



SFML로 작성해보는 프로젝트 I

학습목표

- SFML 라이브러리의 기초 사항을 살펴본다.
- SFML을 이용하여 몇 가지의 클래식 게임을 작성해본다.

학습목차

- 07.1 이번 장에서 만들어 볼 프로그램
- 07.2 SFML이란?
- 07.3 SFML의 설치
- 07.4 SFML 프로젝트 만들기
- 07.5 SFML 기초
- 07.6 Lunar Lander 게임
- 07.7 벽돌깨기
- 07.8 지뢰 찾기 게임 작성

SFML은 어떤 라이브러리인가요?

액체 지향 기법으로 설계된 멀티미디어 라이브러리입니다.

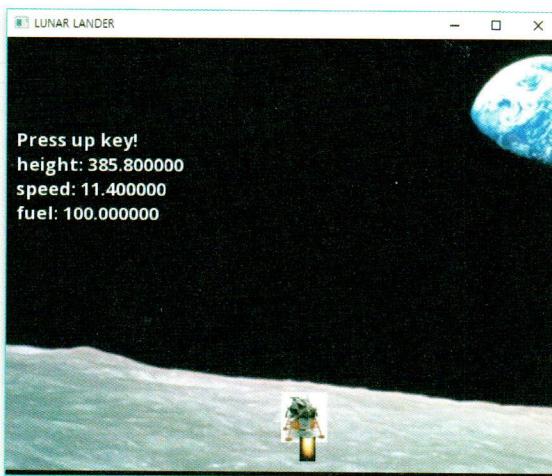


07.1

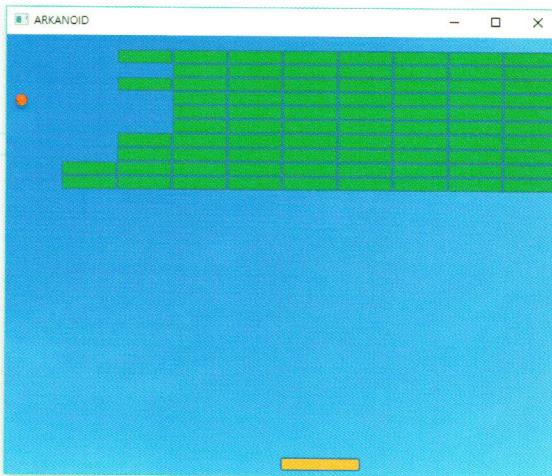
이번 장에서 만들어 볼 프로그램

이번 장에서는 멀티미디어 라이브러리인 SFML을 설치하고 SFML을 사용하여 몇 가지의 클래식 게임들을 작성해본다. 이제까지 우리가 학습한 클래스와 객체, 생성자 등의 지식만 가지고도 기초적인 게임은 작성이 가능하다. SFML을 사용하는 과정에서 우리는 클래스와 객체에 대하여 심도 있게 이해할 수 있을 것이다.

1. “루나 랜더” 게임을 작성해보자.



2. “벽돌깨기” 게임을 작성해보자.



07.2

SFML이란?

C++은 자바나 파이썬과 같은 언어와는 다르게 공식 GUI 플랫폼이 없다. 따라서 C++ 교재에서도 일반적으로 GUI 플랫폼을 소개하지 않는다. 이것은 참으로 안타까운 일이다. 실제로 C++은 빠른 실행 속도 때문에 많은 게임 제작과 GUI 프로그램 작성에 사용되는데 말이다. 마이크로소프트사는 C++로 작성된 MFC과 같은 GUI 플랫폼을 제공하지만 너무 어려운 설계나 특정한 문법으로 말미암아 C++ 교재에서 간단히 다루기에는 무리가 있다.



그림출처: <https://www.sfml-dev.org/tutorials>

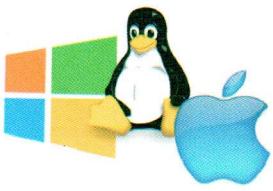
이 책에서는 SFML(Simple and Fast Multimedia Library)을 여러분들에게 소개하고자 한다. SFML은 OpenGL에 기반을 둔, 쉽고 가벼운 라이브러리이다. 여러 플랫폼에서 실행이 가능하고 게임이나 멀티미디어 응용 프로그램 제작에 사용할 수 있다. 이 라이브러리를 통하여 우리는 라이브러리가 객체 지향으로 설계되면 아주 쉽게 사용할 수 있다는 것을 알게 될 것이다. 우리는 비주얼 스튜디오에 외부 라이브러리를 연결하여 사용하는 방법도 학습할 것이다. 기말 프로젝트에도 SFML을 이용해보자.

홈페이지에 설명된 SFML의 장점은 다음과 같다.

- SFML은 PC의 애플리케이션에 간단한 멀티미디어 인터페이스를 제공하여 게임 및 멀티미디어 애플리케이션 개발을 용이하게 한다. 시스템, 윈도우, 그래픽, 오디오 및 네트워크의 5개 모듈로 구성된다.



- SFML을 사용하면 애플리케이션을 Windows, Linux, Mac OS X, Android, iOS와 같은 가장 일반적인 운영 체제에서 즉시 컴파일하여 실행할 수 있다. 많이 사용하는 OS용 SDK는 홈페이지에서 다운로드 할 수 있다.



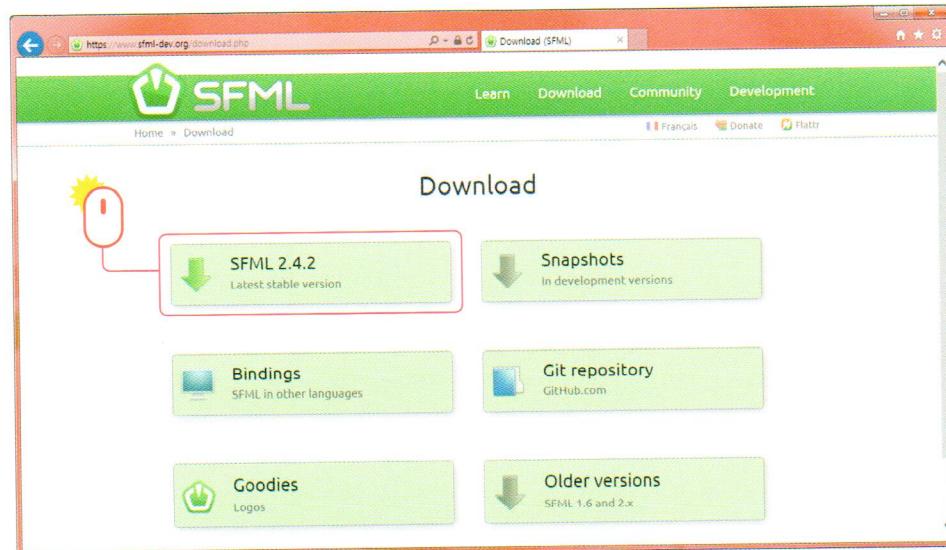
- SFML은 C++, Java, Ruby, Python, Go 등과 같은 다른 많은 언어에서도 사용할 수 있다.



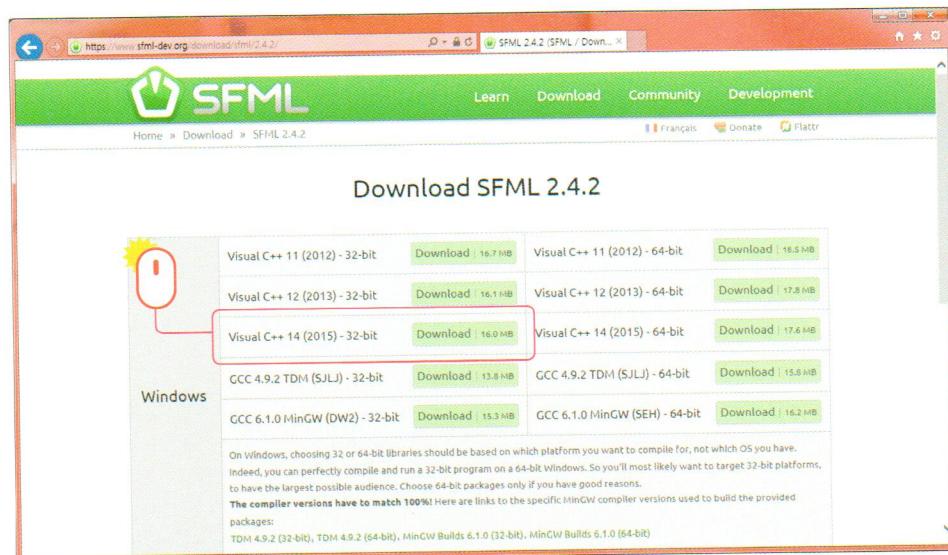
07.3

SFML의 설치

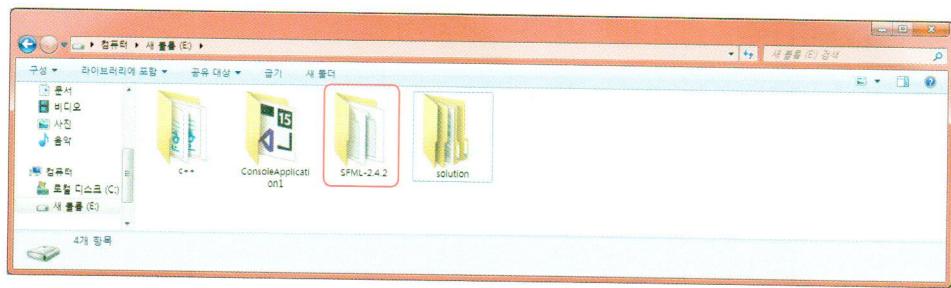
웹페이지 <http://www.sfml-dev.org/download.php>에서 [SFML 2.4.2] 버튼을 누른다. 구체적인 버전 숫자는 변경될 수 있으니 최신 버전을 다운로드하면 된다.



1. Visual C++ 14(2015) 32-bit를 선택하여 다운로드한다. 32비트와 64비트 운영체제에서 모두 실행시키려면 32비트 버전을 선택하여야 한다. 비주얼 스튜디오 2017은 2015와 호환된다.



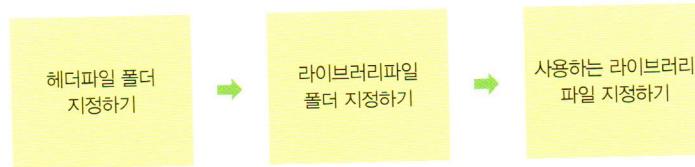
2. 파일을 다운로드하여 적당한 위치에 압축을 푼다. 여기서는 e 드라이브에 압축을 푸는 것으로 가정한다.



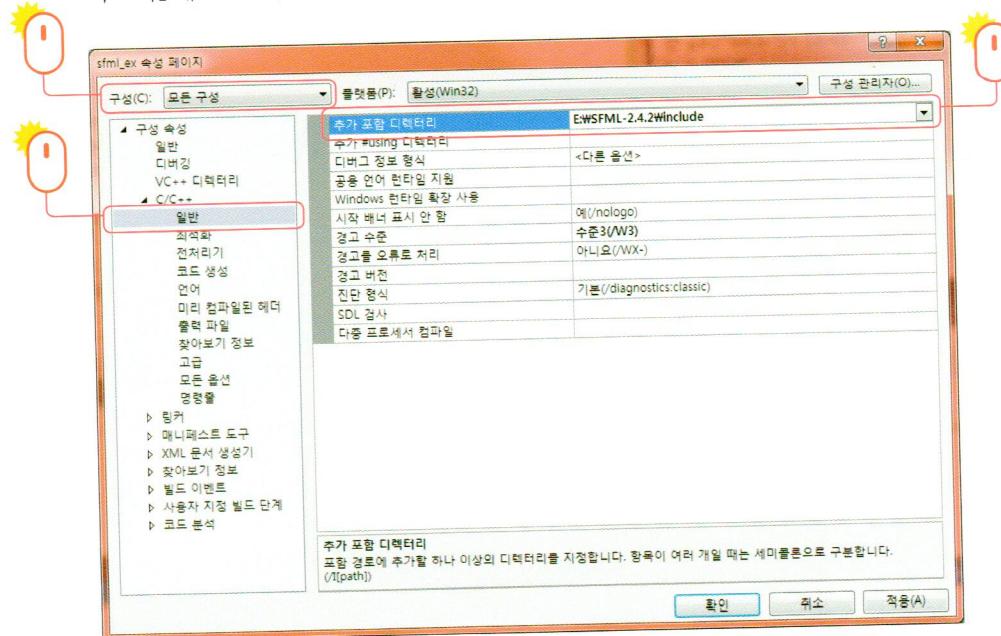
07.4

SFML 프로젝트 만들기

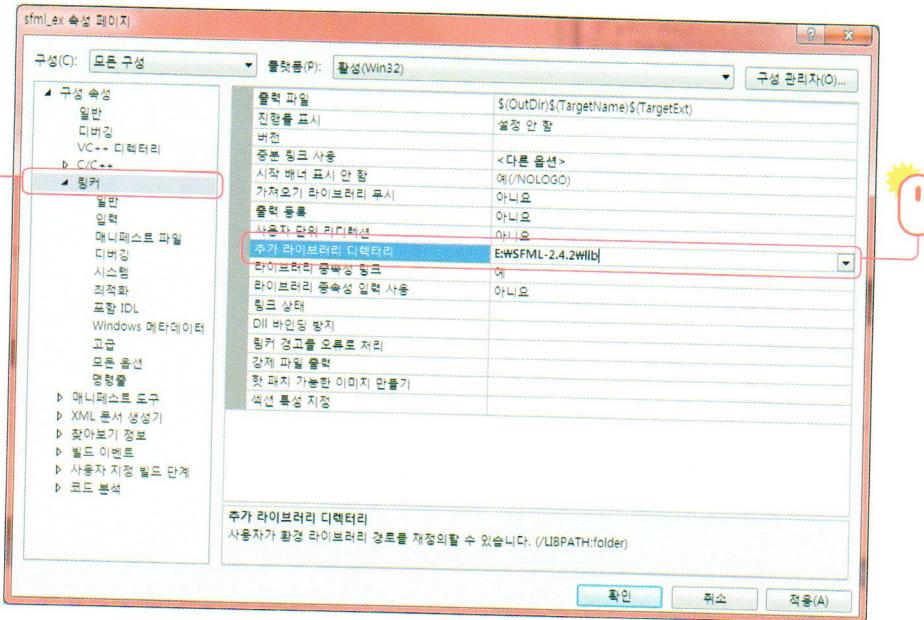
SFML을 사용하여 프로그램을 개발하려면 컴파일러가 헤더 파일과 라이브러리를 찾을 수 있는 위치를 알려주어야 한다. 이 과정은 조금 복잡하지만 외부 라이브러리를 사용하려면 반드시 거쳐야 하는 과정이다. 이 과정이 복잡하다면 출판사 홈페이지를 통하여 미리 만들어진 프로젝트 `sfml_sample`이 제공되니 이것을 다운로드한 후에 소스만 수정하여 사용하여도 된다. 다음과 같은 3단계를 거쳐야 한다.



