Appendix B

Source Code of the Region Tracking Algorithm

In the following section the complete code listing of the proposed region tracking algorithm is given. The source code is written in C++ and does not require any further libraries. All supplementary data structures are listed to allow potential users to easily implement and test the algorithm.

The region tracker assumes that tracking takes place on a two-dimensional grid. This grid has not necessarily to be an image. To adapt the tracker to a specific purpose, only one function, the *homogeneity function*, has to be written. This user-specified procedure is passed to the tracker during initialization using the concept of function pointers.

The homogeneity function tells the tracker whether or not the cell (x, y) meets the homogeneity criterion. Actually, by providing the function, the homogeneity criterion is defined.

The tracker can work on single pixels, defining a certain color class as a homogeneity criterion, for instance, or it can work on blocks of pixels. The homogeneity criterion has not be based on color similarity. For instance, one could define homogeneity criteria based on texture within a block of pixels.

B.1 The Homogeneity Criterion

Assume that 24 bit RGB is used as image format and that a global pointer *BYTE*pImageData* is available, which points to the actual image. Then an example for a homogeneity function is:

```
DWORD  IsGreen(int x, int y) {
   DWORD byteTripleIndex=(y+dx+x)*3;
   DWORD greenIndex=byteTripleIndex+1
   BYTE greenValue=pImageData[greenIndex];
   if (greenValue>200) return 0x00000001;
   return 0;
}
```

Note, that this function is just a pseudo-function. It is not the function, which is used in our system. The homogeneity function has not to care whether (x,y) is a valid image position, the tracker ensures that.

One might wonder that the return value is a 32-Bit value and not a boolean value. The reason is that the region tracker performs a bitwise AND operation between the "classMask" parameter passed at initialization and the mask returned by the homogeneity function to verify whether a specific location in the image meets the homogeneity criterion.

The following pseudo code gives another example of a homogeneity function, in the case of a tracker working on blocks of 8×8 pixels.

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
DWORD getYUV422_88_Mask(int x, int y) {
    int x0=x*8;
    int y0=y*8;
    int greenPixelCounter=0;
    DWORD mask=0;
    for (int ix=0;ix<8;ix++){
        for (int iy=0;iy<8;iy++){
            int ax=x0+ix;
            int ay=y0+iy;
            if (the pixel at ax,ay is green){
                greenPixelCounter++;
            }
        }
    }
    if (greenPixelCounter>=12){
        mask|=0x000000001; //Assume, the first bit means "green"
    }
    return mask;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

B.2 The "Don't touch" Array

Sometimes one might want to forbid the tracker to track regions in certain areas within the image.

For instance, when using an omni-directional catadioptric setup, one might want to delimit the tracker to the areas within the mirror.

Thus, the user provides an array having the same size as the image (when working on pixels, otherwise the size of the grid). Each element is a byte with the value being zero if the tracker is allowed to access the corresponding position and one, if it is not allowed. The array is plain, that is cell (x,y) is at index y*dx+x. A pointer to the array has to be provided during initialization.

B.3 Initializing the Region Tracker

The initialization of the tracker is described by means of an example in the following. Let us assume a resolution of 640×480 pixels with the video format being YUV422. Assume further that the tracker should work on units of 8×8 pixels. Then the initializing code looks as follows:

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#include "FU_RegionTracker.h" int dx=640; int dy=480; int
subsamplingFactor=8; int unitDx=dx/subsamplingFactor; int
unitDy=dy/subsamplingFactor; BYTE*pDT=new BYTE[unitDx*unitDy];//The
"don't touch" array allows to
                                 //define areas in the image, which should
                                 //not be accessed by the region
                                 //tracker.
setmemory(pDT,unitDx*unitDy,1);  //Allow the tracker to work on the
                                 //whole image.
GetMaskFuncXY getMaskXY= getYUV422_88_Mask; //The function pointer
                                            //type GetMaskFuncXY is defined
                                            //in "FU_RegionTracker.h".
                                            //"getYUV422_88_Mask" is your own
                                            //function. The region tracker will call
                                            //this function in order to
                                            //access your images. A more detailed description
                                            //follows in the next section.
DWORD classMask=0x00000001; //Track regions of color class 1
CFU_RegionTracker* pRegionTracker=new
CFU_RegionTracker(unitDx,unitDy,pDT,getYUV422_88_Mask,subsamplingFactor,classMask);
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

The following sections listen the complete source code of the tracker, encompassing the following files:

B.4 FU_RegionTracker.h

This file together with the corresponding cpp-file implements the core of the region tracker.

