Demo: Making a UI
Autoresizing Views

• As on Mac, views in iOS can specify autoresizing behavior using struts and springs

• No Autolayout on iOS

• Particularly important on iOS because of device rotation

• Most of the time, you can get the desired rotation behavior just by configuring the autoresize properties
Outlets & Actions

- Almost the same as on Mac
- Outlets work exactly the same
- Actions are slightly more interesting on iOS
Actions

- On Mac, an action must be a one-argument method that takes in the sender
- On iOS, the sender parameter is not required
- iOS controls can send multiple actions
- Many different events can trigger actions
Actions

- (void) addTarget: (id) target
  action: (SEL) action
  forControlEvents: (UIControlEvents) controlEvents;
Actions

[button addTarget:self action:@selector(tapDown) forControlEvents:UIControlEventTouchUpInside];

[button addTarget:self action:@selector(cancel) forControlEvents:UIControlEventTouchUpInsideOutside | UIControlEventTouchCancel];
Control Events

define:

class ControlEvent {
    @interface Event {
        UIControlEvent touchDown;
        UIControlEvent touchDownRepeat;
        UIControlEvent touchDragInside;
        UIControlEvent touchDragOutside;
        UIControlEvent touchDragEnter;
        UIControlEvent touchDragExit;
        UIControlEvent touchUpInside;
        UIControlEvent touchUpOutside;
        UIControlEvent touchCancel;
        UIControlEventValueChanged;
        UIControlEventEditingDidBegin;
        UIControlEventEditingChanged;
        UIControlEventEditingDidEnd;
        UIControlEventEditingDidEndOnExit;
    }
};

 enum {
    UIControlEventTouchDown = 1 << 0,
    UIControlEventTouchDownRepeat = 1 << 1,
    UIControlEventTouchDragInside = 1 << 2,
    UIControlEventTouchDragOutside = 1 << 3,
    UIControlEventTouchDragEnter = 1 << 4,
    UIControlEventTouchDragExit = 1 << 5,
    UIControlEventTouchUpInside = 1 << 6,
    UIControlEventTouchUpOutside = 1 << 7,
    UIControlEventTouchCancel = 1 << 8,
    UIControlEventValueChanged = 1 << 12,
    UIControlEventEditingDidBegin = 1 << 16,
    UIControlEventEditingChanged = 1 << 17,
    UIControlEventEditingDidEnd = 1 << 18,
    UIControlEventEditingDidEndOnExit = 1 << 19,
    UIControlEventAllTouchEvents = 0x00000FFF,
    UIControlEventAllEditingEvents = 0x000F0000,
    UIControlEventApplicationReserved = 0x0F000000,
    UIControlEventSystemReserved = 0xF0000000,
    UIControlEventAllEvents = 0xFFFFFFFF
};
Demo: Slider
UIApplication

• Manages the main run loop of your application:

  • Waits for events from the mouse, keyboard

  • Dispatches events to the relevant objects
    (of class UIResponder)

• Owner of MainWindow.xib
UIView

- Superclass for all views in Cocoa Touch
- Position and size in `[view frame]`
- `[view addSubview:]` and friends used for programmatic layout
- Often subclassed; we’ll talk about making your own views later
UIWindow

• Manages window frame

• On iOS, there is (usually) just one window
The View Hierarchy

UIWindow

[window subviews]

UIView

[mainView subviews]

[blueView subviews]

Label

Text!
UIApplication

- Subclass of UIView
- Superclass for text fields, buttons, etc.
- Unlike on Mac, there is no implicit concept of value
  - Text-related controls, for instance, only have a text property that can be set
Designing for Touch
Designing for Touch

• When designing for a touch screen, you should take into account the fact that the user is touching the screen
Designing for Touch

• When designing for a touch screen, you should take into account the fact that the user is touching the screen

• ...which means that his finger is obscuring his view of that part of the screen
Designing for Touch

• When designing for a touch screen, you should take into account the fact that the user is touching the screen

• ...which means that his finger is obscuring his view of that part of the screen

• As such, you can't have controls that are too small, because the user will block it when he interacts with it
Designing for Touch

• Additionally, since fingers are large (in terms of pixel area they cover), there is an inherent error bound on the \((x,y)\) coordinates of where the touch registers

• Thus, if two controls are too close together, the user may find it difficult to tap the intended control

• The "fat finger problem"
Universal Apps

• Your application may be iPad-only, iPhone/iPod Touch only, or Universal

• One executable that runs on all device types
  • Great for users

• A bit more work for the developer
Universal Apps - Views

• If a view is created in Interface Builder:
  • One nib for iPad-family
  • One nib for iPhone-family

• If a view is created programmatically:
  • View-drawing code should be dynamic enough to work on both families (or use conditional code paths)
Universal Apps - View Controllers

• Common to have an iPad view controller and an iPhone view controller

• Can also be done in one object (again, conditional code paths)
Universal Apps - Retina Display

- Some devices (currently just the iPhone 4) have a retina display
  - x2 pixels in each dimension
- UIKit abstracts the pixels away from you
  - All drawing done in "points"
- On iPhone 4, for example, a "point" is worth two pixels in each dimension
Universal Apps - Retina Display

• Images for retina displays should have twice the resolution (in each dimension)

• Naming convention: Image.png and Image@2x.png

• [UIImage imageNamed: ...]; automatically looks for an image named using this convention
Universal Apps - Conditional Code Paths

- UIKit provides a lot of automatic handling of different devices/screen resolutions
- Sometimes, you'll need to do things manually
Universal Apps - Conditional Code Paths

• iPhone or iPad?
  • Macro: `UI_USER_INTERFACE_IDIOM()` returns `UIUserInterfaceIdiomPad` or `UIUserInterfaceIdiomPhone`

• Retina or no?
  • `[[UIScreen mainScreen] scale]` returns a CGFloat "scale factor" between points and pixels