Adding Physics Fields to a Scene

Create effects that interact with your scene's physics bodies, such as magnetism, repulsion, friction, or a vortex.

Overview

It is possible to implement a game using only physics bodies, collisions, and applied forces, but sometimes this can be a lot of work. For example, if you wanted two objects to be attracted to each other, you would have to perform gravity calculations and apply those forces every time a new frame is calculated. When you want to apply generalized effects to a scene, you can do this more easily by using physics fields.

A physics field is a node that is placed inside your scene's node tree. When the scene simulates physics, the physics field affects physics bodies that are configured to interact with it.

The following code shows one of these fields, a linear gravity field. In this example, a field node replaces the default gravity provided by the scene's physics world. First, the scene's default gravity is disabled. Then, the scene creates a linear gravity field with a vector pointing toward the bottom of the screen and gives the node a strength equal to Earth's gravity. Finally, it adds the node to itself.

```
physicsWorld.gravity = CGVector(dx:0, dy: 0);

let gravityVector = vector_float3(0,-1,0);

let gravityNode = SKFieldNode.linearGravityField(withVector: gravityVector)

gravityNode.strength = 9.8

addChild(gravityNode)
```

Here are some reasons you might use a field node rather than the default gravity:

>> To vary the strength and direction of gravity independently of each other.

>> To vary the strength of the field using actions.

>> To change the direction of gravity by rotating the field node.

```
gravityNode.zRotation = CGFloat.pi // Flip gravity.
```

>> To enable and disable gravity using the field node's `isEnabled` property, without changing the field node's `strength` or direction.

Limit the Effect of Field Nodes Through Categories and Regions

By default, a field node affects all physics bodies in the scene. However, it doesn't have to be this way. By carefully controlling which physics bodies are affected by a field node, you can create some interesting effects.

Field nodes are categorized just like physics bodies. Similarly, physics bodies declare which field categories affect them. Combining these two properties, you can decide which kinds of fields are implemented by your game and which physics bodies are affected by each of those fields. Typically, you do this by using the field categories to define general effects that you can drop into a game. For example, assume for the moment that you are implementing a game where you want to add gravity to a planet. Gravity should pull objects such as ships and asteroids toward the planet.

The following code creates a category for gravity effects. When a planet is created, a separate radial gravity field node is added to it as a child, and is configured to use the gravity category. Whenever a new physics body is created, the body's mask is configured to determine which fields should affect it. Ships are affected by gravity, and missiles are affected by force fields.

```
let gravityCategory: UInt32 = 0x1 << 0
let shieldCategory: UInt32 = 0x1 << 1
...
let gravity = SKFieldNode.radialGravityField()
gravity.strength = 3
gravity.categoryBitMask = gravityCategory
planet.addChild(gravity)
...
ship.physicsBody?.fieldBitMask = gravityCategory
...
missile.physicsBody?.fieldBitMask = shieldCategory
```

In this example, the force field is a shield object that deflects incoming missiles. You could implement the shield by creating another radial gravity field, but physics bodies are repelled by negative field strength. You probably wouldn't want the field to affect the whole scene, either—only nearby missiles. In this case, you would define the area that a field affects by using these properties:

>> A field node's region property determines the area where the field can affect things. By default, this region covers the entire scene. However, you can choose to give the region a finite shape instead. For the forcefield, a circular area is fine, but other shapes are possible. You can even construct them using constructive set geometry (CSG).

>> A field node's falloff property determines how fast the field loses its strength. You can choose to have the field not lose strength at all or you can choose to have it fall off more quickly than the default. For a radial gravity field, the default is to have it drop off based on the square of the distance to the other body's center. Not all field types are affected by the `falloff` property.

The following code adds a temporary force field to the rocket ship. The field node is created and configured to repel missiles. Then a circular region is added to limit the area affected by the node, and the field is configured so that its strength drops off more quickly as a function of distance. Finally, the field should weaken and disappear so that the ship is not invulnerable. The field is weakened using an action that first animates a drop in strength to nothing, and then removes the shield node. In an actual game, the shield effect might be embellished with a graphical effect rendered on top of the ship.

```
let shield = SKFieldNode.radialGravityField()
shield.strength = -5
shield.categoryBitMask = shieldCategory
shield.region = SKRegion(radius: 100)
shield.falloff = 4
shield.run(SKAction.sequence([
    SKAction.strength(to: 0, duration: 2.0),
    SKAction.removeFromParent()
    ]))
ship.addChild(shield)
```