Sample Code

Implementing playing card overlap and physical characteristics

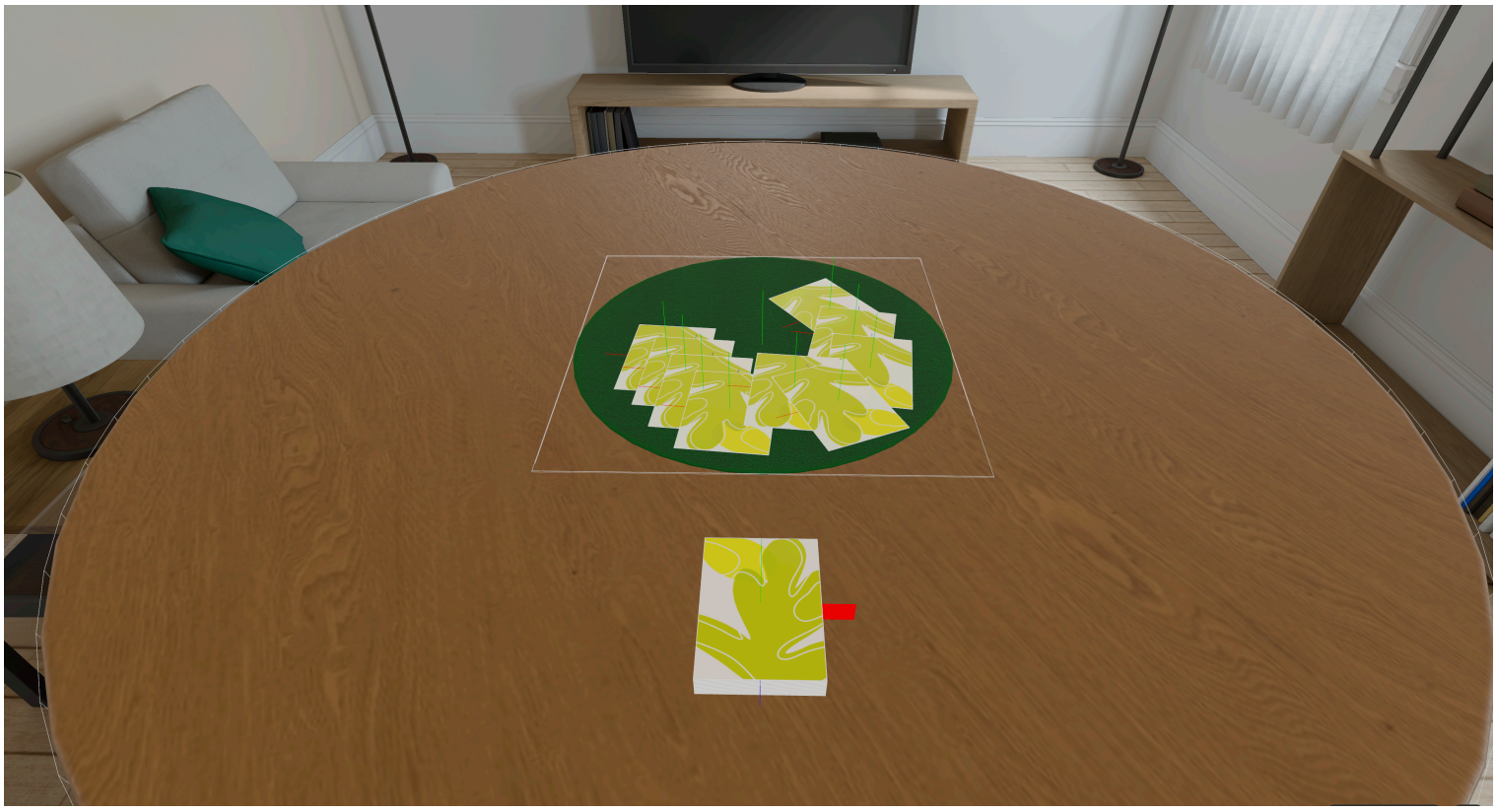
Add interactive card game behavior for a pile of playing cards with physically realistic stacking and overlapping.

Download

visionOS 26.0+ | Xcode 26.0+

Overview

This sample code project demonstrates an effective way an app can implement a 3D volumetric layout to allow for thin equipment, such as playing cards, to pile up in a realistic manner if placed arbitrarily by the player. As the player adds cards into the pile, the app deliberately places them at a nonintersecting, and physically believable height and tilt atop all previous equipment.



