CoasterQueue - Tracking wait times with mobile phones

Josh Fuerst, Rachna Fuerst, Jong Char
Purdue University

Most of the time spent in a theme park is waiting in lines. In most parks, there are only two ways to determine the wait time of a ride. One is by physically walking to the ride entrance, where an estimate is posted. This can waste valuable time only to find out that a ride has an incredibly long wait time. The other is by using an existing crowdsourcing application. However, these applications require *active* user participation to collect data, meaning the users must input the data manually. This can result in a lack of data and poor reviews. In this paper, we present a wait time estimation system for theme parks. This system is based on data collected by a mobile phone's sensors including accelerometer and rotational data. Using this sensor data, we detect *movement patterns* which determines the user's current state. We show that the patterns generated by rides, walking, and waiting are distinct enough to determine what the user is doing. Using sensor data, in conjunction with time, we are able to determine how long a user waited in line and which coaster they were waiting for. Using this method, we have developed a *passive* line tracking system which does not require any user interaction and can be used in any theme park in the world. We comprehensively test our system and the evaluation results show that the system achieves outstanding accuracy for wait times and classifying coasters.

Keywords: Mobile Computing, Wait Time, Roller Coaster, Theme Park, Dynamic Time Warping

1. INTRODUCTION

Theme parks are a highly attended summer attraction all over the world. In 2012, the top 10 theme parks in North America saw an average attendance of almost 10 million guests throughout the season [Jeffers 2012]. However, these parks have one big issue: line wait times. Roller coaster times range anywhere from a few seconds to a few minutes, while the wait times can range from a few minutes to a few hours. Currently, there is no effective way of removing lines from theme parks; so the reality is guests must wait [Corrado]. The wait times vary from park to park and ride to ride, but they have been known to become very large. For the newest ride at popular parks, it is common to wait multiple hours before getting on a ride. We have seen extreme cases, where ride wait times are in excess of 5 hours on holidays such as New Years Eve or the Fourth of July. Most parks only operate between 12 and 14 hours per day, which means a single ride can easily consume 25% of the day. This can be troublesome at a theme park which has over a dozen rides. Therefore, guests frequently seek out the rides which currently have the lowest wait time to increase the number of rides they can enjoy.

In today's world, it is rare to find a person who does not own a mobile device. In fact, a recent survey by Thinkwell Group detailed consumer trends in mobile technology in theme parks [Group 2013]. The study found that in 2013, 76% of theme park guests carried their mobile devices with them in the park. This means, worldwide, Walt Disney Attractions saw over 96 million mobile devices in their parks [Jeffers 2012]. Because of these numbers, we believe that mobile devices are an untapped source of data within theme parks. The survey also showed that 67% of the guests would like to see more mobile integration in theme parks [Group 2013]. As an additional question, guests were asked to rate a collection of desired enhancements. Within the collection, "checking queue wait times" ranked *second* below "front of line access". This survey directly shows that theme park guests use phones and want more information from them. It also shows that guests are concerned with wait times as the top two desired enhancements are directly related. This makes us confident that a mobile application, which tracks and reports

queue times, should be very well received by theme park guests.

Theme parks are well aware of this desire. In fact, almost every park has some type of wait time estimation system in place. Most parks do not have technology involved and use the length of the line to estimate the wait time. This provides users with information that is infrequently updated and often wrong. More technological solutions do exist; however, they are very expensive and many parks have not installed them [Niles 2006]. Even the parks which have paid to install these systems still have issues. In almost all parks, the information is generally presented at the entrance to the ride. This means that guests must first get to the ride entrance *before* they can figure out how long the wait is. This may not seem like a problem but a large amount of planning is done during theme park construction. It is not a wise decision to put all the popular rides in one corner of the park. This will cause congestion and other park areas will have less exposure. Because of this, most of the popular rides are scattered across the park and may require a good deal of time to get to. Knowing a rides queue time in advance can aid guests in determining the most time efficient plan.

There are already many mobile applications which attempt to solve the same problem. These applications use crowd-sourcing to obtain the information [Innetic 2013]. The users must *actively* input their wait times and corresponding rides. The information is then uploaded to a server and available for other users to view. This approach solves the issue of viewing ride times anywhere, but it relies on human input. This results in times being inaccurate and even malicious.

In this paper, we present a new approach to queue time estimation, based on crowd participatory sensing. We have built an application which will estimate queue times using guests mobile devices much like the others. The novelty of our approach is that the queue tracking will be completed *passively*. This means the queue times will be tracked *without* user interaction. This is accomplished by using the mobile device's sensors to gather information about the users current state. This approach poses two major challenges: state detection and ride classification. State detection: users will be doing many things at a theme park so we need to determine when the users are waiting for a ride. Ride classification: not only do we need to know when users are waiting on a ride but we also need to know which ride they are waiting for. We need a way to determine which ride the users have just experienced that way we can pair the wait time with its respective ride.

In this paper, we develop solutions to these challenges. To be specific, we collect accelerometer and rotational sensor data. We show that by analysing the rotational sensor data we can measure the user's leg movements. Using this information, we can accurately determine if a user is walking or waiting. Each roller coaster also has a unique g-force signature. We can use this signature to determine which ride the user has just experienced. Using these combined methods we can estimate how long each user has been waiting and which ride the user waited on.

We evaluated our system using three types of mobile phones (Samsung Galaxy S2, Motorola Droid Razr M, and Google Nexus 4). We evaluate the step detection mechanism first by having users count the number of steps during a timed interval. Our solution was able to track the users steps with an 95% accuracy. Next, we evaluate our ability to determine if a user is walking or waiting. To do this, we compare our log file to a user's provided schedule. Our method was able to recreate the users schedule with a maximum deviation of 78 seconds. Finally, we tested how well our system calculated the wait time vs. the user's real wait time. Our system was able to determine the user's wait time with a 10% error.

Due to cost and time limitations, we could not collect real-time data for roller coasters ourselves. Because of this, we evaluated the ride classification portion using external data. We used the data provided by Gulf Coast Data Concepts as a baseline for the Dynamic Time Warping algorithm [Concepts 2008]. We then generated test queries for each ride by manipulating the data. The manipulation was done by introducing random noise into the signal and adding/removing data. Through our experimental study, our algorithm can accurately detect the ride with a minimum of 95% accuracy. With a large user base, this accuracy is enough to obtain an accurate estimation

of the current wait times for rides.

