Technical Report

Pattern Recognition and Image Processing Group
Institute of Computer Aided Automation
Vienna University of Technology
Favoritenstr. 9/183-2
A-1040 Vienna AUSTRIA
Phone: +43 (1) 58801-18351
Fax: +43 (1) 58801-18392
E-mail: illetsch@prip.tuwien.ac.at

URL: http://www.prip.tuwien.ac.at/

PRIP-TR-106 October 12, 2006

Effective Programming of Combinatorial Maps using COMA - A C++ Framework for Combinatorial Maps

T. Illetschko, A. Ion, Y. Haxhimusa, and W.G. Kropatsch

Abstract

Combinatorial maps and pyramids have been studied in great detail in the past, and it has been shown that this concept is advantageous for many applications in the field of image processing and pattern recognition by providing means to store information of the topological relations of the represented data. In the course of these studies, the properties of combinatorial maps have been investigated using different sets of permutations, different operations and different algorithms. In each case new software had to be created in order to conduct experiments, as the existing programs were designed to work only for a specific model. Due to the complexity of combinatorial maps, the implementation of such a software is a time and resource intensive task. Thus these programming efforts were often responsible for delaying the presentation of new results in the past. This paper presents COMA - a C++ framework for combinatorial maps - that has been created during recent studies of combinatorial maps, motivated by this problem. Using an object oriented approach, COMA was specifically designed to allow an efficient and quick integration of changes to the model of combinatorial maps used, as well as the implementation of new algorithms. As a consequence COMA significantly reduces the amount of time needed to set up new experiments.

1 Introduction

Handling "structured geometric objects" is important for many applications related to geometric modeling, computational geometry, image analysis, etc.; one has often to distinguish between different parts of an object, according to properties which are relevant for the application (e.g. mechanical, photometric, geometric properties) [29]. For e.g. in image analysis, a region is a (structured) set of pixels or voxels, or more generally an abstract cellular complex consisting of dimensions 0, 1, 2, 3 ... (i.e. 0-cells are vertices, 1-cells are edges, 2-cells are faces, 3-cells are volumes, ...) and a bounding relation [27].

The structure, or the topology, of the object is related to the decomposition of the object into sub-objects, and to the relations between these sub-objects. Many topological models have been conceived for representing the topology of subdivided objects, since different types of subdivisions have to be handled: general complexes [8, 9] or particular manifolds [1, 2], subdivided into any cells [17, 13] or into regular ones (e.g. simplices, cubes, etc.) [15, 32]. Few models are defined for any dimensions [3, 31]. Some of them are (extensions of) incidence graphs or adjacency graphs. Their principle is often simple, but they cannot deal with any subdivision without loss of information and operations for handling such graphs are often complex. Other structures are "ordered" [5, 31], and they do not have the drawbacks of incidence or adjacency graphs. A subdivided object can be described at different levels, so several works deal with hierarchical topological models and topological pyramids [12, 3, 28]. For geometric modeling, levels are often not numerous. For image analysis, more levels are needed since the goal is to rise up information which is not known a priori.

The authors in [7, 20] show that 2D combinatorial maps are suitable topological structures to be used in 2D segmentation. Many domains need to work in 3D imagery (e.g. medicine, geology), so the theoretical framework of 2D combinatorial maps has been extended to 3D [10, 4, 23]. These works introduce more than one set of permutations to be used for the combinatorial maps as well as defining different operations to collapse a given combinatorial map.

When studying 2D and 3D combinatorial maps, experimental results are an important step to demonstrate the theoretical results. While software implementations of combinatorial maps exist (e.g. [11]), these libraries have

concentrated on a specific set of permutations and operations. This means that new software or at least a lot of programming effort was necessary whenever a different model of combinatorial maps was studied. This motivated the implementation of "COMA" with the goal of creating a C++ framework that provides all functionality for combinatorial maps of any dimension using an arbitrary set of permutations and operations. Examples for using COMA can be found in [21, 26, 25, 23].

This paper presents the main structure of the COMA library and gives examples on how to use it for implementing a specific model of combinatorial maps or a specific set of experiments. For this purpose Section 2 will give an overview of the architecture of COMA, explaining the independent layers within the library encapsulating the different aspects of a program using combinatorial maps.

A comprehensive overview of the library is presented in Sections 3 and 4. Section 3 contains a description of each class, explaining its purpose within COMA. Examples are provided to show the capabilities provided by each class and how to adapt certain classes in order to implement changes efficiently. In Section 4 a complete example algorithm is presented to demonstrate how to use the library to set up a specific set of experiments.

Finally experiments conducted with COMA are presented in Section 5 and conclusions and an outlook are discussed in Section 6.

The theoretical background of terms mentioned in this report are explained in great detail in [24].

2 Architecture

COMA is a framework providing all functionality to work with combinatorial maps and combinatorial pyramids in an efficient way. The main goal of this library is to offer the means to implement new algorithms and different models of combinatorial maps in a fast manner without having to implement the complete theory of combinatorial maps every time. To achieve this goal the following guidelines were used when designing COMA:

- Using a different set of permutations for a certain dimensionality as well as using different operations should be done in single place and transparent to other parts of the library, especially the algorithm.

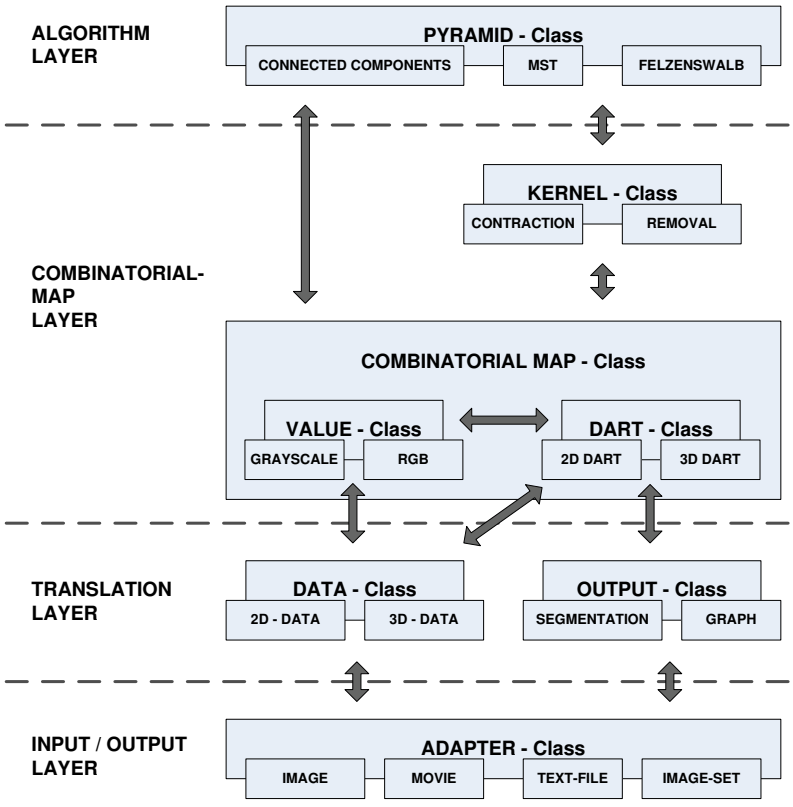


Figure 1: Architecture of the COMA framework.

- The kind of values represented by the combinatorial map (e.g. the color of a voxel represented by a given vertex) should have no impact on other parts of the library. Thus algorithms do not need to be changed when working with different types of input sources.
- When implementing an algorithm with the framework, all functionality of combinatorial maps should be available to the algorithm. At the same time the algorithm should not be responsible for maintaining the consistency of the combinatorial maps or need to have any knowledge of the specific implementation of the combinatorial maps.
- The way in which the data is accessed should be transparent to the library. This applies for both, the input data as well as the data written as the result of an algorithm.
- When implementing a new output format to save the results of an experiment (e.g. saving the pyramid as a set of labeled images, or creating a visualization of combinatorial maps as graphs), this method should automatically be usable for all other algorithms that use the library.

These guidelines led to the architecture shown in Fig. 1. The architecture is divided into four functional layers, each being responsible for different tasks.

2.1 Functional Layers

The following Section gives an overview of the different layers and the classes present in these layers. A more detailed description of each class is provided in the next section.

2.1.1 Input/Output Layer

This layer is responsible for providing a transparent way of reading data from and writing data to different data-sources like images, movies, text-files or sets of images. For this purpose the abstract `coma_adapter` class provides one interface for accessing the data as an n-dimensional array to the library. The handling of the different types of data-sources is taken care of in classes derived from this base class. They implement the functions for reading the data from and writing the data to the file system in each specific format.

2.1.2 Translation Layer

This layer is used for the translation between the geometric, coordinate-based representations of data that is used in most data-files, into the topological representation used by the combinatorial maps. Two classes are defined to achieve this. The `coma_data` class reads values associated with coordinates and assigns them to the i-cells of the combinatorial maps (e.g. associating a given voxel to a specific vertex or volume). In addition it provides a function for mapping indices to darts and i-cells in a unique and bi-directional way. This mapping mechanism is defined for each set of permutations introduced to the library by a class derived from `coma_data` for this purpose.

The second class of this layer is the `coma_output` class. It is used to save the results of an algorithm to the file system. As such it provides functions for saving maps, pyramids and receptive fields. The base class implements the basic operations needed for every output format while rendering the output for each specific output format is defined in the derived classes. (e.g. saving the result as a segmentation image, as a text file containing the labels associated with each vertex, or as an image showing the combinatorial map).

2.1.3 Combinatorial-Map Layer

The actual implementation of combinatorial maps is located at this layer. It consists of the four classes `coma`, `coma_value`, `coma_dart` and `coma_kernel`.

The `coma_value` class is a container storing the values associated with i-cells of a combinatorial map. It provides a generic interface for the most commonly used functions when working with values read from the data-source, without knowing the type of data (e.g. grayscale values, RGB values, etc.). Specializations are used whenever a default function is not adequate for a certain type of data (e.g. calculating the weight of an edge might be different when working with RGB values or grayscale values).

The `coma_dart` class implements the specific set of permutations of a combinatorial map (e.g. α and σ for 2D, or β_1, β_2, and β_3 for 3D) and provides mechanisms for working with these permutations. In addition the values associated with the i-cells that each dart belongs to, can be accessed using a pointer to the corresponding `coma_value` object. The dart class also implements most of the commonly needed operations for darts like traversing

all darts belonging to the same cycle of a permutation or to the same i-cell, calculating the degree and dual-degree of the i-cell the dart belongs to, or retrieving all darts belonging to the receptive field of a given dart.

