ITIS "E.MATTEI " - Sondrio

SARplay

Un’app che mostra le deformazioni del terreno sulla base di dati satellitari

Introduzione

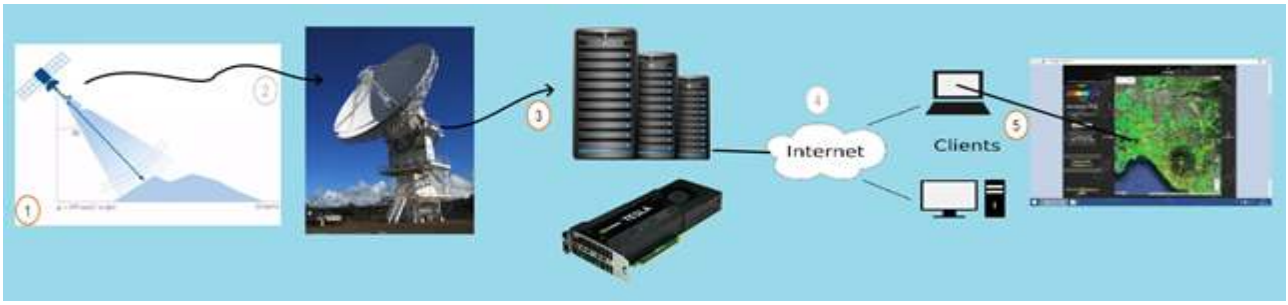
L'applicativo "SARplay" nasce anche grazie a due eventi che si sono svolti presso l'I.T.I.S. "E.Mattei" di Sondrio, durante l'anno scolastico 2015-2016:

· il primo riguarda la collaborazione della nostra scuola con la software house "Visuality Software srl", la quale, in cerca di una sede a Morbegno e di nuovo personale, si è offerta di istruire e trasmettere la sua esperienza nel mondo Apple agli studenti interessati;

· il secondo evento vede coinvolto l'ex professore di Informatica Sabatino Bonanno, il quale, dopo aver insegnato per alcuni anni presso l'I.T.I.S. di Sondrio e il Liceo Scientifico "C. Donegani", è tornato a Napoli, dove attualmente lavora per il CNR-IREA (Consiglio Nazionale delle Ricerche) e l'Università di Roma, e si occupa di elaborazione di immagini satellitari, riguardanti la deformazione della crosta terreste nel tempo

Uno dei compiti dell'istituto IREA (Istituto per il Rilevamento Elettromagnetico dell'Ambiente, dipartimento del CNR) è quello di raccogliere ed elaborare i dati dei satelliti, che vengono poi rappresentati su una mappa, in maniera che essa possa fornire informazioni visive immediate su punti di interesse del territorio. La mappa viene messa a disposizione degli utenti chiedendo l'autorizzazione al CNR, ed è consultabile tramite un web server; tuttavia, la fluidità dell'interfaccia web dipende molto dalla connessione di cui si dispone ed inoltre è parecchio complicata da utilizzare; pertanto un utente medio non sarebbe in grado di farne un buon uso. L'unione di queste due problematiche ci ha spinti a pensare ad un'app per iPad semplice ed intuitiva, da realizzare insieme alla Visuality Software e con la collaborazione dell'IREA

Per poter visualizzare il movimento del territorio nelle zone specificate e messe a disposizione dal CNR occorre una serie di passaggi schematizzabili con lo schema sotto rappresentato.



Dallo schema si evidenziano le 5 fasi che il processo richiede:

1. La fase di acquisizione dei dati è affidata al satellite, il quale esegue i rilevamenti tramite l'uso di radar SAR. I sensori SAR sono associati a specifiche bande dello spettro elettromagnetico. Nelle applicazioni InSAR le bande comunemente utilizzate sono la banda L (frequenza 1-2 GHz, lunghezza d'onda ~24 cm), la banda C (frequenza 5-6 GHz, lunghezza d'onda ~6 cm) e la banda X (frequenza 8-12 GHz, lunghezza d'onda ~3 cm). Il principio di funzionamento di questi sensori è il seguente: un antenna trasmittente propaga nello spazio un'onda elettromagnetica che, incidendo sulla superficie terrestre, subisce un fenomeno di riflessione. Una parte del campo diffuso torna verso la stazione trasmittente, equipaggiata per la ricezione, dove vengono misurate le sue caratteristiche. Il dispositivo è in grado di individuare il bersaglio elettromagnetico (funzione di detecting) e, misurando il ritardo temporale tra l'istante di trasmissione e quello di ricezione, valutare la distanza a cui è posizionato, localizzandolo in modo preciso lungo la direzione di puntamento dell'antenna (direzione di range). Il segnale

radar relativo ad un bersaglio è caratterizzato da due valori: ampiezza e fase. Questi valori permettono di realizzare due immagini. La fase in particolare racchiude l'informazione più importante ai fini delle applicazioni interferometriche.

