Adaptive Sampling of Images Based on Delaunay Triangulation Report

Li Pengxu, Wang Lvdi, Hou Qiming

1. Background: described in proposal
2. User's guide in a nutshell:

   Compression:

   a) Enter an image file name
   b) Click "Load Image"
   c) Click "Initialize"
   d) Click "Double sampling" until satisfied with the quality
   e) Click "Save .voro"

   View results:

   a) Enter an image file name (without .voro extension)
   b) Click "Load .voro"
3. Sampling algorithm in pseudo code

```
PriorityQueue sample_queue;
DelaunayTriangulation triangulator;
Image image;
Integer w0; //w0 is taken to be 2000 in our implementation

//initial triangulation is 2 triangles forming the
image's bounding box
triangulator.initialize();
For i=0 to nSample do
    Triangle t;
    do
        t=sample_queue.extractMin();
        If t==null Then Exit;
    While not t.isInside(t.center())
    For j=0 to nSamplePerTriangle do
        triangulator.addVertex(t, t.center());
End

Procedure triangulator.onTriangleAdded(Triangle t)
    sample_queue.add(t,weight(t));
End

Procedure triangulator.onTriangleRemoved(Triangle t)
```

```
    sample_queue.remove(t);
End

Function weight(Triangle t)
    v=variance(
        image.color(t.v0),
        image.color(t.v1),
        image.color(t.v2));
    Return (v+w0)*t.area()
End
```

Degenerate case: New sample point lies on triangle edge/outside triangle

To make the color value on sample points well-defined, we round all sample point coordinates to integers. Thus, the rounded center of a triangle does not necessarily lie strictly inside it. Given a triangle to sample, we first try all 4 rounding conventions, i.e. round up/down for x/y. When all of them fail to produce a point lies strictly inside the triangle, we just discard it, and try the next best one. This degeneracy handling scheme makes the algorithm not guaranteed to sample all pixels before terminate. But as a compression scheme, sampling all pixels hardly make any sense, and our algorithm usually provides a satisfying result before it terminates.

In the Jan 5 version of our program, the on triangle edge case is not handled, resulting in incorrect behavior at large number of sample points.

Our implementation of Delaunay Triangulation in detail:

The specialty of our problem:

a) All points are integer points, and lies inside a predefined rectangle (the image). It's reasonable to assume the image's resolution is less than 32768x32768.
b) Points are added into the triangulation one by one, and we know which triangle it's added into before incremental triangulation.

So we omitted the point location maintenance part in the standard algorithm, and use exact toLeft and inCircle tests. Since coordinates fit in a 16-bit signed integer, toLeft may be exactly done in 32-bit signed integer (java int) and inCicle in 64-bit (java long).

Complexity analysis

In this section, n denotes number of sample points.

Our algorithm's complexity comes from:

a) Delaunay Triangulation

The triangulation is incremental triangulation without point location. The number of elementary operations down is proportional to number of edge changes during triangulation. Its complexity is input sensitive, since the points are inserted in a deterministic order with respect to the input image. In worst case, it's O(n^2), while practical input tends to yield random-like point patterns, result in O(n) behavior.

b) Sampling

Our adaptive sampling scheme involves maintaining a priority queue that contains all triangles in the current triangulation. Thus, for each edge change reported by the triangulator, O(1) insertion/deletions require to be done in the queue. Also, each time a new sample point is requested, an extractMin operation has to be performed. Number of total edge change, as analyzed above, is worst case O(n^2) and practical case O(n). We use a traditional binary heap for the priority queue, which has O(logn) insertion/deletion/extractMin. We get worst case O(n^2logn) and practical case O(nlogn) for this step.

Thus, total complexity is worst case O(n^2logn) and practical case O(nlogn).

Failed improvement schemes:

a) Multiple samples per triangle

At the point of inserting the first triangle, triangulation is changed. The remaining points would need point-location, or be maintained during incremental triangulation. Though manageable, this is not a trivial task. Also, after the triangulation change, those predetermined points may very likely become poor sample points.

