Thesis Notes

Conor Carroll

2019 - 2020

Contents

Part I
Introduction

1 Foreword

This document will be an archive of my notes and projects as I work through the problem of registering, segmenting, interpolating graphical meshes. This work is being advised by Professor Dana Paquin in the math department, as well as Professor Zöe Wood in the computer science department. The notes taken here will eventually culminate into a masters thesis in the math department, as well as a senior project in the computer science department.

When diagnosing an individual, many physicians will have access to a digital template specific to the patient's medical needs. For example, if a patient was experiencing a cranial defect, the physician would be able to reference a computerized model of a skull. When the individuals are themselves imaged, their scans can be incomplete or noisy for a multitude of reasons. These can include hardware defects, user error, or the patient's condition itself. If the hospital is able to register the individuals scan to the template model, then physicians are able to give a personalized diagnosis that can better fit their patient. More, if the imperfections in the patient's scan can be can be mapped to a template model, repair and interpolation becomes possible on the individuals scan. This allows the possibility of more personalized treatment, such as the development of highly customized prosthetics that are patient specific.

This concept expands beyond the medical context. There is also interest in the preservation of historical artifacts by digitalizing them. Unfortunately, many of these artifacts are either not intact or starting to deteriorate. By scanning the incomplete artifact, then associating it with a similar piece at a similar time, perhaps a graphical mesh can be obtained.

The hope is that these notes will be accessible to anyone who desires to read them. However, there will be an assumption of calculus, and I take liberty to assume other prerequisites as the notes are developed. That said, if I don't define a topic or idea, I will do my best to include a pointer to a source that does.

Best,

Conor

Part II
Background Information

2 Computer Graphics

2.1 Modeling geometry

Computer graphics is a sub field of computer science and mathematics that focuses on how computers can model and generate images. An integral part of this, especially in three-dimensional computer graphics, requires being able to effectively model geometry such that the projection onto our computer screen looks realistic. Unfortunately, lots of the geometry we are interested in rendering is continuous and smooth, which is impossible for a computer to exactly replicate. Instead, we model geometry with polygonal meshes that best fit to the outer surface of our geometry. These meshes consist of a single type of polygons, usually triangles, but can be implemented with other n-gons. For our case, we will be working solely with triangle meshes. To house the mesh data, we place all the vertex locations in what is called a vertex array. An index array is then created which encodes the actual triangles in the mesh. Every value in the index array refers to a location in the vertex array, hence a position. Each trio of values in the index array determines a triangle, allowing us to store the triangle mesh.

In practice, we don't just want the vertex information of the mesh, but would also like the normal vectors. Our calculations for lighting and shading depend on knowing the outward direction of our triangle face, so we would like not just a vertex's position, but also it's normal with respect to the surface we are trying to model. While it is possible to approximate the normal vectors at run-time later in the graphics pipeline, the calculated normal can be poor depending on the quality of the mesh. If the geometry is simple, or derived from equations, we like to pre-calculate our normal vectors. The vertex, normal, and index data is then sent to the graphical processing unit (GPU), which is capable of performing geometric and positional transforms on the data in parallel. However, the data is unchanging. The programmer specifies which transforms they wish to perform, and which mesh they would like to perform them on, and the GPU draws the results to the screen, but does not change the underlying vertex and normal data.

It isn't hard to believe that as our meshes get large, the representation of the geometry becomes more realistic. However, this comes at the cost of requiring more memory in the machine, and more computations on the GPU at run-time. There is ample work in the field that focuses on how objects can be represented realistically with a few number of vertices. Solutions range from minimizing unnecessary topological features to wrapping simple meshes with complex textures that hide the lack of vertex information. Unfortunately, the nature of this project doesn't allow us to use the traditional animation techniques to hide simple meshes, so we will have to get creative and be aware of our computational complexity.

