

Technion – Israel Institute of Technology

Department of Electrical Engineering

Laboratory of Computer Graphics and Multimedia

Project Report for

Content-Aware Image Resizing Using Seam Carving

Author: Tsur Amit – 038164927

Supervisor: Varod Danny, B.Sc.

Abstract

Image resizing is mainly known as cropping or scaling of images. These methods of resizing are not content-aware; they treat each element of the image the same as the other. Now more than ever, with the popularity of the Internet and mobile devices and with the power of modern CPU/GPU, effective image resizing is getting more and more important and necessary.

This project tries to implement a method of resizing an image in a content-aware fashion. Content-aware shrinks/enlarges the image by removing/inserting the least interesting parts of an image, thus resizing an image with regards to its content.

Notes

Throughout the paper, there might be references to actions in horizontal and vertical direction; if an elaboration in one direction is missing, the reader can assume that the same action can be done in respect to the other direction in a symmetric manner. This is also true for enlargement and shrinking.

Introduction

Seam Carving is achieved by firstly assigning a weight to each element of the image. This can be done in various methods and depends on the type of image. Some methods can perform better than others on specific images. The next step is to calculate a cumulative map of weights, used to determine how important/interesting each element of the image is.

We then add or remove to the image the least interesting elements of it, hence keeping the affect on the content at the minimum.

This project focuses on implementing a fast and efficient content-aware image resizing application based on the article by Shai Avidan and Ariel Shamir.

Project Goals

* Implement Shamir and Ariel’s algorithm.
* Build a comfortable GUI to use the implementation.
* Test its many applications.
* Try and analyze different functions.
* Optimize it further to achieve better performance using various methods.

Algorithm

Pixel Value

In order to efficiently work with pixels, we need a way to assign a value to every pixel. This value has to take the structure of the pixel (RGB, for instance) into account.

In our application we’ve chosen to use the pixel’s luminance. Luminance is a photometric measure of the luminous intensity. We’ve used *Luma*, which is the representation of luminance in a video monitor. Luminance of a pixel is achieved with the following:

Y = 0.2126 R + 0.7152 G + 0.0722 B

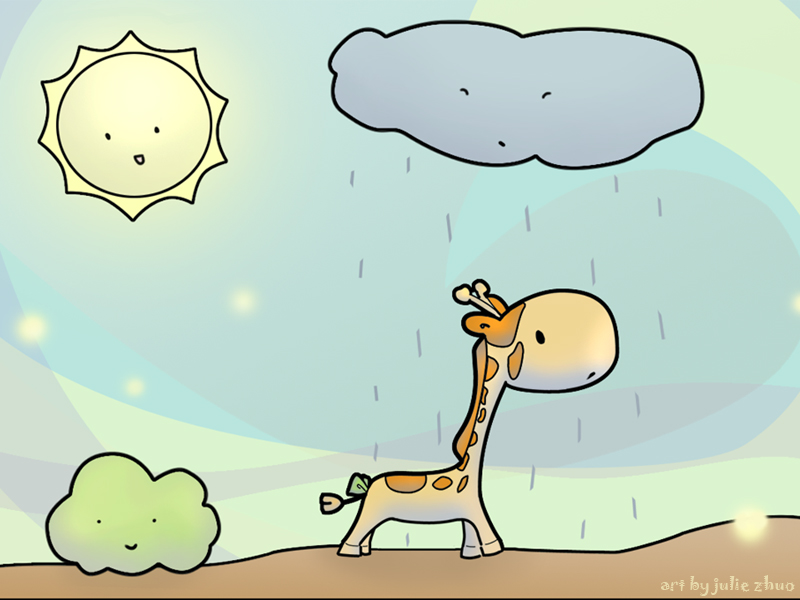
We’ve chosen luminance because it easily and efficiently normalizes pixels in a satisfying way, which allowed us to get a representing, clear energy map of the image using the different energy functions (will be discussed later). Another way to normalize pixels is by using grayscale intensity, which has shown similar results.

