

Hit-Testing in iOS

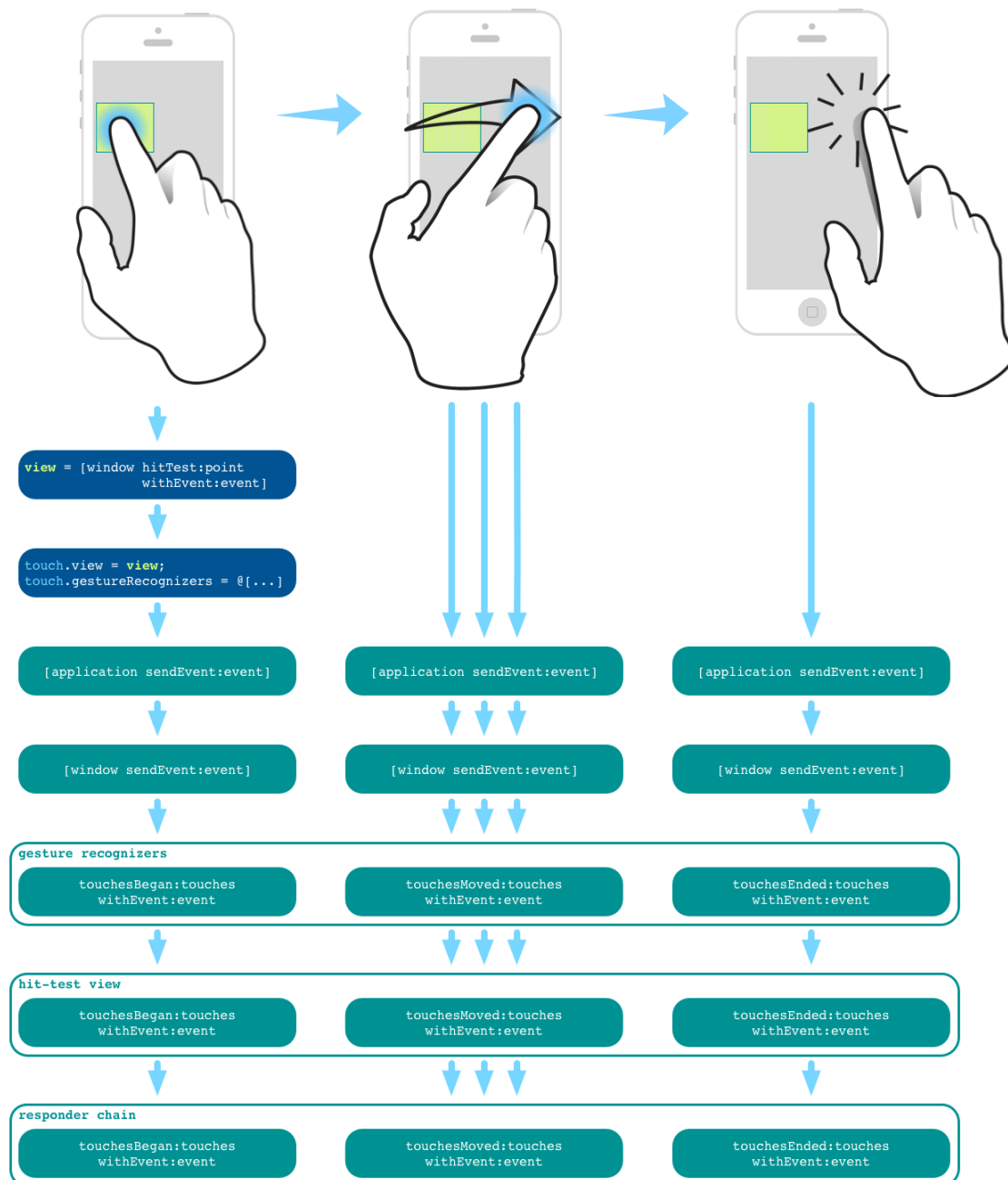
🕒 22 Apr 2014



f t g+

Hit-testing is the process of determining whether a point, such as touch-point, intersects a given graphical object, such as `UIView`, drawn on the screen. iOS uses hit-testing to determine which `UIView` is the frontmost view under the user's finger that should receive the touch event. It implements it by searching the view hierarchy using reverse pre-order depth-first traversal algorithm.

