CHAPTER 39

Elementary Multi-view Stereopsis

Structure from motion, as I have described it, deals in interest points. If your pictures show objects with few interest points on them, or if the interest points are hard to match, the reconstructions might be quite sparse. But structure from motion produces relative camera poses and calibrations, which you can use to produce much denser reconstructions from the images. Doing so is often called *multi-view stereo* or, often, *MVS*. There are many different MVS procedures. I have shown some, broken them out by the type of representation of the dense reconstruction they produce.

39.1 EXAMPLE: MODELLING BUILDINGS FROM PHOTOGRAPHS

Imagine you have a small set of photographs of a building which is made up of straightforward parametric shapes. These could be blocks, or wedges, but you could also choose a parametric family that is good for the building you are studying. There could be one shape – say, a basic apartment building might be a cuboid – or a collection of shapes attached to one another, like the complex building of Figure 39.1. You want to show compelling new pictures of this building from new viewpoints.

Structure from motion alone won't achieve this, because the interest points might be too sparse. Look at the point cloud figures of Chapter 38, for example. You should be able to recover a model, by the following argument. Pass the pictures into a structure from motion pipeline. This yields the relative camera configurations and a cloud of interest points. Assume there are not too many reconstruction errors and not too many points that don't lie on the buildings. With enough work on robust fitting, you should expect to recover the model parameters (Figure 39.1). The reconstruction of this figure used a process somewhat different to that I have sketched, which is why there are edges marked on the photographs of Figure 39.1 **exercises**.

39.1.1 View Dependent Texture Mapping

Generally, you want to show objects as they appear, rather than just their geometry. This means you must deal with the texture of the objects. For the moment, assume that the scene just consists of a plane with a pattern on it. You will need to determine what the pattern looks like in any image you generate.

The most straightforward procedure is to construct a reference plane and then compute the texture on that plane. You could, for example, apply a homography to the image texture to transform it to the reference plane **exercises**. Then when you render the object, you compute a homography from the reference plane to the image, and apply it to the pattern. If the object has a more complicated shape, you could – for example – represent it as a collection of small triangles (a mesh; below). You then transform texture from an image to each triangle, and proceed

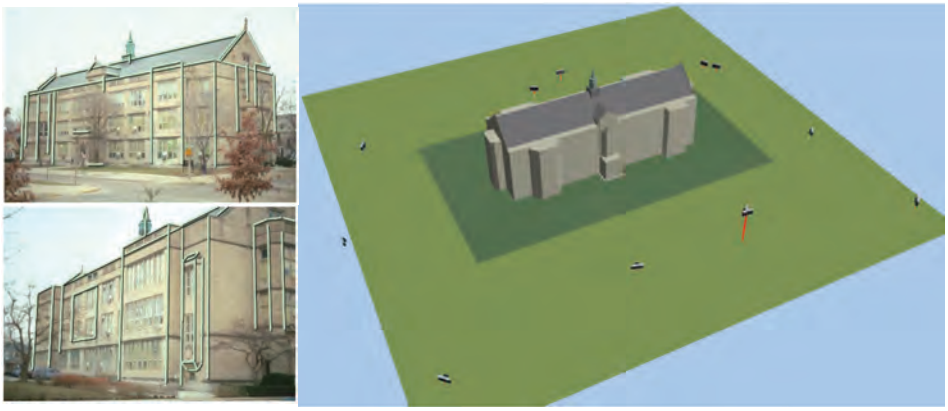


FIGURE 39.1: *On the* **left**, *two photographs of University High School, in Urbana. On the* **right**, *a 3D model of the underlying shape of the building, built using a combination of parametric shapes.* Image credit: *This figure shows parts of Figure 7 and Figure 8 of "Modeling and Rendering Architecture from Photographs: A hybrid geometry- and image-based approach", P.E. Debevec, C.J. Taylor and J. Malik, SIGGRAPH 96, 1996.*

as above. This general strategy is known as *texture mapping*.

It is not a good idea to use a single image to texture map. Doing so can make quite straightforward signal problems very significant. Part of the problem is you do not know the pattern on the plane. If you choose a reference plane and map the image to that, some parts of the image might be quite distorted. This will not be a problem if you view the plane from about the same viewpoint as the image, but might get very serious for other viewpoints. Figure 39.3 illustrates these issues.

For example, imagine you choose a view of the plane from some way away. You wish to move the camera close to the plane. As you get closer, you will need to show the pattern at higher and higher resolution. Eventually, you must run into problems caused by the fact the original picture has very little information about spatial frequencies that are high in the picture you want to generate. Ideally, you would use the pattern in a picture where the camera is close to the plane.

These aliasing problems become particularly difficult when you want to show a picture where the viewing angle has changed. Imagine you choose a view of the plane from a glancing angle, and you want to show a frontal view. You will likely have serious difficulties, because on one side of the view you will have relatively little information about high spatial frequencies and on the other side you will have much more. Ideally, you would use the pattern in a picture where the view angle is about the same as the view angle you want to generate, a procedure known as *view dependent texture mapping*. Typically, view dependent texture mapping blends two or more textures from different views, using blending weights that strongly favor views that are closer to the view being rendered **exercises**. Figure **??** shows the advantages of view dependent texture mapping, which was first used in the context of building reconstruction [].



FIGURE 39.2: *The* **far left** *shows a detail of the model of Figure 39.1, to show the relatively sparse underlying geometry of the model. On* **center left***, this detail rendered using view dependent texture mapping. The texture comes primarily from an image whose view is close to that of the rendered view. As a result, the image shows the effects of small geometric details that aren't present in the underlying model. For example, the windows appear slightly recessed into the wall. On* **center right***, the same geometry shown from a new viewpoint, but with the texture weights the same as center left. The large change in viewing angle has exaggerated the size of the spacing between windows, an aliasing effect. On* **far right***, the second geometry but now rendered using textures close to the view direction. The windows now look like real windows.* Image credit: *This figure shows Figure 13 of "Modeling and Rendering Architecture from Photographs: A hybrid geometry- and image-based approach", P.E. Debevec, C.J. Taylor and J. Malik, SIGGRAPH 96, 1996.*

39.2 PRELIMINARIES

39.2.1 Tasks

How you reconstruct a world depends on what you want to do with it. The key task most representations have in mind is rendering – you want to produce new pictures of the structure that you reconstructed. These pictures will come from any viewpoint, so the task is known as *free viewpoint rendering*.

