Sample Code

Simulating dice rolls as a component for your game

Create a physically realistic dice game by adding interactive rolling and scoring.

Download

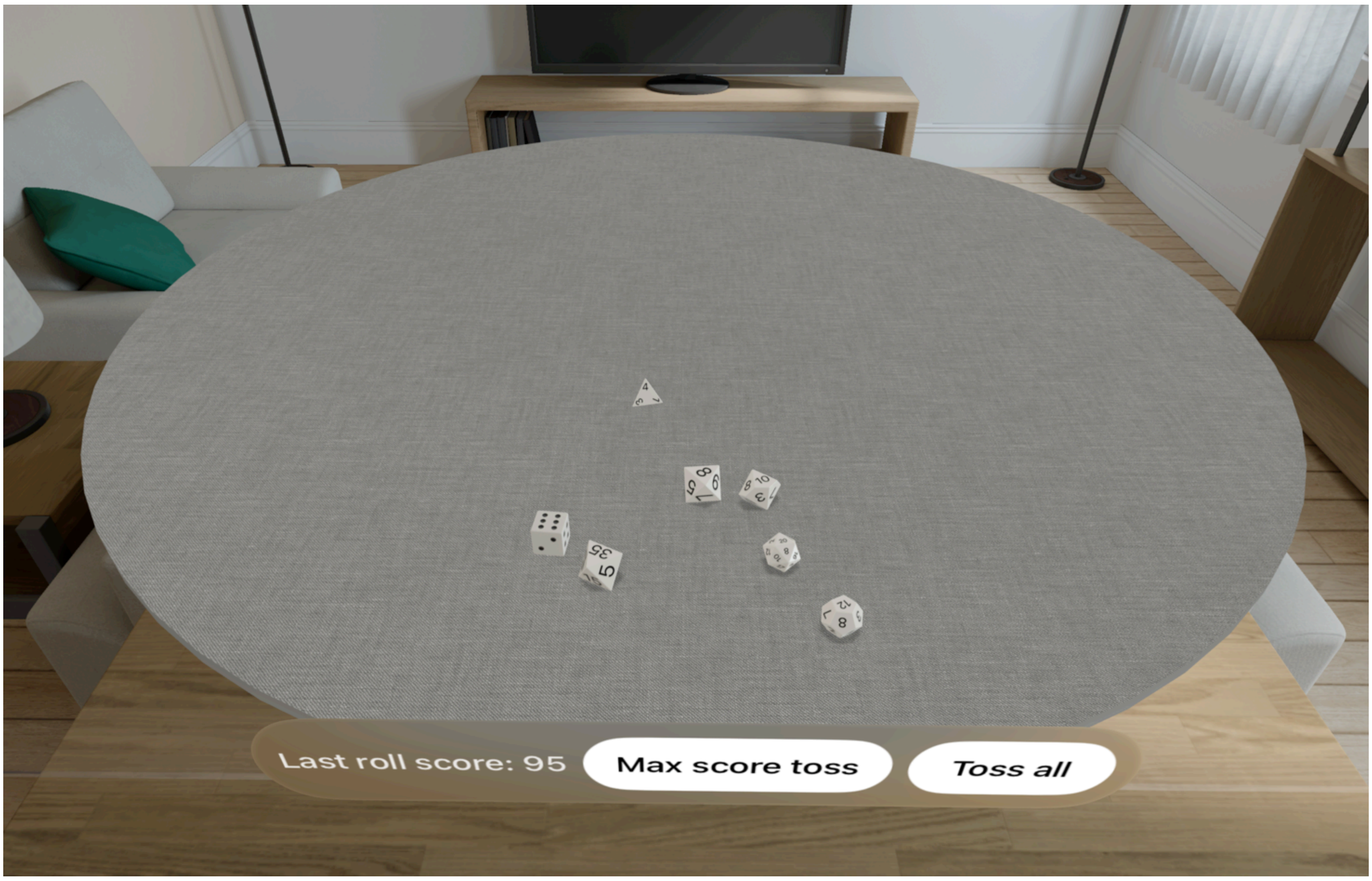
visionOS 26.0+ | Xcode 26.0+

Overview

When reproducing the physicality of a game — whether it's a role-playing game or a boardgame — one key component is how you add realistic dice rolls to your game. When a player interacts with a die, the interaction feels natural as they lift, rotate, toss, and observe each die land on the appropriate face.

