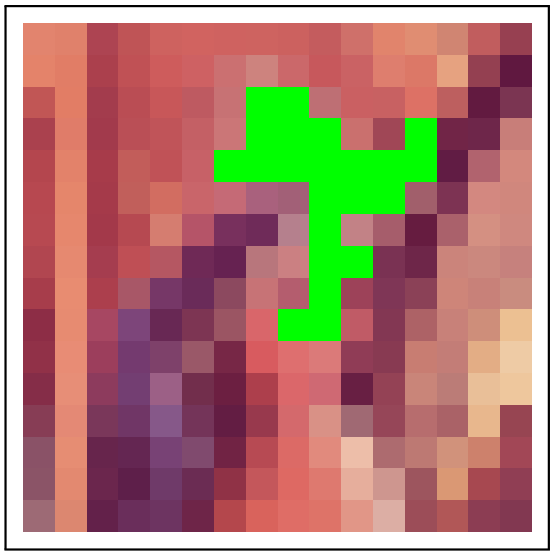
"lena" before.



"label".



"lena" after.

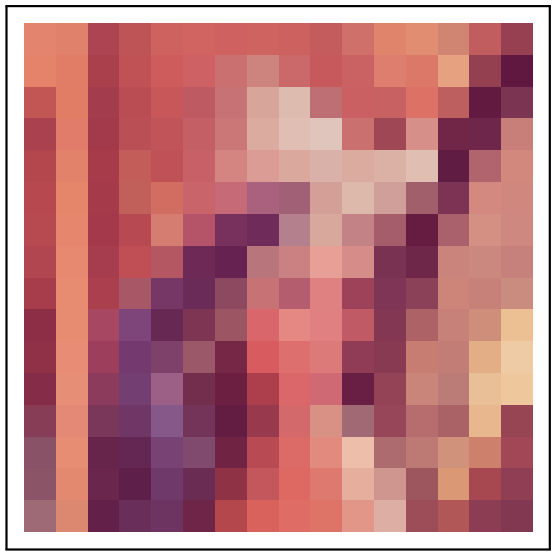


Morphers Examples V

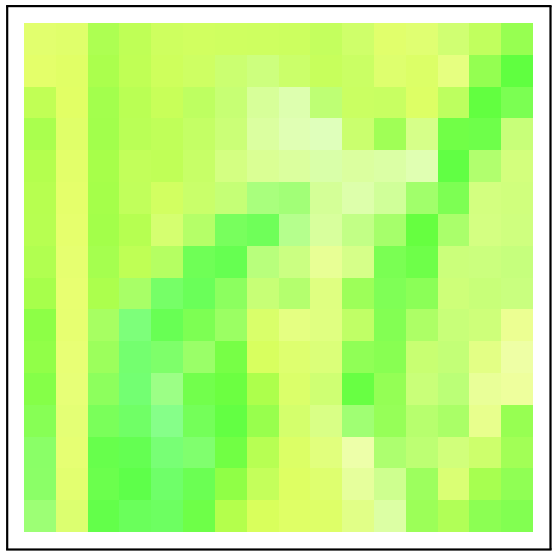
Restriction to a component

```
// ‘lena’ restricted to the ‘‘green’’ channel.
fill(fun::green() << lena;
     255);
```

“lena” before.



“lena” after.



Morphers Examples VI

Recording changes in an image

```
auto lena_rec = record(lena);  // Attach a recorder to ‘lena’.
fill(lena_rec, green);         // Record changes during ‘fill’.
save(lena_rec, "lena-fill.avi");
```

“lena” throughout.

Combining Morphers

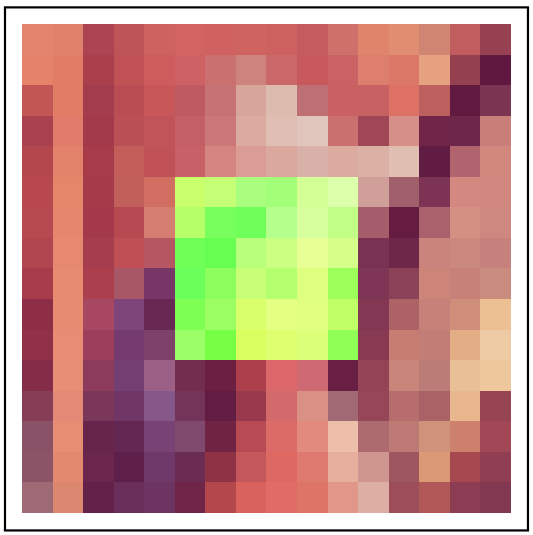
Mask + restriction to channel

```
fill((fun::green() << lena)
       | box2d(5,5, 10,10),
     255);
```

