Reconstructing Roller Coasters

Tyler J. Sellmayer
Stanford University
tsellmay@stanford.edu

Abstract

In this paper, we describe a method of three-dimensional reconstruction whose input is a series of two-dimensional images recorded by a passenger on a roller coaster (a "first-person ride video"), and whose output is a three-dimensional model which approximates the path of the roller coaster's track. We also describe a method for determining the approximate color of the roller coaster track from the same video. We conclude that our methods are irrevocably flawed and cannot be used to achieve our goal of printing scale 3D models of roller coasters.

This builds on previous work in the structure from motion problem, wherein an entire three-dimensional scene is reconstructed from a series of images. We modify this problem by attempting to only reconstruct one element of the scene, the roller coaster's track. We also implement a method of choosing a subset of input frames from a video. Previous SFM work has largely focused on either individually-taken photographs [6, 12] or complete video sequences [2], taking every frame as input.

Our approach relies on a fundamental assumption about first-person ride videos: the camera's path through the world is an approximation of the roller coaster track. In other words, the rider keeps all hands and feet, and his camera, inside the ride at all times. This allows us to use the camera pose location as an approximation of the track location.

1. Introduction

The motivation of this paper is to enable the author to create 3D-printable models of roller coasters automatically. The problem is a modification of the structure-from-motion (SFM) problem as described in [6], simplified because we only need to recover the camera poses for our final output. We do not attempt to recover the 3D world points comprising the roller coaster track itself. To accomplish the camera pose estimation, we rely on MATLAB's structure-from-motion toolkit as described in [12].

We begin by breaking apart a video into individual frames at 30 frames per second of video, using the `avconv` tool [13]. This gives us a collection of frames stored as individual PNG images. Our MATLAB code then reads these images from disk.

We author our own MATLAB code for automated calculation of the color of the track. This color, `track_color_centroid`, is later used to paint the final rendering of points, and in the future could be used to choose an appropriate color for a 3D-printed model of the track.

We extend MATLAB's tutorial code [12] to make use of the high framerate of our input video [3]. Instead of taking each and every frame of video as input, we choose a start point s and desired number of frames n, and introduce a `frameskip` parameter which controls how many video frames we ignore between SFM input frames. We call these ignored frames *inbetweens*. The unignored frames are called *keyframes*.

The SFM pipeline takes these keyframes as input. For each keyframe, it corrects for image distortion (using a manually-tuned set of camera parameters), detects SURF features [1], finds feature correspondences with the previous frame, and uses these correspondences to estimate the fundamental matrix [6, p. 284] between the previous frame and current frame.

This estimated fundamental matrix is then used to compute the relative camera pose adjustment between frames, which is used to update a list of world-coordinate camera poses. After each new frame is successfully processed in this manner, we perform bundle adjustment [15] to increase the quality of our camera poses. Once we have processed n frames, we report the list of camera pose locations in world coordinates and plot them. The plot points are colored with the RGB value from `track_color_centroid` and displayed.

To enhance our results, we enable our algorithm to substitute a keyframe with a nearby inbetween frame when this fundamental matrix estimation fails for any reason, trying up to `frameskip` inbetweens.

In our experimental results, we report plot of camera poses generated under a variety of SURF parameters and

Figure 1. Frame number 3070 from a first-person ride video [3], unedited.

Figure 2. Frame number 2784 from a first-person ride video [3], unedited. Here, the rider is inside a dark tunnel.

`frameskip` values. We then draw conclusions from these results.

2. Problem Statement

Our problem has two independent pieces: estimating the roller coaster track's color, and estimating a three-dimensional model of the roller coaster track's path. We examine these problems separately.

2.1. Estimating Track Color

This problem can be concisely stated as "Given a first-person ride video of a roller coaster, return the RGB value which most closely approximates the paint color of the roller coaster's track."

First-person ride videos have the property that the image of the track always touches the bottom of the frame near the center, as shown in 1. This property is only untrue in cases where the camera is not pointing forward along the track (we found no examples of this) or when the track is not fully visible. For example, the track is not visible at the bottom of the frame when the camera's automatic white balance adjustment causes it to be blacked out (or whited out) in response to changing environment light, as seen in figure 2.

Using this mostly-true property, we can conclude that ideally, the track color will be approximately that color which appears most often in the bottom-center area of our video frames. But our images are noisy, and the lighting changes throughout the video, so this ideal scenario doesn't quite work if we use the pixel colors directly from the recorded images. Instead we bucket these pixel colors into a color palette using nearest-neighbor search [11], then find the palette color whose member pixels occur most often in the bottom-center of the frame. This color we call our track color. The full explanation of this algorithm is in the Technical Content section 3.

