AutoLayout

jano@jano.com.es

AutoLayout

To layout a view means setting the position and size of a view.

Layout

To layout a view means setting the position and size of a view.

Layout

To layout a view means setting the position and size of a view.

Layout

You can do it visually with your mouse, declaratively with code, or by description with the autoresizeMask.

Layout

bounds

center

You can do it visually with your mouse, declaratively with code, or by description with the autoresizeMask.

Layout

bounds
center
} frame

You can do it visually with your mouse, declaratively with code, or by description with the autoresizeMask.

Layout

bounds
center
} frame

```
frame.origin = center - (bounds.size / 2.0)
center = frame.origin + (bounds.size / 2.0)
frame.size = bounds.size
```


You can do it visually with your mouse, declaratively with code, or by description with the autoresizeMask.

Layout

bounds
center
transform
} frame

You can do it visually with your mouse, declaratively with code, or by description with the autoresizeMask.

Layout

bounds
center
} frame
transform
autoresizingMask

You can do it visually with your mouse, declaratively with code, or by description with the autoresizeMask.

Layout

bounds
center
} frame
transform
autoresizingMask

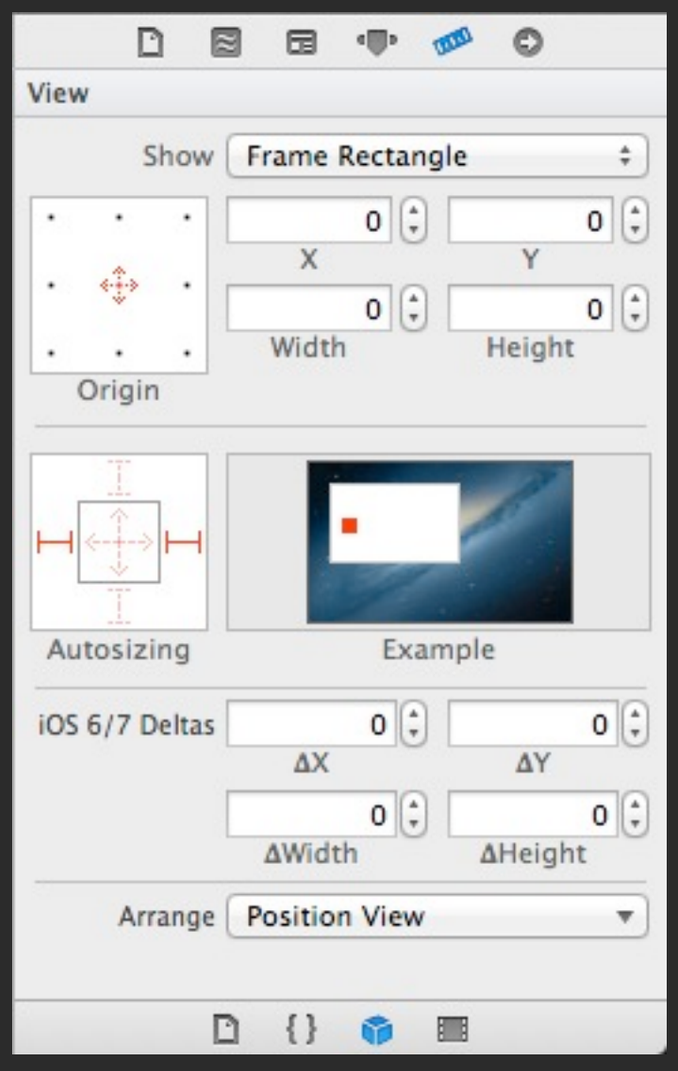
3/3

- bounds: view's position and size in its own coordinate system.
- center: center of the view.
- frame: position and size in the superview coordinate system, synthesized from bounds and center.
- transform: an affine transformation that lets you scale, rotate, or skew with the center as origin. A view's transform property alters how the view is drawn without affecting its bounds and center.

autoresizingMask is descriptive. That means we describe the result we expect, not the steps to accomplish it. For example, objective-c is mostly declarative, with traits of functional thanks to blocks, and even reactive if you add reactive-cocoa. While SQL is descriptive, SELECT USER FROM TABLE tells what we want, not how to get it.

Things autolayout can do that the mask can't:
- content-driven layout. eg: don't shrink an image beyond its natural size
- visual relations. eg: set a view to fill a % of the screen, tie the dimensions of two views
- priorities
- (...)

Layout



bounds
center
} frame
transform
autoresizingMask

3/3

- bounds: view's position and size in its own coordinate system.
- center: center of the view.
- frame: position and size in the superview coordinate system, synthesized from bounds and center.
- transform: an affine transformation that lets you scale, rotate, or skew with the center as origin. A view's transform property alters how the view is drawn without affecting its bounds and center.

autoresizingMask is descriptive. That means we describe the result we expect, not the steps to accomplish it. For example, objective-c is mostly declarative, with traits of functional thanks to blocks, and even reactive if you add reactive-cocoa. While SQL is descriptive, SELECT USER FROM TABLE tells what we want, not how to get it.

Things autolayout can do that the mask can't:
- content-driven layout. eg: don't shrink an image beyond its natural size
- visual relations. eg: set a view to fill a % of the screen, tie the dimensions of two views
- priorities
- (...)

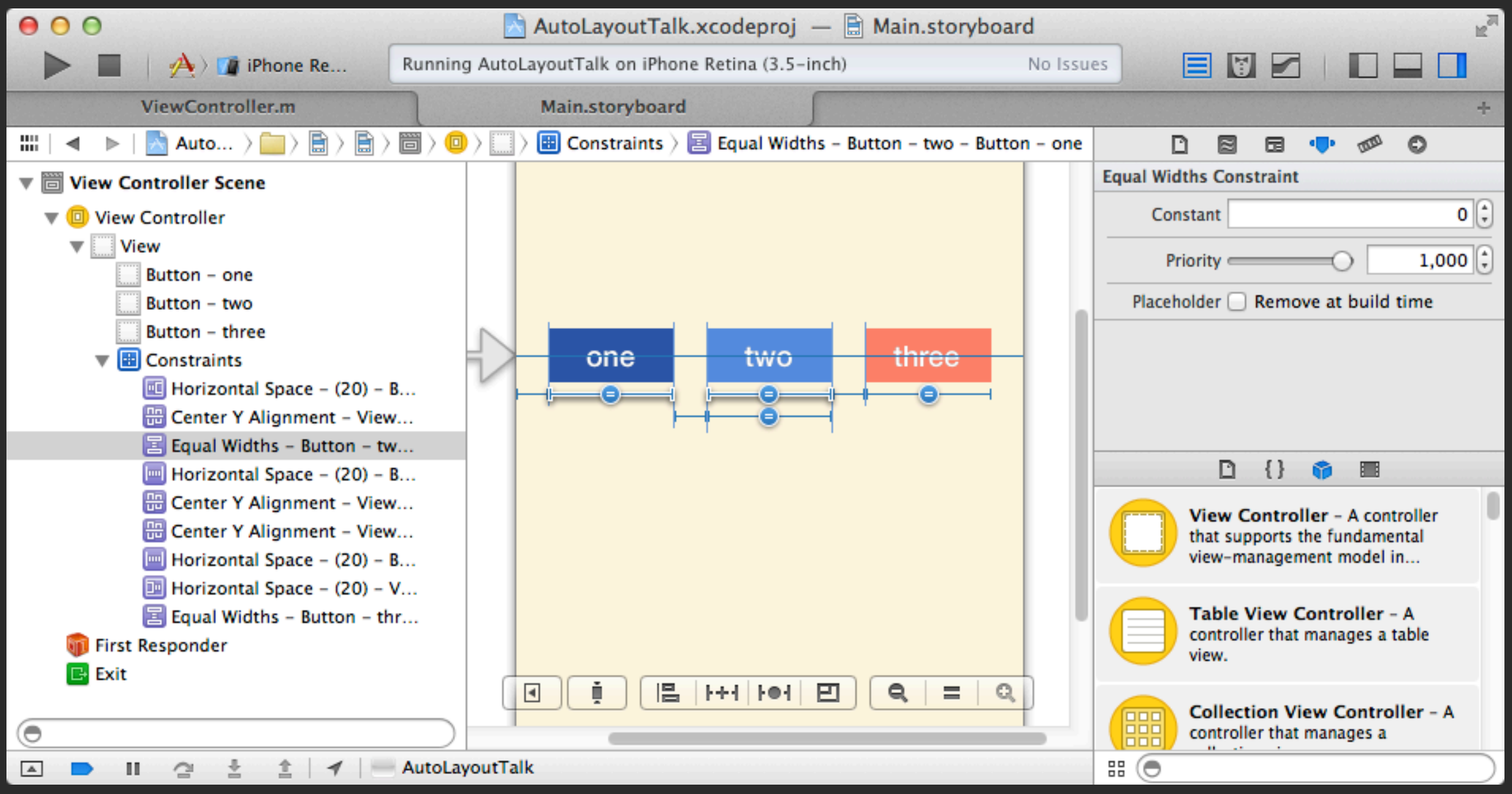
AutoLayout

It requires iOS 6 or OS X 10.7. In OS X since 2011.
Previous operative systems require two versions of the UI.

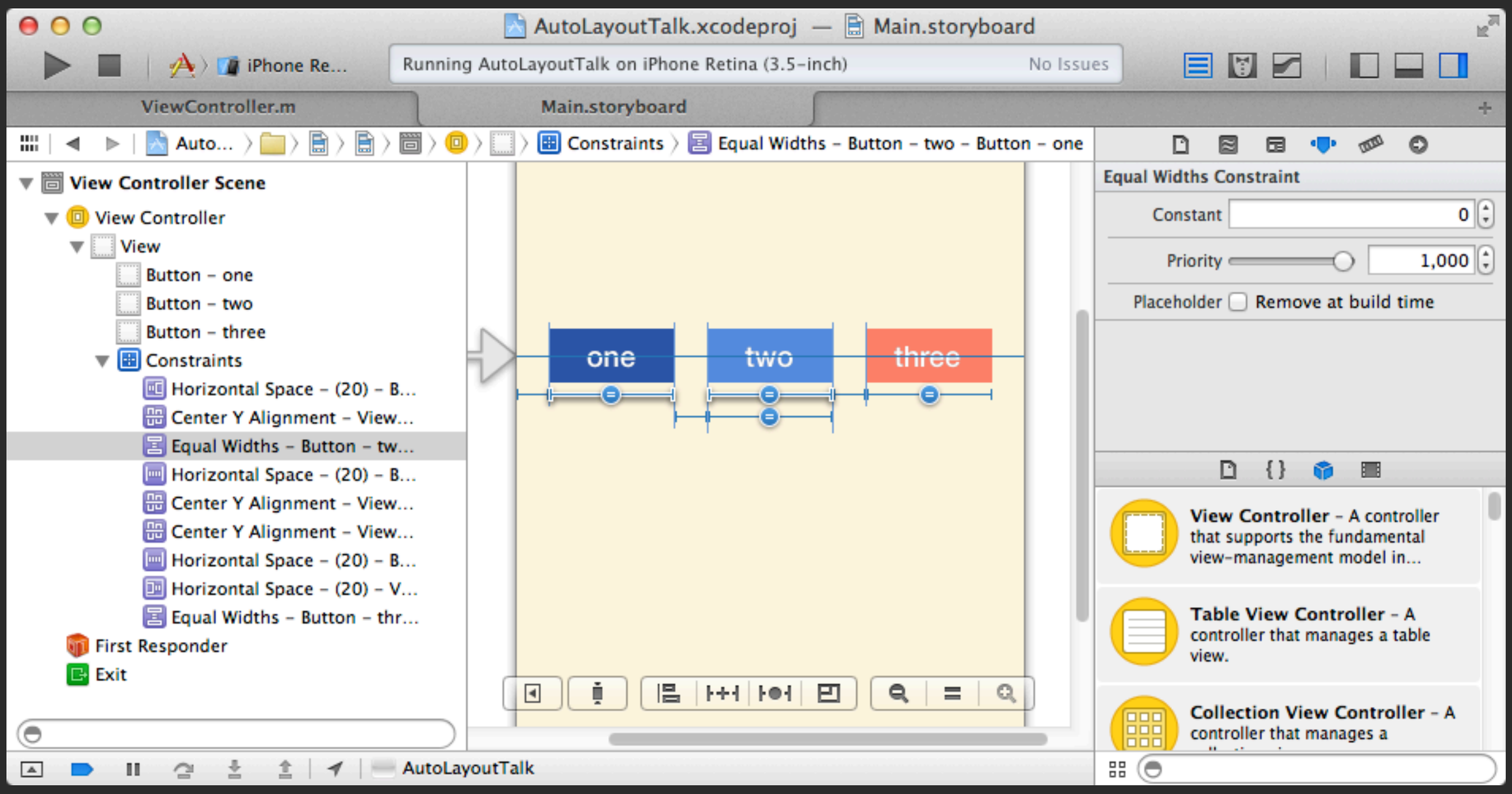
AutoLayout

It requires iOS 6 or OS X 10.7. In OS X since 2011.
Previous operative systems require two versions of the UI.

AutoLayout


AutoLayoutTalk.xcodeproj — Main.storyboard
iPhone Re...
Running AutoLayoutTalk on iPhone Retina (3.5-inch)
No Issues
ViewController.m
Main.storyboard
Auto...
Constraints
Equal Widths - Button - two - Button - one
View Controller Scene
View Controller
View
Button - one
Button - two
Button - three
Constraints
Horizontal Space - (20) - B...
Center Y Alignment - View...
Equal Widths - Button - tw...
Horizontal Space - (20) - B...
Center Y Alignment - View...
Center Y Alignment - View...
Horizontal Space - (20) - B...
Horizontal Space - (20) - V...
Equal Widths - Button - thr...
First Responder
Exit
one
two
three
Equal Widths Constraint
Constant
0
Priority
1,000
Placeholder
Remove at build time
View Controller - A controller that supports the fundamental view-management model in...
Table View Controller - A controller that manages a table view.
Collection View Controller - A controller that manages a
AutoLayoutTalk

AutoLayout



A **description** of a system is more flexible because it adapts to changes in the screen and i18n. Auto layout is descriptive and more expressive than the autoresizingMask. With auto layout we can say make that view the same size of that other view, or, fill 80% of the screen with a view, etc. We do this with constraints.

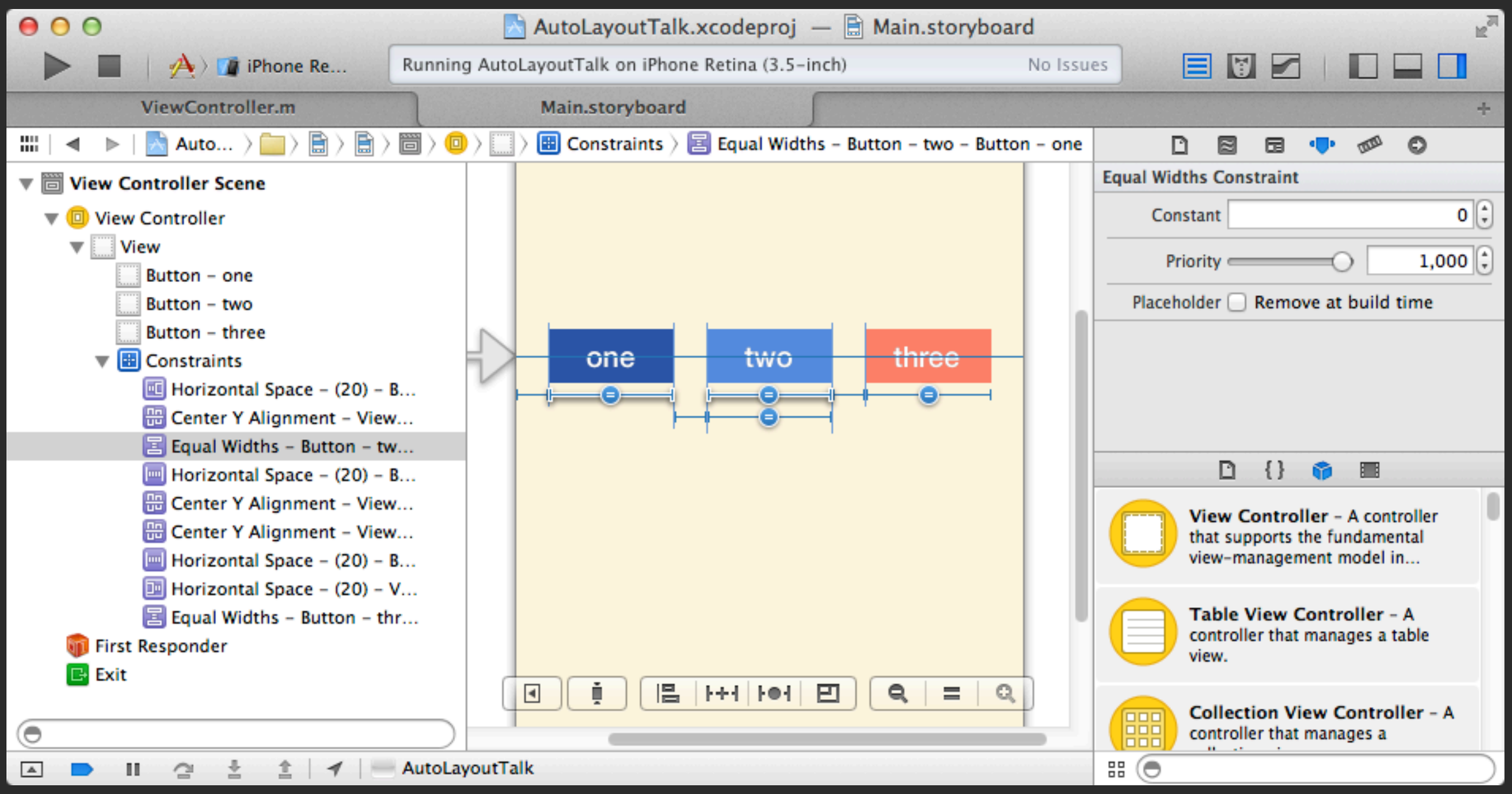
Constraints are geometric relationships between views. For example, a view is half the size of another.

Xcode can only relate two dimensions in the ways provided by the pull-down controls at the bottom. You can't make for example, a square (width=height).

You can mix Storyboards with NIBs.

AutoLayout expresses natural content.

AutoLayout



`view1.attribute1 RELATION multiplier * view2.attribute2 + constant`

A **description** of a system is more flexible because it adapts to changes in the screen and i18n. Auto layout is descriptive and more expressive than the autoresizingMask. With auto layout we can say make that view the same size of that other view, or, fill 80% of the screen with a view, etc. We do this with constraints.

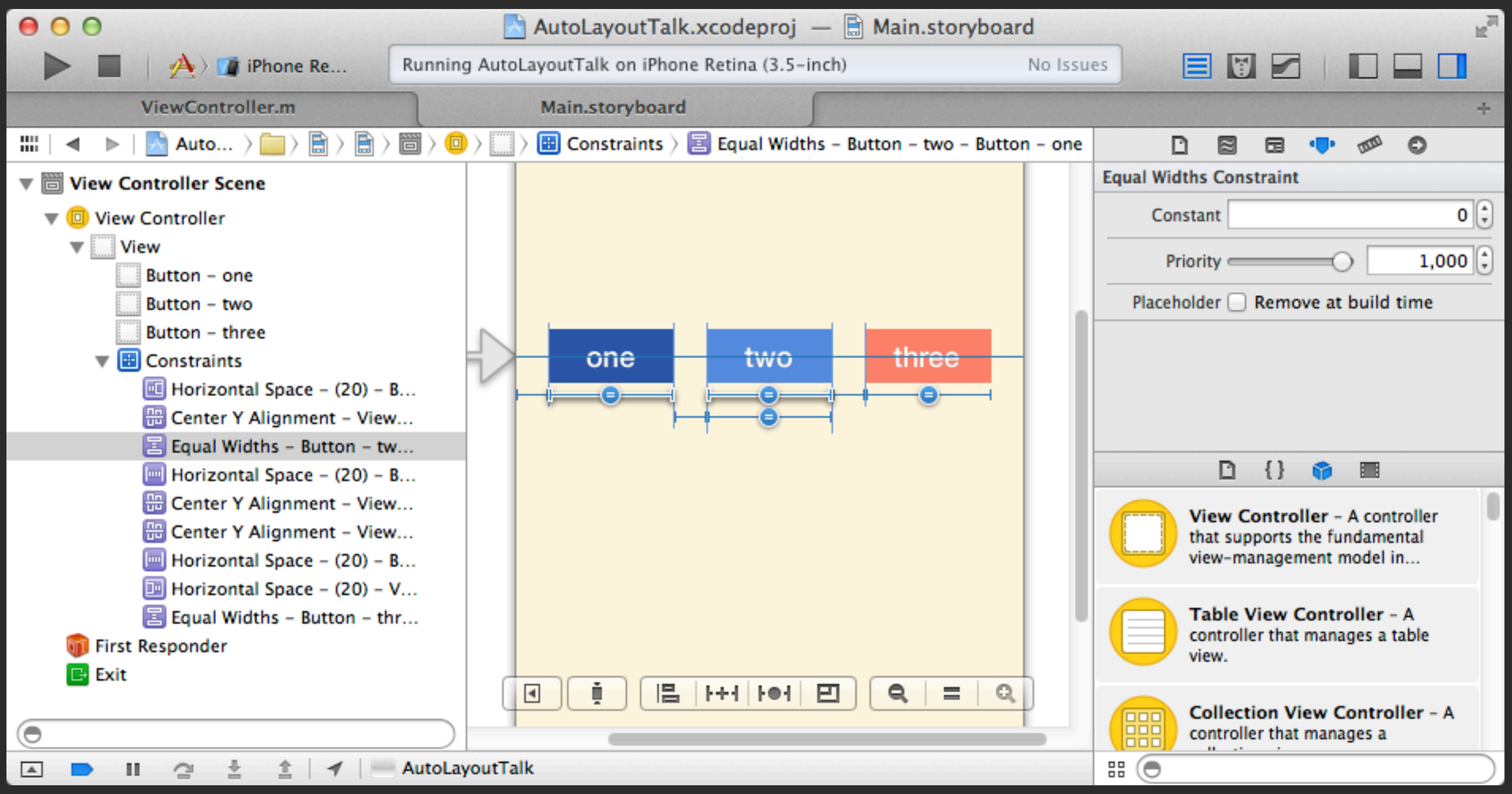
Constraints are geometric relationships between views. For example, a view is half the size of another.

Xcode can only relate two dimensions in the ways provided by the pull-down controls at the bottom. You can't make for example, a square (width=height).

You can mix Storyboards with NIBs.

AutoLayout expresses natural content.

AutoLayout




```
view1.attribute1 RELATION multiplier * view2.attribute2 + constant

button.center.y = 1 * superview.center.y + 0
```

A **description** of a system is more flexible because it adapts to changes in the screen and i18n. Auto layout is descriptive and more expressive than the autoresizingMask. With auto layout we can say make that view the same size of that other view, or, fill 80% of the screen with a view, etc. We do this with constraints.

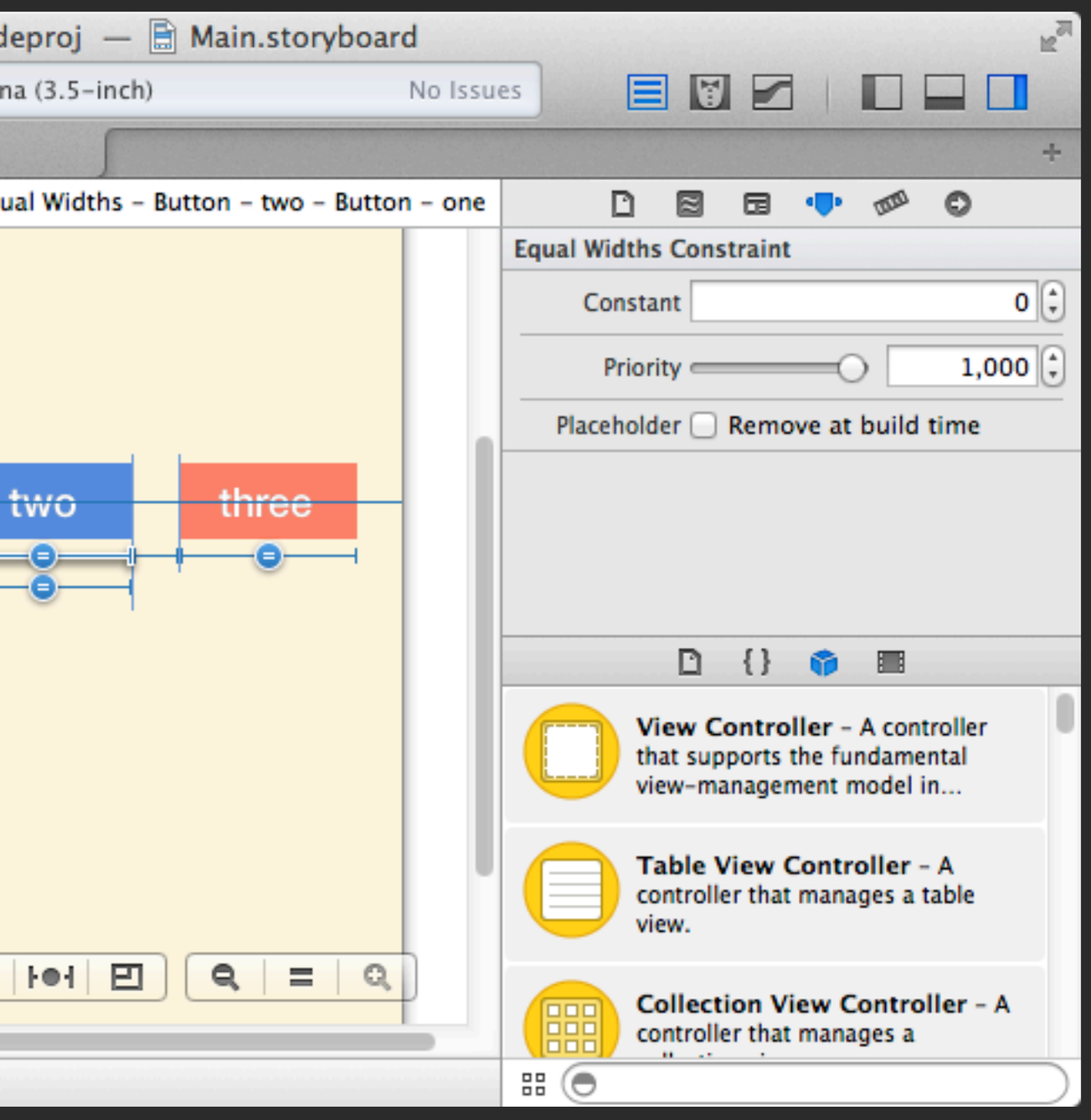
Constraints are geometric relationships between views. For example, a view is half the size of another.