Filename	implements...
FU_Regiontracker.h,.cpp	the core class of the region tracker
FU_Contours.h,.cpp	the class storing the contour information
FV2.h,.cpp	a small class implementing simple 2D vectors
UTIL_TOPFX.h	a fast queue for storing the drops
UTIL_HEAP	a heap structure which is used during initial seeding

Table B.1: The listed files, implementing the region tracker

```
#ifndef FU_REGION_TRACKER
#define FU_REGION_TRACKER
#include "UTIL_Heap.h"
#include "FU_Contours.h"
#include "UTIL_TopfX.h"
#include "FV2.h"

#ifndef BYTE  //8 Bit
   #define BYTE unsigned char
#endif
#ifndef DWORD //32 Bit
    #define DWORD unsigned long
#endif
typedef DWORD (*GetMaskFuncXY)(int x, int y);

//////////////////////////////////////////////////////////////////////
typedef union {
    struct{
        short x;
        short y;
    }s;
    DWORD value;
} DROP;
//////////////////////////////////////////////////////////////////////
typedef struct{
    int x;
    int y;
    BYTE dir;
} EdgePiece;

//////////////////////////////////////////////////////////////////////
class CSeedRegion { public:

   CSeedRegion(){};
   CSeedRegion(int xx,int yy, int ww):x(xx),y(yy),w(ww){};
   int x;
   int y;
   int w;
   bool operator<(CSeedRegion w2){return (w<w2.w);};
   bool operator>(CSeedRegion w2){return (w<=w2.w);};
   bool operator>=(CSeedRegion w2){return (w>=w2.w);};
   bool operator<=(CSeedRegion w2){return (w<=w2.w);};
   bool operator==(CSeedRegion w2){return (w==w2.w);};
   ~CSeedRegion(){};

};
//////////////////////////////////////////////////////////////////////
class CFU_RegionTracker { public:
   CFU_RegionTracker(int dx,int dy,BYTE*pDT_p,GetMaskFuncXY getMaskXY_p,int subSamplingFactor_p,DWORD mask_p);
   virtual ~CFU_RegionTracker();
   void ReInit();
   bool GetCentreOfGravity(FV2 &c);
   void Track();

public:
   CFU_Contours boundaries;

private:
   void GetBoundaries();
   void SeedControl();
   bool FindNBestSeeders(int n);
   int  DetermineSizeOfRegion(int sx, int sy, DWORD mask);
   void Grow();
   void Shrink();
   bool DropNextToFrame(EdgePiece &f, int x,int y);
   bool GetUndeletedEdgePiece(EdgePiece &f);
   void FrameToPoints(IV2&p0,IV2&p1, EdgePiece *f);

private:
   bool bStarting;

   GetMaskFuncXY getMaskXY; //The homogeneity function
   int dx,dy;         //dimension of the image (in picture elements)
   int nCells;        //=dx*dy
   BYTE* occu;        //array of size dx*dy to mark which picture elements are occupied by a drop
   int gdx,gdy;       //dimension of the connectivity grid (gdx=dx+1;gdy=dy+1)
   int nConCells;     //=gdx*gdy
   BYTE *con;         //the connectivity grid: array of size gdx*gdy; it stores the boundaries of the regions
   BYTE *pDT;         //Don’t Touch array
   CUTIL_TopfX<DROP> *pCommonDrops;
   int trackCounter;
   int id;
   CUTIL_Heap<CSeedRegion> seedHeap;
   int shiftx[4];
   int shifty[4];
   int conshiftx[4];
   int conshifty[4];

   int dropAndDirToPointX[4];
   int dropAndDirToPointY[4];
   BYTE nextDir[4][3];
   DWORD mask;
   DWORD iOfFirstUndeleted;
   CUTIL_TopfX<DROP> growList;
   CUTIL_TopfX<DROP> shrinkList;
   IV2 gravitySum;
   int nSize;
   int subSamplingFactor;
};

#endif
```