You may want to change the composition of the scene. For example, you might cut out some part of one reconstruction and place it in another, then render the result. I will use the term *manipulation* for this task.

39.2.2 Representations

There are many ways to represent 3D data. The parametric models of Section 36.3 are no longer in widespread use, because fitting models with high levels of geometric detail tends to involve large numbers of parameters. For this chapter, the options will be per-image depth maps; a point cloud; a set of voxels; and a mesh. I have summarized these representations and their properties below, and you will likely find it useful to memorize this summary. Chapter 36.3 introduces other representations. Each representation makes some operations easy and others hard. A key nuisance in 3D reconstruction is that no representation makes all useful operations easy. Fortunately, it can be quite straightforward to pass from one representa-

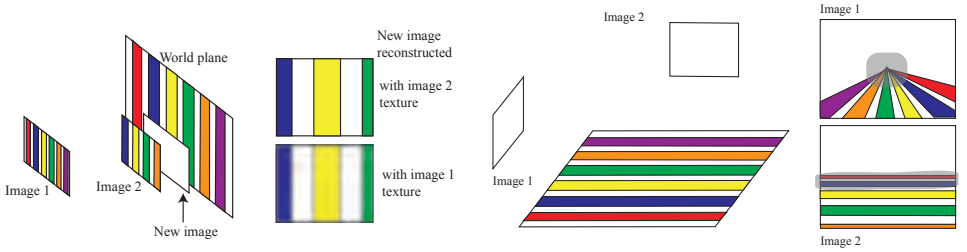


FIGURE 39.3: *It is a bad idea to recover the texture of a surface from a single image. On the* **left**, *two cameras view a textured frontal plane. If you were to, say, use the further view to construct a model of the texture on the plane, when you render a much closer view you will see losses in the high spatial frequencies. As the* **right** *figure shows, different viewing angles lose different details in the texture, so you can't cure this effect by just choosing the view that is closest to the surface. Instead, you should recover the texture from an image whose viewpoint is most like the rendering viewpoint.*

tion to another. MVS pipelines often exploit more than one representation of the reconstruction.

Definition: 39.1 *Per-image depth maps*

A per-image depth map is what it says it is: you associate a depth map with each image in the collection.

Definition: 39.2 *Point clouds*

A point cloud is what it says it is: a collection of points in 3D, usually with a color attached to each point.

Definition: 39.3 *Voxels*

Choose an axis-aligned cuboid in 3D. It is convenient if this is a cube, but that is not required. Subdivide each edge to create a collection of smaller cuboids. It is often convenient that these have the same length on each edge, but that is not required. The result is a *voxel grid* (Figure **??**), and each smaller cuboid is a *cell* or more usually a *voxel* (by analogy with pixel = picture cell, you have voxel = volume cell). Each cell can carry information – so, for example, a label saying whether the cell is `empty` or giving its color. It is usual, and a good idea, to think of a voxel grid as a way to represent a piecewise constant function of 3D.

Definition: 39.4 *Mesh*

A *mesh* is a collection of polygons where many pairs of polygons share edges or vertices. Most meshes are made of triangles, though some are made of quadrilaterals; few meshes use any other polygon. There are many kinds of mesh (Figure **??**). It is common, but not required, to work with *surface meshes*, where the structure imposed by edge and vertex sharing is like a surface. Meshes do not need to be connected, and may have holes in them. It can be useful for a mesh to be *watertight*, which means that it separates space into an inside an outside. Free viewpoint rendering of meshes is easy and extremely fast, with highly adapted hardware available; view dependent texture mapping is difficult; and manipulation is tricky, but numerous procedures are understood. It is quite usual for a mesh to have a normal defined at each vertex. This may appear odd – the mesh is clearly not differentiable at a vertex – but is extremely useful in rendering **exercises**. This normal could be the average of the normals of the polygons sharing the vertex, but there are other ways to obtain normals (below).

Remember this: *Obtaining a set of per-image depth maps from a collection of images is straightforward. The main source of difficulty using per-image depth maps is redundancy: the same surface element is represented in many different images. Free viewpoint rendering of per-image depth maps gets moderate results; view dependent texture mapping is straightforward; and manipulation is tricky. Passing from a per-image depth map to a point cloud is straightforward* **exercises***; to a voxel grid is straightforward* **exercises***; and to a mesh is intricate but well understood* **exercises***, because you can pass to a point cloud and from there to a mesh.*

Remember this: *Obtaining a point cloud from a collection of images requires care. The main source of difficulty using point clouds is that there is no notion of occlusion: points can't occlude one another, because they are too small. Free viewpoint rendering of point clouds gets moderate to poor results ; view dependent texture mapping is very difficult; and manipulation is straightforward. Passing from a point cloud to a per image depth map is tricky* **exercises***; to a voxel grid is straightforward* **exercises***; and to a mesh is intricate but well understood* **exercises**.

Remember this: *Obtaining a voxel grid from a collection of images is straightforward. The main source of difficulty using voxels is managing resolution. Typically voxels are always both too large – when you render them, you see blocky effects – and too small – fine resolution voxel grids contain an awful lot of voxels* **exercises**. *Free viewpoint rendering of voxel grids gets moderate to poor results ; view dependent texture mapping is very difficult; and manipulation is straightforward. Passing from a voxel grid to a per image depth map is tricky* **exercises***; to a point cloud is tricky* **exercises***; and to a mesh is intricate but well understood, though the resulting mesh might not be satisfactory* **exercises**.