This sample code app showcases physics-driven dice simulation with realistic tossing mechanics, rotation handling, and score calculation across multiple dice types with various sides. The API gives you the ability to toss the dice and let the physics simulation handle deciding the final pose and resting orientation. It helps you:

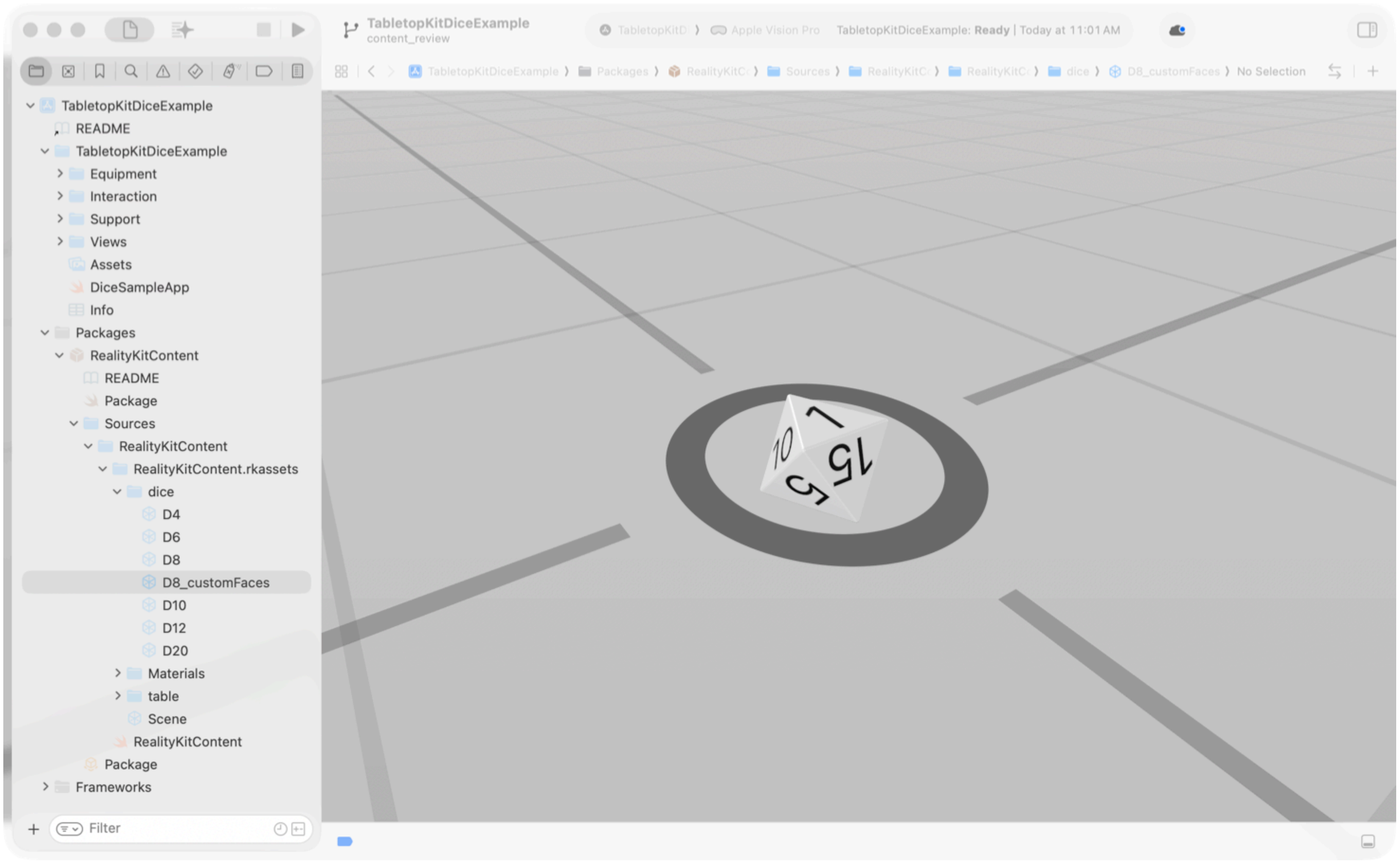
- Determine what face a die lands on naturally.
- Interact with the dice, like lifting and rotating a die.
- Produce outcomes for all of the dice thrown for a particular roll.
- Convert between the logical face of a die and the corresponding resting orientation.



