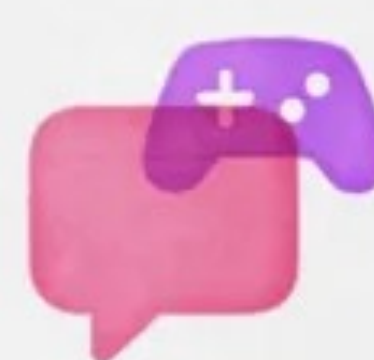


移动平台开发技术

iOS开发中的图像处理与渲染加速

章国锋 2025春夏学期

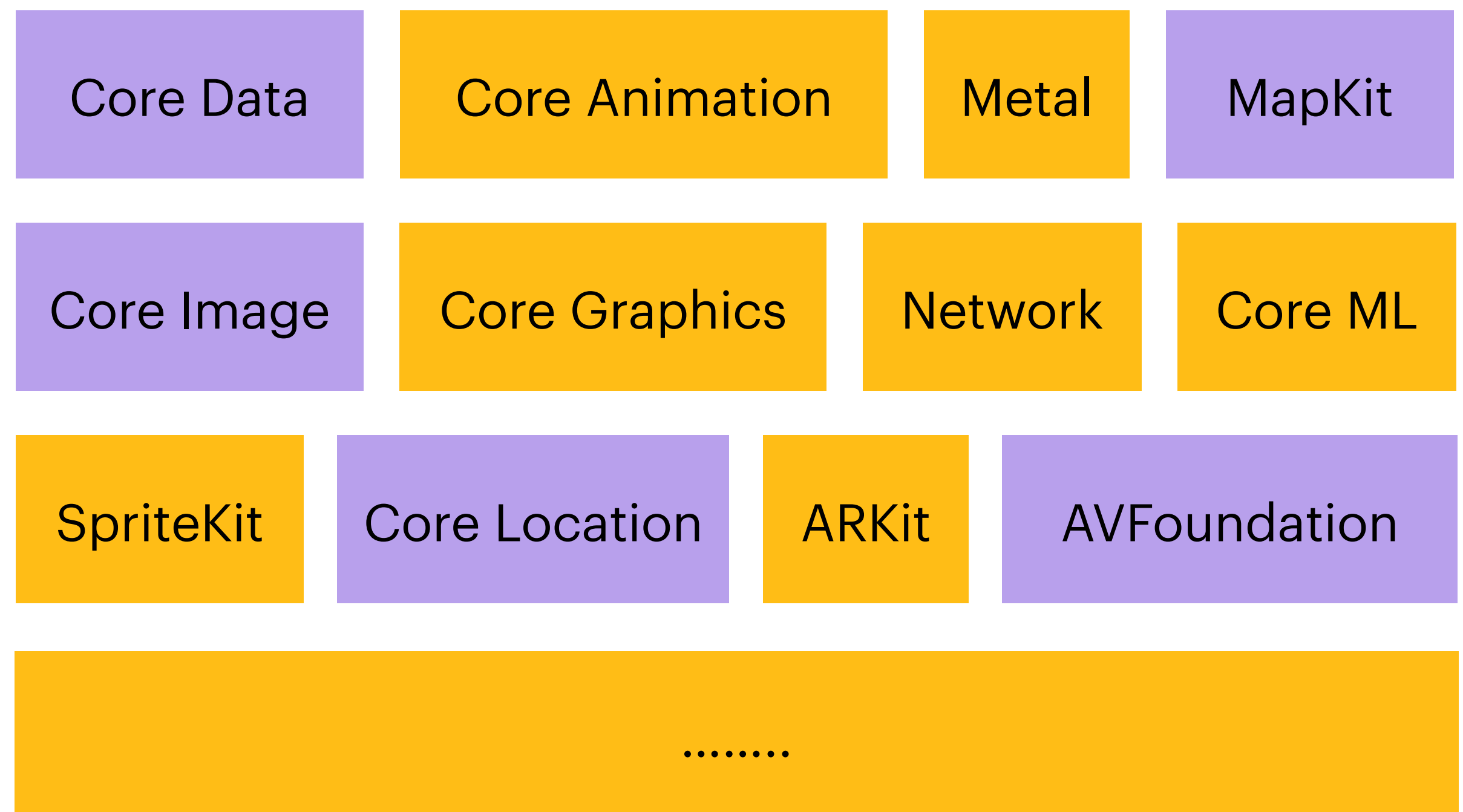


Frameworks

Frameworks



- 苹果为开发者提供了大量方便快捷的框架
- 在开发时候可以直接调用，缩短开发时间
- 之前的课程中我们已经简单了解了部分框架

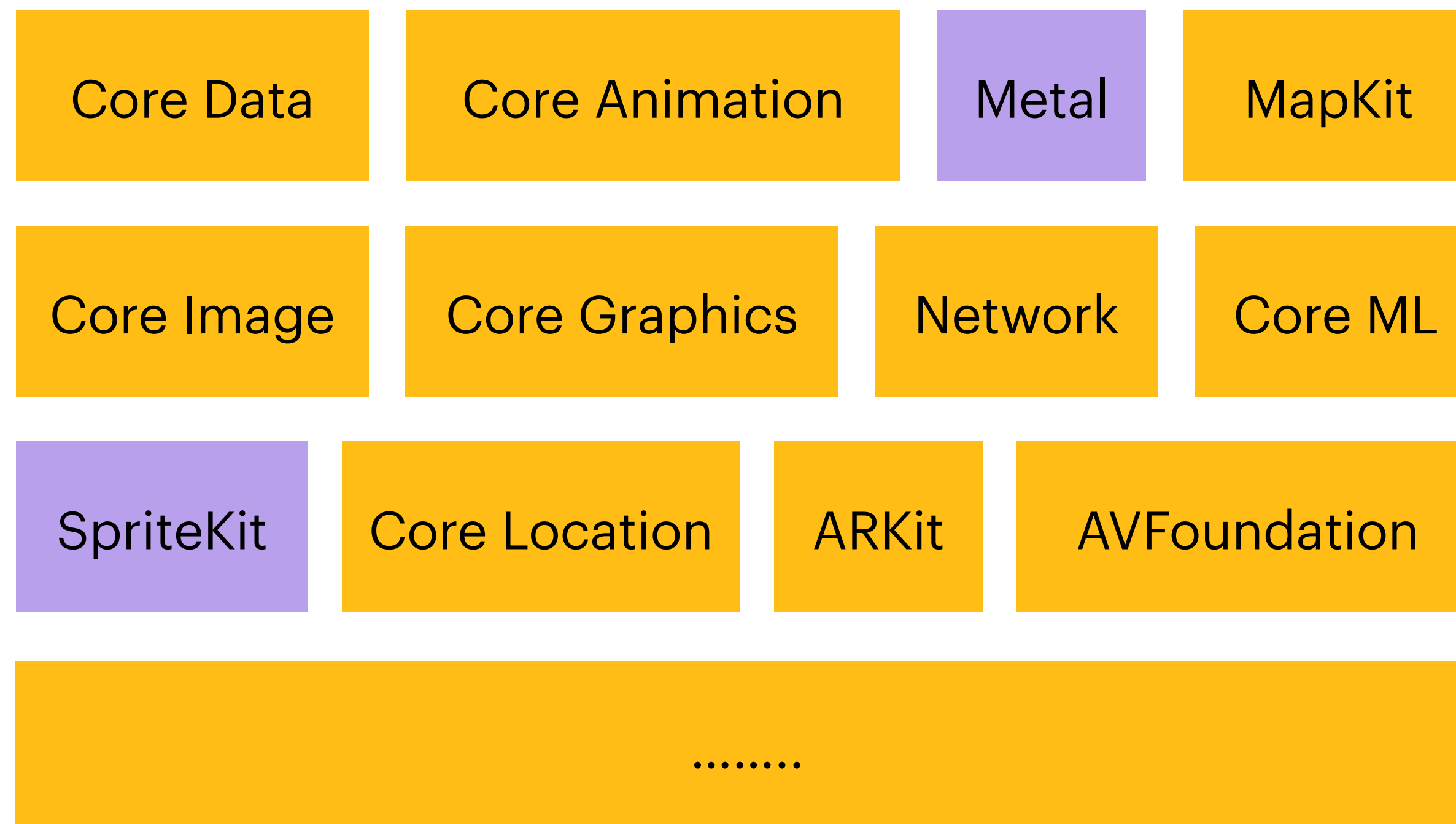


Frameworks

Frameworks



- 苹果为开发者提供了大量方便快捷的框架
- 在开发时候可以直接调用，缩短开发时间
- 之前的课程中我们已经简单了解了部分框架
- 本次课程我们将介绍：
 - **SpriteKit**
 - **Metal**



回顾第二次课程

Review of previous course



提问：在前面的课程中我们如何在SwiftUI中绘制任意图案？

回顾第二次课程

Review of previous course



提问：在前面的课程中我们如何在SwiftUI中绘制任意图案？

Circle()

Ellipse()

Capsule()

RoundedRectangle(cornerRadius: 10))

Rectangle()

.stroke(.red, style: 3)

.fill(.orange)

回顾第二次课程

Review of previous course



浙江大学
ZHEJIANG UNIVERSITY

提问：在前面的课程中我们如何在SwiftUI中绘制任意图案？

```
struct Triangle: Shape {  
    func path(in rect: CGRect) -> Path {  
        var path = Path()  
  
        path.move(to: CGPoint(x: rect.midX, y: rect.minY))  
        path.addLine(to: CGPoint(x: rect.minX, y: rect.maxY))  
        path.addLine(to: CGPoint(x: rect.maxX, y: rect.maxY))  
        path.addLine(to: CGPoint(x: rect.midX, y: rect.minY))  
  
        return path  
    }  
}
```


Apple图形绘制框架

Structure of Graphics frameworks on Apple devices



浙江大学
ZHEJIANG UNIVERSITY

**Useful and easy-to-use
game engine**

SpriteKit

SceneKit

RealityKit

low-level API

Metal

Quartz 2D

Core Graphics

devices

GPU

CPU

(大致框架, 仅供参考)

https://hci.rwth-aachen.de/index.php?option=com_attachments&task=download&id=2435

课程目录



1

SpriteKit 基础

2

GPU 并行计算概念

3

Metal 基础

SpriteKit

活跃气氛：你还在哪里见到Sprite？

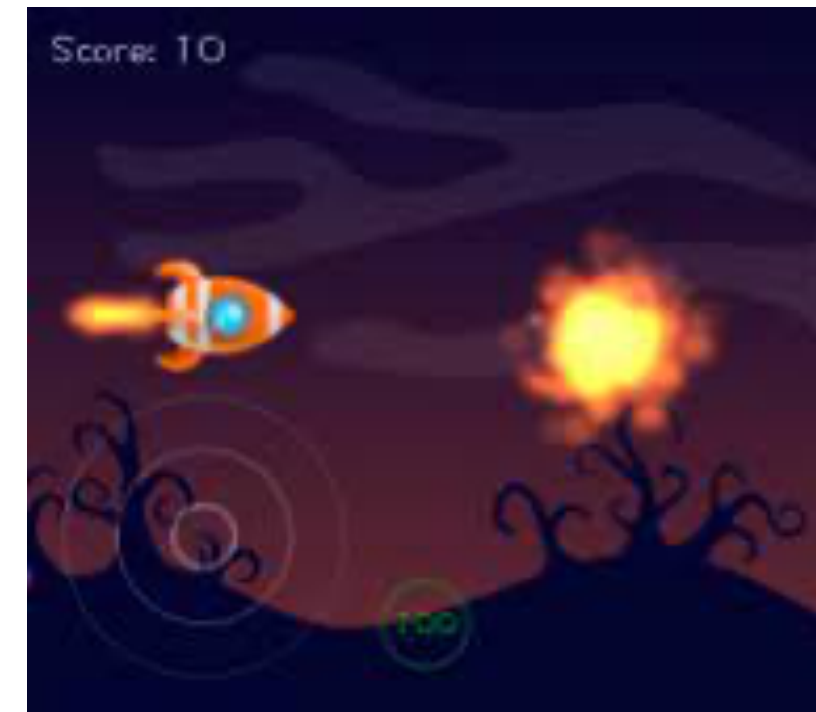
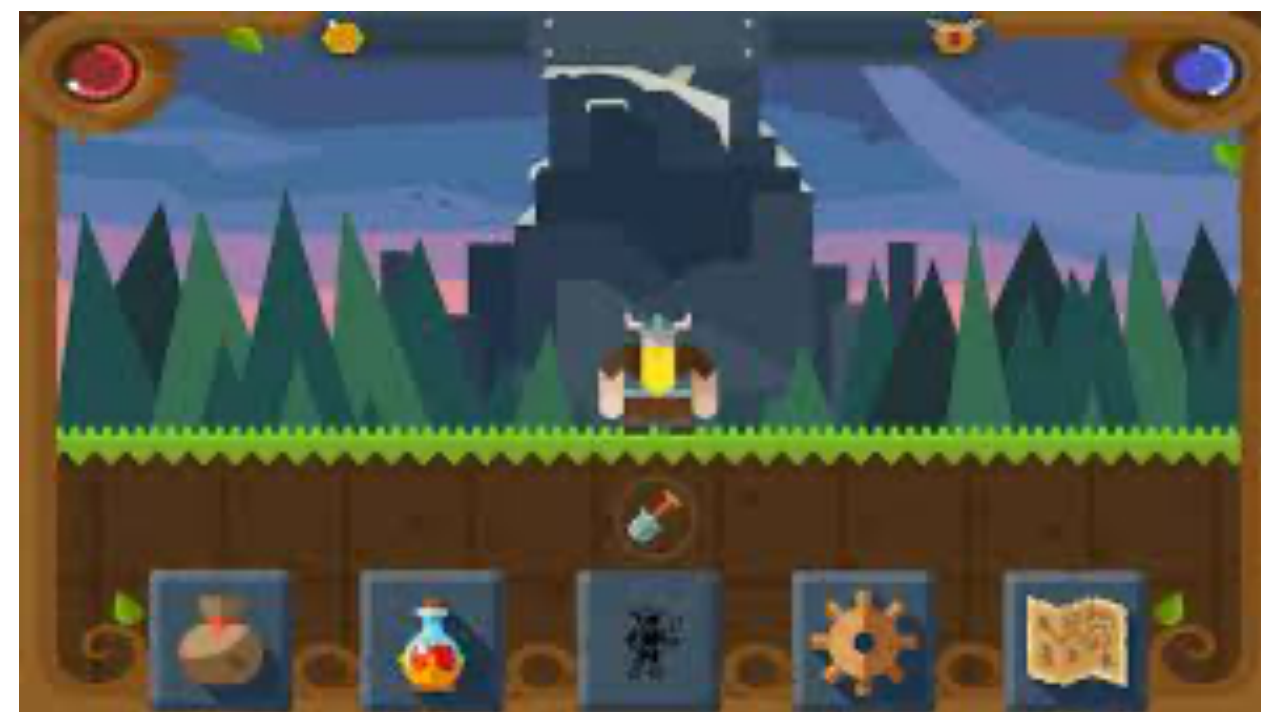
Sprite基本概念

CSS Sprite、雪碧

- **Sprite**：一个计算机图形学概念
 - a sprite is a **two-dimensional bitmap** that is integrated into a larger scene, most often in a 2D video game
 - 大家可以想到很多经典2D游戏



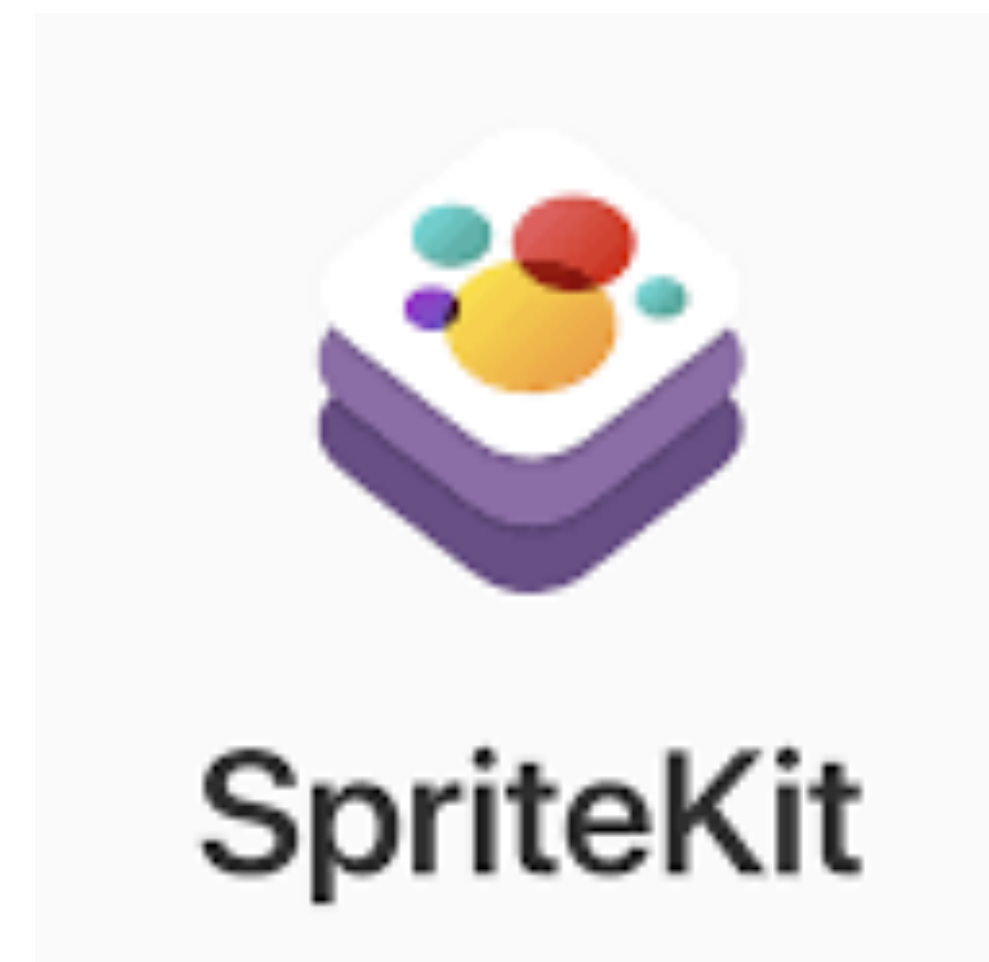
iOS开发中的图形绘制与加速



SpriteKit

SpriteKit基本概念

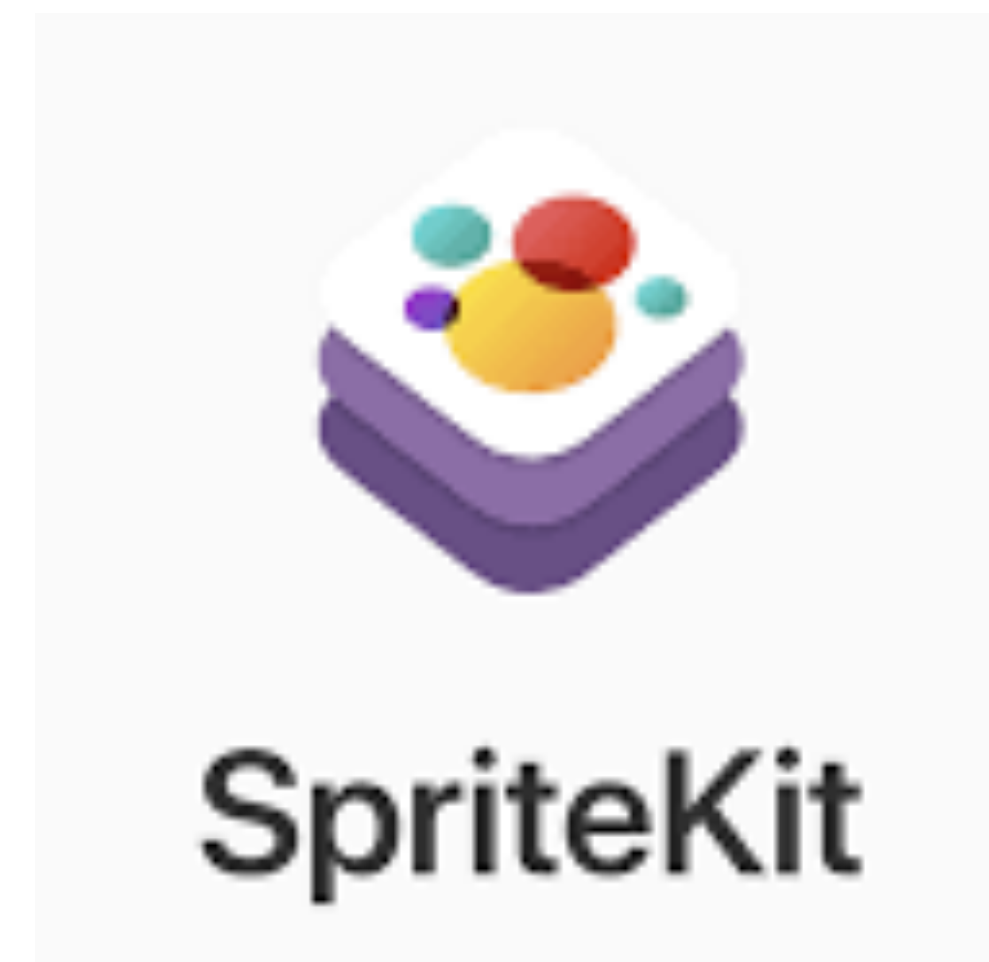
- **SpriteKit**是一个图形渲染和动画基础设施，可以使用它来动画化任意纹理图像（纹理图像也称为精灵**Sprite**）
 - 使用图形硬件加速
 - 针对内容进行任意动画或变更进行了优化
 - 更适合需要灵活处理动画的游戏和应用程序
 - 还提供了其他对游戏非常有用的功能，包括基本的声音播放支持和物理模拟



SpriteKit

基本概念

- **SpriteKit** 使用主机设备提供的图像硬件，以高帧数复合2D图像，iOS和macOS均支持
- **SpriteKit** 支持多种不同的内容，包括：
 - 无纹理或者有纹理的矩形（精灵）
 - 文本
 - 任意基于**CGPath**的形状
 - 视频
 - 裁剪和其他特效的支持



SpriteKit



重要的类

- Scenes
- Nodes
- Actions

Scenes

- 游戏中的内容被组织成由**SKScene**对象表示的**Scenes**。
 - **Scenes**容纳精灵和其他要呈现的内容
 - **Scenes**实现了每帧逻辑和内容处理
 - 在任何给定的时间，视图呈现一个**Scenes**。只要呈现**Scenes**，它的动画和每帧逻辑就会自动执行

SpriteKit

Nodes

- **SKNode**类是大多数SpriteKit内容的基本构建块
- 使用预定义的**SKNode**子类来绘制SpriteKit的游戏中的所有视觉元素
- 每个节点的位置在基于父节点坐标系定义的（相对坐标）
- **SKNode**类不绘制任何东西，但它将其属性应用于其后代

SpriteKit

Actions

- 动作是自包含的对象。每一个动作是一个不透明的（opaque）对象，描述你想对场景做的改变
- 一切动作都是由**SKAction**类实现，它没有可见的子类
- 当场景处理动画帧时，执行Action
- Action最常见的用法是更改节点的属性