This paper makes the first attempt at developing a *passive* way to track queue times with mobile devices. More specifically, we make two distinct contributions. (1) State detection: we develop an algorithm which uses mobile sensors to determine the users current state. Within a theme park, we use this information to track when a user begins waiting in a line and leaves a ride. (2) Ride classification: we develop an algorithm which can determine which ride the user has just experienced using only accelerometer data. Combining our two contributions can lead to an application that can be used by millions of people at a theme park.

The rest of the paper is broken down as follows: we first introduce the related work in section 2. In section 3, we detail our technical solution to this problem. The evaluation results are presented in section 4 followed by the limitations and possible improvements in section 5. Finally, we present our summary in section 6.

2. RELEVANT WORK

In this section, we survey two major areas related to our paper. First, we examine the current state of the art for queue tracking. We show how theme parks and current mobile applications are tracking queue wait times. Then, we examine related research in activity tracking using mobile sensor data.

2.1 Tracking Queue Times

The current state of queue times in theme parks is a grim reality. Currently, the most popular method for queue estimation is the same one which has been used since the creation of theme parks. In this system, employees simply use landmarks to estimate the queue wait time. The park staff knows that if the line reaches turnstile four, then the wait is approximately twenty minutes. More recently, as technology has advanced, newer systems have been created.

The most common electronic queue tracking system uses a card [Niles 2014]. The idea here is that a random guest is given a card when he/she enters the queue for a ride. Before the card is given to the guest, the card is scanned by a park employee. Once that guest gets to the loading platform, he/she gives the card to another park employee. The employee will then scan the card, allowing a software system to record the waiting time and update the park's database with the new data. The data is available on-line in very few parks. Most of the parks just update the display at the ride entrance. Another method is to monitor guests via video footage [Kuklin]. This technology can allow video cameras to be placed in the queue. These cameras can mark guests and track how long they wait in line. In these more advanced parks, the guests are capable of getting more real-time information.

Table I: Comparison of Queue Tracking Systems

Method	Park Cost	User involvement	Accuracy	Automated	User Visibility
Landmarks	None	Passive	Low	No	At Gate
Sensor Systems	High	Passive	High	Yes	At Gate/Online
Active Mobile Apps	None	Active	High/Low	Yes	Mobile Device (online)
Our Application	None	Passive	High/Low	Yes	Mobile Device (online)

All of the previously described systems perform their jobs well, however there are many drawbacks. In the old fashioned parks, there are no electronic systems in place to track queue times. This means that the times are estimated and can go hours without being updated. Guests are also plagued with physically walking to the rides entrance before being presented with the estimated wait time. The few parks which tried to address this issue had to invest in equipment to handle the electronic systems. Theme parks are also geographically separated by many miles. It is very unlikely for a guest to travel a greater distance just to get automated ride wait times. Due

to these reasons, parks are very reluctant to invest in electronic queue tracking systems. Table I summarizes the current methods and their related properties.

There is also an interest in technology related to line mitigation rather than tracking [Corrado]. This technology is being placed in a growing number of parks. A notable one is LegoLand, where the park uses a system called Qbot [Group 2007]. This system allows a user to "check in" to a ride. The device will then inform the user that they can get on the ride at a specific time. All the user has to do is return to the ride at that time and they will get on the ride much faster. Disney world has a similar system in place called FastPass [Corrado]. The goal of these solutions are to completely remove lines from parks altogether (making wait tracking obsolete). This sounds perfect for parks, however there are still issues. These systems still require a huge upfront cost to implement. In addition, parks which currently use this kind of system provide it as an "executive" feature and charge more for it. Since everyone cannot afford the extra costs, there are still customers who wait in line normally and would like to know the current wait time.

In an attempt to remove the park operators from the equation, mobile developers have developed their own solutions. The latest addition to queue tracking systems is mobile applications. The current approach is to track queue times using a crowdsourcing approach [Innetic 2013]. The downfall of this method is that users are reluctant to spend time entering data. Normal users may forget how long they waited or malicious users may input the wrong time in an attempt to deter users from a specific ride. This leads to inaccurate queue times or in the worst case no data at all. Our solution takes all of these into consideration and creates a passive crowdsourcing application. One in which we take advantage of the mobile devices sensors to determine the wait time without interaction by the user. This will lead to more crowdsourced data and more accurate queue times. The end users experience will be much friendlier than current technology.

2.2 Activity Recognition

Currently, there is a modest amount of research detailing activity recognition. Much of this work results in using very complex machine learning (ML) algorithms or dynamic time warping algorithms (DTW) to gain accurate results. However, in most cases their goal is to differentiate between multiple types of activities such as walking, running, sitting, standing, climbing... etc [Dernbach 2012]. Many methods have obtained very accurate results, but at a great cost. We will now discuss Machine Learning and Dynamic Time Warping and their benefits to our project.

Machine Learning: Machine Learning usually occurs in four steps: data collection, feature extraction, training, and matching [Kwapisz et al. 2010]. First, a dataset has to be collected and built. This usually requires many instances of the same data being captured. For ride classification, we want multiple examples of each ride, not just a single example. Next, a set of features must be extracted. Usually this details some sort of data analysis to extract important information about the data. For rides, this could be things like maximum acceleration, standard deviation, number of peaks, time between peaks, etc.

In all cases, feature extraction requires some sort of extra processing to be done on the collected data. In many cases, the processing can be very computation intensive. This could lead to poor performance on the mobile device and possibly a higher battery drain. An argument could be made to upload the data to a server and perform feature extraction there. This is an option but data transmission can also incur a large power loss if high amounts of data are transmitted. Our goal is to keep computation and transmission as low as possible on the mobile device.

