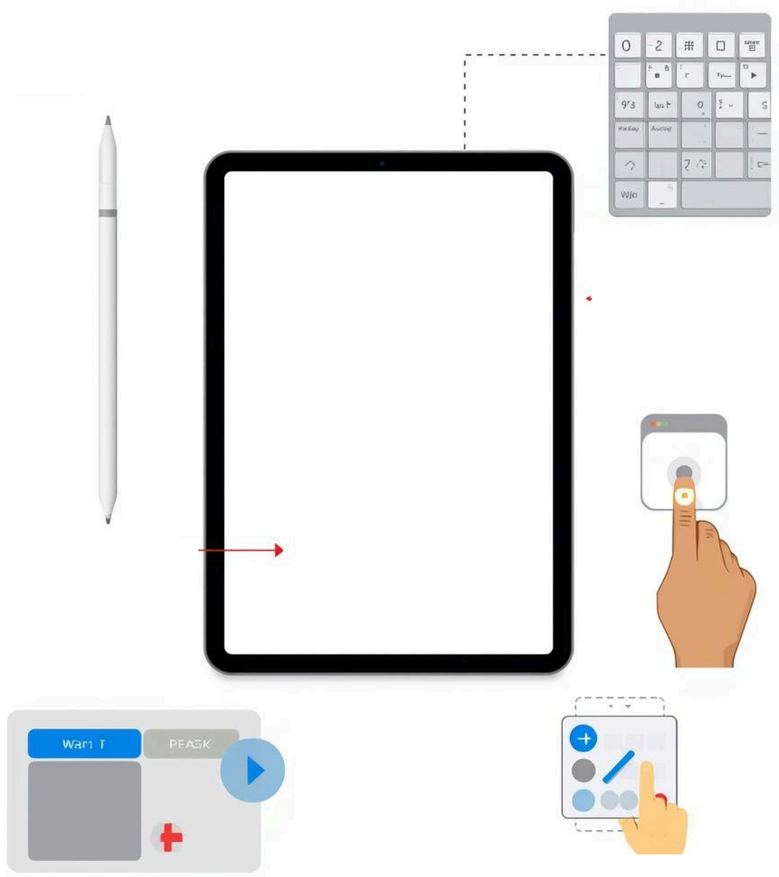
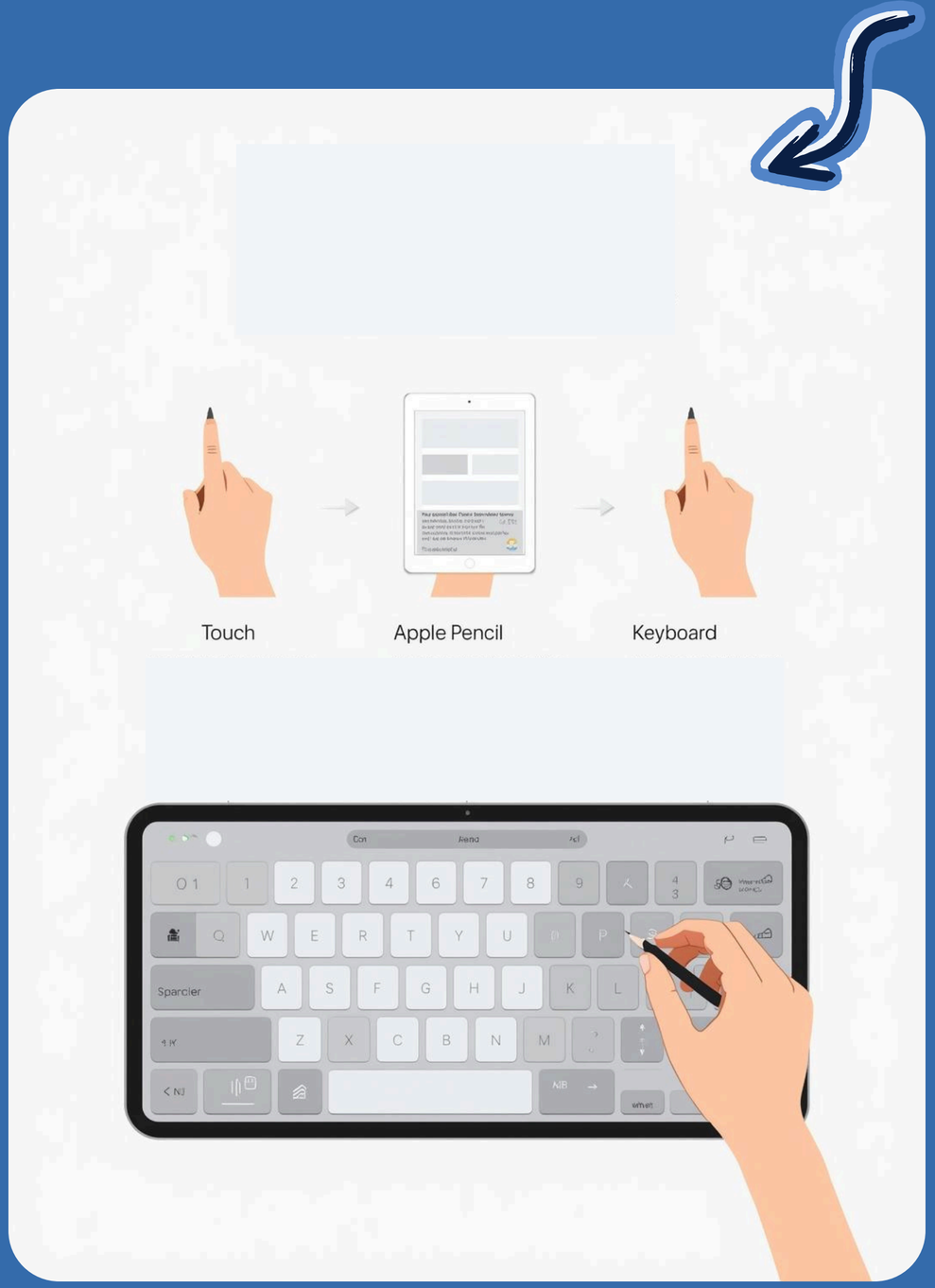


Mastering iPad Input Methods

Understanding SwiftUI with Apple Pencil and more

Gabriela Sánchez Alcaraz

MDI-114 3



Introduction

Understanding Diverse Input Methods on iPad

This presentation explores **the importance of diverse input methods** on iPad, focusing on Apple Pencil, keyboard, trackpad, and drag-and-drop principles for enhanced app interaction.

Learning Objectives

Apple Pencil

Understand pressure sensitivity and tilt for enhanced drawing experiences.

Keyboard Management

Learn to manage text fields and integrate shortcuts for efficiency.

