

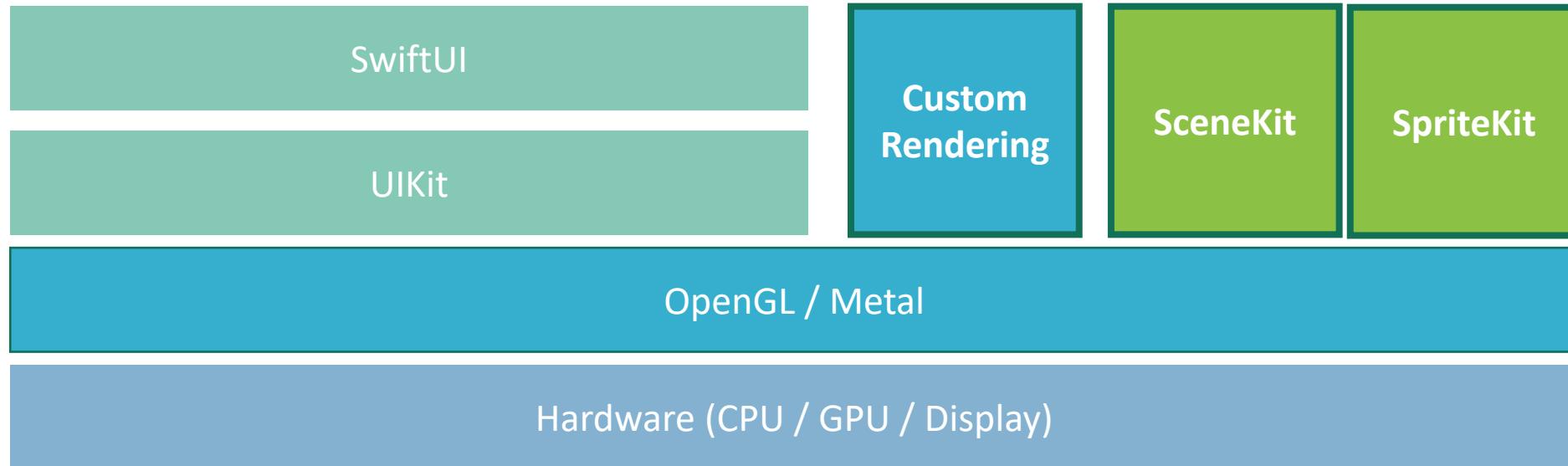
Graphics

Programmieren für iOS



iOS Graphics

- Bisher: Bildschirm mit UI-Elementen (SwiftUI / UIKit) füllen
- Heute: Wie kommen die Pixel wirklich auf den Screen (stark vereinfacht)



Low-Level Graphics

- Heute: Wie befülle ich einen Bereich im Speicher («Buffer») mit Zahlen zwischen 0 und 1, damit es schöne Farben gibt?
- Annahme: Hardware stellt die Werte dann irgendwie dar.



0.8	0.3	0.8	1.0	0.9	0.7	1.0	1.0
0.8	0.3	0.8	1.0	0.9	0.7	1.0	1.0
0.8	0.3	0.8	1.0	0.9	0.7	1.0	1.0
0.0	0.9	0.9	1.0	0.9	0.7	1.0	1.0
0.0	0.9	0.9	1.0	0.9	0.7	1.0	1.0
0.0	0.9	0.9	1.0	0.9	0.7	1.0	1.0
0.0	0.9	0.9	1.0	0.9	0.9	0.3	1.0
0.0	0.9	0.9	1.0	0.9	0.9	0.3	1.0

CoreGraphics

- Buffer (Arrays) können mit normalem Swift-Code gefüllt werden.
- Das Framework **CoreGraphics** bietet viele nützliche Vereinfachungen.
- Aber: Beides ist für grosse Bilder oder hohe Framerates **zu langsam**.
- Lösung: GPU!

```
let buffer = ... // initialize memory

for i in 0 ..< UIScreen.main.size.width {
    for j in 0 ..< UIScreen.main.size.height {
        buffer.setColor(i,j, computeColor(i, j))
    }
}

let img = buffer.toImage()
```

```
let renderer = UIGraphicsImageRenderer(size: UIScreen.main.size)

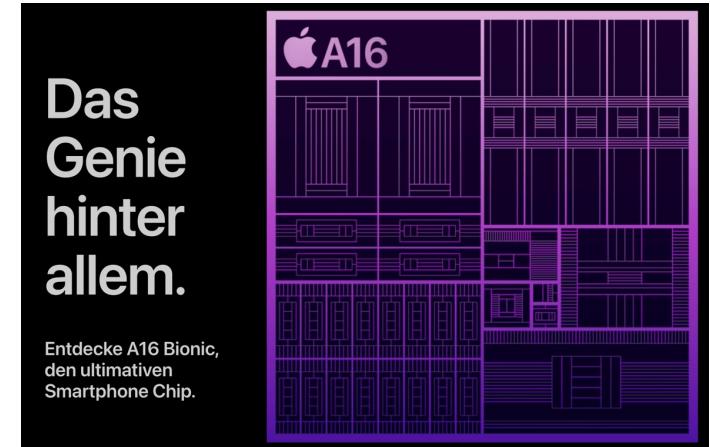
let img = renderer.image { ctx in
    ctx.cgContext.move(to: CGPoint(x: 20.0, y: 20.0))
    ctx.cgContext.addLine(to: CGPoint(x: 260.0, y: 230.0))
    ctx.cgContext.addLine(to: CGPoint(x: 100.0, y: 200.0))
    ctx.cgContext.addLine(to: CGPoint(x: 20.0, y: 20.0))

    ctx.cgContext.setLineWidth(10)
    ctx.cgContext.setStrokeColor(UIColor.black.cgColor)

    ctx.cgContext.strokePath()
}
```