Xcode can only relate two dimensions in the ways provided by the pull-down controls at the bottom. You can't make for example, a square (width=height).

You can mix Storyboards with NIBs.

AutoLayout expresses natural content.

AutoLayout




```
[NSLayoutConstraint
constraintWithItem: buttom
attribute: NSLayoutAttributeCenterY
relatedBy: NSLayoutRelationEqual
toItem: superview
attribute: NSLayoutAttributeCenterY
multiplier: 1.0
constant: 0.0]
```

```
view1.attribute1 RELATION multiplier * view2.attribute2 + constant

button.center.y = 1 * superview.center.y + 0
```

A **description** of a system is more flexible because it adapts to changes in the screen and i18n. Auto layout is descriptive and more expressive than the autoresizingMask. With auto layout we can say make that view the same size of that other view, or, fill 80% of the screen with a view, etc. We do this with constraints.

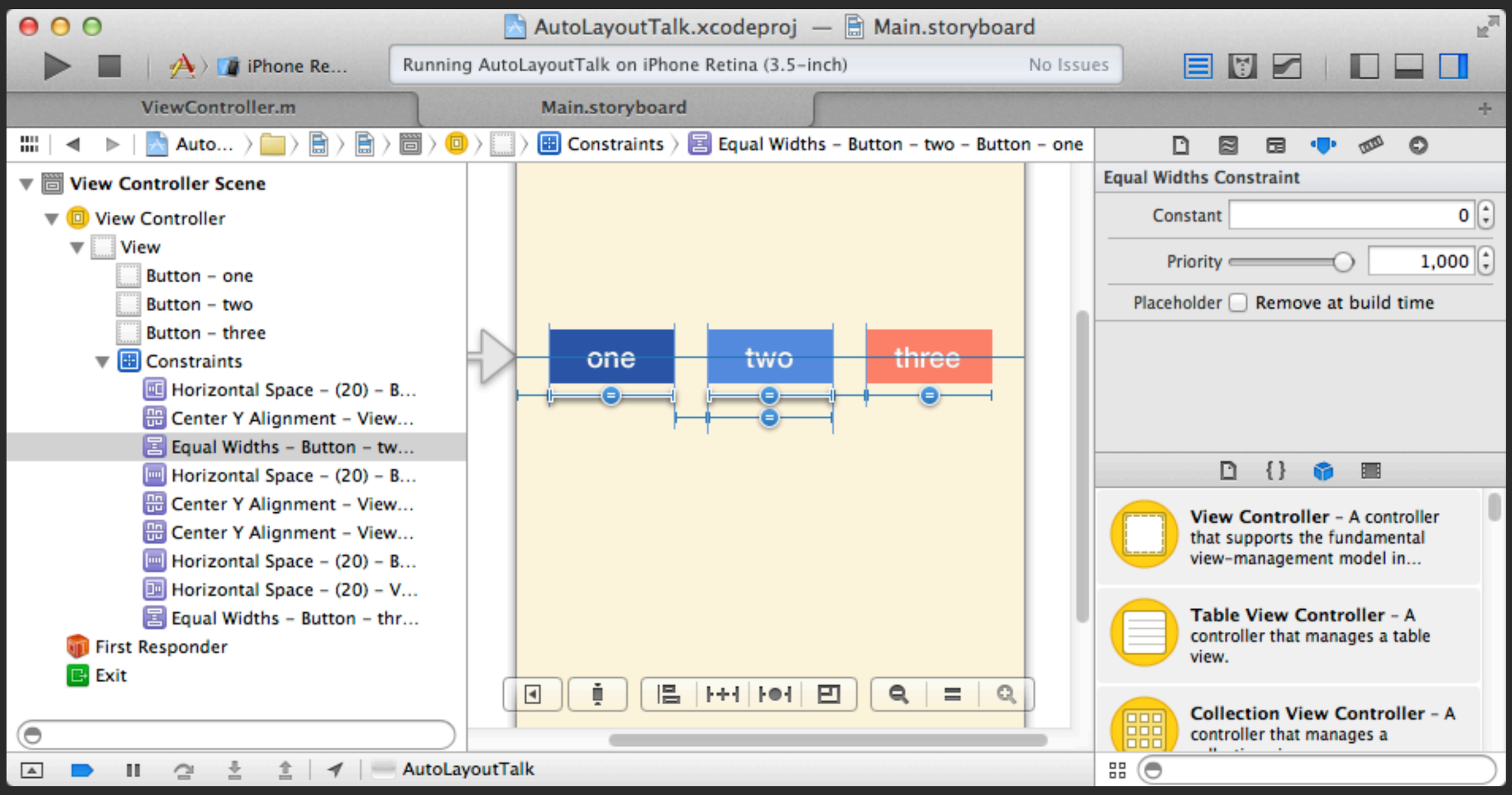
Constraints are geometric relationships between views. For example, a view is half the size of another.

Xcode can only relate two dimensions in the ways provided by the pull-down controls at the bottom. You can't make for example, a square (width=height).

You can mix Storyboards with NIBs.

AutoLayout expresses natural content.

AutoLayout



`view1.attribute1 RELATION multiplier * view2.attribute2 + constant`

`button.center.y = 1 * superview.center.y + 0`

A **description** of a system is more flexible because it adapts to changes in the screen and i18n. Auto layout is descriptive and more expressive than the autoresizingMask. With auto layout we can say make that view the same size of that other view, or, fill 80% of the screen with a view, etc. We do this with constraints.

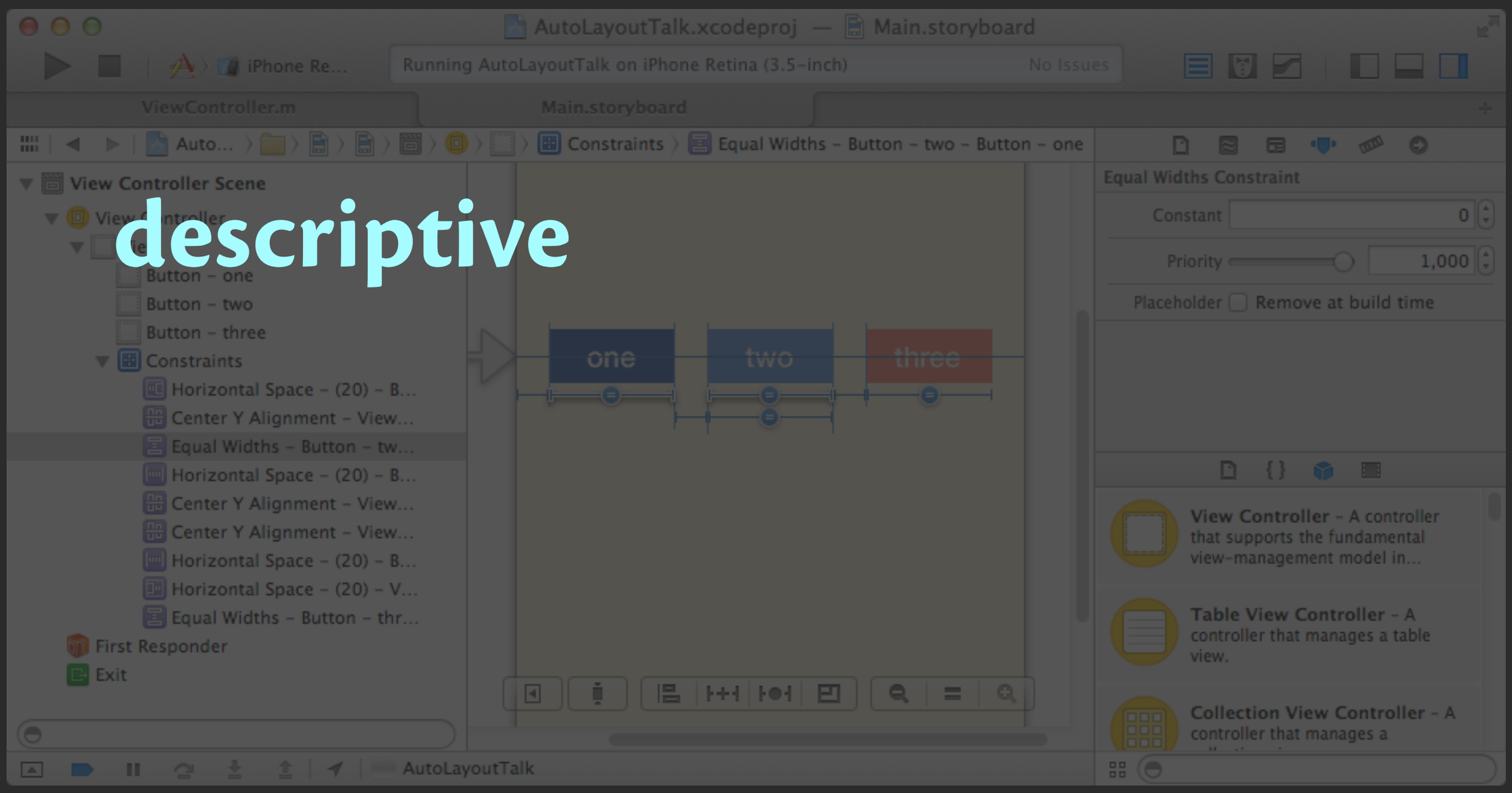
Constraints are geometric relationships between views. For example, a view is half the size of another.

Xcode can only relate two dimensions in the ways provided by the pull-down controls at the bottom. You can't make for example, a square (width=height).

You can mix Storyboards with NIBs.

AutoLayout expresses natural content.

AutoLayout



descriptive

view1.attribute1 RELATION multiplier * view2.attribute2 + constant

button.center.y = 1 * superview.center.y + 0

A **description** of a system is more flexible because it adapts to changes in the screen and i18n. Auto layout is descriptive and more expressive than the autoresizingMask. With auto layout we can say make that view the same size of that other view, or, fill 80% of the screen with a view, etc. We do this with constraints.

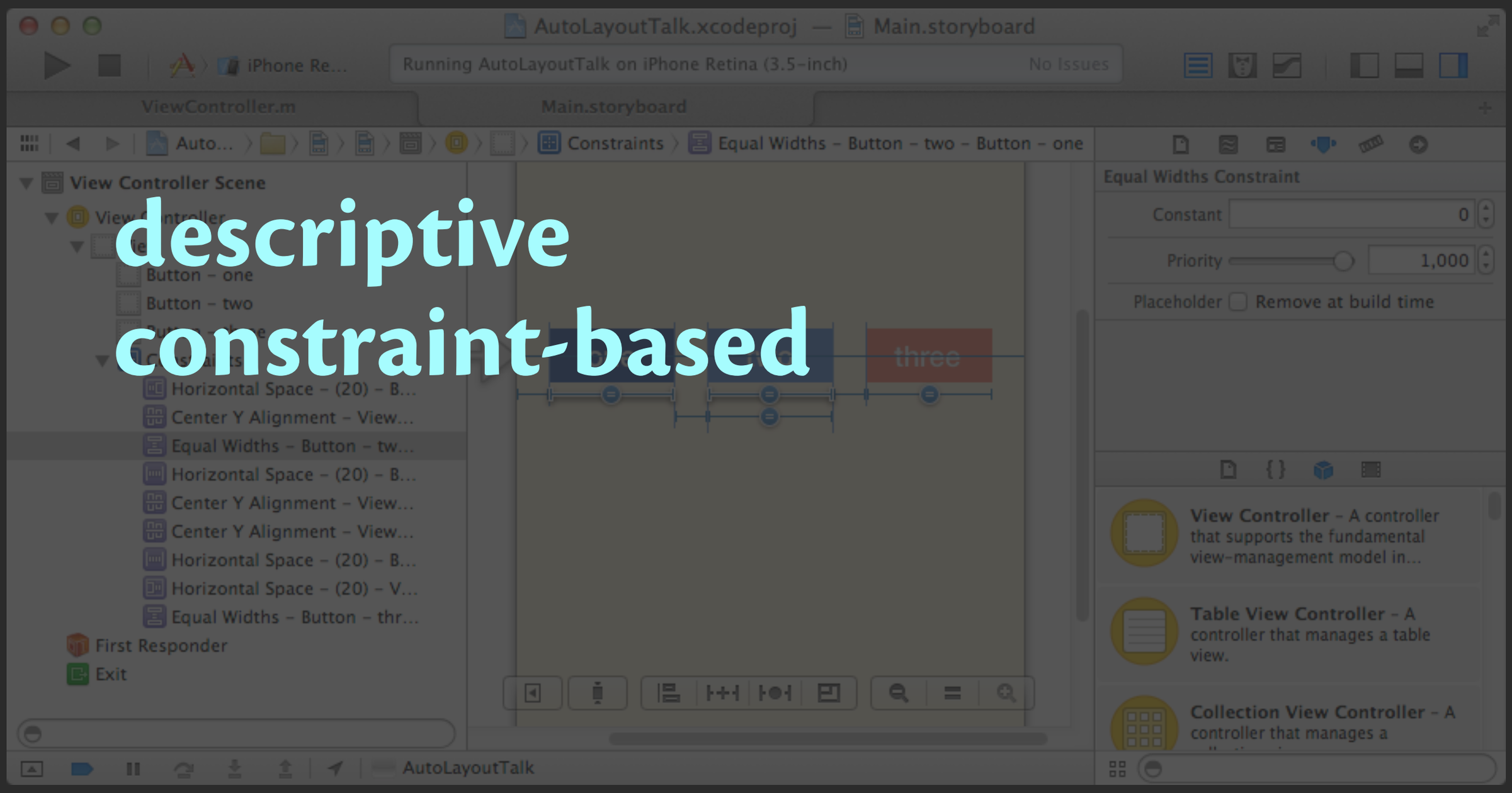
Constraints are geometric relationships between views. For example, a view is half the size of another.

Xcode can only relate two dimensions in the ways provided by the pull-down controls at the bottom. You can't make for example, a square (width=height).

You can mix Storyboards with NIBs.

AutoLayout expresses natural content.

AutoLayout



descriptive
constraint-based

```
view1.attribute1 RELATION multiplier * view2.attribute2 + constant

button.center.y = 1 * superview.center.y + 0
```

A **description** of a system is more flexible because it adapts to changes in the screen and i18n. Auto layout is descriptive and more expressive than the autoresizingMask. With auto layout we can say make that view the same size of that other view, or, fill 80% of the screen with a view, etc. We do this with constraints.

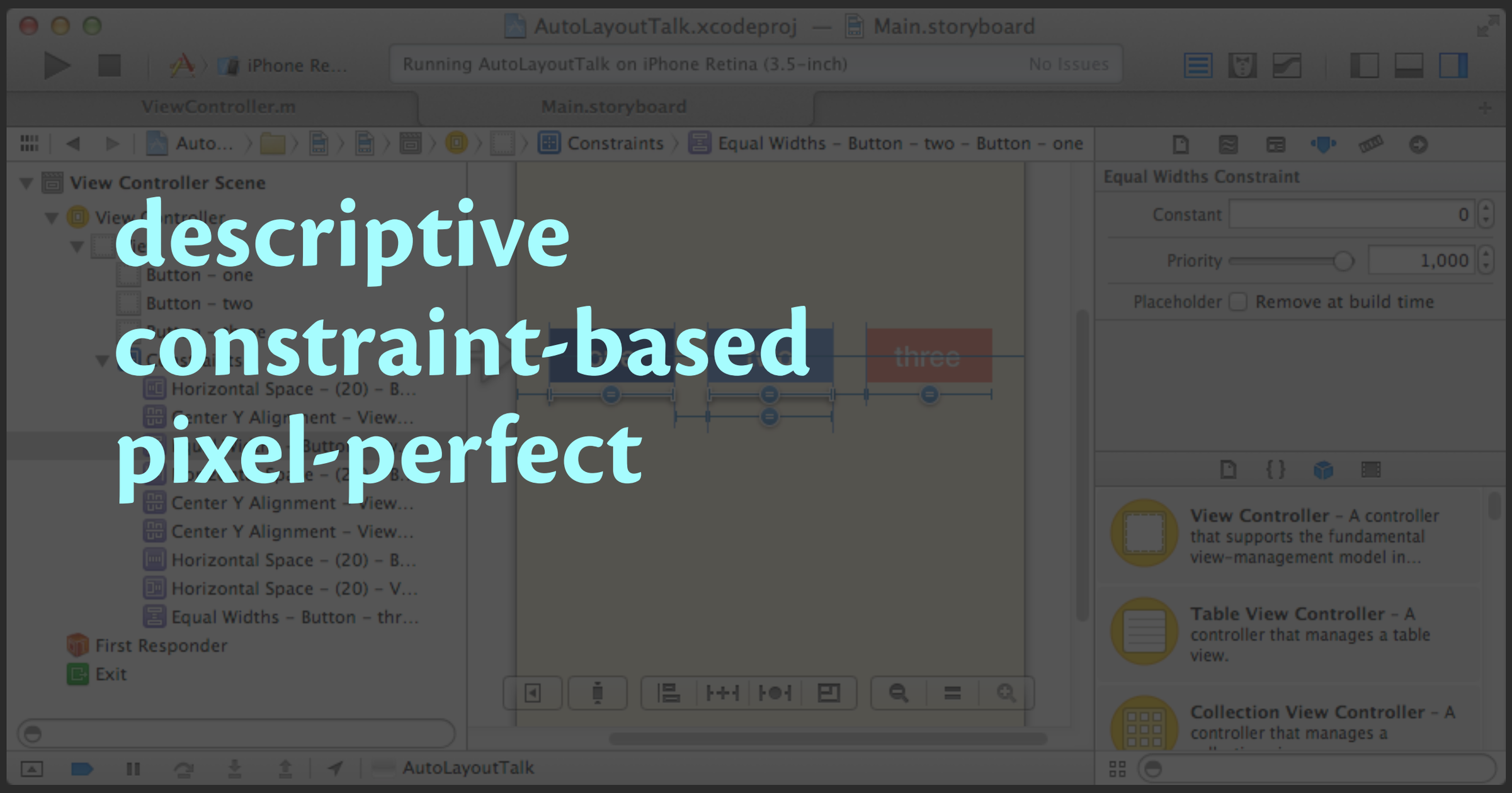
Constraints are geometric relationships between views. For example, a view is half the size of another.

Xcode can only relate two dimensions in the ways provided by the pull-down controls at the bottom. You can't make for example, a square (width=height).

You can mix Storyboards with NIBs.

AutoLayout expresses natural content.

AutoLayout



descriptive
constraint-based
pixel-perfect

```
view1.attribute1 RELATION multiplier * view2.attribute2 + constant

button.center.y = 1 * superview.center.y + 0
```

A **description** of a system is more flexible because it adapts to changes in the screen and i18n. Auto layout is descriptive and more expressive than the autoresizingMask. With auto layout we can say make that view the same size of that other view, or, fill 80% of the screen with a view, etc. We do this with constraints.

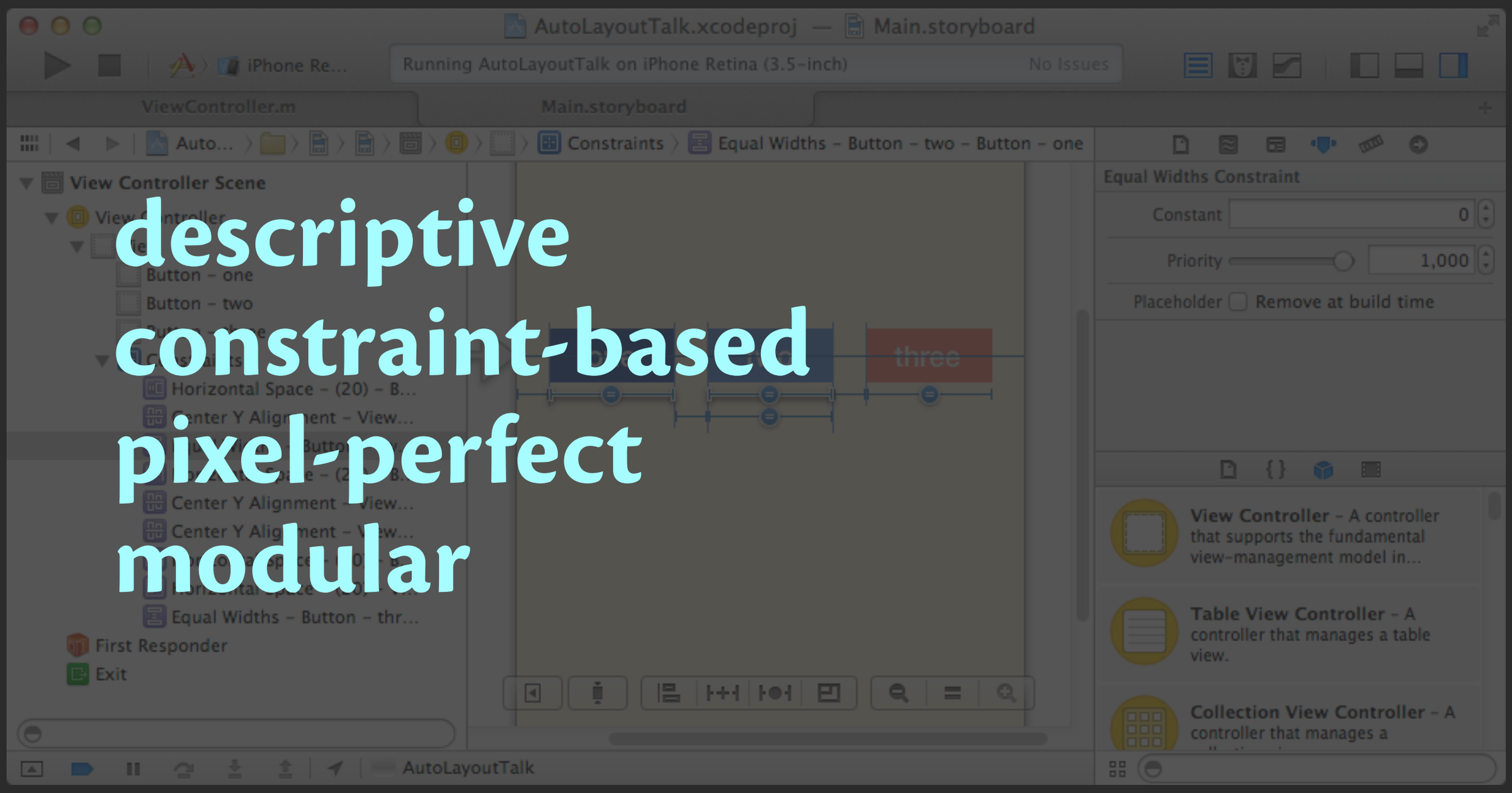
Constraints are geometric relationships between views. For example, a view is half the size of another.

Xcode can only relate two dimensions in the ways provided by the pull-down controls at the bottom. You can't make for example, a square (width=height).

You can mix Storyboards with NIBs.

AutoLayout expresses natural content.

AutoLayout



descriptive
constraint-based
pixel-perfect
modular

```
view1.attribute1 RELATION multiplier * view2.attribute2 + constant

button.center.y = 1 * superview.center.y + 0
```

A **description** of a system is more flexible because it adapts to changes in the screen and i18n. Auto layout is descriptive and more expressive than the autoresizingMask. With auto layout we can say make that view the same size of that other view, or, fill 80% of the screen with a view, etc. We do this with constraints.

Constraints are geometric relationships between views. For example, a view is half the size of another.

Xcode can only relate two dimensions in the ways provided by the pull-down controls at the bottom. You can't make for example, a square (width=height).

You can mix Storyboards with NIBs.

AutoLayout expresses natural content.

What do I need to know?

What do I need to know?

