SwiftUI 레이아웃과 뷰 구성

기본적인 SwiftUI의 구조

ContentView의 Bound는 Body(여기서는 Text)에 의해 결정됨
body와 항상 동일한 bounds를 갖는 ContentView가 존재함.
최 상단에 RootView가 존재하며, RootView의 경우 Safe Area를 제외한 나머지 영역을 의미함.
edgesIgnoringSafeArea같은 modifier를 활용하면 RootView도 Safe Area에 포함될수 있으나,
기본적으로는 safe area을 제외한 영역을 의미함.

레이아웃 프로세스

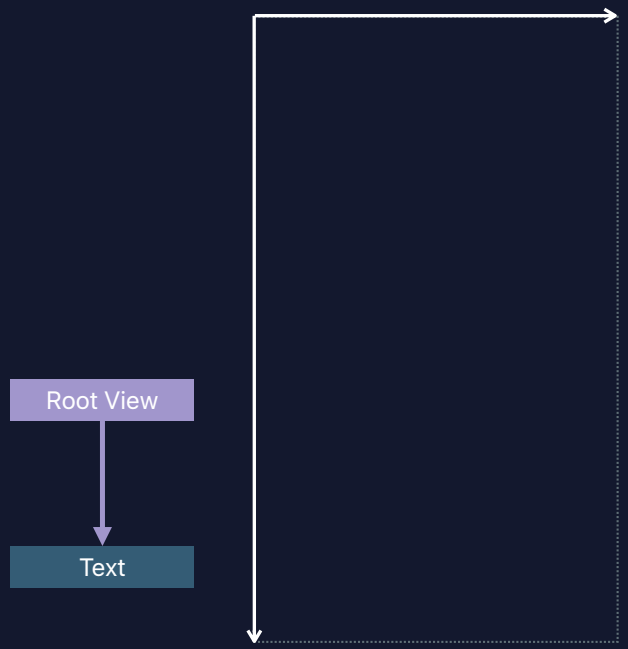
Layout Procedure

1. Parent proposes a size for child
2. Child chooses its own size
3. Parent places child in parent’s coordinate space

1. Parent가 Child에게 사이즈 먼저 제안

1. Parent Proposes Size for Child

```
struct ContentView : View {
    var body: some View {
        Text("Hello World")
    }
}
```