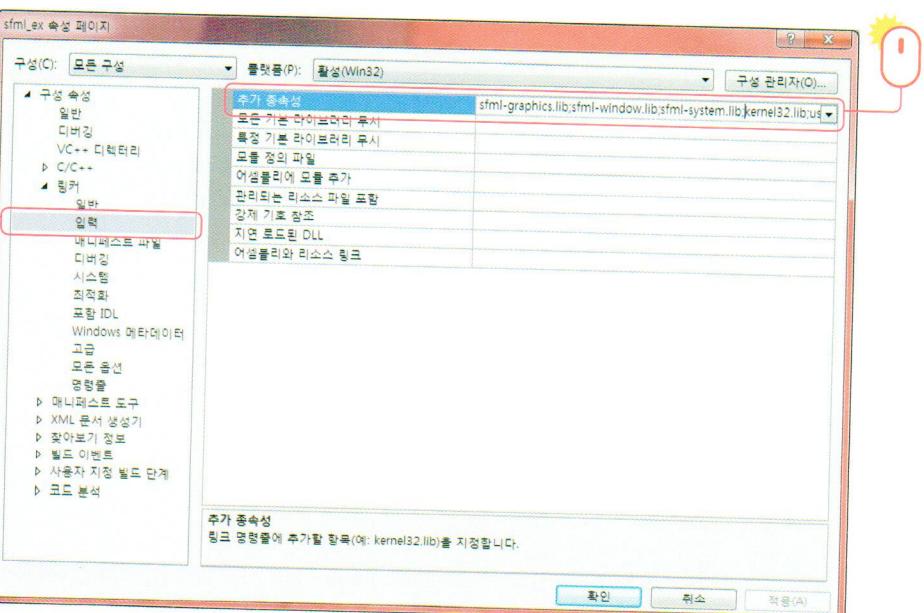
- 비주얼 스튜디오를 이용하여 새로운 프로젝트 `sfml_ex`를 생성한다. 프로젝트의 타입으로는 지금과 동일하게 “Win32 콘솔 응용 프로그램”을 선택한다. 프로젝트 마법사에서 “빈 프로젝트” 체크 박스를 선택한다. 소스 파일 `test.c`를 프로젝트에 추가한다.
- 컴파일러에게 SFML 헤더 파일을 찾을 위치를 알려줘야 한다. 메뉴 [프로젝트] → [`sfml_ex` 속성]을 선택하여 다음과 같은 대화 상자를 연다. 화면 왼쪽 상단의 구성에서 [모든 구성]을 선택한다. [구성 속성] → [C/C++] → [일반] → [추가 포함 디렉토리]에 SFML이 있는 경로를 붙여준다. 즉 “E:\SFML-2.4.2\include”를 추가한다.



3. 컴파일러에게 SFML 라이브러리 파일들을 찾을 위치를 알려줘야 한다. 앞의 화면에서 [구성 속성] → [링커] → [일반] → [추가 라이브러리 디렉토리]에 “E:\SFML-2.4.2\lib”를 추가한다.



4. 다음 단계는 코드에 필요한 SFML 라이브러리 (.lib 파일)을 지정하는 단계이다. SFML은 5개의 모듈(시스템, 윈도우, 그래픽, 네트워크, 오디오)로 구성되며 각각에 대해 하나의 라이브러리가 있다. 구성은 [모든 구성]으로 한다. 라이브러리는 [프로젝트 속성] → [링커] → [입력] → [추가 종속성]의 맨 앞에 sfml-graphics.lib;sfml-window.lib;sfml-system.lib;을 추가한다. 라이브러리 이름 사이에 ; 기호가 있어야 한다.



5. E:\SFML-2.4.2\bin 디렉토리 안의 모든 DLL 파일들을 현재 프로젝트 디렉토리로 복사한다.



6. 소스 파일 test.c 안에 다음 코드를 입력하고 컴파일하여 실행한다.

```
#include <SFML/Graphics.hpp>

int main()
{
    sf::RenderWindow window(sf::VideoMode(200, 200), "SFML works!");
    sf::CircleShape shape(100.f);
    shape.setFillColor(sf::Color::Green);

    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }
    }
}
```

```

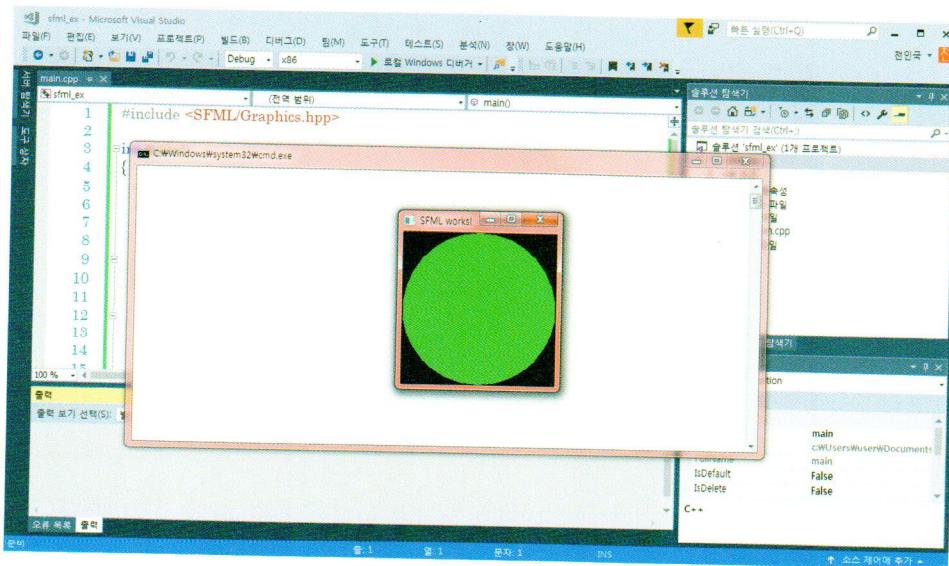
        }

        window.clear();
        window.draw(shape);
        window.display();
    }

    return 0;
}

```

7. 다음과 같은 실행결과를 확인한다.



07.5

SFML 기초

여기서는 sfml을 사용하기 위한 최소한도의 내용만을 다룬다. 보다 자세한 내용은 이 책의 부록이나 sfml 튜토리얼 (<https://www.sfml-dev.org/tutorials/2.4/>)을 참조하기 바란다.

일반적인 구조

SFML을 이용한 응용 프로그램은 다음과 같은 일반적인 구조를 가진다.

```
#include <SFML/Graphics.hpp>
using namespace sf;

int main()
{
    // 그림이 그려지는 화면을 생성한다.
    RenderWindow window(VideoMode(600, 480), "LUNAR LANDER");
    window.setFramerateLimit(60);

    // 게임에 필요한 스프라이트를 생성한다.
    Texture t2;
    Sprite lander;
    t2.loadFromFile("images/spaceship.png");
    lander.setTexture(t2);

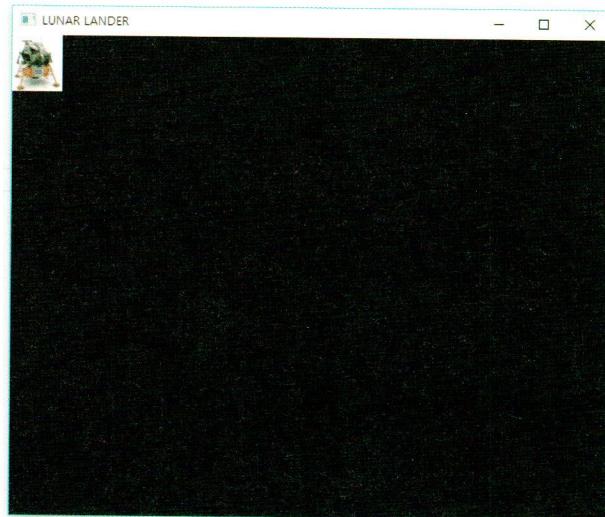
    // 여기서부터 게임 루프이다.
    while (window.isOpen())
    {
        // 이벤트 검사 및 처리
        Event e;
        while (window.pollEvent(e)) {
            if (e.type == Event::Closed)
                window.close();
        }

        // 화면을 지운다.
        window.clear();
    }
}
```

The diagram illustrates the five steps of the SFML game loop:

- ① 윈도우 생성**: The first step is to create a window using `RenderWindow window(VideoMode(600, 480), "LUNAR LANDER");`. This is highlighted in the first code block.
- ② 스프라이트 생성**: The second step is to create a sprite using `Texture t2;` and `Sprite lander;`, and load an image into the texture using `t2.loadFromFile("images/spaceship.png");`. This is highlighted in the second code block.
- ③ 사용자 이벤트 처리**: The third step is to handle user events using `Event e;` and `window.pollEvent(e)`. This is highlighted in the third code block.
- ④ 장면을 업데이트한다.**: The fourth step is to update the scene, which is implied by the code block containing the event loop.
- ⑤ 장면을 그린다.**: The fifth step is to clear the screen using `window.clear();`. This is highlighted in the fifth code block.

```
// 화면에 스프라이트를 그린다.  
window.draw(lander);  
// 화면을 표시한다.  
window.display();  
}  
  
return 0;  
}
```



1. 600 × 480 크기의 윈도우를 생성한다.

```
// 그림이 그려지는 화면을 생성한다.  
RenderWindow window(VideoMode(600, 480), "LUNAR LANDER");  
window.setFramerateLimit(60);
```

RenderWindow 클래스의 생성자가 호출된 것을 알 수 있다. 생성자의 첫 번째 인수인 비디오 모드는 윈도우의 크기를 정의한다. 여기에서는 600x480 크기의 윈도우를 만든다. 생성자의 두 번째 인수는 윈도우 제목이다.

2. 게임에 필요한 스프라이트를 선언한다.

스프라이트(sprite)는 텍스처가 있는 직사각형이라고 생각할 수 있다. 스프라이트는 텍스처(이미지)를 가질 수 있다. 스프라이트를 만들기 전에 유효한 텍스처가 필요하다. SFML에서 텍스처를 캡슐화하는 클래스가 Texture이다. 현재 코드에서 텍스처는 이미지 파일 "images/spaceship.png"을 적재하여 생성하고 있다. 현재 프로젝트 디렉토리 안에 이미지 파일이 있어야 한다.

```

Texture t2; // 텍스처 객체
Sprite lander; // 스프라이트 객체
t2.loadFromFile("images/spaceship.png"); // 이미지 파일을 텍스처 객체로 적재한다.
lander.setTexture(t2); // 스프라이트의 텍스처를 설정한다.

```

3. 게임 루프를 작성한다.

게임 루프(game loop)는 게임에 필요한 작업들을 처리하는 반복 루프이다.

```

while (window.isOpen()) {
    ...
}

```

윈도우가 열려져 있는 동안에는 게임 루프가 무한히 반복된다. 게임 루프에서는 다음과 같이 3가지의 작업이 반복된다.



4. 사용자의 입력 처리

게임 루프 내에서 제일 먼저 하는 일은 발생한 모든 이벤트를 확인하는 것이다. 보류중인 모든 이벤트가 처리 될 수 있도록 while 루프를 사용한다. `pollEvent()` 함수는 이벤트가 처리대기 중이면 `true`를 반환하고 그렇지 않으면 `false`를 반환 한다.

```

// 이벤트 검사 및 처리
Event e;
while (window.pollEvent(e)) {
    if (e.type == Event::Closed)
        window.close();
}

```

이벤트가 생길 때마다 우리는 이벤트 탐색(윈도우가 닫혀 있는지, 키가 눌려 켰는지, 마우스가 움직였는지, 조이스틱이 연결된 상태인지 ...)를 확인하고 반응해야 한다. 여기서는 사용자가 윈도우를 닫을 때 발생하는 이벤트 `Event::Closed`에만 신경 쓰고 있다. `Closed` 이벤트가 발생한 시점에서 윈도우가 열려 있다면 `close()` 함수를 사용하여 명시적으로 윈도우를 닫아야 한다.

5. 장면을 업데이트한다.

게임 루프에서 다음에 해야 할 일은 장면을 업데이트하는 것이다. 예를 들어서 사용자의 입력에 따라서 스프라이트의 위치를 이동시키는 것이다. 현재는 아무 것도 하지 않고 있다.

6. 장면을 그린다.

```
// 화면을 지운다.  
window.clear();  
  
// 화면에 스프라이트를 그린다.  
window.draw(lander);  
  
// 화면을 표시한다.  
window.display();
```

여기서는 윈도우를 지운 후에 모든 스프라이트를 그린다. 마지막으로 화면에 윈도우를 표시한다. 현재는 스프라이트 `lander`만을 그리고 있다. 만약 게임에 100개의 스프라이트가 있다면 여기서 모두 그려야 한다.