The class `coma` represents a complete combinatorial map. It uses the `coma_dart` and `coma_value` objects to build new maps. This is done in two ways:

- Building a grid-like combinatorial map from a given `coma_data` object representing this data.
- Building a reduced combinatorial map from an existing map by applying contraction- and removal-kernels.

Once a combinatorial map has been built, the `coma` class can be used to access all darts and i-cells within this map. For this purpose two kinds of arrays are provided:

- The set D contains all darts of a map. A specific dart can be directly accessed using its index, or an iterator can be used to access all darts in a loop.
- For each i-cell of a given dimension, one array is created. These arrays store one dart for every i-cell of that dimension. Again these arrays can be accessed using either the index of the cell or by using an iterator to loop through all cells. Once a dart has benn retrieved, the other darts belonging to the same i-cell can be accessed using the iterators of the `coma_dart` class.

The fourth class of the combinatorial map layer is the `coma_kernel` class. It provides functions for building a valid kernel to be applied to a combinatorial map. The base class implements the infrastructure needed by all kernels like adding a dart to the kernel or accessing darts in the kernel. The specific conditions for each operation are implemented in the two derived classes `coma_contraction_kernel` and `coma_removal_kernel`.

2.1.4 Algorithm Layer

The algorithm layer implements the functionality for building a pyramid based on combinatorial maps using a given algorithm. In most cases this layer is the only one that must be adapted when implementing a new algorithm.

The class `coma_pyramid` is initialized with an existing combinatorial map as its base level. A new reduced level is then built by implementing an algorithm that select edges as candidates for a contraction. The new level is constructed by first applying these edge contractions and then simplifying the resulting map by selecting additional i-cells that may be contracted or removed according to the conditions for these operations. This way a complete combinatorial pyramid is built by creating new and reduced levels until no more candidate edges are found in the algorithm step.

2.2 Extending COMA

One of the goals when designing COMA, was to reduce the necessary amount of programming to a minimum when implementing a new functionality (e.g. a different set of permutations, different input- and output-formats or building a new algorithm based on combinatorial maps). Table 1 gives an overview of the classes that must be extended when implementing changes to the framework that are typically needed when setting up a new experiment. As can be seen, most changes require only the adaption of a single class. The only extension that has an impact on more than two classes is the implementation of a new operation.

change	coma	dart	value	adapter	data	output	kernel	pyramid
permutations		×			×			
data source				×				
output format						×		
algorithm								×
operation	×	×					×	
value types			×					

Table 1: Classes that must be adapted when extending COMA.

3 Classes

This Section provides a description of the most important classes of the COMA framework. An overview of each class is given as well as examples on how to use each class.

3.1 coma_adapter

The `coma_adapter` class is the only class of the input-/output layer. Its main purpose is to encapsulate all interaction with the actual data-sources by providing a uniform way of loading, saving and accessing data, independent of the specific format in which the data is stored on the file system. E.g. a combinatorial map can be built from a single image, a stack of images, a movie or a simple text-file. This is achieved by mapping each data-source to an nD matrix within the `coma_adapter` class. This matrix is then used by all other classes of COMA.

To avoid that adding support for an additional file format automatically results in changes to `coma_adapter`, derived classes are used for each specific file format. E.g. a class `coma_adapter_image` is used to extend COMA to support reading and writing of 2D images. Most of the functionality is implemented in the base class, while the image specific translation is handled by the derived class `coma_adapter_image`.

Listing 1 shows an example of loading, accessing and saving data from a 2D image using the `coma_adapter` class.

3.2 coma_data

The `coma_data` class is one of the two classes that reside in the translation layer. This layer is responsible for the translation between the coordinate based geometric representations that the input and output data is usually stored in, and the topological representation of combinatorial maps. Introducing such a layer allows all other classes of the framework to remain unaware of the geometric properties of the data.

While `coma_data` handles the translation of input data, the data itself is read using the `coma_adapter` class. Once a data object is initialized this way, three sets of functions are provided that

```
// Create a new adapter object with its filename
coma_adapter_image<> adapter("test_image.ppm");

// Load the image to make its data accessible as a 2D matrix
adapter.load();

// Access the pixel at coordinates (2,3)
vcl_cout << "Value of pixel (2,3):"
  << adapter(2,3) << "\n";

// Change the pixel at coordinates (4,1)
adapter(4,1) = WHITE;

// Save the changes using a different filename
adapter.save("test_output.ppm");
```

Listing 1: Example for the `coma_adapter` class

- bi-directionally calculate a unique index for each dart and i-cell of a grid-like combinatorial map representing the data. E.g. calculate the index for the vertex that represents a specific pixel of a 2D image, identified by its coordinates.
- assign the actual values of the data to a given dart or i-cell identified by such an index. E.g. when using a 2D image as input, assign the color of each pixel to the vertex it is represented by in the initial map.
- calculate the index of the next dart in the cycle of a permutation. E.g. return the index of the next dart in the cycle of the σ permutation starting from a specific dart specified by its index.

Within a grid-like initial map, each dart can be directly associated with a set of coordinates and a direction. This information is used to calculate the index of a dart or i-cell. A helper class `params` is used to encapsulate these parameters.

Note, that the third set of function implies, that the `coma_data` class has knowledge about these permutations. It is therefore one of two classes within COMA that must be extended when a new set of permutations is introduced. While all permutation independent functions are implemented in the abstract base class `coma_data`, the third set of functions is handled by

derived classes: e.g. a class `coma_data_2d` is introduced for 2D combinatorial maps using α and σ.

Listing 2 shows an example of the `coma_data` class.

```
// Create a new adapter object as the data source
// and use this adapter to initialize the coma_data object
coma_adapter_image<> adapter("test_image.ppm");
coma_data_2d<> data(&adapter.load());

// Calculate the index of the dart at (1,2) pointing to the left.
int coord[] = {1,2};
vcl_cout << "Index of the dart at (1,2) pointing left:"
  << data.dart_index(params(coord,LEFT_2D)) << "\n";

// Calculate the index of the vertex, edge, and face associated
// with the dart at coordinates (1,2) pointing to the left.
vcl_cout << "Index of the vertex:" <<
  data.topology_index(params(coord,LEFT_2D, T_VERTEX)) << "\n";
vcl_cout << "Index of the edge:" <<
  data.topology_index(params(coord,LEFT_2D, T_EDGE)) << "\n";
vcl_cout << "Index of the face:" <<
  data.topology_index(params(coord,LEFT_2D, T_FACE)) << "\n";

// Print the value associated with the vertex of
// the dart at coordinates (1,2) pointing to the left.
vcl_cout << "Value associated with the dart:" <<
  data.value(params(coord,LEFT_2D,T_VERTEX)) << "\n";

// Calculate the index of the next dart in the cycle
// of the alpha and the sigma permutations.
int index = data.dart_index(params(coord,LEFT_2D));
vcl_cout << "Index of the alpha successor:" <<
  data.next_dart_by_perm(index, P_ALPHA) << "\n";
vcl_cout << "Index of the sigma successor:" <<
  data.next_dart_by_perm(index, P_SIGMA) << "\n";
```

Listing 2: Example for the `coma_data` class

3.3 coma_value

The `coma_value` class is introduced to COMA to provide a generic interface for working with different types of data (e.g. grayscale pixels or RGB pixels

when processing an image). This way an algorithm can be designed without knowing the exact format of the input data. For this purpose `coma_value` is implemented as an abstract base class that provides the most common functions like comparing, adding and subtracting values. The actual calculations for a given data type are implemented by programming a specialized class for these data types.

```
// Create two grayscale values
coma_value<vxl_byte, 2> value1, value2;
coord[] = {1,2};

// Initialize value1 with color white and index 1
value1.init(WHITE, 1, coord);
// Initialize value2 with color black and index 2
value2.init(BLACK, 2, coord);

// Create two RGB values
coma_value<vil_rgb<vxl_byte>, 2> rgb_value1, rgb_value2;

// Initialize RGB value1 with color red and index 3
rgb_value1.init(RED, 3, coord);
// Initialize RGB value2 with color blue and index 4
rgb_value2.init(BLUE, 4, coord);

// Compare values
if (value.compare(value2) != 0) {
  vcl_cout << "Value 1 and value 2 differ\n";
}
if (rgb_value.compare(rgb_value2) != 0) {
  vcl_cout << "RGB value 1 and RGB value 2 differ\n";
}

// Calculate the distance of the values
vcl_cout << "Distance of value 1 & 2:"
  << value.distance(value2) << "\n";
vcl_cout << "Distance of RGB value 1 & 2:"
  << rgb_value.distance(rgb_value2) << "\n";
```

Listing 3: Example for the `coma_value` class

In addition to storing the actual value of each pixel or voxel, `coma_value` also allows to store coordinates and an index. An example working with RGB-values and grayscale values is shown in listing 3. As can be seen, both

types of values can be used in the same way once they are initialized.

3.4 coma_dart

Darts are the atomic elements that a combinatorial map consists of. The corresponding class `coma_dart` is therefore also the central class of COMA. This class offers all functions for the typical operations on a dart:

- Using a given dart as the starting point for a traversal on the combinatorial map. E.g. examining all darts that belong to the cycle of one of the permutations, retrieving all darts belonging to an i-cell, or processing the darts belonging to the receptive field of a specific dart.
- Working with the properties of an i-cell that is associated with a dart. These properties include the value of the associated pixel or voxel, the degree and dual-degree of the i-cell, flags for self-loops and parallel i-cells, as well as the index of the i-cell or dart.
- Applying a certain operation on an associated i-cell, i.e. contracting or removing the i-cell.

For some of these functions the class must know about the used set of permutations and operations. Therefore this class is the second class of the COMA framework that must be extended when introducing a new set of permutations or operations. In order to keep the necessary amount of programming at a minimum, all generic functions are implemented in a the base class `coma_dart`. The set of permutations and operations supported by the specific model of combinatorial maps is handled by derived classes. Currently two such derived classes exist: `coma_dart_2d` for 2D combinatorial maps using α and σ, and `coma_dart_3d` for 3D combinatorial maps using β_1, β_2, and β_3.

Two ways of traversing the darts of the cycle of a permutation or that belong to the same i-cell are supported by the `coma_dart` class. Each dart object contains a set of pointers that will return the next dart directly, or iterators can be used to loop through all darts of the cycle or i-cell.

