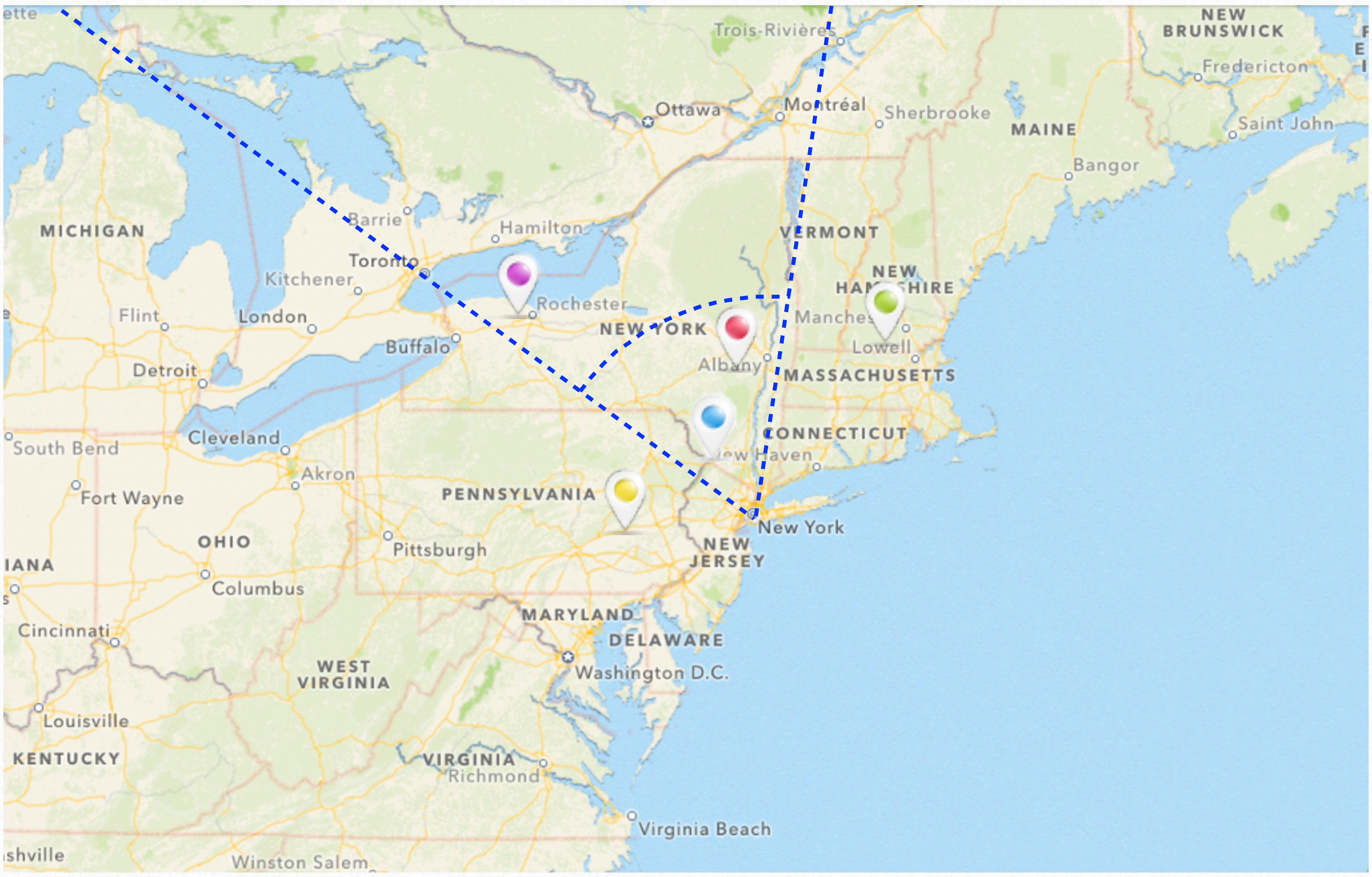
Dalmo Cirne
dalmo.cirne@gmail.com

Augmented Reality Geolocation Math

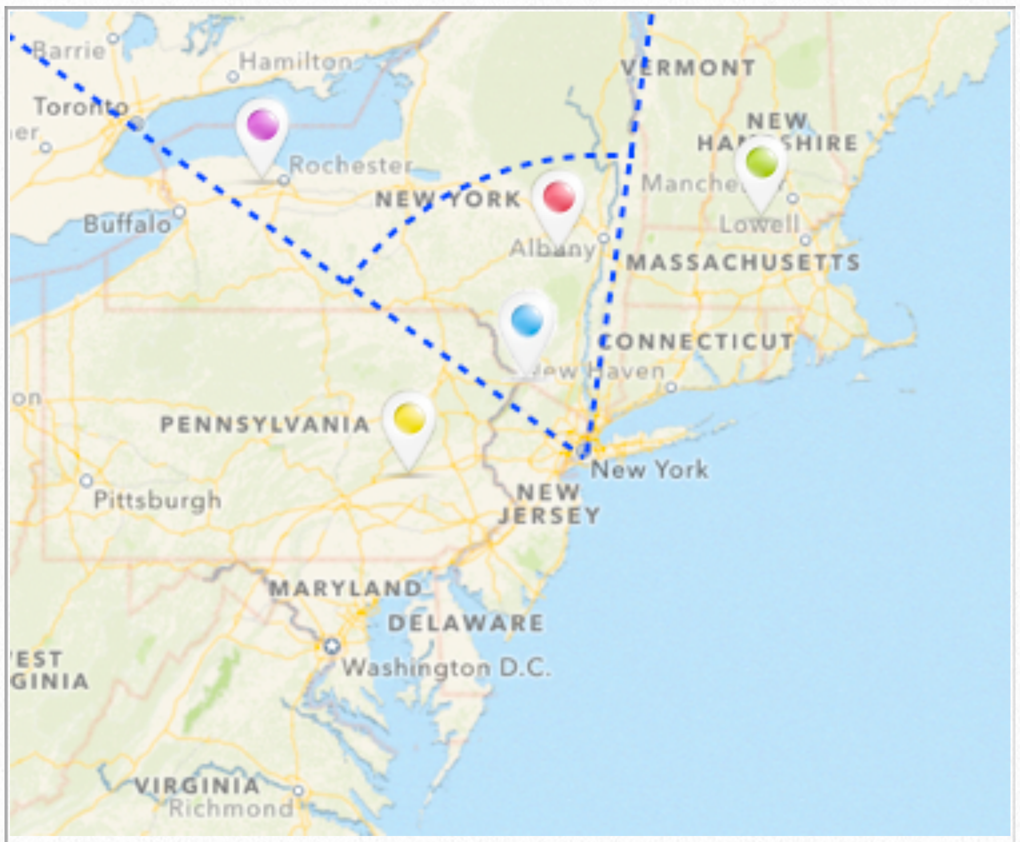


Introduction

Augmented reality combined with modern smartphones equipped with cameras, magnetometers, and the ability to display maps on the screen make a powerful tool that can be used in a plethora of different applications. Here we are going to explain how to build one such kind of application. We will learn how to put geolocation based annotations on the screen, taking into account where a device is, which direction it is pointing, and how far place marks of interest are from it–known as geolocation augmented reality. We will explore the math that goes behind such application, and also learn about a reference implementation for iOS.