SpriteKit



使用方法

- 使用精灵
- 添加动作到节点
- 构建场景
- 使用场景间过渡

1. 使用精灵

- 创建一个精灵的最简单方法是让

SpriteKit为你创建纹理和精灵。

可以把插图存储在**bundle**中，

然后在运行时加载

//从存储在**bundle**中的图像创建一个纹理的精灵

```
let spaceship = SKSpriteNode(imageNamed:"rocket")  
spaceship.position = CGPointMake(100,100)  
self.addChild(spaceship)
```

- 当你使用这种方式时，你可以获得很多默认行为：
 - 精灵以匹配纹理尺寸的**frame**来创建，精灵的**frame**属性指定的矩形定义了它所涵盖的面积
 - 精灵纹理在**framebuffer**中是**alpha-blended**
 - 一个**SKTexture**对象被创建并附加到精灵上

1. 使用精灵

- 调整精灵尺寸，可由3个属性决定：
 - 精灵的size属性。
 - 基准尺寸。根据精灵从SKNode类继承来的xScale与yScale属性进行缩放。
 - 当精灵的框架大于它的纹理时，纹理被拉伸以覆盖框架。
- 对精灵着色
 - 可以使用color和colorBlendFactor属性对它着色
 - 可以使用动作让颜色和颜色混合，成为动画

1. 使用精灵

- 创建一个纹理对象，多个精灵可以复用
 - 使用 **SKTexture**
 - 可以控制的参数更多

```
let bottomLeftTexture =  
    SKTexture(rect: CGRectMake(0.0,0.0,0.5,0.5),  
              inTexture: cornerTextures)  
  
let pulsedRed = SKAction.sequence([  
    SKAction.colorizeWithColor(SKColor.redColor(),  
                                colorBlendFactor:1.0,duration:0.15)  
    SKAction.waitForDuration(0.1)  
    SKAction.colorrizeWithColorBlendFactor(0.0,  
                                             duration :0.15)])  
monsterSprite.runAction(pulsedRed)
```

2. 添加动作到节点

- 绘制精灵仅仅绘制出静态图像
- SpriteKit让场景动起来所使用的主要机制是Action
- 通过Action定义你想对场景所作的改变
- 在大多数情况下，一个Action对执行该Action的节点应用其变化。

2. 添加动作到节点

- 一旦**Action**被创建，它的类型就不能再改变，并且只能有限的改变其属性。在此基础上，**SpriteKit**利用**Action**的不变性有效地执行它们
- 而一个**Action**只有在你告诉一个节点运行它之后才会执行，运行一个节点最简单的方法是调用节点的**runAction**方法

2. 添加动作到节点

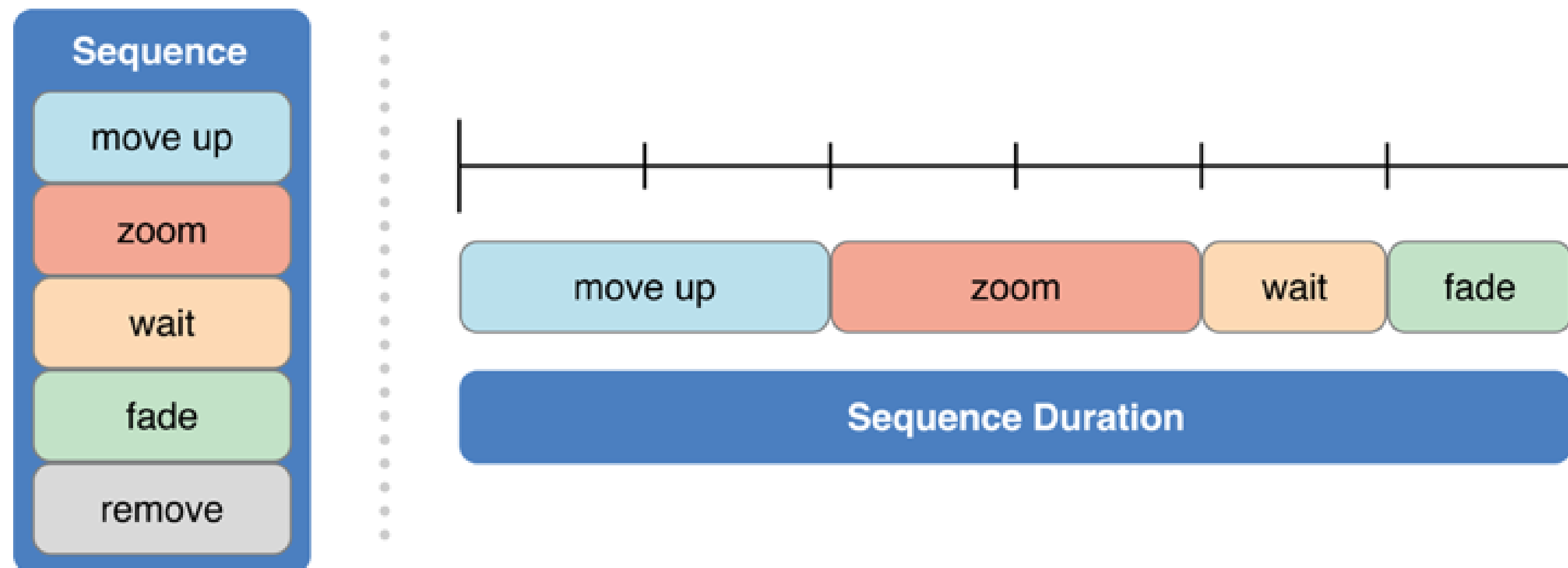
- **SpriteKit**提供了许多标准类型用来改变场景中节点的属性，你可以通过结合Action创建复杂的动画，这些动画仍然通过运行一个单一的**Action**来执行：
 - 序列动作（**sequence action**）：连续运行的动作集合（set）
 - 组动作（**group action**）：同时执行的动作集合
 - 重复动作（**repeating action**）：一个动作重复执行

2. 添加动作到节点

序列动作

```
let moveUp = SKAction.moveByX(0,y:100,duration:1)
let zoom = SKAction.scaleTo(2.0,duration:0.25)
let wait = SKAction.waitForDuration(0.5)
let fadeAway = SKAction.fadeOutWithDuration(0.25)
let removeNode = SKAction.removeFromParent()
let sequence =
SKAction.sequence([moveup, zoom, wait, removeNode])

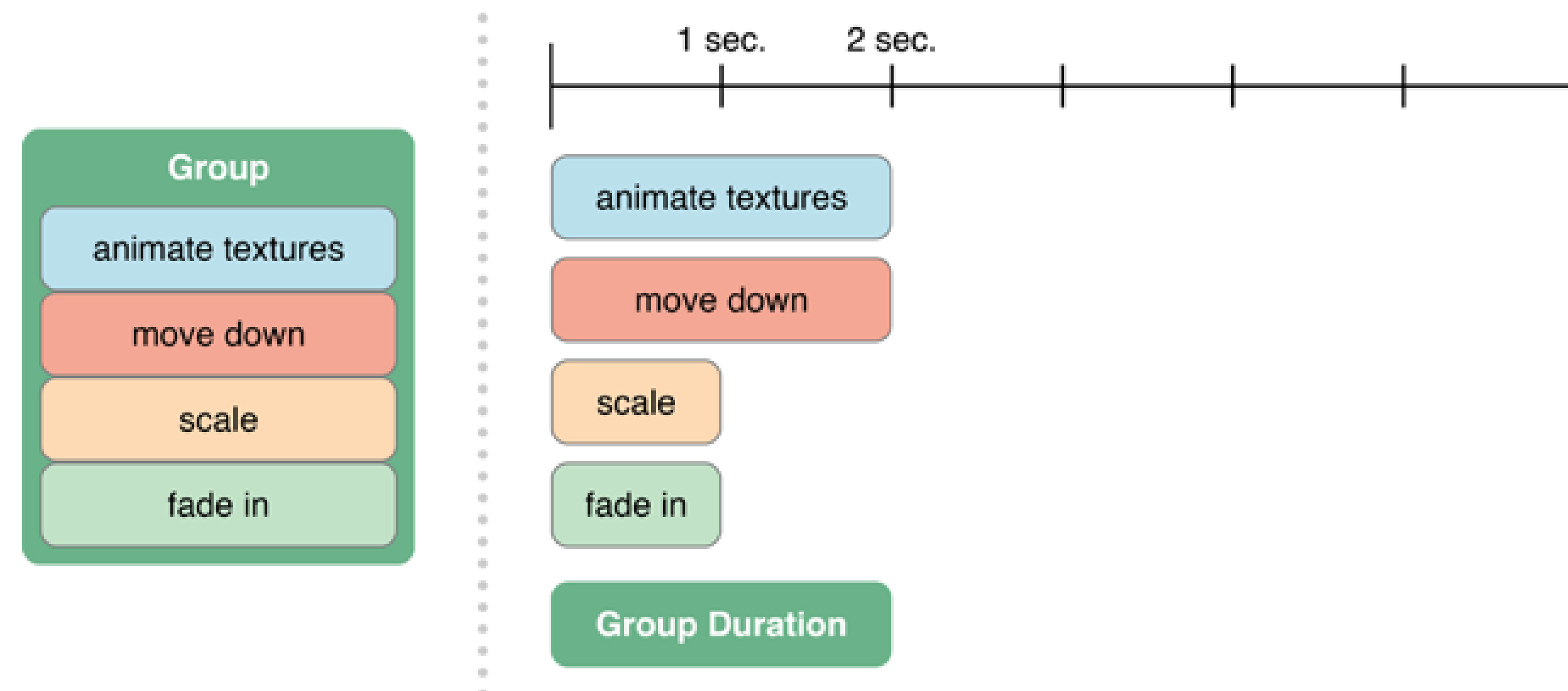
node.runAction(sequence)
```



2. 添加动作到节点

组动作

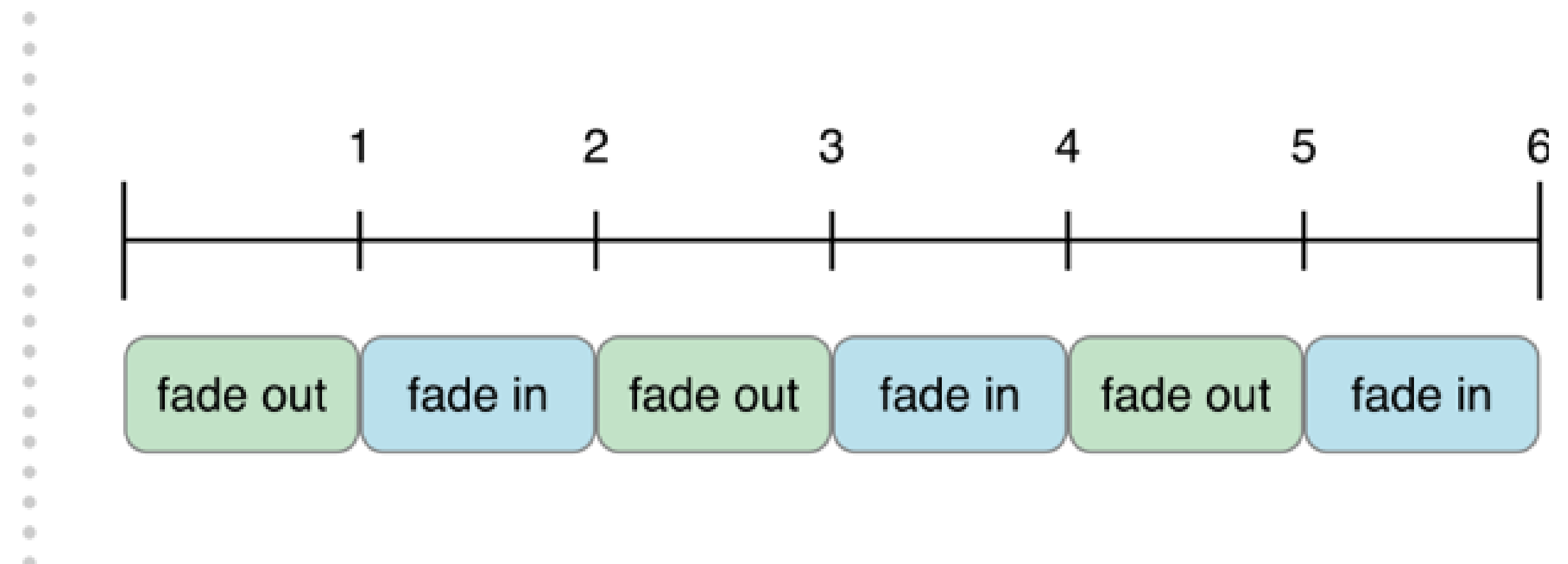
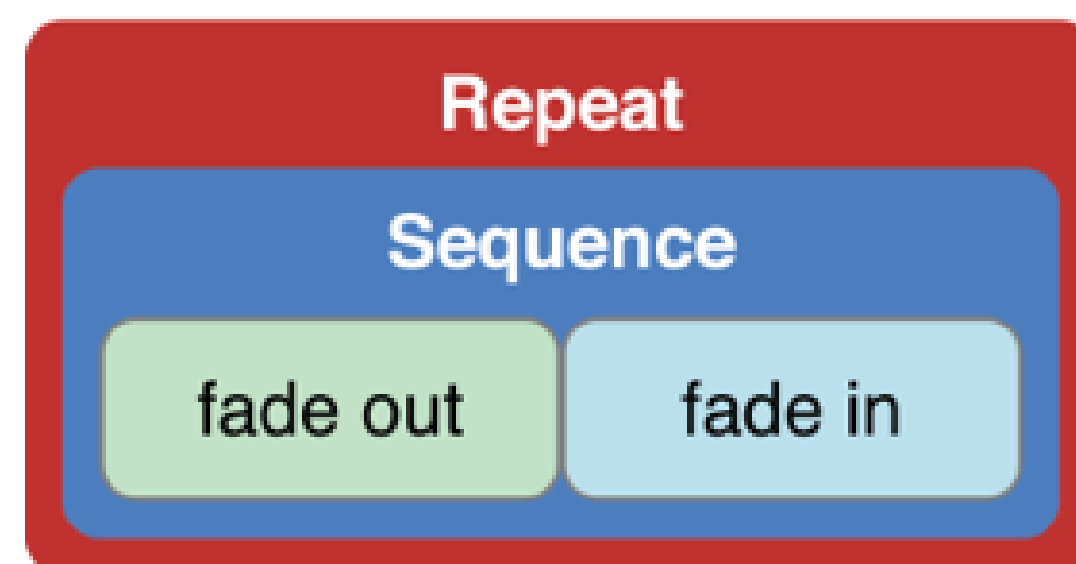
```
let animate =  
SKAction.animateWith(0,y:100,duration:1)  
let moveDown=  
SKAction.moveByX(0,y:-200,duration:2.0)  
let scale = SKAction.scaleTo(1.0,duration:1.0)  
let fadeIn = SKAction.fadeInWithDuration(1.0)  
let group =  
SKAction.group([animate,moveDown,scale,fadeIn])  
  
node.runAction(group)
```



添加动作到节点

重复动作

```
let fadeOut = SKAction.fadeOutWithDuration(1)
let fadeIn = SKAction.fadeInWithDuration(1)
let pulse = SKAction.sequence([fadeOut, fadeIn])
let pulseThreeTimes =
    SKActions.repeatAction(pulse, count: 3)
let pulseForever =
    SKAction.repeatActionForever(pulse)
```

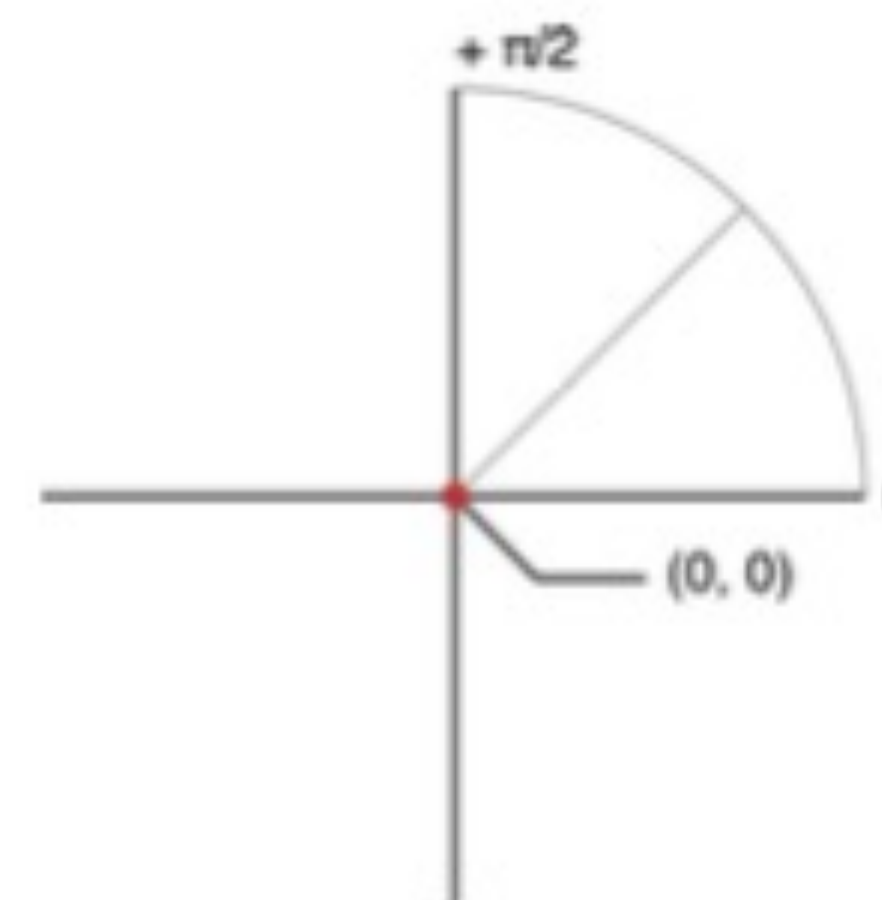
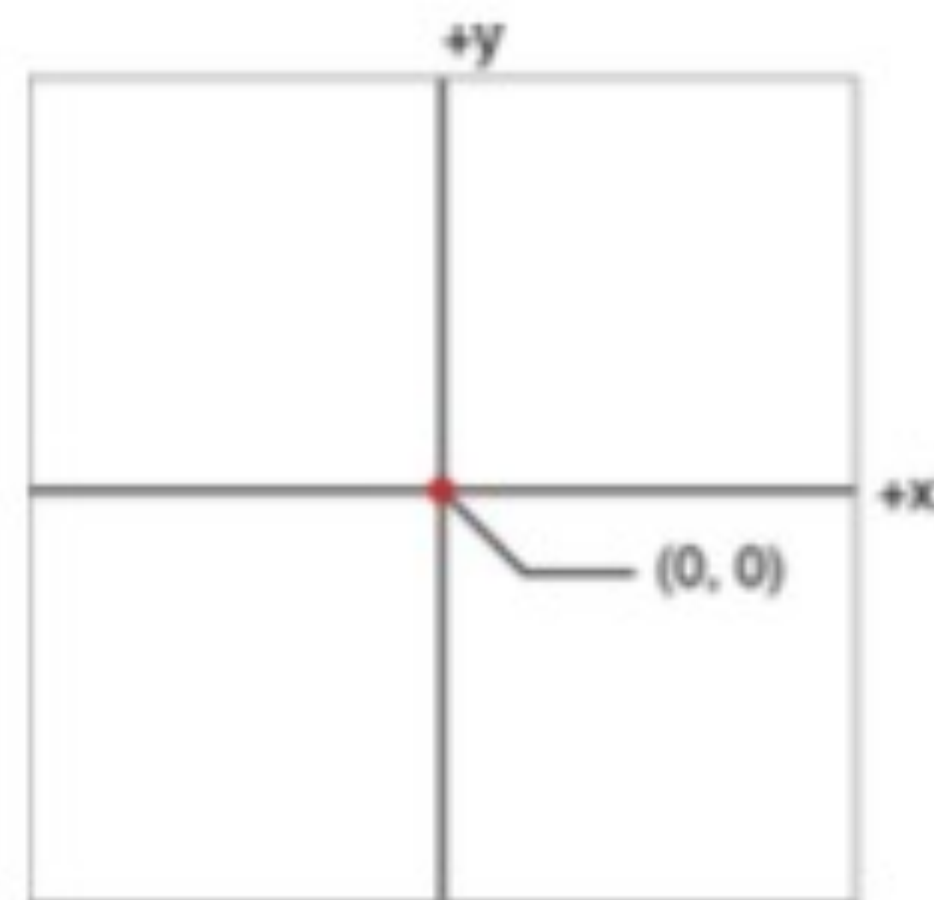


3. 构建场景

- 场景（**SKScene**对象），用来提供**SKView**对象要渲染的内容。
- 场景的内容被创建为树状的节点对象。场景是根节点。
- 在场景由视图呈现时，它运行动作并进行物理模拟，然后渲染节点树。
- 你可以通过子类化**SKScene**类创建自定义的场景。

3. 构建场景

- 节点给子节点提供坐标系。当一个节点被放置在节点树中，它的position属性把它定位在由它的父节点提供的坐标系内。
- **SpriteKit**在iOS和macOS中使用相同坐标系，坐标值用点来测量。
- **SpriteKit**有一个标准旋转约定（rotation convention）。弧度为0的角指定正x轴。沿逆时针方向为正。



3. 构建场景

- 创建场景：
 - 场景由视图呈现
 - 场景的尺寸定义其可见区域
 - 场景被渲染后，内容会被复制呈现到视图
 - 如果场景和视图尺寸不同，则会缩放场景

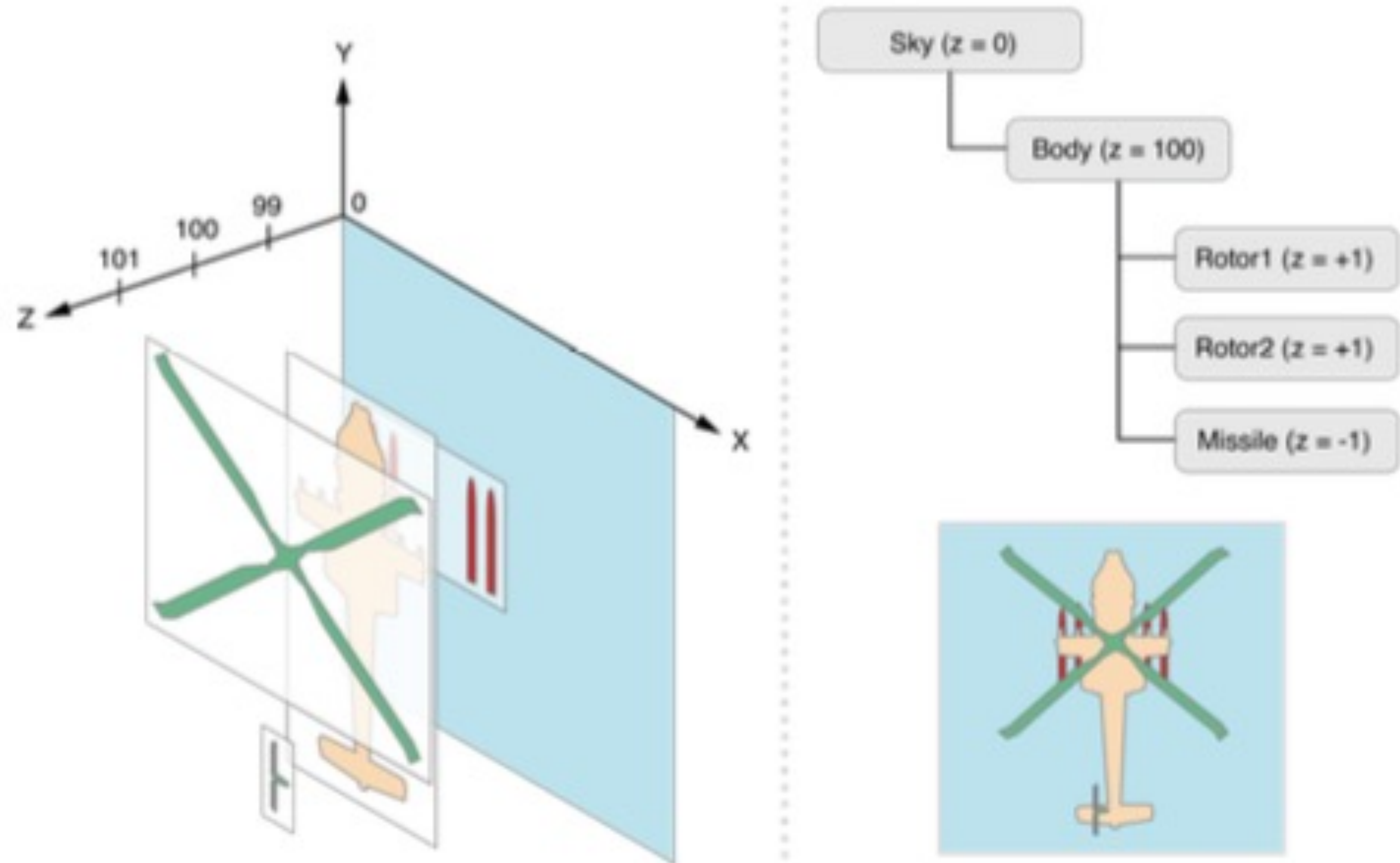
3. 构建场景

- 创建节点树：
 - 通过创建节点之间父子关系来创建节点树
 - 每个节点维护一个有序的子节点列表，读取节点的children属性进行引用
 - 子节点在树中的顺序会影响场景处理的多个方面，包括碰撞检测和渲染，所以必须适当的组织节点树

Method	Description
<code>addChild(_:)</code>	Adds a node to the end of the receiver's list of child nodes.
<code>insertChild(_: at:)</code>	Inserts a child into a specific position in the receiver's list of child nodes.
<code>removeFrom Parent()</code>	Removes the receiving node from its parent.