Recorder + mask

```
auto lena_rec = record(lena);
box2d roi(5,5, 10,10);
fill(lena_rec | roi, green);
```

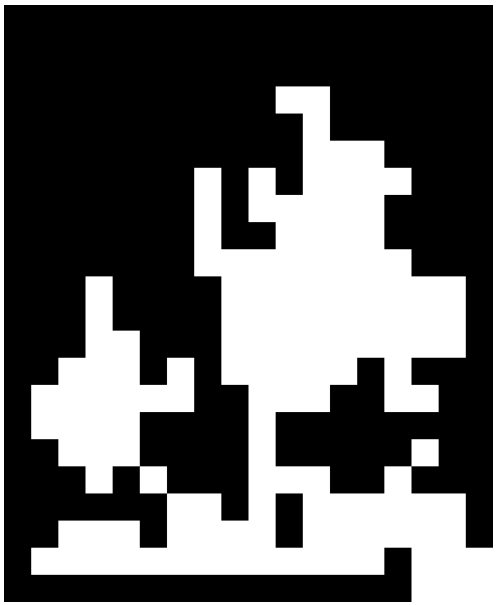
"lena" after.



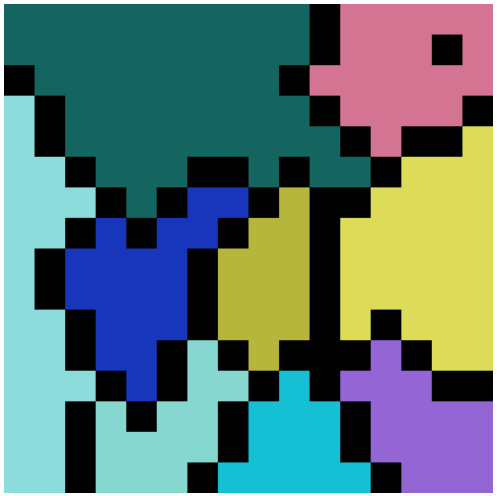
"lena" throughout.

Morphers Applied to Algorithms

Breadth-first thinning.



Watershed transform.

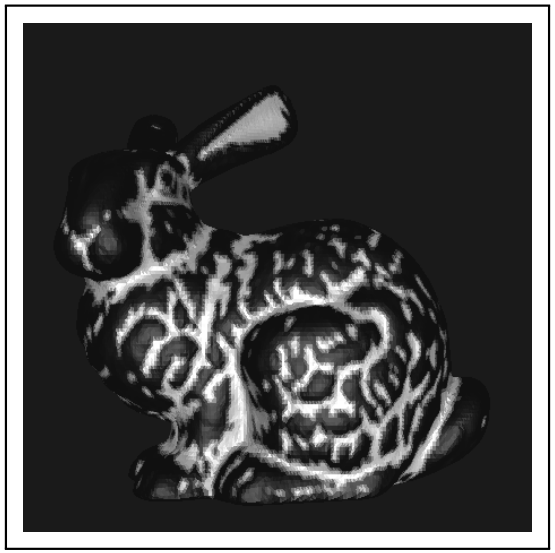


Skeletons by Generic Thinning IV (continued)

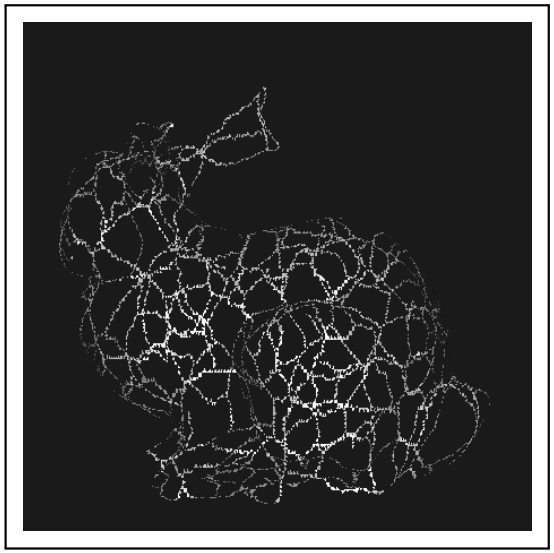
Surface mesh (triangles only)

Mesh.

