**DimensionDriveDownside  
물리 엔진 구현 세부 사항**

1. 게임에서의 동역학

이 게임에서는 현실과 같은 물체의 운동을 표현하기 위해 동역학의 이론을 적용합니다. 그러나 처리를 단순하게 하도록 다음과 같은 가정을 깔고 있습니다.

1. 스테이지 안에 존재하는 모든 물체는 강체(rigid body)입니다.
2. 움직이는 물체는 평행 이동만 고려합니다. 즉, 토크(torque)를 고려하지 않습니다.
3. 충돌 처리에 관여하는 물체들은 모두 균일한 이상적인 상태라고 가정하며, 질점(particle) 취급합니다.

2. 운동의 표현 방법

스테이지 안에서 움직이는 의미 있는 물체는 오로지 **공(ball)**이며, 단순히 크기를 질점(particle) 취급하여 움직임을 표현합니다. **공**을 표현하기 위해 다음과 같은 값을 사용합니다.

* 위치(position) [벡터]
* 속도(velocity) [벡터]
* 가속도(acceleration) [벡터]
* 반지름(radius) [스칼라]

게임 안에서는 반복문을 사용하여 화면을 지속적으로 업데이트하며, 화면을 한 번 그려주는 것을 한 프레임(frame)이라 합니다. 움직임 처리는 프레임마다 이루어지며, 매 프레임이 돌아오는 주기를 단위 시간에 맞춰서 시간 간격(delta time; 이하 dt)으로 이용합니다.

따라서 Handset에 따라 처리 능력이 다르더라도 모든 운동 처리는 단위 시간에 비례하여 이루어지기 때문에 움직임은 같게 됩니다. (물론 처리 능력이 떨어지는 Handset에서는 그만큼 프레임 간격이 벌어지기 때문에 중간 움직임이 생략되어 끊기는 듯 보이는 문제가 있을 수 있습니다.)

참고로 게임 안에서 단위 시간은 1/30초로 설정되어 있습니다.

매 프레임마다 다음과 같은 방법으로 위치와 속도를 반영합니다.

velocity = velocity + acceleration \* dt

position = position + velocity \* dt

수치 적분과 같은 방법이지만 매 프레임마다 값을 누적시킵니다. 시간이 지날수록 실수 연산의 오차도 누적되지만 게임 안의 상황이 시시각각 변하며 공의 움직임이 결정적일 필요가 없으므로 이것으로 충분합니다.

공의 가속도는 뉴턴의 법칙을 이용해 크게 두 가지로 결정합니다.

* 중력
* 항력

공에 작용하는 모든 힘을 더해 알짜힘을 구하고,

F=ma

a=F/m

을 통해 가속도를 구합니다.

받는 힘에 비해 가속도는 질량에 반비례하여 받지만, 중력은 질량에 비례하므로 질량의 의한 효과가 상쇄되어 단순히 중력 가속도만을 고려하여 가속도에 바로 반영합니다. 단, 중력 가속도의 방향은 센서를 이용해 구하며, 그 양은 실제 물리적 수치가 아닌 게임 안에 맞도록 적절히 조절된 값을 사용합니다.

acceleration = gravitational\_acceleration

항력은 공이 계속 가속 받아 무한정 빨라지는 것을 방지하는 장치로 도입하였습니다. 속도에 비례하는 간단한 근사식을 사용하여 구합니다.

acceleration = acceleration + (-velocity \* friction\_coefficient)

항력 계수는 게임 상 공의 재질에 따라 적절히 조정됩니다. 따라서 공이 바뀌면 다른 항력을 받는 느낌을 줄 수 있습니다.

3. 충돌 처리

이 게임에서는 물체끼리의 상호 작용을 처리하기 위해 충돌 처리를 하고 있습니다. 역시 처리를 단순화하기 위해 다음과 같은 가정을 깔고 있습니다.

1. 충돌 처리에 가담하는 물체는 공(ball)과 장애물(obstacle)뿐입니다.
2. 움직이는 물체는 공뿐이고, 장애물은 고정된 물체(fixture)입니다.
3. 물체의 모양(shape)은 정해져 있습니다. 공은 항상 원(circle)이고, 장애물은 항상 직사각형(rectangle)입니다.

충돌 처리 결과는 크게 두 가지입니다.