Although the reference implementation is for iOS, the concepts presented here are transferable to any other capable device or platform.

The use case we will be developing here is of a user standing at a location with longitude a_x and latitude a_y and pointing the back of her device to a direction making an angle α with a line parallel to the latitude lines. The device will have a camera with viewing angle ϕ and a maximum field of interest, from the observer's perspective, denoted by radius r. We want to overlay annotations on the device's screen of all place marks that are situated within the visible area. Furthermore, we want to update the annotations every time the user changes location, rotates the device, or changes the radius of the region of interest.



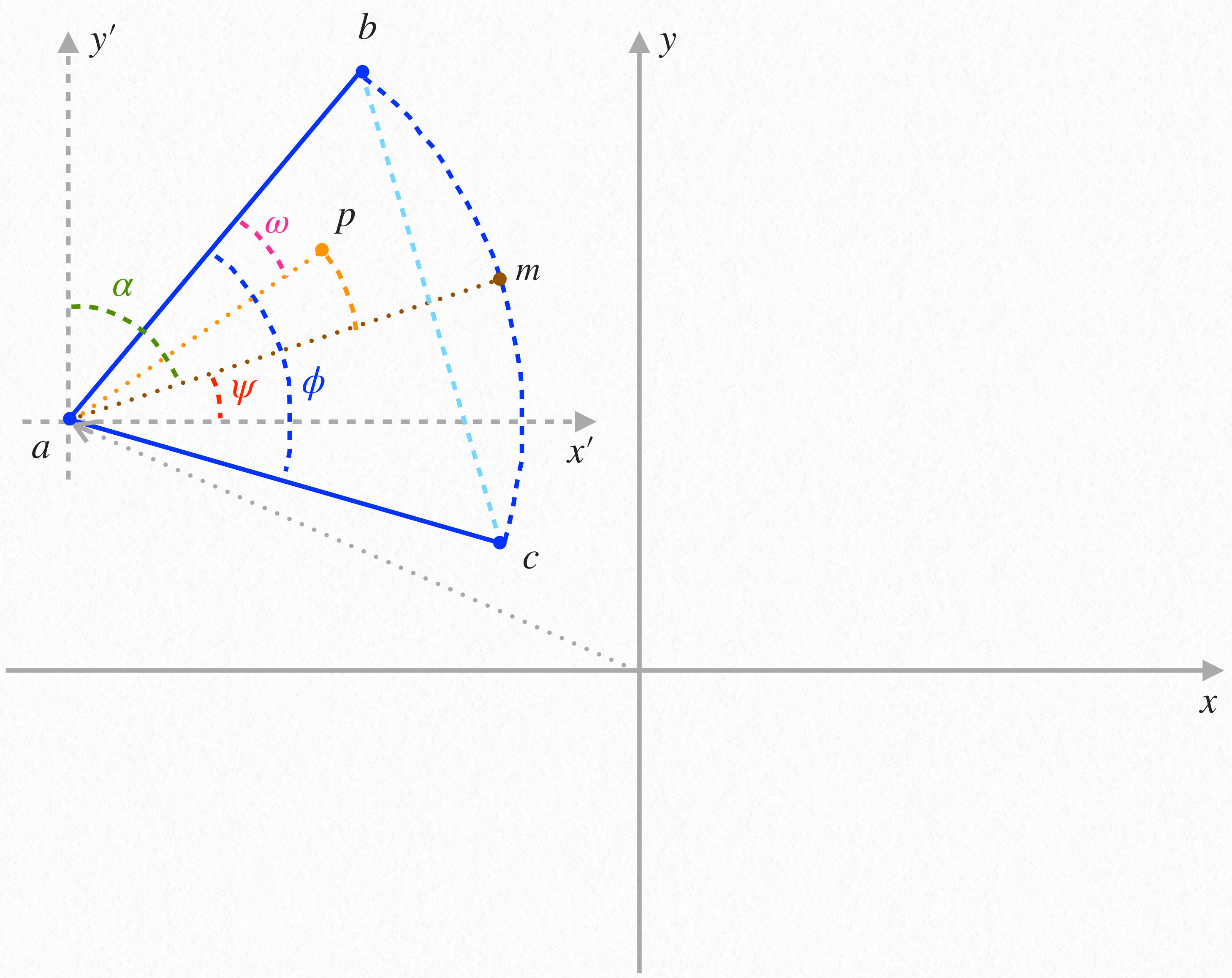
Blue and Red place marks are visible, all others are not.