Multiple initial samples on feature points would be very helpful, but we don't have time to implement it.

b) Random sampling

Random sampling has a tendency of forming irregular triangles (triangles with larger maximal angle). Despite the fact delaunay triangulation tries to minimize them, there're still a considerable number of them manage to stay. Such triangles make this kind of sampling much less efficient.

Due to lack of time, we don't have time to try many different distributions. It's possible some better distributed random sampling may outperform center sampling.

c) Weighting schemes that use internal information of triangles

The strength of this sampling method is that only color information on previous sample points are used to determine the next sample point's location. Thus, only a sequence of colors and a few initial sample points are required to reconstruct the original image. All sample point locations may be determined by an identical sampler with the color sequence as input. More powerful weighting schemes that use internal information, however, violate this rule, and require to have additional information to be stored to make decompression possible. That effectively halves the compression ratio. Though they do provide better quality with fewer number of sample points, they aren't likely to do as well at same amount of storage, which usually means half of less sample points.

4. An abstract documentation of the code:

Almost each file contains a considerable amount of dead code. They're mostly test code or zombie code of failed improvement attempt. For extensibility, we left them in the program. They may just be safely ignored while reading.

File	Description
Vertex.java	Data structure and basic methods for vertices
Triangle.java	Data structure (a special case of DCEL) of triangles and manipulation methods

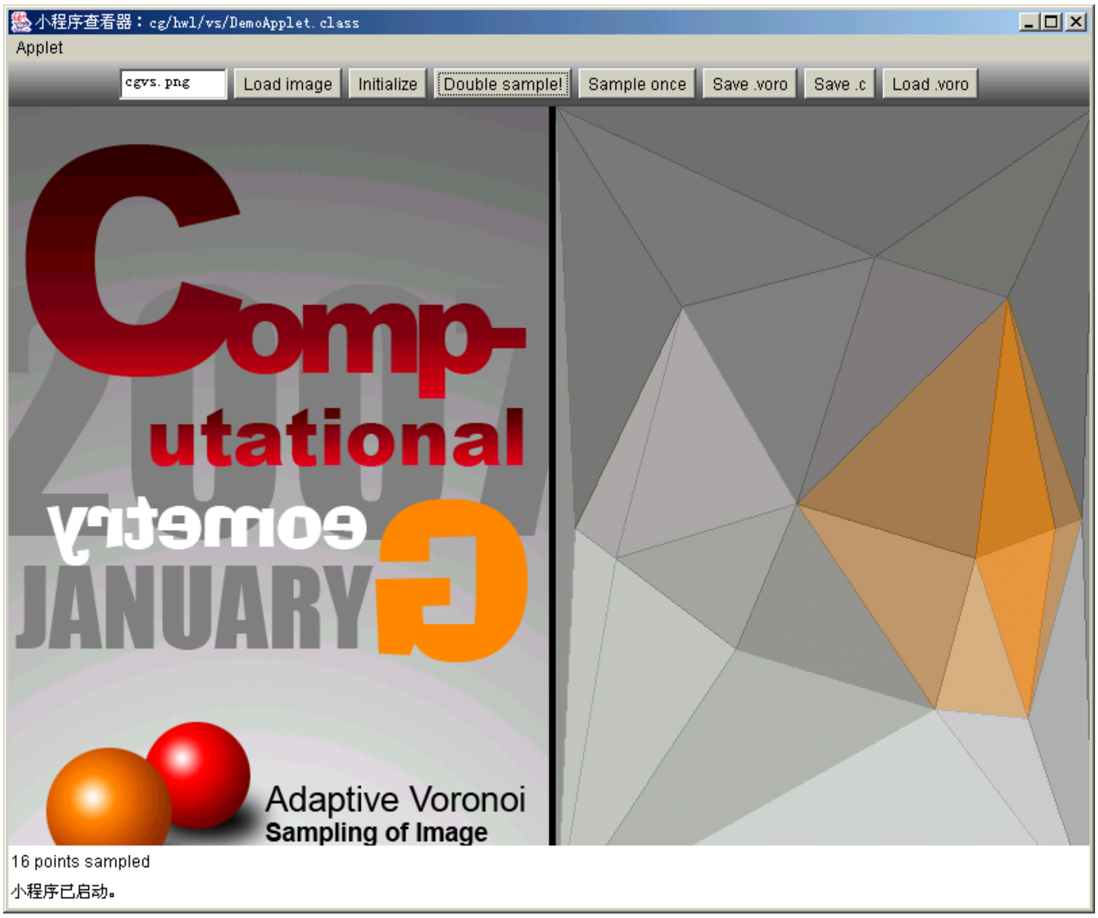
DelaunayTriangulation.java	The incremental Delaunay triangulator.
Sample.java	The sampling controller. It calls the triangulator, and supplies the triangulator with call backs to maintain the sampling priority queue.
DemoApplet.java	The user interface and I/O part.
image files	Test suites