2. Dopo aver raccolto i vari dati questi verranno inviati ad apposite antenne satellitari a terra. Queste faranno da intermediari con i server del CNR, ai quali saranno inviate tutte le informazioni in formato stringa e ancora da elaborare.
3. Il compito dei server è quello di prendere le immagini satellitari ed elaborarle. L'elaborazione consiste nel rappresentare un immagine detta interferometria. L'interferometria è la misurazione delle variazioni della fase del segnale SAR tra due acquisizioni distinte. Una seconda elaborazione consiste invece nella trasformazione dell'interferometria in formato stringa, in modo da rendere semplice la gestione dell'immagine. Questo intero processo, totalmente automatizzato tramite l'uso di programmi in C, è realizzato tramite l'utilizzo di una architettura hardware in parallelo, chiamata CUDA (Compute Unified Device Architecture), sviluppata da NVIDIA. Essa permette di raggiungere alte prestazioni di computing grazie alla potenza di calcolo delle GPU. Nel caso analizzato vengono utilizzate schede video tesla k20.
4. I server oltre che ad occuparsi dell'elaborazione dei dati svolgono anche ruolo di WEB Server e si occupano di gestire una porzione del sito del CNR con il compito di mostrare i dati raccolti.
5. Tramite internet e con l'utilizzo di un browser si è in grado di collegarsi col CNR e visualizzare così su una mappa google i dati sotto forma di punti. Tramite diverse graduazioni di colore si può capire il movimento del terreno.

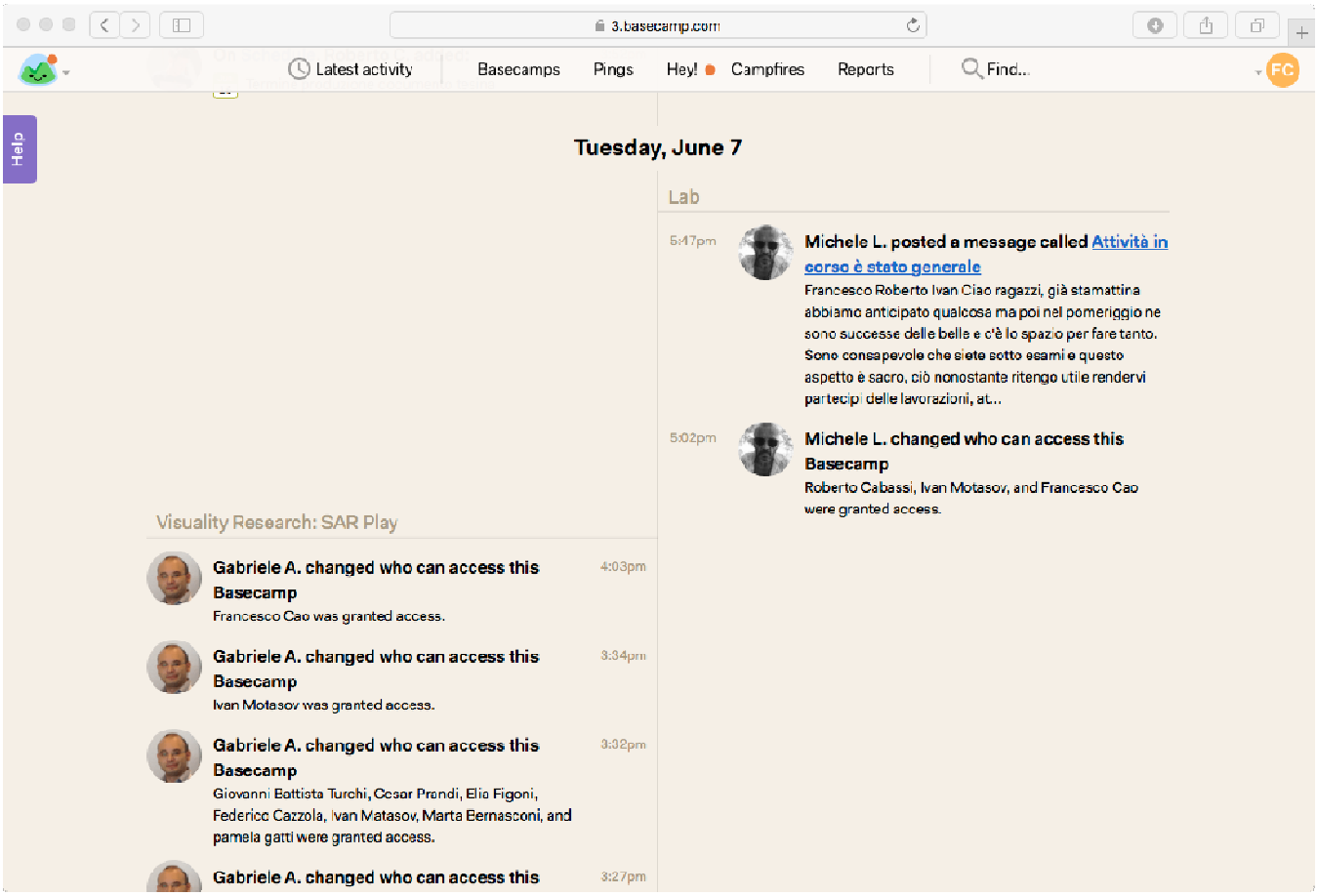
L’app è formata dai seguenti elementi:

- connettore, ovvero una classe che si interpone tra server e applicazione; il suo compito è quello di scaricare un file json dal server, tramite una richiesta http
- database, contenente tutti i punti scaricati dal server tramite il connettore. È formato dalle entità “point”, “zone” e “deformation”
- mappa, contenuta nella view principale, nella quale saranno visibili i punti in base alla zona selezionata. È possibile scegliere due layout diversi (satellitare e mappa semplice)
- posizione corrente, in modo che la mappa visualizzi i punti di deformazione della zona in cui si trova l’utente
- menu delle zone disponibili, costituito da una tabella contenente tutte le zone messe a disposizione dal connettore
- menu dei preferiti, costituito anch’esso da una tabella contenente tutte le zone visitate maggiormente dall’utente. Per poter inserire una zona tra i preferiti, si può utilizzare il bottone apposito, raffigurato da una stella

