Note for seam sarving

October 22, 2015

- Voxel/3d image $\xrightarrow{\text{Marching Cubes}}$ Mesh

"voxel" = "**vo**lume" + "pi**xel**"; "pixel" = "**pi**cture" + "**el**ement".

- **Def. of an isosurface**

2D:

$$z = f(x, y), \quad \{(x, y) \mid f(x, y) = \tau\}$$

3D:

$$w = f(x, y, z), \quad \{(x, y, z) \mid f(x, y, z) = \tau\}$$

, where τ is some constant value (that is interested by the observer).

- `.raw` **format**

It's in an X by Y by Z layout. The spacial locations of the data points are implicit (`http://www.paraview.org/Wiki/ParaView/Data_formats`).

For example, if the binary file contains numbers (after being read in via C++ file stream) 1, 2, ... , 12, and it represents a 2 by 2 by 3 voxel (assume `WIDTH` by `HEIGHT` by `DEPTH`), it should be printed out like this:

```
the 0th y-x plane
1 2
3 4
the 1th y-x plane
5 6
7 8
the 2th y-x plane
9 10
11 12
```

To store the voxel in a 3d array in C/C++, the indexing format is expected to be:

```
I[z][y][x]
```

Operator in 3d

1 Energy function

The gradient of an input 3d image (voxel) $\mathbf{I}$ is

$$\mathbf{G_I} = \left(\frac{\partial}{\partial x}\mathbf{I},\ \frac{\partial}{\partial y}\mathbf{I},\ \frac{\partial}{\partial z}\mathbf{I} \right) \ \in\ \mathbb{R}^3$$

The energy function we use here is the $\mathbf{L_1}$**-norm of the gradient**:

$$e_1(\mathbf{I}) \ =\ ||\mathbf{G_I}|| \ =\ \left|\frac{\partial}{\partial x}\mathbf{I}\right| + \left|\frac{\partial}{\partial y}\mathbf{I}\right| + \left|\frac{\partial}{\partial z}\mathbf{I}\right|$$

it's a mapping $\mathbb{R}^3 \to \mathbb{R}$.

2 Definition of the 2d seam

Let $\mathbf{I}$ be a $w \times h \times d$ (`WIDTH` by `HEIGHT` by `DEPTH`, which correspond to x, y, z respectively) volume. To <u>shrink w (the x-dimension) by 1 pixel</u>, we define a <u>**yz-seam**</u> (or **x-seam**, in correspondence with the denotation) as:

$$\mathbf{s^x} \ =\ \left\{ s^x_{jk} \right\}^{h,\quad d}_{j=1,\ k=1} \ =\ \{ (\ x(j,k),\ j,\ k\ ) \}^{h,\quad d}_{j=1,\ k=1},$$

$$s.t.\ \ \forall j,k, \quad \begin{array}{l} |\ x(j,k) - x(j-1,k)\ | \leq 1 \ \text{ and} \\ |\ x(j,k) - x(j,k-1)\ | \leq 1 \end{array}$$

where $x(\cdot)$ is a $\mathbb{R}^2 \to \mathbb{R}$ mapping: $\{(1,1),\cdots,(h,d)\} \to [1,\cdots,w]$. This means for each yz-seam,

- it consists of $h \times d$ pixels;
- its projection onto xOy plane is an 8-connected region of pixels, expanding in y-direction (and vice versa for projection onto xOz plane)

3 Optimal seam: minimal total energy cost

We look for the optimal yz-seam $\mathbf{s^{x*}}$ that minimizes the total energy cost:

$$\mathbf{s^{x*}} \ =\ \min_{\mathbf{s}} \ \sum_{j=1}^{h}\sum_{k=1}^{d} e_1\left(\mathbf{I}(s^x_{jk})\right)$$

4 Dynamic programming: cumulative minimum energy $\mathbf{M(i,j,k)}$

For the yz-seam,

1

$$\begin{aligned} M(i,j,k) \;=\; e_1(i,j,k) \;+\; \min\{\; & M(i-1,\; j-1,k),\; M(i-1,\; j,k),\; M(i-1,\; j+1,k), \\ & M(i-1,\; j,k-1), \qquad\qquad M(i-1,\; j,k+1) \;\} \end{aligned}$$

or, # 2

$$\begin{aligned} M(i,j,k) \;=\; & e_1(i,j,k) \\ & + \min\{M(i-1,j-1,k),\; M(i,j-1,k),\; M(i+1,j-1,k)\} \\ & \quad M(i-1,j,k-1),\; M(i,j,k-1),\; M(i+1,j,k-1)\;\} \end{aligned}$$

or, # 3

$$\begin{aligned} M(i,j,k) \;=\; e_1(i,j,k) \;+\; \min\{\; & M(\mathbf{i-1},\; j-1,k) \;+\; M(\mathbf{i-1},\; j,k-1), \\ & M(\mathbf{i},\; j-1,k) \;+\; M(\mathbf{i},\; j,k-1), \\ & M(\mathbf{i+1},\; j-1,k) \;+\; M(\mathbf{i+1},\; j,k-1) \;\} \end{aligned}$$

i.e., to pick the minimal value among the 3 pairs of sums — each pair of sum represents a position in x-dimension, which intrinsically treats the y, z dimensions equally.

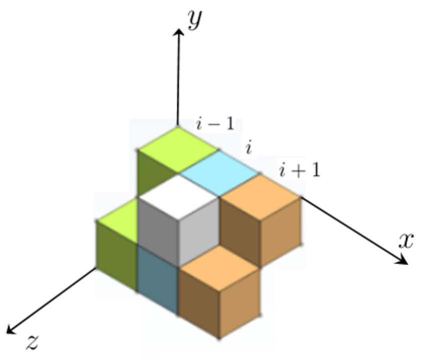


Figure 1: Computing M, where the white pixel stands for $M(i,j,k)$, or `M[z][y][x]`

Both computing M and back-tracking (to get the 2d seam) are done in a level-by-level manner. This also comes from the implicit expectation "when shrinking in x dimension, y and z are treated equally".

*** Illustration of how the two processes work (to be completed if this approach is approved.)**

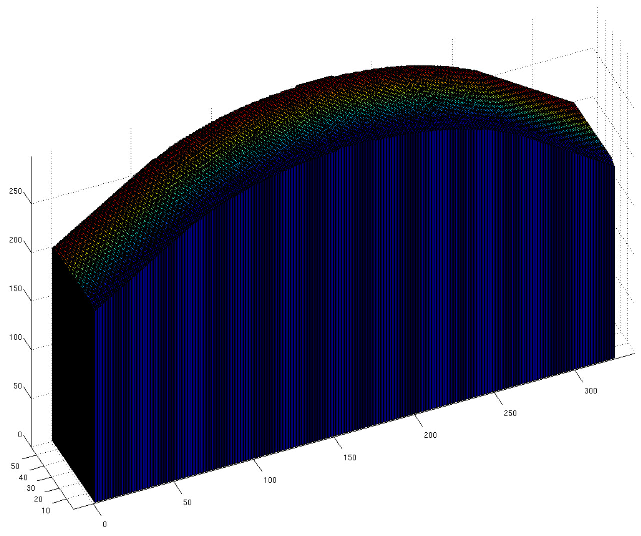


Figure 2: Indices matrix of a `DEPTH` × `HEIGHT` yz-seam.
The top surface depicts the x positions of the seam.

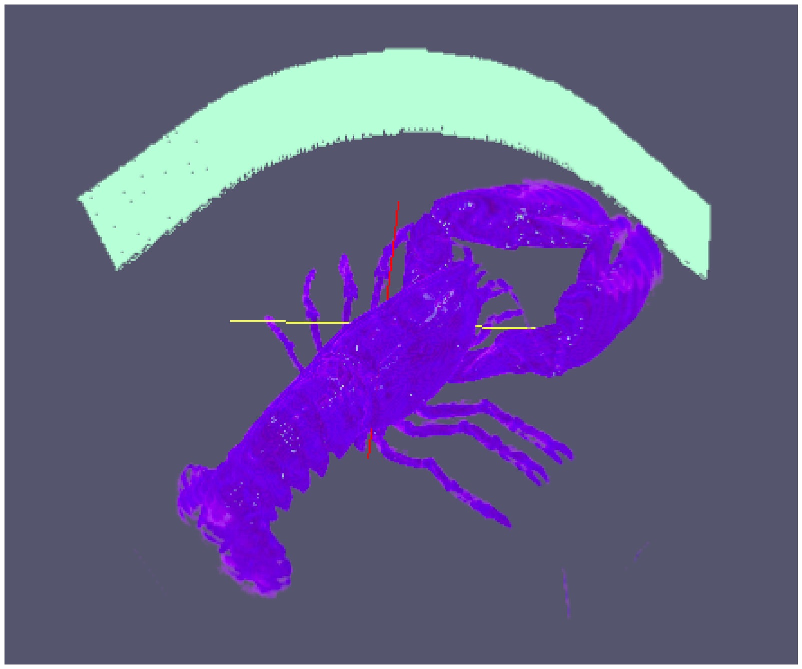


Figure 3: Test on the 301×324×56 `lobster.raw`.
Average energy of the 2d seam here is 0.0 ($<$ 5.998, that of the whole image), which means the program finds the least-varying (non-varying?) pixels in the voxel, during back-tracking for this 2d seam.

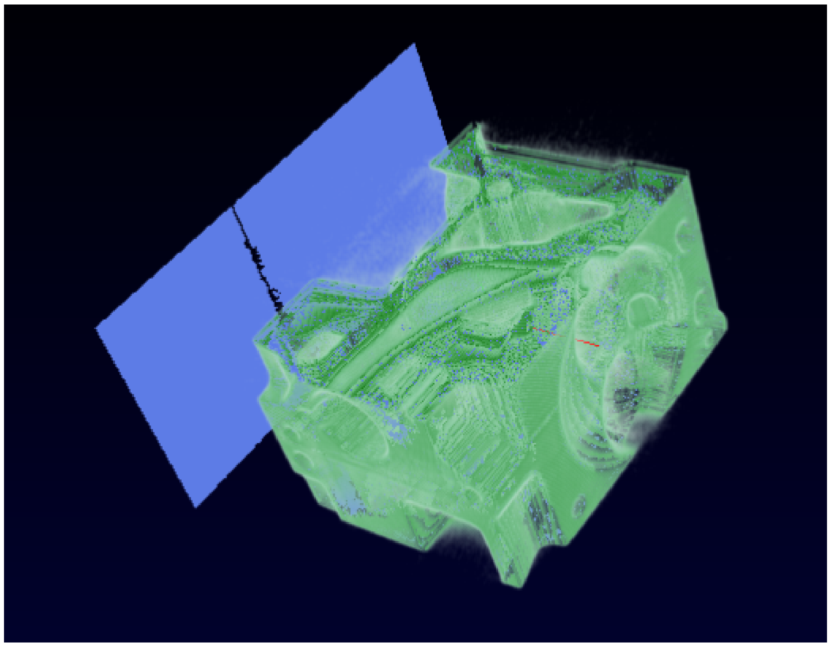


Figure 4: Test on the 256×256×128 `engine.raw`.
Average energy of 2d seam: **0.293472**;
Average energy of the image: 11.8278.

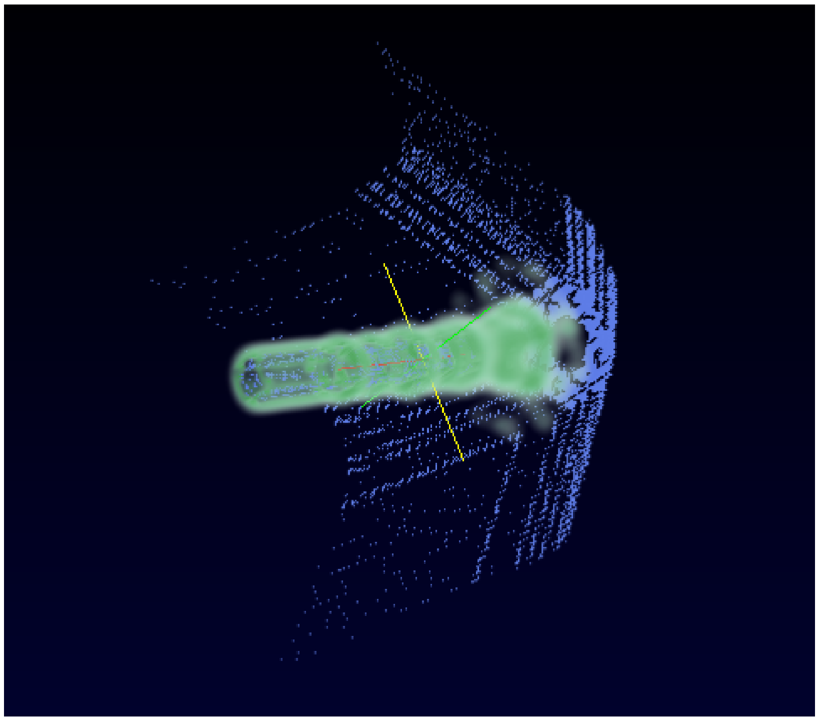


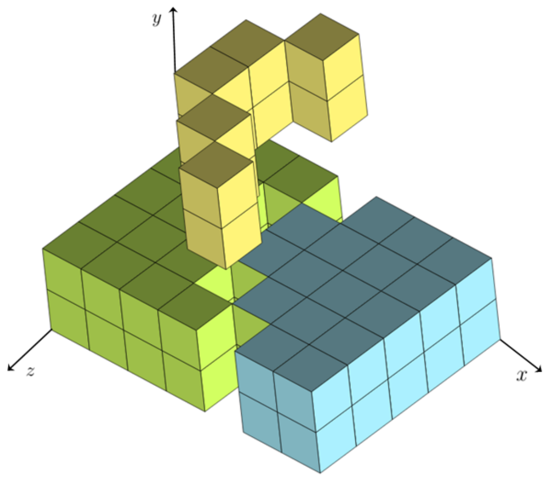
Figure 5: Test on the 64×64×64 `engine.raw`.
Average energy of 2d seam: **0.03662**;
Average energy of the image: 1.95015.

Carving a 2d seam

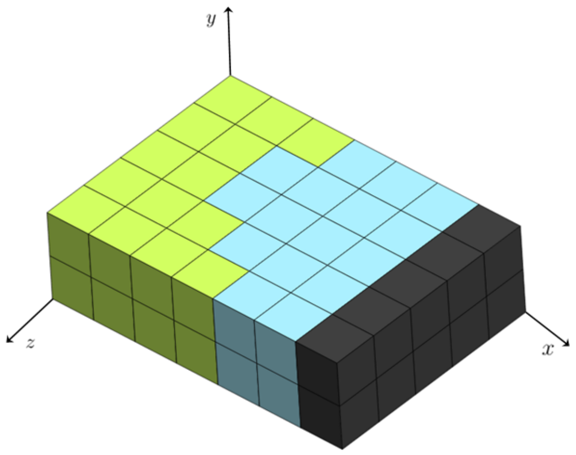
As for carving a yz-seam, we have

- the 3d image array I[DEPTH][HEIGHT][WIDTH];
- a 2d array yz_seam[DEPTH][HEIGHT] storing the x indices of the yz-seam.