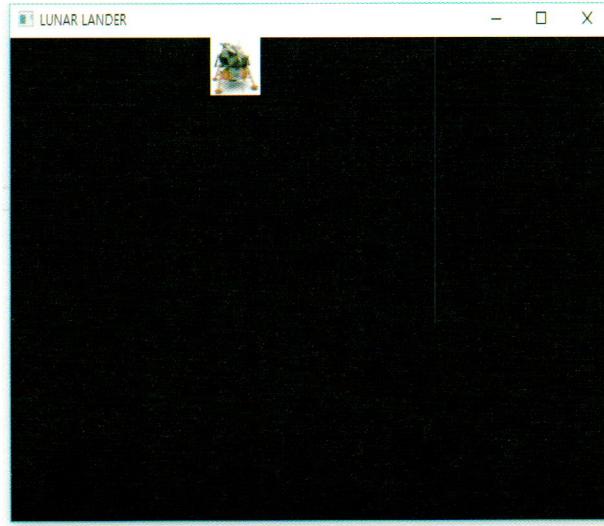
스프라이트를 움직여 보자.

키보드의 화살표키를 움직여서 스프라이트를 움직여 보자. 우리는 언제든지 키보드나 마우스의 상태를 조회할 수 있다. 키보드 상태에 대한 정보를 제공하는 클래스는 `sf::Keyboard`이다. 키보드의 현재 상태(키를 누르거나 해제한 상태)를 검사하는 함수는 `isKeyPressed()`이다. `isKeyPressed()` 함수는 직접 키보드 상태를 읽는다. 이것은 윈도우가 비활성 인 경우에도 `isKeyPressed()`가 `true`를 반환할 수 있음을 의미한다.

다음과 같은 코드를 앞의 코드의 ④ 위치에 추가한다.

```
if (Keyboard::isKeyPressed(Keyboard::Left))  
    lander.move(-10.0, 0.0);  
if (Keyboard::isKeyPressed(Keyboard::Right))  
    lander.move(10.0, 0.0);
```

위의 코드를 추가하면 키보드의 왼쪽 화살표키와 오른쪽 화살표키에 따라 스프라이트가 좌우로 움직이는 것을 볼 수 있다.



스프라이트의 현재 위치는 `setPosition(x, y)` 함수로도 설정이 가능하다.

마우스 상태는 다음과 같이 확인할 수 있다.

```
if (sf::Mouse::isButtonPressed(sf::Mouse::Left)) {  
    // 左쪽 마우스 버튼이 눌려있으면 ...  
}
```

마우스의 현재 위치는 다음과 같이 알 수 있다.

```
sf::Vector2i localPosition = sf::Mouse::getPosition(window);
```

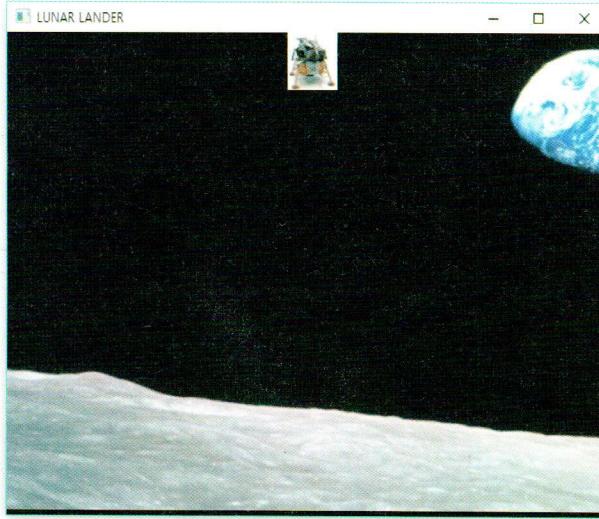
배경 화면을 생성해보자.

위의 코드의 ② 위치에 배경 이미지를 적재하는 코드를 추가해보자.

```
Texture t1;                                // 텍스처 객체  
Sprite background;                          // 스프라이트 객체  
t1.loadFromFile("images/background.png");    // 이미지를 텍스처 객체로 적재한다.  
background.setTexture(t1);                  // 스프라이트의 텍스처를 설정한다.
```

배경 이미지도 ⑤에서 그려주어야 한다.

```
// 화면에 스프라이트를 그린다.  
window.draw(background);  
window.draw(lander);
```



현재까지의 소스는 다음과 같다.

```
#include <SFML/Graphics.hpp>
using namespace sf;

int main()
{
    // 그림이 그려지는 화면을 생성한다.
    RenderWindow window(VideoMode(600, 480), "LUNAR LANDER");
    window.setFramerateLimit(60);

    Texture t1;                                // 텍스처 객체
    Sprite background;                          // 스프라이트 객체
    t1.loadFromFile("images/background.png");    // 이미지를 텍스처 객체로 적재한다.
    background.setTexture(t1);                  // 스프라이트의 텍스처를 설정한다.

    // 게임에 필요한 스프라이트를 생성한다.
    Texture t2;
    Sprite lander;
    t2.loadFromFile("images/spaceship.png");
    lander.setTexture(t2);

    // 여기서부터 게임 루프이다.
    while (window.isOpen())
    {
        // 이벤트 검사 및 처리
        Event e;
        while (window.pollEvent(e)) {
```

```
    if (e.type == Event::Closed)
        window.close();
    }

    if (Keyboard::isKeyPressed(Keyboard::Left))
        lander.move(-10.0, 0.0);
    if (Keyboard::isKeyPressed(Keyboard::Right))
        lander.move(10.0, 0.0);

    // 화면을 지운다.
    window.clear();

    // 화면에 스프라이트를 그린다.
    window.draw(background);
    window.draw(lander);

    // 화면을 표시한다.
    window.display();
}

return 0;
}
```

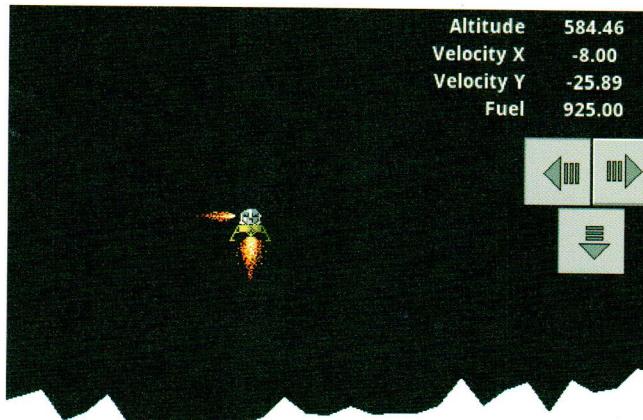
도전문제

착륙선이 위쪽이나 아래쪽으로 움직이도록 키보드 이벤트 처리를 추가해보자.

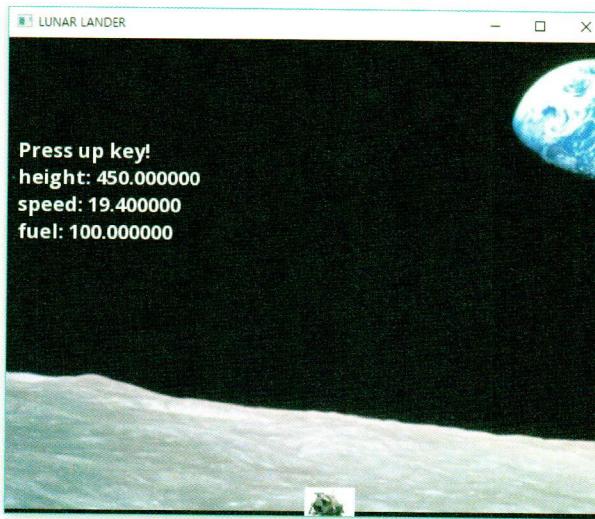
07.6

Lunar Lander 게임

Lunar Lander 게임은 달에 우주선을 착륙시키는 게임이다. 우주선이 하강하는 속도가 너무 빠르면 우주선은 추락한다. 따라서 로켓 분사를 통하여 우주선의 속도를 적절하게 유지하여야 한다. 로켓 분사가 없다면 우주선의 떨어지는 속도가 중력 때문에 계속 빨라질 것이다. 실제 상용 게임의 화면은 다음과 같다.



우리는 간단히 다음과 같이 작성할 것이다. 착륙선은 이미지로 화면에 그려진다. 착륙선의 현재 위치, 현재 속도, 남아 있는 연료의 양을 텍스트로 화면에 표시한다. 코드를 간단하게 하기 위하여 착륙선은 상하로만 움직인다(좌우로는 움직이지 않는다). 사용자가 위쪽 화살표키를 누르면 로켓이 분사된다.



일단 주어진 소스를 실행하여 어떤 게임인지 살펴보자. 착륙 속도가 너무 빠르면 추락으로 판정한다(현재는 구현되어 있지 않다). 즉 높이가 0에 가까워질 때 우주선의 속도가 중요하다. 위쪽 화살표 키를 누르면 한 번씩 로켓 분사가 일어나서 우주선이 위로 움직인다. 연료의 양은 한정되어 있기 때문에 아껴 써야 한다.

전체 소스

전체 소스는 다음과 같다.

```
#include <SFML/Graphics.hpp>
#include <windows.h>
#include <string>
#include <vector>
#include <iostream>
using namespace sf;
using namespace std;

class LunarLander {
private:
    double x, y;                                // 현재 위치
    double velocity;                             // 속도
    double fuel;                                 // 연료
    string status;                               // 현재 상태
    Texture t1, t2;                             // 텍스처 객체
    Sprite spaceship, burst;                     // 스프라이트 객체
    Font font;                                  // 폰트 객체
    Text text;                                  // 텍스트 객체
public:
    LunarLander(double h, double v, double f);   // 생성자
    bool checkLanded();                          // 착륙 검사 함수
    void update(double rate);                   // 상태 업데이트 함수
    void draw(RenderWindow &window);            // 착륙선 그리는 함수
};

// 생성자 함수
LunarLander::LunarLander(double h, double v, double f) {
    x = 300;
    y = h;
    velocity = v;
    fuel = f;
    t1.loadFromFile("images/spaceship.png");
    t2.loadFromFile("images/burst.png");
    spaceship.setTexture(t1);
```

```

        burst.setTexture(t2);
        spaceship.setPosition(x, y);
        burst.setPosition(x + 20, y + 50);
        if (!font.loadFromFile("OpenSans-Bold.ttf")) {
            cout << "폰트 파일을 오픈할 수 없음!" << endl;
        }
        text.setFont(font);
    }

    // 착륙했는지를 검사하는 함수. 만약 높이가 0보다 작으면 착륙한 것이다.
    bool LunarLander::checkLanded() {
        if (y <= 0)
            return true;
        return false;
    }

    // 게임 상태를 업데이트한다.
    void LunarLander::update(double amount) {
        if (fuel <= 0) {
            fuel = 0;
            amount = 0;
        }
        fuel = fuel - amount;
        velocity = velocity - amount + 0.8;
        y = y + velocity;
        if (y > 450) y = 450;
        spaceship.setPosition(x, y);
        burst.setPosition(x + 20, y + 50);
        status = "Press up key!\nheight: " + to_string(y) + "\nspeed: " +
            to_string(velocity) + "\nfuel: " + to_string(fuel);
    }

    // 화면에 착륙선과 불꽃, 현재 상태를 그린다.
    void LunarLander::draw(RenderWindow &window) {
        window.draw(spaceship);
        window.draw(burst);
        text.setString(status);
        text.setCharacterSize(20);
        text.setPosition(10, 100);
        window.draw(text);
    }

    int main()
{

```

```

RenderWindow window(VideoMode(600, 480), "LUNAR LANDER");
window.setFramerateLimit(60);

Texture t;
Sprite background;
t.loadFromFile("images/background.png");
background.setTexture(t);

LunarLander lander(300.0, 1.0, 100.0); // 착륙선 객체 생성
while (window.isOpen())
{
    Event e;
    while (window.pollEvent(e))
    {
        if (e.type == Event::Closed)
            window.close();
    }

    if (Keyboard::isKeyPressed(Keyboard::Up))
        lander.update(3.0);
    else
        lander.update(0.0);

    window.clear();
    window.draw(background);
    lander.draw(window);

    window.display();
    Sleep(100); // 0.1초 동안 잠재운다.
}

return 0;
}

```

소스 설명

여기서는 `main()` 함수가 게임 루프의 역할을 한다. 다음과 같은 작업을 한다.

```

int main()
{
    while (true) {
        사용자 이벤트를 처리한다.
        사용자가 위쪽 화살표 키를 누르면 엔진 분사를 한다.
        착륙선의 속도, 위치, 연료량을 업데이트한다.
}

```

```
화면을 지운다.  
착륙선을 화면에 그린다.  
}  
}
```

우리가 첫 번째 생각해야 할 것은 과연 어떤 클래스가 필요한가이다. 이번 프로그램에서는 단 하나의 객체, 착륙선만 있으면 된다. 따라서 다음과 같이 `LunarLander` 클래스를 작성해보자.

```
class LunarLander {  
private:  
    double x, y;  
    double velocity;  
    double fuel;  
    ...  
public:  
    LunarLander(double h, double v, double f); // 생성자  
    bool checkLanded();  
    void update(double rate);  
    void draw(RenderWindow &app);  
};
```

`LunarLander` 클래스의 멤버 변수로 현재 위치를 저장하고 있는 `x`와 `y`, 현재 속도를 저장하고 있는 `velocity`, 현재 남아 있는 연료량을 저장하고 있는 `fuel`을 가지고 있다. 이들 멤버 변수에 대한 접근자와 설정자 함수도 작성하면 좋다.