The viewing angle ϕ of the device will depend whether its orientation is portrait or landscape. For our calculations we are using:

$$\phi = \begin{cases} \frac{\pi}{3} & \text{if device orientation is landscape} \\ \frac{\pi}{4} & \text{if device orientation is portrait} \end{cases}$$

Although there may be some discussion whether those angles are exact. They are pretty close to the actual angle values and perfectly good for this application.

Before we can proceed we need to find a way to convert longitude and latitude–which are expressed in degrees–to miles (or kilometers). And since the Earth's shape is not a circle, we can expect to have two different conversions: one for latitude and one for longitude.



The Earth's radius at the Equator Line is 3,956.547 miles. Let's denote it by s.

$$s = 3{,}956.547 \text{ miles}$$

Converting latitudes to miles is be simpler since we can approximate latitude lines to an arc of a circle. We know how to calculate the circumference of a circle ($2\pi r$), and we also know a circle has 360°. Thus, if we divide the Earth's circumference by 360 degrees we will have an approximation to miles per degree of latitude t. The accuracy of this approximation is sufficient for our application.

$$t = \frac{2\pi s}{360^\circ} \text{ Miles Per Degree Of Latitude}$$

Calculating miles per degree of longitude g will be not as straightforward. Walking one degree of longitude alongside the Equator Line means walking a greater distance than walking one degree of longitude closer to the poles.

$$g = t\cos\left(a_y \frac{\pi}{180^\circ}\right) \text{ Miles Per Degree Of Longitude}$$

Our user is standing at:

$$a = \begin{bmatrix} a_x \\ a_y \end{bmatrix} = \begin{bmatrix} \text{device's longitude} \\ \text{device's latitude} \end{bmatrix}$$

A user will be pointing the device to a direction, and this direction will make an angle α with the true north (*heading*). Now, this angle α will be measured against the latitude lines, which are vertical, and equivalent to the y axis or any of its parallels in a Cartesian plane. However in a Cartesian plane it is common practice to measure angles against the x axis, therefore we need to convert the angle α to an angle ψ, which will give us the heading relative to an horizontal line parallel to the x axis.

$$\psi = \frac{\pi}{2} - \alpha$$

And since now we know how to convert from degrees to miles and vice-versa, we can calculate points b and c given a distance of interest r .

$$b = \begin{bmatrix} b_x \\ b_y \end{bmatrix} = \begin{bmatrix} \frac{r}{g}\cos\left(\psi + \frac{\phi}{2}\right) + a_x \\ \frac{r}{t}\sin\left(\psi + \frac{\phi}{2}\right) + a_y \end{bmatrix}$$

$$c = \begin{bmatrix} c_x \\ c_y \end{bmatrix} = \begin{bmatrix} \frac{r}{g}\cos\left(\psi - \frac{\phi}{2}\right) + a_x \\ \frac{r}{t}\sin\left(\psi - \frac{\phi}{2}\right) + a_y \end{bmatrix}$$

Given that we have a set of place marks, each one with its own longitude and latitude. Our challenge is to determine which ones would fall within the region of interest, therefore be visible and have a corresponding annotation overlaid on the screen. We will need to iterate over the set of place marks and for each place mark:

$$p = \begin{bmatrix} p_x \\ p_y \end{bmatrix} = \begin{bmatrix} \text{placemark's longitude} \\ \text{placemark's latitude} \end{bmatrix}$$

We will need to calculate whether or not its longitude and latitude coordinates are within the region of interest.

We have that the projection of vector $\overrightarrow{ap}$ on vector $\overrightarrow{ab}$ can be calculated by:

$$\text{proj}_{\overrightarrow{ab}}\overrightarrow{ap} = \|\overrightarrow{ap}\|\cos\omega$$

Also, the dot product between $\overrightarrow{ap}$ and $\overrightarrow{ab}$ is:

$$\overrightarrow{ap}\cdot\overrightarrow{ab} = \|\overrightarrow{ab}\|\|\overrightarrow{ap}\|\cos\omega$$

Therefore we have that:

$$\text{proj}_{\overrightarrow{ab}}\overrightarrow{ap} = \frac{\overrightarrow{ap}\cdot\overrightarrow{ab}}{\|\overrightarrow{ab}\|}$$