Remember this: *Obtaining a mesh from a collection of images is tricky, but understood. The main source of difficulty using meshes is managing the relative complexity of the structure of the mesh. It requires care to maintain which triangles share which edges if you edit or modify a mesh. Free viewpoint rendering of meshes works extremely well for gigantic meshes and has very highly developed hardware support; view dependent texture mapping is tricky; and manipulation is intricate but understood* **exercises**. *Passing from a mesh to a per image depth map is straightforward* **exercises***; to a point cloud is straightforward* **exercises***; and to a voxel grid is straightforward* **exercises**.

39.3 MVS REPRESENTATIONS FROM IMAGE COLLECTIONS

Run a structure from motion pipeline on your images. You now have a sparse point cloud (the reconstructions of the interest points) and the pose and calibration of each camera. You must use this information to come up with a dense representation.

39.3.1 Photometric Consistency

As Chapter 36.3 showed, photometric consistency is powerful if you have two images with known camera poses. There can be ambiguities, however. For example, multiple points along an epipolar line in the second image might have the same intensity a given point in the first image. These ambiguities can be significantly reduced in the multiview stereo case, because you compute photoconsistency using several different images (Figure **??**).

As in the two image case, photoconsistency is often measured in small neighborhoods of a point. Choose a point in the first image and a neighborhood size. In two-view stereo, *if you know the depth at this point*, you can predict where this neighborhood appears in the second image. In multi-view stereo, *if you know the depth at the point*, you can predict where the neighborhood appears in all other images. The strategy will be to search the depths to find the most photoconsistent set of neighborhoods.

Everything is simplified if you assume that this neighborhood, mapped to the other images, has about the same size. This means that the resolution of the images is about the same, they are looking in about the same direction from reasonably nearby points, and so on**exercises**. You now have k neighborhoods, one per image, and must evaluate their photoconsistency.

There are two steps in evaluating photoconsistency: measuring the consistency of each pair of patches; and summarizing the consistency of all pairs. How you measure photoconsistency to some extent depends on what you expect in the images. Assume all images have about the same aperture, exposure time and lighting. This means that the same patch should have the same average intensity in different images. In this case, it is usually enough to measure consistency by computing the SSD (Section 36.3), and summarize by averaging across all pairs **exercises**. SSD is large for bad matches and small for good matches. It is conventional to convert SSD to a score that is small for bad matches and large for good matches by forming $e^{\frac{SSD}{\sigma^2}}$ for some choice of scale. Doing so helps with robustness, too**exercises**.

The same patch could have different intensities in different images. This might happen with significant changes in lighting, aperture or exposure time across images **exercises**. In this case, it is better to use a normalized correlation (Section 36.3) to compare pairs. Doing so essentially compares the spatial structure of local texture patches.

Photoconsistency reasoning has limits. If surfaces aren't diffuse, then the underlying assumption – the same patch will look the same in different images – fails. If you are using a set of images where the lighting varies, the underlying assumption is dubious, but photometric consistency using normalized correlation is often quite well behaved. It will fail disastrously if there is not enough fine scale texture, however – reconstructing large surfaces of uniform texture is notoriously difficult.

39.3.2 Space Carving Voxels

A straightforward procedure to recover a voxel representation is *space carving*. Here is a sketch, that gives the reasoning.



FIGURE 39.4: **Top row** *shows four images of 16 total depicting a gargoyle. The background is deliberately changed from image to image, so that photoconsistency tests segment the object from the background* **exercises**. **Bottom row** *shows renderings of the model recovered using space carving, which occupies some 200, 000 voxels from a total grid of some 5e7 voxels.* Image credit: *This figure shows parts of Figure 5 of "A theory of shape by space carving," by K. N. Kutulakos and S. M. Seitz, ICCV, 1999*

Procedure: 39.1 *Space Carving*

Construct a block of voxels that spans all the geometry of interest, and that has fine enough resolution to be useful. Mark all voxels `unknown`. Now choose a voxel on the outside of this block. Assume that all other voxels are opaque, and check the photometric consistency of this voxel in cameras that can see it. If it is visible in enough images, the photometric consistency estimate should be reliable. If the voxel has the same color in each camera, it is opaque so mark it `opaque`. Otherwise, it is transparent, so remove it. Repeat this until you can remove no more voxels.

The key point is that quite simple tests can identify transparent or empty voxels. In the form I have described, the algorithm is extremely slow. This is because each time you remove a voxel, you expose some new voxels which will require further photoconsistency tests. Making it faster is a question of visiting the voxels in an efficient manner. It is known how to do so [], but even the fast algorithms cannot cope with very large numbers (millions) of images.

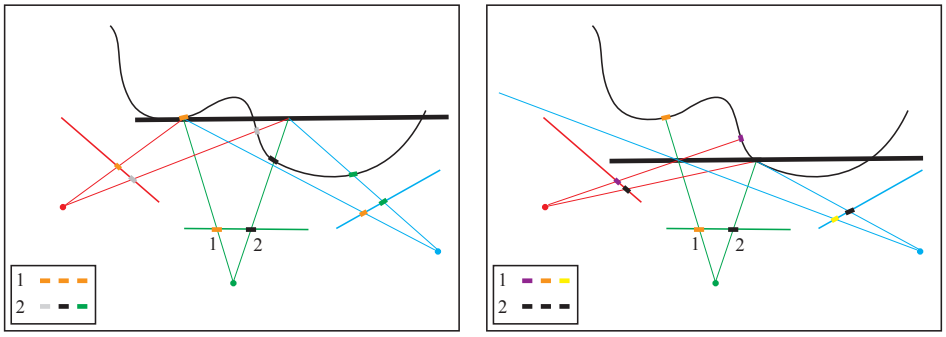


FIGURE 39.5: *A cross-section through three cameras (***red**, **green** *and* **blue***) viewing an object. On the* **left**, *assume that all depths in the* **green** *camera are constant, at the heavy plane. Construct the ray through a pixel in the* **green** *camera to obtain where the ray strikes the plane. I have shown two rays, passing through pixels 1 and 2. You can then determine where these pixels appear in the* **red** *camera and the* **blue** *camera (join the point on the plane with the camera focal point). On the* **left**, *the depth at pixel 1 is the same as the depth to the plane. Each camera then views the same point on the surface, and pixel 1 is photo-consistent between the cameras. However, the depth at pixel 2 is not the same as the depth to the plane. The rays passing through pixel 2 and the* **red** *(resp.* **blue***) focal point strike the surface in different places with different colors. The* **red** *camera sees a gray patch, and the* **blue** *camera sees a green patch. The photo-consistency cue means you can tell that the depth is right for pixel 1. On the* **right**, *the depth at pixel 2 is the same as the depth to the plane, and so it is photo-consistent. Pixel 1 is not. The* **inset** *shows the color of each pixel in* **red**, **green** *and* **blue** *cameras.*

39.3.3 Per Image Depth Maps

One natural representation for a dense reconstruction is a per-image depth map. You associate a depth map with each image in the collection.