`LunarLander` 클래스의 멤버 함수는 생성자와 `checkLanded()`, `update()`, `draw()` 등이 정의된다. 이중에서 가장 중요한 함수는 `update()`이다. 이 함수는 0.1초에 한 번씩 호출되며 착륙선의 위치와 속도, 연료량을 업데이트한다.

```
void LunarLander::update(double amount) {  
    if (fuel <= 0) {  
        fuel = 0;  
        amount = 0;  
    }  
    fuel = fuel - amount; // 연료 감소  
    velocity = velocity - amount + 0.8;  
    y = y + velocity;  
    spaceship.setPosition(x, y);  
    ...  
}
```



이 함수의 매개변수 `amount`는 연료 분사량이다. 연료 분사가 있으면 3.0이 전달되고, 연료 분사가 없으면 0.0이 전달된다. 연료의 양은 `amount`만큼 감소된다. 연료를 분사하면 착륙선의 하강 속도는 늦춰진다. 따라서 연료분사량 `amount`만큼 하강 속도를 감속한다. 중력가속도는 착륙선의 하강 속도를 증가시킨다. 따라서 기존의 착륙선 속도에 중력가속도는 더해진다. 현재 중력가속도는 0.8이라고 생각한다. 1초가 흐른 것으로 가정하므로 착륙선의 높이는 기존의 높이에다가 (속도*1초)를 더한 것이 된다.

화면에 착륙선을 그리는 멤버 함수는 `draw()`이다.

```
// 화면에 착륙선을 그린다.
void LunarLander::draw(RenderWindow &window) {
    window.draw(spaceship);
    window.draw(burst);
    text.setString(status);
    text.setCharacterSize(20);
    text.setPosition(10, 100);
    window.draw(text);
}
```

`draw()`에서는 매개 변수를 통하여 `RenderWindow` 객체를 받고 있다. 이 객체가 있어야 그리기가 가능하다. `draw()`에서는 착륙선과 함께 불꽃 모양도 함께 그려준다. 그리고 화면의 왼쪽에 착륙선의 현재 상태를 텍스트로 표시한다. SFML에서 텍스트를 화면에 표시하려면 `Font` 객체와 `Text` 객체가 필요하다. 한 가지 특이한 사항은 폰트 파일이 현재 프로젝트 디렉토리에 있어야 한다는 점이다. 폰트 파일은 `.ttf`라는 확장자를 가지면 몇 가지의 폰트 파일은 인터넷에서 자유롭게 다운로드받아서 사용할 수 있다. 첨부된 소스에서는 “`OpenSans-Bold`” 파일을 사용하고 있다.

도전문제

1. 착륙선이 지표면에 달았을 때 착륙선의 속도가 빠르면 착륙에 실패했다고 화면에 표시한다.
반대로 착륙선의 속도가 느리면 성공이라고 화면에 표시한다.
2. 착륙선이 왼쪽이나 오른쪽으로 움직이도록 키보드 이벤트 처리를 추가해보자.

07.7

벽돌깨기

“벽돌깨기” 게임을 SFML을 이용하여 작성해보자.



이 게임에서는 벽돌, 공, 패들을 클래스로 작성하여야 한다. 공과 패들은 하나만 있으면 되지만 벽돌은 상당히 많다. 무엇을 사용해야 할까? 2차원 배열을 사용하여도 된다. 하지만 벽돌이 공과 충돌하면 소멸되어야 한다. 따라서 동적 배열인 벡터를 사용하면 편리하다. 공과 충돌한 벽돌은 `erase()` 함수를 이용하여서 벡터에서 제거하면 된다.

전체 소스

전체 소스는 다음과 같다. 각 부분은 상세하게 다시 설명된다.

```
#include <SFML/Graphics.hpp>
#include <vector>
#include <time.h>
using namespace sf;

int main()
{
    srand(time(NULL));
}
```

```

RenderWindow window(VideoMode(600, 480), "ARKANOID");
window.setFramerateLimit(60);

Texture t1, t2, t3, t4;
t1.loadFromFile("images/block.png");
t2.loadFromFile("images/background.png");
t3.loadFromFile("images/ball.png");
t4.loadFromFile("images/paddle.png");

Sprite background(t2), ball(t3), paddle(t4);
paddle.setPosition(300, 460);

const int size = 100;
std::vector<Sprite> blocks(100);
int n = 0;
auto bsize = t1.getSize();
for (int i = 1; i <= 10; i++)
    for (int j = 1; j <= 10; j++)
    {
        blocks[n].setTexture(t1);
        blocks[n].setPosition(i * bsize.x, j * bsize.y);
        n++;
    }

float dx = 3, dy = 3;
while (window.isOpen())
{
    Event e;
    while (window.pollEvent(e)) {
        if (e.type == Event::Closed)
            window.close();
    }

    auto ball_pos = ball.getPosition();

    ball_pos.x += dx;
    for (int i = 0; i < blocks.size(); i++) {
        if (FloatRect(ball_pos.x + 3, ball_pos.y + 3, 6, 6).intersects(
            blocks[i].getGlobalBounds()))
            {
                blocks.erase(blocks.begin() + i);
                dx = -dx;
            }
    }
}

```

```

    }

    ball_pos.y += dy;
    for (int i = 0; i < blocks.size(); i++) {
        if (FloatRect(ball_pos.x + 3, ball_pos.y + 3, 6, 6).intersects(
            blocks[i].getGlobalBounds()))
        {
            blocks.erase(blocks.begin() + i);
            dy = -dy;
        }
    }

    if (ball_pos.x<0 || ball_pos.x>520) dx = -dx;
    if (ball_pos.y<0 || ball_pos.y>450) dy = -dy;

    if (Keyboard::isKeyPressed(Keyboard::Right))
        paddle.move(5, 0);
    if (Keyboard::isKeyPressed(Keyboard::Left))
        paddle.move(-5, 0);

    if (FloatRect(ball_pos.x, ball_pos.y, 12, 12).intersects(paddle.
        getGlobalBounds()))
        dy = -(rand() % 5 + 2);

    ball.setPosition(ball_pos.x, ball_pos.y);

    window.clear();
    window.draw(background);
    window.draw(ball);
    window.draw(paddle);

    for (auto& obj : blocks)
        window.draw(obj);

    window.display();
}

return 0;
}

```

소스 설명

1. 배경, 볼, 패들은 모두 스프라이트 객체로 생성하면 된다. 적당한 이미지 파일을 구해서 텍스처 객체로 만들어둔다.

```

Texture t1, t2, t3, t4;
t1.loadFromFile("images/block.png");
t2.loadFromFile("images/background.png");
t3.loadFromFile("images/ball.png");
t4.loadFromFile("images/paddle.png");

Sprite background(t2), ball(t3), paddle(t4);
paddle.setPosition(300, 460);

```

- 2.** 벽돌은 스프라이트 100개를 벡터에 저장하여 작성한다.

```
vector<Sprite> blocks(100);
```

- 3.** 벽돌 100개를 화면에 2차원 형태로 배치한다. 각 벽돌의 위치는 setPosition() 함수를 사용한다. 각 벽돌의 텍스처를 t1으로 설정한다.

```

for (int i = 1; i <= 10; i++)
{
    for (int j = 1; j <= 10; j++)
    {
        blocks[n].setTexture(t1);
        blocks[n].setPosition(i * bsize.x, j * bsize.y);
        n++;
    }
}

```

- 4.** 게임 루프를 생성한다.

```

while (window.isOpen())
{
    Event e;
    while (window.pollEvent(e))
    {
        if (e.type == Event::Closed)
            window.close();
    }
}

```

사용자가 “Closed” 아이콘을 누르면 전체 응용 프로그램은 종료된다.

- 5.** 공의 현재 위치를 얻어서 벽돌과 충돌하였는지를 검사한다. 이때 `FloatRect` 객체의 `intersect()` 함수를 사용한다.

```

auto ball_pos = ball.getPosition();

ball_pos.x += dx;

```

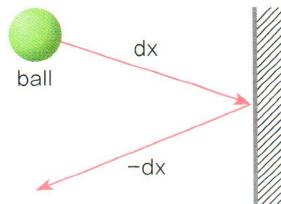
```

for (int i = 0; i < blocks.size(); i++) {
    if (FloatRect(ball_pos.x + 3, ball_pos.y + 3, 6, 6).intersects(blocks[i].
        getGlobalBounds()))
    {
        blocks.erase(blocks.begin() + i);
        dx = -dx;
    }
}

```

`FloatRect` 클래스는 사각형을 나타내는 클래스이다. 현재 공의 위치를 기준으로 조그마한 사각형을 만들어서 벽돌과 충돌했는지를 검사한다. 만약 충돌하였으면 `blocks` 벡터의 `i`번째 요소를 삭제한다. 그리고 공이 벽돌에 맞으면 반사되는 것으로 방향을 변경한다.

공의 `x`방향 속도와 `y`방향 속도는 각각 `dx` 변수와 `dy` 변수에 저장되어 있다. 이것을 음수로 하면 공이 움직이는 방향이 반대가 된다.



6. 공이 화면을 벗어나면 방향을 반대로 한다.

```

if (ball_pos.x<0 || ball_pos.x>520) dx = -dx;
if (ball_pos.y<0 || ball_pos.y>450) dy = -dy;

```

7. 키보드 이벤트를 처리하여서 패들을 움직인다.

```

if (Keyboard::isKeyPressed(Keyboard::Right))
    paddle.move(5, 0);
if (Keyboard::isKeyPressed(Keyboard::Left))
    paddle.move(-5, 0);

```

8. 공과 패들이 충돌하면 약간의 난수를 이용하여 불규칙하게 반사시킨다.

```

if (FloatRect(ball_pos.x, ball_pos.y, 12, 12).intersects(paddle.
    getGlobalBounds()))
    dy = -(rand() % 5 + 2);

ball.setPosition(ball_pos.x, ball_pos.y);

```

9. 화면에 모든 스프라이트를 그려준다. 범위 기반 루프가 사용되고 있다.

```
window.clear();
window.draw(background);
window.draw(ball);
window.draw(paddle);

for (auto& obj : blocks)
    window.draw(obj);

window.display();
```

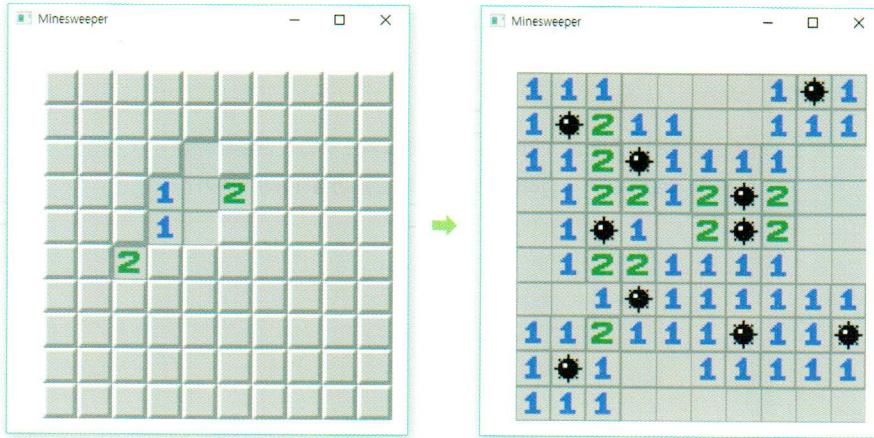
도전문제

- 1 공이 화면의 하단을 벗어나면 게임이 종료되는 것으로 변경해보자.
- 2 공의 속도를 느리게 하려면 어떤 변수의 값을 변경하여야 하는가?

07.8

지뢰 찾기 게임 작성

지뢰 찾기 게임은 예전 윈도우에는 기본으로 포함된 게임이었다. 플레이어는 처음에 사각형의 격자가 제시된다. 무작위로 선택된 일부 사각형은 지뢰를 포함하도록 지정된다. 일반적으로 격자의 크기와 지뢰의 수는 사용자가 미리 설정할 수 있다.



사용자는 각 사각형을 클릭하여 사각형을 열 수 있다. 지뢰가 있는 사각형을 클릭하면 사용자가 진다. 지뢰가 없으면 인근의 8개의 사각형에 있는 지뢰의 개수를 나타내는 숫자가 표시된다. 인접한 지뢰가 없다면 사각형은 공백이 된다. 사용자가 이 정보를 사용하여 다른 사각형의 내용을 추론하고 사각형이 지뢰를 포함하고 있는 것으로 표시할 수 있다.