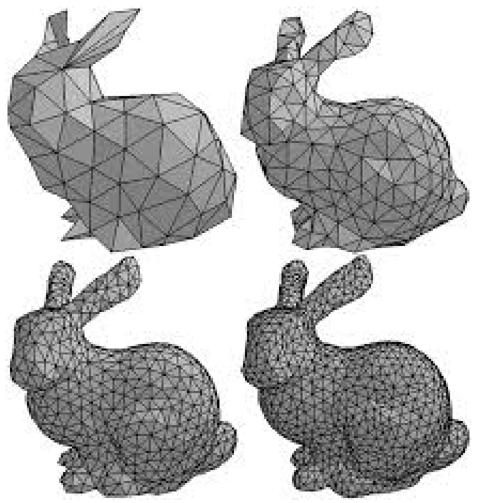

Figure 1: Stanford bunny triangle meshes with increasing numbers of vertices.

2.2 The graphics pipeline

While not a focus of our project, it seems like there would be value in discussing the architecture of the graphics pipeline and how we can use it. The graphics pipeline is a term we use when discussing how a computer renders a scene. It outlines the different stages that a scene undergoes from inception to when it is displayed to a screen. While these stages are usually not accessible, application user interfaces (APIs) such as Direct3D and OpenGL allow programmers to manipulate and control certain aspects of these stages. A note worth mentioning is that the term *graphics pipeline* is somewhat unspecific. It refers to a general framework in which a computer renders a scene, but ultimately depends on the hardware and software of the machine, so it will differ across systems. The example I give will be a simplified overview of the essentials of the pipeline, but be aware that many machines will have more intermediate steps.

The first stage of the pipeline is maneuvering the vertex and normal data. These can be generated by the programmer or read in from an outside source, such as an object file. More importantly, this stage of the pipeline takes place on the central processing unit (CPU), before any data has been sent to the GPU.

The orange boxes in Figure 2 take place on the GPU, and are run entirely in

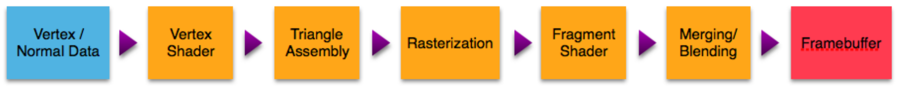

Figure 2: A brief outline of the graphics pipeline

parallel. The vertex shader applies any specified spacial transforms to the vertex data. It dictates the actual spacial setting of the scene. In the triangle assembly stage, the triangles mesh is redrawn with respect to the newly transformed vertex set. These triangles are then projected onto a *screen space*, where the rasterization stage determines which pixels correspond to which triangles. The fragment shader colors the pixels in each projected, triangle fragment. Because the entire process was being carried out in parallel, all the fragments still need to be merged together and blended, which is carried out in the merging stage. What results is a two-dimensional image representing our scene, called a framebuffer, which can be displayed to the screen.

2.3 Marching cubes

We encounter a problem almost immediately as we start our project on registering and interpolating graphical meshes. Our data is volumetric; MRI and CT scans output a series of two-dimensional scans taken at different depths. To be able to draw this data, we need to be able to represent it as a triangular mesh. This means that we have to convert the volumetric data to an array of vertices and an index array that tells us how the mesh fits together. Moreover, we don't have access to normal vectors, so we will have to compute them ourselves. Additionally, when we eventually begin interpolating our meshes, we will generate a set of functions that determine the interpolation. Unfortunately, this isn't immediately useful since it gives us neither vertex nor mesh information. To combat both of these issues, we implement the marching cubes algorithm.

Marching cubes works by taking our coordinate space and discretizing it. We break our domain into a three-dimensional grid that we iterate through. For every box, also called a voxel, we determine whether it's eight corners are either inside or outside whatever surface we wish to represent. If the data is volumetric, we can check the intensity values at each corner and compare them to a predetermined threshold. If the intensity value is above that threshold, we consider the corner outside the surface, and vice versa. In the case that we are given a function, than we redefine it to be a level set of an implicit function. We check each corner to determine if the implicit function's output is positive or negative.