Before explaining how the hit-testing works, it is important to understand when the hit-testing is executed. The following diagram illustrates the high-level flow of a single touch, from the moment a finger touches the screen and until it is lifted from it:



As illustrated in the diagram above, the hit-testing is executed every time a finger touches the screen. And, before any view or gesture recognizer receives the `UIEvent` object representing the event to which the touch belongs.

Note: for unknown reasons, the hit-testing is executed multiple times in a row. Yet, the determined hit-test view remains the same.

After hit-testing completes and the frontmost view under the touch-point is determined, the hit-test view is associated with the `UITouch` object for all phases of the touch event sequence (i.e.: began, moved, ended, or canceled). In addition to the hit-test view, any gesture recognizers attached to that view or any of its ancestor views are associated with the `UITouch` object. Then, the hit-test view begins to receive the sequence of touch events.

An important thing to take in mind is that even if the finger is moved outside the hit-test view bounds over another view, the hit-test view still continue to receive all the touches until the end of the touch event sequence:

"The touch object is associated with its hit-test view for its lifetime, even if the touch later moves outside the view."

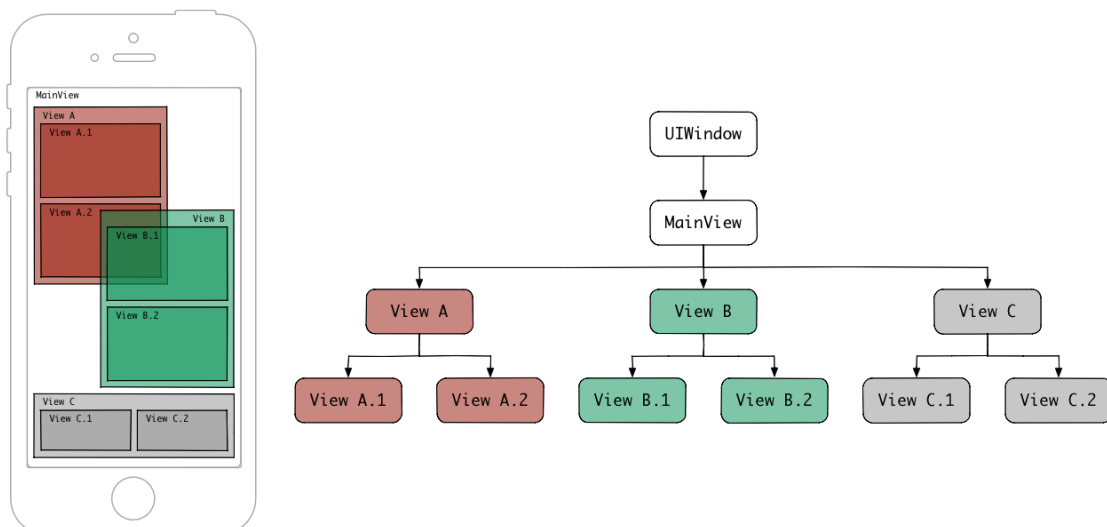
Event Handling Guide for iOS, iOS Developer Library

As mentioned earlier the hit-testing uses depth-first traversal in reverse pre-order (first visiting the root node and then traversing its subtrees from higher to lower indexes). This kind of traversal allows reducing the number of traversal iterations and stopping the search process once the first deepest descendant view that contains the touch-point is found. This is possible since a subview is always rendered in front of its superview and sibling view is always rendered in front of its sibling views with a lower index into the subviews array. Such that, when multiple overlapping views contain specific point, the deepest view in the rightmost subtree will be the frontmost view.

"Visually, the content of a subview obscures all or part of the content of its parent view. Each superview stores its subviews in an ordered array and the order in that array also affects the visibility of each subview. If two sibling subviews overlap each other, the one that was added last (or was moved to the end of the subview array) appears on top of the other."

