게임엔진

제6강 게임루프(Game Loop)

학습 안내

■ 학습 목표

- □ 컴퓨터의 시간 측정 방법을 이해하고, 이를 이용하여 게임 속도를 제어할 수 있다.
- □ 프레임 리스너를 이용하여 게임 루프를 구현하는 방법을 이해한다.
- □ 오우거 엔진의 키입력 처리 방식을 이해한다.

■ 학습 내용

- □ 컴퓨터의 시간 측정
- □ 프레임 레이트, 프레임 시간
- □ 타임 라인
- □ 게임 루프의 다양한 구성 방법
- □ 프레임 리스너의 개념
- □ 프레임 리스너를 이용한 게임 캐릭터의 이동
- □ 캐릭터의 이동 속도 조절
- □ OIS 입력 시스템을 이용한 키보드 입력의 처리

시간 측정

- **게임은 시간에 따른 가상 세계의 시뮬레이션**

- **따라서, 정확한 시간의 측정이 매우 중요함**

- **함수 time()**
  - 표준 C 라이브러리 함수
  - 1970년 1월 1일을 기준으로 현재까지 경과 시간을 “초” 단위로 알려줌
  - 1초는 컴퓨터 게임에서 너무나 긴 시간

- **모든 CPU에 포함된 정밀타이머(high-resolution timer)**
  - CPU 가 켜진 시점으로부터 경과한 클럭틱(tick)의 개수를 저장
  - 3GHz → 초당 3억번 클럭틱이 진행
  - 타이머의 정밀도: 1/30 억 = 0.333ns

- **정밀타이머 이용 윈도우 API 함수**
  - QueryPerformanceCounter() - 경과된 클럭 수
  - QueryPerformanceFrequency() - 현재 CPU의 성능 주파수(ex. 3GHz)

시간 단위와 클록 변수

- **시간 단위?**
  - 초? 밀리초? 또는 하드웨어주기(cycle)

- **시간값을 저장할 단위?**
  - 64비트 정수?
  - 32비트 정수?
  - 32비트 부동 소수?

- **결정의 기준**
  - 필요한 정확도
  - 표현해야 할 절대값의 범위

- **64비트 정수 클럭**
  - 정확도(0.333ns)와 넓은 범위(3Ghz의 경우, 195년에 한번 겹침)

32비트 정수 클럭

- 높은 정밀도로 비교적 짧은 경과시간을 측정할 필요가 있을 경우..

```
// Grab a time snapshot.
U64 tBegin = readHiResTimer();

// This is the block of code whose performance we wish
// to measure.
doSomething();
doSomethingElse();
nowReallyDoSomething();

// Measure the duration.
U64 tEnd = readHiResTimer();
U32 dtCycles = static_cast<U32>(tEnd - tBegin);

// Now use or cache the value of dtCycles…
```

32비트 부동 소수 클럭

- 초단위로 짧은 경과 시간을 저장할 때 많이 사용…
- 정수부, 소수부의 변화에 주의가 필요. 값이 커지면, 소수부에 대한 정확도가 줄어듬.

```
// Start off assuming an ideal frame time (30 FPS).
F32 dtSeconds = 1.0f / 30.0f;

// Prime the pump by reading the current time.
U64 tBegin = readHiResTimer();

while (true) // main game loop
{
    runOneIterationOfGameLoop(dtSeconds);

    // Read the current time again, and calculate the
    // delta.
    U64 tEnd = readHiResTimer();
    dtSeconds = (F32)(tEnd - tBegin) * (F32)getHiResTimerFrequency();

    // Use tEnd as the new tBegin for next frame.
    tBegin = tEnd;
}
```

디버깅 중단에 따른 시간의 처리

- 디버깅을 위해 중단점에서 중단하고, 각종 값을 확인하고 ,다시 프로그램을 실행하는 경우가 많음.
- clock은 계속 진행하기 때문에, delta time이 매우 커지게 됨 → 이 후 프로그램 실행이 오동작할 수 있음.

```
while (true) // main game loop
{
    updateSubsystemA(dt);
    renderScene();
    swapBuffers();
    U64 tEnd = readHiResTimer();
    dt = (F32)(tEnd – tBegin) / (F32)
    getHiResTimerFrequency();

    // If dt is too large, we must have resumed from a
    // break point -- frame-lock to the target rate this
    // frame.
    if (dt > 1.0f/10.0f)
    {
        dt = 1.0f/30.0f;
    }
    tBegin = tEnd;
}
```

Clock 클래스 구현

- **Clock 클래스 기능**
  - getTimeCycles()
    - clock 객체 생성 시점을 기준으로 한 경과 시간을 획득(cycle 단위)
  - calcDeltaSeconds(const Clock& other)
    - 다른 클럭과 이 클럭사이의 절대 시간 차이를 구함.
  - update(F32 dtRealSeconds)
    - 클럭을 dtRealSeconds 로 지정된 시간만큼 진행시킴.
    - 자체적으로 시간이 가는 것이 아니고, 시간을 진행시켜주어야 함.
  - singleStep()
    - 1/30초 만큼 클럭을 진행시킴.
- **여러 개의 clock 객체 인스턴스를 생성해서 운영할 수 있음.**
  - 하나는 “실제시간” 표현에 사용, 다른 하나는 “게임 시간”에 활용, ……