METODOLOGIA DI LAVORO

Dal momento che SARplay nasce come progetto di gruppo, sin da subito c’è stata la necessità di trovare un modo per rimanere in contatto tra tutti e di poter condividere tra tutti il progetto di lavoro. Oltre ai numerosi incontri tenuti durante l’anno, sia con i membri della Visuality, sia solo tra noi ragazzi, rimaneva comunque il problema della condivisione del materiale, con mezzi che non fossero le e-mail. Gli sviluppatori di software utilizzano strumenti come LinkedINper poter rimanere in contatto facilmente e per poter condividere il materiale tramite le apposite repository. Nel nostro caso, la Visuality software usa da tempoBasecamp (https://3.basecamp.org/), pertanto

ha creato uno spazio apposito per il progetto (un Basecamp, appunto).



Le principali funzionalità di Basecamp sono:

- Ping: sono dei messaggi tra due o più utenti, molto comodi per chiarire, specificare o chiedere concetti in maniera immediata; il funzionamento è lo stesso di ogni programma di chat;
- Hey: è una schermata contenente tutte le notifiche;
- Messages: è una chat tra tutti i membri del gruppo;
- To-dos: è una schermata che contiene tutte le cose da fare: ogni utente può creare dei to-do, e delegare ad un altro utente il compito; una volta terminato il compito, è possibile archiviarlo;

- Schedule: contiene tutte le scadenze dei to-dos. È possibile visualizzare tutti i to-dos di tutti i mesi futuri, oltre a quelli già scaduti;
- Check-ins: sono delle domande programmate che vengono inviate all'utente costantemente. Una domanda può essere "Sei bloccato su qualcosa?", oppure "Su cosa stai lavorando?"; in base alla risposta inviata, si possono ricevere chiarimenti, oppure si può organizzare ancora meglio il lavoro rimanente;

- Docs&files: è uno spazio nel quale vengono caricati file utili per il progetto corrente;

Tuttavia, il progetto non viene caricato su Basecamp, in quanto è scomodo per poter tener traccia di tutti gli aggiornamenti: per questo la Visuality usa BitBucket (http://bitbucket.org), un servizio di hosting di materiale pensato apposta per gli sviluppatori di applicazioni.
La repository di BitBucket contiene sempre il progetto più aggiornato degli sviluppatori, infatti ogni volta che un programmatore vuole continuare il proprio lavoro sul progetto, dovrà compiere un operazione di **PULL**, ovvero il download del progetto più aggiornato;
Allo stesso modo se vuole salvare delle modifiche fatte al progetto deve compiere un'operazione di **PUSH**, così che gli altri possano poi aggiornare il loro progetto.
Queste operazioni possono essere fatte a linea di comando, con l'istruzione –git ecc.., ma noi, per semplicità d'uso e di comprensione, abbiamo utilizzato SourceTree.
SourceTree è un programma visuale che permette di compiere queste operazioni, in un apposito menu ti permette inoltre di capire

quali sono le modifiche da te apportate e ti permette di decidere se apportarle tutte o eliminarne alcune.

IL MIO CONTRIBUTO

Il mio compito consisteva nel lavorare sulle mappe, sui punti in esse rappresentati e sulle specifiche di tali punti.
Qui di seguito è presentato un indice del lavoro da me compiuto sull'app e dei mezzi utilizzato per farlo:

1) Presentazione degli strumenti di lavoro:
    a. Le classi utilizzate

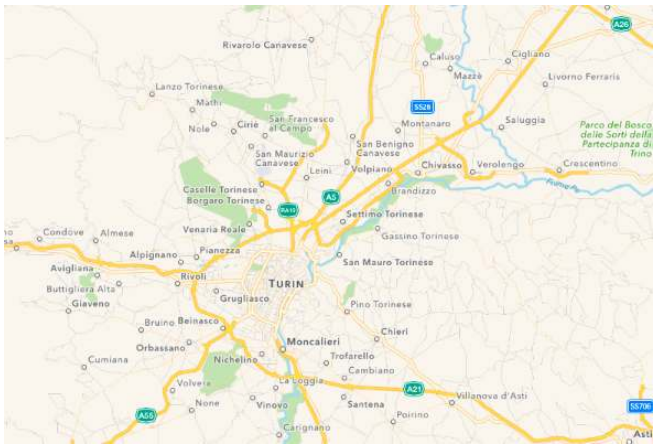
2) Presentazione della features (punto):
    a. Modalità manuale
    b. Modalitàautomatica GPS

3) Specifiche della features:
    a. Significato
    b. Popover: grafica e presentazione
    c. Significato dei dati del popover: il grafico

AMBIENTE DI SVILUPPO E LIBRERIE

Per poter realizzare l'applicazione, io e i miei compagni abbiamo dovuto imparare ad utilizzare XCode per la programmazione software del mondo Apple e il linguaggio SWIFT. Io in specifico, ho utilizzato alcune librerie proprietarie di questo linguaggio:

- **MapKit**: è una semplice API di Apple che permette di integrarele mappe all'interno delle applicazione iOs, senza quindi passare attraverso le API di Google, che vengono utilizzate in tutti gli altri ambiti della programmazione, sia Web