3. 构建场景

节点树绘制顺序



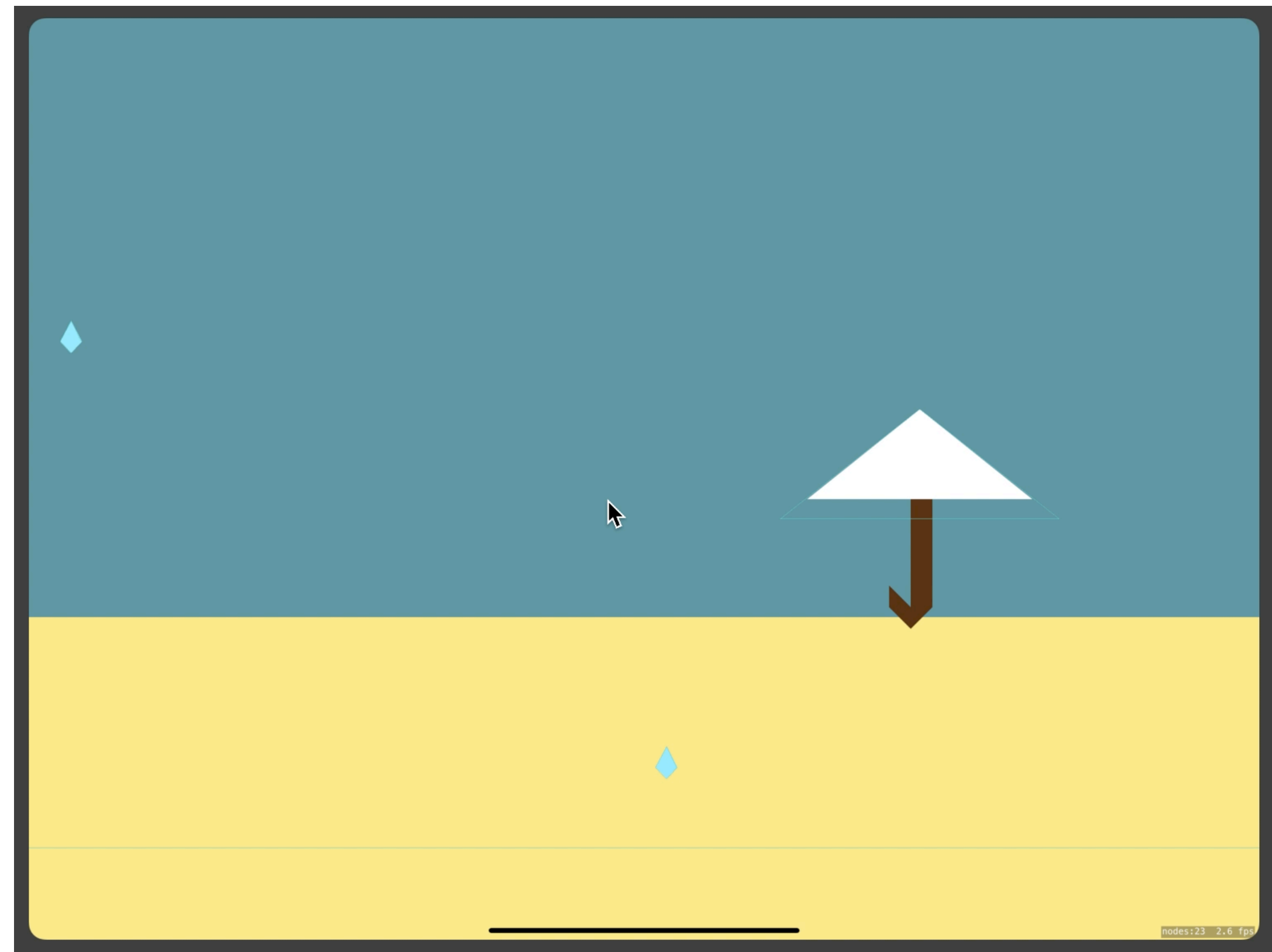
4. 使用场景间过渡

- 默认使用动画过渡
- 过渡发生时，**scene**属性值可能立即更新为指向新的场景，发生动画，移除对旧场景的强引用
- 过渡对象**pausesIncomingScene**和**pausesOutgoingScene**属性允许你在过渡期间播放动画

SpriteKit

5.Demo

- 接下来，本章节实现一个小游戏来巩固 **SpriteKit** 的基础知识。我们会接触到物理引擎（**SKPhysics**）、碰撞、纹理管理、互动以及场景（**SKScene**）。
- 实践之前先看看最终效果：



5.Demo

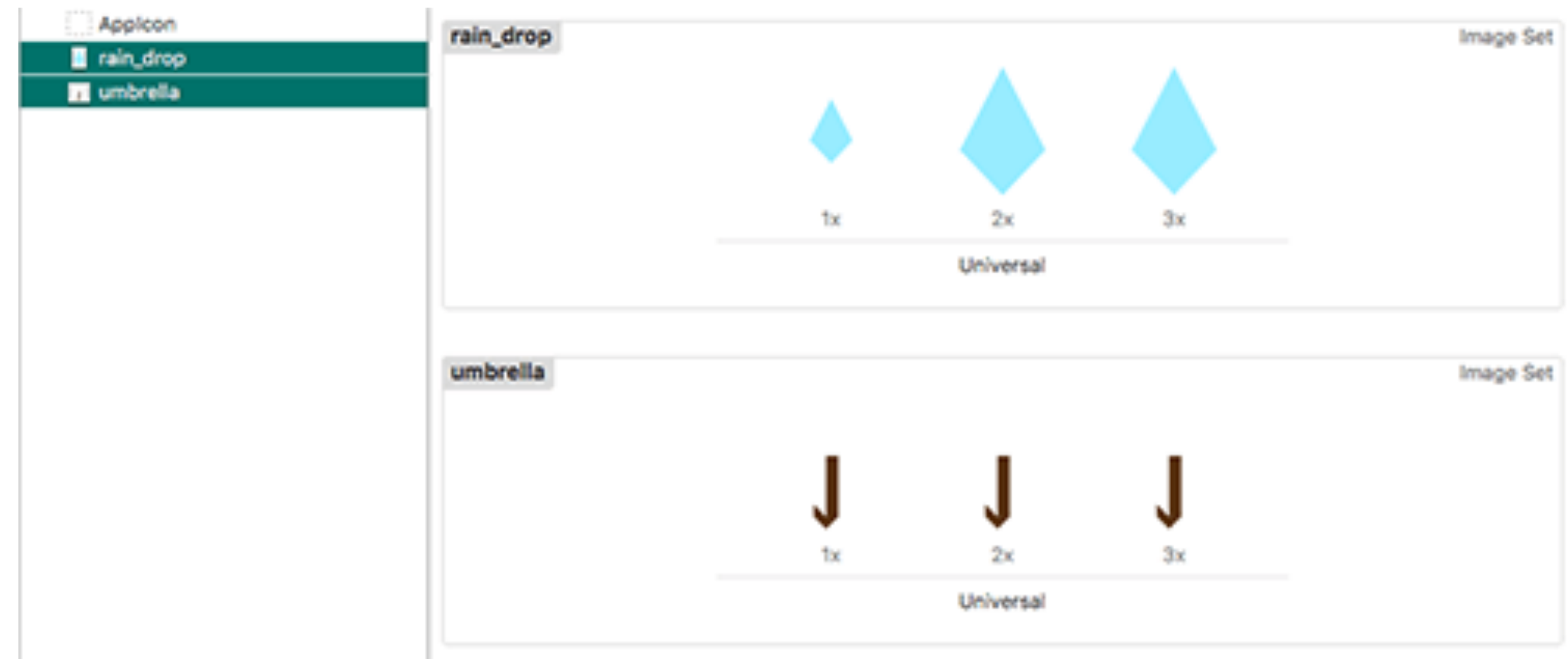
- 这个游戏需要：
 - 添加地面；
 - 添加雨滴；
 - 初始化物理引擎；
 - 添加雨伞对象；
 - 利用 `categoryBitMask` 和 `contactTestBitMask` 来实现碰撞检测；
 - 创建一个全局边界（`world boundary`）来移除落出屏幕的结点（`node`）

SpriteKit

5.Demo

获取资源文件

- 首先我们要获取项目中会用到的资源：雨伞和雨滴。我们先下载好所需要的纹理，将它们添加到 Xcode 左部面板的 **Assets.xcassets** 文件夹中。



5.Demo

创建地面

- 由于背景和地面很简单，我们可以把这些精灵（sprite）放到一个自定义的背景结点（node）中。创建名为 `BackgroundNode.swift` 的 Swift 源文件，并添加以下代码

```
import SpriteKit
public class BackgroundNode : SKNode {
    public func setup(size : CGSize) {
        let yPos : CGFloat = size.height * 0.10
        let startPoint = CGPoint(x: 0, y: yPos)
        let endPoint = CGPoint(x: size.width, y: yPos)
        physicsBody = SKPhysicsBody(
            edgeFrom: startPoint, to: endPoint)
        physicsBody?.restitution = 0.3
    }
}
```


5.Demo

创建地面

- 我们创建的这个对象是一个SKNode实例，我们会把它作为背景元素的容器。我们为其添加了一个SKPhysicsBody实例，这意味着其定义的区域（现在只有一条线），能够和其它的物理实体和物理世界进行交互。

- 接下来我们要把它加入 GameScene 中。切换到 GameScene.swift，添加：

```
private let backgroundNode = BackgroundNode()
```

- 然后，在 sceneDidLoad() 方法中，我们可以初始化背景，并将其加入场景中：

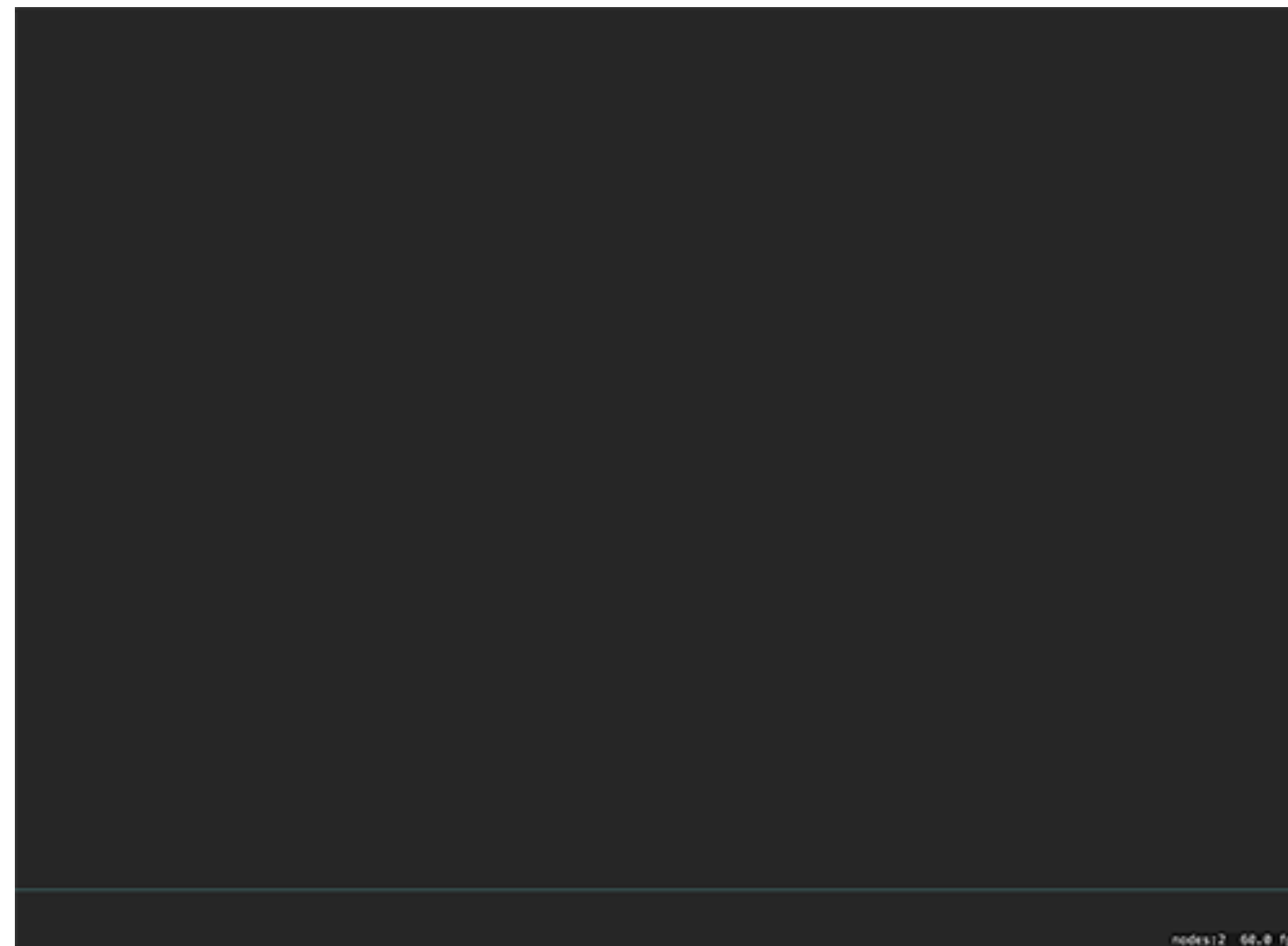
```
backgroundNode.setup(size: size)  
addChild(backgroundNode)
```

SpriteKit

5.Demo

创建地面

- 现在，如果我们运行程序，我们将会看到如图的游戏场景：



5.Demo

创建地面

- 如果你没看见那条线，那说明你在将 node 加入场景时出现了错误，要么就是场景现在不显示物理实体。要控制这些选项的开关，只需要在 `GameViewController.swift`

中修改下列选项即可：

```
if let view = self.view as! SKView? {  
    view.presentScene(sceneNode)  
    view.ignoresSiblingOrder = true  
    view.showsPhysics = true  
    view.showsFPS = true  
    view.showsNodeCount = true  
}
```

- 现在，确保 `showsPhysics` 属性被设为 `true`。这有助于我们调试物理实体。这个背景将会充当雨滴下落反弹时的地面。

5.Demo

添加雨水

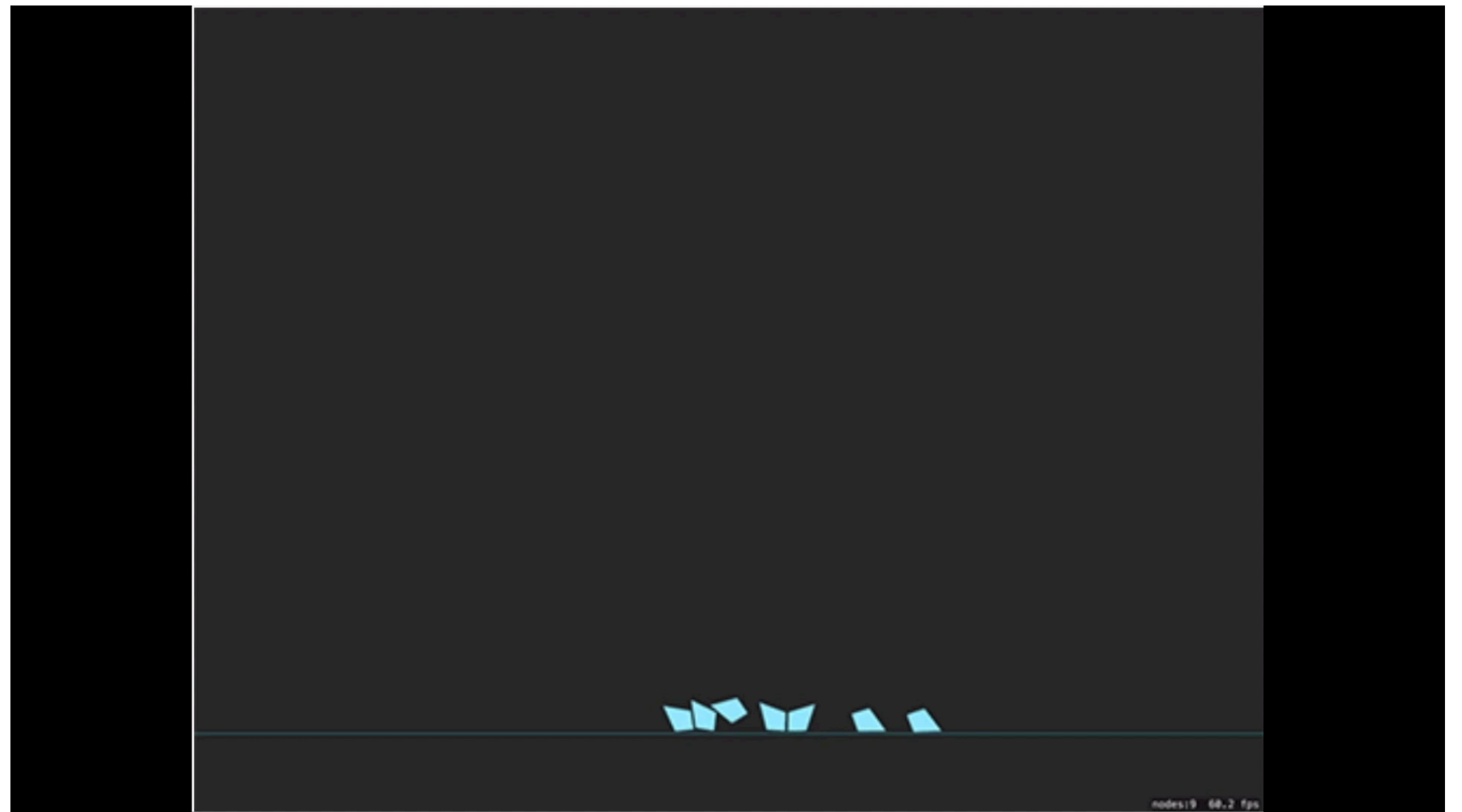
- 在 `GameScene.swift` 中，加入下面这行代码以便创建雨滴：

```
private func spawnRaindrop() {  
    let raindrop = SKSpriteNode(texture:  
raindropTexture)  
    raindrop.physicsBody = SKPhysicsBody(  
        texture: raindropTexture,  
        size: raindrop.size)  
    raindrop.position = CGPoint(  
        x: size.width / 2,  
        y: size.height / 2)  
  
    addChild(raindrop)  
}
```

5.Demo

添加雨水

- 该方法被调用时，会利用我们刚刚创建的 **raindropTexture** 来生成一个新的雨滴结点。然后，将结点位置设置为场景中央，并将其加入场景中。
- 由于我们为雨滴结点添加了 **SKPhysicsBody**，它将会自动受到默认的重力作用并滴落至地面。
- 我们可以在 **touchesBegan**中调用 **spawnRaindrop()**，并看到这样的效果



5.Demo

添加雨水

- 只要我们不断地点击屏幕，雨滴就会源源不断地出现。
- 接下来改变雨水落下的速率，我们把代码从 `touchesBegan` 中删除，并将其绑定到我们的 `update` 循环中。我们希望在 `update` 方法中进行降雨操作。在 `update` 方法的底部，更新 `self.lastUpdateTime` 变量之前，添加如下代码：

```
currentRainDropSpawnTime += dt
if currentRainDropSpawnTime > rainDropSpawnRate {
    currentRainDropSpawnTime = 0
    spawnRaindrop()
}
```


5.Demo

添加雨水

- 但我们不希望所有雨滴都从同一个地方开始往下落。我们可以更新 `spawnRaindrop()` 方法来随机化每个新雨滴的 `x` 坐标，`y` 坐标设置为屏幕顶部。

- 找到 `spawnRaindrop()` 方法中的这行代码：

```
raindrop.position = CGPoint(x: size.width / 2, y: size.height / 2)
```