We are going to process DEPTH × HEIGHT "rows" in x dimension — **for each row, we remove the element that is on the 2d seam, then "shift" all element(s) behind it (farther from yOz plane) by one pixel towards the yOz plane**.



Carve



Shift

Figure 6: Two steps (conceptual) to carve a yz-seam.

```
for (z=0; z<DEPTH; z++) {
   for (y=0; y<HEIGHT; y++) {

      int a = yz_seam_indices[z][y];

      //Shift element(s) I[z][y][a+1...WIDTH-1] to
      //I[z][y][a...WIDTH-2]


      //Set I[z][y][WIDTH-1] as '\0'

   }
}
```

Analysis: on making carving faster

There are three major arrays in this seam carving operator: `I`, `e1I` and `M`, all in dimensions `DEPTH` $\times$ `HEIGHT` $\times$ `WIDTH`. A test on execution time of each of the major steps in the pipeline is done on the $256 \times 256 \times 128$ '`engine.raw`':

Table 1: Execution time (in milliseconds) for each part of the program.

	Stuff to do	Time elapsed
1	3d arrays allocation	$\approx$ 10
2	Read raw image	$\approx$ 300
3	**Compute energy `e1I`**	$\approx$ 200
4	**Compute cumulative energy `M`**	$\approx$ 150
5	**Back track (to find 2d seam)**	$\leq$ 10
6	Write result to .raw	$\approx$ 100

Observation:

1. #3–5 are computation-involved, which (the functions) would be repeatedly called when carving multiple 2d seams.
2. #3 and 4 take much longer, as they're both "point-processing"-like and therefore take $O(n^3)$. **These two parts should be further optimized.**
3. Since "back track" does not take too much time (and considering the way it works), we can keep it as it is.

Experiment on carving multiple yz-seams

First we do an experiment on ‘`nucleon.raw`’ in a more detailed manner, considering its size (41) and shape (the same length in all three dimensions). In Table 2 we list the result of **execution time** and **avgerage energy** when different numbers of yz-seam(s) is/are carved from ‘`nucleon.raw`’.

Table 2: Experiment on ‘`nucleon.raw`’ $(41 \times 41 \times 41)$

1) Execution time is the average of 10 results recorded;
2) The approach is deterministic so “avg. energy” remains the same for each run.

# of yz-seam(s) carved	Execution time (ms)	Avg. energy of pixels
0	——	22.4744
1	**19.2366**	23.0413
2	**28.2732**	23.6146
4	**35.3656**	24.8127
6	**44.8707**	26.1060
8	**54.8796**	27.5076
10	**62.8108**	29.0130
15	**83.8391**	33.0852
20	**103.0176**	37.3794
25	**115.1808**	41.8958
30	**127.1187**	46.5929

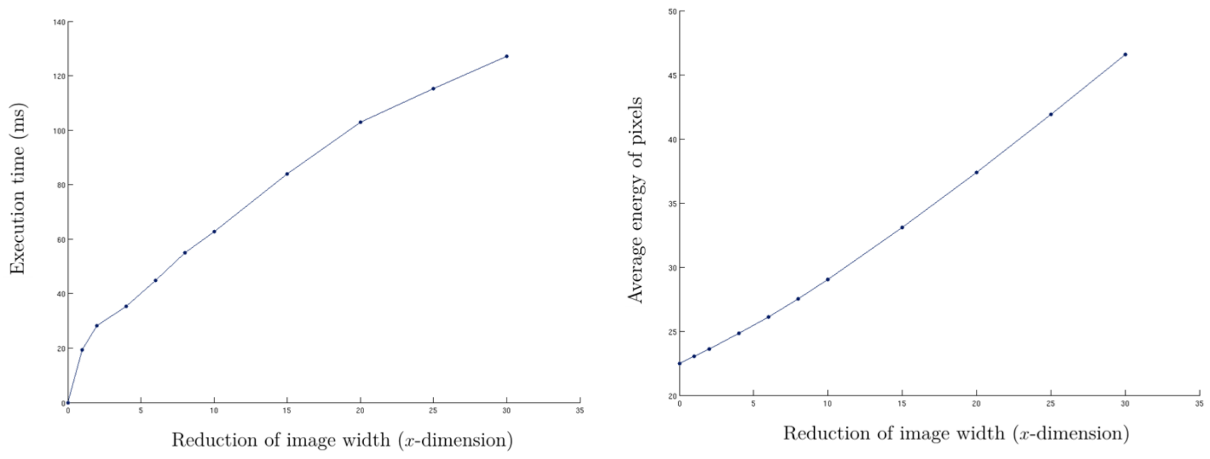
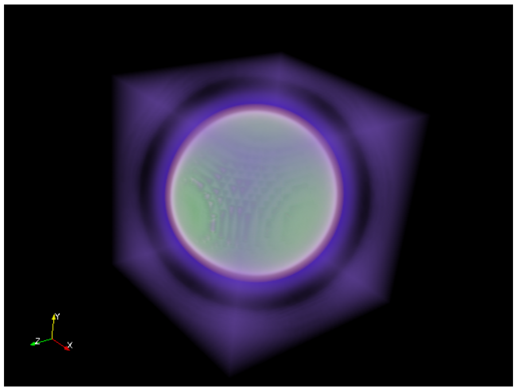
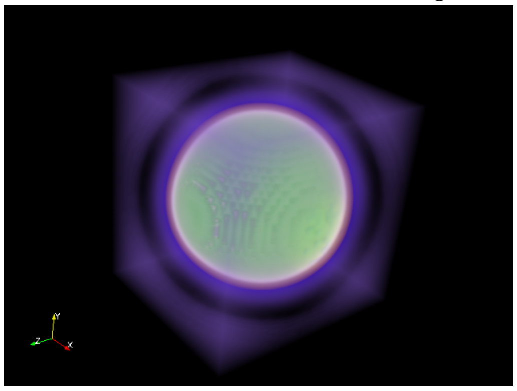
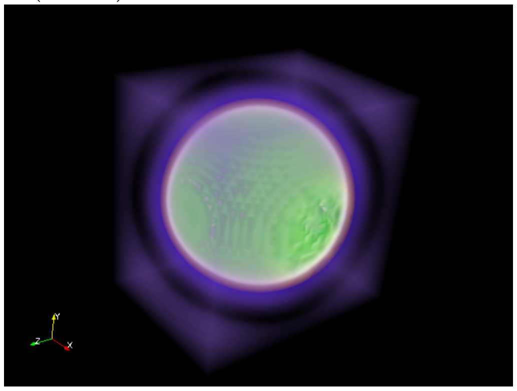
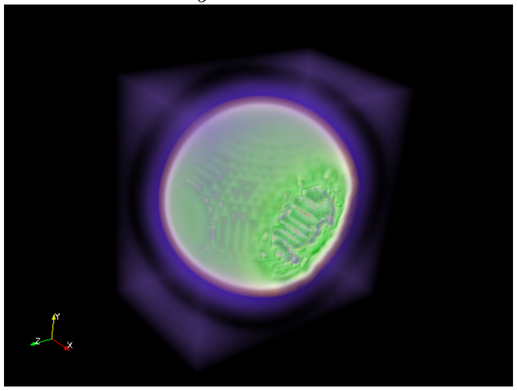


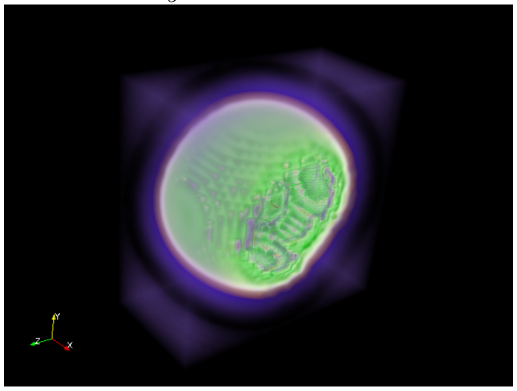
Figure 7: Execution time and average energy vs. number of yz-seam(s) carved.

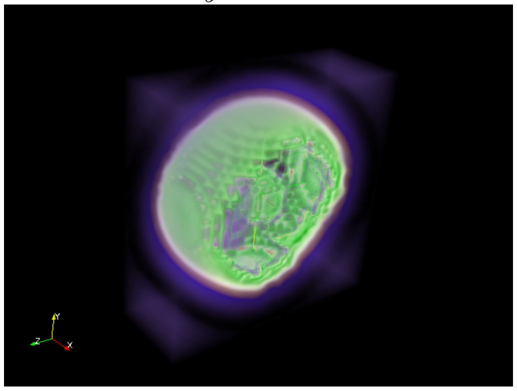

Original size (intact)


5 yz-seams carved


10 yz-seams carved


15 yz-seams carved


20 yz-seams carved


25 yz-seams carved

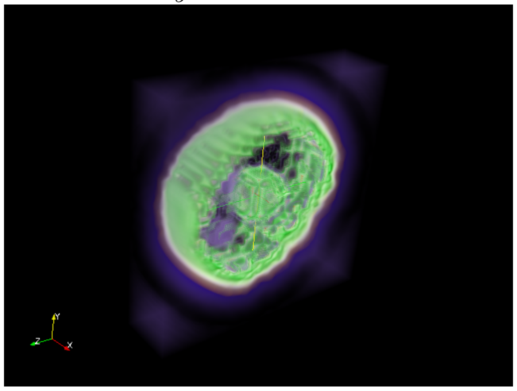

30 yz-seams carved

Figure 8: Paraview display of ‘`nucleon`’ object when multiple yz-seams have been carved.

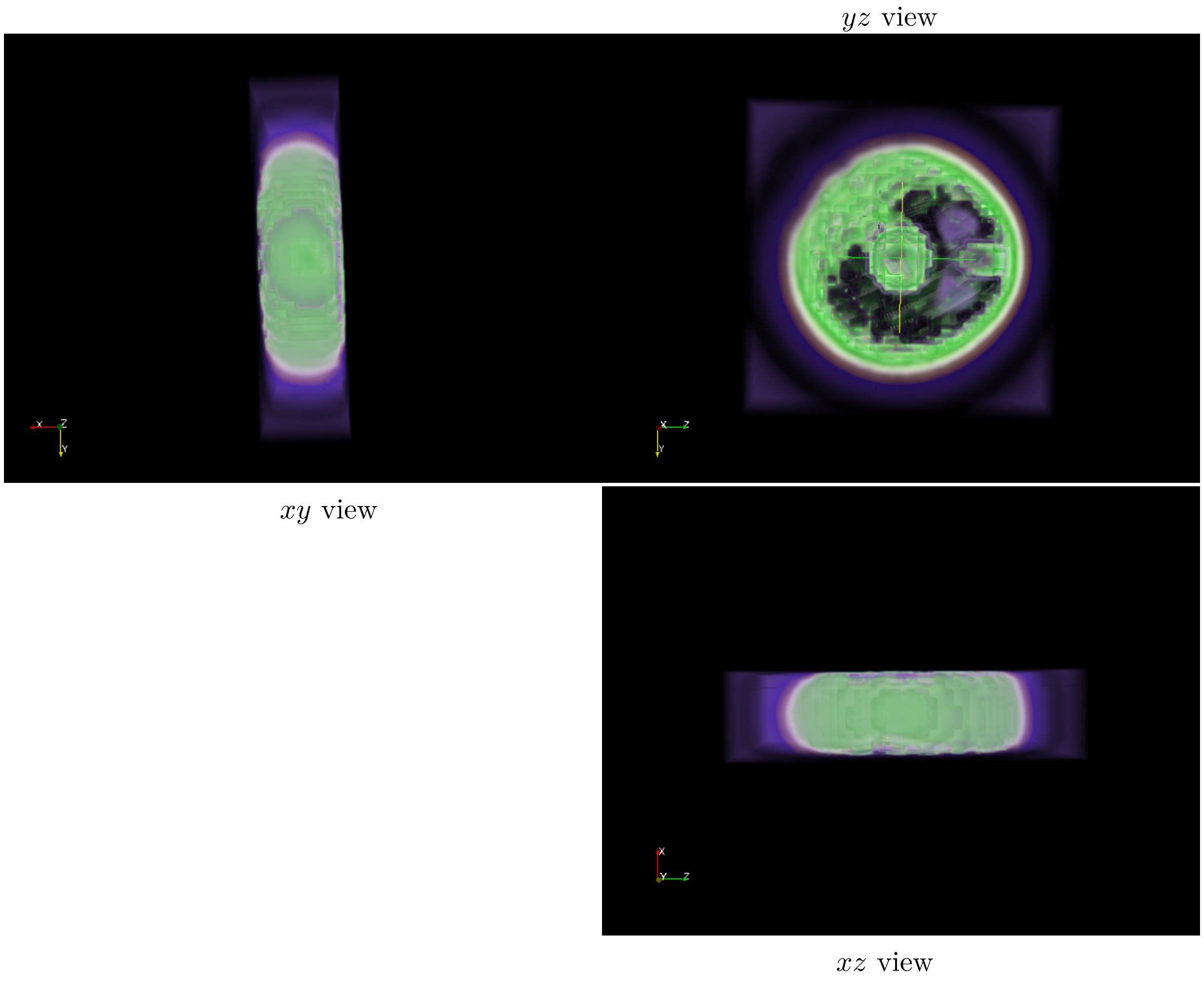


Figure 9: Three-view screenshots, under the scenario of 30 yz-seams get carved away.