The app contains a center pile that cards the player adds become descendants of. Each card has its pose calculated by an algorithm comprised of two physically based solvers:

- Center of mass solver
- Bounded convex hull solver

The center of mass solver takes the input of the existing cards' bounding boxes and estimates the effective center of mass of the current card. It adjusts to account for weight applied from above. The bounded convex hull solver takes the bounding boxes of all of the previously placed cards as input and calculates a support plane for the calculated center of mass of the current card.

Consider the limitations

The sample code project uses two simplifying assumptions to produce a performant algorithm:

- It assumes the layout tilting from a horizontal angle remains small. The sample also assumes the 2D projection of the 3D tilted bounding box remains similar to the flat 2D projection.
- It assumes the cards are thin in the Y dimension — the height — as this allows the first assumption to remain true in a stack of multiple objects. It also allows the sample to ignore side faces of bounding boxes.

The first assumption is sufficient by itself to produce nonintersecting 3D poses, but produces nonphysical and more extremely tilted layouts when players repeatedly stacks cards atop a previously tilted card. The second assumption both improves the physical realism by accounting

for weight atop earlier cards and also tends to reduce overall tilting by leveling cards down when later cards lean against the high end of a previously tilted card.

The limitations mean that this algorithm is fairly ideal for thin playing cards. If given thicker bounding boxes, the algorithm still functions, but eventually shows nonphysical support by points outside the area that faces down (floating) and nonphysical intersections with side faces of higher-tilt thick boxes.

Implement a convex hull support plane solver with clipping

A bounded convex hull solver consists of two steps:

1. Collect all of the possible support points by clipping each previously placed equipment bounding box top plane to the current equipment's flat 2D rectangular footprint.

2. Use a QuickHull-like algorithm search to find the convex hull around all of these points, but limited to only find the single hull face that intersects a specified vertical line.

In `ConvexBoundary2D`, the sample implements general 2D convex planar polygon clipping by iteratively clipping to each edge of the other polygon, which is adequate for the quadrilateral inputs.

```
func clip(corners: inout [Point3DFloat],
                  edges: inout [Plane3DFloat],
                  to edge: Plane3DFloat) {
    let distanceToCorners = corners.map { edge.distance(to: $0) }

    // Clipping by the previous edges may leave no corners outside.
    if distanceToCorners.allSatisfy({ $0 <= 0 }) {
        return
    }

    // The inside corners and outside corners must always form
    // contiguous (but possibly wrapped) sequences.
    let cornerIndexFirstOutside = distanceToCorners[0] >= 0 ?
        ConvexBoundary2D.nextIndex(distanceToCorners.lastIndex(where: { $0 < 0 })!,
        distanceToCorners.firstIndex(where: { $0 >= 0 })!
    let cornerIndexLastOutside = distanceToCorners[0] >= 0 ?
        ConvexBoundary2D.prevIndex(distanceToCorners.firstIndex(where: { $0 < 0 })!,
        distanceToCorners.lastIndex(where: { $0 >= 0 })!
    let cornerIndexLastInside = ConvexBoundary2D.prevIndex(cornerIndexFirstOutside,
    let cornerIndexFirstInside = ConvexBoundary2D.nextIndex(cornerIndexLastOutside,
```

```
    let intersectionFirst = intersection(between: corners[cornerIndexLastInside],
                                         and: corners[cornerIndexFirstOutside],
                                         distance1: distanceToCorners[cornerIndexLas
                                         distance2: distanceToCorners[cornerIndexFir
    let intersectionLast = intersection(between: corners[cornerIndexLastOutside],
                                        and: corners[cornerIndexFirstInside],
                                        distance1: distanceToCorners[cornerIndexLast
                                        distance2: distanceToCorners[cornerIndexFirs
    if cornerIndexFirstOutside <= cornerIndexLastOutside {
        corners.replaceSubrange(cornerIndexFirstOutside...cornerIndexLastOutside, wi
        edges.replaceSubrange(cornerIndexFirstOutside..<cornerIndexLastOutside, with
        return
    }
    if cornerIndexLastInside < corners.count - 1 {
        corners.removeSubrange(cornerIndexFirstOutside..<corners.count)
        edges.removeSubrange(cornerIndexFirstOutside..<corners.count)
    }
    corners.replaceSubrange(0...cornerIndexLastOutside, with: [intersectionFirst, in
    edges.replaceSubrange(0..<cornerIndexLastOutside, with: [edge])
}
```

Find the supporting plane

A QuickHull algorithm consists of starting with a plane defined by any three points known to be on the convex hull — such as those at a maximum or minimum value in any one coordinate axis. Iterate over the points to:

1. Find the point with the highest distance outside the current plane, which must also be on the convex hull.
2. Discard all points inside of all planes defined by the new point and the edges made by pairs of the previous three points.
3. Descend into each of these new planes until you find all planes with no points outside and discard all internal points.

