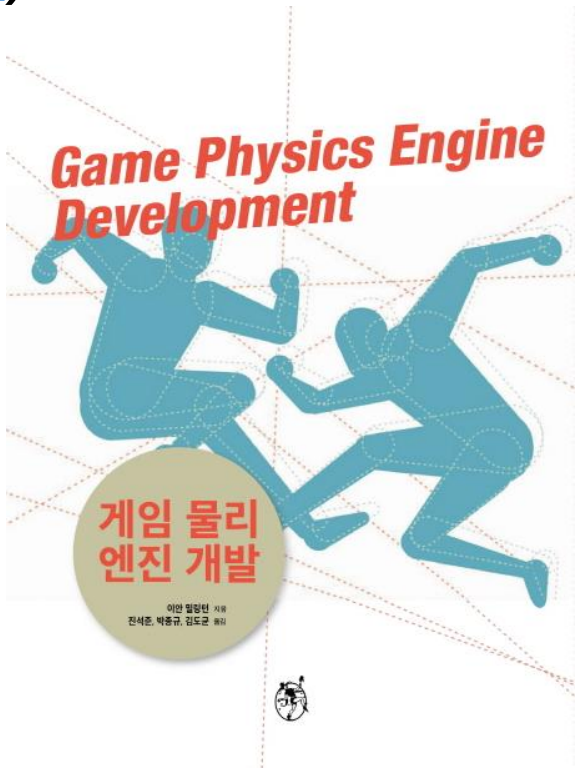
게임 물리 엔진 개발

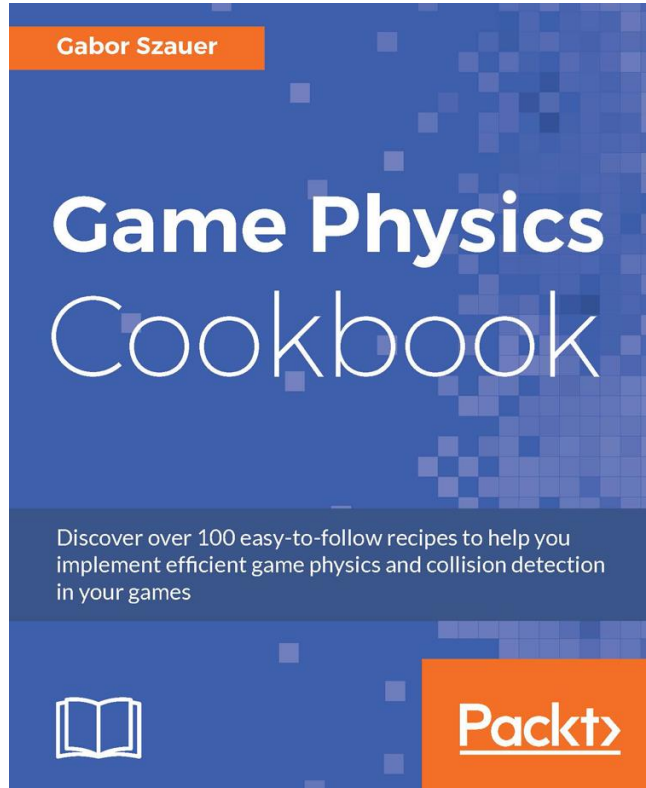
‘게임 물리 엔진 개발’이라는 책을 따라서 게임 물리 엔진을 제작하여 개인 프로젝트에 적용해본 경험을 공유

2007년도에 출간, 2016년도에 번역된 책이고 이 책을 기반으로 발표한 12년도 NDC 발표 자료도 있음
(https://www.slideshare.net/ohyecloudy/ndc12-12668524)

