게임엔진

제8강 애니메이션

학습 안내

■ 학습 목표

□ 애니메이션의 원리를 학습하고, 캐릭터 애니메이션을 구현해봄으로써 게임 애니메이션의 기본 지식을 익힌다.

■ 학습 내용

□ 애니메이션 기초 이론.
□ 오우거 엔진의 애니메이션 기법
□ 캐릭터의 걷기 애니메이션 구현.
□ 캐릭터 이동 속도의 조절.
□ 공간을 돌아다니는 캐릭터 애니메이션 구현.

애니메이션 원리

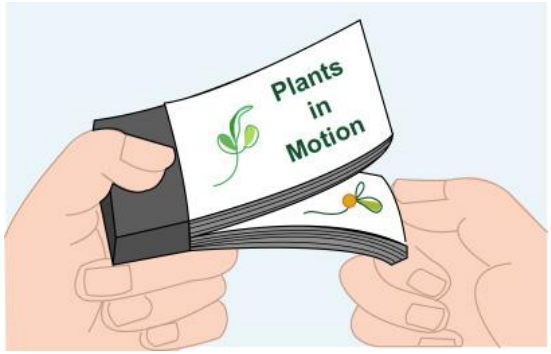
■ 애니메이션은 일련의 정지 이미지(still image)를 연속적으로 보여주어, 보는 사람으로 하여금 이미지들을 연속된 동작으로 착각하도록 함.

■ 잔상효과(persistence of vision)

□ 이미지가 이미 사라졌음에도 불구하고 사람의 눈이나 뇌에 계속 남아 있으려는 경향.

■ 일반적으로 초당 15장 이상의 그림이 보여지면 자연스러운 움직임을 얻을 수 있음. 텔레비젼, 영사기, 비디오 플레이어는 각각이 표시하는 초당 프레임율이 다르지만 모두 잔상효과를 이용함.

Animation frames



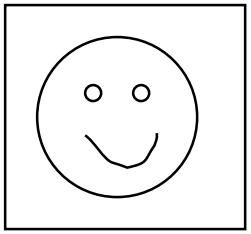
Sequential viewing

애니메이션의 방법

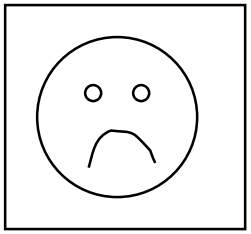
- **키프레임 애니메이션(Key-frame animation)**
  - 수동으로 애니메이션을 구성
- **모션 캡쳐(Motion capture)**
  - 애니메이션을 측정하여 기록함.
- **절차적 애니메이션(Procedural Animation)**
  - 자동으로 애니메이션을 생성.
  - 일종의 시뮬레이션

키프레임(Key Frame)과 보간(Interpolation)

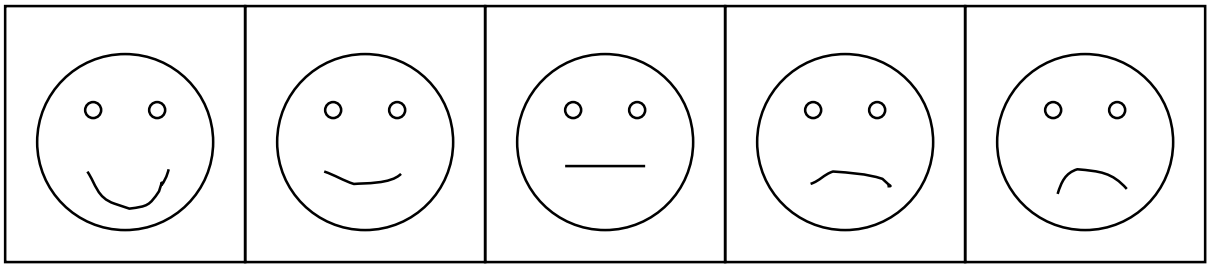
- 움직임의 특징이 되는 키 프레임을 지정(사용자가 수동으로)
- 보간 알고리즘을 이용하여 키 프레임 사이의 중간 프레임들을 생성해냄.



Key Frame #1



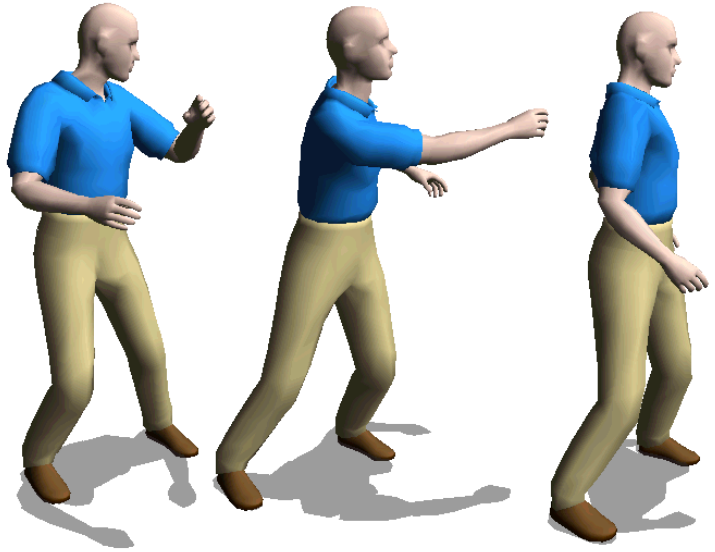
Key Frame #2

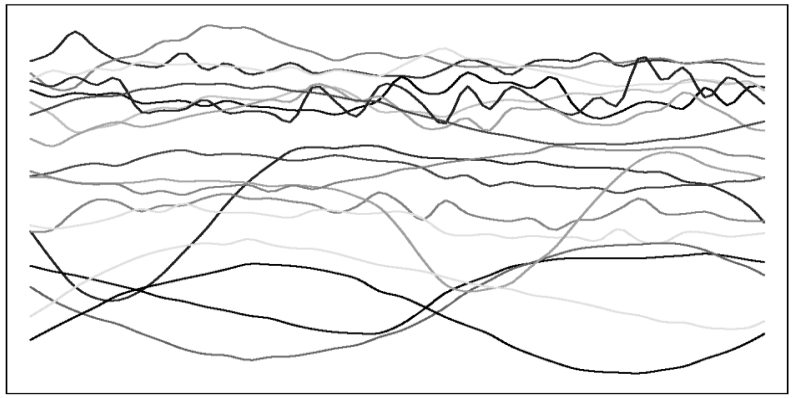


Interpolation

키프레임 애니메이션의 특징

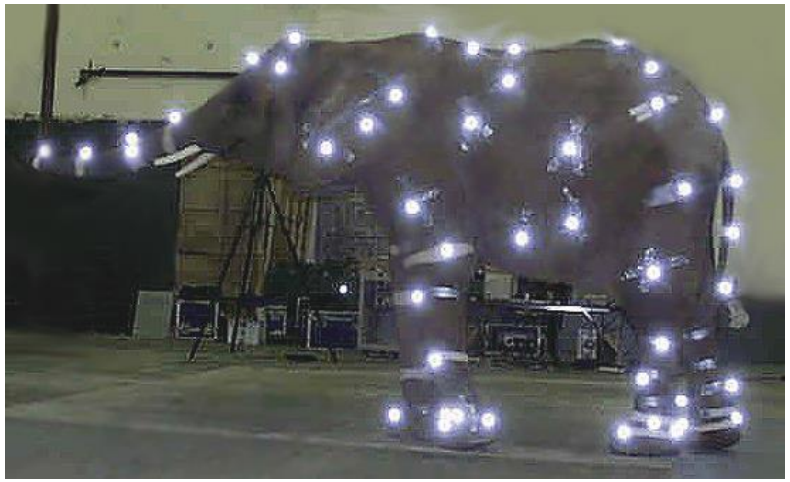
- 숙련된 애니메이터가 필요함.
- 수작업으로 키프레임을 만들어야 하기 때문에, 시간과 비용이 많이 필요.
- 고품질 / 고비용