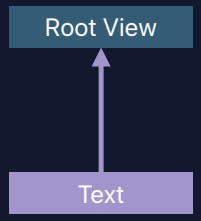
child에게 화살표 크기만큼 사이즈를 제안

2. Child

2. Child Chooses Its Own Size

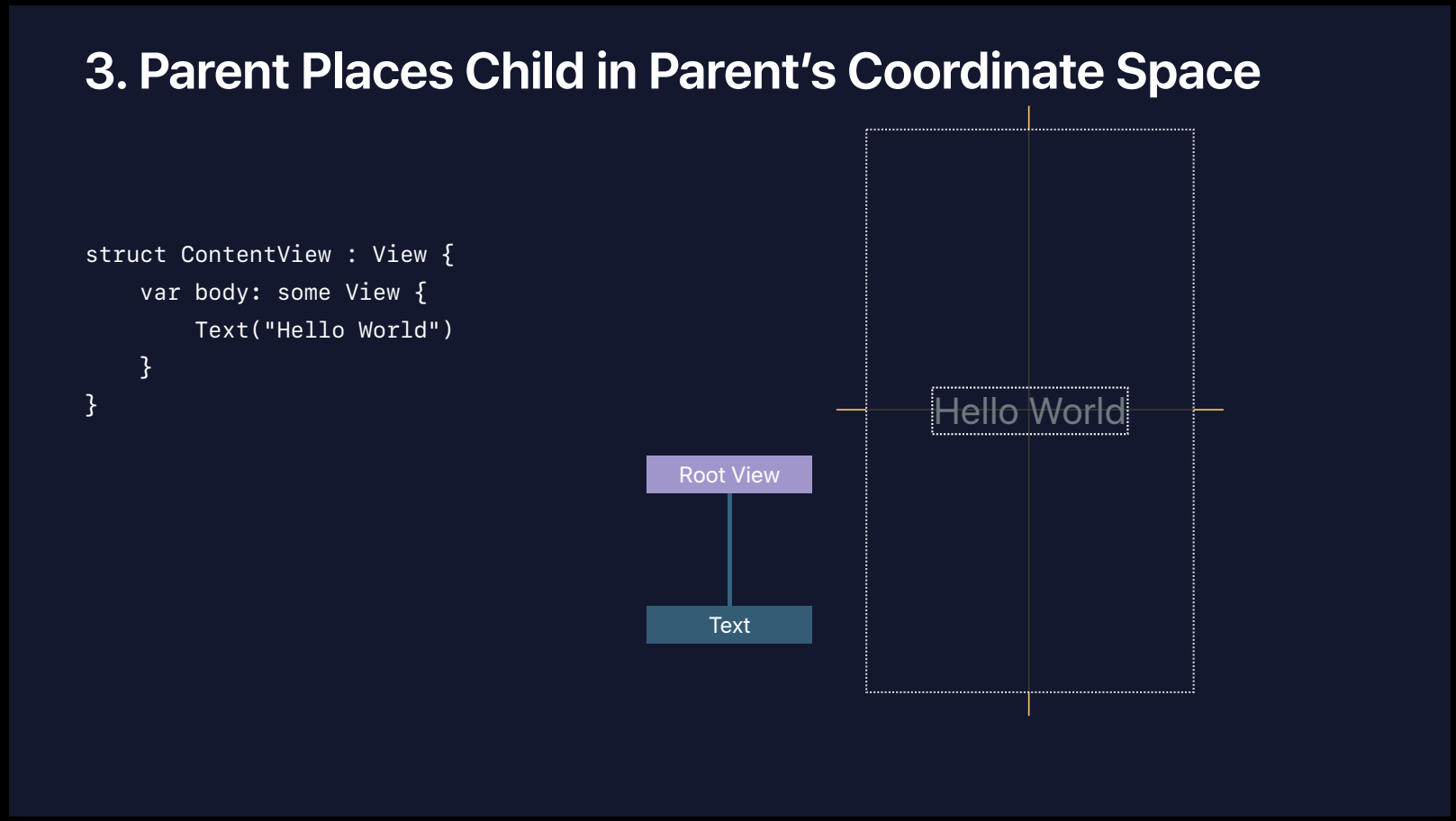
```
struct ContentView : View {
    var body: some View {
        Text("Hello World")
    }
}
```

Hello World



하지만 Text의 경우 Hello World만큼만 필요하기 때문에 제안된 사이즈 대신 텍스트 영역만큼의 사이즈를 사용하기로 결정함.

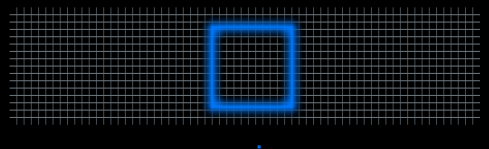
3. 결정된 사이즈를 기반으로 부모가 자식을 부모의 좌표 공간에 배치함



이 과정에서 부모는 자식의 크기 결정을 존중해야한다. 스유에서는 UIKit과 다르게 상위에서 하위 뷰나 레이어 계층에 사이즈를 강제할수 없음

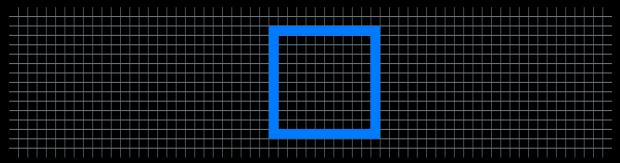
Layout Procedure

1. Parent proposes a size for child
2. Child chooses its own size
3. Parent places child in parent's coordinate space
4. SwiftUI rounds coordinates to nearest pixel



Layout Procedure

1. Parent proposes a size for child
2. Child chooses its own size
3. Parent places child in parent's coordinate space
4. SwiftUI rounds coordinates to nearest pixel



