Some Lessons Learned from Coding the Burns Line Extraction Algorithm in the DARPA Image Understanding Environment [*]

J. Ross Beveridge   Chris Graves   Chris Lesher
Colorado State University
ross/gravesc/lesher@cs.colostate.edu

October 1, 1996

Technical Report CS-96-125

[*]This work was sponsored by the Defense Advanced Research Projects Agency (ARPA) through the Topographic Engineering Center under the auspices of the U.S. Army Research Office Scientific Services Program and monitored by Battelle. (Grant number DAAL03-91-C-0034)

Some Lessons Learned from Coding the Burns Line Extraction Algorithm in the DARPA Image Understanding Environment [*]

J. Ross Beveridge   Chris Graves   Chris Lesher
Colorado State University
ross/gravesc/lesher@cs.colostate.edu

October 1, 1996

Abstract

A complete implementation of the Burns Line Extraction Algorithm has been developed within the IUE. It exercises a number of IUE object classes, including images, image regions, 3D planes and lines, and image line segments. Significant code segments are included showing how the IUE can be used to develop such an application. These are annotated with comments regarding the representational and functional adequacy of the current IUE. Overall, the objects available within the IUE lead to an elegant implementation. Run-times are reported for both the IUE version compared and the straight C version upon which it is based. These numbers demonstrate that a combination of factors conspired to make IUE V1.0 extremely slow. Running a fair sized image could take 22 minutes compared to 15 seconds in straight C. Fortunately, in moving from IUE V1.0 to V1.2 these times drop by well over an order of magnitude and bring the IUE implementation run time to around 90 seconds. Some additional overhead will always be associated with C++ versus C, and the IUE is now approaching the level of performance a user should expect and desire.

[*]This work was sponsored by the Defense Advanced Research Projects Agency (ARPA) through the Topographic Engineering Center under the auspices of the U.S. Army Research Office Scientific Services Program and monitored by Battelle. (Grant number DAAL03-91-C-0034)

Contents

1 Introduction **1**

1.1 Overview of the Burns Algorithm . . . 2

2 Representation, Implementation, and Functionality **2**

2.1 Images & Convolution . . . 5

2.1.1 Implementation . . . 5

2.1.2 Representation and Functionality . . . 7

2.2 Forming Image Regions . . . 7

2.2.1 Implementation . . . 7

2.2.2 Representation & Functionality . . . 7

2.2.3 Label Planes Should Be a Class in the IUE . . . 10

2.3 Pixel Voting . . . 11

2.3.1 Implementation . . . 11

2.3.2 Representation & Functionality: Using Dynamic Attributes . . . 11

2.4 Plane, Line and Line Segment Spatial Objects . . . 11

2.4.1 Implementation . . . 11

2.4.2 Representation & Functionality . . . 13

3 Speed **14**

3.1 Run-times Broken Down by Functionality . . . 14

3.2 Run-time Sensitivity Studies . . . 16

3.3 Conclusions from Run-time Tests . . . 17

4 Summary **21**

4.1 Ease of Use . . . 21

4.2 Final Observations . . . 21

List of Tables

1 Run-time Comparisons on a 128×128 Image 15
2 Run-time Comparisons on a 512×480 Image 15
3 Run-times for different versions of Pixel Connected Components. 16

List of Figures

1 Example of Lines Produced Using the IUE 3
2 Code Segment: Overview of Complete IUE Task 4
3 Code Segment: Estimating The Gradient 5
4 Code Segment: Computing Gradient Magnitude and Two Gradient Orientation Bucket Images. 6
5 Code Segment: Building the Edge Support Regions 8
6 Code Segment: The Floodfill Algorithm 9
7 Code Segment: Voting for the Larger Edge Support Region 12
8 Code Segment 1: This is the favored code for which runtime above is reported. Observe a single `IUE_lattice_pointset_2d` object `tempLat` is used. Each time through the loop it is cleared, and new pixels added by the `floodFill` function. 18
9 Code Segment 2: Create a `new IUE_lattice_pointset_2d` object into which to assemble the pixels of each successive region. 19
10 Code Segment 3: Assemble pixels directly into an `IUE_image_region_2d` object. . . . 19
11 Code Segment 4: Use a sequence of regions rather than pointers to regions. 20
12 Code Segment 5: Compile-time allocate an array of regions. Uncovers a strange dependency run-time dependency between KBV file IO and exiting function where array is declared. 20

1 Introduction

The Burns line extraction algorithm has been chosen as a representative feature extraction algorithm with which to test the facilities of the IUE. There already exist a clean and simple C alone implementation developed by us which is the prototype for the IUE version. Chris Graves has been developing this code for several months, with previous versions of the IUE going back to v0.9b. The current version is relatively simple: the entire `iue_task_burns` file is 800 lines including blanks.

This document has essentially two parts. The first describes the implementation and makes some comments about the representational adequacy of the IUE. On the whole, the IUE's object class representations are excellent, and there were few if any points where there failed to be an obvious choice. This first section will also discuss the functional maturity of the IUE: how much of what the representation suggests will be present actually exist in code today? By functionality, we are referring to the IUE methods available — having chosen a representation, what methods are available to carry out the desired task.

Overall, the object classes, on-line HTML documentation, and code generator have made developing the Burns Algorithm a pleasure. Already the IUE is a rich language for representing algorithms and we hope the code segments presented below will form the basis for a subsequent tutorial on writing code within the IUE. In saying this, obviously the code can be further refined to better exploit the capabilities of the IUE. For instance, the current code does not yet take advantage of the most recent and efficient image accessors.