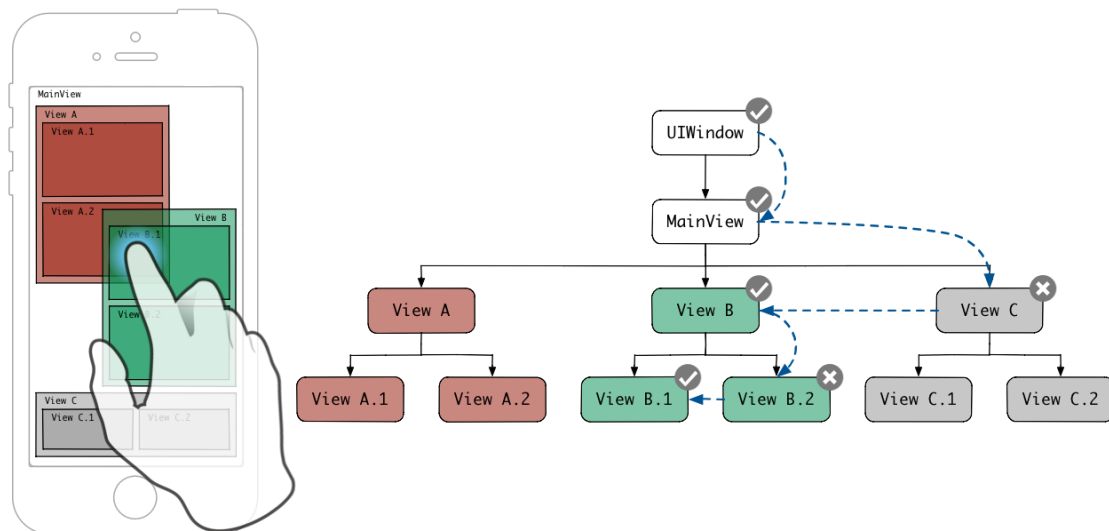
View Programming Guide for iOS, iOS Developer Library

The following diagram shows an example of a view hierarchy tree and its matching UI that is drawn on the screen. The arrangement of tree branches from left to right reflects the order of the subviews array.



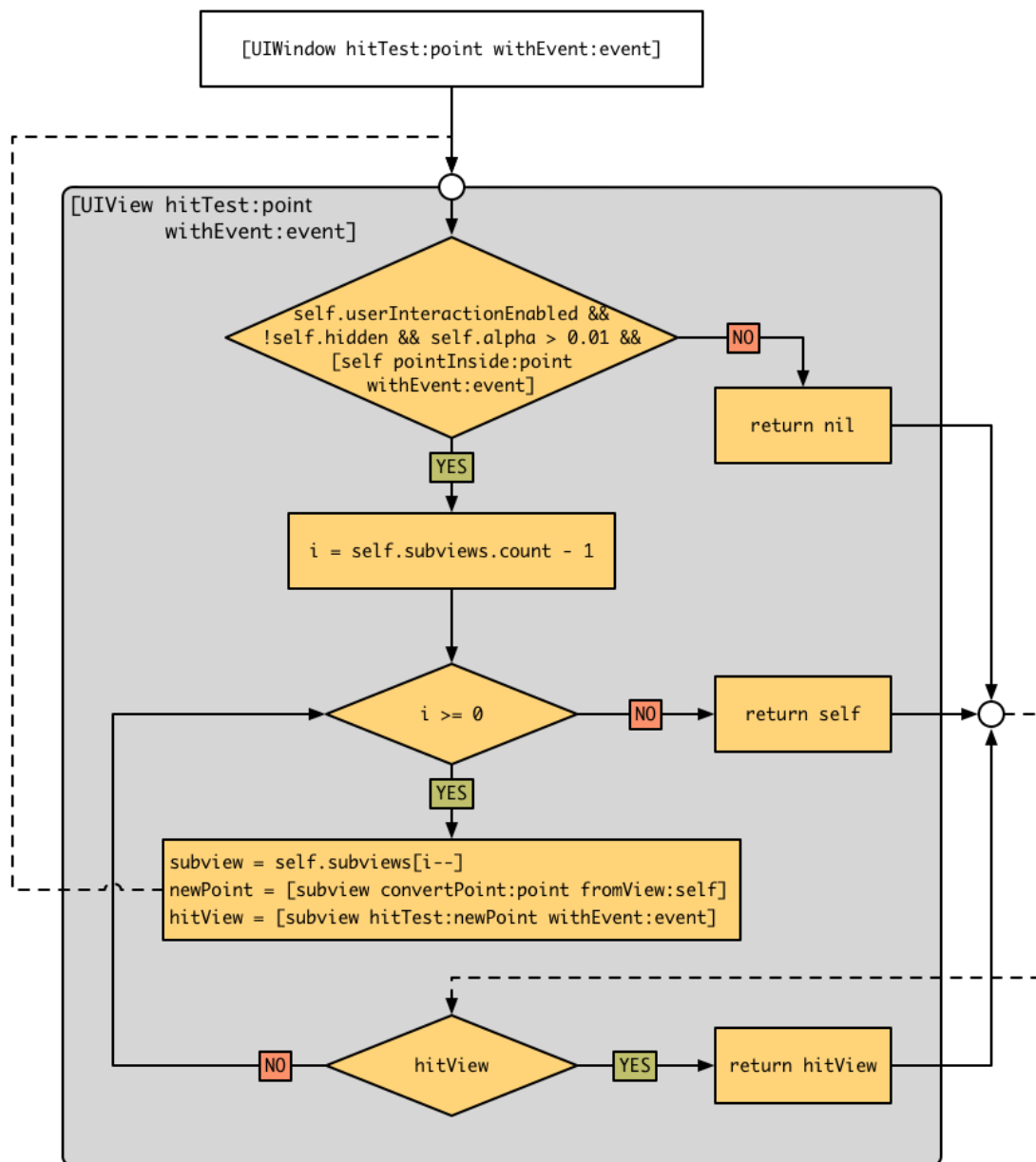
As it can be seen, “View A” and “View B” as well as their children, “View A.2” and “View B.1”, are overlapping. But since “View B” has a subview index higher than that of “View A”, “View B” and its subviews are rendered above “View A” and its subviews. Therefore, “View B.1” should be returned by hit-testing when user’s finger touches “View B.1” in the area where it overlaps with “View A.2”.