Here's the result on '`lobster.raw`':

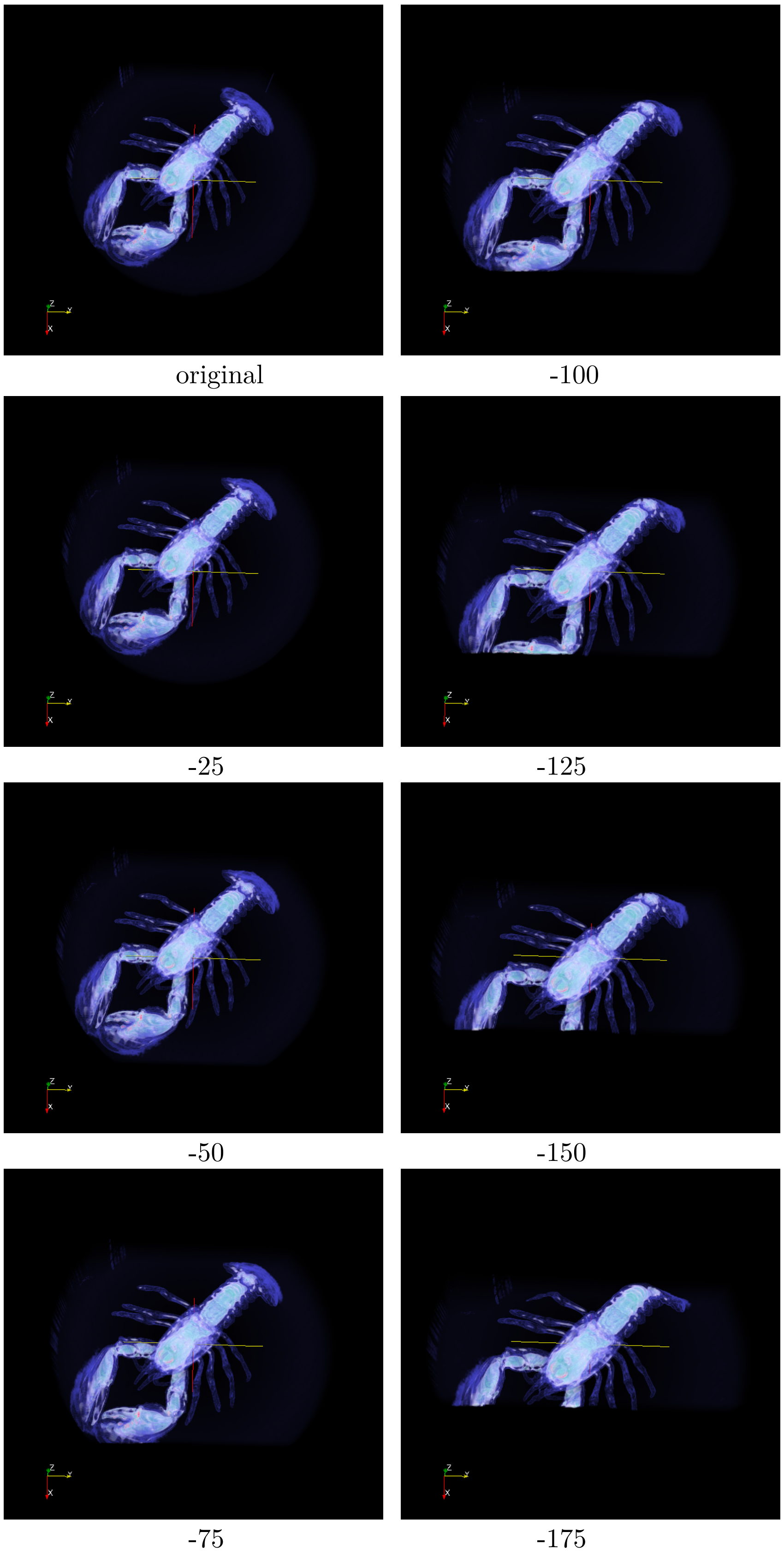


Figure 10: Paraview display of '`lobster`' object when multiple yz-seams have been carved.

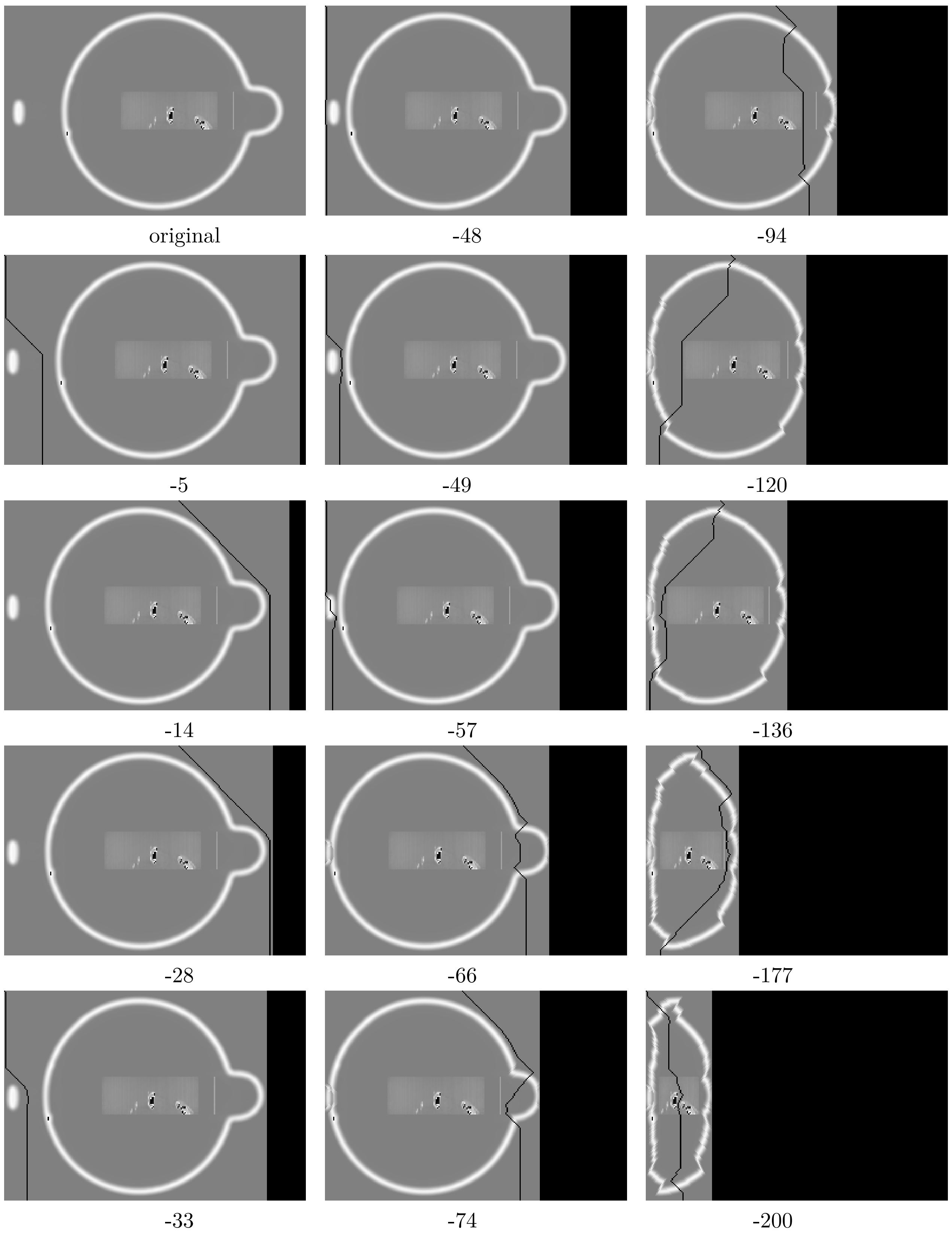


Figure 11: Experiment on a zx-slice (at y=128) of "BostonTeapot"

June 2 —

1. Different energy definitions, cumulative energy schemes

Here illustrates an experiment on

1) three types of energy function (L_1-norm/L_2-norm/L_∞-norm of gradient), and
2) four schemes to compute the cumulative energy matrix M.

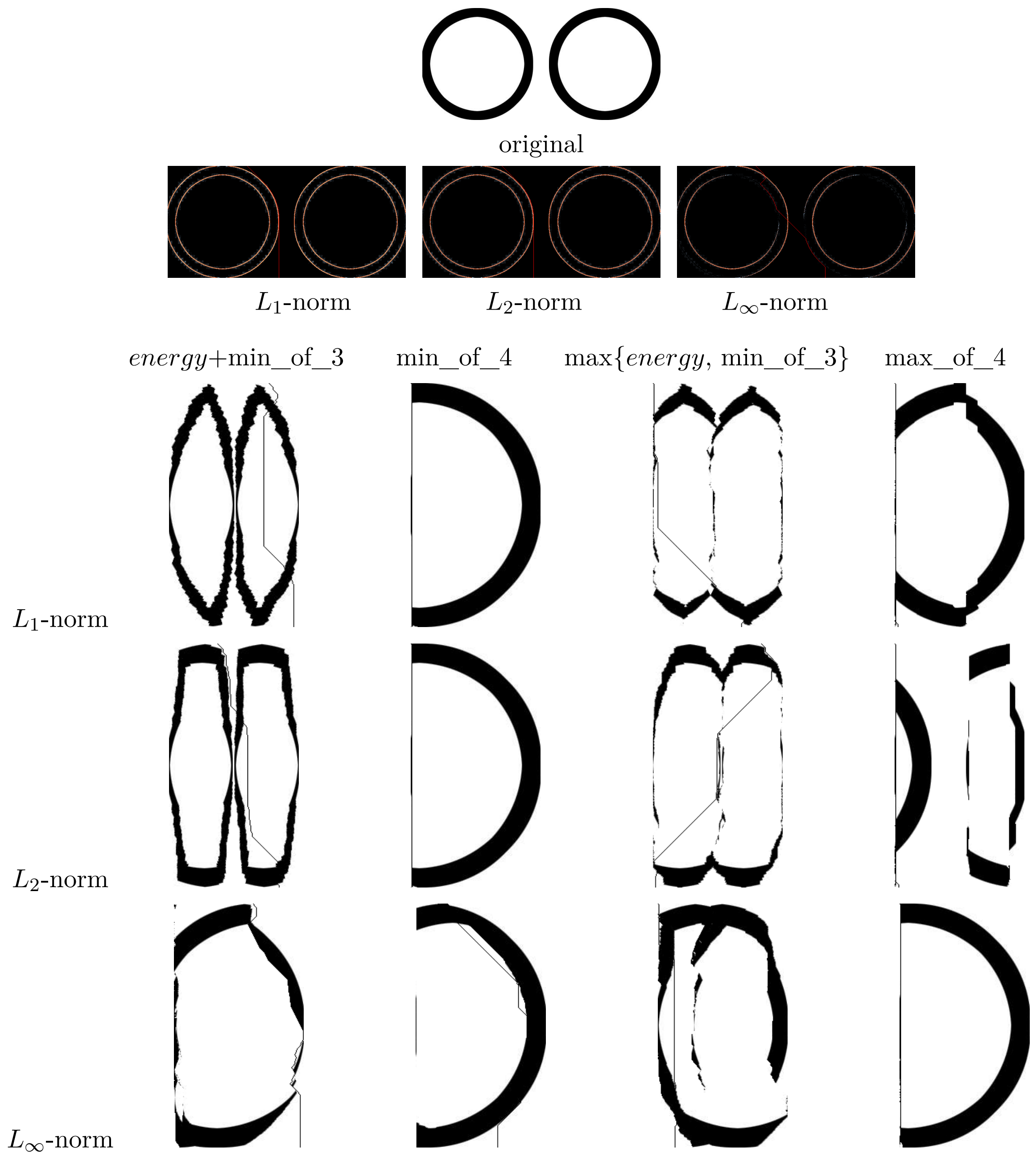


Figure 12: "Two-circle", 75% in width (horizontal) is seam-carved

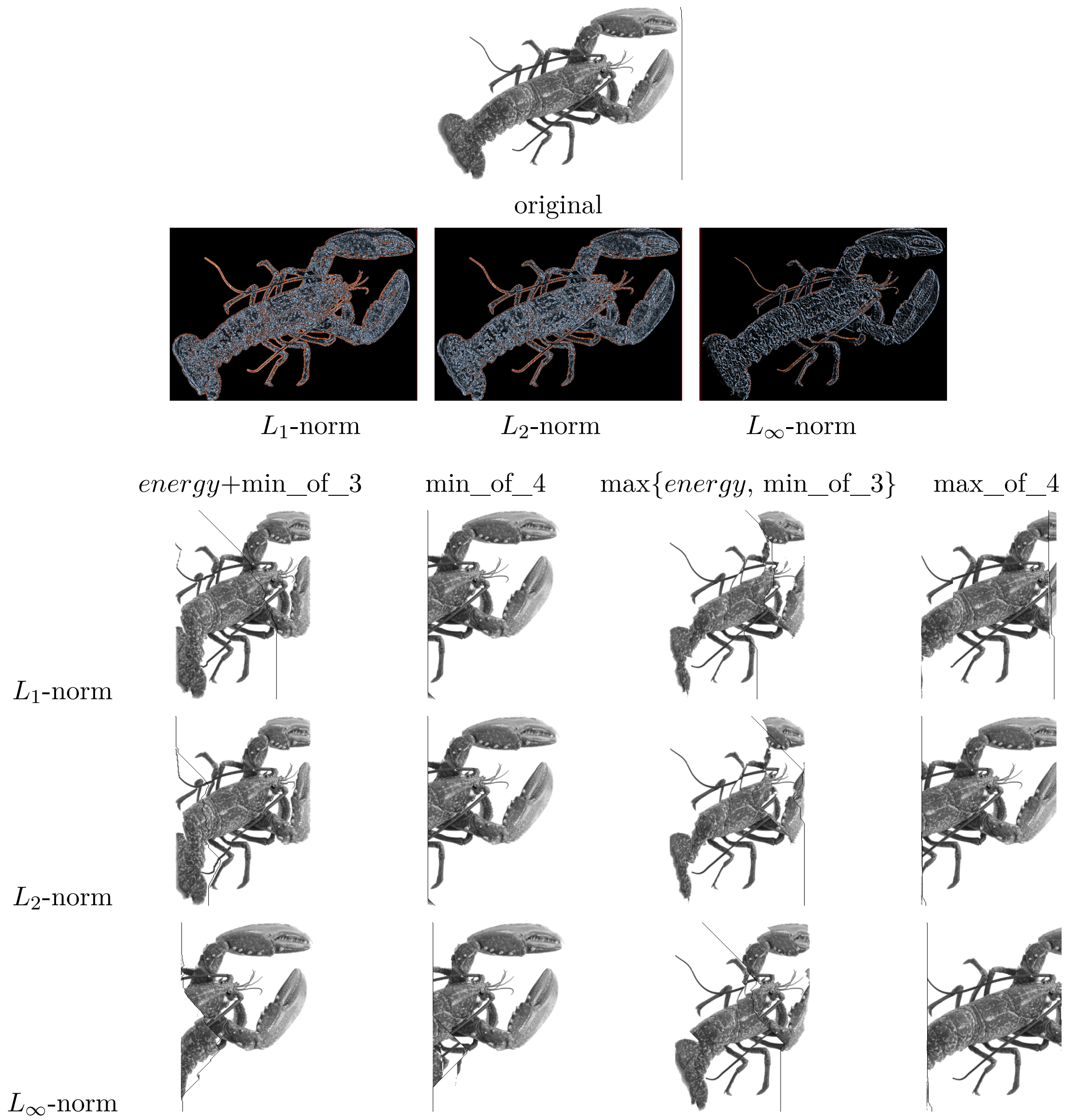


Figure 13: "Lobster", 50% in width (horizontal) is seam-carved

What's new in the 2d code:

- 'e'/'E' key to display color-mapped energy (and press again to switch back);
- 'r'/'R' key to reload.
- '1' to use L_1-norm of gradient (the default, when the program starts);
- '2' to use L_2-norm of gradient;
- '3' to use L_∞-norm of gradient. ('2'/'3' is probably followed by the 'r'/'R' key.)