Curvature.



“Thick” skeleton.



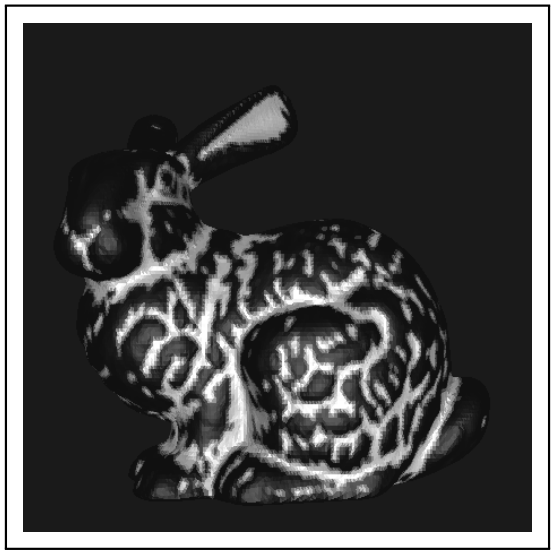
One morpher applied : restriction to triangles.

Skeletons by Generic Thinning IV (continued)

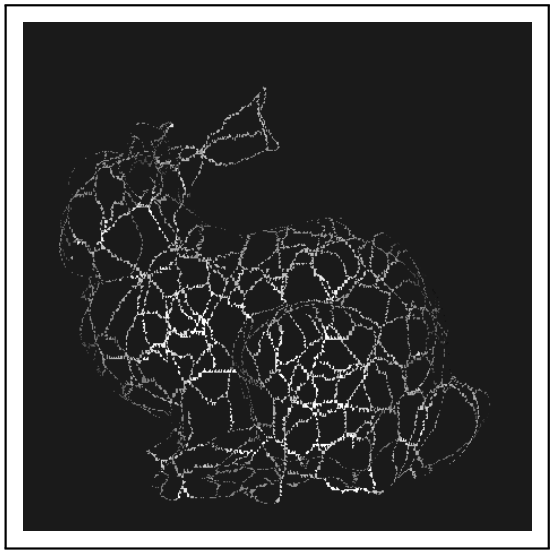
Surface mesh (triangles only)

Mesh.

Curvature.



“Thick” skeleton.

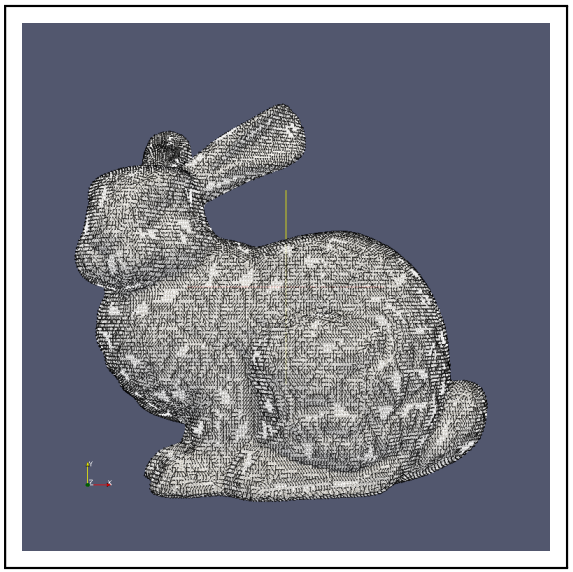


One morpher applied : restriction to triangles.

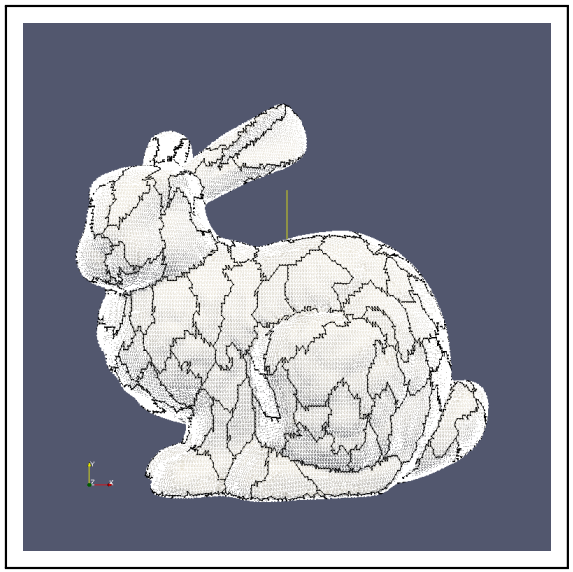
Skeletons by Generic Thinning V

Surface mesh (thin skeleton by 2- and 1-collapse)

2-collapse.