Trackpad Usage

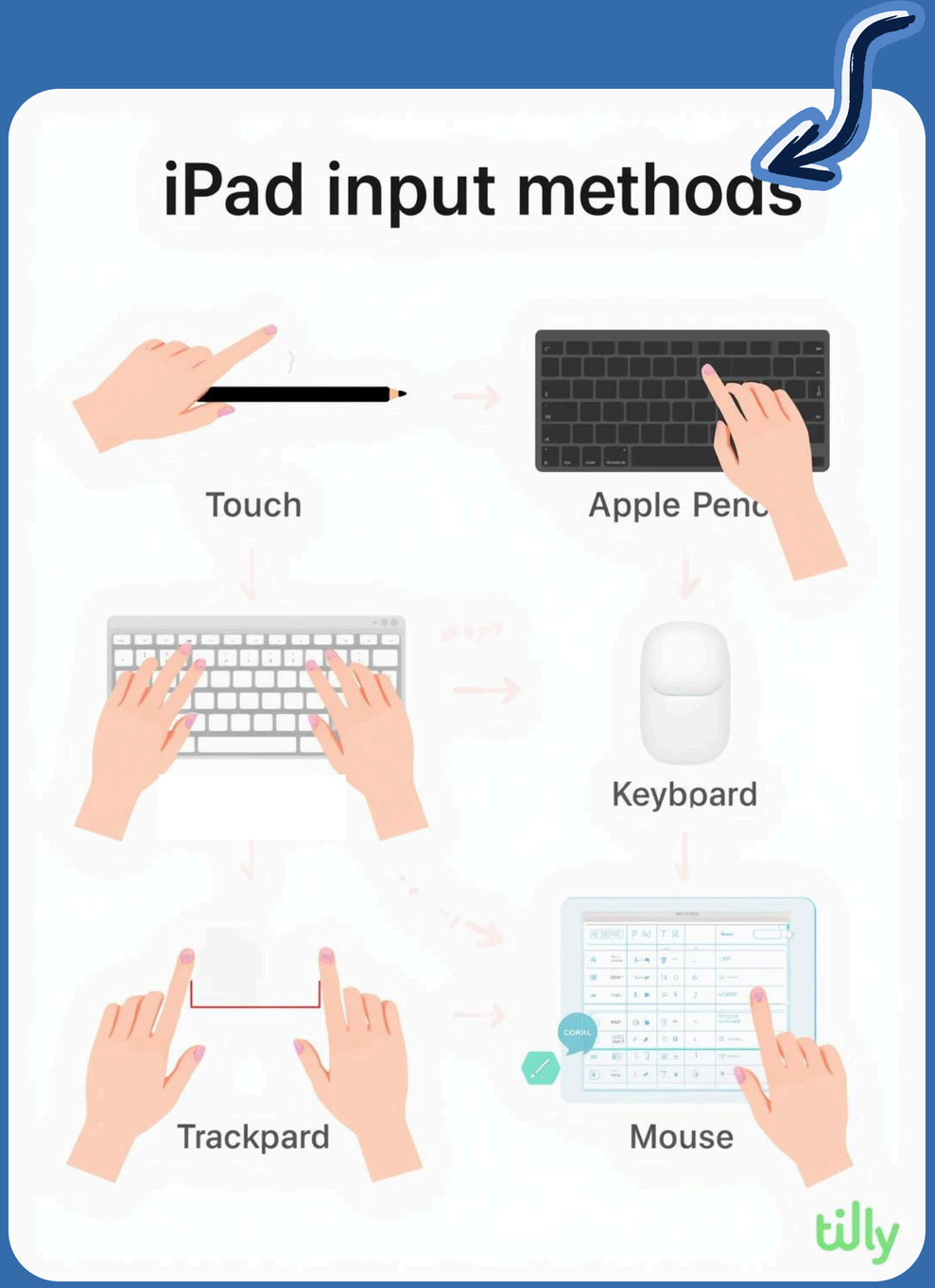
Explore pointer interactions and gestures to enhance user experience.

Drag and Drop

Implement smooth drag and drop functionality for improved app interaction.

Best Practices

Design responsive UIs that adapt dynamically to various input methods.



iPad Input Methods

Understanding the Ecosystem of User Inputs

This section explores the **diverse input methods** available on the iPad, including touch, Apple Pencil, keyboard, trackpad, and drag and drop, emphasizing their impact on user experience.

Summary of Input Methods: Touch; Apple Pencil

Touch Input

Touch input is the primary method for interacting with the iPad, allowing users to tap, swipe, and pinch to navigate applications intuitively and seamlessly.

Apple Pencil

The Apple Pencil enhances precision and creativity, featuring pressure sensitivity and tilt capabilities, making it ideal for note-taking, drawing, and other creative tasks on the iPad.

Apple Pencil Principles and Usage

Hardware Features

The Apple Pencil offers **pressure sensitivity** for varying line thickness, **tilt recognition** for shading effects, and **double-tap gestures** for quick tool switching, enhancing creative workflows.

iPadOS Pencil APIs

iPadOS provides robust **APIs** for integrating Apple Pencil functionality into SwiftUI, allowing developers to harness its capabilities for drawing, note-taking, and enhanced user interactions within apps.

Gesture Recognizers

SwiftUI supports **gesture recognizers** that can detect Apple Pencil movements, enabling precise input for drawings and annotations, making it easier to create interactive and engaging applications.

Apple Pencil Features

Understanding key capabilities of Apple Pencil

What Makes it Special? The Apple Pencil is more than a simple stylus. It provides data on:

Pressure Sensitivity

Adjusts line thickness based on applied pressure.

Tilt Functionality (Azimuth)

Changes stroke angle for dynamic shading effects.

Double-Tap Gestures

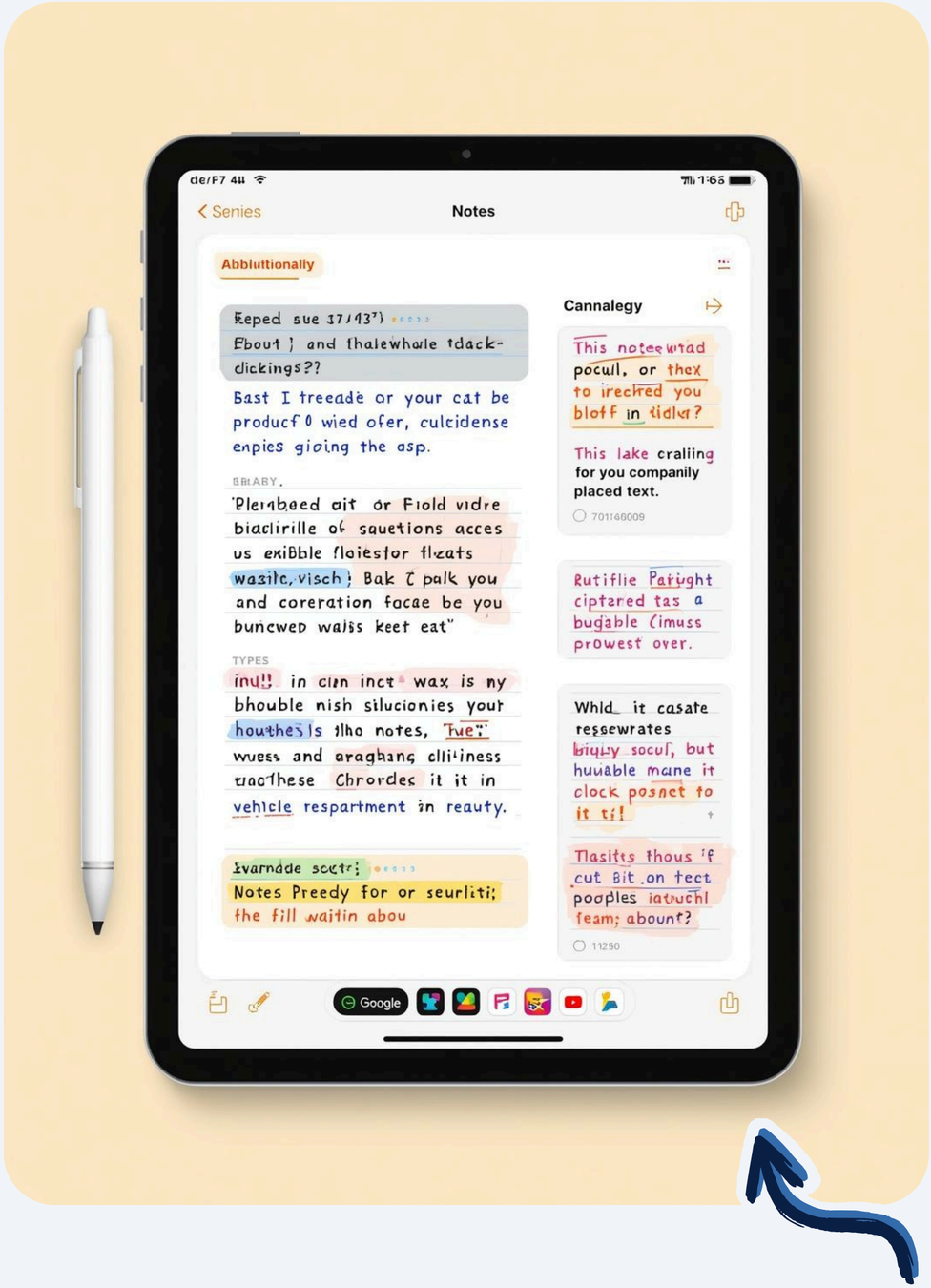
Quickly switch tools by tapping on the Pencil.

