DOWNLOAD PDF CONCEPTS, CONTEXTS, AND TRANSFORMATION

Chapter 1 : iOS Drawing Concepts

Concepts and Transformation | This problem-driven journal focuses on the role of social research in workplace reform and organizational renewal. It presents new perspectives on the relationship.

Providing high-quality graphics not only makes your app look good, but it also makes your app look like a natural extension to the rest of the system. This document describes native rendering. Quartz is the main drawing interface, providing support for path-based drawing, anti-aliased rendering, gradient fill patterns, images, colors, coordinate-space transformations, and PDF document creation, display, and parsing. Core Animation provides the underlying support for animating changes in many UIKit view properties and can also be used to implement custom animations. This chapter provides an overview of the drawing process for iOS apps, along with specific drawing techniques for each of the supported drawing technologies. You will also find tips and guidance on how to optimize your drawing code for the iOS platform. Not all UIKit classes are thread safe. Views define the portion of the screen in which drawing occurs. If you use system-provided views, this drawing is handled for you automatically. If you define custom views, however, you must provide the drawing code yourself. If you use Quartz, Core Animation, and UIKit to draw, you use the drawing concepts described in the following sections. In addition to drawing directly to the screen, UIKit also allows you to draw into offscreen bitmap and PDF graphics contexts. When you draw in an offscreen context, you are not drawing in a view, which means that concepts such as the view drawing cycle do not apply unless you then obtain that image and draw it in an image view or similar. The UIView class makes the update process easier and more efficient; however, by gathering the update requests you make and delivering them to your drawing code at the most appropriate time. There are several actions that can trigger a view update: Moving or removing another view that was partially obscuring your view Making a previously hidden view visible again by setting its hidden property to NO Scrolling a view off of the screen and then back onto the screen Explicitly calling the setNeedsDisplay or setNeedsDisplayInRect: For custom views, you must override the drawRect: During subsequent calls, the rectangle includes only the portion of the view that actually needs to be redrawn. For maximum performance, you should redraw only affected content. After calling your drawRect: If you want to change the contents of the view, however, you must tell your view to redraw its contents. For example, if you were updating content several times a second, you might want to set up a timer to update your view. You might also update your view in response to user interactions or the creation of new content in your view. That method should be called only by code built into iOS during a screen repaint. At other times, no graphics context exists, so drawing is not possible. Graphics contexts are explained in the next section. Coordinate Systems and Drawing in iOS When an app draws something in iOS, it has to locate the drawn content in a two-dimensional space defined by a coordinate system. Apps in iOS sometimes have to deal with different coordinate systems when drawing. In iOS, all drawing occurs in a graphics context. Conceptually, a graphics context is an object that describes where and how drawing should occur, including basic drawing attributes such as the colors to use when drawing, the clipping area, line width and style information, font information, compositing options, and so on. In addition, as shown in Figure , each graphics context has a coordinate system. More precisely, each graphics context has three coordinate systems: The drawing user coordinate system. This coordinate system is used when you issue drawing commands. The view coordinate system base space. This coordinate system is a fixed coordinate system relative to the view. The physical device coordinate system. This coordinate system represents pixels on the physical screen. This initial drawing coordinate system is known as the default coordinate system, and is a 1: Each view also has a current transformation matrix CTM , a mathematical matrix that maps the points in the current drawing coordinate system to the fixed view coordinate system. The app can modify this matrix as described later to change the behavior of future drawing operations. Each of the drawing frameworks of iOS establishes a default coordinate system based on the current graphics context. In iOS, there are two main types of coordinate systems: An upper-left-origin coordinate system ULO , in which the origin of drawing operations is at the upper-left corner of the drawing area, with positive values extending downward and to the right. A