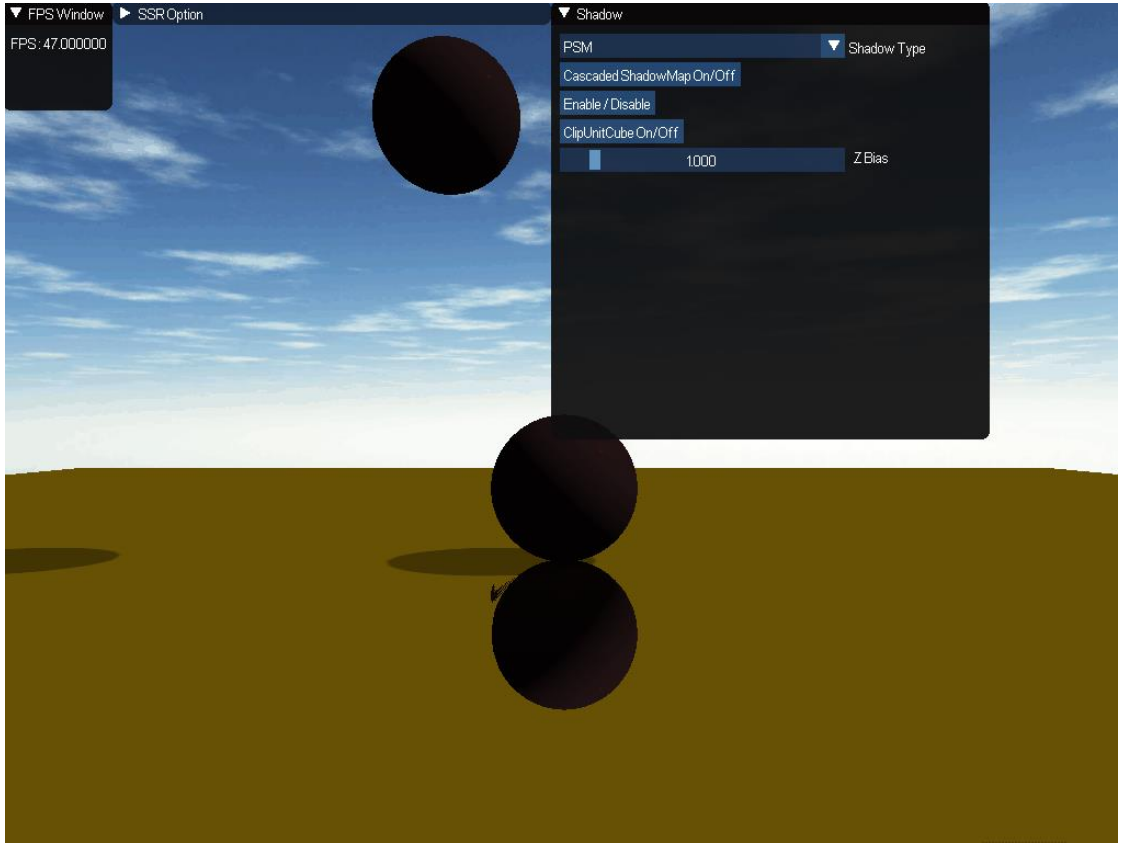


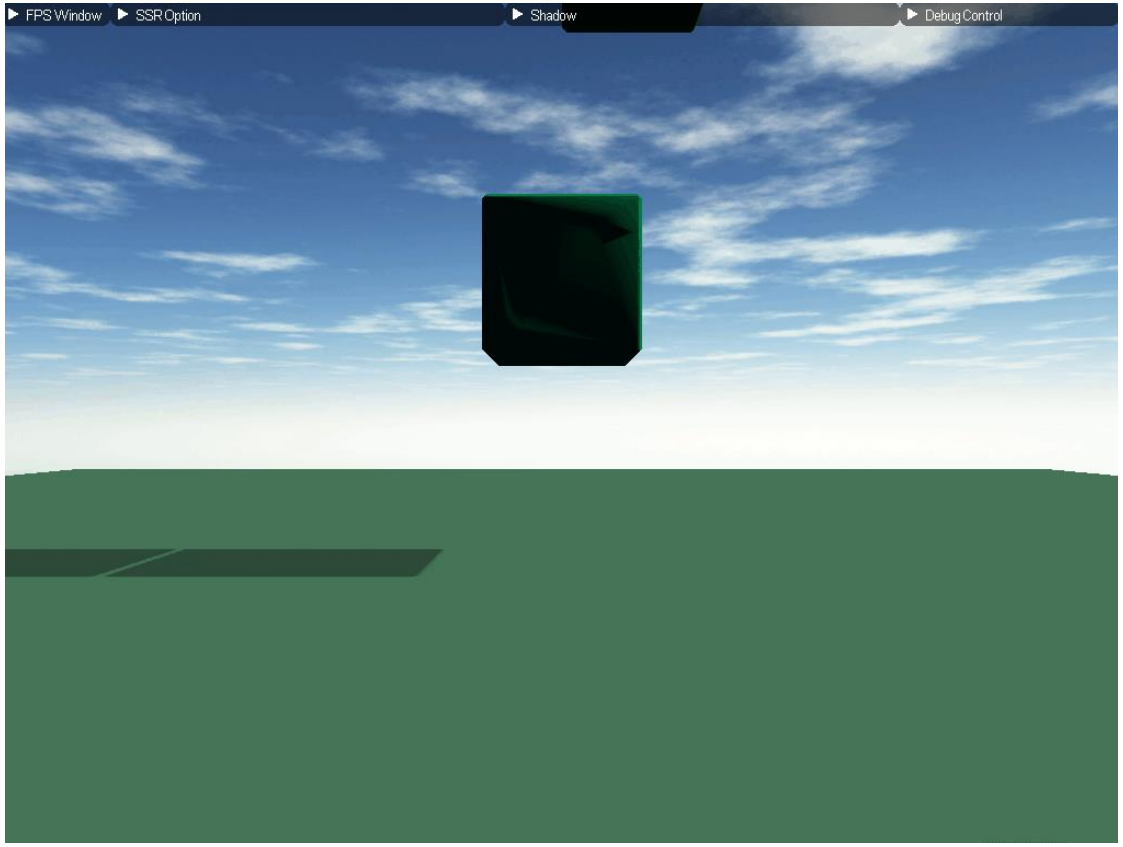
국내 번역서 중 기초적이지만 완성된 형태의 물리 엔진 코드를 제공하는 유일한 책 (…이라고 생각. 다른 책이 있다면 알려주세요.)

원서까지 포함하면 비교적 최근에 출간된 ‘Game Physics Cookbook’ 이 온전한 물리 엔진 코드를 제공하는 것으로 보임



구현하여 프로젝트에 적용한 모습은 대략 다음과 같음.gif





Source code (https://github.com/xtozero/SSR/tree/experimental)

물리는 매우 광범위한 영역을 다루는 학문

위키에서 물리학의 핵심 이론을 살펴보면

핵심 이론 [편집]

물리학이 넓은 범위에 걸친 다양한 주제를 다룸에도 불구하고 모든 물리학자들이 공통적으로 사용하는 핵심 이론들이 있다. 이들 이론에 대한 연구는 여전히 활발히 지속되고 있지만, 그 중에 근본적으로 잘못된 이론이 있으리라고 믿는 물리학자는 거의 없다. 물리 연구의 기본 도구 역할을 하는 이 이론들 각각은 그 적용 범위 내에서 기본적으로 옳은 것으로 믿어지고 있는데, 예를 들어 원자보다 크고 천체에 비해 매우 가벼우며 광속보다 훨씬 느리게 움직이는 일상적인 물체의 움직임은 고전역학으로 비교적 정확히 기술된다. 이러한 특성 때문에 이 이론들은 모든 물리학도들이 기본적으로 이해해야 하는 필수 과목이기도 하다.

- 고전역학 또는 고전 물리학
- 열역학 또는 통계역학
- 전자기학
- 상대성이론
- 양자역학

이론	주요 논제	개념
고전역학	뉴턴의 운동법칙, 라그랑주 역학, 해밀턴 역학, 혼돈 이론, 음향학, 유체역학, 연속체 역학	차원, 공간, 시간, 운동, 길이, 속도, 질량, 운동량, 힘, 에너지, 각운동량, 돌림힘, 보존 법칙, 조화 진동자, 파동, 일, 일률,
전자기학	정전기학, 전기, 자기, 맥스웰 방정식, 광학	전하, 전류, 전기장, 자기장, 전자기장, 전자기파, 자기홀극
열역학과 통계역학	열기관, 기체분자운동론, 상전이, 임계현상	볼츠만 상수, 엔트로피, 자유에너지, 열, 상태합, 온도
상대성이론	특수상대성이론, 일반상대성이론	등가 원리, 사차원 운동량, 기준좌표계, 시공간, 빛의 속도
양자역학	경로 적분 형식, 슈뢰딩거 방정식, 불확정성 원리, 양자 마당 이론	해밀토니언 연산자, 동일입자, 플랑크 상수, 양자 얽힘, 양자 조화 진동자, 파동함수, 영점 에너지

Q. 그럼 게임 물리 엔진이 다루는 영역은 어디인가?

A. 고전역학

고전 역학은 물체에 작용하는 힘과 운동의 관계를 설명

즉 게임 물리 엔진은 게임 물체에 힘을 작용 했을 때 물체가 어떻게 움직이는지를 시뮬레이션 함

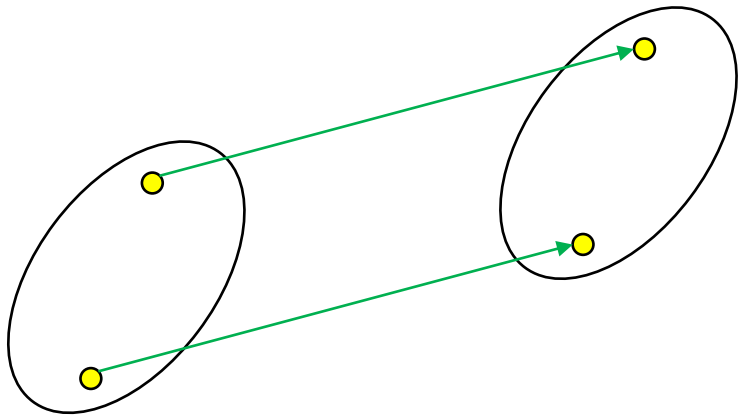
물체는 형태가 고정되어 변하지 않는 강체(Rigid Body), 형태가 변형되는 연체(Soft Body)를 주로 다루는데 여기서는 강체만 다룸