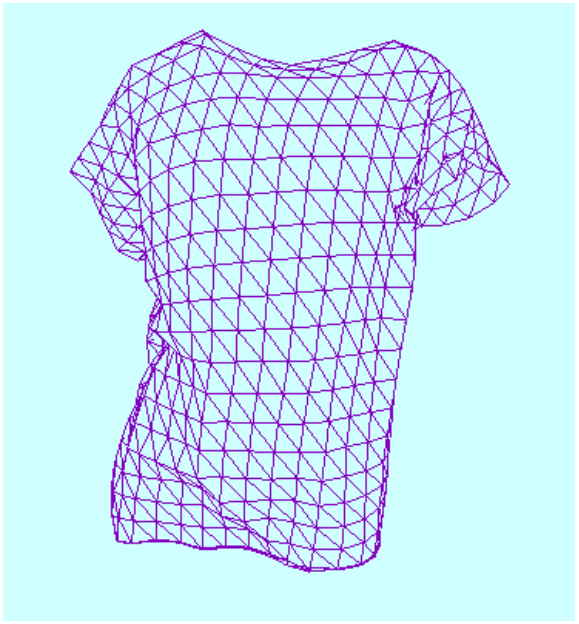
모션캡쳐

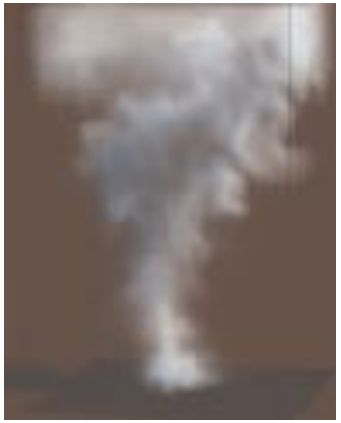
- 사람의 몸에 센서 또는 마커를 부착하여, 오브젝트의 움직임을 동작 데이터로서 그대로 측정하고 기록하는 방식.
  - 광학식 및 기계식
- 기록된 데이터를 가공하는데 긴 시간이 걸림.
- 적절한 품질 / 적절한 비용.



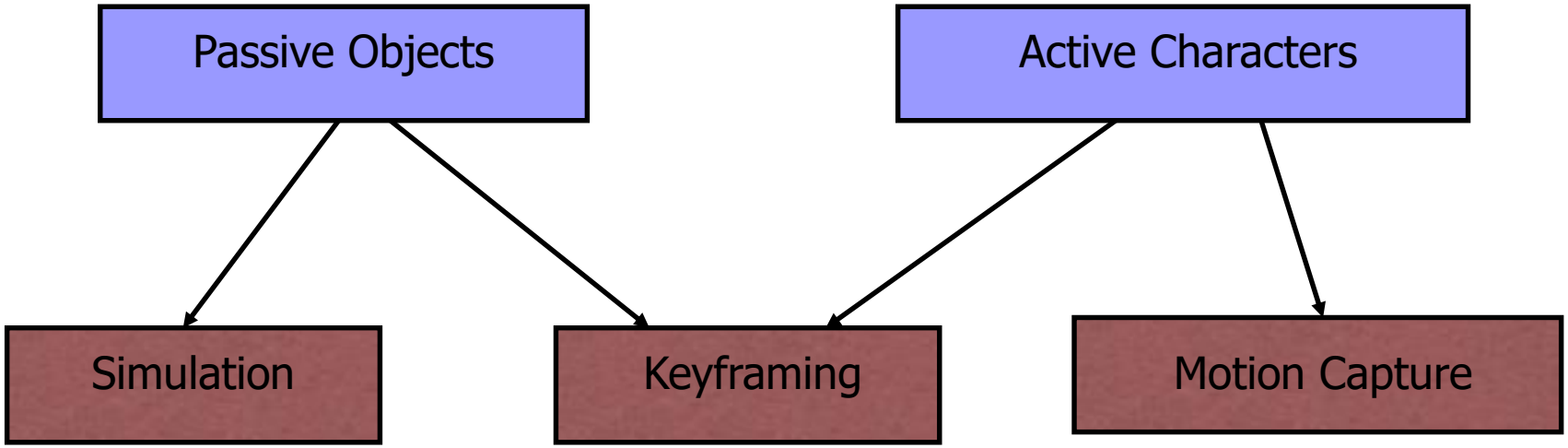
절차적 애니메이션(Procedural Animation)

- 수학적인 시뮬레이션을 이용하여 물체의 움직임을 자동으로 생성하는 방식.
- 입립자 시스템(Particle system)이나 유동 표면(Flexible surface)과 같은 자연 현상이나 사람이 직접 제어하기에는 복잡한 시스템의 애니메이션 구현에 유리함.
  - 유체, 안개, 연기, 구름, 옷 표면
- 실시간 구현을 위해서는 고성능의 하드웨어 필요.





애니메이션 방법의 선택



오우거 엔진의 애니메이션 기법

- **Skeletal Animation**
  - Mesh animation using a skeletal structure to determine how the mesh deforms.
- **Vertex Animation**
  - Mesh animation using snapshots of vertex data to determine how the shape of the mesh changes.
- **SceneNode Animation**
  - Animating SceneNodes automatically to create effects like camera sweeps, objects following predefined paths, etc.

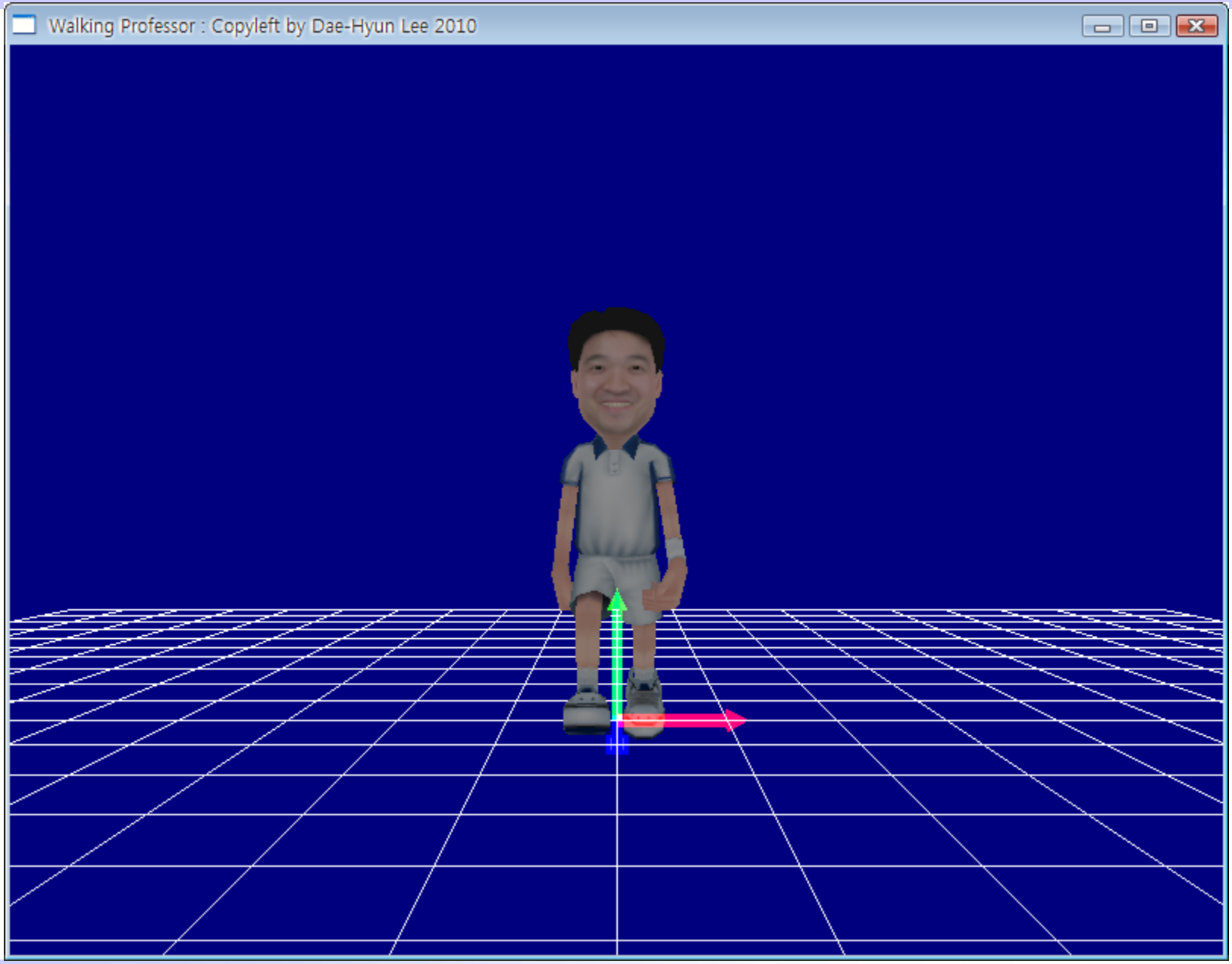
main.cpp


```
MainListener(Root* root, OIS::Keyboard *keyboard):mKeyboard(keyboard),mRoot(root)
{

  mProfessorNode = mRoot->getSceneManager("main")->getSceneNode("Professor");
  mProfessorEntity = mRoot->getSceneManager("main")->getEntity("Professor");
  mAnimationState = mProfessorEntity->getAnimationState("Walk");
  mAnimationState->setEnabled(true);
  mAnimationState->setLoop(true);

}
```

main.cpp


```
bool frameStarted(const FrameEvent &evt)
{
  mProfessorNode->translate(Vector3(0, 0, 50.0f * evt.timeSinceLastFrame));

  return true;
}
```