Once we determine if each corner is inside or outside the surface, we can actually observe how our surface moves through the voxel, and generate a mini mesh to represent it. We see that there are $2^8 = 256$ different configurations for a voxel, which in the grand scheme of computing, a manageable number. However, up to symmetry of rotation, there are actually only fourteen distinct

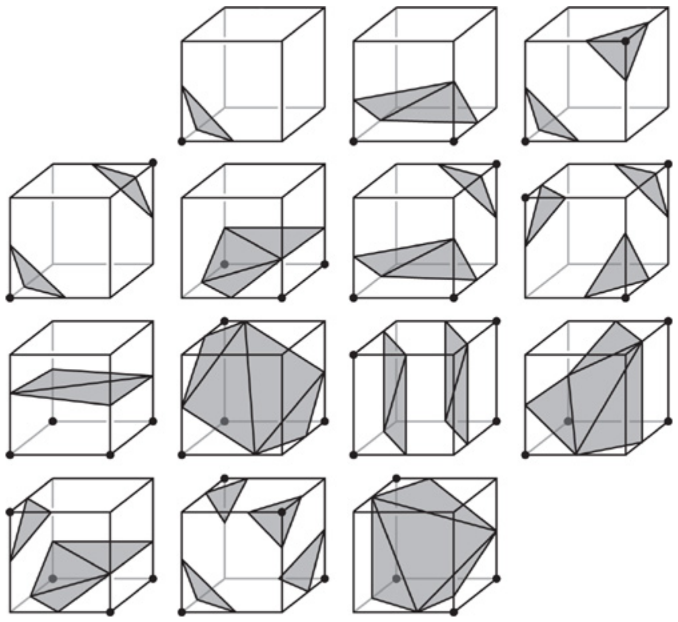

Figure 3: The 14 distinct possibilities for a voxel.

configurations in which the surface intersects the voxel, depicted below. We

then assign an ordering to the vertices of the voxel, and encode that the voxel as an eight bit binary number. The i^{th} bit of our binary number is 1 if and only if the i^{th} corner of the voxel is inside the surface. This number is then used to index into a table that contains the mesh data that we are looking for.

As we do this for every voxel, we piece together the individual meshes to form a total mesh, which we are then able to pass to the GPU to be rendered. We note that the run-time of this algorithm is $O(f(\vec{x})mnl)$, where m is the number of voxels in the x-coordinate, n is the number of voxels in the y-coordinate, and l is the number of voxels in the z-coordinate. We take $O(f(\vec{x}))$ to represent the run-time of the function that checks the status of each corner of the voxel.

We also see that the quality of our mesh increases we increase the number of voxels in our space. Moreover, as the voxel count goes to infinity, our mesh will converge exactly to the surface we are trying to represent.

Below are a few images that we got by graphing the surfaces $f(x,y,z) = x^2+y^2+z^2 = 4$ and $g(x,y,z) = x^2+y^2-z^2 = 4$. Observe the differences as we went from a $32 \times 32 \times 32$ grid to a $8 \times 8 \times 8$ grid.

Algorithm 1: Marching cubes

```
Input  : A grid $G$, and a function $f(\vec{x})$, and a table of mesh
         configurations, $T$
Output : A mesh $M$
M = {};
for voxel $\in G$ do
    index = 0;
    for $i = 0$ to 7 do
        $\vec{c} = i^{th}$ corner of voxel;
        if $f(\vec{c}) < 0)$ then
            index = index $\wedge$ $2^i$;
        end
    end
    M = M + T[index ];
end
return M;
```

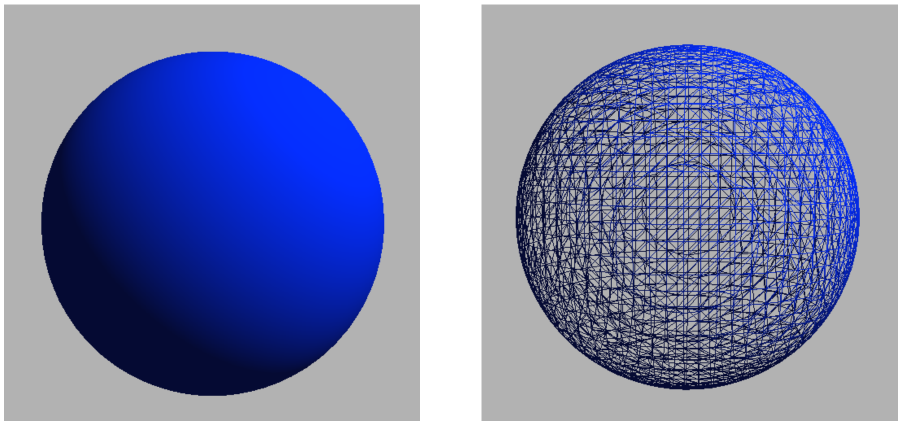

Figure 4: The mesh of f generated by Marching cubes on a $32 \times 32 \times 32$ gridspace.