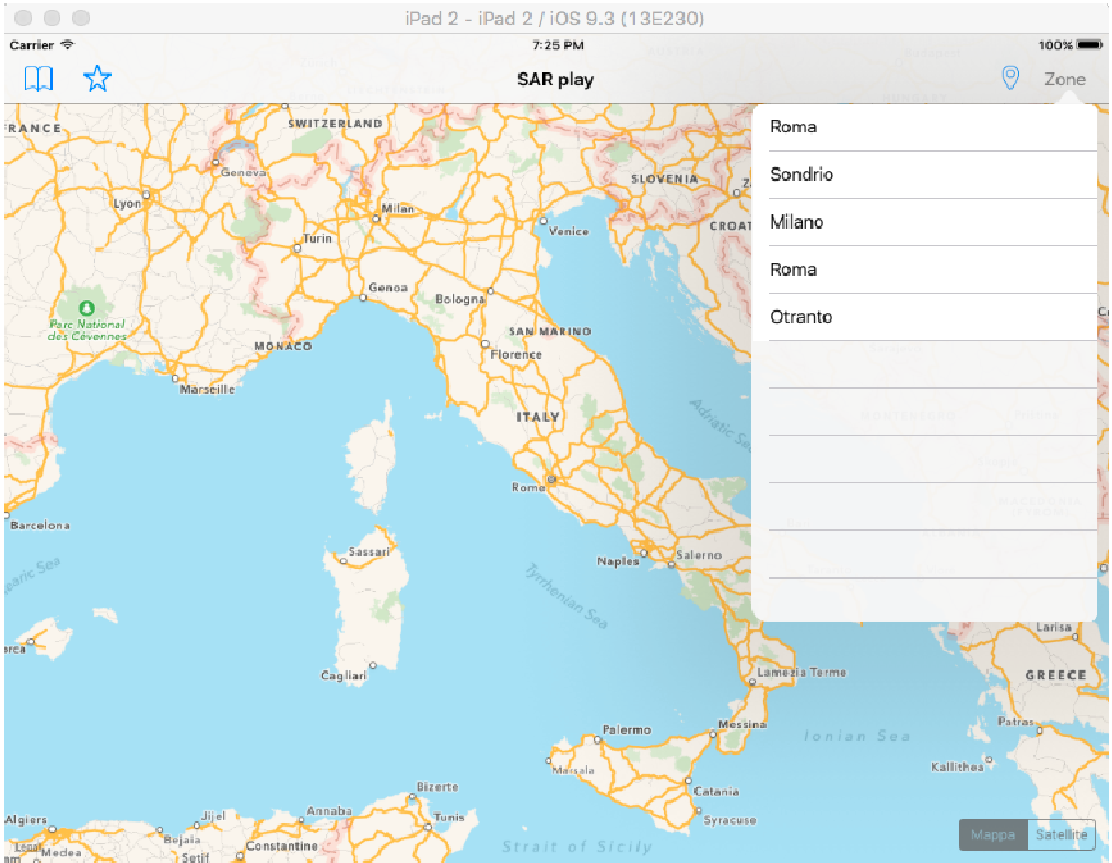
che Android.

MapKit permette, come anche Google, di visualizzare la mappa in due modalità: satellitare, dove si può vedere la mappa fisica, oppure in modalità classica. Le due modalità possono essere scelte grazie a un selettore posto di solito in

basso a destra dell'applicazione.

- **MKAnnotation** e le sue sottoclassi, questa libreria possiede:
  - MKCircle: che è la classe che permette la costruzione del punto fisico;
  - MKCircleRender: che è la classe che permette di visualizzare il punto sulla mappa e dargli un colore;
- **CLLocationManager:** è la classe che permette di implementare la funzione GPS;

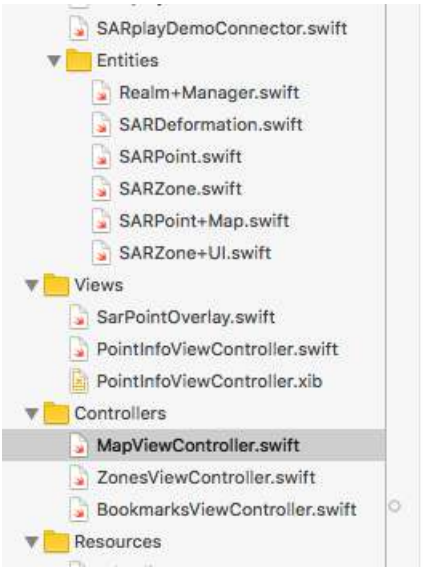
MODALITA' MANUALE



La modalità manuale è quella che permette all'utente

di selezionare una zona tra quelle predefinite (ovvero presenti nel database), nel menu in alto a destra “Zone”;

Di seguito il meccanismo che permette, una volta selezionata la zona, di visualizzare i punti sulla mappa:
La classe MapViewController.swift (quella su cui ho maggiormente lavorato) contiene un attributo **currentZone** di tipo **SARZone**(entità da noi creata)




```
var userPosition: CLLocation?

var pointVelocity: Double?

var sarPointForPopover: SARPoint!

var locationManager: CLLocationManager?
var currentZone: SARZone? {

    willSet {

        print("CLEAR")
        clearData()
    }

    didSet {

        print("RELOAD")
        reloadData()
    }
}

@IBAction func touchDown(sender: AnyObject) {

    if currentZone?.isBookmarked == false{
```

Quando questo attributo si setta, ovvero si cambia il suo valore, dapprima viene azionato il metodo **clearData** e poi il metodo **reloadData**. Questo è possibile grazie a **willSet** che permette di eseguire operazioni mentre l'attributo viene settato, e **didSet**che esegue operazioni una volta che l'attributo è stato settato.