We need to express $\text{proj}_{\overrightarrow{ab}}\overrightarrow{ap}$ as a coordinate of vector $\overrightarrow{ab}$, or in other words, calculate an eigenvalue for it. That is achieved by dividing it by the norm of $\overrightarrow{ab}$. Let's call this eigenvalue λ.

$$\lambda = \frac{\text{proj}_{\overrightarrow{ab}}\overrightarrow{ap}}{\|\overrightarrow{ab}\|} \Rightarrow \lambda = \frac{\overrightarrow{ap}\cdot\overrightarrow{ab}}{\|\overrightarrow{ab}\|^2}$$

Now we need to calculate the projection of vector $\overrightarrow{ap}$ on vector $\overrightarrow{ac}$:

$$\text{proj}_{\overrightarrow{ac}}\overrightarrow{ap} = \frac{\overrightarrow{ap}\cdot\overrightarrow{ac}}{\|\overrightarrow{ac}\|}$$

Dividing the projection by the norm of vector $\overrightarrow{ac}$ given us an eigenvalue and allows us to express its coordinate in terms of $\overrightarrow{ac}$. Let's call this eigenvalue σ.

$$\sigma = \frac{\text{proj}_{\overrightarrow{ac}}\overrightarrow{ap}}{\|\overrightarrow{ac}\|} \Rightarrow \sigma = \frac{\overrightarrow{ap}\cdot\overrightarrow{ac}}{\|\overrightarrow{ac}\|^2}$$

With λ and σ in hand we can proceed to the last step in determining whether a place mark lies within the visible region of interest.

The arc of circumference determining the distance boundary can be calculated using the Pythagorean theorem:

$$a^2 + b^2 = c^2$$

or in our case:

$$(\lambda\overrightarrow{ab})^2 + (\sigma\overrightarrow{ac})^2 = r^2$$

However the length of vectors $\overrightarrow{ab}$ and $\overrightarrow{ac}$ have the same length, and are equal in length to the radius r. Thus, in order to simplify the equation we can divide both sides by r^2.

$$\lambda^2\frac{\cancel{\overrightarrow{ab}^2}}{\cancel{r^2}} + \sigma^2\frac{\cancel{\overrightarrow{ac}^2}}{\cancel{r^2}} = \frac{\cancel{r^2}}{\cancel{r^2}}$$

Leaving us with:

$$\lambda^2 + \sigma^2 = 1$$

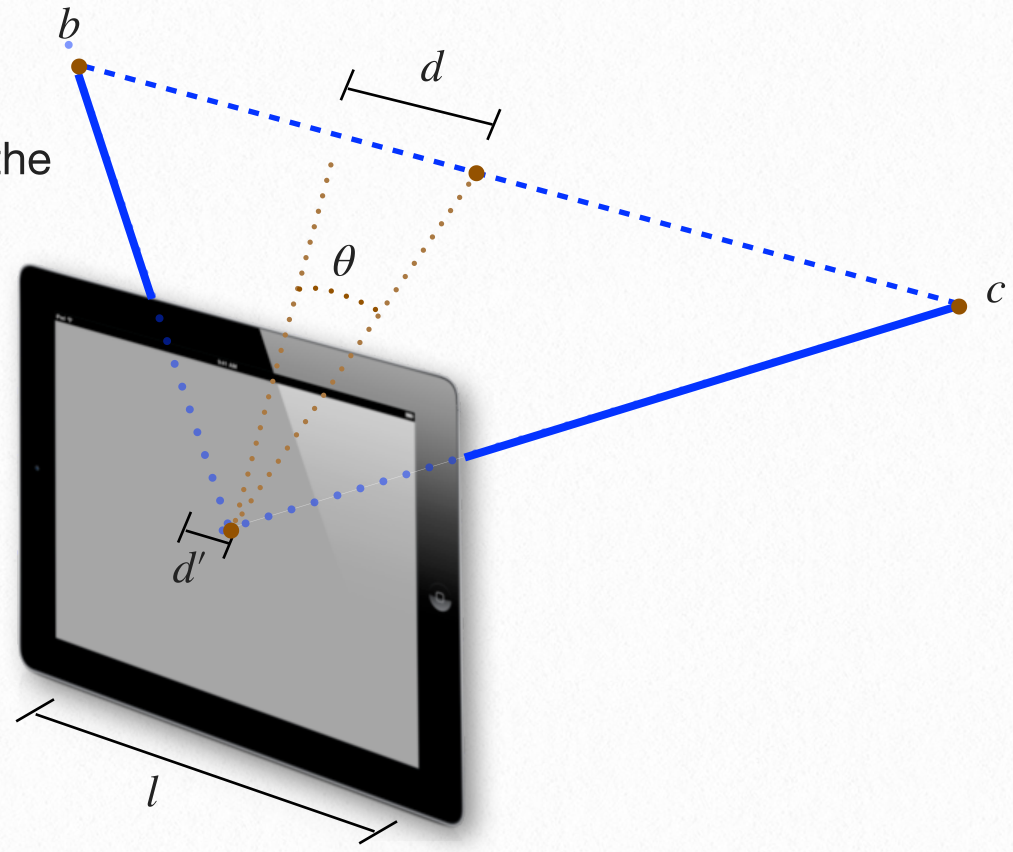
Now we can determine whether a place mark p is within the region of interest, therefore is visible. The eigenvalues λ and σ must be positive numbers (otherwise the place mark would be located behind the observer) and when combined, using the Pythagorean theorem, they have to be smaller or equal to 1, so they are not farther than the boundary set by the radius arc segment.

$$p(\lambda, \sigma) = \begin{cases} \text{Visible} & \forall\ (\lambda > 0) \wedge (\sigma > 0) \wedge (\lambda^2 + \sigma^2 \leq 1) \\ \text{Not Visible} & \text{otherwise} \end{cases}$$

Our next step is to translate what we have developed thus far into device (e.g. iPhone, iPod Touch, iPad or iPad Mini) equations and coordinates.