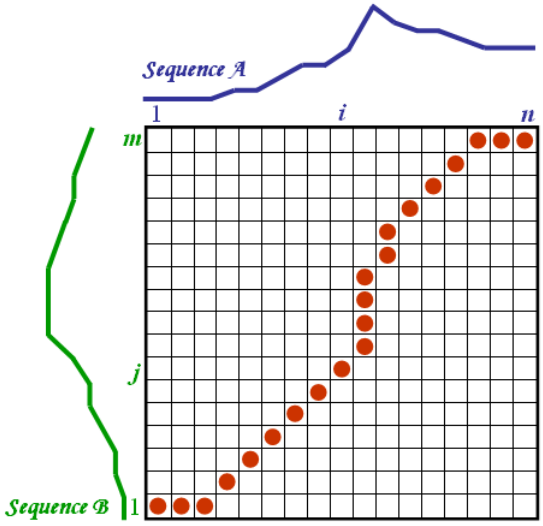
Once the feature extraction is completed for every dataset in the database, we can train the ML algorithm. The training is what allows the algorithm to "learn" what each dataset means. Once the learning is complete, we can give the algorithm a query and it will do its best to match the query to an existing dataset. In this way, we can determine if the query was for *Top Thrill Dragster* or *Millennium Force*.

Machine learning also has another big problem. Each person performs actions differently. For example, two people have different walking speeds. These deviations are harder for ML algorithms to handle [Muscillo et al. 2007]. For an ML implementation to work well, each device would need

to be trained for each user or the extracted features would need to be created to handle those deviations [Muscillo et al. 2007]. This may give accurate results, but requires too much burden on the user. We would also need to devise a way to perform the training, which is simple and easily understood by all users. This alone, poses an almost insurmountable challenge.

Other methods are able to remove this customization and build algorithms that work for any user [Kwapisz et al. 2010]. However, these methods require a large amount of feature extraction to be accurate. This means that large amounts of calculations would need to be performed before the activity could be recognized. We aim to keep all activity recognition on the device to preserve privacy, which means we want to minimize calculations to preserve battery.

One option is to use each data point(g-force value) as a feature. This is beneficial because it does not require extra computation to complete the feature extraction. However, this means that every ride and query must have the exact same number of data points. It may be feasible to pad every ride to be equal to the length of the longest ride, but there still is an issue. Our data collection must always begin at the exact same moment. For example, assume the data collection for the training set of ride1 began at the top of the first hill of the roller coaster. Now lets say for some reason our query data started collecting data a few seconds after the ride has gone over that hill. Now all the features will be mismatched, leading to a false result. Obtaining this precision is not practical. This means when we use time series data we MUST perform a feature extraction which is time independent.



DTW Warping Matrix Example[1]

Dynamic Time Warping: Dynamic Time Warping is another matching algorithm. The reason DTW was considered for our system is because it was designed to be used with time series data. DTW was designed to compare time sequences which may vary in time or speed. In general, DTW is a method that calculates the best match between two given sequences. The sequences are "warped" non-linearly in the time dimension to determine a measure of their similarity.

To achieve a warping between two sequences A and B we can construct an n x m distance matrix as seen in Figure 1. Each cell of the matrix (i,j) represents the distance between the i-th element of sequence A and the j-th element of sequence B. The distance method can vary depending on the application, but we use the absolute distance. To find the best warp, the

[1]Image taken from `http://www.mblondel.org/images/dtw2.gif`

algorithm tries to find the shortest cost path from the bottom left of the matrix to the top right. The cost of a path is the sum of all the distance values visited along the path. The brute force method would be to find all the possible paths and choose the shortest one. This works, but most DTW algorithms utilize dynamic programming to increase efficiency. Overall, if a path stays very close to the matrix diagonal the two signals do not need much warping. However, if the path wanders far from the diagonal the two signals need a much greater warping to match. Some DTW algorithms can also define a warping constraint. This will constrain the maximum deviation from the matrix diagonal. This can be used to increase the efficiency of the algorithm depending on the application.

When using DTW for our purposes, the query sequence will need to be compared to all sequences within the database. The distance for each comparison will be noted and the overall shortest distance is chosen as the matching sequence. The biggest benefit of DTW is that it does not need any feature extraction. The algorithm was designed to be run on raw time series data. This means the data can be sent directly to the server for processing without any extra computation done on the device. Since DTW allows differences in time, smaller sets of data can be transmitted and matched to a longer template data. Thus the algorithm can handle lower computation on the device and less data transmission.

Immediately, it is apparent that this can solve the issue where our query and training data have slightly different start points. This is a huge bonus for our system since triggering data collection at the exact same moment for every ride is not feasible. DTW is also more enticing because we do not need to analyse and determine the appropriate feature sets. DTW is run on the raw data that comes from the accelerometer. Because of these benefits, our system design uses DTW to determine which ride the user just waited on.

3. MODEL DEFINITIONS

Although the idea may seem simple, the design of our system required overcoming some challenges. One of the challenges we had to deal with was detecting whether a user is waiting for a ride or lunch or walking to the next ride, which we discuss in section 3.2. Another challenge we were faced with was classifying which roller coaster the user was just on, which we explain in section 3.3. The system we implemented to overcome these challenges has three major components: activity recognition, coaster classification and the back-end server. In this section, we will describe the implementation and challenges faced for each component.

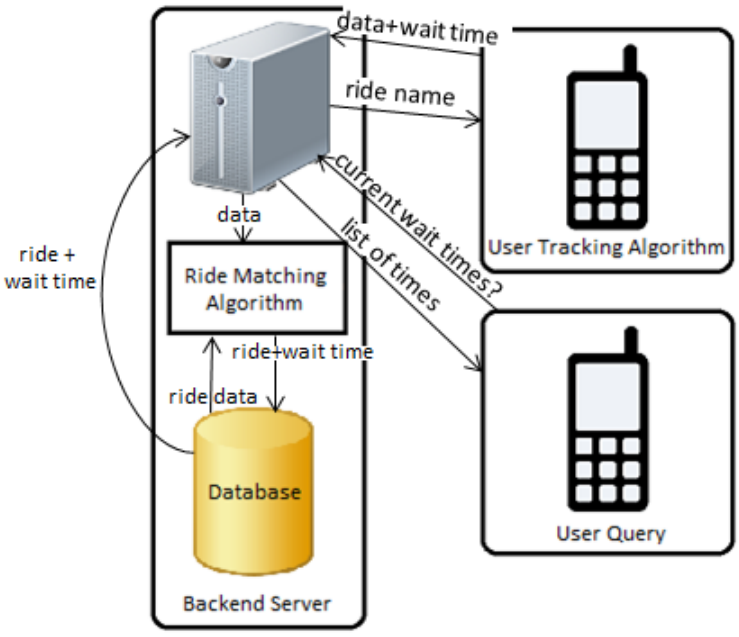
3.1 System Overview

Figure 2 shows the overview of our system. Our system contains three major components: activity recognition, coaster classification, and a back-end server. The back-end server consists of a database which will hold all of the pre-populated ride data. The ride data will be a g-force mapping of each roller coaster. A g-force mapping is simply a time series of gravitational forces the mobile device detects. The coasters will be grouped by the park in which they are contained. The server will then perform the ride classifications using DTW. The resulting wait times will also be stored in the database for later viewing. The uses of our system will fall into two cases: tracking the user's activity and using the application.