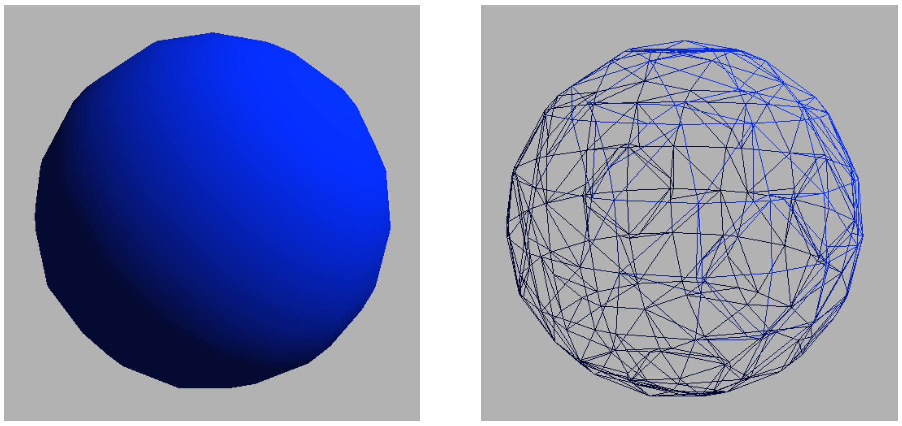

Figure 5: The mesh of f generated by Marching cubes on a $8 \times 8 \times 8$ gridspace.

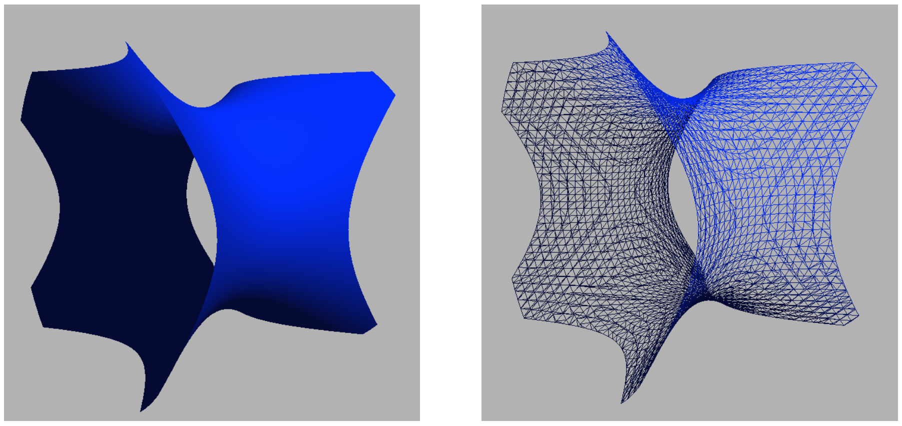

Figure 6: The mesh of g generated by Marching cubes on a $32 \times 32 \times 32$ gridspace.

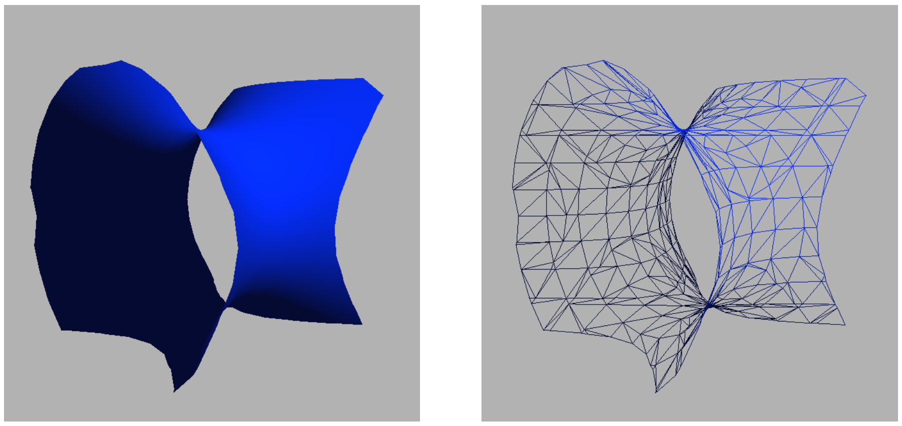
Figure 7: The mesh of g generated by Marching cubes on a $8 \times 8 \times 8$ gridspace.

