آموزش مقدماتی برنامه نویسی بازی

با استفاده از SFML

میلکو جی. میلچیو

به کوشش : عباسعلی طهماسبی

آموزش مقدماتی برنامه نویسی بازی

با استفاده از *SFML*

نویسنده : میلکو جی . میلچیو

به کوشش : عباسعلی طهماسبی

بازی ساز – شهریور ۱۳۹٤

www.bazi-dev.ir

@bazi_dev

درباره نویسنده

میلکو جی . میلچیو از زمانیکه دبیرستانی بود ، یک برنامه نویس بازی بود . او هر روز روی بازی ها کار می کند . بعضی از بازی هائی که او روی آنها کار کرده *Unholy Flesh* و *WhiteCity* هستند . در حال حاضر برای Dream Harvest کار می کند که یک استودیو بازی سازی مستقل همراه با تعداد زیادی از نیروهای با استعداد است . آنها اولین بازی استراتژی خودشان را ساخته اند . میلکو همچنین توسعه SFML را چندسال دنبال کرده و موتور بازی خودش را بر پایه این کتابخانه ، ساخته است .

این کتاب چه چیزهائی را پوشش می دهد

فصل ۱ : شروع کار با SFML – فصلی مقدماتی درباره کتابخانه SFML است . این فصل فرایند ایجاد پنجره و رندر اشکال پایه را بر روی صفحه ، توضیح می دهد . فصل با یک بازی کوچک به پایان می رسد که کدهای آن همراه با جزئیات ، توضیح داده خواهد شد .

فصل ۲ : بارگذاری و استفاده از بافت ها – مفهوم sprite و بافت و تعامل آنها با پنجره را معرفی می کند . در پایان این فصل ، مشکل مدیریت منابع بازی با ساختن یک مدیریت کننده منبع قوی ، حل خواهد شد .

فصل ۳ : انیمیت کردن Sprite ها – در انتهای فصل ، یک کلاس انیمیت کننده کامل ساخته خواهد شد . این کلاس را می توان در خارج از این کتاب هم استفاده کرد .

فصل ٤ : مدیریت یک دوربین دوبعدی – مفهوم دوربین ها را در صحنه معرفی کرده و شیوه های تعامل با آن را بررسی می کند . این فصل ، رندرینگ مستقیم با استفاده از OpenGL بر روی یک پنجره را هم بحث می کند .

فصل ٥ : اکتشاف دنیای صداها و متون – اجزای متن و صدای بازی را بحث می کند . مفهوم صداهای سه بعدی هم بررسی خواهد شد .

فصل ٦ : رندر افکت های ویژه با استفاده از شیدرها – به موضوع شیدرها و استفاده آنها در افکت های ویژه می پردازد . افکت های پس پردازش هم به صورت مختصر با ارائه یک مثال از شیدر پیکسلی کردن ، توضیح داده خواهد شد .

چه چیزی برای این کتاب نیاز است

برای این کتاب به Microsoft Visual Studio 2012 Express مخصوص ویندوز نیاز دارید که می توانید آن را از لینک زیر دانلود کنید :

http://www.microsoft.com/en-gb/download/details.aspx?id=34673

شما همچنین باید SFML v2.1 Visual C++ 11 (2012) 32-bit را از لینک زیر نصب کنید :

http://sfml-dev.org/download/sfml/2.1/

همچنین به یک کارت گرافیک مناسب با رزولیشن 1024 x 768 پیکسل یا بالاتر نیاز دارید که با OpenGL 2.0 سازگار باشد .

این کتاب برای چه کسی است ؟
این کتاب برای افرادی است که در زمینه برنامه نویسی بازی تجربه دارند اما می خواهند برای پروژه بعدی خود ، از SFML استفاده کنند . شاید شما هم برای یک بازی ، ایده ای دارید اما محیط فعلی شما برای برآوردن نیازهای بازی شما خیلی کند است یا اینکه یک راه حل مستقل از پلتفرم در زبان مورد علاقه خود می خواهید . در هر دو مورد ، این کتاب شما را به سرعت به سمت هدفتان ، راهنمائی می کند . ما فرض می کنیم که یک فهم ابتدائی از ساختن بازی دارید ، اما فهمی خوب در حداقل یکی از زبان های پشتیبانی شده (C++) لازم است .

فهرست مطالب

مقدمه

آموزش مقدماتی برنامه نویسی بازی با استفاده از *SFML ، مجموعه ای از* آموزش های کاربردی درباره کتابخانه Simple and Fast Multimedia Library (SFML) است که به شما یاد می دهد چطور به سرعت و به آسانی از این کتابخانه استفاده کنید . کتاب مفاهیم اصلی ساختن بازی را با فراهم کردن بهترین شیوه ها در این زمینه ، ارائه می کند .

ساختن بازی می تواند موضوعی سخت برای فهمیدن باشد . این کتاب دانش کافی درباره SFML را برای شما فراهم کرده است تا در اولین فرصت ممکن ، ایده هایتان را پیاده سازی کنید . این کتاب همچنین شامل تعدادی از مثالهاست که شما می توانید از آنها استفاده کنید و بر طبق نیازتان تغییر دهید .

فصل ۱ : شروع کار با SFML

در این فصل ، مدل های گرافیک و پنجره SFML را بررسی می کنیم و بر روی تقویت مفاهیم پایه تمرکز می کنیم . همچنین خواهیم دید که یک بازی در SFML چه شکلی است و چطور می توانیم برای اداره شکل ها روی صفحه ، ورودی را مدیریت کنیم .
در این فصل موارد زیر توضیح داده می شود :

ایجاد پنجره
حلقه بازی
مدیریت رویداد
رندر و تغییر شکل اشکال هندسی

ایجاد پنجره

هنگام شروع ساختن یک بازی ، اولین چیزی که احتمالا می خواهید انجام دهید ، بازکردن یک پنجره است . در SFML فقط یک خط کد برای ایجاد یک پنجره لازم است :

Main.cpp

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

int main()
{
    sf::Window window(sf::VideoMode(300, 200), "The title");

    return 0;
}
```

تنها چیزی که تابع main انجام می دهد ، مقداردهی اولیه متغییر `window` توسط فراخوان سازنده `sf::Window` است . یک راه دیگر برای بازکردن یک پنجره ، استفاده از سازنده پیش فرض و سپس فراخوان `Window::create()` است . این تابع دقیقا همان آرگومانهای سازنده را می خواهد که توضیح خواهیم داد . اگر روی پنجره ای که از قبل باز است `Window::create()` فراخوانده شود ، باعث بسته شدن پنجره می شود و دوباره با پارامترهای جدیدی ، مقداردهی اولیه خواهد شد .
در مثال قبل ، توجه کنید که هردو `Window` و `VideoMode` در namespace ای بانام `sf` هستند . هر کلاس موجود در SFML تحت این namespace است که کلاس های موجود در SFML را از کلاس های کتابخانه های دیگر جدا می کند .
اگر کدهای این مثال را اجراکنیم ، چیز زیادی نمی بینیم . برنامه بلافاصله پس از ایجاد پنجره ، خارج می شود . علت این است که ما فقط یک پنجره ایجاد کرده ایم بدون اینکه هیچ کاری با آن انجام دهیم . برنامه طبیعتا با رسیدن به انتهای `main()` ، خارج می شود . اینکه ما یک پنجره ایجاد کرده ایم به این معنی نیست که به طور کامل کار می کند . ما باید آن را برطبق چیزی که می خواهیم انجام دهد ، برنامه نویسی کنیم . حالا اجازه دهید با ایجاد تاخیر در نخ اجرائی پنجره ، از به پایان رسیدن تابع main جلوگیری کنیم . SFML یک واسط ساده برای آن تهیه کرده است . کافی است `sf::sleep(sf::seconds(3))` را بعد از کدی که پنجره را ایجاد می کند ، اضافه کنید . حالا پنجره به اندازه زمان داده شده ، قابل دیدن خواهد بود .
ما می توانیم در هنگام ایجاد پنجره تنظیمات متنوعی را تعیین کنیم مثل اندازه ، عنوان ، سبک و تنظیمات گرافیکی پنجره. احتمالا دقت کرده اید که ما دو آرگومان را به سازنده `Window` پاس دادیم : یک نمونه از `VideoMode` و یک رشته متنی برای عنوان پنجره .
سازنده در واقع می تواند تا ۴ پارامتر را دریافت کند اما دوتای آنها یعنی `Style` و `ContextSettings` اختیاری است . بخش های بعدی معنی این آرگومان ها و چگونگی استفاده از آنها را توضیح می دهد .

VideoMode

کلاس `VideoMode` محتوی اطلاعاتی درباره مد ویدئویی پنجره مثل عرض ، ارتفاع و تعداد بیت در هر پیکسل است . این آخرین پارامتر تعداد بیت های مورد استفاده برای نشان دادن رنگ هر پیکسل است و مقدار پیش فرض 32 دارد . به عنوان مثال مقدار 8 یک نتیجه تک رنگ تولید خواهد کرد . اگر بخواهیم یک پنجره تمام صفحه ایجاد کنیم ، مقداری که تعیین می کنیم باید توسط مانیتور و کارت گرافیک پشتیبانی شده باشد . اگر یک آرگومان نامعتبر برای یک پنجره تمام صفحه تعیین کنیم ، ایجاد پنجره با شکست روبه رو می شود . اعتبار کلاس `VideoMode` را می توان با متد `VideoMode::isValid()` بررسی کرد که یک مقدار بولی به عنوان نتیجه بازگشت می دهد .

اگر بخواهیم پنجره ای بر طبق اندازه (یا عمق پیکسل) دسکتاپ ایجاد کنیم ، `VideoMode::getDesktopMode()` به عنوان یک متد استاتیک در دسترس است . اگر بخواهیم پنجره ای تمام صفحه ایجاد کنیم ، باید رزولیشن های در دسترس را با `VideoMode::getFullScreenModes()` چک کنیم . این متد یک `std::vector` از مدهای ویدئویی بازگشت می دهد که ما می توانیم یکی از آنها را انتخاب کنیم یا اینکه به کاربر اجازه دهیم که هرکدام که مناسب است را انتخاب کند .

اما ایجاد یک پنجره تمام صفحه ، به تعیین یک سبک هم نیاز دارد .

Style

آرگومان `Style` یک ماسک بیتی است . یک ماسک ترکیبی از flag هاست که هر flag نشان دهنده یک بیت خاص از ماسک است . در این نمونه flag ها در یک شمارش موجود در فضای نام `sf::Style` ذخیره شده اند . ما می توانیم با ترکیبی از flag ها ماسک دلخواه را بسازیم :

`sf::Style::None`

پنجره هیچ آذین بندی ندارد و نمی تواند با سبک های دیگر ترکیب شود .

`sf::Style::Titlebar`
این یک نوار عنوان اضافه می کند .

`sf::Style::Resize`
این یک دکمه ماکزیمایز اضافه می کند . همچنین پنجره را قادر می کند به صورت دستی دچار تغییر اندازه شود .

`sf::Style::Close`
این یک دکمه بستن را اضافه می کند .

`sf::Style::Fullscreen`
این پنجره را در مد تمام صفحه باز می کند . توجه کنید که این نمی تواند با سبک های دیگر ترکیب شود و یک مد ویدئویی معتبر نیاز دارد .

`sf::Style::Default`
این ترکیب `Titlebar`, `Resize` و `Close` است . این سبک پیش فرض است .

با استفاده از اپراتور بیتی OR می توانید سبک های مختلف را با هم ترکیب کنید . مثلا برای داشتن پنجره ای با نوار عنوان و یک دکمه بستن ، می نویسیم :

```
sf::Uint32 style = sf::Style::Titlebar | sf::Style::Close;
```

تنها چیزی که در اینجا انجام ندادیم ، پاس دادن این سبک به عنوان سومین آرگومان به سازنده Window است .

ContextSettings

آخرین آرگومان یک نمونه از ContextSettings است . این ساختار گروهی از تنظیمات است که فضای رندر مطلوب را توصیف می کند . SFML برای رندرینگ از OpenGL استفاده می کند ، بنابراین این تنظیمات مستقیما مربوط به آن است . تنظیمات در دسترس عبارتند از :

depthBits : این به تعداد بیت های بافر عمق اشاره دارد .
stencilBits : این به تعداد بیت های بافر استنسیل اشاره دارد .
antialiasingLevel : این به تعداد لایه های multisampling اشاره دارد .
majorVersion و minorVersion : این به ورژن مورد نیاز OpenGL اشاره دارد .
هر یک از این تنظیمات به صورت مفصل در فصل ۵ توضیح داده خواهد شد .

غیر فعال کردن مکان نمای ماوس

کلاس Window متدی دارد که قابلیت دیده شدن مکان نما را روشن و خاموش می کند :

```
Window::setMouseCursorVisible()
```

این برای بازی هائی مفید است که به مکان نما نیازی ندارند ، یا هنگامی که می خواهیم تصویر مکان نما را چیزی غیر از پیش فرضش قرار دهیم .

حلقه بازی

هر بازی به یک حلقه نیاز دارد . در غیر این صورت برنامه به انتها رسیده و قادر نخواهیم بود تا چیزی را ببینیم . یک حلقه بازی به شکل زیر خواهد بود :

Main.cpp

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

int main()
{
    sf::Window window(sf::VideoMode(800, 640), "The title");

    //Game loop
    while (window.isOpen())
    {
        /* Game loop stages:

        1. Handle input - handle events from input devices and the window
        2. Update frame - update objects in the scene
        3. Render frame - render objects from the scene onto the window

        */
    }

    return 0;
}
```

یک حلقه بازی ۳ مرحله دارد :

مدیریت ورودی
به روز کردن فریم
رندر فریم

مدیریت ورودی در SFML می تواند با گرفتن رویدادهائی که توسط پنجره ارسال می شوند انجام شود یا اینکه مستقیما دستگاه های ورودی را برای وضعیت حال حاضرشان ، استفسار کرد . هر دو شیوه استفاده های متفاوتی دارد . مثلا ، ممکن است بخواهیم در هنگامی که رویداد فشرده شدن یک کلید اتفاق می افتد ، پنجره را ببندیم یا اینکه تا زمانیکه یک کلید فشرده می شود ، کاراکتر اصلی بازی را به سمت راست حرکت دهیم (استفسار مستقیم کلید)

بعد از اینکه رویدادها را گرفتیم و استفاده کردیم به مرحله به روز کردن فریم می رسیم . این مرحله جائی است که باید منطق بازی را پیش ببریم و وضعیت دنیای بازی را به روز کنیم .
مرحله نهائی حلقه درست بعد از اینکه اشیاء را به روز کردیم ، می آید . در اینجا ما همه چیزی را که از آخرین بار ، ترسیم شده است پاک می کنیم و دوباره همه اشیاء به روز شده را روی صفحه رندر می کنیم .

به مثال حلقه بازی خودمان برگردیم ، این حلقه فعلا کاری نمی کند و اگر کدها را اجرا کنیم معلوم است که پنجره به ورودی واکنشی نشان نمی دهد . علت این است که ما اولین مرحله حلقه بازی یعنی مدیریت ورودی را انجام نداده ایم .

مدیریت ورودی

با استفاده از `bool Window::pollEvent(sf::Event& event)` رویدادها می توانند از پنجره جمع آوری شوند . اگر رویدادی موجود باشد ، تابع `true` بازگشت می دهد ، و متغییر event با اطلاعات مربوط به رویداد پر می شود وگرنه تابع `false` بازگشت می دهد . مهم است که توجه کنیم که ممکن است در یک زمان بیش از یک رویداد وجود داشته باشد ، بنابراین باید اطمینان حاصل کنیم که همه رویدادهای ممکن را گرفته باشیم . یک حلقه رویداد به شکل زیر است :

```
while (window.isOpen())
{
    sf::Event event;
    while (window.pollEvent(event))
    {
    }

    //Update frame

    //Render frame
}
```

حالا اجرای کدها نتیجه بهتری دارد ، ما می توانیم پنجره را حرکت دهیم ، تغییر اندازه بدهیم و آن را مینیمایز کنیم . اما هنوز یک مشکل وجود دارد : دکمه بستن پنجره کار نمی کند . SFML فرض نمی داند که کاربر بعد از اینکه روی دکمه بستن کلیک کرد ، پنجره باید بسته شود . شاید ما بخواهیم پیشرفت بازی کاربر را ذخیره کنیم یا از او ابتدا بپرسیم که برای خروج از بازی مطمئن است یا خیر ؟ این یعنی ما باید خودمان عملکرد دکمه بستن پنجره را پیاده سازی کنیم .
قبل از ادامه موضوع ، توجه کنید که کلاس `Event` در C++ محتوی یک union است . یعنی فقط یکی از اعضایش معتبر است . دسترسی به اعضای دیگر ، باعث انجام نتیجه ای تعریف نشده خواهد شد . با نگاه کردن به نوع رویداد می توانیم عضو معتبر را پیدا کنیم .

انواع رویدادها می توانند به ۴ بخش تقسیم شوند :

پنجره
صفحه کلید
ماوس
جوی استیک (دسته بازی)

رویدادهای مربوط به پنجره

`Event::Closed`

هنگامی که سیستم عامل تشخیص دهد که کاربر می خواهد پنجره را ببندد (دکمه بستن پنجره و ...) این رویداد رها می شود .

`Event::Resized`

هنگامی که سیستم عامل تشخیص دهد که پنجره به صورت دستی دچار تغییر اندازه شده است یا هنگامی که `Window::setSize()` استفاده شود ، این رویداد رها می شود . `Event::size` اندازه جدید پنجره را نگه می دارد .

`Event::LostFocus`

`Event::GainedFocus`

هنگامی که پنجره فوکوس را دریافت کند یا از دست بدهد ، این رویداد رها می شود . پنجره هائی که فوکوس را از دست داده باشند ، رویدادهای صفحه کلید را دریافت نمی کنند .

رویدادهای مربوط به صفحه کلید

`Event::KeyPressed`

`Event::KeyReleased`

هنگامی که در یک پنجره دارای فوکوس ، یک دکمه فشرده شود یا رها شود ، این رویداد رها می شود . `Event::key` کلید فشرده شده/ رها شده را نگه می دارد .

`Event::TextEntered`

هر بار که یک کاراکتر تایپ شود ، این رویداد رها می شود . این یک کاراکتر قابل چاپ از ورودی کاربر فراهم می کند و برای فیلدهای متنی مفید است . `Event::text` مقدار یونیکد UTF-32 کاراکتر را نگه می دارد .

رویدادهای مربوط به ماوس

`Event::MouseMoved`

هنگامی که موقعیت ماوس داخل پنجره دچار تغییر می شود ، این رویداد رها می شود . `Event::mouseMove` موقعیت جدید ماوس را نگه می دارد .

`Event::MouseButtonPressed`

`Event::MouseButtonReleased`

هنگامی که یک دکمه ماوس داخل پنجره فشرده شود ، این رویداد رها می شود . در حال حاضر ۵ دکمه پشتیبانی می شود : چپ ، راست ، وسط ، دکمه اضافی ۱ و دکمه اضافی ۲ . `Event::mouseButton` دکمه فشرده شده / رها شده و موقعیت ماوس را نگه می دارد .

`Event::MouseWheelMoved`
هنگامی که غلطک ماوس داخل یک پنجره حرکت کند این رویداد رها می شود . `Event::mouseWheel` سطح بندی غلطک و موقعیت ماوس را نگه می دارد .

رویدادهای مربوط به جوی استیک (دسته بازی)

`Event::JoystickConnected`
`Event::JoystickDisconnected`
هنگامی که یک جوی استیک اتصال داده شود یا جدا شود این رویداد رها می شود . `Event::joystickConnect` ، ID جوی استیک را نگه می دارد .

`Event::JoystickButtonPressed`
`Event::JoystickButtonReleased`
هنگامی که یک دکمه از روی جوی استیک فشرده شود ، این رویداد رها می شود . SFML تا ۸ جوی استیک و تا ۳۲ دکمه برای هر یک را پشتیبانی می کند . `Event::joystickButton` تعداد دکمه های فشرده شده و ID جوی استیک را نگه می دارد .

`Event::JoystickMoved`
هنگامی که یک محور جوی استیک حرکت کند ، این رویداد رها می شود . حد حرکت می تواند از طریق
`Window::setJoystickThreshold()` تنظیم شود . SFML تا ۸ محور را پشتیبانی می کند .
`Event::joystickMove` محور حرکت کرده ، موقعیت جدید محور و ID جوی استیک را نگه می دارد .

استفاده از رویدادها

بعد از این که توسط فراخوان `Window::pollEvent()` رویدادها را جمع آوری کردیم ، می توانیم با نگاه کردن به `Event::type` نوع آن را چک کنیم . رویداد از نوع `Event::EventType` ، یک شمارش است که داخل کلاس `Event` قرار دارد . در زیر می بینید که چطور رویداد بسته شدن (close) مدیریت می شود :

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

در اینجا تابع `Window::close()` باعث بسته شدن پنجره می شود . اگر یک متغییر `window` به خارج از حوزه برود ، تخریب کننده فراخوانده شده و پنجره بسته می شود .
اگر بخواهیم بیش از یک رویداد را مدیریت کنیم ، منطقی است که از دستور `switch` استفاده کنیم . اجازه دهید تا ببینیم که فشرده شدن و رها شدن کلید صفحه کلید چطور مدیریت می شود :

```
sf::Event event;
while (window.pollEvent(event))
{
    switch (event.type)
    {
    case sf::Event::EventType::Closed:
        window.close();
        break;
    case sf::Event::EventType::KeyPressed:
        //Change the title if the space is pressed
        if (event.key.code == sf::Keyboard::Key::Space)
            window.setTitle("Space pressed");
        break;
    case sf::Event::EventType::KeyReleased:
        //Change the title again if space is released
        if (event.key.code == sf::Keyboard::Key::Space)
            window.setTitle("Space released");
        //Close the window if the Escape key is released
        else if (event.key.code == sf::Keyboard::Key::Escape)
            window.close();
        break;
    default:
        break;
    }
}
```

این کدها نشان می دهند که چطور می توانیم با هرباری که کلید *Space* صفحه کلید فشرده و رها می شود ، رویدادها را گرفته و از آن برای تغییر عنوان پنجره استفاده کنیم . همچنین هنگامی که کلید *Escape* رها می شود ، پنجره بسته می شود . توجه کنید که `Event::key` محتوی عضوی به نام `code` است که شمارشی از نوع `Keyboard::Key` است . شما می توانید از این فرمول برای بقیه انواع رویدادها استفاده کنید . اما `Event::EventType::TextEntered` کمی جالب تر است . مثلا اگر بخواهیم تشخیص دهیم که کی علامت *!* تایپ شده است ، باید ببینیم که آیا دو کلید *Shift + 1* همزمان فشرده شده یا نه ؟ در این نمونه ها ، SFML با فراهم کردن رویداد `TextEntered` ما را از انجام کارهای اضافه نجات داده است . این رویداد تنها هنگامی ارسال می

شود که ترکیبی از کلیدها که نشان دهنده یک کاراکتر است ، فشرده شوند ، یعنی یک کلید تنها (مثلا فقط *Shift*) ممکن است این رویداد را رها نکند . البته اگر K به تنهائی فشرده شود ، رویداد ارسال شده و محتوی کاراکتر خواهد بود .
به مثال زیر نگاه کنید . یک رشته متنی با استفاده از رویداد `TextEntered` و *هنگامی که دکمه* Enter فشرده می شود ، درست شده و برای عنوان پنجره استفاده می شود :

```
sf::String buffer;
while (window.isOpen())
{
    sf::Event event;
    while (window.pollEvent(event))
    {
        switch (event.type)
        {
        case sf::Event::EventType::Closed:
            window.close();
            break;
        case sf::Event::EventType::TextEntered:
            //Add the character directly to the string
            buffer += event.text.unicode;
            break;
        case sf::Event::EventType::KeyReleased:
            //Change the title to the current buffer and clear the buffer
            if (event.key.code == sf::Keyboard::Key::Return)
            {
                window.setTitle(buffer);
                buffer.clear();
            }
            break;
        default:
            break;
        }
    }
}
```

توجه کنید که رشته متنی از نوع `sf::String` است نه از نوع `std::string` . کلاس `sf::String` به صورت اتوماتیک تبدیلات بین انواع رشته های متنی و رمزگذاری آنها را مدیریت می کند . همچنین لازم نیست درباره زبان یا علائم روی صفحه کلید نگران باشیم ، این می تواند هر کاراکتری از هر زبانی را ذخیره کند .

برای پایان دادن به موضوع مدیریت ورودی ، مهم است که ذکر کنیم که یک شیوه دیگر برای جمع آوری رویدادها از پنجره وجود دارد . جدای استفاده از `Window::pollEvent()` می توانیم از `bool Window::waitEvent(Event& event)` هم استفاده کنیم که نخ اجرائی را تا اینکه یک رویداد دریافت شود ، بلوکه می کند . آن فقط هنگامی `false` بازگشت می دهد که چیزی اشتباه اتفاق بیافتد (یک خطا یا یک استثناء و ...) در غیر این صورت همیشه `true` بازگشت می دهد . هنگامی که نیاز داریم که کاربر قبل از اینکه برنامه ادامه پیدا کند ، کاری را انجام دهد ، یا وقتی که بخواهیم ورودی را از روی نخ اجرائی دیگری مدیریت کنیم ، این می تواند مفید واقع شود . در این مورد دومی ، فقط آن نخ اجرائی بلوکه می شود و حلقه بازی اجازه خواهد داشت به اجرا شدن ادامه دهد . حالا که درباره رویدادها بحث کردیم ، اجازه دهید به سمت چیزی جذاب تر بریم : رندرینگ

رندرینگ و تغییر شکل اشکال هندسی

واضح است که اگر هیچ شئ ای برای رندرشدن وجود نداشته باشد ، نیازی به پنجره ها نیست و اگر نمی خواستیم با استفاده از ورودی ، این اشیاء را انیمیت کنیم ، نیازی به رویدادها نبود . SFML چند شیوه برای رندر اشیاء روی صفحه فراهم کرده و ما مهمترین آنها را در این کتاب بررسی می کنیم . قبل از این کار ، باید مطمئن شویم که چرخه رندر ما به درستی جای گرفته و در ترتیب درستی قرار گرفته است .