강체의 운동

강체의 운동은 크게 병진 운동(Translational Motion)과 회전 운동(Rotational Motion)으로 나뉠 수 있음

병진 운동은 모든 질점들이 평행하게 동일 거리를 움직이는 운동

병진 운동의 결과로 물체의 위치(p)가 변하게 된다



물체가 얼마나 빠르게 이동하였는지를 물체의 위치 변화를 시간 간격으로 나눠 표현하는데 이것이 속도($\vec{v}$)

$$\vec{v} = \lim_{\Delta t \to 0} \frac{\Delta x}{\Delta t} = \frac{dx}{dt} = \dot{p}$$

시간 간격에 따라 속도가 변하는 정도를 나타내는 물리량도 있는데 바로 가속도($\vec{a}$)

$$\vec{a} = \lim_{\Delta t \to 0} \frac{\Delta v}{\Delta t} = \frac{dv}{dt} = \ddot{p}$$

뉴턴 운동 제 2법칙을 통해서 힘과 물체의 운동 관계를 정리하면…

$$\vec{F} = \frac{d}{dt}(m\vec{v}) = m\frac{dv}{dt} = m\vec{a}$$

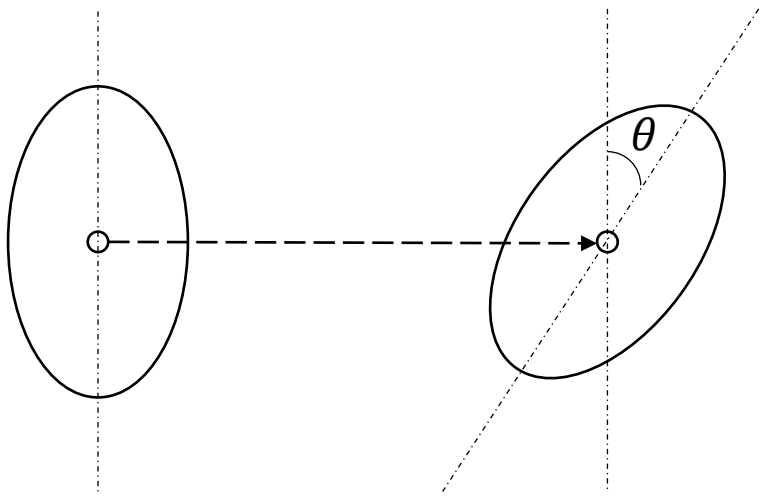
$\vec{F}$는 물체에 작용하는 힘, m은 물체의 질량

즉 힘은 물체의 가속도를 변화시키고 가속도는 속도를 속도는 물체의 위치를 변화

질량은 힘에 저항하는 정도라고 생각할 수 있음

회전운동은 어떤 기준 축을 중심으로 물체가 도는 운동

병진운동으로 물체의 위치가 변하듯 회전운동을 하면 물체의 각도(θ) 가 변함



속도와 마찬가지로 물체의 각도가 얼마나 빠르게 변했는지를 특정 시간 간격동안 각이 변한 정도로 표현할 수 있는데 이것이 각속도(ω)

$$\vec{\omega} = \lim_{\Delta t \to 0} \frac{\Delta \theta}{\Delta t} = \frac{d\theta}{dt} = \dot{\theta}$$

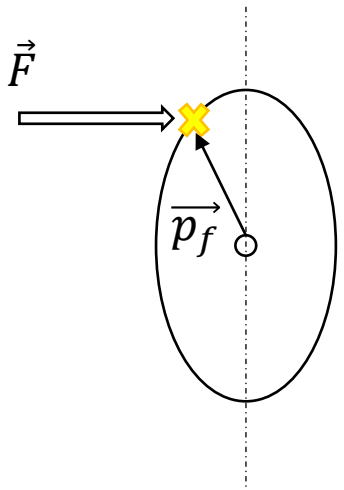
각속도가 얼마나 빠르게 변하는지를 나타내는 물리량도 가속도와 마찬가지로 존재하는데 이것이 각가속도(α)

$$\vec{\alpha} = \lim_{\Delta t \to 0} \frac{\Delta \omega}{\Delta t} = \frac{d\omega}{dt} = \ddot{\theta}$$

이제 힘과 회전운동의 관계를 살펴볼 수 있음, 회전운동에서 물체를 회전시키는 효력을 발생하는 힘은 토크($\vec{\tau}$)라고 하며 병진운동에서 살펴본 힘 $\vec{F}$과 다음과 같이 연관 됨

$$\vec{\tau} = \overrightarrow{p_f} \times \vec{F} = a\hat{d}$$

여기서 $\overrightarrow{p_f}$은 힘이 적용되는 지점이며, 물체의 중심(대체로 무게 중심)을 기준으로 상대적인 위치, a는 토크의 크기, $\hat{d}$는 토크가 작용하는 회전축의 방향



수학자 오일러가 병진운동에서의 뉴턴 운동 제 2법칙과 마찬가지로 회전에 대한 뉴턴 운동 제 2법칙을 다음과 같이 제시

$$\vec{\tau} = I\vec{\alpha}$$

I 는 관성 모멘트로 '물체의 회전 속도를 얼마나 변화시키기 어려운가?'를 나타내며 뉴턴 운동 제 2법칙의 질량과 같다고 생각하면 됨

관성 모멘트는 축이 주어졌을 때 물체를 이루는 알갱이들에 대하여 다음과 같이 계산할 수 있음

$$I_a = \sum_{i=1}^{n} m_i d_{pi\to a}^2$$

n은 알갱이의 개수, $d_{pi\to a}$는 회전축 a로 부터의 거리 I_a는 회전축 a를 기준으로 한 관성 모멘트

관성 모멘트는 질량과 같이 단일 수치로 나타낼 수 없고 회전축에 따라 달라짐

회전축을 정하면 단 하나의 값으로 얻어지겠지만 회전축은 임의로 잡는 것이 가능

Q. 그럼 무한히 많은 값을 갖고 다녀야 하는가?

A. 아니다. 모든 관성 모멘트의 값을 관성 텐서(Inertia tensor)라는 행렬에 채워 넣을 수 있다.

텐서? 행렬을 좀 더 일반화 한 것
0차 텐서가 스칼라, 1차가 벡터, 2차가 행렬이라고 생각할 수 있음

3차원 공간에서 강체의 관성 텐서(Inertia Tensor)는 3 x 3 행렬

대각선 방향 원소는 X축, Y축, Z축 중심으로 회전할 때 관성 모멘트

$$\begin{bmatrix} I_x & & \\ & I_y & \\ & & I_z \end{bmatrix}$$

I_x 는 X축을 중심으로 회전할 때의 관성 모멘트이다. I_y, I_z도 마찬가지

행렬의 나머지 원소는 관성곱(product of inertia)이라고 하며 다음과 같이 정의

$$I_{ab} = \sum_{i=1}^{n} m_i a_{p_i} b_{p_i}$$

a_{p_i} 는 물체의 무게중심으로부터 i번째 파티클까지의 거리의 a축 방향 성분

관성 모멘트와 달리 관성곱은 음수 값이 나올 수 있음, 웬만한 물체는 대부분 이 값이 0

관성곱을 관성 텐서에 넣어주면 자료 구조가 완성

$$I = \begin{bmatrix} I_x & -I_{xy} & -I_{xz} \\ -I_{xy} & I_y & -I_{yz} \\ -I_{xz} & -I_{yz} & I_z \end{bmatrix}$$