In the first case, the mobile device will automatically track the user's activity. This method will periodically transmit data to the server for analysis. The data which is sent will contain the estimated wait time, the ride data (g-force mapping), a unique identifier, and the park the user is at. The server will use this data to determine how long the current device just waited for a specific ride. The location is only used to narrow down the rides that the ride matching algorithm will compare the data to. The unique identifier is generated by our application during installation and has no tie to user information to prevent privacy issues.

In the second case, the user will be using the application actively. The user will request to see the wait times for rides at a specific park. The back-end server will then get the list of rides and associated wait times from the database and send them to the application. The user can then

use this information to determine which ride to go to next.



System Overview

Activity Recognition: The most important piece of our system is the ability to determine what the user is doing. Without this ability, our system would fail. If we cannot determine the user's activities, we cannot determine the user's wait time. As the user is moving about the theme park, we must find a way to accurately determine the user's state. If the user is actively walking around the park, there is no useful information for us to gather. However, if the user quits walking we can infer the user is waiting for something (maybe a ride?). To overcome this challenge, we rely on the rotational sensor of a mobile device. To preserve privacy and keep data transmissions low, the activity recognition portion is performed on the mobile device.

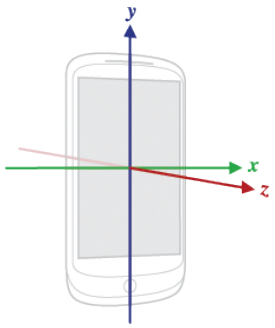
Coaster Classification: The next important piece of the system is the ride classification system. Once we are able to accurately track the users actions, we can determine how long a user has waited for something. The next obstacle to overcome is finding out what the user was waiting for. To do this, we rely on accelerometer data collected from the mobile device. We can use accelerometer data collected while the user was on a ride, to generate a g-force pattern. We then use the data to perform pattern matching on a pre-populated database of ride patterns.

Back-end Server: The back-end server serves three main purposes. It's first purpose is to be the work horse, which performs the computation intensive coaster classification. It's second purpose is to be in charge of storing all the ride data and wait times. The server will collect data from clients and keep a real time log of current wait times for every ride. Finally, it must serve wait times to mobile devices requesting to see them.

3.2 Activity Recognition: What am I doing?

The goal of our system is to track user wait times without any user interaction. We believe this method is much better then the current crowdsourcing approach, which requires users to manually enter ride data. To do this, we need a way to accurately determine what the user is doing. However, we only need to differentiate between two actions: moving and not moving (ie. walking and waiting). Even if the user is waiting for lunch or is taking a break we will track the

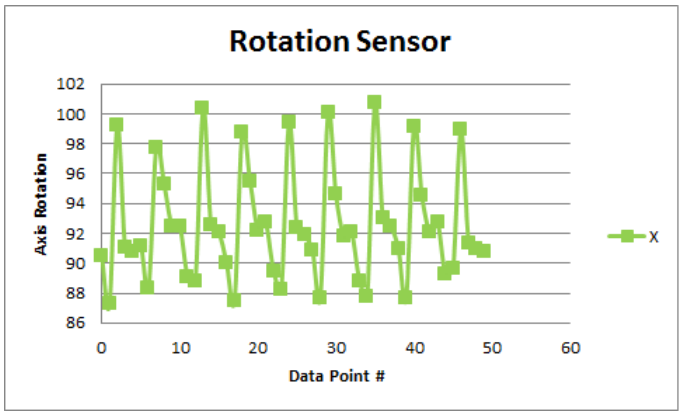
user as waiting and rely on our coaster matching algorithm to detect the data doesn't match a ride. This means that we do not need to differentiate between walking, running, climbing, etc. We can use this to our advantage to greatly simplify the activity recognition algorithms which currently exist.



Android Orientation Sensor Axis[2]

Instead of using GPS enabled location information, we resort to more energy efficient sensing resources like the android rotational sensor (*TYPE_ROTATION_VECTOR*) [Zhou et al. 2012]. This sensor provides information about the rotation of the device. The provided information can be easily translated into: azimuth, pitch, and roll. The azimuth refers to the degree of rotation around the z-axis of the device. The pitch refers to the degree of rotation around the x-axis. The roll refers to the degree of rotation around the y-axis. Figure 3 shows the axis in relation to the mobile device.

We found that the rotational data can be used to accurately track a user's movements. We have developed an algorithm which can translate the rotational data into an approximation of the number of steps the user has taken (much like a pedometer). Through our experimentation, we discovered that as a user walks, the rotation sensor generates a very distinct sinusoidal pattern. Figure 4 shows a data sample taken as the user walked across a room with the device in the front pocket. This pattern directly maps to the number of steps a user takes.

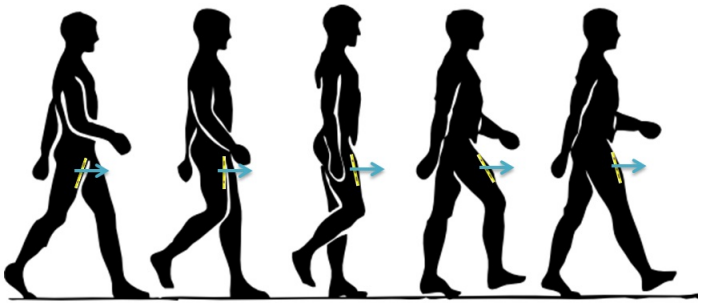


Rotational Sensor Example (X-Axis Only)

Consider the following situation: A theme park guest is carrying a mobile device in the *front* pocket. As the guest walks around the theme park, the phone will be rotating in conjunction

[2]Image taken from `http://developer.android.com/reference/android/hardware/SensorEvent.html#values`

with the user's leg. If the phone is upright in the user's pocket, this rotation will occur primarily around the x-axis. A visual example can be seen in Figure 5. As the guest's leg moves back and forth the mobile device will be generating values which correspond to the angles of rotation. We can use these values to generate a periodic waveform. As the leg extends forward, the waveform will rise to a peak. As the leg swings backwards, the waveform falls to a trough. This means that there is a direct mapping between the peaks/troughs in the waveform, and the number of steps the guest has taken.