رندر فریم

کلاس `Window` را یادتان است ؟ این کلاس واسطی برای ترسیم اشکال SFML فراهم نکرده است . ما باید از کلاسی به نام `RenderWindow` برای انجام این کار استفاده کنیم . این کلاس از کلاس `Window` استخراج شده و کارکرد ترسیم را اضافه کرده است . ما می توانیم آن را ایجاد کرده ، رویدادها را از آن جمع آوری کنیم و .. به همان شیوه ای که با کلاس پایه `Window` انجام می دادیم . مثالی از یک حلقه بازی همراه با چرخه رندر :

Main.cpp

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

int main()
{
    sf::RenderWindow window(sf::VideoMode(640, 480), "The title");

    while (window.isOpen())
    {
        //Handle events
        sf::Event event;
        while (window.pollEvent(event))
        {
            if(event.type == sf::Event::EventType::Closed)
                window.close();
        }

        //Update scene

        //Render cycle
        window.clear(sf::Color::Black);

        //Render objects

        window.display();
    }

    return 0;
}
```

توجه کنید که کلاس `RenderWindow` از مدل گرافیک SFML است یعنی باید به جای `<SFML/Window.hpp>` فایل `<SFML/Graphics.hpp>` را داخل کنیم . اما چون از کلاس `Window` استخراج شده ، بدون تغییر چیزی به جز نوع متغییر ، هنوز می تواند در کدهای ما استفاده شود . چرخه رندر کاملا سرراست است ، و به مراحل زیر تقسیم می شود :

پاک کردن پرده نقاشی که می خواهید روی آن ترسیم را انجام دهید .
ترسیم بر روی پرده نقاشی

نمایش پرده نقاشی

این روال رندر در فریم (هر چرخه حلقه) اتفاق می افتد . اگر با فرایند رندرینگ آشنا نباشید ، ممکن است پاک کردن همه چیز مربوط به آخرین فریم و رندر مجدد همه اشیاء موجود در صحنه (حتی اگر این اشیاء از آخرین بار تغییری نکرده باشند) عجیب و بیهوده به نظر آید . اما کارتهای گرافیک برای انجام این روال بهینه سازی شده اند .
چیز دیگری که باید توجه کنید این است که پرده نقاشی که ما روی آن ترسیم را انجام می دهیم ، دارای بافر دوگانه است . شیوه انجام این کار این است که پرده نقاشی دارای دو طرف است که شما می توانید روی آن ترسیم را انجام دهید . در هنگام رندر فریم ، ما فقط بر روی یکی از این دو طرف کار می کنیم یعنی آن طرفی که روی صفحه نمایش داده نشده است . بعد از اتمام رندرینگ ، پرده را می چرخانیم و کاری را که ترسیم کرده ایم ، نشان می دهیم . در فریم بعدی ، روی طرف دیگر پرده کار می کنیم و همین طور الی آخر ... این تکنیک به ما اجازه می دهد تنها پس از اینکه رندرینگ صحنه به پایان رسید ، آن را نمایش دهیم . در SFML با فراخوان `Window::display()` پرده را می چرخانیم (گاهی اوقات از اصطلاح معاوضه بافرها هم برای این کار استفاده می شود) . متد `Window::display()` همچنین می تواند برای مدت زمانی ، نخ اجرائی را به خواب ببرد تا به یک سرعت فریم مطلوب (frame rate – تعداد فریم ها در هر ثانیه) برسد . می توان با یکبار فراخوان `Window::setFramerateLimit()` در ابتدای برنامه ، سرعت فریم مطلوب را تنظیم کرد . این تابع ضمانت نمی کند که سرعت فریم را دقیقا به همان مقداری که معین شده ، محدود کند بلکه یک مقدار تقریبا نزدیک را اعمال می کند .

`Window::clear()` پرده را برای ترسیم مجدد ، پاک می کند . توجه کنید که این یک آرگومان `sf::Color` می گیرد که یک نمایش RGBA از یک رنگ است . می توانیم با فراخوان سازنده و پاس دادن هر مقدار ، به صورت دستی آن را مقداردهی اولیه کنیم ، یا اینکه می توانیم از رنگ های از قبل آماده استفاده کنیم مثلا : `Color::Red`, `Color::Blue`, `Color::Magenta`

ترسیم اشکال

حالا که با روال رندرینگ اشنا هستیم ، اجازه دهید چند شکل را روی صفحه رندر کنیم . ما با شکل های پایه شروع می کنیم و بعدا پیچیده ها را بررسی خواهیم کرد . هنگامی که می خواهیم یک شکل را ترسیم کنیم ، باید ابتدا شئ را ایجاد کنیم . کد مقداردهی اولیه دو شکل را ببینید . آن را درست قبل از حلقه بازی قرار دهید :

```
sf::CircleShape circleShape(50);
circleShape.setFillColor(sf::Color::Red);
circleShape.setOutlineColor(sf::Color::White);
circleShape.setOutlineThickness(3);

sf::RectangleShape rectShape(sf::Vector2f(50, 50));
rectShape.setFillColor(sf::Color::Green);
```

چند کلاس جدید در این مثال آمده است : `CircleShape`, `RectangleShape` و `Vector2f`

احتمالا حدس می زنید که کلاس `Vector2f` برای چیست ؟ این یک vector دوبعدی است که دو `floats` را نگه می دارد . کلاس های دیگری مثل `Vector2i` (برای اعداد صحیح) ، `Vector2u` (برای unsigned int) ، `Vector3i` (برای vector سه بعدی که اعداد صحیح را نگه می دارد) و `Vector3f` (vector سه بعدی که float را نگه می دارد) وجود دارد . ما حتی می توانیم vector های دوبعدی و سه بعدی بسازیم که انواع سفارشی را نگه دارند . این کار با استفاده از الگوهای کلاس `sf::Vector2<class>` و `sf::Vector3<class>` انجام می شود .

RectangleShape, CircleShape و ConvexShape از کلاس انتزاعی Shape ، استخراج می شوند که توسط گروهی از رئوس (نقاط) تعریف شده است . CircleShape تنها یک چندضلعی با گروهی از رئوس است . با استفاده از دومین آرگومان سازنده ، می توانیم تعیین کنیم که دایره از چند نقطه ساخته شده باشد . این آرگومان اختیاری است و دارای مقدار پیش فرض ۳۰ است . از طرف دیگر ، RectangleShape همیشه ۴ راس دارد . سازنده هر دو شکل ، ابعاد شکل را دریافت می کنند : شعاع دایره و عرض و ارتفاع مستطیل .

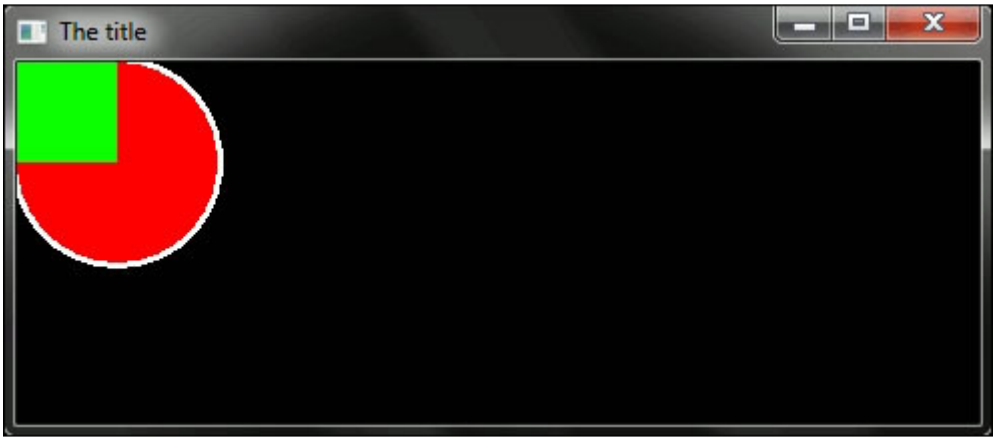
ConvexShape برای شکلی است که می توانیم صریحا رئوسش را مشخص کنیم . هیچ محدودیتی روی تعداد رئوسش وجود ندارد ، آما آنها باید یک شکل محدب (convex) را تشکیل دهند وگرنه شکل به درستی ترسیم نخواهد شد .

شکل ها می توانند دارای رنگ و رنگ اطراف داشته باشند که اینها را می توان با Shape::setFillColor() , Shape::setOutlineColor() و Shape::setOutlineThickness() تغیر داد . این تابع آخری تعداد پیکسل های رنگ اطراف شکل را تنظیم می کند .

برای رندر شکل های قبل ، می توانیم از تابع RenderWindow::draw() استفاده کنیم . چگونگی این کار را ببینید :

```
//Render cycle
window.clear(sf::Color::Black);

window.draw(circleShape);
window.draw(rectShape);

window.display();
```

اجرای این کدها نتیجه زیر را در پی دارد :



توجه می کنید که ترتیب رندر ، یک تفاوت بزرگ ایجاد می کند . اشیاء پس زمینه باید ابتدا رندر شوند سپس هر چیزی که پیش زمینه وجود دارد . در این مثال ، دایره ابتدا رندر شده و بنابراین در پس زمینه خواهد بود ، در حالی که مستطیل در پیش زمینه بر روی دایره نشسته است .

ما می توانیم با تعیین نقاط توسط تابع ConvexShape::setPointCount() ، از ConvexShape استفاده کنیم و با استفاده از ConvexShape::setPoint() این نقاط را به ترتیب تنظیم کنیم . مثالی از یک مثلث :

```
sf::ConvexShape triangle;
triangle.setPointCount(3);
triangle.setPoint(0, sf::Vector2f(100, 0));
triangle.setPoint(1, sf::Vector2f(100, 100));
triangle.setPoint(2, sf::Vector2f(0, 100));
triangle.setFillColor(sf::Color::Blue);
```

بعد از ترسیم آن روی پنجره ، یک مثلث آبی زیبا خواهیم داشت :

در SFML برای شکل های مقعر پشتیبانی وجود ندارد . اما هنوز هم می توان با ایجاد چندین شکل محدب و رندر آنها در جائی مناسب ، اشکال مقعر را ترسیم کرد .

تغییر شکل اشکال هندسی

حالا می دانیم چطور شکل ها را روی صفحه ترسیم کنیم و این عالیست . اما مهم نیست که چندتا از آنها را ترسیم کرده ایم ، چون همه آنها در گوشه بالائی سمت چپ صفحه قرار می گیرند . این یعنی باید موقعیت شکل ها را تغییر دهیم . این می تواند با کمک فراخوان تابع `Shape::setPosition()` انجام شود . همچنین برای چرخاندن (دوران) تابع `Shape::setRotation()` و برای کوچک و بزرگ کردن (مقیاس گذاری) تابع `Shape::setScale()` وجود دارد . در واقع این تابع ها همه بخشی از `sf::Transformable` هستند که کلاس Shape از آن استخراج شده است . استفاده از این تابع ها آسان است :

```
sf::RectangleShape rect(sf::Vector2f(50, 50));
rect.setFillColor(sf::Color::Red);
rect.setPosition(sf::Vector2f(50, 50));
rect.setRotation(30);
rect.setScale(sf::Vector2f(2, 1));
```

یادتان نرود که شکل را رندر کنید . نتیجه را ببینید :

توجه کنید که ما یک مستطیل ایجاد کردیم که در واقع یک مربع با عرض و ارتفاع ۵۰ پیکسل است . اما آن را به نسبت 2:1 مقیاس گذاری کردیم و در نتیجه بزرگتر از اندازه اصلیش رندر می شود . همچنین مستطیل ما کمی کج است چون آن را ۳۰ درجه چرخانده ایم .
در این مثال ؛ ما موقعیت را مستقیما در (50, 50) قرار دادیم ، اما شیوه دیگری هم برای حرکت دادن اشیاء وجود دارد و آن توسط فراخوان `Transformable::move()` و پاس دادن یک بردار است که نشان می دهد ما چه مقدار می خواهیم شئ را از موقعیت حال حاضرش ، حرکت دهیم .
هر چیزی که تابه حال ایجاد کرده ایم ، ساکن بوده است ، پس حالا اجازه دهید کمی زندگی به اشیاء اضافه کنیم . برای این کار باید از به روز کردن فریم استفاده کنیم که تا به حال آن را به کار نگرفته ایم . این بخشی از فریم است درست قبل از اینکه شروع به رندر فریم کنیم . یادآوری می کنم ، یک فریم بازی (چرخه حلقه) به شکل زیر است :

مدیریت ورودی
به روز کردن فریم
رندر فریم

مهم است که قبل از رندر اشیاء ، آنها را به روز کنیم ، در غیر این صورت وضعیت و حالت فعلی آنها به درستی رندر نخواهد شد ، آنها با وضعیتی که از آخرین فریم به دست آورده اند ، رندر خواهند شد .
مثال بعدی نشان می دهد که چطور از ترکیب انتقال و دوران برای ایجاد یک انیمیشن ساده استفاده کنیم :

Main.cpp

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

int main()
{
    sf::RenderWindow window(sf::VideoMode(480, 180), "Animation");
    //Set target Frames per second
    window.setFramerateLimit(60);

    sf::RectangleShape rect(sf::Vector2f(50, 50));
    rect.setFillColor(sf::Color::Red);
    rect.setOrigin(sf::Vector2f(25, 25));
    rect.setPosition(sf::Vector2f(50, 50));

    while (window.isOpen())
    {
        /*Handle events here*/

        //Update frame
        rect.rotate(1.5f);
        rect.move(sf::Vector2f(1, 0));

        //Render frame
        window.clear(sf::Color::Black);
        window.draw(rect);
        window.display();
    }
    return 0;
}
```

چند درس در این مثال وجود دارد . اول اینکه کجا و چگونه سرعت فریم (frame rate) را محدود کنیم – درست بعد از مقداردهی اولیه پنجره . این باعث می شود منطق بازی ما تقریبا به ۶۰ فریم در هر ثانیه محدود شود . یادتان باشد که این حد بالائی سرعت فریم را کنترل می کند . اگر فریم ها بیش از 1/60 ثانیه نیاز داشته باشند تا کامل شوند (مدیریت رویدادها ، به روزکردن اشیاء و رندر) در این صورت سرعت فریم به زیر 60 می آید . اما با کدهای ساده ما ، این خیلی بعید خواهد بود .

حتما به فراخوان جدید `RectangleShape::setOrigin()` توجه کرده اید . مبداء یک شئ تعیین می کند که چطور یک شئ روی صفحه رندر شود . آن به عنوان نقطه مرکز برای انتقال ، دوران و مقیاس گذاری شئ عمل می کند . در مثال قبل ما یک مربع با اندازه 50 x 50 داشتیم . مرکز مربع (25, 25) است ، بنابراین باید آن را به عنوان مبداء شئ تعیین کنیم . در غیر این صورت ، شئ حول مبداء پیش فرضش یعنی (0, 0) شروع به چرخش می کند . آخرین چیزی که باید درخصوص مبداء توجه کنید این است که آن بخشی از کلاس `Transformable` است و بنابراین تمام کلاس های استخراج شده از آن به آن دسترسی خواهند داشت .

فرایند انیمیشن ما ساده است . در هر فریم ، ما مربع را 1.5 درجه می چرخانیم و 1 پیکسل به سمت راست حرکت می دهیم . با تنظیم سرعت فریم به 60 FPS ، می توان تخمین زد که بعد از هر ثانیه ، مربع تقریبا 90 درجه می چرخد (60 x 1.5) و 60 پیکسل به سمت راست حرکت می کند (60 x 1p) . اما اجرای منطق بازی با این شیوه (بسته به سرعت فریم بون) خیلی نامطمئن و خطرناک است . ما در فصل ۳ ، هنگامی که انیمیشن و منطق بازی را اجرا می کنیم ، چگونگی مدیریت زمان را توضیح خواهیم داد .

حالا اجازه دهید ببینیم که چطور می توان شکل ها را به صورت بلادرنگ ، کنترل کرد .

کنترل شکل ها

یک راه برای اینکه باعث حرکت شکل ها شویم ، استفاده از مدیریت رویدادهاست . هنگامی که کاربر کلیدی را فشار دهد ، ما شروع به حرکت دادن شئ می کنیم و هنگامی که کلید رها شود ، می توانیم حرکت شئ را متوقف کنیم . کدهای مثال را ببینید :

```
    bool moving = false;
    while (window.isOpen())
    {
        sf::Event ev;
        while (window.pollEvent(ev))
        {
            if (ev.type == sf::Event::EventType::KeyPressed &&
                ev.key.code == sf::Keyboard::Right)
                moving = true;
            if (ev.type == sf::Event::EventType::KeyReleased &&
                ev.key.code == sf::Keyboard::Right)
                moving = false;
        }
        //Update frame
        if (moving)
        {
            //Move the object
        }

        //Render the object
    }

    return 0;
}
```

متغییر moving تعیین می کند که آیا باید شئ را در فریم فعلی حرکت دهیم یا خیر ؟ هنگامی که ما کلید حرکتی ***Right*** را فشار می دهیم یا رها می کنیم ، مقدار این متغییر تغییر می کند . اما این کدهای زیادی برای یک کار ساده است . خوشبختانه SFML یک واسط خیلی ساده فراهم کرده تا وضعیت حال حاضر دستگاه های ورودی (صفحه کلید ، ماوس و جوی استیک ها) را چک کنیم . این کار با فراخوان `Keyboard::isKeyPressed()` انجام می شود . هنگامی که یک کلید را به عنوان آرگومان به این تابع پاس دهیم ، وضعیت حال حاضر این کلید از جهت فشرده شدن را خواهیم فهمید . اما این تابع به فوکوس پنجره اهمیتی نمی دهد . تصور کنید که بازیکن پنجره را مینیمایز کرده و در اینترنت جستجو می کند . حال اگر بازیکن باز کلید داده شده را فشار دهد ، باز هم تابع `true` بازگشت می دهد . کد قبلی را می توانید با کد زیر جایگزین کنید :

```
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Right))
        {
            //Move the object
        }
```

همین طور که می بینید ، لازم نیست در اینجا وضعیت کلید را ذخیره کنیم ، SFML این کار را برای ما انجام می دهد .
به همین شیوه می توانیم ، وضعیت دیگر دستگاه های ورودی را نیز چک کنیم . ماوس تابع هائی برای گرفتن موقعیتش ، وضعیت دکمه هایش و تنظیم موقعیتش نسبت به دسکتاپ دارد . برای گرفتن موقعیت از `Mouse::getPosition()` استفاده کنید . برای تنظیم موقعیت از `Mouse::setPosition()` استفاده کنید . برای بررسی اینکه آیا دکمه آن فشرده شده یا خیر ، از `Mouse::isButtonPressed()` استفاده کنید .
در خصوص جوی استیک ها ، چون همه تابع ها استاتیک هستند با استفاده از آرگومان `Id` تعیین کنیم که دنبال کدام جوی استیک هستیم . این تابع ها عبارتند از :

`Joystick::isConnected()`

آرگومان : ID

این تابع بررسی می کند که آیا جوی استیک با ID داده شده ، متصل شده یا خیر ؟

`Joystick::hasAxis()`

آرگومان : ID و محور

این تابع بررسی می کند که آیا جوی استیک ، محور تعیین شده را دارد یا خیر ؟

`Joystick::getButtonCount()`

آرگومان : ID

این تابع تعداد دکمه های روی جوی استیک را مشخص می کند .

`Joystick::getAxisPosition()`

آرگومان : ID و محور

این تابع مقدار یک محور را در محدوده [0, 1] می گیرد .

`Joystick::isButtonPressed()`

آرگومان : ID و دکمه
این تابع بررسی می کند که ایا یک دکمه روی جوی استیک مشخص شده ، فشرده شده یا خیر ؟

اجازه دهید حالا آخرین مثال را بحث کنیم که ترکیبی از مطالب این فصل است . ما یک بازی ساده داریم که بازیکن به عنوان یک مربع سبز بازی می کند و او باید بدون اینکه تماسی با شئ قرمز رنگ داشته باشد ، به مربع آبی برسد . تابع زیر یک تابع کمکی است تا اشیاء شبیه به `RectangleShape` را با سادگی و بدون تکرار کدها ، ایجاد کنیم :

```
void initShape(sf::RectangleShape& shape, sf::Vector2f const& pos, sf::Color const& color)
{
    shape.setFillColor(color);
    shape.setPosition(pos);
    shape.setOrigin(shape.getSize() * 0.5f); // The center of the rectangle
}
```

تابع `initShape()` یک شکل ، بردار و رنگ می گیرد و آنها را به شئ `RectangleShape` واگذار می کند . این تابع همچنین نقطه مبداء شکل را مرکز آن قرار می دهد .
مرحله بعدی مقداردهی اولیه اشیاء است :

```
sf::RenderWindow window(sf::VideoMode(480, 180), "Bad Squares");
//Set target Frames per second
window.setFramerateLimit(60);

sf::Vector2f startPos = sf::Vector2f(50, 50);
sf::RectangleShape playerRect(sf::Vector2f(50, 50));
initShape(playerRect, startPos, sf::Color::Green);
sf::RectangleShape targetRect(sf::Vector2f(50, 50));
initShape(targetRect, sf::Vector2f(400, 50), sf::Color::Blue);
sf::RectangleShape badRect(sf::Vector2f(50, 100));
initShape(badRect, sf::Vector2f(250, 50), sf::Color::Red);
```

اولین کاری که باید انجام دهیم ، بازکردن یک پنجره مناسب است . سپس سرعت فریم را به مقدار استاندارد 60 فریم در ثانیه تنظیم می کنیم . سپس از متغییر `sf::Vector2f` به عنوان نقطه شروع بازیکن استفاده می کنیم . بعد از مقداردهی اولیه مربع بازیکن ، هدف را مقداردهی اولیه می کنیم که یک مربع آبی است که کمی در سمت راست دنیای بازی است . آخرین شکل ، مربعی است که بازیکن باید از آن دوری کند و در میانه راه قرار می گیرد .

به روز کردن فریم ، جائی که همه منطق بازی اتفاق می افتد ، به شکل زیر است :

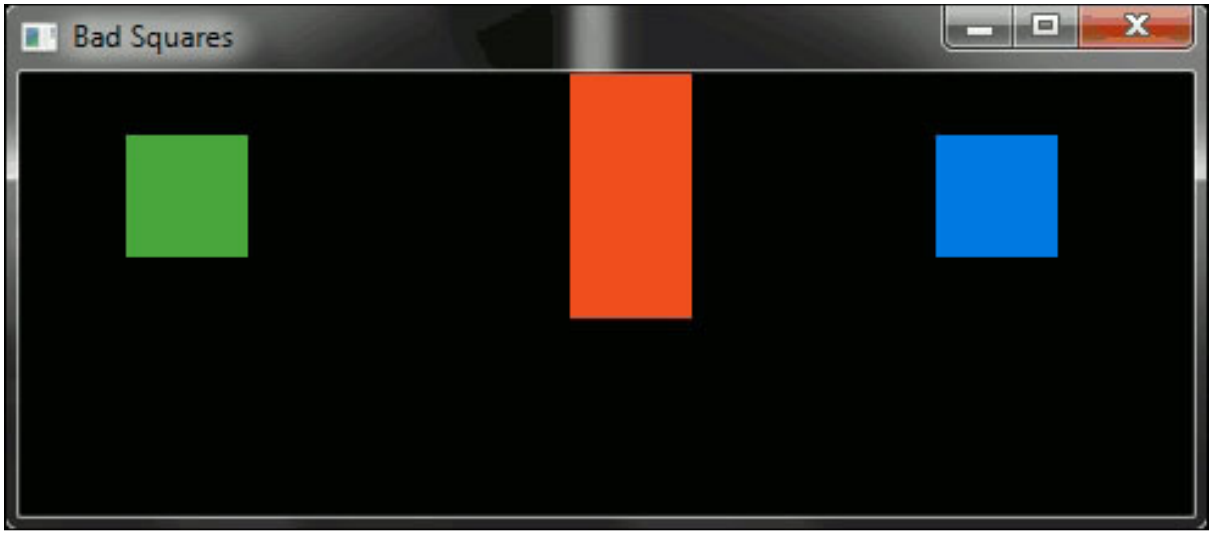
```
//Always moving right
playerRect.move(1, 0);
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Down))
    playerRect.move(0, 1);
if (sf::Keyboard::isKeyPressed(sf::Keyboard::Key::Up))
    playerRect.move(0, -1);

//Target reached. You win. Exit game
if (playerRect.getGlobalBounds().intersects(targetRect.getGlobalBounds()))
    window.close();
//Bad square intersect. You lose. Restart
if (playerRect.getGlobalBounds().intersects(badRect.getGlobalBounds()))
    playerRect.setPosition(startPos);
```

از کدهای قبلی به وضوح می بینیم که بازیکن همیشه به سمت راست حرکت می کند و کنترلی بر این ندارد . شما می توانید آن را تغییر دهید تا بر روی ۴ جهت کنترل داشته باشد . در حال حاضر تنها جهت هائی که بازیکن می تواند حرکت کند ، سمت بالا و پائین است که با کلید های جهتی انجام می شود .
صرف نظر از مدیریت ورودی ، باید بررسی کنیم که کد ، منطقی برای شرایط برنده شدن یا بازنده شدن دارد یا خیر ؟ ما نیاز به شیوه ای داریم تا تشخیص برخورد (collision detection) بین این مربع ها را مدیریت کند . خوشبختانه تمام اشکال در SFML ، دو تابع به نام های `Shape::getGlobalBounds()` و `Shape::getLocalBounds()` دارند که یک `sf::FloatRect` بازگشت می دهند که نشان دهنده محدوده جهانی یا محلی شکل حال حاضر است . محدوده مستطیلی یک شکل (گاهی اوقات جعبه محصور – **bounding box** هم گفته می شود) مستطیلی با کوچکترین مساحت سطح ممکن است که محتوی تمام شکل باشد . جهانی و محلی به تغییر شکل آن شکل اشاره دارد . اگر شکل به هر شیوه ای دچار تغییر شده باشد ، موقعیت ، مقیاس گذاری و دوران در محدوده محلی نادیده گرفته می شود در حالیکه ، در محدوده جهانی اینها در نظر گرفته می شوند . هنگامی که محدوده جهانی را داشته باشیم ، می توانیم از تابع `FloatRect::intersect()` استفاده کنیم که یک `FloatRect` دیگر می گیرد و اگر دو مستطیل با هم برخورد کرده باشند ، `true` بازگشت می دهد .

RectangleShape را با کلاس FloatRect اشتباه نگیرید . این دو کارهای متفاوتی انجام می دهند . اولی برای ترسیم است و دومی کلاسی برای ذخیره مشخصات یک مستطیل (مقادیر بالا ، چپ ، پائین و راست) .

خود منطق بازی ساده است . اگر بازیکن با مربع هدف برخورد کند ، بازیکن برنده خواهد شد (بازیکن باید از بازی خارج شود) . اگر بازیکن با مربع قرمز رنگ برخورد کند ، بازیکن می بازد (بازیکن باید بازی را دوباره شروع کند) .

در خصوص رندر فریم ، چیز خاصی وجود ندارد ، کافی است این ۳ شکل را در هر ترتیبی که دوست دارید ترسیم کنید . نتیجه اجرای بازی را ببینید :



هر چقدر که دوست دارید کدها را تغییر دهید و تنظیمات مختلف را تست کنید (شکل های بیشتری اضافه کنید و منطق بازی را توسعه دهید .)

خلاصه

بعد از تکمیل این فصل ، باید احساس خوبی درباره خودتان داشته باشید . این یعنی برای هرچیزی که این کتاب پیشنهاد می دهد ، آماده هستید . این فصل پایه ای را ساخت که بازی شما بر روی آن خواهد ایستاد . اگر چه مثال ها ساده بودند ، نباید کپی شوند و آن طوری که هستند مورد استفاده قرار گیرند . آنها برای این بود که بفهمید SFML چطور ساخته شده و چطور می توانید از این دانش به نفع خودتان استفاده کنید .

در فصل بعد ، ما بافت ها ، sprite ها و مدیریت منابع را توضیح می دهیم .

فصل ۲ : بارگذاری و استفاده از بافت ها

بافت ها (Texture) خیلی مهم اند هم برای بازی های دوبعدی هم برای بازی های سه بعدی . آنها به ما اجازه می دهند تصاویر را بر روی اشیاء ترسیم کنیم . در این فصل ، به شیوه های بارگذاری بافت ها به داخل حافظه و ترسیم آنها بر روی شکل ها می پردازیم . کلاس `Sprite` وجود دارد و خواهیم دید که چطور با کلاس `Shape` متفاوت است . در نهایت خواهیم دید که چطور منابع را در سرتاسر عمر بازی ، از تخریب ایمن نگه داریم .

در این فصل موارد زیر بررسی خواهند شد :

بارگذاری بافت ها
رندر شکل ها با استفاده از بافت ها
sprite چیست ؟
مدیریت منابع

بارگذاری بافت ها

بافت ها اشیاء کاملا ساده ای هستند . یک بافت دوبعدی در واقع تصویری است که معمولا در حافظه واحد پردازش گرافیکی **(GPU)** ذخیره می شود . SFML برای پردازش و رندر تصاویر ، دو کلاس `Image` و `Texture` را فراهم کرده است .

تصاویر در مقابل بافت ها

مهمترین تفاوت بین تصاویر و بافت ها ، هدف آنهاست : مدیریت و رندرینگ . کلاس `Image` بارگذاری ، ذخیره و دستکاری در پیکسل های تصاویر را مدیریت می کند در حالیکه کلاس `Texture` برای رندرینگ استفاده می شود . این دو کلاس رفتارهای متفاوتی دارند ، اما این موضوع که هر دو داده های یکسانی را نگه می دارند ، فرق نمی کند : آرایه ای از پیکسل ها . SFML راه های ساده ای را برای تولید یکی از دیگری فراهم کرده است . مثلا اگر بخواهیم تصویری را از یک فایل بارگذاری کرده و کمی آن را تغییر دهیم ، می توانیم یک `Texture` را از آن `Image` تولید کنیم . اما اگر بخواهیم دوباره آن شئ `Texture` را تغییر دهیم ، باید آن را به `Image` کپی کنیم . این فرایند می تواند خیلی پرهزینه باشد و باید از انجام آن دوری کنیم .

ایجاد تصاویر

قبل از رفتن به سراغ بافت ها ، راه های تولید و بارگذاری تصاویر را بررسی می کنیم . بسیاری از تابع هائی که در کلاس `Image` می بینیم در کلاس `Texture` نیز موجود است . کد زیر نشان می دهد چطور یک تصویر 50 x 50 ایجاد و آن را با رنگ قرمز پر کنیم :

```
sf::Image image;
image.create(50, 50, sf::Color::Red);
```

دو آرگومان اول تابع `Image::create()` نشان دهنده عرض و ارتفاع تصویر هستند و آخرین آرگومان رنگی است که تصویر را پر می کند . به صورت پیش فرض ، این رنگ ، سیاه با مقدار آلفای 255 است .
تصاویر می توانند توسط پاس دادن آرایه ای از پیکسل ها ، مستقیما تولید شوند . آرایه باید عناصر نوع `Uint8` را نگه دارد ، که یک بایت از حافظه است . چون `Image::create()` به رنگی در فرمت RGBA نیاز دارد ، ما باید مطمئن شویم که آرایه دقیقا برای هر

رنگ ، ۴ بایت نگه می دارد (۱ بایت برای هر جزء رنگی) . هر ۴ بایت متوالی ، نشان دهنده یک پیکسل از تصویر است ، که به صورت سطر در ستون قرار گرفته است . مثال :

```
const unsigned int kWidth = 5, kHeight = 5;

//Array size = width * height * 4(RGBA)
sf::Uint8 pixels[kWidth * kHeight * 4] =
{
    255, 255, 255, 255, //White
    0, 0, 0, 255,       //Black
    255, 0, 0, 255,     //Red
    128, 128, 128, 255,  //Gray

    //...all other pixels
};

sf::Image image;
image.create(kWidth, kHeight, pixels);
```

کد قبلی نشان می دهد که چطور یک تصویر کوچک (5 x 5) ایجاد کنیم . توجه کنید که آرایه محتوی عناصر نوع `sf::Color` نیست ، بلکه محتوی اجزای RGBA یک رنگ است . اما شیوه تعیین رنگ ها در هر دو مورد یکسان است : پاس دادن یک بایت (`Uint8`) برای هر یک از ۴ جزء . این یعنی هر ۴ بایت (`4xUint8`) نشان دهنده یک پیکسل از تصویر است .