Quando si preme una zona nel menu, scatta questo meccanismo, infatti

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) {
    mapView?.currentZone = allZones[indexPath.row]
    dismissViewControllerAnimated(true, completion: nil)// chiudi la finestrella
}
```

come si vede qui sopra viene settato currentZone della classe MapView, e viene modificato nella zona scelta.

Come già detto in precedenza, una volta che il meccanismo viene attivato, vengono chiamati i due metodi **clearData** e **reloadData** che permettono di togliere e mettere fisicamente i punti.

```
// MARK: - Methods

func clearData() {

    self.mapView.removeOverlays(self.mapView.overlays)
}

func reloadData() {

    guard let zone = self.currentZone else {return}
    let points = zone.points


    print("ZONA")
    print(zone.title)

    print("ZONA SCARICATA: \(zone)")

    // Spostamento della mappa

    let lowerLeftCorner = MKMapPointForCoordinate(zone.lowerLeftCorner)
    let upperRightCorner = MKMapPointForCoordinate(zone.upperRightCorner)

    let zoomMapRect = MKMapRectMake(lowerLeftCorner.x, upperRightCorner.y, upperRightCorner.x - lowerLeftCorner.x,
        lowerLeftCorner.y - upperRightCorner.y);

    self.mapView.setVisibleMapRect(zoomMapRect, animated: true)


    for point in points {

        print("Punto(\(point.id) [\(point.latitude), \(point.longitude)])")

        pointVelocity = point.velocity


        dispatch_async(dispatch_get_main_queue(), {

            let overlay = SarPointOverlay.sarPointOverlayWithPoint(point)


            self.mapView.addOverlay(overlay)
        })
    }
```

***clearData*:**come si vede nell'immagine precedente questo metodo ha il semplice compito di togliere i punti dalla mappa, grazie al metodo
removeOverlay.

reloadData:questo metodo è un po' più complesso, esso permette di visualizzare i punti sulla mappa in base alla zona. Dapprima si controlla se currentZone non è nulla grazie all'uso di **guard**, e se questa condizione è vera, grazie all'istruzione

zone.pointsviene creata una query sul DB che ritorna un vettore carico di tutti i punti della zona. A questo punto mentre i punti vengono caricati nel vettore interno al metodo, spostiamo la locazione della mappa in modo da poter vedere i puntini quando saranno caricati. Questo si fa grazie a **lowerLeftCorner** e **upperRightCorner** che sono due attributi della zona; essi rappresentano:

- **lowerLetfCorner**: oggetto di tipo MKMapPointForCoordinate che contiene le coordinate del punto in basso a sinistra, come si vede nella foto;
- **upperRightCorner**: oggetto del medesimo tipo che contiene invece le coordinate del punto in alto a destra

Questi oggetti saranno poi utilizzati dalla classe MKMapRect per creare il rettangolo da visualizzare. A questo punto con il metodo **setVisibleMapRect**della mapView, a cui si passa l'oggetto precedentemente creato, la mappa si sposterà.

L'ultima cosa che manca è quella di caricare i punti sulla mappa:

```
for point in points {
    print("Punto(\(point.id) [\(point.latitude), \(point.longitude)])")
    pointVelocity = point.velocity

    dispatch_async(dispatch_get_main_queue(), {
        let overlay = SarPointOverlay.sarPointOverlayWithPoint(point)

        self.mapView.addOverlay(overlay)
    })
}
```

```
func mapView(mapView: MKMapView, rendererForOverlay overlay: MKOverlay) -> MKOverlayRenderer {
    if let overlay = overlay as? SarPointOverlay, velocity = overlay.point?.velocity {
        let circleRenderer = MKCircleRenderer(circle: overlay)
        let color = colorForVelocity(velocity)
        circleRenderer.strokeColor = color
        circleRenderer.alpha = 0.5