Listing 4 shows an example of using the functions of the `coma_dart` class. For this example an instance of the class `coma_dart_2d`, identified by `dart`,

```
// Print the details of the dart
vcl_cout << "Details of the dart: "
  << dart << "\n";

// Print the details of the pixel associated with the dart
vcl_cout << "Details of the dart: "
  << dart->value() << "\n";

// Print the indices of all darts belonging to
// the same vertex using a topology iterator
vcl_cout << "Darts belonging to the same vertex: ";
coma_dart_2d<>::top_iterator it;
for (it = dart->topology_iterator(T_VERTEX);it.dart;++it) {
  vcl_cout << it.dart << "\n";
}

// Print the the indices of all darts in the cycle of the
// sigma permutation using a permutation iterator
vcl_cout << "Darts belonging to the sigma permutation: ";
coma_dart_2d<>::iterator it2;
for (it2 = dart->perm_iterator(SIGMA_2D);it2.dart;++it2) {
  vcl_cout << it2.dart << "\n";
}

// Print the value of the second vertex of the edge
// using the alpha permutation
vcl_cout << "Pixel at the other end of the edge: "
  << dart->perm(ALPHA_2D)->value() << "\n";

// Print the dual-degree of the face this dart is incident to
vcl_cout << "The degree of the face is: "
  << dart->dual_degree(T_FACE) << "\n";

// Test if the edge that this dart belongs to is a self-loop
if (dart->is_self_loop()) {
  vcl_cout << "The edge is a self-loop\n";
}
```

Listing 4: Example for the `coma_dart` class

is used. It is assumed that this is a valid and initialized dart object. How to retrieve dart objects of a given combinatorial map will be explained in the Section describing the `coma` class.

3.5 coma

The main purpose of the `coma` class is to create new combinatorial maps, to maintain the set of darts D, and to provide efficient methods for accessing the specific darts or i-cells encoded by the map.

There are two possibilities to create a new combinatorial map:

- To create an initial grid-like combinatorial map, a coma object can be initialized by specifying a `coma_data` object.
- To create a reduced combinatorial map, a coma object can be initialized by specifying an existing `coma` object to be simplified as well as a contraction and removal kernel to be applied to this map.

A `coma` object created this way will contain an array `d` that holds all darts of the combinatorial map. These darts can then either be directly accessed using their index, or an iterator can be used to process all darts in the array.

Additionally there is one array for each kind of i-cell encoded by the combinatorial map. These arrays store pointers to one dart for each i-cell. Again the darts can either be accessed using the index of the i-cell or by iterating through all i-cells using an iterator.

Two functions are implemented to enable a quick validation of a combinatorial map. The function `print_details` will print details of all darts and i-cells of the map. This can be used to check small maps manually or observe the results of a particular contraction or removal operation applied to a map. The function `check_map` will conduct a series of automated consistency checks and print a report if errors are found.

Listing 5 shows an example of creating a map from a test image and then accessing darts and i-cells of the combinatorial map.

3.6 coma_kernel

To create a reduced combinatorial map from an existing one, contraction and removal operations are used. Kernels are used to apply multiple such operations in one step. First, all darts belonging to i-cells that are chosen for contraction or removal are stored in a kernel. Once all darts have been

```
// Create an adapter for the test-image
coma_adapter_image<> adapter("test.ppm");
// Create a data object using the adapter object
coma_data_2d<> data(&adapter.load());

// Create a combinatorial map and initialize it
// with the data object to create a grid-like
// initial map representing the test-image.
coma_2d<> map;
map.init(&data, new coma_settings_2d());

// check the consistency of the created map
if (map.check_map() != 0) {
  vcl_cout << "Map is inconsistent!\n";
}

// print a summary of the combinatorial map,
// details of all darts and i-cells, the number
// of edges and the number of darts
vcl_cout << "Summary of the map: "
  << map << "\n";
vcl_cout << "Details of the map: "
  << map.print_details(true, true) << "\n";
vcl_cout << "Number of edges: "
  << map.edges().count() << "\n";
vcl_cout << "Number of darts: "
  << map.d().count() << "\n";

// Print details of the dart with index 52,
// and the face with index 19.
vcl_cout << "Details of dart 52: "
  << map.d().get(52) << "\n";
vcl_cout << "Details of face 19: "
  << map.faces().get(19) << "\n";

// Print the index of every edge
// using an iterator
typename coma_2d<>::coma_array_type::iterator it;

for(it=map.vertices.begin();it!=map.vertices().end();++it) {
  vcl_cout << "Edge with index: "
    << it.dart->value(T_VERTEX)->index << "\n";
}
```

Listing 5: Example for the `coma` class

selected and added to the kernel this way, the new combinatorial map can be created by applying all contraction and removal operations simultaneously.

For this purpose the class `coma_kernel` is provided. It offers functions to add a dart or an i-cell to the kernel, to check whether the conditions for the given operation are met by the dart or i-cell, and to access all darts currently in the kernel. With the exception of the operation specific conditions all functions are implemented in the abstract base class `coma_kernel`. The `check_dart` function that validates the conditions of each operation, is handled by derived classes. Currently such derived classes exist for the contraction and for the removal operation, implementing the 5 conditions introduced by [23]: `coma_kernel_contraction` and `coma_kernel_removal`.

```
// Create an adapter and data object for a test image
coma_adapter_image<> adapter("test.ppm");
coma_data_2d<> data(&adapter.load());

// Create a grid-like initial map for this image
coma_2d<> map;
map.init(&data, new coma_settings_2d());

// Create a contraction kernel
// Initialize it and assign it to the map
coma_contraction_kernel<> *c_kernel =
  new coma_contraction_kernel<>();
c_kernel->init(data.dart_size(), &settings);
map.c_kernel = c_kernel;

// add some edges to the contraction kernel by selecting
// one dart of the edge and specifying T_EDGE
if (c_kernel->add_dart(map.d().get(48), T_EDGE) != 0) {
  vcl_cout << "Edge with dart 48 may not be contracted\n";
}
if (c_kernel->add_dart(map.d().get(-70), T_EDGE) != 0) {
  vcl_cout << "Edge with dart -70 may not be contracted\n";
}

// build new map using the contraction kernel
coma_2d<> new_map;
new_map.init(&map, new coma_settings_2d(), c_kernel,
  NULL, false, true);
```

Listing 6: Example for the `coma_kernel` class

Listing 6 shows an example of a reduced combinatorial map being built by using a contraction kernel.

3.7 coma_output

The `coma_output` class is used to save combinatorial maps in different output formats. E.g. the result of an algorithm could be saved as a segmentation image, a simple text file containing the labels of the surviving vertices, as a visualization of the i-cells encoded by the map, or as a textual description.

Most of the functionality needed for saving a combinatorial map stays the same, regardless of the actual output format. This includes processes like accessing the file system, the calculation of receptive fields, or functions to save each dart or i-cell of a given map. In most cases it is therefore sufficient to provide a function for transforming a dart or i-cell into the needed output-format. E.g. assigning a color and coordinates, or calculating for a given vertex the associated set of pixels in the output-image.

For this reason, the `coma_output` class is implemented as an abstract base class handling functions needed by all output formats. Derived classes are then used for the necessary transformations of each specific format (e.g. `coma_output_segmentation` is used to save segmentation images of whole maps. Since `coma_output` defines a generic interface for all output-classes, each derived class that implements a new output format can immediately be used by all programs based on the COMA framework.

Listing 7 shows an example of a combinatorial map that is saved in multiple output formats and to various file types. This is done by using different output adapters and `coma_output` classes.

Figure 2 shows examples of images created by some output classes. The image of a vase reconstructed using the receptive fields of a combinatorial map can be seen in Figure 2a. Figure 2b shows a segmentation image of the same vase from a reduced combinatorial map. Figure 2c&d display visualizations of a complete 3D combinatorial map and the receptive field of one vertex. In these cases COMA produced the input files for the 3D visualization software "raybooster" that was provided by the Computer Graphics Group of the Institute of Computer Graphics and Algorithms of the University of Technology Vienna. A detailed description of this software is provided by [16, 6].

a) reconstructed image

b) segmentation image

c) 3D visualization

d) 3D visualization of one receptive field

Figure 2: Examples for different output formats of combinatorial maps. Output created as a 2D image (a), a 2D segmentation image (b), a 3D visualization using the raybooster software (c), and the 3D visualization of a receptive field (d).

```
// Create an adapter for a test image and
// initialize a data object with it.
coma_adapter_image<> adapter("test.ppm");
coma_data_2d<> data(&adapter.load());

// Create a grid-like initial map for this image
coma_2d<> map;
map.init(&data, new coma_settings_2d());

// Create an adapter for saving the map
coma_adapter_image<> image_adapter;

// Save the bounding relationship diagram of the map
coma_output_bounding_graph<> output(&image_adapter, &map);
output.save_map("bounding_graph.ppm");

// Save the map as a segmentation image
// by using a different output class
coma_output_segmentation<> output_image(&image_adapter, &map);
output.save_map("segmentation.ppm");

// Additionally save the segmentation as a bsi file
// by using a different adapter
coma_adapter_bsi_file bsi_file;
coma_output_segmentation<> output_bsi(&bsi_file, &map);
output_bsi.save_pyramid("segmentation.bsi");
```

Listing 7: Example for the `coma_output` class

Figure 3, 4, and 5 show examples of using COMA to visualize combinatorial maps as graphs. A bounding relationship diagram of a 3D combinatorial map can bee seen in Figure 5, while Figure 3 and 4 display the darts, vertices, and permutations of 2D combinatorial maps.

Figure 3 shows an initial grid-like combinatorial map G_1 created from a 5×5 image (The background vertex is not visualized). In the same way a map G_2 created from G_1 using edge contractions is given in Figure 4. The details (darts and permutations) of G_1 and G_2 can be found in appendix A.