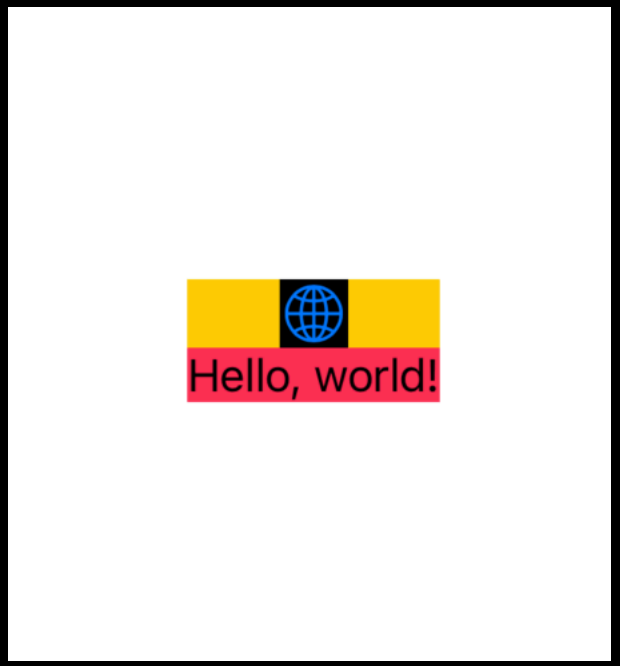
뷰의 레이아웃을 관리하면서 뷰의 모서리가 흐릿하게 처리되는 경우가 생길수 있으나, 스유는 뷰의 모서리를 가장 가까운 pixel로 반올림 함

Stack(VStack/HStack/ZStack)

SwiftUI에서 여러 뷰를 정렬하기 위한 컨테이너.
각 스택의 붙는 대문자는 (Vertical, Horizontal, Z축)을 의미한다.

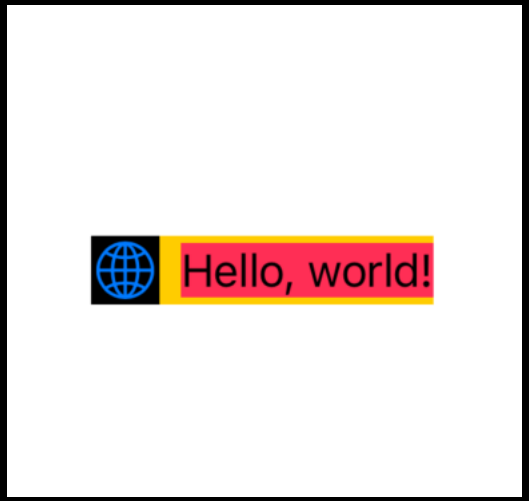
- VStack: 수직 정렬 (세로 방향)

```
struct ContentView: View {
    var body: some View {
        VStack {
            Image(systemName: "globe")
                .imageScale(.large)
                .foregroundStyle(.tint)
                .background(.black)
            Text("Hello, world!")
                .background(.pink)
        }
        .background(.yellow)
    }
}
```



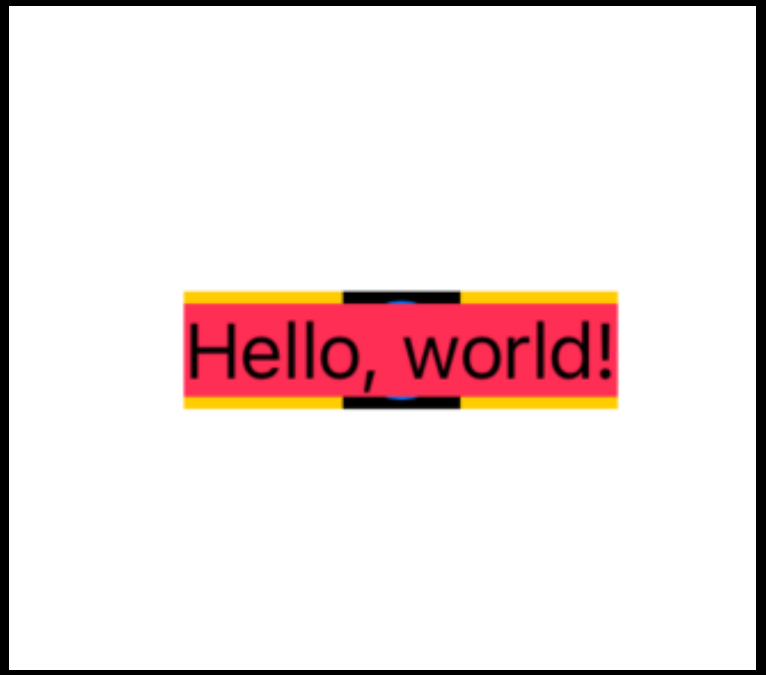
- HStack: 수평 정렬 (가로 방향)

```
9
10  struct ContentView: View {
11      var body: some View {
12          HStack {
13              Image(systemName: "globe")
14                  .imageScale(.large)
15                  .foregroundStyle(.tint)
16                  .background(.black)
17              Text("Hello, world!")
18                  .background(.pink)
19          }
20          .background(.yellow)
21      }
22  }
```