تصاویر می توانند از فایل ها هم بارگذاری شوند . مثال :

```
sf::Image image;
image.loadFromFile("myImage.png");
```

کد فرض می داند که در پوشه کار برنامه ، یک تصویر با نام `myImage.png` وجود دارد . اگر بخواهید از تصاویری که از قبل روی کامپیوتر موجود هستند ، تصویر را ایجاد کنید ، بارگذاری از فایل شیوه ای کارا است . SFML فرمت های فایل زیر را پشتیبانی می کند :
`bmp`, `png`, `tga`, `gif`, `psd`, `hdr`, `pic`, `jpg`

اگر سعی کنیم فایلی را با فرمت دیگر بارگذاری کنیم ، یا فایل مشخص شده موجود نباشد ، `Image::loadFromFile()` مقدار `false` بازگشت می دهد و پیغام زیر در خط فرمان چاپ می شود :

```
Failed to load image "myImage.png". Reason : Unable to open file
```

هنگامی که بارگذاری یک فایل شکست بخورد (به هر علتی) باید کاری انجام دهیم (به کاربر اطلاع دهیم ، برنامه را متوقف کنیم و ..) در کد زیر اگر به درستی تصویر بارگذاری نشود ، از تابع `main()` خارج خواهیم شد . این راهی است امن برای جلوگیری از به وجود آمدن باگ های بیشتر :

```
sf::Image image;
if (!image.loadFromFile("myImage.png"))
    return -1;
```

توصیه می کنم برای تولید بازی های باکیفیت برای کاربر از فرمت هائی مثل PNG استفاده کنید . فرمت های پراتلاف مثل JPEG با ایجاد فشردگی بیشتر باعث کاهش کیفیت تصویر می شوند . تنها وقتی باید از JPEGs استفاده کنیم که اندازه برنامه مهم است یا در خصوص کیفیت تصاویر نگرانی نداریم . مثلا می توانیم از JPEGs برای تصاویر بزرگتر مثل پس زمینه ها استفاده کنیم که با داشتن یک سطح فشردگی مناسب ، فضای زیادی را برای ما ذخیره می کنند اما درجه کیفیت پائین تری دارند .
کلاس `Image` متدهای مفیدی را برای دستکاری یک تصویر فراهم کرده است . توابعی مثل `Image::getPixel()` و `Image::setPixel()` به ما اجازه می دهند پیکسل ها را تغییر دهیم . اگر بخواهیم همه پیکسل های یک تصویر را بخوانیم ، تابع `Image::getPixelPtr()` وجود دارد که ابتدای یک آرایه از پیکسل ها را بازگشت می دهد . این آرایه به همان فرمتی است که در دومین مثالمان برای تولید یک تصویر استفاده کردیم . `Image::flipHorizontally()` و `Image::flipVertically()` همه تصویر را با چرخاندن پیکسل هایش در یک جهت خاص ، تغغیر شکل می دهد . در نهایت می توانیم توسط `Image::saveToFile()` و پاس دادن یک اسم فایل ، تصویر را در فایل ذخیره کنیم . فرمت هائی که برای ذخیره فایل پشتیبانی می شوند عبارتند از :

`bmp`, `png`, `tga`, `jpg`

حالا که می دانیم چطور تصاویر را ایجاد و دستکاری کنیم ، اجازه دهید تا ببینیم که چطور باید بافت ها را از آنها ایجاد کرد .

تولید بافت ها

کلاس `Texture` بیشتر متدهای کلاس `Image` را به اشتراک گذاشته است . مثلا می توانیم بافت ها را به همان شیوه تصاویر ، از فایل ها بارگذاری کنیم :

```
sf::Texture texture;
if (!texture.loadFromFile("myTexture.png"))
    return -1;
```

`Texture::loadFromFile()` عملکرد بیشتری ارائه می کند . هنگام بارگذاری یک بافت از فایل ، می توانیم فقط یک بخش کوچکی از تصویر را انتخاب و بارگذاری کنیم . در `Texture::loadFromFile()` یک آرگومان اختیاری هست که به ما اجازه این کار را می دهد . در کد زیر ما فقط یک مربع 32 x 32 پیکسلی از تصویر اصلی را بارگذاری می کنیم که از گوشه بالائی سمت چپ شروع می شود :

```
sf::Texture texture;
if (!texture.loadFromFile("myTexture.png", sf::IntRect(0, 0, 32, 32)))
    return -1;
```

کد قبل همه تصویر را بارگذاری می کند و سپس فقط یک بافت از مستطیل مشخص شده را ایجاد می کند . اگر بخواهیم از یک تصویر چندین بار استفاده کنیم ، این متد ناکارا می شود . راه دیگر بارگذاری یکباره تصویر (در `Image`) و سپس استفاده از آن برای تولید بافتهایمان است . در کد زیر می بینید که چطور بافت ها را مستقیما از تصویر ایجاد می کنیم :

```
sf::Image image;
image.create(50, 50, sf::Color::Red);

sf::Texture texture;
texture.loadFromImage(image);
```

مثل کلاس Image ، بافت ها می توانند با فراخوان Texture::create() ایجاد شوند . اما باید مراقب ابعادی که برای بافت فراهم می کنیم ، باشیم چون کارت های گرافیک در خصوص اندازه بافت محدودیت دارند . خوشبختانه تابع Texture::getMaximumSize() وجود دارد که یک integer با بیشترین اندازه ممکن یک بافت در کارت گرافیک ، بازگشت می دهد . اگر بیشترین سازگاری را از برنامه خود انتظار داریم ، باید سعی کنیم از بافت هائی استفاده کنیم که اندازه ای برابر با به توان ۲ دارند (32, 64, 256, 1024, ...)

می توانیم با فراخوان Texture::copyToImage() یک تصویر را از یک شئ Texture بسازیم . این کار عملیاتی کند است چون داده ها را از GPU به RAM کپی می کند پس مراقب استفاده از آن در جائی مناسب باشید .

همه چیز تا الان خوب به نظر می رسد اما یک مشکل هنوز باقی مانده است : هنوز ندیده ایم که چطور بافت ها را روی صفحه نمایش دهیم . اجازه دهید این مشکل را برطرف کنیم .

رندر شکل ها با استفاده از بافت ها

اجازه دهید با یک چیز مهم شروع کنیم ، که کاملا برای همه واضح نیست . بافت ها به خودی خود نمی توانند رندر شوند . آنها به سطحی نیاز دارند تا بر آن ترسیم و نگاشت شده و سپس آن سطح بتواند رندر شود . همانطور که در ابتدای فصل گفته شد ، بافت ها فقط مجموعه ای از پیکسل ها هستند که نمی توانند بدون داشتن چند جور ارجاع (مثل موقعیت ، دوران و ...) مستقیما بر روی صفحه رندر شوند . اما در SFML کلاس های قابل رندری وجود دارد که می توانند برای سطحشان از بافت ها استفاده کنند . در حقیقت ما از یکی از این کلاس ها در فصل قبلی استفاده کردیم : shape (شکل های هندسی)

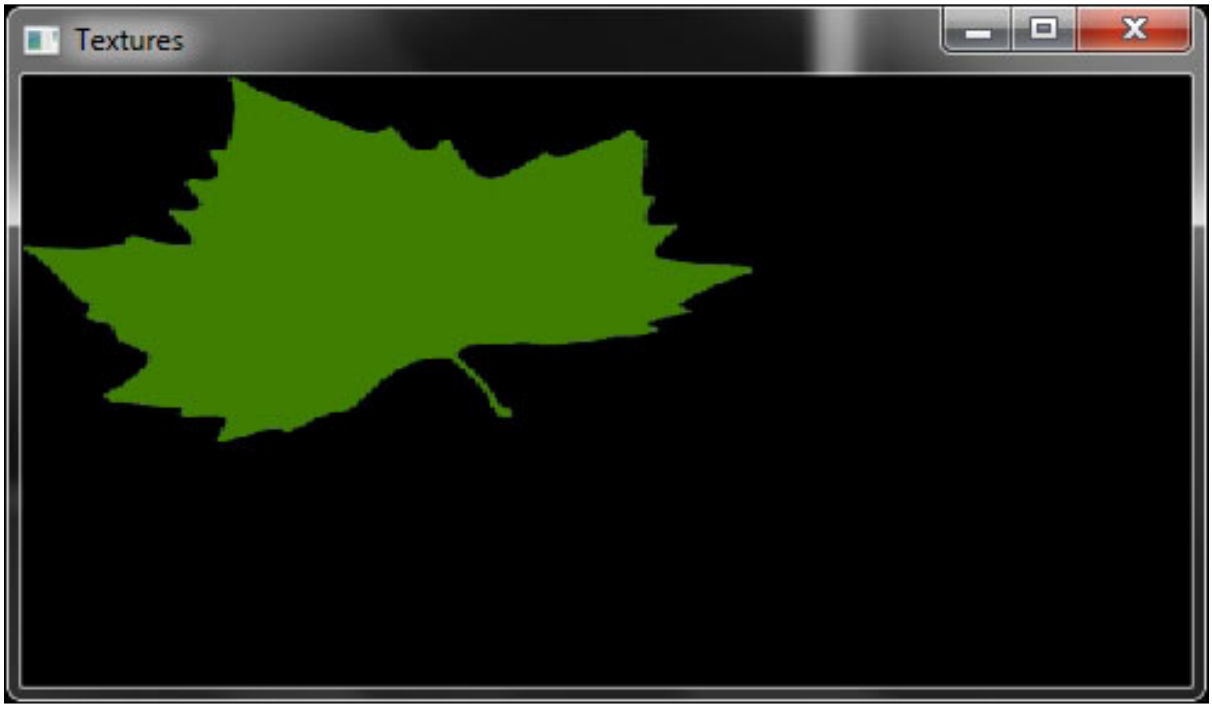
صرف نظر از رنگ و رنگ اطراف ، هر شئ Shape می تواند یک بافت هم داشته باشد . می توان با فراخوان Shape::setTexture() و پاس دادن اشاره گر به بافت ، بافت را به شکل اعمال کرد . آخرین کاری که باید انجام دهیم ، رندر شکل داخل پنجره است :

```
sf::Texture texture;
texture.loadFromFile("myTexture.png");

sf::RectangleShape rectShape(sf::Vector2f(300, 150));
rectShape.setTexture(&texture);

while (window.isOpen())
{
    //Handle events

    window.clear(sf::Color::Black);
    window.draw(rectShape);
    window.display();
}
```

اولین چیزی که باید توجه کرد این است که Shape::setTexture() به جای ارجاع ، یک اشاره گر می گیرد . این موضوع علت این است که ما آدرس بافت را با استفاده از &texture پاس دادیم . شکل سپس اشاره گر را به صورت محلی ذخیره کرده و هر زمان که نیاز به رندر باشد ، از آن استفاده می کند . این یعنی آدرسی که ما به تابع پاس داده ایم باید سرتاسر زندگی شکل ، یک بافت معتبر را نگه داشته باشد . جابه جا کردن بافت در حافظه یا تخریب آن ، باعث ایجاد یک اشاره گر معلق در داخل شئ Shape شده و در نتیجه شاهد رفتاری تعریف نشده و نامشخص خواهیم بود . به همین دلیل همیشه باید مطمئن شویم که بافت از حوزه ای که شئ از آن استفاده می کند ، خارج نمی شود . در آخرین بخش این فصل ، تکنیک مدیریت منابع را بررسی خواهیم کرد .

هنگامی که یک بافت را روی `RectangleShape` قرار می دهیم ، آن سعی می کند با کوچک یا بزرگ کردن خود برای مستطیل مشخص شده ، مناسب و جفت و جور شود . برای مثال ما ، اگر بافت عرض 200 ، ارتفاع 200 داشته باشد و مستطیل درای عرض 300 و ارتفاع 150 باشد ،

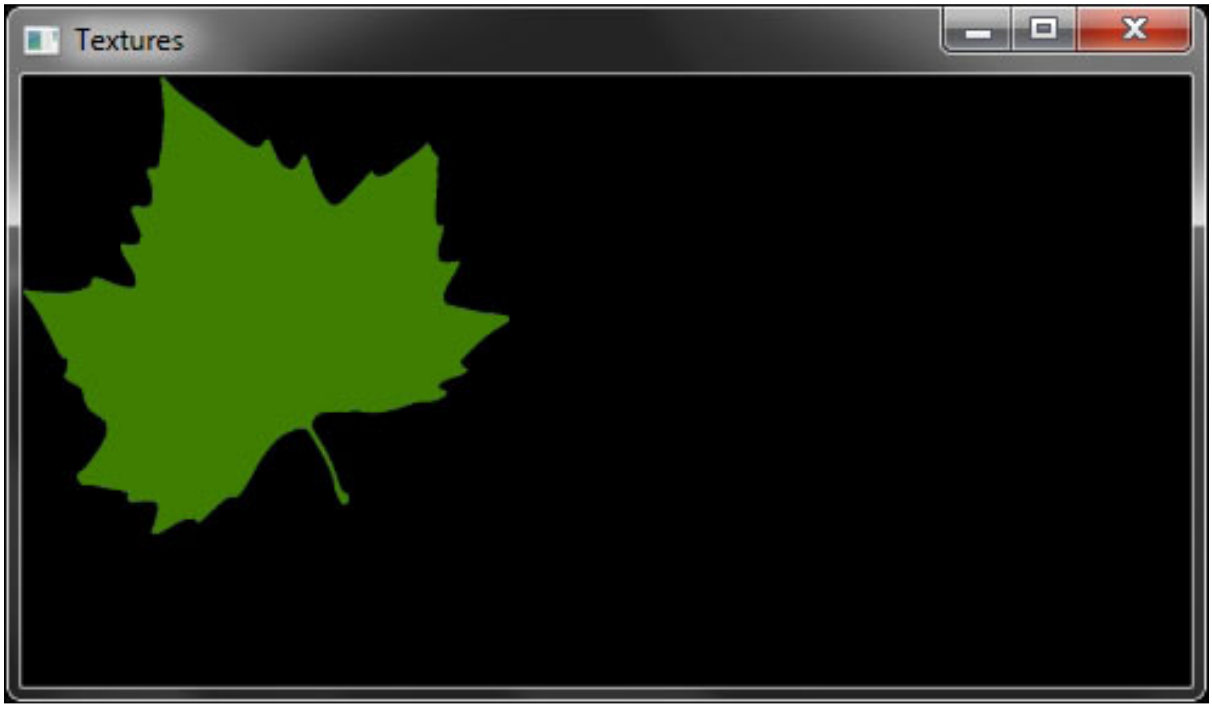
آنگاه بافت بر روی محور x به صورت کشیده تر ظاهر می شود و بر روی محور y فشرده تر خواهد شد :



برای تنظیم شکل `RectangleShape` به اندازه دقیق بافت ، می توان از تابع `Texture::getSize()` استفاده کرد :

```
sf::Vector2u textureSize = texture.getSize();
float rectWidth = static_cast<float>(textureSize.x);
float rectHeight = static_cast<float>(textureSize.y);
sf::RectangleShape rectShape(sf::Vector2f(rectWidth, rectHeight));
rectShape.setTexture(&texture);
```

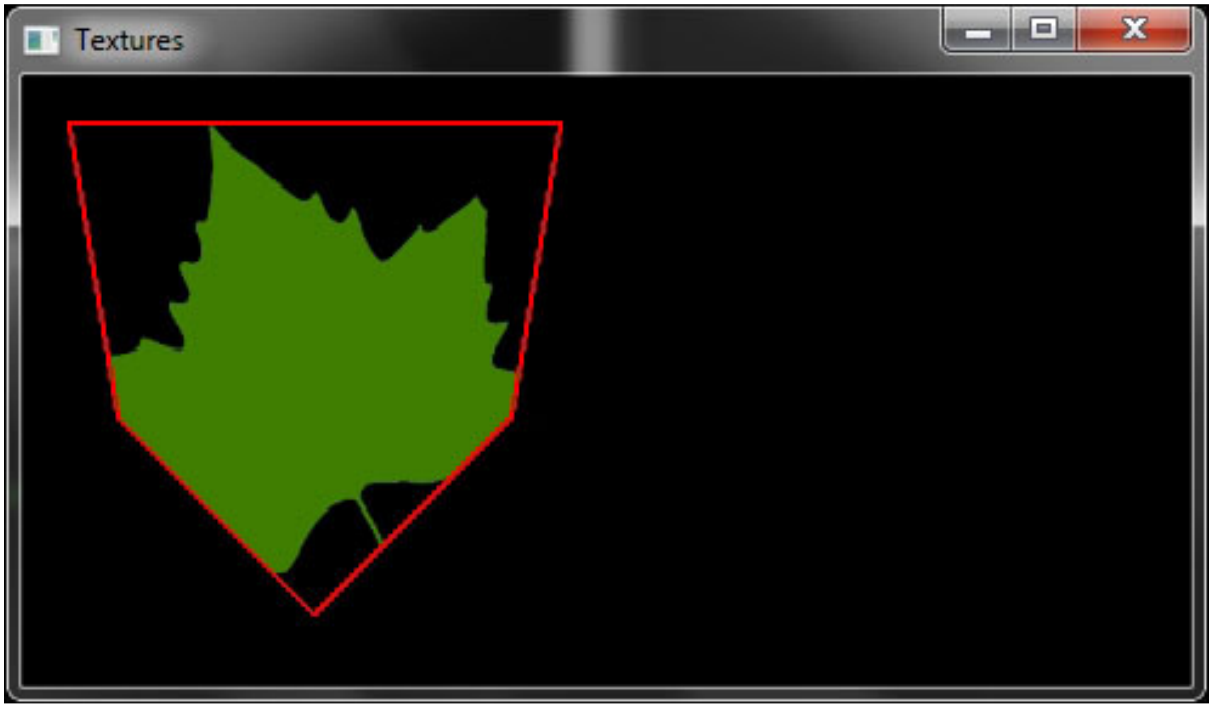
نتیجه را ببینید :



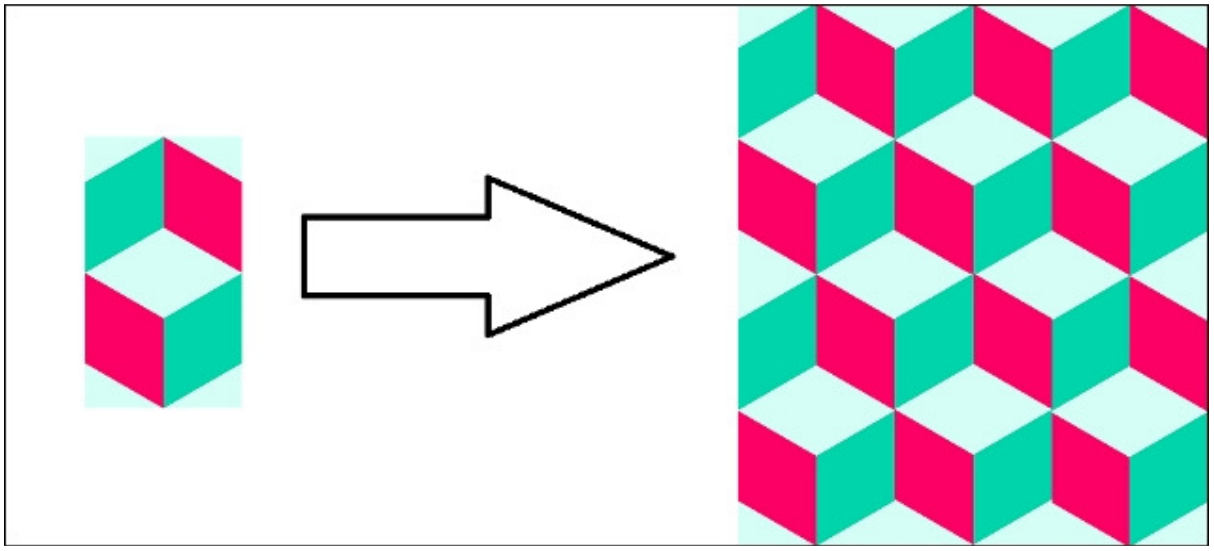
بافت ها همچنین می توانند بر روی اشیاء `CircleShape` و `ConvexShape` هم ترسیم و نگاشت شوند . مثال زیر نشان می دهد که چطور می توانیم اندازه بافتی که توسط یک `ConvexShape` نشان دهده می شود را محدود کنیم :

```
sf::ConvexShape shape(5); //Convex shape has 5 points
shape.setPoint(0, sf::Vector2f(0, 0));
shape.setPoint(1, sf::Vector2f(200, 0));
shape.setPoint(2, sf::Vector2f(180, 120));
shape.setPoint(3, sf::Vector2f(100, 200));
shape.setPoint(4, sf::Vector2f(20, 120));
shape.setTexture(&texture);
shape.setOutlineThickness(2);
shape.setOutlineColor(sf::Color::Red);
shape.move(20, 20); //Move it, so the outline is clearly visible
```

ما یک چندضلعی ساده با ۵ راس تولید کردیم و به آن یک بافت واگذار کردیم . برای وضوح بیشتر خطهای اطراف آن را نشان می دهیم و کمی از لبه پنجره فاصله می دهیم . نتیجه را ببینید :



بافت ها همچنین می توانند چندین بار بر روی یک سطح تکرار شوند . فرض کنید می خواهیم سطح زیر را از این کاشی ایجاد کنیم :

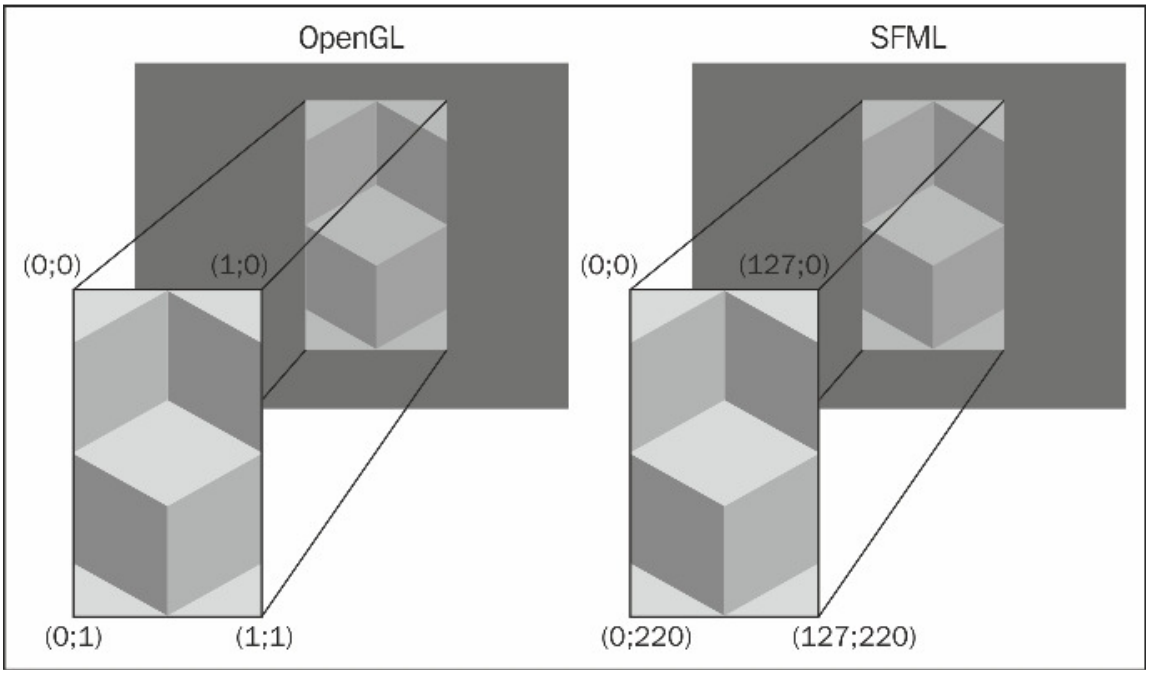


چون کاشی سمت چپ کاملا از همه جهات یکپارچه است ، می توانیم تعداد زیادی از آن را کنار هم قرار داده و یک بافت بزرگتر روی سطح ایجاد کنیم . یک راه انجام این کار ایجاد تصویری بزرگتر در نرم افزارهای ویراش تصویر مثل فوتوشاپ است ، اما تصویر حافظه زیادی را نیاز خواهد داشت . راه دیگر این است که فقط یک کاشی را در حافظه GPU بارگذاری کرده و از آن به عنوان بافت متوالی بر روی سطح داده شده استفاده کنیم .

کاشی ما ابعاد 128;221 دارد ، ما کاشی را ۳ بار روی محور x و ۲ بار روی محور y تکرار می کنیم .یعنی در انتها سحی با اندازه 384;442 خواهیم داشت . برای چنین شکلی ، فقط منطقی است که از کلاس `RectangleShape` استفاده کنیم . ببینید :

```
sf::Texture texture;
texture.loadFromFile("tile.png");

sf::RectangleShape rectShape(sf::Vector2f(128 * 3, 221 * 2));

rectShape.setTexture(&texture);
```

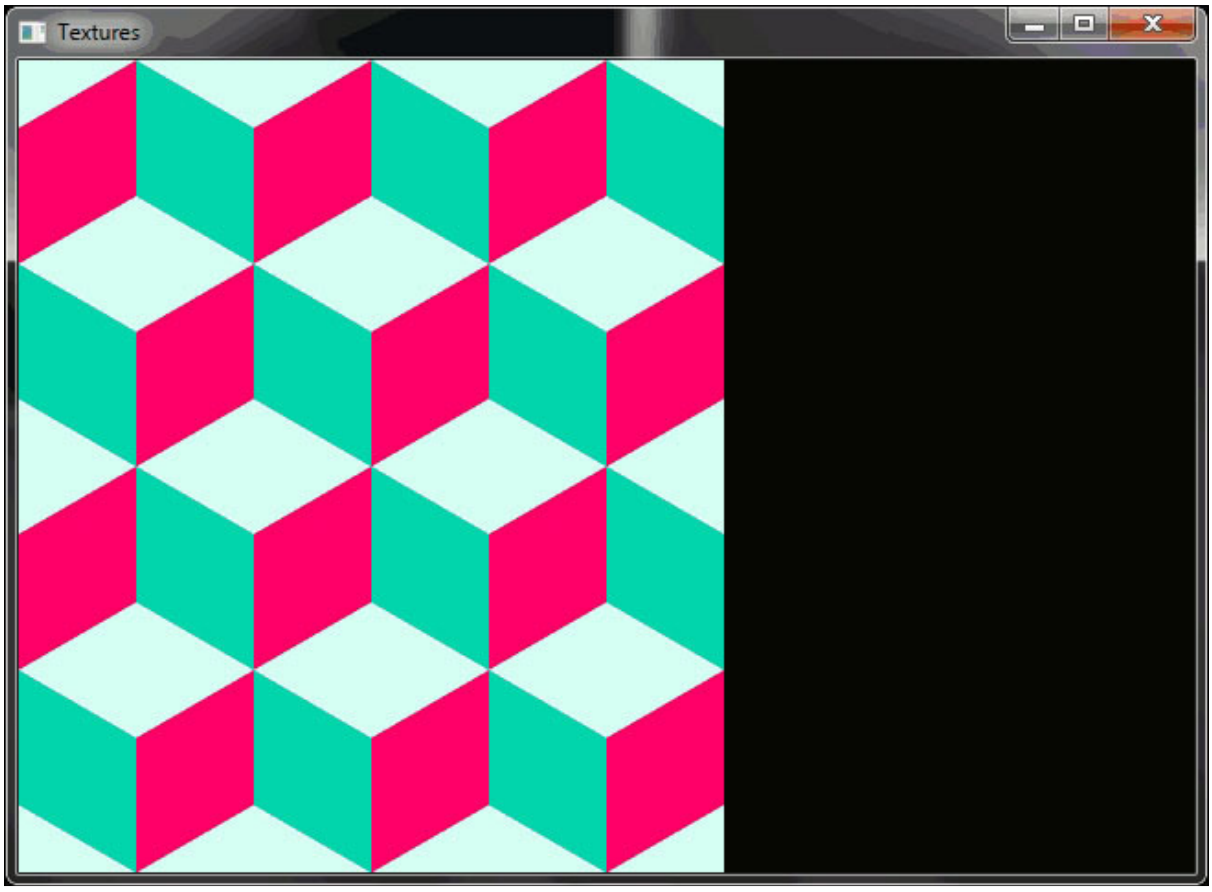
اگر سعی کنیم شکل را با این بافت رندر کنیم ، نتیجه جالب نخواهد بود : بافت کشیده می شود تا روی همه سطح مستطیل را بگیرد . باید کمی بافت را تنظیم کنیم تا آن طوری که ما دوست داریم رفتار کند . تابع `Texture::setRepeated()` وجود دارد که آرگومان بولی می گیرد و اگر با مقدار true فراخوانده شود ، بافت قابل تکرار خواهد شد . اما این کافی نیست . یک مرحله دیگر لازم است .
هنگامی که بافت ها را بر روی سطوح نگاشت و ترسیم می کنیم ، معمولا باید مختصات بافت را برای هر راس از سطح مشخص کنیم . در SFML این کار برای کلاس `Shape` به صورت اتوماتیک انجام می شود . اگر از کتابخانه OpenGL برای رندر یک مربع با یک بافت بر روی آن استفاده کنیم ، باید مختصات بافت را در قالب نرمال سازی شده (0…1; 0…1) مشخص کنیم . SFML از شیوه نرمال سازی شده استفاده نمی کند بلکه از مختصات فضائی پیکسل استفاده می کند .([0… width —1], [0… height —1])
شکل زیر نشان می دهد چطور مختصات بافت بر روی یک سطح نگاشته می شود و سپس بر روی صفحه رندر می شود .