Example of walking rotation[3]

We implement a simple algorithm that tracks the total number of peaks and troughs. The algorithm was designed to take input directly from the rotation sensor. We poll the rotation sensor at 20 Hz. Since android only allows us to "suggest" a value to the operating system, we can only provide a suggested polling time. Android does its best to meet this polling time, but we may get data at a faster or slower rate. We can handle somewhat slower rates, but if the rate becomes too low, we do not have enough data to accurately track steps. We chose 20 Hz because it provided us with more than enough data points to accurately detect steps and it was less detrimental to battery life [Zhou et al. 2012].

As the sensor reports data, we feed it into the algorithm one value at a time. The algorithm counts peaks/troughs that are separated by a predefined *delta* value. It begins by looking for a maximum. Once the current value has fallen at least *delta* from the last seen maximum we can begin looking for the next minimum value. The algorithm then waits for the values to rise at least *delta* above the last seen minimum value. Using this method, we can accurately track the number of crests/troughs seen in the waveform which corresponds to the number of steps the user has taken. This method also allows the algorithm to be easily configurable. Through our tests, we found that a *delta* of 5 degrees was enough to ignore noise, but also not miss small steps.

The next challenge was to map steps to activities. Since we are only worried about two states, the mapping is very straight forward. Walking means the algorithm is detecting steps and waiting means the algorithm is not detecting steps. We decide to break our mappings into timed intervals. Over each interval we count the number of steps taken. If the steps taken is above a threshold value of 10 steps, then we can safely say the user is actively walking. If the number of steps is below the threshold, then the user is not actively walking and we classify them as waiting. We chose 10 steps as the threshold because we found on average a fast walker takes about 25 steps in a 15 second period. Therefore, we brought the threshold down to 10 to account for slow walkers.

During the time spent in a line, the user will experience brief bursts of walking as the line moves forward. We do **not** want to classify the user as walking in these situations because they are still waiting in line. To prevent this, we provide a state transition threshold ST. This means that we must see ST consecutive states before we transition to that state. We have ST set to 3. This means we have to see 3 consecutive walking states, a 45 second period, before we transition

[3]Image taken from `http://www.clker.com/cliparts/6/4/7/7/1197106712447482105johnny_automatic_walking.svg.hi.png`

from waiting to walking. When standing in line for a ride, the user will only experience short bursts of around 10-20 seconds where the user is moving forward in line. This buffering prevents us from classifying a user as walking when the user is just moving forward in line. This method will also rule out invalid steps which will occur as users twist and turn on a ride. Finally, the buffer also adds another possible layer of configuration to the overall algorithm.

Combining all the above steps we are able to successfully track a user with the simple granularity of walking or waiting. The final step is to determine which ride the user was waiting on.

3.3 Coaster Classification: What ride was that?

Ride classification poses the second biggest hurdle we had to overcome. We need a way to determine which ride a user was on based only on data collected from the mobile device's sensors. To do this, we decided to use the device's accelerometer. Similar to activity recognition, there has been much research done on pattern recognition. The two biggest methodologies we discovered are Machine Learning (ML) and Dynamic Time Warping (DTW) [Dernbach 2012; Kwapisz et al. 2010; Muscillo et al. 2007]. These two methodologies were developed to find the best matching of a query to a predefined dataset. As stated in section 2.2 we chose to use DTW for our system.

The system works as follows: we have implemented a data structure which tracks data for only a specific amount of time. As soon as our application is launched, we begin tracking accelerometer data. We only track this data for a specific window, 6 minutes. This means our application always has a log of the last 6 minutes of accelerometer data. We chose 6 minutes because the longest documented coaster time is 5 1/2 minutes. The moment a state transition occurs from walking to waiting, the time is logged. Next, the moment at which the state changes from waiting to walking we know one of three things have happened. First, the user could have been taking a break for lunch and then started walking again. Second, the user could have just gotten off a ride and is proceeding to the next one Finally, the user could currently be on a coaster and the rotations are being logged as steps, we hope to avoid this situation. Once the transition is detected, we send the waiting time along with the data window, up to the back-end server for analysis. The server will determine what ride matches the window using DTW.

In the first situation, the server will not find a match. At this point, we have no other option but to determine the user was not moving for some other reason, possibly eating food or taking a break. In the second situation, we will find a match. We can then use the wait time to update the wait time for the matched ride. In the last situation, the user will only be partway through the ride. The data uploaded to the server will be incomplete and the matching ability will greatly fall. We rely on the robustness of our activity recognition algorithm and the transition threshold to prevent this situation from occurring.

We chose to use the Gesture Recognition Toolkit written by Nick Gillian for our implementation of DTW [Gillian 2013]. This tool-kit was designed for use with gesture recognition. We train the DTW algorithm using each roller coaster as a "gesture". We can then query the trained dataset by providing any length time series data. The algorithm will find the ride with the closest match and report back three metrics: ride name, distance, and likelihood. The distance relates to the amount of warping needed to be done to obtain the match. The lower the distance the closer the match is. The likelihood is a percentage. This is where the algorithm is giving an estimate of the confidence of its guess.

We use the distance and confidence to determine if the guessed ride is a correct match or not. In the event that the distance is extremely high or the likelihood is very low, we ignore this report and do not update any wait times.

4. METHOD DESCRIPTION

During evaluation the mobile portion of our system was tested using Android 4.0.4, 4.1.2, and 4.4.2 on a Samsung Galaxy S2, Motorola Droid Razr M, and Google Nexus 4 respectively. The server portion was run on a free Amazon EC2 instance. The data collection was done using

SOLR/Jetty and the web serving was done using NGIX. The DTW pattern matching was implemented in C++ using the GRT by Nick Gillian.