        return circleRenderer
    }
    else {
        return MKOverlayRenderer(overlay: overlay)
    }
}
```

Si scorre il vettore **points**che contiene i punti della zona corrente; per ogni punto, in unthread diverso, viene creato un oggetto di tipo **SarPointOverlay** che è una sottoclasse di **MKCircle** da me realizzata per poter arricchire di proprietà il punto. A questo punto si aggiunge alla mappa l'overlay. Per fare ciò il metodo **addOverlay**ha bisogno di "contattare" il CircleRender che disegna il punto sulla mappa.
Ogni volta il metodo qui riportato a sinistra prenderà un

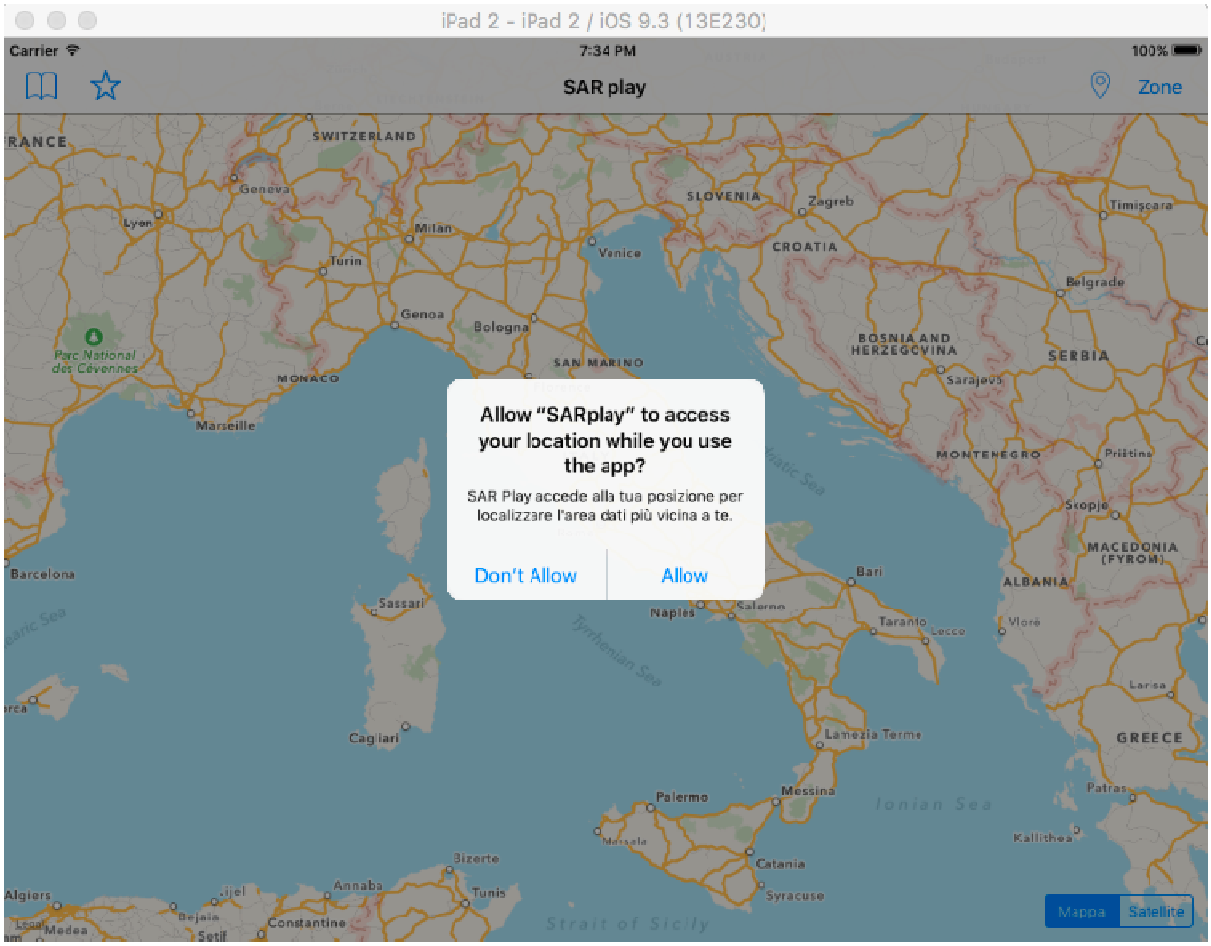
MKCircle e ritornerà un Overlay colorato in base alle proprietà del punto stesso.

Il colore viene dato chiamando il metodo **colorForVelocity** che riceve un valore decimale e ritorna il colore corrispondente secondo le specifiche del CNR.
Alla fine del ciclo tutti i punti saranno visualizzati sulla mappa.

```
func colorForVelocity(value: Double) -> UIColor {

    // Colori secondo le specifiche CNR

    let viola = (1.07143...Double.infinity)
    let blu = (0.64286...1.07143)
    let azzurro = (0.21429...0.64286)
    let verde = ((-0.21429)...0.21429)
    let giallo = ((-0.64286)...(-0.21429))
    let arancio = ((-1.07143)...(-0.64286))
    let rosso = ((-Double.infinity)...(-1.07143))


    if viola.contains(value) {

        return UIColor.purpleColor()
    }

    if blu.contains(value) {

        return UIColor.blueColor()
    }

    if azzurro.contains(value) {

        return UIColor.cyanColor()
    }
```

MODALITA’ AUTOMATICA GPS

In generale il funzionamento che permette di visualizzare i punti è lo stesso visto in precedenza, l’unica cosa che cambia in questa modalità è come settare **currentZone**, infatti ora la zona sarà modificata non più con il click della zona nel menu a destra, ma premendo il bottone della modalità automatica a fianco del precedente.
La prima volta che l’app viene avviata, uscirà la seguente schermata:


iPad 2 - iPad 2 / iOS 9.3 (13E230)
Carrier
7:34 PM
100%
SAR play
Zone
Allow "SARplay" to access your location while you use the app?
SAR Play accede alla tua posizione per localizzare l'area dati più vicina a te.
Don't Allow
Allow
Mappa
Satellite

Premendo su **Allow**, sarà concesso all'applicazione di usare il GPS del nostro dispositivo.

Premendo quindi sul bottone  , verrà chiamato il metodo **automaticType** della classe MapViewController.swift.

```
func automaticType()
{
    // TODO: capire come gestire il bottone quando e' in modalita GPS

    print("AUTOMATIC TYPE")

    TypeLocation.image = UIImage(named:"manuale")

    if let userPosition = self.userPosition {

        let span = MKCoordinateSpanMake(0.5, 0.5)
        let region = MKCoordinateRegion(center: userPosition.coordinate, span: span)
        mapView.setRegion(region, animated:true)


        currentZone = SARZone.zoneContainingLocation(userPosition)
        print("settint zone \(currentZone)")