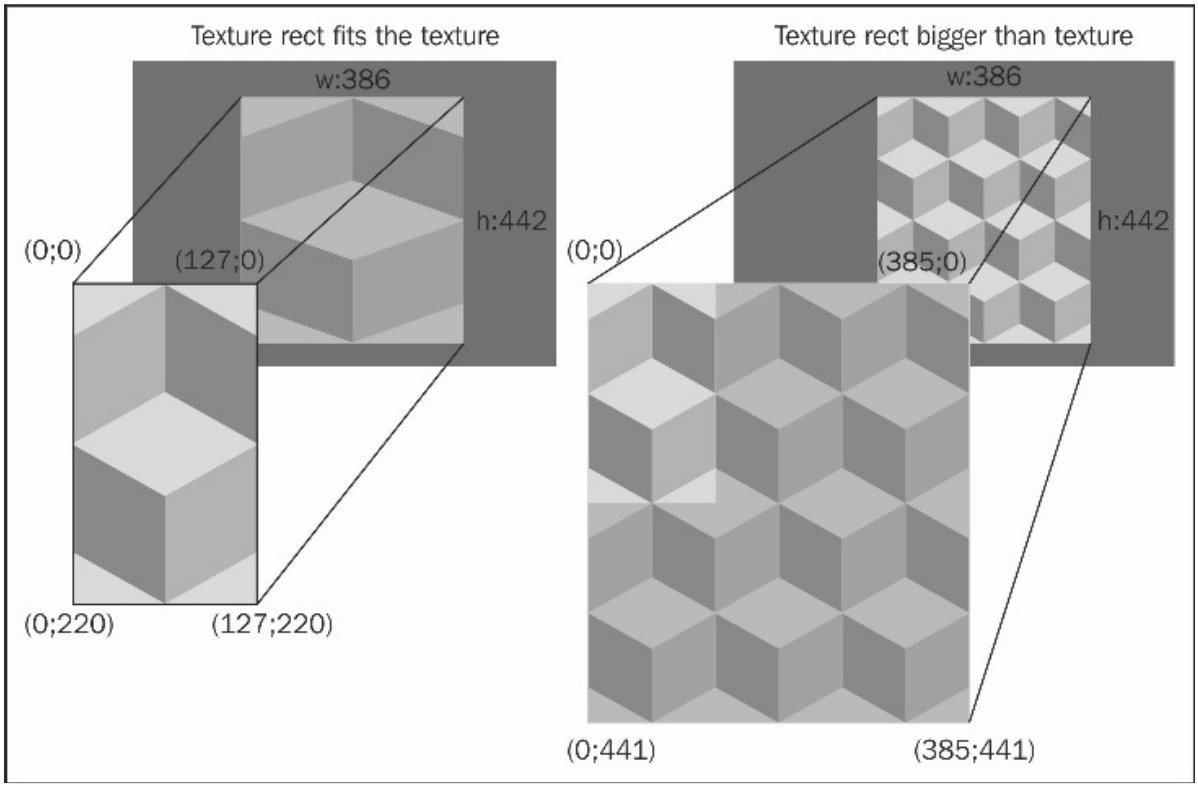
دیدم که هنگامی که سطح (یا شکل) را بزرگتر می کنیم ، چه اتفاقی می افتد : بافت کشیده می شود . ما باید مختصات بافت را طوری تغییر دهیم که بافت را چندین بار بر روی آن سطح تکرار کند . این کار با بزرگتر کردن مستطیل بافت (همه ۴ مختصات بافت در یک ساختار) از خود بافت انجام می شود . کدها را ببینید :

```
sf::Texture texture;
texture.loadFromFile("tile.png");
//Set the texture in repeat mode
texture.setRepeated(true);

sf::RectangleShape rectShape(sf::Vector2f(128 * 3, 221 * 2));
//Bigger texture rectangle than the size of the texture
rectShape.setTextureRect(sf::IntRect(0, 0, 128 * 3, 221 * 2));

rectShape.setTexture(&texture);
```

نتیجه آن چیزی است که می خواستیم :



برای واضح تر شدن مختصات بافت ، شکل زیر نشان می دهد که چطور مستطیل بافت پیش فرض (که کاملا با بافت منطبق است) بر روی یک سطح بزرگتر نگاشته می شود و هنگامی که مستطیل بافت بزرگتر از بافت باشد ، نتیجه چه خواهد بود :

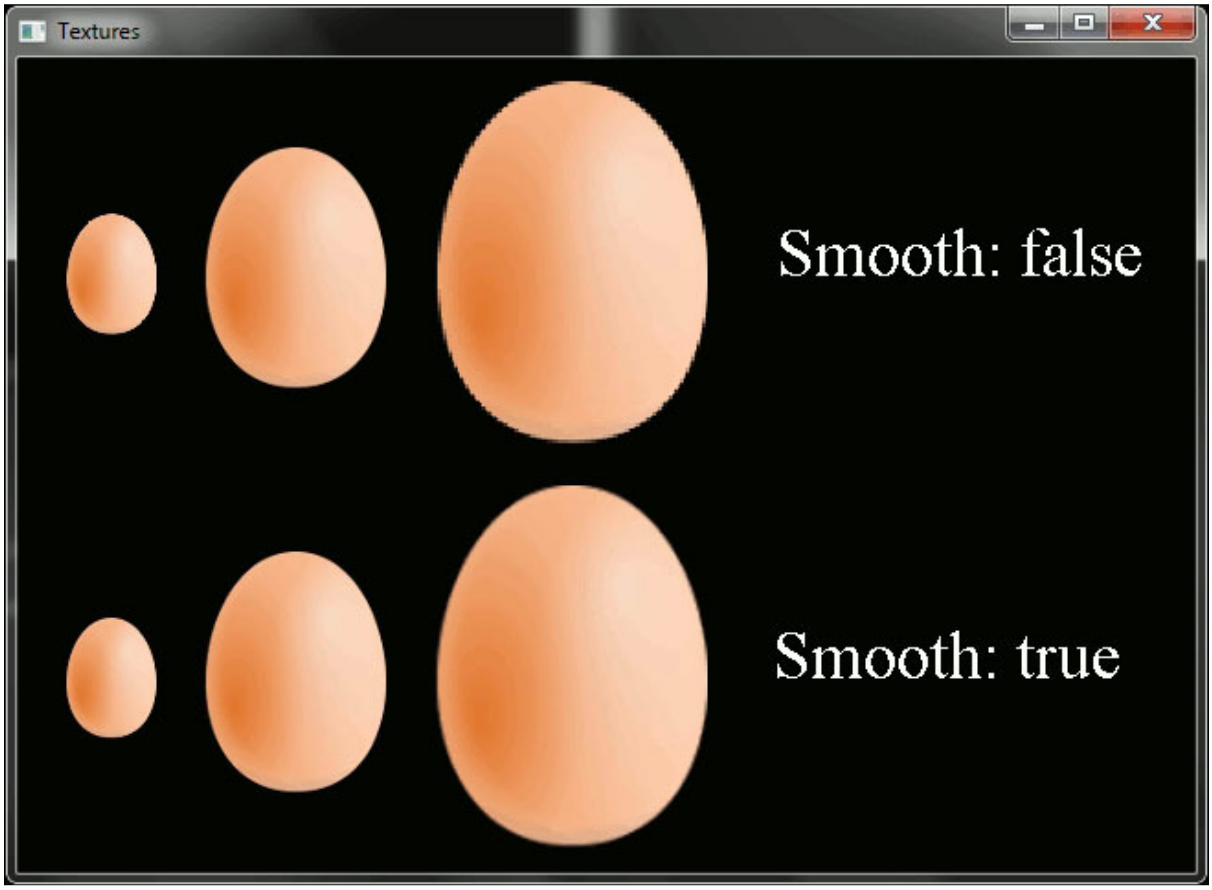


به جز `Texture::setRepeat()` یک مشخصه دیگر وجود دارد که شیوه رندر بافت را دچار تغییر می کند : فیلتر صاف کردن (smooth) که توسط `Texture::setSmooth()` کنترل می شود .

اگر از بافت بر روی سطح با اندازه اصلیش استفاده کنیم ، به این عملکرد نیازی نخواهیم داشت . تابع فیلتر صاف کردن را روی بافت فعال می کند که باعث می شود لبه پیکسل هایش کمتر قابل دیدن شوند . هنگامی که یک پیکسل از بافت نتواند مستقیما بر روی یک پیکسل از صفحه نگاشته شود (به دلیل کوچک و بزرگ شدن و ...) اثر این افکت بهتر قابل دیدن است . برای گرافیک هائی که بافت بر روی سطح با اندازه اصلی رندر می شود ، نباید از این فیلتر استفاده کرد چون باعث لکه لکه شدن بافت می شود . مثال استفاده از این فیلتر :

```
sf::Texture texture;
texture.loadFromFile("myTexture.png");
texture.setSmooth(true);
```

شکل زیر مثالی است که در تصویر بالائی این فیلتر استفاده نشده و در پائینی استفاده شده است :



حالا که می دانیم چطور با بافت ها بازی کنیم ، اجازه دهید درباره sprite ها صحبت کنیم .

sprite چیست ؟

احتمالا اصطلاح sprite را قبلا شنیده اید . sprite سطحی با یک بافت بر روی آن است . اما صبر کنید ، sprite ها کمی متفاوتند . قطع نظر از کلاس `Shape` ، کلاس `Sprite` هم وجود دارد . حال این سوال مطرح می شود که فرق آنها چیست ؟

شکل ها در مقابل sprite ها

مهمترین تفاوت این است که یک sprite همیشه به عنوان یک مستطیل بافت دار رندر می شود . ما می توانیم از شکل ها بدون بافت ها استفاده کنیم ، (فقط تنظیمات مربوط به رنگ داخل و اطراف شکل را انجام دهیم) در حالیکه sprite ها اکیدا نیاز دارند که بافتی به آنها چسبیده باشد . از آنجائی که sprite ها به عنوان مستطیل ها رندر می شوند ، نمی توانیم بخشی از بافت را مثل کاری که با `ConvexShape` انجام دادیم ، ببرُیم .

کلاس `Sprite` تابع `Sprite::setColor()` دارد که شبیه به `Shape::setFillColor()` است . تا زمانیکه شکل یک بافت دارد ، تاثیر هر دو تابع یکسان است . تنها تفاوت این است که اگر sprite بافتی نداشته باشد ، چیزی رندر نخواهد شد در حالیکه شکل با رنگ مشخص شده رندر خواهد شد .

ابعاد sprite توسط بافتش کنترل می شود . در `RectangleShape` اندازه مستطیل را به آنچه می خواستیم تنظیم کردیم ، اما با sprite شکلی وجود ندارد ، فقط بافت است . اگر بخواهیم sprite بزرگتر یا کوچکتر ظاهر شود ، باید مقیاس شئ `Sprite` را تغییر دهیم .

یک سوال مطرح است ؟ چرا باید به جای یک شکل از sprite استفاده کنیم ؟ به نظر می رسد که sprite فقط یک شکل همراه با امکانات کمتر است . علت سادگی آن است . کد زیر را ببینید :

```
//Create a shape with a texture
sf::RectangleShape rectShape(sf::Vector2f(100, 100));
rectShape.setTexture(&texture);

//Create a sprite
sf::Sprite sp(texture);
```

می بینید که ایجاد یک sprite خیلی ساده است . در واقع این مهمترین هدف کلاس Sprite است : رندر یک بافت روی صحنه تا اندازه ای که به سرعت و با آسانی امکان پذیر باشد .

خوب اجازه دهید ببینم که با sprite چه کارهائی می توان انجام داد .

قابلیت تغییر شکل و ترسیم پذیری

کلاس Sprite از دو کلاس Transformable و Drawable استخراج شده است . کلاس Drawable در اصل واسطی است که متد Drawable::draw() را نگه می دارد . همه فرزندانی که این متد را اجرا می کنند ، قادرند خودشان را روی پرده نقاشی (از قبیل RenderWindow) ترسیم کنند . کلاس Transformable موقعیت ، دوران ، مقیاس گذاری و مبداء و تابع های accessor/mutator را برای این فیلدها نگه می دارد . بعضی از آنها عبارتند از :

Transformable::setPosition(), Transformable::getPosition(), Transformable::move()

این تابع ها ممکن است آشنا باشند . چون با آنها در کلاس Shape روبه رو شده ایم . در حقیقت کلاس Shape از Drawable و Transformable ارث می برد . این یعنی می توانیم sprite را به همان شیوه یک شکل ، دستکاری کنیم و می توانیم با فراخوان RenderWindow::draw() یک sprite را ترسیم کنیم . اگر با دقت به RenderWindow::draw() نگاه کنیم ، می بینیم که یک آرگومان Drawable می گیرد نه یک شکل یاsprite . این یعنی هر کلاسی که از Drawable استخراج شود ، می تواند برای ترسیم به یک پنجره پاس داده شود . ما همچنین می توانیم کلاس هائی ایجاد کنیم که از Transformable و/یا Drawable ارث ببرند . مثلا اگر یک sprite دایره بهینه شده بخواهیم ، می توانیم CircleSprite ایجاد کنیم و یک متد draw برای آن پیاده سازی کنیم . سپس پاس دادن یک شئ CircleSprite به فراخوان draw برای RenderWindow اسان خواهد بود .

آخریم مطلب درباره sprite

Sprite چیزهای بیشتری دارد . تمام خصوصیات Texture (فیلتر صاف کردن و تکرار) روی sprite هم کار می کند . برای تکرار یک بافت ، باید مستطیل بافت (نگهدارنده مختصات بافت است) را مثل کاری که با یک شکل کردیم ، تغییر دهیم . برای این کار همان تابع Sprite::setTextureRect() هم در کلاس Sprite وجود دارد . Sprite::getLocalBounds() و Sprite::getGlobalBounds() هم وجود دارند . هر دو جعبه محصور { **Axis-Aligned Bounding Box (AABB)** } یک sprite را محاسبه می کنند . محدوده محلی برای sprite محلی است : آنها در تغییر شکل به حساب نمی آیند . از طرف دیگر محدوده جهانی ، sprite را با موقعیت ، مقیاس گذاری ، دوران و مبداء ؛ تغییر شکل داده و سپس مستطیل را به دست می آورد . همان طور که با شکل ها انجام دادیم ، اینها می توانند برای تشخیص برخورد ابتدائی استفاده شوند چون FloatRect (توسط این دو تابع بازگش داده می شود) یک تابع FloatRect::intersects() در آن دارد .

مدیریت منابع

هنگامی که یک بازی یا برنامه چند رسانه ای می سازیم ، مهم است که منابع را به درستی و کارامد مدیریت کنیم . مطمئن شویم که منابع به صورت ناگهانی تخریب نمی شوند و یا یک بافت را دوبار بارگذاری نمی کنیم . در این بخش درباره ساختن یک پایه و اساس برای یک مدیریت کننده منبع صحبت می کنیم ، که می توانید در برنامه هایتان از آن استفاده کنید .

اجازه دهید با یک موضوع ساده شروع کنیم . اشیاء رو حافظه stack هنگامی که از حوزه خارج شوند ، تخریب خواهند شد . در بعضی زبان ها ، تمام کلاس ها روی حافظه heap تخصیص پیدا می کنند و این مشکلی نیست . اما در C++ اگر بخواهیم اشیاء را روی stack نگه داریم ، که مدیریت حافظه را ساده تر می کند ، باید اطمینان حاصل کنیم که تا زمانیکه آنها مورد استفاده قرار می گیرند ، زنده باقی خواهند ماند . این یک مثال است که در آن ، به محض اینکه تابع خارج شود ، بافت نابود می شود :

```
sf::Sprite createSprite(std::string const& filename)
{
    sf::Texture texture;
    texture.loadFromFile(filename);

    //This is bad. As soon as the function returns
    //the texture will be destroyed
    return sf::Sprite(texture);
}
```

اگر sprite را ترسیم کنیم ، می بینیم که فقط یک مستطیل سفید ترسیم می شود و بافت دیده نخواهد شد . برای مدیریت چرخه زندگی منابع در زمان اجرای برنامه ، داشتن یک مدیریت کننده اختصاصی مفید است . به همین دلیل اجازه دهید یک کلاس `AssetManager` ایجاد کنیم که همه منابع را در برنامه ما بارگذاری ، نگهداری و تخریب می کند . فایل هدر آن به شکل زیر است :

AssetManager.h

```
#ifndef ASSET_MANAGER_H
#define ASSET_MANAGER_H

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

class AssetManager
{
public:
    AssetManager();

    static sf::Texture& GetTexture(std::string const& filename);

private:
    std::map<std::string, sf::Texture> m_Textures;

    //AssetManager is a singleton, so only one instance can exist at a time
    //sInstacne holds a static pointer to the single manager instance
    static AssetManager* sInstance;
};

#endif
```

کلاس AssetManager یک کلاس singleton است (فقط یک نمونه اجازه دارد که موجود باشد) و این موضوع علت این است که یک اشاره گر استاتیک به خودش دارد . در واقع همه تابع های استاتیک این کلاس ، از این اشاره گر به عنوان راهی برای ارجاع به نمونه AssetManager استفاده می کنند . این نمونه درست بعد از مقداردهی اولیه پنجره ایجاد می شود و هنگامی که برنامه خارج می شود ، تخریب خواهد شد . ساختن مدیریت کننده منابع به این شیوه مفید است چون می توانیم AssetManager::GetTexture() را از هر جای برنامه فرابخوانیم بدون اینکه ارجاعی به شئ AssetManager داشته باشیم .

صرف نظر از اشاره گر ، کلاس یک map از بافت ها را نگه می دارد و راه رسیدن به اعضای این map از طریق تابع AssetManager::GetTexture() است . یک map مجموعه ای از مقادیر و کلیدهای یکتاست که هر کلید یک مقدار وابسته دارد . در مورد ما ، کلید های string و مقادیر Texture داریم . کلیدها اسم فایل بافت ها را نگه می دارند و مقادیر ، اشیاء Texture را نگه می دارند . با این شیوه ، به اسانی می توانیم چک کنیم که آیا یک اسم فایل در map وجود دارد و اگر نباشد آن را اضافه کنیم .

سازنده را در فایل منبع کد ببینید :

AssetManager.cpp

```cpp
#include "AssetManager.h"
#include <assert.h>

AssetManager* AssetManager::sInstance = nullptr;

AssetManager::AssetManager()
{
    //Only allow one AssetManager to exist
    //Otherwise throw an exception
    assert(sInstance == nullptr);
    sInstance = this;
}
```

اولین خط بعد از دستور #include ، اشاره گر استاتیک sInstance را با nullptr (که null یا 0 است) مقداردهی اولیه می کند . نصب اشاره گر با nullptr درست بعد از اینکه آن را اعلام کردیم ، عادت خوبی است . این کاری است که در سازنده انجام داده ایم . ما ماکروی assert را فراخوانده ایم که بررسی می کند که آیا یک عبارت true است یا خیر ؟ اگر این طور باشد هیچ اتفاقی نمی افتد ، اما اگر عبارت false باشد ، ماکرو برنامه را قطع می کند . این بررسی از تولید بیش از یک نمونه از کلاس جلوگیری می کند که دقیقا چیزی است که ما می خواهیم . بعد از اینکه مطمئن شدیم فقط یک نمونه وجود دارد ، اشاره گر استاتیک را با this نصب می کنیم .

پیاده سازی AssetManager::GetTexture() را ببینید :

```cpp
sf::Texture& AssetManager::GetTexture(std::string const& filename)
{
    auto& texMap = sInstance->m_Textures;

    //See if the texture is already loaded
    auto pairFound = texMap.find(filename);
    //If yes, return the texture
    if (pairFound != texMap.end())
    {
        return pairFound->second;
    }
    else //Else, load the texture and return it
    {
        //Create an element in the texture map
        auto& texture = texMap[filename];
        texture.loadFromFile(filename);
        return texture;
    }
}
```

توجه کنید که این تابع استاتیک است ، یعنی map را باید از طریق اشاره گر استاتیک `sInstance` به دست آوریم . بعد از این ، توسط فراخوان `map<>::find()` بررسی می کنیم که آیا بافت درخواست شده قبلا بارگذاری شده یا خیر ؟ تابع یک iterator بازگشت می دهد که به جفت پیدا شده اشاره دارد . اگر هیچ جفتی پیدا نشود ، آن به `map<>::end()` اشاره خواهد داشت . اگر اسم فایل پیدا شود ، ما شئ `texture` را در داخل جفت (دومین عنصر) بازگشت می دهیم . اگر اسم فایل پیدا نشود ، یک شکاف در map برای آن بافت ایجاد کرده و بافت را از آرگومان اسم فایل ، بارگذاری می کنیم . این همه کاری است که `AssetManager` انجام می دهد . آن فقط یک map از بافت ها را نگه می دارد و یک واسط برای دسترسی به آنها دارد . با پیشرفت در طول کتاب ، منابع بیشتری مثل فونت ها ، شیدر ها ، صدا ها و موسیقی را به این مدیریت کننده اضافه خواهیم کرد .

و در نهایت چگونگی مقداردهی اولیه و استفاده از `AssetManager` :

Main.cpp

```cpp
#include <SFML/Graphics.hpp>
#include "AssetManager.h"

int main()
{
    sf::RenderWindow window(sf::VideoMode(640, 480), "AssetManager");
    AssetManager manager;

    //Create sprites
    sf::Sprite sprite1 = sf::Sprite(AssetManager::GetTexture("myTexture1.png"));
    sf::Sprite sprite2 = sf::Sprite(AssetManager::GetTexture("myTexture2.png"));
    sf::Sprite sprite3 = sf::Sprite(AssetManager::GetTexture("myTexture1.png"));

    while (window.isOpen())
    {
        //Game loop
    }

    //After main() returns, the manager is destroyed
    return 0;
}
```

در مثال قبلی ، دو فراخوان ابتدائی به `AssetManager::GetTexture()` بافت های جدید را بارگذاری و ذخیره می کند اما در آخرین فراخوان (`sprite3`) ، مدیریت کننده تنها بافت ذخیره شده را بازگشت می دهد و زمان و حافظه مورد نیاز برای بارگذاری مجدد آن را ذخیره خواهد کرد .

خلاصه

در این فصل یاد گرفتیم که SFML در خصوص بافت ها و تصاویر چه امکاناتی را فراهم کرده است . دیدیم که چطور به درستی یک بافت را ایجاد و بارگذاری کنیم و نیز چندین شیوه برای رندر آن بر روی صفحه بررسی شد . هر چند بافت ها از آنچه که دیدیم ، کاربرد بیشتری دارند ، این شروعی مناسب بود .
در فصل بعد ، در خصوص انیمیشن و زمانبندی صحبت می کنیم .

فصل ۳ : انیمیت کردن Sprite ها

اگر می خواهیم این حس را ایجاد کنیم که یک شئ به صورت طبیعی در حال انجام کاری است ، انیمیشن کاملا مهم خواهد بود . مثلا یک sprite از یک آتش را در نظر بگیرید . اگر فقط یک تصویر از یک شعله موجود باشد ، به نظر خواهد رسید که آتش در حال مشتعل شدن و سوختن نیست . اما اگر چندین تصویر موجود باشد ، این خیال ایجاد می شود که آتش وجود دارد . این چیزی است که می خواهیم در این فصل درباره آن بحث کنیم . قبل از این کار ، باید دنیای شبیه سازی های بر مبنای زمان را بررسی کنیم ، چون برای اینکه انیمیشن های ما به درستی کار کنند ، این نیاز خواهد بود .
در این فصل مطالب زیر مورد بحث قرار می گیرند :

گرفتن زمان
انیمیت کردن sprite ها
ساختن یک انیمیت کننده

گرفتن زمان

زمان خیلی مهم است . اما فرض کنید دوست ما "جیمی" اعتقاد دارد که زمان اهمیت زیادی ندارد . یک روز او نشست و یک بازی چندبازیکنه مسابقه ای ساخت که ماشین ها در هر فریم ، دقیقا یک پیکسل حرکت می کردند . او شاد بود که نتیجه بازی را روی کامپیوترش می دید . جیمی بازی را به دوستش "تیمی" داد که به تازگی یک CPU جدید قدرتمند برای کامپیوترش خریده بود . آنها بدون هیچ مشکلی به مسابقه پرداختند ، اما به محض اینکه چراغ سبز شد ، "تیمی" به سرعت از "جیمی" پیش افتاد و او را پشت سر گذاشت . بعدا "جیمی" فهمید که کامپیوتر "تیمی" کدهای بازی را خیلی سریع تر از کامپیوتر خودش اجرا می کند و در نتیجه ، ماشین او کند تر است . اگر در شبیه سازی ها ، زمان نادیده گرفته شود ، در کامپیوترهای مختلف ، فریم ها با سرعت های متفاوتی اجرا می شوند و در نتیجه شبیه سازی ، متفاوت به نظر خواهد رسید . کد "جیمی" قبل از اینکه آن را اصلاح کند به شکل زیر بود :

```
//Car speed = 1 pixel per frame
const float carSpeed = 1.f;

//Advance the car
carSprite.move(carSpeed, 0);
```

این کد در هر فریم اجرا می شود و بسته به سرعت CPU و GPU دارد . اگر یک کامپیوتر این کد را با سرعت 30 فریم در ثانیه اجرا کند ، در نتیجه در یک ثانیه 30 پیکسل جابه جا می شود . در کامپیووتر دیگری که با سرعت 60 فریم در ثانیه این کد اجرا شود ، برای یک ثانیه ، دو برابر مسافت جابه جائی خواهیم داشت . تقریبا همیشه ، فریم ها مقدار زمان متفاوتی برای اجرا در یک کامپیوتر نیاز دارند . مشخص است که ما یک مشکل خواهیم داشت . در سیستم های مختلف ، منطق بازی در سرعت های متفاوتی اجرا می شود . خوشبختانه راه حل خیلی ساده است . راه حل این است که از زمان سپری شده بین فریم ها برای به روز کردن منطق بازی استفاده کنیم . کد زیر همان مثال قبلی است که وابسته به زمان است :

```
//Seconds elapsed since last frame
float deltaTime;

/* Calculate deltaTime here */

//Change the car speed to pixels per second - 30 is reasonable
const float carSpeed = 30.f;

//Advance the car
carSprite.move(carSpeed * deltaTime, 0);
```

تغییر زیادی نکرده است ؟ ما یک متغییر (deltaTime) اضافه کرده ایم که مدت زمان آخرین فریم (به ثانیه) را نگه می دارد .
هنگامی که این متغییر را به Sprite::move() پاس می دهیم ، می گوئیم که می خواهیم این ماشین را با سرعت carSpeed پیکسل در هر ثانیه در جهت افقی حرکت دهیم . اما چطور باید این زمان سپری شده را به دست آورد ؟

sf::Clock و sf::Time

مثل همیشه SFML دوست ماست و دو کلاس دارد که خیلی خوب با زمان کار می کند . Time کلاسی است که یک مدت زمان را نگه می دارد . این یعنی به ما چیزی درباره زمان جاری یا زمان سپری شده از وقتی که برنامه شروع شده است ، نمی گوید . آن فقط متغییری دارد که یک مقدار زمان را نگه می دارد . این می تواند ۵ میکروثانیه باشد یا ۱۰ ماه یا هر چیزی که نشان دهنده یک دوره زمانی است .
ما می توانیم از تابع های sf::milliseconds() , sf::seconds() و sf::microseconds() برای ساختن یک شئ زمان از ثانیه ، میلی ثانیه و میکروثانیه استفاده کنیم . وقتی این شئ را داشتیم ، می توانیم از اپراتورهای ریاضی (جمع ، تفریق و مقایسه) بر روی آن استفاده کنیم . سپس می توانیم با فراخوان تابع های Time::asSeconds() ,
Time::asMilliseconds() و Time::asMicroseconds() شئ Time را به ثانیه ، میلی ثانیه و میکروثانیه تبدیل کنیم . مثالی از چگونگی کارکرد کلاس Time :

```
sf::Time time = sf::seconds(5) + sf::milliseconds(100);
if (time > sf::seconds(5.09))
    std::cout << "It works";
```

کلاس Time برای ذخیره زمان مناسب است اما راهی برای گرفتن آن برای ما فراهم نکرده است . کلاس Clock واسطی برای اندازه گرفتن زمان سپری شده توسط ساعت سیستم عامل فراهم کرده است . استفاده از آن آسان است :

```
sf::Clock clock;

//Run heavy CPU code

sf::Time timePassed = clock.getElapsedTime();
```

قطع نظر از Clock::getElapsedTime() کلاس Clock تابع دیگری دارد که Clock::restart() نام دارد که زمان سپری شده را بازگشت می دهد و ساعت را در همان زمان راه اندازی مجدد می کند .
با در نظر گرفتن این مطالب ، حالا باید اندازه گرفتن زمان فریم واضح باشد . ما در خارج حلقه بازی یک متغییر Clock مقداردهی اولیه می کنیم ، و در شروع فریم ، زمان سپری شده را گرفته و ساعت را راه اندازی مجدد می کنیم . این به ما مقدار زمانی را که فریم قبلی

گرفته است ، می دهد بنابراین می توانیم از آن برای پیش بردن اشیائمان استفاده کنیم . مقدار زمان بین شروع آخرین فریم و شروع فریم جاری معمولا `deltaTime` (یا به طور خلاصه `dt`) گفته می شود . حالا کدها را ببینید :

```
sf::Time deltaTime;
sf::Clock clock;
while (window.isOpen())
{
    //Returns the elapsed time and restarts the clock
    deltaTime = clock.restart();
    float dtAsSeconds = deltaTime.asSeconds(); //Delta time as seconds

    //Handle input

    //Update frame

    //Render frame
}
```

حالا که می دانیم چطور زمان را بگیریم ، باید اطمینان حاصل کنیم که هرجا که منطق بازی به آن نیاز داشت ، از آن استفاده کنیم : در هر حرکت کوتاه ، انیمیشن ، رویدادهای بر پایه زمان (تخریب یک شئ پس از *N* ثانیه) و ... مثال هائی از منطق بازی است که به این زمان نیاز دارد . چون ما زمان بین فریم ها را می دانیم ، می توانیم برای اندازه گرفتن زمان در خارج ساختار فریم ، زمان را در متغییر متفاوتی انباشته و ذخیره کنیم . کد زیر مثالی است که در آن می خواهیم پنجره را 5 ثانیه بعد از باز شدنش ، ببندیم :

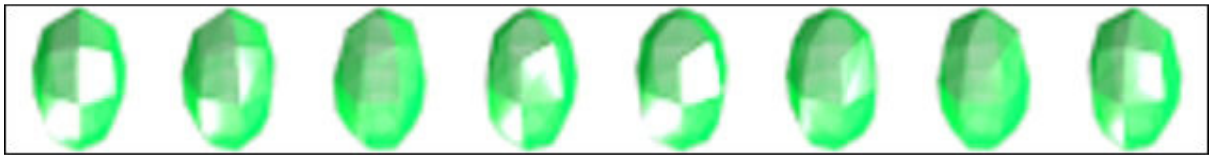
```
sf::Time elapsedTime;
sf::Clock clock;
while (window.isOpen())
{
    sf::Time deltaTime = clock.restart();
    //Accumulate time with each frame
    elapsedTime += deltaTime;

    if (elapsedTime > sf::seconds(5))
        window.close();
}
```

چون ما `Window::pollEvent()` را در حلقه فرانخوانده ایم ، پنجره نمی تواند هیچ رویدادی را از کاربر پردازش کند (دریافت فوکوس ، حرکت دادن ، تغییر اندازه و ...) اما این اجرای منطقی را که فراهم کردیم یعنی بسته شدن پنجره بعد از یک مدت زمان مشخص ، متوقف نمی کند .
حالا با داشتن زمان ، می توانیم با ایمنی به سمت انیمیشن برویم .