전체 소스

```
#include <iostream>
#include <string>
#include <time.h>
#include <SFML/Graphics.hpp>
using namespace sf;
using namespace std;

// 격자의 하나의 사각형을 나타낸다.
class Tile {
public:
    bool open;
```

```

        int number;
    };

    const int TITLE_SIZE = 35;

    // 사각형의 상태를 나타낸다.
    const int BOMB = 9;
    const int HIDDEN = 10;

    int main()
    {
        srand(time(NULL));
        RenderWindow app(VideoMode(400, 400), "Minesweeper");

        // 게임보드를 grid[][] 배열로 나타낸다.
        Tile grid[12][12];
        bool game_ended = false;

        // 12개의 텍스처와 스프라이트를 생성한다.
        Sprite sprites[12];
        Texture t[12];
        for (int k = 0; k < 12; k++) {
            t[k].loadFromFile("images/tile"+to_string(k)+".png");
            sprites[k].setTexture(t[k]);
        }

        // 게임보드에 지뢰를 저장한다.
        for (int i = 1; i <= 10; i++) {
            for (int j = 1; j <= 10; j++)
            {
                grid[i][j].open = false;
                grid[i][j].number = 0;
                if ((rand() % 10) == 1)
                    grid[i][j].number = BOMB;
            }
        }

        // 게임보드에 인근지뢰의 개수를 계산하여 저장한다.
        for (int i = 1; i <= 10; i++) {
            for (int j = 1; j <= 10; j++)
            {
                int n = 0;
                if (grid[i][j].number == BOMB)      continue;

```

```

        if (grid[i + 1][j].number == BOMB) n++;
        if (grid[i][j + 1].number == BOMB) n++;
        if (grid[i - 1][j].number == BOMB) n++;
        if (grid[i][j - 1].number == BOMB) n++;
        if (grid[i + 1][j + 1].number == BOMB) n++;
        if (grid[i - 1][j - 1].number == BOMB) n++;
        if (grid[i + 1][j - 1].number == BOMB) n++;
        grid[i][j].number = n;
    }
}

// 메인루프이다.
while (app.isOpen())
{
    // 마우스가 놓인 사각형의 번호를 계산한다.
    Vector2i pos = Mouse::getPosition(app);
    int x = pos.x / TILE_SIZE;
    int y = pos.y / TILE_SIZE;

    // 윈도우 이벤트를 처리한다.
    Event e;
    while (app.pollEvent(e))
    {
        if (e.type == Event::Closed)
            app.close();

        // 왼쪽 마우스 버튼이 눌렸으면 해당 사각형에 표시한다.
        if (e.type == Event::MouseButtonPressed)
            if (e.key.code == Mouse::Left) {
                grid[x][y].open = true;
                if (grid[x][y].number == BOMB) game_ended = true;
            }
    }

    // 게임 보드를 화면에 그린다.
    app.clear(Color::White);
    for (int i = 1; i <= 10; i++) {
        for (int j = 1; j <= 10; j++)
        {
            int n;
            // 사각형의 상태에 따라 이미지를 지정하여 화면에 그린다.
            if (grid[i][j].open == false && game_ended == false)

```

```

        n = HIDDEN;
    else
        n = grid[i][j].number;
    sprites[n].setPosition(TILE_SIZE * i, TILE_SIZE * j);
    app.draw(sprites[n]);
}
}

app.display();
}

return 0;
}

```

소스 설명

1. 이미지를 준비하자.

다음과 같은 12개의 이미지 파일을 준비한다.



2. 게임 보드를 2차원 객체 배열로 나타내자.

게임 보드의 하나의 사각형을 클래스 `Tile`의 객체로 나타낸다.

```

class Tile {
public:
    bool open;
    int number;
};

Tile grid[12][12];

```

클래스 `Tile`은 사각형이 열려 있는지를 표시하는 `open` 변수와 인근의 지뢰를 나타내는 `number` 변수로 이루어져 있다. 게임 보드는 10×10 크기이지만 경계 조건을 쉽게 검사하기 위하여 12×12 로 정의한다.

3. 2차원 배열 `grid`에 지뢰를 저장한다.

지뢰는 10%의 확률로 저장된다. 난수를 발생하여서 난수를 10으로 나누었을 때 나머지가 1이 나오면 그 위치에 지뢰를 저장한다. 지뢰는 `BOMB` 상수로 표시된다. 지뢰가 없는 사각형에는 0을 저장한다.

```

for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10; j++)
    {
        grid[i][j].open = false;
        grid[i][j].number = 0;
        if ((rand() % 10) == 1)
            grid[i][j].number = BOMB;
    }
}

```

4. 인근 지뢰의 개수를 계산한다.

2차원 배열 `grid[][]`에 인근 지뢰의 개수를 계산하여 저장한다. 게임 보드는 10×10 크기이지만 경계 조건을 쉽게 검사하기 위하여 12×12 로 정의되어 있다.

```

for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10; j++)
    {
        int n = 0;
        if (grid[i][j].number == BOMB) continue;
        if (grid[i + 1][j].number == BOMB) n++;
        if (grid[i][j + 1].number == BOMB) n++;
        if (grid[i - 1][j].number == BOMB) n++;
        if (grid[i][j - 1].number == BOMB) n++;
        if (grid[i + 1][j + 1].number == BOMB) n++;
        if (grid[i - 1][j - 1].number == BOMB) n++;
        if (grid[i - 1][j + 1].number == BOMB) n++;
        if (grid[i + 1][j - 1].number == BOMB) n++;
        grid[i][j].number = n;
    }
}

```

5. 텍스처와 스프라이트 객체를 생성한다.

총 12개의 서로 다른 이미지가 있다. 따라서 크기가 12인 `Texture` 배열과 `Sprite` 배열을 생성하여 이들을 표현한다.

```

Sprite sprites[12];
Texture t[12];
for (int k = 0; k < 12; k++) {
    t[k].loadFromFile("images/tile"+to_string(k)+".png");
    sprites[k].setTexture(t[k]);
}

```

Texture 배열의 각 요소에 디스크에 있는 이미지 파일을 읽어 놓는다. Texture 배열과 Sprite 배열을 연결한다.

6. 사용자로부터 이벤트를 받아서 처리한다.

```
while (app.isOpen())
{
    Vector2i pos = Mouse::getPosition(app);
    int x = pos.x / TILE_SIZE;
    int y = pos.y / TILE_SIZE;

    Event e;
    while (app.pollEvent(e))
    {
        if (e.type == Event::Closed)
            app.close();

        if (e.type == Event::MouseButtonPressed)
            if (e.key.code == Mouse::Left) {
                grid[x][y].open = true;
                if (grid[x][y].number == BOMB) game_ended = true;
            }
    }
}
```

윈도우가 열려 있는 동안에는 마우스의 위치를 받아서 마우스가 놓인 사각형의 번호를 계산한다.

```
Vector2i pos = Mouse::getPosition(app);
int x = pos.x / TILE_SIZE;
int y = pos.y / TILE_SIZE;
```

이벤트 중에서 2가지의 이벤트만 처리한다. 하나는 “close” 이벤트이다. 또 하나는 마우스 버튼 클릭 이벤트 Event::MouseButtonPressed로 특히 왼쪽 마우스 버튼만을 처리한다. 왼쪽 마우스 버튼이 눌리면 해당되는 사각형의 open 변수를 true로 변경한다. 그리고 만약 지뢰가 들어 있는 사각형이라면 게임의 종료를 의미하는 game_ended 변수를 true로 설정한다.

7. 현재 상태를 그린다.

```
app.clear(Color::White);
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10; j++)
    {
```

```
int n;
if (grid[i][j].open == false && game_ended == false)
    n = HIDDEN;
else
    n = grid[i][j].number;
sprites[n].setPosition(TILE_SIZE * i, TILE_SIZE * j);
app.draw(sprites[n]);
}
}

app.display();
```

제일 먼저 하는 작업은 `clear()`를 호출하여서 이전에 그려진 것들을 지우는 것이다. 이중 루프를 실행하면서 `grid[][]`의 각각의 요소를 검사한다. `open이 false이면` `HIDDEN 사각형으로 분류한다.` 그렇지 않으면 인근 지뢰의 개수를 변수 `n에 저장한다.` 스프라이트의 위치를 현재 그리고 있는 사각형의 위치로 변경하고 `draw()`를 호출하여서 사각형을 이미지로 화면에 그린다.



SFML로 작성해보는 프로젝트 II

학습목표

- 상속을 사용하여 게임에 필요한 클래스를 설계할 수 있다.

학습목차

16.1 이번 장에서 만들어 볼 프로그램

16.2 벽돌 깨기 게임 만들기

이제 무엇을 하여야 하나요?

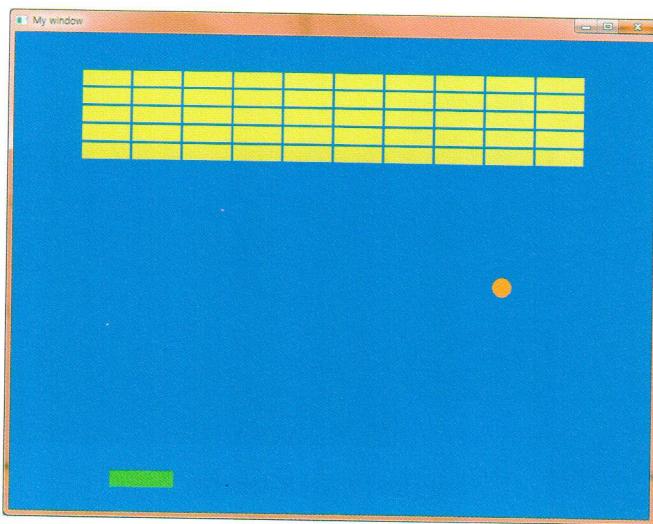
지금까지 학습한 내용을 바탕
으로 기말 프로젝트를 제작해
보세요!



16.1

이번 장에서 만들어 볼 프로그램

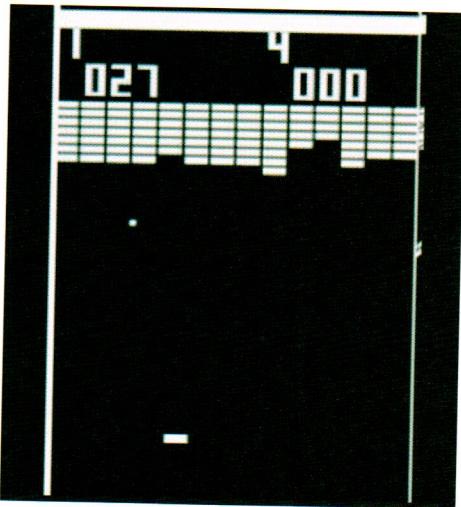
이번 장에서는 지금까지 학습한 내용을 여러 가지로 응용해서 다음과 같이 공을 이용해서 벽돌을 깨는 고전 게임을 작성해보자. 7장에서도 만들어보았지만 이번 장에서는 상속을 이용하여 프로그램을 다시 작성해본다. 사용자는 마우스를 이용해서 패들을 움직이고 공은 패들에 맞으면 반사된다. 공이 벽돌에 맞으면 벽돌은 깨진다.



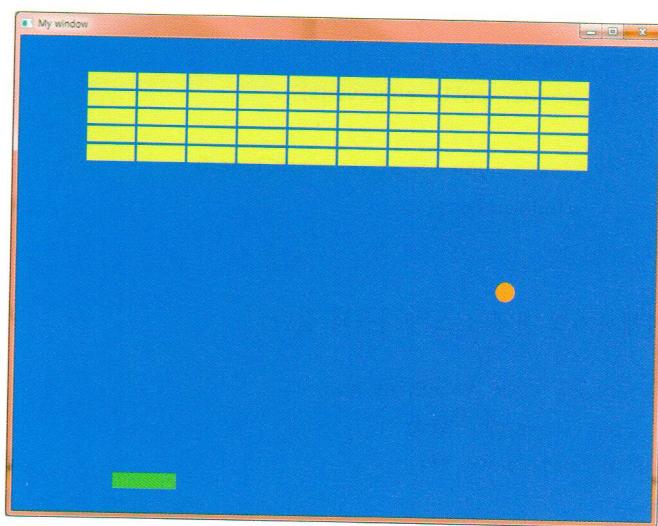
16.2

벽돌 깨기 게임 만들기

최초의 벽돌 깨기 게임은 아타리사가 만든 *Breakout*이었다. 놀라운 점은 이 게임의 프로그래머가 스티브 워즈니악과 스티브 잡스였다는 점이다(스티브 잡스는 아타리사에서 일했었다). 게임에서, 벽돌 층이 화면의 상단 1/3을 채운다. 공이 화면을 가로 질러 이동하면서 화면의 상단과 측면 벽에서 반사된다. 공이 벽돌에 부딪치면 벽돌이 파괴된다. 공이 화면 하단에 닿으면 플레이어가 진다. 플레이어는 공을 반사시키는 패들을 가지고 있다.

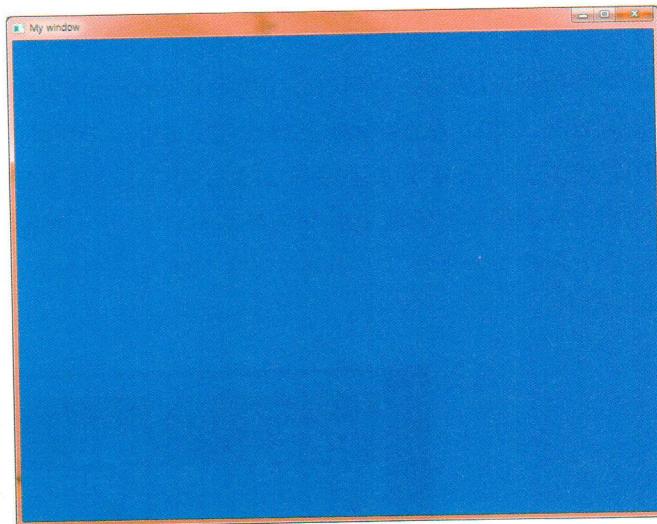


이 책에서는 위의 벽돌 깨기와 유사한 게임을 SFML로 단계적으로 작성해보자.



Step #1 화면을 작성해보자.

다음과 같이 배경색이 청색으로 칠해진 윈도우를 생성해보자. 이것은 앞 절의 소스를 조금만 고치면 가능하다.



화면에 무엇인가를 그리려면 특수한 윈도우인 `sf::RenderWindow`를 사용하여야 한다. 이 클래스는 `sf::Window` 및 모든 기능을 상속받은 자식 윈도우이다. 모든 `sf::Window`(생성, 이벤트 처리, 프레임 속도를 제어, OpenGL 사용)의 기능은 `sf::RenderWindow`에서 사용할 수 있다.

