Introduction to the Color Structure Code and its Implementation

Lutz Priese, Patrick Sturm

March 24, 2003

1 Introduction

One of the most important tasks of an image analysis system is image segmentation, the identification of homogeneous regions in an image. In the literature several methods for segmentation are distinguished. Common are *edge detection*, *split and merge*, *region growing* and *clustering techniques*. Most of the extensive research on image segmentation in the last three decades has been done for gray scale images. However, as the technical equipment for color acquisition becomes cheaper and more common, color image analysis becomes more and more important. Nearly all techniques for gray scale image segmentation have been transferred to color images. Most papers on color segmentation follow the clustering method. Here the pixels are mapped to feature vectors in a feature space. Now statistical methods are applied to find some clusters in this feature space. These clusters, re-mapped to the image, form the color segments. A well-known clustering technique is recursive histogram splitting, applied by many researchers. The advantage of clustering methods is the global view of the data often in form of histograms. However, although histograms provide a global view of the feature data, they do not represent the spatial information of the underlying image. The extension of clusters in feature space is often ambiguous and the statistical methods trying to solve this problem are computationally expensive. Region growing techniques start with initial cells, pixels or small regions. The pure local methods tend to chaining mismatches by merging differently colored segments. The centroid-linkage techniques are sequential methods and are therefore dependent on the choice of starting point and the order in which the pixels are processed. The CSC is a method combining the advantages of local (simplicity and fastness) and global (robustness and accuracy) techniques. It is a hierarchical region growing method that is inherently parallel and therefore independent of the choice of the starting point and the order of processing. It uses local and global information and achieves very robust segmentation results in natural color scenes. This paper describes the color segmentation system called CSC (Color Structure Code) and its implementation.

2 The Hexagonal Island Hierarchy

The CSC follows a hierarchical region growing on a special hexagonal topology that was firstly introduced by Hartmann [HA87]. This hierarchical topology (see figure 1)

is formed by so called *islands* of different levels. One island of level 0 consists of seven pixels (one center pixels and its 6 neighbors). The partition of the image is organized in such a way that the islands are overlapping (each second pixel of each second row is a center of an island of level 0). An island of level $n + 1$ is build up in a similar manner: It consists of seven overlapping islands of level n. Repeating this until one island covers the whole image the number of islands decreases from level to level by factor 4.
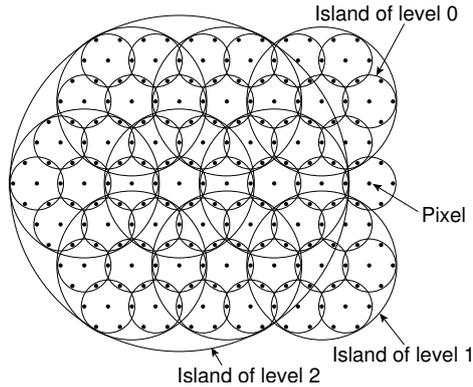

Figure 1: The hexagonal island hierarchy

Operating on a hexagonal topology leads to some difficulties in practice. The most cameras produce images whose pixels are organized in an orthogonal lattice. In order to avoid addition efforts we use the hexagonal island hierarchy only as a logical structure on the orthogonal lattice. To obtain such a logical structure we scale the y-axis of the hexagonal lattice by factor $2/\sqrt{3}$ and move each second row a half length unit to the left (see figure 2a). By that transformation the shape of the hexagonal islands is distorted. The distorted islands can be used directly with the orthogonal lattice. Figure 2b) shows how the hexagonal islands on the orthogonal lattice look like.
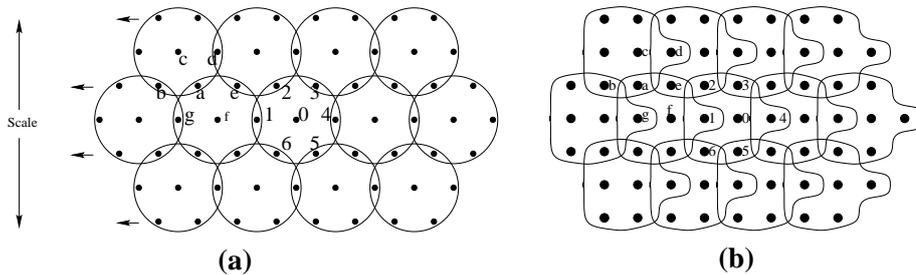

Figure 2: **a** The transformation of the hexagonal lattice. **b** Islands of level 0 upon the orthogonal lattice.

Let us consider the island positions on an orthogonal lattice of size 513×513. The coordinates of an island are identical with the coordinates of its center pixel. All islands are organized in the same way. The first island of each level has always the co-

2

ordinate $(0,0)$. At level 7 this construction is changing. On this level seven islands are sufficient to cover the whole image. Therefore we place the first island at coordinates $(128,0)$. These seven islands are grouped into one island of level 8 (with coordinates $(256,256)$). Figure 3 shows the coordinates of level 0 islands on an orthogonal lattice.
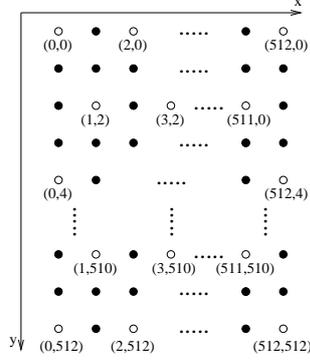
Figure 3: Coordinates of the level 0 islands on an orthogonal lattice of size 513×513

Generally an island hierarchy that is defined on an orthogonal lattice of size $2^m + 1 \times 2^m + 1$ consists of m hierarchy levels. The coordinates of all level n islands are given formally by the function $IC(m,n) : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$:

1. $IC(m,n)$ is defined for all levels n with $n < m - 2$ as:
$$
\begin{aligned}
IC(m,n) \quad := \quad & \{(2^n \cdot 2i, 2^{n+1} \cdot (2j)) \mid i, j \in \mathbb{N},\ 0 \le i \le 2^{m-(n+1)} \wedge 0 \le j \le 2^{m-(n+2)}\} \\
\cup \quad & \{(2^n \cdot (2i+1), 2^{n+1} \cdot (2j+1)) \mid i, j \in \mathbb{N},\ 0 \le i \le 2^{m-(n+1)} \wedge 0 \le j < 2^{m-(n+2)}\}
\end{aligned}
$$

2. On the last but one level ($n = m - 2$) $IC(m,n)$ is defined different:
$$
\begin{aligned}
IC(m,n) \quad := \quad & \{(2^n, 0), (3 \cdot 2^n, 0), (2^n, 2^m), (3 \cdot 2^n, 2^m)\} \\
\cup \quad & \{(0, 2^{m-1}), 2^{m-1}, 2^{m-1}), (2^m, 2^{m-1})\}
\end{aligned}
$$

3. The last level ($n = m - 1$) consists only of one island:
$$
IC(m,n) \quad := \quad \{(2^{m-1}, 2^{m-1})\}
$$

Let us consider an example: The orthogonal lattice is of size 513×513 ($m = 9$, $2^m + 1 = 513$), i.e. the hexagonal island hierarchy consists of 9 levels. In the following we state the island coordinates of level $n = 0$, $n = 1$ and $n = 7$:

$$
\begin{aligned}
IC(9,0) \quad := \quad & \{(2i, 4j) \mid i, j \in \mathbb{N},\ 0 \le i \le 256 \wedge 0 \le j \le 128\} \\
\cup \quad & \{(2i+1, 4j+2) \mid i, j \in \mathbb{N},\ 0 \le i \le 256 \wedge 0 \le j < 128\} \\
= \quad & \{(0,0), (2,0), ..., (512,0), (0,4), (2,4), ..., (512,4), ..., (512,512)\} \\
\cup \quad & \{(1,2), (3,2), ..., (513,2), (1,6), (3,6), ..., (513,6), ..., (513,510)\} \\
IC(9,1) \quad := \quad & \{(4i, 8j) \mid i, j \in \mathbb{N},\ 0 \le i \le 128 \wedge 0 \le j \le 64\} \\
\cup \quad & \{(4i+2, 8j+4) \mid i, j \in \mathbb{N},\ 0 \le i \le 128 \wedge 0 \le j < 64\} \\
= \quad & \{(0,0), (4,0), ..., (512,0), (0,8), (4,8), ..., (512,8), ..., (512,512)\} \\
\cup \quad & \{(2,4), (6,4), ..., (514,4), (2,12), (6,12), ..., (514,12), ..., (514,508)\} \\
IC(9,7) \quad = \quad & \{(128,0), (384,0), (128,512), (384,512)\} \cup \{(0,256), (256,256), (512,256)\}
\end{aligned}
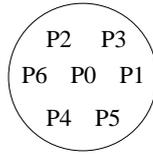$$

3

```
P2   P3
P6  P0  P1
P4   P5
```

Figure 4: Sub-positions of the seven sub-islands resp. sub-pixels within an island

A very important operation is the computation of the seven sub-islands of an island I. Firstly, we assign to each sub-island of I a certain sub-position $p \in \{P_0, ..., P_6\}$ (see fig. 4). A sub-position might be seen as a relative position of a sub-island resp. pixel within its parent island. So if we say the sub-island of island I at sub-position P_0 we reference to the sub-island in the center of I. The coordinates of the sub-islands of I are specified in the following table according to their sub-position. I is of level $n{+}1$ and has the coordinates (x, y):

Sub-Position	X-Coordinate	Y-Coordinate
P_0	x	y
P_1	$x{+}2^{n+1}$	y
P_2	$x{-}2^n$	$y{+}2^{n+1}$
P_3	$x{+}2^n$	$y{+}2^{n+1}$
P_4	$x{-}2^n$	$y{-}2^{n+1}$
P_5	$x{+}2^n$	$y{-}2^{n+1}$
P_6	$x{-}2^{n+1}$	y

3 Color Structure Code

The generation of the CSC operates essentially in three phases. In an *initialization phase* the image is partitioned into small, atomic color regions within islands of level 0. These small color regions are growing in the *linking phase* in a hierarchical manner to complete color segments. Within the linking phase it is possible to detect that color regions connected by a chain of smoothly changing colors have to be split again. This is done in the *splitting phase*.

3.1 Initialization Phase

In the initialization phase color homogeneous regions in level 0 islands of seven pixels are detected and mapped to *initial code-elements*. Such an initial code-element consists of those pixels of level 0 islands that are neighbored and whose colors are similar, i.e. their mutual color distance lies below a certain threshold. A code-element is a data structure describing color regions within an island. In figure 5 an examples is shown: An island, where an edge goes through it, results in two code-elements that describe two differently colored small regions in the island. Hence, a code-element of level 0 describes a small colored region within one island of level 0. Note, this operation is a pure local operation within one island, whose processing can be done independently for each island.
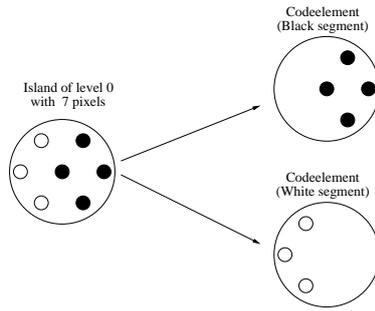
4

Figure 5: Coding of two homogeneous regions into two level 0 code-elements.