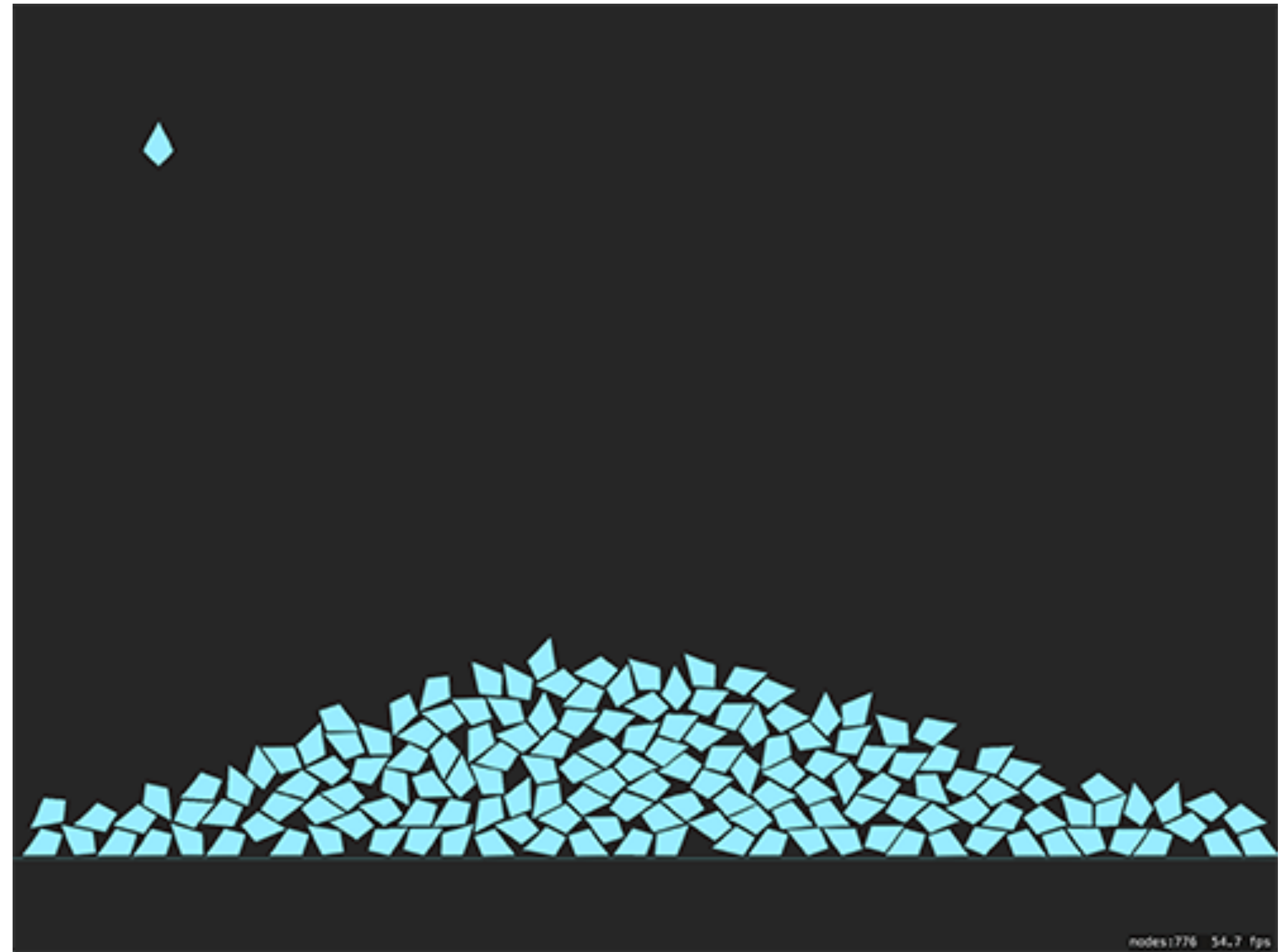
- 将其替换成如下代码：

```
let xPosition =  
    CGFloat(arc4random()).truncatingRemainder(  
        dividingBy: size.width)  
let yPosition = size.height + raindrop.size.height  
  
raindrop.position = CGPoint(x: xPosition, y: yPosition)
```


5.Demo

添加雨水

- 在创建雨滴之后，我们利用 `arc4Random()` 来随机 x 坐标，并通过调用 `truncatingRemainder` 来确保坐标在屏幕范围内。现在运行程序，你应该可以看到这样的效果



5.Demo

监测碰撞

- 雨滴的碰撞包括：
 - 雨滴之间的碰撞
 - 雨滴和地面的碰撞。
- 雨滴碰撞到全局边界的情况，并判断是否要移除雨滴。（我们将引入另一个物理实体来充当全局边界（ world frame ）。任何触碰到边界的对象都会被销毁。）

5.Demo

监测碰撞

- **SKPhysicsBody** 有一个名为 **categoryBitMask** 的属性。这个属性将帮助我们区分互相发生接触的对象。
- 要完成上述工作，我们新创建一个 **Constants.swift** 源文件。这个文件将统一管理我们在整个工程中会用到的硬编码值。在文件中添加如下代码：

```
let WorldCategory : UInt32 = 0x1 << 1
let RainDropCategory : UInt32 = 0x1 << 2
let FloorCategory : UInt32 = 0x1 << 3
```


5.Demo

监测碰撞

- 回到 **BackgroundNode.swift** 文件中，将物理实体更新为刚创建的 **FloorCategory** 。接着，我们还要将地面物理实体设置为可触碰的,故需要将 **RainDropCategory** 添加到地面元素的 **contactTestBitMask** 中。这样我们就能够在雨滴和地面接触时收到回调

```
import SpriteKit
public class BackgroundNode : SKNode {
    public func setup(size : CGSize) {
        let yPos : CGFloat = size.height * 0.10
        let startPoint = CGPoint(x: 0, y: yPos)
        let endPoint = CGPoint(x: size.width, y: yPos)
        physicsBody = SKPhysicsBody(edgeFrom: startPoint,
to: endPoint)
        physicsBody?.restitution = 0.3
        physicsBody?.categoryBitMask = FloorCategory
        physicsBody?.contactTestBitMask =
RainDropCategory
    }
}
```

5.Demo

监测碰撞

- 下一步则是为雨滴元素设置正确的类别，并为其添加可触碰元素。回到 `GameScene.swift` 中，在 `spawnRaindrop` 方法中初始化雨滴物理实体的代码后面添加

```
raindrop.physicsBody?.categoryBitMask = RainDropCategory  
raindrop.physicsBody?.contactTestBitMask = FloorCategory | WorldCategory
```

- 注意，此处我们也添加了 `WorldCategory`，我们此处使用的是位掩码。对于本例中的 `raindrop` 实例，我们希望监听它与 `FloorCategory` 以及 `WorldCategory` 发生碰撞时的信息。现在我们可以 `sceneDidLoad()` 方法中加入全局边界：

```
var worldFrame = frame  
worldFrame.origin.x -= 100  
worldFrame.origin.y -= 100  
worldFrame.size.height += 200  
worldFrame.size.width += 200  
self.physicsBody = SKPhysicsBody(edgeLoopFrom: worldFrame)  
self.physicsBody?.categoryBitMask = WorldCategory
```


5.Demo

监测碰撞

- 在上述代码中，我们创建了一个和场景形状相同的边界，只不过我们将每个边都扩张了 100，相当于创建了一个缓冲区，使得元素在离开屏幕后才会被销毁。注意我们使用的 `edgeLoopFrom`，它创建了一个矩形，其边界可以和其它元素发生碰撞。
- 现在，一切用于检测碰撞的准备都已经就绪了，我们只需要监听它就可以了。为我们的游戏场景添加对 `SKPhysicsContactDelegate` 协议的支持。

```
class GameScene: SKScene, SKPhysicsContactDelegate {
```

5.Demo

监测碰撞

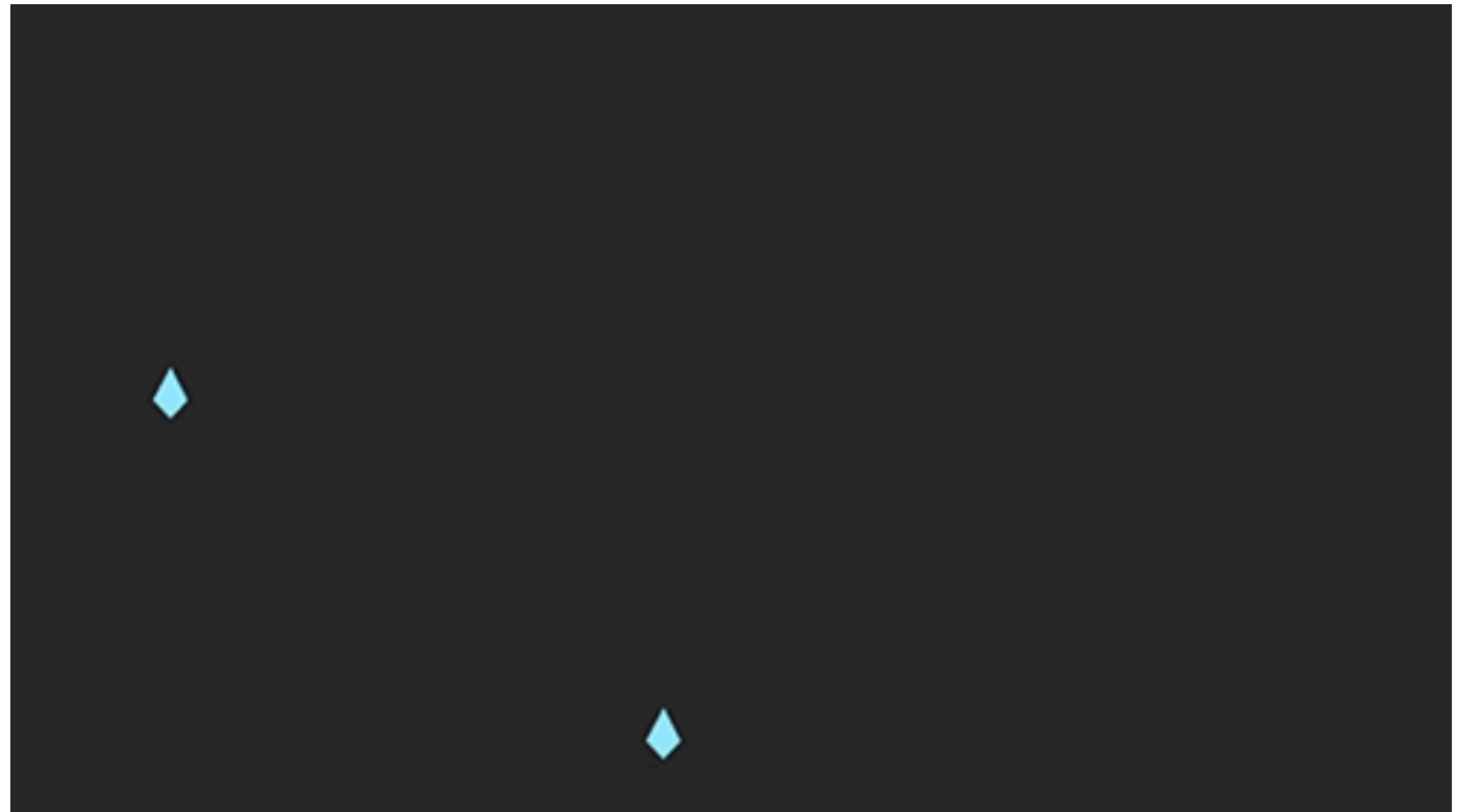
- 接着，我们需要实现 `SKPhysicsContactDelegate` 中的一个方法，`didBegin`。每当带有我们预先设置的 `contactTestBitMasks` 的物体碰撞发生时，这个方法就会被调用。在 `GameScene.swift` 的底部，加入如下代码：

```
func didBegin(_ contact: SKPhysicsContact) {  
    if (contact.bodyA.categoryBitMask == RainDropCategory) {  
        contact.bodyA.node?.physicsBody?.collisionBitMask = 0  
        contact.bodyA.node?.physicsBody?.categoryBitMask = 0  
    } else if (contact.bodyB.categoryBitMask == RainDropCategory) {  
        contact.bodyB.node?.physicsBody?.collisionBitMask = 0  
        contact.bodyB.node?.physicsBody?.categoryBitMask = 0  
    }  
}
```


5.Demo

监测碰撞

- 当一滴雨滴和任何其它对象的边缘发生碰撞后，我们会将其碰撞掩码（collision bitmask）清零。这样做可以避免雨滴在初次碰撞后反复与其它对象碰撞



5.Demo

更新背景

- 目前为止，背景结点都非常的简单。它只是一个 **SKPhysicsBody**，也就是一条线。
我们可以通过创建两个 **SKShapeNode** 达到天空和地面的效果。
- 打开 **BackgroundNode.swift** 并在 **setup(size)** 方法中，初始化 **SKPhysicsBody** 的
下面添加如下代码

5.Demo

更新背景

```
let skyNode = SKShapeNode(rect: CGRect(origin: CGPoint(), size: size))
skyNode.fillColor = SKColor(red:0.38, green:0.60, blue:0.65, alpha:1.0)
skyNode.strokeColor = SKColor.clear
skyNode.zPosition = 0
let groundSize = CGSize(width: size.width, height: size.height * 0.35)
let groundNode = SKShapeNode(rect: CGRect(origin: CGPoint(), size:
groundSize))
groundNode.fillColor = SKColor(red:0.99, green:0.92, blue:0.55,
alpha:1.0)
groundNode.strokeColor = SKColor.clear
groundNode.zPosition = 1
addChild(skyNode)
addChild(groundNode)
```


5.Demo

更新背景

- `skyNode` 的 `zPosition` 设为 0，而地面结点的 `zPosition` 为 1，在渲染时地面就会始终在天空之上。
- 但雨滴的 `zPosition` 存在问题。现在雨滴会被渲染在地面之下。回到 `GameScene.swift` 中，更新 `spawnRaindrop()` 方法中雨滴的 `zPosition`，使之渲染在地面之上。在 `spawnRaindrop()` 方法中，加入下列代码：

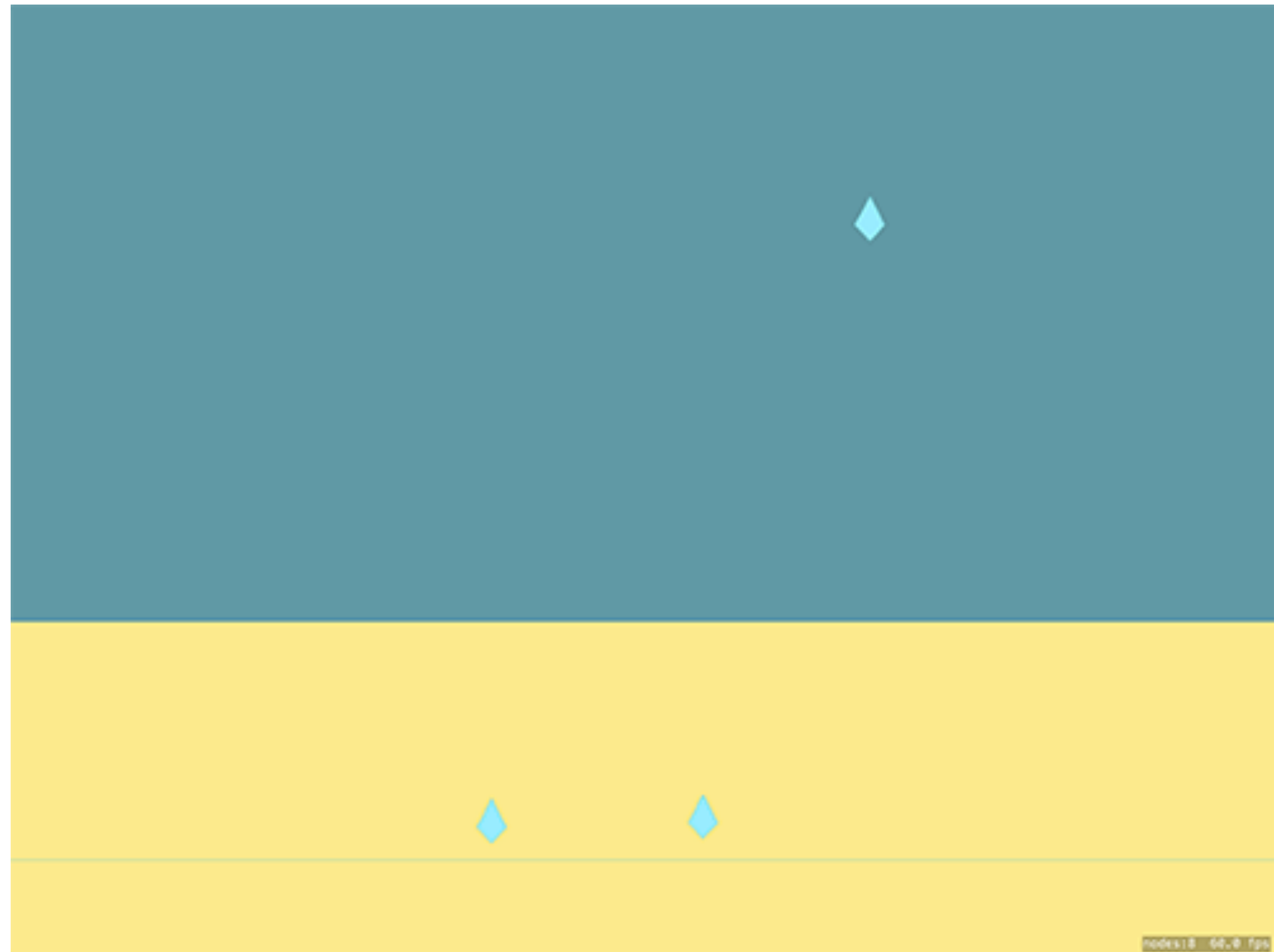
```
raindrop.zPosition = 2
```

SpriteKit

5.Demo

更新背景

— 背景效果



5.Demo

添加交互

- 添加 UmbrellaSprite.swift 源文件，并添加下列代码以生成雨伞的雏形

```
import SpriteKit

public class UmbrellaSprite : SKSpriteNode {
    public static func newInstance() -> UmbrellaSprite {
        let umbrella = UmbrellaSprite(imageNamed: "umbrella")
        return umbrella
    }
}
```


5.Demo

添加交互

- 目前，我们只是使用一个静态方法创建了一个新的精灵结点（sprite node），我们需要再为其添加一个 **physicsBody**。我们在 **newInstance()** 方法中构造一个 **CGPath** 来初始化 **SKPhysicsBody**。

```
let path = UIBezierPath()
path.move(to: CGPoint())
path.addLine(to: CGPoint(x: -umbrella.size.width / 2 - 30, y: 0))
path.addLine(to: CGPoint(x: 0, y: umbrella.size.height / 2))
path.addLine(to: CGPoint(x: umbrella.size.width / 2 + 30, y: 0))
umbrella.physicsBody = SKPhysicsBody(polygonFrom: path.cgPath)
umbrella.physicsBody?.isDynamic = false
umbrella.physicsBody?.restitution = 0.9
```


5.Demo

添加交互

- 目前，我们只是使用一个静态方法创建了一个新的精灵结点（sprite node），我们需要再为其添加一个 **physicsBody**。我们在 **newInstance()** 方法中构造一个 **CGPath** 来初始化 **SKPhysicsBody**。

```
let path = UIBezierPath()  
path.move(to: CGPoint())  
path.addLine(to: CGPoint(x: -umbrella.size.width / 2 - 30, y: 0))  
path.addLine(to: CGPoint(x: 0, y: umbrella.size.height / 2))  
path.addLine(to: CGPoint(x: umbrella.size.width / 2 + 30, y: 0))  
umbrella.physicsBody = SKPhysicsBody(polygonFrom: path.cgPath)  
umbrella.physicsBody?.isDynamic = false  
umbrella.physicsBody?.restitution = 0.9
```

SpriteKit

5.Demo

添加交互

- 然后我们会得到这样一个雨伞的实体



5.Demo

添加交互

- 现在，到 `GameScene.swift` 中来初始化雨伞对象并将其加入场景中。在文件顶部，类变量的下方，加入下面的代码：

```
private let umbrellaNode = UmbrellaSprite.newInstance()
```

- 接着，在 `sceneDidLoad()` 中，将 `backgroundNode` 加入场景的下面，加入如下代码来将雨伞放置在屏幕中央：

```
umbrellaNode.position = CGPoint(x: frame.midX, y: frame.midY)
umbrellaNode.zPosition = 4
addChild(umbrellaNode)
```

- 完成上述操作后，再运行程序，你就能看见雨伞了，同时你会发现雨滴将会被雨伞弹开

5.Demo

手势响应

- 为雨伞添加手势响应，与之相关的是 `GameScene.swift` 中的空方法 `touchesBegan` 和 `touchesMoved`。有三种解决方案：
 1. 在两个方法中都直接根据当前的触摸来更新雨伞的位置，雨伞将会从屏幕的一个位置瞬间移动到另一位置。
 2. 实时设置 `UmbrellaSprite` 对象的终点，并且在 `update` 方法被调用时，逐步向终点方向移动。
 3. 在 `touchesBegan` 或 `touchesMoved` 中通过设置一系列 `SKAction` 来移动 `UmbrellaSprite`，但不推荐这么做。这样做会导致 `SKAction` 对象被频繁创建和销毁，使得性能变差。

5.Demo

手势响应

- 我们这里选择第二个解决方案。 **UmbrellaSprite** 的代码改成下面这样

```
//保存对象移动的终点位置
private var destination : CGPoint!
private let easing : CGFloat = 0.1
//进行刷新操作之前直接对 position 属性进行赋值,以直接移动雨伞
public func updatePosition(point : CGPoint) {
    position = point
    destination = point
}
//缓冲雨伞的移动
public func setDestination(destination : CGPoint) {
    self.destination = destination
}
```

5.Demo

手势响应

- UmbrellaSprite 的 update 代码改成下面这样

```
//计算我们需要向终点方向移动多少距离的逻辑
public func update(deltaTime : TimeInterval) {
    let distance = sqrt(pow((destination.x - position.x), 2) +
pow((destination.y - position.y), 2))
    if(distance > 1) {
        let directionX = (destination.x - position.x)
        let directionY = (destination.y - position.y)
        position.x += directionX * easing
        position.y += directionY * easing
    } else {
        position = destination;
    }
}
```


5.Demo

手势响应

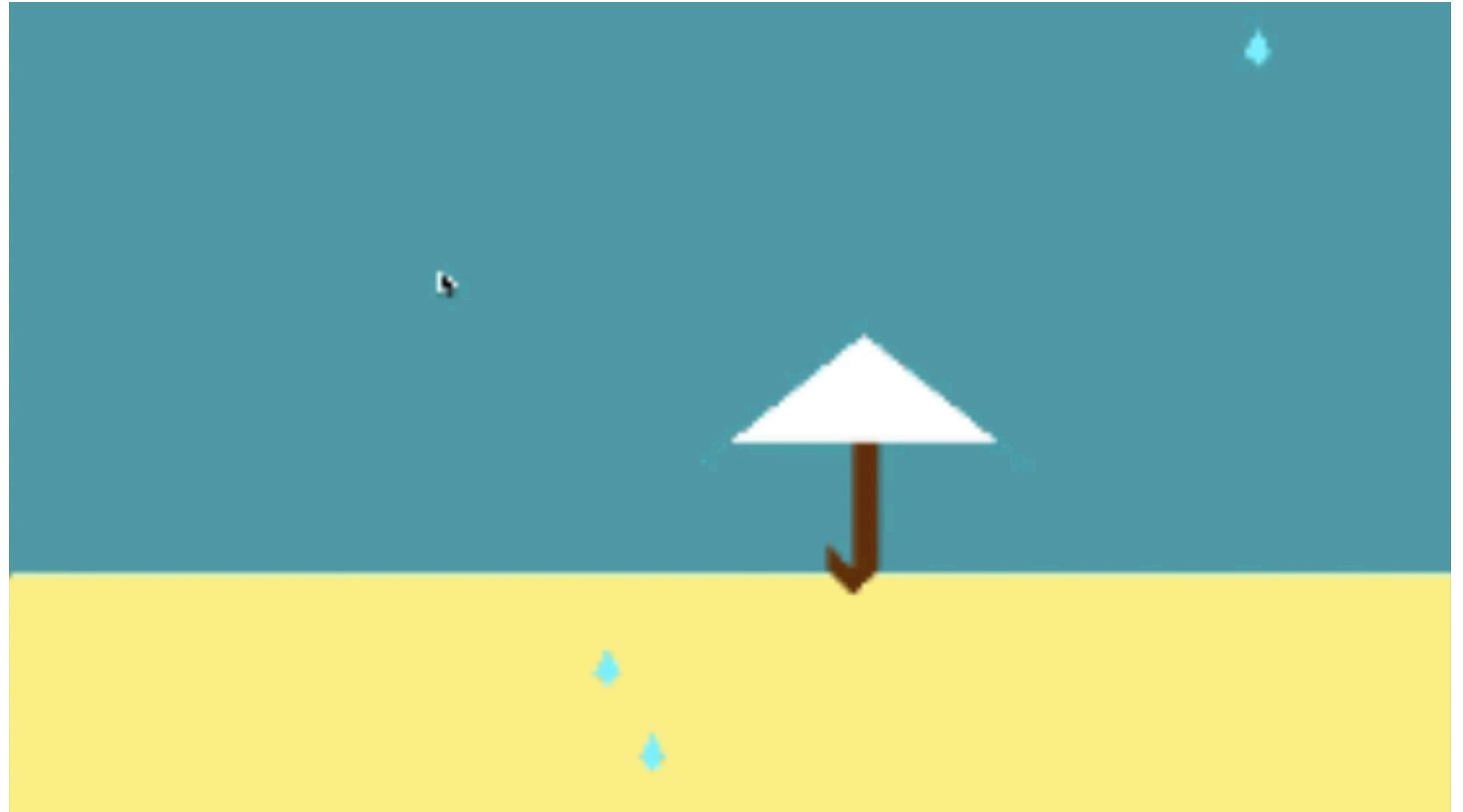
- 回到 `GameScene.swift` 中, 修改 `touchesBegan` 和 `touchesMoved`