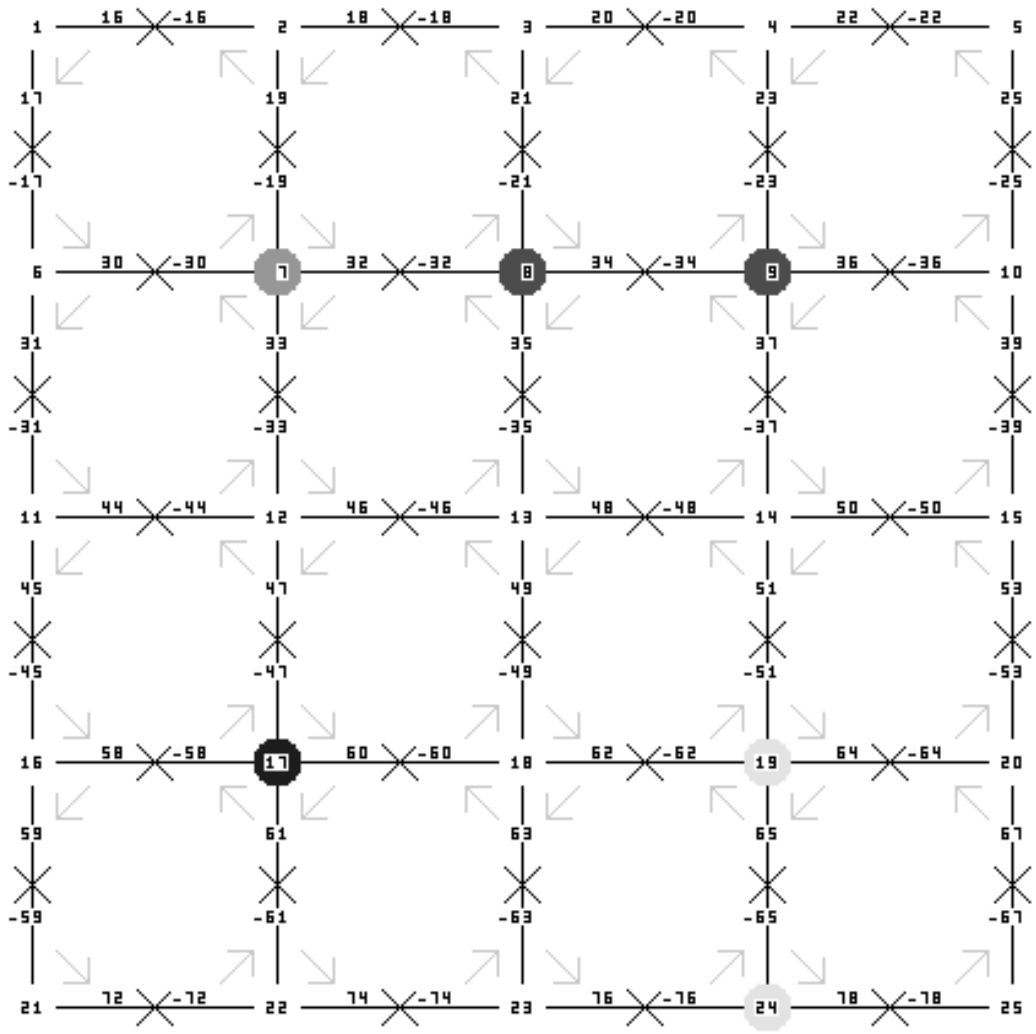


Figure 3: Visualization of a grid-like 2D combinatorial map.

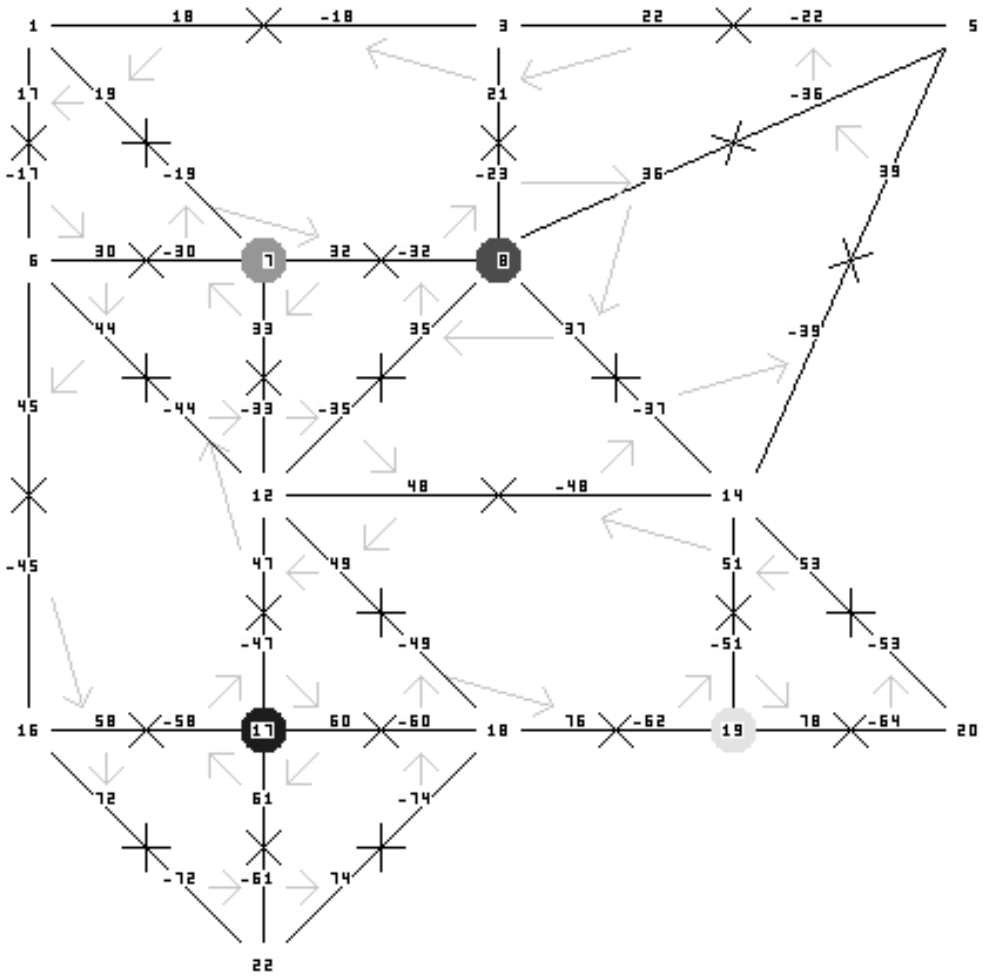


Figure 4: Visualization of a reduced 2D combinatorial map.

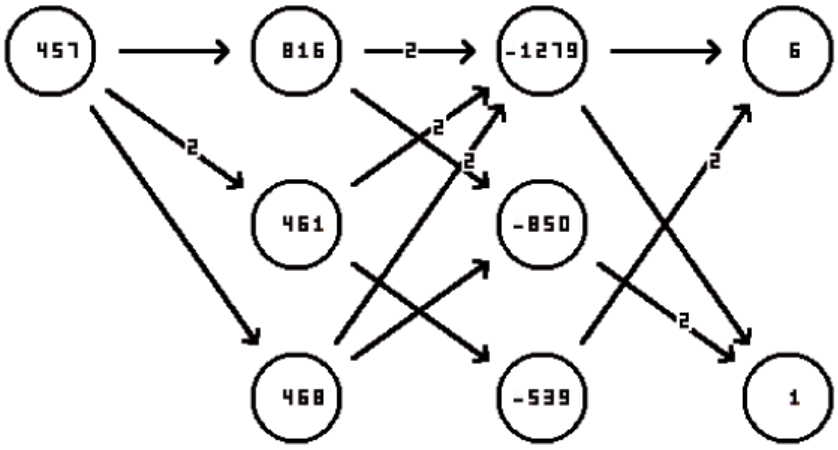


Figure 5: Visualization of a 3D combinatorial map as a bounding relationship diagram.

3.8 coma_pyramid

The `coma_pryamid` class is the only class in the algorithm layer. It is responsible for building combinatorial pyramids where each levels contains a reduced combinatorial map based on the level below. For this purpose the pyramid is initialized with a combinatorial map as the base level of the pyramid. New levels are then created using the algorithm described in [23]:

1. Merging of adjacent voxels that belong to the same component (algorithm step)
2. Eliminating redundant i-cells (simplification steps)

When creating a new level, `construct_c_kernel_by_algorithm` is called first to select all candidate edges that shall be contracted to merge adjacent voxels. All darts belonging to these edges are added to a contraction kernel if they fulfill the conditions for an edge contraction.

Once vertices have been merged this way, the pyramid uses the function `construct_c_kernel_by_simplification` to select all i-cells that can be simplified using a contraction operation. In the same way the function

`construct_r_kernel_by_simplification` is used to identify i-cells that can be eliminated from the combinatorial map using a removal operation. This process is repeated until no more candidates for a simplification can be found in the combinatorial map.

When programming a new algorithm using COMA, it is usually sufficient to replace the function `construct_c_kernel_by_algorithm` in order to implement the algorithm's specific way of deciding which voxels are to be merged at each level. The simplification steps will then be applied automatically to reduce the number of other i-cells in the resulting combinatorial map. Once a pyramid is built, various functions are provided by `coma_pyramid` to access the different levels of the pyramid and to work with the receptive fields of single darts or i-cells.

Listing 8 contains an example that demonstrates how the `coma_pyramid` class is used to build a combinatorial pyramid for a 2D test image. This process is independent from the actual algorithm used for the algorithm step.

```
// Load the test image using an adapter and data class
coma_adapter_image<> adapter(filename);
coma_data_2d<> data(&adapter.load());

// Initialize the base map with the image
coma_2d<> map;
map.init(&data, new coma_settings2d());

// Create a new pyramid and initialize it with
// the map as its base level
coma_pyramid<> pyr;
pyr.init(&map, new coma_settings2d());

// Build the complete pyramid
// This function will create new levels
// until no more edges are found for a contraction
pyr.construct_all_levels();

// Print height of pyramid and a summary of the top level
vcl_cout << "Built a "
  << pyr.top_level()->level << "pyramid\n";
vcl_cout << "Summary of the top level "
  << pyr.top_level() << "\n";

// Retrieve the set of all darts belonging to
// the receptive field of the vertex with index 189
coma_pyramid<>::dart_vector_type field;

pyr.top_level()->receptive_field(field,
  pyr.top_level()->map->d().get(189),
  T_VERTEX);

// Iterate through all darts of the receptive field
// using an iterator
coma_pyramid<>::dart_vector_type::iterator it;

vcl_cout << "Darts in receptive field: ";
for (it=field.begin();it!=field.end();++it) {
  vcl_cout << (*it) << " ";
}
```

Listing 8: Example for the `coma_pyramid` class

4 Implementing an Algorithm

In the previous Sections, an overview of the the COMA framework has been given by describing the architecture and the classes of the library. In this Section, the process of building an algorithm based on COMA is demonstrated. For this purpose it is shown, how an algorithm similar to the one described in [14, 22] can be implemented with this framework.

Whenever new algorithms are implemented, it is necessary to program the specific algorithm step by extending `construct_c_kernel_by_algorithm` of the `coma_pyramid` class. In the case of the algorithm chosen for this example, the standard `coma_value` objects must also be extended, as will be described below. The algorithm works in two steps:

1. All edges are sorted and processed according to their weight in an ascending order. (The specific weight function depends on the values associated with each pixel. E.g. a subtraction if grayscale values are used).