        if currentZone == nil{

            self.presentViewController(alert, animated: true, completion: nil)
        }

    }

}
```

Il metodo automatico, dapprima setta l'immagine del bottone premuto con la modalità manuale, così da far capire che ora si è in automatica, poi dopo di che controlla se il GPS funziona correttamente e la variabile **userPosition**contiene delle coordinate. Se ciò è vero sposta la mappa nella zona dove siamo usando le coordinate della **userPosition,** grazie al **setRegion** della MapView.

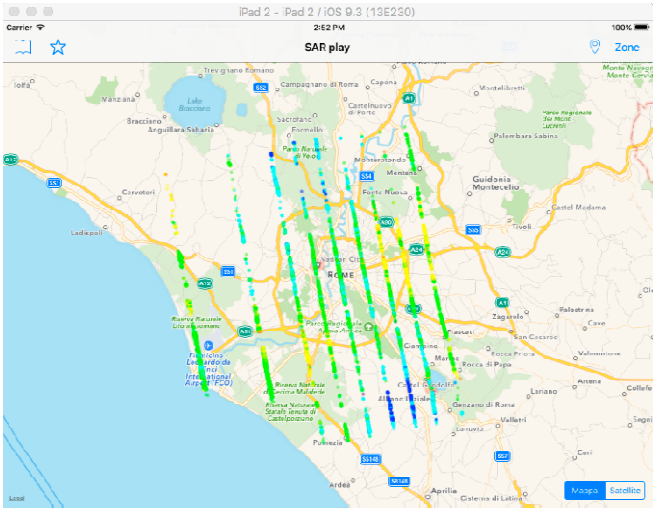
A questo punto manca solo di settare la zona; Dato che abbiamo però solo le coordinate di dove siamo, ci necessita capire in che zona siamo per poter caricare i punti: questo avviene attraverso lo

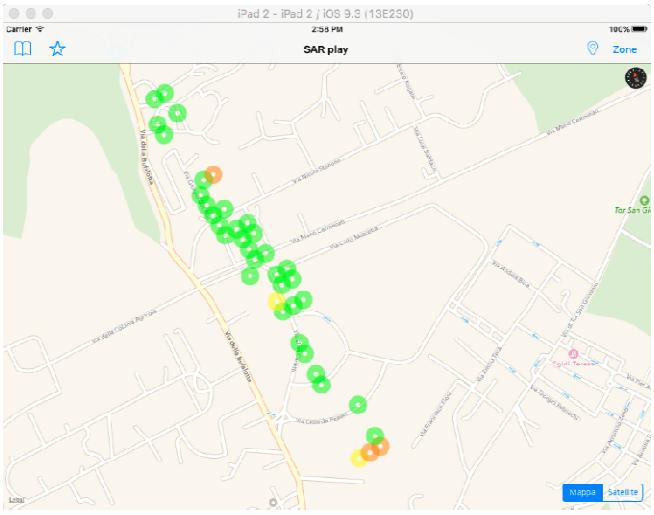
```
class func zoneContainingLocation(location: CLLocation) -> SARZone? {

    let point = MKMapPointMake(location.coordinate.latitude, location.coordinate.longitude)

    let zones = SARZone.allZonesInRealm()

    for zone in zones {

        if MKMapRectContainsPoint(zone.mapRect, point) {

            return zone
        }
    }

    return nil
```

zoneContainingLocationqui sopra riportato, che avendo in input un CLLocation, che corrispondono a due coordinate, restituisce la SARZone corrispondente. Il metodo non è complesso, infatti, controlla se il “rettangolo” delle zone contiene la nostra localizzazione. Se la localizzazione è contenuta, allora currentZone sarà settata e secondo il metodo descritto precedentemente i punti verranno caricati, altrimenti verrà visualizzato un popover che farà capire che la zona non è presente nell’archivio.

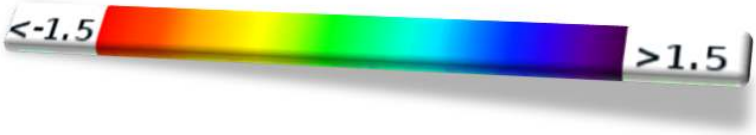
SIGNIFICATO DELLA FEATURES

Le features, così sono chiamati i punti della mappa, permettono all’utente di capire di quanto il terreno si è alzato o abbassato rispetto alla misurazione avvenuta una settimana prima. Qui sotto sono posti alcuni screen che mostrano l’app funzionante.





Lo spostamento del terreno varia di solito da -1.5mm e 1.5 mm, in quanto in una settimana è difficile che il terreno si sposti di più. Qui sotto è presentata la scala dei colori dei punti.



ROSSO: -1,5mm / -1,071mm
ARANCIONE:-1,071mm / -0,64mm
GIALLO: -0,64mm / -0,21mm
VERDE: -0,21mm / 0,21mm
AZZURRO: 0,21mm / 0,64mm
BLU: 0,64mm/ 1,071mm
VIOLA: 1,071mm / 1,5mm

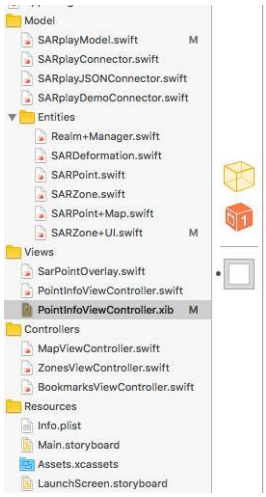
POPOVER: GRAFICA e PRESENTAZIONE

Una volta che le features si sono presentate sulla mappa, premendo sopra di essere, sarà visualizzato un popover, che conterrà le principali caratteristiche del punto:


iPad 2 - iPad 2 / iOS 9.3 (13E230)
Carrier
3:08 PM
100%
SAR play
Zone
ID: csk_roma.fid-68b99caf_1522fa8eedd_1d9
Latitudine: 41.98663
Velocity: -0.0361673
Longitudine: 12.553143
Elevation: 4.14118
2010/9/16
2010/10/14
2010/10/14
2010/10/31
2010/10/31
Map
Satellite

Come si vede nello screen qui sopra, il popover contiene l'**id** del punto, valore univoco nella tabella del DB, la **latitudine** e la **longitudine**, l'**elevation**che rappresenta l'altitudine, e la **velocity**che è il parametro espresso in millimetri che ci permette di capire di quanto si è alzato il terreno, e quindi di dare un colore alla features.

Vediamo ora com'è possibile costruire un popover e dargli delle caratteristiche:
Innanzitutto è necessario che l'overlay si possa premere; Io e i miei compagni abbiamo deciso di utilizzare le **gestureRecognize**che abbinate al **MKAnnotation**permettono di creare eventi sullo schermo. Ora vediamo come il popover viene creato:



Dapprima è necessario creare un file**.xib**, esso rappresenta la view che il popover presenterà al suo interno, e sarà necessario collegarlo a un'altra classe**.swift** per potergli modificare le proprietà man mano.

Quando un overlay viene premuto, viene istanziato un oggetto del tipo PointInfoViewController a cui passeremo un punto, cosi da poter usare le sue proprietà per la costruzione dell'interfaccia.