Procedure: 39.2 *Building per-image depth maps*

For each camera, choose a small cluster of cameras that is: (a) nearby; (b) pointing in about the same direction; and (c) has reasonable baselines to the original camera. Each pixel in this camera defines a ray in space. Each depth along this ray predicts a pixel location in each of the different cameras, because you know the relative configuration of all cameras. Choose the depth that produces pixel locations with the best photoconsistency.

This should look inefficient to you. A simple trick known as *plane sweep stereo* can produce very efficient estimates of photoconsistency. Figure 39.5 illustrates the underlying geometry. The underlying idea of the figure produces a fast algorithm,

because you can construct a homography that maps all pixels in an image to a plane of constant depth, and another that maps the pattern of the plane to a new image.

Procedure: 39.3 *Plane sweep stereo*

Choose the image whose depth map you want to construct. Build a family of planes parallel to this camera's image plane. These planes represent different depths to that camera. For each such plane, construct the homography $\mathcal{H}_d$ that maps the image texture to the plane, and the homography $\mathcal{H}_i$ that maps the plane texture to each of the other image planes. If the true depth at a pixel corresponds to the depth of a chosen plane, then each of the different $\mathcal{H}_i \circ \mathcal{H}_d$ (which map the image plane to the other image plane) should map that pixel to a point with about the same color. Apply a photoconsistency measure. For each pixel, choose the depth where all the $\mathcal{H}_i \circ \mathcal{H}_d$ produce a set of photoconsistent pixel values.

Choosing the depth with the best average photoconsistency is sometimes known as "winner-takes-all" []. Using averaging to summarize the pairwise neighborhood photoconsistency scores may not be a particularly good idea. Imagine one pair of patches at some depth matches very poorly, and the others match well. The poor match could cause the score at that depth not to be the local maximum, and you will get a depth error. This could happen as a result of local occlusions where some patch is occluded in one of the images, but visible in all others. A robust score will resolve this issue [**?**] **exercises**.

The depth maps you obtain using this procedure may not be consistent. For example, an error may cause a point to be closer to one image than it should be. This point will very likely appear in other images, and in those, it might be at the right depth. Such errors can be caused by local occlusions. It is possible to impose a form of consistency on the depth maps []. To get consistency, every image that sees a point should predict the same location in 3D for that point. This produces serious difficulties with the optimization problem, because every image that sees a given point must communicate in some way. This is hard to achieve with millions of images.

Remember this: *Obtaining a per-image depth map representation from a set of images with known camera matrices is reliably well behaved under most circumstances. Surfaces with strong glossy components or very little texture can present problems. Imposing consistency across the depth maps is difficult on a large scale.*

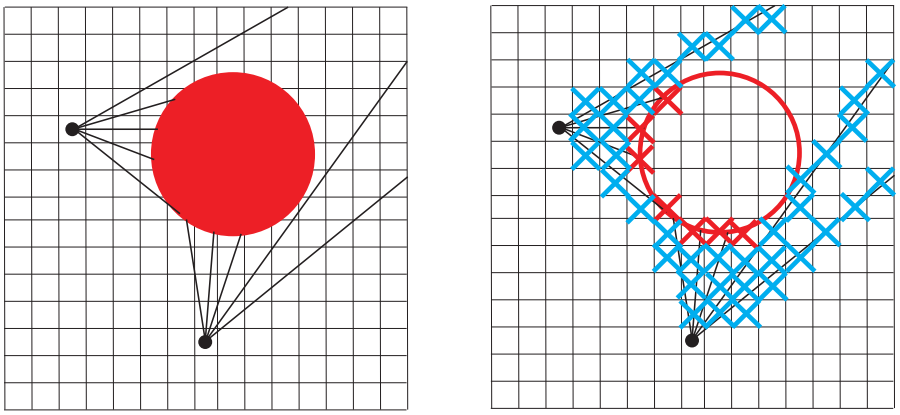


FIGURE 39.6: *Per-image depth maps yield labels for voxels in a grid in two ways. First, every voxel along a ray from the camera center through a pixel to the depth of the closest object must be empty. Second, the voxel at the appropriate depth must be full.* **Left** *shows two cameras in 2D viewing a simple object. The cameras are shown as focal points only, and each camera has five pixels. The object is the red circle. I have worked with a 2D grid because a 3D grid yields an extremely crowded figure.* **Right** *shows the voxel labels that follow. A voxel with a* **red X** *in it is full. A voxel with a* **blue X** *in it is empty. These labels elide a difficult point about sampling, details in text.*

39.3.4 Voxels from Per Image Depth Maps

One way to obtain a consistent 3D reconstruction from a set of per-image depth maps is to use the depth maps to label a set of voxels. Use structure from motion to recover all camera configuration parameters. Now construct a per-image depth map for many (or all) images, discarding the images whose depth maps you can't make or don't trust. Now construct a voxel grid that spans all the cameras and points.

Each voxel starts with an unknown label. The reconstruction is a labelling of voxels as "empty" or "occupied". Choose some pixel in some image. You know the depth to that pixel, and you know the location of the focal point of the camera in 3D. Construct the ray through the focal point and your chosen pixel. Any voxel that lies closer to the focal point along this ray is very likely to be "empty", because you can see through it. There is an outside chance it is "occupied", because the depth might be wrong. Similarly, you have good reason to believe the voxel that lies along the ray at the right depth is "occupied". You can recover many initial labels for voxels from this reasoning (Figure 39.6).