UIView properties

- alignmentRectInsets
- constraints
- contentSize
- compression
- hugging
- translatesAutoresizingMaskIntoConstraints

Content **Hugging** resists padding and stretching beyond its intrinsic size.

Compression [resistance] resists shrinking below its intrinsic size.
High values for both result in a fixed size.

When a view's **translatesAutoresizingMaskIntoConstraints** property is set to YES (the default), the runtime uses that view's autoresizing mask to produce matching constraints in the new Auto Layout system. This generates constraints with class NSAutoresizingMaskLayoutConstraint (unlike those you write using NSLayoutConstraint).

What do I need to know?

UIView properties

Interface Builder > VFL > API

What do I need to know?

UIView properties

Interface Builder > VFL > API

UIViewController lifecycle

What do I need to know?

UIView properties
Interface Builder > VFL > API
Animation

And lots of practice ofc.

UIView Properties

constraints

Layout	`NSLayoutConstraint`
Content	`NSContentSizeLayoutConstraint`
Autosizing	`NSAutoresizingMaskLayoutConstraint`

Content constraints are created with compression and hugging. They also appear for elements with intrinsic content size like images and buttons.
Autosizing constraints are created translating the autoresizing mask to constraints.
Only NSContentSizeLayoutConstraint is a public class.

All three classes appear in the Xcode logs. There are a few internal private related classes, but they are unimportant.

constraints

```
view1.attribute1 RELATION multiplier * view2.attribute2 + constant
```

```
NSLayoutAttributeLeft
NSLayoutAttributeRight
NSLayoutAttributeTop
NSLayoutAttributeBottom
NSLayoutAttributeLeading
NSLayoutAttributeTrailing
NSLayoutAttributeWidth
NSLayoutAttributeHeight
NSLayoutAttributeCenterX
NSLayoutAttributeCenterY
NSLayoutAttributeBaseline

NSLayoutAttributeNotAnAttribute
```

```
NSLayoutRelationEqual
NSLayoutRelationGreaterThanOrEqual
NSLayoutRelationLessThanOrEqual
```

All the views a constraint references must be in the subtree of the view it is being added to. In general, add the constraints to the nearest ancestor for better performance, or to the component it logically belongs to.

Baseline is only available in OS X. It’s used to render content like text.

Some pairings are not allowed, like “leading space equals width”.

constraints

```
view1.attribute1 RELATION multiplier * view2.attribute2 + constant
```

```
NSLayoutAttributeLeft
NSLayoutAttributeRight
NSLayoutAttributeTop
NSLayoutAttributeBottom
NSLayoutAttributeLeading
NSLayoutAttributeTrailing
NSLayoutAttributeWidth
NSLayoutAttributeHeight
NSLayoutAttributeCenterX
NSLayoutAttributeCenterY
NSLayoutAttributeBaseline

NSLayoutAttributeNotAnAttribute
```

```
NSLayoutRelationEqual
NSLayoutRelationGreaterThanOrEqual
NSLayoutRelationLessThanOrEqual
```

All the views a constraint references must be in the subtree of the view it is being added to. In general, add the constraints to the nearest ancestor for better performance, or to the component it logically belongs to.

Baseline is only available in OS X. It's used to render content like text.

Some pairings are not allowed, like "leading space equals width".

constraints

```
view1.attribute1 RELATION multiplier * view2.attribute2 + constant
```

```
NSLayoutAttributeLeft
NSLayoutAttributeRight
NSLayoutAttributeTop
NSLayoutAttributeBottom
NSLayoutAttributeLeading
NSLayoutAttributeTrailing
NSLayoutAttributeWidth
NSLayoutAttributeHeight
NSLayoutAttributeCenterX
NSLayoutAttributeCenterY
NSLayoutAttributeBaseline

NSLayoutAttributeNotAnAttribute
```

```
NSLayoutRelationEqual
NSLayoutRelationGreaterThanOrEqual
NSLayoutRelationLessThanOrEqual
```

All the views a constraint references must be in the subtree of the view it is being added to. In general, add the constraints to the nearest ancestor for better performance, or to the component it logically belongs to.

Baseline is only available in OS X. It's used to render content like text.

Some pairings are not allowed, like "leading space equals width".

constraints

```
view1.attribute1 RELATION multiplier * view2.attribute2 + constant
```

```
NSLayoutAttributeLeft
NSLayoutAttributeRight
NSLayoutAttributeTop
NSLayoutAttributeBottom
NSLayoutAttributeLeading
NSLayoutAttributeTrailing
NSLayoutAttributeWidth
NSLayoutAttributeHeight
NSLayoutAttributeCenterX
NSLayoutAttributeCenterY
NSLayoutAttributeBaseline

NSLayoutAttributeNotAnAttribute
```

```
NSLayoutRelationEqual
NSLayoutRelationGreaterThanOrEqual
NSLayoutRelationLessThanOrEqual
```

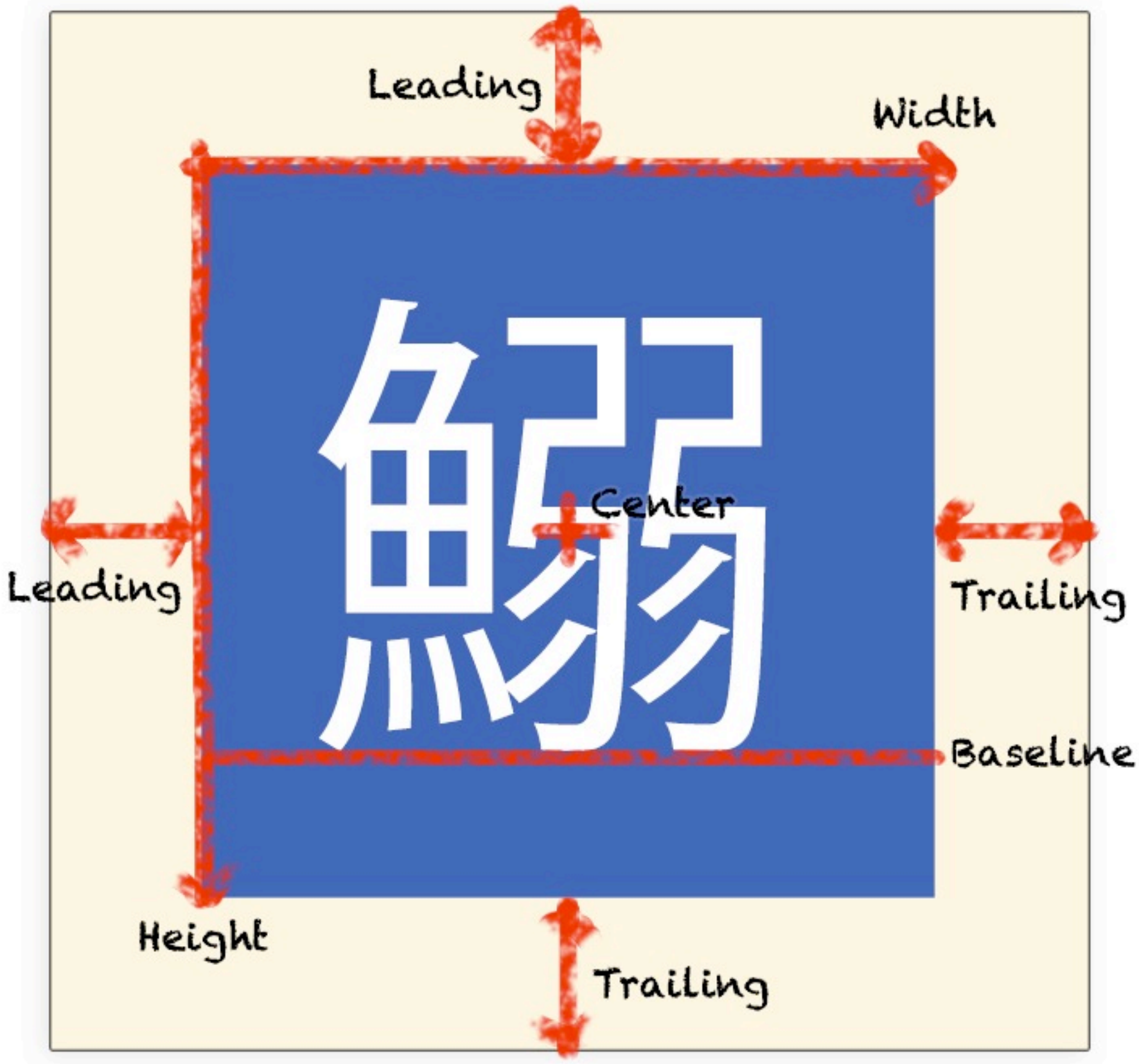
All the views a constraint references must be in the subtree of the view it is being added to. In general, add the constraints to the nearest ancestor for better performance, or to the component it logically belongs to.

Baseline is only available in OS X. It's used to render content like text.

Some pairings are not allowed, like "leading space equals width".

```
view1.attribute
AttributeLeft
AttributeRight
AttributeTop
AttributeBottom
AttributeLeading
AttributeTrailing
AttributeWidth
AttributeHeight
AttributeCenterX
AttributeCenterY
AttributeBaseline

AttributeNotAnAttr
```



All the views a constraint references must be in the subtree of the view it is being added to. In general, add the constraints to the nearest ancestor for better performance, or to the component it logically belongs to.

Baseline is only available in OS X. It’s used to render content like text.

Some pairings are not allowed, like “leading space equals width”.

constraints

NSLayoutConstraint

Tasks

Creating Constraints

```
+ constraintsWithVisualFormat:options:metrics:views:
+ constraintWithItem:attribute:relatedBy:toItem:attribute:multiplier:constant:
```

Accessing Constraint Data

```
priority property
firstItem property
firstAttribute property
relation property
secondItem property
secondAttribute property
multiplier property
constant property
```

Controlling Constraint Archiving

```
shouldBeArchived property
```

```
[NSLayoutConstraint
 constraintWithItem: button
 attribute: NSLayoutAttributeCenterX
 relatedBy: NSLayoutRelationEqual
 toItem: superview
 attribute: NSLayoutAttributeCenterX
 multiplier: 1.0
 constant: 0.0]
```

The second element may be nil. When constraints reference two views, these views must always belong to the same view hierarchy and bounds system.

"A constraint is typically installed on the closest common ancestor of the views involved in the constraint. It is required that a constraint be installed on a common ancestor of every view involved. The numbers in a constraint are interpreted in the coordinate system of the view it is installed on. A view is considered to be an ancestor of itself."

Items are typed as nil to allow NSView and UIView.

constraints

```
view1.attribute1 RELATION multiplier * view2.attribute2 + constant
```

```
constraint = [NSLayoutConstraint
              constraintWithItem: view
              attribute: NSLayoutAttributeWidth
              relatedBy: NSLayoutRelationEqual
              toItem: nil
              attribute: NSLayoutAttributeNotAnAttribute
              multiplier: 1.0
              constant: 100.0];
[view addConstraint: constraint];

constraint = [NSLayoutConstraint
              constraintWithItem: view
              attribute: NSLayoutAttributeHeight
              relatedBy: NSLayoutRelationEqual
              toItem: nil
              attribute: NSLayoutAttributeNotAnAttribute
              multiplier: 1.0
              constant: 80.0];
[view addConstraint: constraint];
```

size=100x80

MACROS.

constraints

```
view1.attribute1 RELATION multiplier * view2.attribute2 + constant
```

```
CONSTRAINT_SIZE(view,100,80);
```

MACROS.

constraints

```
view1.attribute1 RELATION multiplier * view2.attribute2 + constant
```

linear equations

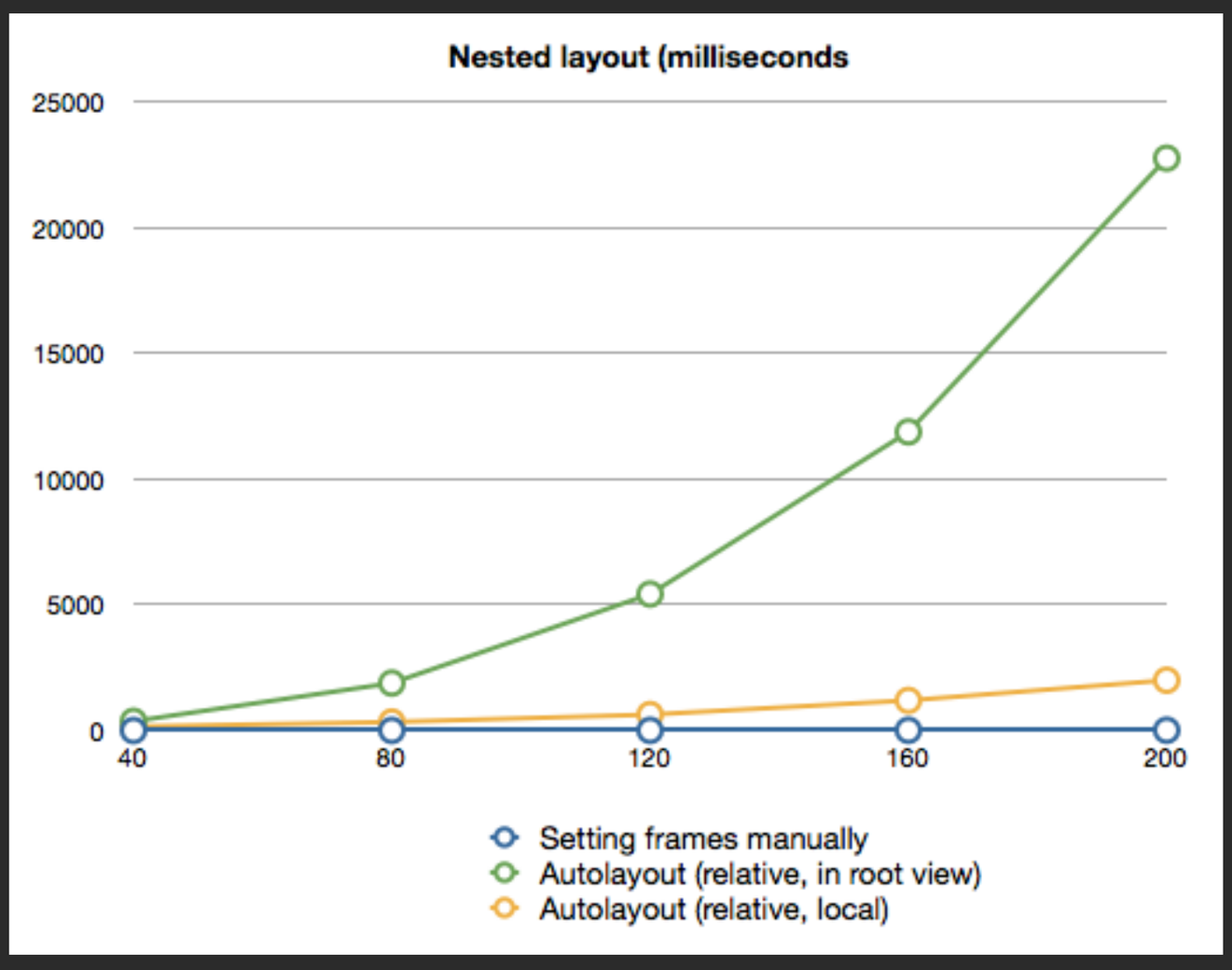
Cassowary Linear Arithmetic Constraint Solving Algorithm

Pro tip: Use local flat hierarchies.

Constraints form a system of linear equations of polynomial complexity. Thus, the time to render increases fast with the number of views. It is solved with the Cassowary algorithm. Results are cached, so additional constraints are solved incrementally (it doesn't recalculate the whole thing).

constraints

`view1.attribute1 RELATION multiplier * view2.attribute2 + constant`



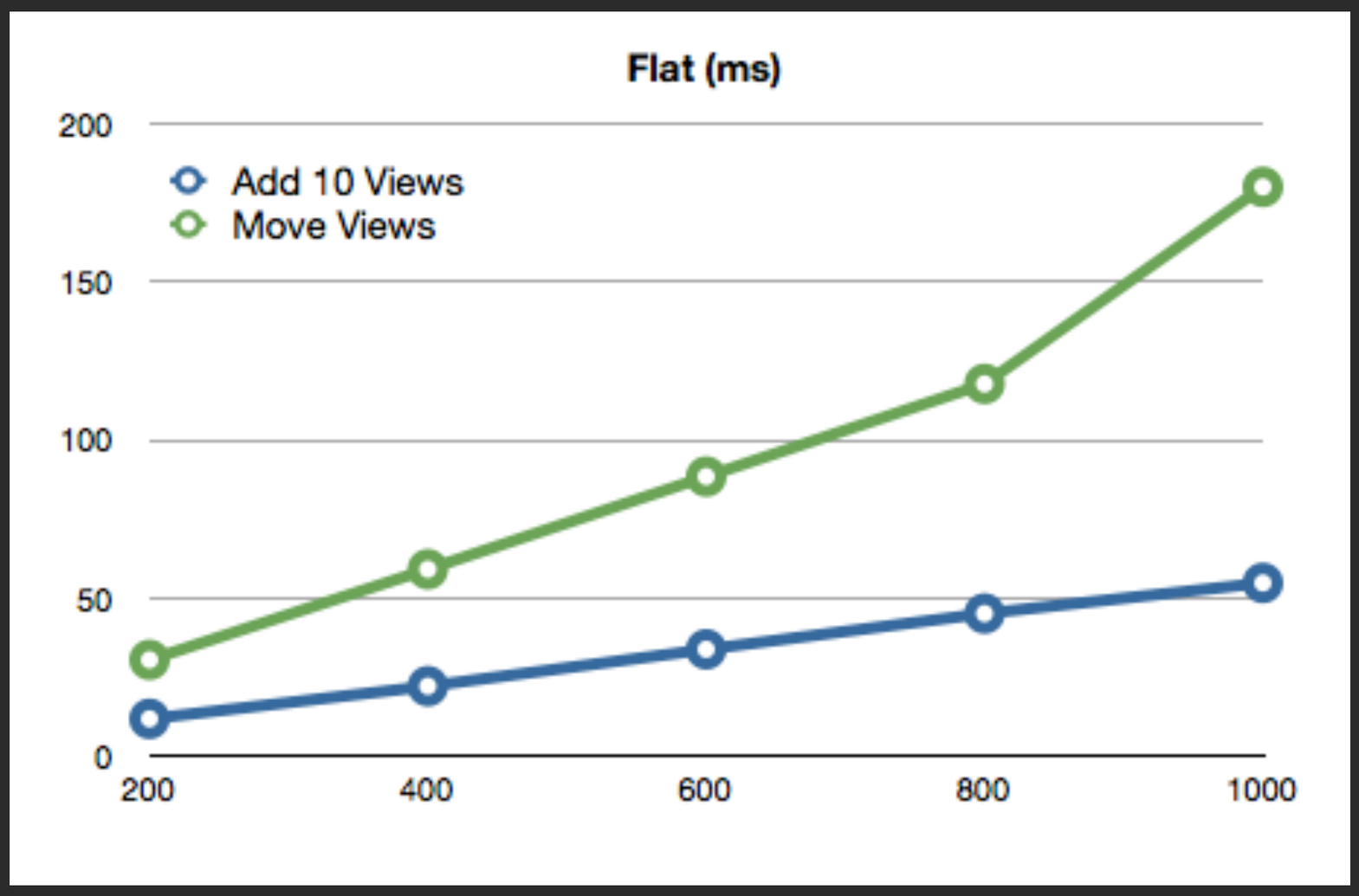
http://pilky.me/view/36

Autolayout is faster with constraints added to the closest parent.
If we add all constraints to the root view, we see a polynomic curve.

http://pilky.me/view/36

constraints

```
view1.attribute1 RELATION multiplier * view2.attribute2 + constant
```



http://pilky.me/view/36

If the hierarchy is flat, adding more views increases the render time linearly.

http://pilky.me/view/36

Unsatisfiability means no layout can satisfy the required constraints. eg: a view can't be left and right of another view. If the runtime can't satisfy constraints, it makes a choice and output a message.

Ambiguity means there are multiple layouts that satisfy all constraints. The results are unpredictable. You have to solve it adding constraints, or maybe changing priorities so one of them wins. You need 2 constraints per axis for x,y, width,height.

Invalid constraints terminate the application with the proper error message. For example, a syntax error in the visual language, or a constraint that relates height to width.

Xcode lets you create ambiguous layout (some constraints are missing), but not unsatisfiable or invalid layouts.

If translatesAutoresizingMaskIntoConstraints=NO, the view is NOT initialized to the size of the frame. And if you don't set a size with intrinsicContentSize or a constraint, the view becomes size 0,0, thus, invisible.

Unsatisfiability means no layout can satisfy the required constraints. eg: a view can't be left and right of another view. If the runtime can't satisfy constraints, it makes a choice and output a message.

Ambiguity means there are multiple layouts that satisfy all constraints. The results are unpredictable. You have to solve it adding constraints, or maybe changing priorities so one of them wins. You need 2 constraints per axis for x,y, width,height.

Invalid constraints terminate the application with the proper error message. For example, a syntax error in the visual language, or a constraint that relates height to width.

Xcode lets you create ambiguous layout (some constraints are missing), but not unsatisfiable or invalid layouts.

If translatesAutoresizingMaskIntoConstraints=NO, the view is NOT initialized to the size of the frame. And if you don't set a size with intrinsicContentSize or a constraint, the view becomes size 0,0, thus, invisible.

Unsatisfiability means no layout can satisfy the required constraints. eg: a view can't be left and right of another view. If the runtime can't satisfy constraints, it makes a choice and output a message.

Ambiguity means there are multiple layouts that satisfy all constraints. The results are unpredictable. You have to solve it adding constraints, or maybe changing priorities so one of them wins. You need 2 constraints per axis for x,y, width,height.

Invalid constraints terminate the application with the proper error message. For example, a syntax error in the visual language, or a constraint that relates height to width.

Xcode lets you create ambiguous layout (some constraints are missing), but not unsatisfiable or invalid layouts.

If translatesAutoresizingMaskIntoConstraints=NO, the view is NOT initialized to the size of the frame. And if you don't set a size with intrinsicContentSize or a constraint, the view becomes size 0,0, thus, invisible.

Unsatisfiability means no layout can satisfy the required constraints. eg: a view can't be left and right of another view. If the runtime can't satisfy constraints, it makes a choice and output a message.

Ambiguity means there are multiple layouts that satisfy all constraints. The results are unpredictable. You have to solve it adding constraints, or maybe changing priorities so one of them wins. You need 2 constraints per axis for x,y, width,height.

Invalid constraints terminate the application with the proper error message. For example, a syntax error in the visual language, or a constraint that relates height to width.

Xcode lets you create ambiguous layout (some constraints are missing), but not unsatisfiable or invalid layouts.

If translatesAutoresizingMaskIntoConstraints=NO, the view is NOT initialized to the size of the frame. And if you don't set a size with intrinsicContentSize or a constraint, the view becomes size 0,0, thus, invisible.

Unsatisfiability means no layout can satisfy the required constraints. eg: a view can't be left and right of another view. If the runtime can't satisfy constraints, it makes a choice and output a message.

Ambiguity means there are multiple layouts that satisfy all constraints. The results are unpredictable. You have to solve it adding constraints, or maybe changing priorities so one of them wins. You need 2 constraints per axis for x,y, width,height.

Invalid constraints terminate the application with the proper error message. For example, a syntax error in the visual language, or a constraint that relates height to width.

Xcode lets you create ambiguous layout (some constraints are missing), but not unsatisfiable or invalid layouts.

If translatesAutoresizingMaskIntoConstraints=NO, the view is NOT initialized to the size of the frame. And if you don't set a size with intrinsicContentSize or a constraint, the view becomes size 0,0, thus, invisible.