Sprite ها در عمل

انیمیشن ها در قالب های زیادی موجود هستند . یک شیوه برای انیمیشن ، ترسیم یک سلسله از تصاویر پشت سر هم که به مقدار کمی با هم تفاوت دارند و نمایش آنها روی صفحه یکی پس از دیگری است . اگر چه این شیوه هنوز خیلی کاربرد دارد ، راه های زیباتری هم وجود دارد . مثلا ترسیم (یا مدل سازی به صورت سه بعدی) فقط اعضای بدن یک کاراکتر و سپس انیمیت کردن تکنیکی است که زمان زیادی را برای آرتیست ها ذخیره می کند . این همچنین نتیجه ای روان تر تولید می کند چون نباید همه فریم های انیمیشن ترسیم شود .
در این کتاب قصد داریم شیوه قدیمی را بررسی کنیم ، چون راه حل آسانتری برای برنامه نویسان است و در بسیاری از موارد ، برای دادن زندگی به هر sprite ، کافی است .

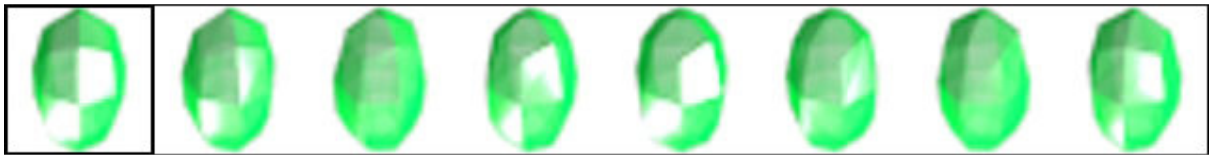
همانطور که گفته شد ، شیوه قدیمی انیمیشن شامل گروهی از تصاویر است که باید در طول زمان تغییر پیدا کنند . برای مثال ، از یک کریستال استفاده می کنیم که حول مرکزش می چرخد . معمولا یک انیمیشن در یک فایل (به عنوان صفحه **sprite**) نگهداری می شود که هر فریم از انیمیشن ذخیره شده و در اغلب موارد ، هر فریم یک اندازه دارد : اندازه شئ . در مثال ما ، sprite اندازه 32 x 32 پیکسل دارد و دارای ۸ فریم است که در یک ثانیه پخش می شود . صفحه sprite به شکل زیر است :



کد زیر آماده سازی انیمیشن ما را نشان می دهد :

```
sf::Vector2i spriteSize(32, 32);
sf::Sprite sprite(AssetManager::GetTexture("spriteSheet.png"));
//Set the sprite image to the first frame of the animation
sprite.setTextureRect(sf::IntRect(0, 0, spriteSize.x, spriteSize.y));

int framesNum = 8; //Animation consists of 8 frames
float animationDuration = 1; //1 second
```

اول از همه توجه کنید که ما از کلاس `AssetManager` فصل ۲ برای بارگذاری صفحه sprite استفاده می کنیم . خط بعدی ، مستطیل بافت sprite را اولین تصویر موجود در صفحه قرار می دهد . منظور ما از بافت صفحه sprite را ببینید :



سپس ، این مستطیل بافت را لحظه به لحظه حرکت می دهیم تا چرخش کریستال ، شبیه سازی شود . در کد قبلی تعداد فریم ها را ۸ قرار دادیم ، (به تعدادی که در صفحه sprite وجود دارد) و زمان انیمیشن را در مجموع یک ثانیه قرار دایم یعنی هر فریم تقریبا 0.125 ثانیه می ماند (مدت انیمیشن تقسیم بر تعداد فریم ها) . حالا می دانیم که چه کاری می خواهیم انجام دهیم . پس اجازه دهید شروع کنیم :

```
while (window.isOpen())
{
    //Returns the elapsed time and restarts the clock
    sf::Time deltaTime = clock.restart();

    //Handle input

    //Accumulate time with each frame
    elapsedTime += deltaTime;
    float timeAsSeconds = elapsedTime.asSeconds();

    //Get the current animation frame
    int animFrame = static_cast<int>((timeAsSeconds / animationDuration) * framesNum) % framesNum;
    //Set the texture rectangle, depending on the frame
    sprite.setTextureRect(sf::IntRect(animFrame * spriteSize.x, 0, spriteSize.x, spriteSize.y));

    //Render frame
}
```

در این کد ، ما ابتدا زمان سپری شده از آخرین فریم اندازه گرفتیم و سپس آن را به زمان انباشته شده اضافه می کنیم . دو خط آخر کد ها ، در واقع همه کار را انجام می دهد . اولی در نگاه اول خیلی ترسناک است ، اما آن یک راه ساده برای انتخاب فریم درست است و بر این اساس است که چقدر زمان سپری شده و انیمشن چقدر طولانی است . فرمول `timeAsSeconds / animationDuration` زمان را نسبت به مدت انیمیشن به ما می دهد . فرض کنید 0.4 ثانیه سپری شده و مدت انیمیشن ما ۱ ثانیه است . این به ما 0.4 ثانیه می دهد . ضرب این 0.4 ثانیه در تعداد فریم ها ، نتیجه زیر را دارد :

0.4 * 8 = 3.2

این به ما فریمی را که در لحظه باید باشیم و اینکه تا کی باید آنجا باشیم را می دهد . شاخص فریم جاری ، بخش کامل 3.2 (یعنی 3) است و بخش کسری (0.2) مدت زمانی است که باید در این فریم بمانیم . در این مورد ، ما فقط فریم جاری را می خواهیم ، بنابراین آن را با تبدیل کل عبارت به `int` به دست می آوریم . اگر عدد مثبت باشد ، این کار آن را به سمت پائین گرد می کند . آخرین بخش ، `% frameNum` آنجاست تا هنگامی که انیمیشن به آخرین فریمش رسید ، انیمیشن را مجددا شروع کند . بنابراین هنگامی که 2.3 ثانیه سپری شد ، نتیجه زیر را خواهیم داشت :

2.3 * 8 = 18.4

ما ۱۹ فریم برای نمایش نداریم ، بنابراین ، ما فریمی را نشان می دهیم که متناظر با مقیاس ما باشد [0...7] . در این نمونه :

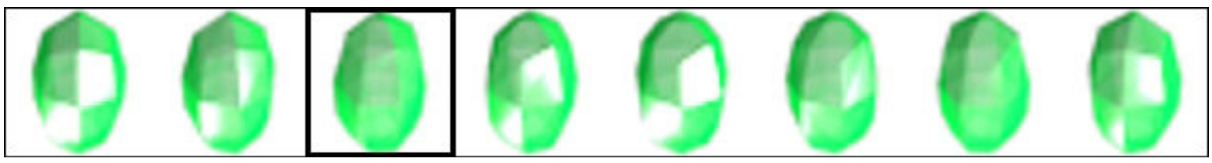
18 / 8 = 2 (و ۲ باقیمانده)

چون اپراتور `%` باقیمانده تقسیم را می گیرد ، ما فریم با شاخص 2 را نشان می دهیم که سومین فریم است . (ما به عنوان برنامه نویس ، شمارش را از صفر شروع کردیم . یادتان هست ؟)

آخرین خط کد ، مستطیل بافت را فریم جاری قرار می دهد . فرایند کاملا سرراست است ، چون ما فقط فریم هائی روی محور X داریم ، نیاز به نگرانی درباره مختصات Y مستطیل نداریم ، و بنابراین آن را صفر قرار می دهیم . X توسط محاسبه `animFrame * spriteSize.x` به دست می آید که فریم جاری را عرض فریم ضرب می کند . در مورد مثال ما ، فریم جاری دو بود و عرض فریم هم 32 . پس داریم :

2 * 32 = 64

مستطیل بافت به شکل زیر خواهد بود :



آخرین کاری که باید انجام دهیم ، رندر sprite است . اگر همه چیز به روانی انجام شود ، ما باید یک کریستال چرخان با ۸ فریم روی صفحه داشته باشیم . با این تکنیک می توانیم همه انواع sprite ها را انیمیت کنیم بدون توجه به اینکه چند فریم دارد یا اینکه انیمیشن چقدر طولانی است . یک مشکل با این شیوه وجود دارد : کدها خیلی آشفته به نظر می رسد ، و فقط برای یک انیمیشن مفید است . اگر چندین انیمیشن برای یک sprite بخواهیم (مثلا چرخش کریستال در جهت عمودی) و بخواهیم این انیمیشن ها را عوض کنیم ، چه

کار باید کرد ؟ در حال حاضر باید همه کدها را برای انیمیشن و هر sprite انیمیت شده ، تکرار کنیم . در بخش بعدی ، با ساختن یک سیستم انیمیشن کامل ، در خصوص چگونگی دوری کردن از این مشکلات صحبت خواهیم کرد .

ساختن یک انیمیت کننده

قبل از شروع انجام کار مهم است که بدانیم دقیقا چه کاری می خواهیم انجام دهیم بنابراین اجازه دهید مشخصات انیمیت کننده را لیست کنیم :

انیمیت کننده باید یک sprite را از یک شئ یا چندین شئ بافت انیمیت کند .
انیمیت کننده باید انیمیشن ها را با متغییر مدت زمان و تعداد فریم ها پشتیبانی کند .
انیمیت کننده باید چندین انیمیشن را نگه دارد .
انیمیت کننده باید بتواند انیمیشن ها را عوض کند .
هر sprite باید شئ انیمیت کننده مخصوص به خودش داشته باشد .
استفاده از انیمیت کننده باید آسان باشد .
انیمیت کننده باید بتواند تولید مستطیل بافت را به صورت اتوماتیک انجام دهد .

چون می خواهیم چندین انیمیشن را در هر انیمیت کننده اجرا کنیم ، باید یک ساختار انیمیشن ایجاد کنیم که هر انیمیشن را به درستی نگه دارد . ما به جای کلاس از `struct` استفاده می کنیم تا اندازه کد کاهش پیدا کند . هر انیمیشن باید یک مدت زمان پخش ، لیستی از فریم ها ، یک بافت (مورد استفاده توسط انیمیشن) ، اطلاعات تکرار (آیا انیمیشن تکرار می شود ؟) و یک اسم داشته باشد که به عنوان ارجاع به آن انیمیشن مورد استفاده قرار می گیرد . بر اساس این طراحی ساختار انیمیشن ما به شکل زیر خواهد بود :

```
struct Animation
{
    std::string m_Name;
    std::string m_TextureName;
    std::vector<sf::IntRect> m_Frames;
    sf::Time m_Duration;
    bool m_Looping;

    Animation(std::string const& name, std::string const& textureName,
        sf::Time const& duration, bool looping)
        : m_Name(name), m_TextureName(textureName),
        m_Duration(duration), m_Looping(looping)
    { }

    //Adds frames horizontally
    void AddFrames(sf::Vector2i const& startFrom,
        sf::Vector2i const& frameSize, unsigned int frames)
    {
        sf::Vector2i current = startFrom;
        for (unsigned int i = 0; i < frames; i++)
        {
            //Add current frame from position and frame size
            m_Frames.push_back(sf::IntRect(current.x, current.y, frameSize.x, frameSize.y));
            //Advance current frame horizontally
            current.x += frameSize.x;
        }
    }
};
```

متد `Animation::AddFrames()` را توجه کنید . این متد کار ما را در هنگام تولید انیمیشن آسان می کند . آن موقعیت ، اندازه و تعداد فریم ها را می گیرد و از آن موقعیت به صورت افقی می چرخد تا فریم ها را به داخل vector با نام `m_Frames` اضافه کند . این در اصل یک راه میان بر است که می توانیم به جای اینکه فریم ها را یکی یکی اضافه کنیم ، از این راه استفاده کنیم .
حالا می توانیم به سراغ انیمیت کننده `Animator` برویم که به طور واقعی از انیمیشن ها برای انیمیت کردن sprite ما استفاده می کند . اولین کاری که انیمیت کننده باید انجام دهد ، تولید و ذخیره انیمیشن ها برای استفاده های بعدی است . همچنین چون انیمیشن ها وابسته به زمان هستند ، انیمیت کننده باید یک متد `Animator::Update()` داشته باشد که در هر فریم با زمان سپری شده فراخوانده می شود . همچنین راهی برای عوض کردن انیمیشن ها نیاز است .
با در نظر داشتن این توضیحات ، داده های خصوصی ما به شکل زیر خواهد بود :

```
private:
    //Returns the animation with the passed name
    //Returns nullptr if no such animation is found
    Animator::Animation* FindAnimation(std::string const& name);

    void SwitchAnimation(Animator::Animation* animation);

    //Reference to the sprite
    sf::Sprite& m_Sprite;
    sf::Time m_CurrentTime;
    std::list<Animator::Animation> m_Animations;
    Animator::Animation* m_CurrentAnimation;

};
```

فعلا `Animator::FindAnimation()` و `Animator::SwitchAnimation()` را نادیده بگیرید ، بعدا دوباره به اینها برمی گردیم . چیزی که فعلا درباره اش صحبت می کنیم ، داده ها هستند . همین طور که می بینید ، هیچ نمونه ای از sprite در `Animator` نیست بلکه فقط یک ارجاع وجود دارد . این یعنی sprite در خارج کلاس ساخته می شود و سپس به سازنده `Animator` پاس داده می شود .
غیر از ارجاع `Sprite` ، یک شمارشگر انباشتگی زمان هم وجود دارد (همان که در بخش قبلی این فصل استفاده کردیم) ، همچنین لیستی از انیمیشن ها ، و اشاره گری به انیمیشن جاری که در حال پخش است . توجه کنید که ما از یک `vector` برای ذخیره انیمیشن ها استفاده نکردیم بلکه از یک list استفاده شد . این اغلب مخصوص C++ است اما شما نمی توانید اشاره گرها و ارجاع ها به عناصر را در `vector` نگه دارید : یک بار که شروع به اضافه کردن یا حذف عناصر از آن کردیم ، آنها نامعتبر خواهند شد . کلاس `list` این مشکل را ندارد چون پیاده سازی آن طوری است که اشاره گرها و ارجاع ها معتبر می مانند حتی پس از حذف یا اضافه عناصر از آن .
حالا اجازه دهید کارکردی را که کلاس `Animator` فراهم کرده است بررسی کنیم . این کار را با نگاه به توابع عضو عمومی آن انجام می دهیم :

```
public:
    struct Animation{ ... };

    Animator(sf::Sprite& sprite);

    Animator::Animation& CreateAnimation(std::string const& name,
        std::string const& textureName, sf::Time const& duration, bool loop = false);

    void Update(sf::Time const& dt);

    //Returns if the switch was successful
    bool SwitchAnimation(std::string const& name);

    std::string GetCurrentAnimationName() const;
```

اولین چیزی که چشمان شما را تیز می کند ساختار Animation است . این همان ساختاری است که قبلا در این فصل درباره آن صحبت کردیم . آن داخل کلاس Animator اعلام شده است چون هر دو کلاس به هم متصل هستند . به علاوه کلاس Animation در خارج از Animator استفاده زیادی ندارد . خط بعدی محتوی اعلام سازنده است که یک ارجاع به Sprite می خواهد . این ارجاعی است که عضو m_Sprite را مقداردهی اولیه می کند . Animator::CreateAnimation() از پارامترهای داده شده ، یک انیمیشن تولید می کند ، آن را به لیست اضافه کرده و یک ارجاع به آن بازگشت می دهد . به زودی خواهیم دید این دقیقا چطور کار می کند .

Animator::Update() تمام منطق پشت انتخاب فریم درست را مدیریت می کند .

Animator::SwitchAnimation(string) سعی می کند انیمیشن جاری را با انیمیشنی که اسمش تعیین شده است ، عوض کند .

وقت آن است که به پیاده سازی این تابع ها و شیوه استفاده از آنها نگاه کنیم . منطقی است که از سازنده شروع کنیم :

```
Animator::Animator(sf::Sprite& sprite)
    :m_Sprite(sprite), m_CurrentTime(), m_CurrentAnimation(nullptr)
{
}
```

ما فقط داده هایمان را مقداردهی اولیه کرده ایم . توجه کنید که ارجاع Sprite باید در لیست مقداردهی اولیه نصب شود در غیر این صورت کد کامپایل نمی شود . شیوه کار کردن ارجاع ها در C++ این طور است که باید به محض اینکه امکانش فراهم شد ، مقداردهی اولیه شوند . چیز دیگری که باید توجه کرد مقداردهی اولیه انیمیشن جاری با nullptr است . این همیشه یک عادت خوب است که اشاره گرها را با nullptr مقداردهی اولیه کرد .

Animator::CreateAnimation() کمی جالب تر است . نگاه کنید :

```
Animator::Animation& Animator::CreateAnimation(std::string const& name,
    std::string const& textureName, sf::Time const& duration, bool loop)
{
    m_Animations.push_back(
        Animator::Animation(name, textureName, duration, loop));

    //If we don't have any other animations, use that as current animation
    if (m_CurrentAnimation == nullptr)
        SwitchAnimation(&m_Animations.back());

    return m_Animations.back();
}
```

Animator::CreateAnimation() با استفاده از پارامترها ، یک انیمیشن تولید می کند . همانطور که قبلا گفتیم ، هر انیمیشن به یک اسم احتیاج دارد تا بتوانیم از خارج از کلاس به آن ارجاع داشته باشیم . همچنین هر انیمیشن یک بافت و یک مدت زمان پخش می خواهد . پارامتر loop تعیین می کند که آیا انیمیشن باید تکرار شود یا اینکه فقط یک بار پخش شود . با این پارامترها ، ما یک نمونه جدید از انیمیشن را مقداردهی اولیه کرده و آن را در m_Animations قرار می دهیم . اگر این اولین انیمیشن ما باشد ، آن را انیمیشن جاری قرار می دهیم . این باعث می شود هنگامی که متد Animator::Update() فراخوانده می شود ، چیزی برای پخش داشته باشیم .

Animator::SwitchAnimation(Animation*) دقیقا همین کار را انجام می دهد : یک انیمیشن می گیرد و آن را انیمیشن جاری قرار می دهد . چند لحظه دیگر می بینیم که این متد چطور کار می کند . آخرین خط متد یک ارجاع به انیمیشن تولید شده بازگشت می دهد . این فراخواننده را قادر می کند انیمیشن را با اضافه کردن فریم ها یا تغییر بعضی از مقادیر ابتدائی ، دستکاری کند .

همانطور که وعده داده بودیم ، یکی از متدهای overload شده `Animator::SwitchAnimation()` به صورت زیر است :

```
void Animator::SwitchAnimation(Animator::Animation* animation)
{
    //Change the sprite texture
    if (animation != nullptr)
    {
        m_Sprite.setTexture(AssetManager::GetTexture(animation->m_TextureName));
    }

    m_CurrentAnimation = animation;
    m_CurrentTime = sf::Time::Zero; //Reset the time
}
```

از آنجائیکه بالقوه به یک انیمیشن جدید سوئیچ می کنیم ، باید بافت sprite را با بافت انیمیشن جدید تغییر دهیم . اما چون هر دو `animation` و `m_CurrentAnimation` می توانند `nullptr` باشند ، باید آن را ابتدا چک کنیم . اگر `animation` برابر با `nullptr` نباشد ، می توانیم با اطمینان از اسم بافت برای فراخواندن `AssetManager` و درخواست بافت استفاده کنیم . در این مورد ، اگر بافت انیمیشن قبلی ، همین یکی باشد ، هزینه این تغییر کم خواهد بود چون با ارجاع ها سروکار داریم .
سپس برای استفاده های بعدی ، اشاره گر `m_CurrentAnimation` را با `animation` نصب می کنیم و زمان را راه اندازی مجدد می کنیم .
یک ورژن overload شده از `Animator::SwitchAnimation(Animation*)` هم وجود دارد که به جای `Animation*` یک `string` می گیرد . این تابع عمومی است و برای تغییر انیمیشن بر اساس اسمش ، استفاده می شود . پیاده سازی این تابع را ببینید :

```
bool Animator::SwitchAnimation(std::string const& name)
{
    auto animation = FindAnimation(name);
    if (animation != nullptr)
    {
        SwitchAnimation(animation);
        return true;
    }

    return false;
}
```

این متد کاملا سرراست است و سعی می کند با اسم داده شده ، انیمیشن را پیدا کند . اگر شکست بخورد ، `false` بازگشت می دهد . اگر موفق شود ؛ از این اشاره گر animation برای سوئیچ به آن انیمیشن استفاده می کند . این کار توسط فراخوان تابع overload شده که قبلا بحث شد ، انجام می شود . در نهایت یک `true` بازگشت می دهد تا به فراخواننده نشان دهد که این انیمیشن وجود دارد و تعویض با موفقیت بوده است .
`Animator::FindAnimation()` خیلی قابل توجه نیست اما به آن نگاهی می اندازیم :

```
Animator::Animation* Animator::FindAnimation(std::string const& name)
{
    for (auto it = m_Animations.begin(); it != m_Animations.end(); ++it)
    {
        if (it->m_Name == name)
            return &*it;
    }

    return nullptr;
}
```

تنها چیزی که باید اینجا توجه کنید این است که این متد اولین انیمیشن با اسم یکسان را بازگشت می دهد ، یعنی اینکه هنگامی که انیمیشن ها را تولید می کنیم باید خیلی مراقب باشیم . در این پیاده سازی ، از انیمیشن های هم نام باید دوری کرد ، چون دسترسی به آنها به صورت جداگانه غیر ممکن می شود . همچنین اگر انیمیشن هائی با اسم یکسان پیدا کردیم باید یک استثناء اعلام کنیم تا اشکال زدائی آسان تر شود .

گاهی اوقات لازم است که از خارج کلاس بررسی کنیم که کدام انیمیشن در حال حاضر در حال پخش است . برای این کار ما یک متد ساده پیاده سازی می کنیم که اسم انیمیشنی که در حال حاضر در حال پخش است را بازگشت می دهد :

```
std::string Animator::GetCurrentAnimationName() const
{
    if (m_CurrentAnimation != nullptr)
        return m_CurrentAnimation->m_Name;

    //If no animation is playing, return empty string
    return "";
}
```

حالا که اغلب متدهای کلاس را بررسی کردیم ، اجازه دهید مهمترین آنها را ببینیم . متد `Animator::Update()` :

```
void Animator::Update(sf::Time const& dt)
{
    //If we don't have any animations yet return
    if (m_CurrentAnimation == nullptr)
        return;

    m_CurrentTime += dt;

    //Get the current animation frame
    float scaledTime = (m_CurrentTime.asSeconds() / m_CurrentAnimation->m_Duration.asSeconds());
    int numFrames = m_CurrentAnimation->m_Frames.size();
    int currentFrame = static_cast<int>(scaledTime * numFrames);

    //If the animation is looping, calculate the correct frame
    if (m_CurrentAnimation->m_Looping)
        currentFrame %= numFrames;
    else if (currentFrame >= numFrames)//if the current frame is greater than the number of frames
        currentFrame = numFrames - 1; //Show last frame

    //Set the texture rectangle, depending on the frame
    m_Sprite.setTextureRect(m_CurrentAnimation->m_Frames[currentFrame]);
}
```

این کدها کاملا شبیه آن چیزی است که در بخش قبلی برای اجرای انیمیشن انجام دادیم . آن با بررسی `m_CurrentAnimation` شروع می شود . چون این مقدار می تواند `nullptr` باشد ، در این مورد لازم نیست کاری انجام دهیم . سپس زمان سپری شده را به زمان محلی اضافه می کنیم . ما این زمان را به مدت زمان انیمیشن تقسیم می کنیم تا `scaledTime` را به دست آوریم که برای تعیین

اینکه کدام فریم باید نمایش داده شود ، مورد استفاده قرار می گیرد . همانطور که در بخش قبلی هم انجام دادیم ، ما فریم جاری را با ضرب این زمان در مجموع تعداد فریم های انیمیشن و سپس گرد کردن این عدد به دست می آوریم .
بخش بعدی جدید است . اگر بخواهیم انیمیشن تکرار شود ، از اپراتور `%` استفاده می کنیم در غیر این صورت ، فقط یکبار انیمیشن را پخش می کنیم و از آخرین فریم استفاده می کنیم تا اینکه انیمیشن عوض شود یا مجددا پخش شود .
حالا که فریم جاری را داریم ، فقط باید مستطیل بافت را برای sprite نصب کنیم . چون ما همه فریم ها را در داخل انیمیشن نگهداری می کنیم ، بازیابی آنها آسان است : برای این کار از `Animation::m_Frames` به عنوان آرایه استفاده می کنیم و اندیس آن را با `currentFrame` فرامی خوانیم .
خوب ، یک انیمیت کننده sprite قدرتمند ایجاد کردیم . سپس به چند مثال نگاه می کنیم که چگونگی استفاده از این انیمیت کننده را در عمل به ما نشان می دهند .

استفاده از انیمیت کننده

اجازه دهید با مثال کریستال چرخان شروع کنیم . بخش مقداردهی اولیه را ببینید :

```
sf::Vector2i spriteSize(32, 32);
sf::Sprite sprite;
Animator animator(sprite);
//Create an animation and get the reference to it
auto& idleAnimation = animator.CreateAnimation("Idle", "spriteSheet.png", sf::seconds(1), true);
//Add frames to the animation
idleAnimation.AddFrames(sf::Vector2i(0, 0), spriteSize, 8);
```

برای تولید یک انیمیت کننده باید توجه کنید که یک sprite باید از قبل موجود باشد چون انیمیت کننده از یک ارجاع به sprite در سازنده اش استفاده می کند . هنگامی که یک انیمیت کننده برای sprite ایجاد کردیم ، یک انیمیشن را اضافه می کنیم . چون داریم مثال قبلی را دوباره تکرار می کنیم ، فقط یک انیمیشن وجود دارد . اسم انیمیشن `Idle` خواهد بود و از بافت `spriteSheet.png` استفاده می کند . این انیمیشن یک ثانیه زمان برای تکمیل شدن نیاز دارد و باید تکرار شود .
وقتی که ارجاع به انیمیشن را داشتیم ، چند فریم را از بافت اضافه می کنیم . ما می دانیم که sprite چقدر بزرگ است ، برای گرفتن همه فریم ها از بافت ، از (0, 0) شروع کرده و پی در پی به صورت افقی به اندازه ۸ بار حرکت می کنیم ، هر بار به اندازه 32 پیکسل به جلو حرکت خواهیم کرد . همه اینها در داخل متد `Animation::AddFrames()` انجام می شود . تنها کاری که باقی مانده فراخواندن متد update انیمیت کننده `Animator` در بخش به روز کردن فریم حلقه بازی است :

```
sf::Clock clock;
while (window.isOpen())
{
    //Returns the elapsed time and restarts the clock
    sf::Time deltaTime = clock.restart();

    animator.Update(deltaTime);

    window.clear(sf::Color::Black);

    window.draw(sprite);

    window.display();
}
```

چندین انیمیشن

ایجاد یک انیمیشن کاملا سرراست است . این طور نیست ؟ با چندین انیمیشن ، چیزها خیلی پیچیده نخواهد بود . فرض کنید دو بافت `spriteSheet.png` و `myTexture.png` را دایم ، و می خواهیم ۴ انیمیشن داشته باشیم که هریک از آنها از یکی از این بافت ها استفاده کنند . راه اندازی اولیه را ببینید :

```
Animator animator(sprite);
//Idle animation with 8 frames @ 1 sec looping
auto& idleAnimation = animator.CreateAnimation("Idle", "spriteSheet.png", sf::seconds(1), true);
idleAnimation.AddFrames(sf::Vector2i(0, 0), spriteSize, 8);

//IdleShort animation with 8 frames @ 0.5 sec looping
auto& idleAnimationShort = animator.CreateAnimation("IdleShort", "spriteSheet.png", sf::seconds(0.5f), true);
idleAnimationShort.AddFrames(sf::Vector2i(0, 0), spriteSize, 8);

//IdleSmall animation with 5 frames @ 1.5 sec looping
auto& idleAnimationSmall = animator.CreateAnimation("IdleSmall", "myTexture.png", sf::seconds(1.5f), true);
//Adding frames multiple times from different locations
idleAnimationSmall.AddFrames(sf::Vector2i(64, 0), spriteSize, 3);
idleAnimationSmall.AddFrames(sf::Vector2i(64, 32), spriteSize, 2);

//IdleOnce animation with 8 frames @ 0.5 sec not looping
auto& idleAnimationOnce = animator.CreateAnimation("IdleOnce", "myTexture.png", sf::seconds(0.5f), false);
idleAnimationOnce.AddFrames(sf::Vector2i(0, 0), spriteSize, 8);
```

به تمام شیوه های متفاوتی که با آن می توانیم انیمیشن را ایجاد کنیم ، توجه کنید . ما به زمان ، بافت ، اسم ، تعداد فریم ها یا حتی موقعیت های فریم محدود نیستیم . حتی می توانیم توسط فراخوان `Animation::AddFrames()` و پاس دادن 1 به عنوان آخرین پارامتر ، فریم ها را یکی یکی از بافت ، اضافه کنیم . داشتن تعداد زیادی از انیمیشن ها چیز جالبی است اما راهی برای عوض کردن انیمیشن ها نیاز است . کد زیر نشان می دهد که چطور در رویداد فشرده شدن کلید ، انیمیشن را عوض کنیم :

```
sf::Event ev;
while (window.pollEvent(ev))
{
    if (ev.type == sf::Event::KeyPressed)
    {
        if (ev.key.code == sf::Keyboard::Key::Num1)
            animator.SwitchAnimation("Idle");
        else if (ev.key.code == sf::Keyboard::Key::Num2)
            animator.SwitchAnimation("IdleShort");
        else if (ev.key.code == sf::Keyboard::Key::Num3)
            animator.SwitchAnimation("IdleSmall");
        else if (ev.key.code == sf::Keyboard::Key::Num4)
            animator.SwitchAnimation("IdleOnce");
    }
}
```

این کار به سادگی فراخواندن متد `Animator::SwitchAnimation()` و پاس دادن اسم انیمیشن جدید است . یادتان باشد پاس دادن اسم انیمیشنی که در حال پخش است ، باعث راه اندازی مجدد زمان شده ، و انیمیشن از ابتدا شروع می شود . این یعنی فراخواندن این متد در طول هر فریم ایده خوبی نیست . اگر بخواهیم اسم انیمیشنی که در حال پخش است را چک کنیم ، می توانیم `Animator::GetCurrentAnimationName()` را فراخوانده و فقط اگر انیمیشن جدید با این متفاوت باشد ، انیمیشن را عوض کنیم .