2.2. Estimating Track Structure

As stated above, we use the locations in world space of our camera as an approximation of the track position. This lets us use the camera pose estimation stage of structure-from-motion [12] as the basis of our algorithm. We mark every `frameskip`th frame as a *keyframe* with the frames between them called *inbetweens*.

Before processing any frame, we first undistort it using manually-tuned camera parameters for correcting fisheye distortion and a calculated intrinsic camera matrix K. To obtain the approximate focal length of our camera, we compute the average width-at-widest-point of the roller coaster track's image in pixels across a random subset of frames, and use this average width to obtain a ratio between a width in world-space (the real width of the track, which we assume to be 48 inches) and a width in the image plane in pixels. We use this ratio to convert the known focal length of our camera (14mm, according to [5]) to pixels. We assume square pixels and zero skew, so this focal length is all we need to compute our intrinsic camera matrix K.

For each keyframe, we attempt to compute the camera pose relative to that of the previous frame. MATLAB's pose estimation toolkit assumes that the camera poses are exactly 1 unit distance apart, an assumption which is corrected by performing bundle adjustment [15] after each frame's pose is added.

During the computation of the relative camera pose, MATLAB's `helperEstimateRelativePose` function [12] attempts to estimate the fundamental matrix F [6, p. 284]. This estimation can throw an exception when there are not enough matching correspondence points to complete the eight-point algorithm, or when there is no fundamental matrix found which creates enough epipolar inliers to meet the requirement set by our `MetricThreshold` parameter. When an exception occurs, we do not want to simply stop calculating. Instead, we make use of the inbetween frames, retrying the relative camera pose computation with each inbetween frame after our second keyframe until we find one that succeeds, or until we run into the next keyframe, whichever comes first. If we run out of inbetweens without successfully computing a relative camera pose, we terminate our SFM computations immediately and return a partial result. In our full results (see section 6)

we report the mean and maximum numbers of unsuccessful fundamental matrix computations per keyframe for each of our experimental runs.

SFM requires feature correspondences for the fundamental matrix calculation [6], which requires features. We choose to use SURF [1] as our feature detection algorithm because it provides scale- and rotation-invariant features. This is necessary because roller coasters often rotate the rider relative to the environment (which rotates the projected images of features in our scene between frames), and because the camera moves forward along the z-axis between frames which changes the scale of the projected images of features in our scene. In our results we report experiments with controlling the SURF parameters `NumOctaves`, `NumScaleLevels`, and `MetricThreshold` as defined in [7].

To avoid needless reimplementation of past work, we use MATLAB's built-in toolkit [12] for computing correspondences between features, estimating camera poses, tracking views, triangulating 3D world points, and performing bundle adjustment [15]. Together, these produce a final set of camera poses, including camera location and orientation in world-space. We then plot the camera locations and color our plot using the RGB value of the calculated track color.

3. Technical Content

3.1. Splitting Video Into Individual Frames

Our video is downloaded from YouTube [3]. We run the following command to split it into its individual PNG format images at a rate of 30 frames per second of video [13]:

```
$ avconv -i video.mp4 -r 30 -f image2 \
  output_dir/%05d.png
```

We manually select the range of frames $[f_1, \ldots, f_e]$ from the video which comprises the first-person ride video, excluding the copyright notice at the beginning and the credits at the end.

3.2. Calculating Track Color

3.2.1 Determining The Color Palette

We first decide on a palette size. For our experiments, we use palette size 10, meaning we will calculate 10 centroids in the RGB space.

Our code examines a random subset of t frames $[s_1, \ldots, s_t] \subset [f_1, \ldots, f_e]$, and takes a random subset of q pixels in each frame $[p_{1,1}, \ldots, p_{q,1}, p_{1,2}, \ldots, p_{q,2}, \ldots, p_{q,t}]$, where each pixel $p_{i,j}$ is represented as a triplet of values between 0 and 255, indicating the red, green, and blue values comprising the color of that pixel, respectively. This is a standard representation of colors in RGB space.

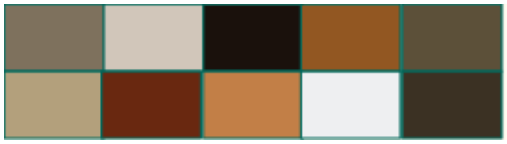


Figure 3. Ten color centroids calculated from a random subset of pixels in [3]. Notice that these colors are similar to those found in figure 1.

We take this set of several thousand pixels, and run k-means clustering [10] on it. This gives us k centroids in RGB space, and we use the colors those centroids represent as our color palette. Because of the randomness in this algorithm, we do not get the exact same palette every time. One example of a $k = 10$ color palette is seen in seen in figure 3.