B.5 FU_RegionTracker.cpp

This file constitutes the core of the region tracker source code. Here the growing and shrinking is implemented.

```
/////////////////////////////////////////////////////////////////////
//FU_RegionTracker.cpp
/////////////////////////////////////////////////////////////////////

#include "stdafx.h" #include "FU_RegionTracker.h"

/////////////////////////////////////////////////////////////////////
/**
* INPUT: int dx_p,dy_p                       : the size of the image being tracked, if you don’t work on pixels but on block of pixels
*                                                dx_p,dy_p is the size of the subsampled grid
*         BYTE*pDT_p                           : the "Dont't Touch" array. The user has to create this array the size being dx*dy,
*                                                containing BYTEs indicating whether the tracker is allowed
*                                                to access the corresponding cell (value=0) or whether it is not allowed (value=1)
*         getMaskXY_p                          : a function pointer to the user specified homgeneity function
*         subSamplingFactor_p                  : the factor of subsampling, for instance when tracking on blocks of 4x4 pixels,
*                                                the factor is 4, when tracking on single pixels the factor is 1.
*                                              : This factor is required to produce contour coordinates which correspond to the plain pixels.
*         mask_p                               : The mask which is used to logical "AND"-combine the results of the homgeneity function.
*                                                Only regions with AND-combinations not equal to zero are tracked.
*/
CFU_RegionTracker::CFU_RegionTracker(int dx_p, int dy_p, BYTE*pDT_p, GetMaskFuncXY getMaskXY_p, int subSamplingFactor_p,DWORD mask_p)
   :seedHeap(false),shrinkList(dx_p*dy_p),growList(dx_p*dy_p)
{
   id=1;                                  //the id of the region
   mask=mask_p;                           //the mask of the region
   subSamplingFactor=subSamplingFactor_p; //the subsampling factor (the number of pixels per unit along the x, and y-axis, respectively)
   dx=dx_p;                               //the horizontal number of units (one unit has subSamplingFactor pixels in horizontal and vertical direction)
   dy=dy_p;                               //the vertical number of units (one unit has subSamplingFactor pixels in horizontal and vertical direction)
   nCells=dx*dy;
   occu=new BYTE[nCells];
   gdx=dx+1;                              //the dimensions of the connectivity grid...
   gdy=dy+1;
   nConCells=gdx*gdy;
   con=new BYTE[nConCells];               //the connectivity grid
   getMaskXY=getMaskXY_p;                 //image access function
   pDT=pDT_p;
   if (!pDT){                             //DT= "Dont Touch"
                                          //this array must have size dx*dy bytes. A cell different from zero indicates that the corresponding unit in the image
                                          //should not be covered by a region
      //error message
      return; //pDT must exist
   }
   bStarting=true;
   dropAndDirToPointX[0]=0; dropAndDirToPointY[0]=1;
   dropAndDirToPointX[1]=1; dropAndDirToPointY[1]=1;
   dropAndDirToPointX[2]=1; dropAndDirToPointY[2]=0;
   dropAndDirToPointX[3]=0; dropAndDirToPointY[3]=0;
   shiftx[0]= 0  ; shifty[0]= 1;
   shiftx[1]= 1  ; shifty[1]= 0;
   shiftx[2]= 0  ; shifty[2]=-1;
   shiftx[3]=-1  ; shifty[3]= 0;
   nextDir[0][0]=3;
   nextDir[0][1]=0;
   nextDir[0][2]=1;
   nextDir[1][0]=0;
   nextDir[1][1]=1;
   nextDir[1][2]=2;
   nextDir[2][0]=1;
   nextDir[2][1]=2;
   nextDir[2][2]=3;
   nextDir[3][0]=2;
   nextDir[3][1]=3;
   nextDir[3][2]=0;
   conshiftx[0]=  1; conshifty[0]= 0;
   conshiftx[1]=  0; conshifty[1]=-1;
   conshiftx[2]= -1; conshifty[2]= 0;
   conshiftx[3]=  0; conshifty[3]= 1;
   trackCounter=0;
   pCommonDrops=new CUTIL_TopfX<DROP>(dx*dy);
   gravitySum.x=gravitySum.y=0;
   boundaries.ReserveDefaultMemory();
   ReInit();
}
//////////////////////////////////////////////////////////////////////
void CFU_RegionTracker::ReInit() {
   ZeroMemory(con,sizeof(con[0])*nConCells);
   ZeroMemory(occu,sizeof(occu[0])*nCells);
   nSize=0;
   gravitySum.x=gravitySum.y=0;
   growList.Clean();
   shrinkList.Clean();
   boundaries.Reset();
}
//////////////////////////////////////////////////////////////////////
CFU_RegionTracker::~CFU_RegionTracker() {
   delete [] pCommonDrops;
   delete [] con;
   delete [] occu;
}
//////////////////////////////////////////////////////////////////////
/**
* You might modify the beginning of the function. Currently a complete reinitialization
* is performed every 300 frames. In some applications you might not need any reinitialization.
*/ void CFU_RegionTracker::Track() {
    if (bStarting){
        if ((trackCounter%30)==0) ReInit();
        bStarting=false;
    }else{
        if ((trackCounter%300)==0) ReInit();
    }
   SeedControl();
   Shrink();
   Grow();
   GetBoundaries();
   trackCounter++;
}
//////////////////////////////////////////////////////////////////////

void CFU_RegionTracker::SeedControl() {
   bool foundAnySeedPixel=false;
   if (growList.IsEmpty()&&shrinkList.IsEmpty()){
      if (FindNBestSeeders(10)){
          foundAnySeedPixel=true;
      }
   }
}
//////////////////////////////////////////////////////////////////////
bool CFU_RegionTracker::FindNBestSeeders(int n) {
   seedHeap.Clear();
   for (int i=0;i<nCells;i++){
      if (!occu[i]&&!pDT[i]){
         DWORD c=getMaskXY(i,0);
         if (c&mask){
            int ax=i%dx;
            int ay=i/dx;
            int aSize=DetermineSizeOfRegion(ax,ay,mask);
            CSeedRegion p(ax,ay,aSize);
            seedHeap.Insert(p);
         }
      }
   }
   ZeroMemory(occu,dx*dy);
   if (seedHeap.GetSize()){
      for (int i=0;i<nCells;i++){
           occu[i]&=0x7F;
      }
   }
   bool foundAny=false;
   for (int k=0;k<n;k++){
      if (seedHeap.GetSize()){
         CSeedRegion p=seedHeap.ExtractTop();
         DWORD o=p.y*dx+p.x;
         occu[o]=id;
         DROP d;
         d.s.x=(short)p.x;
         d.s.y=(short)p.y;
         growList.Append(d);
         nSize++;
         gravitySum.x+=p.x;
         gravitySum.y+=p.y;
         foundAny=true;
      }else{
         break;
      }
   }
   Grow();
   GetBoundaries();
   return foundAny;
}
//////////////////////////////////////////////////////////////////////