Instead of starting with one seed pixel, the CSC starts concurrently in all level 0 islands of the image. The result of the initialization phase is a set of code-elements, each one describing a small color patch. In the following linking phase these small color patches are checked for continuity and grow hierarchically to complete, connected color segments. The data structure of a code-element will be given in section 4.2. At this moment it is sufficient to consider a code-element as a data structure that stores information about the shape and the mean-color of a region.

3.2 Linking Phase

In the linking phase code-elements of level n are linked to new code-elements of level $n{+}1$ in seven, neighbored overlapping islands of the hexagonal island structure. Code-elements will be linked if the regions represented by them are connected and similar in color. The connectivity of code-elements can easily be determined within the hexagonal island structure.
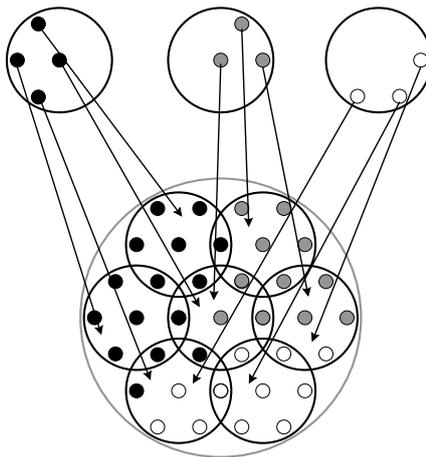

Figure 6: The linking of code-elements.

Two code-elements are connected if they share a common sub-region in their common sub-island. On level 1 this simply means that they share a common pixel. The

5

linking operations are repeated for all islands on every level, starting from level 1 and ending on the topmost level. On the topmost level one island covers the whole image. By repeated linking those connected and color similar code-elements form a code-element-tree. A new code-element c of level n is stored together with pointers to the code-elements on level n–1 from which c was formed, the so called sub-code-elements. Code-elements (root-code-elements) that do not find any partner for linking on some level n form the root of such a code-element-tree. Thus, a connected homogeneous region (segment) is represented by a tree in our CSC data structure. The larger a region is the higher is its root level in the hierarchical data structure. The root contains raw information about the size, location, and mean color of a region. More details can be obtained by descending the tree.

The linking of code-elements within one island is similar to the operation in the initialization phase. Instead of linking single pixels, regions are linked. Again all operations within one island can be done independently of the other islands. The segmentation results are not depending on the order of execution. All small color regions of the initialization phase are growing concurrently within one level. The overlapping of the islands leads to efficient connectivity checks of code-elements. The hexagonal island structure assures that the regions are growing in all directions in contrast to quad-tree-like structure.

The test if two code-elements have a common sub-region can be done by testing whether both code-elements possess a sub-code-pointer on a common sub-code-element. But this test is very time consuming. Therefore two additional pointers are stored with each code-element c. These pointers refer to the parent-code-elements of c (parent-pointers). p is a parent-code-element of s if s is a sub-code-element of p. Due to the overlapping structure of the hexagonal island hierarchy each code-element has at most two different parents. The use of parent-pointers leads to very efficient detection of connected code-elements. If we want to find those code-elements that are connected with a code-element v_1 we do the following: We descend from code-element v_1 to one of its sub-code-elements, let say s. If s has a second parent-pointer v_2 in addition to v_1 then v_1 and v_2 are connected via the common sub-code-element s. This operation is more efficient than searching for a common sub-code-element.

3.3 Splitting Phase

A typical error in local region growing techniques is the linking of differently colored regions due to a chain of connected pixels with smoothly changing colors. Segmentation algorithms that only use local information are unable to detect region boundaries with low contrast. A good segmentation algorithm has to use local and global information. We solve this problem by additional color similarity checks between connected code-elements on every linking level. If the color distance lies above a certain threshold the two code-elements would not be linked although they are connected by a chain of color similar pixels. Consider the example in figure 7. If the color distance between the regions r_1 and r_2 is too high, they would not be linked, although all their sub-regions on level n–1 are locally homogeneous. It is the global view at this level that makes it possible to detect the smooth transition of one color to the other. The fact that r_1 and r_2 would not be linked results in two different complete segments. Due to the overlapping structure they possess a common sub-region, thus $r_1 \cap r_2 \neq \emptyset$. Therefore

6

r_1 and r_2 have to be explicitly separated, which means the common sub-region s has to partitioned between r_1 and r_2.
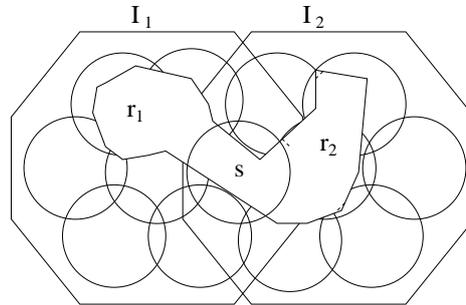

Figure 7: Two regions r_1 and r_2 with their common sub-region s.
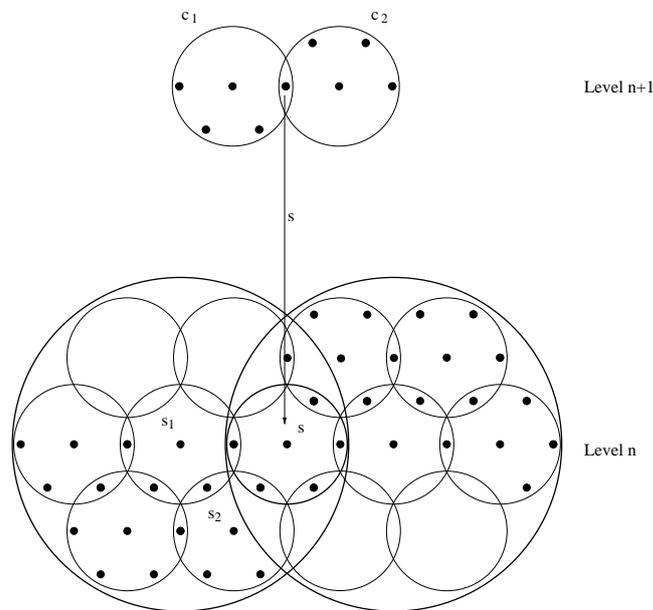

Figure 8: The splitting of two code-elements c_1 and c_2. Both code-elements have a common sub-code-element s.