همانطور که دیدید ، کلاس `Animator` مدیریت انیمیشن را آسان تر می کند . دیگر تولید چندین sprite مشکلی نیست چون می توانیم به هر تعدادی که بخواهیم ، انیمیت کننده ایجاد کنیم .

خلاصه

انیمیشن های Sprite حالا کاملا ساده به نظر می رسند . این طور نیست ؟ فقط به یاد داشته باشید در خصوص انیمیشن ها ، چیزهای بیشتری برای اکتشاف وجود دارد . نه تنها تکنیک های متفاوتی برای انجام آنها وجود دارد ، تکمیل آنچه که تاکنون انجام داده ایم ، ممکن است زمان بر باشد . خوشبختانه ، آنچه که تاکنون انجام دادیم ، در بیشتر نمونه ها کار خواهد کرد ، بنابرین بهتر است با اشتیاق شروع کنید .

در آینده به بررسی دوربین ها و رندرینگ OpenGL می پردازیم .

فصل ٤ : مدیریت یک دوربین دوبعدی

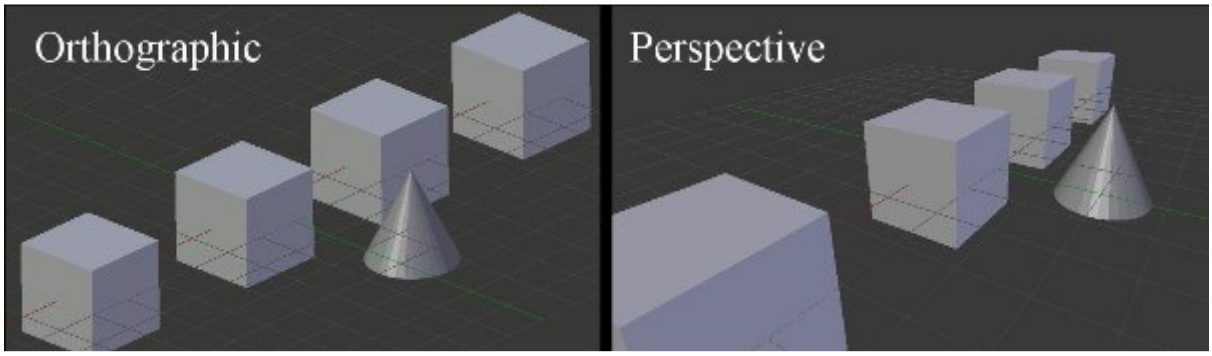
در این فصل ما در خصوص دوربین ها و **OpenGL** صحبت می کنیم و اینکه چطور می توانیم از اینها به نفع خودمان استفاده کنیم . ما مباحث مربوط به دوربین ها را مفصل مطرح می کنیم اما در خصوص OpenGL ، تنها به صورت مختصر درباره ادغام آن در **SFML** بحث خواهیم کرد . **OpenGL API** خیلی بزرگ است و نمی توان آن را در یک کتاب پوشش داد چه برسد به یک فصل . اگر نمی دانید چطور کدهای OpenGL بنویسید یا نمی خواهید از OpenGL استفاده کنید ، می توانید آخرین بخش این فصل را نادیده بگیرید . اما اگر می خواهید بدانید که OpenGL چطور می تواند به شما کمک کند ، ببینید دومین بخش این فصل چه چیزهائی را توصیه می کند .

در این فصل مطالب زیر توضیح داده می شود :

دوربین چیست ؟
مدیریت دوربین ها با `sf::View`
OpenGL چیست ؟
استفاده از OpenGL در داخل SFML

دوربین چیست ؟

شانس اینکه با دوربین ها در ساختن بازی روبه رو نشوید خیلی کم است . آنها یکی از ضروریات هر بازی هستند . در اصل یک دوربین ، نقطه ای در فضاست که از طریق آن شما می توانید به دنیای بازی نگاه کنید . پارامترهای زیادی در رابطه با دوربین ها وجود دارد ، هم در فضای دوبعدی هم در فضای سه بعدی ، اما در این فصل ، بر روی آنچه که SFML ارائه می دهد ، متمرکز می شویم .
قبل از رفتن به سراغ کدها ، اجازه دهید چند مطلب را بیان کنیم . چون SFML اغلب برای بازی های دوبعدی استفاده می شود ، کلاس `camera` تنها از یک تصویر آرتوگرافیک (**orthographic**) استفاده می کند . در این تصویر ، هر شئ به شکلی ظاهر می شود که انگار هیچ دورنمای سه بعدی ندارد . در مقابل تصویر پرسپکتیو (perspective – دارای دورنمای سه بعدی) ، چگونگی ظاهر شدن اشیاء روی صفحه را بر اساس فیزیک چشم انسان تغییر می دهد (اشیاء در فاصله دورتر ، کوچکتر ظاهر می شوند) . این نوع تصویر در بازی های سه بعدی استفاده می شود و به دنیای دوبعدی تعلق ندارد . مقایسه ای کوچک بین این دو نوع دید را ببینید :



استفاده از تصویر پرسپکتیو برای بازی های دوبعدی منطقی نیست چون تصویر sprite ها ناقص می شود . اما می توانیم با استفاده از OpenGL یک دوربین سفارشی تولید کنیم ، این موضوعی است که بعدا در این فصل بحث خواهد شد .

کی باید از دوربین استفاده کنیم ؟

لازم نیست که همیشه دوربین را دستکاری کنیم . اگر بازی فقط یک صفحه دارد ، دلیلی برای تغییر دوربین نیست چون همیشه در مرکز صفحه ساکن خواهد بود . مثال دیگر رفت و آمد در طول منو اصلی بازی است که باز هم موقعیت دوربین ساکن است و نیاز به انجام کاری نیست .

اما فرض کنید یک بازی نقش آفرینی (**RPG**) می سازیم ، که در آن یک دنیای بزرگ برای اکتشاف وجود دارد . در این مورد قطعا باید یک دوربین پیاده سازی کنیم چه برای حرکت همراه با کاراکتر ما یا نسبت به آن . در اصل هنگامی که یک دنیا (یا مرحله) برای اکتشاف داریم ، باید استفاده از دوربین را بررسی کنیم .

حالا که مفهوم اولیه یک دوربین را می دانیم ، اجازه دهید تا ببینیم که چطور باید یکی از آنها را پیاده سازی کنیم .

SFML چطور یک دوربین را پیاده سازی می کند ؟

اگر بخواهیم دوربین پیش فرضی را که همراه با هر نمونه `sf::Window` می آید ، تغییر دهیم ، باید با کلاس `sf::View` سروکار داشته باشیم . کلاس `View` دقیقا شبیه به یک نوع دوربین رفتار می کند : این کلاس آنچه را که بازیکن می تواند در دنیا ببیند را توسط تنظیم پارامترهائی ، محدود می کند . چگونگی تولید و استفاده از `View` را ببینید :

```
auto wSize = window.getSize();
sf::View view(sf::FloatRect(0, 0, wSize.x, wSize.y));

//Initialize view

window.setView(view);
```

سازنده کلاس `View` یک پارامتر `FloatRect` می گیرد ، که ناحیه مطلوب از چشم انداز دنیا را تنظیم می کند . اگر ناحیه دید بزرگتری داشته باشیم ، محتویات آن کوچک می شوند تا درون پنجره جا شوند و برعکس . در این مثال ناحیه مطابق با اندازه پنجره است بنابراین تغییری در چگونگی رندر اشیاء نمی دهد .

در نهایت وقتی که همه چیز را در `View` تنظیم کردیم ، لازم است با فراخوان `RenderWindow::setView()` ، به پنجره بگوئیم که از آن استفاده کند .

حالا اجازه دهید تا ببینیم که کلاس `View` چه کارهائی می تواند انجام دهد .

مدیریت دوربین ها با استفاده از sf::View

مهمترین خصوصیت کلاس `View` توانائی آن برای تغیر مرکز ناحیه دید است . به صورت پیش فرض ، مرکز دید ، مرکز ناحیه دید است یعنی اگر ناحیه دید ما اندازه (640, 480) داشته باشد ، مرکز دید (320, 240) خواهد بود . این باعث می شود اشیاء رندر شده با موقعیت (0; 0) در گوشه بالائی سمت چپ ظاهر شوند . این همان رفتاری است که هنگام استفاده از `View` پیش فرض یک پنجره SFML به دست می آوردیم . برای تغییر مرکز دید می توانیم `View::setCenter()` یا `View::move()` را فرابخوانیم .

مثال را ببینید :

```
auto wSize = window.getSize();
sf::View view(sf::FloatRect(0, 0, wSize.x, wSize.y));

//The view is centered around the world point (0; 0)
view.setCenter(sf::Vector2f(0, 0));

window.setView(view);

sf::Vector2f spriteSize = sf::Vector2f(32, 32);
sf::Sprite sprite(AssetManager::GetTexture("myTexture.png"));
sprite.setOrigin(spriteSize * 0.5f); // Sprite origin at it's center
```

اگر ما مرکز دید را در موقعیت (0; 0) قرار دهیم ، این موقعیت در دنیا در مرکز صفحه ظاهر می شود . sprite در کدنویسی به صورت پیش فرض در موقعیت (0; 0) است ، بنابراین در مرکز صفحه ظاهر می شود . نتیجه را ببینید :

معمولا ، قرار دادن مرکز دید روی کاراکتر اصلی ، تکنیکی ساده و موثر برای جابجائی موقعیت دوربین است . این کار فقط به دو خط کد در بخش به روزرسانی فریم حلقه بازی نیاز دارد :

```
view.setCenter(sprite.getPosition());
window.setView(view);
```

توجه کنید که ما باید `RenderWindow::setView()` را فراخوانده و `view` را دوباره پاس دهیم چون `RenderWindow` فقط یک کپی از `view` را نگه می دارد . فقط تغییر دادن `view` قدیمی بر روی `view` ذخیره شده در `RenderWindow` تاثیری نخواهد گذاشت .

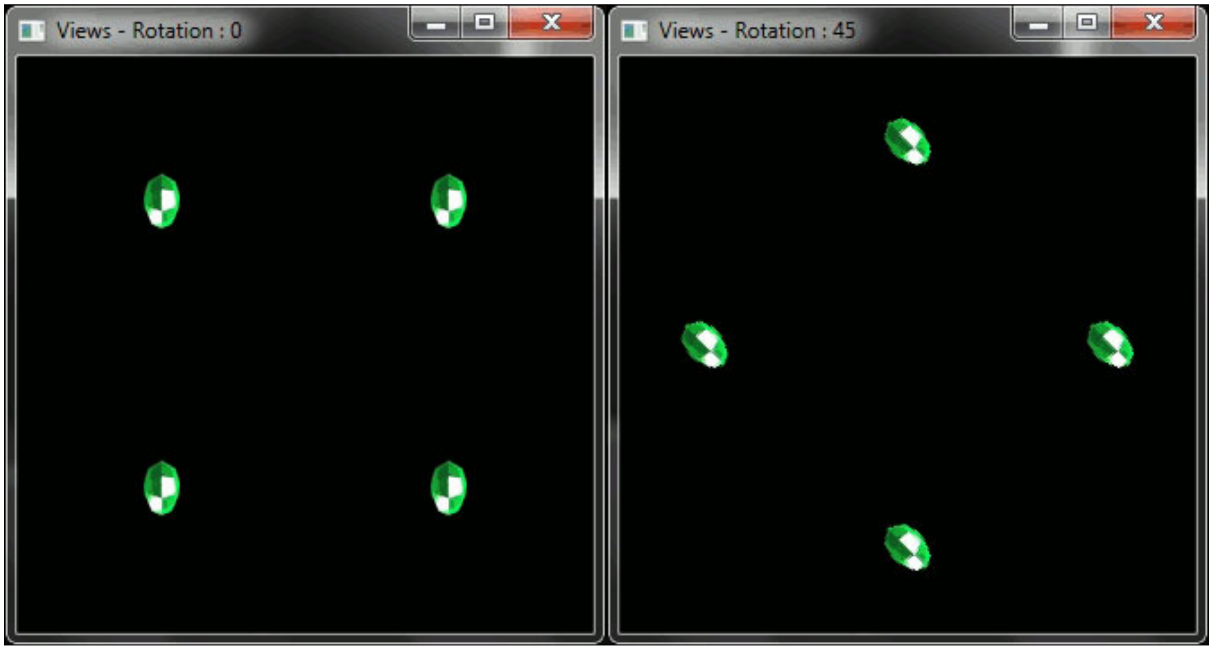
چرخاندن و کوچک و بزرگ کردن یک دوربین

دو تبدیل دیگر وجود دارد که می توان بر روی هر دوربین انجام داد : چرخش و کوچک و بزرگ کردن . هر دوی اینها استفاده های محدودی دارند ، اما هنگامی که خصوصیات ویژه ای را از دوربینمان انتظار داریم ، به کار می آیند . برای چرخاندن یک دوربین بسته به نوع چرخشی که می خواهیم انجام شود ، `View::setRotation()` یا `View::rotate()` را فرامی خوانیم . متد `View::setRotation()` یک مقدار مطلق را برای چرخاندن `View` واگذار می کند ، در حالیکه ، `View::rotate()` معمولا هنگامی استفاده می شود که می خواهیم به تدریج و در طول یک دوره زمانی ، چرخش را افزایش دهیم .

چرخش همانطوری که انتظار داریم ، انجام می شود : آن صحنه را (همه اشیاء را) حول مرکز دید می چرخاند . در زیر آماده سازی تستی را که قصد داریم انجام دهیم ، ببینید :

```
auto wSize = window.getSize();
//The view is centered around the world point (0; 0)
//The view has the size of the window
sf::View view(sf::Vector2f(0, 0), sf::Vector2f(wSize.x, wSize.y));

//Set rotation view.setRotation(...);

window.setView(view);

sf::Vector2f spriteSize = sf::Vector2f(32, 32);
auto& texture = AssetManager::GetTexture("myTexture.png");

//Top left
sf::Sprite sprite1(texture);
sprite1.setOrigin(spriteSize * 0.5f);
sprite1.setPosition(sf::Vector2f(-80, -80));

//Top right
sf::Sprite sprite2(texture);
sprite2.setOrigin(spriteSize * 0.5f);
sprite2.setPosition(sf::Vector2f(80, -80));

//Bottom right
sf::Sprite sprite3(texture);
sprite3.setOrigin(spriteSize * 0.5f);
sprite3.setPosition(sf::Vector2f(80, 80));

//Bottom left
sf::Sprite sprite4(texture);
sprite4.setOrigin(spriteSize * 0.5f);
sprite4.setPosition(sf::Vector2f(-80, 80));
```

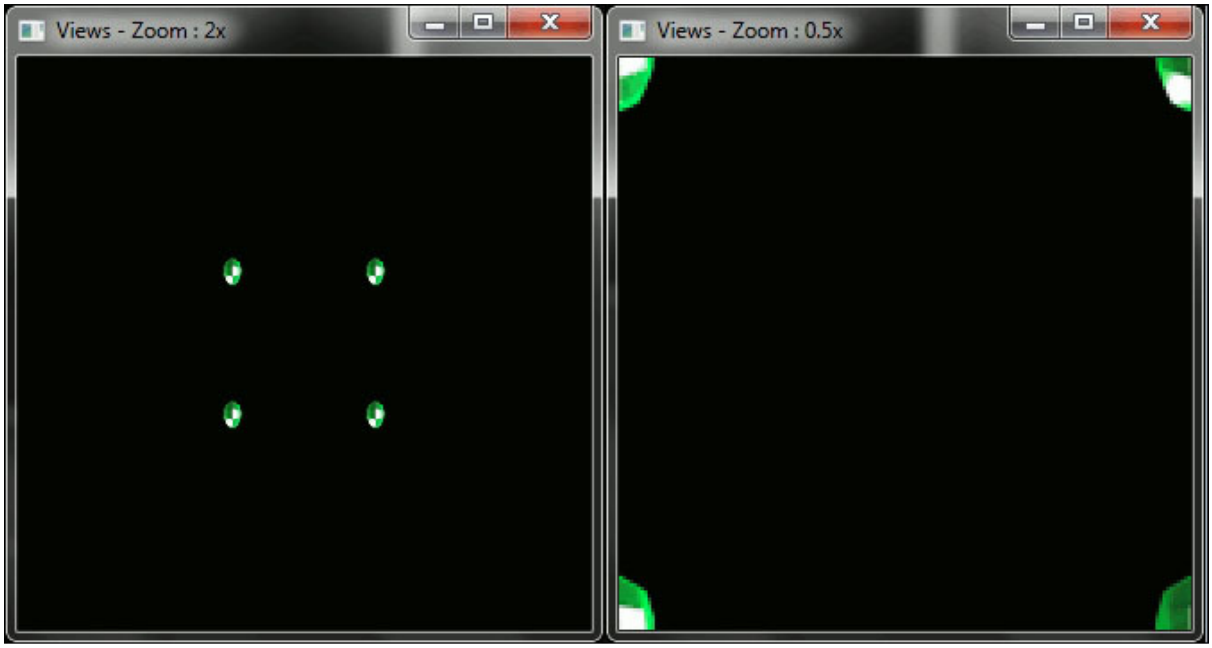
این بار ، دوربین با یک سازنده متفاوت فراخوانده شده است . به جای پاس دادن یک مستطیل ، ما یک موقعیت مرکز و یک اندازه را پاس می دهیم . این همان کار را انجام می دهد ، اما به ما اجازه می دهد ، نقطه مرکز را آسانتر تعیین کنیم . بعد از مقداردهی ابتدائی دوربین ، چهار sprite در چهار گوشه صفحه حول نقطه (0;0) ایجاد می کنیم . نتیجه زیر تست هائی است که با چرخش صفر و ۴۵ درجه بر روی صحنه انجام شده است :



چرخاندن دوربین استفاده محدودی در ساختن بازی دارد . آن می تواند برای انیمیت کردن یک رویداد خاص مثلا مردن کاراکتر اصلی (یک زوم آرام همراه با چرخش) یا آسیب رسیدن (لرزش خفیف دوربین) استفاده شود . ما همچنین می توانیم در یک بازی از بالا به پائین ، از آن برای چرخاندن دنیا حول یک کاراکتری که در مرکز قرار گرفته است ، استفاده کنیم . به طور کلی ، دوران مفید است اما موارد استفاده کمی دارد .

دوربین ها می توانند کوچک و بزرگ هم شوند . با کوچک و بزرگ کردن دوربین ، می توانیم چیزهای بیشتر یا کمتری از دنیای بازی را نمایش دهیم ، البته بسته به جهت کوچک یا بزرگ کردن . در خصوص بازی ها ، این خصوصیت را معمولا زوم – بزرگنمائی (**zoom**) می گویند . زوم مستقیما مرتبط با اندازه دوربین است . هنگامی که به دوربین می گوئیم که یک ناحیه را دوبار بزرگتر از پنجره نشان بده ، با فاکتور ۲ زوم را انجام داده ایم . از طرف دیگر ، اگر بخواهیم اشیاء را نزدیکتر نشان دهیم ، باید زوم را با فاکتور `1/x` انجام دهیم که `x` فاکتور زوم است . برای کوچک کردن دوربین به اندازه دوبرابر به فاکتور 1 / 2 = 0.5 نیاز داریم .

اجازه دهید کوچک و بزرگ کردن را با استفاده از دو مثال همراه با sprite نشان دهیم که برای چرخش استفاده شدند :



تنظیم زوم با فراخوان `View::zoom()` همراه با یک فاکتور زوم انجام می شود یا اینکه توسط تغییر اندازه دوربین با `View::setSize()` .

هنگامی که می خواهیم در طول تعدادی از فریم ها ، حرکتی مداوم داشته باشیم ، اغلب از `View::zoom()` , `View::move()` استفاده می شود .

مهم است که توجه کنید که `View` فاکتور زوم را ذخیره نمی کند ، بلکه فقط اندازه دوربین را ذخیره می کند . این یعنی فراخوان `View::zoom()` با فاکتوری متفاوت با `1` ، اندازه دوربین را هر بار تغییر می دهد ، حتی اگر همان فاکتور را پاس دهیم . متد زوم را تنظیم نمی کند ، بلکه فقط اندازه دوربین را با آن فاکتور تغییر می دهد . برای مثال اگر `View::zoom()` را دو بار با فاکتور `1/2` فرابخوانیم ، نتیجه زیر را خواهیم داشت :

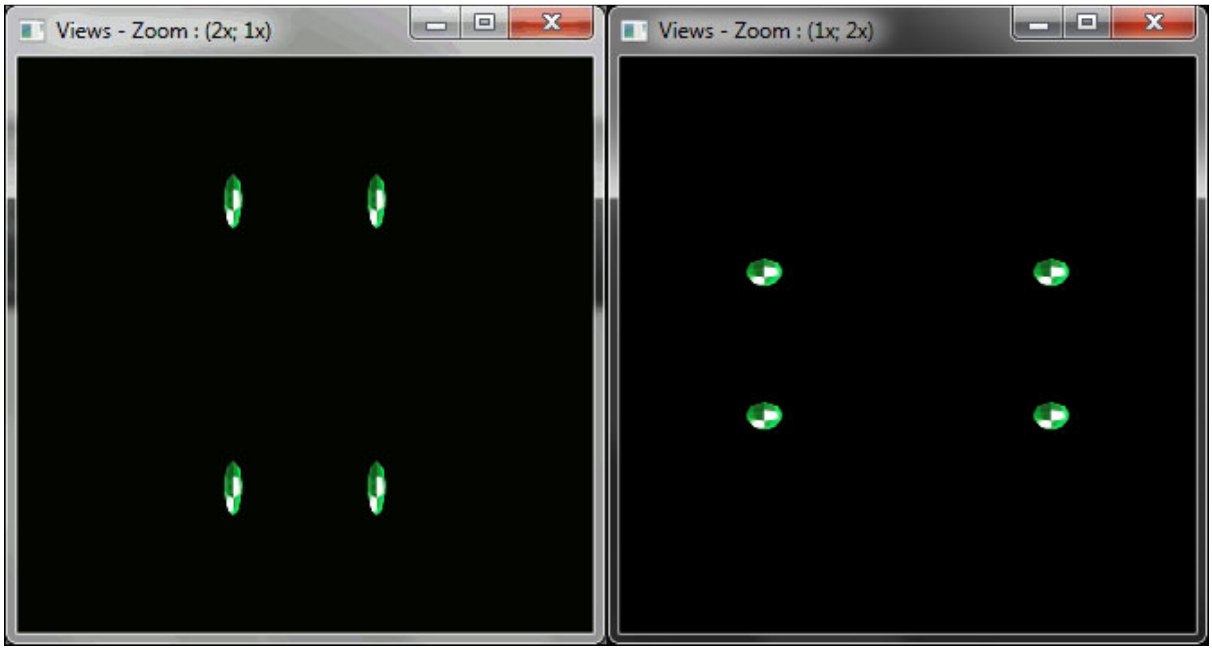
(1/2) * (1/2) * = (1/4) * size

همین کار با فراخوان `View::zoom()` و فاکتور `1/4` انجام می شود .

چون `View::zoom()` فقط یک فاکتور می گیرد ، از اندازه حال حاضر دوربین استفاده می کند تا تخمین بزند که چقدر باید عرض و ارتفاع دوربین را تغییر دهد . اگر بخواهیم عرض و ارتفاع را با دو فاکتور مختلف تغییر دهیم ، باید از `View::setSize()` همراه با مقادیر عرض و ارتفاعی که می خواهیم ، استفاده کنیم . مثال را ببینید :

```
auto wSize = window.getSize();
sf::View view(sf::Vector2f(0, 0), sf::Vector2f(wSize.x, wSize.y));

//First example
view.setSize(wSize.x * 2, wSize.y);
//Second example
view.setSize(wSize.x, wSize.y * 2);

window.setView(view);
```

این کد نتیجه زیر را دارد :



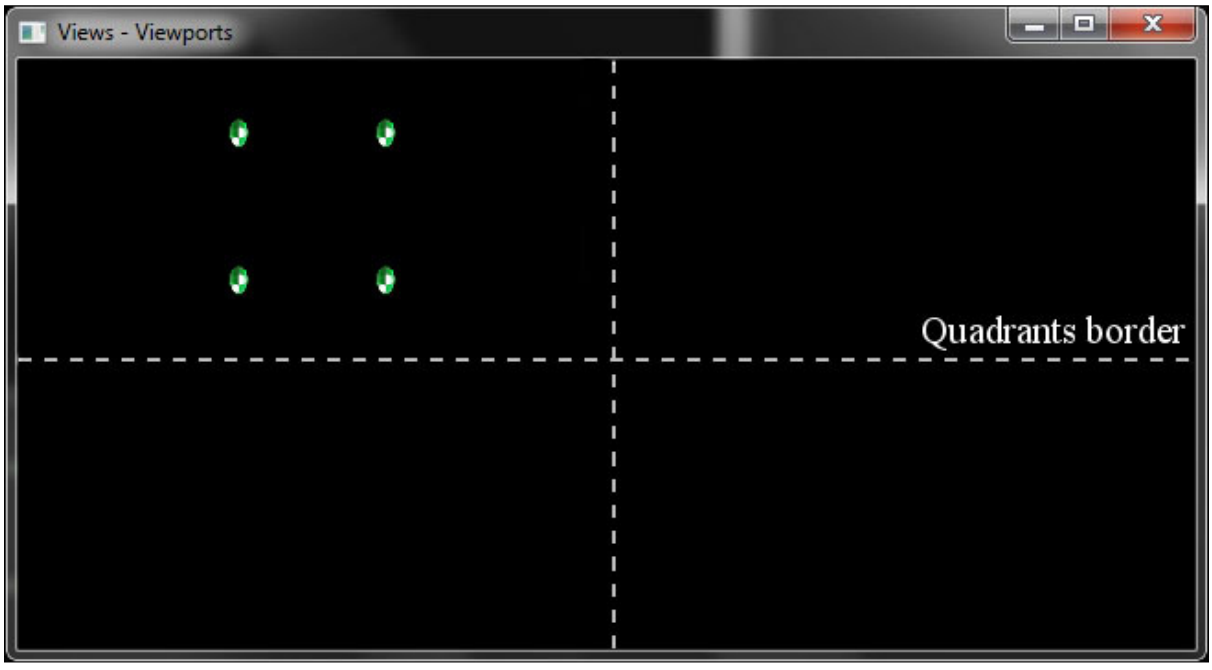
همانطور که می بینید ، هر دو تصویر روی محورهای *x* و *y* فشرده تر به نظر می رسند . علت این است که ما سعی کرده ایم ، دوبرابر پیکسل ها را روی محوری که صفحه به ما اجازه داده است ، جا دهیم . احتمالا این نتیجه برای شما آشناست چون شبیه به قرار دادن بافت بر روی سطحی است که ابعاد متفاوتی با بافت دارد .
چند چیز دیگر وجود دارد که می توان با `View` انجام داد . اینها را در بخش بعدی بررسی می کنیم .