`sf::RenderWindow`는 쉽게 그림을 그릴 수 있도록 `sf::Window`에 높은 수준의 기능을 추가한다. 여기서는 2개의 함수에 초점을 맞추어보자. `clear()`와 `draw()`이다. `clear()` 함수는 선택한 색상으로 전체 윈도우를 지우고 `draw()`는 여러분이 전달한 객체를 화면에 그린다. 아래의 반복 루프에서는 이벤트를 처리한다. 현재는 윈도우를 닫는 이벤트만을 처리하고 있다.

```
Event event;
while (window.pollEvent(event))
{
    if (event.type == sf::Event::Closed)
        window.close();
}
```

현재까지의 전체 소스는 다음과 같다.

```
#include <SFML/Graphics.hpp>
using namespace std;
using namespace sf;
```

```

int main()
{
    RenderWindow window(VideoMode(800, 600), "My window");
    window.setFramerateLimit(60);

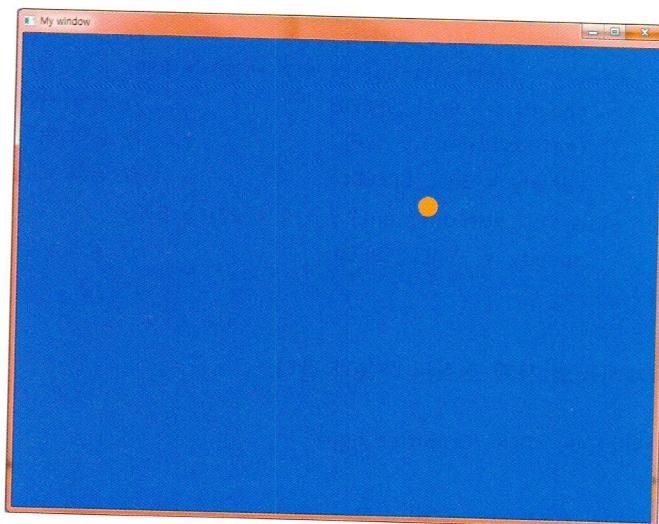
    while (window.isOpen())
    {
        window.clear(sf::Color::Blue);

        Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }
        window.display();
    }
    return 0;
}

```

Step #2 공을 화면에 그려보자.

공을 클래스로 정의하고 객체를 생성하여 화면에 표시해 보자. 우리는 상속을 사용한다.



Ball 클래스는 CircleShape이라는 클래스를 상속받아서 작성하자. 우리가 추가한 멤버 변수로는 speedx와 speedy가 있다. 이것은 볼이 이동하는 x방향 속도와 y방향

속도를 나타낸다. `Ball` 클래스 `Ball(float x, float y)` 생성자와 `update()` 함수를 가지고 있다.

```
class Ball : public CircleShape
{
    ...
}
```

`Ball(float x, float y)` 생성자는 외부로부터 공의 위치를 받아서 객체를 생성한다. 객체가 생성되면서 부모 클래스의 생성자인 `CircleShape(12.0)`이 호출된다. 12.0은 공의 반지름이다.

```
Ball(float x, float y) : CircleShape(12.0)
{
    setPosition(x, y); // 공의 위치 설정
    setFillColor(Color(255,128,0)); // 공의 색상 설정
    setOrigin(0, 0); // 공의 기준점 설정
}
```

`update()` 함수는 공의 현재 위치를 이동한다. 공이 벽에 부딪치면 공의 방향을 변경하는 코드도 가지고 있다.

```
void update()
{
    move(speedx, speedy); // 공을 이동시킨다.
    if ((getPosition().x) < 0) // 공의 왼쪽 벽에 부딪치면
        speedx = BALL_SPEED; // 공의 x방향 속도를 양수로 한다.
    else if ((getPosition().x + 2 * 20) > 800) // 공의 오른쪽 벽에 부딪치면
        speedx = -BALL_SPEED; // 공의 x방향 속도를 음수로 한다.
    if (getPosition().y < 0) // 공이 위쪽벽에 부딪치면
        speedy = BALL_SPEED; // 공의 y방향 속도를 양수로 한다.
    else if ((getPosition().y + 2 * 20) > 600) // 공이 아래쪽벽에 부딪치면
        speedy = -BALL_SPEED; // 공의 y방향 속도를 음수로 한다.
}
```

현재까지의 전체 소스는 다음과 같다.

```
#include <SFML/Graphics.hpp>

using namespace std;
using namespace sf;
const float BALL_SPEED = 5.0;
```

```

class Ball : public CircleShape
{
public:
    float speedx = BALL_SPEED, speedy = BALL_SPEED;
    Ball(float x, float y) : CircleShape(12.0)
    {
        setPosition(x, y);
        setFillColor(Color(255,128,0));
        setOrigin(0, 0);
    }
    void update();
};

void Ball::update()
{
    move(speedx, speedy);
    if ((getPosition().x) < 0)
        speedx = BALL_SPEED;
    else if ((getPosition().x + 2 * 20) > 800)
        speedx = -BALL_SPEED;
    if (getPosition().y < 0)
        speedy = BALL_SPEED;
    else if ((getPosition().y + 2 * 20) > 600)
        speedy = -BALL_SPEED;
}

int main()
{
    Ball ball ={ 800.0 / 2, 600.0 / 2 };

    RenderWindow window(VideoMode(800, 600), "My window");
    window.setFramerateLimit(60);

    while (window.isOpen())
    {
        window.clear(sf::Color::Blue);

        Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }
    }
}

```

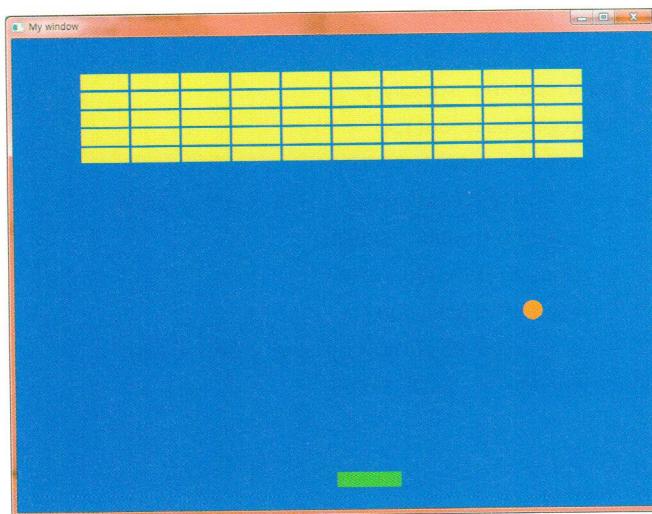
```

        ball.update();
        window.draw(ball);
        window.display();
    }
    return 0;
}

```

Step #3 패들과 벽돌을 화면에 그려보자.

패들과 벽돌을 화면에 그려보자. 패들과 벽돌은 모두 사각형이므로 RectangleShape에서 상속받자.



패들을 나타내는 Paddle 클래스를 살펴보자. 멤버 변수로는 패들의 초기 위치를 나타내는 init_x와 init_y가 추가되었다. 이들은 모두 실수형으로 정의된다. OpenGL에서는 모든 좌표가 기본적으로 실수이기 때문이다.

```

class Paddle : public RectangleShape
{
    float init_x, init_y;
public:
    Paddle(float x, float y): init_x(x), init_y(y)
    {
        setSize({ 80.0, 20.0 });           // 사각형의 크기 설정
        setPosition(x, y);               // 사각형의 위치 설정
        setFillColor(Color(0,255, 64));   // 사각형의 색상 설정
        setOrigin(0, 0);                 // 사각형의 기준점 설정
    }
}

```

`update()` 함수는 패들의 위치를 설정하는데 사용된다. 패들은 x방향으로만 움직이므로 마우스에서 위치를 받아서 사각형의 위치를 설정한다.

```
void update(int x)
{
    setPosition(x, init_y);
}
};
```

벽돌을 나타내는 `Brick` 클래스도 `RectangleShape`을 상속받아서 작성된다. 멤버 변수로는 `deleted`가 있는데 공이 벽돌을 맞추는 경우, 벽돌이 소멸되어야 하기 때문이다.

```
class Brick : public RectangleShape
{
public:
    bool deleted = false;
    Brick(float x, float y)
    {
        setSize({ 60.0, 20.0 });
        setPosition(x, y);
        setFillColor(Color::Yellow);
        setOrigin(0, 0);
    }
};
```

중요한 코드는 바로 벽돌을 여러 개 생성하는 코드이다. 이 책에서 강조하였지만 동적 배열인 벡터(`vector`)를 사용하는 것이 좋다. `main()` 함수에서 `Brick` 객체를 여러 개 생성하여서 `push_back()` 함수를 이용하여 벡터에 추가한다. 각 `Brick` 객체의 위치는 조금씩 다르게 된다.

```
vector<Brick> bricks;
for (int x=0; x < 10; x++)
    for (int y=0; y < 5; y++)
        bricks.push_back( Brick( x*(60+3)+20, y*(20+3)+40));
```

벡터에서 각 `Brick` 객체를 꺼내서 화면에 그릴 때는 범위-기반 반복 루프를 사용해보자. `auto` 키워드도 사용한다.

```
for (auto& brick : bricks)
    window.draw(brick);
```

현재까지의 전체 소스는 다음과 같다.

```
#include <SFML/Graphics.hpp>

using namespace std;
using namespace sf;
const float BALL_SPEED = 5.0;

class Paddle : public RectangleShape
{
    float init_x, init_y;
public:
    Paddle(float x, float y): init_x(x), init_y(y)
    {
        setSize({ 80.0, 20.0 });
        setPosition(x, y);
        setFillColor(Color(0,255, 64));
        setOrigin(0, 0);
    }
    void update(int x)
    {
        setPosition(x, init_y);
    }
};

class Ball : public CircleShape
{
public:
    float speedx = BALL_SPEED, speedy = BALL_SPEED;
    Ball(float x, float y) : CircleShape(12.0)
    {
        setPosition(x, y);
        setFillColor(Color(255,128,0));
        setOrigin(0, 0);

    }
    void update();
};

void Ball::update()
{
    move(speedx, speedy);
    if ((getPosition().x) < 0)
        speedx = BALL_SPEED;
```

```

        else if ((getPosition().x + 2 * 20) > 800)
            speedx = -BALL_SPEED;
        if (getPosition().y < 0)
            speedy = BALL_SPEED;
        else if ((getPosition().y + 2 * 20) > 600)
            speedy = -BALL_SPEED;
    }

    class Brick : public RectangleShape
    {
public:
    bool deleted = false;
    Brick(float x, float y)
    {
        setSize({ 60.0, 20.0 });
        setPosition(x, y);
        setFillColor(Color::Yellow);
        setOrigin(0, 0);
    }
};

int main()
{
    Ball ball ={ 800.0 / 2, 600.0 / 2 };
    Paddle paddle ={ 800.0 / 2, 550.0 };
    vector<Brick> bricks;

    for (int x=0; x < 10; x++)
        for (int y=0; y < 5; y++)
            bricks.push_back( Brick( x*(60+3)+20, y*(20+3)+40));

    RenderWindow window(VideoMode(800, 600), "My window");
    window.setFramerateLimit(60);

    while (window.isOpen())
    {
        window.clear(sf::Color::Blue);

        Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }
    }
}

```

```

    }

    ball.update();

    window.draw(ball);
    window.draw(paddle);

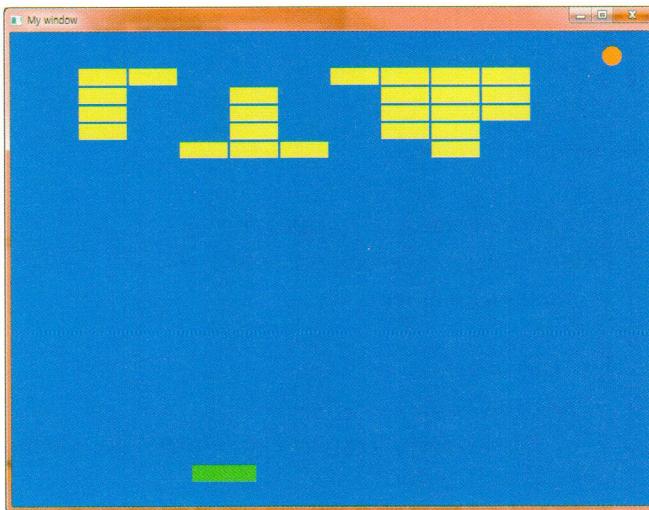
    for (auto& brick : bricks)
        window.draw(brick);

    window.display();
}

return 0;
}

```

Step #4 패들을 움직이고 충돌을 처리하자.



지금까지는 화면에 그림만 그렸다. 여기서는 패들을 마우스에 따라서 움직이고 공이 패들에서 반사되게 하며, 공이 벽돌이 부딪치면 벽돌이 소멸되도록 하자.

패들이 마우스를 따라서 움직이게 하는 것은 아주 쉽다. 마우스의 좌표는 다음과 같은 코드로 얻을 수 있고 패들의 update() 함수를 호출해주면 된다.

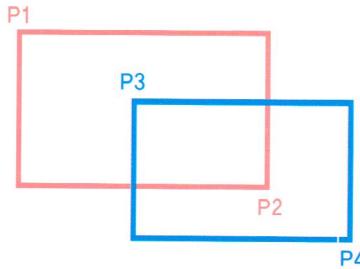
```

sf::Vector2i position = sf::Mouse::getPosition(window);
paddle.update(position.x);

```

공이 패들에 반사되게 하려면 공이 패들과 충돌하였는지를 검사하여야 한다. 게임에서

충돌 검사는 아주 중요한 부분으로 공을 감싸는 사각형과 패들을 감싸는 사각형이 겹치는지를 검사하면 된다. 우리가 그냥 작성해도 그렇게 어렵지는 않다. 여기서는 SFML이 제공하는 함수를 이용하자.



Ball 클래스의 멤버 함수로 `isIntersecting()`을 작성한다. `isIntersecting()`은 공 객체와 패들 객체가 충돌하였는지 `getGlobalBounds().intersects()`을 호출하여 알아낸다. `getGlobalBounds()` 함수는 공을 감싸는 사각형 객체를 반환하고 이 객체가 가지는 `intersects()`를 호출하면 사각형이 겹치는지를 알 수 있다.

```
bool isIntersecting(Paddle& paddle)
{
    return getGlobalBounds().intersects(paddle.getGlobalBounds());
}
```

공이 패들과 충돌하면 반사되어야 한다. 이것은 다음과 같은 Ball 클래스의 멤버 함수로 구현된다.

```
void handleCollision(Paddle& paddle)
{
    if (!isIntersecting(paddle)) return;
    speedy = -BALL_SPEED;
    if (getPosition().x < paddle.getPosition().x)
        speedx = -BALL_SPEED;
    else
        speedx = BALL_SPEED;
}
```

공과 패들이 충돌하면 그냥 볼의 속도를 음수로 만들면 공이 위쪽으로 방향을 변경한다. 공의 x좌표와 패들의 x좌표를 비교하여 공의 x축 방향도 변경한다.