Consider the general case of Figure 8. Two connected code-elements c_1 and c_2 are not color similar. They possess the common sub-code-element s. s has to be partitioned between c_1 and c_2. At first, the color value of s is compared with those of c_1 and c_2. s assigned to that code-element, say c_1. This is simply a pointer deletion. This does not mean that the whole region represented by s has to be separated from s_1 and s_2. This may not be an accurate border between c_1 and c_2. Although s has been deleted from c_1 it is still connected with the region represented by c_1 via s_1 and s_2 due to the overlapping structure. Now, s has to be separated from s_1 and s_2. This done in a recursive procedure in the same way as with c_1 and c_2. The only difference is that the color values to compare with are those of c_1 and c_2 as they represent the most global

7

information. When splitting s and s_1 it is possible that their common sub-region is assigned to s_1. That is why not necessarily the entire region of s is assigned to c_2. s can loose some sub-regions in the recursive descent. The recursion stops at level 0. With this simple and elegant recursive algorithm (main operation is the deletion of pointers) very accurate borders between segments can be found.

4   Implementation of the CSC

The Color Structure Code (CSC) is implemented in C++ by using object oriented programming techniques like classes and templates. Firstly, a short overview of the most important classes will be given:

1. **ipcIslandHierarchy:** The class `ipcIslandHierarchy` contains some important methods for obtaining informations about the hexagonal island hierarchy. Thus an instant of the class is able to compute for example:

    (a) the sub-islands of a given island,

    (b) the parent islands of a given island,

    (c) the pixels of a given island of level 0 and

    (d) the necessary number of hierarchy levels of a given image of size $Width \times Height$ pixels.

2. **ipcCodeelementBase:** An instance of the class `ipcCodeelementBase` (code-element-base) stores a forest of code-element-trees and allows easy access to them. Also this class provides a method to generate region images resp. labeled images from the code-element-trees. In a region image each pixel is displayed with the mean color of the region it belongs to. A labeled image is a kind of gray image where each pixel obtains a label (gray value) that identifies the region it belongs to.

3. **ipcCSC:** The class *ipcCSC* contains methods that realize the CSC segmentation algorithm.

In the following sections all three classes will be discussed. The classes are not described in detail. But we will show what the purpose of the classes is and how they are used.

4.1   Class: ipcIslandHierarchy

The class `ipcIslandHierarchy` is defined in the files `ipcislhierarchy.h` (Definition) and `ipcislhierarchy.cpp` (Implementation).

It contains useful methods for operating with the hexagonal island hierarchy. Essentially it provides information about the relations between hexagonal islands. In this section we show

1. how a hexagonal island hierarchy is created,

2. how the sub-islands of an island are computed and

3. how the islands of hierarchy can be enumerated.

4.1.1 The structure: ipcIsland

An island is represented by the C++ structure `ipcIsland`. The structure stores the `level` and the coordinates `(x,y)` of an island. All three components are stored as integer values (32 bit):

```
struct ipcIsland {
  int level,x,y;
};
```

4.1.2 Creating the Island Hierarchy

The island hierarchy will be created by calling its constructor. The method has two parameters: the width and the height of the image that should be segmented (starting image). For example if we want to create a hexagonal island hierarchy that covers an image of size 500×400 we have to write

```
ipcIslandHierarchy hierarchy(500,400);
```

In section 2 we defined a hexagonal island hierarchy just for images of size $2^m + 1 \times 2^m + 1$. Thus the constructor changes the size from 500×400 to 513×513 pixels automatically.

4.1.3 Computation of sub-islands and sub-pixels

An important operation on the hexagonal island hierarchy is the computation of the sub-islands of a given island `I`. The method that computes this operation is `getSubIslands`. The method has two parameters. The first parameter denotes the island for that the sub-islands should be computed. The second parameter is an array with seven entries in which the sub-islands of `I` are stored. Let us consider an example. We want to compute the seven sub-islands of the level 8 island `I` with co-ordinates $(256, 256)$. Firstly, we create an island hierarchy object and initialize the island variable `I`:

```
// Creating the island hierarchy object
ipcIslandHierarchy hierarchy(500,400);

// Initialization of an island
ipcIsland I;
I.level=8;
I.x=256;
I.y=256;
```

Then we define the array `sub` which has seven entries to store all sub-islands of `I`. After that, we call the method `getSubIslands` of the object `hierarchy`. This method computes the seven sub-islands and stores them into the array `sub`:

```
// Definition of the sub-island array
icpIsland sub[7];
```

9

```
// Compute the sub-islands of I
hierarchy.getSubIslands(I,sub);
```

After applying this program the array entry `sub[i]` contains the sub-island at the sub-position P_i. Thus `sub[0]` denotes the center island of I. If we want to compute the sub-island at a certain sub-position, lets say P_3, we can also call the method `getSubIsland`. The sought sub-island will be stored in the variable `subIsland`:

```
ipcIsland subIsland;
// Computing the sub-island of I at sub-position P3
hierarchy.getSubIsland(I,3,subIsland);
```