2. Summary of related papers

2008

(1) *Fast Image/Video Upsampling* (ACM Transactions on Graphics) —— to upsample local regions, getting a "magnifying" effect. Here, seam carving is mentioned as one of the ways to do upsampling that do not use patch information.

☞ **Clue: perform upsampling/enlarging on the 3d object (or part of it).**

(2) ***Optimized Scale-and-Stretch for Image Resizing*** (ACM Transactions on Graphics) —— a (content-preserving) warping method which first partitions the original image into a grid mesh (many "quad"s), then deforms it to the target dimensions.

Gives better results than (improved) seam carving and is cited 90 times.

☞ **Clue: (1) partitioning/segmentation-based seam carving on 3d image;**
(2) "significant map" (gradient$\times$saliency), and "forward energy", to replace gradient.

2009

(3) *Optimized Image Resizing Using Seam Carving and Scaling* (ACM Transactions on Graphics) —— first do seam carving and then scaling, optimizing a well-defined image distance function. It protects important regions and visual effect, but runs slower (40-180s to resize a 500$\times$500 to half size).

☞ **Clue: could we have a definition of "object protection" for the 3d image, by measuring some "distance function" between I and T? This may include feature extraction that aligns two images to two features with the same dimension.**

(4) *CONTEXT SALIENCY BASED IMAGE SUMMARIZATION* (IEEE International Conference on Multimedia and Expo) —— based on "content saliency" and a grid representation, to warp the original image to a smaller "summarization". Gives better result on some examples than seam carving, but the work seems to be on outside the object (i.e., removing the surrounding background).

☞ **Clue: the usage of "saliency".**

2010

(5) *Scene Carving: Scene Consistent Image Retargeting* (ECCV) —— generalize seam carving by using a user-provided **relative depth map**, to preserve **scene consistency**. It first splits the image (the scene) into multiple layers.

☞ **Clue: maybe to create a depth map 3d matrix at each pixel of the 3d image, as an alalogy, as an extra feature added to seam carving process?**

2011

(6) *A Distortion-Sensitive Seam Carving Algorithm for Content-Aware Image Resizing* (Journal of Signal Processing Systems) —— use local gradient information along with a thresholding technique to guide the seam selection; a mechanism to halt seam carving when further processing would introduce unacceptable visual distortion in the resized image; anti-aliasing filter to reduce the artifacts cauesed by seam removal.

☞ **Clue: thresholding the gradient energy; taking an average of two adjacent "columns" when carving a seam.**

(7) *Scale and Object Aware Image Retargeting for Thumbnail Browsing* (IEEE International Conference on Computer Vision) —— **Cyclic Seam Carving (CSC)**; to augment the energy function with the proposed scale and object aware saliency.

2012

(8) *Live Image Composing* (SIGGRAPH Asia) —— to compose two images together, by first finding a boundary using seam carving and then using Poission image editing to obtain seamless composed result.

☞ **Clue: application related to object composing/connecting in 3d images.**

2013

(9) *Depth-Aware Image Seam Carving* (IEEE Transactions on Cybernetics) —— take advantage of modern depth camera to improve seam carving, by cutting the **near objects less seams** while removing **distant objects more seams**.

☞ **Clue: introduction of the depth map.**

2014

(10) *Saliency-Based Parameter Tuning for Tone Mapping* (European Conference on Visual Media Production) —— a parameter-tuning algorithm to **minimize the saliency distortion** caused by tone mapping.

Table 3: Some differences between 2d and 3d images

Observed feature	2d	3d
Scene condition	background and foreground	centered object surrounded by nothing but air
# of objects	> 1	$= 1$
$\frac{\texttt{Foreground}}{\texttt{Background}}$ ratio	low (in general)	very high
Color information	RGB/RGBA	intensity-only

❒ Intuitive/straight goal: to make the seam-carved 3d image the most **visually similar** to the original one.

$\Downarrow$

to minimize the "difference"/content distortion caused by carving each 2d seam.

$\Downarrow$

So if we need to remove some pixels (which form the 2d seam) away, they should probably be: (1) less-important ones, and (2) not destroying the geometric shape/properties of the image.

(By June 26th...)

1. Code description and instructions

In the updated code, functionalities are integrated into `main.cpp`:

1. **Two seam-generation approaches** (computation of `M` and "backtracking" — '**frommid**' (the default option; start with the first 1d seam on some middle xz-slice) and '**volume**'.

   ☞ Usage: **'v'/'V'** to switch between the two (compute the next optimal seam but do not carve).

2. **Energy functions** (switch by pressing '1'–'6'; compute the next optimal seam but do not carve).

   ☞ Usage:

   '1' Gradient (L_1-norm), the default option;
   '2' Extended gradient (3x3x3 neighbors, added four "diagonal" terms);
   '3' Laplacian (3x3x3 neighbors);
   '4' LoG (5x5x5 neighbors, simply approximated by a spatial mask);
   '5' Line detector (3x3x3 neighbors);
   '6' To be completed (could be making use of the histogram of the 3d image).

3. **"Bilinear" (interpolation) scaling** (for comparison)

   ☞ Usage: **'7'** to do scaling from `WIDTH` to `WIDTH − 1`, so includes the carving action.

   ☞ Usage: **'c'/'C'** carves to 75% of the current width, by bilinear scaling.

4. **Rendering modes**

   ☞ Usage:

   '8' one xz-slice (without color mapping); use **'UP'/'DOWN'** to navigate to different "layers" (y values);
   '9' look from $y+$ and render all xz-slices (with color+α mappings);
   '0' volume rendering mode (with color+α mappings), the default option.

5. **Carve the current yz-seam**

   ☞ Usage: **'Backspace'** to carve.

6. **Hide/partially display/fully display the current seam**

   ☞ Usage: **'s'/'S'** to switch among the three modes (in order).

7. **Zoom-in/out**

   ☞ Usage: **'i'/'I'** and **'o'/'O'**

8. **Reload the original 3d image**

   ☞ Usage: **'r'/'R'** (the original `.raw` image is fisrt loaded into '`I_original`' and remains in the memory until the end).

9. **Snapshot the GLUT frame buffer**

   ☞ Usage: **'w'/'W'** to take a snapshot and save it to `.png` file.

10. ***Rotate the scene**

    ☞ Usage: click+drag the **mouse**

2. Experiment

With the selections of both seam-generation approaches and energy function being discussed, we still have some other factors that might improve/damage the quality of retargetting:

- do/do not use, in computing the energy `E`, **terms correlated with the y-direction** (suppose using ‘frommid’ mode, carving x-direction and implicitly treating each xz-slice separately).
- loose/tighten the constraint when generating the 2d seam.
- do/do not introduce **randomness** when selecting the starting 1d seam (‘frommid’) and x-position (‘volume’).

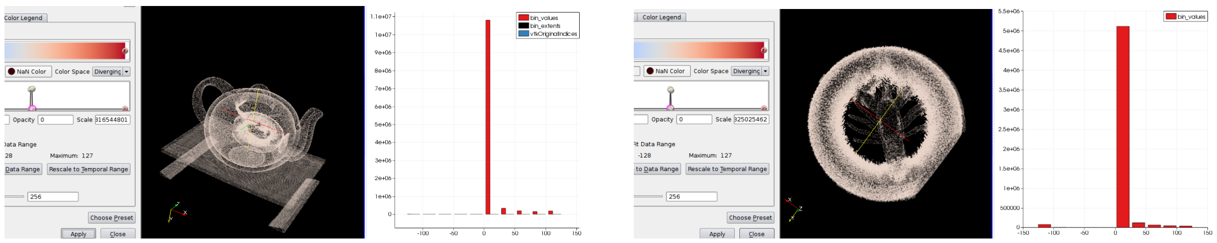


Figure 14: Histograms of the original `.raw` 3d images.

Note on June 26:

(0) Continue completing the experiment (4 groups);

(1) Check out “Transfer Function Design” (and volume rendering);

(2) Wrap up the math of the idea (summarized in the note);

(3) 3d texturing using OpenGL;

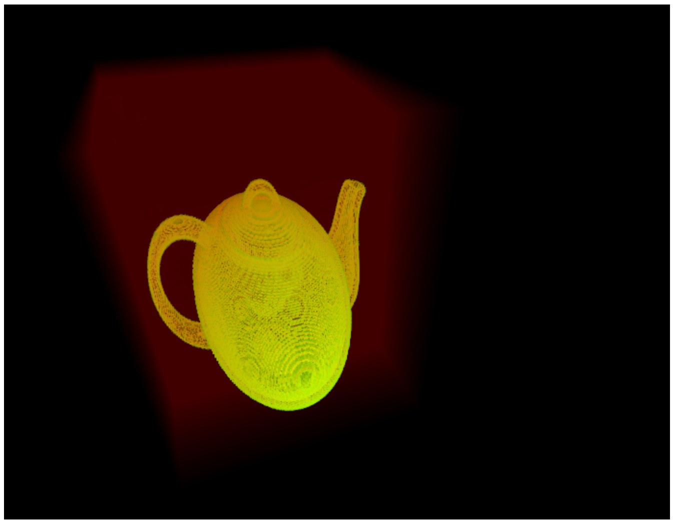


Figure 15: Comparison: simple scaling to 75% of width.

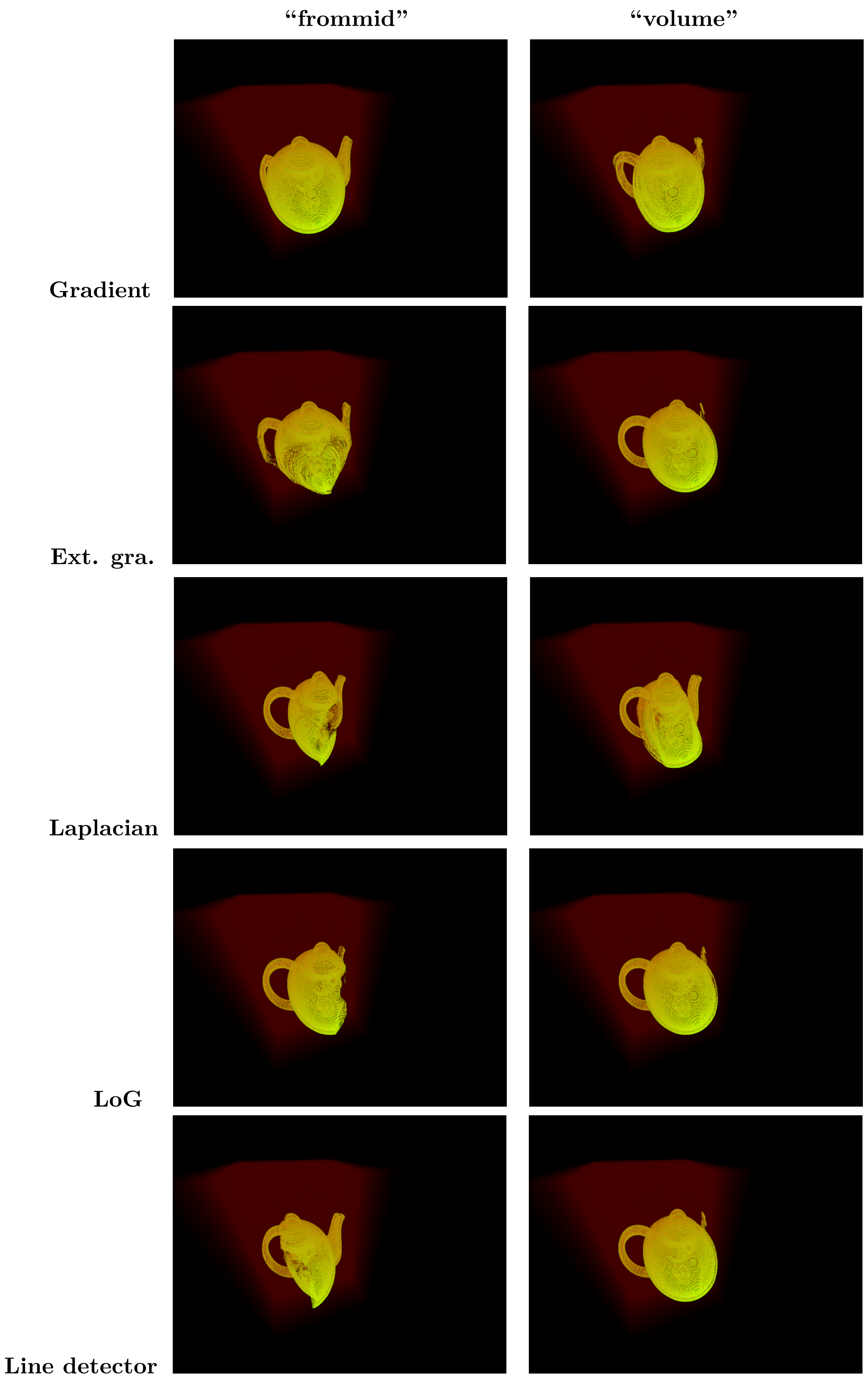


Figure 16: **Group #1**: with**out** y-related terms, **no randomness** (choose exactly the top 1)

