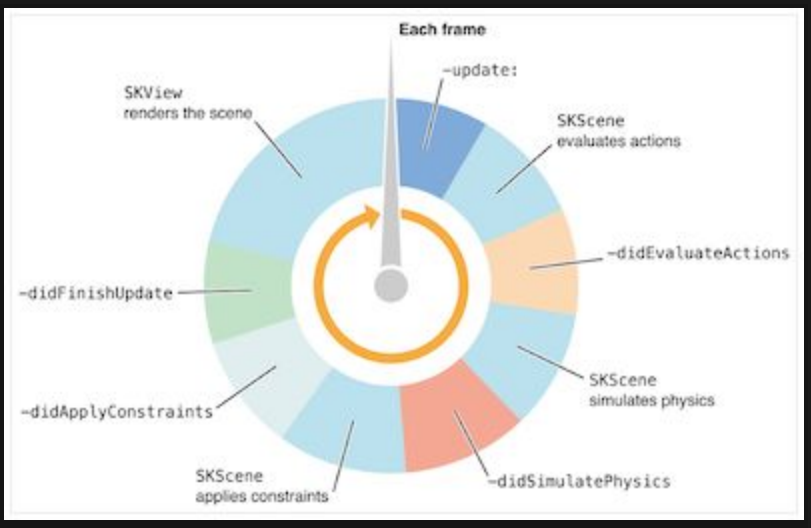
Responding to Frame-Cycle Events

Implement per-frame app logic, such as the scene's update function that's called every frame.

Overview

When a scene is presented via `presentScene(_:)`, SpriteKit calls you every frame if you implement any of the functions shown in the following image:



If you don't implement any of the frame-cycle functions, SpriteKit will only render the scene when something within it has changed, improving energy efficiency and allowing your game or app to perform other operations instead.

Decide Which Events to Implement

Each time through the rendering loop, the scene's contents are updated and then rendered. You can't override the rendering behavior; instead you update the nodes in the scene. However, the scene includes methods you can override to customize scene processing, and you can use actions and physics to alter properties of nodes in the tree. Here are the steps in the rendering loop:

1. The scene's `update(_:)` method is called with the time elapsed so far in the simulation. This is the primary place to implement your own in-game simulation, including input handling, artificial intelligence, game scripting, and other similar game logic. Often, you use this method to make changes to nodes or to run actions on nodes.
2. The scene processes actions on all the nodes in the tree. It finds any running actions and applies those changes to the tree. In practice, because of custom actions, you can also hook into the action mechanism to call your own code. You

cannot directly control the order in which actions are processed or cause the scene to skip actions on certain nodes, except by removing the actions from those nodes or removing the nodes from the tree.

3. The scene's `didEvaluateActions()` method is called after all actions for the frame have been processed.
4. The scene simulates physics on nodes in the tree that have physics bodies. Adding physics to nodes in a scene is described in `SKPhysicsBody`, but the end result of simulating physics is that the position and rotation of nodes in the tree may be adjusted by the physics simulation. Your game can also receive callbacks when physics bodies come into contact with each other.
5. The scene's `didSimulatePhysics()` method is called after all physics bodies for the frame have been simulated.
6. The scene applies any constraints associated with nodes in the scene. Constraints are used to establish relationships in the scene. For example, you can apply a constraint that makes sure a node is always pointed at another node, regardless of how it is moved. By using constraints, you avoid needing to write a lot of custom code in your scene handling.
7. The scene calls its `didApplyConstraints()` method.
8. The scene calls its `didFinishUpdate()` method. This is your last chance to make changes to the scene.
9. The scene is rendered.

Use a Scene Delegate Instead of Subclassing a Scene

Subclassing `SKScene` and overriding frame-cycle functions is a simple way to implement your app's logic, but you avoid subclassing by using a delegate. All of the frame-cycle functions defined by SKScene have an `SKSceneDelegate` protocol equivalent function. So, if you provide a scene delegate, the delegate's frame-cycle methods are called instead of those on the scene. For example, an iOS app might use a view controller as the scene delegate.

Implement Post-Processing

A scene can process the actions on the scene tree in any order. For this reason, if you have tasks that you need to run each frame and you need precise control over when they run, you should use the `didEvaluateActions()` and `didSimulatePhysics()` methods to perform those tasks. Typically, you make changes during post-processing that require the final calculated positions of certain nodes in the tree. Your post-processing can take these positions and perform other useful work on the tree.