Both methods work only with islands of level $n > 0$. For islands of level 0 one can compute their sub-pixel coordinates instead. This is done by calling the method `getSubPixels`. To store the coordinates of the sub-pixels two arrays have to been created. One array stores the x-coordinates (`xcoord`) and the other the y-coordinates (`ycoord`) of the sub-pixels. Firstly, we initialize the variable `I`. For this example we choose the level 0 island with coordinates $(1, 2)$:

```
// Initialization of an island
ipcIsland I;
I.level=0;
I.x=1;
I.y=2;

// Definition of the arrays storing the coordinates
int xcoord[7];
int ycoord[7];

// Compute the sub-pixels of I
hierarchy.getSubPixels(I,xcoord,ycoord);
```

(`xcoord[i]`,`yccord[i]`) specifies the pixel coordinate of the island `I` at sub-position P_i. Thus (`xcoord[0]`,`yccord[0]`) is the coordinate of the center pixel of I.

4.1.4 Island Iterators

Island iterators are objects that provide methods for iterating the islands of the hexagonal island hierarchy in a certain order. An island iterator can be considered as a kind of pointer that refers to a certain island. Two different kinds of island iterators exist in the implementation: top-down and bottom-up island iterators. The difference between both is the order in which the hierarchy levels are processed. The top-down island iterator starts at the top most level and ends at level 0. Analogical the bottom-up island iterator starts at level 0 and ends at the top most level. Each island iterator stores the level and the coordinates of the island (*current island*) it refers to.

As an example we present a short program that iterates over all islands with a bottom-up island iterator. The level and the coordinates of each island are printed to the console. Firstly, the method `createIteratorBU` of the class `ipcIslandHierarchy` is called to create a bottom-up island iterator. One should not forget to destroy the island iterator object when it is not needed anymore.

```
// Create an island hierarchy
ipcIslandHierarchy hierarchy(500,400);

// Create a bottom-up island iterator
ipcBottomUpIslandIterator* islandIterator =
                            hierarchy.createIteratorBU();
```

After creation, the island iterator has to be initialized by calling its `begin` method.

```
islandIterator->begin();
```

Two nested do-while-loops are necessary to iterate over all islands. The outer do-while-loop iterates over all hierarchy levels. By calling the `nextLevel`-method of the island iterator the program goes to the next hierarchy level. `nextLevel` returns `false` if the iterator already refers to the last hierarchy level. Otherwise it returns true. The inner do-while-loop iterates over all islands of the current level. Therefore the program calls the method `nextIsland` to switch to the next island within the current level. The method returns `false` if the iterator refers to the last island within the current level. Otherwise it returns `true`. The method `currentIsland` returns the current island:

```
// Iterate over all hierarchy levels
do
{
  // Iterate over all islands within the current hierarchy level
  do
  {
    // Get a reference on the current island
    ipcIsland& currentIsland = hierarchy->currentIsland();

    // Print level and coordinates of the current island
    cout << "(Level=" << currentIsland.level
         << ",x=" << currentIsland.x
         << ",y=" << currentIsland.y << ")" << endl;
  } while (islandIterator->nextIsland();
} while (islandIterator->nextLevel());
delete islandIterator;
```

To iterate over all islands in top-down order only a small modification of the program has to be done. Instead of calling the method `createIteratorBU` the program has to call the method `createIteratorTD`. Also the type of the island iterator has to be changed to `ipcTopDownIslandIterator`:

```
// Create a top-down island iterator for island hierarchy
ipcTopDownIslandIterator* islandIterator =
                                   hierarchy.createIteratorTD();
```

Normally a programmer that uses the CSC has not to do anything with island iterators. They are just used within the algorithm.

4.2 The Class: ipcCodeelementBase

The generic class `ipcCodeelementBase<ipcFeatureType>` is responsible for storing the code-element-trees that are generated by the CSC. `ipcFeatureType` is a template parameter that denotes the data type of the colors that are stored within the code-elements. Therefore the implementation is independent of the chosen color space. If a code-element-base should be created the programmer has to choose a concrete data structure for `ipcFeatureType`. The following example demonstrates the instantiation of a code-element-base that is able to store HSV colors:

```
ipcCodeelementBase<ipcHSV> base(...);
```

The structure `ipcHSV` is defined in the include file `ipccolors.h`:

```
struct ipcHSV {
  short hue;
  unsigned char val,sat;
};
```

If the code-element-base should store 8 bit gray values instead we use the following instantiation:

```
ipcCodeelementBase<unsigned char> base(...);
```

The file `ipcodebase.h` contains both the definition and the implementation of the class. The class itself is a template class which has to be a data type of a color vector.

4.2.1 The Structure: ipcCodeelement

Code-elements are described by the following generic C++ structure:

```
template <class ipcFeatureType>
struct ipcCodeelement {
  unsigned char form,numOfSubCEs,flags;
  ipcFeatureType data;
  ipcCodeelement<ipcFeatureType>* parents[2];
  ipcSubCodeelement<ipcFeatureType>* subCEs;
  ipcCodeelement<ipcFeatureType>* next;
};
```

In the following list the purpose of each structure field is described:

12

- `form`: For a code-element of level 0 the lower seven bits of `form` represent the pixels of a level 0 island that are part of the code-element. Each bit-position accords to a certain sub-position within a level 0 island (see figure 4). Generally, the bit at position i denotes whether the pixel at sub-position P_i is a sub-pixel of the code-element or not. Consider the white code-element in figure 5. The pixels at the sub-positions P_2, P_4 and P_6 forms the white region. Thus the bits at the positions 2, 4 and 6 are set within the field *form*. In that case the value of $form$ is $2^2 + 2^4 + 2^6$. The bits of *form* have a different meaning for code-elements of a higher level than 0. Then the seven bits describe which of the seven sub-islands contain sub-regions of the code-element. Therefore *form* represents a vague shape of the region represented by the code-element.