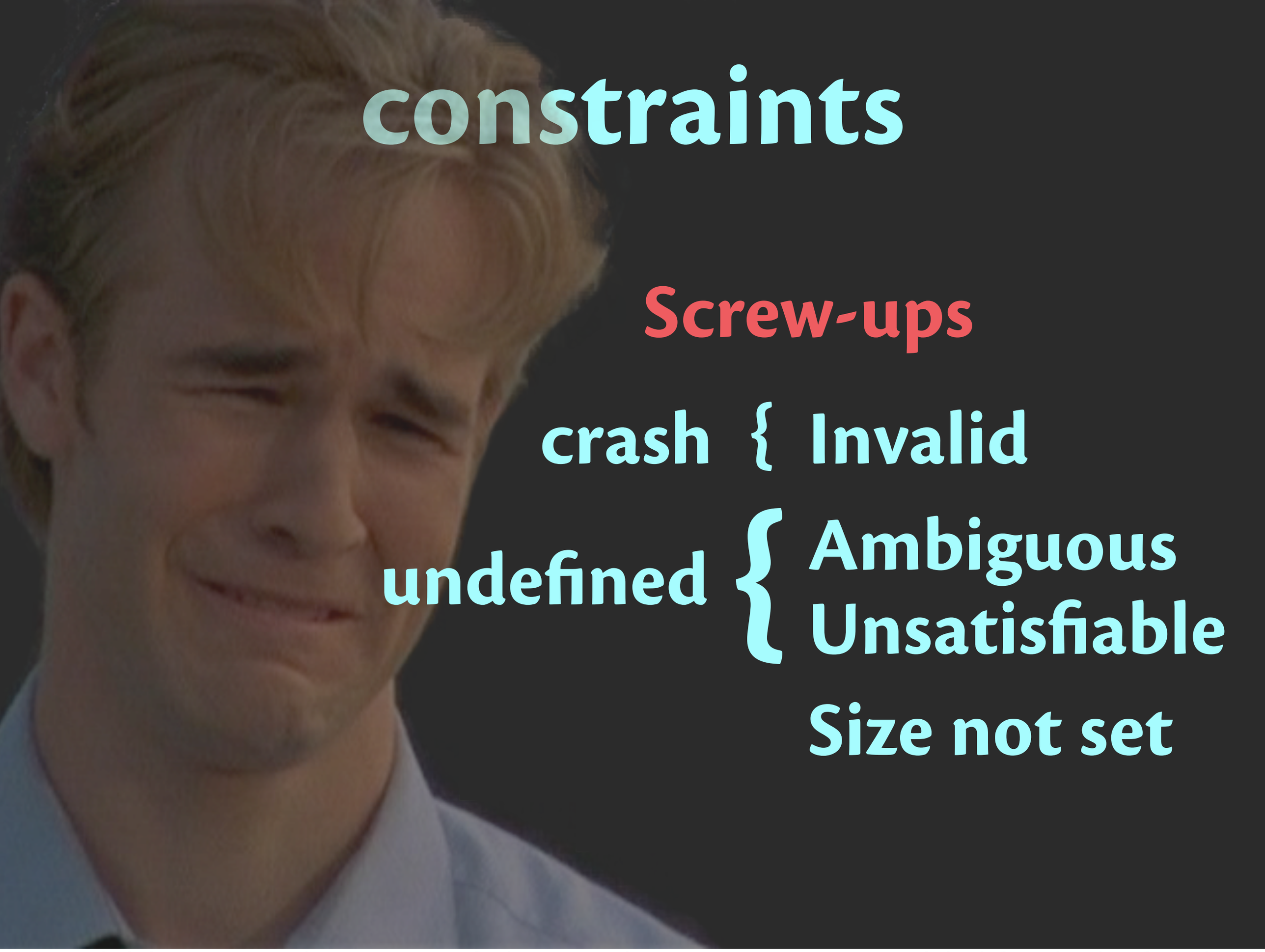
Unsatisfiability means no layout can satisfy the required constraints. eg: a view can't be left and right of another view. If the runtime can't satisfy constraints, it makes a choice and output a message.

Ambiguity means there are multiple layouts that satisfy all constraints. The results are unpredictable. You have to solve it adding constraints, or maybe changing priorities so one of them wins. You need 2 constraints per axis for x,y, width,height.

Invalid constraints terminate the application with the proper error message. For example, a syntax error in the visual language, or a constraint that relates height to width.

Xcode lets you create ambiguous layout (some constraints are missing), but not unsatisfiable or invalid layouts.

If translatesAutoresizingMaskIntoConstraints=NO, the view is NOT initialized to the size of the frame. And if you don't set a size with intrinsicContentSize or a constraint, the view becomes size 0,0, thus, invisible.



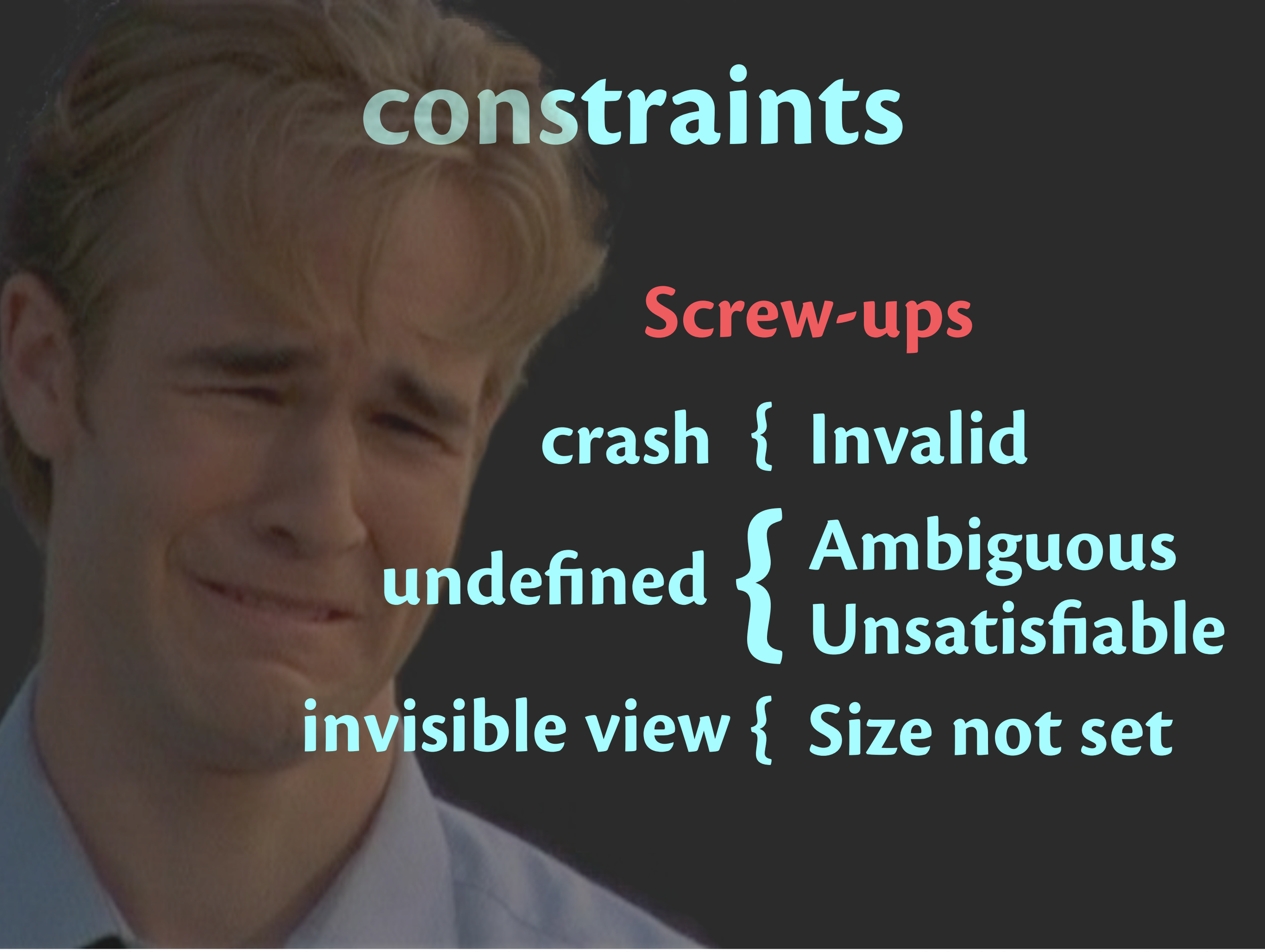
Unsatisfiability means no layout can satisfy the required constraints. eg: a view can't be left and right of another view. If the runtime can't satisfy constraints, it makes a choice and output a message.

Ambiguity means there are multiple layouts that satisfy all constraints. The results are unpredictable. You have to solve it adding constraints, or maybe changing priorities so one of them wins. You need 2 constraints per axis for x,y, width,height.

Invalid constraints terminate the application with the proper error message. For example, a syntax error in the visual language, or a constraint that relates height to width.

Xcode lets you create ambiguous layout (some constraints are missing), but not unsatisfiable or invalid layouts.

If translatesAutoresizingMaskIntoConstraints=NO, the view is NOT initialized to the size of the frame. And if you don't set a size with intrinsicContentSize or a constraint, the view becomes size 0,0, thus, invisible.



Unsatisfiability means no layout can satisfy the required constraints. eg: a view can't be left and right of another view. If the runtime can't satisfy constraints, it makes a choice and output a message.

Ambiguity means there are multiple layouts that satisfy all constraints. The results are unpredictable. You have to solve it adding constraints, or maybe changing priorities so one of them wins. You need 2 constraints per axis for x,y, width,height.

Invalid constraints terminate the application with the proper error message. For example, a syntax error in the visual language, or a constraint that relates height to width.

Xcode lets you create ambiguous layout (some constraints are missing), but not unsatisfiable or invalid layouts.

If translatesAutoresizingMaskIntoConstraints=NO, the view is NOT initialized to the size of the frame. And if you don't set a size with intrinsicContentSize or a constraint, the view becomes size 0,0, thus, invisible.

Unsatisfiable

Errors 〉 Errors 〉 ViewController.m 〉 @implementation ViewController

```
#import "ViewController.h"

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    NSDictionary *viewsDictionary = NSDictionaryOfVariableBindings(btnOne, btnTwo);
    for (NSString *format in @[@"H:|[btnOne][btnTwo]|",@"H:|[btnTwo][btnOne]|"]){
        [self.view addConstraints:
         [NSLayoutConstraint constraintsWithVisualFormat:format
          options:0 metrics:0 views:viewsDictionary]];
    }
}

@end
```

Errors.app

```
2013-07-03 14:59:25.205 Errors[1050:a0b] Cannot find executable for CFBundle 0xa0b2d70 </Applications/Xcode5-DP2.app/Contents/
Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator7.0.sdk/System/Library/AccessibilityBundles/
CertUIFramework.axbundle> (not loaded)
2013-07-03 14:59:25.287 Errors[1050:a0b] Unable to simultaneously satisfy constraints.
    Probably at least one of the constraints in the following list is one you don't want. Try this: (1) look at each constraint and
try to figure out which you don't expect; (2) find the code that added the unwanted constraint or constraints and fix it. (Note: If
you're seeing NSAutoresizingMaskLayoutConstraints that you don't understand, refer to the documentation for the UIView property
translatesAutoresizingMaskIntoConstraints)
(
    "<NSLayoutConstraint:0xa0a14a0 'IB auto generated at build time for view with fixed frame' H:|-(42)-[UIButton:0xa0c11e0](LTR)
(Names: '|':UIView:0xa0b1530 )>",
    "<NSLayoutConstraint:0xa08c9f0 H:|-(0)-[UIButton:0xa0c11e0]   (Names: '|':UIView:0xa0b1530 )>"
)

Will attempt to recover by breaking constraint
<NSLayoutConstraint:0xa08c9f0 H:|-(0)-[UIButton:0xa0c11e0]   (Names: '|':UIView:0xa0b1530 )>
```

All Output

Invalid

Errors 〉 Errors 〉 ViewController.m 〉 -viewDidLoad

```
#import "ViewController.h"

@implementation ViewController

- (void)viewDidLoad
{
    [super viewDidLoad];

    NSDictionary *viewsDictionary = NSDictionaryOfVariableBindings(btnOne, btnTwo);
    for (NSString *format in @[@"$#*(&$*(#$&(*#$&(*$#^$&#($^&*#$^$*#&^$*#^$*("]){
        [self.view addConstraints:
         [NSLayoutConstraint constraintsWithVisualFormat:format
          options:0 metrics:0 views:viewsDictionary]];
    }
}

@end
```

Errors 〉 Thread 1 〉 13 main

```
2013-07-03 15:02:48.923 Errors[1128:a0b] Cannot find executable for CFBundle 0x99ad060 </Applications/Xcode5-DP2.app/Contents/
Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator7.0.sdk/System/Library/AccessibilityBundles/
CertUIFramework.axbundle> (not loaded)
2013-07-03 15:02:48.988 Errors[1128:a0b] *** Terminating app due to uncaught exception 'NSInvalidArgumentException', reason: 'Unable
to parse constraint format:
Expected a view
$#*(&$*(#$&(*#$&(*$#^$&#($^&*#$^$*#&^$*#^$*(
^'
*** First throw call stack:
(0x16ad9b8 0x142e8b6 0x11e717b 0x1094e0b 0x109409e 0x1093c5c 0x1093bee 0x6b32 0x3339fc 0x333c98 0x268f99 0x269334 0x26959e 0xf94211b
0x273697 0x229824 0x22ab5e 0x240a6c 0x240fd9 0x22c7d5 0x35a4906 0x35a4411 0x16293e5 0x162911b 0x1653b30 0x165310d 0x1652f3b 0x22a2b1
0x22c4eb 0x706d 0x1d0d725)
libc++abi.dylib: terminating with uncaught exception of type NSException
(lldb)
```

All Output

Ambiguous

view.hasAmbiguousLayout
view.exerciseAmbiguityInLayout

```
for (UIView *view in self.subviews) {
    if ([view hasAmbiguousLayout]){
        NSLog(@"<%@:0x%0x>", view.description, (int)self);
    }
}
```

Use it on UIViewController.loadView.

instrinsicContentSize

Suggested size for the view.

```
- (CGSize) intrinsicContentSize {
    return mySize;
}

[self invalidateIntrinsicContentSize];


UIImage *img = UIImage imageNamed:@"Icon.png"];
UIImageView *iv = [[UIImageView alloc] initWithImage:img];
NSLog(@"%@", NSStringFromCGSize(iv.intrinsicContentSize));
```

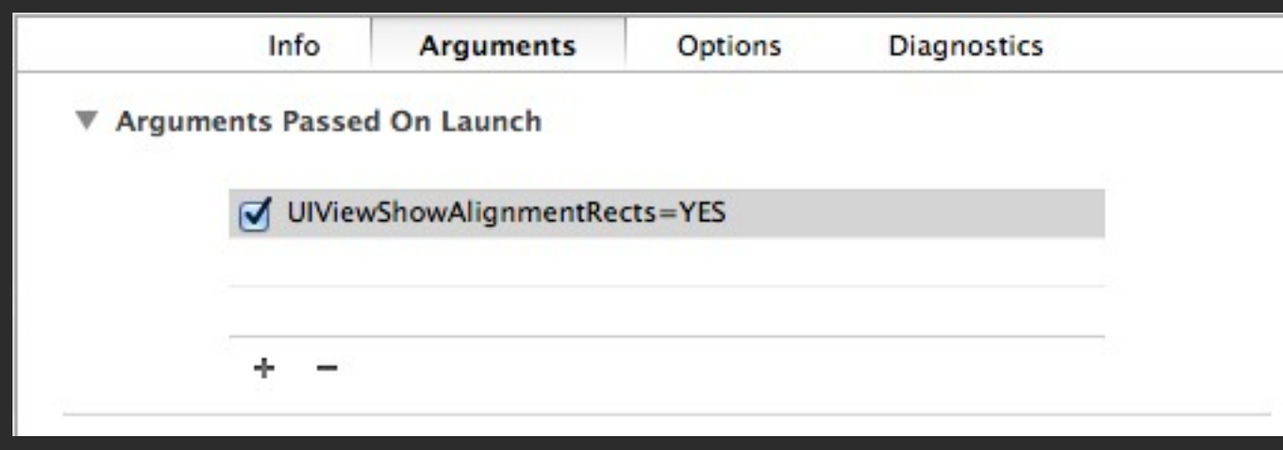
Alignment rectangle

UIView

```
- (UIEdgeInsets)alignmentRectInsets

- (CGRect)frameForAlignmentRect:(CGRect)alignmentRect
- (CGRect)alignmentRectForFrame:(CGRect)frame
```

Show rect lines:



The alignment rectangle is the area where constraints are applied to.
It is equal to the frame plus the UIEdgeInsets returned by alignmentRectInsets. The UIEdgeInsets is 0,0,0,0 by default.

If we add an element to a view (eg: a subview with a badge number), and we want the view to report the an area that covers both the view and the new element, we have to override alignmentRectInsets.

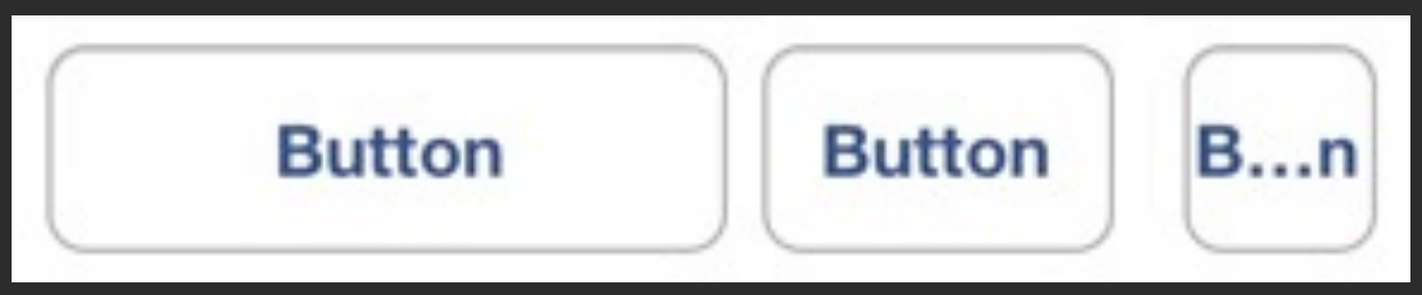
If you need a complex calculation between alignment rect and frame, use frameForAlignmentRect: and alignmentRectForFrame:. They should be mathematically inverse.

Set the UIViewShowAlignmentRects using NSUserDefaults, or using Xcode.

Autolayout doesn't support the mix of a transform that doesn't preserve the rectangle, and an alignment rect with non zero UIEdgeInsets.

Hug & compress

Hugging resists stretching
Compression resists shrinking




```
UILayoutConstraintAxis axis = UILayoutConstraintAxisHorizontal;
UILayoutPriority p = UILayoutPriorityDefaultHigh;
[button setContentCompressionResistancePriority:p forAxis:axis];
[button setContentHuggingPriority:p forAxis:axis];
```

This sets the behavior of the intrinsic size when there are other forces.
High values for both result in a fixed size.

Priority is a number 0-1000, but you can use a enum.

The priority of compression and hugging override the priorities of the user layout constraints.