By applying depth-first traversal in reverse pre-order allows stopping the traversal once the first deepest descendant view that contains the touch-point is found:



The traversal algorithm begins by sending the `hitTest:withEvent:` message to the `UIWindow`, which is the root view of the view hierarchy. The value returned from this method is the frontmost view containing the touch-point.

Following flow-chart illustrates the hit-test logic.



And the following code shows possible implementation of the native

`hitTest:withEvent:` method:

```

- (UIView *)hitTest:(CGPoint)point withEvent:(UIEvent *)event {
    if (!self.userInteractionEnabled || self.isHidden ||
        !self.pointInside:point withEvent:event) {
        return nil;
    }
    if ([self pointInside:point withEvent:event]) {
        for (UIView *subview in [self.subviews reverse]) {
            CGPoint convertedPoint = [subview convertPoint:point fromView:self];
            UIView *hitTestView = [subview hitTest:convertedPoint withEvent:event];
            if (hitTestView) {
                return hitTestView;
            }
        }
        return self;
    }
    return nil;
}

```

The `hitTest:withEvent:` method first checks if the view is allowed to receive the touch. A view is allowed to receive the touch if:

- The view is not hidden:

```
self.hidden == NO
```

- The view has user interaction enabled:

```
self.userInteractionEnabled == YES
```

- The view has alpha level greater than 0.01:

```
self.alpha > 0.01
```

- The view contains the point:

```
pointInside:withEvent: == YES
```

Then, if the view is allowed to receive the touch, this method traverses the receiver's subtree by sending the `hitTest:withEvent:` message to each of its subview from last to first until one of them returns non `nil` value. The first non `nil` value returned by one of the subviews is the frontmost view under the touch-point and is returned by the receiver. If all receiver subviews returned `nil` or the receiver has no subviews the receiver returns itself.

Otherwise, if the view is not allowed to receive the touch, this method returns `nil` without traversing the receiver's subtree at all. Therefore, the hit-test process may not visit all the views in the view hierarchy.

