Frame and layouts examples

Overview

In the previous video, you learned about the important tools for modifying the layout of your user interface. In this reading, you'll work through examples of using those tools. By the end, you will understand when to consider modifying a view layout. You will be also able to identify the frame of a view element and explain the different tools.

To try out these examples, create a SwiftUI app project in Xcode. Set the target to your favorite iPhone and open **`ContentView.swift`**. The code that defines the content view will be something like this:

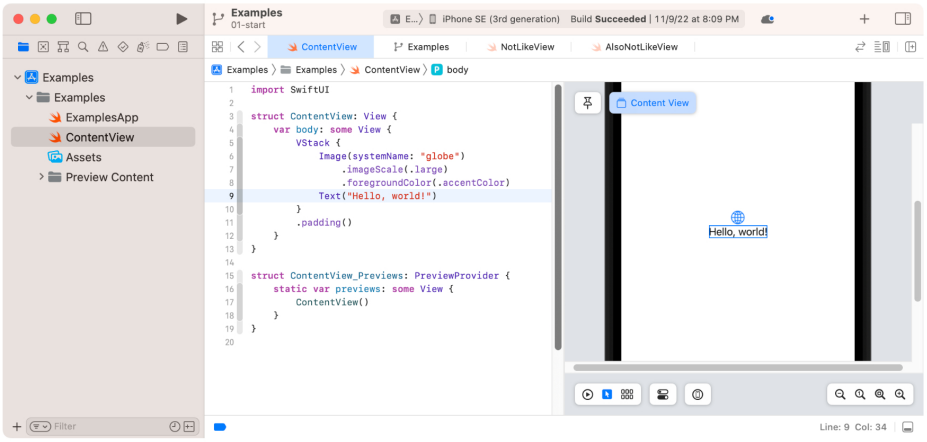
```
struct ContentView: View {
    var body: some View {
        VStack {
            Image(systemName: "globe")
                .imageScale(.large)
                .foregroundColor(.accentColor)
            Text("Hello, world!")
        }
        .padding()
    }
}
```

The examples that follow use or modify the code inside the **`VStack`**.

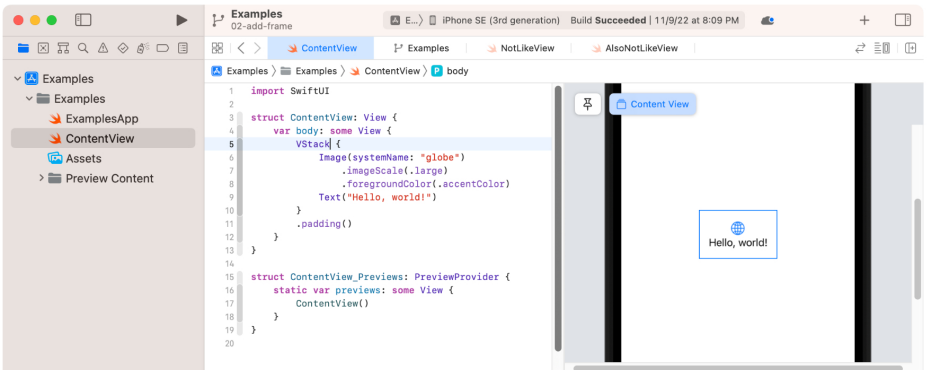
Frame, the basic unit of layouts

Every view that is displayed on the screen is a rectangle. Views that don't appear to be a rectangle are drawn in a rectangle. This rectangle is called the frame of the view.

In the Xcode project you created, set the mode of the Canvas to Selectable (the middle of the three buttons on the lower left). Now select the line with the **`TextView`** in the code editor. The Canvas displays a box around the text. This box is the frame of the **`TextView`**.



Select the **`VStack`** and now the rectangle encloses the image and **`TextView`**.