3.2.2 Finding The Track

Once we have established our color palette, we need to determine which of the colors in the palette most closely approximates the color of our track. To accomplish this, we must rely on our knowledge that in first-person ride videos, the roller coaster track usually touches the bottom of the image frame, near the center, and almost never touches the left or right sides of the frame.

We first select a new random subset of t frames $[r_1, \ldots, r_t] \subset [f_1, \ldots, f_e]$. In each frame, we examine only the bottom 10 rows of pixels. We split this 10-pixel-high strip horizontally into g 10-pixel-high segments. For an image of width W, this gives us g regions $[\gamma_1, \ldots, \gamma_g]$ each of size $10 \times \frac{W}{g}$. Our ultimate goal with these regions is to find which palette color is least often present in the left- and right-most regions.

Rather than just counting every pixel, we choose to count only those pixels which lie on either side of an edge. This increases the number of pixels we count that represent track (which is made of hard-edged steel parts, in focus, and relatively large in the frame, giving it more sharp edges) compared to the number of pixels we count in noisy background regions (which tend to be out of focus, motion-blurred, or so far away that their edges are not distinguishable at the camera's resolution, giving them few sharp edges). We count the pixels $(e_x - 1, e_y), (e_x + 1, e_y)$ which lie on either side of the edge, rather than the pixel (e_x, e_y) which lies directly on the edge, because we want to capture the colors *inside* the regions more than we want to capture the colors of the edges themselves. We call the set of points $(e_{i,x} - 1, e_{i,y}), (e_{i,x} + 1, e_{i,y})$ for all edge pixels e_i our set of *half-edge pixels*.

Furthermore, because we care about hue more than saturation or value when determining which pixels to count, we perform the edge-finding computation on the 'hue' layer of

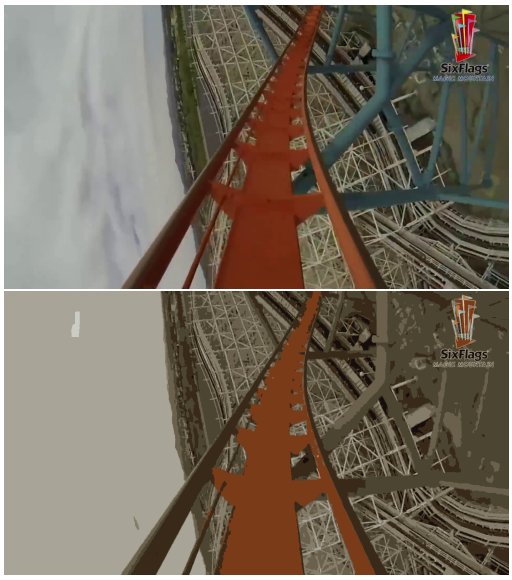


Figure 4. Top: Frame number 3854 from a first-person ride video [3], unedited. Bottom: the same image with each pixel's color replaced by its nearest RGB-space neighbor from the color palette in figure 3.

Table 1. A histogram of half-edge pixel counts over 5 regions and 10 colors.

1612	163	15	14	244	117	72	0	59	818
471	47	276	18	253	112	41	5	0	433
683	0	70	55	32	285	142	124	0	251
182	0	168	1	281	104	0	26	45	441
312	141	74	0	94	183	18	0	14	663

our images after converting them to HSV (hue-saturation-value) format. We use MATLAB's `edge` function to accomplish this [8].

We use a nearest-neighbors search [11] to bucket our half-edge pixels' RGB values into our color palette. To illustrate this concept, we provide figure 4 where every pixel in the image has been replaced with its bucketed color.

We then count the number of half-edge pixels in each color palette bucket, in each segment $[\gamma_1, \ldots, \gamma_g]$. This gives us a two-dimensional histogram where each cell represents the total number of half-edge pixels of one particular color c_i in one particular region γ_j across all the frames in $[r_1, \ldots, r_t]$. Table 1 gives an example of one such histogram. From this histogram we can find a color column which most closely matches the pattern we expect for our track color. As stated above, we expect the track color to appear almost-only in the center region(s), and almost-never in the far left and far right regions. We get good results simply choosing the color whose far-left region and far-right region counts have the minimum sum. In table 1 we see that the column $[0, 5, 124, 26, 0]^T$ satisfies this condition, so our track color is the one corresponding to that column. In this example, that color happens to be the reddish-orange palette color which covers the bulk of the track in figure 4.

3.3. Calculating Track Width

Now that we know the color palette and the track color, we can examine yet another random subset of frames and compute the average width (in pixels) of our image of the track. This will allow us to scale our model appropriately by adjusting the focal length parameter in our camera matrix K. We again examine only the bottom 10 rows of pixels of each frame. In each row of pixels, we find the leftmost and rightmost pixel whose color lands in the track-color bucket, and consider the distance between these two pixels to be the track width at that row. We simply take the mean of these track widths for each of the 10 bottom rows in each image as our average track width.