Overlaying Augmented Reality Annotations on the Device's Screen

So far we have calculated which place marks are visible. Now we need to calculate their respective coordinates, sizes, and perspectives on the device's screen.



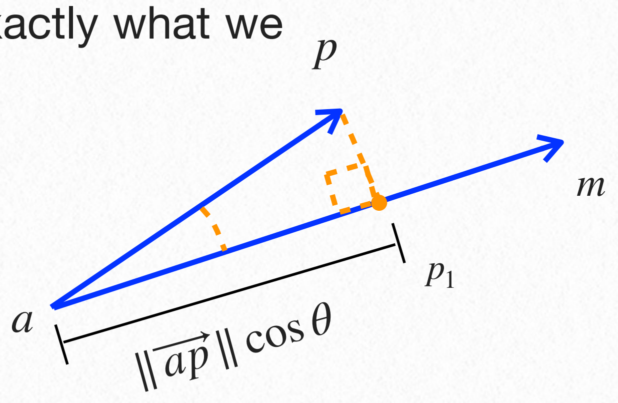
If we calculate the mid-point m of the boundary arc segment and determine the vector $\overrightarrow{am}$, we can calculate the distance d, to represent how far a place mark p is from the vector $\overrightarrow{am}$. Point m would be equivalent to the center of the device's screen, and points b and c equivalent to the screen's left and right margins. Thus, once having d computed we can determine its proportionally equivalent d' on the screen.

$$m = \begin{bmatrix} \frac{r}{g}\cos\psi + a_x \\ \frac{r}{t}\sin\psi + a_y \end{bmatrix}$$

We know that the dot product between two vectors is equal to the product of their Norms times the cosine of the angle θ between them:

$$\overrightarrow{am} \cdot \overrightarrow{ap} = \|\overrightarrow{am}\| \|\overrightarrow{ap}\| \cos\theta$$

The angle θ is the only unknown in the equation, and is exactly what we need to discover in order to calculate distance d.



$$\theta = \arccos\left(\frac{\overrightarrow{am} \cdot \overrightarrow{ap}}{\|\overrightarrow{am}\|\|\overrightarrow{ap}\|}\right)$$

Now that we know θ we can proceed to calculate d:

$$d = \|\overrightarrow{ap}\| \sin\theta$$

The length of the vector $\overrightarrow{bc}$ is equivalent to the length l of the screen. With that information we can calculate d':

$$\frac{d}{\|b-c\|} = \frac{d'}{l} \Rightarrow d' = \frac{ld}{\|b-c\|}$$

For each place mark we will need to calculate a coordinate (x, y) representing the center of the augmented reality annotation (annotation for short), and dimensions (w, h) representing its *width* and *height*, respectively.

Having calculated the distance d' above, we can compute coordinate x:

$$x = \frac{l}{2} + d'$$

In the equation above, $\frac{l}{2}$ gives us the middle of the screen on the longitudinal (Cartesian x) axis, and d' tells us how far from it our annotation should be from it. This leaves us with y, w, h yet to be calculated.

However, before we continue forward, let me introduce a scale factor s varying from $[0,1]$ that will become very important determining the remaining unknowns.

When plotting annotations on the screen, it would be nice to draw them smaller and closer to the top of the screen if a place mark is farther from the observer, and draw them larger and closer to the bottom of the screen if a place mark is closer to the observer. This is where the scale factor s becomes important.

We can define an annotation's maximum size to have width $defaultWidth$ and height $defaultHeight$, then by multiplying its size by the scale factor s we can make the size of the annotation drawn on the screen to be inversely proportional to the place mark's distance to the observer (which is the length of vector $\overrightarrow{ap}$), behaving as we described in the previous paragraph.