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {  
    let touchPoint = touches.first?.location(in: self)  
    if let point = touchPoint {  
        umbrellaNode.setDestination(destination: point)  
    }  
}  
  
override func touchesMoved(_ touches: Set<UITouch>, with event: UIEvent?) {  
    let touchPoint = touches.first?.location(in: self)  
    if let point = touchPoint {  
        umbrellaNode.setDestination(destination: point)  
    }  
}
```


SpriteKit

5.Demo

最终效果



课程目录



1

SpriteKit 基础

2

GPU 并行计算概念

3

Metal 基础

GPU 并行计算概念

GPU VS CPU

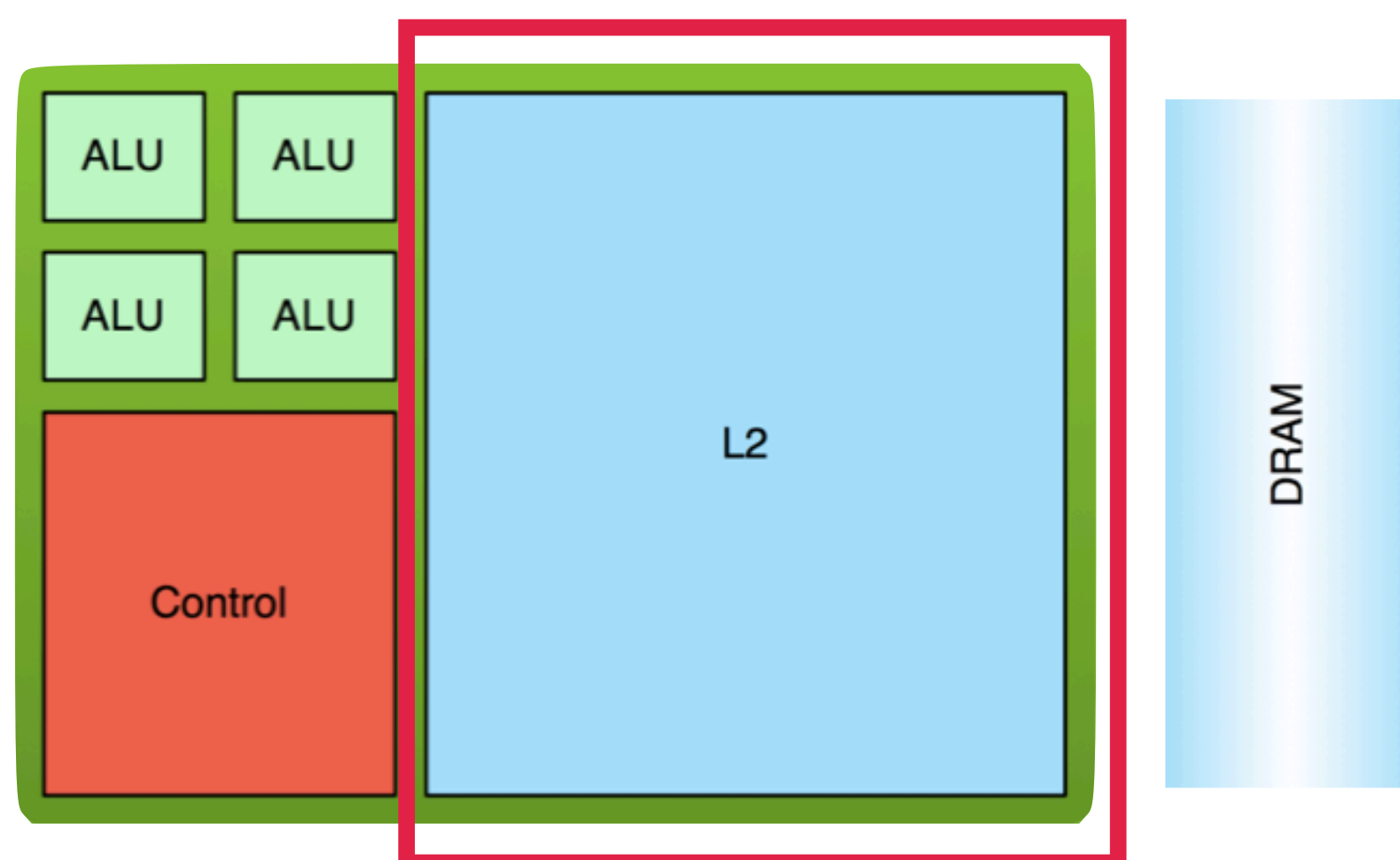
- 无论是CPU还是GPU，在进行计算时，都需要用 Core 来做算术逻辑运算。核心中有ALU（逻辑运算单元）和寄存器等电路。
- GPU 并行编程的核心在于**线程**，一个线程就是程序中的一个单一指令流，一个个线程组合在一起就构成了并行计算网格，成为了并行的程序。右图展示了多核 CPU 与 GPU 的计算网格



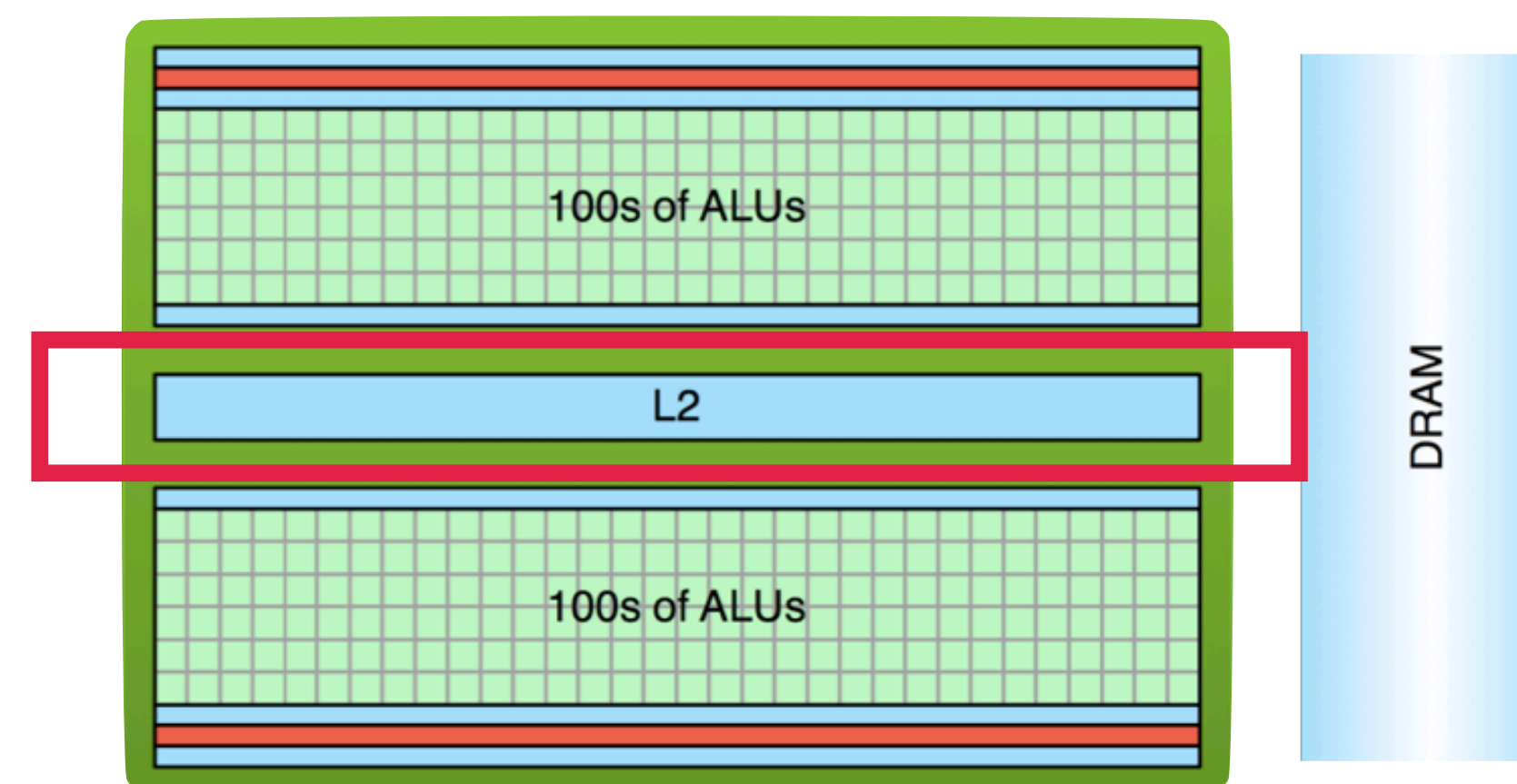
Figure 3 The GPU Devotes More Transistors to Data Processing

GPU 并行计算概念

GPU VS CPU



- CPU:
 - 读写缓存更快（缓存模块大），读写主存稍慢
 - 更适合乱序执行

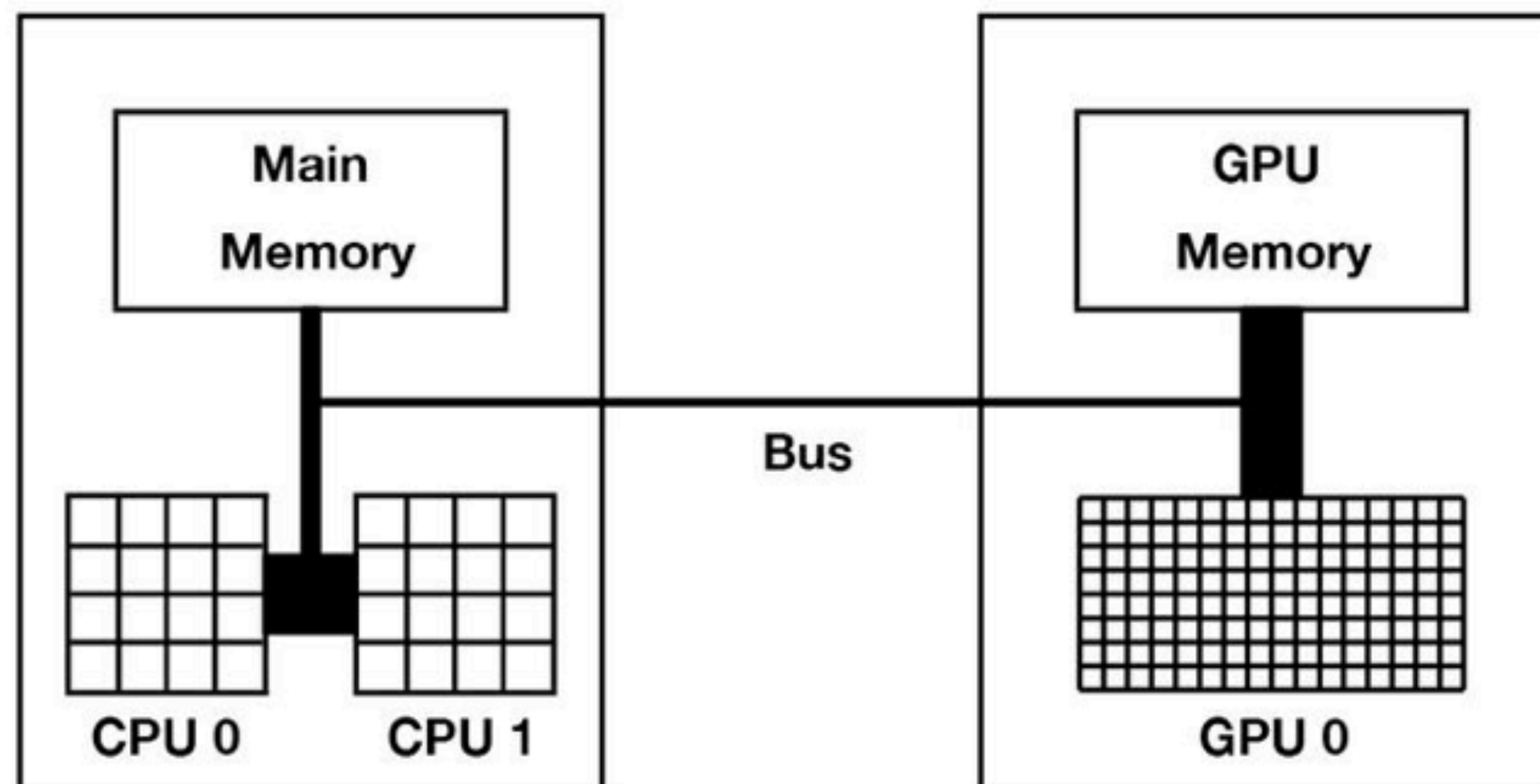


- GPU:
 - 适合大量数据并行、数据吞吐率大的计算
 - 主要是计算模块（缓存模块小）

GPU 并行计算概念

GPU VS CPU

- 通常个人电脑只有2-8个核心，而GPU则有成百上千个核心
- CPU的单核处理能力更强，但是GPU则可以靠数据并行取胜
- 大量计算时，CPU主要从主存（Main Memory）中读写数据，并通过总线（Bus）与GPU交互。GPU除了有超多计算核心外，也有自己独立的存储，被称之为显存。



GPU 并行计算概念

Demo: 数组加法

- 我们从一个数组数组开始说起:
 - 这是一个在CPU上运行的C语言数组加法程序
 - 显然，整个数组的计算是顺序执行的

```
void add_arrays(const float* inA,  
               const float* inB,  
               float* result,  
               int length)  
{  
    for (int index = 0; index < length ; index++)  
    {  
        result[index] = inA[index] + inB[index];  
    }  
}
```


GPU 并行计算概念



Demo: 数组加法

- 下面的是一个GPU上运行的Kernel程序:
 - For循环消失了
 - 每个kernel只进行一个单元的计算
 - Index使用传入的参数获得

```
void add_arrays(const float* inA,
               const float* inB,
               float* result,
               int length)
{
    for (int index = 0; index < length ; index++)
    {
        result[index] = inA[index] + inB[index];
    }
}
```

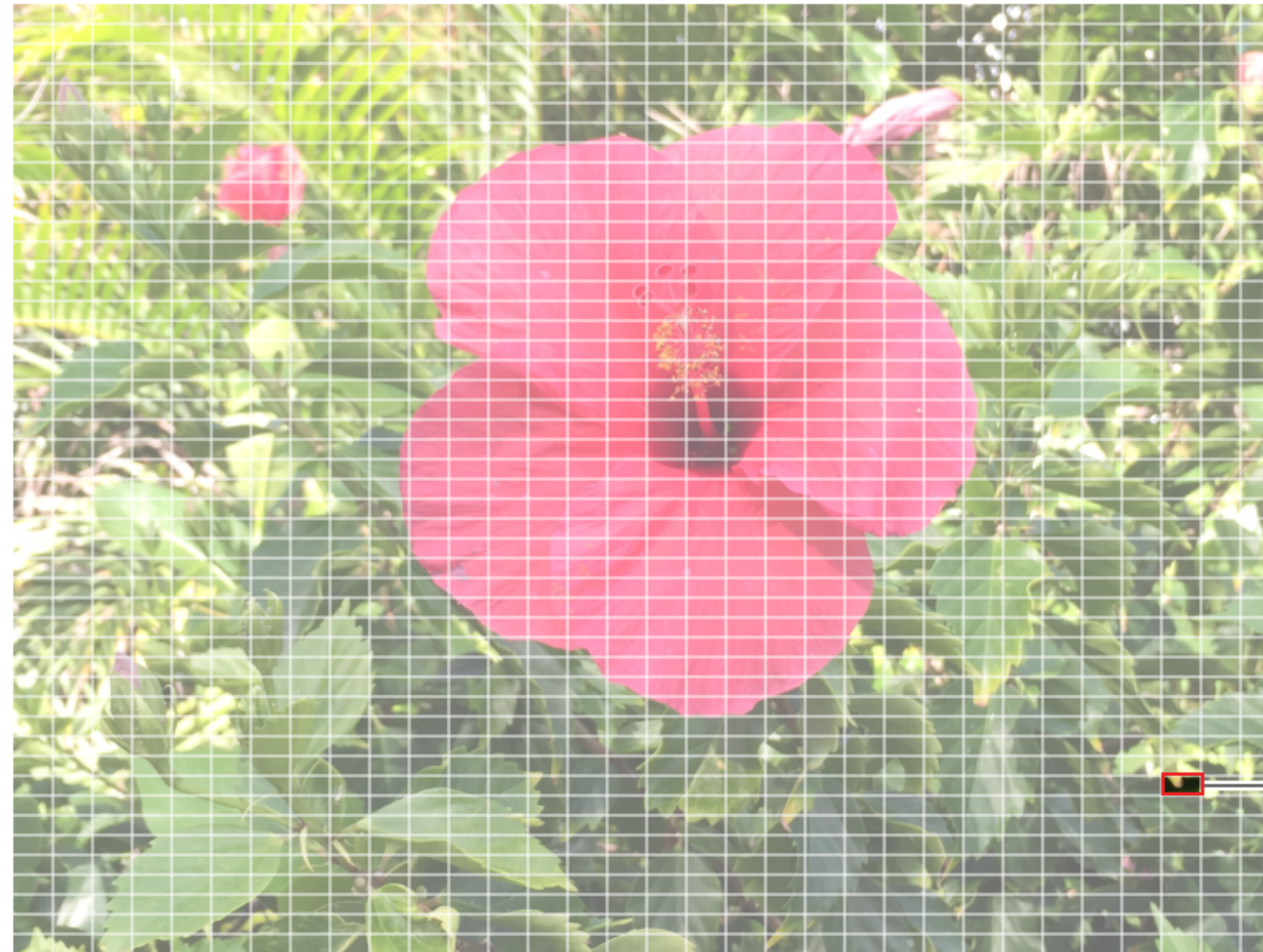
```
kernel void add_arrays(device const float* inA,
                      device const float* inB,
                      device float* result,
                      uint index
                      [[thread_position_in_grid]])
{
    result[index] = inA[index] + inB[index];
}
```


GPU 并行计算概念

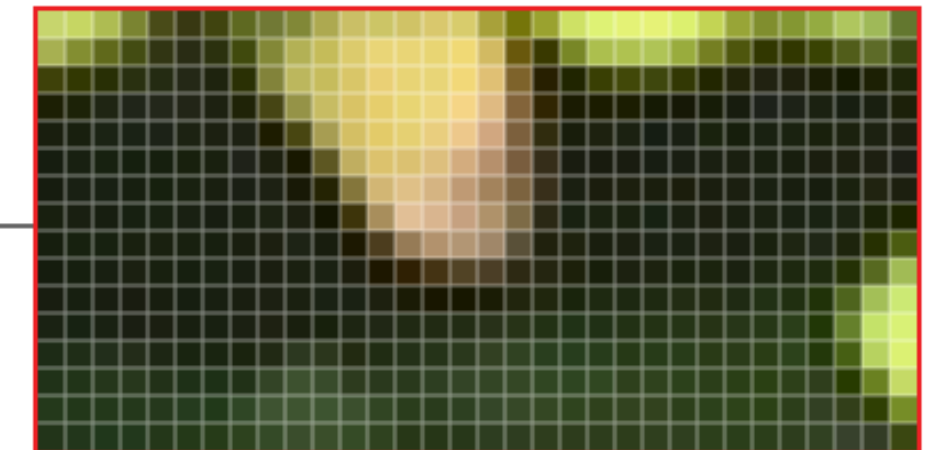
Metal GPU 编程概念

- **Grid**
 - Grid是所有要计算的子任务
 - 图像处理中就是所有像素单元
- **Thread**
 - 每个GPU Kernel进行计算的实例
- **ThreadGroup**
 - 数十个Thread会以一组ThreadGroup的形式组织起来
 - 一组ThreadGroup中的Thread是一起执行的

Grid: 1024 x 768 pixels / 32 x 48 threadgroups



Threadgroup: 32 x 16 threads



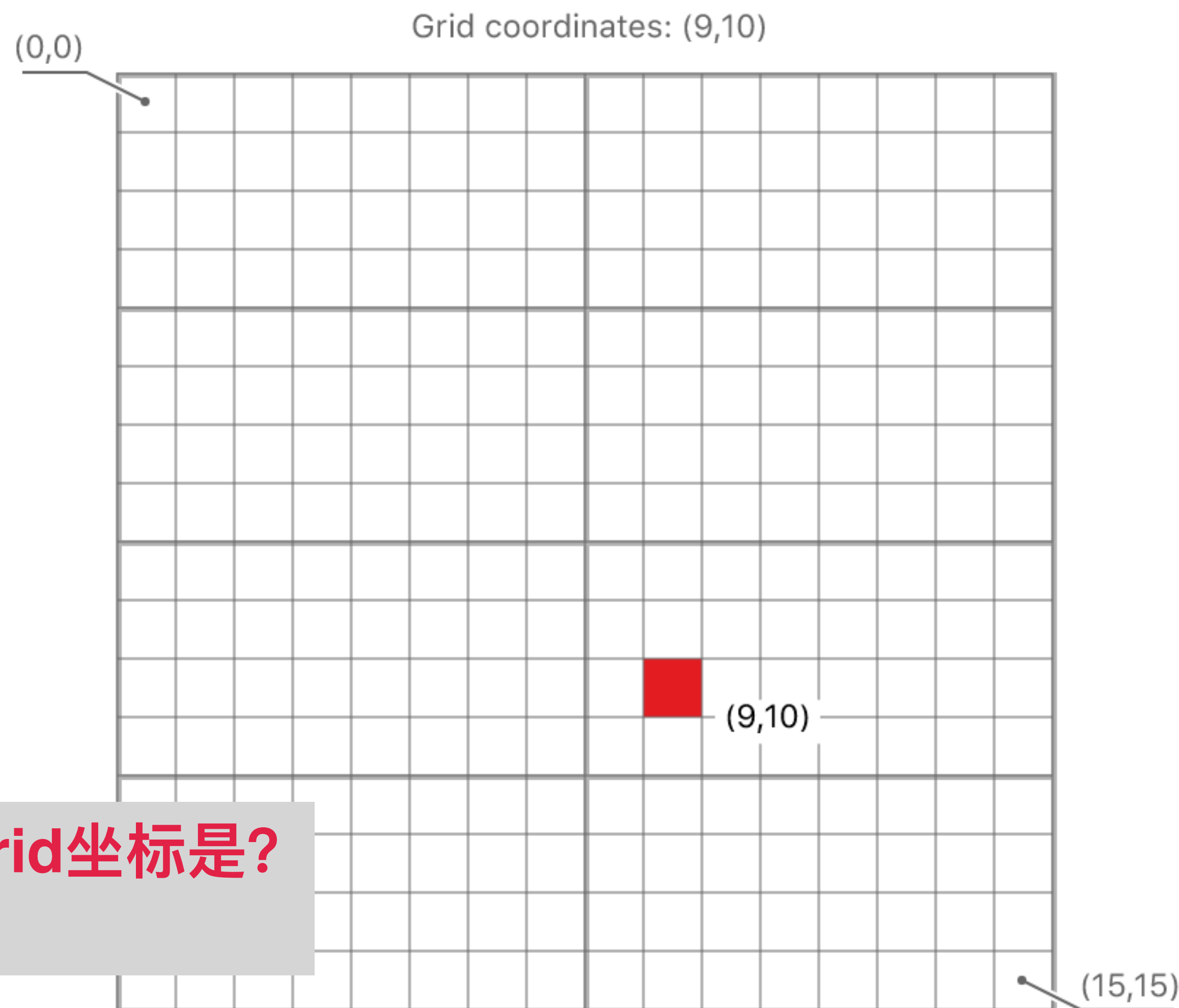
GPU 并行计算概念

Metal GPU 编程概念

- 在 Metal 中，你可以使用
- `[[ thread_position_in_threadgroup ]]`
 - thread相对于threadgroup的局部坐标 (1, 2)
- `[[ thread_position_in_grid ]]`
 - thread相对于Grid的全局坐标 (9, 10)
- `[[ threadgroup_position_in_grid ]]`
 - threadgroup本身相对于grid的坐标 (思考题)

思考题🤔：红点所在的ThreadGroup相对Grid坐标是？
(起始组坐标为0, 0)

(1, 2)



GPU 并行计算概念

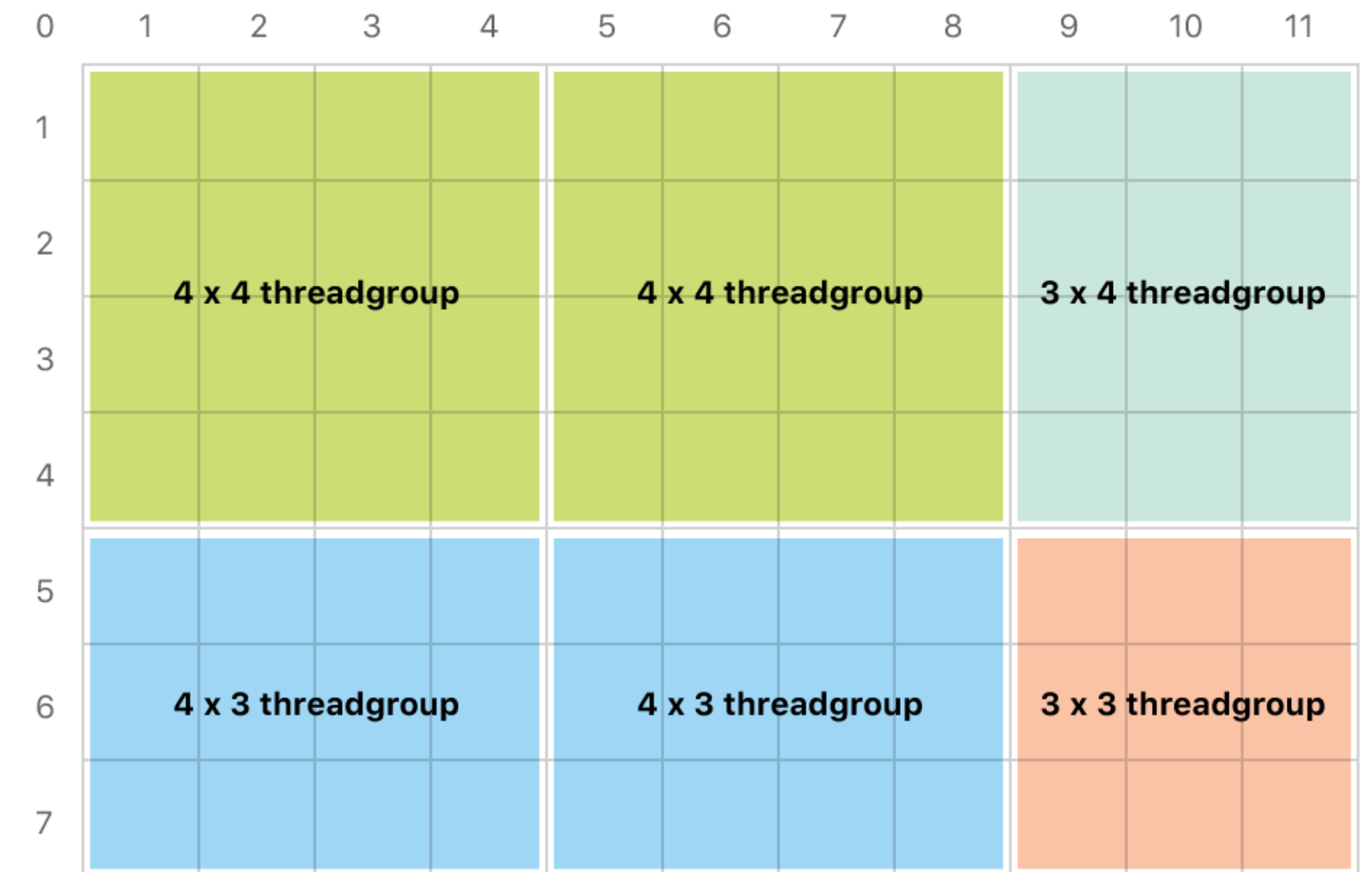
Metal GPU 编程概念

- 考虑这样一个11 * 7 的Grid。划分Group之前，我们需要首先计算：