The `BoundedConvexHull` class implements `findSupportPlane(at:)` to find the single convex hull plane beneath a specified center point.

Because this use case only requires finding a single plane of the convex hull, it's unnecessary to find the full convex hull. Instead, the algorithm focuses on locating the single face that contains, in 2D, the supplied center point by iterating the following steps:

1. Find the point with the highest distance outside the current plane, which must also be on the convex hull, and discard all points below or on the plane.

2. Find the pair of previous points that, together with the new point, surround the center point of interest.

3. Descend into this new plane until no nondiscarded vertices are outside, and then until no original vertices are also outside.

Rechecking the original points is necessary because concentrating on the one sector that contains the center in a single descent can produce a final plane that doesn't account for possible support points in other discarded sectors. Each outer iteration, starting from all points, finds at least one — but often two or three — of the three actual support points, so three outer iterations may be required.

The support planes produced by this method don't account for the fact that tilting a box somewhat distorts its 2D projected boundary, potentially leaving one or more support points outside or, at some tilt directions, failing to consider support points in a sliver of space outside of the flat 2D boundary. The first issue can result in highly tilted boxes floating without realistic support. The second can result in boxes that are highly tilted at a nonaxis-aligned direction slightly intersecting adjacent boxes.

Because this method doesn't add the side planes of boxes to the convex hull vertices, it also doesn't consider some valid support points on thick tilted boxes, with possible intersections resulting.

Implement a center of mass solver

The `CenterOfMassSolver` structure implements `calculateCentersOfMass(boxes:)` to calculate the effective center of mass. It first finds all added boxes that stack atop each original box, based on flat 2D rectangle overlapping. Then, it calculates a contact point and weight for each added box, which the solver assumes to either:

- Apply its full weight at its center of mass if that lies within the base rectangle because the added box expects to lie flat atop it.
- Apply a fraction of its weight at its center of mass clamped to the nearest edge of the base rectangle because the added box expects to split its weight between multiple support points

Finally, the method calculates the weighted average of the base box's center and all weighted contact points atop it.

```
func calculateCentersOfMass(boxes: [OrientedBox]) -> [Point3DFloat] {
    if boxes.isEmpty {
        return []
    }
    if boxes.count == 1 {
        return [CenterOfMassSolver.calculateCenterOfMass(boxes[0])]
```

```
    }

    var centers: [Point3DFloat] = []
    centers.reserveCapacity(boxes.count)
    for (index, box) in boxes.enumerated() {

        // Calculate the average center of mass of this box, and all boxes
        // that are stacked atop it, recursively.
        //
        // The sample assumes that a box that is stacked atop is to rest
        // all or part of its weight on the base depending on whether its
        // center lies inside the base box's area.
        let orientedBox = OrientedRect2DFloat(box)
        var center = CenterOfMassSolver.calculateCenterOfMass(box)
        var weightedOffset = Vector3DFloat.zero
        var totalWeight: Float = 0
        for indexNext in (index + 1)..<boxes.count {

            let boxNext = boxes[indexNext]
            let orientedBoxNext = OrientedRect2DFloat(boxNext)
            if !orientedBox.overlaps(orientedBoxNext) {
                // Other boxes that don't directly overlap, but sit atop a
                // box that overlaps, or a chain of such indirect overlaps,
                // might also apply a fraction of their weight at some
                // contact point.
                // However, indirect overlaps contribute exponentially less
                // and are increasingly likely to be spurious as the count
                // of links in the chain increases, so it's simpler and
                // reasonably accurate to just ignore them.
                continue
            }

            // Each box applies its weight at its own center and clamps to the
            // base box rectangle. This becomes inaccurate if the actual
            // average point of contact isn't near the center of this box,
            // and can produce nonphysical forces when it lies on the
            // opposite side of the base box's center.
            //
            // In theory, a more sophisticated analysis of the stacking
            // heights, and overlap regions, of the 2D rectangles can
            // produce better guesses at the likely contact points, and
            // fewer cases with nonphysical forces applied.
            let contact = calculateCenterOfContactAndWeight(of: boxNext, on: box)
```

```
            weightedOffset += Vector3DFloat(contact.center) * contact.weight
            totalWeight += contact.weight
        }

        if totalWeight > 0 {
            center = (center + weightedOffset) / (1.0 + totalWeight)
            // Clamp to a fractional inset area of the base box's area.
            center = CenterOfMassSolver.clampCenterOfMass(center, to: box, inset: ir
        }
        centers.append(center)
    }
    return centers
}
```

The method becomes physically inaccurate when the model of contact points doesn't match the 3D reality. Notably, it produces nonphysical forces in the wrong direction when the underlying card is tilted and the card atop has a real contact point on the opposite side of the card — typically the side tilted up — from its center.