Common use cases for overriding `hitTest:withEvent:`

The `hitTest:withEvent:` method could be overridden when the touch events intended to be handled by one view should be redirected to another view for all phases of that touch event sequence.

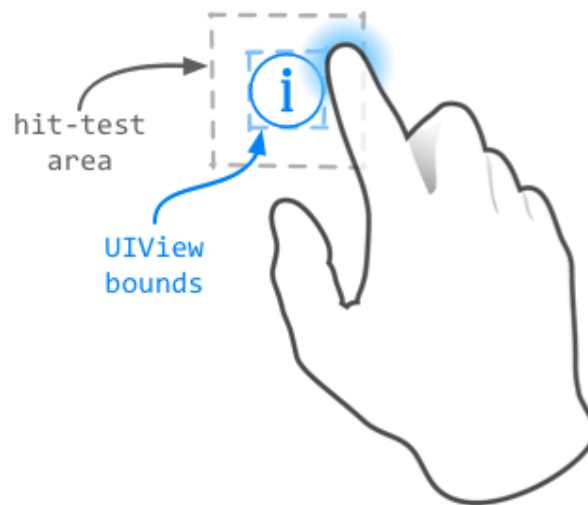
Because the hit-test is executed before, and only before, the first touch event of the touch event sequence is sent to its receiver (the touch with the `UITouchPhaseBegan` phase), overriding `hitTest:withEvent:` to redirect events will redirect all touch events of that sequence.

Increasing view touch area

One use case which could justify overriding the `hitTest:withEvent:` method is when the touch area of a view should be larger than its bounds. For example, the following illustration shows a `UIView` having size of 20x20. This size may be too small to handle nearby touches. Therefore, its touch area may be

increased by 10 points in each direction by overriding the

`hitTest:withEvent:` method:



```
- (UIView *)hitTest:(CGPoint)point withEvent:(UIEvent *)e
{
    if (!self.userInteractionEnabled || self.isHidden ||
        return nil;
    }
    CGRect touchRect = CGRectInset(self.bounds, -10, -10)
    if (CGRectContainsPoint(touchRect, point)) {
        for (UIView *subview in [self.subviews reverseOrder]) {
            CGPoint convertedPoint = [subview convertPoint:point toView:self];
            UIView *hitTestView = [subview hitTest:convertedPoint withEvent:e];
            if (hitTestView) {
                return hitTestView;
            }
        }
        return self;
    }
    return nil;
}
```

Note: in order for this view to be hit-tested correctly, the parent view's bounds should contain the desired touch area of its subview, or its `hitTest:withEvent:` method should be also overridden to include the desired touch area.

Passing touch events through to views below

Sometimes it is necessary for a view to ignore touch events and pass them through to the views below. For example, assume a transparent overlay view placed above all other application views. The overlay has some subviews in the form of controls and buttons which should respond to touches normally. But

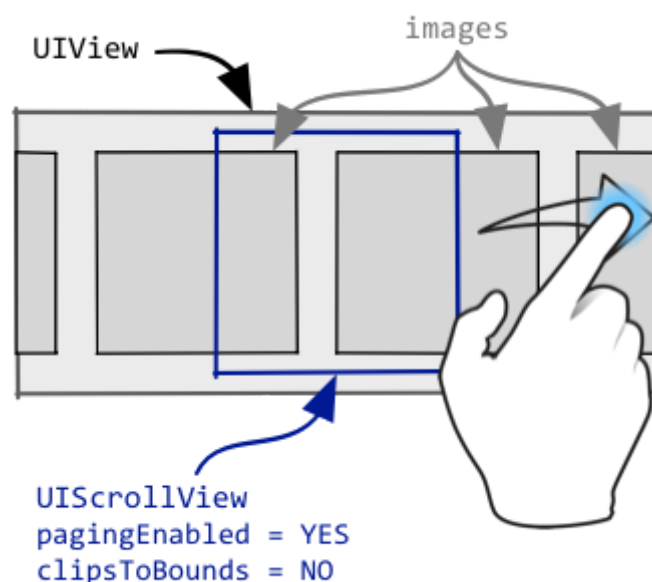
touching the overlay somewhere else should pass the touch events to the views below the overlay. To accomplish this behavior the overlay