int CFU_RegionTracker::DetermineSizeOfRegion(int sx, int sy, DWORD mask) {
   int masse=0;
   long o=sy*dx+sx;
   occu[o]|=0x80;
   masse++;
   CUTIL_TopfX<DROP>*pTopf=pCommonDrops;
   DROP d;d.s.x=sx;d.s.y=sy;
   pTopf->Append(d);
   while(!pTopf->IsEmpty()) {
      DROP d=pTopf->ExtractLast();
      long x=d.s.x;
      long y=d.s.y;
      for (long k=0;k<4;k++){
         long ax=x+shiftx[k];
         long ay=y+shifty[k];
         o=ay*dx+ax;
         if ((ax>=0&&ax<dx&&ay>=0&&ay<dy)&&(!pDT[o])&&(occu[o]==0)){
            DWORD amask=getMaskXY(ax,ay);
            if (amask&mask){
               occu[o]|=0x80;
               masse++;
               DROP d;
               d.s.x=ax;
               d.s.y=ay;
               pTopf->Append(d);
            }
           }
      }
   }
   return masse;
}
//////////////////////////////////////////////////////////////////////
void CFU_RegionTracker::Grow() {
   CUTIL_TopfX<DROP>*pResultingDrops=&shrinkList;
   CUTIL_TopfX<DROP>*pDropPool=&growList;
   DWORD o;
   while(!pDropPool->IsEmpty()) {
       DROP d=pDropPool->ExtractLast();
       bool generated=false;
       for (long k=0;k<4;k++){
          long ax=d.s.x+shiftx[k];
          long ay=d.s.y+shifty[k];
          o=ay*dx+ax;
          bool rand=false;
          if (ax<0) rand=true;
          else if (ay<0) rand=true;
          else if (ay>=dy) rand=true;
          else if (ax>=dx) rand=true;
          if (!rand) rand=pDT[o];
          DWORD amask=0;
          if (!rand){
             if (!occu[o]){
                amask=getMaskXY(ax,ay);
                if (amask&mask){
                      occu[o]=id;
                      DROP nd;
                      nd.s.x=ax;
                      nd.s.y=ay;
                      pDropPool->Append(nd);
                      gravitySum.x+=ax;
                      gravitySum.y+=ay;
                      nSize++;
                }else{
                   rand=true;
                }
            }else{
                if (occu[o]!=id) rand=true;
            }
         }
         if (rand){
            generated=true;
            int gx=d.s.x+dropAndDirToPointX[k];
            int gy=d.s.y+dropAndDirToPointY[k];
            DWORD o0=gy*gdx+gx;
            BYTE conmask=1<<k;
            DWORD color=color;
            con[o0]|=conmask;
         }
      }
      if (generated) {
         pResultingDrops->Append(d);
      }
   }
}
//////////////////////////////////////////////////////////////////////
void CFU_RegionTracker::Shrink() {
   CUTIL_TopfX<DROP>*pResultingDrops=&growList;
   CUTIL_TopfX<DROP>*pDropPool=&shrinkList;
   DWORD o;
   //Clearing
   for (int i=0;i<pDropPool->n;i++){
       DROP d=pDropPool->GetAt(i);
       o=d.s.y*dx+d.s.x;
       occu[o]=0;
       gravitySum.x-=d.s.x;
       gravitySum.y-=d.s.y;
       nSize--;
   }
   while(!pDropPool->IsEmpty()) {
      DROP d=pDropPool->ExtractLast();
       DWORD ao=d.s.y*dx+d.s.x;
      DWORD amask=getMaskXY(d.s.x,d.s.y);
      if (amask&mask){
         pResultingDrops->Append(d);
      }else{
         bool generated=false;
         for (long k=0;k<4;k++){
            long ax,ay;
            ax=d.s.x+shiftx[k];
            ay=d.s.y+shifty[k];
            o=ay*dx+ax;
            if ((ax>=0&&ax<dx&&ay>=0&&ay<dy)&&(!pDT[o])&&(occu[o]==id)){
                occu[o]=0;
                    DROP nd;
                    nd.s.x=ax;
                    nd.s.y=ay;
                    gravitySum.x-=ax;
                    gravitySum.y-=ay;
                    nSize--;
                pDropPool->Append(nd);
            }
         }
      }
   }
   //Setting
   for (i=0;i<pResultingDrops->n;i++){
       DROP d=pResultingDrops->GetAt(i);
       o=d.s.y*dx+d.s.x;
       occu[o]=id;
       gravitySum.x+=d.s.x;
       gravitySum.y+=d.s.y;
       nSize++;
   }
}
//////////////////////////////////////////////////////////////////////
bool CFU_RegionTracker::DropNextToFrame(EdgePiece &f, int x,int y) {
   int gx[4];
   int gy[4];
   gx[0]=x;gy[0]=y+1;
   gx[1]=x+1;gy[1]=y+1;
   gx[2]=x+1;gy[2]=y;
   gx[3]=x;gy[3]=y;
   for (int i=0;i<4;i++){
      BYTE mask=1<<i;
      DWORD to=gy[i]*gdx+gx[i];
      if (con[to]&mask){
         f.x=x;
         f.y=y;
         f.dir=i;
         return true;
      }
   }
   return false;
}
//////////////////////////////////////////////////////////////////////
bool CFU_RegionTracker::GetUndeletedEdgePiece(EdgePiece &f) {
   DWORD i=iOfFirstUndeleted;
   DWORD n=shrinkList.n;
   DROP*p=shrinkList.pMem;
   while(i<n){
      if (DropNextToFrame(f,p[i].s.x,p[i].s.y)){
         iOfFirstUndeleted=i;
         return true;
      }
      i++;
   }
   return false;
}
//////////////////////////////////////////////////////////////////////
void CFU_RegionTracker::GetBoundaries() {
   CFU_Contours&b=boundaries;
   b.Reset();
   EdgePiece f;
   IV2 p0,p1;
   iOfFirstUndeleted=0;
   int everyX=2;
   while(GetUndeletedEdgePiece(f)){
      FrameToPoints(p0,p1,&f);
      int ldir=f.dir;
      bool found;
      int counter=0;
      do{
        if ((counter%everyX)==0){
             b.point[b.nPoints].x=p0.x*subSamplingFactor;
             b.point[b.nPoints].y=p0.y*subSamplingFactor;
             b.nPoints++;
        }
        DWORD o0=p0.y*gdx+p0.x;
        found=false;
```

```
         if (con[o0]){
            for (int i=0;i<3;i++){
               BYTE testDir=nextDir[ldir][i];
               BYTE mask=1<<testDir;
               if (con[o0]&mask){
                  p1.x=p0.x+conshiftx[testDir];
                  p1.y=p0.y+conshifty[testDir];
                  found=true;
                  BYTE burner=~mask;
                  con[o0]&=burner;
                  ldir=testDir;
                  break;
               }
            }
         }
         p0=p1;
           counter++;
       }while(found);
       b.inci[++b.nLines]=b.nPoints;
   }
   b.SmoothAndGrowInsitu(1.0f);
}

//////////////////////////////////////////////////////////////////////
void CFU_RegionTracker::FrameToPoints(IV2&p0,IV2&p1, EdgePiece *f) {
   switch(f->dir){
   case 0: p0=IV2(f->x,f->y+1);  p1=IV2(f->x+1,f->y+1);   break;
   case 1: p0=IV2(f->x+1,f->y+1);p1=IV2(f->x+1,f->y);     break;
   case 2: p0=IV2(f->x+1,f->y);  p1=IV2(f->x,f->y);       break;
   case 3: p0=IV2(f->x,f->y);    p1=IV2(f->x,f->y+1);     break;
   }
}
//////////////////////////////////////////////////////////////////////
bool CFU_RegionTracker::GetCentreOfGravity(FV2 &c) {
   if (!nSize) return false;
   c.x=gravitySum.x/(float)nSize;
   c.y=gravitySum.y/(float)nSize;
   c*=subSamplingFactor;
   return true;
}
```