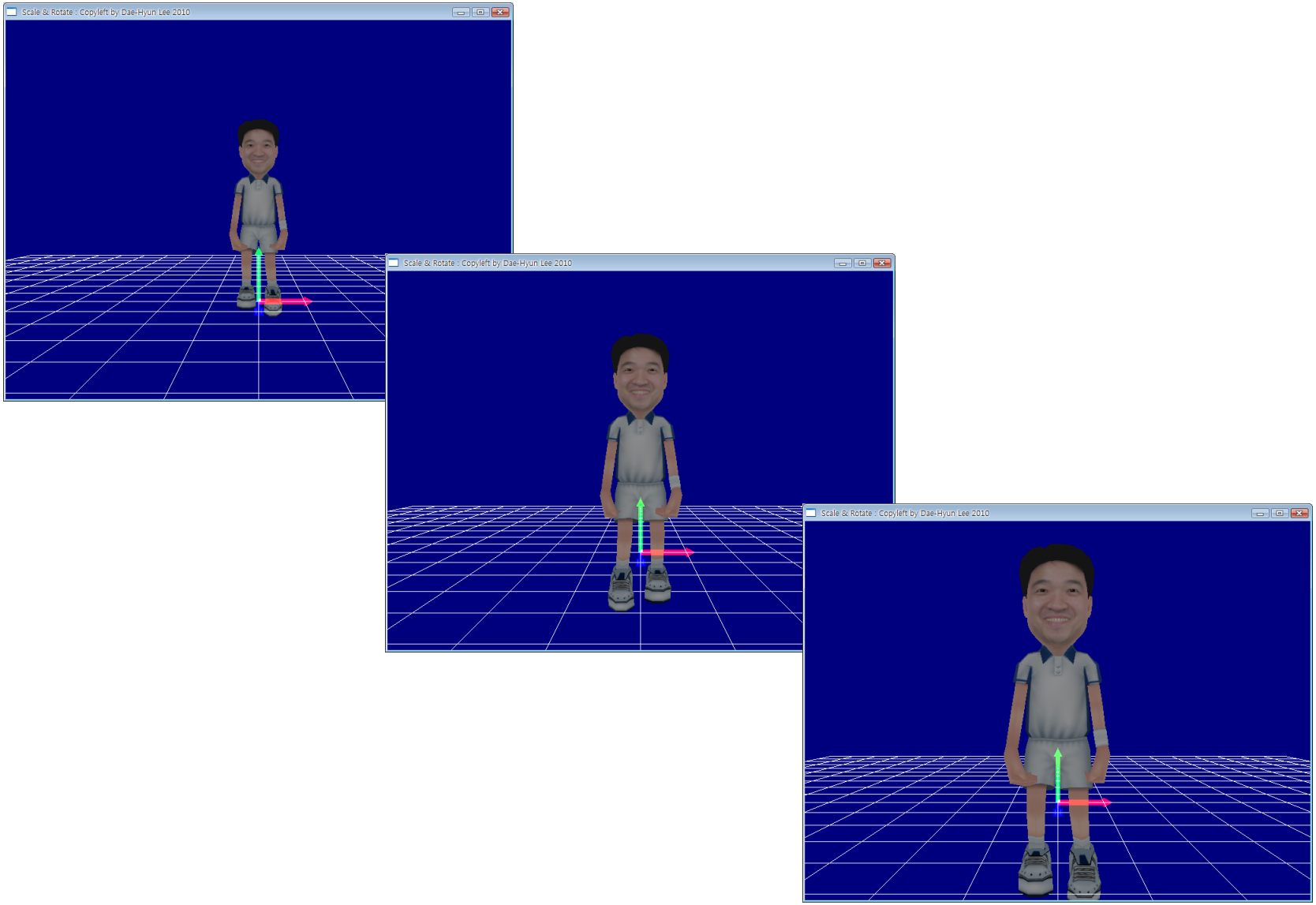
실행 결과: 애니메이션이 실행되지 않는다?



main.cpp


```
bool frameStarted(const FrameEvent &evt)
{

  mAnimationState->addTime(evt.timeSinceLastFrame);
  mProfessorNode->translate(Vector3(0, 0, 50.0f * evt.timeSinceLastFrame));

  return true;
}
```

실행 결과: 캐릭터가 카메라를 향해 전진함.



애니메이션 설정

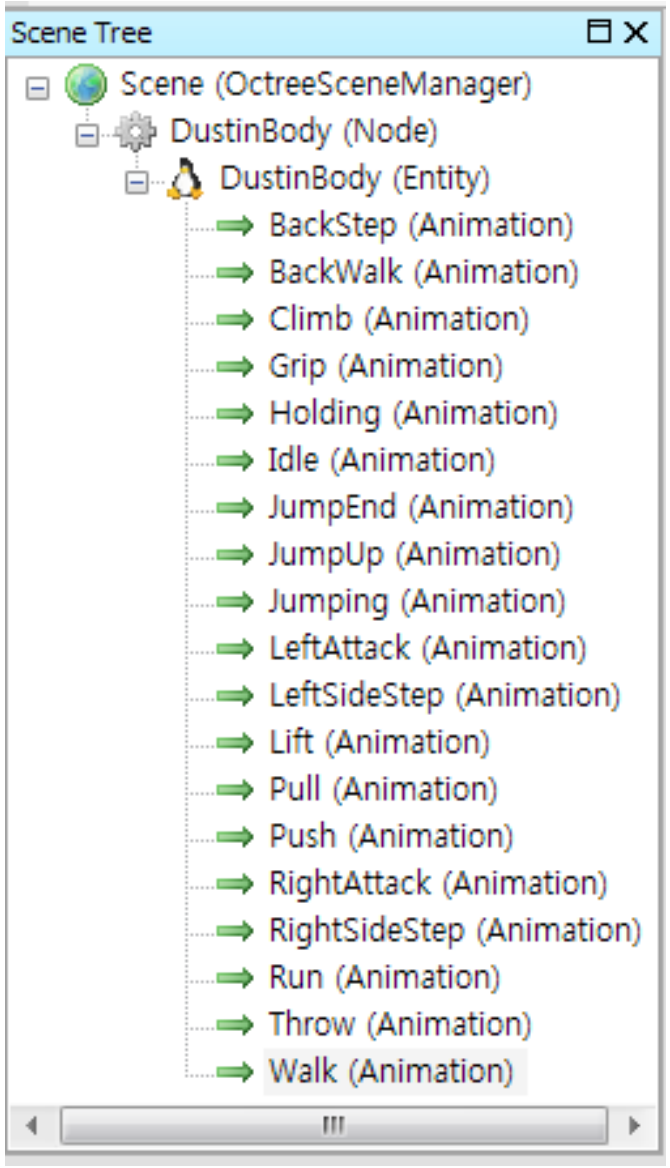
```
mProfessorEntity = mRoot->getSceneManager("main")->getEntity("Professor");

mAnimationState = mProfessorEntity->getAnimationState("Walk");


mAnimationState->setEnabled(true);

mAnimationState->setLoop(true);
```

- 애니메이션을 "Walk" 상태로 설정한다.

- "Walk" 상태의 애니메이션의 계속 반복.
- "false" 면 한번만 애니메이션을 수행.