1-collapse.



Two morphers applied :

1. Restriction to triangles.
2. Extension to edges and vertices.

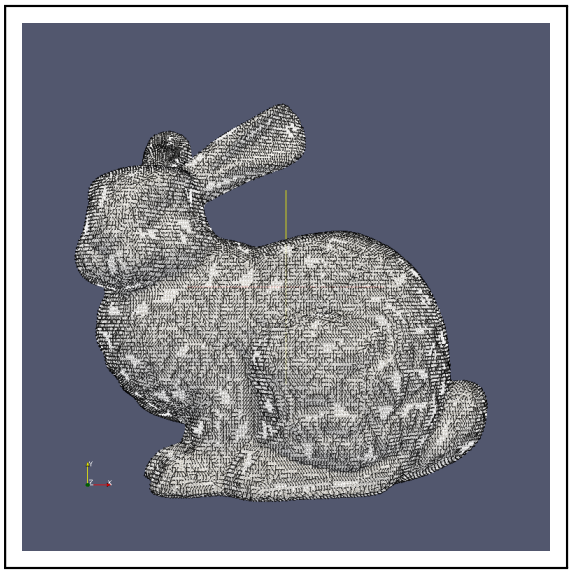
Two morphers applied :

1. Restriction to edges.
2. Extension to vertices.

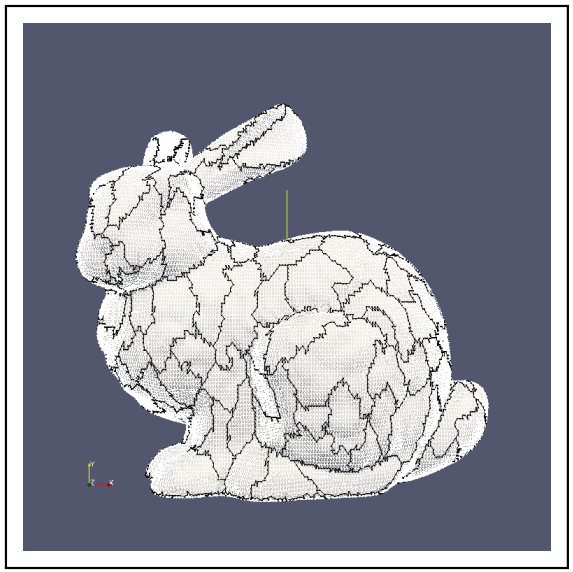
Skeletons by Generic Thinning V

Surface mesh (thin skeleton by 2- and 1-collapse)

2-collapse.



1-collapse.



Two morphers applied :

1. Restriction to triangles.
2. Extension to edges and vertices.

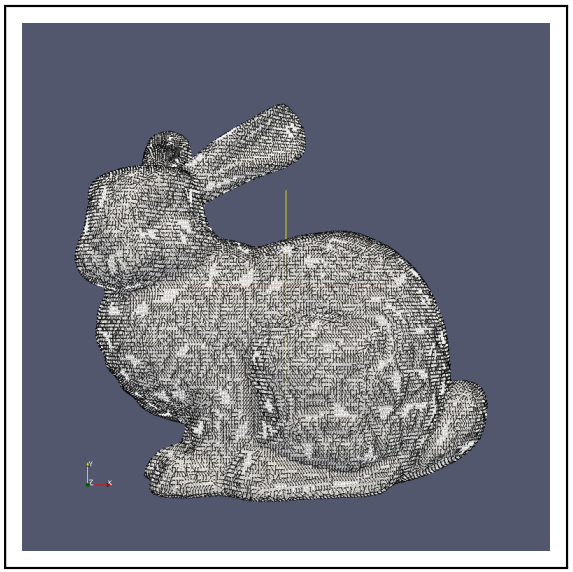
Two morphers applied :