- `numOfSubCEs`: In this field the number of sub-code-elements of the code-element are stored.

- `flags`: This field provides 8 bits that are used as flags. The most important flag (lowest bit in *flags*) is the root flag. It states that a code-element is a root of a code-element-tree. Such a root-code-element has no parents. Other flags are used during the segmentation algorithms. But after the segmentation the bits 1-7 of `flags` are not set and may be used without restrictions by the user.

- `data`: The field *data* stores the mean color of all sub-code-elements resp. sub-pixels. Thus *data* describes approximately the mean color of the region that is represented by the code-element. The type of `data` (`ipcFeatureType`) is defined by the template parameter since the color could be a gray value, a HSV color or a RGB color.

- `next`: This pointer refers to the next code-element within the same island.

- `parents`: The array `parents` has two entries. Each entry contains one parent pointer of the code-element. If a code-element has no parents both parent pointers refer to `NULL` (undefined).

- `subCEs`: The pointer `subCEs` refers to the first element of the sub-code-element-list. In this sub-code-element list the sub-code-pointers of the code-element are stored.

4.2.2 The Structure: ipcSubCodeelement

The items of the sub-code-element-lists are of type `ipcSubCodeelement`. Each code-element of level > 0 owns such an list:

```
template <class ipcFeatureType>
struct ipcSubCodeelement {
  ipcCodeelement<ipcFeatureType>* ce;
  int pos;
```

```
  ipcSubCodeelement<ipcFeatureType>* next;
}
```

Assume that `c` is a code-element and `s` is an item of its sub-code-element-list. `s->ce` refers then to a sub-code-element of `c` and `s->pos` denotes the sub-position of `s` within the island of `c`. The pointer `next` refers to the next item within the sub-code-element-list.

4.2.3   Implementation Details of the Code-Element-Base

The code-element-trees are stored in the code-element-base. It consists of three different tables: The *index table*, the *code-element table* and the *sub-code-element table* (see figure 9).

All code-elements that are generated by the CSC are stored in the code-element table. Each entry of the table is of type `ipcCodeelement<ipcFeatureType>`. Code-elements that was generated in the same island are linked via their `next` pointer (see previous section). They form a linear linked list. In order to save memory space it would also be possible to store all code-elements that belongs to the same island one after another in a compact block representation. But this would lead to some difficulties in the splitting phase. Sometimes new code-elements are generated in the splitting phase and have to be inserted into the code-element table. The insertion of code-elements is more simple by using a list-representation instead of a block-representation.

The index table provides direct access to all code-elements that was generated in the same island. Each island has a certain entry in the index table. The hierarchy level and the coordinates of an island are used to compute the position of its entry in the index table by a hash function. Each entry is of type `ipcIndexEntry`:

```
template <class ipcFeatureType>
struct ipcIndexEntry {
  ipcCodeelement<ipcFeatureType>* firstCE;
  ipcCodeelement<ipcFeatureType>* lastCE;
}
```

Since the code-elements of the same island form a linear linked list the pointer `firstCE` refers to the first and `lastCE` refers to the last element of that list.

The sub-code-pointers of each code-element (level $n > 0$) are stored along with their sub-positions in the sub-code-element table.

The advantage of using tables is that memory allocation can be done in the initialization phase of the algorithm. If someone want to segment several images of the same size one after and another memory allocation has to be done only once. This trick leads to higher performance than by using dynamically memory management routines of the operating system.

During the linking phase the algorithm builds a so called *segment list*. The segment list is a "real" linear linked list whose entries are not stored in a table. Each entry contains a pointer to a root-code-element (root of a code-element-tree) which represents a detected segment. Further each list entry contains the island in which

Segment-List

next	Flags, Form, Data etc.	subCE

| Level | x | y |

| Level | x | y |

NIL

NULL

ce	pos	next

NULL

firstCE	lastCE

Island Position
(level,x,y)

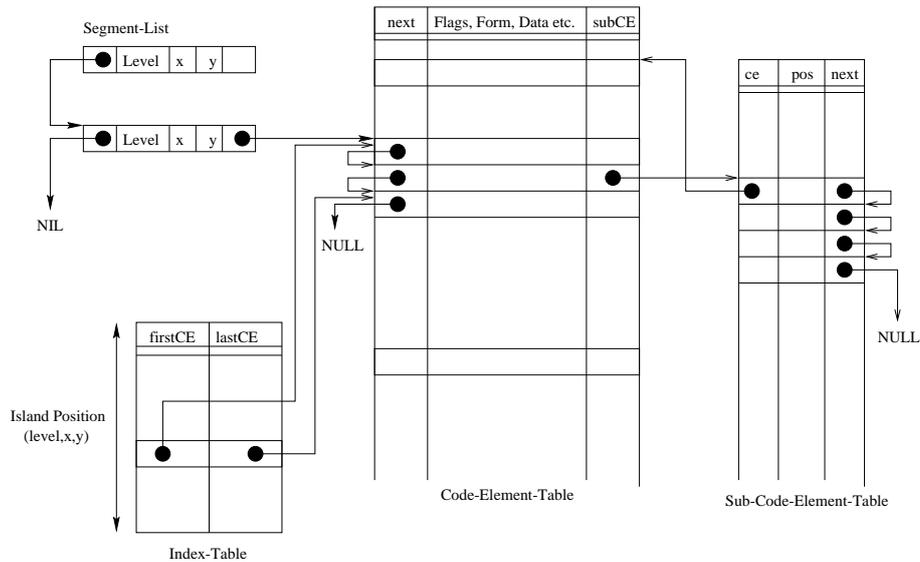Code-Element-Table

Sub-Code-Element-Table

Index-Table

Figure 9: Construction of the code-element base

the root-code-element was generated. The island defines the position of the segment within the image. Each item of the segment list is of type:

```
template <class ipcFeatureType>
struct ipcCSCSegment {
  ipcIsland island;
  ipcCodeelement<ipcFeatureType>* ce;
  ipcCSCSegment* next;
};
```

It is not time critical to use a linear list instead of a fourth table since there exists significant less root-code-elements than code-elements. The entries of the segment list are sorted by their hierarchy level: Large segments (High level) are stored at the beginning of the list and small segments (Low level) at the end of the list.