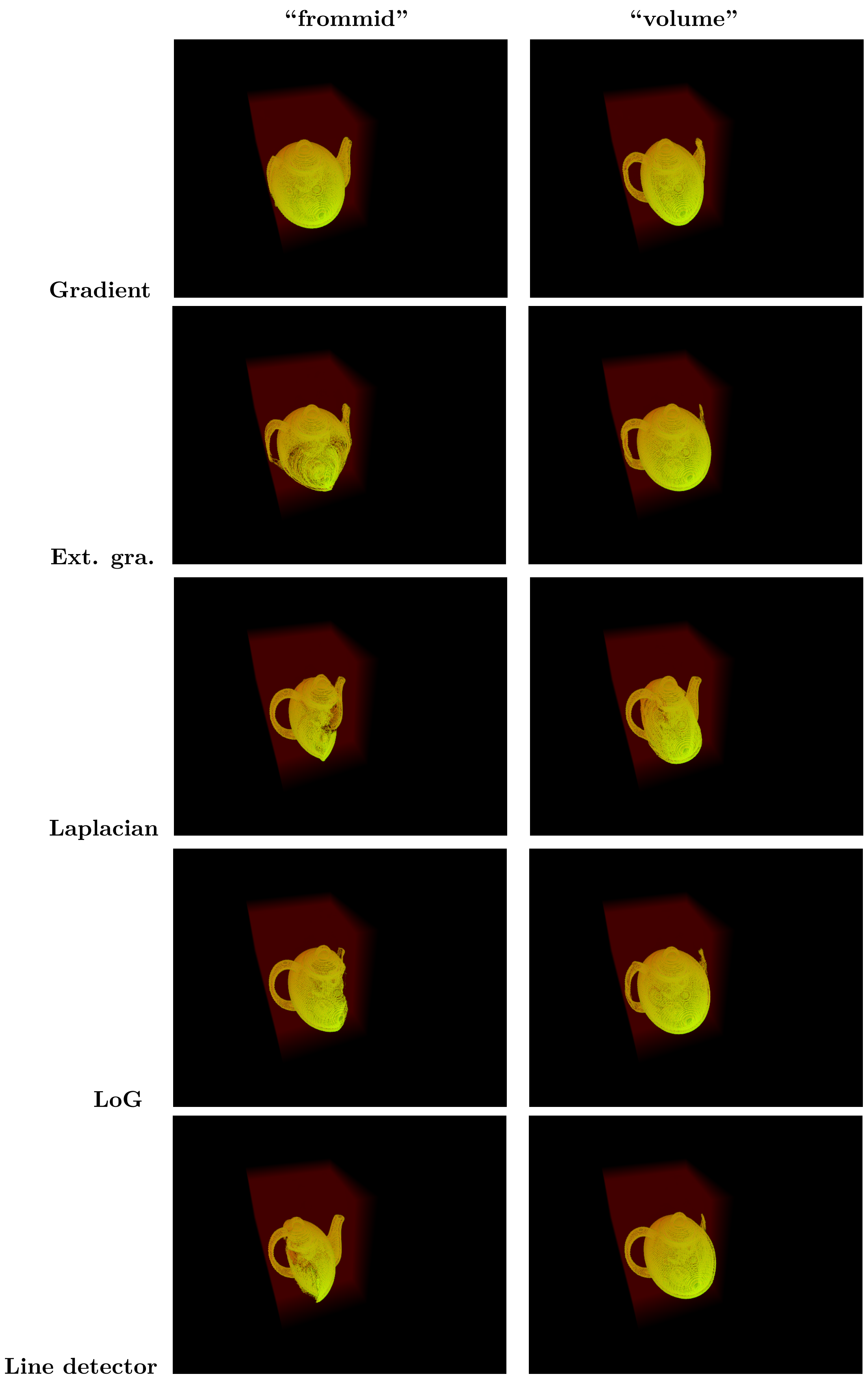


Figure 17: **Group #2**: with**out** y-related terms, **use randomness** (choose one from the top 15%)

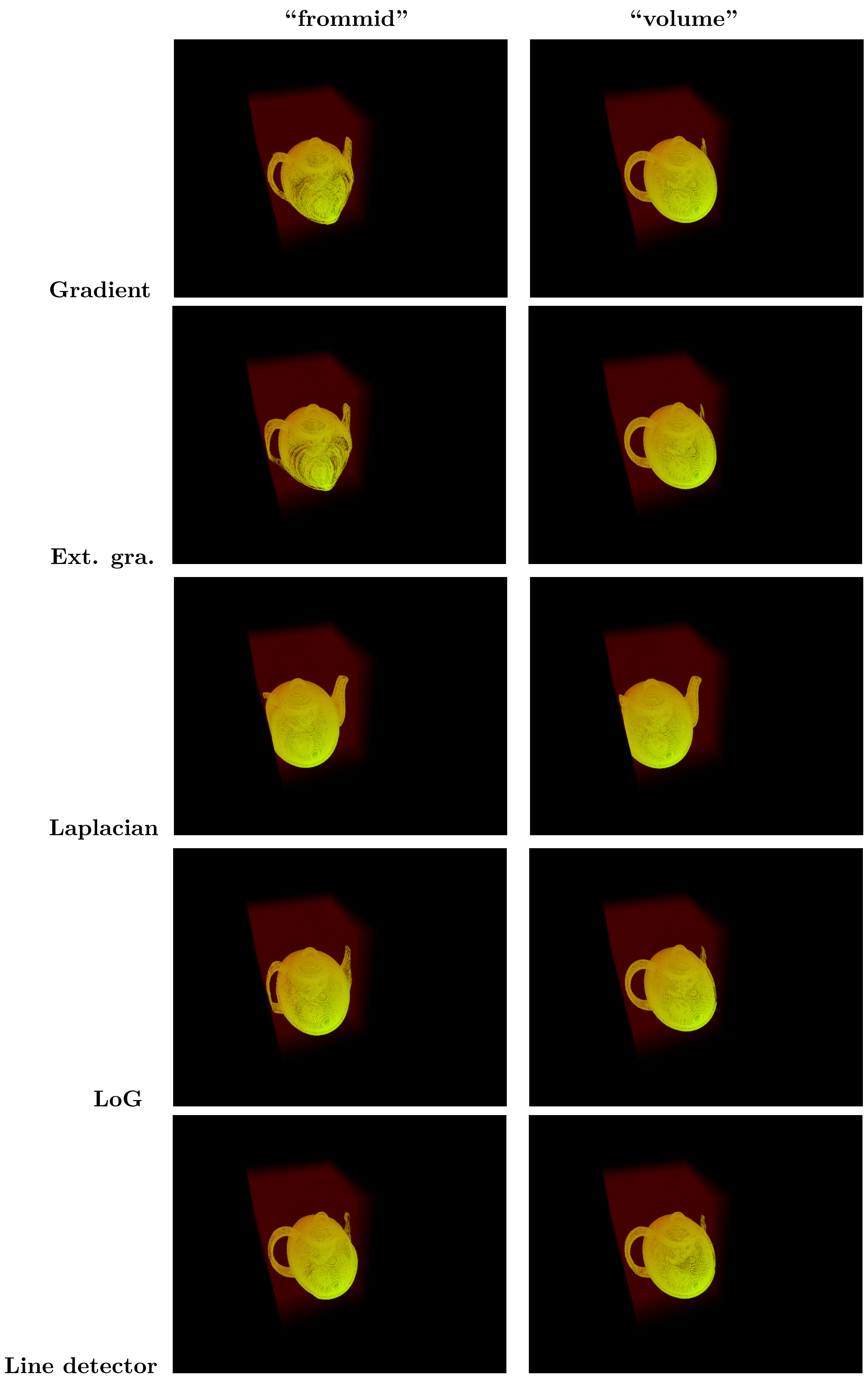


Figure 18: **Group #3**: **with** y-related terms, **no randomness**

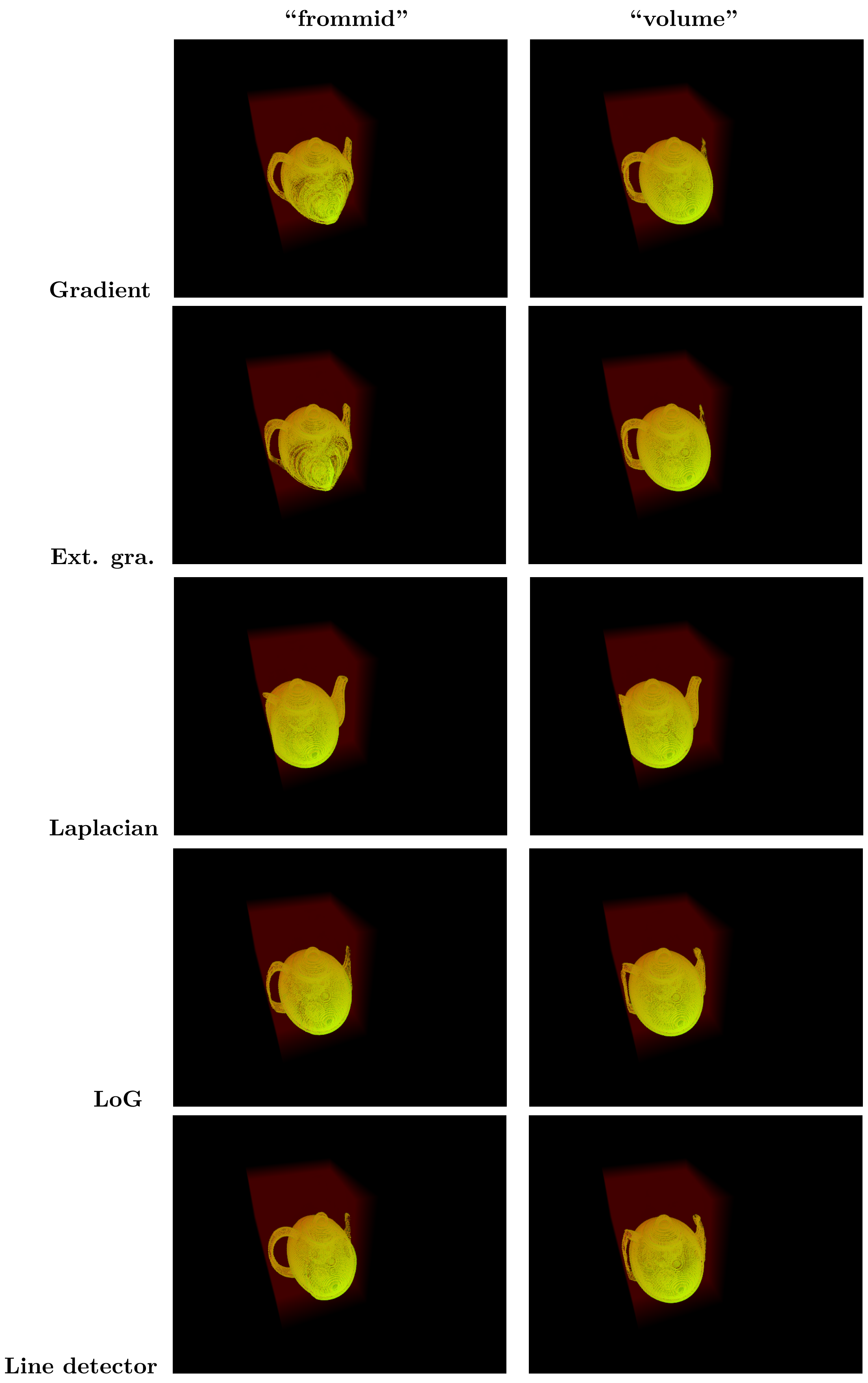


Figure 19: **Group #4**: **with** y-related terms, **use randomness**

Table 4: Options setting for each group of the experiment

Group \ Option	y-related energy term(s)	Randomness
1	✗	✗
2	✗	✓
3	✓	✗
4	✓	✓

Observation ("raw" conclusion):

First, note that 1) every observation here is partially subjective (already knew details of these four groups) and completely aesthetics-based (not quantitatively). 2) each figure gives the result of carving 25% of the width concecutively with the corresponding setting, while in the program it's okay to change the setting along the way of carving.

1. Overall, randomness does benefit the effect (#2 beats #1 and #4 beats #3), so **it's worth introducing randomness in the algorithm, at least in a certain degree**.

2. The simplest "gradient" (the first energy option) seems, and had been observed, to perform well. However, when $|\frac{\partial}{\partial y}I|$ is back to the equation (groups #3 and #4), gradient leads the algorithm to destroying the teapot's body. Meanwhile, since **it's unpredictable that how the object is placed when the 3d image was taken**, the algorithm should not be asked to recognize which one(s) of the x, y and z terms of the gradient is/are to be neglected. This means any energy function should be, and is best to be, in its original and complete form (as in groups #3 and #4), with no "prejudice" on any of the three dimensions. Therefore, from such perspective **"gradient" is not an ideal choice**.

3. From observing groups #3 and #4, **the best two options of the energy function are "LoG" and "Line detector"**, which give comparable quality and are more robust to the number of seams carved. **Any of the two, or some combination of them, can be considered as the default energy function**.
   ——LoG adds "Gaussian smoothing" to the pure Laplacian filter, and here we may also pre-filter the image using Gaussian before carving with (such as) the "Line detector" energy function.
   (should do such smoothing every time, so not changing the image data — only aims at improving the carving. will be tested soon)

- Last but important, appears that **"volume" mode performs well**, or better than "from-mid" — if we look at the last four cases in group #4. So **there's still a chance to propose a seam-generation approach like this** (but needed to be improoved), which may look general and intuitive from the theoratical perspective.

On 3d texture-mapping (July 7):

In OpenGL, textures are used to map color onto geometry. **The only difference between a 2d texture and a 3d texture is <u>the way in which you address the color</u>.**

2d texture ⇔ 2d texcoord, 3d texture ⇔ 3d texcoord.

(so you could, for example, specify a 3d texture and then draw a single quad that has some part of that texture mapped onto it.)

The teximage that you create is just an image that's stored efficiently in memory, and interpolated efficiently in memory as well.

- **What volume rendering does** is it tries to **think of the 3d image as some gaseous volume**, so parts can be seen through, other parts are opaque. It's a **physical model that we map onto the data**, by using two pieces:

(1) a transfer function that maps data values to color (RGBA) values, and

(2) a light model that tells us how a ray of light passes through the data values and accumulates color. In OpenGL, the standard way to do (2) is a `glBlendFunc`.

But, OpenGL still thinks about the world as drawing "geometry", such as the triangle. What folks do when they're making an opengl volume renderer is to use some kind of **proxy geometry**. For example, a stack of quads (or polygons; it need not be quads). They then use the teximage to rapidly index and interpolate an image of RGBA. What people often do is use **a set of polygons that line up with the ray cast from the viewpoint into the volume**.

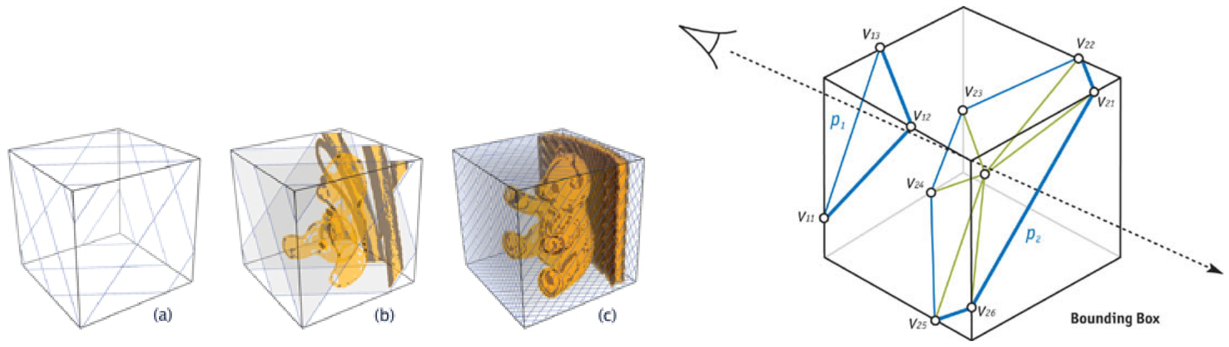


Figure 20: "View-Aligned Slicing"

The number of slices is a variable that affects the quality of the approximation, **but if you compute those slices quickly, it's nicer than using axis-aligned slices because it gives the effect of looking into the volume**. If you just used such as xy-planes you'd see gaps as you rotate. While with a 3d texture, you can specify 3d coordinates for the view-aligned slices and then get a visualization that has no gaps (and interpolates the data between slices for you).

Namely, each viewpoint should correspond to a specific stack of polygons, or "view-aligned slices" — you might want to **cache them a bit** or **delay recomputation**.