3 Registration

3.1 Overview

Registration is a term that refers to trying to fit data sets together. If two seemingly different data sets have the same underlying structure, then we would like to be able to map one data set to the other. Let $A, B \subseteq X$ be data sets, and $d : \mathcal{P}(X) \times \mathcal{P}(X) \to [0, \infty)$ be a semi-metric. Then the registration problem can be realized as finding

$$\varphi = \arg\min_{\psi} \Big[d(A, \psi(B)) \Big] \tag{1}$$

where ψ iterates through all function from X to X.

However, this is much easier to formalize mathematically than it is to implement. The set of functions from X to itself will likely be enormous, which means the algorithm for determining φ needs to be strategic as to minimize run-time. Fortunately, context will allow us to make certain assumptions about our data sets that will decrease the number of functions that we need to check.

There are two key types of transformation that we would like to consider: rigid and non-rigid. Rigid transformations only make sense if X is a metric space, as they preserve distance between points. We see rotations, reflections, and translations in this category. Non-rigid rotations encompass all other transforms, such as scaling and basis-splines (b-splines).

3.2 Point-Set Registration

Point set registration refers to when the sets $A, B \subseteq \mathbf{R}^n$ for some n. A and B need not be the same size, which means we could possibly be aligning A to a small portion of B.

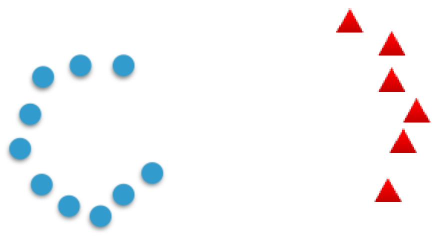

Figure 8: Two distinct data sets, A and B.

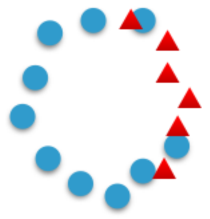

Figure 9: The two data sets A and B after registration.

4 Pattern Recognition and Machine Learning

4.1 Introduction

The crux of pattern recognition and machine learning is trying to predict a new data point from an existing data set. Suppose we have a set $X = (\vec{x}_1, \ldots, \vec{x}_N)^T$ with corresponding outputs $\vec{t} = (t_1, \ldots, t_N)$, then given some new input $\widehat{x}$, we want to accurately predict the corresponding output $\widehat{t}$.

As an example, let X and $\vec{t}$ consist of scalars and we will try to fit our data set to an M-degree polynomial,

$$y = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M.$$

Now this polynomial is currently unknown, but we can consider it a function of it's coefficients $\vec{w} = (w_0, \ldots, w_M)$ and define

$$y(x, \vec{w}) = w_0 + w_1 x + \cdots + w_M x^M,$$

where we try to determine the "best" vector $\vec{w}$. A standard way to determine $\vec{w}$ is to minimize an error function. A common one we see is the *sum of squares* error function,

$$E(\vec{w}) = \frac{1}{2}\sum_{n=1}^{N}\Big(y(x_n, \vec{w}) - t_n\Big)^2.$$

By recognizing $E(\vec{w})$ is positively quadratic in $\vec{w}$, we can solve for the unique global minimum numerically, the most common solution being the Gauss-Newton algorithm. This algorithm is governed by setting the partial derivatives equal to zero.

$$\frac{\partial E(\vec{w})}{\partial w_k} = \sum_{n=1}^{N}\Big((y(x_n, \vec{w}) - t_n)(x_n^k)\Big) = 0$$