Pixel Energy

After being normalized, each pixel must also have another numerical value, a value which depends not only on the pixel itself, but also on its environment, indication of how ‘interesting’ it is. The numerical value is called *Energy*.

We’ve implemented three ways to represent the energy. The three functions are similar in their core; they all use small convolution masks in order to approximate the first derivative of the image brightness function, thus enhancing edges. Additional material on the energy functions can be found in the references chapter.

Below are an image (which will be the main example image of the paper) and its energy bitmaps generated using the three different energy functions. The images are in a non-scaled resolution of 800x600; these dimensions were chosen in order to convey the results to their fullest extent.



Original Image:

This is the original and untouched image. We will use it as our main example image, along with other images.



Sobel Energy Bitmap:

Sobel is the default energy function. As you can see, The edges are very bright and clear. Notice that there aren’t many objects in the picture; we will also supply energy bitmaps of images (not drawing) which contain a lot more edges and information, thus presents poorer results.



Prewitt Energy Bitmap:

Prewitt is very similar to Sobel in its tolerance; it’s very ‘aggressive’ in edge detection, although a little less aggressive than Sobel.



Roberts:

Unlike Sobel and Prewitt, Roberts is a lot less aggressive in its edge detection; notice that the stripes in the background of the Sobel and Prewitt bitmaps are almost completely invisible.

Less aggressive edge detection has its pros and cons, we will discuss it leter.

Seam

Seam is an 8-connected path of pixels. A vertical seam is a seam from top to bottom which contains exactly one pixel from each row of the image, and more specifically, for an image  of size  we define a vertical seam to be:



Where x is a mapping .

Cumulative Energy

After calculating the energy, we find the cost of the minimal seam. We define the cost of a seam:



In order to fine the minimal seam cost, we use dynamic programming. We traverse the image from the second row to the last, and calculate the cumulative minimum energy  for all possible connected seams for each entry 



After calculated, the cumulative energy is saved in the cumulative energy map.

Optimal Seam

After the completion of the process, we scan the last row for the minimal value. The minimal value is the cost of the minimal (optimal) seam. We then backtrack from the minimal entry in an 8-connected manner to the minimal neighbor until we reach the top of the image again. The minimal path defines the minimal seam.

Index Map

Vertical index map of image  of size  is a sized  integer array.

For any indices , the following holds:

 The pixel in the image is removed in the  seam removal

The index map is used in order to efficiently resize the image. The usage will be explained throughout the paper.

Seam Removal

In order to shrink an image of size  to size  where , we remove the first  optimal seams.

Removing the least interesting part of an image is supposed to leave a minimal impact on its content.

Seam Addition:

In order to enlarge an image of size  to size  where , we add the first  optimal seams.

We do not add the exact seam. For every pixel of the seam, we add the average of that pixel and its left/top neighbor pixel. This is done in order to make the new pixel ‘blend’ with its surrounding, rather than looking exactly like the original pixel, which will look artificial.

As with removing, the impact on the content is kept to a minimum.

Implementation

Environment

Software

Development began on Windows 7 Beta 1 using Visual Studio 2008 and C# 3 (.NET Framework 3.5 SP1) and was completed using Windows 7 RC Visual Studio 2010 Beta 1 and C# 4 (.NET Framework 4 Beta 1). The transition to .NET 4 was crucial as the .NET 4 Beta 1 implementation of the Parallel library performs much better than the .NET 3.5 Parallel extension beta. The user interface has been implemented using the Windows Presentation Framework (WPF). There’s has been several uses of the .NET 4 Parallel library in order to greatly improve performance.

Hardware

The computer used for development uses a Pentium i7 920 CPU (8 cores), 3 GB DDR 3 1033 MHz of memory and a Radeon 4870HD graphics adapter.

The program was initially implemented without the .NET Parallel extension, and was performing very bad. After some code has been refactored to use the Parallel extension, performance has improved by hundreds of percents, and indeed there is indication for the utilization of many cores throughout the operation of the program.