```
let w = pipelineState.threadExecutionWidth
let h = pipelineState.maxTotalThreadsPerThreadgroup / w
let threadsPerThreadgroup = MTLSizeMake(w, h, 1)
```

- **threadExecutionWidth** （真实设备一般为32）
 - GPU上一个SIMD最多并行执行的thread数量
- **maxTotalThreadsPerThreadgroup** （因GPU而异）
 - 单个threadgroup中最多的thread数量

设置好 **gridSize** 和 **threadsPerThreadgroup** 之后，就可以直接传入 **computeCommandEncoder.dispatchThreads**



GPU 并行计算概念

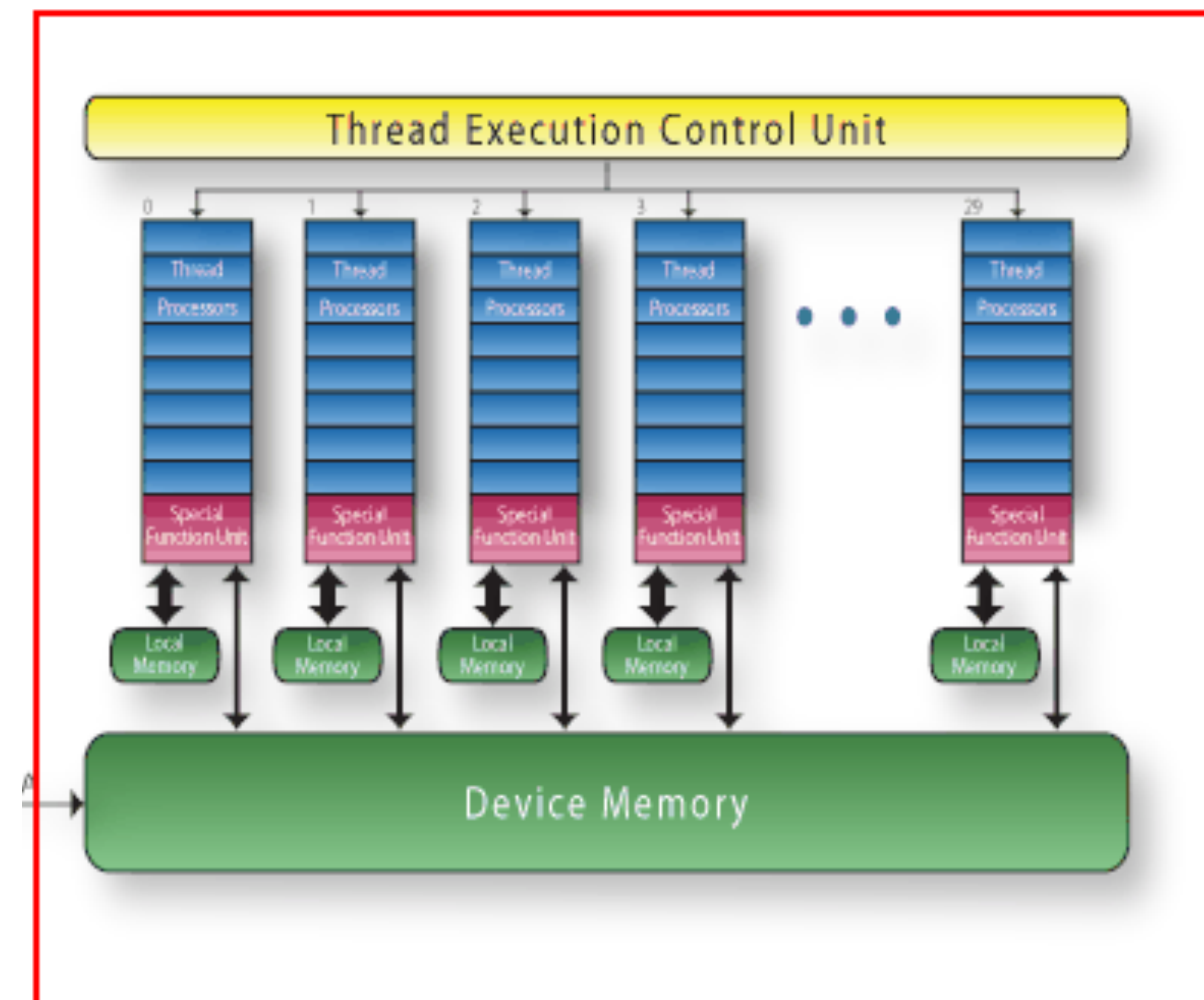


浙江大学
ZHEJIANG UNIVERSITY

GPU computer

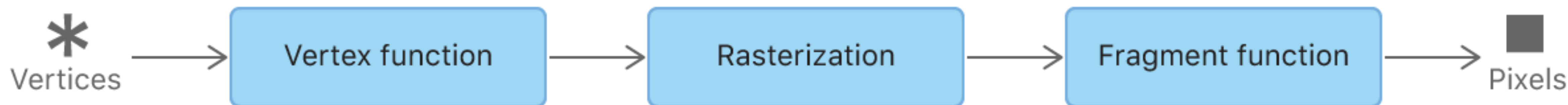
Metal GPU 编程概念

- **threadExecutionWidth** (真实设备一般为32)
 - GPU上一个SIMD最多并行执行的thread数量
- **maxTotalThreadsPerThreadgroup** (因GPU而异)
 - 单个threadgroup中最多的thread数量
- GPU硬件是会划分成多个SIMD模块, 所以需要区分 **maxTotalThreadsPerThreadgroup** 和 **threadExecutionWidth**



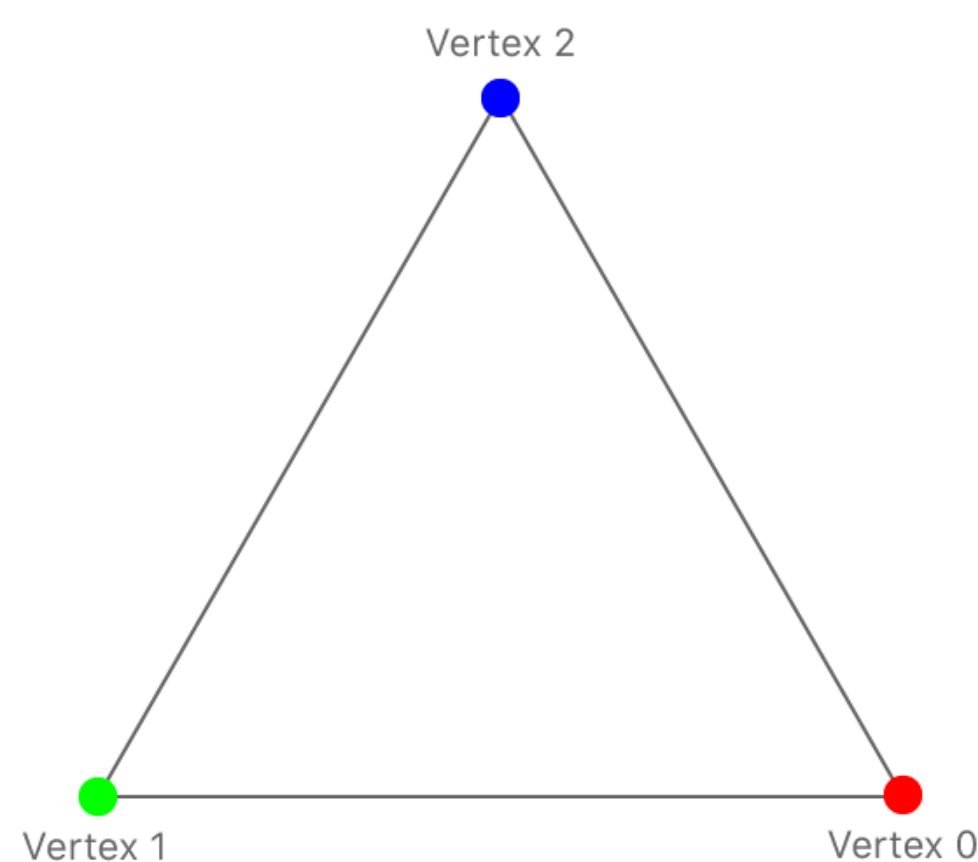
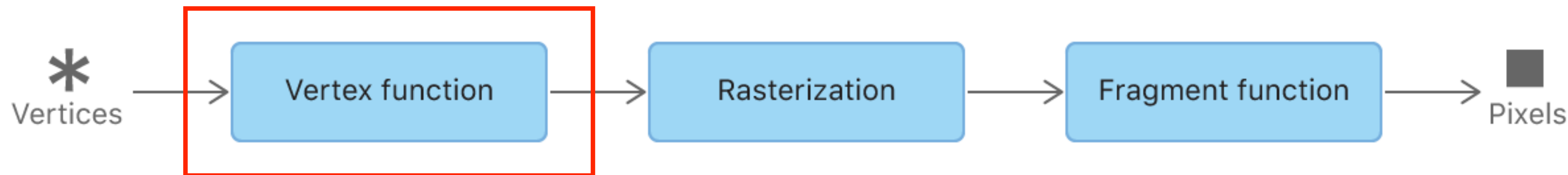
GPU 绘制

Metal GPU 绘制流程

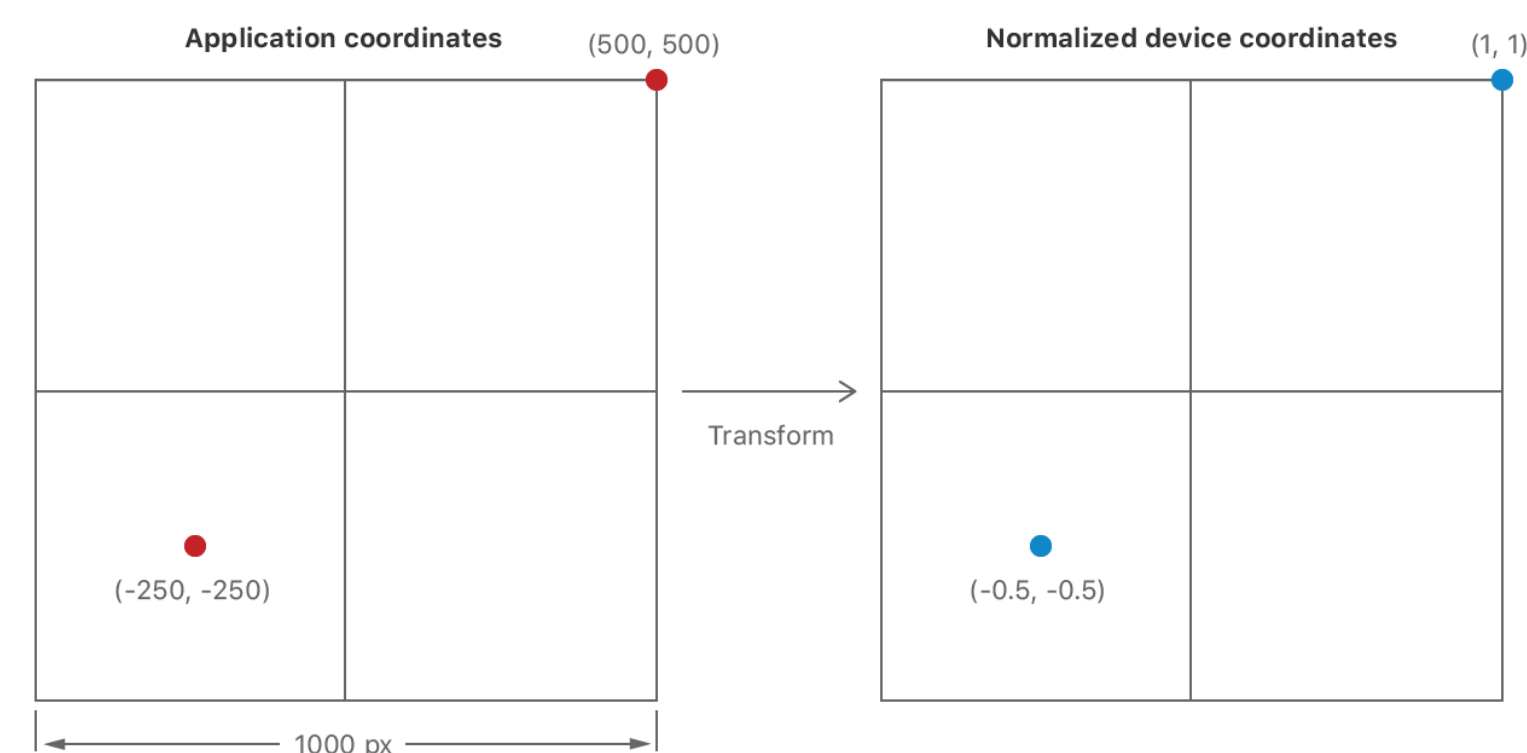


- Metal 的绘制流水线和其他API（例如OpenGL）类似
- 根据需要，绘制流水线可能有多个阶段。最主要的三个阶段是：顶点着色器、光栅化、片元着色器。

GPU 绘制

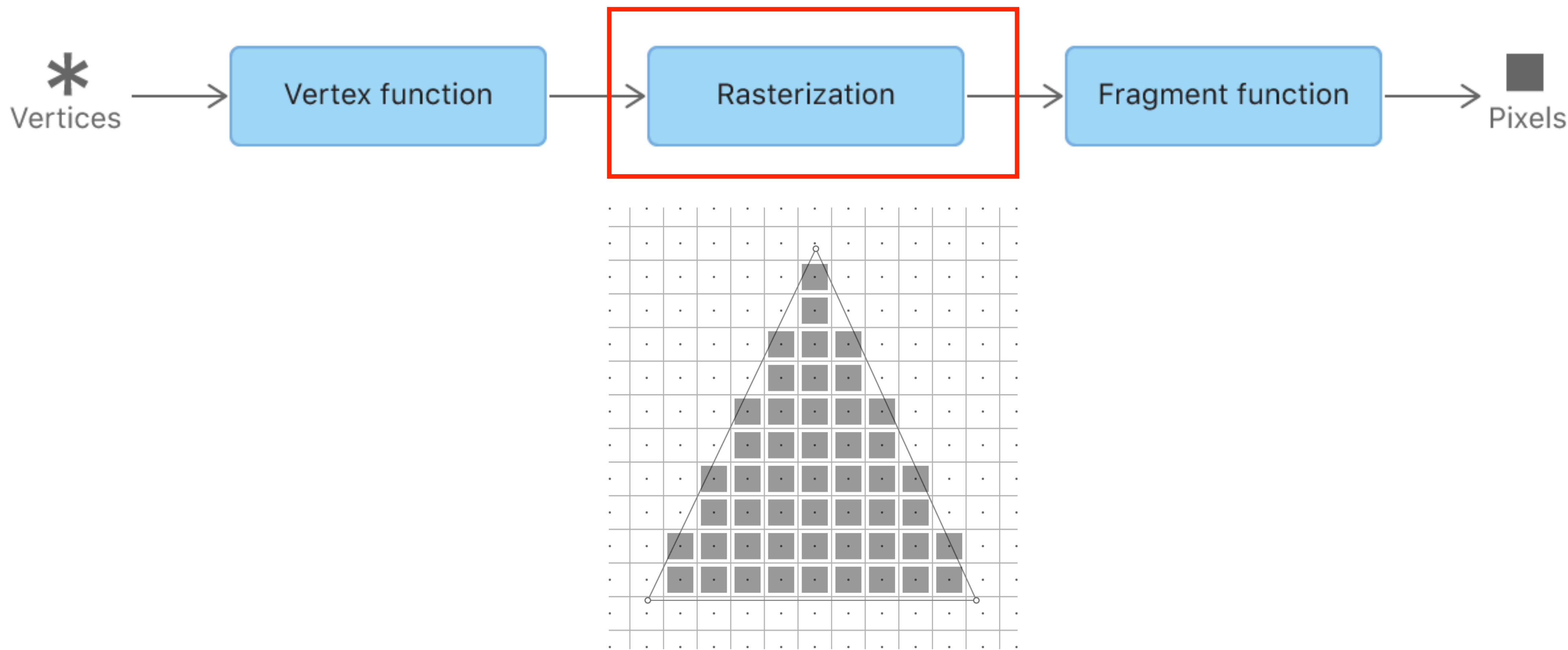


场景中的点

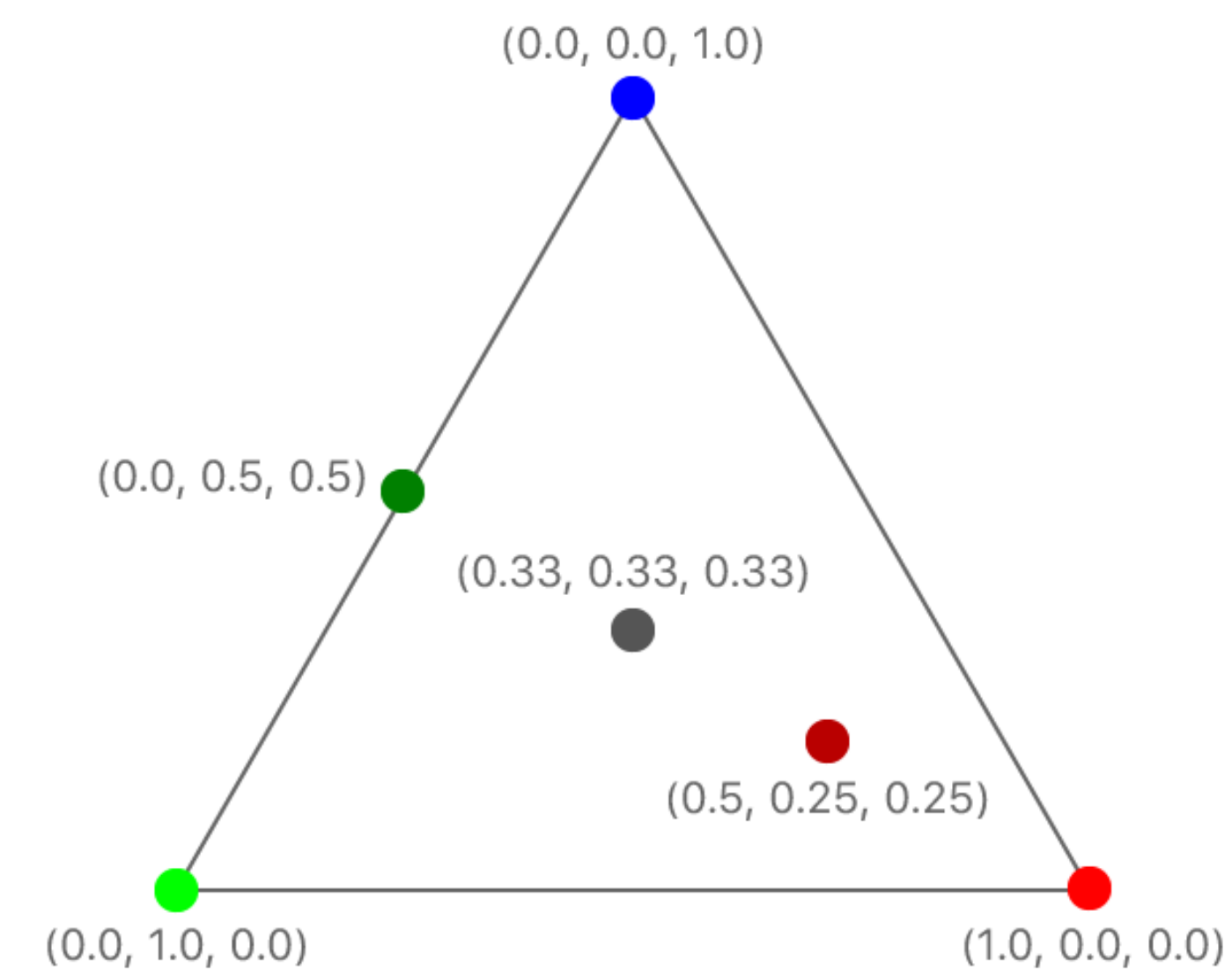
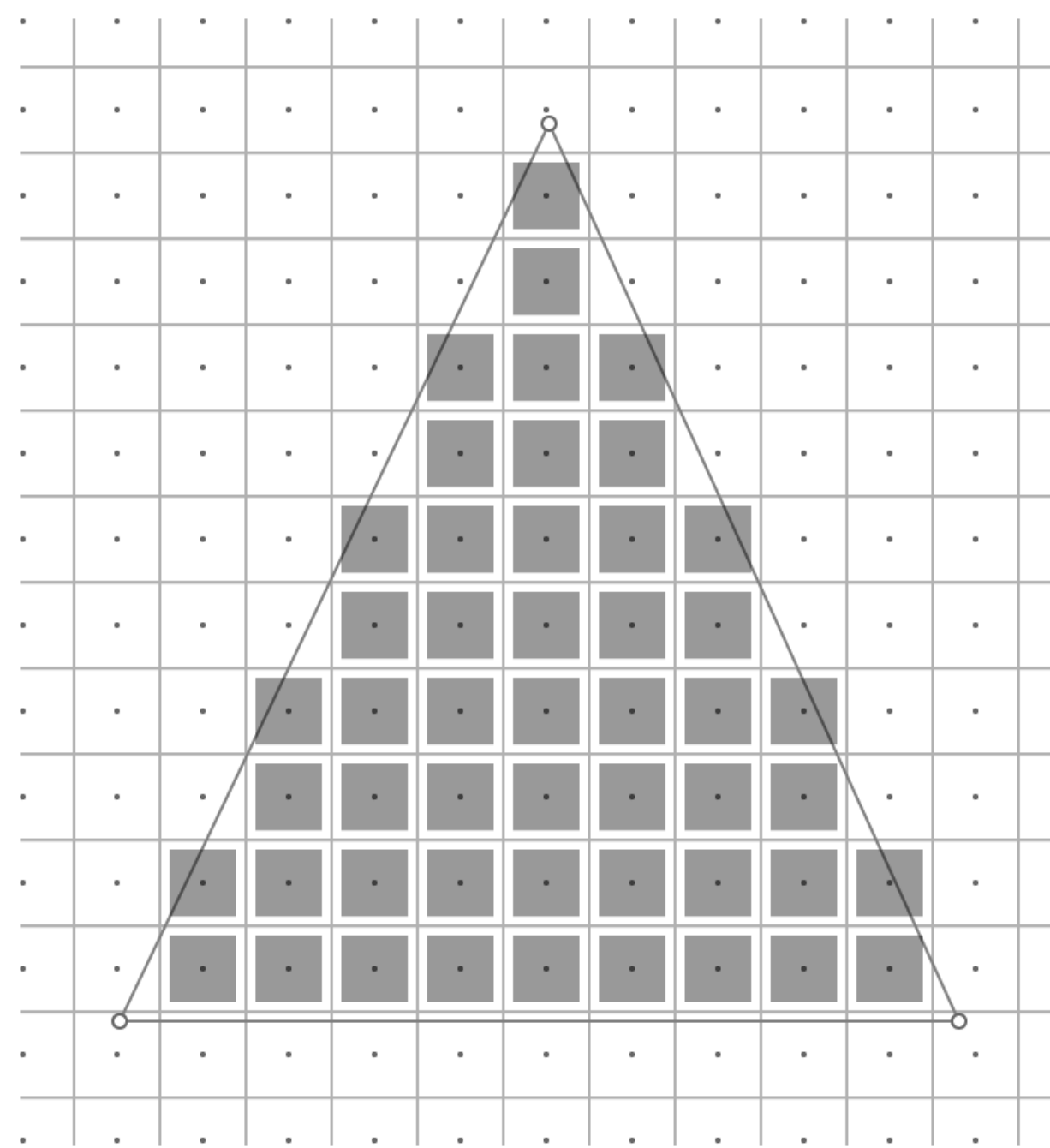
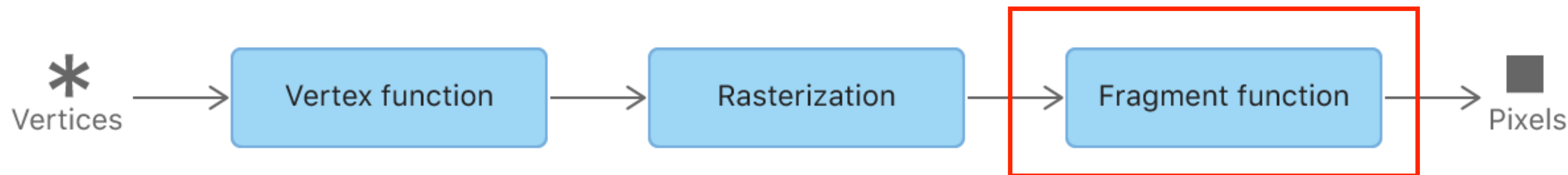


屏幕上的点

GPU 绘制



GPU 绘制



课程目录



1

SpriteKit 基础

2

GPU 并行计算概念

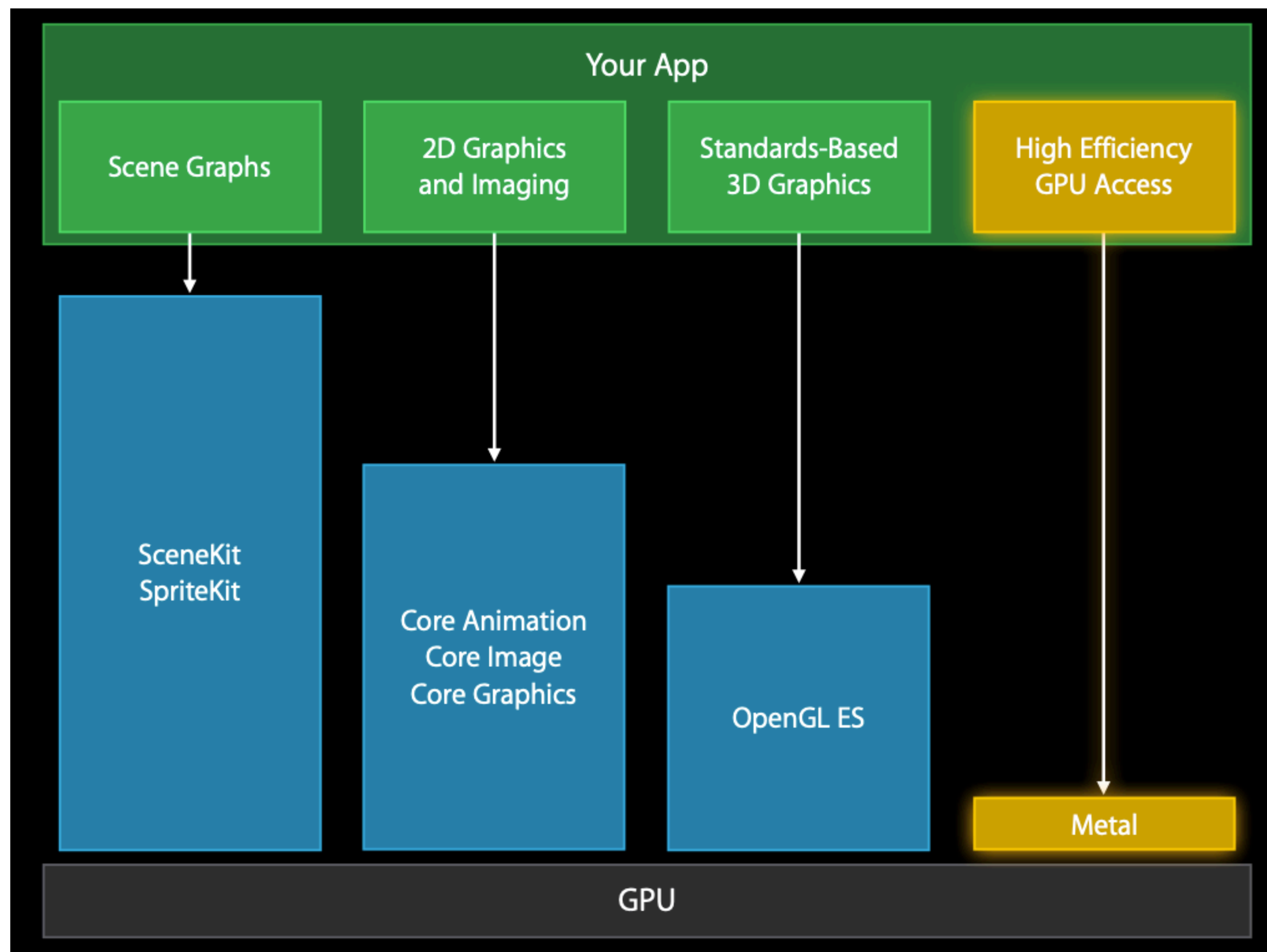
3

Metal 基础

Metal

Metal 基础概念

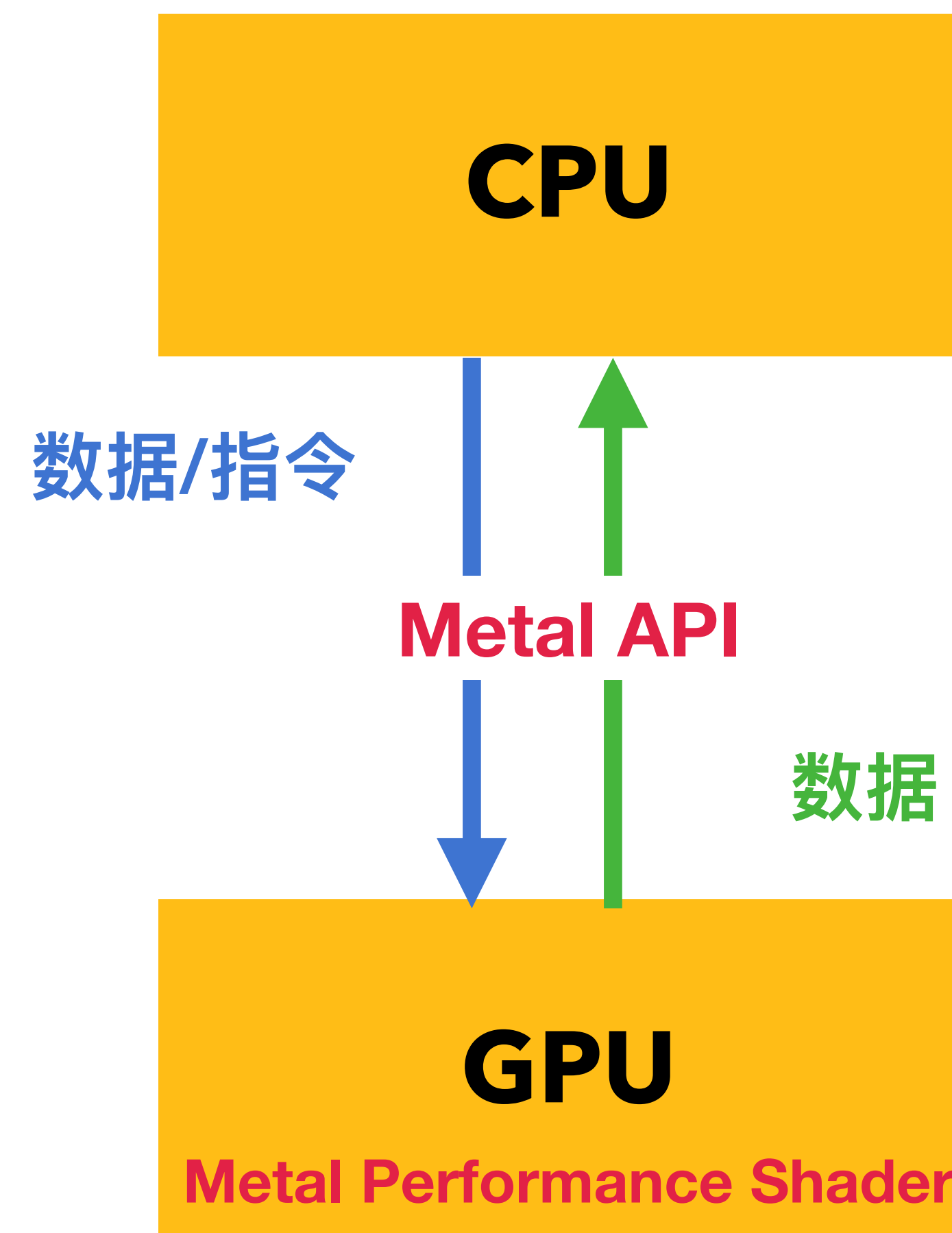
- **Metal** API使得程序员能直接操纵GPU
Primitive，控制数据进出，因此能减少很多不必要的计算



Metal

Metal Objects

- Metal API 中提供了一系列对象：
 - **Device**: 控制GPU
 - **Command Encoder**: 将API翻译到GPU硬件指令
 - **Command Buffer**: GPU硬件指令