• ZStack: 겹쳐 쌓기 (z축 방향)

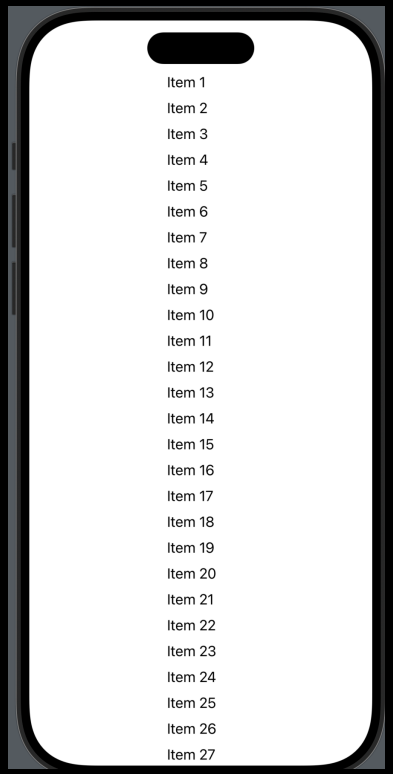
```
struct ContentView: View {
    var body: some View {
        ZStack {
            Image(systemName: "globe")
                .imageScale(.large)
                .foregroundStyle(.tint)
                .background(.black)
            Text("Hello, world!")
                .background(.pink)
        }
        .background(.yellow)
    }
}
```



+ 추가적으로 Lazy한 스택들도 3가지가 존재한다.
+ Lazy 키워드가 지연된다는 뜻은 알겠는데, 정확히 뭘 의미하고, 일반 스택과 어떤 차이가 있을까?

스크롤뷰를 사용해서 보면 확연한 차이를 알수 있음.
공식문서에는 필요한 항목만 생성하는 뷰라고 나온다. 이와 곁들어서 VStack의 공식문서 내용을 확인해보면,
 일반적인 스택은 화면에 노출되지 않아도 하위 뷰를 한번에 렌더링하기 때문에, 하위뷰의 갯수가 적을경우 일반 Stack을 사용하고, 많을경우 LazyStack을 사용하라고 설명하고 있음.