프레임레이트(Frame Rate)와 시간 델타(Time Delta)

프레임

- 특정 순간에 화면에 그려지는 하나의 그림

프레임 레이트(Frame Rate)

- 3D 화면을 연속적으로 얼마나 빨리 보여주는가?
- 단위: Hz, FPS(Frame Per Sec) - 초당 몇 개의 프레임을 보여주는가?
- 영상물: 24 FPS
- 북미, 일본의 게임물: 30FPS, 60FPS
- 유럽: 50FPS

시간 델타(Time Delta)

- 두 프레임 사이에 경과한 시간
- 한장의 프레임을 그리는 데 걸리는 시간
- 프레임 시간(Frame Time)
- 씬의 구성요소 밀도에 따라 시간이 달라지는 것이 문제임.

$$\Delta t = \frac{1}{FPS}$$

time delta가 중요한 이유

- **객체 위치 계산의 핵심 요소**
  - x : 객체의 위치
  - v : 객체의 속도(등속 운동 가정)

$$x_{\text{다음프레임}} = x_{\text{현재프레임}} + v\Delta t$$

초창기의 CPU 종속적 게임

- delta time 개념이 없음.
- 그냥 물체의 움직임을 pixel 값의 변화로 표시
- 문제점은?
  - CPU 성능에 따라, 물체의 움직이는 속도가 달라짐.
  - single player 게임에서는 문제가 아닐수도…

delta time의 측정

- **기본적인 측정 방법**
  - 프레임 그리기 전, 후의 CPU 정밀 타이머의 값의 차이를 측정하면 됨.

- **문제점은?**
  - 앞선 프레임의 delta time을 다음 프레임에서 객체의 위치 계산에 사용함. 미래의 delta time을 예측해서 사용하는 것임.
  - 따라서, 두개의 delta time의 차이가 난다면, 오차가 발생할 수 밖에 없음.
  - 이동평균을 이용하여 계산하는 방법도 괜찮음.

$x_1 = x_0 + v\Delta t_0$

$x_1 = x_0 + v\Delta t_1$

Frame #1

$x_2 = x_1 + v\Delta t_1$

$x_2 = x_1 + v\Delta t_2$

Frame #2

Δt_1

Δt_2

프레임 레이트의 조절

- **delta time을 계산, 예측하지 말고, 아예 고정!!(예를 들어, 30FPS=33.3333ms)**
  - 측정된 시간이 목표시간 보다 짧으면?
    - 쉰다..
  - 측정된 시간이 목표시간 보다 길면?
    - 프레임을 포기한다. 프레임을 그리지 않는다. Frame Skip→뚝뚝 끊겨보이게 됨.
  - 평균적인 frame rate가 목표 frame rate가 비슷할 경우에 제대로 작동함.

- **프레임 레이트를 고정했을 경우의 장점**
  - 물리 엔진은 일정 간격으로 업데이트를 해야 안정성있는 최적의 성능을 발휘.
  - 화면 티어링(tearing)의 방지
  - 녹화 및 재생 기능의 안정성 - 디버깅의 주요 도구
    - 게임 플레이 도중 발생하는 모든 이벤트의 시각을 기록하게 됨.
    - 프레임 레이트가 일정하지 않으면 정확한 순서대로 진행되지 않을 수 있음.

가상 타임 라인(Abstract Timeline)

- **시스템에서 사용되는 여러 개의 시간축**

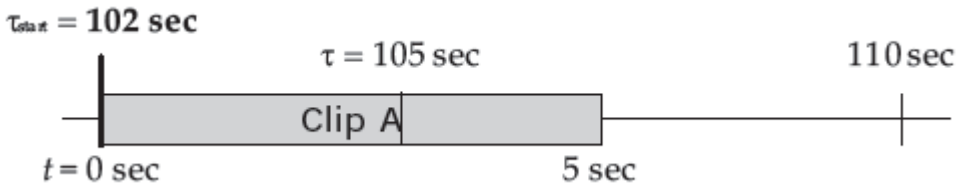
- **로컬 타임 라인**
  - 특정 시스템, 또는 객체들이 갖는 타임라인
  - 예)애니메이션 클립 타임라인, 오디오 클립 타임라인