다수의 물체들은 관성 텐서를 쉽게 계산할 수 있는 식이 존재, 다음은 육면체 블록의 관성 텐서

$$I = \begin{bmatrix} \frac{1}{12}m(d_y^2 + d_z^2) & 0 & 0 \\ 0 & \frac{1}{12}m(d_x^2 + d_z^2) & 0 \\ 0 & 0 & \frac{1}{12}m(d_x^2 + d_y^2) \end{bmatrix}$$

다른 형태의 물체에 대한 관성 텐서는 여기를 참고
https://en.wikipedia.org/wiki/List_of_moments_of_inertia#List_of_3D_inertia_tensors

이제 강체의 운동을 계산하는 함수 Integrate를 살펴보면…

```
void RigidBody::Integrate( float duration )
{
    m_lastFrameAcceleration = m_acceleration;
    m_lastFrameAcceleration += m_forceAccum * m_inverseMass;

    CXMFLOAT3 angularAcceleration =
    XMVector3TransformNormal( m_torqueAccum, m_inverseInertiaTenserWorld );

    m_velocity += m_lastFrameAcceleration * duration;
    m_rotation += angularAcceleration * duration;

    m_velocity *= powf( m_linearDamping, duration );
    m_rotation *= powf( m_angularDamping, duration );

    m_position += m_velocity * duration;

    CXMFLOAT4 dq = m_rotation * duration;
    dq = XMQuaternionMultiply( m_orientation, dq ) * 0.5f;
    m_orientation += dq;
}
```

$$\vec{F} = m\vec{a}$$

$$\vec{a} = \frac{1}{m}\vec{F}$$

이제 강체의 운동을 계산하는 함수 Integrate를 살펴보면…

```
void RigidBody::Integrate( float duration )
{
    m_lastFrameAcceleration = m_acceleration;
    m_lastFrameAcceleration += m_forceAccum * m_inverseMass;

    CXMFLOAT3 angularAcceleration =
    XMVector3TransformNormal( m_torqueAccum, m_inverseInertiaTenserWorld );

    m_velocity += m_lastFrameAcceleration * duration;
    m_rotation += angularAcceleration * duration;

    m_velocity *= powf( m_linearDamping, duration );
    m_rotation *= powf( m_angularDamping, duration );

    m_position += m_velocity * duration;

    CXMFLOAT4 dq = m_rotation * duration;
    dq = XMQuaternionMultiply( m_orientation, dq ) * 0.5f;
    m_orientation += dq;
}
```

$\vec{\tau} = I\vec{\alpha}$

$\vec{\alpha} = I^{-1}\vec{\tau}$

이제 강체의 운동을 계산하는 함수 Integrate를 살펴보면…

```
void RigidBody::Integrate( float duration )
{
    m_lastFrameAcceleration = m_acceleration;
    m_lastFrameAcceleration += m_forceAccum * m_inverseMass;

    CXMFLOAT3 angularAcceleration =
    XMVector3TransformNormal( m_torqueAccum, m_inverseInertiaTenserWorld );

    m_velocity += m_lastFrameAcceleration * duration;
    m_rotation += angularAcceleration * duration;

    m_velocity *= powf( m_linearDamping, duration );
    m_rotation *= powf( m_angularDamping, duration );

    m_position += m_velocity * duration;

    CXMFLOAT4 dq = m_rotation * duration;
    dq = XMQuaternionMultiply( m_orientation, dq ) * 0.5f;
    m_orientation += dq;
}
```

$$\vec{a} = \lim_{\Delta t \to 0} \frac{\Delta v}{\Delta t} = \frac{dv}{dt} = \ddot{p}$$

$$\vec{\alpha} = \lim_{\Delta t \to 0} \frac{\Delta \omega}{\Delta t} = \frac{d\omega}{dt} = \ddot{\theta}$$

이제 강체의 운동을 계산하는 함수 Integrate를 살펴보면…

```
void RigidBody::Integrate( float duration )
{
    m_lastFrameAcceleration = m_acceleration;
    m_lastFrameAcceleration += m_forceAccum * m_inverseMass;

    CXMFLOAT3 angularAcceleration =
    XMVector3TransformNormal( m_torqueAccum, m_inverseInertiaTenserWorld );

    m_velocity += m_lastFrameAcceleration * duration;
    m_rotation += angularAcceleration * duration;

    m_velocity *= powf( m_linearDamping, duration );
    m_rotation *= powf( m_angularDamping, duration );

    m_position += m_velocity * duration;

    CXMFLOAT4 dq = m_rotation * duration;
    dq = XMQuaternionMultiply( m_orientation, dq ) * 0.5f;
    m_orientation += dq;
}
```

$$\vec{v} = \lim_{\Delta t \to 0} \frac{\Delta x}{\Delta t} = \frac{dx}{dt} = \dot{p}$$

$$\vec{\omega} = \lim_{\Delta t \to 0} \frac{\Delta \theta}{\Delta t} = \frac{d\theta}{dt} = \dot{\theta}$$

충돌 후 강체의 운동

여기까지가 힘에 의한 강체의 운동

강체에 작용하는 물리적 현상이 온전히 힘에 의한 것은 아님

강체는 운동을 통해서 다른 강체와 부딪힐 수 도 있음

물체가 서로의 운동에 간섭하는 현상이 충돌

그럼 충돌이 발생 했을 경우 강체의 운동이 어떻게 변하는가?