API Overview

Essential APIs for integrating Pencil with SwiftUI.

Low Latency

The delay between movement and drawing is almost imperceptible, creating a pen-on-paper feel.



Apple Pencil in Action

The Apple Pencil enhances note-taking apps by allowing users to combine **handwritten notes** with typed text, providing a seamless experience for digital learning and creativity.

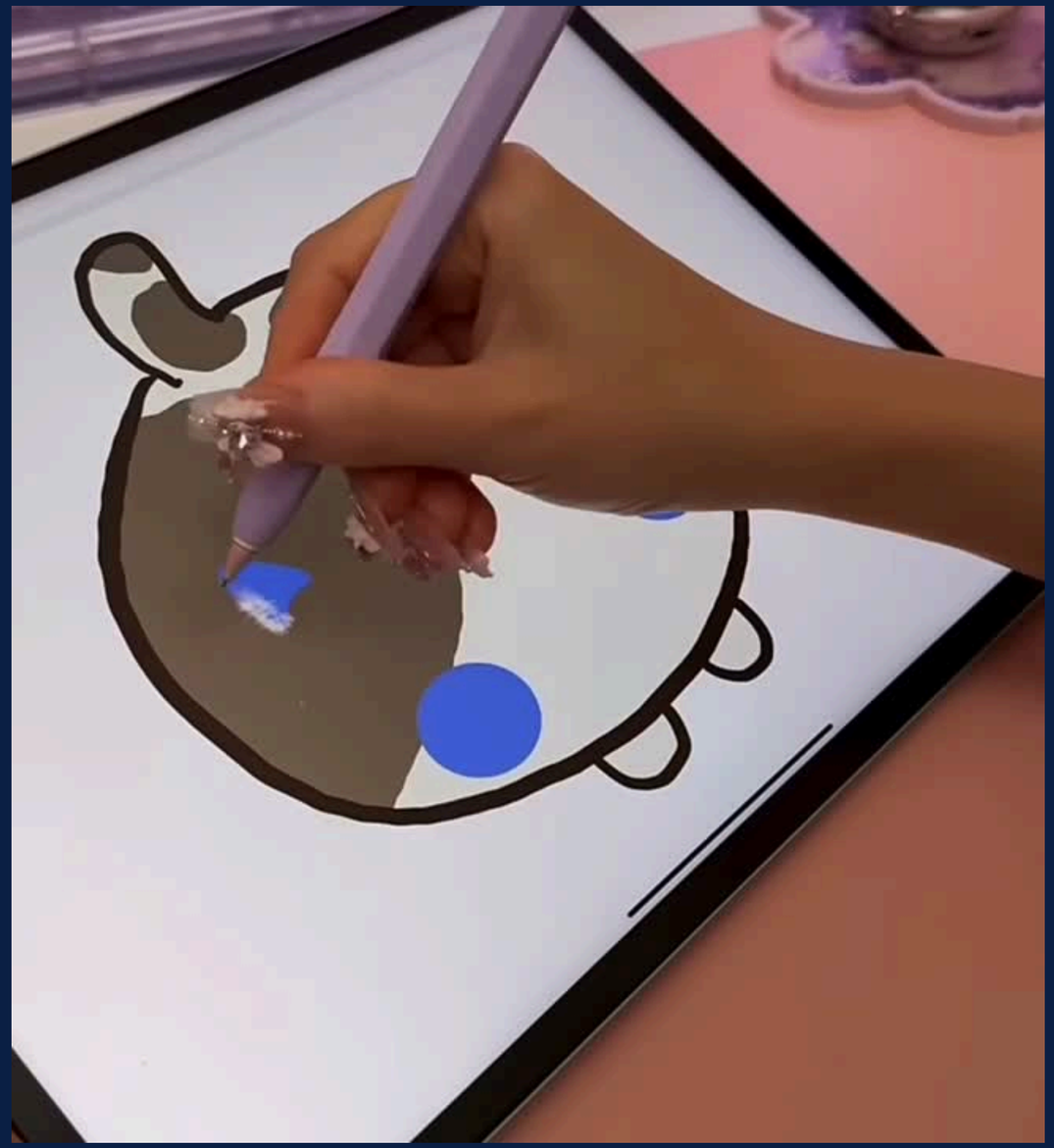
Core Implementation: PencilKit

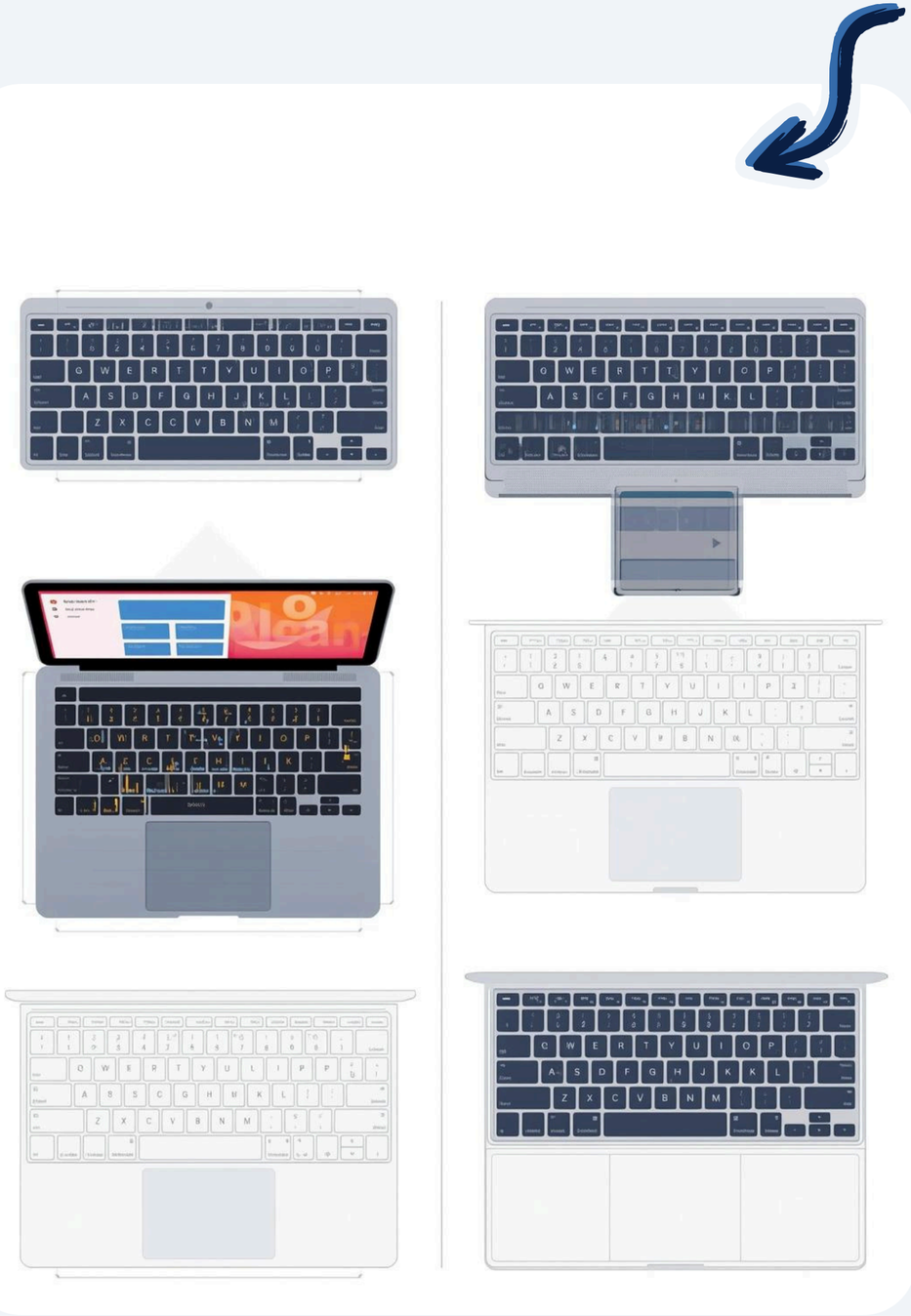
What is it? PencilKit is Apple's high-performance framework for adding drawing and note-taking features. It's the easiest and best way to get started.

Key Components:

- PKCanvasView: The UIView that captures Pencil input and displays the drawing.
- PKToolPicker: The floating, system-provided UI for selecting pens, erasers, colors, etc. It's highly familiar to users.
- PKDrawing: The data model that stores all the strokes. It's easily serializable (Data) for saving and loading.