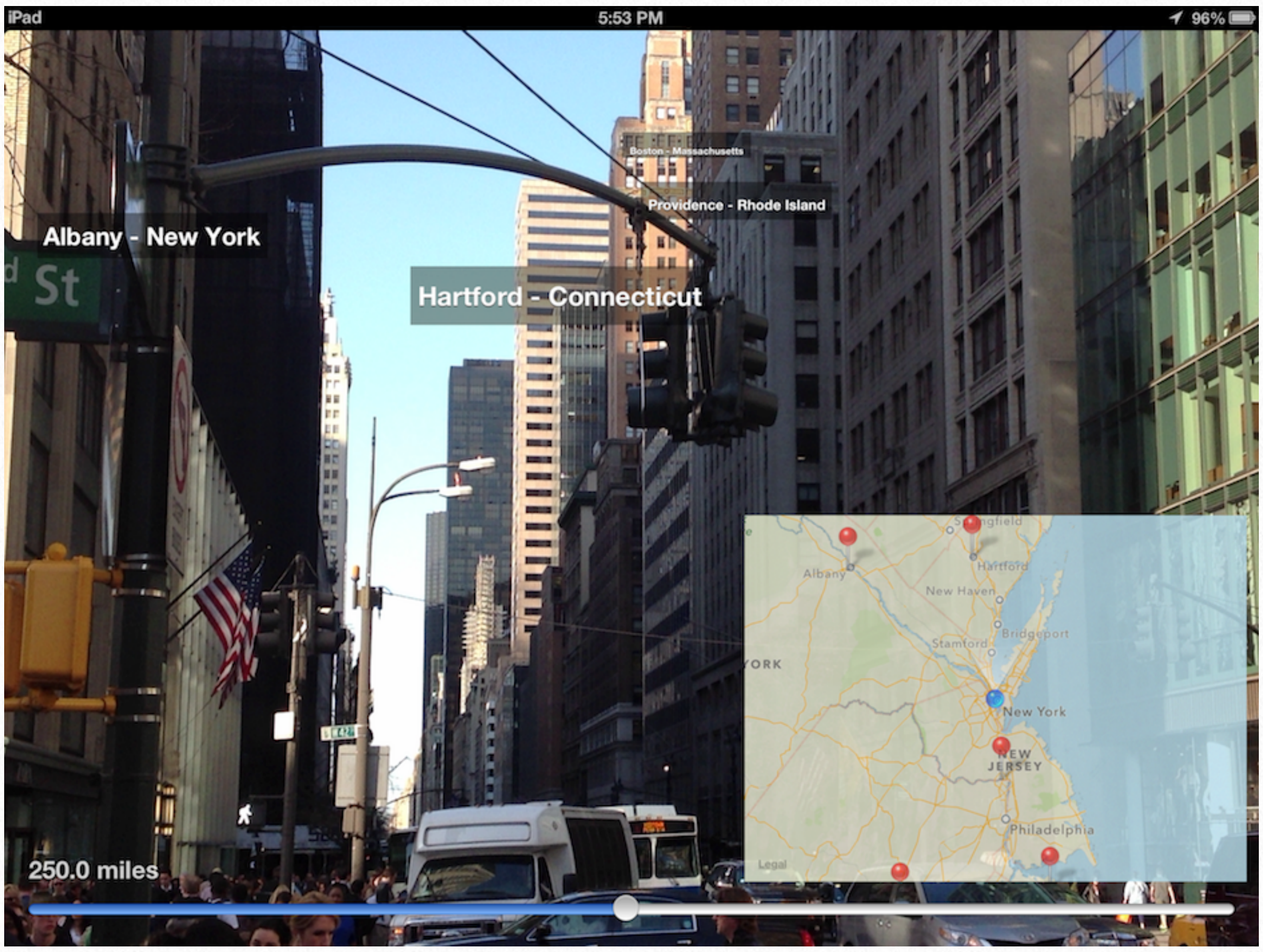
The scale factor s is calculated by:

$$s = 1 - \frac{\|\overrightarrow{ap}\|}{r}$$

And the dimension (w, h) of a place mark are given by:

$$w = s \times defaultWidth$$

$$h = s \times defaultHeight$$



The same idea we used for the dimensions of an annotation can be carried over to determine its y coordinate. We just need to establish a maximum y coordinate $(yMax)$ on the screen and multiply it by the scale factor s.

$$y = s \times yMax$$

As you can see from the screenshot, the annotations closer to the observer appear in larger size and closer to the bottom of the screen, and as distances from the observer grow, the annotations get smaller and closer to the top of the screen.