- **실제 시간 타임 라인**
  - 기준 타임 라인

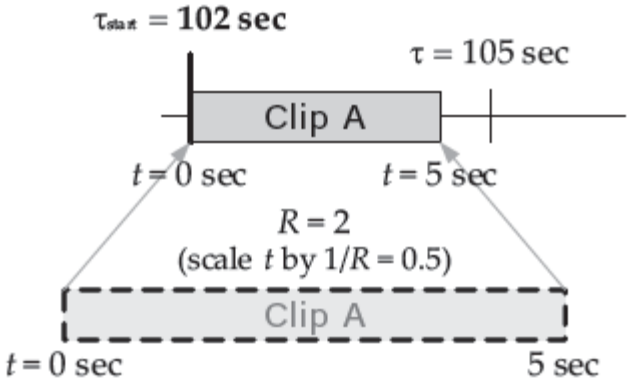
- **게임 시간 타임 라인**
  - 실제의 시간(실시간 타임라인)은 계속 진행되지만, 게임 시간(게임 타임 라인)은 중간에 멈출 수도 있고, 느리게 움직일 수도 있음.
  - 게임 시간을 멈추든지 또는 느리게 진행시키면서, 렌더링 엔진과 카메라는 다른 타임라인을 이용해 정상적으로 움직이면 효과적인 디버깅이 가능해짐.

타임라인 매핑

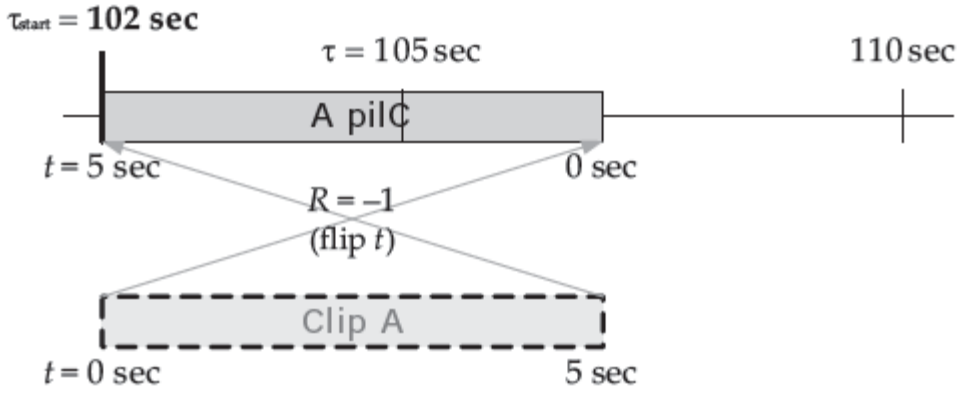
- 애니메이션을 다양한 방법으로 수행 가능



일반적인 애니메이션



2배 빠른 애니메이션



거꾸로 재생 애니메이션

게임 루프

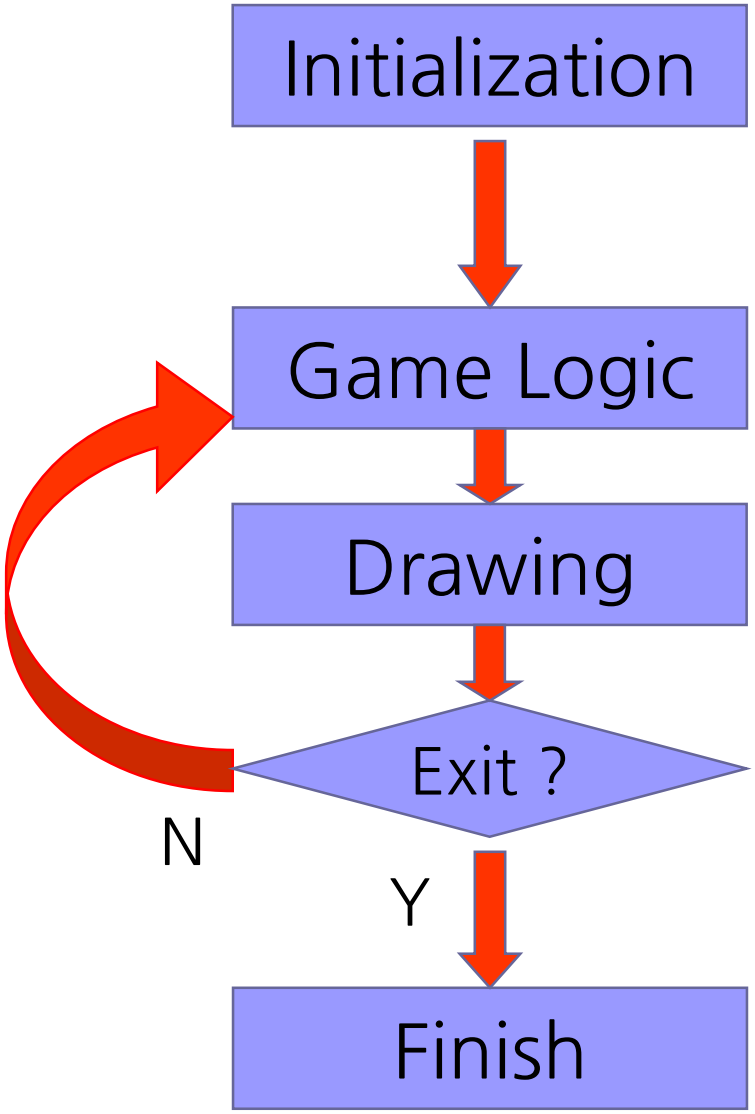
- **게임의 하부 시스템**
  - 장치 I/O, 렌더링, 애니메이션, 충돌 감지 및 처리, 강체 물리 시뮬레이션, 멀티플레이어 네트워크

- **하부 시스템들은 주기적인 갱신(update)이 필요**
  - 시간의 진행에 따른 하부 시스템들의 변화를 반영해야 함.
  - 시스템 별로 갱신 주기가 다름.
  - 애니메이션: 30,60Hz
  - 물리시뮬레이션: 120Hz
  - AI : 초당 한두번 정도?