The second part of this document addresses the issue of speed. To summarize briefly what is laid out in detail below, the IUE has seen a dramatic improvement in performance over the past year. In an earlier draft of this technical report, prior to testing with IUE V1.2, it was reported that a 523×480 image took 22 minutes to run through the algorithm. By comparison, a straight C implementation took 15 seconds. At the time these runs were made, this caused us considerable concern.

However, as was noted in our earlier draft, there are many opportunities for evolving C++ compiler technology to extract one to two order of magnitude increases in run time speed. As we can now report, just such a dramatic improvement has been witnessed between IUE V1.0 and IUE V1.2. **The task previously taking 22 minutes now completes in less than 2 minutes**. The run times, as given in detail below, suggest the IUE version of the Burne Line Extraction algorithm is now running 6 to 8 times slower than the straigth C version. While obviously one would like to see no penalty when moving from C to C++, such is not a realistic expectation. While continuing efforts are sure to be made to improve IUE run time performance, the IUE is now within the proper order of magnitude range relative to straight C.

1.1 Overview of the Burns Algorithm

The Burns Algorithm was first described in [BHR86]. The algorithm takes in a single image and returns straight line segments representing regions of aligned gradient orientation in the image. Figure 1 shows a 512 by 480 image as well as the straight line segments produced for this image using the IUE.

The Burns algorithm uses the following general types of objects/operations:

1. Byte Images.
2. Convolution to extract gradient magnitude and gradient orientation images.
3. Image segmentation into regions based on 4-connected orientation directions.
4. Label plane to identify each line-support region.
5. 3D-plane built from each orientation region, weighted by grad mag.
6. 2D-line extracted by intersecting 3D-plane with average-intensity-value (over region) 3D plane. (actually a 3D-line, but only need X-Y plane components)
7. 2D-line segment created when 2D-line is clipped to region.

This array of objects/operations makes means that the Burns algorithm exercises various levels of the IUE ranging from basic image convolution up through creation of spatial-objects. The steps carried out by the algorithm are laid out below.

2 Representation, Implementation, and Functionality

This section presents examples pulled from the full code of the Burns Algorithms. The complete code may be obtained from our ftp site: `ftp.cs.colostate.edu` and directory `pub/vision/iue`.

The overall structure of the Burns Algorithm is indicated in Figure 2. This example shows contents of the file `iue_task_burns.cc` with most all the internal code removed. It shows that the overall structure is first the definition of the IUE task, followed by the implementation of the flood fill algorithm used to perform the connected components pixel grouping, followed by the `main` code body.

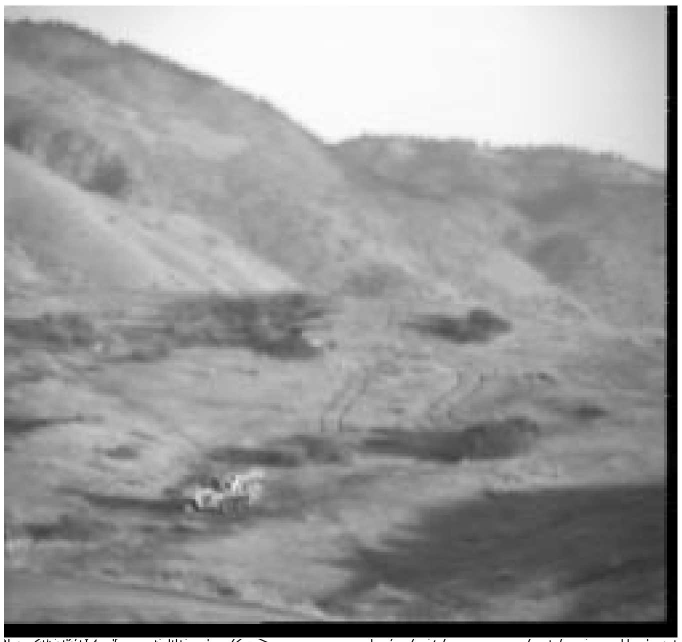
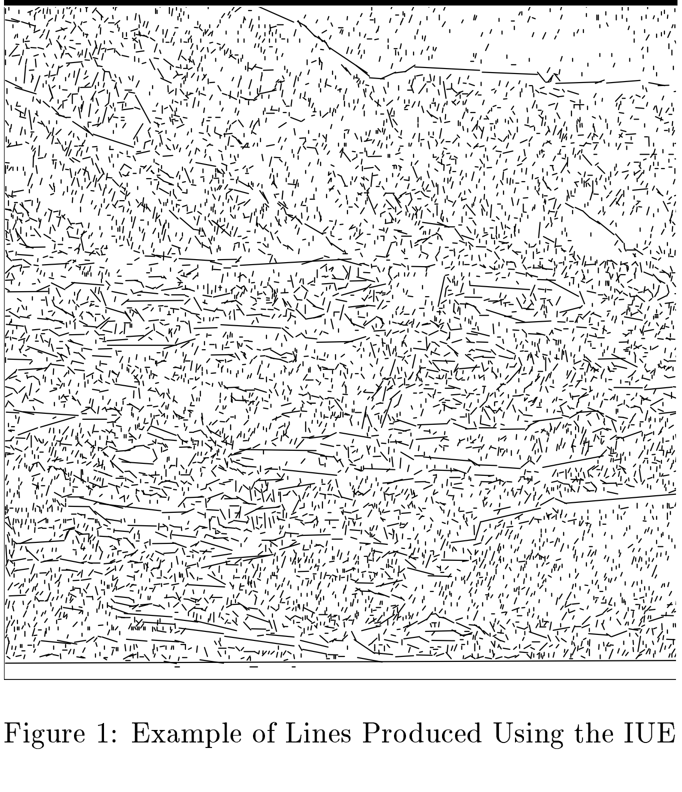