It is entirely possible that two cameras might label the same voxel with different labels **exercises**. At heart, this comes from the fact that voxels represent space in discrete chunks, and is a sampling issue. If a ray from one camera passes close enough to a surface, that camera will label that voxel as "empty". But a different camera might see the surface at an appropriate depth to label it full.

The initial labels are not enough to draw conclusions. Some voxels may remain unlabelled. Many voxels will be seen by more than one camera. If each camera gives a voxel the same initial label, there is nothing to do. But if the initial labels disagree, remove all labels. Now find final labels by solving a discrete optimization problem.

Set up the optimization problem rather like that in Section 33.1.3. Index the voxels by i, j and k. At each voxel, place a variable x_{ijk} which can take the values -1 ("empty") or 1 ("occupied"). The voxel at ijk has a set of neighbors $\mathcal{N}(ijk)$. The neighbors are usually the voxels up, down, left, right, front and back of the voxel at ijk when these exist. Write

$$\phi(x_{ijk}) = \begin{cases} \text{high} & \text{if there is an initial label and } x_{ijk} \text{ disagrees with that label} \\ 0 & \text{otherwise} \end{cases}$$

for a per-voxel cost. Construct a cost $\psi(x_{ijk}, x_{uvw})$ for two voxels x_{ijk} and x_{uvw} where x_{uvw} is a neighbor of x_{ijk}. There are a number of cases. First, the cost $\psi(x_{ijk}, x_{uvw})$ should be high if (a) neither voxel has an initial label and (b) the two voxels disagree with one another (so $x_{ijk}x_{uvw} = -1$). Second, if the two voxels have initial labels that disagree and $x_{ijk}x_{uvw} = -1$ and the x's agree with the initial labels, the cost should be low but not zero. In all other cases, the cost should be zero. Now choose the voxel labels to minimize

$$\sum_{ijk \in \text{voxels}} \phi(x_{ijk}) + \sum_{ijk \in \text{voxels}} \sum_{uvw \in \mathcal{N}(ijk)} \psi(x_{ijk}, x_{uvw}).$$

This optimization problem should not look easy to you. It turns out that polynomial time algorithms are known, because it is cheaper to agree than disagree []. Solutions can produce quite good reconstructions, but have the characteristic problem with voxels. When the voxels are large, the number of voxels is manageable but the solution can be blocky. When the voxels are small, the solution gets better but the number of voxels becomes unmanageable quite quickly. It is possible to attack this difficulty by modifying the optimization problem [].

A better solution is available. You should think of the voxel grid as a way to approximate a function which is positive inside the object and negative outside. The initial labels are constraints on that function. The optimization problem then chooses a function that is "well-behaved" and is also largely consistent with the constraints. With considerable reformulation along these lines (Section 36.3), excellent reconstructions are available.

39.3.5 Obtaining Point Clouds with Patches

Building dense point clouds mirrors building per-image depth maps in important ways. What follows is a sketch of a standard, very widely used procedure, originally due to [].

Use structure from motion to recover all camera configuration parameters. Choose an image, and so a camera. Obtain a small cluster of cameras that is: (a) near the chosen camera; (b) pointing in about the same direction as the chosen camera; and (c) has reasonable baselines to the original camera. Assume you have a rectangle in your chosen image. Associate a depth $c(\mathbf{p})$ and a unit normal $\mathbf{n}(\mathbf{p})$

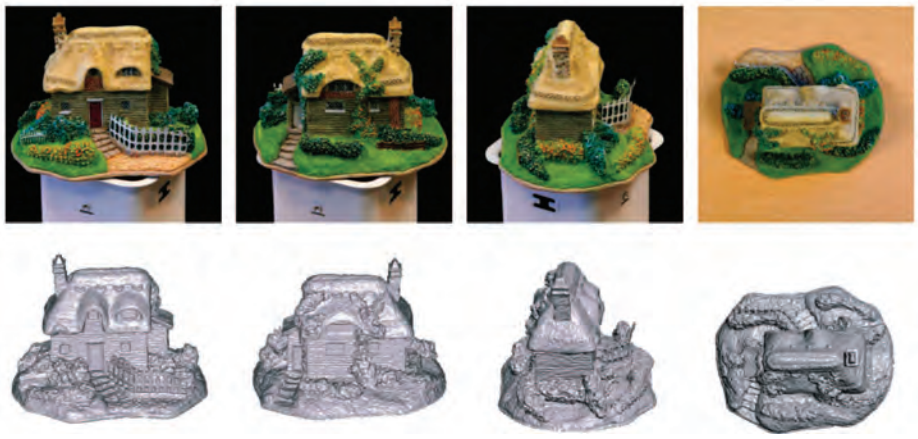


FIGURE 39.7: **Top row** *shows four images showing a plaster model of a cottage. The three images on the* **left** *are part of the reconstruction process; the image on the* **right** *is a new view, not used in reconstruction. Reconstruction uses the global optimization problem. Below the images are renderings of the voxel grid (empty is translucent; full is gray) from the image viewpoints. Note how the reconstruction of the overhead view, which isn't in the original set of images, is consistent with the picture.* Image credit: *This figure shows Figure 1 of "Multiview Stereo via Volumetric Graph-Cuts and Occlusion Robust Photo-Consistency," by G. Vogiatzis, C. Hernandez Esteban, P. H. S. Torr and R. Cipolla, IEEE TPAMI, 2007.*

with the center of the rectangle. You can then think of this rectangle as a 3D *patch* – a flat rectangle in 3D that is tangent to some surface (Figure 39.9). You can project this patch into the other images. In 3D, the patch supplies a small set of points, and you can tell whether these points lie on the object by checking photometric consistency.