`hitTest:withEvent:` method could be overridden to return one of its subviews containing the touch-point and `nil` in all other cases, including the case when the overlay contains the touch-point:

```
- (UIView *)hitTest:(CGPoint)point withEvent:(UIEvent *)e
    UIView *hitTestView = [super hitTest:point withEvent:
        if (hitTestView == self) {
            hitTestView = nil;
        }
        return hitTestView;
    }
```

Passing touch events to subview

A different use case may require parent view redirect all touch events to its only child view. This behavior may be desired when the child view partially occupies its parent view, but should respond to all touches occurring in its parent. For example, assume a carousel of images that consist of a parent view and a `UIScrollView` with `pagingEnabled` set to `YES` and `clipsToBounds` set to `NO` to create carousel effect:



To make the `UIScrollView` to respond to touches occurring not only inside its own bounds, but also inside the bounds of its parent view, the parent's `hitTest:withEvent:` method could be overridden in the following way:

```
- (UIView *)hitTest:(CGPoint)point withEvent:(UIEvent *)e
    UIView *hitTestView = [super hitTest:point withEvent:
```



```

if (hitTestView) {
    hitTestView = self.scrollView;
}
return hitTestView;
}

```

14 Comments

smnh

1 Login ▾

♥ Recommend 4

🔗 Share

Sort by Best ▾



Join the discussion...

**Vijaya Prakash Kandel** • 3 years ago

Thanks for the wonderful post, really enjoyed it.

6 ^ | v • Reply • Share ›

**idrisr** • 2 years ago

Great visuals. They helped immensely to understand this concept.

1 ^ | v • Reply • Share ›

**Danny** • 2 years ago

Really love how you explain things in this article, but I'm a bit confused with the last scrollview example? Wouldn't this make more sense?

```

- (UIView *)hitTest:(CGPoint)point withEvent:(UIEvent *)event
{
    UIView *hitTestView = [super hitTest:point withEvent:event];

```

```

    if (hitTestView) {
        hitTestView = self.carouselScrollView;
    }

```

```

    return hitTestView;
}

```

1 ^ | v • Reply • Share ›

**smnh** Mod ➔ Danny • 2 years agoYou are right Danny, I will fix that
Thanks!

^ | v • Reply • Share ›

**yebingwei** • 3 years ago

Thanks a lot

1 ^ | v • Reply • Share ›



warpling • 3 years ago

This is such a fantastic explanation and the visuals are perfect—helped me a lot. Thank you so much.

1 ^ | v • Reply • Share ›



윤병인 • 4 months ago

Thank You

^ | v • Reply • Share ›



Seyoung • 8 months ago

Thank you. It's an amazing post. Could you share how you created those beautiful graphics?? Photoshop?

^ | v • Reply • Share ›



joshuali • a year ago

mark

^ | v • Reply • Share ›



maple • 2 years ago

Thanks for your post, i finally understand hittest in iOS.

^ | v • Reply • Share ›



Kris • 2 years ago

Thanks for the post, this actually inspired me to make custom subclasses for increasing hit-test area for my projects that I eventually open-sourced: <https://github.com/kgaidis/....> I also mentioned your blog in the README!

^ | v • Reply • Share ›



iknowsomething.com • 2 years ago

I wish Apple were explaining like that - marvellous!

My question is, however, where did you get that iPhone and hands stencils? I'm looking for something like that.

^ | v • Reply • Share ›



smnh Mod ➔ **iknowsomething.com** • 2 years ago

Thanks,

Hand gestures: <http://www.mobiletuxedo.com...>

iPhone: <http://freebiesbug.com/psd-...>




1 ^ | v • Reply • Share ›



chris • 2 years ago

Great article, thanks. How do you determine the id of the uiview below another? I'd like to change the bottom uiview when pan gesture another on top. thanks

^ | v • Reply • Share ›

 [Subscribe](#)  [Add Disqus to your site](#) [Add Disqus](#) [Add](#)  [Privacy](#)

comments powered by Disqus

github.com/smnh

[Google+](#)

Copyright © 2015