In short,

- the easy part is setting up the texture;
- the hard part is setting up the geometry;
- but it should make things look nicer from an arbitrary viewpoint.

Let M be the **image domain**, namely $M \in \mathbb{R}^3$, then

$$\begin{aligned}
f:\ & M \mapsto \mathbb{R} && \text{a 3d point matched to an intensity value, } I = f(x,y,z) \\
t_\alpha\ & \mathbb{R} \mapsto \mathbb{R} && \text{an intensity value matched to an opacity, } \alpha = t_\alpha(I) \\
t_\alpha \circ f:\ & M \mapsto \mathbb{R} && \text{composition of the two: assign each 3d point with an} \\
& && \text{opacity, } \alpha = t_\alpha\Big(f(x,y,z)\Big)
\end{aligned}$$

, thus f denotes some **intensity value**; and the domain of f, M, is the "finite subset" $\Big\{ (x,y,z) \,\Big|\, x,y,z \in \mathbb{Z},\ \ 0 \leqslant x < \texttt{WIDTH},\ 0 \leqslant y < \texttt{HEIGHT},\ 0 \leqslant z < \texttt{DEPTH} \Big\} \in \mathbb{R}^3$. The outcome of seam carving is **to change the image domain from M to $M' = M - \{\text{seam}\}$**.

- **"Fact" behind the scene:**

—— **carving $t \circ f$ is better than carving f ONLY IF t is designed right**.

Designing a good TF is not always easy. As the volume rendering goes, the user doesn't know exactly what t is, and so they want to vary it. Anyway, we **cannot assume the TF is a good one**.

- **Assumption on a right-designed t:**

—— **is representative of some measure of visual importance dictated by the user**.

This is not unlike the idea in 2d seam carving where the user selects an object to remove/p-reserve.

- **Goal:** asking **when t changes, if carving $t \circ f$ can be accomplished (by carving f + using the relationship between t and f) without having to apply t to f**.

It's to understand the **relationship** between carving f and carving $t \circ f$. NO assumptions on whether one will be good/bad/successful/failure. So in other words, with such relationship carving $t \circ f$ may be performed more efficiently, or, it can help user determint t.

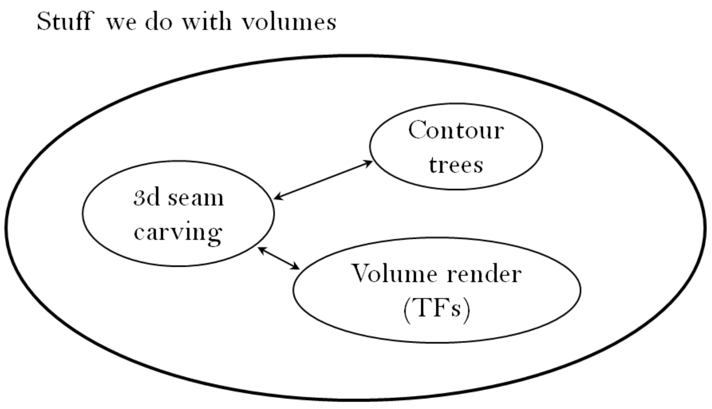


Figure 21: Relationship between SC and "something else we do with volumes"

Aug. 4th

1 Statistical feature of the whole volume

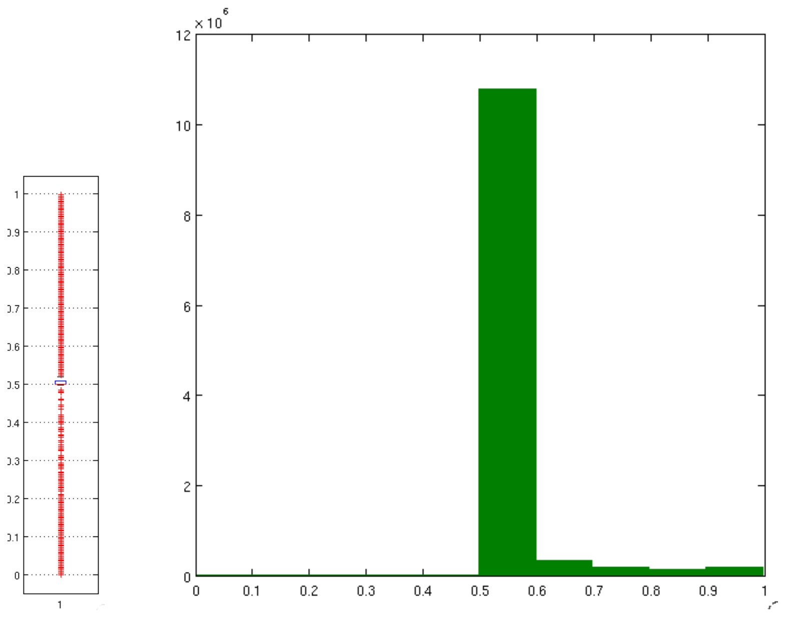


Figure 22: Boxplot and histogram of the whole (original) volume.

2 Statistical feature of each 2d seam (when carving f)

Goal: to observe statistical feature of the seam's intensity values, so as to see if seams generated by carving f can be candidates when carving $t \circ f$.

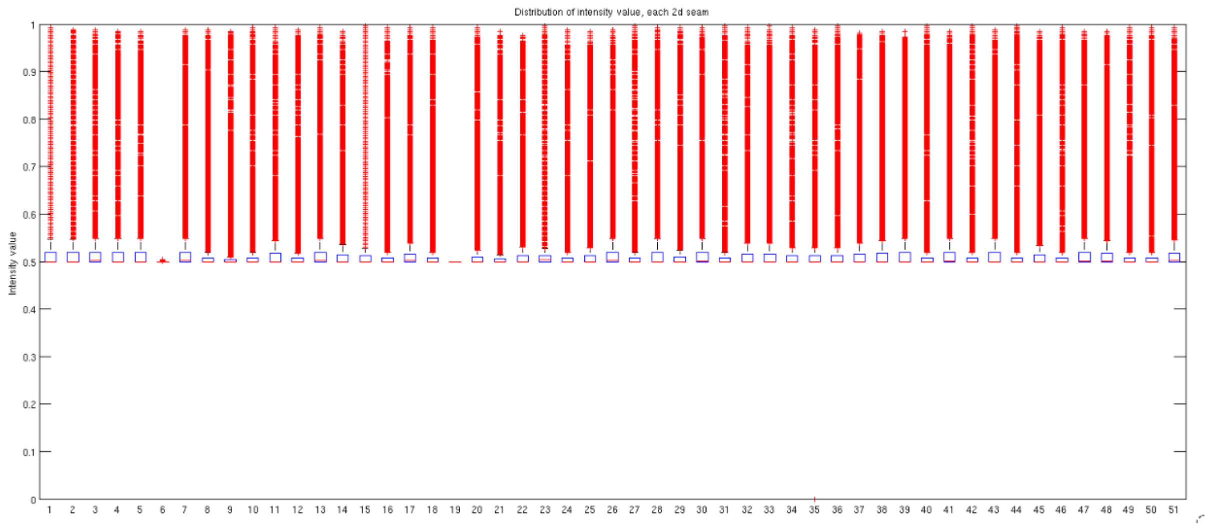


Figure 23: Boxplot: each of the first 50 seams carved from f.

It appears that these concecutive 2d seams have similar statistical distributions.

3 Generate opacity TF based on seams carved from f

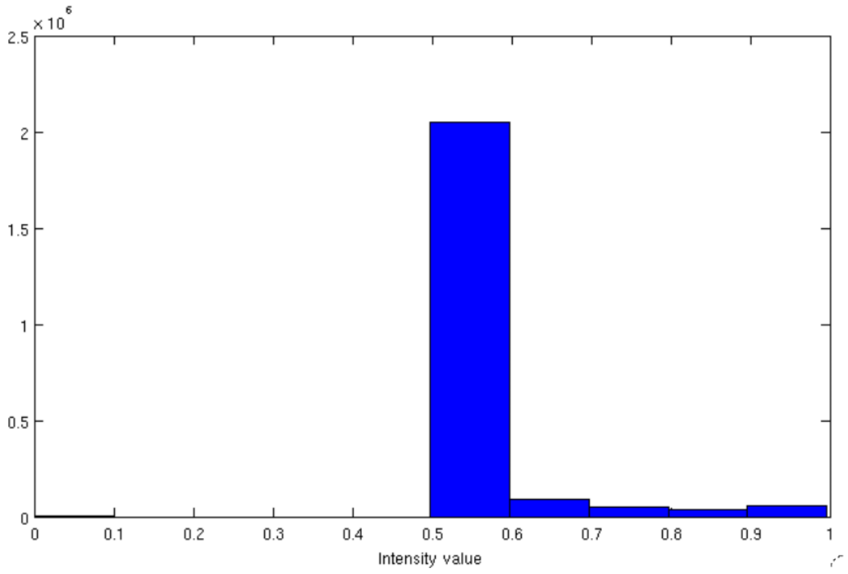


Figure 24: Histogram of the first 50 seams (cumulative, computed together).

The above figure gives the histogram of all intensity values of the first 50 seams (simply combined together, non-weighted). Such distribution (and histogram of each single seam) **looks very alike that of the original volume's**.

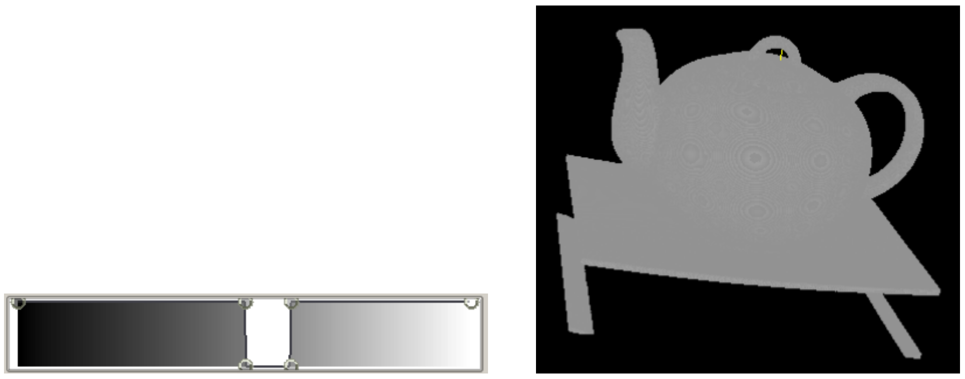


Figure 25: TF trial #1 based on histogram of seams carved/original volume.

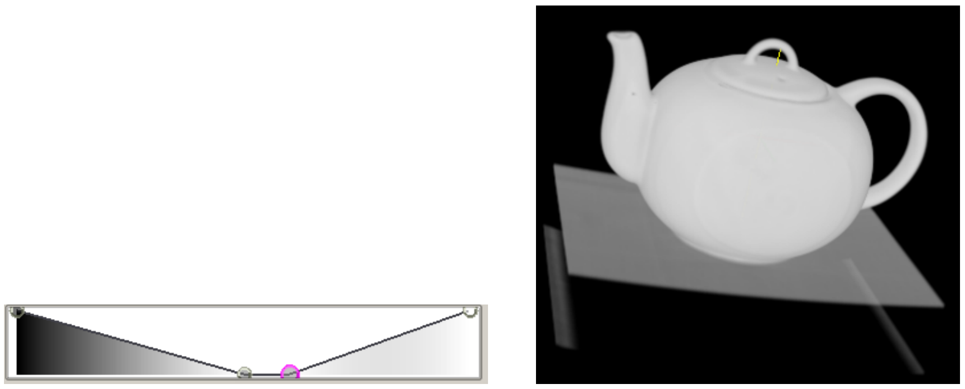


Figure 26: TF trial #2 based on histogram of seams carved/original volume.

Aug. 6th Statistical feature of seams in 2d

(1) `BostonTeapot__slice.png`

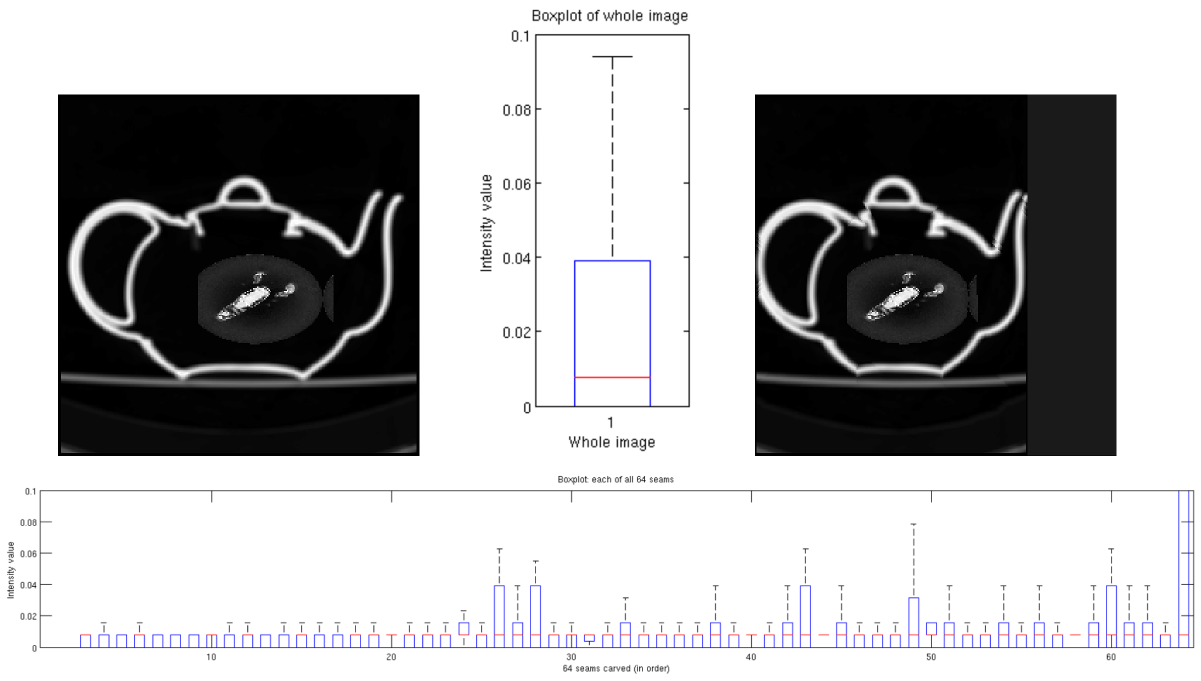


Figure 27: Boxplot: whole image vs. first 64 seams.