The app provides a tabletop gaming experience in a volumetric window that allows a player to toss an individual die or a group of dice. After tossing an individual or group of dice, the sample shows the total score.

Create a die with face mapping

Before creating the representation of dice with `TabletopKit`, the sample app includes corresponding USDZ files for each of the dice in the `RealityKit` content bundle, including a custom 8-sided die, shown here:



The face mapping system allows you to assign custom values to each face of a die. The sample includes standard sequential mappings for most dice, but also demonstrates custom scoring with a custom 8-sided die:

```swift
let customOctahedronFaceMap: FaceMap<TossableRepresentation.OctahedronFace> = [
    .a: 1,
    .b: 5,
    .c: 10,
    .d: 15,
    .e: 20,
    .f: 25,
    .g: 30,
    .h: 35
]
```

The sample creates different types of dice by using `TossableRepresentation`. Each die type uses a specific geometric representation — like an octahedron with eight sides — and face mapping that determines the score value of each face on the die:

```swift
func customOctahedronDie(index: Int, height: Float = 0.02) -> Die {
    Die(index: index,
        entityName: "dice/D8_customFaces",
        representation: TossableRepresentation.octahedron(height: height),
        faceMap: customOctahedronFaceMap)
```

Each die implements the EntityEquipment protocol and includes methods for calculating scores, determining resting orientations, and finding the highest-scoring face. The `Die` class encapsulates the 3D model entity, tossable representation, and the face mapping for the die. After a toss ends, the sample calls `calculateScore(for:)` to get the score for the die state:

```
func calculateScore(for state: RawValueState) -> Int {
    guard let currentFace = faceType.init(rawValue: state.rawValue) else {
        fatalError("Invalid rawValue in state")
    }

    guard let score = faceMap.value(for: currentFace) else {
        fatalError("Wrong face map used for this die")
    }
    return score
}
```

Configure the game

The game configuration involves setting up the tabletop environment and adding seven dice to the table. The `Game` class manages the overall game state and coordinates between TabletopKit and the UI:

```
var setup = TableSetup(tabletop: tabletop)
setup.add(seat: PlayerSeat(index: 0, position: .init(x: 0, z: +0.5), rotation: .init
setup.add(equipment: tetrahedronDie(index: 1))
setup.add(equipment: cubeDie(index: 2))
setup.add(equipment: octahedronDie(index: 3))
setup.add(equipment: customOctahedronDie(index: 4))
setup.add(equipment: decahedronDie(index: 5))
setup.add(equipment: dodecahedronDie(index: 6))
setup.add(equipment: icosahedronDie(index: 7))

tabletopGame = TabletopGame(tableSetup: setup)
tabletopGame.claimAnySeat()
```

The app uses a volumetric window style with a 2 x 2 x 2 meter default size to provide adequate space for dice interaction. The `RoundTabletop` provides the surface for dice physics simulation.

The game automatically positions the table within the volumetric space, ensuring that the table surface aligns with the bottom of the volume and the closest edge stays against the front boundary so it remains close to the player interacting with the dice. When the player tosses the dice at the boundary, they bounce off of the volume bounds and back onto the table surface.