lower-left-origin coordinate system LLO , in which the origin of drawing operations is at the lower-left corner of the drawing area, with positive values extending upward and to the right. These coordinate systems are shown in Figure Although the drawing functions and methods of the Core Graphics and AppKit frameworks are perfectly suited to this default coordinate system, AppKit provides programmatic support for flipping the drawing coordinate system to have an upper-left origin. This implicit graphics context establishes a ULO default coordinate system. Points Versus Pixels In iOS there is a distinction between the coordinates you specify in your drawing code and the pixels of the underlying device. These logical coordinate systems are decoupled from the device coordinate space used by the system frameworks to manage the pixels onscreen. This behavior leads to an important fact that you should always remember: One point does not necessarily correspond to one physical pixel. The purpose of using points and the logical coordinate system is to provide a consistent size of output that is device independent. For most purposes, the actual size of a point is irrelevant. The goal of points is to provide a relatively consistent scale that you can use in your code to specify the size and position of views and rendered content. How points are actually mapped to pixels is a detail that is handled by the system frameworks. For example, on a device with a high-resolution screen, a line that is one point wide may actually result in a line that is two physical pixels wide. The result is that if you draw the same content on two similar devices, with only one of them having a high-resolution screen, the content appears to be about the same size on both devices. On a standard-resolution screen, the scale factor is typically 1. On a high-resolution screen, the scale factor is typically 2. In the future, other scale factors may also be possible. In iOS prior to version 4, you should assume a scale factor of 1. Native drawing technologies, such as Core Graphics, take the current scale factor into account for you. For example, if one of your views implements a drawRect: Thus, any content you draw in your drawRect: In iOS, when you draw things onscreen, the graphics subsystem uses a technique called antialiasing to approximate a higher-resolution image on a lower-resolution screen. The best way to explain this technique is by example. When you draw a black vertical line on a solid white background, if that line falls exactly on a pixel, it appears as a series of black pixels in a field of white. If it appears exactly between two pixels, however, it appears as two grey pixels side-by-side, as shown in Figure For example, if you draw a one-pixel-wide vertical line from 1. If you draw a two-pixel-wide line, you get a solid black line because it fully covers two pixels one on either side of the specified point. As a rule, lines that are an odd number of physical pixels wide appear softer than lines with widths measured in even numbers of physical pixels unless you adjust their position to make them cover pixels fully. Where the scale factor comes into play is when determining how many pixels are covered by a one-point-wide line. On a low-resolution display with a scale factor of 1. To avoid antialiasing when you draw a one-point-wide horizontal or vertical line, if the line is an odd number of pixels in width, you must offset the position by 0. If the line is an even number of points in width, to avoid a fuzzy line, you must not do so. To draw a line that covers only a single physical pixel, you would need to make it 0. A comparison between the two types of screens is shown in Figure Of course, changing drawing characteristics based on scale factor may have unexpected consequences. A 1-pixel-wide line might look nice on some devices but on a high-resolution device might be so thin that it is difficult to see clearly. It is up to you to determine whether to make such a change. Obtaining Graphics Contexts Most of the time, graphics contexts are configured for you. Each view object automatically creates a graphics context so that your code can start drawing immediately as soon as your custom drawRect: As part of this configuration, the underlying UIView class creates a graphics context a CGContextRef opaque type for the current drawing environment. If you want to draw somewhere other than your view for example, to capture a series of drawing operations in a PDF or bitmap file , or if you need to call Core Graphics functions that require a context object, you must take additional steps to obtain a graphics context object. The sections below explain how. For more information about graphics contexts, modifying the graphics state information, and using graphics contexts to create custom content, see Quartz 2D Programming Guide. The first parameter of many of these functions must be a CGContextRef object. Flipping the Default Coordinate System discusses flip transforms in detail. In a manner similar to drawRect: This graphics context establishes a ULO default coordinate system. Both of these approaches require that you first call a function that creates a graphics contextâ€”a bitmap context or a PDF context, respectively. The returned object serves

as the current and implicit graphics context for subsequent drawing and state-setting calls. When you finish drawing in the context, you call another function to close the context. Core Graphics has corresponding functions for rendering in a bitmap graphics context and for drawing in a PDF graphics context. The context that an app directly creates through Core Graphics, however, establishes a LLO default coordinate system. However, if you do use the Core Graphics alternatives and intend to display the rendered results, you will have to adjust your code to compensate for the difference in default coordinate systems. See Flipping the Default Coordinate System for more information. Because iOS is designed to run on embedded hardware and display graphics onscreen, the RGB color space is the most appropriate one to use.

Chapter 2 : "Contexts, Concepts, and Logic of Domains" by Pascal Hitzler

This special issue continues the discussion initiated by Piaget and Vygotsky on the nature of and relationship between spontaneous and scientific concepts, and comes out of a sense of continuity with and a reconsideration of ideas presented by Vygotsky over 60 years ago.

Chapter 3 : Threshold Concepts and Transformational Learning - - SensePublishers

Transformation also occurs in the activities or processes between contexts. Double Transfer Paradigm: students need to transfer what they learned from the instructional method to learn from the resource, and they need to transfer what they learned from the resource to solve the target problem.

Chapter 4 : Chinaâ€™s Urban Communities: Concepts, Contexts and Well-Being - Harvard Graduate Scho

Over the last decade the notion of 'threshold concepts' has proved influential around the world as a powerful means of exploring and discussing the key points of transformation that students experience in their higher education courses and the 'troublesome knowledge' that these often present.

Chapter 5 : Holbrook Mahn (Editor of Concepts, Contexts, and Transformation)

ReseÃ±a del editor. This special issue continues the discussion initiated by Piaget and Vygotsky on the nature of and relationship between spontaneous and scientific concepts, and comes out of a sense of continuity with and a reconsideration of ideas presented by Vygotsky over 60 years ago.

Chapter 6 : Transformational Leadership - Leadership Training from blog.quintoapp.com

Holbrook Mahn is the author of Concepts, Contexts, and Transformation (avg rating, 0 ratings, 0 reviews, published).

Chapter 7 : Concepts and Transformation. International Journal of Action Research and Organizational Re

Mascarenhas, O A , 'An introduction to business transformation strategies: concepts, constructs, and contexts', in Business transformation strategies: the strategic leader as innovation manager, SAGE Publications India Pvt Ltd, New Delhi, pp. , viewed 25 October , doi: /n1.

Chapter 8 : Willem L. Wardekker (Editor of Concepts, Contexts, and Transformation)

This problem-driven journal focused on the role of social research in workplace reform and organizational renewal. It presented new perspectives on the relationship between theory and practice in social science.

Chapter 9 : Concepts and definitions | EIGE

3 THE CHANGE CONCEPTS FOR PRACTICE TRANSFORMATION: OVERVIEW CHANGING CARE DELIVERY ORGANIZED, EVIDENCE-BASED CARE â€¢ Use planned care according to patient need. â€¢ Identify high risk patients and ensure they are receiving appropriate care and case management services.