The evaluation of our system was done by breaking the system into several logical sections and then evaluating each section separately. After each section was evaluated, we then tested the system as a whole with a full system live test. We first evaluate the effectiveness of our step detection algorithm in section 4.1. In section 4.2, we test and evaluate our overall activity recognition algorithm. We then test and evaluate the our roller coaster classification algorithm in section 4.3. Finally in section 4.4, we present the overall performance of our wait time tracking system.

Table II: Step detection accuracy over various time intervals

Duration	**# Runs**	**Avg. Counted Steps**	**Avg. Reported Steps**	**Accuracy**
15 Seconds	6	26.0	26.2	99%
30 Seconds	6	53.0	52.5	99%
60 Seconds	6	104.5	102.0	97%
120 Seconds	6	211.5	208.6	98%

4.1 Step Detector

The base component of our system is the step detection algorithm. As discussed in section 3.2, the step detection algorithm utilizes rotation data to determine the number of steps the user has taken in a given time interval. A simple test was employed to determine the accuracy of this algorithm. First, a set of test users was determined, two members from our group. Each test user was instructed to perform a set of walks with the mobile device in their front pocket. Each user performed four walk intervals: 15 seconds, 30 seconds, 60 seconds, and 120 seconds. During the walk the user was instructed to manually count the number of steps taken. At the end of the time interval, the manual calculation was compared to the count given by our algorithm. Each walk interval was performed three times by each user. This gives us a total of twelve data points per user. The results from our analysis of the users can be seen in Table II. Our methodology performed exceptionally well with an average detection accuracy of around 95%. This is more then accurate enough to successfully detect a user's movement for our purposes.

4.2 Activity Recognition

After we have a strong base built with our step detection algorithm, the next portion to evaluate is the activity recognition algorithm. This is the portion of our system which determines if the user is actively walking or waiting. This portion was the most difficult to test. The activity tracking was designed specifically for tracking users within theme parks but we did not have access to a theme park at the time of writing because all nearby parks were closed. We eventually discovered that a professor's daily activities are very similar to those of a theme park guest. In a theme park, guests move from one ride to another. The rides are separated by waiting in line. While waiting in line, the users are subject to small movements (ie. user's fidgeting) and brief movements as lines progress forward. Similarly, professors move from class to class throughout the day as they teach. Each movement is separated by the time spent in class (ie. waiting). While in class, professors often fidget and undergo small movements as they move around the classroom much like queues moving forward. Due to these similarities, we decided to test our algorithm by letting some teaching assistants (TA) in our group use it throughout a day. The users were instructed to keep their phones on them at all times and stored in their front pocket while not in use. Otherwise the users were encouraged to use their phones as they normally would. The TAs were also asked to write down the exact times they began a large movement from one location to another. The data collected by our system was then compared to the noted schedule of the TA.

This evaluation was completed by 2 group members over the course of a single day. After analysis of the results, we found that our application was able to easily map to the user's schedule. On a granular scale (50 minute class periods) we were able to reconstruct the user's activities with 100% accuracy. This means we were always able to detect when the user moved from one class to the next. However, there were some small discrepancies in the exact timing. The maximum difference between the manually recorded time and the phone detection time was 00:01:18 with an average time difference of 00:00:48. Since we require our algorithm to see a state change for at least 45 seconds before the state change is allowed, there is a slight time shift between when the users started/stopped and when the application detected it. This delay is only a few seconds from the average time difference. If we adjust our timings taking the 45 second delay into account we can detect the times within an average of a few seconds. This shows that we can accurately determine when a user is walking and waiting. In addition we can do this within a worst case time window of a couple minutes. During the standalone evaluation, the activity recognition algorithm performed very well. We were able to reconstruct an entire days schedule based only on the information obtained from our application data.

We also note that at the time of reporting the wait time may be obsolete. The only information we have is how long the user who just got off the ride waited. This is not necessarily an accurate representation of the time a new incoming user will wait. This is a known "issue" and not within our scope to solve. Other wait time applications also suffer from the same issue. We leave this to future work.

4.3 Coaster Classification

The final component to our system is the coaster classification algorithm. As discussed in section 3.3, this algorithm uses DTW to determine which ride the user was just on. This was the hardest portion of our system to test due to time constraints. We were not able to collect data ourselves to test the validity of this solution. To perform the evaluation, we used data collected by Gulf Coast Data Concepts (GCDC). GCDC is a company which manufactures accelerometers. As a test they took some of their devices to Disney World and carried them onto roller coasters. They published their findings on-line and were nice enough to provide the raw data to us. Using this data, we devised a plan to test our classification algorithm. We were able to get raw accelerometer data for seven rides within Walt Disney World: Tower of Terror, Mission Space, Big Thunder Mountain, Test Track, Space Mountain, Expedition Everest, and Rockin Roller Coaster.

Our system uses DTW from the Gesture Recognition Toolkit implemented by Nick Gillian [Gillian 2013]. This tool kit provides a fully functional classification algorithm which uses DTW. For our tests, we trained the DTW algorithm using the raw data provided by GCDC. The matching algorithm was trained using every coaster except Test Track. One ride was left out of the training data in order to ensure we can detect false positives. Using the seven coasters we then created a set of test queries. For each coaster, we created five queries by adding 0%, 3%, 5%, 10%, and 15% noise to the data. This means each data point could differ in magnitude by the given percentage. Since our data collection takes places over a given time window, there can be significant random data occurring before and after the ride data. To simulate this, we took accelerometer data collected from the step detection evaluation and placed it at the beginning and end of the ride data. These queries were then fed into the matching algorithm. The results can be seen in Table III. The algorithm reports the predicted match along with a match likelihood and distance. The likelihood is the measure of the algorithm's confidence in its guess. The higher the number, the more confident the prediction. The distance is the measure of how much the matched pattern had to be manipulated before it was considered a match. The shorter the distance, the better the match is. Out of the 30 test queries which had possible matches, 29 were matched correctly, leading to a 95.8% detection rate for true positives with our implementation. It is also valuable to note that the only incorrect prediction occurred within the highest noise group.