Moreover, we are guaranteed an optimal solution for the coefficients, namely w^*, and we want to use $y(x, w^*)$ to be able to predict outputs for new inputs. However, we quickly run into an issue of model choice. What degree polynomial should we use? If we allow the degree to get close the the number of data points, then we have an issue of overfitting; this is where the model exactly fits the data and disregards the underlying pattern. There is no issue in defining a high degree polynomial such that $y(x_k) = t_k$ for all k. To combat this, we introduct a *regulating term*,

$$\tilde{E}(\vec{w}) = \frac{1}{2}\sum_{n=1}^{N}\Big(y(x, \vec{w}) - t_n\Big)^2 + \frac{\lambda}{2}||\vec{w}||^2.$$

The idea is to penalize large coefficients in $\vec{w}$, where the parameter λ governs the relative importance of this term. We will find that this term arises naturally when considering the fitting problem statistically.

4.2 Probability Theory

4.2.1 Introduction

Let X be a random variable for instances $\{x_1, \ldots, _M\}$ and Y be a random variable for instances $\{y_1, \ldots, y_L\}$. Suppose we had N points on a grid that consisted of M columns and L rows. Then we would define $P(X = x_i) = \frac{c_i}{N}$, where c_i is the number of points that fall into columns i. Similarly, $P(Y = y_j) = \frac{r_j}{N}$ where r_j is the number of points that fall into row j. Then the *joint probability* would be $P(X = x_i, Y = y_j) = \frac{n_{ij}}{N}$ where n_{ij} is the number of points that fall into the ij-cell of our grid. For notation convenience we will simply write, $P(x_i, y_j)$.

By visualizing the grid, it is straightforward to derive the *sum rule* or *marginal probability rule*, which states,

$$P(x_i) = \sum_{j=1}^{L} P(x_i, y_j)$$

Now we introduce the notation $P(x_i|y_j)$, which represents conditional probability. Verbally, $P(x_i|y_j)$ is the "probability that $X = x_i$ given that $Y = y_j$." In our example, $P(x_i|y_j) = \frac{n_{ij}}{r_j}$, and in general,

$$P(x_i|y_j) = \frac{P(x_i, y_j)}{P(y_j)}.$$

We can also quickly derive the *product rule of probability*,

$$P(x_i, y_j) = \frac{n_{ij}}{N} = \frac{n_{ij}}{c_i} \times \frac{c_i}{N} = P(y_j|x_i)P(x_i),$$

and similarly,

$$P(x_i, y_j) = P(x_i|y_j)P(y_j).$$

Combining the above, we can derive Baye's Theorem,

$$P(y_j|x_i) = \frac{P(x_i|y_j)P(y_j)}{P(x_i)}$$

Of course, all these properties generalize to non-discrete setting, and we can rewrite the sum rule, product rule, and Baye's Theorem as follows:

- $P(X) = \sum_Y P(X, Y)$
- $P(X, Y) = P(Y|X)P(X)$
- $P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}$

4.2.2 PDF's and CDF's

Briefly mentioned earlier, sometimes our random variables will not be discrete, and in this case it is beneficial to consider a *probability density function* (PDF). Let $p : \mathbb{R}^d \to [0, 1]$ be such that,

$$\int_{\mathbb{R}^d} p(\vec{x})d\vec{x} = 1.$$

Then we say $P(\vec{x} \in A \subseteq \mathbb{R}^d) = \displaystyle\int_A p(\vec{x})d\vec{x}$. All of our familiar rules generalize to the multivariate case as well, consider a PDF of two variables,

$$p(x) = \int p(x, y)dy \text{ and}$$
$$p(x, y) = p(x|y)p(y).$$

4.2.3 Expectation of a function

We can also consider the expectation of a function under a probability distribution. Given p, then the "expection of f" is,

$$\mathbb{E}[f] = \sum_x p(x)f(x)$$

if X is a discrete random variable. If X is a continuous random variable, then

$$\mathbb{E}[f] = \int p(x)f(x)dx.$$

We note that the expected value is a functional. Suppose that we only have a data set $\{x_1, x_2, \dots, x_N\}$, then we can approximate the expected value of a function as