Design

The design of the program is nothing complex. There weren’t any special challenges as the program has few components and fewer connections between components, as can be seen in the class diagram. Most of the class are ‘regular’ classes, there’s a static class for utilities, a static class for constants, an abstract class for the energy function and it’s extenders.

Major Classes

We will list the major classes and their important members. We will elaborate on each of these throughout the paper.

Seam

The Seam class represents a seam. The class holds the following data on the seam:

* The direction of the seam (horizontal or vertical).
* The starting index of the seam (height if horizontal and width if vertical).
* The absolute coordinates of the seam’s pixels.
* The value of the seam (sum of cumulative energy of all pixels, as defined earlier).

Energy Function

Abstract class, represents an energy function. It holds the energy map and supplies the following methods:

* Compute energy: Computes the energy map according to the energy function.
* Compute local energy: Refreshes the energy map to reflect changes made to the images. Its main use is to support user input (will be discussed later).
* Get pixel energy: Gets the energy of a pixel. It is abstract because the energy of a pixel depends on the energy function.

There are three other classes (Sobel, Prewitt and Roberts) which derive from Energy Function.

Cumulative Energy

This class holds the cumulative energy (the only implemented seam function) of the image and contains methods to build and maintain it. The class is also responsible for calculating and building the seams and calculating the index maps. It holds the following data:

* The energy function that has been chosen.
* Horizontal and vertical cumulative energy maps.
* Horizontal and vertical Boolean maps, used to determine whether some pixel has been used in a horizontal / vertical seam.
* Horizontal and vertical lists containing the cumulative energies of the calculated seams, sorted by cumulative energy in an ascending order.

And the following methods:

* Compute cumulative energy map: computes the entire cumulative energy map in a specific direction.
* Build seam: Builds a seam.

Seam Image

This class represents an image. The class hold all the information about the image and supply many methods. Notable data members:

* Bitmap representing the image.
* Bitmap representing the energy.
* Bitmaps representing the horizontal and vertical index maps.
* Energy function.
* Seam function (cumulative energy).
* Horizontal and vertical index maps.
* Lists of calculated horizontal and vertical seams.
* Lists of removed and added seams.
* Bitmap representing the old image (will be discussed).
* Current cache configuration (will be discussed).
* Dirty flag (will be discussed).

Methods:

* Compute energy map: Uses the energy function to calculate the energy map.
* Cache refresh: Refreshes all the caches (energy, cumulative energy and index maps).
* Generate bitmaps: Generates bitmaps for energy map and horizontal and vertical index maps.
* User input: Refines the image energy according to user input (will be discussed).
* Get k best seams: Calculates the first k seams in some direction.
* Calculate index maps: Uses the seam function to calculate index maps (also build seams).
* Carve: Carves seams.
* Add: Adds seams.

Flow & Deeper Into Algorithm

In order to best understand how the application produces data and manage it, it is best to learn how it works through a flow of its operation. I will list all the possible actions and how exactly they are performed in the flow of the program:

Open File:

Upon opening an image, a SeamImage class is created to maintain it. Upon creating a SeamImage object, the pixel’s energy is also calculated using the chosen (or default, which is Sobel) energy function and saved.

User Input:

Before actually altering the image, the user can use two brushes in order to change the image energy. The purpose and applications of this feature will be discussed. The user can also crop the image. Cropping the image will force a refresh of the caches.

Set Cache Limit:

The application manages many caches (seams, index maps, energies…). It is up to the user to decide how dependent on the cache the application is going to be. It can be set to work in a ‘real-time’ mode and in a ‘precached’ mode. The effects of the cache will be discussed.