운동량 보존 법칙에 따르면 외부에서 힘이 가해지지 않는다면 뉴턴의 운동 법칙에 따라 총 운동량은 바뀌지 않음

두 물체가 충돌할 때 두 물체의 운동량의 합은 일정

$$m_1 v_1 + m_2 v_2 = m_1 v_1' + m_2 v_2'$$

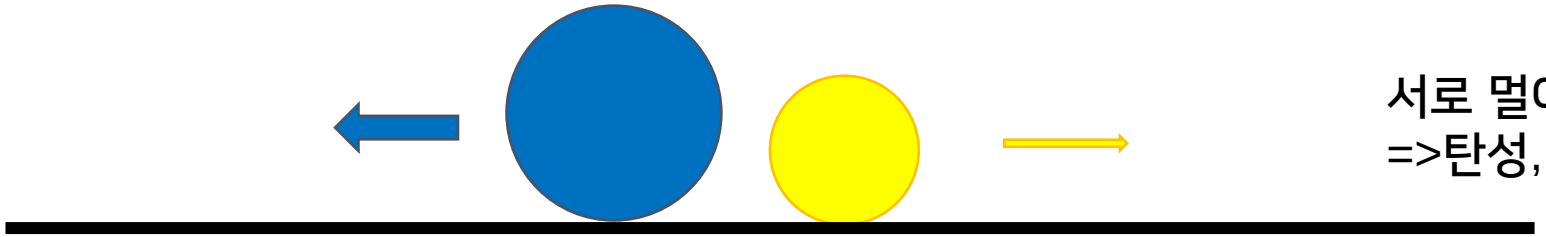
여기서 v_i 는 충돌 전 속도 v_i'는 충돌 후 속도

물체의 충돌 전후 속도의 비율은 반발 계수라하고 다음과 같이 정의됨

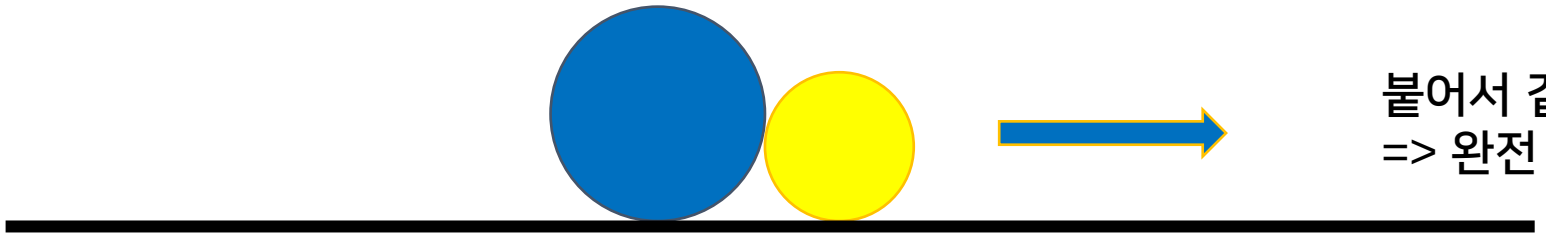
$$c = \frac{v_2' - v_1'}{v_1 - v_2} \ (0 \leq c \leq 1)$$

반발 계수는 0 ~ 1사이의 값을 가지며 0일 경우 완전 비탄성 충돌, 1일 경우 탄성 충돌, 그 외의 경우 비탄성 충돌

서로 접근하는 물체가 충돌했을 때



서로 멀어지는 방향으로 운동
=>탄성, 비탄성 충돌



붙어서 같은 방향으로 운동
=> 완전 비탄성 충돌

운동량 보존 법칙을 통해서 알 수 있는 것은 충돌로 인해서 속도가 변함

$$m_1 v_1 + m_2 v_2 = m_1 v_1' + m_2 v_2'$$

강체이기 때문에 충돌후에도 모양이나 크기가 변형되지 않음. 질량은 변하지 않음.

여기서 충격량(Impulse)이라는 물리량을 소개할 필요가 있음, 충격량은 어떤 정해진 시간 동안 운동량이 변화한 정도

$$g = p_1 - p_0 = \int_{t_0}^{t_1} \dot{p}(t)dt$$

여기서 p_i는 어떤 물체의 운동량 t_i는 시간, 운동량은 물체의 질량과 속도의 곱으로 나타낼 수 있으므로 충격량을 다음과 같이 정리할 수 있음

$$g = mv_1 - mv_0 = m\Delta v$$

그리고 힘이 시간에 일정하다면 다음과 같은 식으로 통해 충격량과 힘을 상호 변환 가능

$$g = m\Delta v = F\Delta t$$

이제 충격량과 힘, 속도간의 관계를 살펴보았는데 지금까지 다룬 충격량은 사실 선(Linear) 충격량으로 강체의 운동을 다루기 위해서는 각(angular) 충격량을 살펴볼 필요가 있음

각 충격량도 선 충격량과 크게 다르지 않음

각 충격량은 어떤 정해진 시간안에 각 운동량의 변화
여기서 각 운동량 $L = I\vec{\omega}$ 이므로 각 충격량은

$$u = I\overrightarrow{\omega_1} - I\overrightarrow{\omega_0} = I\Delta\vec{\omega}$$

마찬가지로 토크가 시간에 일정하다면 $u = \tau\Delta t$으로 표기할 수 있고 토크와 힘의 관계($\vec{\tau} = \overrightarrow{p_f} \times \vec{F}$) 에 의해 다음과 같이 정리 됨

$$u = \overrightarrow{p_f} \times g$$

선 충격량과 각 충격량을 구하면 충돌이 발견했을 때 강체의 운동이 어떻게 변하는지 계산할 수 있음

즉 물리엔진에서 충돌을 해결하는 과정은 충격량을 계산하는 과정

충격량을 계산하기 위해서 물체의 충돌을 검사하면서 추가적으로 계산해야 하는 정보가 두 가지 존재

경계 구(Bounding Sphere)의 충돌 검사 코드를 보면서 추가로 계산해야 하는 정보가 무엇인지 살펴보겠음

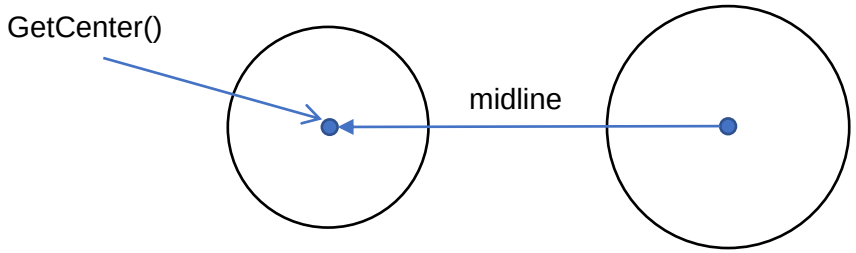
```
unsigned int SphereAndSphere( const BoundingSphere& lhs, RigidBody* lhsBody, const BoundingSphere& rhs, RigidBody* rhsBody,
CollisionData* data )
{
```

충돌여부 판단

```
	if ( data->m_contactsLeft <= 0 )
	{
		return COLLISION::OUTSIDE;
	}

	const CXMFLOAT3& lhsPos = lhs.GetCenter( );
	const CXMFLOAT3& rhsPos = rhs.GetCenter( );

	CXMFLOAT3 midline = lhsPos - rhsPos;
	float size = XMVectorGetX( XMVector3Length( midline ) );

	if ( size <= 0.f || size >= lhs.GetRadius( ) + rhs.GetRadius( ) )
	{
		return COLLISION::OUTSIDE;
	}

	CXMFLOAT3 normal = midline / size;

	Contact* contact = data->m_contacts;
```

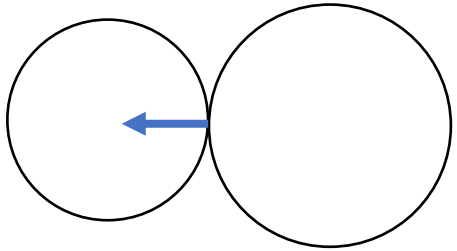
추가 정보 계산

```
	contact->SetContactNormal( normal );
	contact->SetContactPoint( lhsPos - CXMFLOAT3( midline * 0.5f ) );
	contact->SetPenetration( lhs.GetRadius( ) + rhs.GetRadius( ) - size );
	contact->SetBodyData( lhsBody, rhsBody, data->m_friction, data->m_restitution );

	data->AddContacts( 1 );
	return COLLISION::INTERSECTION;
}
```

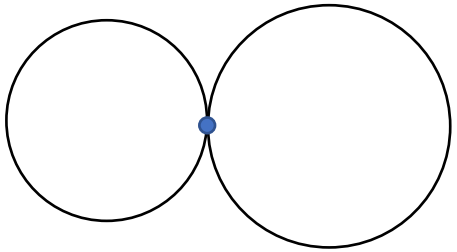


??? 두 가지라고 했는데 세 가지?