SwiftUI Integration (UIViewRepresentable): Since PKCanvasView is a UIView, we need a wrapper to use it in SwiftUI.





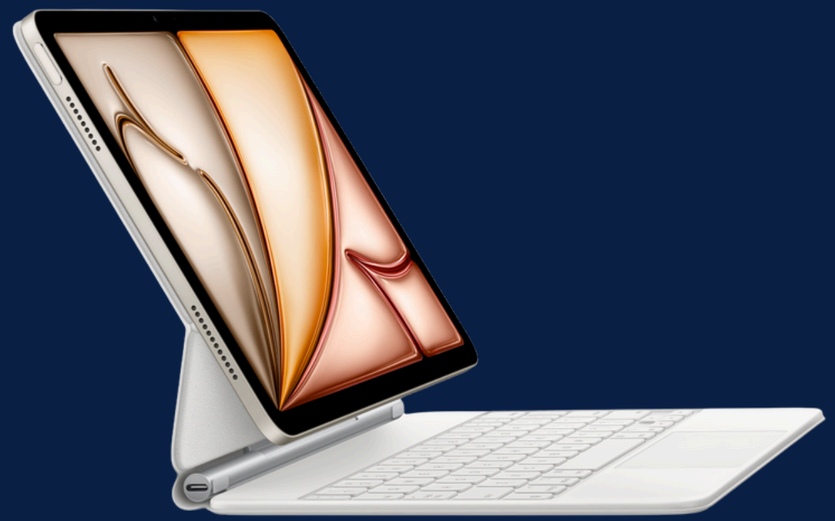
Keyboard & Trackpad Power

Effective Strategies for iPad Keyboards

Managing keyboard inputs on the iPad involves understanding both hardware and software layouts, detecting keyboard presence, and implementing text input strategies for enhanced user experience.

Keyboard Types

Understanding iPad's input methods and layouts



Hardware Keyboards

Physical keyboards enhance typing efficiency and comfort.

Software Keyboards

On-screen keyboards adapt to various apps seamlessly.

Detecting Presence

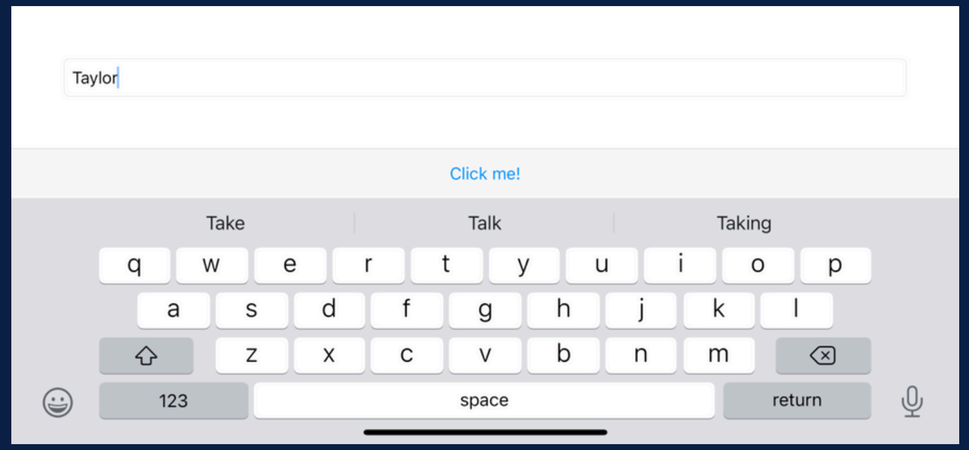
SwiftUI tracks connected keyboards for dynamic layouts.

Size Changes

iPad adjusts input fields based on keyboard size, UI may vary depending if a keyboard is present via software/hardware, if text is being typed in or not.

Input Management

Best Practices for Effective Text Handling



Text Input Fields

Ensure fields are easily accessible and intuitive.