퐁(Pong)의 게임 루프

```
void main() // Pong
{
     initGame();
     while (true) // game loop
     {
          readHumanInterfaceDevices();
          if (quitButtonPressed())
          {
              break; // exit the game loop
          }
          movePaddles();
          moveBall();
          collideAndBounceBall();
          if (ballImpactedSide(LEFT_PLAYER))
          {
               incremenentScore(RIGHT_PLAYER);
               resetBall();
          }
          else if (ballImpactedSide(RIGHT_PLAYER))
          {
               incrementScore(LEFT_PLAYER);
               resetBall();
          }
          renderPlayfield();
     }
}
```

일반적인 기본 게임 루프



게임 루프 구조 형태 (1)

- **윈도우 메시지 펌프형**
  - OS의 여러 다른 메시지도 처리해야 함.
  - 윈도우 메시지는 오는대로 처리하고, 더 이상 처리할 윈도우 메시지가 없을 때 게임 엔진 처리
  - 게임 윈도우 크기를 바꾸거나, 이리저리 끌고 다니면?

```
while (true)
{
    // Service any and all pending Windows messages.
    MSG msg;
    while (PeekMessage(&msg, NULL, 0, 0) > 0)
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    // No more Windows messages to process - run one
    // iteration of our "real" game loop.
    RunOneIterationOfGameLoop();
}
```

게임 루프 구조 형태 (2)

■ 이벤트 기반 업데이트

□ 이벤트 - 게임의 상태에 변화가 생기는 것(ex. 조이패드 버튼 누르기, 폭발, 적 캐릭터가 플레이어 캐릭터를 발견한 순간)

□ 이벤트 핸들링
- 이벤트가 발생하면, 연관있는 객체들에게 알려줌(Nonification)
- 객체들은 이벤트에 대한 처리(handling)를 하게 됨.

□ 게임 객체들의 상태 갱신에 주로 사용됨.

□ 객체다마 갱신 주기가 다를 경우, 이벤트 기반 업데이트 방식이 효과적임.

□ 렌더링을 이벤트 핸들링으로 처리하려면?
- 주기적으로 (예를 들어, 1/60 초마다) 타이머 이벤트를 발생하여, 렌더링 시스템에 전달함.