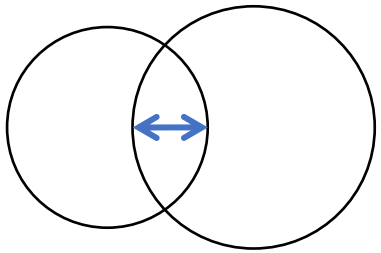
1. Contact Normal : 충돌 점에서의 법선 벡터



2. Contact Point : 충돌 점



번외. Penetration : 관통 깊이 (겹침 해소 과정에서 필요, 충격량 계산에서는 사용 X)



두 가지 추가 정보를 가지고 다음과 같은 순서로 충격량을 계산

1. 단위 충격량당 속도의 변화를 계산
2. 기대되는 속도의 변화량을 계산
3. 요구되는 속도 변화량을 기반으로 발생해야 하는 충격량을 계산
4. 충격량을 기반으로 선 성분과 각 성분을 분리하고 이를 각각의 물체에 반영

1. 단위 충격량당 속도의 변화를 계산

Q. 단위 충격량당 발생하는 속도 변화는 몇 가지가 있는가?
A. 두 가지

선 성분
앞에서 선 충격량과 속도의 변화의 관계를 살펴보았음

$$g = m\Delta v$$

이때 속도의 변화는 $\Delta v = \frac{g}{m}$, 즉 단위 충격량(충격량의 크기가 1)에서는…

$$\Delta v = \frac{1}{m}$$

각 성분

각 충격량과 각 속도의 변화는 다음과 같음

$$u = I\Delta\vec{\omega}$$

각 속도의 변화는 $\Delta\vec{\omega} = \mathrm{I}^{-1}\mathrm{u}$라고 할 수 있음, 이제 단위 충격량에서 각 충격량을 구하면 각 속도를 구할 수 있음

$$u = \overrightarrow{p_f} \times \hat{g}$$

p_f : 물체 무게중심에 대한 충돌점의 상대적 위치(충돌점 – 무게중심)

$\hat{g}$: 단위 충격량 여기서는 충돌점에서의 법선 벡터와 같다.

이제 단위 충격량당 각 속도와 선 속도를 하나의 선속도로 통합하면 됨, 각 속도로 인해 유도되는 회전 유도 속도($\dot{q}$)를 다음과 같이 계산할 수 있음

$$\dot{q} = \vec{\omega} \times \overrightarrow{p_f}$$

정리하자면 단위 충격량당 속도의 변화는 다음과 같이 계산

$$\frac{1}{m} + ((\mathrm{I}^{-1}(\overrightarrow{p_f} \times \hat{g})) \times \overrightarrow{p_f})$$

```
CXMFLOAT3 Contact::CalculateFrictionlessImpulse( const CXMFLOAT3X3* inverseInertiaTensor )
{
      CXMFLOAT3 deltaVelWorld = XMVector3Cross( m_relativeContactPosition[0], m_contactNormal );
      deltaVelWorld = XMVector3TransformNormal( deltaVelWorld, inverseInertiaTensor[0] );
      deltaVelWorld = XMVector3Cross( deltaVelWorld, m_relativeContactPosition[0] );

      float deltaVelocity = XMVectorGetX( XMVector3Dot( deltaVelWorld, m_contactNormal ) );

      deltaVelocity += m_body[0]->GetInverseMass( );

      if ( m_body[1] )
      {
            deltaVelWorld = XMVector3Cross( m_relativeContactPosition[1], m_contactNormal );
            deltaVelWorld = XMVector3TransformNormal( deltaVelWorld, inverseInertiaTensor[1] );
            deltaVelWorld = XMVector3Cross( deltaVelWorld, m_relativeContactPosition[1] );

            deltaVelocity += XMVectorGetX( XMVector3Dot( deltaVelWorld, m_contactNormal ) );

            deltaVelocity += m_body[1]->GetInverseMass( );
      }

      return CXMFLOAT3( m_desiredDeltaVelocity / deltaVelocity, 0.f, 0.f );
}
```

2. 충돌 후 기대되는 속도의 변화량을 계산

여기서는 분리 속도(Separating Velocity)가 도입

분리 속도의 반대 방향인 접근 속도(Closing Velocity)를 먼저 보면 접근 속도는 두물체가 서로 움직이는 속도의 합으로 다음과 같이 계산

$$v_c = v_a \cdot (\widehat{p_b - p_a}) + v_b \cdot (\widehat{p_a - p_b})$$

여기서 v_c는 접근 속도 p는 물체의 위치 벡터 $\hat{p}$는 단위 벡터를 의미

위 식을 다음과 같이 간단하게 할 수 있음

$$v_c = -(v_a - v_b) \cdot (\widehat{p_a - p_b})$$

접근 속도의 방향을 반대로 하면 분리 속도인데 부호만 바꾸면 됨

$$v_s = (v_a - v_b) \cdot (\widehat{p_a - p_b})$$

분리 속도는 충돌 후에 반발 계수에 비례하여 변하게 되는데 이를 통해서 충돌 후 기대되는 속도의 변화량을 알 수 있음

$$v_s' = -cv_s$$
$$\Delta v_s = v_s' - v_s$$
$$\Delta v_s = -cv_s - v_s$$
$$\Delta v_s = -(1 + c)v_s$$

$$c = \frac{v_2' - v_1'}{v_1 - v_2} \ (0 \le c \le 1)$$

다음 코드는 $v_a \cdot (\widehat{p_a - p_b})$ 을 계산

```
CXMFLOAT3 Contact::CalculateLocalVelocity( int bodyIndex, float duration )
{
	RigidBody* thisBody = m_body[bodyIndex];

	CXMFLOAT3 velocity = XMVector3Cross( thisBody->GetRotation( ), m_relativeContactPosition[bodyIndex] );
	velocity += thisBody->GetVelocity( );

	CXMFLOAT3 contactVelocity = XMVector3TransformNormal( velocity, XMMatrixTranspose( m_contactToWorld ) );

	CXMFLOAT3 accVelocity = thisBody->GetLastFrameAcceleration( ) * duration;

	accVelocity = XMVector3TransformNormal( accVelocity, XMMatrixTranspose( m_contactToWorld ) );

	accVelocity.x = 0;

	contactVelocity += accVelocity;

	return contactVelocity;
}
```

회전 유도 속도

$v_a \cdot (\widehat{p_a - p_b})$ 에서 $v_b \cdot (\widehat{p_a - p_b})$ 를 빼면 분리 속도

```
m_contactVelocity = CalculateLocalVelocity( 0, duration );
if ( m_body[1] )
{
		m_contactVelocity -= CalculateLocalVelocity( 1, duration );
}
```

