iOS 17 App Development Essentials

Neil Smyth

Payload publishing

iOS 17 App Development Essentials

iOS 17 App Development Essentials

ISBN-13: 978-1-951442-80-4

Rev: 1.0

Chapter 21

21. SwiftUI Stacks and Frames

User interface design is primarily a matter of selecting the appropriate interface components, deciding how those views will be positioned on the screen, and then implementing navigation between the different screens and views of the app.

As is to be expected, SwiftUI includes a wide range of user interface components for developing an app such as button, label, slider and toggle views. SwiftUI also provides a set of layout views to define how the user interface is organized and how the layout responds to changes in screen orientation and size.

This chapter will introduce the Stack container views included with SwiftUI and explain how they can be used to create user interface designs with relative ease.

Once stack views have been explained, this chapter will cover the concept of flexible frames and explain how they can be used to control the sizing behavior of views in a layout.

21.1 SwiftUI Stacks

SwiftUI includes three stack layout views in the form of VStack (vertical), HStack (horizontal), and ZStack (views are layered on top of each other).

A stack is declared by embedding child views into a stack view within the SwiftUI View file. In the following view, for example, three Image views have been embedded within an HStack:

```
struct ContentView: View {
    var body: some View {
        HStack {
            Image(systemName: "goforward.10")
            Image(systemName: "goforward.15")
            Image(systemName: "goforward.30")
        }
    }
}
```

Within the preview canvas, the above layout will appear, as illustrated in Figure 21-1:

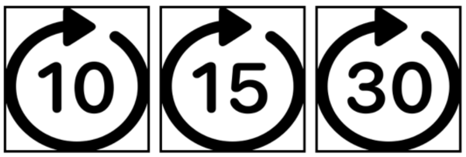

Figure 21-1

A similarly configured example using a VStack would accomplish the same results with the images stacked vertically:

```
VStack {
    Image(systemName: "goforward.10")
    Image(systemName: "goforward.15")
    Image(systemName: "goforward.30")
}
```

To embed an existing component into a stack, either wrap it manually within a stack declaration or right-click on the component in the editor and, from the resulting menu (Figure 21-2), select the appropriate option:

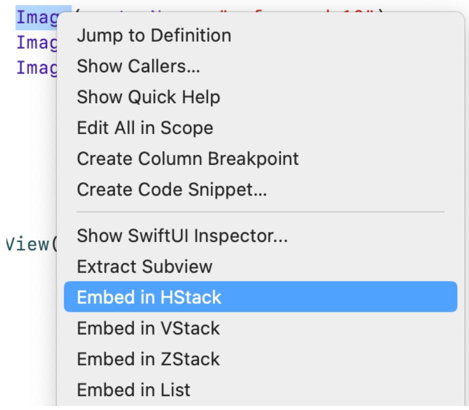

Figure 21-2

Layouts of considerable complexity can be designed simply by embedding stacks within other stacks, for example:

```
VStack {
    Text("Financial Results")
        .font(.title)

    HStack {
        Text("Q1 Sales")
            .font(.headline)

        VStack {
            Text("January")
            Text("February")
            Text("March")
        }

        VStack {
            Text("$1000")
            Text("$200")
            Text("$3000")
        }
    }
}
```

The above layout will appear, as shown in Figure 21-3:

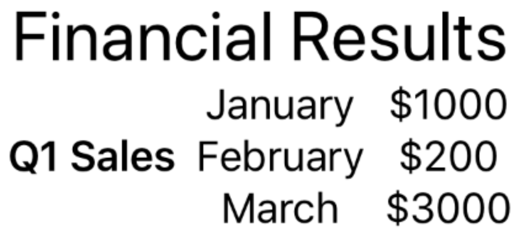

Figure 21-3

As currently configured, the layout needs some additional work, particularly in terms of alignment and spacing. The layout can be improved in this regard using a combination of alignment settings, the Spacer component, and the padding modifier.

21.2 Spacers, Alignment and Padding

To add space between views, SwiftUI includes the Spacer component. When used in a stack layout, the spacer will flexibly expand and contract along the axis of the containing stack (in other words, either horizontally or vertically) to provide a gap between views positioned on either side, for example:

```
HStack(alignment: .top) {
    Text("Q1 Sales")
        .font(.headline)
    Spacer()
    VStack(alignment: .leading) {
        Text("January")
        Text("February")
        Text("March")
    }
    Spacer()
.
.
```

In terms of aligning the content of a stack, this can be achieved by specifying an alignment value when the stack is declared, for example:

```
VStack(alignment: .center) {
            Text("Financial Results")
                .font(.title)
```

Alignments may also be specified with a corresponding spacing value:

```
VStack(alignment: .center, spacing: 15) {
            Text("Financial Results")
                .font(.title)
```