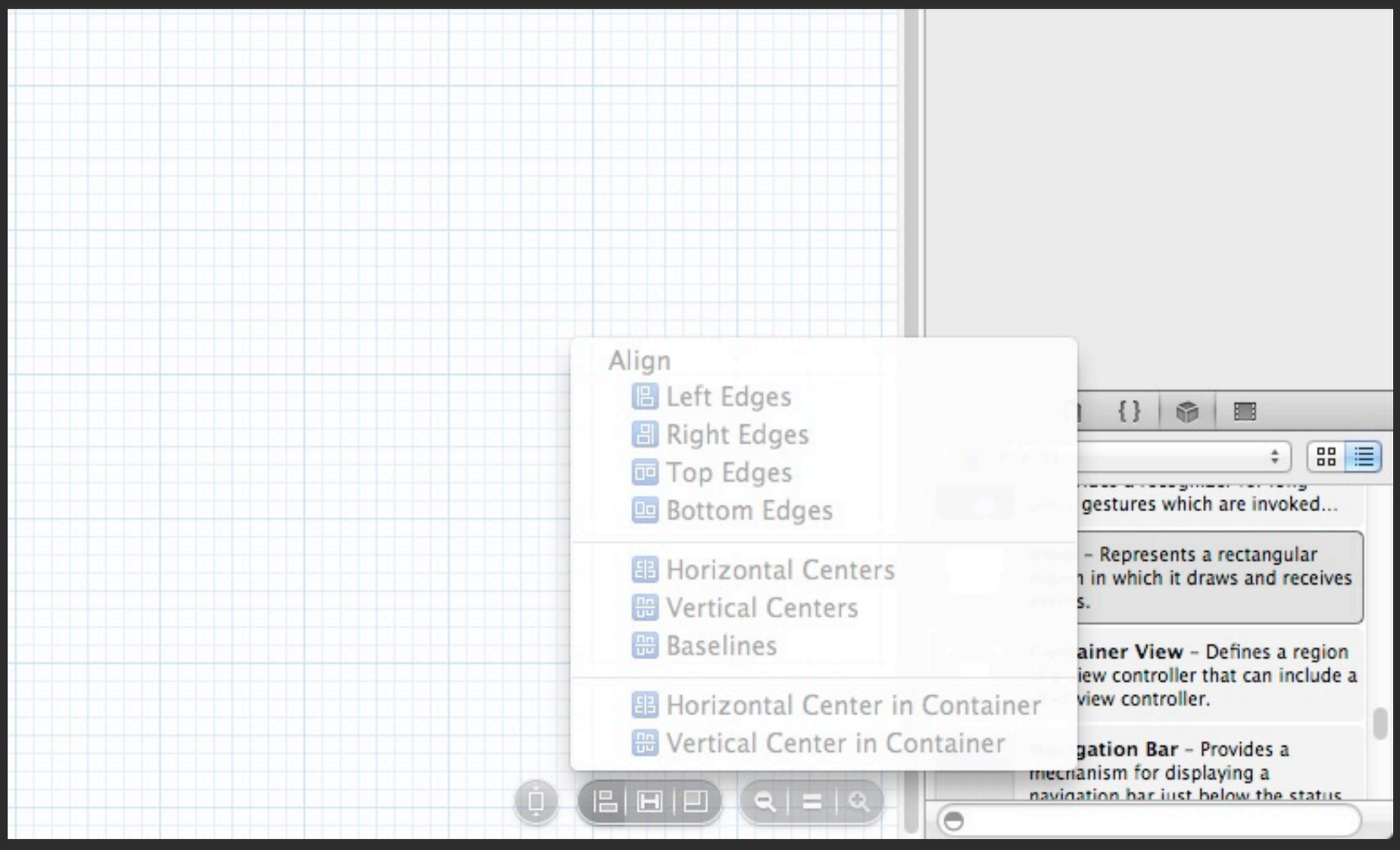
UIView properties

Interface Builder > VFL > API

Animation

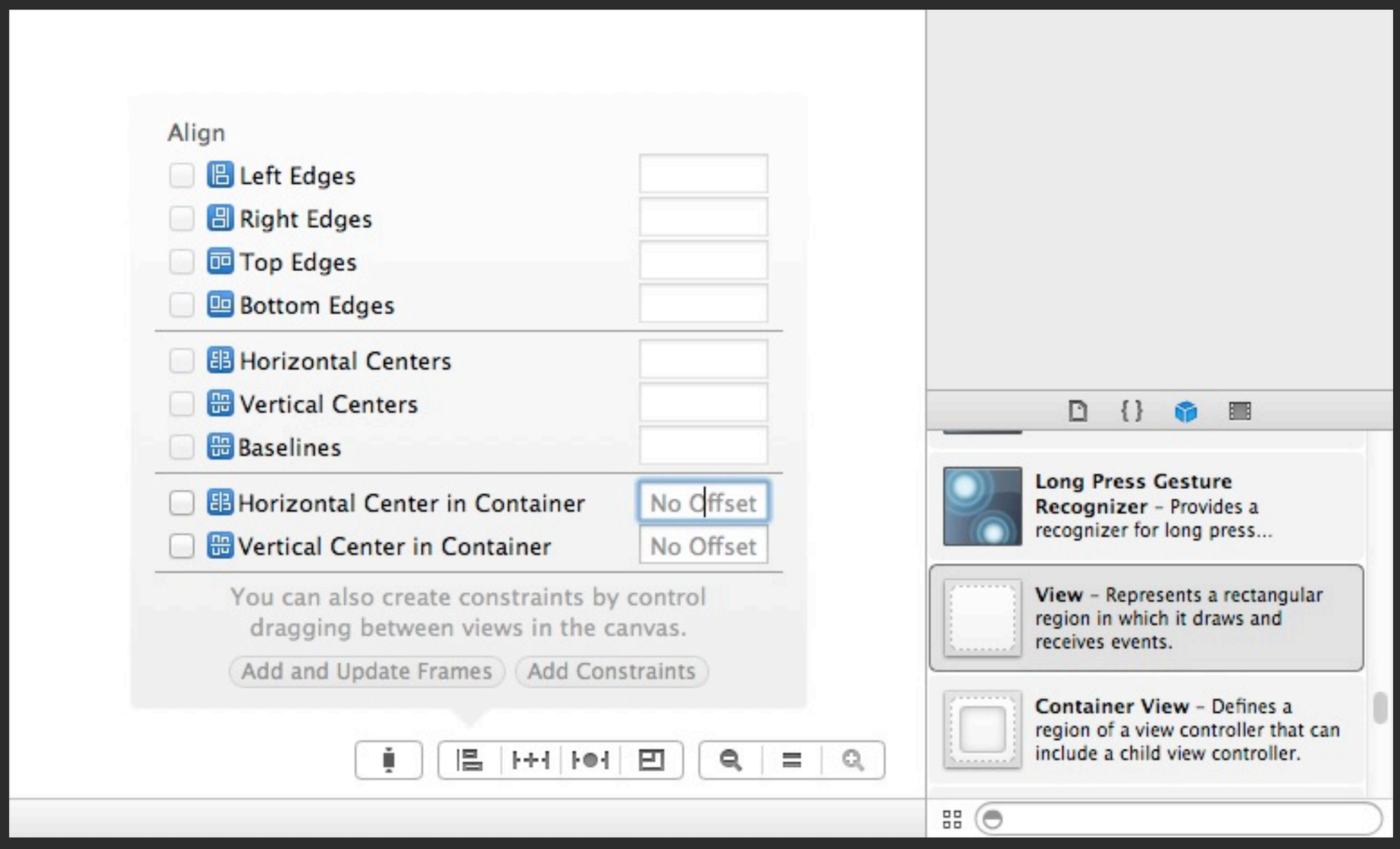
Interface Builder

Interface Builder



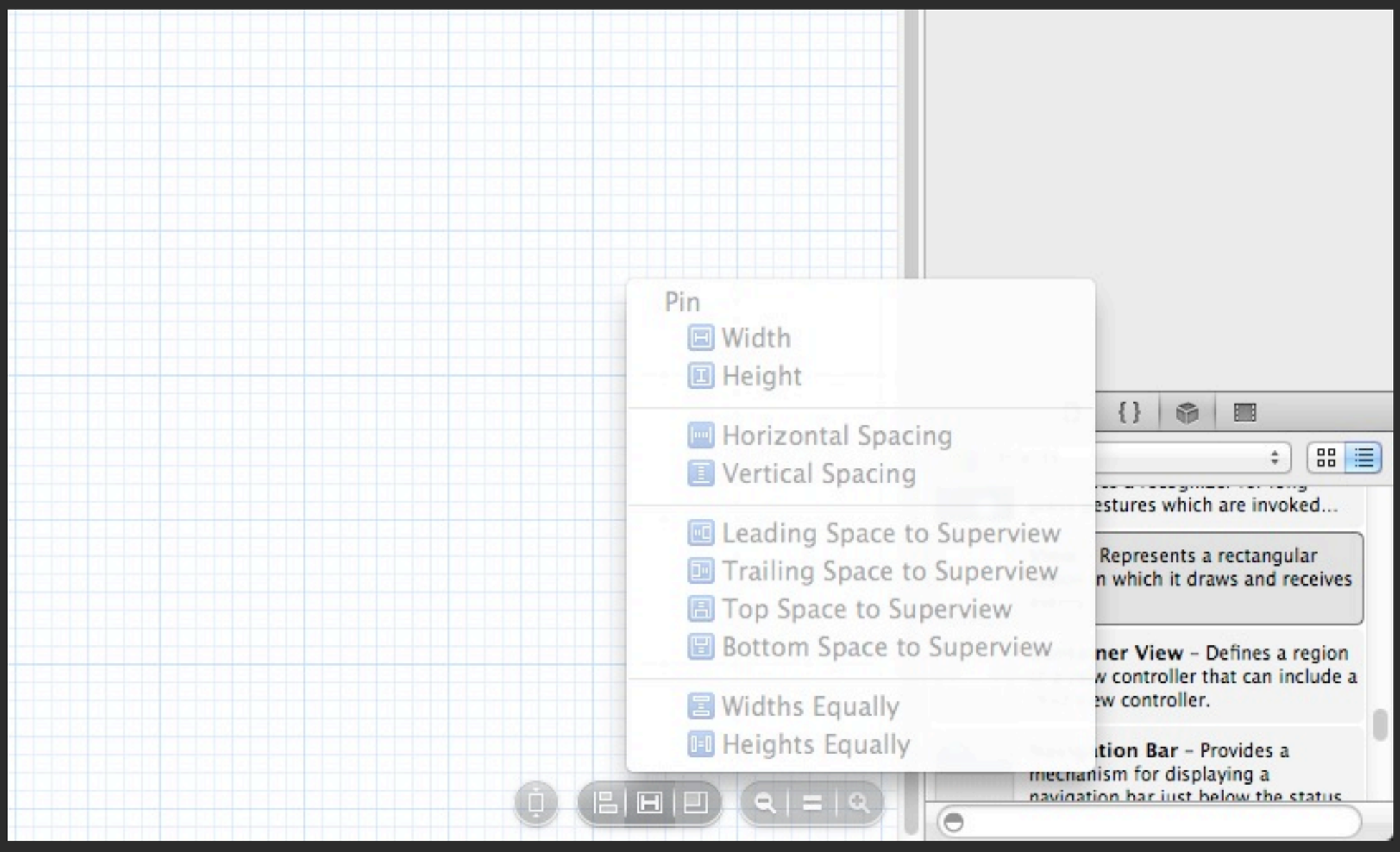
ALIGN

Interface Builder



ALIGN

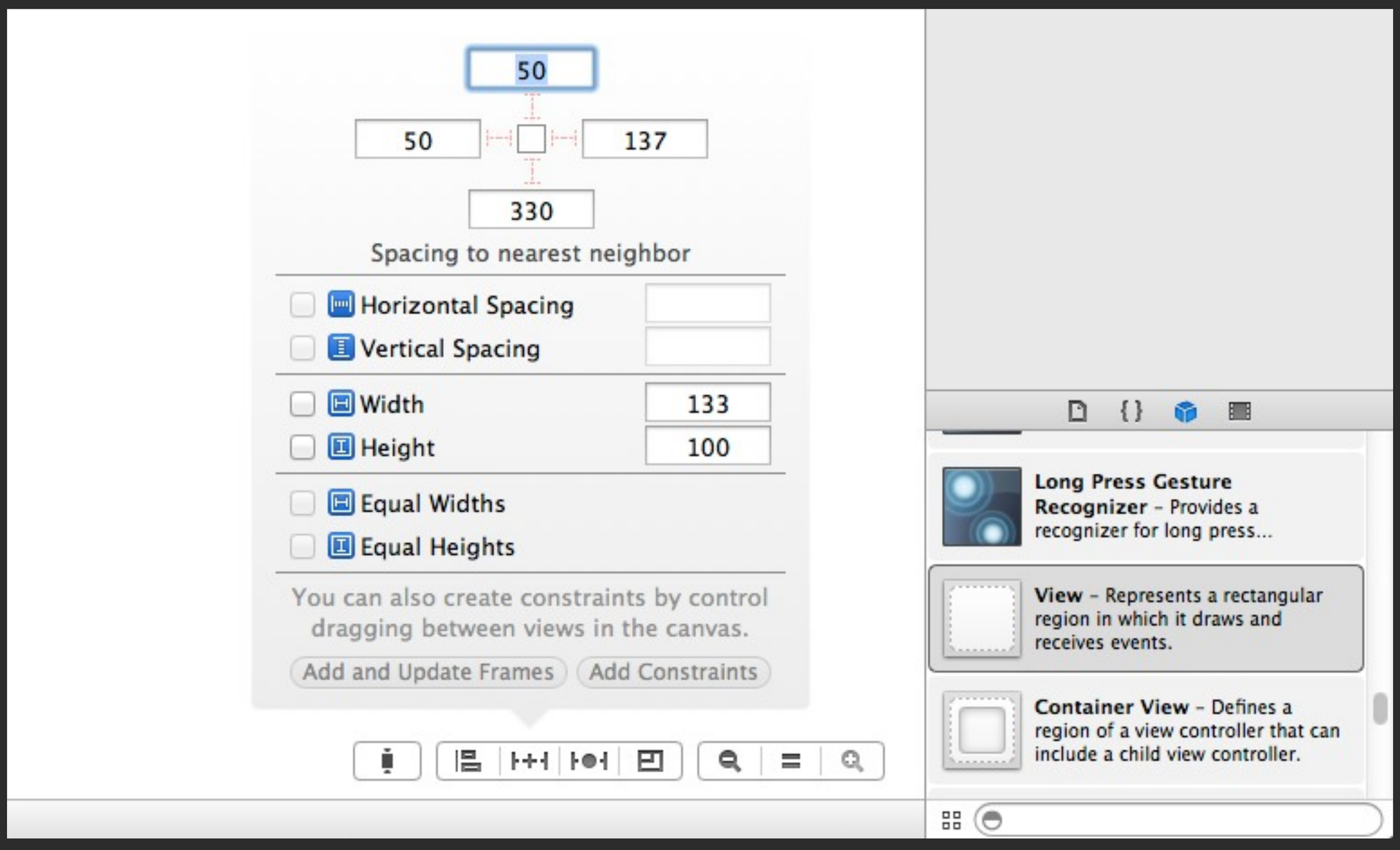
Interface Builder



PIN

Xcode 4, Xcode 5, and a shortcut: When pressing Ctrl + clicking an element, dragging to another view, and releasing, we get an options menu.

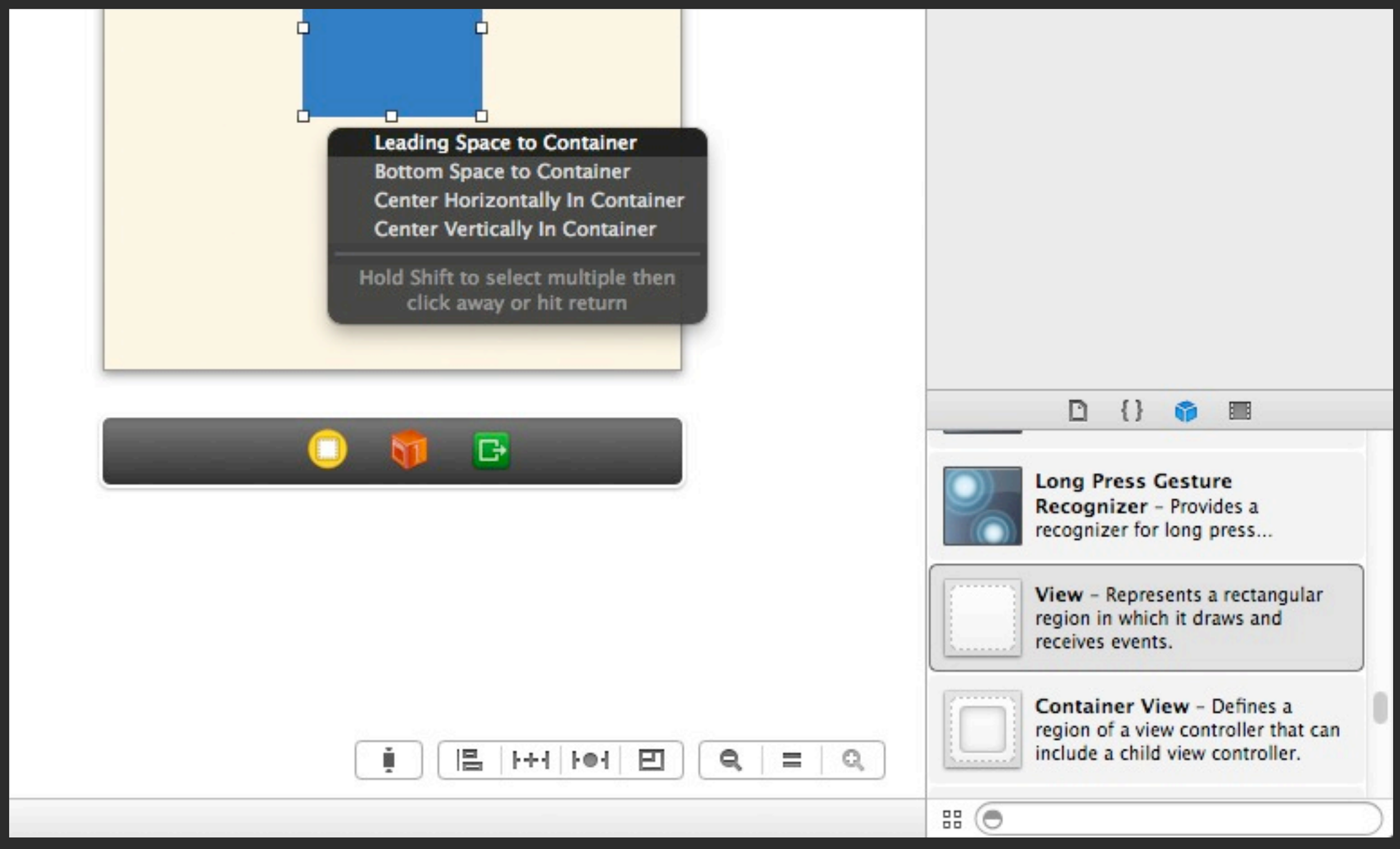
Interface Builder



PIN

Xcode 4, Xcode 5, and a shortcut: When pressing Ctrl + clicking an element, dragging to another view, and releasing, we get an options menu.

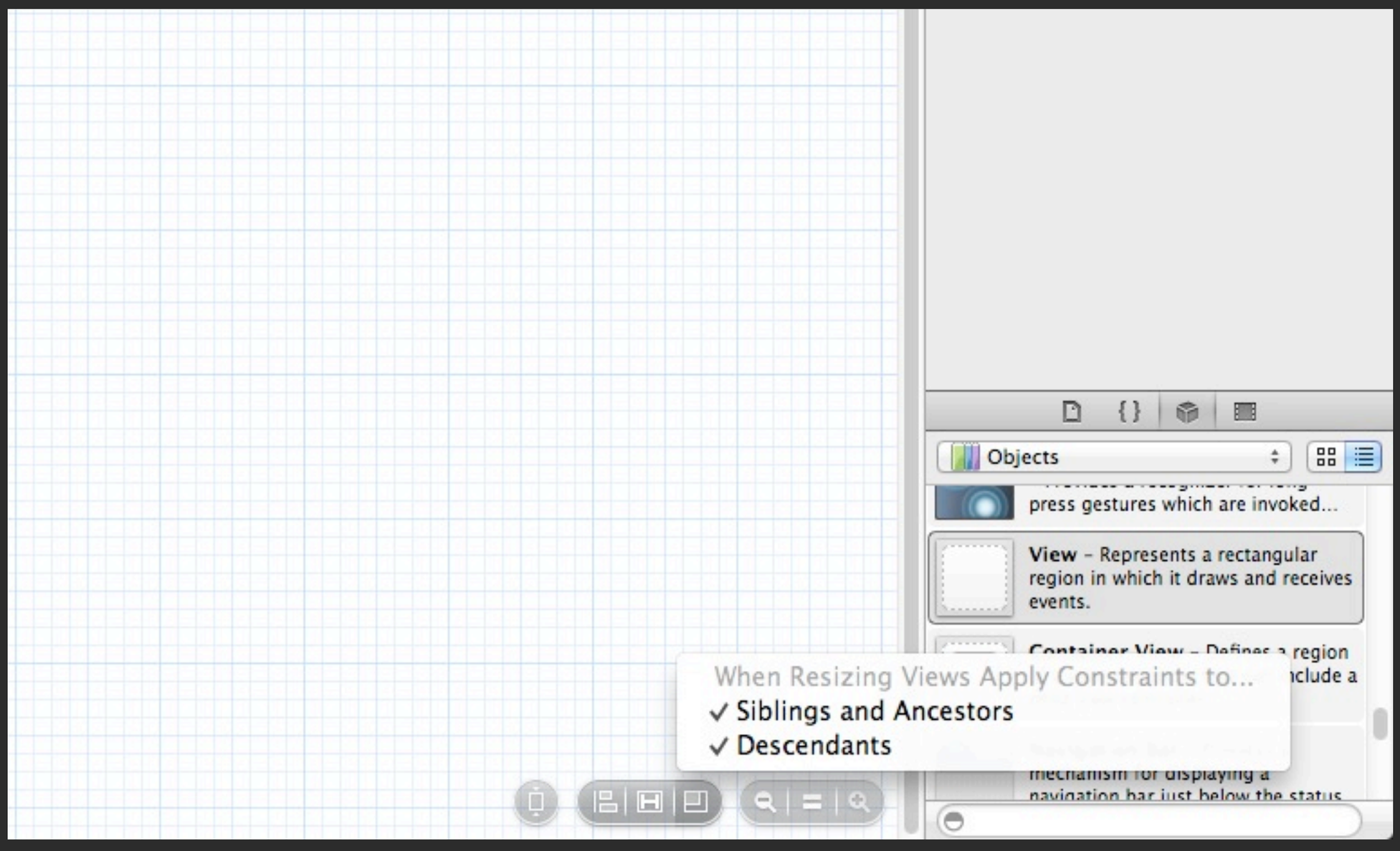
Interface Builder



PIN

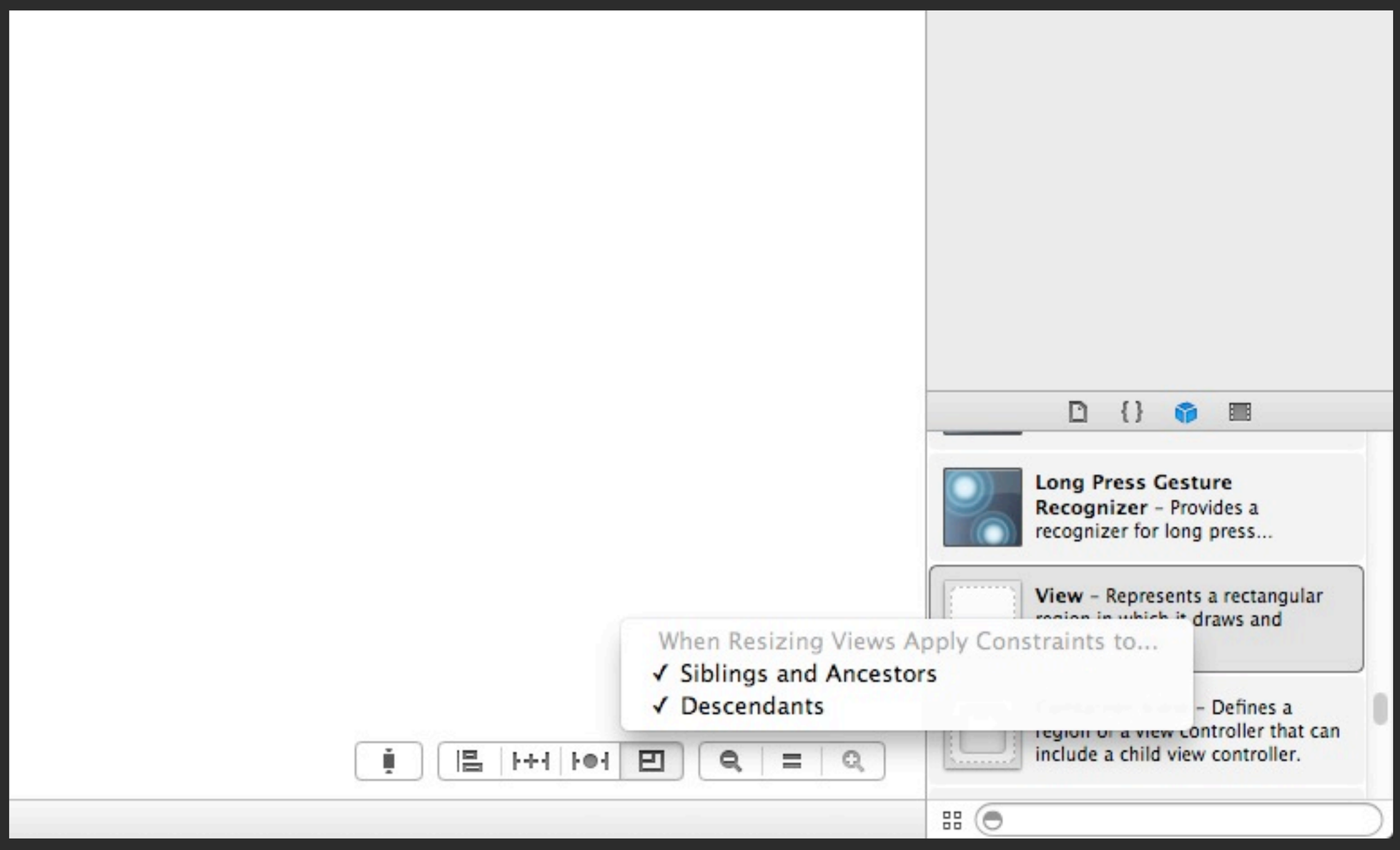
Xcode 4, Xcode 5, and a shortcut: When pressing Ctrl + clicking an element, dragging to another view, and releasing, we get an options menu.

Interface Builder



RESIZING

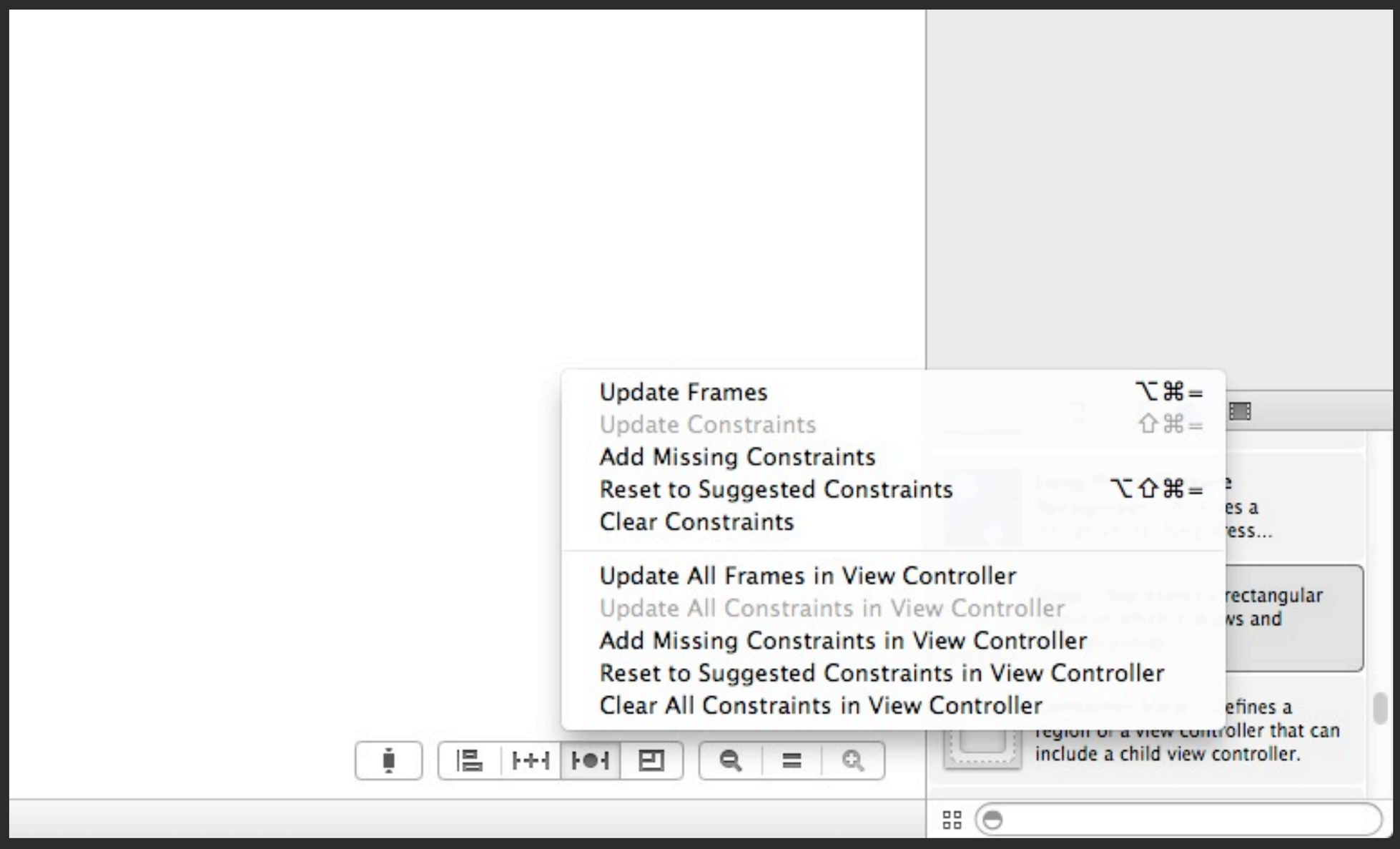
Interface Builder



RESIZING

Interface Builder

Interface Builder



RESOLVE

Interface Builder

IB > VFL > API
Constraints colors
IB can't create ambiguous layouts
Add a constraint before deleting another
Preserve intrinsic size
Don't optimize until everything is in place

UIView properties
Interface Builder > **VFL** > API
Animation

Visual Format Language

VFL

```
[NSLayoutConstraint
    constraintsWithVisualFormat:@"H:|-[buttonA]-|"
    options:0
    metrics:nil
    views:@{ @"buttonA" : buttonA }];
```

VFL

```
[NSLayoutConstraint
    constraintsWithVisualFormat:@"H:|-[buttonA]-|"
    options:0
    metrics:nil
    views:@{ @"buttonA" : buttonA }];
```

Parenthesis are used for relations that are not simply a number.

There is no compile check. If you mess up you terminate the application.

VFL

```
[NSLayoutConstraint
    constraintsWithVisualFormat:@"H:|-[buttonA]-|"
    options:0
    metrics:nil
    views:@{ @"buttonA" : buttonA }];
```

H V [view] | - @ ()

Parenthesis are used for relations that are not simply a number.

There is no compile check. If you mess up you terminate the application.

VFL

```
[NSLayoutConstraint
    constraintsWithVisualFormat:@"H:|-[buttonA]-|"
    options:0
    metrics:nil
    views:@{ @"buttonA" : buttonA }];
```

Alignment of the elements in the layout, and the direction the ASCII is read.

The alignment should be perpendicular to the axis you are specifying (H or V), otherwise you are undoing your work.