B.6 FV2.h

This file defines a simple 2D point or vector. It is used to represent the extracted field contours.

```
//////////////////////////////////////////////////////////////////////
#ifndef FV2_H
#define FV2_H
#define FLOAT float
//////////////////////////////////////////////////////////////////////
/*The class for a floating point 2D point or vector*/ class FV2 {
public:
    FV2();
    FV2(float x,float y);
    ~FV2();
    FV2 operator-(FV2&v);
    FV2 operator+(FV2&v);
    FV2 operator*(FLOAT l);
    FV2 operator/(FLOAT l);
    FLOAT operator*(FV2 v);
    FV2 operator-();
    void operator /=(FLOAT f);
    void operator *=(FLOAT f);
    void operator +=(FV2&v);
    void operator -=(FV2&v);
    FV2 operator~(); //yields the orthogonal vector (left turning)
public:
   FLOAT x;
   FLOAT y;
};
//////////////////////////////////////////////////////////////////////
FV2 operator*(FLOAT f,FV2 v);
//////////////////////////////////////////////////////////////////////
/*The class for an integer 2D point or vector*/ class IV2 { public:
    IV2();
    IV2(int x,int y);
    ~IV2();
public:
   int x;
   int y;
};
//////////////////////////////////////////////////////////////////////
#endif
```