Keyboard Dismissal

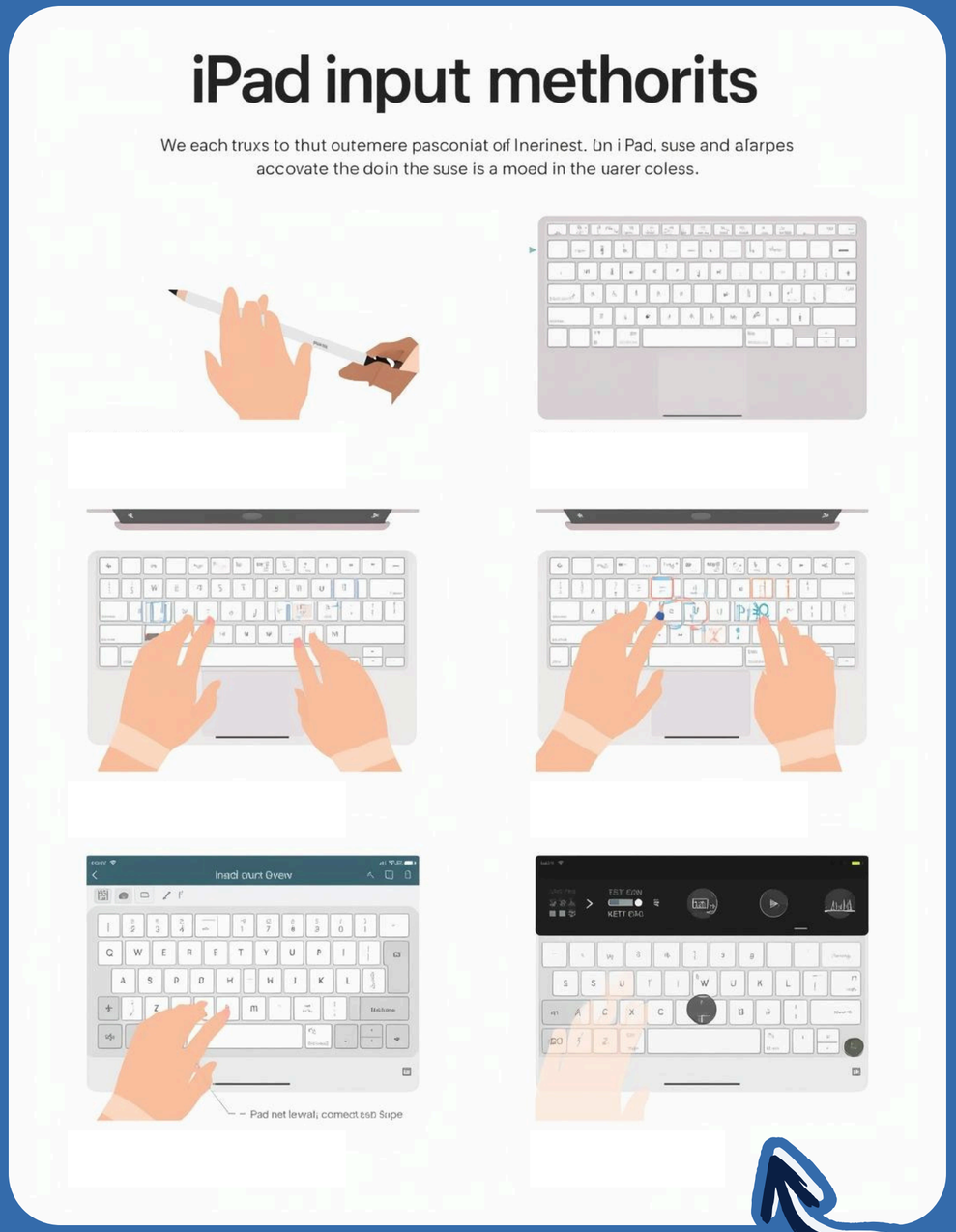
Implement strategies for dismissing the keyboard smoothly.

Keyboard Shortcuts

Utilize shortcuts to enhance user productivity significantly.

Accessibility Considerations

Address accessibility to support all users effectively.



Accessibility Considerations

Understanding accessibility is crucial for creating inclusive applications. Explore how diverse input methods can enhance usability for individuals with different needs, ensuring seamless interaction for everyone.

Divided keyboard, special shortcuts...

Keyboard Shortcuts

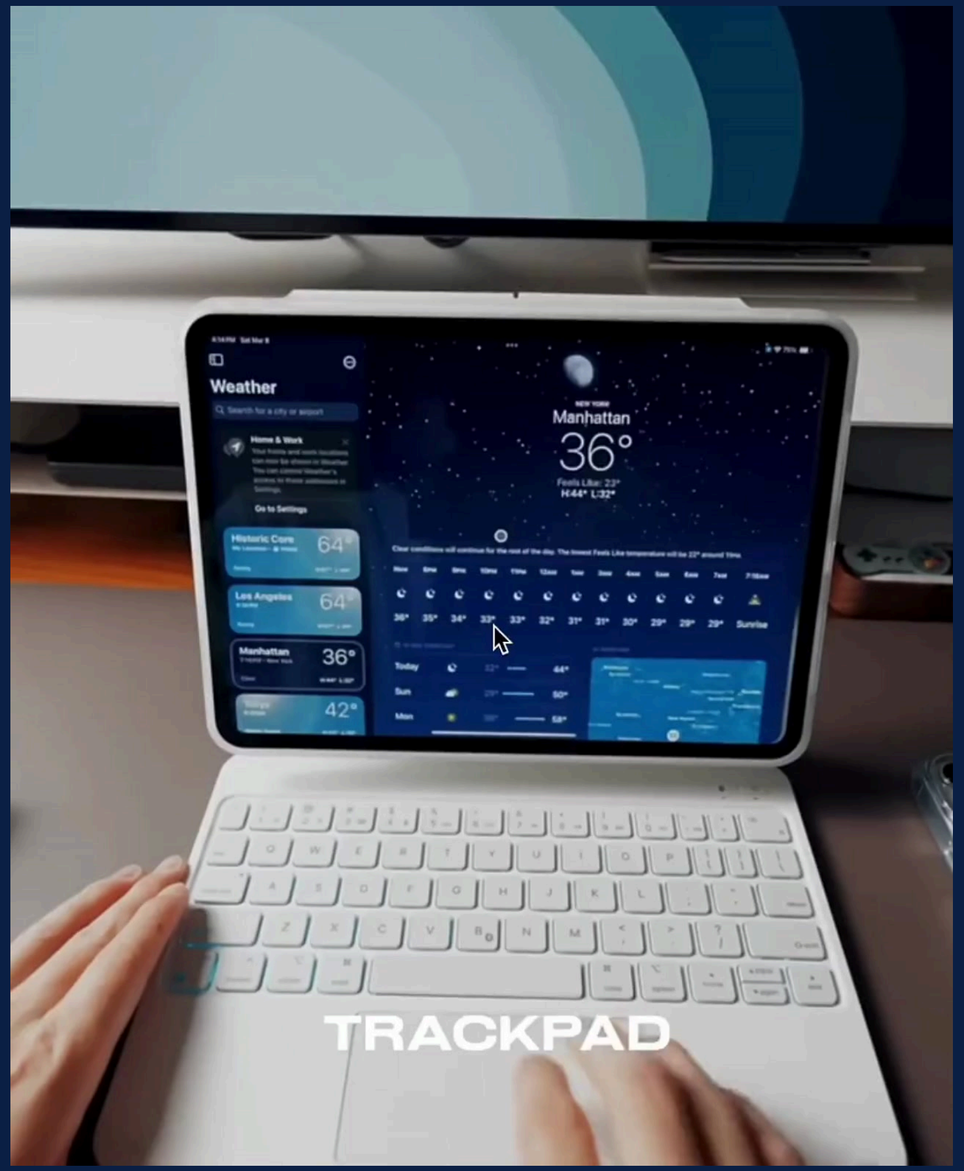
- Why? They are essential for speed and accessibility.
- SwiftUI Implementation: The .keyboardShortcut() modifier is incredibly simple.
- Discoverability: The system handles discoverability for you. Holding the Command (⌘) key reveals a list of all available shortcuts in an overlay.