Figure 1: Example of Lines Produced Using the IUE

```
#include "iue_task_burns.h"
#include "../stopwatch.h"

stopwatch stopWatch;

IUE_TASK(burns)
(
   ...  Parameters ...

{ // begin IUE_task_burns

...

} // end IUE_task_burns

IUE_INT
floodFill()
{
   ...
   return numPix;
}

void
main(int argc, char *argv[])
{
   stopWatch.start();

   ...

   IUE_task_burns(...);


   stopWatch.printTotal("Finished writing out ASCII lines");

} // end main
```

Figure 2: Code Segment: Overview of Complete IUE Task

```
for (y = 0; y < yMax; y++) { // Iterate through image
   for (x = 0; x < xMax; x++) {
      if(y == yMax-1 || x == xMax-1) // last row/col are zeroed out
         horz = vert = 0;
      else {
         if(gradOp == sobel) {
         < code omitted for brevity >
         }
         else if(gradOp == oneByTwo) {
            inImg.get_pixel(pix,x,y);
            inImg.get_pixel(pix2,x,y+1);

            horz = pix - pix2;

            inImg.get_pixel(pix,x,y);
            inImg.get_pixel(pix2,x+1,y);

            vert = pix - pix2;
         }
      }
```

Figure 3: Code Segment: Estimating The Gradient

2.1 Images & Convolution

2.1.1 Implementation

The Burns Algorithm takes as input a single image. The code segment used to compute the gradient orientation using a simple one-by-two mask is shown in Figure 3. Note no explicit mask is used, but instead adjacent pixel values are gotten from the image and their difference taken. A similar implementation of the Sobel operator also exists in the full algorithm.

After the horizontal and vertical components of the gradient at each pixel have been estimated, the next step is to produce a gradient magnitude image and two gradient orientation images. These gradient orientation images are derived by partitioning the gradient orientation into fixed width *buckets*. Typically 8 buckets are used. These bucket labels are used later to generate connected regions of similar gradient orientation: these are called line-support regions. Two orientation images are produced, one with the bucket boundaries shifted 50% relative to the others. This code segment where this is done is shown in Figure 4.

```
      // assign magnitude and orientation
      pix = (IUE_UINT8)(sqrt((IUE_DOUBLE)(horz*horz+vert*vert))+0.5);
      outGM->set_pixel(pix,x,y);

      if (horz == 0 && vert == 0) {
         outGM->set_pixel(0,x,y);
         outGO1->set_pixel(0,x,y);
         if(use2ndGradO)
            outGO2->set_pixel(0,x,y);
      }
      else if(pix < magThresh) {
         outGO1->set_pixel(0,x,y);
         if(use2ndGradO)
            outGO2->set_pixel(0,x,y);
      }
      else { // compute gradient orientation (-PI to PI)
         tempOrient1 = atan2((double)vert,(double)horz);

         // shift to zero-2PI range
         tempOrient1 += PI;
         tempOrient2 = tempOrient1;

         // account for bucket offset-- first image
         tempOrient1 += buckOffset1/100. * ((2. * PI)/numBuckets);

         // account for offset shift over 2PI
         if (tempOrient1 > (2. * PI))
            tempOrient1 -= (2. * PI);

         // compute integer bucket value of orientation and store
         tempOrient1 *= numBuckets/(2. * PI);
         outGO1->set_pixel((IUE_UINT8) floor(tempOrient1) + 1,x,y);

         if(use2ndGradO) {
            // account for bucket offset-- second image
            tempOrient2 += buckOffset2/100. * ((2.* PI)/numBuckets);

            // account for offset shift over 2PI
            if (tempOrient2 > (2. * PI ))
               tempOrient2 -= (2. * PI);

            // compute integer bucket value of orientation and store
            tempOrient2 *= numBuckets/(2. * PI);
            outGO2->set_pixel((IUE_UINT8) floor(tempOrient2) + 1,x,y);
         }
      } // end else
   }
} // end double for
stopWatch.printLap("Finished gradient images");
```

2.1.2 Representation and Functionality

Overall, the IUE image handling capabilities have worked very well, and images are the most mature and robust of all the classes with which we have worked so far.

In future versions, a case could be made for using the functionality of the filter classes to perform the gradient magnitude and orientation estimation outside of the Burns Algorithm. However, this is not a critical issue at this point. Moreover, while it may be tempting to break the algorithm into pieces in this way, it actually has been found to be cumbersome to use such a version. The UMass version of the Burns Algorithm under Khoros is highly modular, with separate glyphs for different steps. One consequence is many students find using the algorithm confusing and make mistakes in setting it up. Hiding the details in this case makes for a nicer end product.

There does not appear in our experience to be a significant advantage in the current version of images over the previous version. This is in part because we are not yet taking advantage of the `get-row` access capabilities. Perhaps the underlying representation is much better, but something that has been lost are simple accessors that return a value as was done in previous versions. This would make aspects of coding a lot cleaner, and more in line with other IUE methods.

The IUE is still missing practical functionality for various image types; IO methods are needed for various image formats.

2.2 Forming Image Regions

2.2.1 Implementation

A recursive floodfill algorithm is used to perform the connected components of each region in the orientation images. Floodfill is a standard computer graphics algorithm and while not quite as efficient as some others, it is simple to understand, to code, and it run in quite reasonable time. The essential idea is that seed pixels are repeatedly selected from the image, regions grown or flooded until no further pixels can be added. This process is repeated until all pixels have been examined. A label-plane serves two purposes in this algorithm. First, it keeps track of the region number to which each pixel belongs. Second, it is used to keep track of pixels already included in regions. The code for the region formation is shown in Figure 5. Note this code uses a function called `floodFill` which is shown in Figure 6.