Procedure: 39.4 *Building a patch*

Given a location and an image, construct a rectangle of pixels – for example, a 5×5 window – at that location in the image. Use $c(\mathbf{p})$ and $\mathbf{n}(\mathbf{p})$ to project this patch into the other cameras, and compute photometric consistency with those images. Finally, adjust $c(\mathbf{p})$ and $\mathbf{n}(\mathbf{p})$ so that photometric consistency is maximised. If the photometric consistency is sufficiently good, accept the patch.

To get a dense point cloud, you need to have patches overlap and cover the parts of all images you want to reconstruct. A relatively straightforward search expands from one patch to a collection of overlapping patches. Construct a grid for each image, and use this set of grids to keep track of patch overlaps. Label every cell in the grid as untouched.

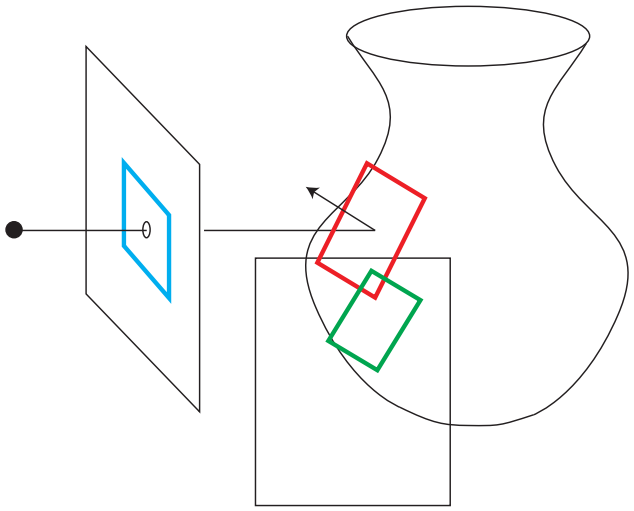


FIGURE 39.8: *If you supply the rectangular patch (***blue***) in the image plane on the* **left** *with a depth for its center, and a normal, you can interpret it as a small rectangle tangent to a surface (***red***) and predict its location in other images (***green***). If there are enough images, and you know their relative configurations, you can choose the depth and normal for the patch using photoconsistency between the predicted patches in each image.*

Procedure: 39.5 *Expanding to a collection of patches*

Expand a patch by building patches that overlap it in either the original image or in any other image where the patch is visible. Find the cell the working patch projects to in each image's grid, and check neighboring cells. For each empty cell, try to build a patch centered on the center of the cell. If this succeeds, record the new patch in the grids. Because the patch is visible in more than one image, it will appear in multiple cells. Keep expanding patches until you have touched every cell.

You can warm start the process of building a patch when you expand, because the patch in the neighboring cell likely overlaps the working patch. This means that the depth and normal for the new patch are likely similar to those for the working patch. Call two patches whose centers are close in 3D and whose normals are similar *neighbors*. Not every patch that results from the expansion process is an accurate reconstruction, as there may be matching errors or occlusion problems. The pool of patches can be filtered with a set of straightforward tests.

Patches that appear in the same cell might occlude one another. They are unlikely to if they are neighbors. If two patches appear in the same cell, but are not neighbors, test whether either patch has weak photometric consistency or is

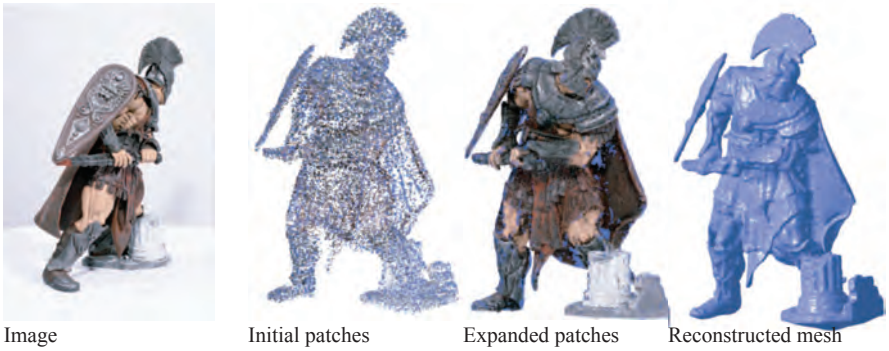


FIGURE 39.9: *On the* **left**, *an image of a model roman (Scaevola, perhaps?).* **Center left** *shows initial patch matches, built around original interest point matches;* **center right** *shows matches obtained by the expansion procedure of Procedure 39.5; and* **right** *shows a mesh obtained using the method of Section 39.3.6.* Image credit: *This figure shows parts of Figure 1 of "Accurate, Dense, and Robust Multi-View Stereopsis", by Y. Furukawa and J. Ponce, IEEE TPAMI, 2010.*

visible from relatively few images **exercises**. If so, discard it. Further, you can render each patch as a polygon and test using a z-buffer whether it is actually visible in a cell **exercises**. If not, discard it. Finally, discard any patch that has few neighbors in surrounding grid cells **exercises**.

Each patch yields a small set of points and each patch has a normal. Attach this normal to each of the points, and you have a point cloud, but where every point has a normal attached. Such points are sometimes called *surfels*. Having surfels is extremely useful if you wish to turn the point cloud into a mesh.

Remember this: *Obtaining a dense point cloud from a from a set of images with known camera matrices is reliably well behaved under most circumstances. Surfaces with strong gloss components or very little texture can present problems.*

39.3.6 Turning Point Clouds into Meshes

Start with a cloud of surfels (points with normals). Assume the points lie near a closed surface (which doesn't have to be connected **exercises**). You can represent this surface with a function $\phi(\mathbf{x})$ that is (say) positive on the inside, negative on the outside, and zero on the boundary. This representation is sometimes known as an *implicit function representation* or an *implicit surface*.

Useful Fact: *An implicit surface represents a closed surface using* $\phi(\mathbf{x}) = 0$ *for some function* ϕ.

There is a standard procedure for turning an implicit function into a mesh, known as *marching cubes*. This will be useful below. All readers should know that that this procedure exists and is extremely widely used in reconstruction tools. Those who need more are referred to [].