CPU vs. GPU

- Obwohl alles auf einem Chip zusammengeführt wird, haben **CPU** (Central Processing Unit) und **GPU** (graphics processing unit) deutlich andere Fähigkeiten und Aufgaben.
- Die Architektur von GPUs ist darauf ausgelegt, viele einfache Operationen schnell und parallel auszuführen.
- A15 GPU: 1224 GFLOPS
 - 1 GFLOPS = 1 Milliarde Fließkomma-Operationen pro Sekunde
- A15 CPU: 2 x 3 GHz (2 High-Performance Cores)
 - + Low-Performance Cores

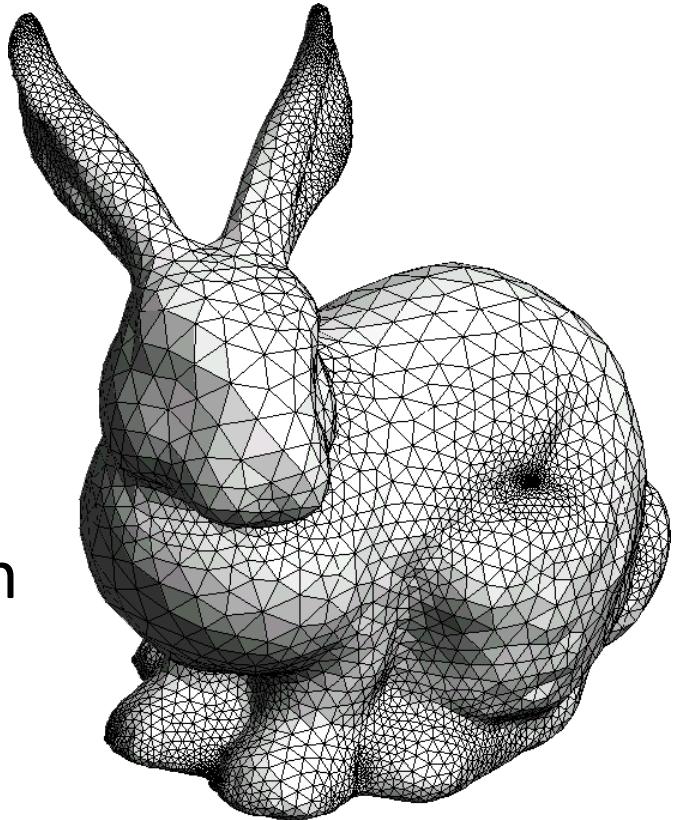
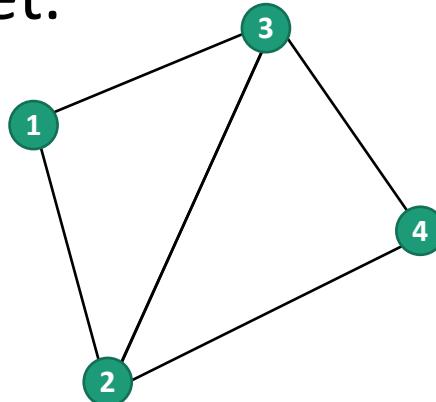


Graphics Pipeline – Was macht eine GPU?

- Die GPU führt in wesentlichen diese drei Schritte aus:
 1. Geometry Processing
 2. Rasterizer
 3. Fragment Processing

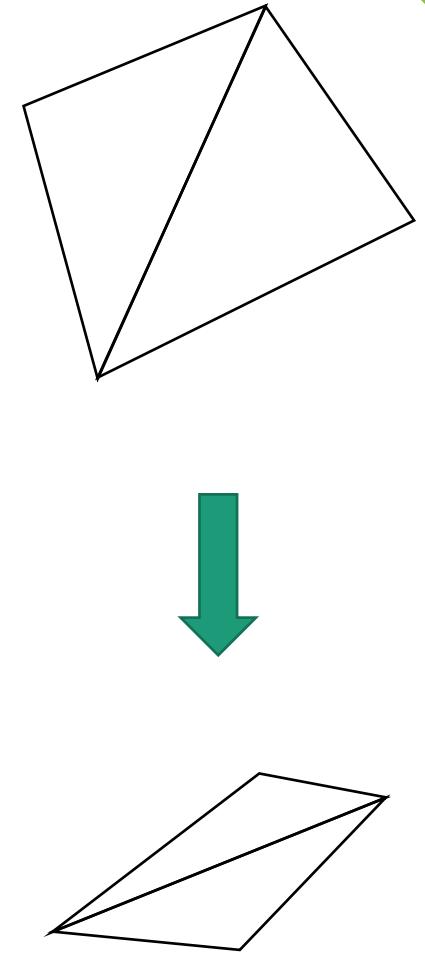
Datengrundlage:

- Objekte auf der GPU werden fast ausschliesslich durch Dreiecke repräsentiert. Deren Ecken werden als **Vertices** bezeichnet.
- **Mesh:** 3D-Modell aus Dreiecken oder Polygonen.