1. Restriction to edges.
2. Extension to vertices.

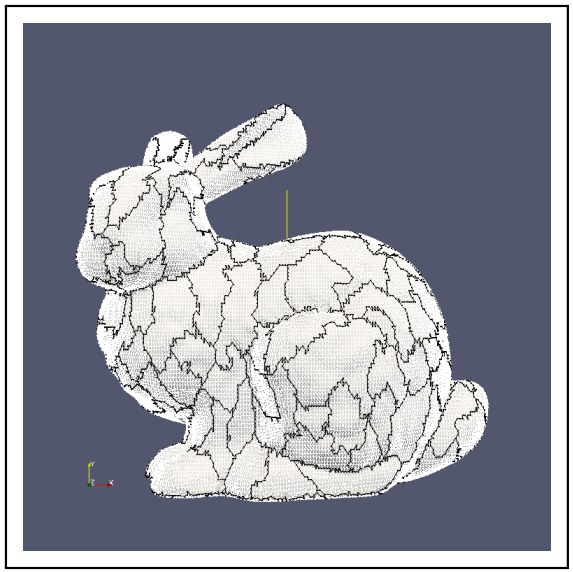
Skeletons by Generic Thinning V

Surface mesh (thin skeleton by 2- and 1-collapse)

2-collapse.



1-collapse.



Two morphers applied :

1. Restriction to triangles.
2. Extension to edges and vertices.

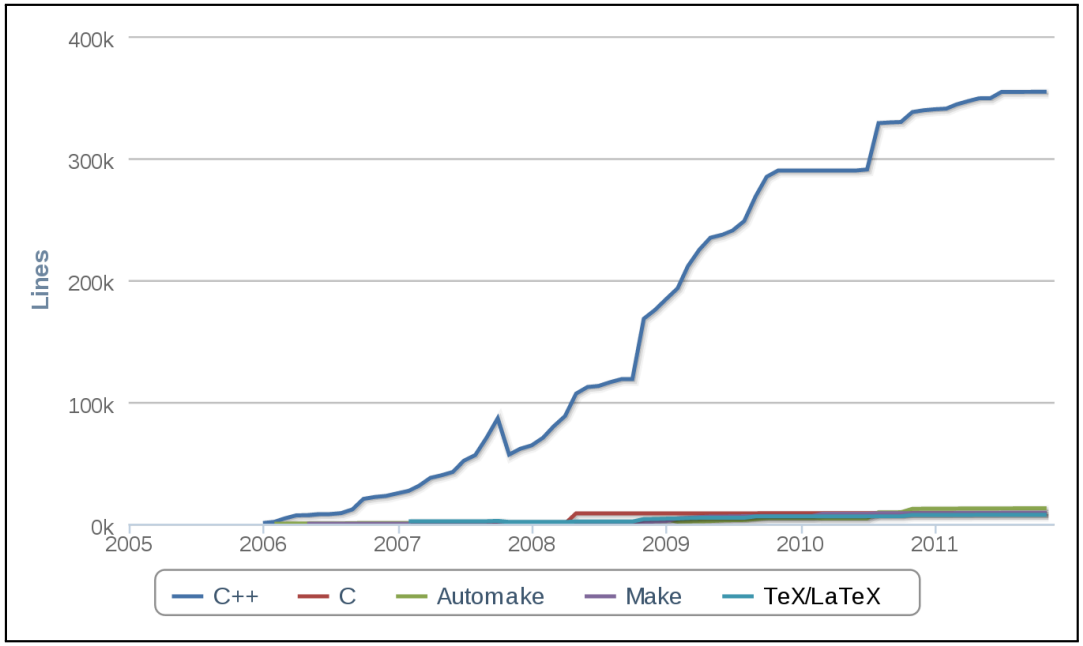
Two morphers applied :

1. Restriction to edges.
2. Extension to vertices.

Epilogue

The Olena Project

- Ideas effectively translated into code.
- Free software (GNU General Public License).
- Available on the Web (`http://olena.lrde.epita.fr`).



Applications of Olena

Projects

- SCRIBO (document image analysis),
- MELIMAGE (medical imaging).

Integration

Debian GNU/Linux, MacPorts (pending)

Students Beginners (every year).

Perspectives

- Experiment with other (uncommon) data structures.
- Fast implementations (parallelization, vectorization) in a generic context.
- Delivering more tools—especially a dynamic-static bridge.
- Transfer of ideas to domains connected to IP.
- Transfer of ideas to fields other than IP.

Genericity as a Solution for Image Processing