VFL

```
[NSLayoutConstraint
    constraintsWithVisualFormat:@"H:|-[buttonA]-|"
    options:0
    metrics:nil
    views:@{ @"buttonA" : buttonA }];
```

```
NSLayoutFormatAlignAllLeft
NSLayoutFormatAlignAllRight
NSLayoutFormatAlignAllTop
NSLayoutFormatAlignAllBottom
NSLayoutFormatAlignAllLeading
NSLayoutFormatAlignAllTrailing
NSLayoutFormatAlignAllCenterX
NSLayoutFormatAlignAllCenterY
NSLayoutFormatAlignAllBaseline

NSLayoutFormatAlignmentMask
```

```
NSLayoutFormatDirectionLeadingToTrailing
NSLayoutFormatDirectionLeftToRight
NSLayoutFormatDirectionRightToLeft

NSLayoutFormatDirectionMask
```

NSLayoutFormatOptions

Alignment of the elements in the layout, and the direction the ASCII is read.

The alignment should be perpendicular to the axis you are specifying (H or V), otherwise you are undoing your work.

VFL

```
[NSLayoutConstraint
    constraintsWithVisualFormat:@"H:|-[buttonA]-distance-|"
    options:0
    metrics: @{ @"distance": @50 }
    views:@{ @"buttonA" : buttonA }];
```

Pass constants.

VFL

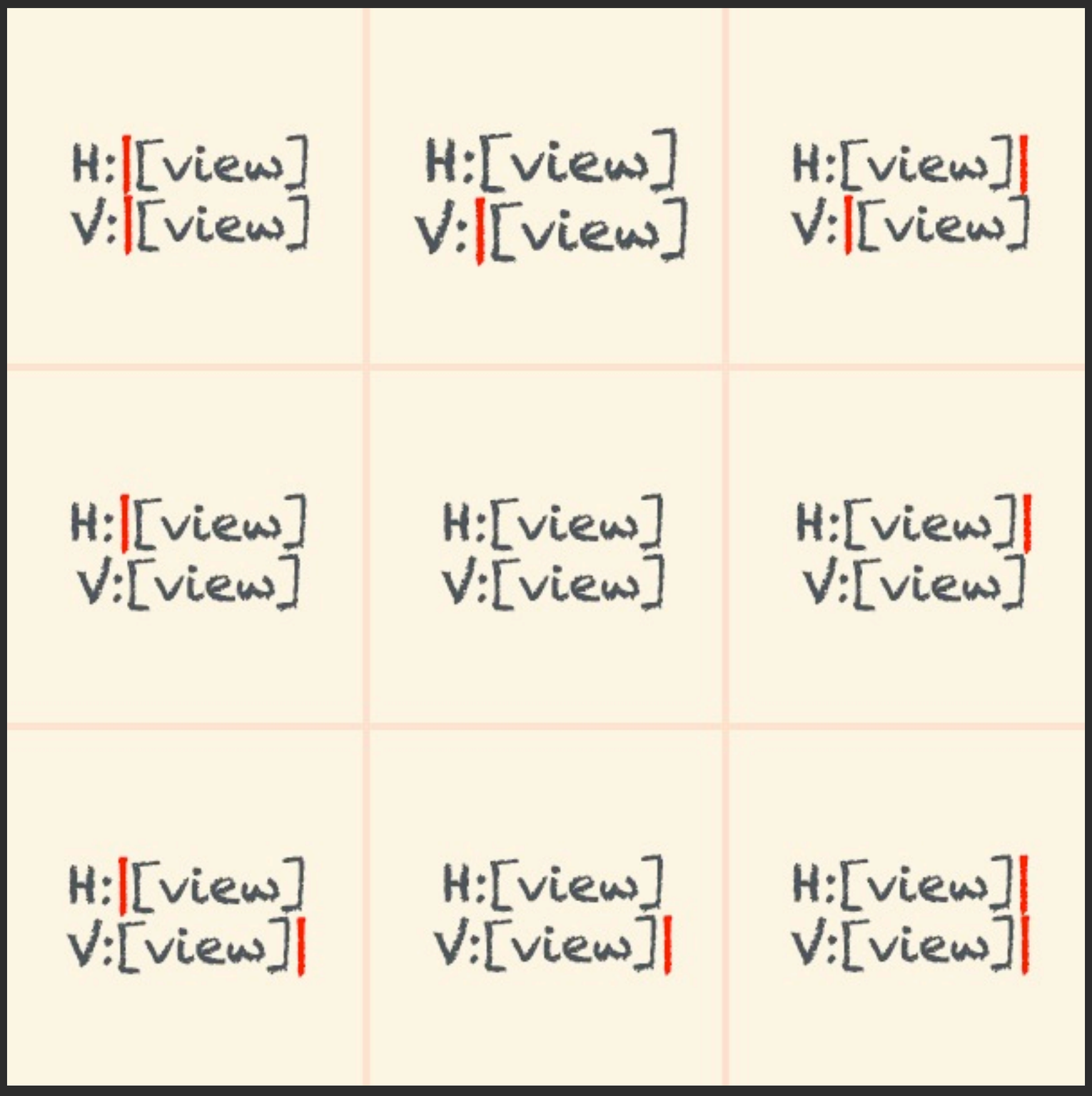
```
[NSLayoutConstraint
    constraintsWithVisualFormat:@"H:|-[buttonA]-distance-|"
    options:0
    metrics: @{ @"distance": @50 }
    views:@{ @"buttonA" : buttonA }];
```

VFL

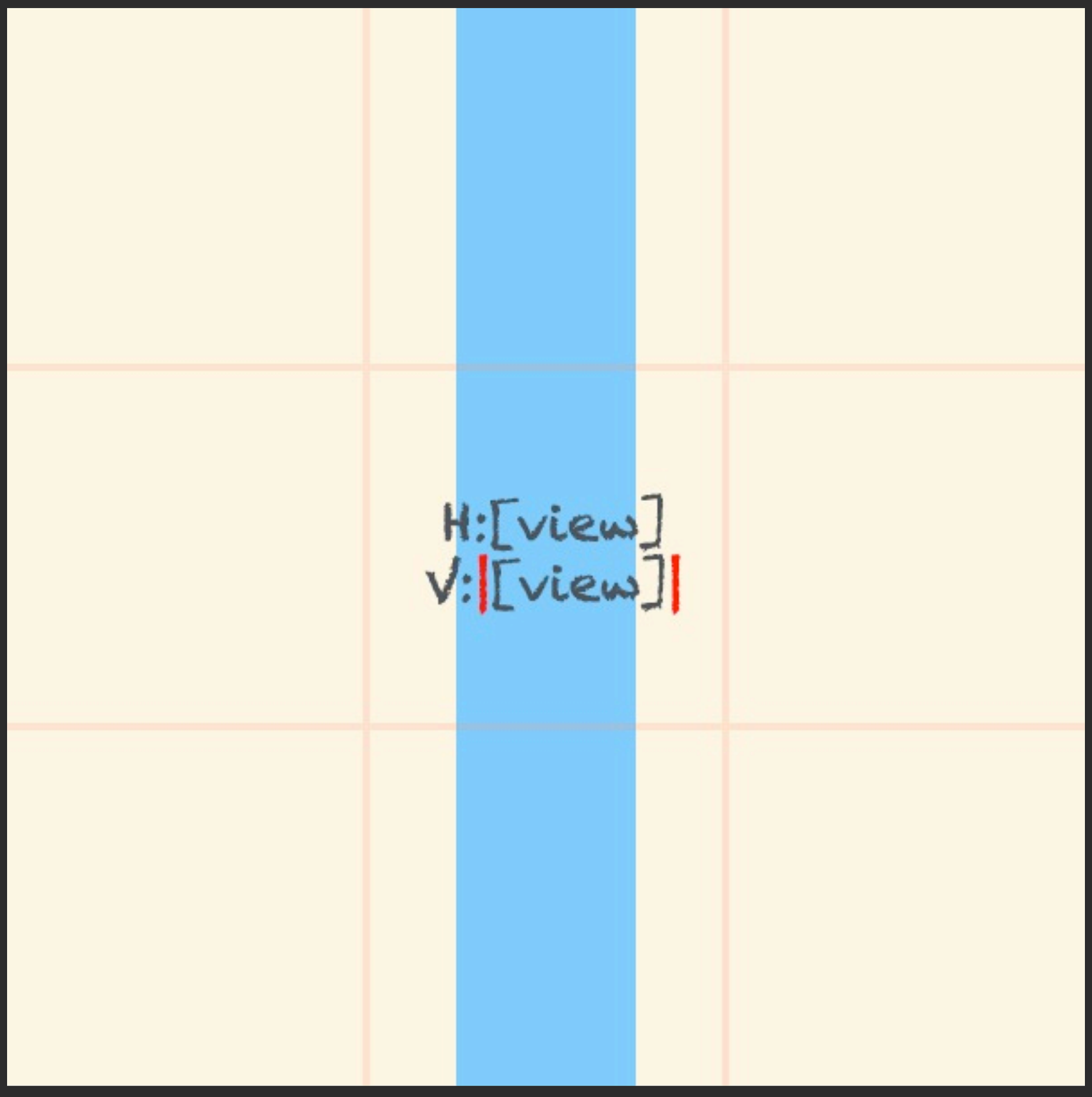
```
[NSLayoutConstraint
    constraintsWithVisualFormat:@"H:|-[buttonA]-distance-|"
    options:0
    metrics: @{ @"distance": @50 }
    views:NSDictionaryOfVariableBindings(buttonA)];
```

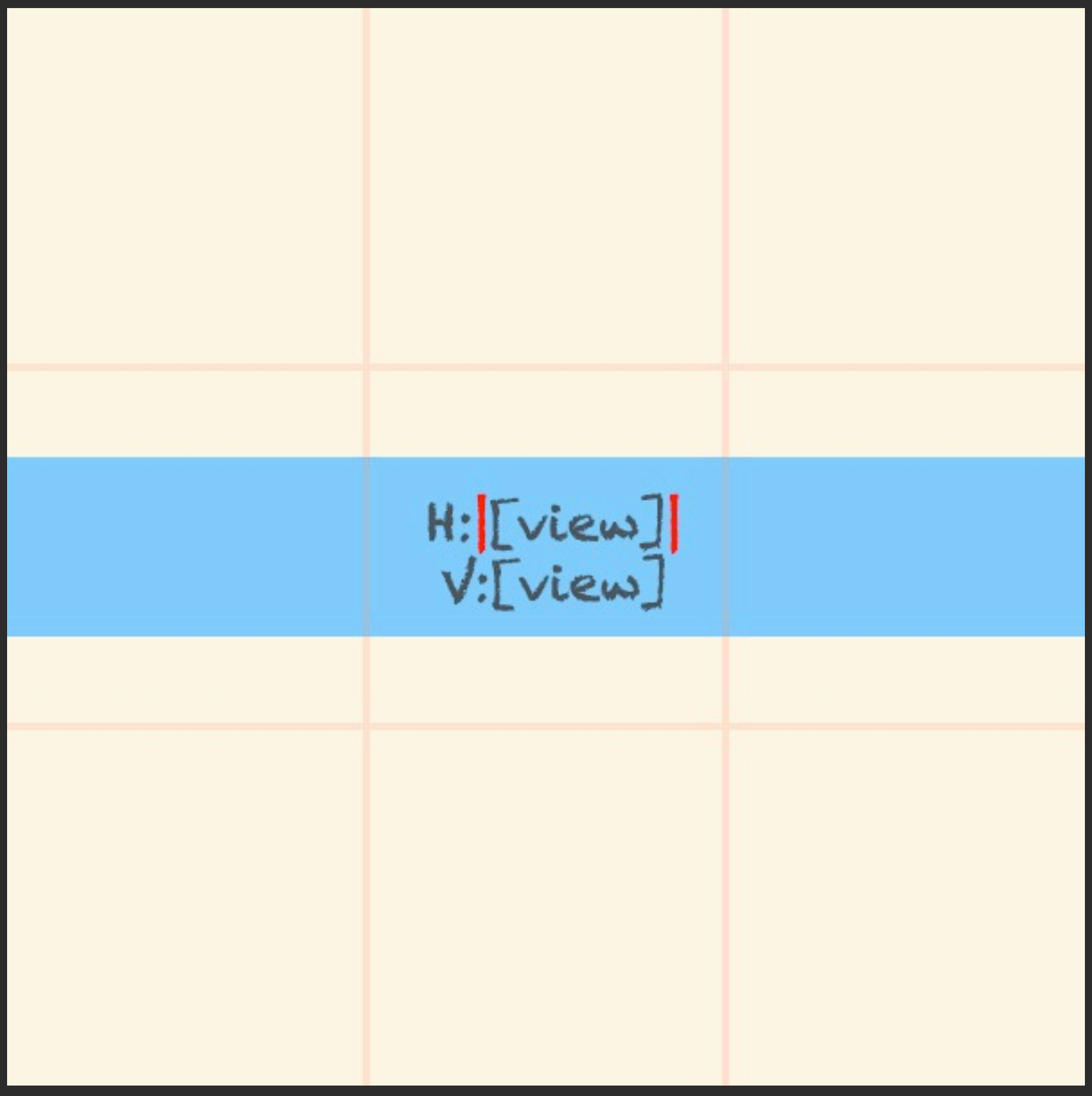
VFL

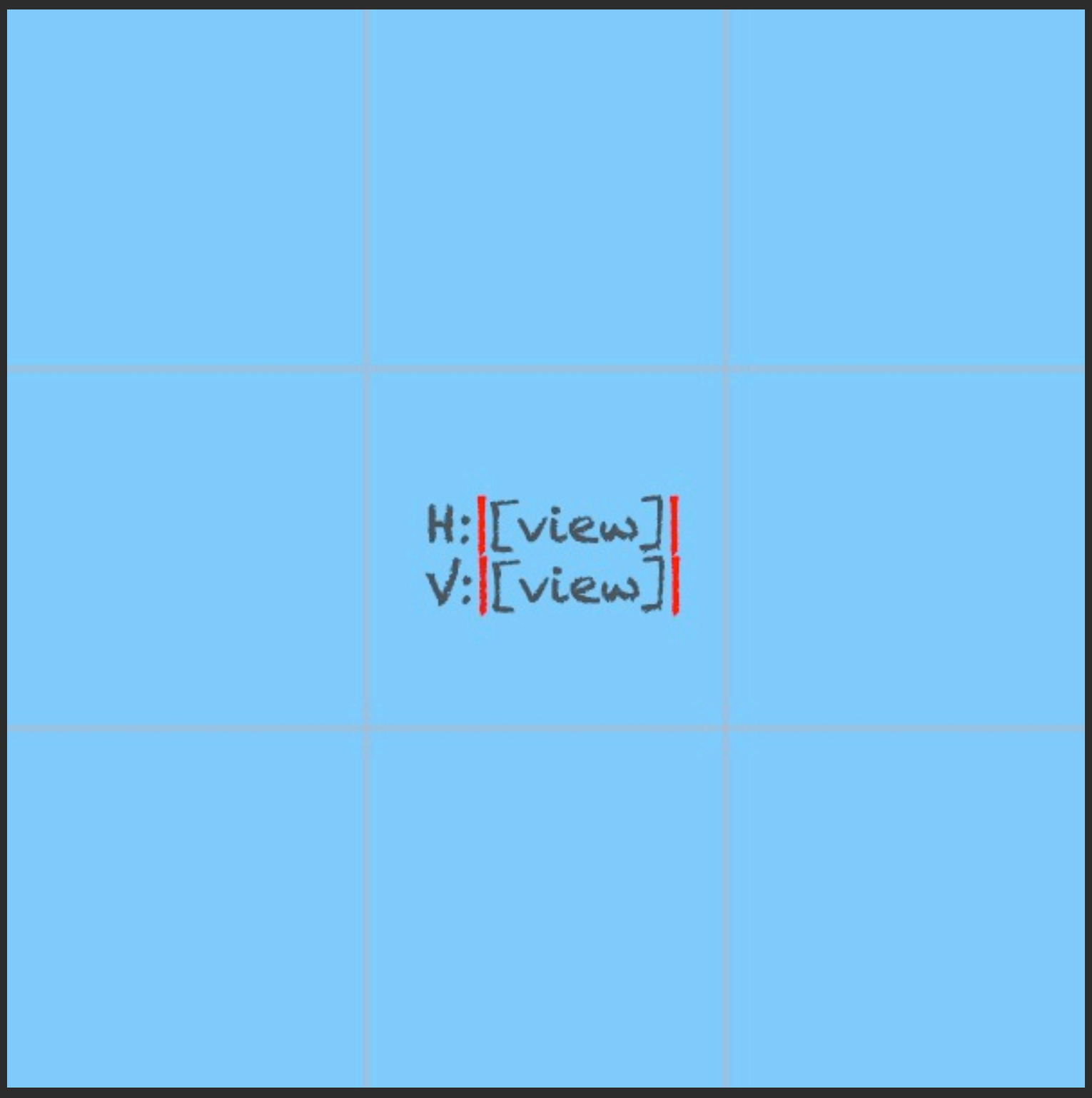
```
H V [view] | - @ ()
```

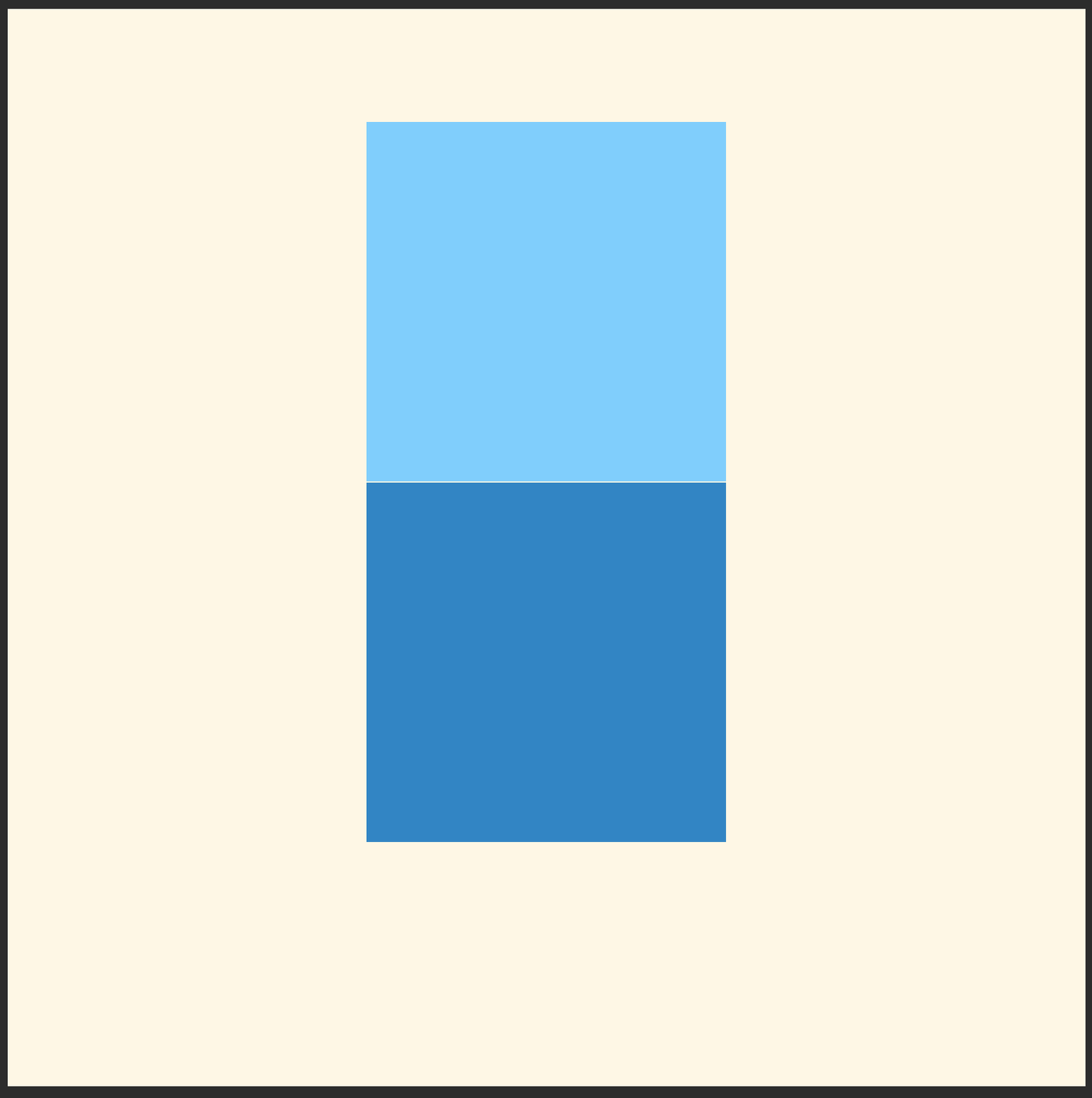

H:|[view]
V:|[view]
H:[view]
V:|[view]
H:[view]|
V:|[view]
H:|[view]
V:[view]
H:[view]
V:[view]
H:[view]|
V:[view]
H:|[view]
V:[view]|
H:[view]
V:[view]|
H:[view]|
V:[view]|

Alignment.

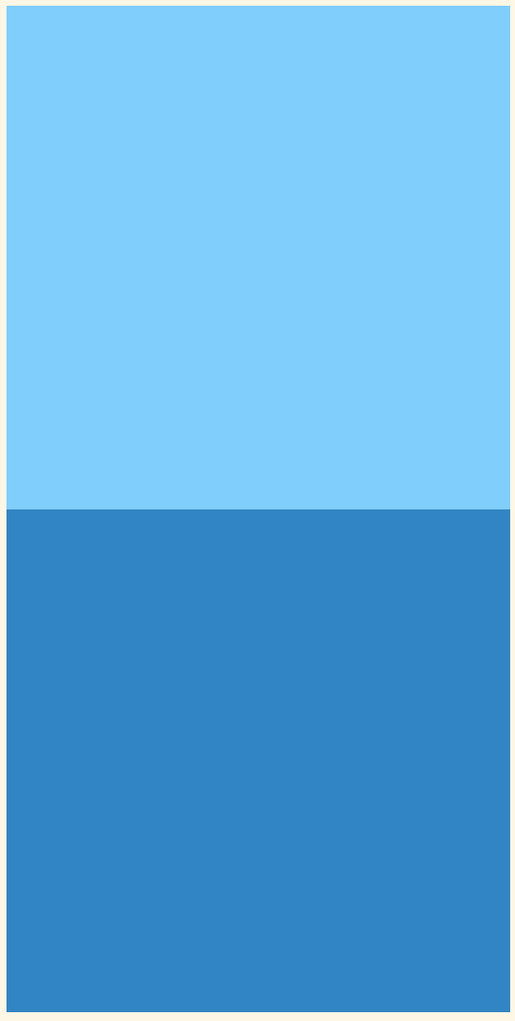

H:[view]
V:|[view]|


H:|[view]|
V:[view]

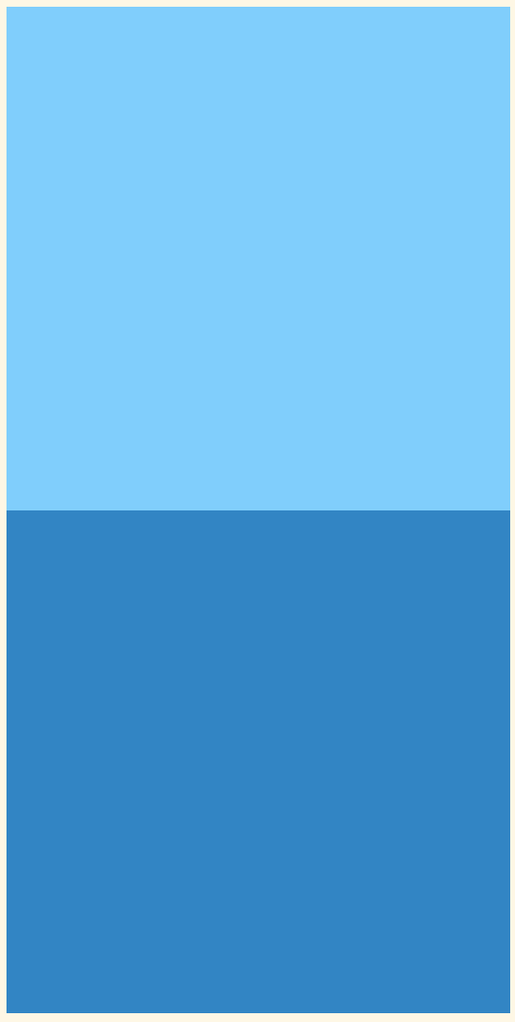

H:|[view]|
V:|[view]|



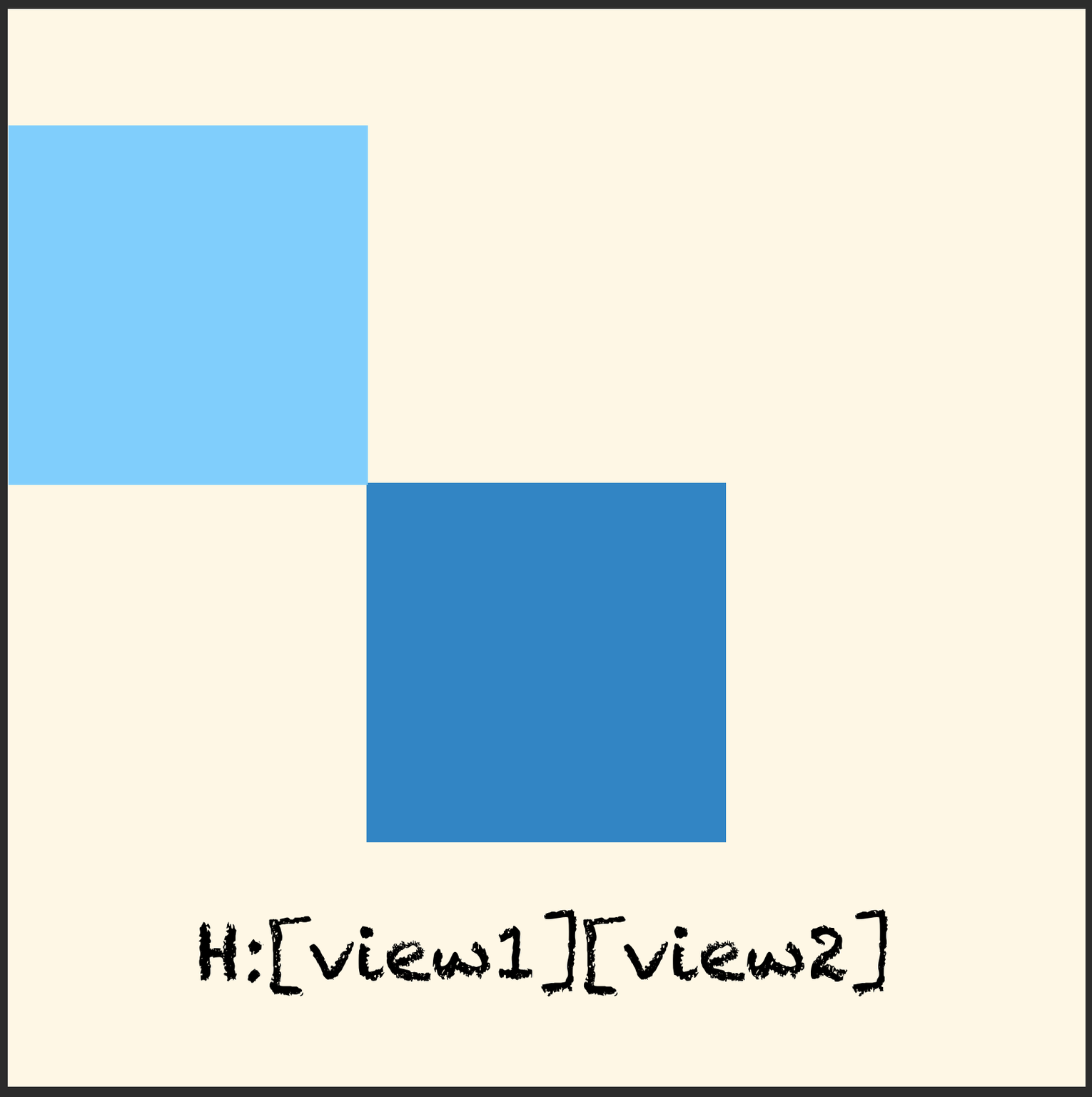
Two squares. Deal with it.