2.2.2 Representation & Functionality

The current representation of image regions is less than adequate. However, it is understood that this class is is only partially implemented. Each defined constructor requires a pointer to an image,

```
//**********************************************************************
// REGIONS

// Create region image one
for (y = 0; y < yMax-1; y++) // -1 because last row/col always 0
   for (x = 0; x < xMax-1; x++) {
      outGO1->get_pixel(pix,x,y);
      // do not process pixels without orientation, and do not
      // process pixels already covered
      if(pix > 0 && labelPlane1(x,y) == -1) {
         tempLat = emptyLat;
         // if region size meets minimum, add to sequence
         if(floodFill(*outGO1,labelPlane1,tempLat,
                      xMax-1,yMax-1,x,y,pix,regLabel) >= pixPerReg) {
            tempReg = new IUE_image_region_2d(tempLat,&inImg);
            regions.append(tempReg);
            DAtype<IUE_INT>::put(*regions.last(),"votes",0);
            regLabel++;
         }
         else // region too small, tag with 999999 so all pixels skipped
            floodFill(*outGO1,labelPlane1,tempLat,
                      xMax-1,yMax-1,x,y,pix,999999);
      }

   }// end double for

// Return all too-small regions back to -1 label
for(a2dIter=labelPlane1.begin(); a2dIter!=labelPlane1.end(); a2dIter++)
   if(*a2dIter == 999999)
      *a2dIter = -1;

stopWatch.printLap("Finished region segmentation 1");
```

Figure 5: Code Segment: Building the Edge Support Regions

```
IUE_INT
floodFill(IUE_scalar_image_2d_of<IUE_UINT8> &go,
          IUE_array_2d<IUE_INT> &labelPlane,
          IUE_lattice_pointset_2d &tempLat,
          IUE_INT xMax, IUE_INT yMax, IUE_INT x, IUE_INT y,
          IUE_UINT8 pixel, IUE_INT label)
{
   IUE_INT
      numPix = 0;

   IUE_UINT8
      pix;

   // if flood pixel value == current x,y pixel value &&
   //  current pixel does not already belong to a region.
   go.get_pixel(pix,x,y);
   if(pix == pixel && labelPlane(x,y) != label) {
      numPix++; // increase number of pixels for this lattice-2d
      labelPlane(x,y) = label;
      if(label != 999999)
         tempLat.insert(x,y);

      // flood up
      if(y > 0) // not in first row
         numPix += floodFill(go,labelPlane,tempLat,
                             xMax,yMax,x,y-1,pixel,label);
      // flood forward
      if(x < xMax-1) // not in last col
         numPix += floodFill(go,labelPlane,tempLat,
                             xMax,yMax,x+1,y,pixel,label);
      // flood back
      if(x > 0) // not in first col
         numPix += floodFill(go,labelPlane,tempLat,
                             xMax,yMax,x-1,y,pixel,label);
      //flood down
      if(y < yMax-1) // not in last row
         numPix += floodFill(go,labelPlane,tempLat,
                             xMax,yMax,x,y+1,pixel,label);
   }
   else
      return 0;

   return numPix;
}
```

Figure 6: Code Segment: The Floodfill Algorithm

which clearly is meant to tie together the region and corresponding image. However, there are only a few methods defined for regions that depend on image information. Of these, the only useful method for my application is `mean()`, which failed to work. `Mean()` depends on the soft slot `intensity_distribution`, which is not yet initialized in any of the constructors, and is disabled otherwise.

Missing is an iterator or iterators, that will return both the X-Y (in the 2D case) cell address and the intensity value for that cell. An iterator is available for `IUE_cell_lattice_2d` of course (parent to `IUE_image_region_2d`), but this has nothing to do with the image and only returns the coordinates. To obtain a mean value across the image, one must extract the X-Y coordinates from the cell-lattice-2d iterator, and then get the intensity value from the image itself. If regions are to be tied to a specific image upon creation, the intensity values should be readily available, or even stored for each region cell.

Image features should be convertible to spatial objects. A natural conversion of a 2D region would be to a discrete-functional surface. Access to the intensity values is necessary for this.

For the Burns Algorithm, a region-of-interest-2d class would be much more useful. The Burns algorithm requires information from three different images in the formation/use of each ROI. The ROI is formed on the gradient orientation image, but the orientation values are irrelevant beyond formation of the region. Values from the intensity image and gradient magnitude image are then gathered for each ROI to form a plane. An ROI class that is sibling to image-region-2d would be perfect. In fact, all methods and constructors could mirror those of image-region-2d with one change in philosophy: drop the image pointer requirement in the constructors and add it to the methods that return image related information. I used regions in my application, but only accessed the cell-lattice-2d methods.

2.2.3 Label Planes Should Be a Class in the IUE

A label plane produced during a region segmentation is common enough, and should have its own class perhaps. Natural inclination would lead me to hang it off off the spatial-index class; however, according to IUE documentation, this is just meant to efficiently index a collection of IUE-objects. Here we represent a label plane as just an array of integers, initialized with the image size. Accessing the LP with image coordinates produces an integer value, which indexes a sequence of regions producing the appropriate region covering the image location. Using a spatial-index would improve my implementation; however, I predict no gains in run-time.