Remember this: *Marching cubes is a procedure to turn an implicit surface into a mesh. The procedure intersects the surface with each voxel in a grid. Assuming the grid is fine enough, the intersection between the surface and each voxel falls into a manageable number of classes. For each voxel, the procedure chooses the class of the intersection, then assembles the resulting pieces of surface into a mesh. Strong codes are available.*

One strategy for constructing a surface would be to seek some $\phi(\mathbf{x})$ that vanished on every point. However, because the points may not lie exactly on the surface, it is difficult to use this strategy effectively. Instead, $\phi(\mathbf{x})$ can be efficiently recovered from the normal field if it is sufficiently smooth. If necessary, smooth the normals attached to the points using Gaussian smoothing **exercises**. Write $\mathbf{N}_i$ for the normal associated with the i'th point $\mathbf{x}_i$

A geometric construction is useful here.

Useful Fact: *If* $\mathbf{x}$ *is a point lying on the surface* $\phi(\mathbf{x}) = 0$*, then the outward pointing unit normal to the surface at* $\mathbf{x}$ *is given by*

$$\frac{\nabla\phi}{\|\nabla\phi\|}$$

evaluated at $\mathbf{x}$ **exercises**.

Use the normal estimated at the points to estimate $\phi(\mathbf{x})$. In principal, you can do this by seeking $\phi(\mathbf{x})$ such that

$$\sum_i \|\nabla\phi(\mathbf{x}_i) - \mathbf{N}_i\|^2$$

is minimized. There is no need to normalize $\nabla\phi(\mathbf{x}_i)$ because the cost function tends to force this to be close to a unit vector. However, it is easy to come up with wholly

unhelpful solutions to this problem **exercises**. These unhelpful solutions involve $\phi(\mathbf{x}_i)$ that has large local spikes very close to each point. They can be controlled by requiring that $\phi(\mathbf{x})$ is a sufficiently smooth function.

Once you have $\phi(\mathbf{x})$, it can be turned into a mesh using a procedure known as *marching cubes*, sketched in Section **??**. The resulting procedure accepts a set of surfels and produces a mesh. It is known as *Poisson surface reconstruction*, and is due to [**?**]. All readers should know that this exists and is extremely widely used in reconstruction tools. There are some further details, heavily compressed, below for some readers. Those who need even more are referred to the paper.

Remember this: *Poisson surface reconstruction accepts a set of surfels and produces a mesh representing a surface that lies close to the points and has approximately their normals. It uses the normals attached to each surfel to produce an implicit surface that passes close to the points. The surface is obtained by solving an optimization problem. The implicit surface is then turned into a mesh using marching cubes. The procedure is very widely used in reconstruction tools.*

The function $\phi(\mathbf{x})$ chosen to approximate the surface will be chosen from some finite dimensional function space. You are forced to choose this finite dimensional space, and the details of the choice are very important. The points are close to some surface, and so large volumes of 3D space will have no points in them. This means that controlling the representation of $\phi(\mathbf{x})$ close to the points is more important than controlling it far away from any point. Represent the points with an octtree **exercises**, subdividing cells that have points in them down to some maximum depth **exercises**. The maximum depth is a parameter of the model that you must choose.

Choose a "bump" function $F(\mathbf{x})$ which (a) looks like a Gaussian for a bounded range but then (b) falls to, and stays at, zero **exercises**. Place a scaled and translated copy of this function at each leaf of the octtree, to obtain a collection of functions $f_u(\mathbf{x})$ where u indexes the leaves of the octree. The appropriate space of functions is spanned by these $f_u(\mathbf{x})$, so $\phi(\mathbf{x}) = \sum_u a_u f_u(\mathbf{x})$ for some set of constants a_u. Every $f_u(\mathbf{x})$ is to some points; every point has a $f_u(\mathbf{x})$ nearby; and the resolution of the functions is chosen by the maximum depth of the octtree.

Some work with vector calculus **exercises** establishes that, if ϕ minimizes

$$\sum_i \| \nabla\phi(\mathbf{x}_i) - \mathbf{N}_i \|^2$$

then $\nabla^2\phi$ is "the same" as $\nabla \cdot \mathbf{N}$ in some sense. What "the same" means is quite delicate. You cannot insist these functions are equal, because ϕ is chosen from a finite dimensional space. Instead, the correct criterion is that the difference $\nabla^2\phi - \nabla \cdot \mathbf{N}$, *when projected into working function space*, should be minimized. Equivalently, choose ϕ so that the residual that you can represent using the function space is as small as possible. Solving this problem boils down to solving a linear system whose matrix is quite sparse and quite well structured **exercises**.

Procedure: 39.6 *Passing from a point cloud to a mesh*

First, use Poisson surface reconstruction to produce an implicit function representation. Then use marching cubes to produce a mesh. .

39.3.7 Refining Meshes with Photometric Consistency

You have a mesh built from a point cloud. You would like to adjust the mesh so that it best represents an object or scene that appears in a collection of images. You can do this by moving mesh vertices to improve the photometric consistency. Doing so requires some care with details, because the photometric consistency is expensive to evaluate and can have multiple local minima.

The original mesh is likely quite good, if not perfect, because it was constructed from points that have good photometric consistency. This means the vertices should not move too far from when they started. Efficiency will be a prime consideration computing photometric consistency. The following strategy, due to [], has a good reputation for effectiveness. Write N for the number of vertices, $\hat{\mathbf{v}}_i$ for the i'th vertex of the initial mesh, $\hat{\mathbf{n}}_i$ for the initial normal at that vertex, $\mathbf{v}_i$ for the adjusted position of the vertex and $\mathbf{n}_i$ for the adjusted normal. At each vertex $\hat{\mathbf{v}}_i$, find the d_i such that the photometric consistency of $\hat{\mathbf{v}}_i + d_i\hat{\mathbf{n}}_i$ is maximized (as in Section 39.3.1). Now build a cost that approximates the photometric consistency of the mesh by

$$E_p(\mathbf{v}_1, \ldots, \mathbf{v}_N) = \sum_i (\mathbf{v}_i - \hat{\mathbf{v}}_i + d_i\hat{\mathbf{n}}_i)^T(\mathbf{v}_i - \hat{\mathbf{v}}_i + d_i\hat{\mathbf{n}}_i)$$

(here smaller is better).