B.7 FV2.cpp

This file implements a simple 2D point or vector. It is used to represent the extracted field contours.

```
#include "stdafx.h"
#include "FV2.h"
//////////////////////////////////////////////////////////////////////
// Konstruktion/Destruktion
//////////////////////////////////////////////////////////////////////
FV2::FV2() {
}
//////////////////////////////////////////////////////////////////////
FV2::FV2(FLOAT x,FLOAT y):x(x),y(y) {
}
//////////////////////////////////////////////////////////////////////
FV2::~FV2() {
}
//////////////////////////////////////////////////////////////////////
FV2 FV2::operator*(FLOAT l) {
    return FV2(x*l,y*l);
}
//////////////////////////////////////////////////////////////////////
FV2 FV2::operator/(FLOAT l) {
    return FV2(x/l,y/l);
}
//////////////////////////////////////////////////////////////////////
FV2 FV2::operator -() {
    return FV2(-x,-y);
}
////////////////////////////////////////////////////////////////////////
void FV2::operator +=(FV2&v) {
    x+=v.x;
    y+=v.y;
}
////////////////////////////////////////////////////////////////////////
void FV2::operator -=(FV2&v) {
    x-=v.x;
    y-=v.y;
}
```

```
////////////////////////////////////////////////////////////////////////
void FV2::operator /=(FLOAT f) {
    x/=f;
    y/=f;
}
////////////////////////////////////////////////////////////////////////
void FV2::operator *=(FLOAT f) {
    x*=f;
    y*=f;
}

////////////////////////////////////////////////////////////////////////
FLOAT FV2::operator*(FV2 v) {
    return x*v.x+y*v.y;
}
/////////////////////////////////////////////////////////////////////
FV2 FV2::operator~() {
  return FV2(-y,x);
}
/////////////////////////////////////////////////////////////////////
FV2 FV2::operator-(FV2&v) {
    return FV2(x-v.x,y-v.y);
}
/////////////////////////////////////////////////////////////////////
FV2 FV2::operator+(FV2&v) {
    return FV2(x+v.x,y+v.y);
}

/////////////////////////////////////////////////////////////////////
FV2 operator*(FLOAT f,FV2 v){
    return FV2(v.x*f,v.y*f);
}
/////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////
IV2::IV2() {
}
/////////////////////////////////////////////////////////////////////
IV2::IV2(int x,int y):x(x),y(y) {
}
/////////////////////////////////////////////////////////////////////
IV2::~IV2() {
}
```

B.8 FU_Contours.h

This file defines a class to represent the extracted boundary contours. In each frame, the tracker yields new field contours. The field contours are defined as an array of points and an incidence structure that partitions the points into segments.

```
#ifndef FU_CONTOURS_H
#define FU_CONTOURS_H
#include "FV2.h"

#define MAX_CONTOUR_POINTS   10000
#define MAX_FU_LINES         2000

/** CFU_Contours stores several closed curves (referred to as lines
in the source code), each consisting of a sequence of points. */
class CFU_Contours { public:
    int GetNumberOfLines();
    int GetNumberOfPoints();
    void Reset();
    CFU_Contours();
    virtual ~CFU_Contours();
    void ReserveDefaultMemory();
    void AccessLine(FV2*&p,int&nP,int iLine);
    void SmoothAndGrowInsitu( float extent);
    void Draw(void*pDrawingContext);
    FV2 *point;
    int *inci;
    int nPoints;
    int nLines;
};

#endif
```