2. For each edge the weight is compared to a threshold associated with the vertices it connects. The edge is contracted if the weight is below this threshold. Whenever two vertices are merged, a new threshold is calculated for the resulting single vertex.

4.1 Implementing the Threshold Functions

To implement the functionality of calculating the threshold for a vertex created by the contraction of an edge, the class `coma_value` is extended. One way to do this, is to derive a class `fs_value` that contains a pointer to a container structure storing all values needed for the calculation of the threshold. Listing 9 contains the definitions for these two classes.

The `coma_value` class is only extended by adding a pointer to a container object called `fs_container`. This way no other changes need to be applied to the original value class.

When calculating the threshold of vertices connected by an edge, three cases are differentiated:

```
// Extension of coma_value.
template <class VALUE_TYPE=vxl_byte>
class fs_value : public coma_value<VALUE_TYPE, 2> {
public:
  // Pointer to the container used to calculate the threshold
  fs_container_type     *fs_container;

  // destructor
  ~fs_value();
  // constructs a new fs_value object and initializes it
  // using the given value, index and coordinates
  fs_value(const value_type &_value, int _index,
    const int _coordinates[num_coord]);
};

// Definition of the class used for the threshold calculation
template <class FS_VALUE>
class fs_container {
public:
  // Values needed for the threshold (the maximum weight, the
  // parameter 'k', the number of vertices and an index.
  double max_weight;
  int k;
  int count;
  int index;

  // Constructor setting k and the unique index
  fs_container(int _k, int _index);

  // Functions used for the calculation of the threshold:
  // The internal contrast, the internal contrast of two
  // components, and the function tau
  const double INT(void) const;
  const double MINT(const fs_container_type &c) const;
  const double tau() const;

  // Function for adding a single vertex to this container
  void add_value(const double weight, fs_value_type *v);
  // Function for merging two containers
  void merge_container(fs_container_type *c);
};
```

Listing 9: Extension of the class `coma_value` for a new algorithm

1. Both vertices represent a single pixel. In this case the edge may be contracted if the weight of the edge is smaller or equal than the global parameter k.

2. One vertex represents a single pixel, while the other one represents a component c of more than one pixel. In this case, the edge may be contracted if the weight of the edge is smaller or equal to $\min(k, INT(c) + \tau(c))$. $\tau()$ is k divided by the number of pixels, and $INT()$ is equal to the largest weight of an edge that has been contracted for merging pixels belonging to c.

3. The vertices represent two components c_1, c_2 of more than one pixel. In this case the edge may be contracted if the weight of the edge is smaller or equal than $MINT(c_1, c_2)$, with $MINT(c_1, c_2) = \min(INT(c_1) + \tau(c_1), INT(c_2) + \tau(c_2))$.

The functions $INT()$, $MINT()$, and $\tau()$ are implemented in the class `fs_container`. Additionally the functions `merge_container` and `add_value` are provided to update the number of pixels and the maximum weight whenever an edge is contracted.

4.2 Implementing the Algorithm Step

The implementation of the algorithm itself is done by creating `fs_pyramid`, a class that is derived from `coma_pyramid`. To program a new way of choosing which vertices shall be merged, `construct_c_kernel_by_algorithm` is overwritten in the new class, as this function is responsible for the algorithm step that selects the edges that will be contracted when building the next level of the pyramid.

For the algorithm described in this Section, the algorithm step is divided into two functions: `construct_c_kernel_by_algorithm` and `compare_darts`.

An implementation of `construct_c_kernel_by_algorithm` is shown in listing 10. First an ordered list is created by processing all darts of the combinatorial map using an iterator on the array "d". Each dart and the weight of the edge the darts are incident to is inserted into a standard C++ map, which will automatically create an ordered list. A second iterator is

```
template<class COMA>
coma_contraction_kernel<typename fs_pyramid<COMA>::dart_type>
  *fs_pyramid<COMA>::construct_c_kernel_by_algorithm(COMA *_map){

  // Define iterators and temporary variables
  typename COMA::coma_array_type::iterator it;
  dart_type *dart;
  std::multimap<double,dart_type *> o;
  typename std::multimap<double,dart_type *>::iterator o_it;

  // Build a list of all edges in this combinatorial map that is
  // sorted by their weight
  for (it=map->d().begin();it!=map->d().end();it++) {
    o.insert(std::pair<double, dart_type*>(
      it.dart->value()->distance(*(it.dart->alpha()->value())),
      it.dart));
  }

  // Process all darts in the sorted list and call
  // compare_darts for each to check the three cases
  for (o_it=o.begin();o_it!=o.end();++o_it) {
    dart = o_it->second;

    if (compare_darts(dart, dart->alpha()) != 0) {
      // An edge that is selected for contraction by
      // compare_darts will always pass the the conditions
      // for contraction. Therefore the return value of
      // add_dart does not need to be checked.
      c_kernel->add_dart(dart, T_EDGE);
    }
  }

  return c_kernel;
}
```

Listing 10: Implementing the algorithm step

then used to call `compare_darts` for each dart within this ordered list. The edge is added to the contraction kernel based on the result of this function.

Listing 11 contains the implementation of the function `compare_darts`. First it determines whether each of the two vertices represents either a single pixel or a region of more than one pixel by retrieving the associated `fs_container` objects. The weight of the edge is then compared against

the correct internal contrast according to the 3 distinguished cases of the algorithm described at the beginning of this Section. This is done using the functions provided by the container object. If the weight is smaller or equal to the internal contrast, the containers are updated and 1 is returned to indicate that the edge should be added to the contraction kernel.

```
template<class COMA>
int fs_pyramid<COMA>::compare_darts(dart_type *dart1,
                                    dart_type *dart2) {
  // Temporary variables.
  double single_m_int = 0;
  dart_type *tmp = NULL;

  // Select the containers associated with each vertex
  // connected by the edge. These functions will return
  // NULL if the vertex represents only a single pixel.
  fs_container_type *c1 = dart1->value()->fs_container;
  fs_container_type *c2 = dart2->value()->fs_container;

  // Calculate the weight of the current edge.
  double weight = dart1->value()->distance(*(dart2->value()));

  // Case 3: Both vertices represent more than one pixel
  if ( c1 != NULL && c2 != NULL ) {
    // In the case of an edge self-loop, the edge is incident to
    // the same vertex twice. In this case both containers have
    // the same index and the edge is not added to the kernel
    if ( c1->index == c2->index ) return 0;

    // Compare the weight of the edge to the internal contrast
    // of the two vertices by using the MINT function provided
    // by fs_container
    if ( weight <= c1->MINT(*c2) ) {
      // The two vertices are merged by contracting the edge.
      // First the weight of the edge is added to the container.
      c1->add_value(weight, NULL);

      // Then the containers are merged and one is deleted.
      c1->merge_container(c2);
      delete c2;

      return 1;
    }
```

```
    return 0;
  }

  // Case 2: One vertex represents more than one pixel.
  if ( c1 != NULL || c2 != NULL ) {
    // For easier processing c1 is set to always holds the
    // fs_container object. The darts are swapped if necessary.
    if ( c2 ) {
      c1 = c2; c2 = NULL;
      tmp = dart1; dart1 = dart2; dart2 = tmp;
    }

    // The internal contrast is calculated using the INT and
    // tau functions provided by fs_container.
    single_m_int = (double)c1->INT()+c1->tau();

    // If the weight of the edge is smaller or equal to the
    // internal contrast the edge may be contracted.
    if ( weight <= single_m_int ) {
      // Add the weight to the container.
      c1->add_value(weight, dart2->value());

      return 1;
    }
    return 0;
  }

  // Case 1: Both vertices represent a single pixel.
  // The edge may be contracted if the weight is smaller than
  // parameter 'k'. 'settings' is a struct of fs_pyramid storing
  // the global parameter 'k' and the last index for a container.
  if ( weight <= settings->k ) {
    // Create a new container with a unique index
    c1 = dart1->value()->fs_container =
      new fs_container_type(settings->k,settings->index++);

    // The weight of the edge is added.
    c1->add_value(weight, dart2->value());

    return 1;
  }
  return 0;
}
```

Listing 11: Calculating the threshold

5 Experiments

An implementation of a connected components algorithm based on COMA is presented for the experiments of this report. This algorithm is using no permutation specific operations. It is therefore suitable to demonstrate how a program that is based on COMA can be built for input data of an arbitrary dimension. This algorithm will also be used to give a performance comparison of processing 2D and 3D data with combinatorial maps using COMA.

5.1 Experimental Setup

The connected components algorithm used for the experiments is modified, as it would otherwise merge all neighboring pixels/voxels of the same color in one step. This would lead to very low pyramids that would only consist of few levels. Instead the modified algorithm selects at most one edge to be contracted for each vertex. This is done by iterating through all edges of the map in a random order. An edge gets selected if it connects two vertices with the same label and neither of these two vertices is incident to an edge that is already part of the contraction kernel (Different strategies for selecting edges to be contracted are presented in great detail in [19, 18, 30]). All other edges of a vertex will be kept until the next level. All simplification steps will be applied before the next level is constructed.

Listing 12 shows the algorithm step for this algorithm. This implementation uses only functions that operate on i-cell. It is therefore independent of the set of permutations used for the combinatorial map or its dimension. (Note, that this is only true if the kind of i-cells exists for the given dimensionality. E.g. working with volumes is not possible for a 2D combinatorial map).

5.2 Experimental Results

In order to study the performance of COMA, 2D and 3D combinatorial pyramids for initial maps of different sizes that representing the same configuration are compared. The bottom-up construction times as well as the allocated memory for thebuilding process is measured and compared. The sizes of the initial maps are chosen in such a way, that the 2D and 3D combinatorial

```
template<class COMA>
coma_contraction_kernel<typename COMA::dart_type>
*coma_pyramid<COMA>::construct_c_kernel_by_algorithm(COMA *_map){

  // Define two iterators to process the edges of the map
  typename COMA::coma_array_type::iterator
    it, it_end = _map->topology_array(T_EDGE).end();

  // one dart for each vertex connected by an edge
  dart_type *vertex_dart1, *vertex_dart2;

  // Process all edges of the map using an iterator
  for (it=_map->topology_array(T_EDGE).begin();it!=it_end;++it) {

    // Assign one dart for each vertex to the temporary variables
    // Using the function 'vertex_by_edge' will return the next
    // vertex belonging to the same edge. It can therefore be
    // used independent of the underlying model for a
    // combinatorial map or the dimension of the input data.
    vertex_dart1 = (dart_type*)it.dart;
    vertex_dart2 = (dart_type*)it.dart->vertex_by_edge();

    // Compare of of both vertices
    if (compare_darts(it.dart,
                      (dart_type*)it.dart->vertex_by_edge())) {

      // If they are identical, the edge is added to the kernel
      c_kernel->add_dart(it.dart, T_EDGE)
    }
  }

  return c_kernel;;
}
```

Listing 12: A connected component algorithm for 2D and 3D

pyramids are comparable: For each 2D map a 3D map with approximately the same number of vertices is selected for the experiments. Table 2 lists the 4 pairs of pyramids used in the experiments.