- 실제로 "Walk" 상태의 애니메이션 동작이 일어나도록 설정한다.

Professor Character의 애니메이션 종류



애니메이션 시간 업데이트

```
bool frameStarted(const FrameEvent &evt)
{
```

- 애니메이션 루프는 시간이 진행됨에 따라, 동작한다.
- 현재 시간을 갱신하지 않으면, 시간 진행이 없는 것으로 간주하여, 애니메이션이 일어나지 않는다.
- 따라서, 매 프레임마다, 경과된 시간을 알려주어야 한다.
- addTime() 함수를 이용하여, 이전 애니메이션 시간 대비하여, 현재까지 경과된 시간을 알려줌으로써, 다음 애니메이션 동작을 표시.

```
mAnimationState->addTime(evt.timeSinceLastFrame);
```

- 직전에 수행되었던, frameStarted() 함수의 실행 완료 시점이후, 현재 frameStarted() 함수 호출이 될때까지의 경과 시간.

캐릭터 노드의 이동 - Frame Time Independent Movement

- 노드의 이동량에 직전 프레임과 현재 프레임 시간차이를 곱한다.

```
 mProfessorNode->translate(Vector3(0, 0, 50.0f * evt.timeSinceLastFrame));
… 후략 …
}
```

WalkingAroundProfessor
공간을 돌아다니는 캐릭터

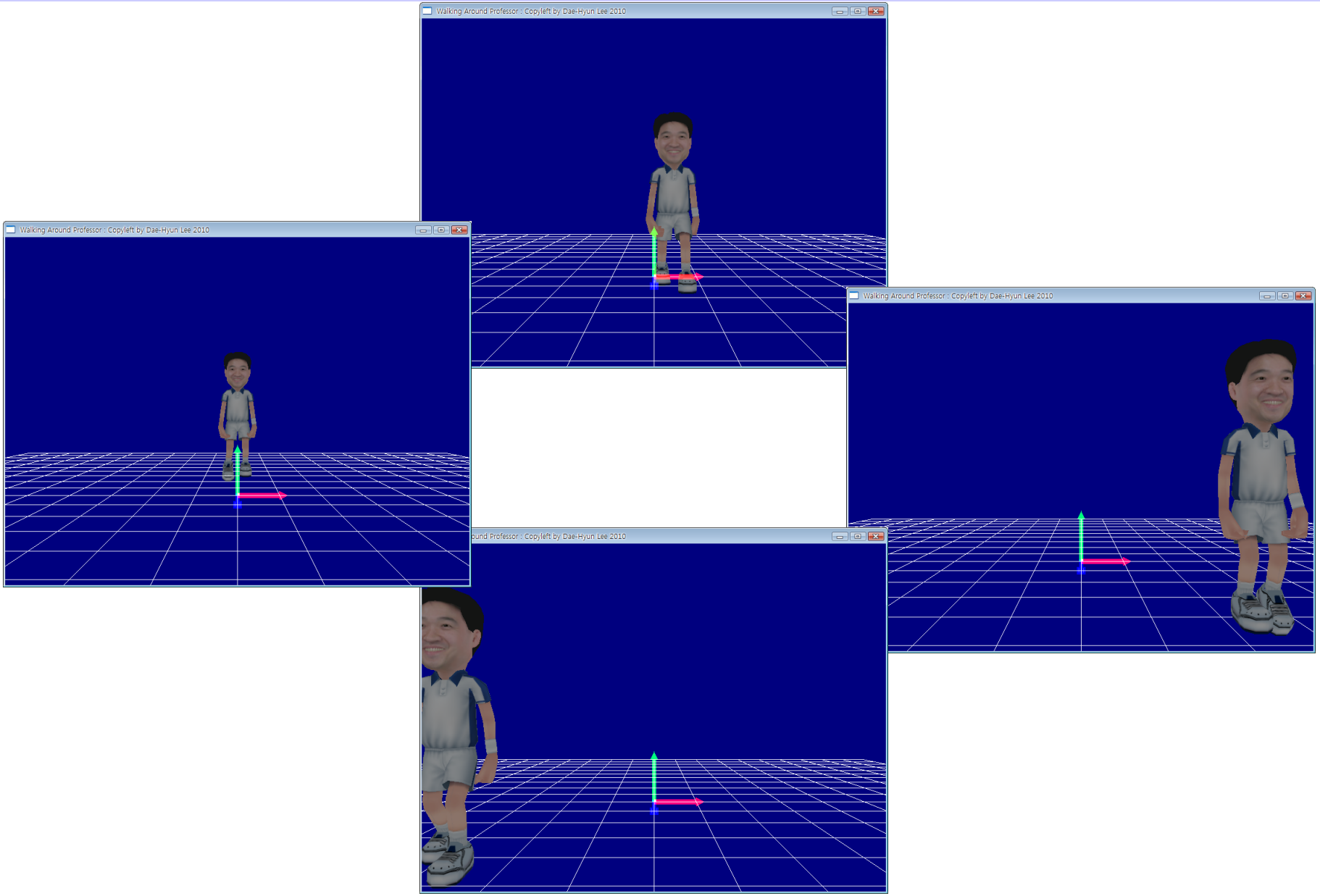
main.cpp


```
MainListener(Root* root, OIS::Keyboard *keyboard) : mKeyboard(keyboard), mRoot(root)
{

  ... 중략 ...


  mWalkSpeed = 80.0f;
  mDirection = Vector3::ZERO;

  mWalkList.push_back( Vector3( 150.0f,  0.0f, 200.0f  ) );
  mWalkList.push_back( Vector3( -150.0f,  0.0f,  200.0f  ) );
  mWalkList.push_back( Vector3( 0.0f,  0.0f,  -200.0f  ) );
  mWalkList.push_back( Ogre::Vector3::ZERO );

  ... 중략 ...

}
```


```
bool frameStarted(const FrameEvent &evt)
{
  if (Vector3::ZERO == mDirection)
  {
    if (nextLocation())
    {
      mAnimationState = mProfessorEntity->getAnimationState("Walk");
      mAnimationState->setLoop(true);
      mAnimationState->setEnabled(true);
    }
  }
  else // Vector3::ZERO != mDirection
  {
    Real move = mWalkSpeed * evt.timeSinceLastFrame;
    mDistance -= move;
    if (mDistance <= 0.0f)
    { // 목표 지점에 다 왔으면…
      mProfessorNode->setPosition( mDestination );
      mDirection = Vector3::ZERO;
      if (! nextLocation( ) )
      {
        mAnimationState->setEnabled(false);
        mAnimationState = mProfessorEntity->getAnimationState( "Idle" );
        mAnimationState->setLoop( true );
        mAnimationState->setEnabled( true );
      }
    }
    else // mDistance > 0.0f
    {
      mProfessorNode->translate( mDirection * move );
    }
  }
  ... 중략 ...
}
```


```
bool nextLocation(void)
{


  if (mWalkList.empty())  // 더이상목표지점이없으면false 리턴
    return false;
  mDestination = mWalkList.front(); // 큐의가장앞에서꺼내기
  mWalkList.pop_front(); // 가장앞포인트를제거
  mDirection = mDestination - mProfessorNode->getPosition( ); // 방향계산
  mDistance = mDirection.normalise( ); // 거리계산


  ... 중략 ...
}
```

실행 결과: 캐릭터가 공간 상의 네점 사이를 이동한다.



걷기 목표 포인트 리스트의 생성

- 캐릭터 현재 이동 방향을 나타내는 벡터.
- 이 값이 ZERO면, 정지 상태임을 나타낸다.

```
mWalkSpeed = 80.0f;
mDirection = Vector3::ZERO;

mWalkList.push_back( Vector3( 150.0f,  0.0f, 200.0f  ) );
mWalkList.push_back( Vector3( -150.0f,  0.0f,  200.0f  ) );
mWalkList.push_back( Vector3( 0.0f,  0.0f,  -200.0f  ) );
mWalkList.push_back( Ogre::Vector3::ZERO );
```

- std::deque<Vector3> mWalkList; 로 선언됨.