Preprocess:

When ready to alter the image, the user clicks the ‘Done Editing’ button. Pressing the button will begin the calculating of data. If the user had specified some input regarding the energy, there will be a refinement of the energy according to that input. The cache will then be built according to the cache level selected by the user. For any cache type, the program behaves as follows:

* ‘real-time’: No actual cache is kept. Every action (carve/add seams) requires calculation of index maps and seams. In order to carve k seams, k seams in that direction are built and index maps are set accordingly (not entirely, only k rows / columns). After every action (carve/add k seams), the energy map is refreshed and cumulative energy map is rebuilt.
  + The energy map is refreshed in order to preserve changes made to it by the user, otherwise is could have been rebuilt. Using parallel code, rebuilding the energy map can be cheaper than maintaining it.
  + The cumulative energy map is rebuilt for the same reason. It calculated in a trapezoidal manner, meaning that multiple changes to the image nearly invalidate the entire map.
* ‘precached’: All the possible information regarding the image will be collected and used until depletion or invalidation. This means that for an image of width , for instance, carving  () seams and then another  required no additional cache calculation and no changes to the energy. When trying to carve another  seams (), a recalculation of the caches will be invoked.
  + Changing direction invalidates the caches and forces a recalculation. This is due to the fact that horizontal and vertical seams can collide in more than one location, meaning that the index map of the opposite direction could be destroyed. There is a way to overcome this issue, but it hasn’t been implemented. You can read about it in the original paper.
  + Changing action type (carve or add) also invalidates the caches. SUPPLY EXPANATION TO THIS.
* In between: Some of the data is cached and is used until depletion or invalidation.

Note that while using cached data requires less calculation and is overall faster, it produces inferior results because of use of stale data. After carving k seams, since pixel energy and cumulative energy is environment dependent, the existing data needs to be refreshed in order to reflect the changes. ‘real-time’ mode is slower due to recalculation of caches after every operation, but is also more optimal.

Order of Cache Calculation/Refresh:

1. Energy map is being built/refreshed. If refreshed, a list of carved/added seams is used in order to refresh it.
2. Cumulative energy map is being (re)built. As the pixel’s energy had changed, the cumulative energy has to reflect these changes.
3. Seams/Index Maps are (re)built: After building the cumulative energy maps (as defined earlier), we keep sorted lists of vertical and horizontal cumulative energies. We use these lists in order to find the starting index of the next optimal seam. We traverse these lists and build the seams in the required direction. While building the seams we also set the index maps.

Building Seams / Index Maps:

As described above, we get the starting index and begin building the seam. The start index is actually the rightmost (horizontal) or bottom (vertical) pixel of the seam, meaning that in order to find the entire seam we need to traverse to the leftmost (top) part of the image. We use dynamic programming to find the neighbor (left right or straight are the three possible directions to traverse) with the lowest energy that hasn’t been used yet. If a left or right neighbor is already used, we try to skip it and get the next left or right neighbor. This is done in order to always have an option instead of finding out that all of the neighbors are used. After finding the lowest neighbor, we mark it as used and continue from it. We continue the procedure until reaching the opposing edge of the image. Since the lists of cumulative energy are sorted, while setting pixels as used during the calculation of the  seam, we also set  in the appropriate index map, where  is the location of the current pixel, thus easily building the index maps.

Carve:

After the calculation of the data, the user can now carve seams.

Carving seams produces a new image of the target dimensions. Seams are carved using the index maps. For example, when carving k seams, we traverse the index map in the relevant direction, and for every pixel , we copy it to the target image iff , thus effectively removing k seams.

In order to support cache, we keep track of how many seams were removed, and in order to remove  seams after another k has been removed, we traverse the index map again and copy all pixels in coordinates that satisfy . As long as the cache is valid and hasn’t depleted, for each carve operation, we traverse the original image (just after the last cache rebuild), and copy all the pixel’s which haven’t been removed yet. When running out of cache or invalidating it, the cache is rebuilt, the lists of removed/added pixels are emptied, and a new original image is set.