1. Geometry Processing

- Vorbereitung: CPU kopiert Koordinaten und weitere Daten der Vertices (2D oder 3D) auf GPU in einen Buffer. Koordinaten sind in lokalem Koordinatensystem des Objektes definiert.
- Processing für jeden Vertex:
 - Koordinaten werden mit Matrix-Multiplikationen zu Positionen auf Bildschirm umgerechnet.
 - Lokal -> Welt -> Kamera -> Projektion
 - Clipping: Dreiecke, die ausserhalb vom Bildschirm landen, werden ignoriert.
 - Zusatzinformationen pro Vertex können berechnet werden.



Matrix-Multiplikationen

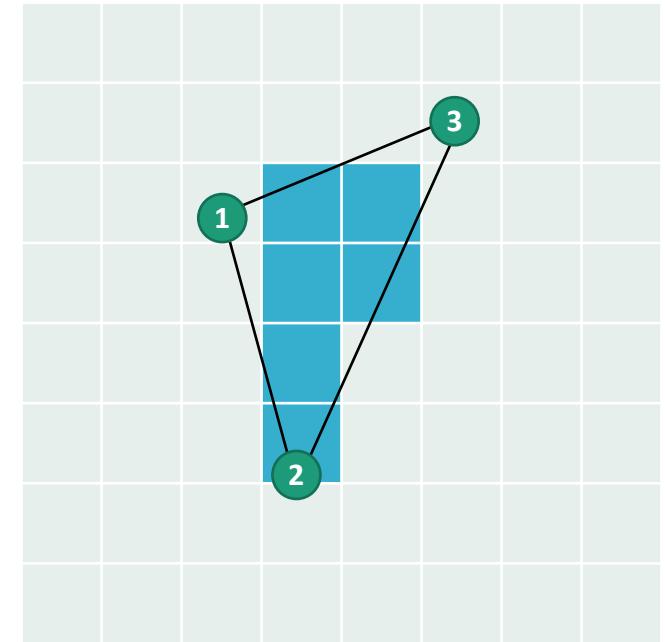
- Mini-Refresher: Verschiebung, Rotation, Skalierung etc. lassen sich sehr gut als Matrizen repräsentieren.

$$\begin{array}{c} \text{Verschiebung} \\[1ex] \left(\begin{array}{cccc} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right) * \left(\begin{array}{c} 1 \\ 2 \\ 3 \\ 1 \end{array} \right) = \left(\begin{array}{c} 6 \\ 2 \\ 3 \\ 1 \end{array} \right) \\[1ex] \text{Resultat} \\[1ex] \text{Position} \end{array}$$

- I.d.R. eine Matrix für relative Position von Objekt, eine für Kamera-Position, eine für Projektion von 3D auf Screen
- Endresultat = Position auf Screen
- Funktioniert auch in 2D (bsp. ScrollView)

2. Rasterizer

- Dreieck wird in Pixels (**Fragments**) aufgeteilt.
- Zentrum des Pixels muss innerhalb vom Dreieck liegen. Jedes Pixel (auch auf Kante) wird nur einem Dreieck zugeordnet.
- Interpolation: Zusatzinformationen der Vertices werden für jedes Fragment interpoliert.



3. Fragment Processing

- Pro Fragment wird eine beliebige Farbe berechnet. Häufig werden dafür konstante Farben, Sampling und Lighting kombiniert.
- Texture Sampling: Farbe eines Pixel von einem Bild (**Textur**) auslesen.
- Lighting / Shading: Original-Farbe basierend auf Einfallwinkel und Intensität des Lichtes modifizieren.

Graphics Programming

- Erste GPUs: Fixes Geometry Processing und wenige Operationen für Fragment Processing.
- Heute: Geometry und Fragment Processing der GPU kann programmiert werden. Diese Programme werden **Vertex Shader** und **Fragment Shader** genannt.
- Die GPU wird mit einem Framework gesteuert und programmiert, dafür stehen je nach Plattform unterschiedliche Optionen zur Auswahl:
 - **OpenGL (iOS, Android, Windows, macOS, Web)**
 - **Metal (iOS, macOS)**
 - DirectX (Windows, Xbox)
 - Vulkan (Windows, Linux, macOS, Android)
 - Gnm, Gnmx (Playstation)

OpenGL / Metal



- 1992 veröffentlicht
- Meistverbreitete Lösung
- Seit iOS 12 deprecated
- GLSL (OpenGL Shading Language)
 - Basiert auf C
 - Runtime-compiled Strings
- Globale Funktionen und State Machines



- 2014 von Apple veröffentlicht
- Für Apple-Hardware optimiert
- Im Allgemeinen performanter
- MSL (Metal Shading Language)
 - Basiert auf C++
 - Precompiled
- Objekt-orientierte API



Demo

SpriteKit / SceneKit

- Fazit: OpenGL / Metal == Viel Boilerplate-Code
- Die beiden Frameworks **SpriteKit** und **SceneKit** basieren auf Metal und ermöglichen High-Performance Rendering für Spiele, ohne dass man sich mit Metal herumschlagen muss.