1. 물체가 다른 물체를 뚫고 지나가지 못합니다.
2. 물체가 다른 물체와 부딪혀 튕겨져 나갑니다.

또한 프레임이 이산적이기 때문에, 다음과 같은 상황도 처리해야 합니다.

before frame

after frame

corrected position

penetration depth

단순히 현재 위치에 변위를 더하기 때문에, 공이 장애물을 관통하는 경우가 생깁니다. 따라서 충돌 시점에서 관통된 변위를 구해 올바른 방향으로 튕겨나간 것으로 교정할 필요가 있습니다.

다음과 같은 방법으로 충돌을 판정하고 위치를 교정합니다.

1. 현재 위치와, 변위를 더해 새로 이동할 위치를 잇는 선분 P를 구합니다.
2. 장애물 중 하나를 골라 한 변의 선분 Q를 구합니다.
3. 선분 P와 Q의 교점을 찾습니다.  
   교점이 없다면, 변이 공의 이동 경로를 가로막지 않은 것입니다.
4. 교점이 있다면, 새 위치에서 교점을 빼 관통 깊이(penetration depth) 벡터를 구합니다.
5. 변을 구성하는 선분 Q로부터 법선 N을 구하고, 관통 깊이 벡터를 법선 N에 대해 mirror하여 반사된 벡터 V2를 구합니다.
6. 교점에서 현재 위치를 빼 충돌 직전까지의 변위 벡터 V1을 구하고, 여기에 위에서 구한 V2를 더해 교정된 위치를 구합니다.
7. 충돌 후 운동 방향이 바뀌므로 속도도 V2와 같은 방향이 되도록 만듭니다.
8. 장애물의 나머지 변에 대해서도 같은 처리를 수행합니다.
9. 다른 장애물에 대해서도 같은 처리를 수행합니다.

간단하게 요약하면, 현재 위치와 새 위치를 잇는 이동 경로와 겹치는 선분(변)이 있다면 충돌된 것이며 뚫고 들어간 부분만큼 위치와 속도를 교정합니다.

이때, 공은 질점(particle) 취급을 하고 있으므로 공의 중심점 좌표만 가지고는 공이 정말 충돌했는지 판단할 수 없으며, 공의 크기에 따라 충돌 지점도 달라집니다. 이 문제는 공의 위치는 그대로 두고, 변을 구할 때 법선 방향으로 공의 반지름(radius)만큼 이동시켜 해결합니다.

ball position – edge line

ball position – translated edge line

이렇게 하면 공의 반지름만큼 파고들어가도 충돌이 검출되지 않는 문제를 해결할 수 있습니다.

4. 게임 내 물체 관리

충돌 처리 할 때 공과 어떤 장애물(변)이 충돌했는지 판별하려면, 모든 장애물과 공을 다 검사해봐야만 합니다. 하지만 충돌하는 장애물은 몇 개에 불과한데 모든 장애물을 다 검사하면 계산 능력을 낭비하게 됩니다.

그러나 어떤 장애물이 공과 충돌할 가능성이 있는지 알려면 결국 충돌 검사를 해야 하는 딜레마가 있습니다.

이 문제를 해결하기 위해, 충돌할 가능성이 없는 장애물을 검사 대상에서 빠르게 제외시킬 수 있도록 Quad Tree 구조를 도입합니다.

Space partitioning by Quad Tree

Quad Tree는 개념적인 직사각 공간을 재귀적으로 밭 전(田)자 모양으로 나누어, 자식 노드가 4개가 되도록 하는 트리입니다.

트리의 각 노드에는 이 공간에 속한 장애물들의 목록이 들어있습니다. 만약 공이 위 그림에서 보이는 파란색 노드 안에서만 움직였다면, 그 외 노드(흰 부분)에 속한 장애물과는 충돌할 가능성이 없습니다. 따라서 파란색 노드 안에서만 충돌 처리를 하면 되기 때문에 처리량이 줄어듭니다.

어떤 노드의 부모 노드는 자식 노드의 공간을 모두 포함하고 있으므로, 공이 이동하는 변위를 모두 포함하는 가장 작은 노드를 빠르게 찾아내서 그 안에 있는 장애물과만 충돌 처리를 하면 훨씬 효율적입니다.