4.2.4 Enumeration of all Segments

The detected segments (of type ipcSegment) are stored in the segment list of the code-element-base. A segment is represented by a root-code-element and an island in which this code-element was generated. The following example shows how the iteration over all segments have to be done. The example function iterateOverAllSegments is called with a pointer to a code-element-base. Firstly, the method getFirstSegment of the code-element-base is called to retrieve a pointer to the first item of the segment list. The iteration over the segment is now done by simply traversing the list:

```
void iterateOverAllSegments(ipcCodeelementBase<ipcHSV>* ceBase) {
  // Definition of a pointer that refers to a segment
```

15

```
ipcCSCSegment<ipcHSV>* segment;

// Get the first segment from the code-element-base
segment=ceBase->getFirstSegment();

// iterate over all segments
while (segment!=NULL) {
  // Do something with the segment
  ...
  // Switch to the next segment in the list
  segment=segment->next;
}
}
```

4.2.5 Accessing the Pixels of a Segment

In this section we show how the pixels of a code-element-tree can be computed. The pixels are obtained by traversing the code-element-tree recursively. This done by the function `traverse`. The parameters of the function are:

1. a reference to the island hierarchy `h`,

2. a pointer to a root-code-element `ce`,

3. and an island in which the root-code-element was detected.

Firstly, the function checks if the code-element `ce` was already processed. This is done by testing the highest bit of `ce->flags`. Note, only the lowest bit of `ce->flags` is reserved. If the highest bit of `ce->flags` is set the code-element was already processed and the function returns immediately. Otherwise the bit is set by the function. This kind of labeling is used to avoid multiple processing of code-elements. Since code-elements may have two parents they could be reached via several paths from the root-code-element. If `ce` is of a higher level than 0 the function has to iterate over its sub-code-elements. This is done by traversing the sub-code-element-list of `ce`. For each sub-code-element the function `traverse` has to be called recursively to realize a recursive descent of the code-element-tree.

The following program the pointer `subCE` refers to an item of the sub-code-element-list of `ce`. `subCE->ce` denotes a sub-code-element of `ce` and `subCE->pos` denotes its sub-position within the island `isl`. With that information one can compute the sub-island of `isl` (`subIsland`) in which `subCE->ce` was generated. This is done by calling the method `getSubIsland` of the island hierarchy. Now the function `traverse` can be called recursively with the parameters `h`, `subCE->ce` and `subIsland`. The recursion stops at level 0.

```
void traverse(const ipcIslandHierarchy& h,
              ipcCodeelement<ipcHSV>* ce,
              const ipcIsland& isl) {
  // Leave the function immediately if ce was already processed.
  // We use for that purpose the highest bit of ce->flags
```

16

```
  if (ce->flags & 128) return;

  // Otherwise a flag is set for ce.
  // The flag denotes that ce is now processed.
  ce->flags|=128;

    // Is ce a code-element of a higher level than 0?
  if (isl.level>0) {
    ipcSubCodeelement<ipcHSV>* subCE;

    // Get a pointer to the first element of the
    // sub-code-element-list of ce.
    subCE=ce->subCEs;

    // Iterate over all sub-code-elements of ce
    while (subCE!=NULL) {
      ipcIsland subIsland;

      // Compute the sub-island at sub-position subCE->pos
      hierarchy.getSubIsland(island,subCE->pos,subIsland);

      // Recursive Descent
      traverse(h,subCE->ce,subIsland);
    }
    // Switch to the next sub-code-element of ce
    subCE=subCE->next;
  }
  else computePixels(h,ce,isl);
}
```

If the code-element `ce` is a level 0 code-element the function `traverse` calls the function `computePixels`. The function is called by using the code-element `ce` and the island `isl` of `ce` as parameters. Firstly, the function computes all seven pixel coordinates within the level 0 island `isl`. This is done by calling the `getSubPixels` method of the island hierarchy. The x-coordinates of the pixels are stored in the array `xcoord` and the y-coordinates of the pixels are stored in the array `ycoord`. Now the function iterates over all 7 sub-positions (`sub`) and tests whether the bit at position `sub` is set in `ce->form` or not. If the sub-th bit is set in `ce->form` the coordinate of the pixel at sub-position P_sub of island `isl` is printed to the console:

```
void computePixels(const ipcIslandHierarchy& hierarchy,
                   ipcCodeelement<ipcHSV>* ce,
                   const ipcIsland& isl) {
  // Array to store the X-coordinates of the pixels
  int xcoord[7]; //
  // Array to store the Y-coordinates of the pixels;
  int ycoord[7];
```

```
  // Compute the pixel coordinates of isl
  hierarchy.getSubPixels(isl,xcoord,ycoord);

  // Iterate over the sub-positions of the island
  for (int sub=0; sub<7; sub++) {
    // Check if the pixel at sub-position sub is set in
    // the form field of the code-element ce
    if (ce->form & (1<<sub)) {
      cout << "(" << xcoord[pos] << "," << ycoord[pos] << ")" << endl;
}
```