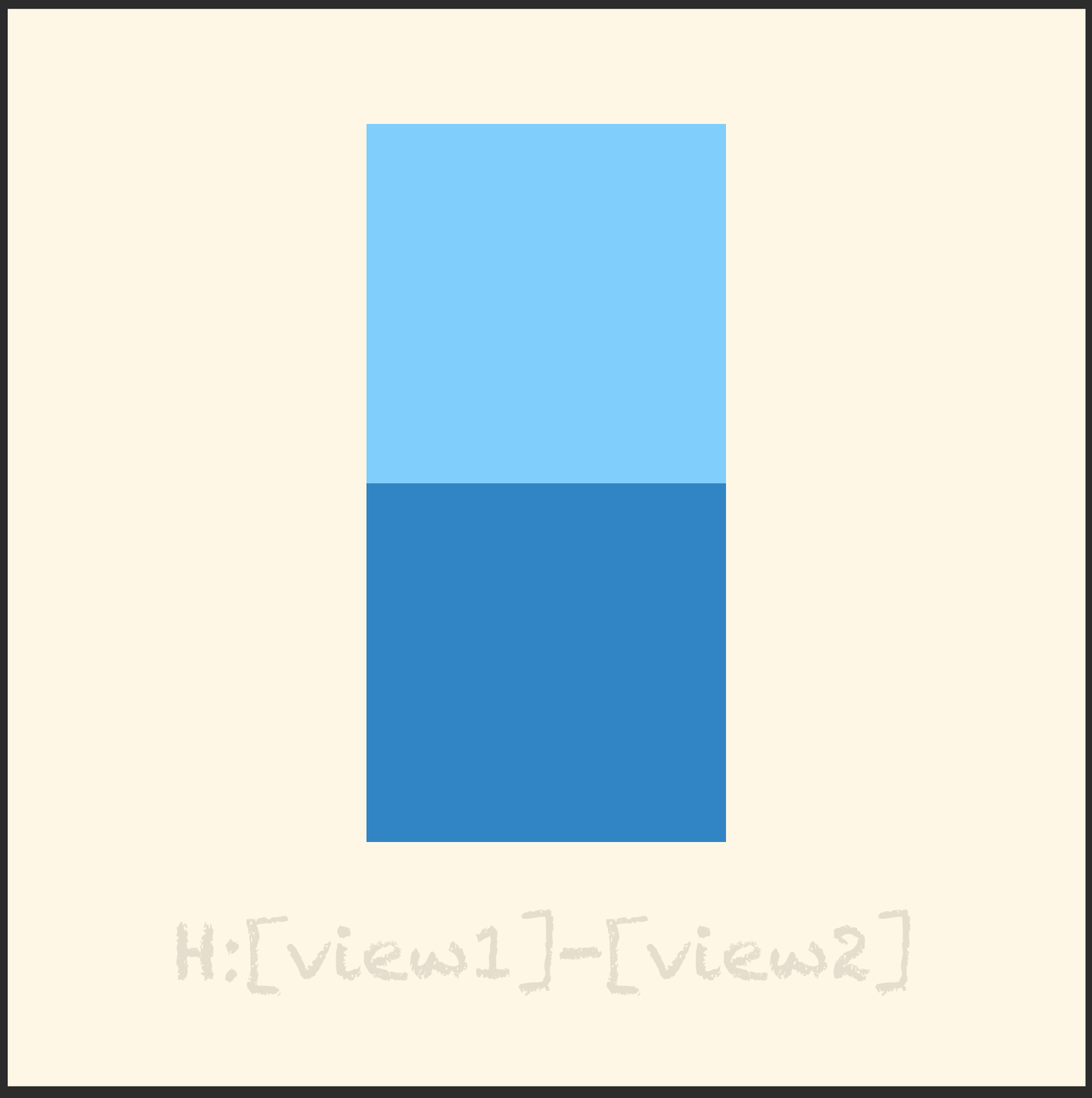


H:[view1][view2]

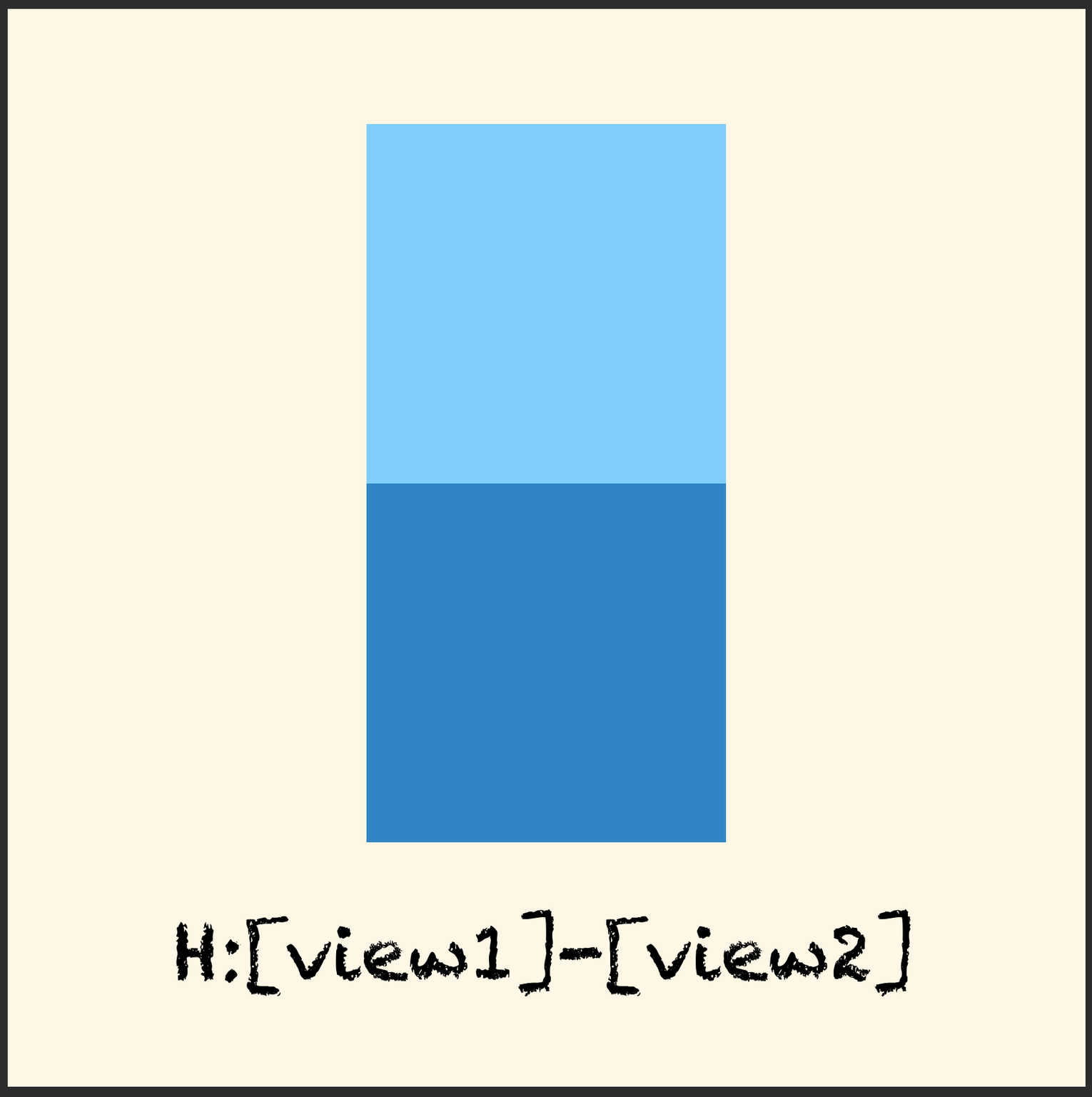


H:[view1][view2]

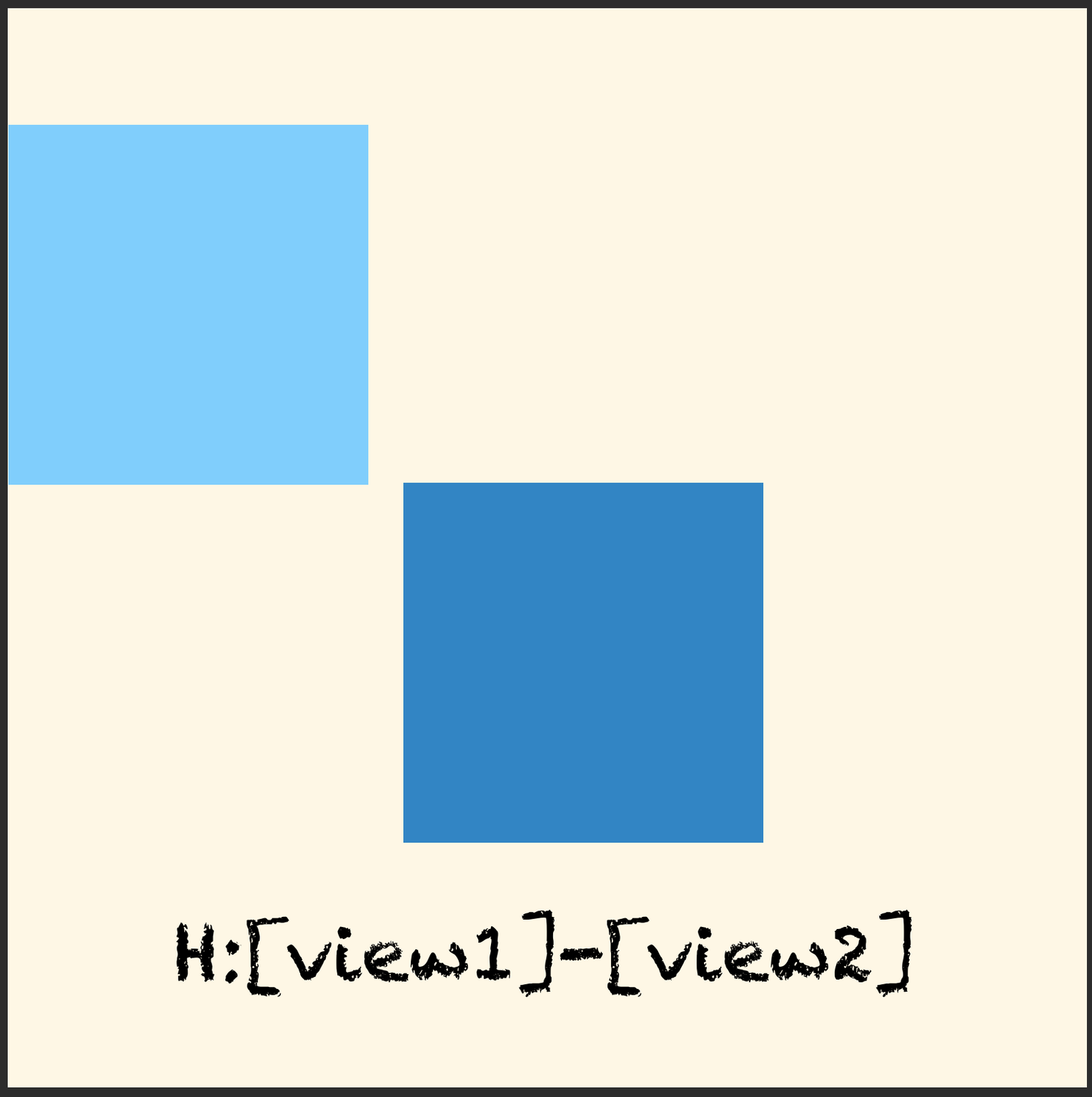

H:[view1][view2]



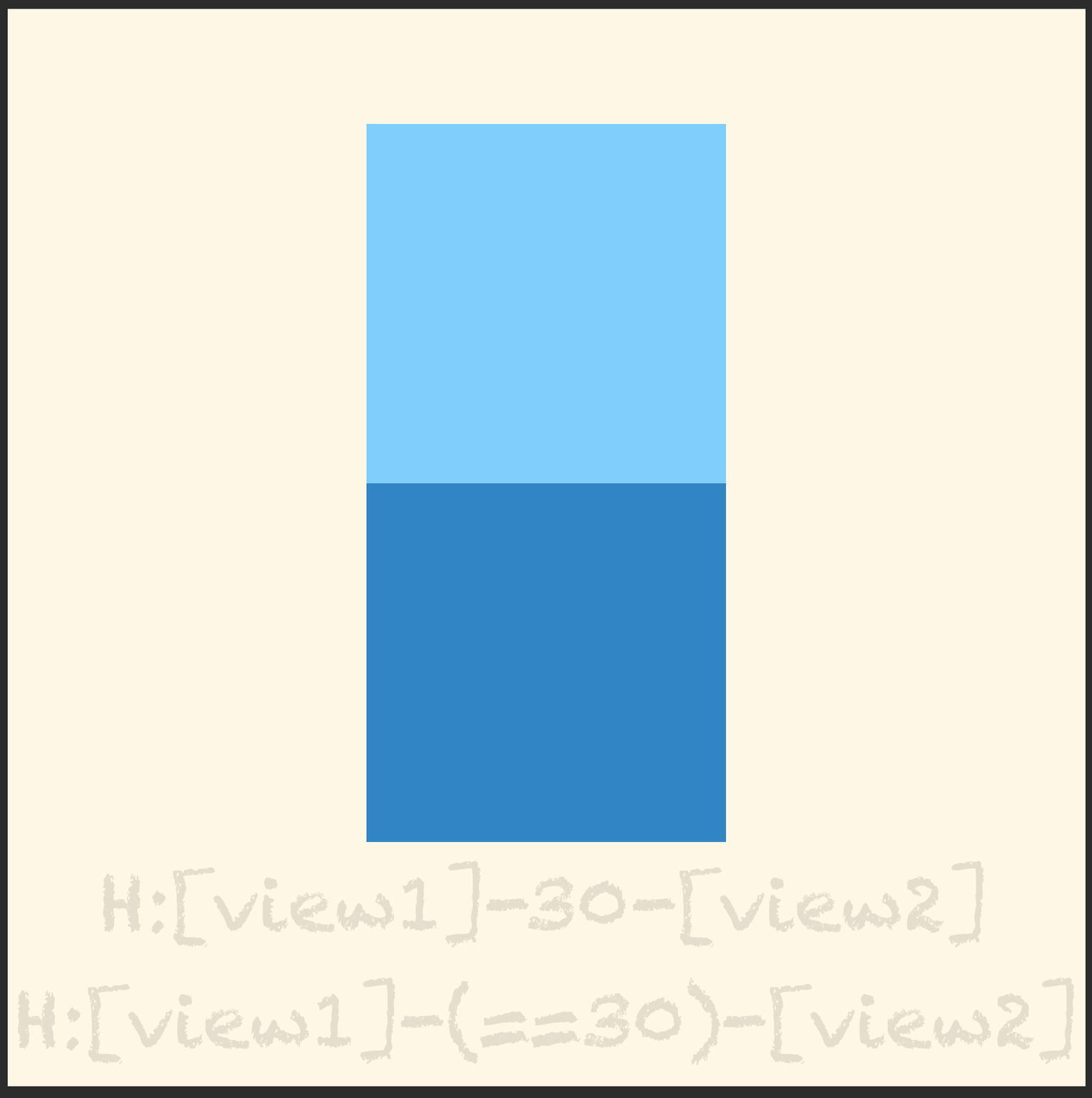
Two views with a standard separation (20 according to Apple's HIG).



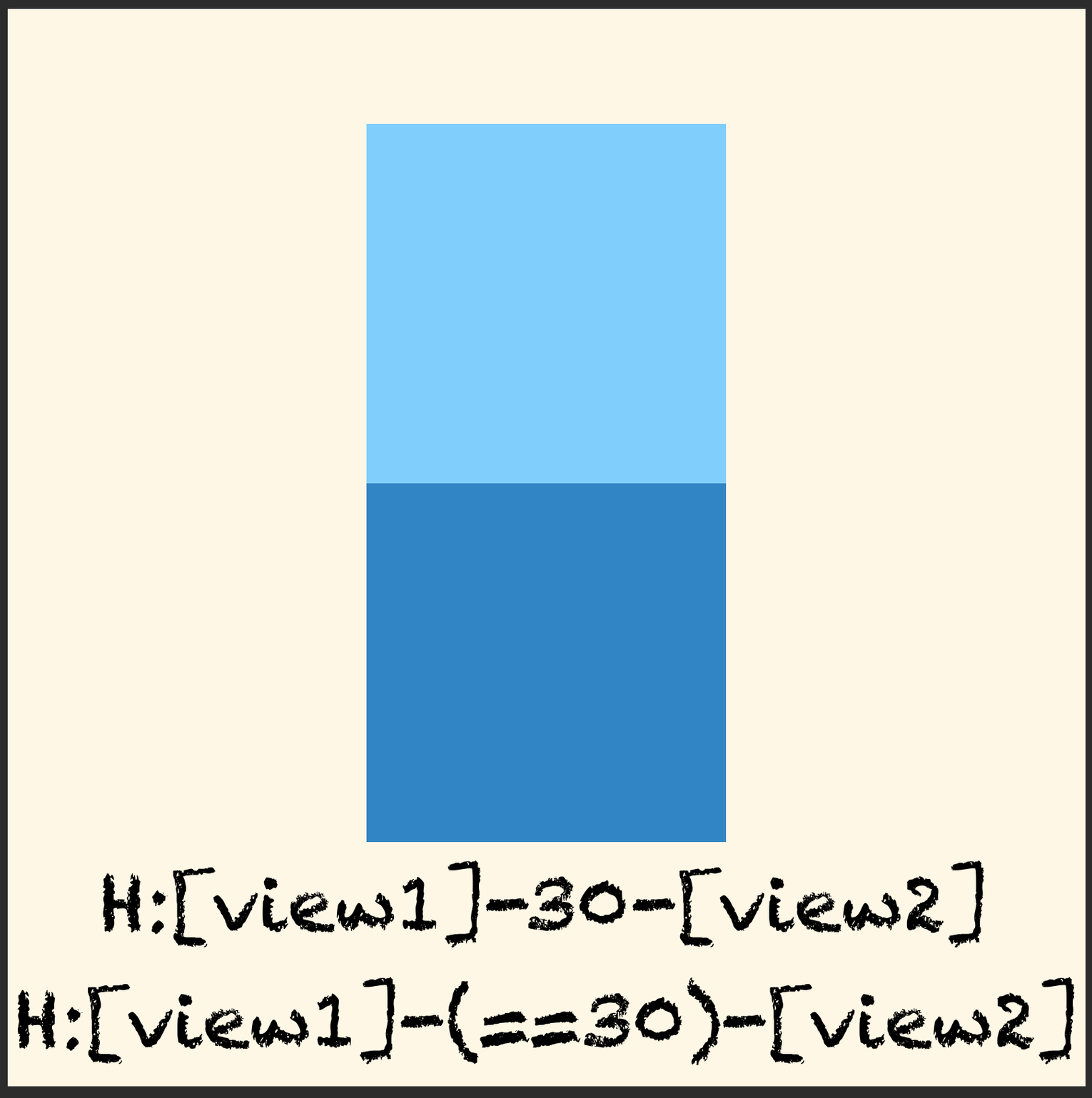
Two views with a standard separation (20 according to Apple's HIG).



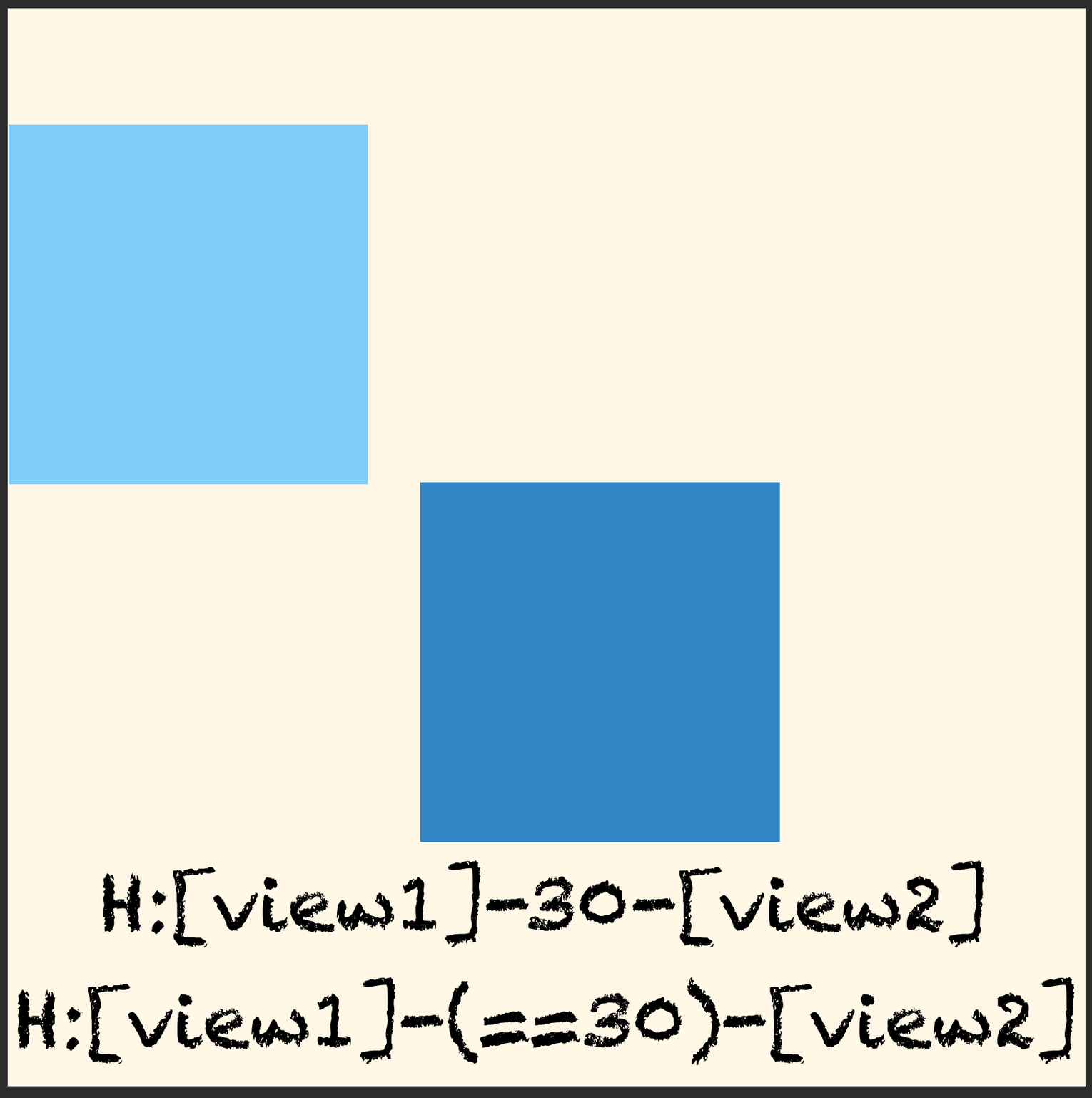
Two views with a standard separation (20 according to Apple's HIG).