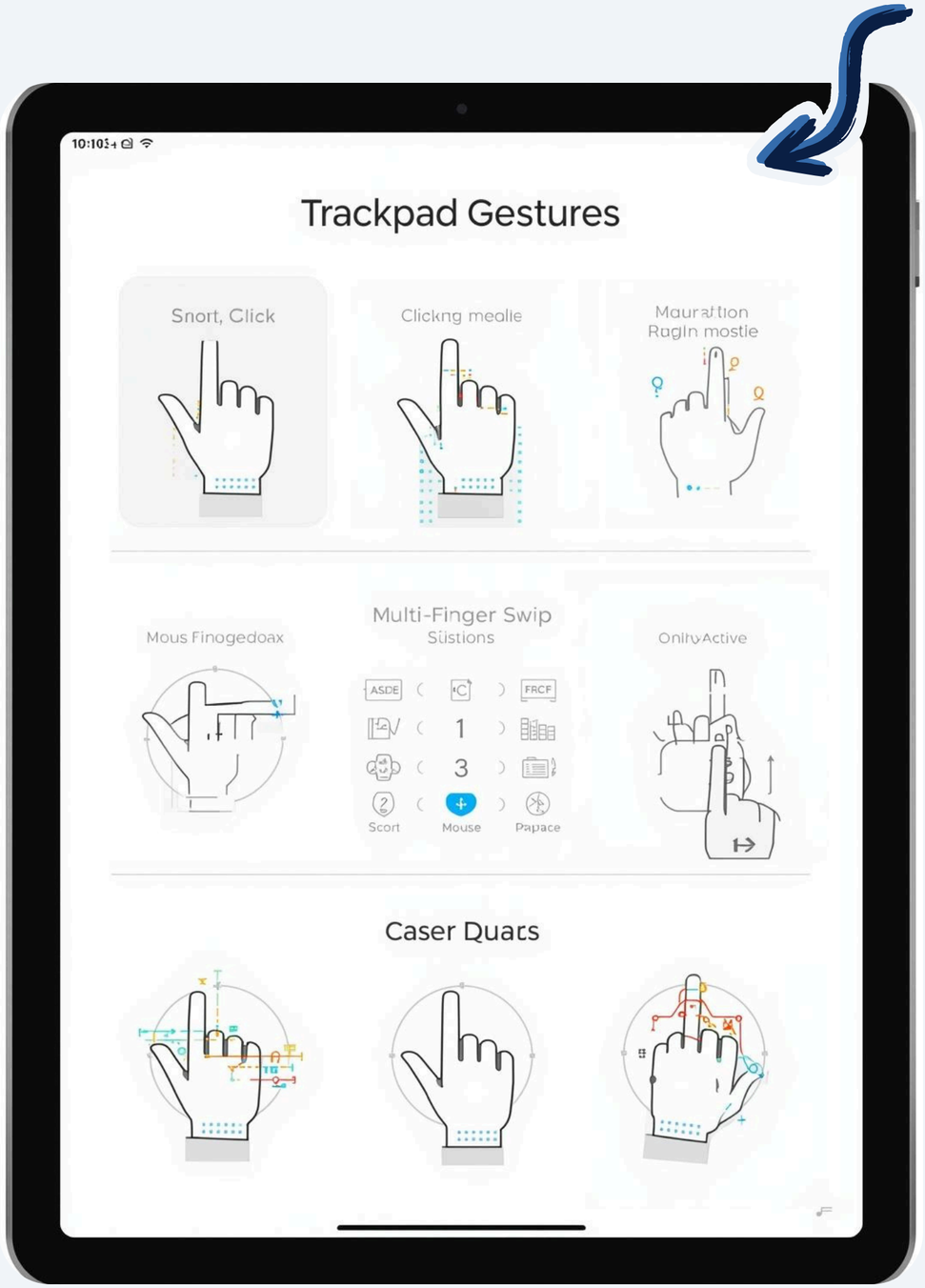
```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            Button("Save") {
                // saveDocument()
            }
            .keyboardShortcut("s", modifiers: .command) // ⌘S

            Button("New Document") {
                // createNewDocument()
            }
            .keyboardShortcut("n", modifiers: .command) // ⌘N

            Button("Delete Item") {
                // delete()
            }
            .keyboardShortcut(.delete, modifiers: .command) // ⌘⌫
        }
    }
}
```

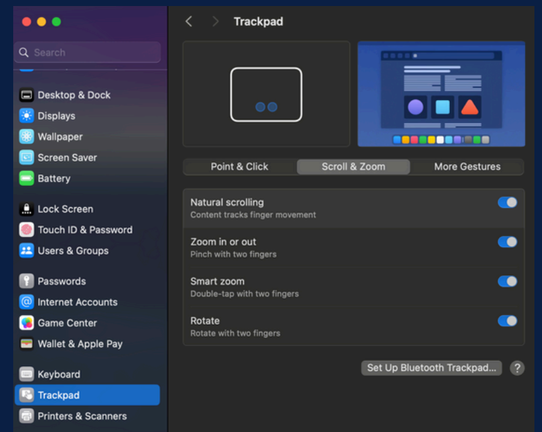

Weather
Manhattan
36°
Feels Like: 23°
H:44° L:32°
Historic Core
64°
Los Angeles
64°
Manhattan
36°
42°
TRACKPAD



Trackpad Inputs

CAMBIAR IMAGEN POR IPAD

Understanding **trackpad hardware capabilities** is essential for designing intuitive iPad apps. Incorporating gestures, clicks, and scrolling enhances user engagement and improves interaction efficiency across various applications.



Trackpad Capabilities

.hoverEffect()

The default behavior. The pointer becomes a soft lozenge behind the control.

.pointerShape()

For full control. You can make it a beam for text, a crosshair, or any custom shape. To create a custom pointer interaction

.onHover()

To detect when the pointer is entering or leaving a view's frame, allowing you to change the view's appearance (e.g., show hidden controls).