A class for label plane is a good idea from a run-time standpoint, at least for this algorithm. For a 512x480 image, I was able to create as many as 28,000 regions, of which approximately 15,000 lines were created. A label plane class would eliminate the need for regions in this case, and regions-of-interest for that matter. A label plane would be created with a pointer to an image, but not necessarily tied to a particular image. All contiguous regions (according to connectivity) would be uniquely labeled and indexed by location. An option would be to form non-contiguous regions based on distinct image values. In any event, given a label plane, I would like to apply it to any arbitrary image. Methods would be similar to region-of-interest described above. Given a pointer to an image and either a region label or coordinates, I would like to know the size of region, mean intensity value, and so on. In addition, an ROI or region object should be available as a method return. Run-time would be significantly reduced with this class.

2.3 Pixel Voting

The two sets of regions are created in order to protect against orientation bucket boundaries slicing through the average orientation of a particular aligned gradient region in the image. In other words, if the boundary splits the region in one segmentation, it will not in the other. A consequence is more lines are produced than are really desired, and the way this is handled is each pixel expresses a preference for the more significant of the two regions in which it participates. In short, each pixel votes for the larger of the two regions. This will allow the final line segments produced by the algorithm to be filtered based upon how well they are supported by their member pixels: how many votes they receive.

2.3.1 Implementation

The code segment for the voting is shown in Figure 7.

2.3.2 Representation & Functionality: Using Dynamic Attributes

An interesting aspect of this part of the algorithm is it demonstrates the usefulness of dynamic attributes. It would be awkward and undesirable to create a new hard slot for each region indicating the votes cast for that region. Instead, a dynamic attribute is used to accumulate the votes.

2.4 Plane, Line and Line Segment Spatial Objects

2.4.1 Implementation

After the line-support regions have been created, a line segment abstraction for each region is computed in the following manner. First, a best-fit 3D plane is fit to the image intensity data in the region. Often this fit is weighted by the gradient magnitude at each pixel. Next, an infinite line is created by intersecting this plane with a horizontal plane with height equal to the average

```
//*************************************************************************
//VOTING
// if there are 2 region plane images then vote for each region in both
if(use2ndGradO) {
   for (y = 0; y < yMax-1; y++) // iterate through labelPlanes
      for (x = 0; x < xMax-1; x++) {

         reg1 = labelPlane1(x,y); // get reg. num. of pixel for plane one
         reg2 = labelPlane2(x,y); // get reg. num. of pixel for plane two
         if( (reg1 > -1) && (reg2 > -1) ) { // Making decisive vote
            // if region 2 at least as big as region 1, cast vote
            // for region 2
            if(regions(reg2)->area() > regions(reg1)->area()) {
               // cast vote for region 2
               vote = DAtype<IUE_INT>::get((*regions(reg2)).get("votes"));
               DAtype<IUE_INT>::put(*regions(reg2),"votes",vote+1);
            }
            else { // cast vote for region 1 if >= region 2 size
               vote = DAtype<IUE_INT>::get((*regions(reg1)).get("votes"));
               DAtype<IUE_INT>::put(*regions(reg1),"votes",vote+1);
            }

         } // end if
         else if(reg1 > -1) {         // a vote for reg of first plane
            vote = DAtype<IUE_INT>::get((*regions(reg1)).get("votes"));
            DAtype<IUE_INT>::put(*regions(reg1),"votes",vote+1);
         }
         else if(reg2 > -1) {      // a vote for reg of second plane
            vote = DAtype<IUE_INT>::get((*regions(reg2)).get("votes"));
            DAtype<IUE_INT>::put(*regions(reg2),"votes",vote+1);
         }
      } // end double for
   stopWatch.printLap("Finished voting");
} //endif
```

Figure 7: Code Segment: Voting for the Larger Edge Support Region

intensity in the region. Finally, a bounded line segment is computed by intersecting the region with the infinite line. The actual code for these processes is fairly long and is not included in this report.

A large portion of the code in this section clips the infinite lines to the regions. As the IUE matures, and intersection is supported between these two spatial object classes, this complex operation will reduce to a single call to the IUE's own intersection code. However, intersection between `IUE_parametric_line_2d` and `IUE_image_region_2d` is not currently implemented.

2.4.2 Representation & Functionality

We have found the spatial-objects for planes and lines to be excellent. Lines have been formed using the method outlined in the AAI Vision Tutor Guide. In the abstract, two planes are intersected, producing a 3D line. This line is projected straight down to the XY plane to give the desired 2D line. In practice, doing the linear algebra for plane coefficients produces the final 2D line coefficients given the average intensity value, so the plane intersection is unnecessary.

However, we took the intersection routine for a test drive anyway. The plane constructor is straightforward, as is the intersection routine. We also tried the 3D line constructor using two planes. All the lines produced were perfect according to theory — parallel with the $Z = 0$ plane. As a further test, we extracted the 2D line from the 3D line, checking in every case that the Z component of the tangent vector was zero. Comparing compared these lines with the 2D lines created directly from coefficients, the results were perfect: an assert function based on equality between the two differently formed 2D lines never fails.

Two form bounded 2D line segments, the infinite lines are clipped first to the bounding box of region, and then to the region pixels themselves. There was a problem during this process due to the current implementation since it was possible only to get the discrete version of the axis-aligned extents box for the region. In the IUE, it appears that "cells" are represented by the point in the lower left corner, e.g. the $(0, 0)$ cell is the space from $X = Y = 0$ up to but not including $X = Y = 1$: a one by one box. When returning the discrete bounding box, all cells on the right-hand side of the regions are essentially lost.