Handle the dice interaction

The sample app manages dice interaction through the `DiceInteraction` class, which implements `TabletopInteraction.Delegate`. The interaction system supports both single die tossing and group tossing, where multiple dice move together and get tossed simultaneously. The sample uses a hexagonal positioning pattern to ensure dice don't overlap during group interactions.

The `controlledDie` is a single die that the player interacts with in the sample app. When the interaction begins with a die, and the "Toss all" option is in a selected state, the system sets up additional dice for tossing them as a group with the controlled die:

```
case .started:
    /// Group the dice together when the person selects "Toss All".
    for (index, die) in extraDiceToToss.enumerated() {
        interaction.addAction(.moveEquipment(die,
                                             childOf: controlledDie,
                                             pose: hexagonPoses[index]))
    }
```

The sample app monitors gesture completion and initiates the toss when the player releases the dice:

```
case .update:
    /// Run the tossing simulation when the player releases the dice.
    if interaction.value.gesture?.phase == .ended {
        interaction.toss(equipmentID: controlledDie.id,
                         as: controlledDie.tossableRepresentation)

        for die in extraDiceToToss {
            interaction.toss(equipmentID: die.id,
                             as: die.tossableRepresentation)
        }
    }
```

After the player completes their toss, the system calculates and updates the final score:

```
case .ended:
    /// Calculate and store the result.
    if interaction.value.phase == .ended {
        game.updateLastRollScore(for: [controlledDie] + extraDiceToToss)
    }
```

Process the dice toss and score

The sample includes toggle controls for testing different scenarios:

Max score toss
: Forces all dice to land on their highest-scoring faces.

Toss all
: Enables group tossing of all dice simultaneously.

The physics simulation provides realistic dice behavior including rotation, bouncing, and the dice settling into their final positions. The toss outcome processing occurs in the `onTossStart(interaction:outcomes:)` method, where the sample processes the physics simulation and final die states. When determining the outcome, the sample uses `predeterminedOutcome` to get the highest score for the toss. If the player didn't select the option for the highest score, the sample calls the function `face(for:)` to get the die face for the `TabletopInteraction.TossOutcome`:

```
func onTossStart(interaction: TabletopInteraction,
                 outcomes: [TabletopInteraction.TossOutcome]) {

    let allTossedDice = [controlledDie] + extraDiceToToss

    for outcome in outcomes {
        guard let die = allTossedDice.first(where: { $0.id == outcome.id }) else {
            fatalError("Outcome ID does not match any tossed dice")
        }

        let face = if predeterminedOutcome {
            die.faceWithHighestScore()
        } else {
            // Roll the score that the physics simulation determines.
            outcome.tossableRepresentation.face(for: outcome.restingOrientation)
        }

        // Set the new final pose and score on the die.
        interaction.addAction(.updateEquipment(die,
                                               rawValue: face.rawValue,
                                               pose: outcome.pose))

        if die.id != controlledDie.id {
            // If this was one of the extra dice, ensure that its pose is
            // back in table space.
            interaction.addAction(.moveEquipment(matching: die.id,
                                                 childOf: .tableID))
        }
    }
}
```

Score calculation uses the face mapping system to determine point values for each die. The game tracks the total score by summing individual die scores and displays the result in the interface:

```
func updateLastRollScore(for tossedDice: [Die]) {
    tabletopGame.withCurrentSnapshot { snapshot in
        var score = 0
        for die in tossedDice {
            score += die.calculateScore(for: snapshot.state(for: die))
        }
        lastRollScore = score
    }
}
```

See Also

Interactions

`class TabletopInteraction`

A protocol for objects that manage the entire flow of players interacting with equipment.

`struct TossableRepresentation`

An object that represents geometric shapes that the player can throw during gameplay, such as dice.

`struct TableSnapshot`

A snapshot of the current state of the table.

`struct TableVisualState`

A structure that represents the appearance of an object on the table.

`struct TableCursor`

A cursor conveys information about one equipment that is currently being controlled by an interaction.

`struct TableCursorIdentifier`

A unique identifier for cursors.