분리 속도를 계산하면 $\Delta v_s = -(1+c)v_s$를 통해 충돌 후 기대되는 속도의 변화량을 계산할 수 있음

```
void Contact::CalculateDesiredDeltaVelocity( float duration )
{
	constexpr float velocityLimit = 0.25f;

	float velocityFromAcc = 0;

	if ( m_body[0]->IsAwake( ) )
	{
		velocityFromAcc += XMVectorGetX( XMVector3Dot( m_body[0]->GetLastFrameAcceleration( ) * duration, m_contactNormal ) );
	}

	if ( m_body[1] && m_body[1]->IsAwake( ) )
	{
		velocityFromAcc -= XMVectorGetX( XMVector3Dot( m_body[1]->GetLastFrameAcceleration( ) * duration, m_contactNormal ) );
	}

	float thisRestitution = m_restitution;
	if ( fabsf( m_contactVelocity.x ) < velocityLimit )
	{
		thisRestitution = 0.f;
	}

	m_desiredDeltaVelocity = -m_contactVelocity.x - thisRestitution * ( m_contactVelocity.x - velocityFromAcc );
}
```

3. 요구되는 속도 변화량을 기반으로 발생해야 하는 충격량을 계산

발생해야 하는 충격량은 매우 간단하게 계산할 수 있음

충돌 후 기대되는 속도의 변화 v와 단위 충격량에 따른 속도 변화 d를 알기 때문에 충격량은 다음과 같이 계산

```
return CXMFLOAT3( m_desiredDeltaVelocity / deltaVelocity, 0.f, 0.f );
```

$$g = \frac{v}{d}$$

4. 충격량을 기반으로 선 성분과 각 성분을 분리하고 이를 각각의 물체에 반영

앞에서 다뤘던 공식을 떠올리며 코드를 보면…

```
CXMFLOAT3 impulsiveTorque = XMVector3Cross( m_relativeContactPosition[0], impulse );
rotationChange[0] = XMVector3TransformCoord( impulsiveTorque, inverseInertiaTensor[0] );
velocityChange[0] = impulse * m_body[0]->GetInverseMass( );

m_body[0]->AddVelocity( velocityChange[0] );
m_body[0]->AddRotation( rotationChange[0] );

if ( m_body[1] )
{
	impulsiveTorque = XMVector3Cross( impulse, m_relativeContactPosition[1] );
	rotationChange[1] = XMVector3TransformCoord( impulsiveTorque, inverseInertiaTensor[1] );
	velocityChange[1] = impulse * -m_body[1]->GetInverseMass( );

	m_body[1]->AddVelocity( velocityChange[1] );
	m_body[1]->AddRotation( rotationChange[1] );
}
```

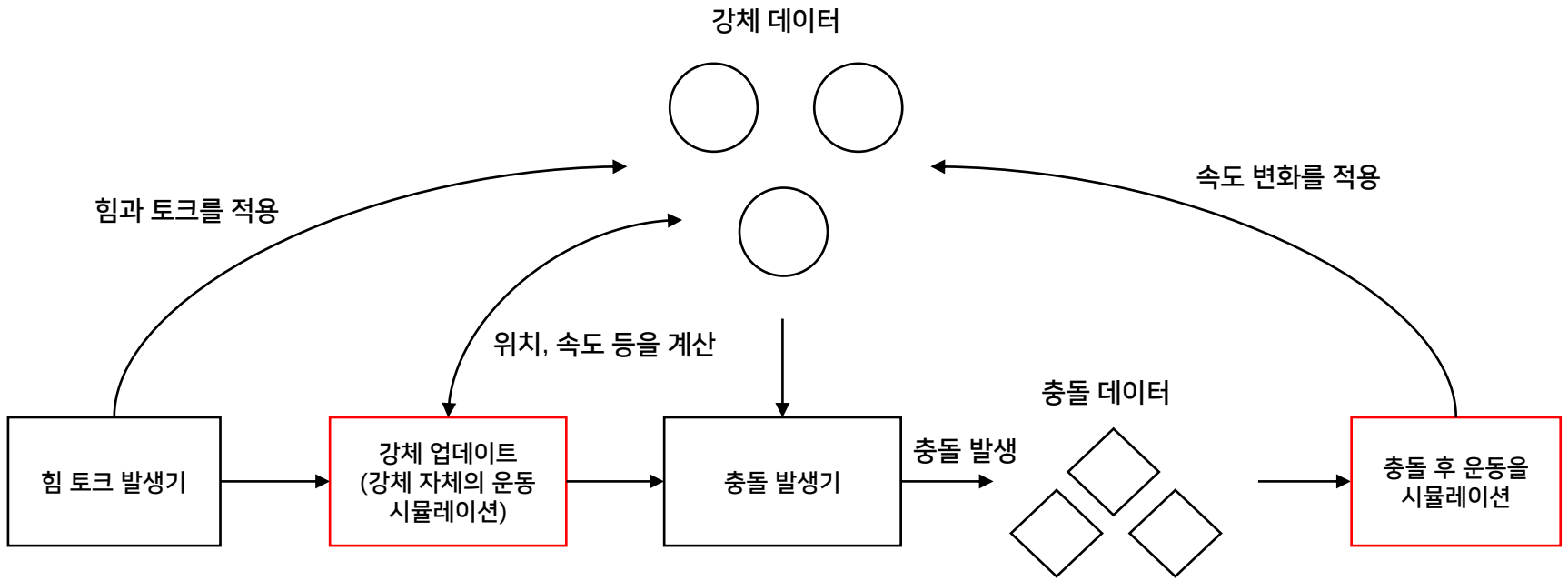
$$\Delta\vec{\omega} = (\mathrm{I}^{-1}(\overrightarrow{p_f} \times \hat{g}))$$

$$\Delta v = \frac{g}{m}$$

작용 반작용의 법칙에 따라 방향만 반대로 바꿔서 적용한다.

강체 자체의 운동과 충돌 후의 운동을 시뮬레이션하는 것이 여기서 다루는 물리 엔진의 기본적인 부분이라고 할 수 있음

물리 엔진의 전체적인 동작 과정을 그림으로 간략하게 표현하면 다음과 같은데 이 ppt는 붉은 박스 부분만 다루었음
(세부 내용에 관심있으시면 책을 참고)



구현하면서 겪은 시행착오

구현해볼 마음이 없으신 분은 패스

샘플 코드의 BVH 구현 중 잘 동작하지 않았던 부분

충돌 발생기의 충돌 가속을 위해서 BVH(Bounding Volume Hierarchy)를 사용

책에 실린 코드가 예제 프로그램에서는 실제로 사용되지 않고 있음

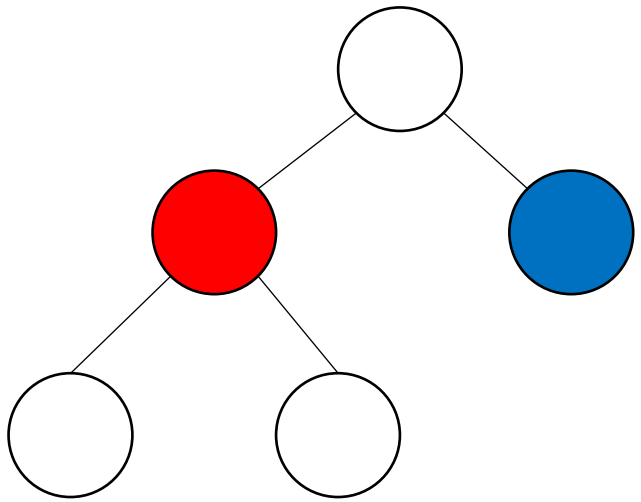
```
for (Box *box = boxData; box < boxData+boxes; box++)
{
    if (!cData.hasMoreContacts()) return;
    cyclone::CollisionDetector::boxAndHalfSpace(*box, plane, &cData);

    // Check for collisions with each shot
    for (AmmoRound *shot = ammo; shot < ammo+ammoRounds; shot++)
    {
        if (shot->type != UNUSED)
        {
            if (!cData.hasMoreContacts()) return;

            // When we get a collision, remove the shot
            if (cyclone::CollisionDetector::boxAndSphere(*box, *shot, &cData))
            {
                shot->type = UNUSED;
            }
        }
    }
}
```