Since our initial clipping was to the discrete box, we lost some representation of the region. When clipping to the region, a parameterized point on the line is checked to see if it is "in" the region. This is the stopping criteria since the line is guaranteed to pass through the region. The "in" function simply truncates the point to a discrete version using the floor function — in accordance with the cell representation — and checks to see if the point is in the cell-lattice. Several cases were found where the "in" function failed to clip the line to the region. If the line is initially clipped using the correct bounding box, this problem won't occur. This problem was solved by first rounding the point on the line to a discrete point before calling the region.in() method. In any event, these problems will be fixed when it is possible to use the correct but yet to be implemented axis-aligned box.

3 Speed

3.1 Run-times Broken Down by Functionality

Part of our experimentation has been looking at differences in run-time due to code style. Tests were run using two versions of the IUE: 1.0b1 and 1.2alpha. It was found that version 1.2alpha was much faster than 1.0b1 (which was itself much faster than version 0.9).

Times were measured on a Sparc 20 running Solaris with 256MB of RAM. The times are obtained using a stopwatch facility we built on top of the `clock()` function. The times are check-pointed after completion of key steps in the algorithm and some of the check-pointing calls could be seen in the code segments shown above. Here let us review briefly the stages in the algorithm for which times are reported:

Gradient images Includes reading the input image from a file, convolving the image with the gradient mask, and generating three additional images: one gradient magnitude and two gradient orientation images.

Region Segmentation Performs a connected components operation using a flood-fill style algorithm to build aligned-gradient-orientation regions. A label-plane is constructed along with a sequence of regions. In the nomenclature of the Burns algorithm, these are called line-support regions. Region segmentation is done twice and the two runtimes are always nearly identical.

Voting Each image pixel belongs to two line-support regions. In this step, each pixel "votes" for the larger of these two regions; more votes mean a larger and more salient line structure. This part of the code uses dynamic attributes to accumulate votes.

Compute Lines One IUE-parametric-line-segment-2d is created for each line-support region. Conceptually, this is accomplished by taking the intersection of a plane fitted to the sampled intensity surface in the line-support region and a horizontal plane with height set to the average intensity of the line-support region. This infinite line is then clipped so as to lie inside the line-support region.

Tables 1 and 2 show the fastest times obtained for IUE versions 1.0b1 and 1.2alpha on a small and medium sized image, respectively. Run-times for a straight C version of the algorithm running stand-alone are also shown.

Algorithm Step	Straight C Code	IUE V1.0b1	IUE V1.2α	Improvement V1.0 / V1.2	Penalty V1.2 / Straight C
Gradients	0.19	2.0	0.25	800%	132%
Segmentation 1	0.10	23.0	1.89	1,217%	1,890%
Segmentation 2	0.10	23.0	1.80	1,277%	1,800%
Voting	0.17	4.0	1.16	345%	682%
Extract Lines	0.45	59.0	2.98	1,980%	662%
Total Elapsed	1.01	113.0	8.08	1,399%	800%

Table 1: Run-time Comparisons on a 128×128 Image. Run times are in seconds on a Sparc 20 and are for a stand-alone straight C implementation of the Burns Line Extraction algorithm as compared to the IUE implementation in IUE Versions 1.0 and 1.2

Algorithm Step	Straight C Code	IUE V1.0b1	IUE V1.2α	Improvement V1.0 / V1.2	Penalty V1.2 / Straight C
Gradients	2.78	23.0	3.79	607%	136%
Segmentation 1	1.76	309.0	22.71	1,361%	1,290%
Segmentation 2	1.71	314.0	24.02	1,307%	1,405%
Voting	1.92	47.0	11.78	399%	614%
Extract Lines	6.30	643.0	30.98	2,076%	492%
Total Elapsed	14.47	1,364.0	93.28	1,462%	645%

Table 2: Run-time Comparisons on a 512×480 Image. Run times are in seconds on a Sparc 20 and are for a stand-alone straight C implementation of the Burns Line Extraction algorithm as compared to the IUE implementation in IUE Versions 1.0 and 1.2

Note that the timings for the two versions were done when the machine had different loads. For IUE version 1.0b1 the machine was virtually unloaded, whie for the 1.2alpha test the machine had an average load of 1.37. So this means that the improvement in speed between the two versions is even better than the timings indicate, since on an unloaded machine the times for the 1.2alpha version would be lower.

The last two columns in Tables 1 and 2 show the relative speedup for IUE V1.2 versus V1.0 and the relative slow-down of V1.2 relative to the straight C code. Note that the improvements seen in the past year in the IUE have moved IUE V1.2 more than half the distance between the worst case performance of V1.0 and the best case performance of the straight C code.

Code Segment	Figure	Pixel Collector	Re-use	Sequence Representation	Run-time 1.0b1	Run-time 1.2alpha
1	8	IUE_lattice_pointset_2d	Yes	Sequence of Pointers	21.0	1.85
2	9	IUE_lattice_pointset_2d	No	Sequence of Pointers	29.5	2.01
3	10	IUE_image_region_2d	No	Sequence of Pointers	38.0	1.84
4	11	IUE_lattice_pointset_2d	No	Sequence of Regions	45.5	3.70
5	12	IUE_lattice_pointset_2d	No	Array of Regions	21.0	2.65

Table 3: Run-times for different versions of Pixel Connected Components.

3.2 Run-time Sensitivity Studies

Most of the time is spent either building the connected components for the edge support region or in extracting the straight line segments from these regions. This section presents a first set of sensitivity studies carried out using slight variations of the connected components code. These studies illustrate how small changes in choice of objects and coding style alter run-time. The only differences between the cases are the changes in the connected components section and the specific code is shown.

The code segment in Figure 8 is the one considered best. The run-times reported above are for this case. Enough code is shown to include the data types and comments. The variations upon this code segment in Figure 8 are tested to show runtime implications of using different object classes and different object creation/deletion strategies.