5. Detailed user's guide

The batch files:

File	Description
t.bat	Compile and run the applet
r.bat	Run the applet
c.bat	Compile and run the saved .c file

The user interface:



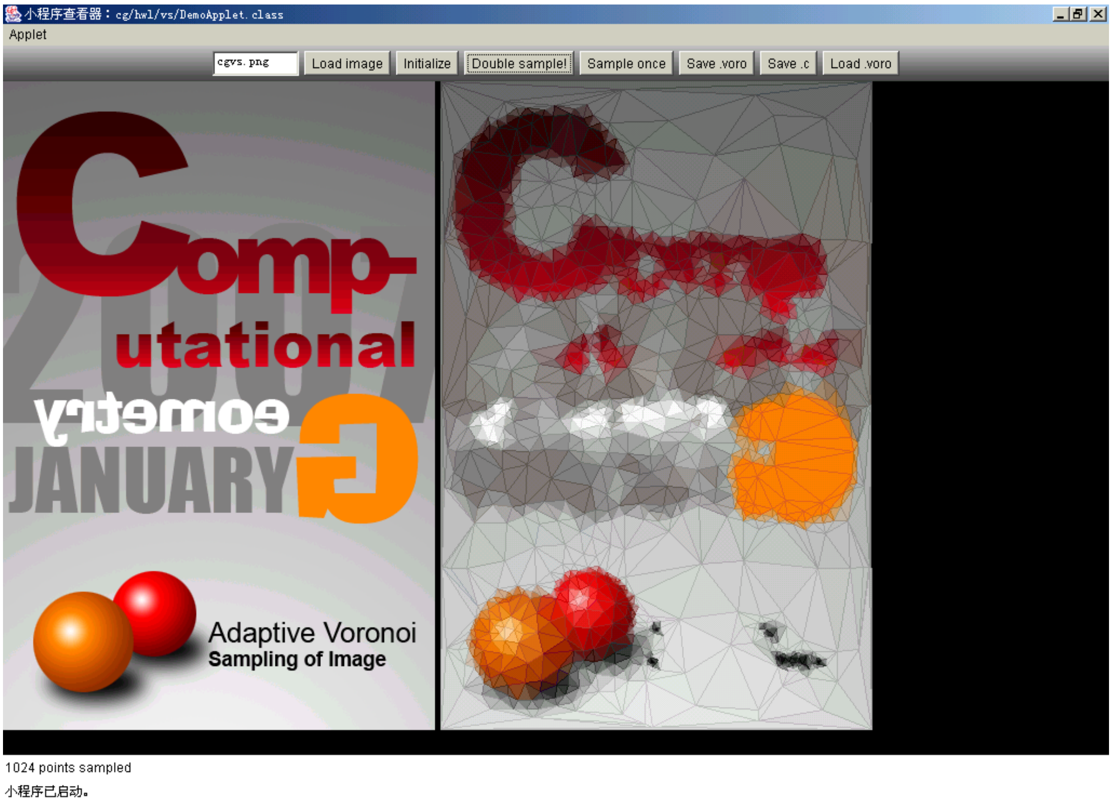
Button	Description
Load Image	Load the image file, file name should be entered in the text box
Initialize	Initialize the sampling
Double sample	Double the amount of sample points
Sample once	Add one sample point
Save .voro	Save sampled image to <image file>.voro
Save .c	Save sampled result to a .c file that may be compiled and run using c.bat. It views the sampling result with smooth shading.