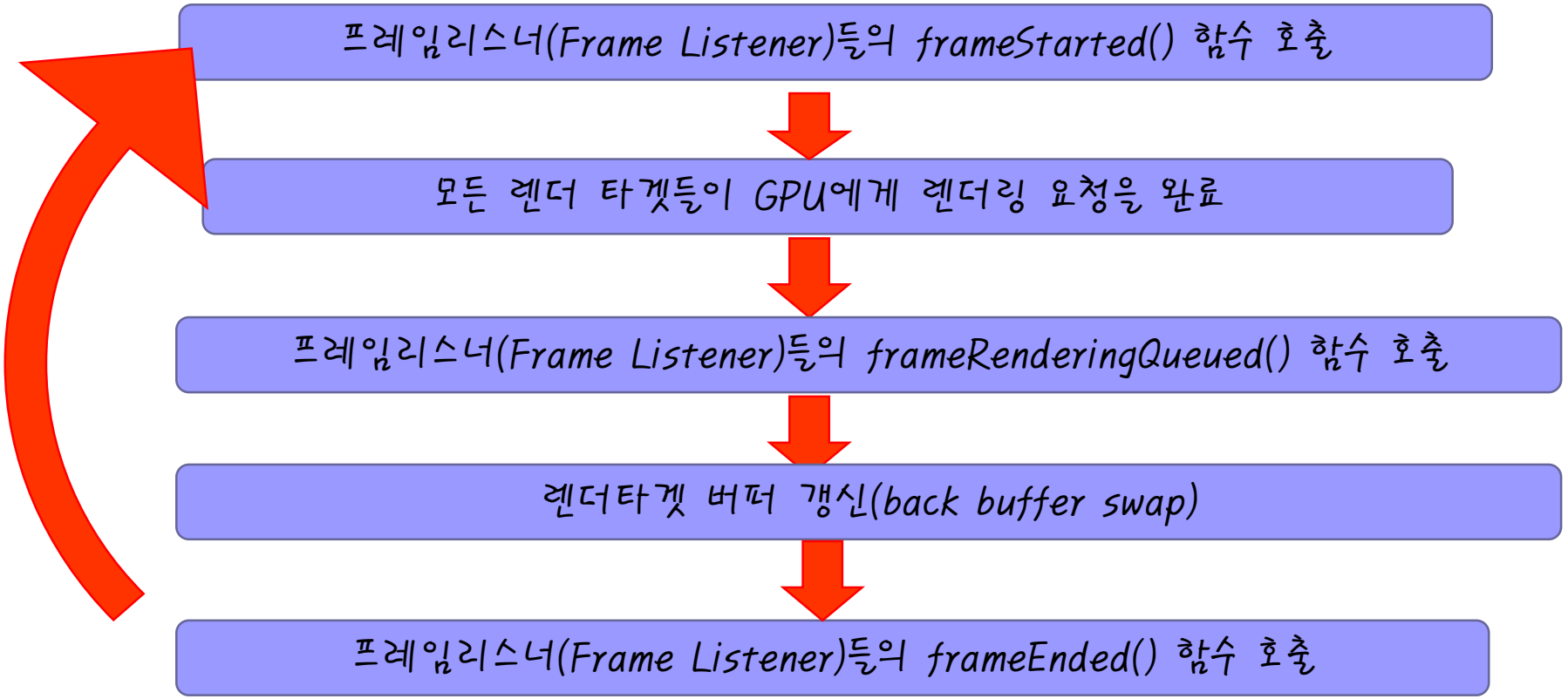
콜백 주도 프레임워크

- **프레임워크**
  - 부분적으로 구성된 “실행가능한” 애플리케이션
  - 프레임워크의 기본기능을 교체하거나, 새로운 기능을 추가할 수 있는 구조
  - 콜백 함수
    - 프레임워크가 호출하는 함수
    - 개발자가 원하는 기능을 함수로 구현해서 덧붙이는 구조.
    - 예) Ogre3D 엔진의 FrameListener 구조

오우거 엔진의 메인 렌더링 루프

- Root::startRendering() 함수에서 이루어짐.

- 메인 루프 수행 내용



- 루프의 중단
  - frameStarted(), frameEnded(), frameRenderingQueued() 에서 하나라도 false 리턴.

frameRenderingQueued()

- Ogre 1.7 부터 FrameListener에서 사용되기 시작함.
- GPU에 rendering command를 issue한 후, 그 결과가 넘어오기 전까지 기다리는 시간을 이용하여, frame 구성 로직을 CPU를 이용할 수 있음.
- 이에 따라, frameStarted(), frameEnded() 만을 이용하는 것보다 성능 측면에서 유리함.

Ogre::FrameListener 클래스

- **프레임리스너(frame listener)**
  - 장면이 화면에 렌더링되기 직전 및 직후에 호출되는 함수를 지니고 있는 객체
  - 한 프레임의 렌더링 전후에 처리해야 할 일을 프레임 리스너를 통해서 구현할 수 있다.
  - 프레임리스너 객체를 생성한 후, 반드시 프레임리스너로 등록을 시켜야 비로소 렌더링 전후에 호출이 된다.
  - 여러 개의 프레임리스너 객체가 존재할 수 있다.