Two alternative IUE object classes are used to gather pixels into sets subsequently used to construct regions of type `IUE_image_region_2d`: `IUE_lattice_pointset_2d` and `IUE_image_region_2d`. The choice of object class can affect the runtime. As expected, changes which lead to underlying changes in how memory is managed also alter run-times. One key issue explored in the tests is what to do when objects are conceptually needed for a short time and are then discarded. One option is to create a single scratch object and use it repeatedly. The other option is to create a new object each time one is needed and to delete it when we are finished using it. Observe that in Figure 8, the former approach is taken.

Table 3 summarizes the results of the sensitivity study. Here is an explanation of what is reported in each column of the table.

Figure The figure in which the code segment is displayed.

Pixel Collector what type of IUE object is used to collect pixels into regions are they are being constructed by the recursive flood-fill algorithm.

Re-use 'Yes' indicates a single scratch object is re-used to assemble each successive region. 'No' indicates a new pixel collector object is created each time through the loop.

Sequence Representation The final result is a sequence of regions which can be represented as a sequence of pointers, `IUE_indexed_sequence_via_array<IUE_image_region_2d *>`, a sequence of regions, `IUE_indexed_sequence_via_array<IUE_image_region_2d>`, or an array of regions, `IUE_array_1d<IUE_image_region_2d>`.

Run-time 1.0b1 The average number of seconds required to perform connected components for the two segmentations using IUE 1.0b1.

Run-time 1.2alpha The average number of seconds required to perform connected components for the two segmentations using IUE 1.2alpha.

Code segment 3, in Figure 10, uses `IUE_image_region_2d` objects directly to assemble the pixels included in the region. This is not as efficient as the previous code segments, even though it conceptually limits the number of objects being created and builds the regions in the most obviously appropriate object class. Even neglecting to `delete` unwanted objects, the run-time is 28 seconds. Since regions found to be too small should be deleted, including this `delete` further increases run-time to 38 seconds. This segment therefore requires twice the time of the first code segment.

Code segment 4, in Figure 11, is similar to code segment 2, but uses a sequence of regions rather than pointers to regions. This forces a more significant `new` operation during the sequence append, and also adds the delete on `tempReg`. In developing this code segment, a bug was found in the dynamic attribute mechanism that caused dynamic attributes to fail to be copied into the sequence's region. This bug has been reported and is being addressed. This version is easily the slowest of the five.

Code segment 5, in Figure 12, is similar to code segment 1 with one change: a fixed size array of regions is used instead of a sequence of regions to return the result of the connected components operation. In this case, enough memory is allocated ahead of time in an array to contain the maximum number of expected regions: 10,000 in the example. Then a `new` operation is done "in place", similar to the way a sequence append works using STL code (we think). Thus, object initialization functions will execute but now new memory need be allocated. One might suspect some time would be saved by giving the address of this already existing object. However the run-times are not much different from those for code segment 1.

3.3 Conclusions from Run-time Tests

It appears in all the above examples that IUE memory deallocation is quite expensive; several of the code-segments above show significant savings is run-time when dropping even a single `delete`

```
   IUE_array_2d<IUE_INT>
      labelPlane1(0,xMax-1,0,yMax-1,-1),
   IUE_indexed_sequence_via_array<IUE_image_region_2d *>
      regions;
   IUE_lattice_pointset_2d
      tempLat,
      emptyLat;

   // Create region image one
   for (y = 0; y < yMax-1; y++) // -1 because last row/col always 0
     for (x = 0; x < xMax-1; x++) {
       outG01->get_pixel(pix,x,y);
       // do not process pixels without orientation, and do not
       // process pixels already covered
       if(pix > 0 && labelPlane1(x,y) == -1) {
---->    tempLat = emptyLat;
         // if region size meets minimum, add to sequence
         if(floodFill(*outG01,labelPlane1,tempLat,
                      xMax-1,yMax-1,x,y,pix,regLabel) >= pixPerReg) {
---->      tempReg = new IUE_image_region_2d(tempLat,&inImg);
           DAtype<IUE_INT>::put(*tempReg,"votes",0);
---->      regions.append(tempReg);
           regLabel++;
         } else {
           // region too small, tag with 999999 so all pixels skipped
           floodFill(*outG01,labelPlane1,tempLat,
                     xMax-1,yMax-1,x,y,pix,999999);
         }
       }
     }
```

Figure 8: Code Segment 1: This is the favored code for which runtime above is reported. Observe a single `IUE_lattice_pointset_2d` object `tempLat` is used. Each time through the loop it is cleared, and new pixels added by the `floodFill` function.

```
     if(pix > 0 && labelPlane1(x,y) == -1) {
---->  tempLat = new IUE_lattice_pointset_2d;
       if(floodFill(*outG01,labelPlane1,*tempLat,
                    xMax-1,yMax-1,x,y,pix,regLabel) >= pixPerReg) {
         tempReg = new IUE_image_region_2d(*tempLat,&inImg);
         DAtype<IUE_INT>::put(*tempReg,"votes",0);
         regions.append(tempReg);
         regLabel++;
       } else  {
         floodFill(*outG01,labelPlane1,*tempLat,
                   xMax-1,yMax-1,x,y,pix,999999);
       }
       delete tempLat;
     }
```