	Requires OpenGL and recommends to compile with gcc.
Load .voro	Load and view previous saved sample result

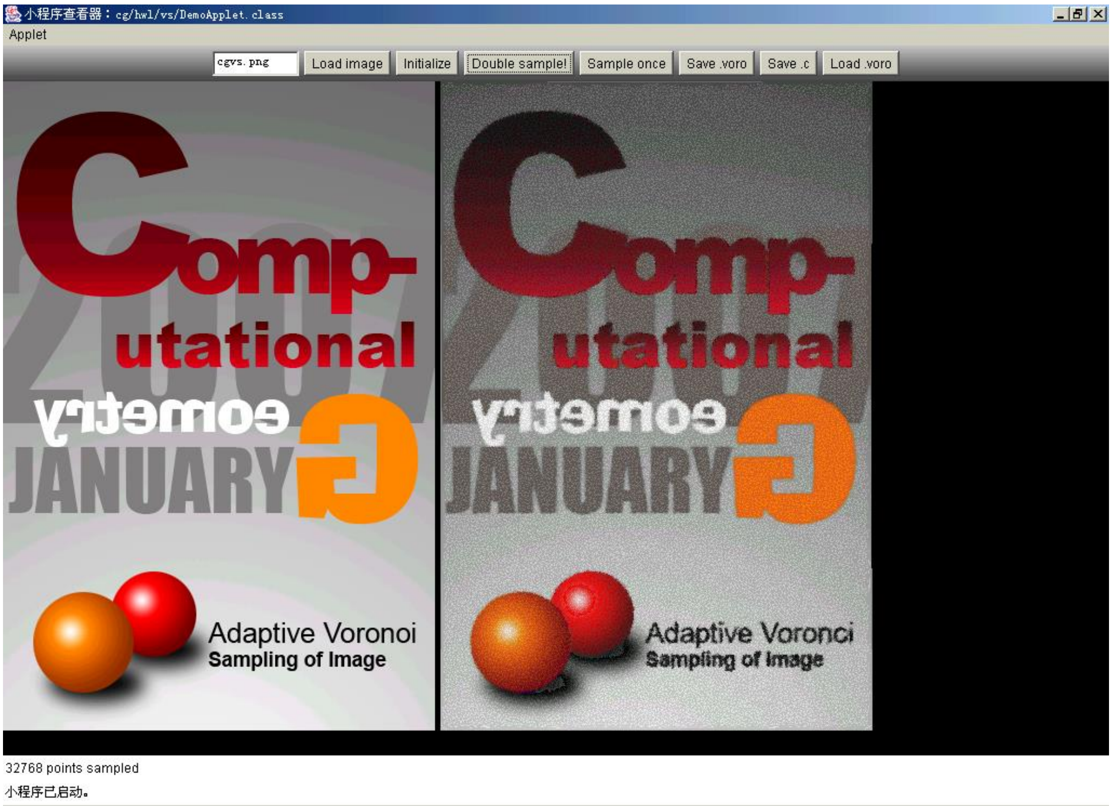
6. Results

cgvs.png, at 128 samples

cgvs.png, at 1024 samples



cgvs.png, 32768 samples:



girl0.png, 8192 samples

girl1.png, 32768 samples

7. Comparisons

All test suites are 24-bit png images. Note that jpg images are already compressed, during which considerable high frequency noise are introduced even at maximal quality. That makes them poor test suites for our algorithm.

Test suite	Resolution	Original png	.voro + rar
cgvs.png 32768 samples	400x600	139782 Bytes	60185 Bytes
girl0.png 8192 samples	488x475	228811 Bytes	16431 Bytes
girl1.png 32768 samples	439x576	305844 Bytes	65209 Bytes

8. Our collaboration

Li Pengxu: The delaunay triangulator

Wang Lvdi: GUI and overall architecture, proposal

Hou Qiming: Sampling, report