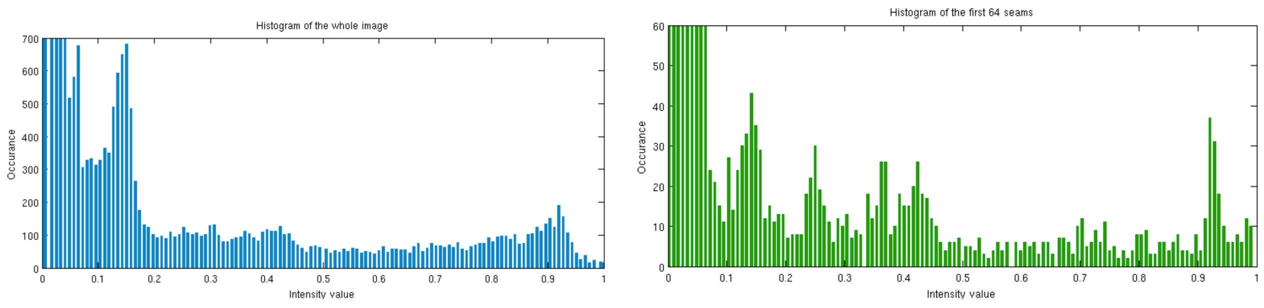


Figure 28: Histogram: whole image vs. first 64 seams.

(2) `lobster.png`

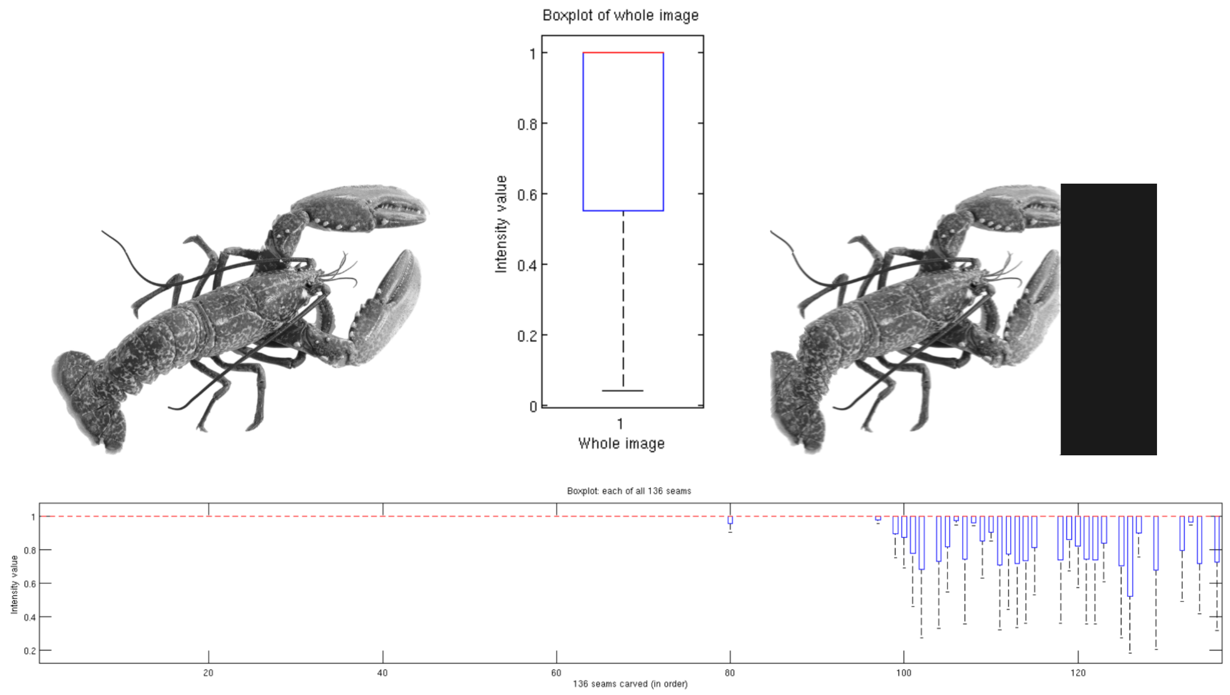


Figure 29: Boxplot: whole image vs. first 136 seams.

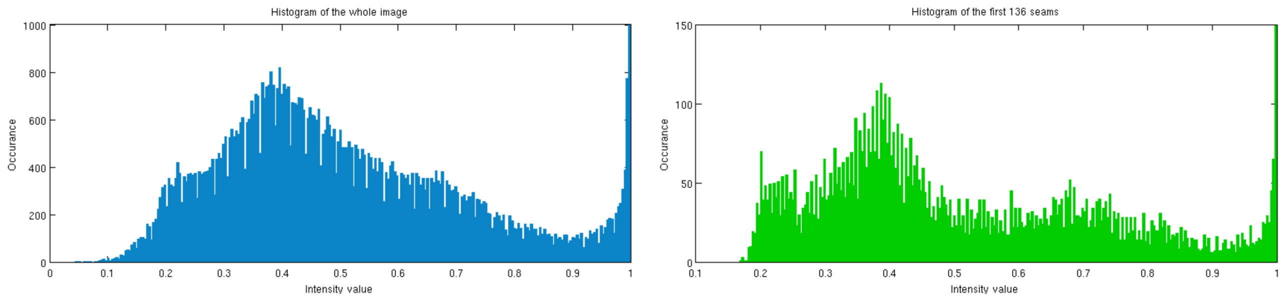


Figure 30: Histogram: whole image vs. first 136 seams.

(3) `lena.png`

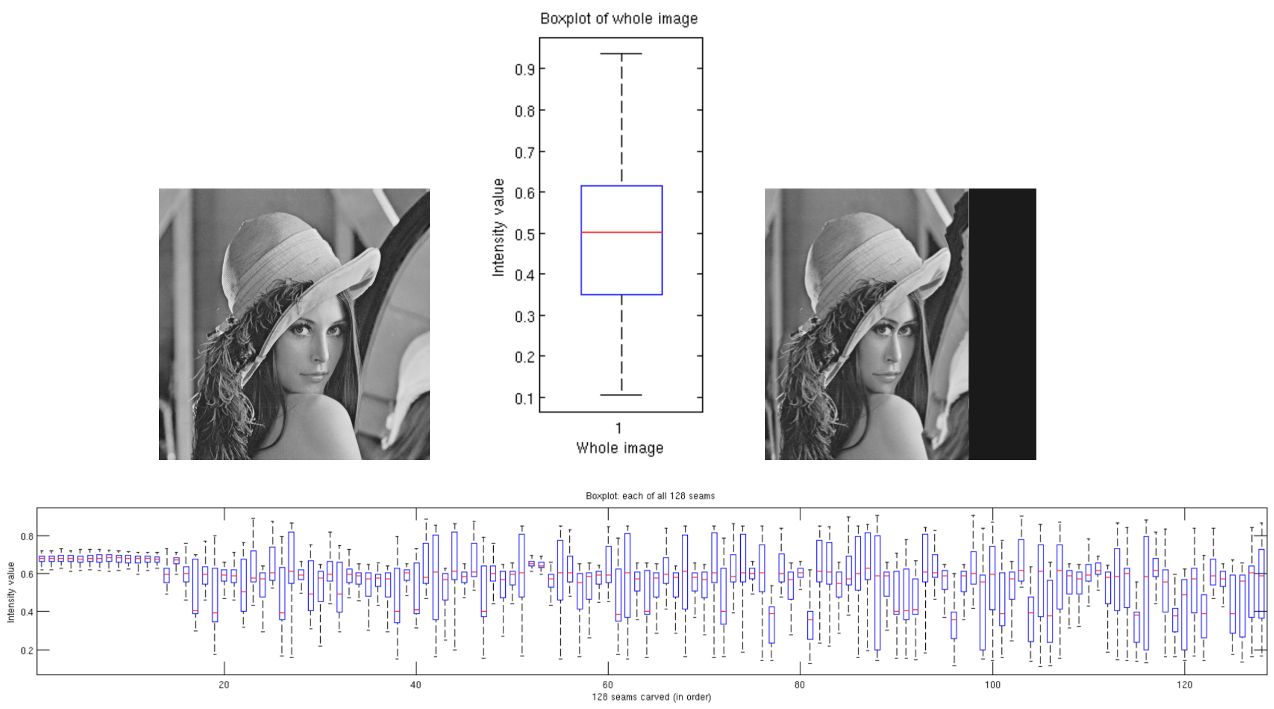


Figure 31: Boxplot: whole image vs. first 128 seams.

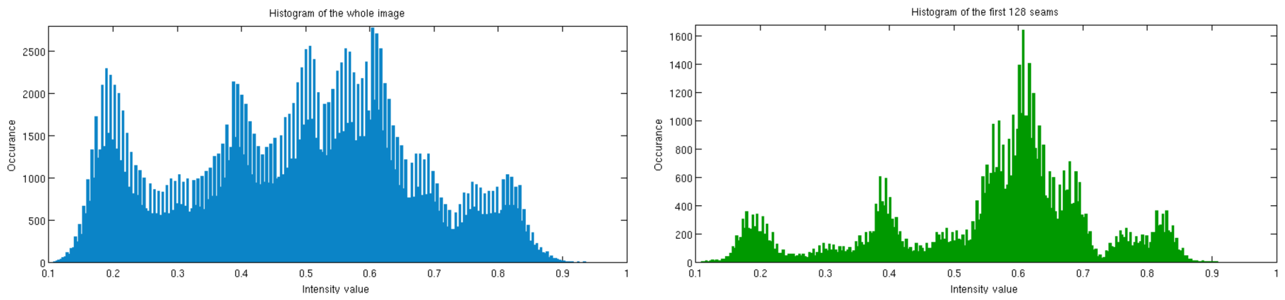


Figure 32: Histogram: whole image vs. first 128 seams.

(4) `train.png`

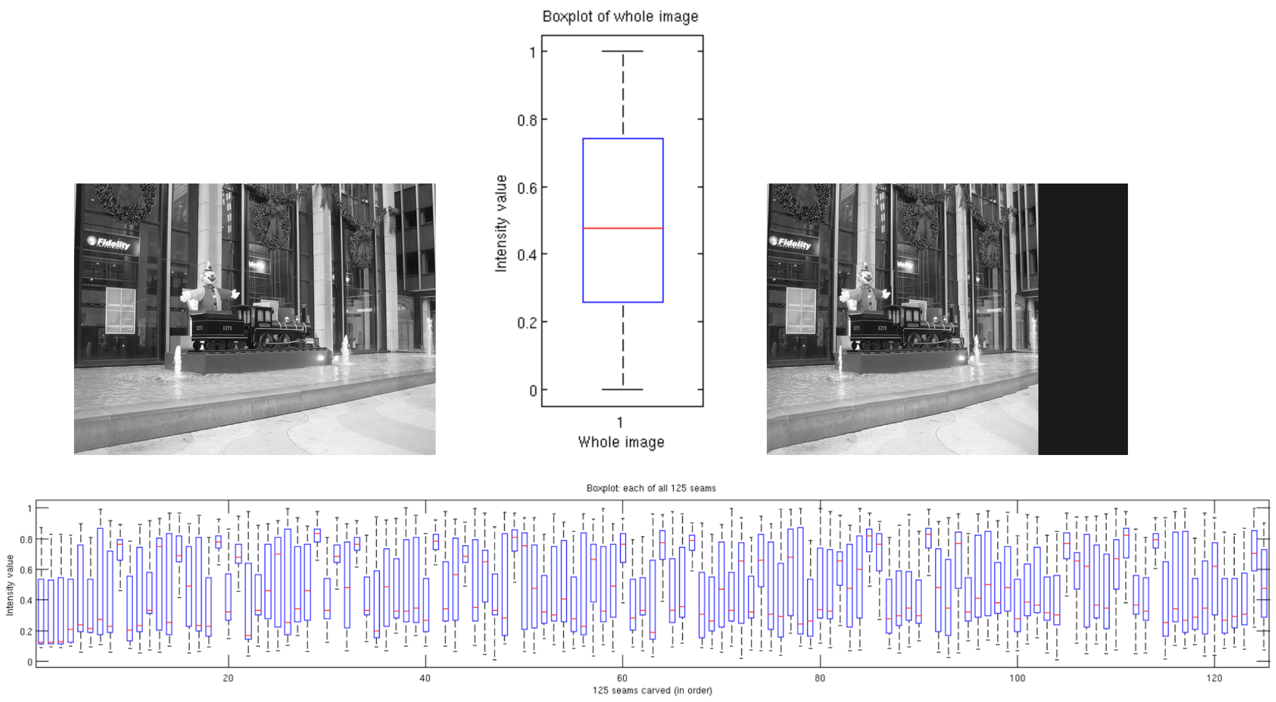


Figure 33: Boxplot: whole image vs. first 125 seams.

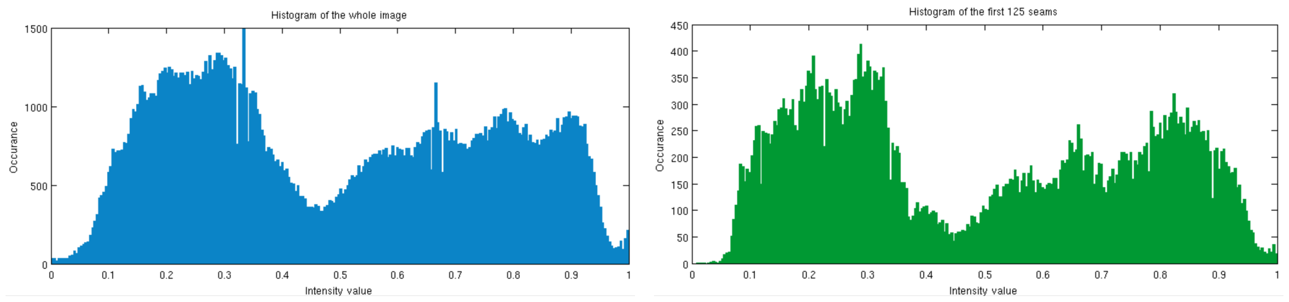


Figure 34: Histogram: whole image vs. first 125 seams.

Aug. 12 —

Topology

Smooth function: a function that has continuous derivatives up to some desired order over some domain.

retraction

✐ What is d-dimensional manifold

For an d-manifold M, $\forall p \in M$, $\exists$ an open set where all points in the open set are in d-dimensional ...

✐ Contour of scalar functions

Given a continuous/smooth function
$f : M \to \mathbb{R}$, a contour at value k is:

$$f^{-1}(k) \;=\; \{\; p \in M \mid f(p) = k \;\}$$

If M is a 2-manifold, $f^{-1}(k)$ should be 1-manifolds.

✐ Basic concepts in combinatorial topology

Topology is:

- a branch of geometry;
- the study of **properties of figures that endure when figures are subjected to continuous transformations**;
- "Rubbder sheet geometry".

The **disk**:

- is in one piece;
- has only one boundary curve.

The **annulus**:

- is in one piece;
- has **two** boundary curves;
- **divides the (2d) plane into two parts**.

• ***cell:*** any figure **topologically equivalent to a disk**.

• First priciple of **combinatorial topology**: to study complicated figures ("**complexes**") that can be **constructed from cells** by **gluing and pasting them together along their edges**.