Table 3 and Table 4 list the bottom-up construction times for the 2D and 3D combinatorial pyramids. The times are given for the accumulated construction times up to each level and for each level alone. These times are

Number of vertices	2D initial map		3D initial map	
	size	darts	size	darts
≈ 125	11×11	520	$5 \times 5 \times 5$	3.072
≈ 1.000	32×32	4.216	$10 \times 10 \times 10$	25.272
≈ 8.000	89×89	32.032	$20 \times 20 \times 20$	199.272
≈ 64.000	253×253	257.040	$40 \times 40 \times 40$	1.569.672

Table 2: Sizes of the four pairs of 2D and 3D combinatorial pyramids for the experiments.

also shown in Figure 6. As can be seen by comparing the diagrams for the 2D pyramids (Figure 6a) and for the 3D pyramids (Figure 6b), the times for pyramids with the same number of vertices differ by a factor of ~ 20. This is expected as the number of darts in grid-like 3D combinatorial maps is about 6 times the number of darts needed for a grid-like 2D combinatorial map (a vertex in a grid consists of 24 darts for 3D and 4 darts for 2D). Another reason for the higher times is the additional permutation for 3D combinatorial maps. However, if the factor is taken into consideration the times for 2D and 3D pyramids are similar. This shows, that the performance of algorithms built with COMA is stable with respect to the dimensionality and the size of the input data.

Table 5 and Table 6 show the memory allocated by the construction of 2D and 3D combinatorial pyramids. For each pyramid and level the total amount of memory allocated up to this level as well as the memory used by that level alone (in brackets) is given in kilobytes. A graphical comparison is shown in Figure 7.

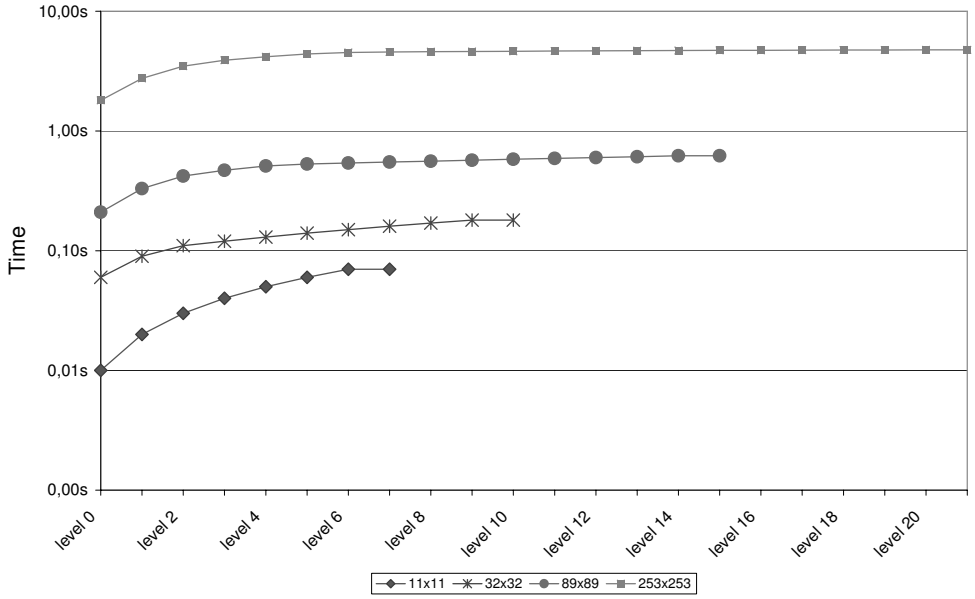
The amount of memory used for a pyramid is mainly dependent on the number of darts in the initial combinatorial map. This can be seen by comparing the memory used for 2D and 3D pyramids with initial maps containing the same number of vertices. The 3D pyramids are about 4 to 5 times larger than the 2D pyramids. This corresponds to the number of darts for these initial maps (there are 6 times more darts per vertex for 3D), when taking into account, that additional memory is needed by COMA to store the values associated with each vertex and for some internal structures to maintain the sets of darts and i-cells. This additional memory is constant for 2D and 3D pyramids.

level	11×11	32×32	89×89	253×253
0	0,010s (0,010s)	0,060s (0,060s)	0,210s (0,210s)	1,800s (1,800s)
1	0,020s (0,010s)	0,090s (0,030s)	0,330s (0,120s)	2,750s (0,950s)
2	0,030s (0,010s)	0,110s (0,020s)	0,420s (0,090s)	3,480s (0,730s)
3	0,040s (0,010s)	0,120s (0,010s)	0,470s (0,050s)	3,900s (0,420s)
4	0,050s (0,010s)	0,130s (0,010s)	0,510s (0,040s)	4,170s (0,270s)
5	0,060s (0,010s)	0,140s (0,010s)	0,530s (0,020s)	4,400s (0,230s)
6	0,070s (0,010s)	0,150s (0,010s)	0,540s (0,010s)	4,530s (0,130s)
7	0,070s (0,000s)	0,160s (0,010s)	0,550s (0,010s)	4,570s (0,040s)
8	**0,070s (0,000s)**	0,170s (0,010s)	0,560s (0,010s)	4,600s (0,030s)
9		0,180s (0,010s)	0,570s (0,010s)	4,610s (0,010s)
10		**0,180s (0,000s)**	0,580s (0,010s)	4,630s (0,020s)
11			0,590s (0,010s)	4,660s (0,030s)
12			0,600s (0,010s)	4,670s (0,010s)
13			0,610s (0,010s)	4,680s (0,010s)
14			0,620s (0,010s)	4,700s (0,020s)
15			**0,620s (0,000s)**	4,710s (0,010s)
16				4,720s (0,010s)
17				4,730s (0,010s)
18				4,740s (0,010s)
19				4,750s (0,010s)
20				4,760s (0,010s)
21				**4,760s (0,000s)**

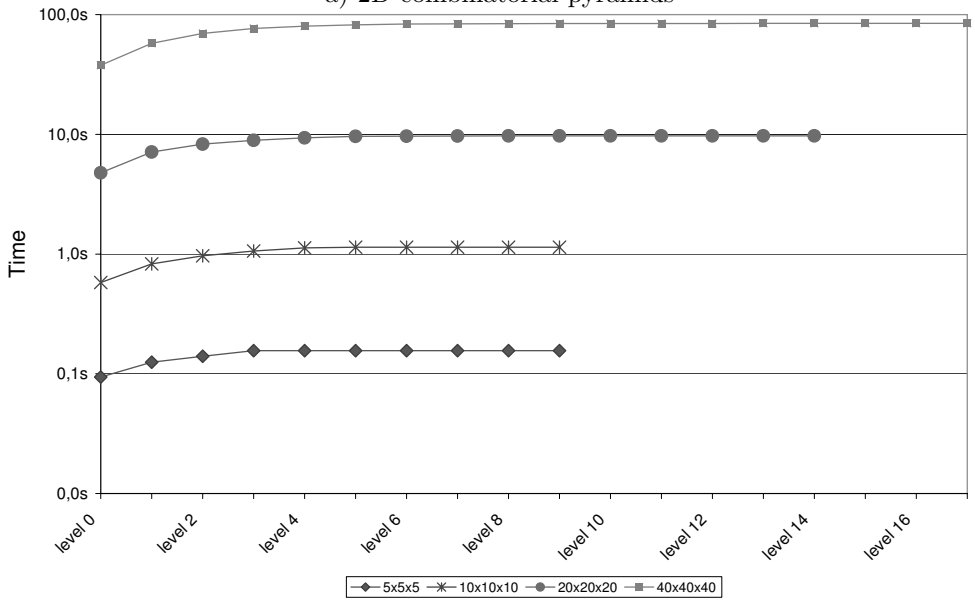
Table 3: Bottom-up construction times of 2D combinatorial pyramids. For each pyramid and level the time is given as the accumulated total construction time up to this level and as the time for each level alone.

level	$5 \times 5 \times 5$	$10 \times 10 \times 10$	$20 \times 20 \times 20$	$40 \times 40 \times 40$
0	0,094s (0,094s)	0,578s (0,578s)	4,766s (4,766s)	37,812s (37,812s)
1	0,125s (0,031s)	0,828s (0,250s)	7,125s (2,359s)	57,578s (19,766s)
2	0,140s (0,015s)	0,968s (0,140s)	8,281s (1,156s)	69,734s (12,156s)
3	0,156s (0,016s)	1,062s (0,094s)	8,906s (0,625s)	76,625s (6,891s)
4	0,156s (0,000s)	1,125s (0,063s)	9,359s (0,453s)	80,187s (3,562s)
5	0,156s (0,000s)	1,140s (0,015s)	9,609s (0,250s)	82,265s (2,078s)
6	0,156s (0,000s)	1,140s (0,000s)	9,656s (0,047s)	83,343s (1,078s)
7	0,156s (0,000s)	1,140s (0,000s)	9,687s (0,031s)	83,843s (0,500s)
8	0,156s (0,000s)	1,140s (0,000s)	9,702s (0,015s)	84,093s (0,250s)
9	**0,156s (0,000s)**	**1,140s (0,000s)**	9,702s (0,000s)	84,218s (0,125s)
10			9,702s (0,000s)	84,296s (0,078s)
11			9,702s (0,000s)	84,359s (0,063s)
12			9,702s (0,000s)	84,374s (0,015s)
13			9,702s (0,000s)	84,405s (0,031s)
14			**9,702s (0,000s)**	84,421s (0,016s)
15				84,421s (0,000s)
16				84,421s (0,000s)
17				**84,421s (0,000s)**

Table 4: Bottom-up construction times of 3D combinatorial pyramids. For each pyramid and level the time is given as the accumulated total construction time up to this level and as the time for each level alone.