Notes:

* Traversal of the index maps is done in a parallel manner.
* When using ‘real-time’ cache settings, removing a single pixel k times produces better results than removing k pixels at once. This is due to the fact that in ‘real-time’ the data is refreshed after every operation. When using other cache settings, as long as there’s more data in the cache, the ‘granularity’ of removal has no effect, since order of removal has been pre-determined.

Add Seams:

Like removing seams, adding seams, too, is done using the index maps. In order to add k seams, we traverse the relevant index map and for every pixel with index less or equal to k, we copy it, and then add its average with the next (yet to be copied) pixel. If there isn’t a next pixel to average with, we simply copy the pixel twice.

As with removing seams, we keep track of added seams in order to allow the use of cache.

Notes:

* In the current implementation, using ‘real-time’ cache for seam addition will result in a stretching artifact This is due to the fact that ‘real-time’ cache refreshes the caches after every operation. Supposed we have added k seams in ‘real-time’ mode and now want to add another k’. Whether or not k > k’, since we are adding the seams with the least energy, refreshing the cache will render these previously added k seams as the seams with the least energy, meaning that adding another k’ will result in adding all or a subset of the added seams *again*, making the image look as if it was stretched, rather than enlarged with relation to its content. An example of the stretching artifact will be given later on.

In order to prevent the stretching artifact, when adding seams we set the cache settings to ‘precache’. This way we can add as many seams as the image’s width/height.

The original paper explains how to better implement seam addition in order to prevent the stretching from happening (using more sophisticated index maps, which better support image enlarging).

Applications:

We will elaborate on some of the main applications of Seam Carving and provide example images.

1. Image Resizing: The main application of Seam Caring. We will begin with size **reduction**.



This is the original image, with its width reduced by about 150 pixels (almost 20%) and its height untouched. Notice that apart from some minor artifacts (circled in red), none of the interesting parts of the image has been damaged.

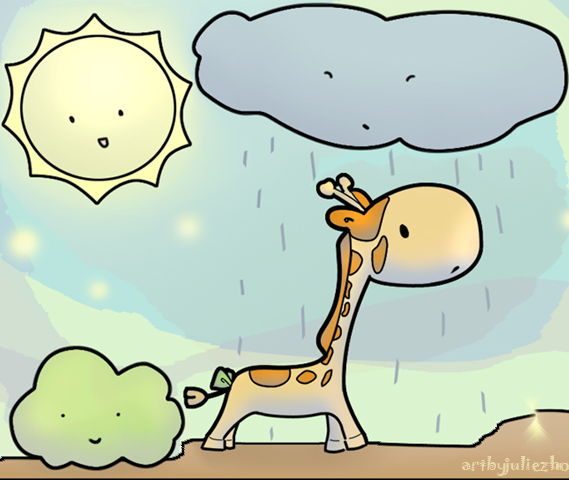


After reducing about 100 more pixels, we can easily notice that the grey cloud on the top right corner has been deformed; it’s important to mention that as long as it was possible, seams removed did not contain pixels from any of the four major objects in the image.



This is the image after removing an additional 120 pixels. The image illustrates two facts:

1. Removing seams critically depends on the content of the image (we will discuss it more in the paper).
2. Less interesting objects (objects with few edges) such as the bush and the cloud gets removed earlier than more interesting objects (such as the sun and the giraffe).



This is the original image, reduced from 800x600 to 570x480. There are still some minor artifacts but the important objects remained untouched. Further seam reduction will damage the interesting objects in this image.

The image used throughout this paper produces very satisfying results due to the lack of details in it, mostly because it’s a drawing, which means there are very few and distinct edges in it.

We will now show the results on a less optimal image:



In this image we spot 3 main differences from the previous image we used:

1. It’s not a drawing. There objects are not smooth and vectored, there are many edges and curves which will toughen the classification of objects as ‘interesting’.
2. It’s a mess. There are abundant details, objects and information, not to mention edges.
3. It’s deceptive. One of the most interesting object (the black horse) is rather smooth and black, thus containing few edges when compared to the foliage and the fence in the background. In addition, the right horse is facing the camera, hiding its entire body.