Figure 9: Code Segment 2: Create a `new` `IUE_lattice_pointset_2d` object into which to assemble the pixels of each successive region.

```
     if(pix > 0 && labelPlane1(x,y) == -1) {
---->  tempReg = new IUE_image_region_2d;
       if(floodFill(*outG01,labelPlane1,*tempReg,
                    xMax-1,yMax-1,x,y,pix,regLabel) >= pixPerReg) {
         DAtype<IUE_INT>::put(*tempReg,"votes",0);
         regions.append(tempReg);
         regLabel++;
       } else {
         floodFill(*outG01,labelPlane1,*tempReg,
                   xMax-1,yMax-1,x,y,pix,999999);
         delete tempReg;
       }
```

Figure 10: Code Segment 3: Assemble pixels directly into an `IUE_image_region_2d` object.

```
   IUE_indexed_sequence_via_array<IUE_image_region_2d>
      regions;

   if(pix > 0 && labelPlane1(x,y) == -1) {
     tempLat = new IUE_lattice_pointset_2d;
     if(floodFill(*outG01,labelPlane1,*tempLat,
                  xMax-1,yMax-1,x,y,pix,regLabel) >= pixPerReg) {
       tempReg = new IUE_image_region_2d(*tempLat,&inImg);
---->  DAtype<IUE_INT>::put(*tempReg,"votes",0);
---->  regions.append(*tempReg);
---->  delete (tempReg);
       regLabel++;
     } else {
       floodFill(*outG01,labelPlane1,*tempLat,
                 xMax-1,yMax-1,x,y,pix,999999);
     }
     delete tempLat;
   }
```

Figure 11: Code Segment 4: Use a sequence of regions rather than pointers to regions.

```
IUE_array_1d<IUE_image_region_2d> regions(0,10000);

if(pix > 0 && labelPlane1(x,y) == -1) {
  tempLattice = new IUE_lattice_pointset_2d;
  if(floodFill(*outG01,labelPlane1,*tempLattice,
               xMax-1,yMax-1,x,y,pix,regLabel) >= pixPerReg) {
    regions(regLabel) = IUE_image_region_2d(*tempLattice, &inImg);
    DAtype<IUE_INT>::put(regions(regLabel), "votes", 0);
    regLabel++;
  } else {
    floodFill(*outG01,labelPlane1,*tempLattice,
              xMax-1,yMax-1,x,y,pix,999999);
  }
  delete tempLattice;
}
```

Figure 12: Code Segment 5: Compile-time allocate an array of regions. Uncovers a strange dependency run-time dependency between KBV file IO and exiting function where array is declared.

operation. It also appears that at least in the cases tested, object creation with `new` is not a source of appreciable overhead. Neither of these findings is particularly obvious, and more work must be done to better understand what is taking place.

4 Summary

4.1 Ease of Use

The following are some personal observations of Chris Graves after spending many months working with the early releases of the IUE. First, The IUE is large, complex and is not fully implemented yet. Two properties of the IUE make it manageable in its current form: the associated HTML documentation and the uniformity imposed by the latex code generator.

I, Chris Graves, find the HTML pages to be faithful to the corresponding source code. For the most part, I only had to inspect source code when the class documentation header indicated “partial” implementation. Any criticism of the HTML pages are addressed in the recent release of v1.0b2, which improves on this already excellent tool.

When inspecting source code, the uniform style evident throughout the IUE makes this an easy task once the format is understood.

4.2 Final Observations

Our experience confirms what is well understood within the technical community developing the IUE: that the emphasis to date has been on representation primarily, functionality secondarily, and concerns of efficiency last. The one area where efficiency has been a concern from the beginning is the image classes, and the results shown above suggest images are doing fine.

The IUE design and it’s hundreds of classes covers a great array of IUE areas. The current IUE version still seeks to flesh out the representation of many classes and functionality lags representation. Even with the tremendous effort underway, many functionalities promised by the representation are not yet present. The intersection routines for lines and regions are one example. At one level this seems a failing, but at another it points to one of the great strengths of the IUE design. This strength is how clearly the representation uniformly suggests an incredible array of functionalities. This, in turn, will encourage development of shared implementations as opposed to the more current norm of building most components from scratch.

Provided the IUE continues to mature and develop, the functionality will come. There is still a bit of a circular dependency, with more functionality needed to attract users while more users are needed to expand the functionality. In our experience so far, it has been exciting to find so much already in place. For example, even though they were not in the end needed, the functionality to handle 3D planes and plane intersections represents a dramatic step forward in supporting user development. These classes are easy to use and the routines work flawlessly. Even with significant run-time slowdown, such convenience is vastly preferable to the alternative — coding all from scratch.

Overall, getting the IUE to the point it has reached is a significant achievement. The code generator, the HTML documentation, the templated code, and perhaps more importantly their complete integration through the code-generation process, is a fantastic advancement. It is the envy of some of our software engineering faculty who have taken serious note of the IUE and are preparing to study it as a test case. Few systems have such excellent hyper-text documentation. The IUE project is at the vanguard in this respect. We have taken the time in our own use of the IUE to learn the tools and find they are essential when managing a complex object oriented system. Given the grand design of the IUE, it is no small achievement that the IUE has reached its current form given the current state of C++ compilers, and how they deal with templated code.

Lastly, as should be clear from what we have said above, our early experience gave us great fear as to the practicality of IUE. Its use of templates, it could be argued, pushed the G++ compiler over the edge back in 1995. The most devastating consequence was that of seeing a 15 second algorithm turn into a 22 minute algorithm. The good news has clearly been that in less than one year we have seen this problem largely disappear. Run time on this case is down to a manageable 90 seconds, and while everyone hopes for further improvements, the IUE is now operating in an acceptable range in terms of run time efficiency.

References

[BHR86] J. B. Burns, A. R. Hanson, and E. M. Riseman. Extracting straight lines. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI–8(4):425 – 456, July 1986.