```
print(sarPoint)

guard let sarPoint = self.sarPoint else {return}

idLabel.text = sarPoint.id
latitudeLabel.text = "\(sarPoint.latitude)"
longitudeLabel.text = "\(sarPoint.longitude)"
velocityLabel.text = "\(sarPoint.velocity)"
elevationLabel.text = "\(sarPoint.elevation)"

self.preferredContentSize = CGSizeMake(450, 400)
```

Dapprima si controlla che il punto sia arrivato, con l'istruzione **guard**, se è vero, si settano le varie label dello xib sopra presentato con i valori del punto corrente.
Per ultima cosa fissiamo la grandezza dello xib:
450x400 è la misura ideale, sia considerando i dispositivi su cui andrà, sia per il contenuto e lo spazio vuoto presente.

IL GRAFICO

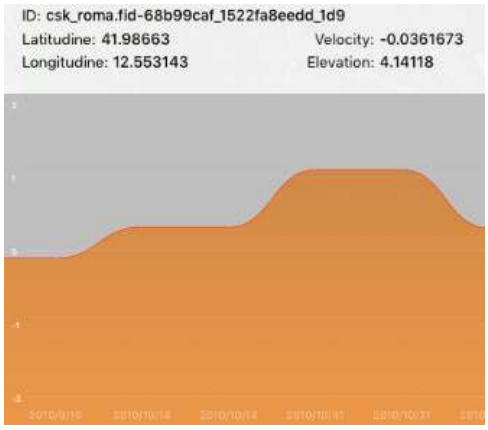
Il grafico contenuto nel popover rappresenta i dati contenuti nella lista **pointDeformation** che è un attributo del punto **SARPoint**.
Essa è formata da:

- value: che contiene alcuni valori di innalzamento precedenti a quello corrente;
- date: che contiene la data corrispondente del valore sopra;

Ora vediamo come si costruisce il grafico reale.
Innanzitutto è necessario importare una libreria, la **ScrollableGraphView** che ci permette di costruire grafici complessi e migliori esteticamente.

```
// Creazione Grafico
graphView = ScrollableGraphView(frame: graphV.frame)

// Posizionamento del Grafico
graphView.frame = CGRectMake(0, 0, graphV.frame.size.width, graphV.frame.size.height)

graphView.backgroundFillColor = UIColor(red:0.75, green:0.75, blue:0.75, alpha:1.0)
  graphView.layer.cornerRadius = 7

graphView.rangeMin = -1.5
graphView.rangeMax = 1.5

graphView.lineWidth = 0.5
graphView.lineColor = UIColor.redColor()
graphView.lineStyle = ScrollableGraphViewLineStyle.Smooth
graphView.shouldFill = true
graphView.fillType = ScrollableGraphViewFillType.Gradient
graphView.fillColor = UIColor.blackColor().colorWithAlphaComponent(0.8)
graphView.fillGradientType = ScrollableGraphViewGradientType.Linear
graphView.fillGradientStartColor = UIColor.brownColor().colorWithAlphaComponent(0.8)
graphView.fillGradientEndColor = UIColor.orangeColor().colorWithAlphaComponent(0.6)
graphView.dataPointSpacing = 80
graphView.dataPointSize = 0.5
graphView.dataPointFillColor = UIColor.whiteColor()

graphView.referenceLineLabelFont = UIFont.boldSystemFontOfSize(8)
graphView.referenceLineColor = UIColor.whiteColor().colorWithAlphaComponent(0.2)
graphView.referenceLineLabelColor = UIColor.whiteColor()
graphView.dataPointLabelColor = UIColor.whiteColor().colorWithAlphaComponent(0.5)

let graphData = array(forPoint: sarPoint)

print(graphData)

graphView.setData(graphData.data, withLabels: graphData.labels)

self.graphV.addSubview(graphView)
```

Dapprima s'istanza un oggetto del tipo ScrollableGraphView e gli si passa la finestra in cui dovrà adattarsi. Dopodiché è necessario modificare i colori delle parti del grafico a nostro piacimento.Come ultima cosa è necessario passare i dati al grafico, si suddivide quindi la lista in due array e si passano al grafico con il metodo **setData**.



Durante la realizzazione del grafico, mi sono accorto che i dati nella lista non erano in ordine di tempo, mi è stato quindi necessario riordinarle. Questo mi è stato facilmente possibile grazie al metodo **sort** della lista, che permette di ordinarla con due semplici istruzioni.