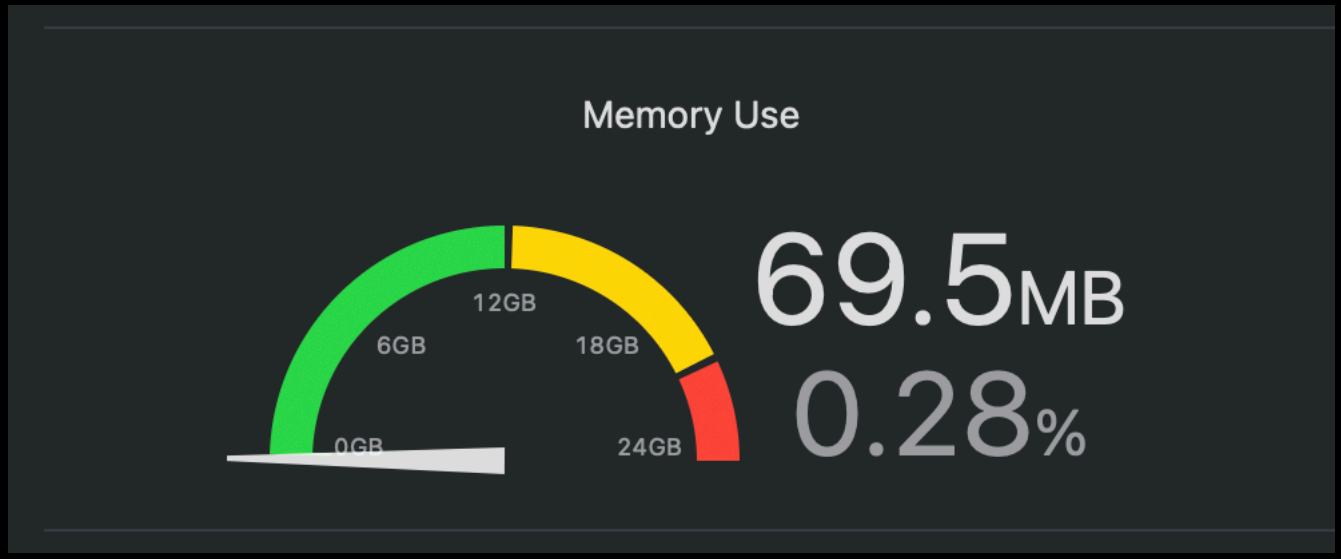
```
struct ContentView: View {
    var body: some View {
        ScrollView {
            VStack(
                alignment: .leading,
                spacing: 10
            ) {
                ForEach(
                    1...1000,
                    id: \.self
                ) { item in
                    Text("Item \(item)")
                        .onAppear {
                            print("Item \(item) appeared")
                        }
                }
            }
        }
    }
}

#Preview {
```



- 간단하게 1000개의 데이터를 생성했다 가정하고 비교를 해보면 명확하게 차이를 알수 있음

VStack
모든 아이템이 출력됨. 측 처음에 모든 Text가 렌더링되었다는것을 의미함.

```
Item 50 appeared
Item 49 appeared
Item 48 appeared
Item 47 appeared
Item 46 appeared
Item 45 appeared
Item 44 appeared
Item 43 appeared
Item 42 appeared
Item 41 appeared
Item 40 appeared
Item 39 appeared
Item 38 appeared
Item 37 appeared
Item 36 appeared
Item 35 appeared
Item 34 appeared
Item 33 appeared
Item 32 appeared
Item 31 appeared
Item 30 appeared
Item 29 appeared
Item 28 appeared
Item 27 appeared
Item 26 appeared
Item 25 appeared
Item 24 appeared
Item 23 appeared
Item 22 appeared
Item 21 appeared
Item 20 appeared
Item 19 appeared
Item 18 appeared
Item 17 appeared
Item 16 appeared
Item 15 appeared
Item 14 appeared
Item 13 appeared
Item 12 appeared
Item 11 appeared
Item 10 appeared
Item 9 appeared
Item 8 appeared
Item 7 appeared
Item 6 appeared
Item 5 appeared
Item 4 appeared
Item 3 appeared
Item 2 appeared
Item 1 appeared
```



LazyVStack

- 화면에 보이는 뷰들만 렌더링되어 표시됨.
- 확실이 모든 화면이 렌더링되지 않기 때문에 메모리 사용량 차이가 발생하는 모습 확인 가능

```
Item 19 appeared
Item 6 appeared
Item 2 appeared
Item 9 appeared
Item 3 appeared
Item 4 appeared
Item 5 appeared
Item 27 appeared
Item 10 appeared
Item 21 appeared
Item 24 appeared
Item 12 appeared
Item 8 appeared
Item 25 appeared
Item 22 appeared
Item 7 appeared
Item 15 appeared
Item 13 appeared
Item 26 appeared
Item 16 appeared
Item 23 appeared
Item 18 appeared
Item 14 appeared
Item 1 appeared
Item 11 appeared
Item 17 appeared
Item 20 appeared
```