3.4. Camera Pose Estimation

The camera pose estimation task is the first part of the Structure-From-Motion problem as described in [12]. We operate on grayscale version of our frames, color is not important for this part. We use a manually-tuned parameter for correcting the radial distortion caused by the GoPro's fish-eye lens effect, giving us less distorted frames, like the one seen in figure 5.

We use SURF [1] to detect features as seen in figure 6. We modify the parameters `NumOctaves`, `NumScaleLevels`, and `MetricThreshold` as defined in [7] in our experiments. SURF provides rotation- and scale-invariant features, which is useful to us because we want to find feature correspondences between frames in an environment where our camera is rotating and moving through space (because it's on a roller coaster!). As described in the Problem Statement 2 section, we use MATLAB's toolkit to accomplish fundamental matrix estimation, triangulation, and bundle adjustment. We extend the tutorial code [12] to make use of the relatively large number of frames available to us.

We begin our SFM camera pose update loop with the intention of operating only on every `frameskip`th frame $[f_1, f_{1+frameskip}, f_{1+2*frameskip}, ...]$, called our *keyframes*. At each step in the loop, we first find feature correspondences between the next keyframe and the current last-used frame. Figure 7 shows one such set of correspondences. When these feature correspondences provide enough high-quality points for us to estimate the fundamental matrix and meet our threshold for epipolar inlier count, the calculation succeeds and our loop continues on to the next keyframe. When the feature correspondences do not meet this requirement, or if MATLAB's `estimateFundamentalMatrix` function [9] fails for

Figure 5. Above: Frame number 839 in [3], unedited. Below: The undistorted frame.

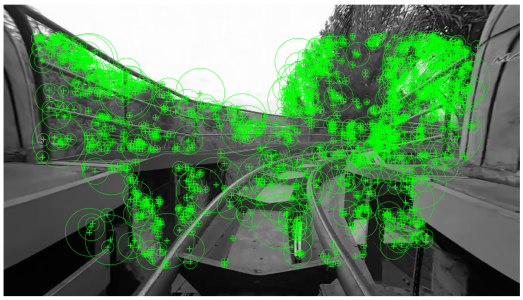


Figure 6. SURF features in frame 91.

any other reason, we re-attempt the calculation with an *inbetween* instead of the failed keyframe. We try subsequent inbetween frames in the same order they appear in the video.

For example when `frameskip` $= 20$ the algorithm will first attempt to compute the fundamental matrix between frames 1 and 21. If this computation fails, we try the computation again between frames 1 and 22, then 1 and 23, and so on, until the computation succeeds, or until we reach frame 41 (which is the next keyframe) and terminate. Examples of feature correspondences which led to a failed fundamental matrix calculation can be seen in figure 8.

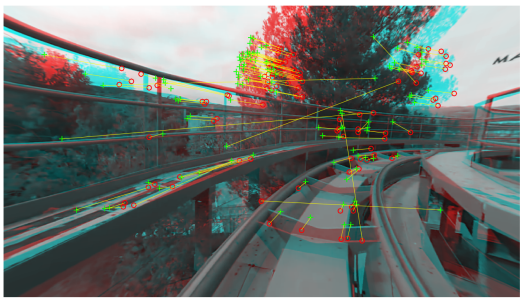
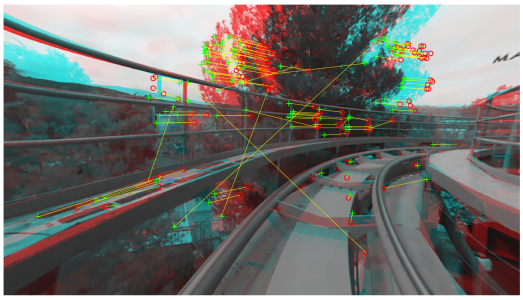


Figure 7. SURF feature correspondences between frames 192 (red) and 212 (blue).



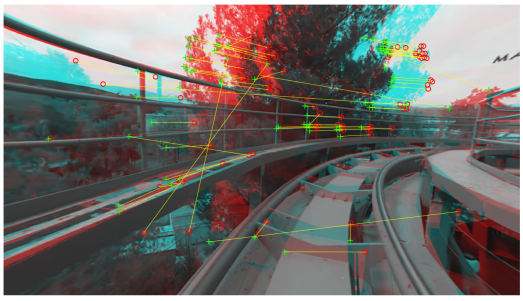


Figure 8. Top: SURF feature correspondences between frames 212 (red) and 232 (blue). Bottom: SURF feature correspondences between frames 212 (red) and 241 (blue). Both of these sets of features lead to a failed fundamental matrix estimate calculation. Notice the yellow lines which are much longer than the distance between the image centers - these are incorrect correspondences. Having so many of these creates the situation where `estimateFundamentalMatrix` finds only fundamental matrices that do not meet the threshold for number of epipolar inliers.