We did however, have more trouble with false positives. If we cannot detect the ride with a

Table III: Coaster Classification algorithm results under various noise manipulations

Ride	Prediction	Likelihood	Distance(Absolute)
Rockin Roller Coaster	Rockin Roller Coaster	0.9516	19.2978
Space Mountain	Space Mountain	0.9389	24.5728
Mission Space	Mission Space	0.9446	29.1795
Big Thunder Mountain	Big Thunder Mountain	0.9669	14.1209
Expedition Everest	Expedition Everest	0.9538	17.9809
Tower of Terror	Tower of Terror	0.9264	28.7797
Test Track	Big Thunder Mountain	0.8696	54.8771
Rockin Roller Coaster 3%	Rockin Roller Coaster	0.9458	21.6058
Space Mountain 3%	Space Mountain	0.9327	28.5226
Mission Space 3%	Mission Space	0.9261	39.7736
Big Thunder Mountain 3%	Big Thunder Mountain	0.9535	20.8092
Expedition Everest 3%	Expedition Everest	0.9471	21.3412
Tower of Terror 3%	Tower of Terror	0.9226	29.7474
Test Track 3%	Big Thunder Mountain	0.8663	68.5389
Rockin Roller Coaster 5%	Rockin Roller Coaster	0.9414	23.7251
Space Mountain 5%	Space Mountain	0.9293	31.5925
Mission Space 5%	Mission Space	0.9116	49.9490
Big Thunder Mountain 5%	Big Thunder Mountain	0.9426	26.2914
Expedition Everest 5%	Expedition Everest	0.9385	25.8123
Tower of Terror 5%	Tower of Terror	0.9194	30.3318
Test Track 5%	Big Thunder Mountain	0.8619	82.7235
Rockin Roller Coaster 10%	Rockin Roller Coaster	0.9299	28.1598
Space Mountain 10%	Space Mountain	0.9139	44.3897
Mission Space 10%	Mission Space	0.8784	76.8821
Big Thunder Mountain 10%	Big Thunder Mountain	0.9192	43.5992
Expedition Everest 10%	Expedition Everest	0.9210	36.4777
Tower of Terror 10%	Tower of Terror	0.9146	31.0126
Test Track 10%	Space Mountain	0.8597	116.9300
Rockin Roller Coaster 15%	Rockin Roller Coaster	0.9207	34.3665
Space Mountain 15%	Space Mountain	0.9046	58.4378
Mission Space 15%	Expedition Everest	0.8646	101.6790
Big Thunder Mountain 15%	Big Thunder Mountain	0.9002	60.4038
Expedition Everest 15%	Expedition Everest	0.9097	48.2443
Tower of Terror 15%	Tower of Terror	0.9072	33.2130
Test Track 15%	Space Mountain	0.8580	150.5750

high confidence, we do not want to accept the associated wait time as it could introduce large errors. As seen above, the likelihood of the Test Track queries were still rather high with an average of 86%. These numbers are a bit troublesome as they are very close to the likelihood values of the true positives. In order to ensure the best matching rate, we need an accurate way to reject false positives from our system.

We hope that with live data, all the correct predictions will still have likelihoods above 90%. We could then use this metric alone as the rejection threshold. If we see correct and incorrect predictions with similar likelihoods we can also use the distance metric to reject a prediction. If we look at the distances within each noise group, we can see that there is a significant jump between the correct and incorrect predictions. This could also be utilized to help reject false positives.

4.4 Live Tests

In this final section, we discuss a simulated 'live' testing environment. The following test was performed by all group members. Since we are not able to visit a park in the allotted time frame, we opted to create our own. This park consisted of three rides. The rides are simulated by moving

the phone about in a specific pattern. We devised three specific patterns which are described in Table IV. Each ride performs three actions in order to create a unique action sequence. 5 circles is performed by the user's arm creating a full circle pivoting around the shoulder with the phone in that hand 5 times. 5 tosses means the user had to toss their phone in the air and catch it 5 times. 5 side/side means the users moved the phone across their body in left/right pattern 5 times. The actions were performed back to back without any interruption. For each ride, the action sequence was performed 10 times by a single group member to create the training data for the DTW algorithm.

Table IV: Custom Ride Actions and Results

Ride	Action 1	Action 2	Action 3	Avg. Reported Wait	%Error
1	5 Circles	5 Tosses	5 Side/Side	645 seconds	7.5%
2	5 Tosses	5 Circles	5 Side/Side	682 seconds	13.8%
3	5 Side/Side	5 Tosses	5 Circles	660 seconds	10.0%

To simulate a roller coaster's queue, paths were drawn in a parking lot using chalk. Each user was instructed to start at the entrance, place their phone in the front pocket and proceed to the first "ride". The entrance of each rides' queue was clearly marked. Upon entering the queue, the users were instructed to note their time. To simulate a line moving forward in small bursts, we wrote precise wait instructions on the ground. Each ride's queue had positions marked where the user would move forward then wait again for some allotted time. The wait times and physical length between wait points were different for each ride. Each rides wait time was set to 10 minutes to provide a consistent base. At the end of the queue, the user removed the mobile device from their pocket and performed the ride's action sequence. After this, the phone was placed back in their front pocket and the user proceeded to the next queue.

During this simulation, the entire system functionality was tested. Our application tracked users as they walked and waited within the "park". The application also tracked the wait times of each user and attempted to perform the matching based on the collected accelerometer data. After completion of the test, we analysed the results and found our application did very well. We were able to achieve 100% accuracy on the pattern matching for the rides. This means we were always able to detect which ride the user was on. It is worth noting that this test did not check detection of false positives. We were also able to detect the wait time with an average of 10% error. The error is because the end of the waiting is not detected until the user begins walking AFTER exiting the ride. This means the ride time will be included in the waiting time as well. In order to offset this error, we will populate our ride database with the official ride length (these are posted by every park on-line). Using this information, we can perform the ride matching then subtract the official ride time from the calculated wait time. There are also times in which the users must sit in the ride longer and wait before they are permitted to exit. This will also add incorrect time to the detected wait. This is a known issue which cannot be prevented with our current implementation. We may be able to mitigate the error by further analysing the accelerometer data and check the time elapsed from the end of the coaster to the beginning of the walking. Theme park guests are normally not concerned with exact wait times but a rough estimate (15 minute intervals) so the added time will not be a huge hindrance to our application performance. Due to this fact, the error mitigation is left to future work.

5. LIMITATIONS AND FUTURE WORK

We feel this project has very high promise, however it was developed during a single semester. Due to the time restrictions, we were not able to accomplish everything needed to make this project a fully working system. Below, we will discuss limitations of our current work and improvements which we intend to do as future work with the hopes of publishing our system on the Android Market.