Move Professor & Ninja
캐릭터 움직이기

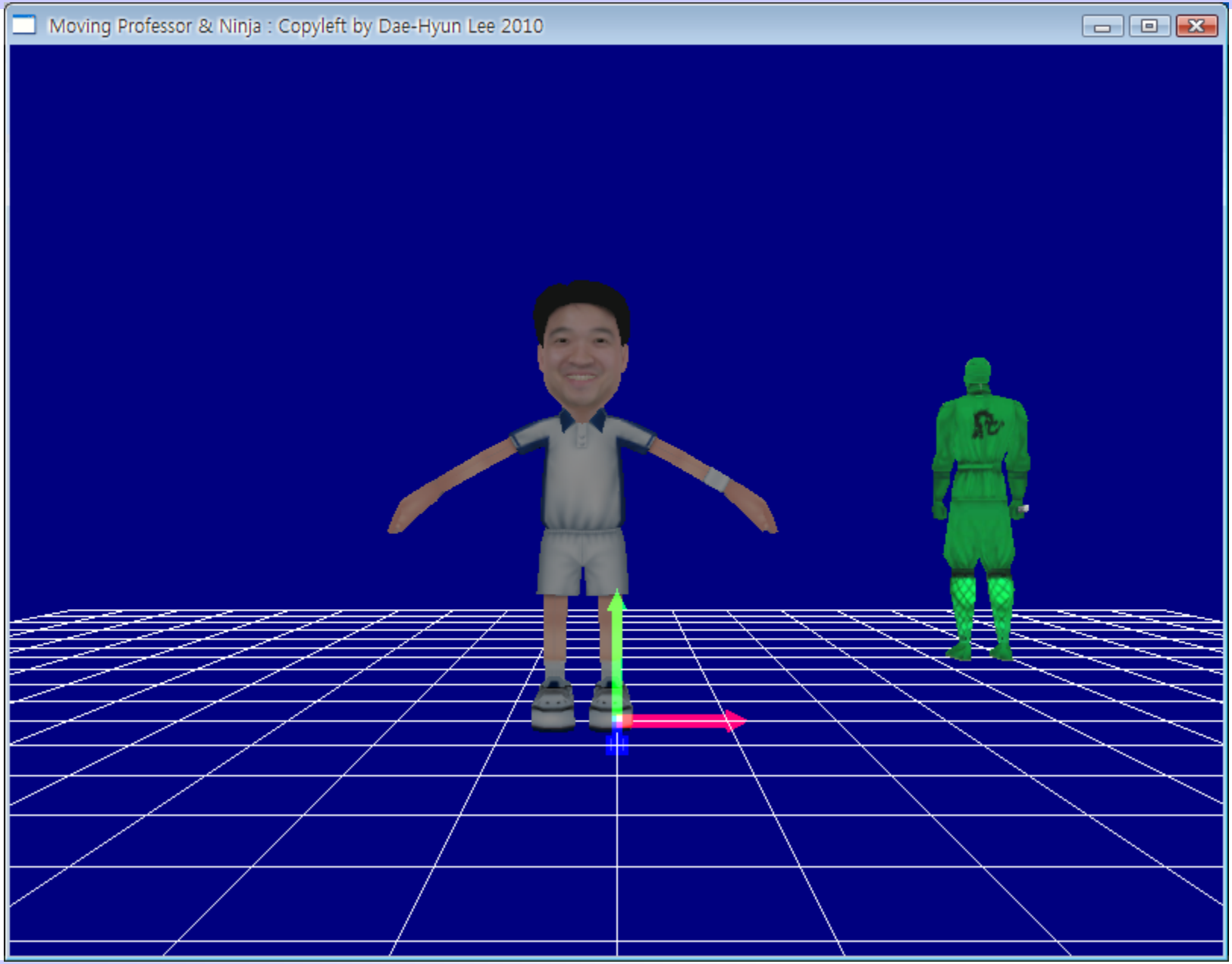
main.cpp (1)


```
class MainListener : public FrameListener
{
  … 중략 …
public:
  … 중략 …
  MainListener(Root* root, OIS::Keyboard *keyboard) : mKeyboard(keyboard), mRoot(root)
  {
    mProfessorNode = mRoot->getSceneManager("main")->getSceneNode("Professor");
    mNinjaNode = mRoot->getSceneManager("main")->getSceneNode("Ninja");
  }

  bool frameStarted(const FrameEvent &evt)
  {
    static float professorVelocity = -50.0f;
    if (mProfessorNode->getPosition().x < -200.f || mProfessorNode->getPosition().x > 200.f)
      professorVelocity *= -1;
    mProfessorNode->translate(professorVelocity * evt.timeSinceLastFrame, 0, 0);

    static float ninjaVelocity = 100.0f;
    if (mNinjaNode->getPosition().x < -400.f || mNinjaNode->getPosition().x > 400.0f)
      ninjaVelocity *= -1;
    mNinjaNode->translate(ninjaVelocity * evt.timeSinceLastFrame, 0, 0);
    return true;
  }
};
```

main.cpp (2)


```
class LectureApp {
  … 중략 …
public:
 … 중략 …
  void go(void)
  {
    … 중략 …
    mSceneMgr = mRoot->createSceneManager(ST_GENERIC, "main");
    mCamera = mSceneMgr->createCamera("main");

    … 중략 …

    mESCListener =new ESCListener(mKeyboard);
    mRoot->addFrameListener(mESCListener);

    mMainListener = new MainListener(mRoot, mKeyboard);
    mRoot->addFrameListener(mMainListener);

    … 중략 …
  }
};
```

실행 화면



캐릭터를 움직이는 프레임 리스너의 구현 (1)

```
class MainListener : public FrameListener
{


 … 중략 …
public:
  … 중략 …
  MainListener(Root* root, OIS::Keyboard *keyboard) : mKeyboard(keyboard), mRoot(root)
  {
    mProfessorNode = mRoot->getSceneManager("main")->getSceneNode("Professor");
    mNinjaNode = mRoot->getSceneManager("main")->getSceneNode("Ninja");
  }
```

• FrameListener 클래스로부터 상속.

이름을 이용하여 씬매니저 획득

이름을 이용하여 씬노드 획득

```
mSceneMgr = mRoot->createSceneManager(ST_GENERIC, "main");
mCamera = mSceneMgr->createCamera("main");
```

씬매니저 생성 시 이름을 부여

카메라 이름 부여

→ 나중에 카메라 객체를 이름으로 액세스가능

FrameListener 멤버 함수

Public Member Functions

virtual bool frameStarted (const FrameEvent &evt)
Called when a frame is about to begin rendering.

virtual bool frameRenderingQueued(const FrameEvent &evt)
Called after all render targets have had their rendering commands issued, but before the render windows have been asked to flip their buffers over

virtual bool frameEnded (const FrameEvent &evt)
Called just after a frame has been rendered.

virtual ~FrameListener ()

```
struct FrameEvent
{
  Real timeSinceLastEvent;
  Real timeSinceLastFrame;
};
```