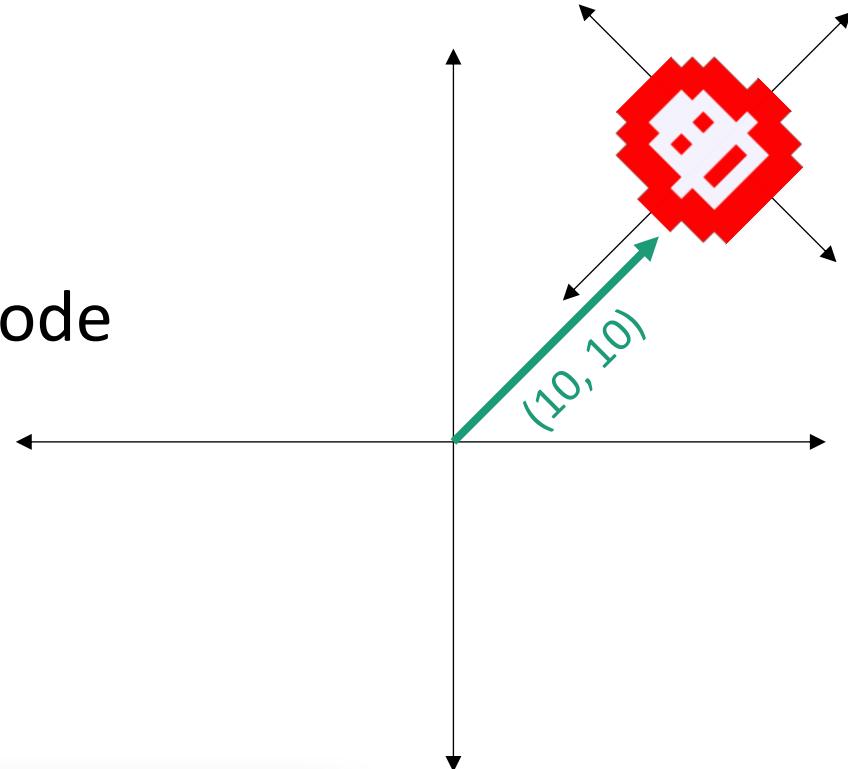
SpriteKit (2D)



SceneKit (3D)

SpriteKit

- Die Anwendung platziert **Sprites** (Grafikobjekte) und andere Objekte als Node in einer Szene
- Framework erstellt und aktualisiert alle Buffers, Dreiecke, Texturen, Shaders, etc.



```
let image = SKSpriteNode(imageNamed: "sprite.png")
image.position = CGPointMake(10, 10)
image.zRotation = .pi / 4.0

// Add the image to the scene.
scene.addChild(image)
```

Actions & Physics

- Nodes können mit **Actions** vielseitig animiert und modifiziert werden
- Falls den Nodes ein **Physics Body** zugewiesen wird, simuliert SpriteKit Kräfte, Kollisionen, etc.

```
spinnyNode.run(SKAction.repeatForever(  
    SKAction.rotate(byAngle: .pi, duration: 1)  
)  
  
spinnyNode.run(SKAction.scale(by: 0.1, duration: 2))  
  
spinnyNode.run(SKAction.sequence([  
    SKAction.wait(forDuration: 0.5),  
    SKAction.fadeOut(withDuration: 0.5),  
    SKAction.removeFromParent()  
])))
```

```
let body = SKPhysicsBody(rectangleOf: CGSize(width: w, height: w * 2))  
body.mass = 100  
body.friction = 0.3  
spinnyNode.physicsBody = body  
  
body.applyImpulse(CGVector(dx: 10, dy: 0))  
body.applyTorque(0.3)
```

SceneKit

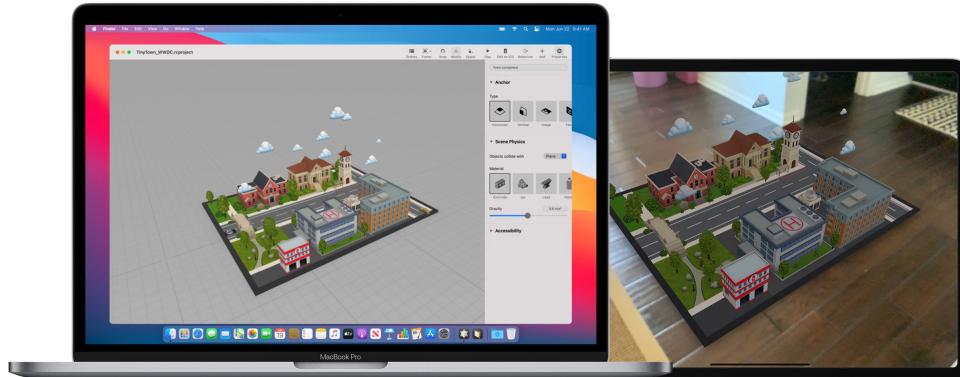
- Die gleichen Grundkonzepte wie SpriteKit: Nodes, Actions, Physics, ...
- Viele weitere APIs für 3D-Meshes, Licht, Rendering, etc.