 - **Command Queue**: GPU硬件指令序列
 - **State**: 有多种State, 表示不同的buffer状态/texture状态等
 - **Code**: Shader程序 (Kernel程序) 打包
 - **Resources**: 材质/数据/顶点信息等



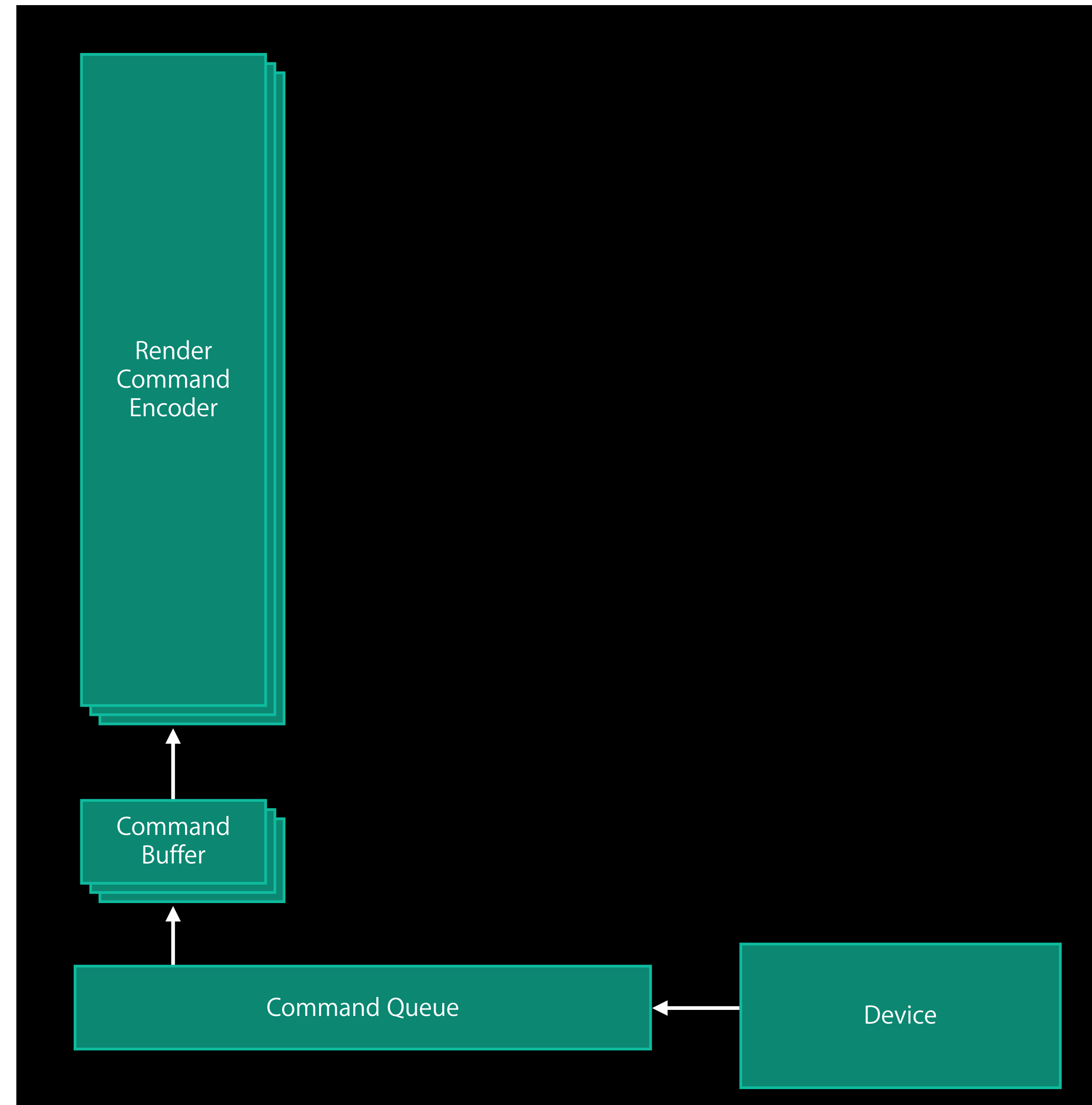
Metal



浙江大学
ZHEJIANG UNIVERSITY

Metal 调用流程

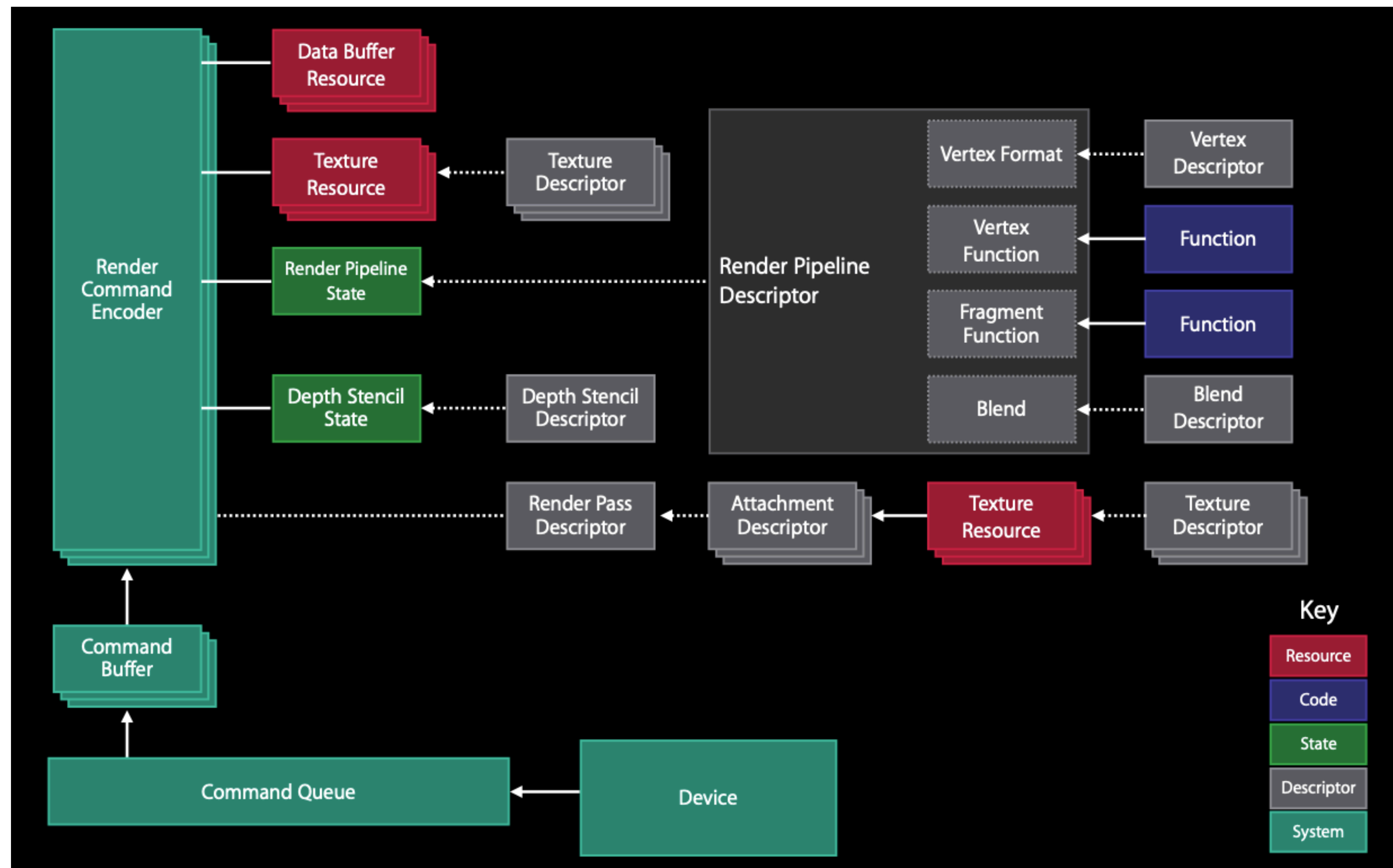
- 从 **Device** 中获取对应的 **CommandQueue**
- 创建 **CommandBuffer** 和对应的 **Encoder**



Metal

Metal 调用流程

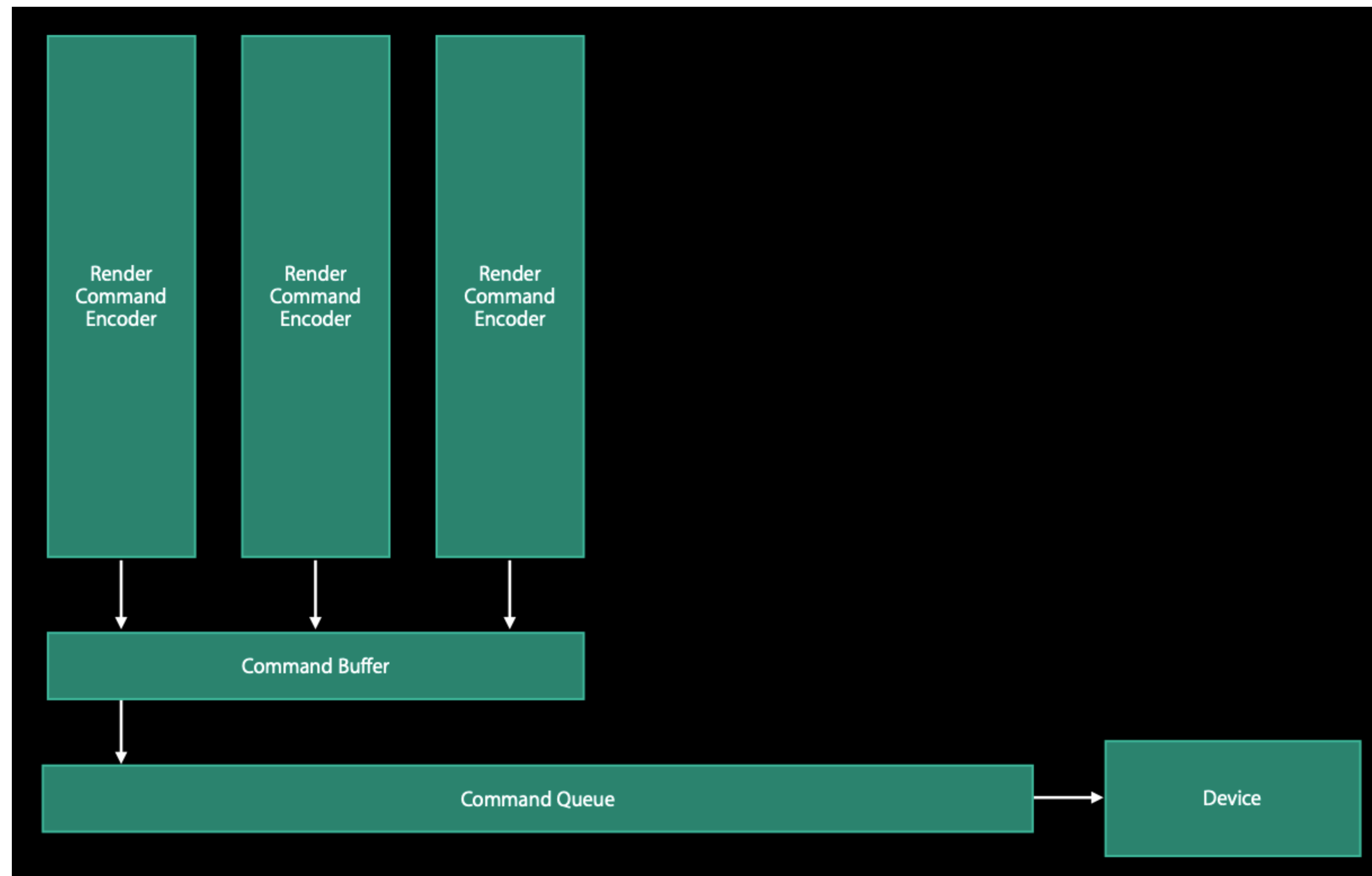
- 资源载入/渲染管线定义/其他渲染参数定义，都可以通过各类 **Descriptor** 设置到 **Encoder** 中



Metal

Metal 调用流程

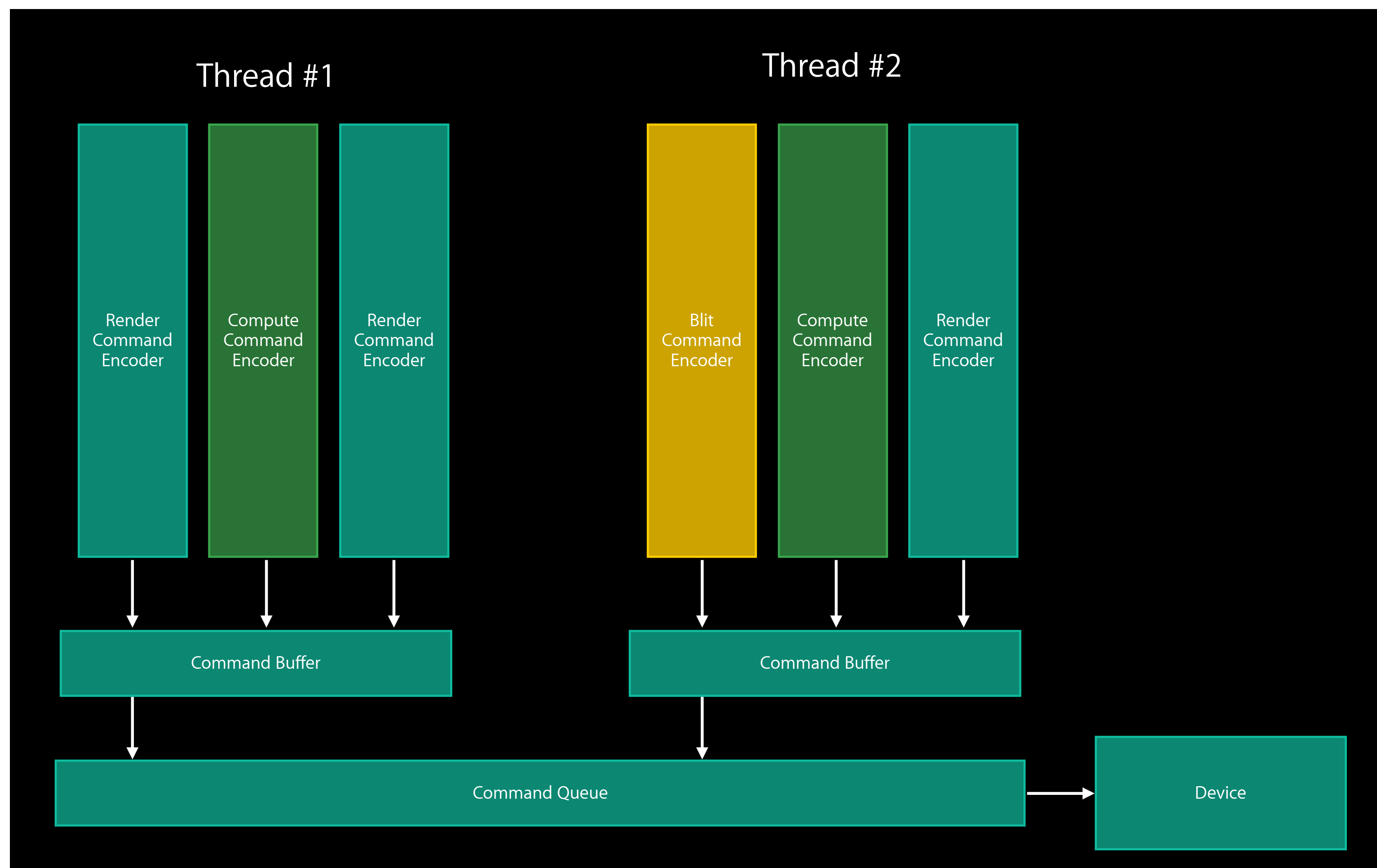
- 多个 **Encoder** 可以一起作用在一个 **CommandBuffer** 上，然后再传递给 **CommandQueue**



Metal

Metal 调用流程

- 不同的Encoder（用于计算的/用与图形渲染的/..）可以作用在同一个Buffer上
- 不同的 CPU Thread 上可以并行处理不同的 Buffer，然后依次加入 CommandQueue



Metal 调用流程

- 学习 **Metal** 的过程中，一定要区分 **Command Submission** 和 **Resource Update**
 - 指令传递和数据传递是分开控制的
- **Encoder** 将用户需要的指令（数据/渲染流程/各种设置）转换成硬件指令（**Hardware commands**）
 - 便于精细控制
 - 指令立即生成，但是还没有执行
- 然后将硬件指令存在 **Command Buffer** 中
 - 可以重复使用，提高效率

Metal 数据模型

- 有两类数据模型
 - **Texture** (formatted images)
 - **Buffer** (unformatted memory)
 - 几乎可以表示任意数据
- 数据的大小/格式在定义后不可以变化
 - 内容可以变化
 - 数据传递都是显式的，开发者可以很清晰地控制数据流
- **Buffer** 更新
 - **CPU** 可以直接访问数据，不需要lock API

Metal 渲染流程

- **Metal** 最核心的作用是替代 OpenGL ES，成为 iOS 和 Mac 上的底层图形接口
- **Metal Shading Language** 的语法是基于 C++ 14 演变的
 - 特点是各类 **Attributes** 和 **Argument Tables**
- **Metal** 渲染也符合前面描述的流程，Apple 还提供了很多优化API，有图形学基础的同学可以自主尝试

Metal Demo: GPU加法



整体效果与流程

- 我们要实现前面提到的数组加法
- 具体流程如下:
 - 获得GPU设备接口
 - 初始化各类Buffer、Queue、Encoder
 - 填充数据
 - 设置指令
 - 发送指令
 - 验证结果