a) 2D combinatorial pyramids



b) 3D combinatorial pyramids

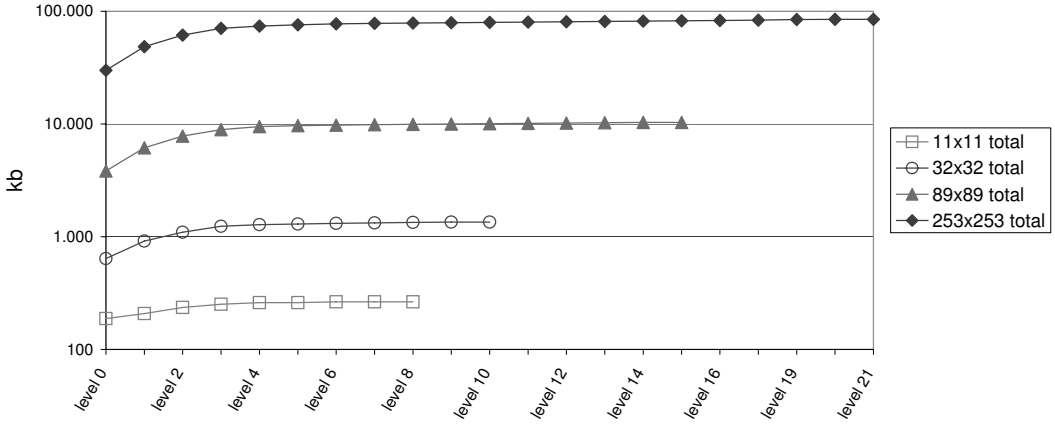
Figure 6: Bottom-up construction times (given in seconds) for 2D (a) and 3D (b) combinatorial pyramids with initial maps of different sizes.

level	11×11	32×32	89×89	253×253
0	1.996 (1.996)	2.924 (2.924)	9.560 (9.560)	58.994 (58.994)
1	2.016 (20)	3.200 (276)	11.876 (2.316)	77.504 (18.510)
2	2.044 (28)	3.380 (180)	13.548 (1.672)	90.584 (13.080)
3	2.060 (16)	3.520 (140)	14.652 (1.104)	99.548 (8.964)
4	2.068 (8)	3.564 (44)	15.228 (576)	102.940 (3.392)
5	2.068 (0)	3.580 (16)	15.396 (168)	104.996 (2.056)
6	2.072 (4)	3.596 (16)	15.504 (108)	106.308 (1.312)
7	2.072 (0)	3.612 (16)	15.592 (88)	107.228 (920)
8	**2.072 (0)**	3.624 (12)	15.668 (76)	107.776 (548)
9		3.632 (8)	15.740 (72)	108.296 (520)
10		**3.632 (0)**	15.804 (64)	108.816 (520)
11			15.868 (64)	109.336 (520)
12			15.936 (68)	109.856 (520)
13			16.000 (64)	110.368 (512)
14			16.064 (64)	110.884 (516)
15			**16.064 (0)**	111.392 (508)
16				111.904 (512)
17				112.420 (516)
18				112.932 (512)
19				113.444 (512)
20				113.956 (512)
21				**113.956 (0)**

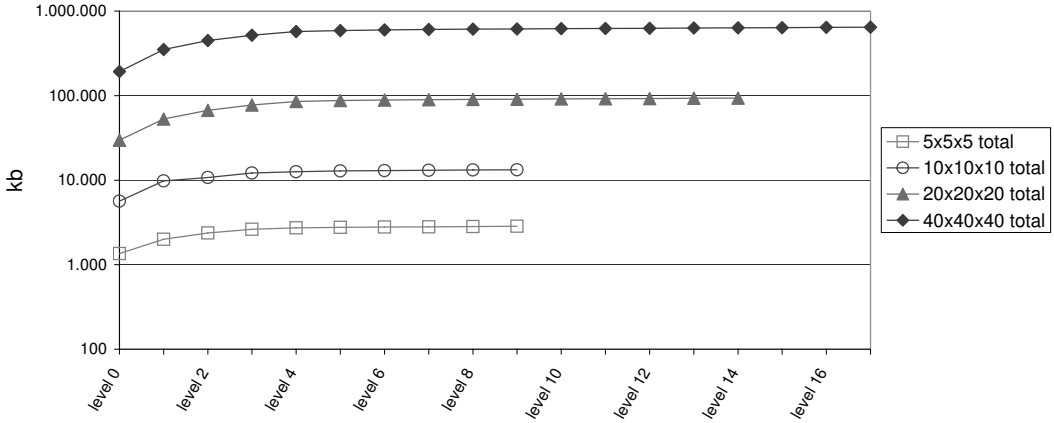
Table 5: Memory (specified in kilobytes) allocated by 2D pyramids.

level	$5 \times 5 \times 5$	$10 \times 10 \times 10$	$20 \times 20 \times 20$	$40 \times 40 \times 40$
0	1.356 (1.356)	5.640 (5.640)	29.720 (29.720)	192.640 (192.640)
1	2.004 (648)	9.820 (4.180)	52.944 (23.224)	351.660 (159.020)
2	2.372 (368)	10.728 (908)	66.980 (14.036)	449.256 (97.596)
3	2.628 (256)	12.156 (1.428)	77.356 (10.376)	518.000 (68.744)
4	2.728 (100)	12.584 (428)	85.148 (7.792)	574.056 (56.056)
5	2.768 (40)	12.864 (280)	87.448 (2.300)	589.832 (15.776)
6	2.788 (20)	13.004 (140)	88.616 (1.168)	599.724 (9.892)
7	2.808 (20)	13.108 (104)	89.596 (980)	606.512 (6.788)
8	2.828 (20)	13.200 (92)	90.340 (744)	611.632 (5.120)
9	**2.848 (20)**	**13.268 (68)**	90.868 (528)	615.908 (4.276)
10			91.396 (528)	619.696 (3.788)
11			91.920 (524)	623.408 (3.712)
12			92.444 (524)	627.040 (3.632)
13			92.964 (520)	630.628 (3.588)
14			**93.488 (524)**	634.192 (3.564)
15				637.760 (3.568)
16				641.328 (3.568)
17				**644.896 (3.568)**

Table 6: Memory (specified in kilobytes) allocated by 3D pyramids.



a) total memory allocation in kilobytes for 2D pyramids of different sizes



a) total memory allocation in kilobytes for 3D pyramids of different sizes

Figure 7: Memory (in kilobytes) allocated for 2D (a) and 3D (b) pyramids.

6 Conclusion and Outlook

In this technical report, COMA is presented. COMA is a C++ library with the goal of providing an efficient framework for implementing software that uses combinatorial maps. For this purpose an object oriented architecture has been designed, that encapsulates the different functionalities needed by such programs. This architecture is explained and it is shown that only few classes must be adapted when new functionalities are introduced, like adding a new set of permutations, supporting new input or output formats or implementing a new algorithm. Additionally a comprehensive overview of each class within COMA is provided, demonstrating the purpose of each class and giving examples for using each class.

To demonstrate the capabilities COMA provides for the task of programming a new algorithm, an example algorithm implemented with COMA is presented. It is shown that only two classes of the framework must be extended in order to build an algorithm with COMA, and that the complexity of the combinatorial maps itself is completely transparent to the algorithm.

Finally experimental results based in a connected component algorithm that was implemented using COMA are presented. It is shown, that this algorithm works for different sets of permutations and dimensionality of the input data without having to change the source code. Performance measurements in terms of bottom up construction times as well as allocated memory are presented for 2D and 3D combinatorial pyramids built with this algorithm. These numbers demonstrate, that the performance is mainly dependent on the number of darts and the number of permutations used for a specific model of combinatorial maps.

The examples and experiments show, that COMA can be used to set up new experiments using different permutations, operations or algorithms in a fast and efficient way, thus speeding up the process of verifying theoretical results in this field. Further they show, that algorithms that do not use the permutations of a combinatorial map directly, will work on any model of combinatorial map. Therefore COMA allows to compare different models and operations directly, by invoking the same algorithm for each model.

The measured performance during the experiments also indicates, that COMA in its current version has high demands regarding processing time and memory allocation (e.g. using 645 megabytes of memory and 84 seconds to

construct a combinatorial pyramid for an initial $40 \times 40 \times 40$ 3D combinatorial map). While this is sufficient for scientific experiments, it might be too high for real-time applications. Future work based on these results will therefore include optimizing COMA. This will include using parallel computing as well as using prebuilt maps to accelerate building the initial grid-like maps that are the dominating factor of the total construction time.

A Combinatorial Maps

This appendix contains details of combinatorial maps used in this technical report. The set of darts D is given as well as the permutations.

A.1 Darts and Permutations of G_1

Details of G_1:

D = { 16, 17, -3, 3, 18, 19, -16, -5, 5, 20, 21, -18, -7, 7, 22, 23, -20, -9, 9, 24, -24, 25, -22, 30, 31, -28, 28, -17, 32, 33, -30, -19, 34, 35, -32, -21, 36, 37, -34, -23, 38, -38, 39, -36, -25, 44, 45, -42, 42, -31, 46, 47, -44, -33, 48, 49, -46, -35, 50, 51, -48, -37, 52, -52, 53, -50, -39, 58, 59, -56, 56, -45, 60, 61, -58, -47, 62, 63, -60, -49, 64, 65, -62, -51, 66, -66, 67, -64, -53, 72, -70, 70, -59, 74, 75, -75, -72, -61, 76, 77, -77, -74, -63, 78, 79, -79, -76, -65, 81, -81, -78, -67 }

α : {-81 81}, {3 -3}, {5 -5}, {7 -7}, {9 -9}, {-24 24}, {28 -28}, {-38 38}, {42 -42}, {-52 52}, {56 -56}, {-66 66}, {70 -70}, {-75 75}, {-77 77}, {-79 79}, {16 -16}, {17 -17}, {18 -18}, {19 -19}, {20 -20}, {21 -21}, {22 -22}, {23 -23}, {25 -25}, {30 -30}, {31 -31}, {32 -32}, {33 -33}, {34 -34}, {35 -35}, {36 -36}, {37 -37}, {39 -39}, {44 -44}, {45 -45}, {46 -46}, {47 -47}, {48 -48}, {49 -49}, {50 -50}, {51 -51}, {53 -53}, {58 -58}, {59 -59}, {60 -60}, {61 -61}, {62 -62}, {63 -63}, {64 -64}, {65 -65}, {67 -67}, {72 -72}, {74 -74}, {76 -76}, {78 -78}