Demos

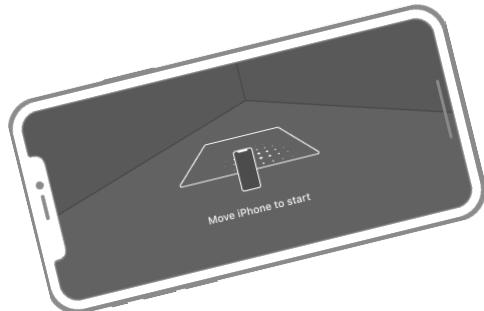
Extended Reality

- Unter dem Begriff **Extended Reality (XR)**, werden Virtual und Augmented Reality zusammengefasst
- **Virtual Reality (VR)**: Simulierte Welt durch 3D-Rendering in Echtzeit, in der Regel mit 3D-Brillen
- **Augmented Reality (AR)**: Kombination von echten und virtuellen Inhalten, häufig optisch (modifiziertes Kamerabild oder teil-transparenter Bildschirm, aber auch akustisch, haptisch, etc.)



ARKit

- SpriteKit, SceneKit und die meisten Metal-Anwendungen verwenden das Konzept einer **Kamera**, deren Position und Rotation die Projektion von globalen Koordinaten auf den Screen definieren.
- Das Framework **ARKit** berechnet aus den Kameras, Beschleunigungs- und Rotations-Sensoren eine virtuelle Kamera, mit der die Illusion von Objekten im Raum gerendert werden kann.

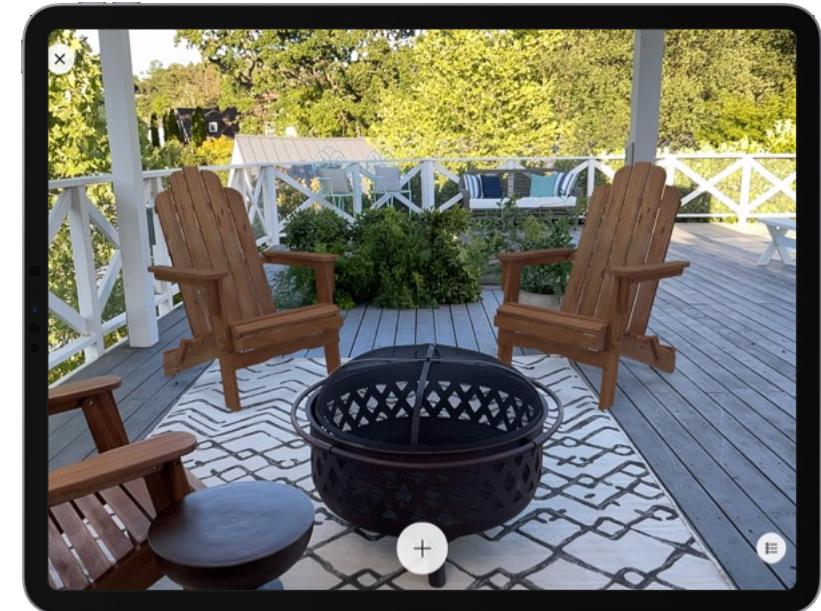


$$\begin{pmatrix} 0.3 & 0.5 & 0.1 & 1.4 \\ 3.4 & 0.9 & 4.4 & 2.3 \\ 7.5 & 5.2 & 0.6 & 0.2 \\ 0.1 & 0.6 & 1.4 & 1.0 \end{pmatrix}$$

- Rendering mit einem beliebigen Framework: Metal, SpriteKit, SceneKit, oder **RealityKit**.