$$\mathbb{E}[f] \simeq \frac{1}{N}\sum_{n=1}^{N} f(x_n)$$

with asymptotic convergence. This intuitively makes sense, as the expected value of a function would be the average of it's outputs. The variance of a function is defined as,

$$\text{var}[f] = \mathbb{E}\Big[(f - \mathbb{E}[f])^2\Big]$$

which we can simplify to,

$$\text{var}[f] = \mathbb{E}[f^2] - \Big(\mathbb{E}[f]\Big)^2.$$

The variance represents how much variability f has about it's expected value. The covariance between two random variables is defined as,

$$\begin{aligned}\text{cov}[x, y] &= \mathbb{E}_{xy}\Big[(x - \mathbb{E}[x])(y - \mathbb{E}[y])\Big] \\ &= \mathbb{E}_{xy}[xy] - \mathbb{E}[x]\mathbb{E}[y].\end{aligned}$$

The covariance expresses the extent that x and y vary together.

5 Inpainting

5.1 Introduction

Inpainting refers to the reconstruction of missing or damaged data, with historical applications being the reparation of art mediums: paintings and sculptures. However, as data has increased in variation and sophistication, inpainting has found a niche in image processing and computer graphics. Today, we see uses in the repair of images, videos, and meshes.

5.2 2-D inpainting

5.3 3-D inpainting

The 3-D inpainting problem consists of a damaged or incomplete model S, where we use H to represent to damaged regions of the model. The goal of the inpainting problem is to fill in the regions given my H in such a way that remains faithful to the geometry and topology of the surface S. While some techniques interpolate solely based on the surface S, many of these require immense amounts of pre processing on depth maps and mapping a hierarchy of landmarks - both of which are algorithmically expensive. A way to circumvent the algorithmic expense is to provide a template surface to S.

We consider our model S to be a set of points in $\mathbb{R}^3$, with a template surface $M \subset \mathbb{R}^3$. In practice, these points exists as meshes in world space (the coordinate system where our virtual world is created), and are potentially skewed in location, size, and orientation. Consequently, we register our images with respect to some estimator function

$$\varphi = \arg\min_{\psi} d\big(\, \psi(M), S\,\big). \tag{2}$$

It is critical to note the immediate reliance on a robust registration algorithm. Point clouds are notoriously difficult to register both quickly and reliably, fortunately we are able to view our meshes in world space. Thus, we can manually orient our meshes, and it is fairly simple to align and scale S and M's bounding boxes. In doing so, we are able to address major impediments modern point-cloud registration algorithms have on accurately computing global transformations.

The actual process of inpainting proceeds from the outside in, starting at the boundary of the hole, ∂H, and working toward the center. A standard approach to this problem would be an iterative approach. If we let $M' = \varphi(M)$, then for a given point $h \in \partial H$, we gather h's corresponding point $k \in M'$ given by

$$k = \arg\min_{x \in M'} ||x - h||. \tag{3}$$

We can estimate damaged points adjacent to h by borrowing the gradient of M' at the point k,

$$h' = h + \alpha \nabla M'(k), \tag{4}$$

where α is a scalar dictating the size of the step and is allowed to be negative in the case that the gradient points away from the damaged region. By iterating this for every point in ∂H, we than create a new boundary consisting of h'-s, and we iterate the process on this new boundary until the hole is filled.

It's fairly clear to see that the above method has between speed and accuracy - for large steps we will fill in the hole quickly, but lose lots of information regarding the surface. Likewise, if we allow small steps, we have a potentially slow algorithm, but would be able to have a much more detailed reparation. We must also question how well the algorithm respects the underlying structure of S, and it doesn't really. In order to recreate S accurately, we are assuming M' is a similar enough that we can just borrow the gradients at corresponding points. With our context of inpainting medical meshes, this doesn't seem like a reasonable assumption; we are likely to see varying curves and surfaces in the geometry among different patients.

A way the literature has combated this limitation is to use M' to create a depth dictionary corresponding to the damaged region H in S. For a given depth, the points in M' corresponding to H are written as a vector