3.5. Output

Our output is a set of points, plotted in a 3D view, colored to match our calculated track color. Each point represents the estimated camera pose location from one frame of video. An example plot is given in figure 9.

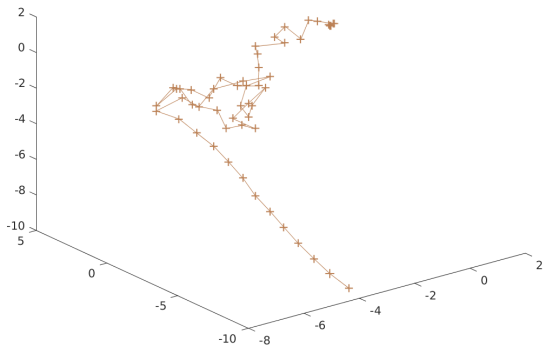


Figure 9. Plot of 53 estimated camera pose locations, generated with `frameskip` $= 16$, `NumOctaves` $= 4$, `NumScaleLevels` $= 3$, `MetricThreshold` $= 1100.0$.

4. Experimental Setup and Results

4.1. Implementation And Setup

For all our experiments, we operate on the same set of images from [3]. This data set is roughly 14GB, and we're running our code on Stanford's corn servers, so this uses 99% of our filesystem quota. We did not have a computer with enough available storage to hold another data set.

All our code is implemented in MATLAB. We implement the following functions:

- `main.m` – This is the main function of our code. Responsible for generating figures 4, 5 and all of the camera pose plots, as well as running all other functions described below. Controls which image files are used as input to the other functions. Defines the experiment parameter `TEST_FRAMESKIP`, the desired number of inbetween frames between keyframes (also referred to as `frameskip` above.)

- `random_subset_images.m` – Used by `main.m` to select random subsets of image paths, which are then passed to `cluster_colors`, `track_color`, and `average_track_width`.

- `cluster_colors.m` – Determines the color palette of a set of images using k-means [10], as described in section 3.2.1. Defines the experiment parameter `NUM_COLORS`, the number of color centroids in the palette. Returns a `NUM_COLORS` $\times$ 3 matrix of RGB values, with each row representing one palette color.

- `track_color.m` – Determines which color centroid from the output of `cluster_colors` best approximates the color of the roller coaster track, as described in section 3.2.2. Defines the experiment parameter `BOTTOM_STRIP_SEGMENTS`.

- `average_track_width.m` – Determines the width of the track, as described in section 3.3.

- `sfm.m` – Computes camera poses using the SFM pipeline as described in section 3.4. Defines the experiment parameters `NUM_OCTAVES`, `NUM_SCALE_LEVELS`, and `METRIC_THRESHOLD`.

For our color experiments, we run our `main` MATLAB function once per experiment, altering either the `NUM_COLORS` parameter in `cluster_colors.m` or the `BOTTOM_STRIP_SEGMENTS` parameter in `track_color.m` for each experiment. We record the output of `main`, which includes the list of color centroids, the histogram of pixels-per-centroid-per-segment, and the final track color.

For our SFM experiments, we run the same MATLAB function (`main`) once per experiment, changing one parameter out of `NumOctaves`, `NumScaleLevels`, `MetricThreshold`, and `TEST_FRAMESKIP` each time. We conduct testing in an exploratory manner, altering only one parameter at a time, searching for a configuration which gives us the longest possible series of camera poses before running into the failure condition of being unable to calculate a fundamental matrix for a keyframe or any of its subsequent inbetweens.

4.2. Results

4.2.1 Track Color Estimation

We experimented with computing the track color for various values of `NUM_COLORS` and `BOTTOM_STRIP_SEGMENTS`. These results are so uninteresting - the track color always comes out as a shade of orange (as seen in figures 1, 3, and 4) or nearly white (the color of the sky in those same figures) - that it is wholly unnecessary to present more than one full example. We present this example in tables 2 and 3.

We also present a table of track color centroids as calculated under different parameters in table 4. Remember that these color coordinates are on a scale of 0-255 with $(0, 0, 0)$ representing pure black and $(255, 255, 255)$ representing pure white.

4.2.2 Camera Pose Plots

We present a representative subset of our camera pose estimation results. Figures 9, 10, 11, 12, and 13 show a variety of camera pose location plots. Note that the long, straight path of points in each plot corresponds to the long, straight lift hill of the roller coaster. The jumbles of points are badly-reconstructed camera locations on and after the peak of the lift hill, or during the initial turn out of the ride shelter

Table 2. A histogram over 5 segments and 10 colors.