Although we define the horses as the most interesting part of the image, the foliage and fence produce many edges and the cluttered mess makes the horses appear uninteresting.



This is the Sobel bitmap our image. Notice that the scenery (foliage, ground and fence) renders the left horse as a “energy hole”, effectively marking it as a passage for optimal seams (both horizontal and vertical since the scenery is speeded across the entire image).

In this energy bitmap it can be clearly seen that in order to improve seam carving result, a reduction of the energy intensity is required. As we have previously seen, Sobel is considered an aggressive edge detector. We will try the less aggressive Roberts.



We can see that the Roberts energy bitmap is much darker than the Sobel bitmap. Using less aggressive energy function produces an energy map that is less sensitive to clutter.

We will now try resizing the image using the Sobel and Roberts energy functions.



This is the original image reduced from 800x520 to 740x520 (60 pixels were removed). As expected, most of the seams removed included the black horse (most of its legs were removed, as they form the longest seam path that intersects with it). We can also spot some artifacts on the bars of the fence and some extra thin trunks (marked on the left is a trunk and a fence bar that formed some seam paths).



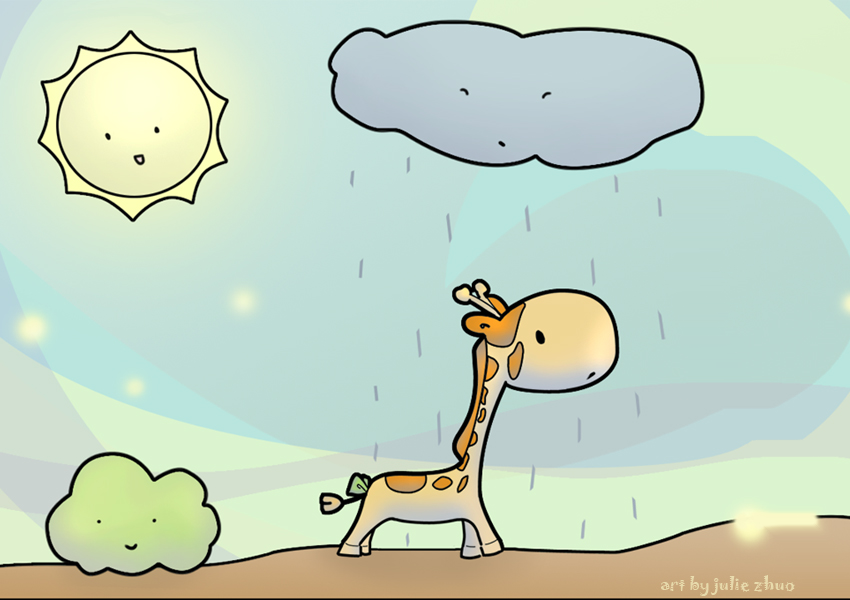
In the same reduction using the Roberts energy function, we spot the following differences:

1. The tree trunks and (most of) the fence bars are intact. There is an exception of one bar which we will discuss.
2. Both horses had more pixels removed from them.

The ‘normalization’ of the energy created more optimal paths for seams to pass through; we can see in the Roberts map that leaves no longer ‘glow’ with energy, which means that the leaves themselves can be distinguished from the space between leaves or leaves surface (which contain less edges than their outline). Using the Roberts energy map, we could use more information related to the content of the image in order to carve. The deformation of one of the fence bar can be associated with its proximity to the black horse.

We have managed to see and demonstrate two facts:

1. Some images responds better the seam carving than other, it depends on the density of the image (object-wise) and the type of the image (drawing, image, picture, etc…).
2. Using an energy function that better fits a specific image can produce better output.



Stretching Artifact:

This is our original image, after adding 50 vertical seams (increasing its width by 50). With cache level set to ‘real-time’, we constantly add the same seams, creating a stretching artifact.