A more sophisticated analysis of the 2D stacking height and action overlap areas of bounding boxes can produce better guesses at the true contact points and weights — and a correspondingly more physically accurate 3D layout for complex piles.

Produce a card layout

By splitting the problem into two independent parts — calculating centers of mass and then finding support planes at each center — the calculation of a 3D layout becomes straightforward. In `Messy Pile`, the `layoutChildren(snapshot:visualState:)` method first calculates the centers of mass of all descendants equipment based on 2D poses only. For each descendants card, it then:

1. Adds the table plane and all previously 3D layed out descendants to the convex hull.
2. Calculates the support plane at its center of mass.
3. Sets the 3D pose of this descendants to sit flat on the support plane.

```
func layoutChildren(for snapshot: TableSnapshot, visualState: TableVisualState) -> E
    let sortedChildrenIDs = snapshot.equipmentIDs(childrenOf: id)
    let boxes: [OrientedBox] = sortedChildrenIDs.map {
        let state = snapshot.state(matching: $0)!
        return .init(pose: state.pose, boundingBox: Rect3DFloat(state.boundingBox))
    }
```

```
    // Calculate an effective center of mass for each box,
    // accounting for the weight of supported boxes on top.
    let centersOfMass = CenterOfMassSolver().calculateCentersOfMass(boxes: boxes)

    var poses: [EquipmentPose3D] = []
    poses.reserveCapacity(sortedChildrenIDs.count)
    for (index, childID) in sortedChildrenIDs.enumerated() {
        // For each descendants, collect all possible supporting points of
        // previously placed descendants, and then find the support plane
        // underneath this box's center of mass.
        // The support plane is the upward-facing face of the convex
        // hull around the point cloud that intersects the vertical
        // line at the 2D center of mass.
        let box = boxes[index]
        if index == 0 {
            poses.append(EquipmentPose3D(id: childID, pose: makePose(pose2d: box.pos
            continue
        }
        let boundary = makeConvexBoundary2D(boundingBox: box.boundingBox, pose2d: bo
        let convexHull = BoundedConvexHull(boundary: boundary)
        var countAdded = 0
        for indexPrev in 0..<index {
            let boxPrev = boxes[indexPrev]
            let posePrev = Pose3DFloat(poses[indexPrev].pose)
            let boundaryPrev = makeConvexBoundary2DOfTopPlane(boundingBox: boxPrev.b
            let topPlane = makeTopPlane(boundingBox: boxPrev.boundingBox, pose: pose
            if convexHull.addPlanarPolygon(polygon: boundaryPrev, plane: topPlane) {
                countAdded += 1
            }
        }
        if countAdded == 0 {
            poses.append(EquipmentPose3D(id: childID, pose: makePose(pose2d: box.pos
            continue
        }
        let supportPlane = convexHull.findSupportPlane(at: centersOfMass[index])
        let pose = makePoseOnPlane(pose2d: box.pose, boundingBox: box.boundingBox, p
        poses.append(EquipmentPose3D(id: childID, pose: pose))
    }
    return .volumetric(layout: poses)
}
```

See Also

Equipment

`protocol` `Equipment`

A protocol for equipment that players directly interact with in a game.

`struct` `EquipmentCollection`

A collection of equipment whose state can be inspected and modified.

`protocol` `EntityEquipment`

A protocol for equipment in a game that you render using RealityKit.

`struct` `EquipmentIdentifier`

A unique identifier for equipment.

`protocol` `EquipmentState`

A protocol for the equipment data that TabletopKit syncs between players.

`struct` `EquipmentStateCollection`

A collection of equipment states that can be inspected and modified.

`struct` `BaseEquipmentState`

A state for equipment that contains no equipment-specific data.

`protocol` `CustomEquipmentState`

A specialized protocol for the equipment state that allows to accommodate custom data that TabletopKit syncs between players.

`protocol` `MutableEquipmentState`

A protocol for equipment data that TabletopKit syncs between players, and that can be mutated.

`struct` `CardState`

A state for cards that contains face up and down information.

`struct` `DieState`

A state for dice that contains the current value.

`struct` `RawValueState`

A state for equipment that contains a game-specific value.

`enum ControllingSeats`

The seats that can manipulate or interact with the equipment.