B.9 FU_Contours.cpp

This file implements the class representing the extracted boundary contours. In each frame, the tracker yields new field contours. The field contours are defined as an array of points and an incidence structure that partitions the points into segments.

```
///////////////////////////////////////////////////////////////////////
#include "Stdafx.h"
#include "FU_Contours.h"
//#include "..\..\Common\ImageWnd.h"
extern class CImageWnd*gpImageWnd;
///////////////////////////////////////////////////////////////////////
CFU_Contours::CFU_Contours() {
   nPoints=0;
   nLines=0;
   inci=0;
   point=0;
}
///////////////////////////////////////////////////////////////////////
CFU_Contours::~CFU_Contours() {
   if (point) delete [] point;
   if (inci)  delete [] inci;
}
///////////////////////////////////////////////////////////////////////
/**Reset the contours, that is clear all points and lines. *Prepare
the data structure for storing new contour data. */ void
CFU_Contours::Reset() {
   nPoints=0;
   nLines=0;
   if (inci) inci[0]=0;
}
///////////////////////////////////////////////////////////////////////
/**Reserve memory for MAX_FU_LINES lines and MAX_CONTOUR_POINTS
points*/ void CFU_Contours::ReserveDefaultMemory() {
   point=new FV2[MAX_CONTOUR_POINTS];
   inci= new int[MAX_FU_LINES+1];
}
///////////////////////////////////////////////////////////////////////
/**Draw the contours. Add your own source code here.*/ void
CFU_Contours::Draw(void*pDrawingContext) {
//add system dependent drawing code//
   CDC*pDC=(CDC*)pDrawingContext;

    for (int l=0;l<nLines;l++){
      int nP;
      FV2*p;
      AccessLine(p,nP,l);

        for (int i=0;i<nP;i++){
               //add system dependent drawing code://
         CPoint sp=gpImageWnd->WorldToScreen(V2(p[i].x,p[i].y));
            if (!i){
                //add system dependent drawing code://
            pDC->MoveTo(sp);
            }else{
                //add system dependent drawing code://
            pDC->LineTo(sp);
            }
        }

    }
}
//////////////////////////////////////////////////////////////////////
/**Smooth and enlarge the lines *Input:  extent : the amount of how
much the lines are enlarged
*                If this value is zero, the lines are just smoothed.
*Remarks: Each line is a closed curve. For enlarging the lines each
*         point is moved in direction of the normal at that point.
*         This function should be optimized. Identical caluclations
*         are performed several times.
*/ void CFU_Contours::SmoothAndGrowInsitu( float extent) {
    for (int i=0;i<nLines;i++){
        FV2*p;
        int nP;
        AccessLine(p,nP,i);
       for (int i=0;i<nP;i++){
           int im_2=(i+nP-2)%nP; //Indices to neighboring points...
           int im_1=(i+nP-1)%nP;
           int ip_1=(i+nP+1)%nP;
           int ip_2=(i+nP+2)%nP;

           FV2 v_m1=p[i]-p[im_1];//Tangent vectors,...
           FV2 v_m2=p[i]-p[im_2];
           FV2 v_p1=p[ip_1]-p[i];
           FV2 v_p2=p[ip_2]-p[i];

         //Calculate the smoothed tangent vector...
           FV2 v=0.1f*v_m1+0.4f*v_m2+ 0.4f*v_p1+ 0.1f*v_p2;
         //Vector v should be normalized here
         //However, the points have approximately the same distance.
         //Thefore, instead of an expensive normalization to length one,
         //we multipy the following constant, assuming an average distance
         //of 8 pixels between the points.
           v*=0.125f;
         //The function also will work, if the assumption is not true,
         //since it can be tuned with the parameter "extent".
           FV2 n=~v*extent; //~v yields the vector orthogonal to v (left turning)
         //Calcualte the new smoothed point
         p[i]=0.25*p[im_1]+0.25*p[ip_1]+0.5*(p[i]+n);
       }
    }
}
//////////////////////////////////////////////////////////////////////
/**Call this function to access the points of line "iLine". *Input:
*        iLine   : the zero-based index of the line
*Output by reference: *
*        p       : a reference to a pointer which will receive the
*                  adrress of the first point of the line
*        nP      : a reference to an interger which receives the
*                  number of points of line "iLine".
* *Example:   Draw the points of the third line *
*            FV2*p;
*            int nP;
*            AccessLine(p,nP,2);
*            for (int i=0;i<nP;i++){
*                DrawPoint(p[i].x,p[i].y);
*            }
*/ void CFU_Contours::AccessLine(FV2 *&p, int &nP, int iLine) {
    int i0=inci[iLine+0];
    int i1=inci[iLine+1];
    nP=i1-i0;
    p=&point[i0];
}
/////////////////////////////////////////////////////////////////////
/**returns the number of lines*/ int
CFU_Contours::GetNumberOfLines() {
   return nLines;
}
/////////////////////////////////////////////////////////////////////
/**returns the number of points*/ int
CFU_Contours::GetNumberOfPoints() {
   return nPoints;
}
```