걷기 목표 지점의 획득

```
bool nextLocation(void)
{

  if (mWalkList.empty())  // 더 이상 목표 지점이 없으면 false 리턴
    return false;


  mDestination = mWalkList.front(); // 큐의 가장 앞에서 꺼내기


  mWalkList.pop_front(); // 가장 앞 포인트를 제거


  mDirection = mDestination - mProfessorNode->getPosition( ); // 방향 계산


  mDistance = mDirection.normalise( ); // 거리 계산


  return true;
}
```

캐릭터 이동 구현

- 정지 상태이면…

- 다음 이동 목표 지점을 가져온다.

- 이동 목표 지점이 있으면, 애니메이션 상태를 걷기(“Walk”) 상태로 설정한다.

```
if (Vector3::ZERO == mDirection)
{

  if (nextLocation()) {

    mAnimationState = mProfessorEntity->getAnimationState("Walk");
    mAnimationState->setLoop(true);
    mAnimationState->setEnabled(true);
  }
```

```
else
{
  Real move = mWalkSpeed * evt.timeSinceLastFrame; // 이동량 계산
  mDistance -= move; // 남은 거리 계산
  if (mDistance <= 0.0f)
  { // 목표 지점에 다 왔으면…
    mProfessorNode->setPosition( mDestination ); // 목표 지점에 캐릭터를 위치
    mDirection = Vector3::ZERO; // 정지 상태로 들어간다.
    if (! nextLocation( ) )
    {
      mAnimationState->setEnabled(false); // "Walk" 애니메이션 정지!!
      mAnimationState = mProfessorEntity->getAnimationState( "Idle" );
      mAnimationState->setLoop( true );
      mAnimationState->setEnabled( true );
    }
  }
  else
  {
    mProfessorNode->translate( mDirection * move );
  }
}
```

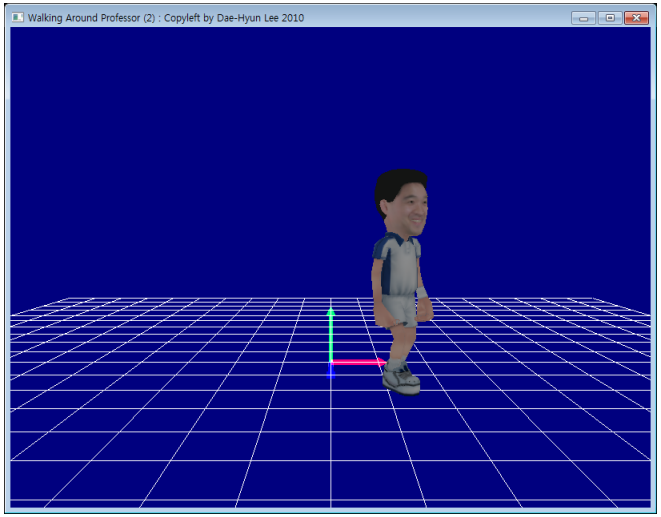
- 캐릭터가 이동 중이면….
- 다음 이동 목표 지점이 없으면, 애니메이션 상태를 "Idle" 상태로 설정한다.
- 캐릭터를 mDirection 방향으로, move 만큼 이동시킨다.

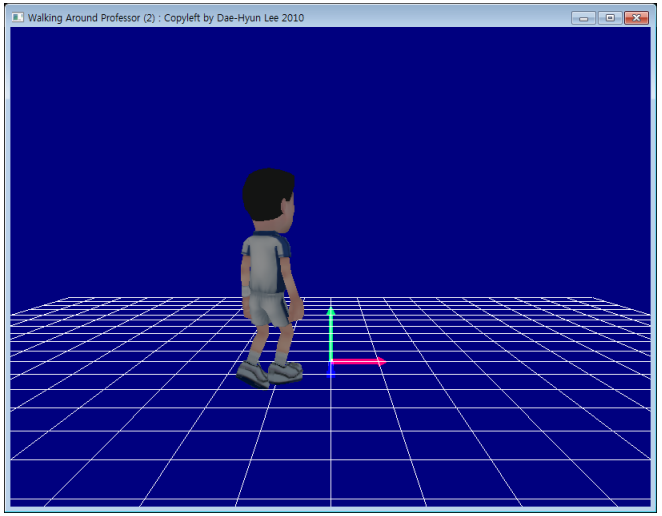
WalkingAroundProfessorCorrectFacing
걷는 방향으로 바라보기

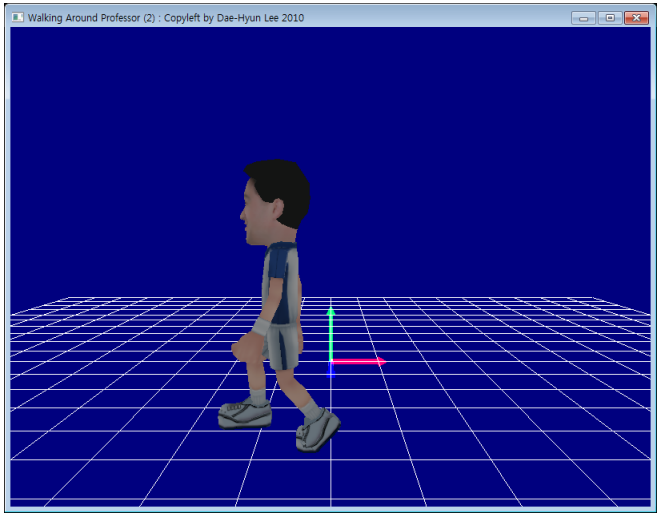
class ProfessorController


```
bool nextLocation(void)
{
  if (mWalkList.empty())  // 더 이상 목표 지점이 없으면 false 리턴
    return false;

  mDestination = mWalkList.front(); // 큐의 가장 앞에서 꺼내기
  mWalkList.pop_front(); // 가장 앞 포인트를 제거
  mDirection = mDestination - mProfessorNode->getPosition(); // 방향 계산
  mDistance = mDirection.normalise(); // 거리 계산

  Quaternion quat = Vector3(Vector3::UNIT_Z).getRotationTo(mDirection);
  mProfessorNode->setOrientation(quat);

  return true;
}
```