이렇게 일일이 검사…

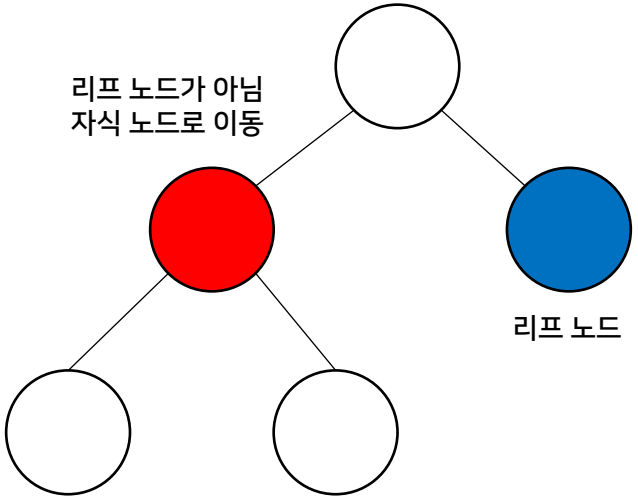
문제가 된 함수는 GetPotentialContactsWith 함수

해당 함수는 충돌할 가능성이 있는 모든 강체의 짝을 찾는 함수

함수의 동작을 간단히 그림으로 살펴보면…

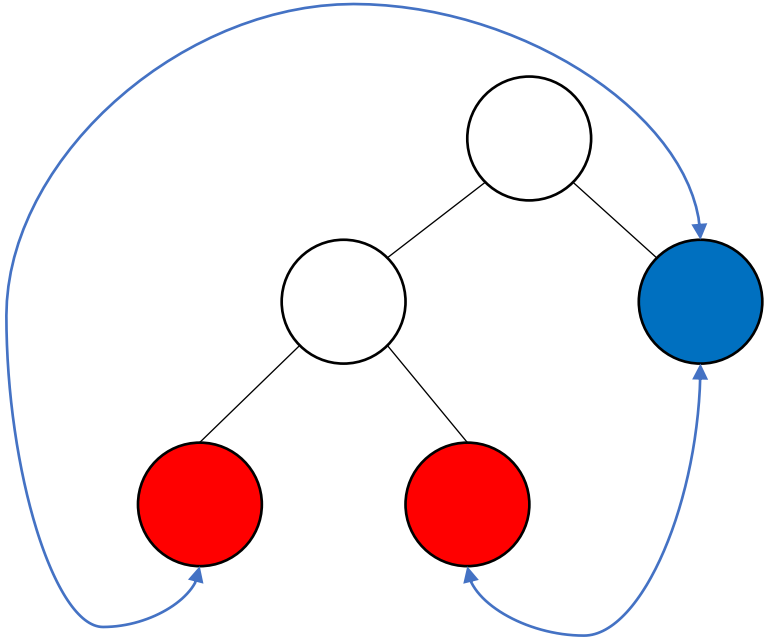


루트 노드의 자식 노드끼리 다음의 조건을 검사
1. 리프 노드인지?
2. 리프 노드가 아니라면 노드의 볼륨 크기가 다른 자식 노드 보다 큰지



리프 노드이면 충돌할 가능성이 있는지 체크할 대상으로 선정

리프 노드가 아니거나 노드의 볼륨 크기가 다른 노드 보다 크다면 자식 노드로 진행



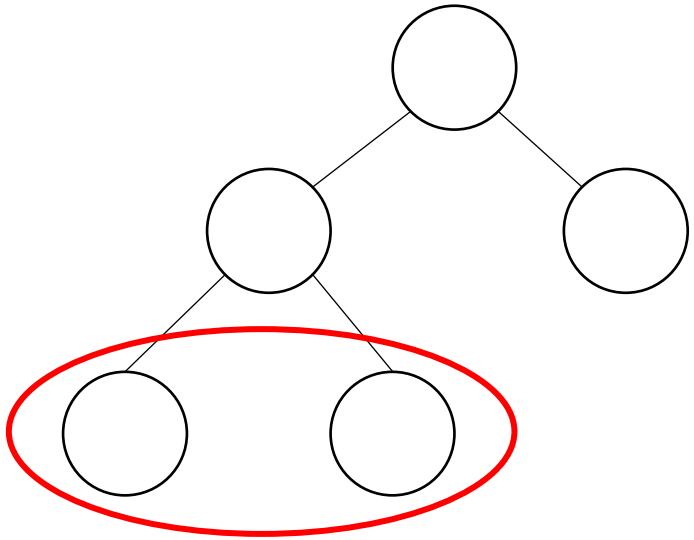
자식 노드가 리프 노드라면 충돌할 가능성이 있는지 검사

왼쪽 그림의 화살표로 묶인 쌍을 검사하게 됨

이를 재귀적으로 반복

문제는 자식 노드끼리는 충돌 가능성이 있다고 판단되지 않을 수 있음

이전 그림에서 이 두 노드끼리는 충돌 가능성 검사가 이뤄지지 않음



그래서 충돌 검사가 제대로 이뤄지지 않아 물체가 통과하는 현상이 발생

다음과 같이 자식 노드간 충돌 가능성을 검사하도록 수정.

```
template <typename BoundingVolumeClass, typename RigidBody>
unsigned int BVHNode<BoundingVolumeClass, RigidBody>::GetPotentialContactsWith( const BVHNode<BoundingVolumeClass, RigidBody>* other, PotentialContact<RigidBody>* contacts,
unsigned int limit ) const
{
	if ( Overlaps( other ) == false || limit == 0 )
	{
		return 0;
	}

	if ( IsLeaf( ) && other->IsLeaf( ) )
	{
		contacts->m_body[0] = m_body;
		contacts->m_body[1] = other->m_body;
		return 1;
	}

	if ( other->IsLeaf( ) || ( ( IsLeaf( ) == false ) && m_volume.GetSize( ) >= other->m_volume.GetSize( ) ) )
	{
		unsigned int count = m_children[0]->GetPotentialContactsWith( m_children[1], contacts, limit );

		if ( limit > count )
		{
			count += m_children[0]->GetPotentialContactsWith( other, contacts + count, limit - count );
		}

		if ( limit > count )
		{
			count += m_children[1]->GetPotentialContactsWith( other, contacts + count, limit - count );
		}

		return count;
	}

	// 생략…
}
```

Reference

게임 물리 엔진 개발

https://ko.wikipedia.org/wiki/고전물리학

https://ko.wikipedia.org/wiki/뉴턴_운동_법칙

https://ko.wikipedia.org/wiki/운동량

https://ko.wikipedia.org/wiki/각가속도

https://ko.wikipedia.org/wiki/각속도