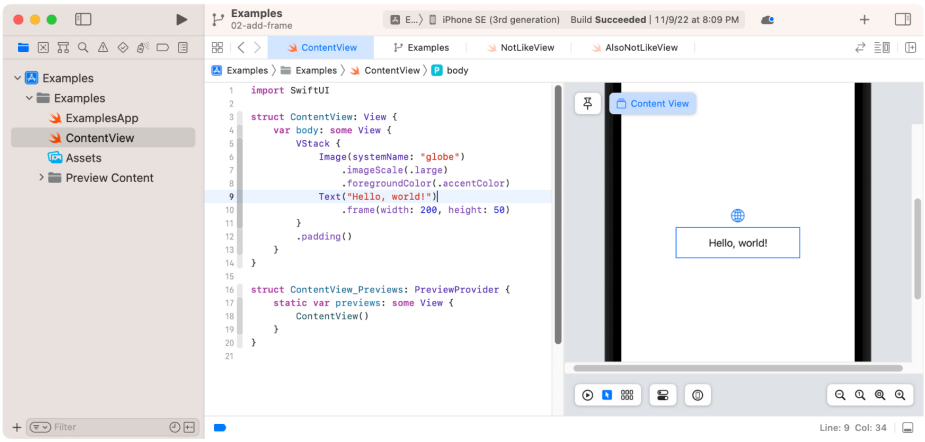


Setting the frame

Now you'll set the frame for the `TextView`. Change the code as follows:

```
1   Text("Hello, world!")
2       .frame(width: 200, height: 50)
```

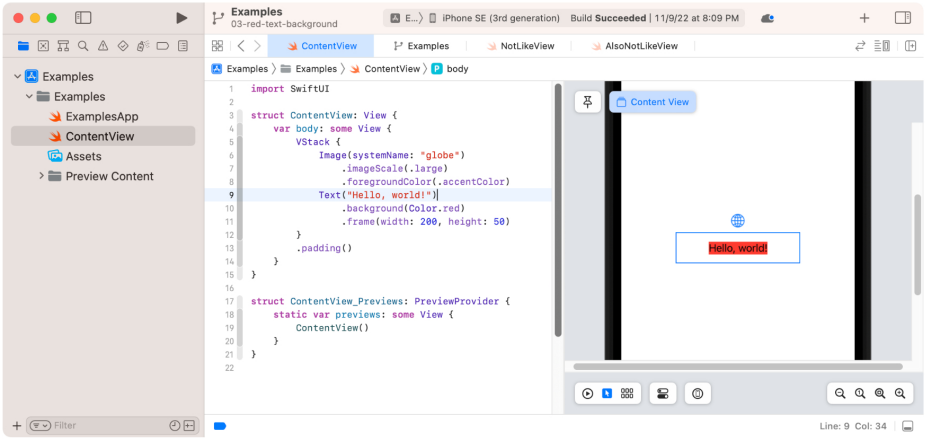
The frame for the `Text` appears to change on the canvas, but that's not what is really happening.



Set the background color of the `Text` to red:

```
1   Text("Hello, world!")
2       .background(Color.red)
3       .frame(width: 200, height: 50)
```

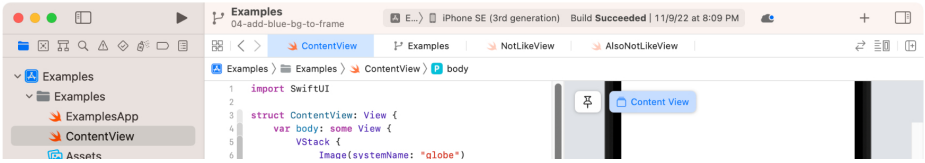
The red background is displayed only in the original `Text` frame.



Now add a blue background to the `.frame()` modifier:

```
1   Text("Hello, world!")
2       .background(Color.red)
3       .frame(width: 200, height: 50)
4       .background(Color.blue)
```

The `.frame()` modifier creates a new invisible view with the specified size. The `TextView` is inside that new view.



When to modify your layout

You may be wondering if all these invisible views can slow your app down. The basic answer is no. SwiftUI is very efficient when working with views, including the invisible ones for view modifiers.

However, SwiftUI is very good at laying out an interface on many different devices and platforms. The best idea is to use a modification only when it's important. For example, a logo may need to be a certain size or you want a portion of the app to display a certain way.

Start with no modifications. Add them when you need them. When you do add them, preview them on any device that your app runs on.