Viewport

هر دوربین یک viewport وابسته به خود دارد . viewport ناحیه ای از پنجره است که در آن منظره نمایش داده می شود . این ناحیه با یک مستطیل نمایش داده می شود که از مختصات های نرمال سازی شده [0...1] استفاده می کند . به صورت پیش فرض ، viewport برابر با اندازه دوربین (view) است (0, 0, 1, 1)
ما می توانیم این را با فراخوان `View::setViewport()` تغییر دهیم . فرض کنیم می خواهیم صحنه را فقط در یک چهارم سمت چپ و بالای صفحه ، رندر کنیم . کد زیر این کار را انجام می دهد :

```
auto wSize = window.getSize();
sf::View view(sf::Vector2f(0, 0), sf::Vector2f(wSize.x, wSize.y));

view.setViewport(sf::FloatRect(0, 0, 0.5f, 0.5f));

window.setView(view);
```

نتیجه این کد را ببینید :



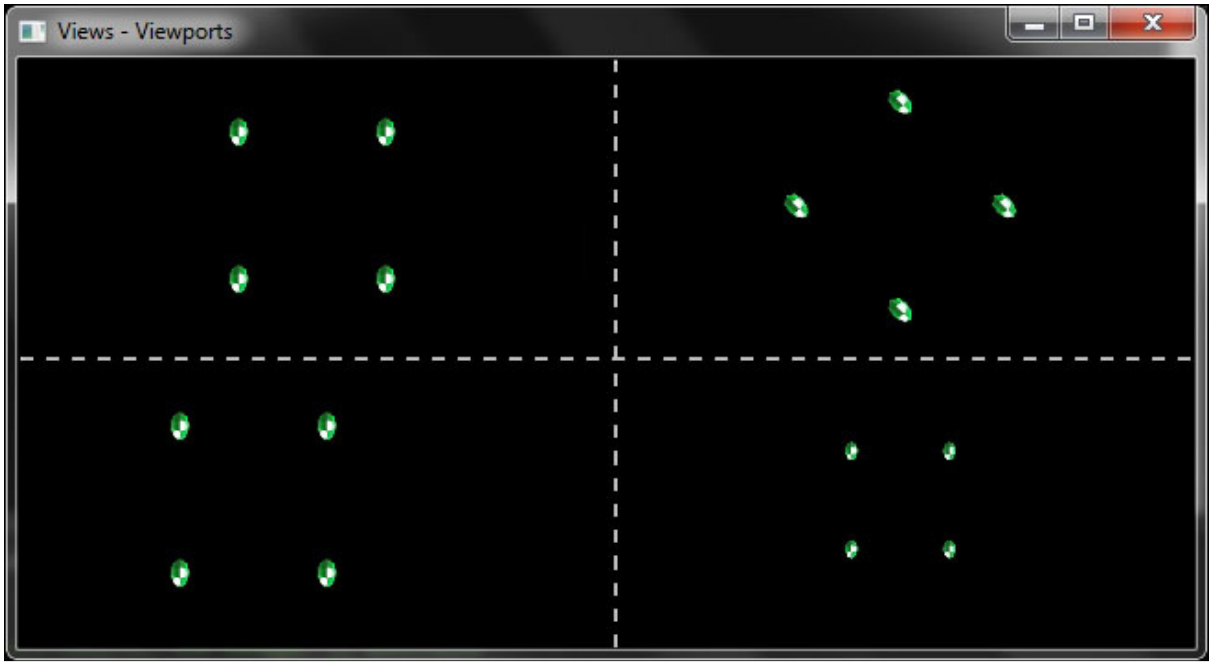
چون می توانیم چندین دوربین ایجاد کنیم ، می توانیم در بخش رندر فریم ، بین آنها سوئیچ کنیم و مکررا صحنه را رندر کنیم . در این شیوه ، صحنه طوری نمایش داده می شود که انگار در دوربین های متفاوتی ظاهر شده اند . در مثال قبلی ، می توانیم همین صحنه را در همه چهار ربع صفحه رندر کنیم اما با تغییر شکل های متفاوت . برای انجام این کار ، چهار دوربین را در چهار گوشه صفحه با viewport های زیر نصب می کنیم :

گوشه بالائی سمت چپ : (0, 0, 0.5, 0.5)
گوشه بالائی سمت راست : (0.5, 0, 0.5, 0.5)
گوشه پائینی سمت چپ : (0, 0.5, 0.5, 0.5)
گوشه پائینی سمت راست : (0.5, 0.5, 0.5, 0.5)

وقتی که اینها را داشتیم ، تغییر شکل آنها را دستکاری کرده و در یک vector با نام `viewList` قرار می دهیم . سپس بخش رندر فریم به شکل زیر خواهد بود :

```
window.clear(sf::Color::Black);

for (auto it = viewList.begin(); it != viewList.end(); ++it)
{
    //Set the view
    window.setView(*it);

    //Render sprites
}

window.display();
```

نتیجه بستگی به این دارد که چطور دوربین ها را دستکاری کرده باشیم . مثال زیر را می توانید در کدهای نمونه ای که برای این کتاب فراهم شده ، پیدا کنید :



همان طور که می بینید ، viewport ها شدیدا برای داشتن چندین دوربین به صورت هم زمان ، مفید اند . این باعث می شود ، ایجاد صفحه نمایش دوبخشی برای بازی های چندبازیکنه (multiplayer) خیلی آسان باشد . قطع نظر از این ، می توانیم از دوربین های متفاوتی برای رابط کاربری (UI) استفاده کنیم . برای انجام این کار ، بازی را با دوربین بازی رندر می کنیم ، سپس به دوربین UI سوئیچ می کنیم (که همراه با دنیای بازی حرکت نمی کند) و رابط کاربری بازی را رندر می کنیم . نقشه ها هدفی عالی برای دوربین ها هستند چون ما در اصل دنیا را در داخل پنجره دیگری (یا دوربین) رندر می کنیم .

به طور کلی دوربین ها خلی مفیدند ، اما یک نقطه ضعف دارند : هنگامی که شروع به دستکاری دوربین می کنیم ، مختصات پنجره با محتویات دوربین متناظر نخواهد بود . مثلا اگر سعی کنیم یک رویداد کلیک را مدیریت کنیم ، برای دوربین های مختلف ، موقعیت ماوس به همان نقطه موجود در صحنه اشاره نخواهد کرد . مثل همیشه ، SFML با معرفی نقشه برداری مختصات ما را نجات می دهد .

نقشه برداری مختصات

هنگامی که یک دوربین متصل به یک پنجره داشته باشیم ، می توانیم `RenderWindow::mapPixelToCoords()` را همراه با یک موقعیت از پنجره فرابخوانیم و این متد ، بردار موقعیت را به مکانی در صحنه (مختصات دنیای بازی) تبدیل کند . مثال زیر تبدیل مختصات ماوس به مختصات دنیای بازی را در هنگام رویداد فشرده شدن یک دکمه نشان می دهد :

```
sf::Event ev;
while (window.pollEvent(ev))
{
    if (ev.type == sf::Event::MouseButtonPressed)
    {
        sf::Vector2f sceneCoords = window.mapPixelToCoords(
            sf::Vector2i(ev.mouseButton.x, ev.mouseButton.y));

        //Do something at that location in the scene
    }
}
```

مهم است که به یاد داشته باشید که این تابع فقط با `View` کار می کند که در حال حاضر به پنجره اعمال شده است . اگر از چندین دوربین برای پنجره استفاده می کنیم ، به این موضوع باید توجه کنیم .
همچنین می توانیم برعکس هم عمل کنیم : از یک مکان در صحنه ، مختصات صفحه را به دست آوریم . این کار توسط `RenderWindow::mapCoordsToPixel()` انجام می شود . هنگامی که می خواهیم یک مکان را از یک صحنه به دوربین های دیگری انتقال دهیم ، این مفید است . فرض کنید می خواهید یک نمودار سلامتی بر روی کاراکترها نمایش دهید ، مکان آنها را از صحنه می گیریم ، مکان را به یک پیکسل صفحه تبدیل کرده سپس آن پیکسل را به مختصات دوربین تبدیل می کنیم و نمودار سلامتی را در آنجا نمایش می دهیم .

این مباحث ، موضوع `view` را پوشش دادند . بخش بعدی این فصل ، ادغام کدهای OpenGL در داخل SFML را بررسی خواهد کرد .

OpenGL چیست ؟

OpenGL یک API گرافیکی مستقل از پلتفرم است که به عنوان واسطی برای ارتباط با کارت گرافیک استفاده می شود . مهمترین خصوصیت هر API گرافیکی ، توانائی آن برای رندر اشیاء بر روی صفحه است . درحالیکه مطمئنا OpenGL این کار را انجام می دهد ، خصوصیات مفید دیگری هم دارد . چون تکنولوژی GPU به سرعت تغییر کرده است ، همه خصوصیات ورژن های جدید OpenGL در همه کارت های گرافیک پشتیبانی نمی شوند .

آیا شما باید از OpenGL استفاده کنید ؟

SFML بیشتر کارکردی که OpenGL فراهم کرده است را پشتیبانی می کند . در حقیقت SFML به صورت درونی از OpenGL برای پیاده سازی عملکردهایش استفاده می کند . اما چون SFML یک کتابخانه سطح بالاست ، با استفاده از آن همیشه هزینه اتلاف کارائی وجود دارد . در بیشتر موارد ، این موضوع مشکل بزرگی نیست . اما بعضی مواقع ، نیاز به کارائی بیشتری است تا FPS هدف به دست آید . در این مورد ، SFML ادغامی ساده از کدنویسی OpenGL را فراهم کرده بدون اینکه درباره بیشتر چیزها نگرانی داشته باشیم .

شاید بخواهید یک باره ده هزار sprite را روی صفحه رندر کنید ، یا بخواهید خصوصیتی را به کلاس `Window` اضافه کنید . OpenGL مکانی برای شروع کار خواهد بود .

دلیل دیگر برای استفاده از OpenGL ، ساختن بازی های سه بعدی است . SFML چند ابزار را فراهم کرده که می توانند در محیط سه بعدی استفاده شوند ، اما آنها برای تولید یک محصول سه بعدی کامل (مدل های سه بعدی ، نورپردازی ، سایه ها و ...) کافی نیستند . بنابراین باید از OpenGL استفاده کنیم .

OpenGL یک API خیلی بزرگ است و این فصل به شما یاد نمی دهد که چطور از خصوصیات آن استفاده کنید . این فصل فقط یک راهنمای کلی درباره چگونگی استفاده از آن در کنار SFML به شما می دهد .

استفاده از OpenGL در داخل SFML

قبل از اینکه شروع به استفاده از فراخوان های OpenGL کنیم ، باید مطمئن شویم که فضای گرافیکی مقداردهی اولیه شده باشد . این فضا داده هائی را نگه می دارد (وضعیت ها ، framebuffer پیش فرض و ..) که اجازه می دهد OpenGL کار کند . هنگامی که یک نمونه از `window` را تولید می کنیم ، این به صورت اتوماتیک انجام می شود :

Main.cpp

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

int main()
{
    sf::ContextSettings settings;
    settings.depthBits = 24;
    settings.stencilBits = 8;
    settings.majorVersion = 3;
    settings.minorVersion = 0;
    settings.antialiasingLevel = 2;
    sf::Window window(sf::VideoMode(640, 480), "OpenGL", sf::Style::Default, settings);

    //Window is ready to receive OpenGL calls here

    while (window.isOpen())
    {
        //Game loop
    }

    return 0;
}
```

برای استفاده از فراخوان های OpenGL باید `<SFML/OpenGL.hpp>` را داخل پروژه خود کنیم . اگر بخواهیم همه چیز را فقط با استفاده از OpenGL رندر کنیم ، نیازی به استفاده از کلاس `RenderWindow` نیست . کلاس `Window` مناسب خواهد بود . از طرف دیگر ، اگر بخواهیم به دلایلی OpenGL را همراه با مدل گرافیک استفاده کنیم ، در این صورت باید از `RenderWindow` استفاده کنیم . در این مثال قصد داریم فقط از مدل window (پنجره) استفاده کنیم که کلاس `Window` را نگه می دارد .

توجه کنید که پنجره یک نمونه از `ContextSettings` (پارامتری اختیاری است) را می گیرد . ما کمی درباره این در فصل ۱ صحبت کردیم اما اجازه دهید تا ببینیم که این تنظیمات چه معنی دارند .

ساختار ContextSettings دارای ۵ فیلد است که ما می توانیم آنها را تغییر دهیم .

depthBits

این فیلد به ما اجازه می دهد تعداد بیت ها در هر پیکسل را برای بافر عمق پیشنهاد دهیم .

محدوده مجاز : [0, 8, 16, 24, 32]

stencilBits

این فیلد به ما اجازه می دهد تعداد بیت ها در هر پیکسل را برای بافر استنسیل پیشنهاد دهیم .

محدوده مجاز : [0, 8]

majorVersion

این فیلد به ما اجازه می دهد ورژن بزرگ OpenGL را پیشنهاد دهیم .

محدوده مجاز : [1...4]

minorVersion

این فیلد به ما اجازه می دهد ورژن کوچک OpenGL را پیشنهاد دهیم .

محدوده مجاز : [1...5]

antialiasingLevel

این فیلد به ما اجازه می دهد سطح multisampling را پیشنهاد دهیم .

محدوده مجاز : [0...16] . معمولا توان ۲ بهترین نتیجه را می دهد : 1, 2, 4, 8, 16

این مقادیر SFML را مجبور به استفاده از آنها نمی کند و اگر سعی کنیم از آنها بر روی سخت افزاری استفاده کنیم که اینها را پشتبانی نمی کند ، هیچ استثنائی اعلام نمی شود . SFML نزدیکترین و احتمالا بهترین گزینه ای که روی سیستم پشتیبانی می شود را انتخاب می کند . ما می توانیم بررسی کنیم که چه گزینه هائی انتخاب شده اند . این کار با گرفتن `ContextSettings` از پنجره انجام می شود :

```
auto wSettings = window.getSettings();
std::cout << "depthBits: " << wSettings.depthBits << std::endl;
std::cout << "stencilBits: " << wSettings.stencilBits << std::endl;
std::cout << "antialiasingLevel: " << wSettings.antialiasingLevel << std::endl;
std::cout << "version: " << wSettings.majorVersion << "." << wSettings.minorVersion << std::endl;
```

نتیجه این کد را ببینید :

```
depthBits: 24
stencilBits: 8
antialiasingLevel: 2
version: 4.4
```

بعد از نصب همه چیز ، می توانیم حلقه بازی را ایجاد کنیم :

```
while (window.isOpen())
{
    sf::Event ev;
    while (window.pollEvent(ev))
    { /* Handle events */ }

    //Update frame

    //Set red clear color;
    glClearColor(1, 0, 0, 1);
    //Clear the screen and the depth buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    //Render things here

    //SwapBuffers
    window.display();
}
```

چون پنجره SFML طوری ساخته شده که با OpenGL کار کند ، ادغام آن به سادگی کدهای بالاست . اما هنگام ترکیب مدل گرافیک و OpenGL کمی متفاوت می شود . هر بار که بین رندرینگ با SFML و رندرینگ با OpenGL سوئیچ می کنیم ، باید وضعیت OpenGL را ذخیره و بازیابی کنیم . هنگامی که شکل ها را هم با استفاده از مدل گرافیک و هم با OpenGL ترسیم می کنیم ، کد زیر باید استفاده شود :

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

//Draw shape using OpenGL

window.pushGLStates();

//Draw shape using SFML

window.popGLStates();

//Continue drawing using OpenGL

//SwapBuffers
window.display();
```

کد قبلی برای مثال های کوچکی که تعداد زیادی منابع نیاز ندارند یا کلاس ندارند ، مناسب است .

فرض کنید یک کلاس `GameObejct` داریم که دارای دو متد `GameObject::update()` و `GameObject::render()` است . در متد `update()` ما منطق بازی را مدیریت می کنیم و در `render()` شئ را رندر می کنیم . حلقه اصلی ما به شکل زیر می شود :

```
//Update frame
for (auto it = gObjects.begin(); it != gObjects.end(); ++it)
{
    it->update();
}

//Render frame
window.clear(sf::Color::Black);

for (auto it = gObjects.begin(); it != gObjects.end(); ++it)
{
    it->render(window);
}

window.display();
```

حالا فرض کنید چند کارکرد بیشتر دیگر از کلاسمان می خواهیم که دوست باریم با استفاده از OpenGL آنها را پیاده سازی کنیم . ساختار کلاس ما به شکل زیر می شود :

```
class GameObjectGL : public GameObject
{
    void render(sf::RenderWindow& window) override
    {
        //Render object using OpenGL
    }
};

class GameObjectSFML : public GameObject
{
    void render(sf::RenderWindow& window) override
    {
        window.pushGLStates();
        //Render object using Graphics module
        window.popGLStates();
    }
};
```

چون `RenderWindow::pushGLStates()` یک عملیات پرهزینه است ، به نظر می رسد کلاس `GameObjectSFML` ما بسیار ناکارامد است ، این طور نیست ؟ ذخیره و بازیابی وضعیت ها در OpenGL برای هر شئ ای که از SFML استفاده می کند ، اتلاف منابع است . یک راه حل برای این مشکل ایجاد دومین تابع رندر در کلاس پایه است که می تواند `GameObject::renderGL()` نامیده شود . در حلقه اصلی کمی چیزها فرق خواهد کرد :

```
        //Render frame
        window.clear(sf::Color::Black);

        //Call GameObject::renderGL() on all objects

        window.pushGLStates();

        //Call GameObject::render() on all objects

        window.popGLStates();

        window.display();
```

با این شیوه بسیاری از فراخوان های غیر ضروری درایور را حذف کرده و کد داخل کلاس `GameObject` واضح تر خواهد شد . هر شئ ای که از OpenGL استفاده نمی کند ، این متد را خالی دریافت می کند در حالیکه بقیه `GameObject::render()` را خالی دریافت می کنند .

OpenGL در چندین پنجره

SFML به ما اجازه می دهد از چندین پنجره در هر برنامه استفاده کنیم . برای بازی ها ، داشتن بیش از یک پنجره کاملا غیر معمول است ، اما برای برنامه های چند رسانه ای ، خیر . هنگامی که می خواهیم یک شئ را در یک پنجره خاص رندر کنیم ، `RenderWindow::draw()` را فرامی خوانیم . اما هنگامی که می خواهیم از OpenGL استفاده کنیم ، باید مشخص کنیم که کدام پنجره توسط این فراخوان دچار تاثیر می شود . این کار به سادگی توسط `Window::setActive()` انجام می شود . هنگامی که می خواهیم رندرینگ را روی یک پنجره شروع کنیم ، کافی است `setActive()` را فراخوانده و شروع به استفاده از OpenGL کنیم .

در اینجا درس ما درخصوص OpenGL به پایان می رسد .

خلاصه

در این فصل ، ما درخصوص اهمیت دوربین ها و چگونگی کارکرد آنها در SFML صحبت کردیم . دیدم که چطور کلاس `View` را تغییر شکل داده و چطور صفحه نمایش دو قسمتی را برای بازی های چندبازیکنه آماده کنیم . در بخش دوم این فصل درباره OpenGL و اینکه چطور با SFML ارتباط دارد ، صحبت کردیم .

در فصل بعد ، سه جزء حیاتی بازی ها یعنی صدا ، موسیقی و متن را بررسی خواهیم کرد .

فصل ۵ : اکتشاف دنیای صداها و متون

از هنگام شروع کار با SFML راه طولانی آمده ایم ، و قبل از رسیدن به انتها ، مسیر کمی باقی مانده است . این فصل خصوصیات صوتی SFML را تست می کند که در داخل مدل صوتی (audio) قرار دارد . این شامل صدا ، موسیقی و محیط صدای سه بعدی است . آخرین بخش این فصل چگونگی رندر متن بر روی صفحه را توضیح خواهد داد .

در این فصل مباحث زیر بررسی می شود :

مقدمه ای بر مدل صوتی

صدا در مقابل موسیقی

صدا در عمل

`sf::SoundSource` و صدا در محیط سه بعدی

شروع کار با `sf::Text`

مقدمه ای بر مدل صوتی

تا کنون فقط از مدل های پنجره ، گرافیک و سیستم SFML استفاده کرده ایم . مدل پنجره ، پنجره های سیستم عامل و خصوصیات مربوط به آنها را مدیریت می کند . مدل گرافیک ترسیم اشیاء بر روی صفحه را برای ما آسان می کند . مدل سیستم کلاس های vector را نگه داشته و خصوصیات سیستم عامل مثل کلاس های `Clock` و `Time` را بسته بندی کرده است . دو مدل دیگر در SFML وجود دارد : صوتی و شبکه . این فصل درباره مدل صوتی و خصوصیات آن است . مهمترین کلاس های موجود در این مدل ، `sf::Sound` و `sf::Music` است . اینها به ما راهی برای پخش صداها و موسیقی می دهند . خصوصیات دیگری هم وجود دارد که به شما اجازه می دهد صداهای سه بعدی را پخش کنید ، یعنی اینکه صدا بسته به موقعیت و جهت شنونده ، از جهت های مختلفی پخش می شود . ما همه این خصوصیات را به صورت مفصل در این فصل بحث می کنیم . چند چیز دیگر در این مدل وجود دارد که خارج از حد این کتاب است مثل کلاس `SoundRecorder` که می تواند صداها را از دستگاه های ورودی (مثلا میکروفون) ضبط کند .

حالا آماده ایم که کار با مدل صوتی را شروع کنیم . اولین چیزی که باید درباره آن صحبت کنیم ، این است که دو کلاس وجود دارد که شما می توانید با کمک آنها صدا را پخش کنید : کلاس `Sound` و `Music` .

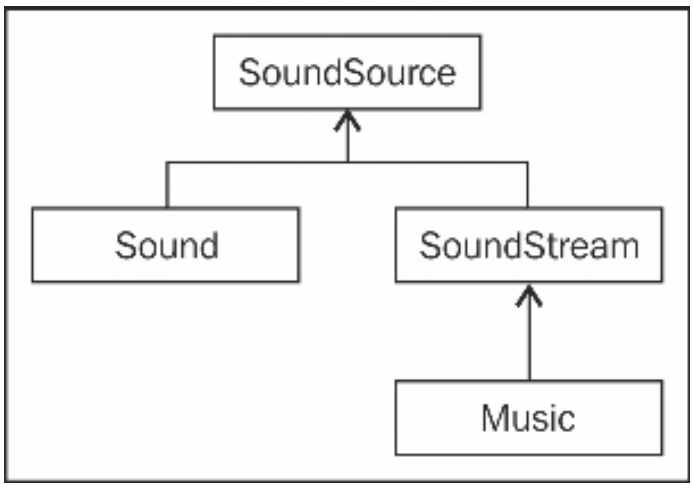
از بیرون به نظر می رسد که اینها یک کار انجام می دهند : پخش یک فایل صوتی ، اما مشهور است که می گویند " درباره یک کتاب با جلدش قضاوت نکن ! "

صدا در مقابل موسیقی

در نگاه اول وجود این دو کلاس ممکن است عجیب باشد اما آنها کاملا اهداف متفاوتی دارند و این به خاطر نوع پیاده سازی آنهاست . کلاس Sound تمام داده هایش را داخل حافظه سیستم بارگذاری می کند و این کار باعث می شود پخش نمونه صوتی خیلی سریع انجام شود . از آن سمت ، کلاس Music یک استریم به یک فایل را روی هارد دیسک (یا RAM) باز می کند و قطعات کوچکی از داده را بارگذاری می کند که یکی پس از دیگری پخش می شوند . به علت این نوع طراحی ، کلاس Music تاخیر در بازپخش دارد چون داده ها را از یک چنین مکان کندی ، انتقال می دهد .

هر دو کلاس مزیت های متفاوتی دارند . کلاس Sound تقریبا بلافاصله پخش می شود اما حافظه زیادی از سیستم می گیرد . در حالیکه کلاس Music در هنگام پخش کند تر است اما RAM زیادی نمی خواهد . هر دو کلاس در موقعیت های متفاوتی مفید هستند ، برای مثال اگر فایل صوتی آنقدر کوچک باشد که بتوان در حافظه سیستم ذخیره کرد ، باید آن را با استفاده از کلاس Sound بارگذاری کرد . هنگامی که صدا باید بلافاصله پس از فراخوان متد play() پخش شود ، این وضعیت قابل اجراست . هنگامی که فایل خیلی بزرگ است و باید بلافاصله پخش شود ، باید حافظه را قربانی کنیم . این موضوع علت مهم بودن مدیریت منابع است . کلاس Music اغلب برای فایل های صوتی بزرگ استفاده می شود ، در جائی که اگر بازپخش در هنگام شروع کمی با تاخیر همراه باشد ، زیاد مهم نخواهد بود . این کلاس اغلب برای موسیقی های پس زمینه بازی استفاده می شود .

Sound و Music در ساختار درختی وراثتشان ، یک کلاس دارند : SoundSource . این کلاس یک عملکرد عمومی صوتی فراهم کرده از قبیل تغییر در زیر و بمی صدا و موقعیت یابی سه بعدی . کلاس Sound از این به صورت مستقیم ، استخراج می شود در حالیکه Music از SoundStream استخراج می شود. در نهایت SoundStream از SoundSource استخراج می شود . تصویر زیر رابطه اینها را نشان می دهد :



در ادامه این فصل ، چیزهای بیشتری درباره SoundSource و SoundStream می گوئیم .

تمام این بخش را می توان به صورت زیر خلاصه کرد :

از Sound برای پخش افکت های صوتی (صدای شلیک اسلحه ، صدای قدم برداشتن کاراکتر و ...) و از Music برای پخش موسیقی پس زمینه استفاده کنید .

حالا که فرق بین این دو کلاس را فهمیدید ، اجازه دهید کمی کدنویسی کنیم .

صدا در عمل

ما با افکت های صوتی شروع می کنیم و بعدا به سراغ موسیقی می رویم .

کلاس sf::Sound

یک صدا از دو کلاس تشکیل شده : `Sound` و `SoundBuffer` .

کلاس `SoundBuffer` منبعی در حافظه است و `Sound` بسته بندی است که منبع را پخش می کند . این درست یادآور رابطه `Sprite` و `Texture` است . `Sprite` از `Texture` به عنوان یک منبع استفاده می کرد . با این نوع طراحی ، چندین نمونه `Sound` می توانند از یک نمونه `SoundBuffer` استفاده کنند و مقدار حافظه مورد نیاز را به طور قابل ملاحظه ای کاهش دهند . توضیح کافی است . بینیم چطور یک صدا ایجاد می شود :

Main.cpp

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


int main()
{
    sf::Window window(sf::VideoMode(640, 480), "Audio");

    sf::SoundBuffer sBuffer;
    if (!sBuffer.loadFromFile("mySound.ogg"))
        return -1; //Failed to load

    sf::Sound sound(sBuffer);

    while (window.isOpen())
    {
        //Game loop
    }

    return 0;
}
```

ما از همان شیوه بارگذاری `Texture` برای `SoundBuffer` استفاده کردیم . بعضی از فرمت های صوتی که پشتیبانی می شوند عبارتست از :

OGG, WAV, FLAC , …

توجه کنید که MP3 به علت مجوزش پشتیبانی نمی شود .

مثل `sf::Texture` ، بافرهای صوتی هم متدی برای بارگذاری از حافظه دارند
(`SoundBuffer::loadFromMemory()`) و همچنین برای بارگذاری از استریم
(`SoundBuffer::loadFromStream()`)

یک بافت می تواند از آرایه ای از پیکسل ها هم بارگذاری شود اما در خصوص بافر های صوتی ، این منطقی نیست . در عوض کلاس `SoundBuffer` متدی برای بارگذاری از آرایه ای از نمونه ها دارد : `SoundBuffer::loadFromSamples()`

به محض تولید `sf::Sound` توسط سازنده اش و پاس دادن بافر صوتی به آن ، می توانیم آن را با `Sound::play()` پخش کنیم . متد بسته به وضعیت حال حاضر `Sound` کار متفاوتی را انجام می دهد . متد صدا را در نخ اجرائی دیگری پخش می کند بنابراین نخ اجرائی جاری بلوکه نمی شود .

هر شئ `SoundSource` یک `SoundSource::Status` (شمارش) مربوط به خود دارد . آن می تواند یکی از این سه وضعیت باشد : **متوقف شده** ، **مکث شده** یا **در حال پخش** . این سیستم وضعیت به صورت درونی توسط کلاس `SoundSource` استفاده می شود تا رفتار متدهایش را کنترل کند . با فراخوان متد `getState()` می توان وضعیت یک شئ `Sound` یا `Music` را گرفت .

به `Sound::play()` برگردیم ، در وضعیت متوقف شده یا مکث شده ، متد از موقعیت حال حاضر پخش ، شروع به پخش مجدد می کند . اگر در وضعیت در حال پخش باشد ، پخش را دوباره از ابتدا انجام می دهد . این یعنی هنگامی که می خواهیم یک صدا را چندین بار پخش کنیم ، باید مراقب باشیم . بهترین راه حل این کار این است که هر بار که صدا باید پخش شود ، چندین نمونه از `Sound` ایجاد کنیم . چون همه این نمونه ها از یک `SoundBuffer` استفاده می کنند ، ایجاد صداها سریع خواهد بود بنابراین نباید نگران حافظه یا ضربه خوردن به کارائی برنامه باشیم .

کلاس `Sound` برای توقف و مکث صدا متدهای `Sound::stop()` و `Sound::pause()` را دارد . متد `Sound::stop()` صدا را متوقف کرده و اگر در حال حاضر در وضعیت در حال پخش یا مکث شده باشد ، موقعیت پخش را به ابتدا برمی گرداند . اگر صدا در وضعیت متوقف شده است ، متد کاری انجام نمی دهد . به همین شکل ، `Sound::pause()` بازپخش صدا را متوقف می کند اما موقعیت پخش را به ابتدا برنمی گرداند . اگر صدا در وضعیت متوقف شده یا مکث شده باشد ، متد کاری انجام نمی دهد .

ما می توانیم با `Sound::setLoop()` تعیین کنیم که صدا قابلیت تکرار و پخش مجدد داشته باشد یا خیر . اگر صدا تکرار شود ، در انتهای پخش از شروع دوباره پخش خواهد شد . به صورت پیش فرض هیچ صدائی دارای قابلیت تکرار نیست .

کلاس `sf::Sound` خصوصیت دیگری دارد که با فراخوان `Sound::setPlayingOffset()` به کارگیری می شود . این متد `sf::Time` می گیرد و موقعیت پخش را با این زمان تنظیم می کند . این می تواند در هر دو وضعیت مکث شده و در حال پخش ، استفاده شود .

چیز مهم دیگری وجود دارد که باید درباره آن مفصل صحبت کنیم و آن مدیریت منابع است . مثل کلاس های `Texture` و `Sprite` ، نمونه `SoundBuffer` هم باید هنگامی که `Sound` از آن استفاده می کند ، زنده و موجود باقی بماند . برای مدیریت آسان تر طول عمر یک شئ `sf::Texture` ما از کلاس `AssetManager` خودمان استفاده کردیم ، حالا قصد داریم این کار را برای `SoundBuffer` هم انجام دهیم .

معرفی AssetManager 2.0

مدیریت کننده منابع قبلی ما فقط بافت ها را بارگذاری و نگهداری می کرد . قصد داریم با اضافه کردن پشتیبانی برای اشیاء `SoundBuffer` آن را بهبود ببخشیم . هنگام بارگذاری این منابع ، هر دو کلاس شبیه هم هستند بنابراین کدهای ما برای `SoundBuffer` هم تقریبا مثل بافت هاست . فایل به روز شده `AssetManager.h` را ببینید :

AssetManager.h