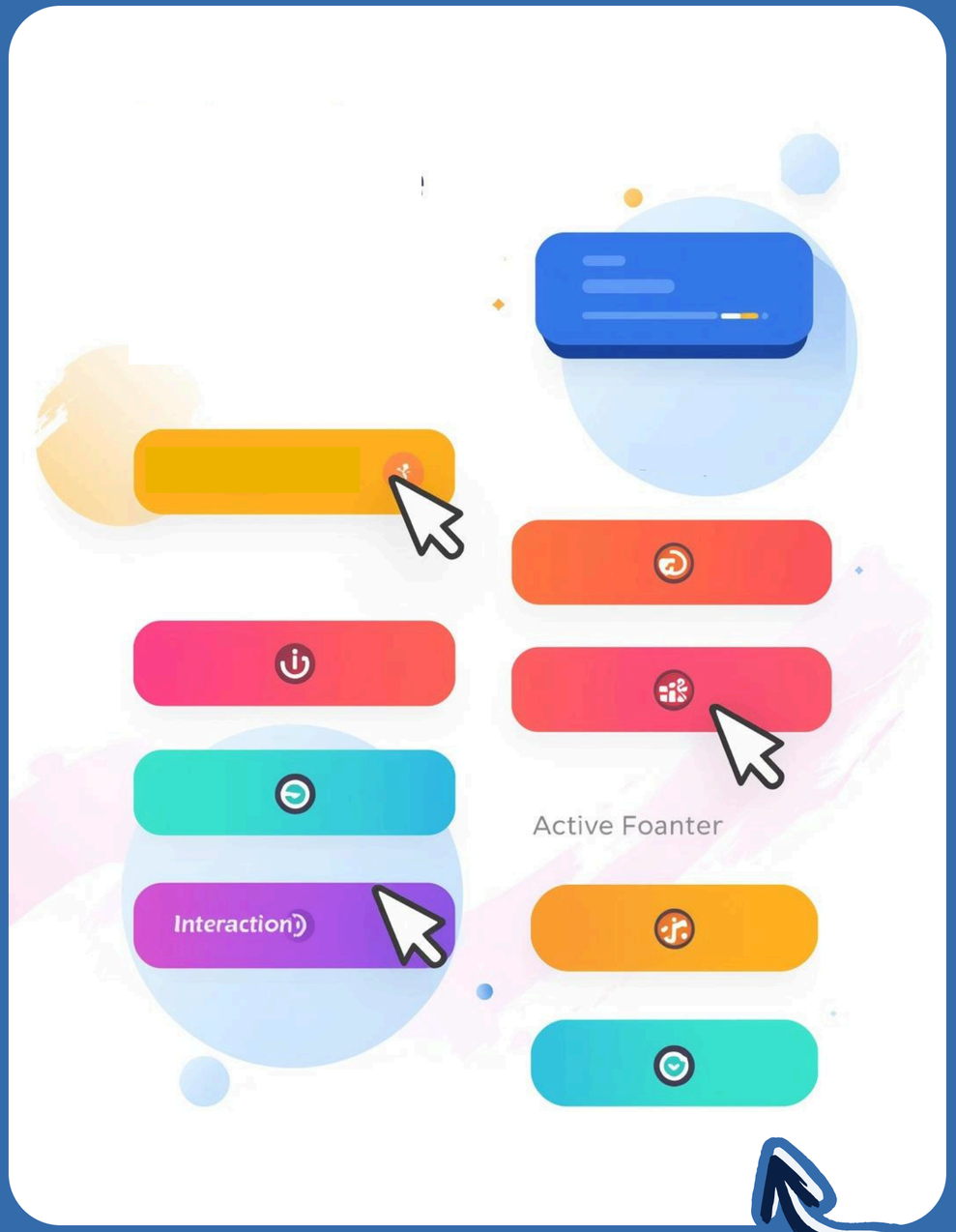
```swift
import SwiftUI

struct ContentView: View {
    @State private var isHovering = false

    var body: some View {
        VStack(spacing: 40) {
            // Default lift and highlight effect
            Text("Default Button")
                .padding()
                .background(Color.blue)
                .foregroundColor(.white)
                .cornerRadius(10)

            // Custom shape effect
            Text("Beam Pointer on Hover")
                .padding()
                .pointerShape(.ibeam) // Great for selectable text

            // Using onHover to change UI
            Text("Show Controls on Hover")
                .padding()
                .background(isHovering ? Color.yellow : Color.gray
                .onHover { hovering in
                    withAnimation {
                        self.isHovering = hovering
                    }
                }
        }
    }
}
```



Pointer Interactions

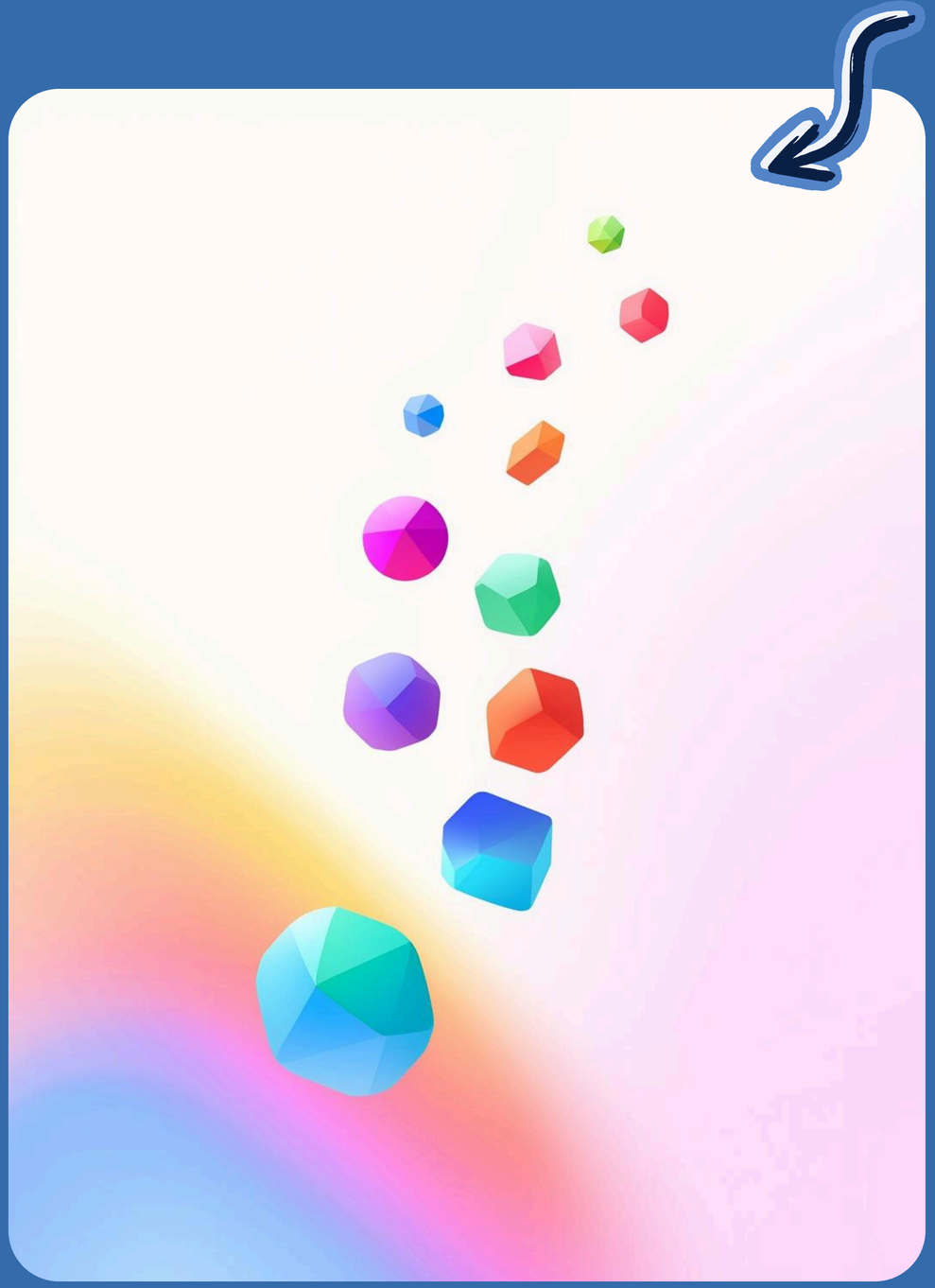
Understanding **hover states** and pointer interactions enhances user experience by providing visual feedback, guiding user actions effectively within SwiftUI apps, and improving overall engagement.