"Frame Time"
한 frame을 생성하는데 걸린 시간

캐릭터를 움직이는 프레임 리스너의 구현 (2)

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

  static float professorVelocity = -50.0f;

  if (mProfessorNode->getPosition().x < -200.f || mProfessorNode->getPosition().x >
    200.f)
     professorVelocity *= -1;
  mProfessorNode->translate(professorVelocity * evt.timeSinceLastFrame, 0, 0);


  static float ninjaVelocity = 100.0f;

  if (mNinjaNode->getPosition().x < -400.f || mNinjaNode->getPosition().x > 400.0f)
    ninjaVelocity *= -1;
  mNinjaNode->translate(ninjaVelocity * evt.timeSinceLastFrame, 0, 0);

  return true;
}
```

초기속도 : 초당 50 cm, -x 방향

범위를 벗어나면 방향 전환

씬노드 x위치

(속도) x (시간) = 이동거리

startRendering() 함수 반복 수행 계속함.

프레임 리스너의 생성 및 등록

```
class LectureApp {
  … 중략 …
  void go(void)
  {
    … 중략 …
    mESCListener = new ESCListener(mKeyboard);
    mRoot->addFrameListener(mESCListener);

    mMainListener = new MainListener(mRoot, mKeyboard);
    mRoot->addFrameListener(mMainListener);

    mRoot->startRendering();

    … 중략 …
  };
```

- 프레임 리스너를 생성.
- 프레임 리스너를 등록.
- 렌더링 루프에서 프레임 리스너를 계속 호출하게 됨.

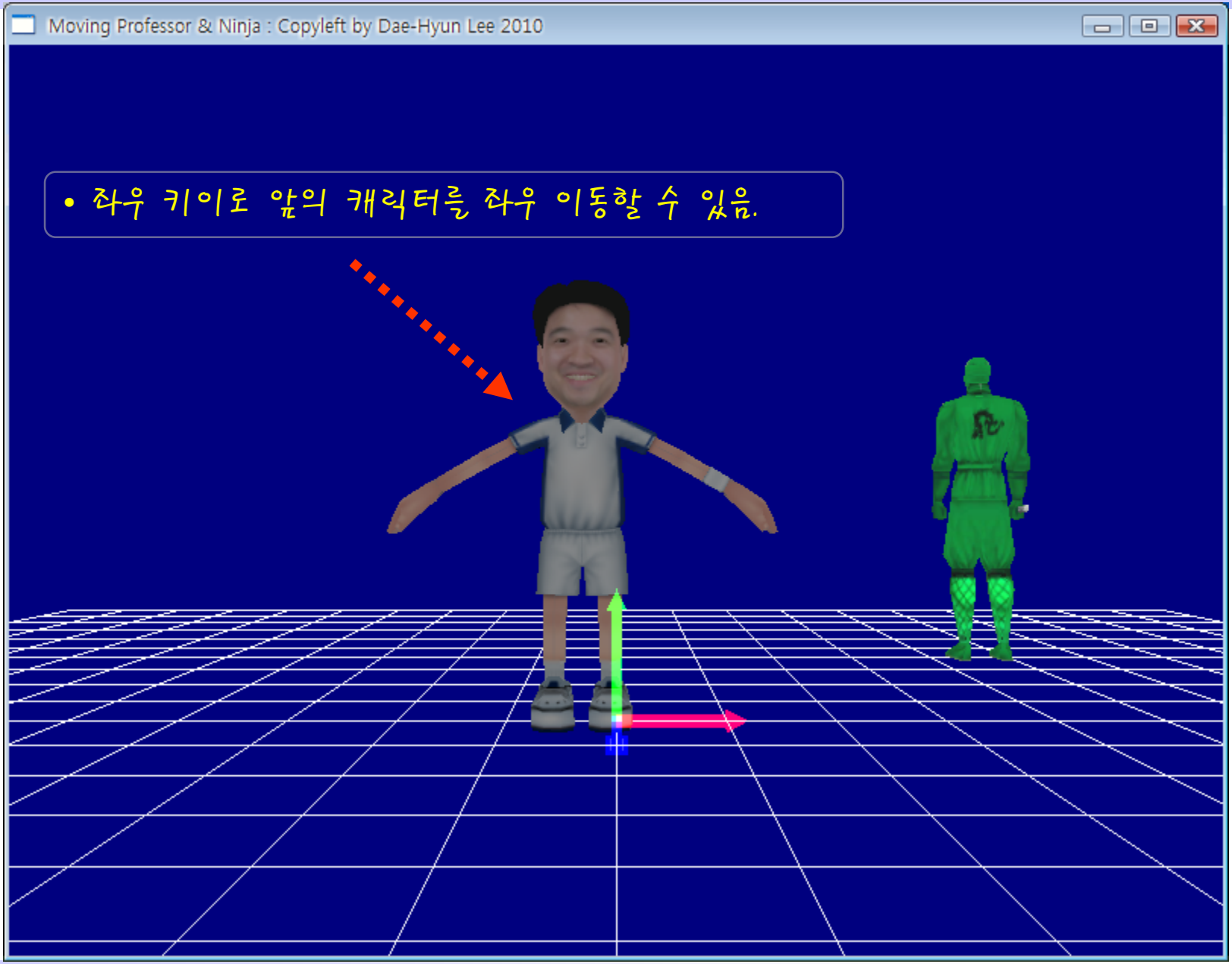
실습

Move with Key
방향키로 캐릭터 움직이기

main.cpp - professorNode 처리부 변경