실행 결과 - 걷는 방향으로 바라보면서 이동

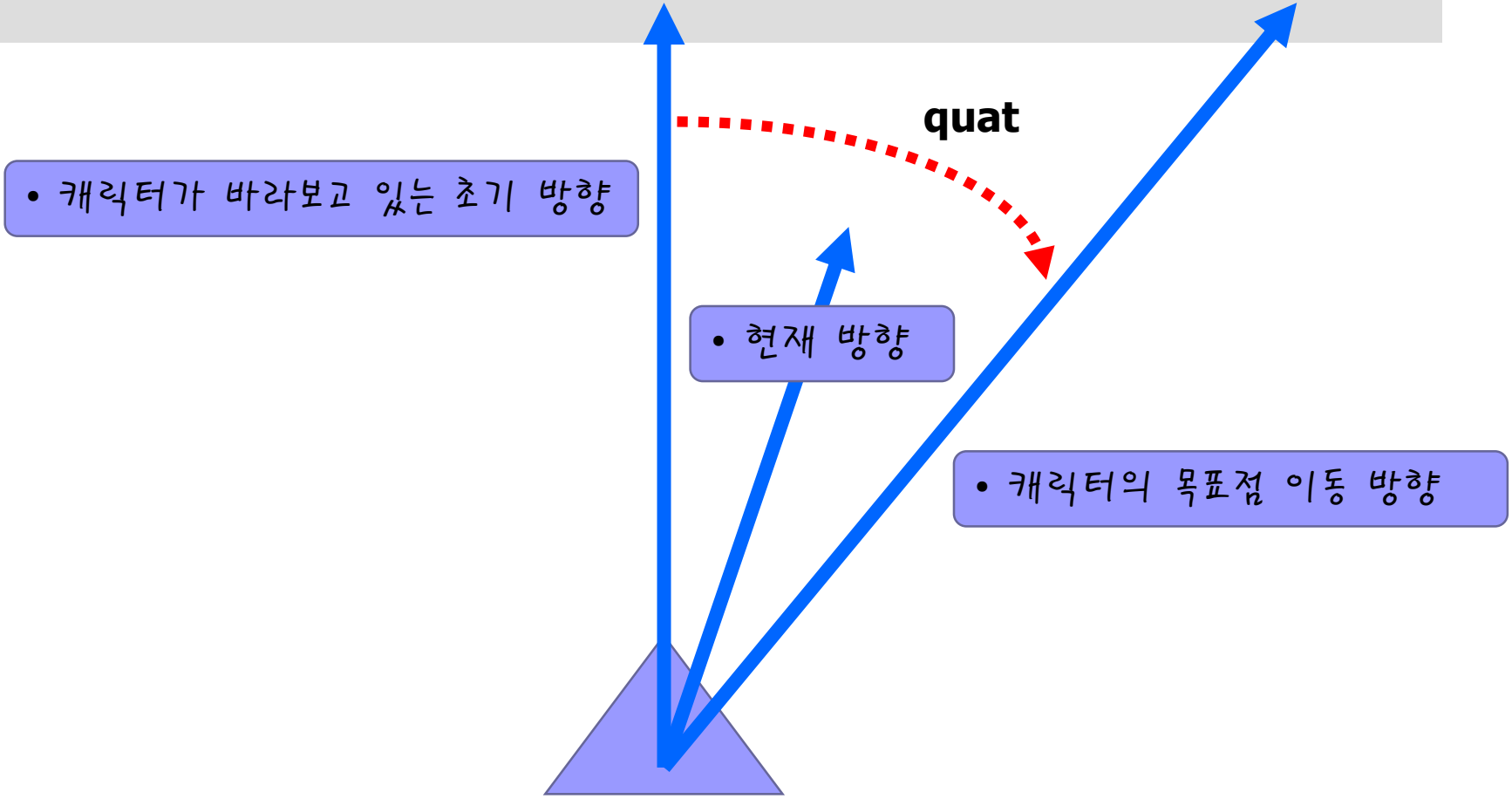






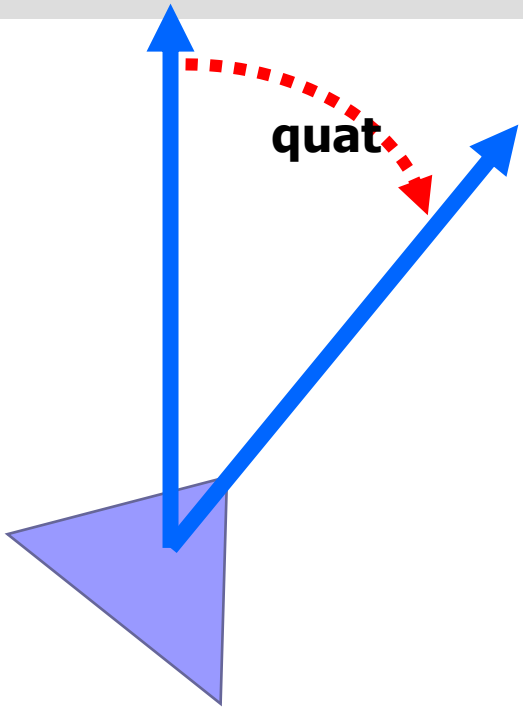
목표 방향으로 바라보려면, 몇 사원수값만큼 회전해야 하는가?

```
quat = Vector3(Vector3::UNIT_Z).getRotationTo(mDirection);
```

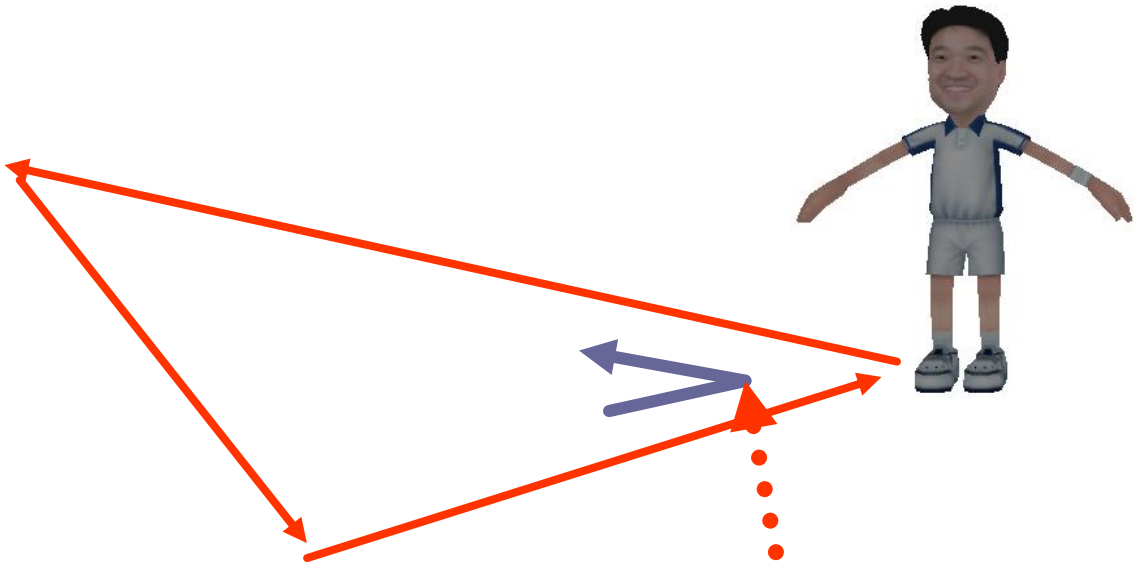


캐릭터 노드를 구해진 사원수만큼 회전

```
mProfessorNode->setOrientation(quat);
```

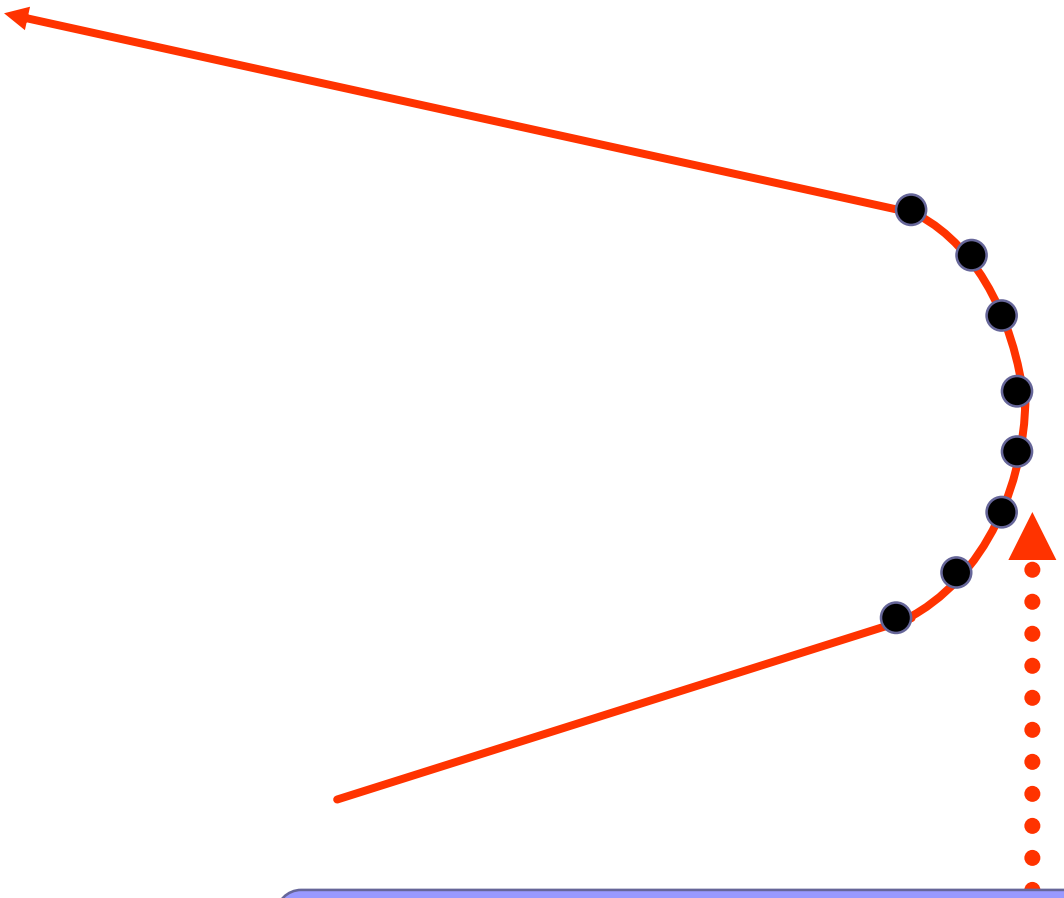


방향 전환시 문제점은?