902	1	153	87	4	191	775	3	467	428
454	4	45	206	7	146	344	98	293	721
429	4	20	114	6	492	172	7	203	1084
375	6	60	233	0	333	0	0	103	1131
918	0	148	338	0	334	0	0	97	730

Table 3. The color centroids for each of the 10 colors in 2.

Red	Green	Blue
81.1008	76.3207	57.9355
187.2241	128.2615	83.6437
72.8899	45.4787	24.6535
244.1506	242.3728	241.9210
171.5434	90.5595	36.1833
127.8087	105.3556	84.7148
36.8842	28.1582	19.3491
177.7524	181.0321	182.5577
184.3315	152.2799	126.4728
129.8242	55.6536	23.2184

Table 4. Estimated track colors.

#colors	#segments	Red	Green	Blue
3	3	161.7516	117.9050	85.0151
5	3	243.3168	241.6485	241.0941
7	3	243.2237	242.1020	242.0921
14	3	244.6650	243.3350	243.2677
3	5	161.7516	117.9050	85.0151
5	5	243.3168	241.6485	241.0941
7	5	142.3456	67.5242	28.2738
14	5	244.6650	243.3350	243.2677

area. Only figure 13 represents our coaster somewhat accurately - watch the video in [14] and observe the presence of the roller coaster's second hill. We were unable to find any configuration of our code which could reconstruct past frame 2689 of the video [3] without hitting the failure condition of being unable to find an inbetween that allows for successful fundamental matrix estimation. For full results, see the link in section 6.

5. Conclusions

5.1. Color Extraction

Our relatively low level of success in determining track color (see table 4, often the track color is determined to be white, which is actually the color of the sky in our video) suggests that our goal was not achieved, and our method was too reliant on manual tuning of the `NUM_COLORS` and `BOTTOM_STRIP_SEGMENTS` parameters. We suggest that future work on this topic should ignore our results and use better, existing image segmentation algorithms like the ones found in [4]. Should anyone choose to use our method, we offer some suggestions and conclusions about our results.

In section 2.1, we introduced a method for extracting the

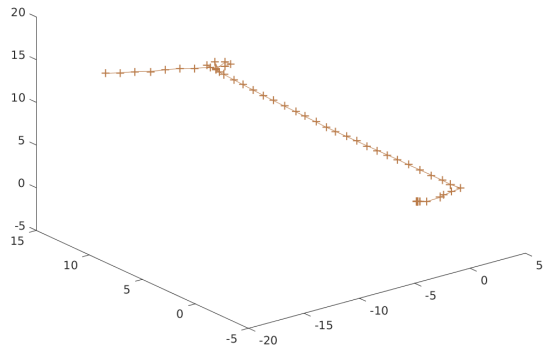


Figure 10. Plot of 62 estimated camera pose locations (between frames 91 and 249) in [3], generated with `frameskip` $= 4$, `NumOctaves` $= 4$, `NumScaleLevels` $= 3$, `MetricThreshold` $= 900.0$.

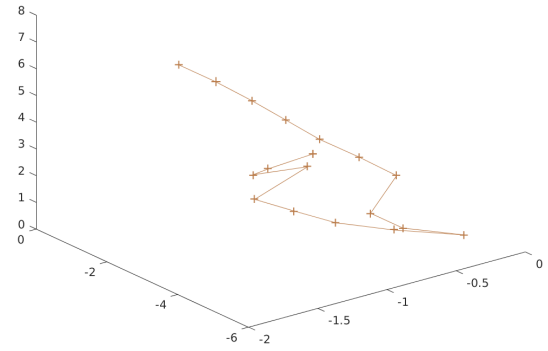


Figure 11. Plot of 19 estimated camera pose locations (between frames 91 and 153) in [3], generated with `frameskip` $= 8$, `NumOctaves` $= 4$, `NumScaleLevels` $= 3$, `MetricThreshold` $= 900.0$.

color of a roller coaster track from a first-person ride video. This method is extensible to other color extraction problems, where the general region that contains an object is known across multiple views. For instance, one might extract a person's eye color by overlaying a grid on the image, then creating a 2D histogram similar to table 1 with each row referring to a single cell in the grid. An existing face-detection algorithm would be used to determine the general location of the eye in each frame. After doing the nearest-neighbors color palette bucketing and counting half-edge pixels to create the histogram, whichever color was present near the center of the eye, but not present at all in the bound-