5.1 Step Detection

Currently, we assume the phone is in the user's front pocket. We feel that this will cover a majority of the cases; however, there are other locations for the device such as back pockets and cargo pockets. We would like to handle these other locations to increase the detection rate. At the time of writing this paper, the only way to determine steps was via accelerometer data. It was discovered that the latest version of Android (KitKat 4.4) has added a new composite sensor with step detection capability. Since this version of android requires the latest devices, using it would greatly limit the number of devices which could run our application. For these reasons, its evaluation is left to future work.

5.2 Obsolete Data

It is a known issue that our application can only report data after it has occurred. This means our data is already obsolete at the time it has been reported. There is currently no way to solve this issue. Wait time is dependent on ride throughput which can be affected by many factors. A single person taking an extra few minutes to board the train causes everyone to wait longer. The only way to get a better idea of timing is to construct an estimation system. As our system grows we aim to monitor ride data. Eventually we can build an estimation system by observing previous wait times during similar conditions (day of week, time of day, month, etc). Using this information, we can use the reported wait times along with past data collected to create a better estimated wait time.

5.3 Full System

Currently, our system only performs the basic interactions needed. We need to implement a GUI which will allow users to view live data and see the ride wait times. This will require a real server to be set up. Currently, we are using the free version of Amazon EC2 which is limited. Some work needs to be completed before the application can be a success.

The biggest downfall of our system is that we could not properly test our system at a theme park. We plan to get a season pass to a theme park this summer to do a full system test and gather data. This will enable us to properly tune our algorithms to provide the best results for real data.

5.4 Device Capabilities

It was discovered during the final stages of testing that the Samsung Galaxy S2's accelerometer was only capable of detecting a maximum of 2G's. This means that it would never correctly match during the final live testing stage. Because of this issue, a check was put into place which will disable our tracking system if the accelerometer is not able to detect at least 5G's. We are currently unsure of how this will effect the amount of data which we will be able to collect.

6. CONCLUSION

In this paper, we present a passive crowd-participated roller coaster queue tracking system using mobile phones. Our system efficiently utilizes energy efficient mobile sensors which encourages and attracts participatory users. We comprehensively evaluate the system through a prototype system deployed on the Android platform with three types of mobile phones. Over a 2-day experiment period, the evaluation results demonstrate that our system can accurately detect wait times and classify coasters. The proposed system provides an energy efficient and cost effective approach to tracking wait times for a theme park which attracts participatory contribution by the community.

7. ACKNOWLEDGEMENT

Special thanks to Dr. Yung-Hsiang Lu for his help and guidance in the completion of this paper.

REFERENCES

Concepts, G. C. D. 2008. Raw accelerometer data. http://www.gcdataconcepts.com/ftp/WDW_rides.zip.

Corrado, H. Theme parks and waiting lines. http://web.archive.org/web/20140412161055/http://www.tejix.com/PaperQueueLines.html.

Dernbach, S. 2012. Simple and complex activity recognition through smart phones. Tech. rep., Whitworth University.

Gillian, N. 2013. Gesture recognition toolkit. http://web.archive.org/web/20140428172557/http://www.nickgillian.com/wiki/pmwiki.php?n=GRT.GestureRecognitionToolkit.

Group, T. 2013. Thinkwell group launches annual guest experience trend report. http://web.archive.org/web/20140408021002/http://thinkwellgroup.com/news/thinkwell-group-launches-annual-guest-experience-trend-report/.

Group, T. L. 2007. Q-bot ride reservation device. http://web.archive.org/web/20140408021514/http://www.legoland.co.uk/QBOT/.

Innetic, I. 2013. Ride hopper park wait times. http://web.archive.org/web/20140408021338/https://play.google.com/store/apps/details?id=com.ridehopper.

Jeffers, G. 2012. Global attractions attendance report. Tech. rep., Themed Entertainment Association (TEA).

Kuklin, P. How people-counting technology can change the game in amusement park queuing. http://web.archive.org/web/20140408021849/http://blog.lavi.com/2013/07/16/people-counting-technology/.

Kwapisz, J. R., Weiss, G. M., and Moore, S. A. 2010. *Activity recognition using cell phone accelerometers*. Vol. 12. SIGKDD Explorations.

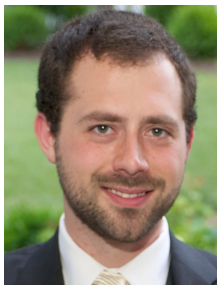
Muscillo, R., Conforto, S., Schmid, M., Caselli, P., and D'Alessio, T. 2007. Classification of motor activities through derivative dynamic time warping applied on accelerometer data. In *Proceedings of the 29th Annual International Conference of the IEEE EMBS*. IEEE.

Niles, R. 2006. Disney world's wait time system. http://web.archive.org/web/20140412160848/http://www.themeparkinsider.com/news/response.cfm?ID=3062.

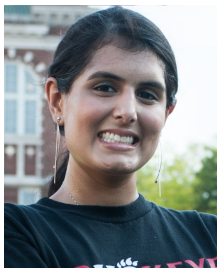
Niles, R. 2014. Disney world's wait time system. http://web.archive.org/web/20140412160957/http://www.themeparkinsider.com/news/response.cfm?ID=945509941.

Zhou, P., Zheng, Y., and Li, M. 2012. How long to wait?: Predicting bus arrival time with mobile phone based participatory sensing. In *MobiSys*.

Josh Fuerst is an Edison Engineer at GE Aviation in Evendale, Ohio. Josh received his M.S. in Computer Science from Purdue University with a research focus on security. He received his B.S. in Computer Engineering from the University of Cincinnati. In his spare time, he likes to involve himself in personal projects such as building a UC Coffee table out of coke bottle caps and small projects with the raspberry pi.



Rachna Fuerst is an Edison Engineer at GE Aviation in Evendale, Ohio. Rachna received her M.S. in Computer Science from Purdue University with a research focus on cryptography. She received her B.S. in Computer Engineering from the University of Cincinnati. In her spare time, she loves to play bridge and tennis.



Jong Char is currently pursuing his M.S. in Computer Science at Purdue University, where he also obtained his B.S. degree in Computer Science. He is also currently working full time as a Software Architect at Day Research. His areas of interest are machine learning and data mining.