공이 벽들과 충돌하면 벽돌 객체는 삭제되어야 한다. 이것을 위해서 벡터 안의 모든 벽돌 객체와 공과의 충돌을 검사한다. 충돌이 발생한 벽돌 객체는 `deleted` 변수가 `true`로 설정된다.

```
for (auto& brick : bricks)
    brick.handleCollision(ball);
```

이어서 STL 알고리즘의 `remove_if()`와 `erase()` 함수를 사용한다. `remove_if()` 함수는 조건을 만족하는 객체들을 표시하고 `erase()` 함수는 이들 객체를 벡터에서 삭제한다.

```
bricks.erase(remove_if(begin(bricks), end(bricks),
    [](&Brick& b)
{
    return b.deleted;
}), end(bricks));
```

조건을 표시하기 위하여 람다식이 사용되었다. 람다식은 임시 함수 객체를 생성하는 기법이다. `[]`가 나오고 뒤에 매개 변수가 나온다. 여기서의 매개 변수는 `b`이다. 그리고 함수 몸체가 나온다. 여기서는 `b.deleted`를 반환하는 부분이 함수 몸체이다. 따라서 벡터 안의 객체 중에서 `deleted`가 `true`인 객체들만 추출되고 이들 객체가 `erase()`에 의하여 벡터에서 삭제된다.

전체 소스는 다음과 같다.

```
#include <SFML/Graphics.hpp>

using namespace std;
using namespace sf;
const float BALL_SPEED = 5.0;

class Paddle : public RectangleShape
{
    float init_x, init_y;
public:
    Paddle(float x, float y): init_x(x), init_y(y)
    {
        setSize({ 80.0, 20.0 });
        setPosition(x, y);
        setFillColor(Color(0,255, 64));
        setOrigin(0, 0);
    }
    void update(int x)
    {
        setPosition(x, init_y);
    }
}
```

```

};

class Ball : public CircleShape
{
public:
    float speedx = BALL_SPEED, speedy = BALL_SPEED;
    Ball(float x, float y) : CircleShape(12.0)
    {
        setPosition(x, y);
        setFillColor(Color(255,128,0));
        setOrigin(0, 0);

    }
    void update();
    bool isIntersecting(Paddle& paddle);
    void handleCollision(Paddle& paddle);
};

void Ball::update()
{
    move(speedx, speedy);
    if ((getPosition().x) < 0)
        speedx = BALL_SPEED;
    else if ((getPosition().x + 2 * 20) > 800)
        speedx = -BALL_SPEED;
    if (getPosition().y < 0)
        speedy = BALL_SPEED;
    else if ((getPosition().y + 2 * 20) > 600)
        speedy = -BALL_SPEED;
}
bool Ball::isIntersecting(Paddle& paddle)
{
    return getGlobalBounds().intersects(paddle.getGlobalBounds());
}
void Ball::handleCollision(Paddle& paddle)
{
    if (!isIntersecting(paddle)) return;
    speedy = -BALL_SPEED;
    if (getPosition().x < paddle.getPosition().x)
        speedx = -BALL_SPEED;
    else
        speedx = BALL_SPEED;
}

```

```

class Brick : public RectangleShape
{
public:
    bool deleted = false;
    Brick(float x, float y)
    {
        setSize({ 60.0, 20.0 });
        setPosition(x, y);
        setFillColor(Color::Yellow);
        setOrigin(0, 0);
    }
    bool isIntersecting(Ball& ball)
    {
        return getGlobalBounds().intersects(ball.getGlobalBounds());
    }
    void handleCollision(Ball& ball)
    {
        if (!isIntersecting(ball)) return;
        deleted = true;
    }
};

int main()
{
    Ball ball ={ 800.0 / 2, 600.0 / 2 };
    Paddle paddle ={ 800.0 / 2, 550.0 };
    vector<Brick> bricks;

    for (int x=0; x < 10; x++)
        for (int y=0; y < 5; y++)
            bricks.push_back( Brick(x*(60+3)+20, y*(20+3)+40));

    RenderWindow window(VideoMode(800, 600), "My window");
    window.setFramerateLimit(60);

    while (window.isOpen())
    {
        window.clear(sf::Color::Blue);

        Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)

```

```
        window.close();
    }
sf::Vector2i position = sf::Mouse::getPosition(window);
// window is a sf::Window
paddle.update(position.x);

ball.update();
ball.handleCollision(paddle);
for (auto& brick : bricks)
    brick.handleCollision(ball);
bricks.erase(remove_if(begin(bricks), end(bricks),
    [](&Brick& b)
{
    return b.deleted;
}), end(bricks));

window.draw(ball);
window.draw(paddle);

for (auto& brick : bricks)
    window.draw(brick);

window.display();
}
return 0;
}
```




부록

SFML 기초

여기서는 `sfml`을 사용하기 위한 기초적인 내용만을 다룬다. 보다 자세한 내용은 `sfml` 튜토리얼 (<https://www.sfml-dev.org/tutorials/2.4/>)을 참조하여야 한다. 여기 내용도 위의 튜토리얼을 요약한 것이다.

SFML 윈도우 열기 및 관리

SFML의 윈도우는 `Window` 클래스에 의해 정의된다. `Window` 클래스의 생성자를 호출하면 윈도우가 생성되고 화면에 나타난다.

```
#include <SFML/Window.hpp>

int main()
{
    sf::Window window(sf::VideoMode(800, 600), "My window");
    ...
    return 0;
}
```

생성자의 첫 번째 인수인 비디오 모드는 윈도우의 크기(제목 표시 줄과 테두리를 제외한 내부 크기)를 정의한다. 여기에서는 800x600 픽셀 크기의 윈도우를 만든다. 생성자의 두 번째 인수는 단순히 윈도우 제목이다. 클래스 앞에서는 이름 공간 지정자인 `sf`를 붙이는 것이 좋다.

이벤트 처리하기

아무것도 없는 앞의 코드를 실행하려고하면 우리는 화면에서 아무 것도 볼 수 없다. 첫째, 프로그램이 즉시 종료되기 때문이다. 둘째, 이벤트 처리가 없으므로 이 코드에 무한루프를 추가하더라도 이동하지 못하고 크기를 조정하거나 닫을 수 없는 윈도우를 보게 된다.

이 프로그램을 좀 더 재미있게 만들 수 있는 몇 가지 코드를 추가해 보자.

```
#include <SFML/Window.hpp>

int main()
{
    sf::Window window(sf::VideoMode(800, 600), "My window");

    // 윈도우가 오픈되어 있는 한 프로그램을 실행한다.
    while (window.isOpen())
    {
        // 윈도우의 이벤트를 처리한다.
    }
}
```

```

sf::Event event;
while (window.pollEvent(event))
{
    // "close" 이벤트를 처리한다.
    if (event.type == sf::Event::Closed)
        window.close();
}

return 0;
}

```

위의 코드는 윈도우를 하나 만들고 사용자가 윈도우를 닫을 때 종료된다. 어떻게 작동하는지 살펴보자. 먼저, 윈도우를 닫을 때까지 애플리케이션을 새로 고치거나 업데이트하는 루프를 추가했다. 대부분의 `sfml` 프로그램은 이런 종류의 루프를 가지며 메인 루프 또는 게임 루프(game loop)라고 불린다.

게임 루프 내에서 제일 먼저 하는 일은 발생한 모든 이벤트를 확인하는 것이다. 보통 `pollEvent()` 함수는 이벤트가 처리될 수 있도록 `while` 루프를 사용한다. `pollEvent()` 함수는 이벤트가 처리대기 중이면 `true`를 반환하고 그렇지 않으면 `false`를 반환한다.

이벤트가 생길 때마다 우리는 이벤트 타입(윈도우가 닫혀 있는지, 키가 눌려 졌는지, 마우스가 움직였는지, 조이스틱이 연결된 상태인지 ...)를 확인하고, 관심이 있다면 그에 따라 반응해야 한다. 여기서는 사용자가 윈도우를 닫을 때 발생하는 이벤트 `Event::Closed`에만 신경쓰고 있다. `Closed` 이벤트가 발생한 시점에서 윈도우가 열려 있다면 `close()` 함수를 사용하여 명시적으로 윈도우를 닫아야 한다. 이렇게 하면 애플리케이션의 현재 상태 저장이나 메시지 표시와 같이 윈도우를 닫기 전에 해야 하는 작업을 수행할 수 있다. 윈도우를 닫은 후에는 메인 루프가 종료되고 프로그램이 종료된다.

우리는 아직 화면에 아무 것도 그리지 않았다. 화면에 무엇인가를 그리는 작업은 나중에 설명된다.

키보드 상태 확인하기

여기서는 키보드, 마우스 및 조이스틱과 같은 입력 장치의 상태를 확인하는 방법에 대해 설명한다. 이것은 이벤트와 혼동되어서는 안 된다. 우리는 언제든지 키보드나 마우스의 상태를 조회할 수 있다. 키보드 상태에 대한 정보를 제공하는 클래스는 `sf::Keyboard`이다. 키보드의 현재 상태(키를 누르거나 해제한 상태)를 검사하는 하나의 함수 `isKeyPressed()`만 포함되어 있다. 정적 함수이므로 인스턴스화 할 필요가 없다.

`isKeyPressed()` 함수는 윈도우의 포커스 상태를 무시하고 직접 키보드 상태를 읽는

다. 이것은 윈도우가 비활성인 경우에도 `isKeyPressed()`가 `true`를 반환할 수 있음을 의미한다.

```
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left))
{
    // 왼쪽 화살표키가 눌려있으면 주인공 캐릭터를 이동한다.
    character.move(1, 0);
}
```

마우스 상태 확인하기

마우스 상태에 대한 액세스를 제공하는 클래스는 `sf::Mouse`이다. `sf::Keyboard`처럼 `sf::Mouse` 클래스도 정적 함수 `isButtonPressed()`와 같은 정적 함수만을 포함하고 있다. 다음 코드로 버튼이 눌러져 있는지 확인할 수 있다.

```
if (sf::Mouse::isButtonPressed(sf::Mouse::Left))
{
    // 왼쪽 마우스 버튼이 눌려있으면 총알을 발사한다.
    gun.fire();
}
```

마우스 버튼 코드는 열거형 `sf::Mouse::Button`에 정의된다. SFML은 왼쪽, 오른쪽, 가운데 버튼을 지원한다.

바탕 화면 또는 윈도우에 상대적인 마우스의 현재 위치를 가져 올 수 있다.

```
// 바탕 화면에 상대적인 마우스 위치를 가져온다.
sf::Vector2i globalPosition = sf::Mouse::getPosition();

// 윈도우에 상대적인 마우스 위치를 가져온다.
sf::Vector2i localPosition = sf::Mouse::getPosition(window);

// 바탕 화면에 상대적으로 마우스 위치를 설정한다.
sf::Mouse::setPosition(sf::Vector2i(10, 50));

// 윈도우에 상대적으로 마우스 위치를 설정한다.
sf::Mouse::setPosition(sf::Vector2i(10, 50), window);
```

SFML에서의 그리기

SFML에서 무엇인가를 그리려면 그래픽 모듈을 사용한다. 그래픽 모듈이 제공하는 2차원 도형을 그리려면 특수한 윈도우 클래스인 `sf::RenderWindow`를 사용해야 한다. 이 클래스는 `sf::Window`에서 파생되며 모든 함수를 상속한다. `sf::RenderWindow`은 쉽게

도형을 그리는 데 도움이 되는 높은 수준의 함수를 추가한다. `clear()` 함수는 전체 윈도우를 선택한 색상으로 지우고, `draw()` 함수는 지정된 도형을 화면에 그린다.

```
#include <SFML/Graphics.hpp>

int main()
{
    // 윈도우를 생성한다.
    sf::RenderWindow window(sf::VideoMode(800, 600), "My window");

    // 윈도우가 오픈되어 있는 한 프로그램을 실행한다.
    while (window.isOpen())
    {
        // 윈도우의 이벤트를 처리한다.
        sf::Event event;
        while (window.pollEvent(event))
        {
            // "close" 이벤트를 처리한다.
            if (event.type == sf::Event::Closed)
                window.close();
        }

        // 윈도우를 배경색으로 지운다.
        window.clear(sf::Color::Black);

        // 여기서 무엇인가를 그린다.
        window.draw(...);

        // 현재 프레임을 종료한다.
        window.display();
    }

    return 0;
}
```

무언가를 그리기 전에 `clear()`를 호출하는 것은 필수적이다. 그렇지 않으면 이전 프레임의 내용이 그대로 화면에 표시된다.

`display()`를 호출하는 것도 필수적이며 마지막 호출 이후에 그려진 내용을 가져와 윈도우에 표시한다. 사실, SFML에서는 도형들이 윈도우에 직접적으로 그려지는 것이 아니라 숨겨진 버퍼 메모리에 그려진다. 이 버퍼 메모리는 `display()`를 호출할 때 윈도우에 복사된다. 이러한 기술을 이중 버퍼링이라고 한다. 이제 그릴 준비가 된 메인 루프

가 생겼으니, 무엇을 실제로 그릴 수 있는지 살펴보자. SFML은 4가지 종류의 도형을 제공한다. 그 중 3가지(스프라이트, 텍스트, 도형)는 이미 사용할 준비가 되었다.