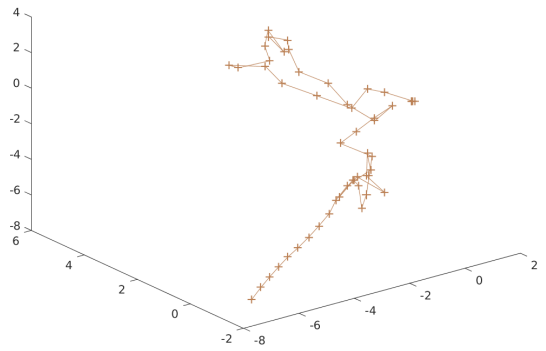


Figure 12. Plot of 52 estimated camera pose locations (between frames 91 and 833) in [3], generated with `frameskip` $= 16$, `NumOctaves` $= 4$, `NumScaleLevels` $= 3$, `MetricThreshold` $= 900.0$.

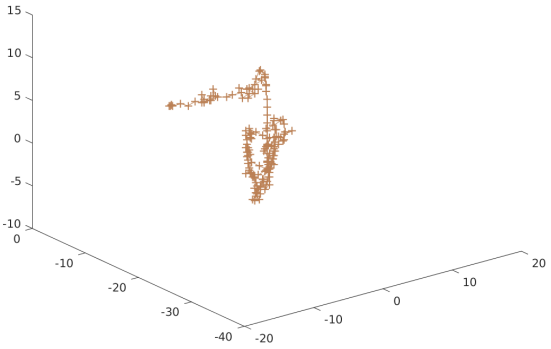


Figure 13. Plot of 165 estimated camera pose locations (between frames 91 and 2641) in [3], generated with `frameskip` $= 16$, `NumOctaves` $= 4$, `NumScaleLevels` $= 3$, `MetricThreshold` $= 2000.0$. A video of these points is available in [14].

ary regions around the eye, would most likely represent the closest approximation of the person's eye color. Further refinements could be made by detecting circles to more accurately locate the iris and pupil.

For our specific roller coaster problem, our goal was to find a close approximation to our track color. Here we define "close approximation" in the context of our original goal, which was to create 3D-printed models of roller coasters, so we only need to approximate up to the color resolution of available 3D-printing filaments. In general this means we only need to be able to accurately distinguish *orange* from *red* or *brown* or any other primary or secondary color, and we have achieved this level of accuracy in this paper, but only by manually tuning the `NUM_COLORS` and `BOTTOM_STRIP_SEGMENTS` parameters until we got the desired result. This is less useful than simply picking the color manually.

5.2. Camera Pose Estimation

In section 3.4 we describe our approach to reconstructing camera poses for a series of frames taken from a video. Our experimentation with modifying the `frameskip` parameter reveals that `frameskip` $= 18$ gives the longest reconstructable sequences, but the resulting plot looks far less smooth than with `frameskip` $= 16$. For our ultimate goal of creating 3D-printed models of the track, we want a smooth-looking plot, so we chose `frameskip` $= 16$ and explored the behavior when other parameters are modified.

Increasing SURF's `NumOctaves` parameter allows SURF to find larger blob features [7]. This is useful when our roller coaster moves past objects that are large in the frame, such as the tree seen in figure 8. Modifying this did not affect our results very much. Running with `NumOctaves` $= 5$, `NumScaleLevels` $= 3$, `MetricThreshold` $= 2000.0$, `frameskip` $= 16$ allowed our algorithm to reconstruct 168 total frames, whereas the same configuration with `NumOctaves` $\in [3, 4, 6]$ only allowed our algorithm to reconstruct 165 total frames. This is only a 1.8% increase.

Increasing SURF's `NumScaleLevels` parameter allows SURF to find a greater quantity of small blobs [7]. It cannot be less than 3, but increasing it above 3 did not improve our results. Running with `NumOctaves` $= 5$, `NumScaleLevels` $= 4$, `MetricThreshold` $= 2000.0$, `frameskip` $= 16$ allowed our algorithm to reconstruct 40 total frames, and running with `NumOctaves` $= 5$, `NumScaleLevels` $= 5$, `MetricThreshold` $= 2000.0$, `frameskip` $= 16$ allowed our algorithm to reconstruct 167 total frames.