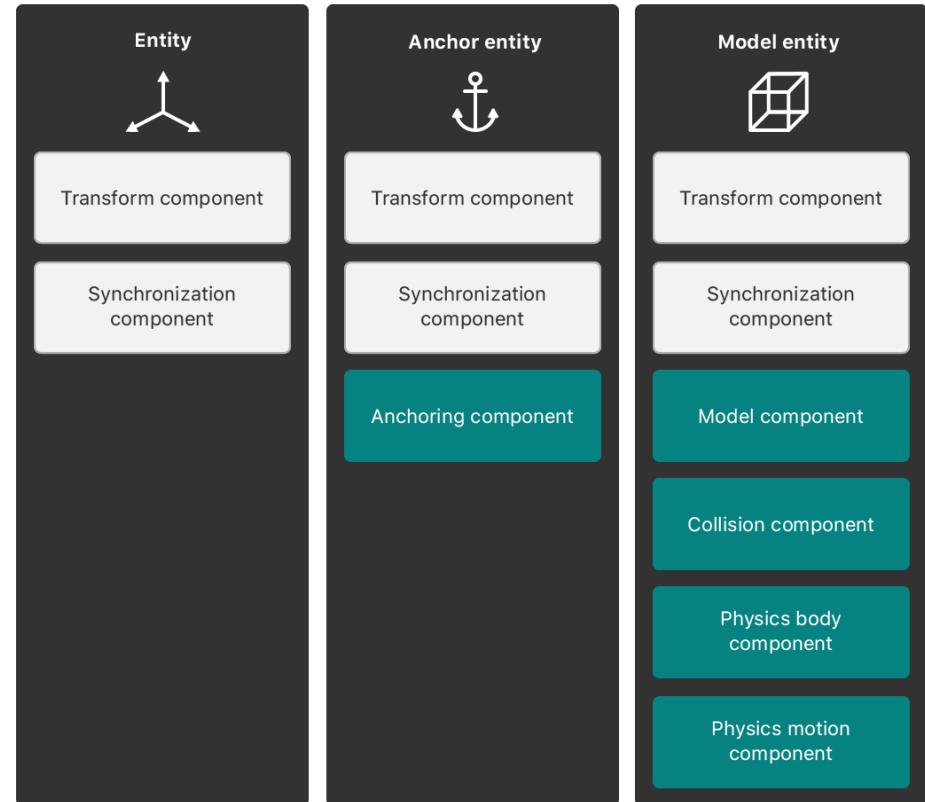
RealityKit

- Neues, speziell für AR entwickeltes 3D-Graphics-Framework.
- Grundfunktionen wie SceneKit: 3D-Meshes, Animationen, Physik, ...
- Rendering-Effekte für mehr Realismus
 - Körnung
 - HDR
 - Grund-Schatten
 - Motion-Blur
 - Tiefenschärfe
 - Okklusion von Personen
 - Beleuchtung



ECS

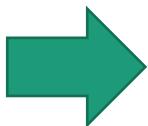
- RealityKit modelliert Objekte als Entities nach dem Software-Architektur Pattern **Entity-Component-System (ECS)**.
- Statt komplexer Vererbung wird die einfache Klasse **Entity** mit verschiedenen **Components** erweitert.
- Ein **System** agiert auf allen Entities, die bestimmte Components enthalten.
 - Model → Rendering
 - Physics Body → Schwerkraft



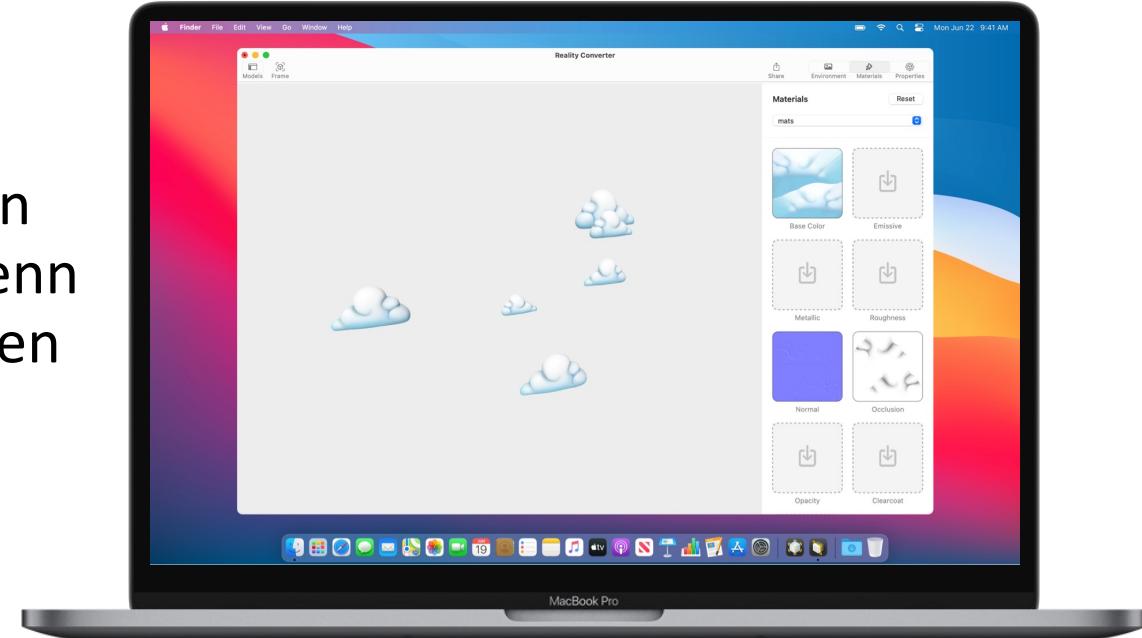
```
let entity = ModelEntity(mesh: .generateSphere(radius: 0.25))  
entity.components[FlyingComponent.self] = FlyingComponent()  
entity.components[SwarmComponent.self] = SwarmComponent()
```

AR Creation Tools

- Komplete AR-Welt kann in Code erstellt werden, zusätzliche Tools vereinfachen den Prozess
 - Reality Converter: 3D-Objekte für AR optimieren
 - Object Capture: macOS-API, aus Bildern ein 3D-Model erstellt werden kann, wenn genügend Winkel aufgenommen wurden
 - Reality Composer: AR Szene einfach «zusammenklicken».