```
class MainListener : public FrameListener {
  … 중략 …

  bool frameStarted(const FrameEvent &evt)
  {

 … 중략 …

    static float professorVelocity = 50.0f;

    if (mKeyboard->isKeyDown(OIS::KC_LEFT))
      mProfessorNode->translate(-professorVelocity * evt.timeSinceLastFrame, 0, 0);
    else if (mKeyboard->isKeyDown(OIS::KC_RIGHT))
      mProfessorNode->translate(professorVelocity * evt.timeSinceLastFrame, 0, 0);

 … 중략 …

  }

 … 중략 …
};
```

실행 화면


Moving Professor & Ninja : Copyleft by Dae-Hyun Lee 2010
• 좌우 키이로 앞의 캐릭터를 좌우 이동할 수 있음.

OIS InputManager의 생성 및 키보드 입력 장치 등록

```
class LectureApp {
  … 중략 …
  void go(void)
  {
   … 중략 …

    size_t hWnd = 0;
    mWindow->getCustomAttribute("WINDOW", &hWnd);

    mInputManager = OIS::InputManager::createInputSystem(hWnd);

    mKeyboard = static_cast<OIS::Keyboard*>
        (mInputManager->createInputObject(OIS::OISKeyboard, false));


   … 중략 …
   }
 … 중략 …
};
```

- 윈도우 핸들러 획득
- InputManager 생성
- 키보드 입력 장치의 생성
- 무버퍼 입력

ESC 처리 프레임 리스너의 구현

```
class ESCListener : public FrameListener {
  OIS::Keyboard *mKeyboard;

public:
  ESCListener(OIS::Keyboard *keyboard) : mKeyboard(keyboa

  bool frameStarted(const FrameEvent &evt)
  {
    mKeyboard->capture();

    return !mKeyboard->isKeyDown(OIS::KC_ESCAPE);
  }
};
```

→ 매 프레임마다 입력을 폴링함

키이가 눌린 상태인가?

```
namespace OIS
{
 //! Keyboard scan codes
 enum KeyCode
 {
  KC_UNASSIGNED  = 0x00,
  KC_ESCAPE      = 0x01,
  KC_1           = 0x02,
  KC_2           = 0x03,
  KC_3           = 0x04,
  KC_4           = 0x05,
  KC_5           = 0x06,
  KC_6           = 0x07,
  KC_7           = 0x08,
  KC_8           = 0x09,
  KC_9           = 0x0A,
  KC_0           = 0x0B,
  KC_MINUS       = 0x0C,    /
  KC_EQUALS      = 0x0D,
  KC_BACK        = 0x0E,    //
  KC_TAB         = 0x0F,
  KC_Q           = 0x10,
  KC_W           = 0x11,
  KC_E           = 0x12,
  KC_R           = 0x13,
  KC_T           = 0x14,
  KC_Y           = 0x15,
```

방향키 입력에 따른 이동 속도 방향 조정

```
bool frameStarted(const FrameEvent &evt)
 {
   … 중략 …
   static float professorVelocity = 50.0f;

   if (mKeyboard->isKeyDown(OIS::KC_LEFT))
     mProfessorNode->translate(-professorVelocity * evt.timeSinceLastFrame, 0, 0);
   else if (mKeyboard->isKeyDown(OIS::KC_RIGHT))
     mProfessorNode->translate(professorVelocity * evt.timeSinceLastFrame, 0, 0);

  … 중략 …
 }
```

왼쪽 방향키

-x 방향으로 이동

학습 정리

- **컴퓨터 시간 측정**
  - 정밀하고 정확한 측정이 필요.
- **프레임 레이트 조절**
  - 프레임 시간에 따라 객체의 상태를 갱신하는 방법
  - 강제로 일정 프레임 시간을 보장시키는 방법
- **게임 루프의 구성**
  - 초기화, 게임로직, 렌더링, 종료프로세스로 구성.
  - 게임로직과 렌더링이 무한 반복.
- **프레임 리스너**
  - 오우거 엔진의 렌더링 프로세스 전후에 실행됨.
  - 게임 로직을 구현해 넣을 수 있는 곳.
- **일정한 속도의 캐릭터 이동**
  - 프레임 타임을 이용하여, 캐릭터의 이동 속도를 설정함.
- **OIS 입력 시스템**
  - 키보드 및 마우스 입력 처리 가능.
  - 프레임 리스너 내에서 입력 처리 함으로써 사용자의 입력을 게임 로직과 연결시킬 수 있음.