Reference Implementation

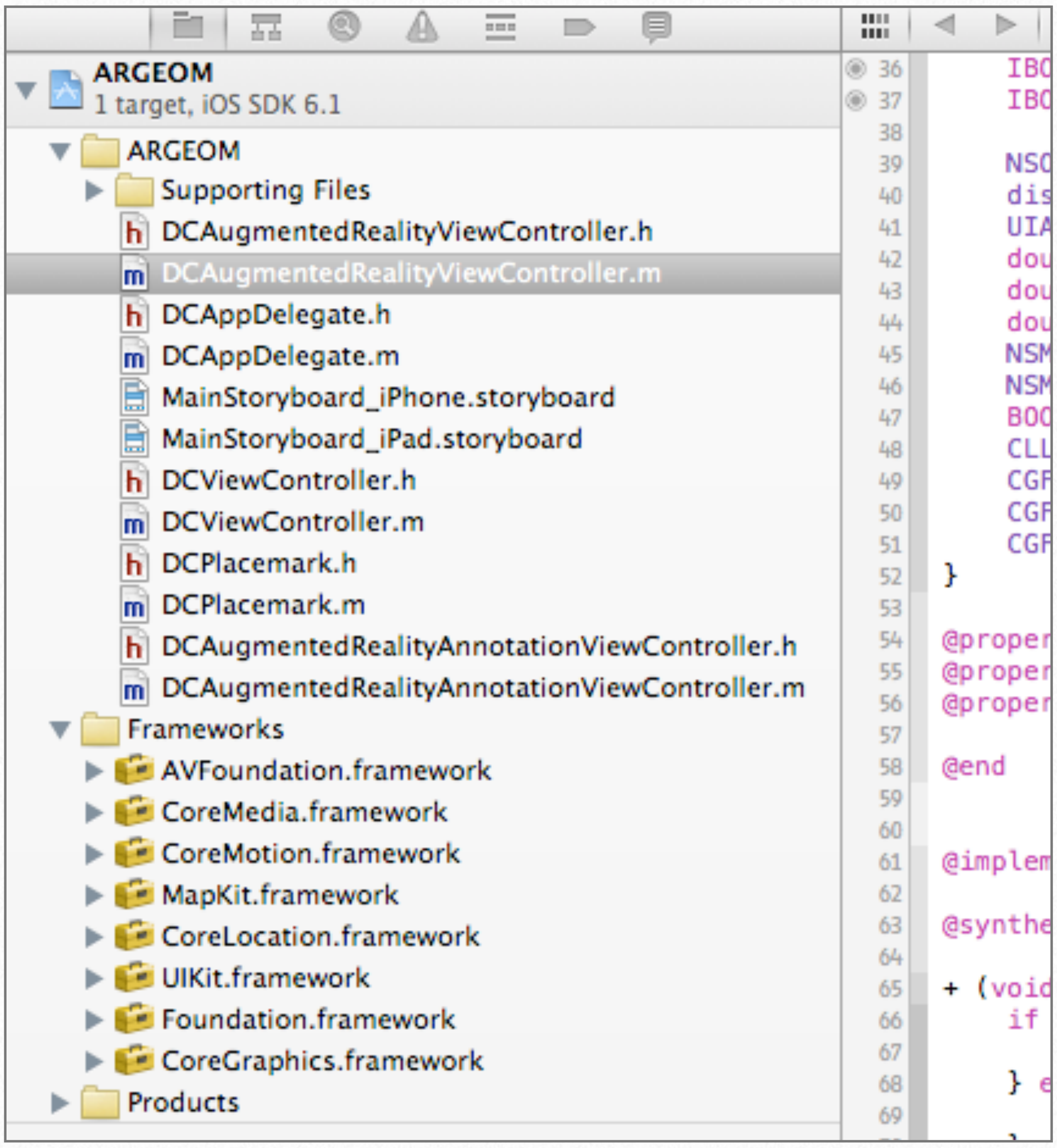
A reference implementation for iOS is available in GitHub. You can clone or fork it from:

https://github.com/dcirne/ARGEOM

Let's begin by taking a look at the source code files and explaining a little about of what each of them do.

As you can see, we are using some of *Cocoa Touch*'s frameworks in this project. For example:

- *AVFoundation* to capture the input from the device's camera.
- *MapKit* to place a map on the screen.
- *CoreMotion* to detect device movement and acceleration (including the gesture to tell whether the device is parallel or perpendicular to the floor.)
- *CoreLocation* for user location and device *heading*.



```
ARGEOM
1 target, iOS SDK 6.1
  ARGEOM
    Supporting Files
    DCAugmentedRealityViewController.h
    DCAugmentedRealityViewController.m
    DCAppDelegate.h
    DCAppDelegate.m
    MainStoryboard_iPhone.storyboard
    MainStoryboard_iPad.storyboard
    DCViewController.h
    DCViewController.m
    DCPlacemark.h
    DCPlacemark.m
    DCAugmentedRealityAnnotationViewController.h
    DCAugmentedRealityAnnotationViewController.m
  Frameworks
    AVFoundation.framework
    CoreMedia.framework
    CoreMotion.framework
    MapKit.framework
    CoreLocation.framework
    UIKit.framework
    Foundation.framework
    CoreGraphics.framework
  Products
```

This project also uses *Storyboards* to represent user interfaces (iPhone and iPad), and *ARC* for memory management. Last but not least, the deployment target is iOS 5.0 or greater.

If you already have the project open in Xcode, you should be able to select a device and hit *Run* to see it working.

DCViewController is the root view controller of this app and also has the responsibility of loading the collection [NSArray] of place marks. There are two choices of place marks collections to be loaded, and it is very easy to pick which one you want. The first one is calling the method:

```
- (NSArray *)loadPlacemarks;
```

In its implementation you can see a simplistic way of entering a collection of place marks. You can manually enter title, subtitle, latitude, and longitude.

The other choice is to call the method:

```
- (void)loadPlacemarks:(PlacemarksLoaded)completionBlock;
```

This method loads the list of U.S. State Capitals from a CSV file in a background thread, parses it, allocates an instance of **DCPlacemark** for each capital, and adds it to the collection of place marks. Once the collection is complete, it invokes the completion block on the main thread passing all loaded place marks as parameter.

In order to keep the selection easy, you can comment out the compiler directive

```
#define USE_EXTERNAL_PLACEMARKS
```

comment it out if you want the simple, manual collection of place marks; or uncomment it if you want the list of U.S. State Capitals.

DCPlacemark contains the representation of a place mark, including its longitude, latitude, title and other properties.

DCAugmentedRealityAnnotationViewController is used to overlay an augmented reality annotation on the device's screen.

DCAugmentedRealityViewController is the place where most of the calculations happen. Using it is as simple as presenting the view controller and calling its start method passing a collection [NSArray] of **DCPlacemark**s as parameter:

```
- (void)startWithPlacemarks:(NSArray *)placemarks;
```

When you hold your device parallel to the floor you will see the standard iOS map view from MapKit and the place marks as pins on the map. However, if you move your arm to hold the device perpendicular to floor, the map is resized, placed at the bottom-right corner, and the augmented reality mode gets started. Depending on the direction you are pointing the device to, and the distance set in the slider, you will see annotations appearing on the screen.