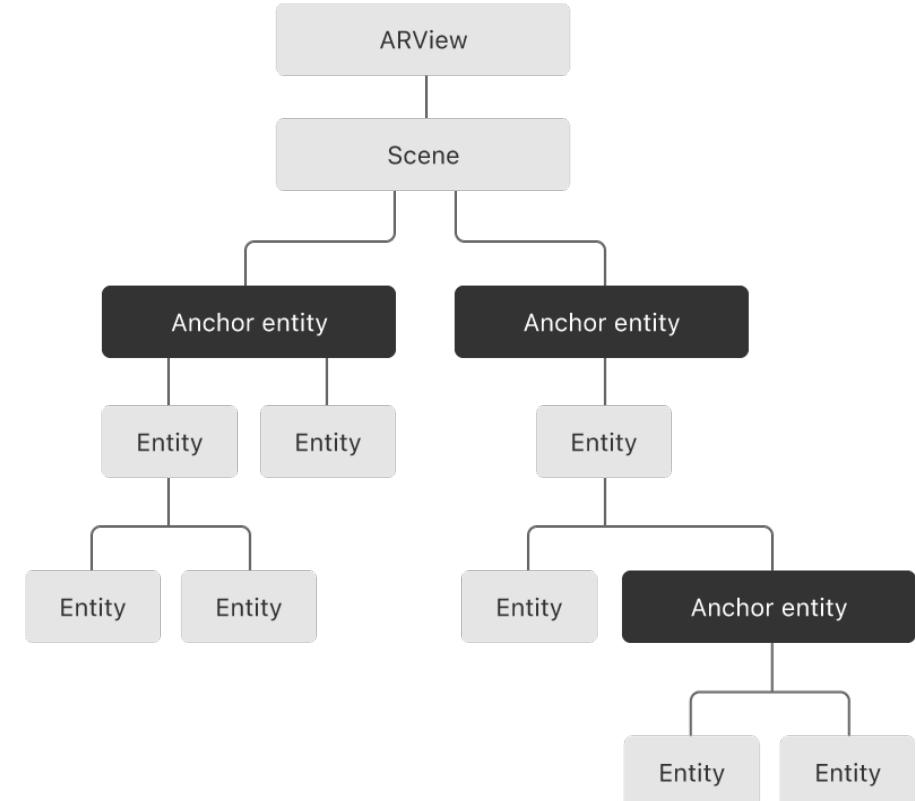


Demo



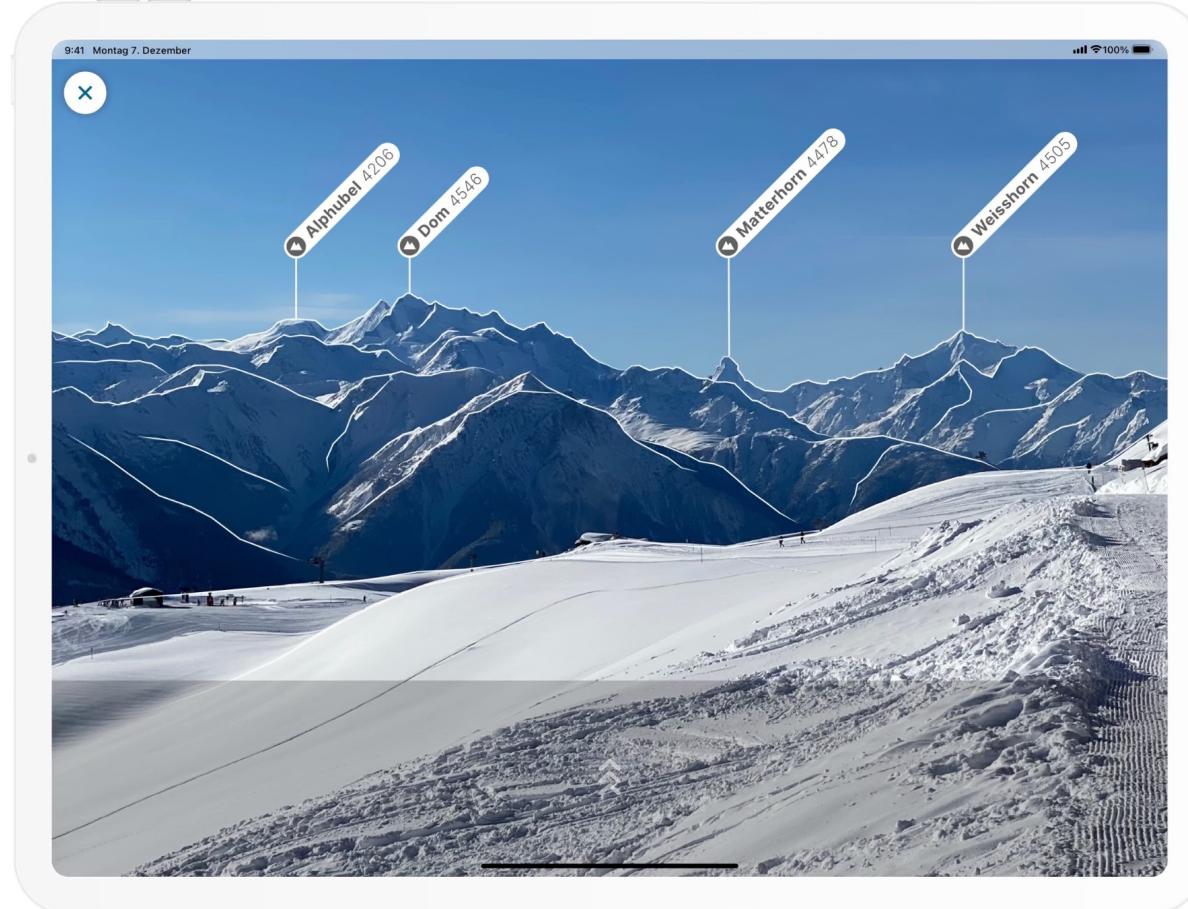
Anchors

- Virtuelle Objekte werden mit **Anchors** in der Szene platziert.
- Verschiedene Subklassen
 - Plane: Horizontale/vertikale Ebene
 - Object: Referenz 3D-Modell
 - Image: Referenz 2D-Bild
 - Mesh: Rekonstruktion eines Objektes
 - Face: Gesicht in Front-Kamera
 - Body: Körper/Skeleton
 - Participant: Andere User