σ : {-81 3 5 7 9 -24 28 -38 42 -52 56 -66 70 -75 -77 -79}, {16 17 -3}, {18 19 -16 -5}, {20 21 -18 -7}, {22 23 -20 -9}, {24 25 -22}, {30 31 -28 -17}, {32 33 -30 -19}, {34 35 -32 -21}, {36 37 -34 -23}, {38 39 -36 -25}, {44 45 -42 -31}, {46 47 -44 -33}, {48 49 -46 -35}, {50 51 -48 -37}, {52 53 -50 -39}, {58 59 -56 -45}, {60 61 -58 -47}, {62 63 -60 -49}, {64 65 -62 -51}, {66 67 -64 -53}, {72 -70 -59}, {74 75 -72 -61}, {76 77 -74 -63}, {78 79 -76 -65}, {81 -78 -67}

A.2 Darts and Permutations of G_2

Details of G_1:

D = { 17, 3, 18, 19, -5, 21, -18, 7, 22, -9, -24, -22, 30, -28, -17, 32, 33, -30, -19, 35, -32, 36, 37, -23, 38, 39, -36, 44, 45, 42, 47, -44, -33, 48, 49, -35, 51, -48, -37, 52, -52, 53, -39, 58, -56, -45, 60, 61, -58, -47, -60, -49, -62, -51, -66, -64, -53, 72, 70, 74, 75, -75, -72, -61, 76, 77, -77, -74, 78, 79, -79, 81 }

α : {17 -17}, {3 -5}, {18 -18}, {19 -19}, {21 -23}, {7 -9}, {22 -22}, {-24 38}, {30 -30}, {-28 42}, {32 -32}, {33 -33}, {35 -35}, {36 -36}, {37 -37}, {39 -39}, {44 -44}, {45 -45}, {47 -47}, {48 -48}, {49 -49}, {51 -51}, {52 -52}, {53 -53}, {58 -58}, {-56 70}, {60 -60}, {61 -61}, {-62 76}, {-66 81}, {-64 78}, {72 -72}, {74 -74}, {75 -75}, {77 -77}, {79 -79}

σ : {17 18 19 -5}, {3 7 -24 42 -52 -66 70 -75 -77 -79}, {21 -18 22 -9}, {-22 38 39 -36}, {30 -28 -17 44 45}, {32 33 -30 -19}, {35 -32 36 37 -23}, {47 -44 -33 48 49 -35}, {51 -48 -37 52 53 -39}, {58 -56 -45 72}, {60 61 -58 -47}, {-60 -49 76 77 -74}, {-62 -51 78 79}, {-64 -53 81}, {74 75 -72 -61}

References

[1] S. Ansaldi, L. de Floriani, and B. Falcidieno. Geometric Modeling of Solid Objects by Using a Face Adjacency Graph Representation. *Computer Graphics*, 19(3):131–139, 1985.

[2] B. Baumgart. A Polyhedron Representation for Computer Vision. In *AFIPS National Computer Conferenc Proc.*, volume 44, pages 589–596, Anaheim, May 1975.

[3] Y. Bertrand, G. Damiand, and C. Fiorio. Topological Encoding of 3D Segmented Images. In G. Borgefors, I. Nyström, and G. S. di Baja, editors, *International Conference on Discrete Geometry for Computer Imagery*, volume 1953 of *Lecture Notes in Computer Science*, pages 311–324. Springer-Verlag, Germany, 2000.

[4] A. Braquelaire, G. Damiand, J.-P. Domenger, and F. Vidil. Comparison and convergence of two topological models for 3d image segmentation. In *Workshop on Graph-Based Representations in Pattern Recognition*, number 2726 in Lecture Notes in Computer Science, pages 59–70, York, England, June 2003.

[5] E. Brisson. Representing Geometric Structures in D Dimensions: Topology and Order. *Discrete and Computational Geometry*, 9:387–426, 1993.

[6] S. Bruckner. Efficient volume visualization of large medical datasets. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, May 2004.

[7] L. Brun and W. G. Kropatsch. Irregular Pyramids with Combinatorial Maps. In F. J. Ferri, J. M. Iñesta, A. Amin, and P. Pudil, editors, *Advances in Pattern Recognition, Joint IAPR International Workshops on SSPR'2000 and SPR'2000*, volume 1876 of *Lecture Notes in Computer Science*, pages 256–265, Alicante, Spain, August 2000. Springer, Berlin Heidelberg, New York.

[8] P. Cavalcanti, P. Carvalho, and L. Martha. Non-manifold Modeling: An Approach Based on Spatial Subdivision. *Computer-Aided Design*, 29(3):209–220, 1997.

[9] G. Crocker and W. Reinke. An Editable Nonmanifold Boundary Representation. *Computer Graphics and Applications*, 11(2):39–51, 1991.

[10] G. Damiand. *Définition et étude d'un modèle topologique minimal de représentation d'images 2d et 3d*. Thèse de doctorat, Université Montpellier II, Décembre 2001.

[11] G. Damiand and F. Vidil. Moka - modeleur de cartes. http://www.sic.sp2mi.univ-poitiers.fr/moka/, 2006.

[12] L. De Floriani, E. Puppo, and P. Magillo. A Formal Approach to Multiresolution Hypersurface Modeling. In R. Straber, W. andKein and R. Rau, editors, *Geometric Modeling: Theory and Practice*, pages 302–323. Springer-Verlag, 1997.

[13] D. Dobkin and M. Laszlo. Primitives for the manipulation of three-dimensional subdivisions. *Algorithmica*, 4(1):3–32, 1989.

[14] P. F. Felzenszwalb and D. P. Huttenlocher. Image Segmentation Using Local Variation. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 98–104, June 1998.

[15] V. Ferruci and A. Paoluzzi. Extrusion and Boundary Evaluation for Multidimensional Polyhedra. *Computer-Aided Design*, 23(1):40–50, 1991.

[16] S. Grimm, S. Bruckner, A. Kanitsar, and M. E. Gröller. A refined data addressing and processing scheme to accelerate volume raycasting. *Computers & Graphics*, 28(5):719–729, 2004.

[17] L. Guibas and J. Stolfi. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. *ACM Trans. on Graphics*, 4(2):74–123, 1985.

[18] Y. Haxhimusa, R. Glantz, and W. G. Kropatsch. Constructing Stochastic Pyramids by MIDES - Maximal Independent Directed Edge Set. In E. Hancock and M. Vento, editors, *4th IAPR-TC15 Workshop on Graph-based Representation in Pattern Recognition*, volume 2726 of *Lecture Notes in Computer Science*, pages 35–46, York, UK, June-July 2003. Springer, Berlin Heidelberg, New York.

[19] Y. Haxhimusa, R. Glantz, M. Saib, G. Langs, and W. G. Kropatsch. Logarithmic tapering graph pyramid. In *Proceedings of the 24th DAGM Symposium on Pattern Recognition*, pages 117–124, London, UK, 2002. Springer-Verlag.

[20] Y. Haxhimusa, A. Ion, W. G. Kropatsch, and L. Brun. Hierarchical Image Partitioning using Combinatorial Maps. In D. Chetverikov, L. Czuni, and M. Vincze, editors, *Joint Hungarian-Austian Conference on Proceedings on Image Processing and Pattern Recognition, HACIPPR 2005 - OAGM 2005/KPAF 2005*, pages 179–186, Veszprm, Hungary, 11-13, May 2005. OCG.

[21] Y. Haxhimusa, A. Ion, W. G. Kropatsch, and T. Illetschko. Evaluating minimum spanning tree based segmentation algorithms. In A. Gagalowicz and W. Philips, editors, *Proceedings of the 11th International Conference on Computer Analysis of Images and Patterns*, volume 3891 of *Lecture Notes in Computer Science*, pages 579–586, France, September 2005. Springer.

[22] Y. Haxhimusa and W. Kropatsch. Hierarchical Image Partitioning with Dual Graph Contraction. In B. Michaelis and G. Krell, editors, *Proc. of 25th German Association for Pattern Recognition*, volume 2781 of *Lecture Notes in Computer Science*, pages 338–345, Magdeburg, Germany, September 2003. Springer-Verlag.

[23] T. Illetschko. Minimal Combinatorial Maps for analyzing 3D Data. Master's thesis, Pattern Recognition and Image Processing Group, Institute of Computer Aided Automation, Vienna University of Technology, July 2006.

[24] T. Illetschko. Minimal Combinatorial Maps for analyzing 3D Data. Technical Report PRIP-TR-110, Institute f. Computer Aided Automation 183/2, Pattern Recognition and Image Processing Group, TU Wien, Austria, 2006.

[25] T. Illetschko, A. Ion, Y. Haxhimusa, and W. G. Kropatsch. Collapsing 3d combinatorial maps. In O. S. F. Lenzen and M. Vincze, editors, *Proceedings of 30th Austrian Association for Pattern Recognition Workshop*, pages 85–93. Oesterreichische Computer Gesellschaft, March 2006.

[26] T. Illetschko, A. Ion, Y. Haxhimusa, and W. G. Kropatsch. Distinguishing the 3 primitive 3D-topological configurations: simplex, hole, tunnel. In O. Chum and V. Franc, editors, *CVWW'06: Proceedings of the Computer Vision Winter Workshop 2006*, pages 22–27, Prague, Czech Republic, February 2006. Czech Society for Cybernetics and Informatics.

[27] V. A. Kovalevsky. Finite topology as applied to image analysis. *Computer Vision, Graphics, and Image Processing*, 46:141–161, 1989.

[28] W. Kropatsch. Building Irregular Pyramids by Dual Graph Contraction. *IEE-Proc. Vision, Image and Signal Processing*, 142(6):366–374, 1995.

[29] W. G. Kropatsch, Y. Haxhimusa, and P. Lienhardt. Hierarchies relating topology and geometry. In I. H. I. Christensen and H.-H. Nagel, editors, *Cognitive Vision Systems*, volume 3948, pages 199–220. Springer Verlag, 2006.

[30] W. G. Kropatsch, Y. Haxhimusa, Z. Pizlo, and G. Langs. Vision pyramids that do not grow too high. *Pattern Recognition Letters*, 26(3):319–337, 2005.

[31] P. Lienhardt. N-dimensional Generalized Combinatorial Maps and Cellular Quasi-manifolds. *Int. J. of Comp. Geom. and Appl.*, 4(3):275–324, 1994.

[32] A. Paoluzzi, F. Bernardini, C. Cattani, and V. Ferrucci. Dimension Independent Modeling with Simplicial Complexes. *ACM Trans. on Graphics*, 12(1):56–102, 1993.