B.10 UTIL_HEAP.h

This file defines a template class for a heap structure. It is used, when the tracker performs initial seeding. Here, all regions are determined and inserted in the heap, with the largest regions being on top of the tree structure.

```
#ifndef UTIL_HEAP
#define UTIL_HEAP
//////////////////////////////////////////////////////////////////////////////////////////
//This template class provides a simple heap, whose maximum size is
//fixed.
//We made this limitation because some dynamic data structures, i.e. STL,
//are often not fully platform independent.
//Replace the array "T*a" by a dynamic data structure to have
//a dynamic heap.
//////////////////////////////////////////////////////////////////////////////////////////
template <class T>class CUTIL_Heap { public:

   T*a;
   int nE;          //actual number of elements
   int maxSize;     //maximum number of elements
   bool bLowOnTop;  //lowest or greatest element at top?

   ////////////////////////////////////////////
   CUTIL_Heap(bool bLowOnTop_p=true,int maxSize_p=10000)
   {
       maxSize=maxSize_p;
      bLowOnTop=bLowOnTop_p;
       nE=0;
      a=new T[maxSize];
   };
   /////////////////////////////////////////////
   ~CUTIL_Heap(){delete [] a;};
   ////////////////////////////////////////////
   inline int GetSize()       {return nE;}
   ////////////////////////////////////////////
   inline void Clear()        { nE=0;}
   inline int LeftSon(int i)  { return (i<<1)+1;}
   inline int RightSon(int i) { return (i<<1)+2;}
   inline int GetFather(int i){ return (i-1)/2 ;}
   ////////////////////////////////////////////
   bool IsHeapAt(int i) {
       int l=LeftSon(i);
       int r=RightSon(i);
       int n=GetSize();
       if (bLowOnTop){
           if (l<n){
               if (a[l]<a[i]) {
                   return false;
               }
           }
           if (r<n){
               if (a[r]<a[i]) {
                   return false;
               }
           }
       }else{
           if (l<n){
               if (a[l]>a[i]) {
                   return false;
               }
           }
           if (r<n){
               if (a[r]>a[i]) {
                   return false;
               }
           }

       }
       return true;
   }
   /////////////////////////////////////////////////////////////////////
   bool CheckHeap(){
       for (int i=0;i<GetSize();i++){
           if (!IsHeapAt(i)) return false;
       }
       return true;
   }
   /////////////////////////////////////////////////////////////////////
   void Heapify(int i){
      int l=LeftSon(i);
      int r=RightSon(i);
      int extrem;
      int n=GetSize();
      if (bLowOnTop){
         if (l<n&&a[l]<a[i]) extrem=l;
         else extrem=i;
         if (r<n&&a[r]<a[extrem]) extrem=r;
      }
      else{
         if (l<n&&a[l]>a[i]) extrem=l;
         else extrem=i;
         if (r<n&&a[r]>a[extrem]) extrem=r;
      }
      if (extrem!=i) {
         T tmp=a[i];
         a[i]=a[extrem];
         a[extrem]=tmp;
         Heapify(extrem);
      }
   }
   /////////////////////////////////////////////
   void BuildHeap(){
      int n=nE;
      if (n<=1) return;
      int lastparent=GetFather(n-1);
      for (int i=lastParent;i>=0;i--){
         Heapify(i);
      }
   }
```

```
////////////////////////////////////////////
T ExtractTop(){
   int n=nE;
   if (n<1){
      //should not happen
      return T();
   }
   T top=a[0];
   a[0]=a[n-1];
   nE--;
   Heapify(0);
   return top;
}
////////////////////////////////////////////
T RemoveAt(int i){
   int n=nE;
   T t=a[i];
   a[i]=a[n-1];
   a.RemoveAt(n-1);
    if (i!=n-1){
        i=Climb(i);
       Heapify(i);
   }
   return t;
}
////////////////////////////////////////////
void ReplaceTop(T e){
   if (!nE) {
      //should not happen:
      return;
   }
   a[0]=e;
   nE--;
   Heapify(0);
}
////////////////////////////////////////////
void Insert(T key){
   if (nE>=maxSize) {
      //Error message: OVERFLOW
      return;
   }
   a[nE++]=key;
   int i=nE-1;
   if (i==0) return;
   int f=GetFather(i);
   if (bLowOnTop){
      while (i>0&&a[f]>key){
         a[i]=a[f];
         i=f;
         if (i!=0) f=GetFather(i);
      }
   }else{
      while (i>0&&a[f]<key){
         a[i]=a[f];
         i=f;
         if (i!=0) f=GetFather(i);
      }
   }
   a[i]=key;
}
////////////////////////////////////////////
int Climb(int i){
   if (i==0) return 0;
    T key=a[i];
   int f=GetFather(i);
   if (bLowOnTop){
      while (i>0&&a[f]>key){
         a[i]=a[f];
```

```
            i=f;
            if (i!=0) f=GetFather(i);
         }
      }else{
         while (i>0&&a[f]<key){
            a[i]=a[f];
            i=f;
            if (i!=0) f=GetFather(i);
         }
      }
      a[i]=key;
       return i;
   }
};
/////////////////////////////////////////////////////////////////////////////////////////////
#endif
```

B.11 UTIL_TOPFX.h

This file implements an efficient queue for the drops. It takes advantage of the fact, that the maximum number of elements in the queue cannot exceed the number of cells.

```
#ifndef UTIL_TOPFX
#define UTIL_TOPFX

template <class ElementTyp> class CUTIL_TopfX { public:
   CUTIL_TopfX(long nnmax){nmax=nnmax;n=0;size=sizeof(ElementTyp);pMem=new ElementTyp[nmax];};
   virtual ~CUTIL_TopfX(){delete [] pMem;};
   void Append(ElementTyp el);
   void Delete(long i);
   void DeleteLast();
   bool IsEmpty(){return n==0;};
   void Clean(){n=0;}
   ElementTyp GetAt(long i);
   ElementTyp GetLast();
   ElementTyp ExtractLast();
   inline GetIndexOfLast(){return n-1;};
   ElementTyp* GetAndAppend();
   long nmax;
   long n;
   long size;
   ElementTyp*pMem;
};
///////////////////////////////////////////////////////////////////////////
template <class ElementTyp> void
CUTIL_TopfX<ElementTyp>::Append(ElementTyp el) {
   pMem[n]=el;
   n++;
}
///////////////////////////////////////////////////////////////////////////
template <class ElementTyp> void
CUTIL_TopfX<ElementTyp>::Delete(long i) {
   pMem[i]=pMem[n-1]
   n--;
}
///////////////////////////////////////////////////////////////////////////
template <class ElementTyp> void
CUTIL_TopfX<ElementTyp>::DeleteLast() {
   n--;
}
///////////////////////////////////////////////////////////////////////////
template <class ElementTyp> ElementTyp
CUTIL_TopfX<ElementTyp>::ExtractLast() {
    ElementTyp a=pMem[n-1];
    n--;
    return a;
}
//////////////////////////////////////////////////////////////////////////
template <class ElementTyp> ElementTyp
CUTIL_TopfX<ElementTyp>::GetAt(long i) {
   return pMem[i];
}
//////////////////////////////////////////////////////////////////////////
template <class ElementTyp> ElementTyp
CUTIL_TopfX<ElementTyp>::GetLast() {
   return pMem[n-1];
}
//////////////////////////////////////////////////////////////////////////
template <class ElementTyp> ElementTyp*
CUTIL_TopfX<ElementTyp>::GetAndAppend() {
   return &pMem[n++];
}
#endif
```