스프라이트와 텍스처

텍스처는 이미지이다. 그러나 2차원 도형에 매핑되는 역할을 하기 때문에 일반적으로 “텍스처”라고 부른다. 스프라이트(sprite)는 텍스처가 있는 직사각형에 지나지 않는다.



스프라이트를 만들기 전에 유효한 텍스처가 필요하다. SFML에서 텍스처를 캡슐화하는 클래스가 `sf::Texture`이다. 텍스처의 유일한 역할은 텍스처를 적재하고 업데이트하는 것이다.

텍스처를 적재하는 가장 일반적인 방법은 디스크의 이미지 파일에서 적재하는 것으로, `loadFromFile()`을 호출한다.

```
sf::Texture texture;
if (!texture.loadFromFile("image.png"))
{
    // error...
}
```

`loadFromImage()`은 이미지 데이터를 저장하고 조작하는 데 도움이 되는 유ти리티 클래스 `sf::Image`에서 텍스처를 적재한다. SFML은 가장 일반적인 이미지 파일 형식을 지원한다.

이 모든 로딩 함수는 선택적 인수를 가지며 이미지의 더 작은 부분을 적재하려는 경우 사용할 수 있다.

```
// (10, 10)에서 시작하는 32x32 사각형만을 적재한다.
if (!texture.loadFromFile("image.png", sf::IntRect(10, 10, 32, 32)))
{
    // error...
}
```

이 `sf::IntRect` 클래스는 사각형을 나타내는 간단한 유ти리티 유형이다. 생성자는 원

쪽 상단의 좌표와 사각형의 크기를 받는다. 텍스처가 만들어지면 스프라이트를 만들 수 있다.

```
sf::Sprite sprite;
sprite.setTexture(texture);
```

스프라이트를 화면에 그리는 코드는 다음과 같다.

```
// 메인 루프 안에 window.clear()와 window.display() 사이에 있어야 한다.
window.draw(sprite);
```

흰색 사각형 문제

성공적으로 텍스처를 적재하고 스프라이트를 올바르게 구성했으며 화면에는 흰색 사각형밖에 보이지 않는다. 어떻게 된 것일까?

이것은 흔한 실수이다. 스프라이트의 텍스처를 설정할 때 스프라이트는 텍스처 객체에 대한 포인터만을 저장한다. 따라서 텍스처가 파괴되거나 메모리의 다른 곳으로 이동하면 스프라이트가 유효하지 않은 텍스처 포인터를 사용하게 된다. 예를 들어서 다음과 같이 코드를 작성하면 이러한 문제가 발생한다.

```
sf::Sprite loadSprite(std::string filename)
{
    sf::Texture texture;
    texture.loadFromFile(filename);

    return sf::Sprite(texture);
} // 텍스처 객체가 여기서 파괴되기 때문에 오류이다.
```

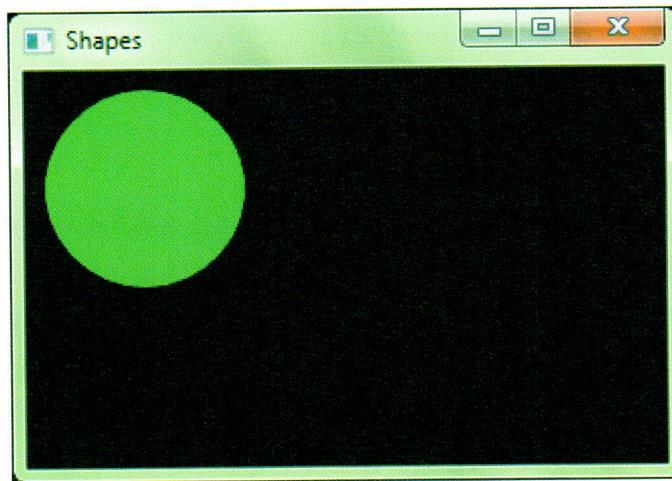
텍스처 객체의 수명을 정확하게 관리하고 스프라이트에 의해 사용되는 텍스처 객체들은 반드시 살아 있는지를 확인해야 한다. 또 가능한 한 적은 개수의 텍스처를 사용하는 것이 좋은 전략이며 그 이유는 간단하다. 텍스처를 변경하는 것은 비용이 많이 드는 작업이기 때문이다. 동일한 텍스처를 많이 사용하면 최상의 성능을 얻을 수 있다.

원 그리기

이제부터는 도형을 그리는 방법을 살펴보자. 원은 `sf::CircleShape` 클래스에 의해 표현된다. 반지름이 50인 원을 그리는 코드는 다음과 같다. 도형의 색상은 `setFillColor()`로 변경할 수 있다.

```
sf::CircleShape shape(50);

// 도형의 색상을 녹색으로 변경한다.
shape.setFillColor(sf::Color(100, 250, 50));
shape.setRadius(50);
window.draw(shape);
```



도형 객체가 생성되고 나서 화면에 도형을 그리려면 `window.draw(shape);`을 호출 한다.

사각형 그리기

사각형을 그리려면 `sf::RectangleShape` 클래스를 사용할 수 있다.

```
// 120x50 사각형을 정의한다.
sf::RectangleShape rectangle(sf::Vector2f(120, 50));

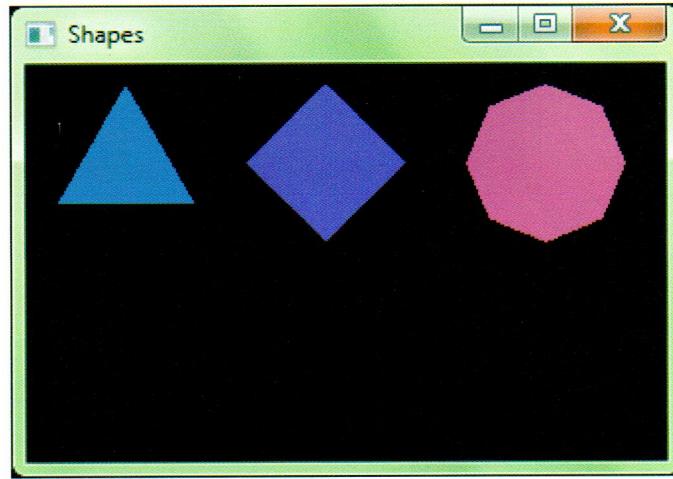
// 크기를 100x100으로 변경한다.
rectangle.setSize(sf::Vector2f(100, 100));
```

정다각형

실제로는 정다각형을 위한 전용 클래스가 없다. 하지만 `sf::CircleShape` 클래스는 두 번째 인수로 점들의 개수를 받는다. 이것을 이용하여 다각형을 그릴 수 있다. `sf::CircleShape(100, 3)`은 삼각형이고, `sf::CircleShape(100, 4)`는 사각형이다.

```
// 삼각형을 정의한다.
sf::CircleShape triangle(80, 3);
```

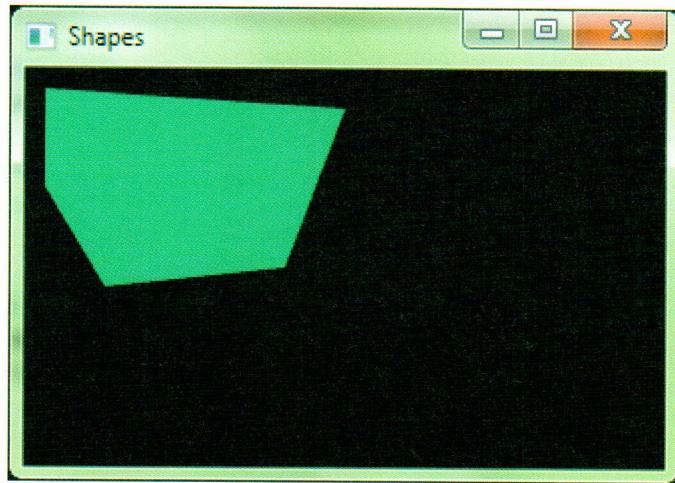
```
// 사각형을 정의한다.  
sf::CircleShape square(80, 4);  
  
// 팔각형을 정의한다.  
sf::CircleShape octagon(80, 8);
```



볼록다각형

볼록한 모양을 만들려면 먼저 필요한 점의 수를 설정한 다음 점을 정의해야 한다.

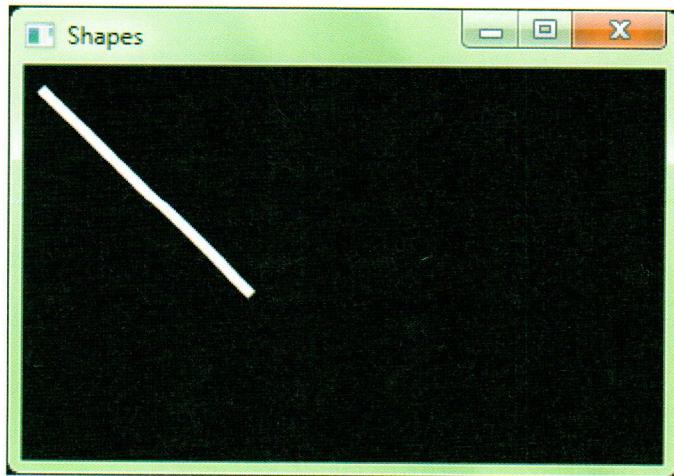
```
// 비어 있는 도형을 만든다.  
sf::ConvexShape convex;  
  
// 5개의 점으로 이루어진다.  
convex.setPointCount(5);  
  
// 점들을 정의한다.  
convex.setPoint(0, sf::Vector2f(0, 0));  
convex.setPoint(1, sf::Vector2f(150, 10));  
convex.setPoint(2, sf::Vector2f(120, 90));  
convex.setPoint(3, sf::Vector2f(30, 100));  
convex.setPoint(4, sf::Vector2f(0, 50));
```



선 그리기

두께가 있는 선은 사각형 객체로 그릴 수 있다.

```
sf::RectangleShape line(sf::Vector2f(150, 5));
line.rotate(45);
```



글꼴 불러 오기

텍스트를 그리기 전에 사용 가능한 글꼴을 가져와야 한다. 글꼴은 `sf::Font` 클래스에 캡슐화되어 글꼴을 적재하고, 글꼴을 가져오며, 속성 읽기와 같은 세 가지 주요 기능을 제공한다. 글꼴을 적재하는 가장 일반적인 방법은 `loadFromFile()` 함수를 호출하여 디스크의 파일에서 읽는 것이다.

```
sf::Font font;
if (!font.loadFromFile("arial.ttf"))
{
    // 오류
}
```

SFML은 자동으로 시스템 글꼴을 적재하지 않으므로 `font.loadFromFile("Courier New")`은 작동하지 않는다. 그 이유는 첫째, SFML은 글꼴 이름이 아닌 파일 이름을 필요로 하기 때문이다. 둘째, SFML은 시스템의 글꼴 폴더에 대한 액세스 권한이 없기 때문이다. 글꼴을 적재하려면 다른 모든 리소스 (이미지, 사운드, ...)와 마찬가지로 글꼴 파일을 애플리케이션에 포함시켜야 한다.

텍스트 그리기

텍스트를 그리려면 `sf::Text` 클래스를 사용하면 된다.

```
sf::Text text;

// 폰트를 선택한다.
text.setFont(font);

// 출력할 문자열을 지정한다.
text.setString("Hello world");

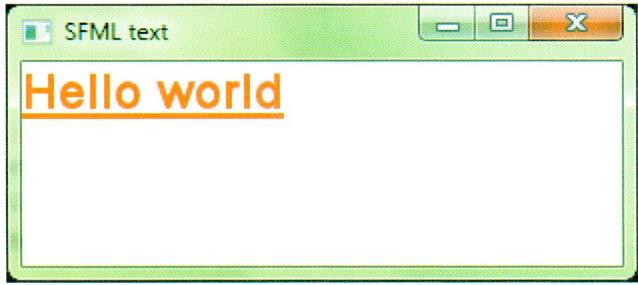
// 글자 크기를 설정한다.
text.setCharacterSize(24); // 포인트가 아닌 픽셀 크기

// 색상을 지정한다.
text.setFillColor(sf::Color::Red);

// 텍스트 스타일을 설정한다.
text.setStyle(sf::Text::Bold | sf::Text::Underlined);

...

// 메인 루프 안에서 window.clear()와 window.display() 사이에서 그린다.
window.draw(text);
```



한글 출력하기

한글과 같은 비 아스키 문자를 올바르게 처리하는 것은 까다로울 수 있다. 이 때는 문자열 앞에 L을 붙이면 된다.

```
text.setString(L"한글");
```

SFML에서의 시간 측정

SFML에서는 시간을 sf::Time 클래스로 나타낸다. Time 클래스는 날짜를 나타내는 것 이 아니라 어떤 시점에서의 시각을 저장한다. 예를 들어서 경과된 시간을 측정하는 코드는 다음과 같다.

거의 모든 프로그램에서 필요한 것을 수행하는 방법을 살펴보자. 경과한 시간을 측정 한다. SFML은 시간 측정을 위해 매우 간단한 클래스 sf::Clock를 가지고 있는데 즉, 두 가지 기능만 가지고 있다 : getElapsedTime()은 시계가 시작된 아래 경과된 시간을 검색하고 restart()는 시계를 다시 시작한다.

```
sf::Clock clock; // 시계를 시작한다.  
...  
sf::Time elapsed1 = clock.getElapsedTime();  
std::cout << elapsed1.asSeconds() << std::endl;  
clock.restart();  
...  
sf::Time elapsed2 = clock.getElapsedTime();  
std::cout << elapsed2.asSeconds() << std::endl;
```

다음은 게임 루프를 반복할 때마다 경과된 시간을 사용하여 게임 로직을 업데이트하는 예제이다.

```
sf::Clock clock;  
while (window.isOpen())  
{  
    sf::Time elapsed = clock.restart();
```

```
    updateGame(elapsed);
    ...
}
```