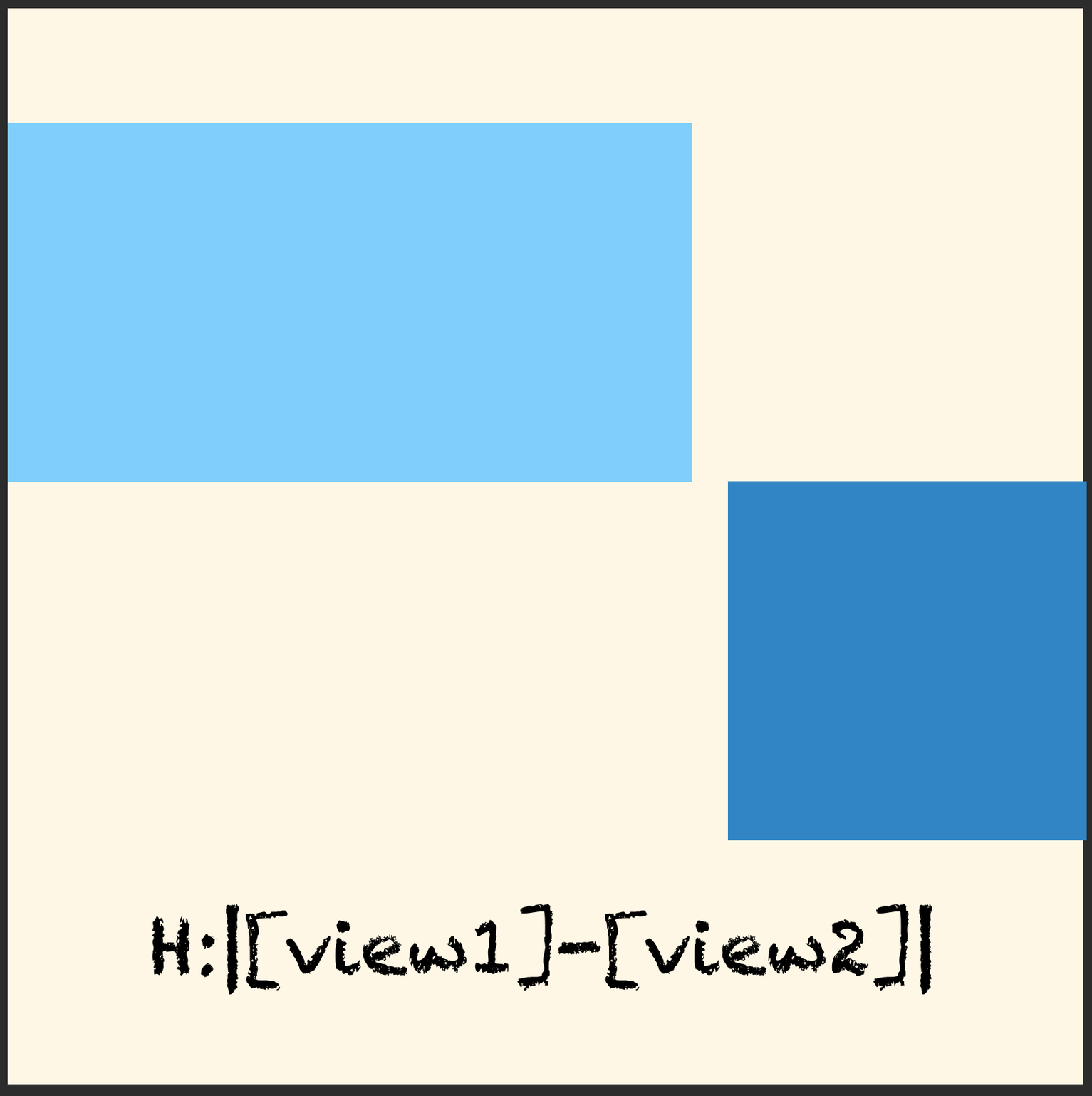
Those two are the same.



Those two are the same.

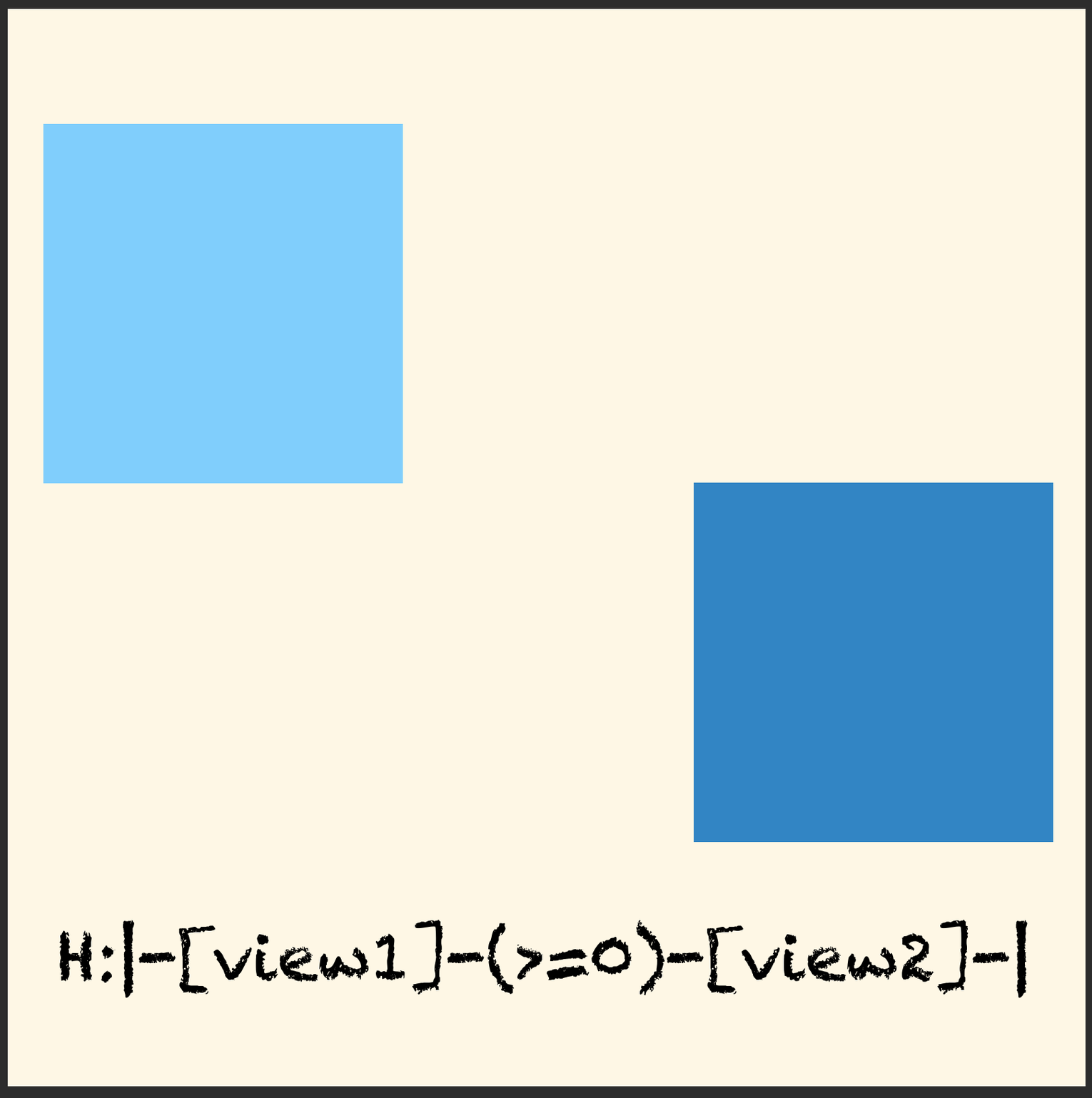


Those two are the same.

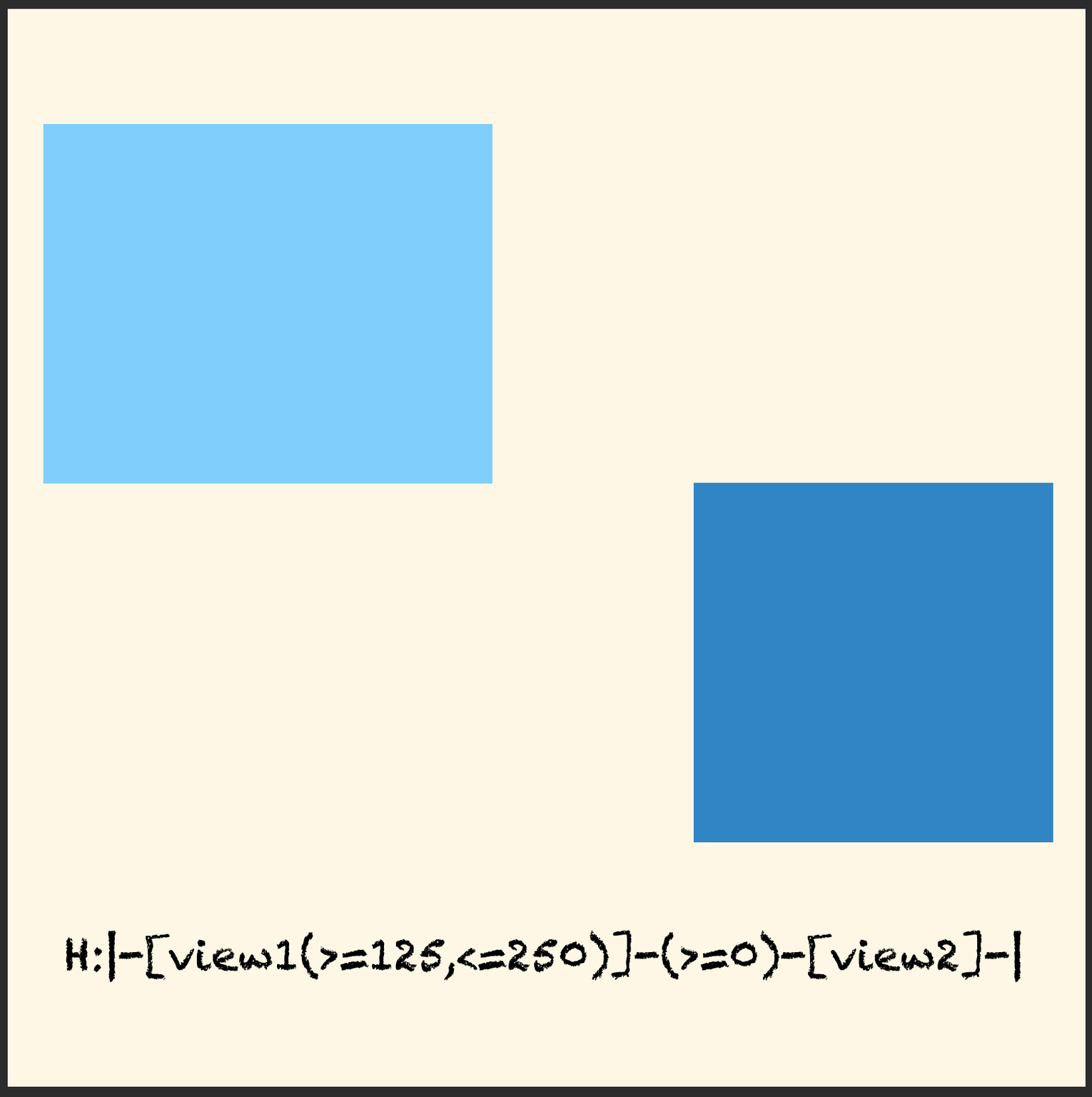


I didn’t animate these.

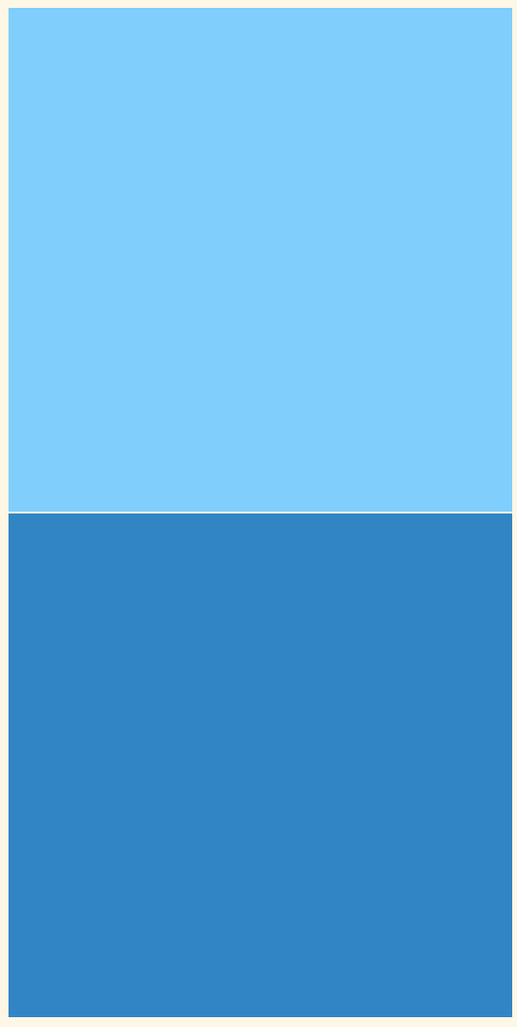
They both hug the edges so one has to resize. Which one is undefined unless one of them has low hugging priority, in which case, that one stretches.



We are setting variable space in the middle.



Size between 125 and 250.



H:[view1(>=view2)][view2]

view 1 is equal or bigger horizontally than view 2.
Nothing happens because nothing needs to happen.

H:[button(100@20)]

H:|[view1]-(>=50@30)-[view2]|

H:|-[view1(==view2)]-[view2]-|

H:[view1(view2)]

...

Bored of drawing.

H:[button(100@20)] Width 100, priority 20.
H:[view1(view2)] Both views have same width.

100x100 Square

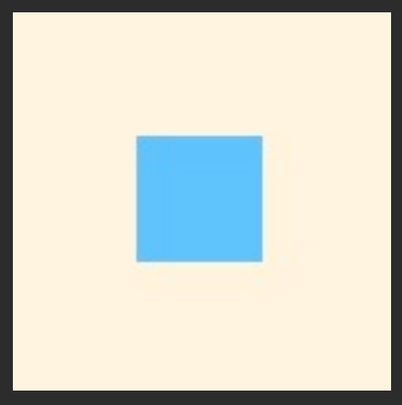
```
- (void)viewDidLoad
{
    [super viewDidLoad];

    self.blueView.translatesAutoresizingMaskIntoConstraints = NO;

    [self.blueView setContentHuggingPriority:UILayoutPriorityDefaultHigh
forAxis:UILayoutConstraintAxisHorizontal];
    [self.blueView setContentCompressionResistancePriority:UILayoutPriorityDefaultHigh
forAxis:UILayoutConstraintAxisVertical];

    [self.blueView removeConstraints:self.blueView.constraints];
    [self.blueView.superview removeConstraints:self.blueView.superview.constraints];

    NSArray *constraints = @[ @"H:[blueView(100)]", @"V:[blueView(100)]"];
    NSDictionary *views = @{@"blueView":self.blueView};
    for (NSString *format in constraints)
    {
        [self.view addConstraints:
         [NSLayoutConstraint
          constraintsWithVisualFormat: format
                              options: 0
                              metrics: nil
                                views: views]];
    }
}
```



I'm setting the size with constraints. A better way is to use intrinsicContentSize.

```
- (CGSize)intrinsicContentSize {
    return CGSizeMake(100,100);
}
```

UIView properties
Interface Builder > VFL > **API**
Animation

UIView API

UIView API

Opting in to Constraint-Based Layout

```
+ requiresConstraintBasedLayout
– translatesAutoresizingMaskIntoConstraints
– setTranslatesAutoresizingMaskIntoConstraints:
```

Managing Constraints

```
– constraints
– addConstraint:
– addConstraints:
– removeConstraint:
– removeConstraints:
```

Measuring in Constraint-Based Layout

```
– systemLayoutSizeFittingSize
– intrinsicContentSize
– invalidateIntrinsicContentSize
– contentCompressionResistancePriorityForAxis:
– setContentCompressionResistancePriority:forAxis:
– contentHuggingPriorityForAxis:
– setContentHuggingPriority:forAxis:
```

- requires constraints, translates masks to constraints

- add/remove constraints

- contentSize, compression, hugging.

systemLayoutSizeFittingSize: accepts UILayoutFittingCompressedSize,UILayoutFittingExpandedSize as parameters; it returns the largest or smallest size that fits the constraints.

UIView API

Aligning Views with Constraint-Based Layout

- `alignmentRectForFrame:`
- `frameForAlignmentRect:`
- `alignmentRectInsets`
- `viewForBaselineLayout`

Triggering Constraint-Based Layout

- `needsUpdateConstraints`
- `setNeedsUpdateConstraints`
- `updateConstraints`
- `updateConstraintsIfNeeded`

Debugging Constraint-Based Layout

- `constraintsAffectingLayoutForAxis:`
- `hasAmbiguousLayout`
- `exerciseAmbiguityInLayout`

- align views
- update constraints
- debugging

CALayer API

CALayer

- `layoutIfNeeded`

-layoutIfNeeded forces Autolayout to run immediately, rather than deferring until the end of the run loop.

UIViewController

viewDidLoad
- autolayout-
viewDidLayoutSubviews
viewDidAppear

[self.view layoutIfNeeded]

Auto Layout does its calculations after viewDidLoad. If you need the frame size use viewDidAppear: or viewDidLayoutSubviews, or call [self.view layoutIfNeeded] inside viewDidLoad.

UIView properties

Interface Builder > VFL > API

Animation

Animation

Without Auto Layout, animation is accomplished changing a view's frame over time. With Auto Layout, the constraints dictate the view frame, so you have to animate the constraints instead. This indirection makes animation harder to visualize.

Animation

Without Auto Layout, animation is accomplished changing a view's frame over time. With Auto Layout, the constraints dictate the view frame, so you have to animate the constraints instead. This indirection makes animation harder to visualize.

Animation

#238: Animate the constant

Animate constraint.constant after constraint creation using periodic calls (CADisplayLink, dispatch_source_t, dispatch_after, NSTimer).

constant

```
self.someConstraint.constant = 10.0;
[UIView animateWithDuration:0.25 animations:^{
    [self.view layoutIfNeeded];
}];
```

Animation

#238: Animate the constant.

#238: Call layoutIfNeeded in a block.

Change the constraints and call [view layoutIfNeeded] inside an animation block. This interpolates between the two positions ignoring constraints during the animation.

```
[UIView animateWithDuration:0.5 animations:^{
    [view layoutIfNeeded];
}]
```

layoutIfNeeded

```
- (BOOL)continueTrackingWithTouch:(UITouch *)touch
withEvent:(UIEvent *)event
{
    CGPoint touchPoint = [touch locationInView:self];
    [UIView animateWithDuration:0.1f animations:^(){
        NSLayoutConstraint *constraint =
            [trackView constraintNamed:THUMB_POSITION_TAG];
        constraint.constant = touchPoint.x;
        [trackView layoutIfNeeded];
    }];
    return YES;
}
```

That code drags a view on touch.

Animation

#238: Animate the constant.
#238: Call layoutIfNeeded in a block.
Animate layers instead of views.

Animate layers instead views. Layer transforms don't trigger the Auto Layout.
Applying a transform to a view triggers auto layout immediately.

Layer animation

```
// jumpy
[UIView animateWithDuration:0.3 delay:0
options:UIViewAnimationOptionAutoreverse
animations:^{
    v.transform = CGAffineTransformMakeScale(1.1, 1.1);
} completion:^(BOOL finished) {
    v.transform = CGAffineTransformIdentity;
}];

// smooth
CABasicAnimation* ba = [CABasicAnimation
animationWithKeyPath:@"transform"];
ba.autoreverses = YES;
ba.duration = 0.3;
ba.toValue = [NSValue
valueWithCATransform3D:CATransform3DMakeScale(1.1, 1.1, 1)];
[v.layer addAnimation:ba forKey:nil];
```

Setting the transform makes the view jumpy because it’s calculating the layout on each frame.
Instead we animate the layer, which doesn’t call autolayout.

Animation

#238: Animate the constant.
#238: Call layoutIfNeeded in a block.
Animate layers instead of views.
Drop constraints, use autosizing masks.

Remove all constraints and use autosizing masks. For the later, you have to set view.translatesAutoresizingMaskIntoConstraints = YES.

Animation

#238: Animate the constant
#238: Call layoutIfNeeded in a block
Animate layers instead of views.
Drop constraints, use autosizing masks.
Use a container view.

Use the constraints on the superview, and animate the subview inside the superview.

Animation

#238: Animate the constant.
#238: Call layoutIfNeeded in a block.
Animate layers instead of views.
Drop constraints, use autosizing masks.
Use a container view.
Use constraints that don’t interfere.

Use constraints that don't interfere with the intended animation.

For example, if you want to animate the size, only add constraints that set the position, and don’t set the contentSize.

Constraints can be removed or added at any time. Just call addConstraint: or enumerate view.constraints and call removeContraint:.

Animation

#238: Animate the constant.
#238: Call layoutIfNeeded in a block.
Animate layers instead of views.
Drop constraints, use autosizing masks.
Use a container view.
Use constraints that don't interfere.
Set frame in viewDidLayoutSubviews.

Auto Layout is applied in UIView.layoutSubviews, so once done, change it in UIViewController.viewDidLayoutSubviews.

Animating Rotations

Fading in/out during rotation

```
- (void)willRotateToInterfaceOrientation:(UIInterfaceOrientation)to
duration:(NSTimeInterval)duration
{
    // fade away old layout
    [UIView animateWithDuration:duration animations:^{
        for (UIView *view in @[settingsView, creditsView])
        view.alpha = 0.0f;
    }];
}

- (void) didRotateFromInterfaceOrientation:(UIInterfaceOrientation)from
{
    // update the layout for the new orientation
    [self updateViewConstraints];
    [self.view layoutIfNeeded];

    // fade in the new layout
    [UIView animateWithDuration:0.3f animations:^{
        for (UIView *view in @[settingsView, creditsView])
        view.alpha = 1.0f;
    }];
}
```

It takes twice as long to execute.

Update and animate changes

```
- (void) willAnimateRotationToInterfaceOrientation:
(UIInterfaceOrientation)to
duration:(NSTimeInterval)duration
{
   [UIView animateWithDuration:duration animations:^{
      [self updateViewConstraints];
      [self.view layoutIfNeeded];
   }];
}
```

This solution animates the layout update during reorientation, using the reorientation animation timing. This coordinates the two updates, so they finish simultaneously and draw the least attention to the updates.

Calling updates

```
UIDeviceOrientation newOrientation;

- (void) updateViewConstraints
{
    [super updateViewConstraints];
    [self.view removeConstraints:self.view.constraints];
    if (newOrientation==UIDeviceOrientationPortrait){
        // ...
    }
}

- (void) willRotateToInterfaceOrientation:
(UIInterfaceOrientation)toInterfaceOrientation
duration:(NSTimeInterval)duration
{
    newOrientation = toInterfaceOrientation;
    [self updateViewConstraints];
}
```

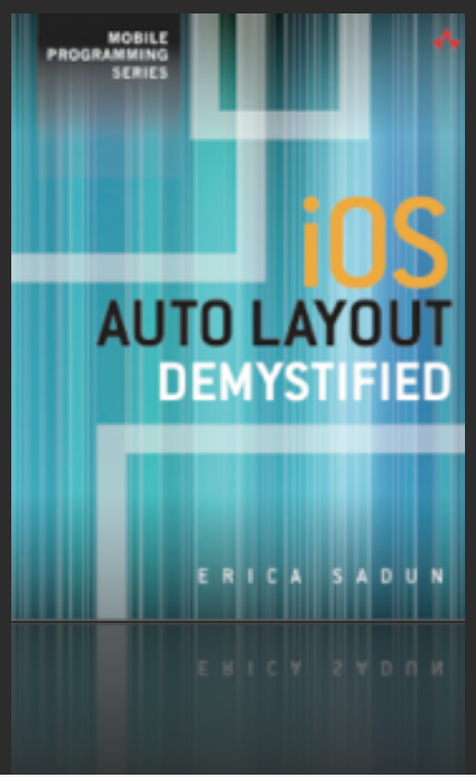
Change the constraints on rotation.

References

#202 WWDC 2012: Introduction to Auto Layout for iOS and OS X
#228 WWDC 2012: Best Practices for Mastering Auto Layout
#232 WWDC 2012: Auto Layout by Example
#406 WWDC '13 Taking Control of Auto Layout in Xcode 5

Cocoa Auto Layout Guide

iOS Auto Layout Demystified



$16

238 pages