Just moving every vertex to minimize this cost is unwise, because some of the d_i might be wrong. Write $\mathcal{N}_i$ for the set of vertices that are neighbors to the i'th vertex (these are vertices that share an edge with vertex i), and $|\mathcal{N}_i|$ for the number of neighbors. At the i'th initial vertex, compute

$$\hat{\mathbf{m}}_i = \frac{\sum_{u \in \mathcal{N}_i} \hat{\mathbf{v}}_u}{|\mathcal{N}_i|}.$$

Now use

$$E_r(\mathbf{v}_1, \ldots, \mathbf{v}_N) = \sum_i (\mathbf{v}_i - \hat{\mathbf{m}}_i)^T(\mathbf{v}_i - \hat{\mathbf{m}}_i)$$

as a cost to discourage $\mathbf{v}_i$ from moving too far from its original position. Notice E_r encourages the mesh to flatten out and to have evenly spaced vertices **exercises**. Finally, obtain the new vertex positions by minimizing

$$E_p(\mathbf{v}_1, \ldots, \mathbf{v}_N) + \lambda E_r(\mathbf{v}_1, \ldots, \mathbf{v}_N)$$

where λ balances photometric consistency and smoothness. Some alternative approaches are explored in the reading list.

39.4 RESOURCES

Resource pointers: 39.1 *Compilations*

Github user walsvid offers a list of MVS tools at `https://github.com/walsvid/Awesome-MVS`. Some of these tools use learned methods (Chapter **??**), others use variants of the methods in this chapter and are called "traditional methods". Yudong Jin offers a list of MVS tools at `https://github.com/krahets/awesome-mvs`.

https://github.com/walsvid/Awesome-MVS

There is a substantial volume of open source software related to structure from motion. There are a number of pipelines, and a variety of toolkits.

Resource pointers: 39.2 *CMVS*

CMVS is a revised version of the original implementation of the methods sketched in Section 39.3.5. The original (PMVS2) was developed by Yasutaka Furukawa under the supervision of Jean Ponce; CMVS was developed by Yasutaka Furukawa under the supervision of Brian Curless and Steve Seitz. You can find CMVS at `https://www.di.ens.fr/cmvs/`.

Resource pointers: 39.3 *VisualSFM*

VisualSFM is a GUI oriented pipeline for SFM reconstruction (though a command line interfaceis available). VisualSFM uses the CMVS code referred to above for patch-based multiview stereo. The author is Changchang Wu. You can find VisualSFM at `http://ccwu.me/vsfm/`

Resource pointers: 39.4 *OpenMVS*

OpenMVS is an open source MVS pipeline at `https://github.com/cdcseacave/openMVS`. The lead author is Dan Cornea [**?**].

Resource pointers: 39.5 *MVE*

MVE is an open source MVS pipeline at `https://github.com/simonfuhrmann/mve`, due to S. Fuhrmann, F. Langguth and M. Goesele [**?**].

Resource pointers: 39.6 *Poisson Surface Reconstruction*

Michael Khazdan publishes open source code for Poisson surface reconstruction at `https://github.com/mkazhdan/PoissonRecon`. There are a variety of other surface reconstruction procedures there, too.

Resource pointers: 39.7 *Marching Cubes*

There are a variety of marching cubes algorithms available. I am in no position to recommend "the best", and likely what is best depends on your system, etc. The following codes have strong reputations.

- scikit-image's implementation, with manual page at `https://scikit-image.org/docs/0.25.x/auto_examples/edges/plot_marching_cubes.html`.
- PyMCubes, by the person with github handle pmneila, J. Matelsky, S. Jennen, and H. Shayani, at `https://github.com/pmneila/PyMCubes`.
- A marching cubes implementation which extends to a marching tetrahedra algorithm, by the person with github handle scrawk, is at `https://github.com/Scrawk/Marching-Cubes`.
- A C++ marching cubes implementation by A. Paris, O. Seiskari, A. Quaglino, N. Kondo and U. Ries is at `https://github.com/aparis69/MarchingCubeCpp`.

Resource pointers: 39.8 *Visualizing and Processing Meshes*

Meshes are interesting in their own right. Meshlab, at `https://www.meshlab.net/#about`, is

> the open source system for processing and editing 3D triangular meshes. It provides a set of tools for editing, cleaning, healing, inspecting, rendering, texturing and converting meshes. It offers features for processing raw data produced by 3D digitization tools/devices and for preparing models for 3D printing.

Meshlab is by G. Ranzuglia, M. Callieri, M. Dellepiane, P. Cignoni, R. Scopigno, and is described in [**?**].

39.4.1 Reading List

-

39.5 YOU SHOULD

39.5.1 remember these facts:

Per image depth maps: useful facts . . . 719
Point clouds: useful facts . . . 719
Voxels: useful facts . . . 719
Meshes: useful facts . . . 719
Building per-image depth maps is reliable . . . 723
Building dense point clouds from calibrated images is reliable . . . 728
An implicit surface is the set of points where a function vanishes . . 729
Marching cubes turns an implicit surface into a mesh . . . 729
The normal to an implicit surface is obtained from the gradient. . . 729
Poisson surface reconstruction produces an implicit surface from surfels 730

39.5.2 remember these procedures:

Space Carving . . . 721
Building per-image depth maps . . . 722
Plane sweep stereo . . . 723
Building a patch . . . 726
Expanding to a collection of patches . . . 727
Passing from a point cloud to a mesh . . . 731

39.5.3 exploit these resources:

Compilations . . . 732
CMVS . . . 732
VisualSFM . . . 732
OpenMVS . . . 732
MVE . . . 733
Poisson Surface Reconstruction . . . 733
Marching Cubes . . . 733
Visualizing and Processing Meshes . . . 734

39.5.4 be able to:

EXERCISES

QUICK CHECKS

LONGER PROBLEMS

PROGRAMMING EXERCISES

39.1. (a)