Increasing SURF's `MetricThreshold` parameter increases the minimum threshold for feature 'strength' [7]. This gives us a greater quantity of high-quality features which are more likely to find strong correspondences in subsequent keyframes. This also makes us less likely to find correspondences in general, because it may be that the same feature has 'strength' higher than `MetricThreshold` in one image but not in the other. The higher we set this threshold the more likely it becomes that we will fail to find the same feature in two consecutive images, thus the less likely we are to find a correspondence. When the threshold is much higher than 2000 (we tested with `MetricThreshold` $= 4000$), we will reach a failure state earlier, because we will be unable to calculate a fundamental matrix

due to too few features. When the threshold is much lower than 2000 (we tested with `MetricThreshold` $\in [800, 850, 900, 1000, 1100]$) we will reach a failure state earlier. With low thresholds we obtain so many erroneous feature correspondences that they will cause MATLAB's `estimateFundamentalMatrix` function to fail with an exception because there are never enough epipolar inliers for any of the sampled fundamental matrices [9]. Unfortunately, we cannot provide a specific suggestion for a good `MetricThreshold` parameter, because the effects of this value are entirely dependent on the quality and structure of the input images. We can suggest that future work start by doubling `MetricThreshold` until the quality of their output degrades, then doing binary search to find a good-enough `MetricThreshold` between the two best values.

Choosing a high `MetricThreshold` also increases the (totally subjective) smoothness of our point plot. This is because the high quality features are less likely to be incorrectly corresponded with the wrong feature in an adjacent frame. This is especially important because the scene in and around a roller coaster is full of repetitive elements, like the repeating structure of the track, the similar pieces of support steel, and repeating patterns in the nearby rides and buildings. These elements are often incorrectly matched as correspondences, as seen in figure 8. Modifying `NumOctaves` and `NumScaleLevels` also helps with this by narrowing the range of feature scales we detect, reducing the occurrence where a nearby feature in one frame is incorrectly corresponded with a far-away feature in another frame.

5.3. Final Word

Overall, we consider these experiments a failure. Our camera pose estimation is not robust enough to create smooth models of the entire track. There are large portions of first-person ride videos which are totally inscrutable to our methods, including frames like the ones seen in figure 2 and 14 which have been nearly destroyed by the camera's auto white-balance feature. We were unable to find a configuration of SURF parameters and `frameskip` value which reduced the reconstruction error sufficiently to make a smooth-looking track model, so none of our results are worthy of being 3D printed. Also, the processing takes so long (on the order of 1 hour per 150 frames successfully processed, though we did not take explicit notes of our timing), and needs to be manually re-calibrated for each video (because the scale and quality of features in different videos varies widely depending on video quality and camera resolution), that this is not faster or better than simply constructing the model manually in some 3D modeling software.

Figure 14. Frame number 2814 from a first-person ride video [3], unedited.

6. Code And Full Results

MATLAB code and the full experimental results of this paper are available at `https://github.com/tsell/reconstructing-roller-coasters`.

References

[1] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool. Surf: Speeded up robust features. *Computer Vision and Image Understanding*, 110(3):346–359, 2008.

[2] P. Beardsley, P. Torr, and A. Zisserman. 3d model acquisition from extended image sequences. *Computer Vision*, pages 683–695, 1996.

[3] FrontSeatCoasters. Six flags magic mountain goliath pov hd roller coaster on ride front seat gopro steel 2013. Web, 2014. https://www.youtube.com/watch?v=N_uV0Q2UH98.

[4] M. Frucci and G. S. di Baja. From segmentation to binarization of gray-level images. *Journal of Pattern Recognition Research*, 1:1–13, 2008.

[5] GoPro. Hero3 field of view (fov) information. Web, 2016. https://gopro.com/support/articles/hero3-field-of-view-fov-information.

[6] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2003.

[7] MathWorks. detectsurffeatures: Detect surf features and return surfpoints object. Web, 2016. http://www.mathworks.com/help/vision/ref/detectsurffeatures.html.

[8] MathWorks. edge: Find edges in intensity image. Web, 2016. http://www.mathworks.com/help/images/ref/edge.html.

[9] MathWorks. estimatefundamentalmatrix: Estimate fundamental matrix from corresponding points in stereo images. Web, 2016. http://www.mathworks.com/help/vision/ref/estimatefundamentalmatrix.html.

[10] MathWorks. kmeans: K-means clustering. Web, 2016. http://www.mathworks.com/help/stats/kmeans.html.

[11] MathWorks. knnsearch: Find k-nearest neighbors using data. Web, 2016. http://www.mathworks.com/help/stats/knnsearch.html.

[12] MathWorks. Structure from motion from multiple views. Web, 2016. http://www.mathworks.com/help/vision/examples/structure-from-motion-from-multiple-views.html.

[13] J. Nielsen. How to extract images from a video with avconv on linux. Web, 2015. http://www.dototot.com/how-to-extract-images-from-a-video-with-avconv-on-linux/.

[14] T. Sellmayer. Rotate camera points fig. 13. Web, 2016. https://www.youtube.com/watch?v=N_uV0Q2UH98.

[15] B. Triggs, P. McLauchlan, R. Hartley, and A. Fitzgibbon. Bundle adjustment: A modern synthesis. *Proceedings of the International Workshop on Vision Algorithms*, pages 298–372, 1999.