- 캐릭터 방향 전환시 순간적으로 방향이 바뀜으로써, 어색하다.

부드러운 방향 전환을 하려면?



- 방향 전환 점에, 여러 개의 이동점을 둠으로써 부드럽게 회전시킨다.
- 일일이 점을 지정해야 된다는 점이 문제…

Slerp() 함수를 사용한 부드러운 방향 전환 방법

- **구면보간(Spherical Linear Interpolation)**
  - 벡터의 방향 전환을 부드럽게 하기 위해 필요한 여러 개의 사원수값을 생성.
  - Quaternion Ogre::Quaternion::Slerp (Real fT, const Quaternion & rkP, const Quaternion & rkQ, bool shortestPath = false)
    - fT: 회전 비율(0부터 1까지의 값으로 지정)
    - rkP: 시작 사원수
    - rkQ: 종료 사원수
    - shortestPath: 가장 짧은 경로를 통해서 회전함.

    - 함수의 리턴값은 시작 사원수로부터 종료 사원수까지 회전을 할 때, ft로 주어진 회전 비율까지로 회전을 하고자 할 때, 필요한 사원수가 된다.

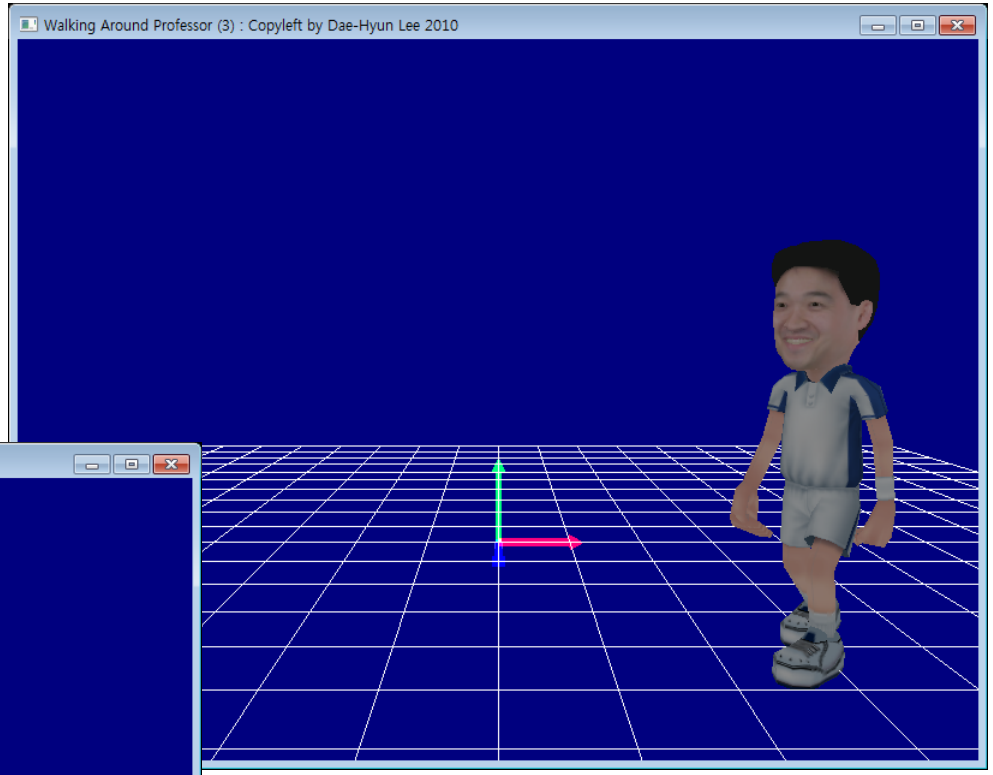
WalkingAroundProfessorSmoothRotation
부드러운 방향 전환

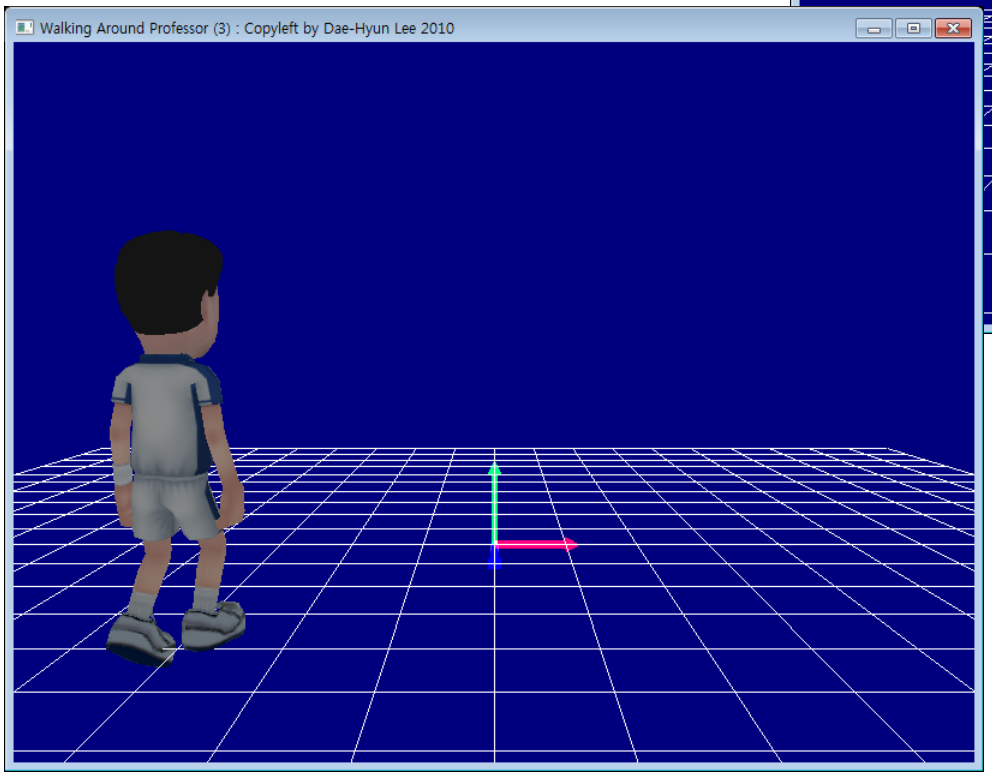
class ProfessorController


```
bool frameStarted(const FrameEvent &evt)
{
  ... 중략 ...

  else if (mRotating)
  {
    static const float ROTATION_TIME = 0.3f;
    mRotatingTime = (mRotatingTime > ROTATION_TIME) ? ROTATION_TIME : mRotatingTime;
    Quaternion delta = Quaternion::Slerp(mRotatingTime / ROTATION_TIME, mSrcQuat, mDes
   tQuat, true);
    mProfessorNode->setOrientation(delta);
    if (mRotatingTime >= ROTATION_TIME)
    {
      mRotatingTime = 0.0f;
      mRotating = false;
      mProfessorNode->setOrientation(mDestQuat);
    }
    else
      mRotationTime += evt.timeSinceLastFrame;
  }

 ... 중략 ...
}
```


```
bool nextLocation(void)
{
  ... 중략 ...


  mSrcQuat = mProfessorNode->getOrientation();
  mDestQuat = Vector3(Vector3::UNIT_Z).getRotationTo(mDirection);
  mRotating = true;
  mRotatingTime = 0.0f;


  return true;
}
```

실행 결과: 코너에서 부드러운 회전





- 현재 구면보간을 이용하여 회전 중인가?

```
bool mRotating;
```

- 시작 사원수와 종료 사원수.