```
guard let device: MTLDevice =  
    MTLCreateSystemDefaultDevice()  
    else { exit(0) }
```

```
let adder = MetalAdder(device: device)  
adder?.prepareData()  
adder?.sendComputeCommand()  
adder?.verifyResults()
```


Metal Demo: GPU加法



Kernel 函数

```
kernel void add_arrays(device const float* inA,  
                       device const float* inB,  
                       device float* result,  
                       uint index  
                       [[thread_position_in_grid]])  
{  
    result[index] = inA[index] + inB[index];  
}
```

Metal Demo: GPU加法



整体效果与流程

- 为了调用前面写的 Kernal, 我们至少需要实现右边这样的功能

```
class MetalAdder {  
    var device: MTLDevice  
    var addFuncCPS: MTLComputePipelineState  
    var commandQueue: MTLCommandQueue  
  
    var bufferA: MTLBuffer  
    var bufferB: MTLBuffer  
    var bufferResult: MTLBuffer  
  
    init?(device: MTLDevice) { }  
    func prepareData() { }  
    func sendComputeCommand() { }  
    func verifyResults() { }  
}
```

Metal Demo: GPU加法



获得设备

- **MTLDevice** 是对 GPU 的一种抽象
- 使用 **MTLCreateSystemDefaultDevice()** 方法可以获得设备的默认 GPU
- Mac 等多GPU的设备则会返回其中的一个

```
guard let device: MTLDevice = MTLCreateSystemDefaultDevice()  
    else { exit(0) }
```


Metal Demo: GPU加法



初始化 Metal 相关对象

- 初始化 **library** (存放Kernel函数)
- 初始化 **metal** 的流水线 (**ComputePipelineState**)
- 初始化 **CommandQueue** (**device**的指令序列)
- 初始化 **Buffer** (存放数据)

```
let arrayLength: Int = 1 << 24
let bufferSize = arrayLength * MemoryLayout<Float>.size

init?(device: MTLDevice) {
    self.device = device
    do {
        let defaultLib = device.makeDefaultLibrary()!
        let addFunc = defaultLib.makeFunction(name: "add_arrays")!
        try self.addFuncCPS = device.makeComputePipelineState(
            function: addFunc)
        self.commandQueue = device.makeCommandQueue()!

        self.bufferA = device.makeBuffer(length: bufferSize,
                                           options: .storageModeShared)!
        self.bufferB = device.makeBuffer(length: bufferSize,
                                           options: .storageModeShared)!
        self.bufferResult = device.makeBuffer(length: bufferSize,
                                                options: .storageModeShared)!
    } catch {
        return nil
    }
}
```

Metal Demo: GPU加法



初始化数据

- 往 Buffer 里

填充随机数

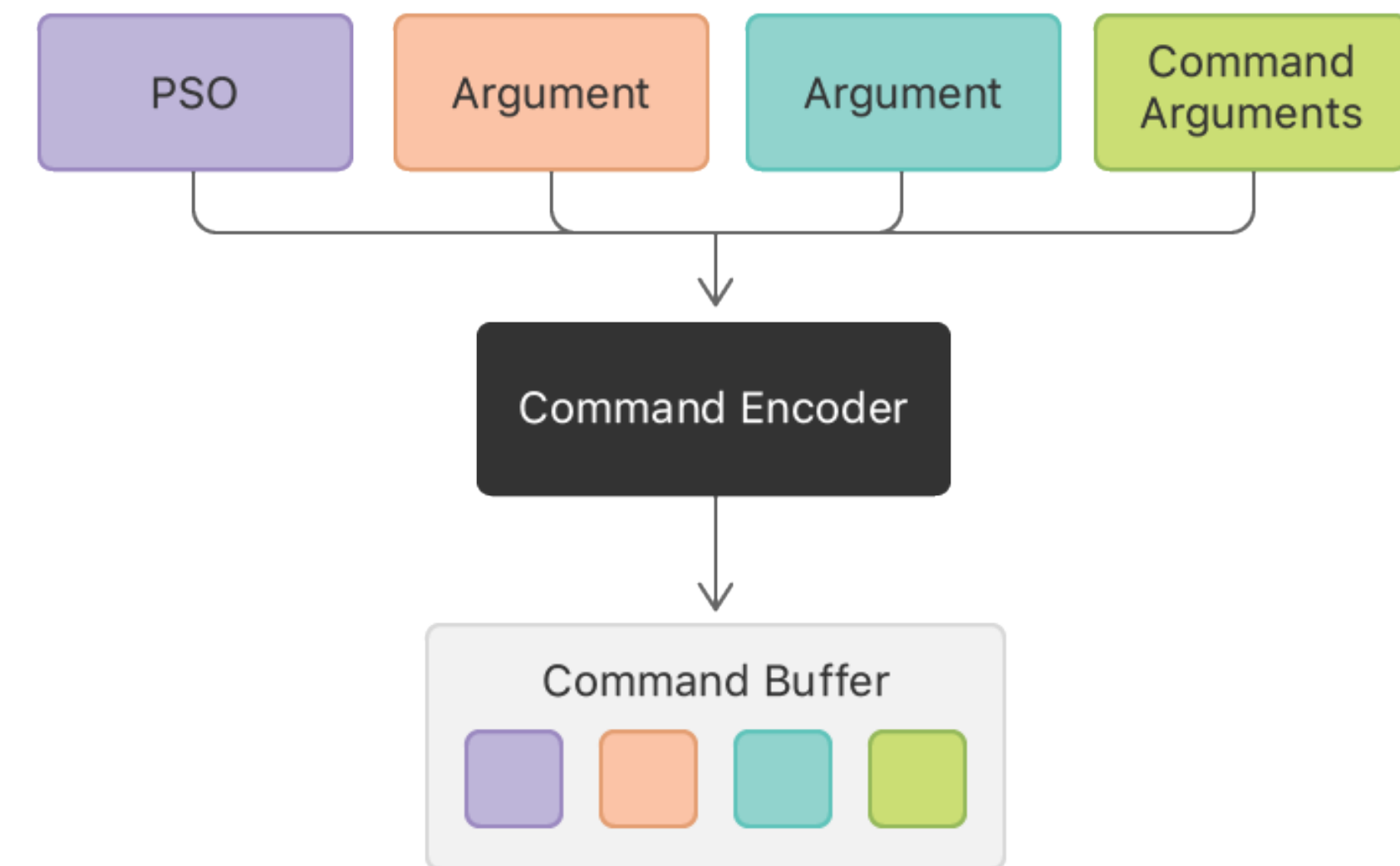
据

```
func prepareData() {  
    func generate_random_float(buffer: MTLBuffer) {  
        let dataPtr =  
            buffer.contents().assumingMemoryBound(to: Float.self)  
        for index in 0...arrayLength {  
            dataPtr[index] = Float(arc4random()) / Float(UINT32_MAX)  
        }  
    }  
    generate_random_float(buffer: self.bufferA)  
    generate_random_float(buffer: self.bufferB)  
}
```

Metal Demo: GPU加法

创建 Encoder

- 用 **CommandQueue** 创建默认的 **commandBuffer**
- 用 **CommandBuffer** 创建默认的 **Encoder**
- 把指令塞入 **Encoder**



```
let commandBuffer = self.commandQueue.makeCommandBuffer()!  
let computeEncoder = commandBuffer.makeComputeCommandEncoder()!
```

```
computeEncoder.setComputePipelineState(self.addFuncCPS)  
computeEncoder.setBuffer(bufferA, offset: 0, index: 0)  
computeEncoder.setBuffer(bufferB, offset: 0, index: 1)  
computeEncoder.setBuffer(bufferResult, offset: 0, index: 2)
```


Metal Demo: GPU加法



浙江大学
ZHEJIANG UNIVERSITY

创建 Encoder

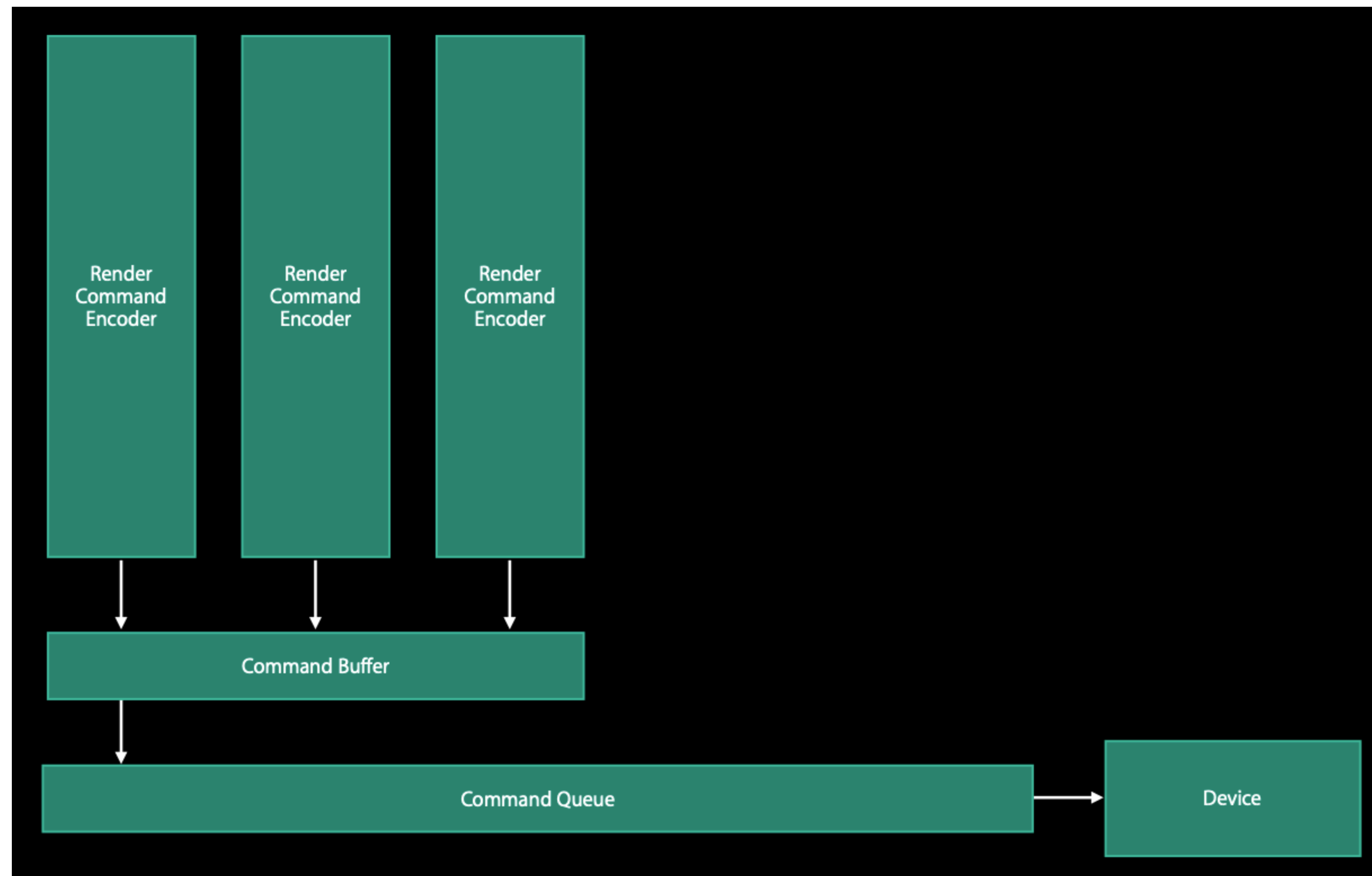
```
kernel void add_arrays(device const float* inA,  
                      device const float* inB,  
                      device float* result,  
                      uint index  
                      [[thread_position_in_grid]])  
{  
    result[index] = inA[index] + inB[index];  
}
```

```
let commandBuffer = self.commandQueue.makeCommandBuffer()!  
let computeEncoder = commandBuffer.makeComputeCommandEncoder()!  
  
computeEncoder.setComputePipelineState(self.addFuncCPS)  
computeEncoder.setBuffer(bufferA, offset: 0, index: 0)  
computeEncoder.setBuffer(bufferB, offset: 0, index: 1)  
computeEncoder.setBuffer(bufferResult, offset: 0, index: 2)
```

Metal

Metal 调用流程

- 多个 **Encoder** 可以一起作用在一个 **CommandBuffer** 上，然后再传递给 **CommandQueue**



Metal Demo: GPU加法



设置 threadgroup大小

- `arrayLength` (数据长度), 就是我们的 `Grid` 大小。
- 计算 `threadGroupSize`
- 把指令塞入 `Encoder`

```
let gridSize = MTLSizeMake(arrayLength, 1, 1)
```

```
let threadGroupSize = MTLSizeMake(  
    min(addFuncCPS.maxTotalThreadsPerThreadgroup, arrayLength), 1, 1)
```

```
computeEncoder.dispatchThreads(gridSize,  
                                threadsPerThreadgroup: threadGroupSize)
```


Metal Demo: GPU加法



发送指令，开始执行

- 结束 **Encoder** 的编码
- 发送 **commandBuffer** 中的指令
- 等待程序运行完成

```
computeEncoder.endEncoding()  
commandBuffer.commit()  
commandBuffer.waitUntilCompleted()
```

Metal Demo: GPU加法



验证数据

- 在 CPU 上验证数据正确性

```
func verifyResults() {  
    let a = bufferA.contents().assumingMemoryBound(to: Float.self)  
    let b = bufferB.contents().assumingMemoryBound(to: Float.self)  
    let result = bufferResult.contents().assumingMemoryBound(to: Float.self)  
    for index in 0..  
arrayLength {  
        if a[index] + b[index] != result[index] {  
            print("Compute ERROR")  
            return  
        }  
    }  
    print("Compute results as expected\n");  
}
```

Metal Demo: GPU加法



验证数据

- 在 CPU 上验证数据正确性

```
func verifyResults() {
    let a = bufferA.contents().assumingMemoryBound(to: Float.self)
    let b = bufferB.contents().assumingMemoryBound(to: Float.self)
    let result = bufferResult.contents().assumingMemoryBound(to: Float.self)
    for index in 0..
```


QUESTIONS?