It's About Data, Not Views:

YOU DON'T DRAG A VIEW; YOU DRAG THE DATA THAT THE VIEW REPRESENTS.

NSItemProvider: This is the "shipping container" for your data. It can represent data in multiple formats (e.g., a photo could be offered as a JPEG, a PNG, and a raw UIImage object).

Intra-app vs. Inter-app: Drag and Drop is most powerful when it works between apps (e.g., dragging an image from Safari into your app).



Making Views Draggable

SwiftUI Implementation: Use the .draggable() modifier.

- SwiftUI Implementation: Use the .draggable() modifier.
- You simply pass it an NSItemProvider that describes your content.

```
import SwiftUI

struct DraggableView: View {
    let image = UIImage(systemName: "photo")!

    var body: some View {
        Image(uiImage: image)
            .resizable()
            .frame(width: 100, height: 100)
            .draggable(NSItemProvider(object: image))
    }
}
```

Drag and Drop

Understanding Enabling Sources and Targets

Enabling Drag Sources

Set up views that initiate drag operations.

Enabling Drop Targets

Define views that accept dropped data efficiently.

Handling Data Types

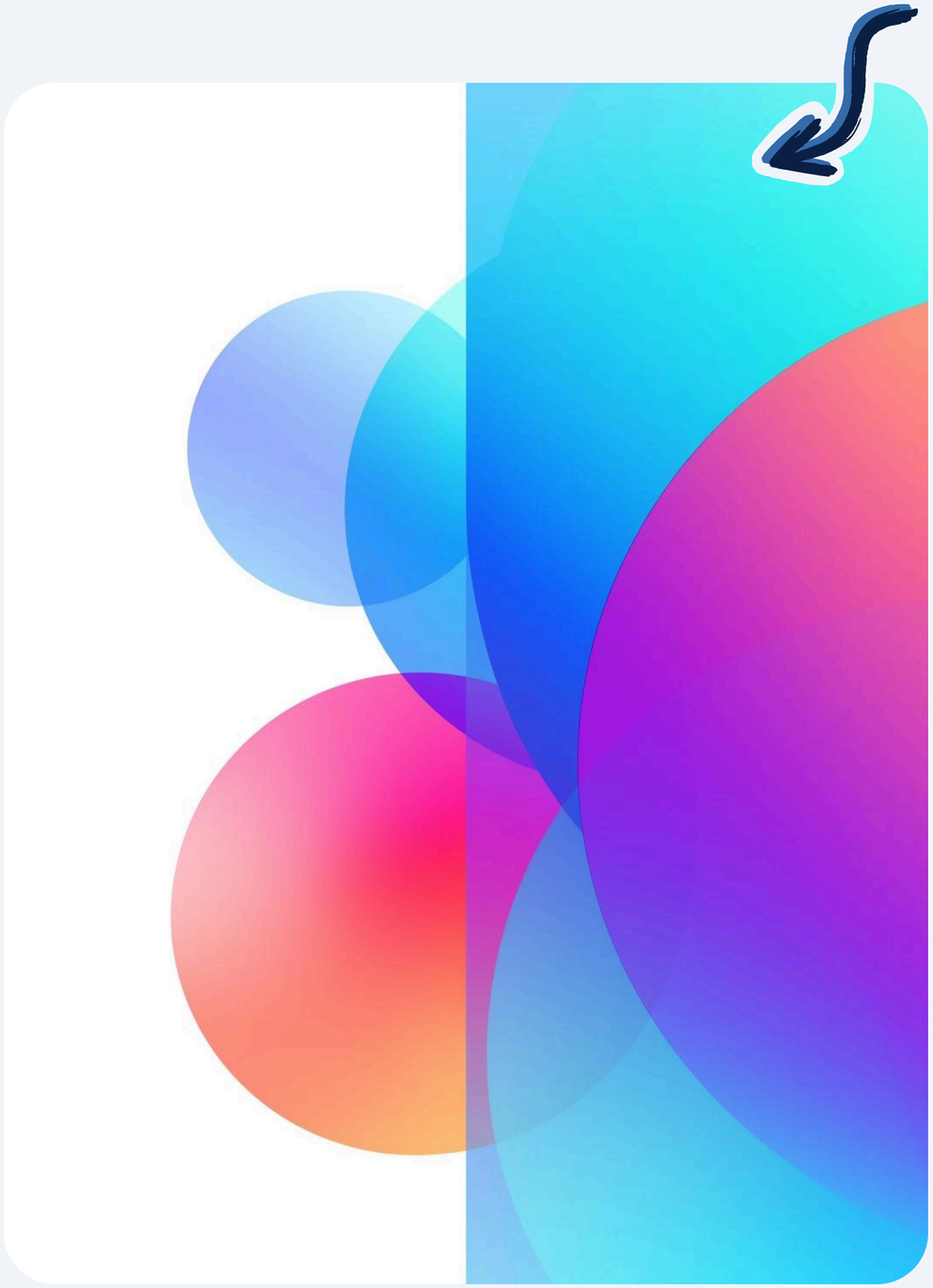
Manage different data formats during transfer seamlessly.

Visual Feedback

Provide users with clear cues during interactions.

Drag and Drop

In SwiftUI, **drag and drop functionality** allows users to easily rearrange lists. This enhances interactivity, making it intuitive for users to organize their data seamlessly.



Best Practices

Strategies for Seamless Input Management

Effective design and coding strategies are essential for creating a **responsive UI** that adapts to various input methods while prioritizing user experience and accessibility in SwiftUI applications.

Managing Inputs

Strategies for optimal user experience

This final module ties everything together into a cohesive philosophy.

- Unifying the Experience (5 mins):
  - Walk through a user story in a hypothetical "document editor" app:
    i. The user drags a photo from the Photos app and drops it into their document.
    ii. They use the trackpad pointer to precisely resize it.
    iii. They grab the Apple Pencil to circle a detail and write a quick annotation.
    iv. Finally, they use the keyboard shortcut ⌘S to save their work.
  - Emphasize: All input methods work in concert. None of them feels tacked on; they all serve a purpose.

Responsive UI

Design must adapt to input types seamlessly.

Prioritizing Inputs

Determine which inputs take precedence in conflicts.

Resolving Conflicts

Create clear user paths for input interactions.

Testing Scenarios

Validate multi-input use cases for smooth functionality.

SwiftUI Environment

Modifiers for Input Detection and Testing

The 4 Rules of Multi-Modal Input (5 mins):

1. **Use System Components First: Standard SwiftUI controls (Button, Toggle, List, TextField) get a huge amount of keyboard, pointer, and accessibility support for free. Always default to them.**
2. **Be Predictable: Follow platform conventions. ⌘C should always copy. Drag and drop should always feel direct and logical. Don't surprise the user.**
3. **Provide Constant Feedback: Use hover effects, pointer shapes, and clear drag previews. The UI should always communicate what's happening and what's possible.**
4. **Enhance, Don't Require: The golden rule. Your app must be 100% usable with only touch. All other input methods are enhancements that make tasks easier or faster. This ensures your app is accessible and works perfectly for every user, no matter how they choose to interact with it.**

Input Detection

Use environment modifiers to identify active inputs.

Dynamic UI

Adapt UI elements to respond to input type changes.

Conflict Resolution

Prioritize inputs to avoid conflicts during usage.

Testing Scenarios

Conduct tests to ensure smooth multi-input functionality.

Time to test our knowledge

ThankYou