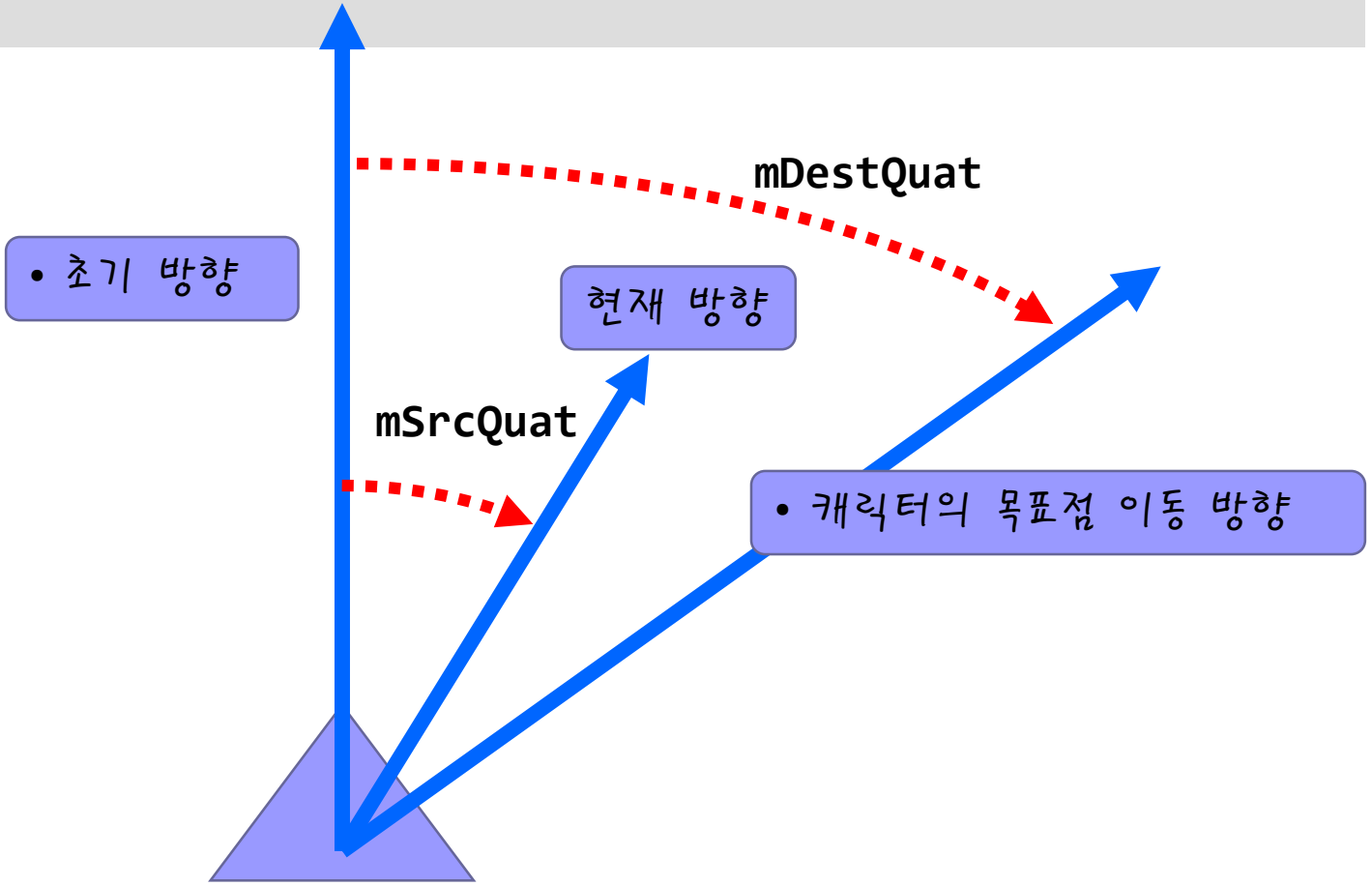
```
Quaternion mSrcQuat, mDestQuat;
```

```
float mRotatingTime;
```

- 회전 경과 시간

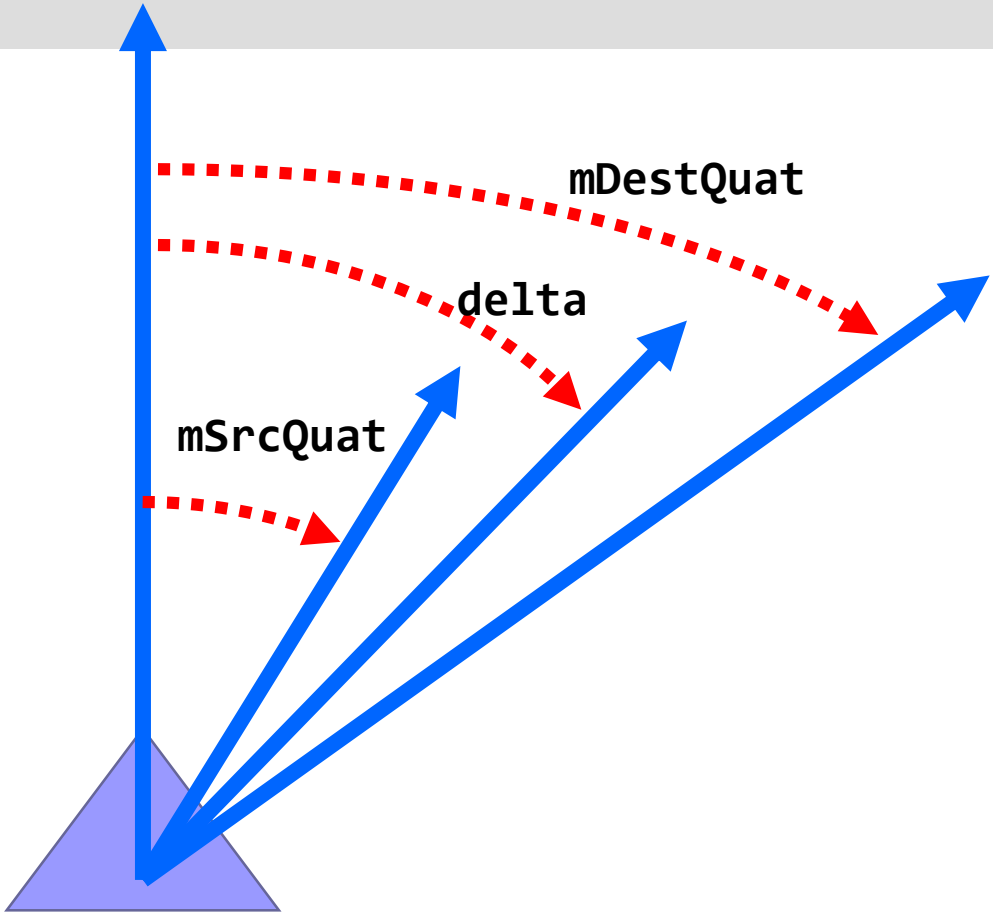
현재 회전값 및 목표 회전값 계산

```
mSrcQuat = mProfessorNode->getOrientation();
mDestQuat = Vector3(Vector3::UNIT_Z).getRotationTo(mDirection);
```



회전값 delta 계산

```
delta = Quaternion::Slerp(mRotatingTime / ROTATION_TIME,
                          mSrcQuat, mDestQuat, true);
```



시간 경과 비율 만큼 회전

```
else if (mRotating)
{
  static const float ROTATION_TIME = 0.3f;


  mRotatingTime = (mRotatingTime > ROTATION_TIME) ? ROTATION_TIME : mRotatingTime;

  Quaternion delta =
        Quaternion::Slerp(mRotatingTime / ROTATION_TIME, mSrcQuat, mDestQuat, true);

  mProfessorNode->setOrientation(delta);

  if (mRotatingTime >= ROTATION_TIME)
  {
    mRotatingTime = 0.0f;
    mRotating = false;
    mProfessorNode->setOrientation(mDestQuat);
  }
  else
     mRotatingTime += evt.timeSinceLastFrame;
}
```

회전에 허용되는 시간

경과시간에 따른 회전 비율 계산

(0 ~ 1)

학습 정리

■ 애니메이션 방법

- □ 키프레임 애니메이션(Key-frame animation)
- □ 모션 캡쳐(Motion capture)
- □ 절차적 애니메이션(Procedural Animation)

■ 키프레임 애니메이션

- □ 애니메이션 동작 중에서 특징이 되는 프레임만을 제작.
- □ 중간 프레임은 보간을 통해서 생성.

■ 애니메이션 구현

- □ getAnimationState(), setLoop(), setEnabled()
- □ 애니메이션 실행을 위해서는 반드시 애니메이션 시간을 증가시켜야 함 - addTime()

■ 캐릭터 이동 속도 조절

- □ 캐릭터 이동량 = 초당 이동량 * (직전 프레임과 현재 프레임간의 시간차이)

■ Slerp 구면 보간

- □ 두 개의 회전값 사이를 부드럽게 보간하는 방법.
- □ 캐릭터의 방향 전환에 적용됨.