• Euler's formula for a cell: $F - E + V = 1$

Group meeting on Fall 2015

August 19

1. Seam carving project:
   - directions: 1) TF generation based on carved seams/image after being carved; 2) topological change (contour tree) if the volume is carved.
   - $\geqslant$ two more weeks of learning topology/contour tree/reeb graph, then implement CT in C++ and do the experiment.
   - how to seam carve a volume? (what the best approach is, and reasons)
   - 2 credit hours.
2. TA on 604:

   Five reasons to join lab sessions:

   (a) **Before you start**: take minutes to read lab instructions in the room, and clarify confusion(s).

   (b) **Already started**: prepare/come up with questions to ask or discuss, making sure that you are on the right track.

   (c) **About to finish**: test your code again on lab machines, and check anything is missing/to be improved (e.g., extra credits).

   (d) **Feedback in detail**: check with TA/instructor about grading questions from last assignment.

   (e) **Flexibility**: non-mandatory; can come/leave at any time you like.
3. Courses taking:
   - CPSC 804 and CPSC 678
   - optional: MATH 656 (Topology), ECE 847 (Digital Image Processing)

Web folder permission:

- dachaos@imp17:/web/home/dachaos [11] chmod -R 755 public_html/
- drwxr-xr-x 2 dachaos cuuser 4.0K Aug 18 21:06 public_html

Add library directory (with .so file in):

- export LD_LIBRARY_PATH=/home/dachao/Documents/libtourtre-master:$LD_LIBRARY_PATH

Contour tree notes by Sep. 10

`I.` In general (ground-truths):

- JT, ST and the upcoming CT are all drawn as **undirected** graphs.
- Nodes should be at the correct height, or in a line from bottom to top.
- "is a leaf" simply means "`node.outdegree() == 1`"
- Properties:
  - Join tree has the correct DOWN degree;
  - Split tree has the correct UP degree;

  thus, at the end of each iteration we add an edge by looking into

  JT if it's a lower leaf, or **ST if it's an upper leaf**.

`II.` Initial enqueuing (before the `while` loop):

- Not any kind of priority queue, meaning that **the order doesn't matter**. All eligible candidates can be enqueued in a random order.
- Enqueuing constraint: $\boxed{\texttt{down}\text{degree\_in\_}J_T + \texttt{up}\text{degree\_in\_}S_T \;\texttt{==}\; 1}$, namely (in the JT/ST) those who are either
  1. a global extremum, or
  2. a local extremum on one tree but a regular point on the other.

`III.` Rewritting Algorithm 4.2 in *Carr's paper*:

Algorithm to merge J_T and S_T

```
1  Leaf queue initialization:
       For each node $i$
           If down-degree in $J_T$ + up-degree in $S_T$ is equal to 1 *,
               enqueue $i$.
2  While leaf queue's size is larger than 1 do
     • Dequeue the first node $i$ (at front of the leaf queue).
     • If $i$ is a lower leaf in $J_T$,
           find incident arc $\textcircled{i}—\textcircled{j}$ in $J_T$;
       Else
           find incident arc $\textcircled{i}—\textcircled{j}$ in $S_T$;
     • Add $\textcircled{i}—\textcircled{j}$ to $C$, the contour tree.
     • $J_T \leftarrow J_T \ominus i$, $S_T \leftarrow S_T \ominus i$.
     • If node $j$ now satisfies enqueuing condition,
           enqueue node $j$.
```

* we can name it as the ***enqueuing condition***.

What is contour tree?

What is join tree?

What is split tree?

Magic: Stealing Pixels Secretly

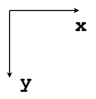
—— a not-too-short tutorial for Seam Carving

October 7, 2015

Consider an N-by-N (N=4) image:

$\mathbf{I} =$

201	209	213	199
211	220	227	188
194	180	49	52
190	174	150	107



A simple *energy function* defined on it:

$$e_1(\mathbf{I}) \;=\; \left|\frac{\partial}{\partial x}\mathbf{I}\right| + \left|\frac{\partial}{\partial y}\mathbf{I}\right|$$

, where $\frac{\partial}{\partial x}\,\mathbf{I}(x,y) = \mathbf{I}(x+1,y) - \mathbf{I}(x,y)$.
OR $\frac{\partial}{\partial x}\,\mathbf{I}(x,y) = \frac{1}{2}\cdot[\mathbf{I}(x+1,y) - \mathbf{I}(x,y) \;+\; \mathbf{I}(x+1,y) - \mathbf{I}(x,y)] \;=\; \frac{1}{2}\cdot(\mathbf{I}(x+1,y) - \mathbf{I}(x-1,y))$.

$\mathbf{E} =$

12	15	28	25
26	47	217	179
18	135	104	58
20	30	144	98

Well, what would be the best ***vertical seam***?

Reminder, def. of a vertical seam:

$$\mathbf{s^x} \;=\; \{s_j^x\}_{j=0}^{\texttt{HEIGHT}-1} \;=\; \{x(j),\ j\}_{j=0}^{\texttt{HEIGHT}-1}$$
$$s.t.\ \forall j,\ \ |x(j) - x(j-1)| \le 1$$

1-by-N image:

12	15	28	25

Sort them! Find the minimum. This is a vertical seam of length 1.

12	15	28	25

2-by-N image:

12	15	28	25
26	47	217	179

The vertical seam now should consist of two pixels (`HEIGHT` $= 2$, one at each row).

Recall that we want to find an optimal seam that minimizes the seam (energy) cost, i.e. $\mathbf{s}^* = \min_{\mathbf{s}} E(\mathbf{s}_1) + E(\mathbf{s}_2)$. Let's first look at the second row —— there are 4 position.

An immediate question: **starting from each of these four positions, what would the optimal seam path**?

Just to make us on the same page, a vertical seam that starts from the third position at the second row looks like:

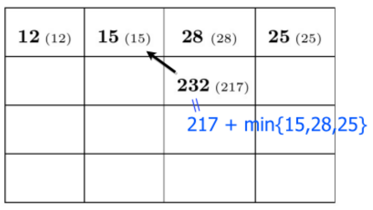
1st row	12	15	28	25	12	15	28	25	12	15	28	25
2nd row	26	47	217	179	26	47	217	179	26	47	217	179

It's much easier to answer which option is the best among these three: it's the left one! ($217 \longrightarrow 15$) The math we implicitly did here is finidng $\min\{15, 28, 25\}$. We actually divide to problem, at the second row, into four **sub-problems**.

If we think of solving the problem recursively as the problem size (# of rows) increases, i.e. trying out all possibilities, it will be slow —— there are too many!! (grows exponentially)

Solution: record the "footprint". Record what decisions we make at each step as we go one hop down, by **cumulating the energy along the best seam so far**. It's store in the ***cumulative energy array***

$$\begin{aligned} M(i,j) &= e_1(i,j) \\ &\quad + \min\left(M(i-1,j-1), M(i,j-1), M(i+1,j-1)\right) \end{aligned}$$



fill it out:

12 (12)	**15** (15)	**28** (28)	**25** (25)
38 (26)	**59** (47)	**232** (217)	**204** (179)
44 (18)	**173** (135)	**163** (104)	**262** (58)
64 (20)	**74** (30)	**307** (144)	**261** (98)

Get the position (col number) $x^* = \min_x M(\texttt{HEIGHT-1}, x)$ by sorting the last row (find some way to deal with ties!). x^* will be where we start to "backtrack" the optimal seam, from the last row.

And we're done!

Dynamic Programming

Part of the challenge is that the name itself doesn't really convey what the technique is about (like "divide and conquer", the name tells you what it does). Dynamic programming is kind of a historical artifact:

- the "programming" part does not mean "computer programming", it means "mathematical programming" in the sense of optimization. The word "planning" might be more appropriate there.
- "dynamic" comes from the fact that this technique was originally used to solve multi-stage/decision-making processes.

So the name kindof tells you nothing about what the technique is. The technique is basically **solving problems recursively**. (compare with: greedy algorithm, exhausted search)

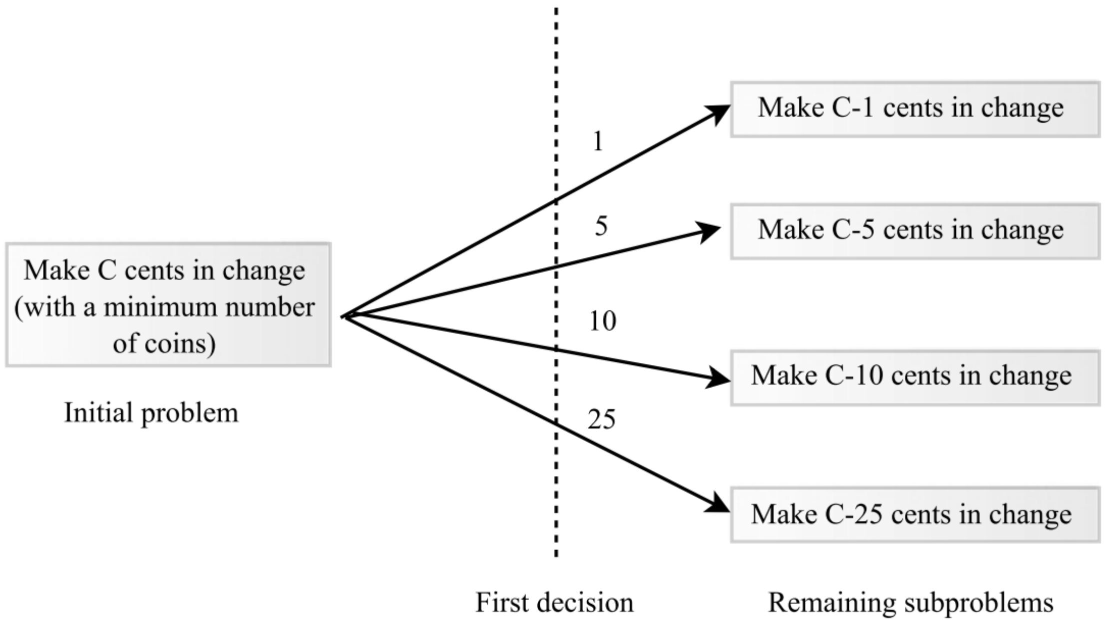
Example: Making Change

- We have N different denominations of coins
  (e.g. US coin system, 1 cent, 5 cents, 10 cents, 25 cents)
- Can use as many coins of each denominations as we wish.
- What is the **minimum number of coins** we need in order to construct exactly `C` cents worth of change?

What's the usual way to do this? Just the largest value first, be greedy. For US coin system it does work, it does give you a minimum number of coins. If we come up with a strange country such that for greedy it's not always optimal:

1 cent, 11 cents, 20 cents

22 cents? Greedy is not optimal.


Make C cents in change (with a minimum number of coins)
Initial problem
1
5
10
25
Make C-1 cents in change
Make C-5 cents in change
Make C-10 cents in change
Make C-25 cents in change
First decision
Remaining subproblems

There are different ways of approaching the algorithm problem from a high level; one of the main techniques we have for solving a problem is a kind of **sequential construction** —— you **build the solution one step at a time**. Each step of the algorithm you try to **pick one thing to add to your solution**. $\longrightarrow$ All the algorithm has to do is to figure out what's the first coin to put into the solution. And then it's going to figure out what's the next one (to put into the solution).

If I could just figure out how to make that first decision, then after that what am I left with? — a remaining sub-problem that is the exactly same form of the original. It's smaller, it involes a smaller number of `C`. So I could just **recursively solve that**. This type of probalms they have sort of a "recursive structure", so a very plausible way to solve these problems is **to make that first decision, and just use recursion to finish things off**.

- **Greedy:** initial decision can be made safely and irrevocably; is really great because it has such a simple rule to make each decision. They are nice for just really really small simple problems, but in real world, most problems are mean enough that g.a. don't give you the optimal solution.
- **Exhaustive search:** Recursively try all possible first decisions, usually in some sort of "greedy" order, and usually with some sort of pruning (heuristics) to speed things up.
  It's going to grow exponentially fast (an immense set of solution). This is really slow though, usually. What's the key of making this run quickly? Pruning! To start with the biggest value (greedy) usually gives you a "descent" solution. **Search everything, but keep in your hand the best (or best k) solution(s) and as you do the search anytime you realize that nothing ahead of me is not gonna improve on the best solution, just backup and prune the search.**
- **Dynamic Programming:** Solve all possible subproblems from smallest to largest. Suppose "someone" tells you the answer of each of the sub-problem. That is what dynamic programming is all about!

  What it does is it actually solves all the problems from the bottom up: what if I want to make change for 1 cent? What if 3 cents?... by the time I reach the actual value of `C` (that I actually care about), I've already solve everything smaller!! That makes the initial problem really easy because we already know the solutions to the smaller problems that I would reach via the that first decision.

A lot of people think of it as a "bottom-up" approach, or an approach kindof involves tabular solutions to all possible problems. It seems like you might be wasting time, because you're solving a bunch of problems that you don't really care about, i.e. you're solving everything; however, the benefit of that is when you actually reach the big problem you originally care about, you can easily solve it —— easily make that first decision, because these "subproblems" are already solved.

Page 0:

Today this useful tool that I'm going to talk about is an image manipulation approach, which I refer to as "stealing pixels away"! Let's see what it's about.

Page 1:

Image manipulation has been around ever since photography has been around. As we can imagine it was very difficult to do at the beginning, but today we have computers, with terrific tools to help us manipulate the images.

Often times, the manipulation we'd like to do is to **change the aspect ratio of an image**: on one hand, the display device changes from a cellphone to a computer screen, to maybe the screen in the theater; on the other hand, for example, if you want to change the header background of my personal website, you probably have to "sqeeze"-and-"stretch" the background photo to the same resolution as the header. (so in order to show the stuff better we need to change them a little bit)

Page 2:

One simple operation is to **crop** it. Here's an image of this polar bear. Cropping gives you the region of interest, but something else has to give (especially when objects are located far from each other). Well now we may think what about just **scale** the image? As we can see some artifacts/distortion that make the image look NOT so real, even if some interpolation is done to reduce the aliasing. **What we want, is a way to change the size of the image while still preserving the content, that's why it is referred to as "content-aware" resizing**.

Page 3:

This so-called "seam carving" approach defines (what we call) a "seam path", which is just a set of pixels that go from top of the image to the buttom — they contain 1 pixel at a row, and they're connected.

Page 4:

What's interesting is if you take such a seam and remove it from the image, what happens is the image remains "intact" BUT, as I removed one pixel for each row so **the width is one pixel smaller**. And as we do it concecutively, the image changes its aspect ratio.

So obviously the big question is **which seam do we remove**? The right seam to remove is actually the one that would be **the least noticable**, such that it would be able to trick your eyes. Well "least noticable" means it contains the least amount of "information", or say "content". We want to show what's important in the image, and specifically, the **edges**.

Page 5:

So the math behind the scene is we define an **energy function** based on some common edge-detection operator, such as gradient. And use dynamic programming to find the optimal seam path that touches as few edges as possible.