 - GeoAnchor: Koordinaten



GeoAnchor

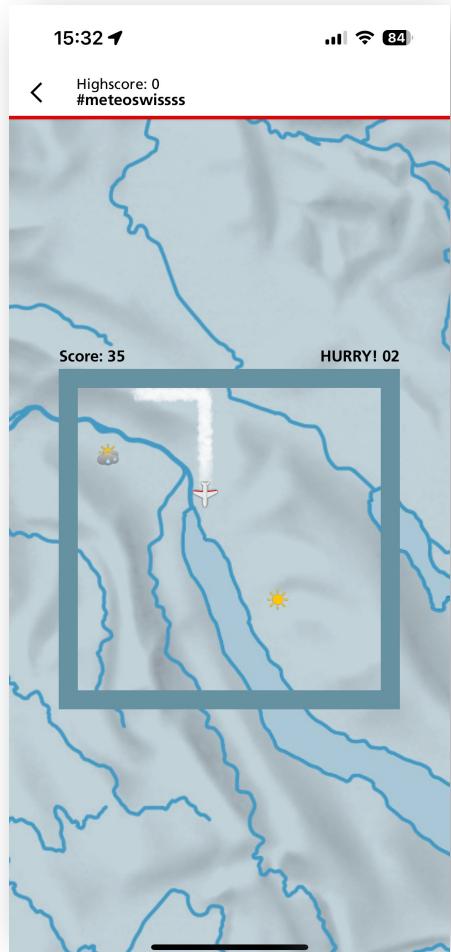
- Achtung: Nur in einigen Städten (primär USA) verfügbar!
- iOS kombiniert GPS und Kompass mit «Street View» - Aufnahmen für genaue Position.
- Beispiel swisstopo App: GPS und Kompass manuell zu Transformations-Matrix umrechnen (ungenauer).



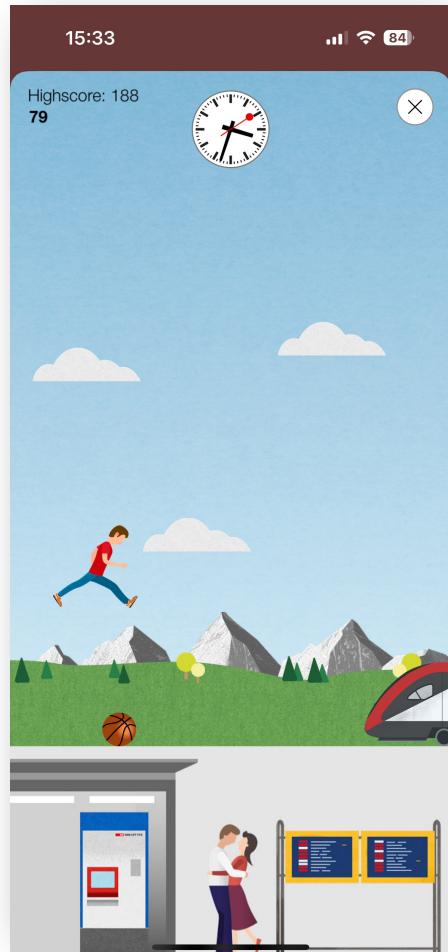
«Zusammenfassung»



SwiftUI



OpenGL



SpriteKit



SceneKit