The program uses *CoreMotion* to monitor the device's motion and detect whether it is parallel or perpendicular to the floor. Then we use this information to start/stop the augmented reality visualization mode. This is handled by the following methods:

```
- (void)startMonitoringDeviceMotion;
- (void)stopMonitoringDeviceMotion;
- (void)handleDeviceAcceleration:(CMAccelerometerData *)accelerometerData
                            error:(NSError *)error;
```

One important point worth mentioning is that by default iOS devices assume the top of the screen, in portrait mode, to represent due north. Thus, if we are holding a device in any other orientation, or if we rotate the device, we need to set the heading orientation to the appropriate value. This is accomplished in app by the method:

```
- (void)updateLocationManagerHeadingOrientation;
```

Now, probably the most important method in this class, and highly likely in the whole project, is the one where majority of the calculations described in this paper are performed. Let's take a moment to dive deeper look into the code:

```
- (void)calculateVisiblePlacemarksWithUserLocation:(CLLocation *const)location
                                           heading:(CLHeading *const)heading
                                   completionBlock:(PlacemarksCalculationComplete)completionBlock {
    .
    .
    .
    // Calculations are performed on a separate thread in a serial queue
    dispatch_async(placemarksQueue, ^{
        // Blocks to perform vector operations
        double(^dotProduct)(double *, double *) = ^(double *vector1, double *vector2)...
        double(^norm)(double *) = ^(double *vector)...
        void(^makeVector)(double **, CGPoint, CGPoint) = ^(double **vector, CGPoint point1,
        CGPoint point2)...
        CLLocationDistance(^calculateDistanceBetweenPoints)(CGPoint, CGPoint) = ^(CGPoint
        point1, CGPoint point2)...
        .
        .
        .
        // Loops through place marks calculating which ones are visible and which
        // ones are not
        for (DCPlacemark *placemark in self.placemarks) {
            pointP = CGPointMake(placemark.coordinate.longitude,
                                 placemark.coordinate.latitude);
            makeVector(&vectorAP, pointA, pointP);

            lambda = dotProduct(vectorAP, vectorAB) / pow(norm(vectorAB), 2);
            sigma = dotProduct(vectorAP, vectorAC) / pow(norm(vectorAC), 2);

            if ((lambda > 0) && (sigma > 0) && (pow(lambda, 2) + pow(sigma, 2) <= 1)) {
                thetaDirection = calculateDistanceBetweenPoints(pointB, pointP) <=
                                 calculateDistanceBetweenPoints(pointC, pointP) ? -1.0 : 1.0;
                theta = acos(dotProduct(vectorAM, vectorAP) / (norm(vectorAM) *
                             norm(vectorAP))) * thetaDirection;
                dPrime = l * norm(vectorAP) * sin(theta) / norm(vectorBC);
                distanceFromObserver = [placemark
                                        calculateDistanceFromObserver:location.coordinate];
                scale = 1.0 - distanceFromObserver / distance;

                placemark.bounds = CGRectMake(0,
                                              0,
                                       defaultAugmentedRealityAnnotationSize.width * scale,
                                       defaultAugmentedRealityAnnotationSize.height * scale);
                placemark.center = CGPointMake(lOver2 + dPrime, yMax * scale);

                [visiblePlacemarks addObject:placemark];
            } else {
                [nonVisiblePlacemarks addObject:placemark];
            }

            free(vectorAP);
        }
        .
        .
        .
        dispatch_async(dispatch_get_main_queue(), ^{
            completionBlock([visiblePlacemarks copy], [nonVisiblePlacemarks copy]);
        });
    }
}
```

This method calculates all visible and non-visible place marks in a background serial dispatch queue and when finished it dispatches `completionBlock` on the main queue.

In this reference implementation the completion block contains a call to a single method. This method overlays the visible annotations on the screen and also remove the ones no longer visible.

```
 - (void)overlayAugmentedRealityPlacemarks:(NSArray *const)visiblePlacemarks
                       nonVisiblePlacemarks:(NSArray *const)nonVisiblePlacemarks;
```

We have covered a lot of ground in this paper. We have learned the math necessary to determine whether a place mark is within a region of interest, we went one step further and translated it to fit inside a device's screen, and last but not least we saw a reference implementation for iOS.

I hope you have enjoyed reading this paper and playing with the reference implementation as much as I did writing them.

Dalmo Cirne
dalmo.cirne@gmail.com
http://dalmocirne.blogspot.com

References

Dot Product. (n.d.). In *Wikipedia*. Retrieved November 13, 2012, from
http://en.wikipedia.org/wiki/Dot_product#Geometric_interpretation

Norm. (n.d.). In *Wikipedia*. Retrieved November 9, 2012, from
http://en.wikipedia.org/wiki/Norm_(mathematics)

Unit Vector. (n.d.). In *Wikipedia*. Retrieved November 9, 2012, from
http://en.wikipedia.org/wiki/Unit_vector

Earth Radius. (n.d.). In *Wikipedia*. Retrieved January 27, 2013, from
http://en.wikipedia.org/wiki/Earth_radius

Longitude. (n.d.). In *Wikipedia*. Retrieved January 27, 2013, from
http://en.wikipedia.org/wiki/Longitude

Latitude. (n.d.). In *Wikipedia*. Retrieved January 27, 2013, from
http://en.wikipedia.org/wiki/Latitude