Spacing around the sides of any view may also be implemented using the *padding()* modifier. When called without a parameter, SwiftUI will automatically use the best padding for the layout, content, and screen size (referred to as *adaptable padding*). The following example sets adaptable padding on all four sides of a Text view:

```
Text("Hello, world!")
    .padding()
```

Alternatively, a specific amount of padding may be passed as a parameter to the modifier as follows:

```
Text("Hello, world!")
    .padding(15)
```

Padding may also be applied to a specific side of a view with or without a specific value. In the following example, a specific padding size is applied to the top edge of a Text view:

```
Text("Hello, world!")
    .padding(.top, 10)
```

Making use of these options, the example layout created earlier in the chapter can be modified as follows:

```
VStack(alignment: .center, spacing: 15) {
        Text("Financial Results")
            .font(.title)

        HStack(alignment: .top) {
            Text("Q1 Sales")
                .font(.headline)
            Spacer()
            VStack(alignment: .leading) {
                Text("January")
                Text("February")
                Text("March")
            }
            Spacer()
            VStack(alignment: .leading) {
                Text("$1000")
                Text("$200")
                Text("$3000")
            }
            .padding(5)
        }
        .padding(5)
    }
    .padding(5)
}
```

With the alignments, spacers, and padding modifiers added, the layout should now resemble the following figure:

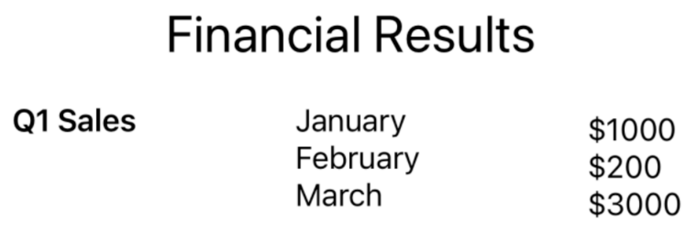

Figure 21-4

A later chapter entitled *"SwiftUI Stack Alignment and Alignment Guides"* will cover more advanced stack alignment topics.

21.3 Grouping Views

The Group component groups together multiple views and is useful when performing an operation on multiple views (for example, a set of related views can all be hidden in a single operation by embedding them in a Group and hiding that view):

```
Group {
    Text("$1000")
    Text("$200")
    Text("$3000")
}
```

21.4 Dynamic HStack and VStack Conversion

The choice between an HStack and a VStack during the design phase does not need to be final. SwiftUI allows the type of stack used in a layout to be changed dynamically from within the app code. We achieve this by creating an AnyLayout instance and assigning it either an HStackLayout or VStackLayout container. In the example below, we have assigned an AnyLayout instance to a state variable and configured it to stack children using the HStackLayout container. We then use this layout to arrange two Image views. Finally, we add two Button views to switch between horizontal and vertical orientation by changing the container type assigned to myLayout:

```
@State var myLayout: AnyLayout = AnyLayout(VStackLayout())

var body: some View {

    VStack {
        myLayout {
            Image(systemName: "goforward.10")
                .resizable()
                .aspectRatio(contentMode: .fit)
            Image(systemName: "goforward.15")
                .resizable()
                .aspectRatio(contentMode: .fit)
        }

        HStack {
            Button(action: {
                myLayout = AnyLayout(HStackLayout()) }){
                Text("HStack")
            }
            Button(action: {
                myLayout = AnyLayout(VStackLayout()) }) {
                Text("VStack")
            }
        }
    }
}
```

21.5 Text Line Limits and Layout Priority

By default, an HStack will attempt to display the text within its Text view children on a single line. Take, for example, the following HStack declaration containing an Image view and two Text views:

```
HStack {
    Image(systemName: "airplane")
    Text("Flight times:")
    Text("London")
}
.font(.largeTitle)
```

If the stack has enough room, the above layout will appear as follows:

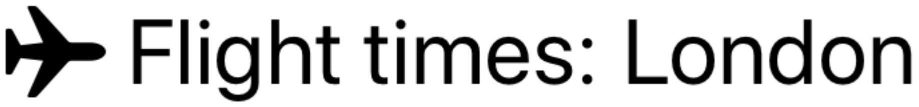

Figure 21-5

If a stack has insufficient room (for example if it is constrained by a frame or is competing for space with sibling views) the text will automatically wrap onto multiple lines when necessary:

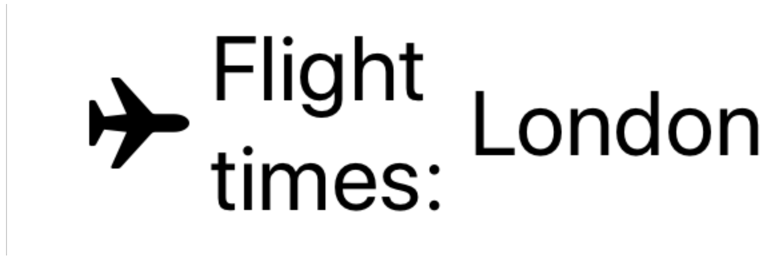

Figure 21-6

While this may work for some situations, it may become an issue if the user interface is required to display this text in a single line. The number of lines over which text can flow can be restricted using the *lineCount()* modifier. The example HStack could, therefore, be limited to 1 line of text with the following change:

```
HStack {
    Image(systemName: "airplane")
    Text("Flight times:")
    Text("London")
}
.font(.largeTitle)
.lineLimit(1)
```

A line limit restriction can also be specified as a range that provides the maximum and minimum number of lines over which text can be displayed:

```
.lineLimit(1...4)
```

When an HStack has insufficient space to display the full text and is not permitted to wrap the text over enough lines, the view will resort to truncating the text, as is the case in Figure 21-7:

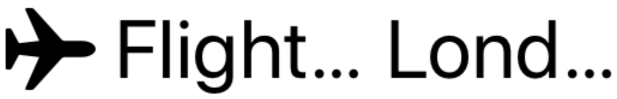

Figure 21-7

Without any priority guidance, the stack view will decide how to truncate the Text views based on the available space and the length of the views. Obviously, the stack can only know whether the text in one view is more important than the text in another if the text view declarations include some priority information. This is achieved by making use of the *layoutPriority()* modifier. This modifier can be added to the views in the stack and passed values indicating the priority level for the corresponding view. The higher the number, the greater the layout priority, and the less the view will be truncated.

Assuming the flight destination city name is more important than the "Flight times:" text, the example stack could be modified as follows:

```
HStack {
    Image(systemName: "airplane")
    Text("Flight times:")
    Text("London")
        .layoutPriority(1)
}
.font(.largeTitle)
.lineLimit(1)
```

With a higher priority assigned to the city Text view (in the absence of a layout priority, the other text view defaults to a priority of 0), the layout will now appear, as illustrated in Figure 21-8:

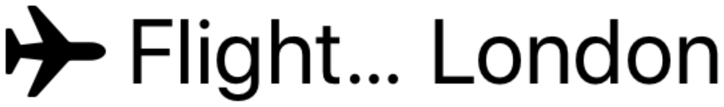

Figure 21-8

21.6 Traditional vs. Lazy Stacks

So far in this chapter, we have only covered the HStack, VStack, and ZStack views. Although the stack examples shown so far contain relatively few child views, it is possible for a stack to contain large quantities of views. This is particularly common when a stack is embedded in a ScrollView. ScrollView is a view that allows the user to scroll through content that extends beyond the visible area of either the containing view or the device screen.

When using the traditional HStack and VStack views, the system will create all child views at initialization, regardless of whether those views are currently visible to the user. While this may not be an issue for most requirements, this can lead to performance degradation when a stack has thousands of child views.

SwiftUI also provides "lazy" vertical and horizontal stack views to address this issue. These views (named LazyVStack and LazyHStack) use the same declaration syntax as the traditional stack views but are designed only to create child views as needed. For example, as the user scrolls through a stack, views that are currently off-screen will only be created once they approach the point of becoming visible to the user. Once those views pass out of the viewing area, SwiftUI releases them so they no longer take up system resources.

When deciding whether to use traditional or lazy stacks, start using the traditional stacks and switch to lazy stacks if you encounter performance issues relating to a high number of child views.

21.7 SwiftUI Frames

By default, a view will be sized automatically based on its content and the requirements of any layout in which it may be embedded. Although much can be achieved using the stack layouts to control the size and positioning of a view, sometimes a view is required to be a specific size or to fit within a range of size dimensions. To address this need, SwiftUI includes the flexible frame modifier.

Consider the following Text view, which has been modified to display a border:

```
Text("Hello World")
    .font(.largeTitle)
    .border(Color.black)
```

Within the preview canvas, the above text view will appear as follows:

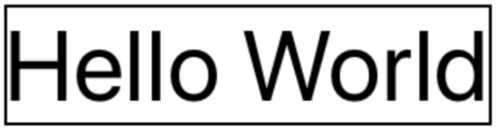

Figure 21-9

Without a frame, the text view has been sized to accommodate its content. If the Text view was required to have height and width dimensions of 100, however, a frame could be applied as follows:

```
Text("Hello World")
    .font(.largeTitle)
    .border(Color.black)
    .frame(width: 100, height: 100, alignment: .center)
```

Now that the Text view is constrained within a frame, the view will appear as follows:

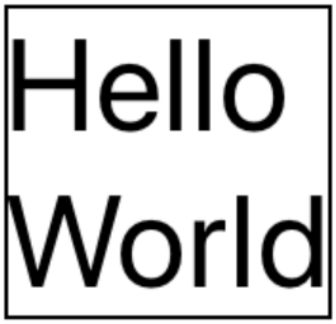

Figure 21-10

In many cases, fixed dimensions will provide the required behavior. In other cases, such as when the content of a view changes dynamically, this can cause problems. Increasing the length of the text, for example, might cause the content to be truncated:

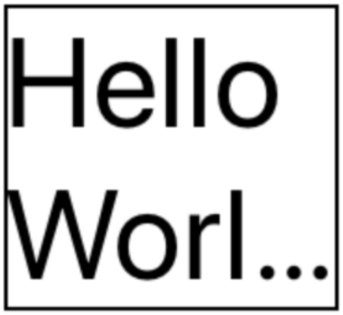

Figure 21-11

This can be resolved by creating a frame with minimum and maximum dimensions:

```
Text("Hello World, how are you?")
            .font(.largeTitle)
            .border(Color.black)
            .frame(minWidth: 100, maxWidth: 300, minHeight: 100,
                   maxHeight: 100, alignment: .center)
```

Now that the frame has some flexibility, the view will be sized to accommodate the content within the defined minimum and maximum limits. When the text is short enough, the view will appear, as shown in Figure 21-10 above. Longer text, however, will be displayed as follows:

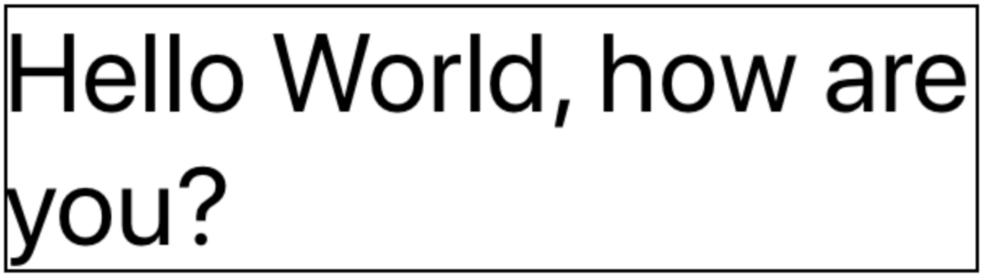

Figure 21-12

Frames may also be configured to take up all the available space by setting the minimum and maximum values to 0 and infinity respectively:

```
.frame(minWidth: 0, maxWidth: .infinity, minHeight: 0,
        maxHeight: .infinity)
```

Remember that the order in which modifiers are chained often impacts the appearance of a view. In this case, if the border is to be drawn at the edges of the available space it will need to be applied to the frame:

```
Text("Hello World, how are you?")
    .font(.largeTitle)
    .frame(minWidth: 0, maxWidth: .infinity, minHeight: 0,
          maxHeight: .infinity)
    .border(Color.black, width: 5)
```

By default, the frame will honor the safe areas on the screen when filling the display. Areas considered outside the safe area include the camera notch on some device models and the bar across the top of the screen displaying the time and WiFi and cellular signal strength icons. To configure the frame to extend beyond the safe area, simply use the *edgesIgnoringSafeArea()* modifier, specifying the safe area edges to ignore:

```
.edgesIgnoringSafeArea(.all)
```

21.8 Frames and the Geometry Reader

Frames can also be implemented so that they are sized relative to the size of the container within which the corresponding view is embedded. This is achieved by wrapping the view in a GeometryReader and using the reader to identify the container dimensions. These dimensions can then be used to calculate the frame size. The following example uses a frame to set the dimensions of two Text views relative to the size of the containing VStack:

```
GeometryReader { geometry in
    VStack {
        Text("Hello World, how are you?")
            .font(.largeTitle)
            .frame(width: geometry.size.width / 2,
                height: (geometry.size.height / 4) * 3)
        Text("Goodbye World")
            .font(.largeTitle)
            .frame(width: geometry.size.width / 3,
                height: geometry.size.height / 4)
    }
}
```

The topmost Text view is configured to occupy half the width and three-quarters of the height of the VStack, while the lower Text view occupies one third of the width and one-quarter of the height.

21.9 Summary

User interface design mainly involves gathering components and laying them out on the screen in a way that provides a pleasant and intuitive user experience. User interface layouts must also be responsive so that they appear correctly on any device regardless of screen size and, ideally, device orientation. To ease the process of user interface layout design, SwiftUI provides several layout views and components. In this chapter we have looked at layout stack views and the flexible frame.

By default, a view will be sized according to its content and the restrictions imposed on it by any view in which it may be contained. When insufficient space is available, a view may be restricted in size, resulting in truncated content. Priority settings can be used to control how much views are reduced in size relative to container sibling views.

For greater control of the space allocated to a view, a flexible frame can be applied to the view. The frame can be fixed in size, constrained within a range of minimum and maximum values, or, using a Geometry Reader, sized relative to the containing view.