4.2.6 Drawing a Region Image

The class `ipcCodeelementBase` has also methods for drawing region images resp.
labeled images. Each pixel of a region image is displayed in the mean color of the
segment it belongs to. The method `drawRegionImage` generates such a region
image from the code-element-trees. The first four parameters of the function defines a
clipping rectangle. The clipping rectangle is specified by its upper-left corner (x,y), its
width (w) and its height (h). The last parameter is a pointer that refers to the memory
(image) where the region image should be stored. The following example shows how
the method is used. Firstly, we have to allocate a memory block that should store the
region image. We assume that the starting image has a size of 256×256 pixels. Now
we draw the region image by calling the method `drawRegionImage` of the code-
element-base. In this example the clipping rectangle has the same size as the starting
image. Thus the complete region image will be drawn:

```
  // Definition of the code-element-base
  ipcCodeelement<ipcHSV> ceBase(...)

  // Create Region Image
  ipcHSV regionImage[256][256];

  // Draw Region Image
  ceBase.drawRegionImage(0,0,256,256,*regionImage);
```

   A labeled image can be produced in a similar way. A labeled image can be consid-
ered as a gray value image. Each gray value is a label that identifies a certain segment.
The following code draws a labeled image from the code-element-base:

```
  // Definition of the code-element-base
  ipcCodeelement<ipcHSV> ceBase(...)

  // Create Region Image
  int labeledImage[256][256];
  map<int,ipcHSV> featureMap;

  // Draw Region Image
```

```
ceBase.drawLabeledImage(0,0,256,256,*labeledImage,featureMap);
```

By calling the method `drawLabeledImage` the mean color of a segment is assigned to the appropriate label. This relation is stored in the `featureMap`. Assume i is a label of a segment. Then `featureMap[i]` specifies its mean color.

4.3 The Class: ipcCSC

The class `ipcCSC<ipcPixelType,ipcFeatureType,ipcLinkControl>` is a generic class that does the CSC segmentation. The file `ipccsc.h` contains both its definition and implementation. By instantiating the class an island hierarchy object of type `ipcIslandHierarchy` and a code-element-base object of type `ipcCodeelementBase` will be created automatically. Thus an user of the CSC has just to create an instance of the class `ipcCSC`. Because the CSC should be able to work with different color spaces we define the class `ipcCSC` as a generic class. Thus the type of the color vectors used in the implementation can be defined by the programmer. `ipcPixelType` defines the type of the pixels stored in the starting image and `ipcFeatureType` specifies the type of the color vectors stored in the code-element-base. Normally the same type for both template arguments is used. The following definition specifies a CSC class for segmentation of HSV images:

`ipcCSC<ipcHSV,ipcHSV,ipcTrafficSignLinkControl>`.

The third template argument `ipcTrafficSignLinkControl` is a special class (link control class). A link control task is able to compute the mean color in a certain color space, i.e. HSV. It also contains the color similarity function that states whether two colors of a certain color space are similar or not. `ipcTrafficSignLinkControl` is an implementation of a link control class. It operates on the HSV color space. Its color similarity function was used in a project on traffic sign recognition in real-time. This color similarity function is suitable for segmentation of the most natural color scenes. A programmer has therefore the possibility to change the color similarity function without modifying the code of the segmentation algorithm. He has just to write a new link control class and use it as the third template parameter of the class `ipcCSC`.

In the following we present a small program that shows how to segment a HSV image. The HSV image of size 256×256 pixels. It is stored in the array `original`. Then an instance of the class `ipcCSC` is created. Its method `segmentation` is called to start the segmentation. As parameters the method obtains the width and height of the original image. Further it obtains the original image itself. This method also creates an instance of `ipcIslandHierarchy` and `ipcCodeelementBase`. One can get a pointer to these objects by calling the `getIslandHierarchy` resp. `getCEBase` method of the `ipcCSC` class. After processing the method `segmentation` the code-element-base within the CSC object contains the segmentation result. After that the region image can be drawn:

```
#include <ipccolor.h>
#include <ipccsc.h>
#include <ipctrafficsignlinkcontrol.h>
```

```
int main(int argc, const char* argv[]) {
  // Definition of the original hsv image of size 256x256
  ipcHSV original[256][256];

  // Definition of the region image of size 256x256
  ipcHSV region[256][256];

  // Read the image from file
  ...

  // Creation of an instance of ipcCSC
  ipcCSC<ipcHSV,ipcHSV,ipcTrafficSignLinkControl> csc;

  // Start the segmentation
  csc.segmentation(256,256,original);

  // Get the code-element-base
  ipcCodeelementBase<ipcHSV>* base = csc.getCEBase();

  // Draw the region image
  base->drawRegionImage(0,0,256,256,region);

  return 0;
}
```

References

[HA87] G. Hartmann. Recognition of Hierarchically Encoded Images by Technical and Biological Systems. In: Biological Cybernetics, 57:73-84, 1987.

[HP76] S. L. Horowitz, T. Pavlidis. Picture Segmentation by a Traversal Algorithm. Journal of the ACM, 23:368-388, 1976.

[PR93] L. Priese, V. Rehrmann. A Fast Hybrid Color Segmentation Method. In: S.J. Pöppl and H.Handels, editors, Mustererkennung 1993, pages 297-304. Springer Verlag, 1993. 15. DAGM-Symposium, Lübeck, 27.-29.Sept. 1993.

[RE98] V. Rehrmann, L. Priese. Fast and Robust Segmentation of Natural Color Scenes. 3rd Asian Conference on Computer Vision, Hongkong, 8-10th January 1998.