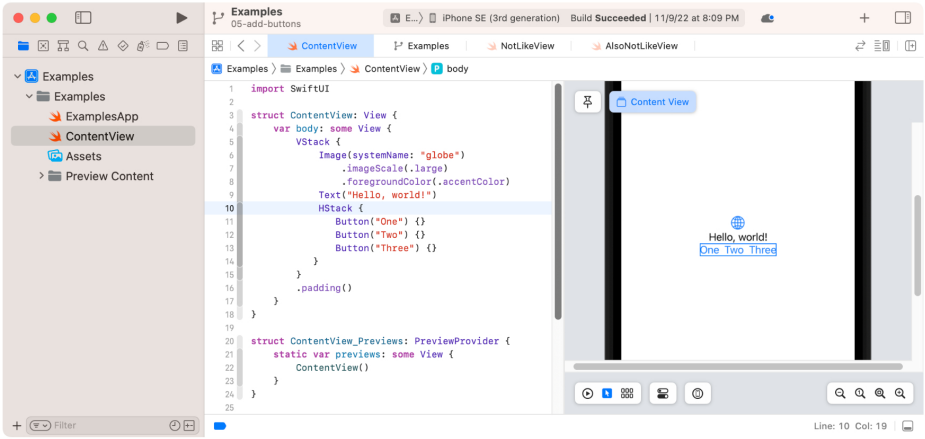
Organizing groups of views

You can think of your interface as different groups of related views. For example, the stopwatch in the Apple Clock app has two vertical groups. In the digital version, the first is a horizontal group that displays the time. Next is a horizontal group with a lap button, a control to switch between the digital and analog versions and another button to start and stop timing.

The code you've been working with is a vertical group, a **`VStack`**. For a horizontal group, you use **`HStack`**. The code that follows is a horizontal group of three buttons.

```
HStack {
    Button("One") {}
    Button("Two") {}
    Button("Three") {}
}
```

Remove the background colors and frame from the **`Text`**. Now add the **`HStack`** with the buttons below the **`Text`**.



The current code describes a view that is grouped in the vertical direction. It contains three elements: an image, some text and a horizontal group of buttons.

Designing your interface as groups of horizontal and vertical views makes designing for SwiftUI easier.

Adding space

The **`Text`** and the buttons of the **`HStack`** are too close together. A **`Spacer`** adds space. Where that space is added depends on the direction of the views in the group. For example, if you add a **`Spacer`** between the first and second buttons, it adds space in the horizontal direction.

To add space in the vertical direction, add the **`Spacer`** between the **`Text`** and the **`HStack`** with the buttons:

```
Text("Hello, world!")
Spacer()
HStack {
    Button("One") {}
```

```
    Button("Two") {}
    Button("Three") {}
}
```

Now the buttons are at the bottom of the screen and the `Text` is at the top.

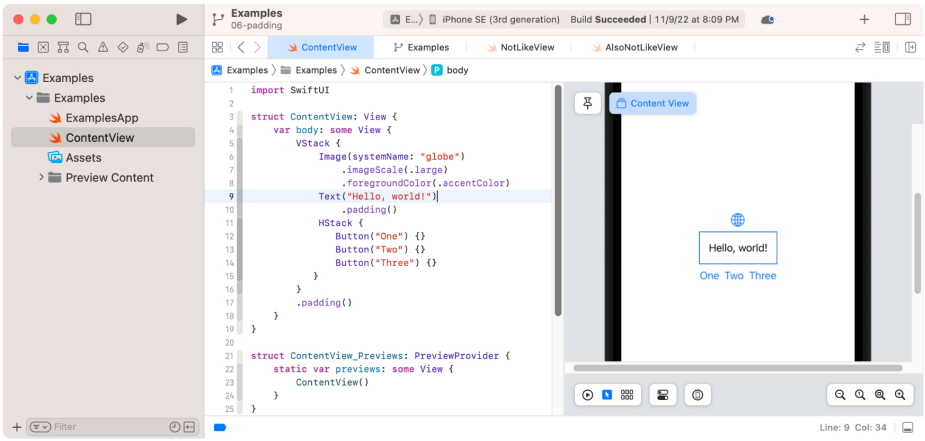
Adding padding

Sometimes a `Spacer` adds too much space. The `Spacer` between the `Text` and button group pushed the buttons to the bottom of the screen. It also moved the `Text` and image to the top .Sometimes you want less space between views. The most common reason is matching the Apple Human Interface Guidelines for spacing.

For this, you can use the `.padding()` modifier. By default, this adds the system-defined amount of space on all four sides of a view. It's also possible to choose which sides and how much space.

Remove the `Spacer` between the `Text` and button group. Now, add padding to the `Text`:

```
Text("Hello, world!")
    .padding()
HStack {
    Button("One") {}
    Button("Two") {}
    Button("Three") {}
}
```



Now there is the recommended space between the bottom of the `Text` view and the top of the button group. There is also space between the top of the `Text` view and the image.

Alignment

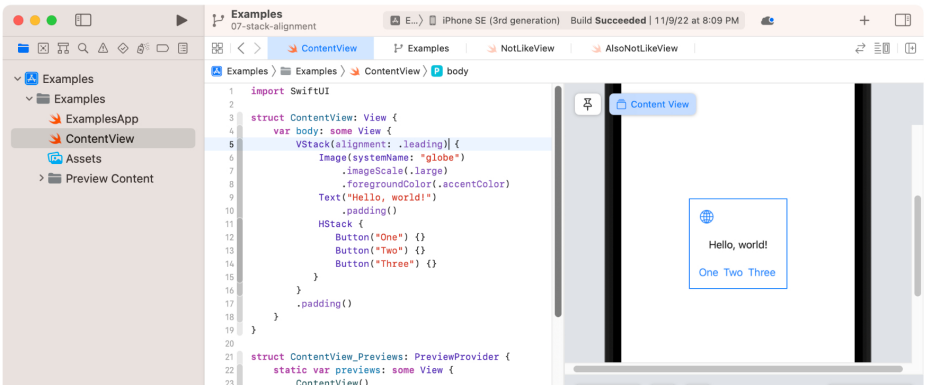
The current interface is centered on the iPhone screen. Sometimes you want your views to align in different ways.

There are two main ways to change the alignment.

The first uses the Stack views alignment argument. A `VStack` arranges its child views vertically but aligns them in the horizontal direction. Similarly, an `HStack` arranges its child views horizontally, but aligns them in a vertical direction.

Change the `VStack` so it aligns items to its leading edge:

```
VStack(alignment: .leading) {
```



Although it looks like the text and buttons are still centered, that's not really the case. Examine the frames for the **`Text`** and for the **`HStack`** in the Canvas. They are both as wide as the **`VStack`**.

Add a .frame modifier to the **`VStack`** with a width of 300.

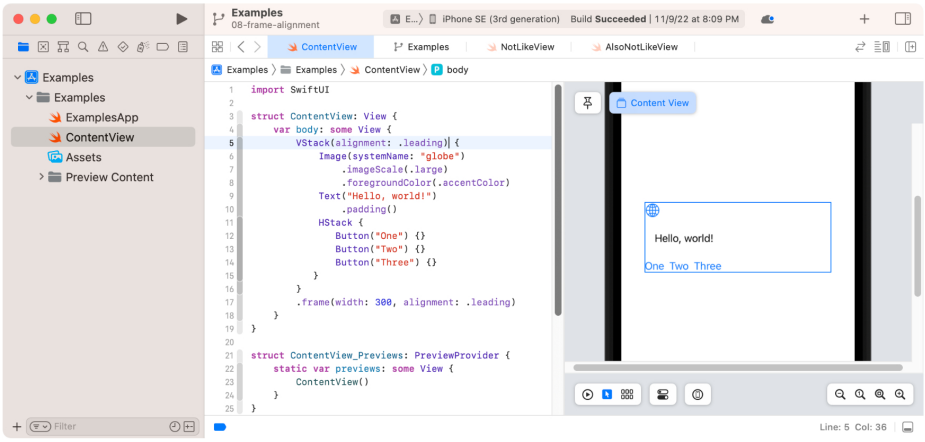
```
struct ContentView: View {
    var body: some View {
        VStack(alignment: .leading) {
            …
        }
        .frame(width: 300)
    }
}
```

The views inside the **`VStack`** don't change position. The **`.frame`** modifier added a new invisible view.

Now you can try the second way to change alignment. The frame modifier also takes an argument for alignment.

Add a leading alignment to the **`VStack`** frame:

```
.frame(width: 300, alignment: .leading)
```



There are many different types of alignment including leading, trailing, top, center and so on. Use the code you've created to experiment with the different types. Try using them in the stacks and in frames. Try adding frames with different alignments.

Conclusion

In this reading, you explored tools you can use to modify the layout of your interface. You learned that a frame is just another invisible view and how to identify one on the XCode Canvas. You also learned when to modify your layout. Plus, you experimented with stacks of views, tried adding space with the **`Spacer`** and then with padding and, finally, you explored two ways to align views.

Mark as completed

Like Dislike Report an issue