```
#ifndef ASSET_MANAGER_H
#define ASSET_MANAGER_H

#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>
#include <map>

class AssetManager
{
public:
    AssetManager();

    static sf::Texture& GetTexture(std::string const& filename);
    static sf::SoundBuffer& GetSoundBuffer(std::string const& filename);

private:
    std::map<std::string, sf::Texture> m_Textures;
    std::map<std::string, sf::SoundBuffer> m_SoundBuffers;

    //AssetManager is a singleton, so only one instance can exist at a time
    //sInstacne holds a static pointer to the single manager instance
    static AssetManager* sInstance;
};

#endif
```

ما از مدل صوتی برای بارگذاری بافرهای صوتی استفاده کردیم ، بنابراین باید فایل هدر مدل صوتی را داخل پروژه کنیم . در `AssetManager` علاوه بر کدهائی که بافت ها را مدیریت می کنند یک map برای ذخیره بافرهای صوتی و یک متد برای بارگذاری و بازیابی آنها می بینیم . پیاده سازی `AssetManager::GetSoundBuffer()` را ببینید :

```
sf::SoundBuffer& AssetManager::GetSoundBuffer(std::string const& filename)
{
    auto& sBufferMap = sInstance->m_SoundBuffers;

    auto pairFound = sBufferMap.find(filename);
    if (pairFound != sBufferMap.end())
    {
        return pairFound->second;
    }
    else
    {
        //Create an element in the SoundBuffer map
        auto& sBuffer = sBufferMap[filename];
        sBuffer.loadFromFile(filename);
        return sBuffer;
    }
}
```

این کد همان ساختار کدهای `AssetManager::GetTexture()` را دارد . ابتدا سعی می کنیم تا بفهمیم آیا یک بافر صوتی با اسم داده شده ، از قبل وجود دارد یا خیر . اگر باشد ، آن را بازگشت می دهیم . اگر نباشد ، یک ورودی در map بافرهای صوتی ایجاد کرده ، بافر را با یک فایل بارگذاری کرده و بافر جدید را بازگشت می دهیم .

خارج `AssetManager` می توانیم به شکل زیر صداها را ایجاد کنیم :

```
int main()
{
    sf::Window window(sf::VideoMode(640, 480), "Audio");
    //Remember, we need an instance of the asset manager
    AssetManager manager;

    sf::Sound sound(AssetManager::GetSoundBuffer("mySound.ogg"));
    sound.play();

    while (window.isOpen())
    {
    }

    return 0;
}
```

با استفاده از مدیریت کننده منابعمان ، می توانیم مطمئن باشیم که منابع ما به صورت اتفاقی از حافظه حذف یا جابه جا نمی شوند . همچنین منابع با اسم یکسان را می توان ذخیره کرده و هنگامی که این منبع را در چندین مکان بخواهیم ، کافی است آن را دریافت کنیم چون قبلا یک بار بارگذاری شده است . این شیوه مقدار زیادی از حافظه را ذخیره می کند .

وقت آن است تا ببینیم که کلاس `Music` چه چیزی برای ما تدارک دیده است !

sf::Music و sf::SoundStream

پخش یک فایل صوتی با استفاده از کلاس Music به سادگی زیر است :

```
sf::Music music;
if (!music.openFromFile("myMusic.ogg"))
    return -1;
music.play();
```

هنگامی که یک استریم موسیقی را از یک فایل باز می کنیم ، فرمت های صوتی پشتیبانی شده همان هائی هستند که برای Sound بودند . همچنین Music::play() صدا را در نخ اجرائی دیگری اجرا می کند بنابراین نیازی به نگرانی درباره بلوکه شدن نخ اجرائی جاری نیست .

یک فایل موسیقی می تواند از یک فایل بارگذاری شده در حافظه (Music::openFromMemory()) یا از InputStream (Music::openFromStream()) استریم شود . کلاس موسیقی از کلاس SoundStream استخراج شده است ، که وظایفی عمومی را برای استریم صوتی پیاده سازی می کند . آن همه داده ها را در حافظه سیستم ، بارگذاری نمی کند بلکه قطعات کوچکی از منبع را بارگذاری کرده و با آنها کار می کند . هنگامی که به ناحیه ای از صوت می رسد که در حافظه وجود ندارد ، داده های بیشتری را از استریم درخواست کرده و شروع به کار با آنها می کند . کلاس Music بر روی این واسط کار می کند و فقط متدهائی برای باز کردن یک استریم (با متدهای Music::open*()) و تغذیه اینها با SoundStream پائینی فراهم کرده است .

چون با داده های استریم سروکار داریم ، هنگای که SoundStream در حال استفاده از آنهاست ، داده ها نمی توانند تخریب شوند ، هنگامی که از داده هائی استفاده می کنیم که قبلا در حافظه سیستم تخصیص پیدا کرده اند ، باید مطمئن شویم که زنده و موجود باقی می مانند .

همانطور که قبلا گفتیم ، کلاس Music کارکرد بیشتری از SoundStream فراهم نکرده است البته به جز متدهائی که استریم ها را باز می کند . تمام عملکرد جذاب در کلاس پایه SoundStream قرار گرفته ، متدهائی مثل SoundStream::play() , SoundStream::pause() و SoundStream::stop() .

این سه متد دقیقا به شیوه کلاس Sound کار می کنند . متدهای SoundStream::setPlayingOffset() و SoundStream::setLoop() هم وجود دارند .

در خصوص AssetManager ، نباید یک ورودی برای اشیاء Music اضافه کنیم چون منابع داخلی آن مثل SoundBuffer قابل استفاده مجدد نیستند . اگر از Music::openFromMemory() برای بارگذاری فایل های موسیقی استفاده می کردیم ، این منابع را به مدیریت کننده منابعمان اضافه می کردیم اما استفاده از این گزینه بسیار نادر است چون بارگذاری همه یک فایل از نظر حافظه خیلی پرهزینه خواهد بود .

حالا وقت آن است که به صداها در محیط سه بعدی نگاهی بیاندازیم .

sf::SoundSource و صدا در محیط سه بعدی

همانطور که قبلا بحث شد کلاس های Sound و SoundStream از SoundSource استخراج شده اند . ابتدا درخصوص چند خصوصیت عمومی صحبت می کنیم و سپس درباره چگونگی استفاده از صداهای سه بعدی صحبت خواهیم کرد .

خصوصیات عمومی صداها

با تولید صحنه های پیچیده و پیچیده تر ، چیزهای بیشتری از صداها و موسیقی می خواهیم . برای مثال ممکن است بعضی از صداها بلندتر از بقیه باشد ، و بخواهیم هنگام اجرای بازی بلندی صدای آنها را کمتر کنیم . این کار را می توان با SoundSource::setVolume() انجام داد . مقادیر صدای پشتیبانی شده از صفر (بی صدا) تا 100 است و هر منبع صدا با مقدار پیش فرض 100 شروع می شود . می توانیم بلندی صدای جاری را با SoundSource::getVolume() به دست آوریم .

یکی دیگر از مشخصات صدا، زیر و بمی است . زیر و بم فرکانس صدا را دریافت می کند . برای مثال نت موسیقی *C4* زیروبمی پائین تری از نت *E4* دارد . می توانیم فاکتور زیروبمی یک منبع اصلی صدا را با فراخوان SoundSource::setPitch() تغییر دهیم . این متدی برای افزودن زیروبمی است که سرعت بازپخش را کم و زیاد می کند . مقدار پیش فرض فاکتور 1 است که زیروبمی صدای اصلی را تغییری نمی دهد . می توان با فراخوان SoundSource::getPitch() زیروبمی جاری را به دست آورد .

صدا در محیط سه بعدی

هر صدائی که تاکنون استفاده کرده ایم ، با بلندی صدای کامل پخش شده است (فرض می کنیم که بلندی صدا را با SoundSource::setVolume() تغییری نداده ایم) . در این بخش درباره این موضوع صحبت می کنیم که چطور صداها را در دنیای بازی قرار دهیم طوری که نسبت به یک شنونده صدا پخش شوند (مثلا کاراکتر اصلی) . با انجام این کار ، می توانیم یک محیط صوتی واقعی را شبیه سازی کنیم طوری که اگر چیزی در سمت چپ کاراکتر منفجر شود ، صدای انفجار را فقط از بلنگوی سمت چپ بشنویم .

قبل از ادامه بحث ، مهم است توجه کنید که یک صدا اگر فقط یک کانال داشته باشد (یک صدای mono) می تواند سه بعدی شود . سه بعدی کردن منابع دارای چندین کانال زیاد منطقی نیست چون آنها به صورت صریح تعریف کرده اند که چطور از اسپیکرها استفاده کنند .

تولید یک محیط صوتی نیازمند دو چیز است : کسی یا چیزی که صداها را پخش کند و کسی یا چیزی که به آنها گوش کند . در بخش های قبلی این فصل ، درباره چگونگی پخش صدا و موسیقی صحبت کردیم . اما آن صداها سه بعدی نبودند و نیازی به یک شنونده نداشتند . آنها فقط با بلندی صدای کامل از بلندگوها پخش می شدند . اگر بخواهیم آنها را در یک دنیا و محیط قرار دهیم و محیطی واقعی ایجاد کنیم ، باید یک شنونده نصب کنیم که توانائی گوش دادن به آنها را داشته باشد . این کار خصوصیاتی مثل جهت و تضعیف قدرت صوت را در هنگام پخش صدا ، فعال می کند .

نصب یک شنونده

در هر لحظه ای از اجرای برنامه ما ، ما فقط به یک شنونده برای شنیدن صداها نیاز داریم . SFML یک کلاس استاتیک برای مدیریت شنونده صدا فراهم کرده است : sf::Listener

شنونده سه مشخصه دارد که ما می توانیم آنها را دستکاری کنیم : موقعیت ، جهت و بلندی صدای سراسری

موقعیت شنونده توسط فراخوان `Listener::setPosition()` تنظیم می شود . این تابع یک بردار سه بعدی می گیرد چون SFML فرض نمی کند که ما در حال ساخت یک بازی دوبعدی هستیم . در بازی دوبعدی دنیای ما مسطح خواهد بود و نیازی به هر سه محور نداریم . ما می توانیم از محورهای *x* و *y* *استفاده کرده و* *z = 0* را استفاده کنیم . بنابراین اگر یک sprite به عنوان کاراکتر اصلیمان داشته باشیم ، باید هرباری که sprite حرکت می کند ، موقعیت شنونده را با موقعیت sprite تنظیم کنیم :

```
sf::Sprite heroSprite(AssetManager::GetTexture("myHero.png"));

while (window.isOpen())
{
    /* Update the hero Position here */

    //Set the listener to the hero's position
    sf::Vector2f heroPos = heroSprite.getPosition();
    sf::Listener::setPosition(heroPos.x, heroPos.y, 0);
}
```

قطع نظر از موقعیت ، می توانیم با فراخوان `Listener::setDirection()` جهت شنونده را تنظیم کنیم . این جهتی را که شنونده به آن سمت است تعریف می کند . اگر بازی ما سبک تیراندازی بالا به پائین باشد ، کاراکتر اصلی می تواند 360 درجه بچرخد . با تابع `Listener::setDirection()` شنونده می تواند چنین رفتاری را مدیریت کند . این تابع یک بردار سه بعدی می خواهد که نشان دهنده جهت شنونده است . چون این فقط یک جهت است ، بردار پاس داده شده باید نرمال سازی شده باشد اما SFML این را اجبار نمی کند .

در خصوص مثالی که در آن یک sprite قهرمان بازی بود ، به شکل زیر می توانیم بردار جهت را برای شنونده محاسبه کنیم :

```
#define PI_RADIANS 3.1415f
#define PI_DEGREES 180.f

sf::Sprite heroSprite(AssetManager::GetTexture("myHero.png"));

while (window.isOpen())
{
    /* Update the hero Position here */

    //Transform the rotation to radians
    float heroRot = heroSprite.getRotation() * PI_RADIANS / PI_DEGREES;
    //Set the listener's direction from the hero's rotation
    sf::Listener::setDirection(std::cos(heroRot), std::sin(heroRot), 0);
}
```

این کدها از فرمول تبدیل یک زاویه به یک بردار دوبعدی جهت با استفاده از مقادیر سینوس و کسینوس آن زاویه ، استفاده می کند . مهم است که توجه کنید که زاویه باید به رادیان باشد تا تابع مثلثاتی کار کند . این موضوع علت تبدیل زاویه از درجه به رادیان با استفاده از فرمول زیر است :

Angle_Radians = Angle_Degrees* PI_Radians / PI_Degrees

آخرین خصوصیتی که می توانیم برای یک شنونده نصب کنیم ، بلندی صدای سراسری است . این می تواند با فراخوان `Listener::setGlobalVolume()` انجام شود . این تابع یک `float` در محدوده [0...100] می خواهد .

همچنین تابع های `Listener::get*()` برای هر یک از مشخصات گفته شده در این بخش نیز وجود دارد .

منابع صوتی

ما اولین بخش از محیط صوتی یعنی شنونده را توضیح دادیم که مسئول گوش کردن به صداهای در حال پخش در محیط است . اما بدون اینکه صدائی در حال پخش باشد ، داشتن شنونده بیهوده خواهد بود . در این بخش کارکرد کلاس `SoundSource` در را رابطه با صدای سه بعدی را بررسی می کنیم .

اگر ما هیچ کدام از مشخصات `Listener` و `SoundSource` تک کاناله را دست نزنیم ، آنها در همان مکان می مانند و طوری ظاهر خواهند شد که انگار با بلندی صدای کامل در حال پخش اند . اما هنگامی که شروع به حرکت `Listener` از مبداء (0; 0; 0) کنیم ، صدا را طوری دریافت می کنیم که انگار محو می شود . این کاملا ایده آل نیست . ما در این بخش شیوه های حل این موضوع را بررسی می کنیم .

مهمترین مشخصه هر نمونه `SoundSource` موقعیت آن است . موقعیت مهمترین فاکتوری است که تعیین می کند که با چه بلندی و از کدام جهت ، منبع پخش شود . ما می توانیم موقعیت یک `SoundSource` را توسط فراخوان `SoundSource::setPosition()` تنظیم کنیم . این خیلی شبیه به کلاس `Listener` است چون که یک بردار سه بعدی می خواهد . در اینجا هم از قرارداد *z = 0 استفاده می کنیم :*

```
sf::Sprite zombie(AssetManager::GetTexture("zombie.png"));
sf::Sound growl(AssetManager::GetSoundBuffer("growl.ogg"));

/*Update zombie's position here*/

//Update sound's position
sf::Vector2f zombiePos = zombie.getPosition();
growl.setPosition(zombiePos.x, zombiePos.y, 0);
```

این مثال هم دقیقا شبیه مثال `Listener` است ، با این تفاوت مهم که اینجا می تواند چندین صدا در محیط باشد ، و بنابراین باید هر موقعیت صدا به صورت جداگانه تنظیم شود . در این مثال صدا به sprite متصل نشده است ، و ما باید مطمئن شویم که هر باری که sprite حرکت می کند ، موقعیت `growl` را به روز کنیم تا باعث شود صدا از مکان درست بیاید .

همچنین می توانیم یک مقدار بولی نصب کنیم که مشخص کند آیا موقعیت ، در نسبت با موقعیت `Listener` است یا خیر . به صورت پیش فرض ، موقعیت هر صدا مطلق است ، اما می توانیم با فراخوان `SoundSource::setRelativeToListener()` باعث شوید در نسبت با `Listener` باشد . این وقتی مفید است که می خواهیم یک صدا را با بلندی صدای کامل پخش کنیم (صدای برداشتن قدم ، خش خش لباس ، شکلیک اسلحه و ...) . برای انجام این کار کافی است صدا را در مبداء (0; 0; 0) قرار داده ، و آن را در نسبت با شنونده قرار دهیم . با این کار ، صدا همیشه بر روی شنونده پخش می شود .

قطع نظر از موقعیت ، هر `SoundSource` یک مشخصه دیگر به نام کوتاه ترین فاصله دارد . این نشان دهنده دورترین مسافت از شنونده است که از آنجا صدا می تواند با بیشترین درجه بلندیش ، شنیده شود . برای مثال اگر صدائی داشته باشیم که 40 واحد از شنونده دور است و دارای کوتاهترین فاصله 45 است ، با بلندی صدای کامل پخش خواهد شد . اما اگر صدا دارای کوتاهترین فاصله پائین تر از 40 باشد ، با بلندی صدای محو شده پخش خواهد شد که بسته به فاکتور تضعیف است (در پاراگراف زیر بحث می شود) . می توانیم کوتاه ترین فاصله هر `SoundSource` را توسط فراخوان `SoundSource::setMinDistance()` تنظیم کنیم . این یک پارامتر `float` می خواهد که نشان دهنده مسافت در مختصات محیط است (مقدار `0` ممنوع است) . مقدار پیش فرض کوتاه ترین فاصله 1 است .

آخرین مشخصه ای که می توانیم برای یک منبع صدا نصب کنیم ، تضعیف است . تضعیف ، فاکتوری است که تعیین می کند هنگامی که صدا دورتر از کوتاهترین فاصله اش از شنونده است ، تا چقدر سریع ، محو شود . برای مثال فاکتور 1 صدا را خیلی آهسته محو می کند ، در حالیکه با فاکتور 100 ، صدا هیچ دوره انتقالی بین بیشترین درجه بلندیش و صفر نخواهد داشت . فاکتور 0 نشان دهنده این است که صدا یا در بیشترین درجه بلندیش توسط شنونده شنیده شده یا اینکه پخش نشده است . تضعیف می تواند توسط فراخوان `SoundSource::setAttenuation()` تنظیم شود . مقدار پیش فرض تضعیف 1 است .

خلاصه خصوصیات صوتی

بعد از این همه صحبت درباره صداهای سه بعدی ، فرصت خوبی است تا با ارائه یک مثال ، همه چیز را مرور کنیم . ما یک محیط ساده با یک شنونده و یک منبع صوتی ایجاد می کنیم که می توانیم با استفاده از ماوس آن را کنترل کنیم (موقعیت توسط مختصات ماوس کنترل می شود ، و صدا با یک کلیک دکمه ماوس پخش می شود) . اجازه دهید با نصب شنونده و صدا همراه با نشان دهنده های تصویری مناسب برای آنها (`CircleShape`) شروع کنیم :

```
sf::RenderWindow window(sf::VideoMode(640, 480), "Audio");
AssetManager manager;

//Listener at the center of the window
sf::Listener::setPosition(window.getSize().x / 2.f, window.getSize().y / 2.f, 0);
//The listener is facing UP (-Y)
sf::Listener::setDirection(0, -1, 0);

//Shape for the listener (world representation)
sf::CircleShape shapeListener(20);
shapeListener.setFillColor(sf::Color::Red);

sf::Sound sound(AssetManager::GetSoundBuffer("mySound.ogg"));
//Sound will start to fade away from the listener when it's
//more than 160 pixels away from the listener ( 640 / 4 = 160)
sound.setMinDistance(window.getSize().x / 4.f);
//Sound will fade quite quickly, once it passes the 160 pixel boundary
sound.setAttenuation(20.f);

//Shape for the sound (world representation)
sf::CircleShape shapeSound(10);
shapeSound.setFillColor(sf::Color::White);
```

چون ما از `AssetManager` برای بارگذاری بافرهای صوتی استفاده می کنیم ، ایجاد یک نمونه از آن درست بعد از ایجاد پنجره مهم است . سپس شنونده را در مرکز پنجره قرار داده و طوری تنظیم می کنیم که به سمت بالا – لبه پنجره (جهت *y*) باشد . همچنین یک شکل ایجاد می کنیم تا در موقعیت مرکز رندرکننده نمایش داده شود . این به ما یک نشانه تصویری می دهد که معلوم می کند شکل دقیقا در کجا ایستاده است . بعد از آن یک صدا تولید کرده و مشخصات کوتاهترین فاصله و تضعیف آن را تنظیم می کنیم . برای تکمیل کار ، یک شکل هم برای صدا ایجاد می کنیم . با انجام این کار می توانیم به حلقه بازی برسیم . اولین چیزی که باید مدیریت شود ، رویدادهاست :

```
//Handle events
sf::Event ev;
while (window.pollEvent(ev))
{
    //Close window on close button click
    if (ev.type == sf::Event::Closed)
        window.close();
    //Play the sound on mouse button click
    else if (ev.type == sf::Event::MouseButtonPressed)
        sound.play();
}
```

ما در هنگام کلیک یکی از دکمه های ماوس ، صدا را پخش می کنیم . این کار به سادگی توسط فراخوان `Sound::play()` انجام می شود . حالا بخش به روز کردن فریم را ببینید :

```
        //Get 2D listener position
        sf::Vector2f listenerPos(sf::Listener::getPosition().x, sf::Listener::getPosition().y);
        //Set listener position (constant for this example)
        shapeListener.setPosition(listenerPos);

        //Set sound position
        sf::Vector2f soundPos(static_cast<sf::Vector2f>(sf::Mouse::getPosition(window)));
        sound.setPosition(soundPos.x, soundPos.y, 0);
        shapeSound.setPosition(soundPos);
```

دو خط اول مسئول وضعیتی هستند که در آن `Listener` موقعیتش را در هر فریم تغییر می دهد . اما در این مثال خاص ، شنونده ساکن است و بنابراین می توانیم آنها را نادیده بگیریم . خط های بعدی صدا را به موقعیت ماوس حرکت می دهند (نسبت به پنجره جاری) . توجه کنید که ما sprite را هم به روز می کنیم .

آخرین کار ، رندر صحنه است . ما فقط دو شکل داریم :

```
        //Render frame
        window.clear();

        window.draw(shapeListener);
        window.draw(shapeSound);

        window.display();
```

با اجرای برنامه باید دو دایره ببینید که یکی از آنها با ماوس حرکت می کند . این موجود صدای ماست . برای پخش صدا ، یکی از دکمه های ماوس را کلیک می کنیم . سعی کنید فاصله های مختلفی بین صدا و شنونده را امتحان کنید تا متوجه شوید محو شدن صدا چطور کار می کند . همچنین تغییر تضعیف و کوتاهترین فاصله ، افکت های متفاوتی تولید می کند . آنقدر کدها را دستکاری و تست کنید تا با مفهوم صدای سه بعدی آشنا شوید .

در اینجا بحث ما درباره صداها و موسیقی به پایان می رسد . متن چیزی است که قصد داریم در فصل بعدی به آن بپردازیم . متون خیلی مهم اند چون ما نیاز به راهی داریم تا آمار آیتم های داخل بازی و اسم کاراکترها را نمایش دهیم .

شروع کار با sf::Text

متن یکی از مهمترین خصوصیات یک بازی است . بله درست است بازی هائی وجود دارد که در آنها از متن هیچ استفاده ای نمی شود آما تعداد اینها خیلی کم است . اگر بازی می سازید ، نیاز دارید در طول فرایند ساختن بازی متن ها را در نقطه ای رندر کنید (حتی برای اطلاعات اشکال زدائی و دیباگ بازی) .در این بخش درباره متن و فونت ها صحبت می کنیم .

رابطه `Texture`—`Sprite` و `SoundBuffer`—`Sound` را یادتان هست ؟ بله درست حدس زدید ، کلاس های `sf::Font` و `sf::Text` هم از همین مدل استفاده می کنند . در این مورد کلاس `Font` یک منبع است و ما باید در زمانی که کلاس `Text` از آن استفاده می کند ، آن را ایمن نگهداری کنیم . احتمالا فرایند بارگذاری فونت ها و ایجاد متون بر اساس آنها را تصور کرده اید ! با ایجاد یک برچسب متنی ساده شروع می کنیم :

```
sf::Font font;
//Try to load a font and exit if there was an error
if (!font.loadFromFile("awesomeFont.ttf"))
    return -1;

sf::Text text("Look at my awesome font.", font);
```

هنگامی که یک نمونه از Text را داشتیم ، می توانیم آن را توسط فراخوان متد RenderWindow::draw() ترسیم کنیم . در واقع ، مثل کلاس های Sprite و Shape ، کلاس Text هم از sf::Drawable و sf::Transformable ارث می برد بنابراین می توانیم به همان شیوه ، موقعیت ، دوران ، کوچک و بزرگ کردن و مبداء آن را تغییر دهیم .

در یک شئ Text می توانیم تعدادی از خصوصیات مخصوص متون را تنظیم کنیم . یکی از مهمترین آنها اندازه کاراکتر است . این مشخصه تعیین می کند که کاراکترها چقدر بزرگ باشند (به پیکسل) و توسط Text::setCharacterSize() تنظیم می شود . ما همچنین می توانیم یک آرگومان سومی را در سازنده مشخص کنیم که اندازه کاراکتر را در هنگام تولید تعیین می کند (مقدار پیش فرض 30 است) .

به علاوه ، می توانیم رشته متنی را که توسط شئ text نمایش داده می شود ، تغییر دهیم . این کار توسط Text::setString() انجام می شود . این یک شئ sf::String نیاز دارد که تبدیلی مجازی از std::string و std::wstring دارد . بنابراین پاس دادن هر سه این آرگومان ها کار خواهد کرد :

```
sf::String someString;

/*Fill the 'someString' variable here*/

text.setString(someString);
text.setString("This is a normal string");
text.setString(L"This is a wide-char string");
text.setString(std::string("This is a normal string"));
text.setString(std::wstring(L"This is a wide-char string"));
```

رنگ شئ Text را می توان توسط فراخوان Text::setColor() تنظیم کرد . همچنین می توان با سبک فونت را مشخص کرد :

Text::Style::Bold , Text::Style::Italic , Text::Style::Regular , Text::Style::Underlined

این کار با متد Text::setStyle() انجام می شود . آرگومان مورد انتظار ترکیبی از اعضای شمارش Text::Style است . مثلا کد زیر یک متن bold (پررنگ) با خطی در زیر آن ایجاد می کند :

```
text.setStyle(sf::Text::Bold | sf::Text::Underlined);
```

یکی از خصوصیات جالب کلاس Text ، توانائی گرفتن موقعیت جهانی یک کاراکتر خاص در متن است . این کار توسط Text::findCharacterPos() انجام می شود . این ، اندیس کاراکتر موجود در رشته متنی را می خواهد و موقعیت آن را بازگشت می دهد . این در جاهائی مفید است که می خواهیم یک گرافیک را روی یک کاراکتر قرار دهیم ، یا به عنوان نشان دهنده موقعیت مکان نما در یک جعبه ورودی متن .

تمام خصوصیاتی که می توان به یک شئ Text اعمال کرد یک ورژن *getter* *به شکل کلی* Text::get*() دارند .

حالا اجازه دهید درباره فونت ها صحبت کنیم . چون آنها منابعی هستند که باید از فایل ها بارگذاری شوند ، تا زمانیکه شئ Text از آنها استفاده می کند ، باید موجود و زنده باقی بمانند . خوشبختانه ما AssetManager را داریم که دقیقا این کار را برای همه منابع ما انجام می دهد .

AssetManager 3.0

اضافه کردن یک ورودی برای Font به کدهای موجود ، زیاد متفاوت از دو منبع دیگر یعنی Texture و SoundBuffer نیست . ما نیاز داریم که ابتدا یک std::map را اضافه کنیم و سپس یک متد که منبع را بارگذاری و نگهداری کند . فایل هدر ما بعد از اضافه کردن map و متد به شکل زیر خواهد بود :

```
class AssetManager
{
public:
    AssetManager();

    static sf::Texture& GetTexture(std::string const& filename);
    static sf::SoundBuffer& GetSoundBuffer(std::string const& filename);
    static sf::Font& GetFont(std::string const& filename);

private:
    std::map<std::string, sf::Texture> m_Textures;
    std::map<std::string, sf::SoundBuffer> m_SoundBuffers;
    std::map<std::string, sf::Font> m_Fonts;

    //AssetManager is a singleton, so only one instance can exist at a time
    //sInstacne holds a static pointer to the single manager instance
    static AssetManager* sInstance;
};
```

پیاده سازی متد AssetManager::GetFont() :

```
sf::Font& AssetManager::GetFont(std::string const& filename)
{
    auto& fontMap = sInstance->m_Fonts;

    auto pairFound = fontMap.find(filename);
    if (pairFound != fontMap.end())
    {
        return pairFound->second;
    }
    else
    {
        //Create an element in the Fonts map
        auto& font = fontMap[filename];
        font.loadFromFile(filename);
        return font;
    }
}
```

فرایندی که این متد دنبال می کند ، خیلی ساده است : اگر از قبل فونتی با نام داده شده موجود باشد ، آن را بازگشت می دهد در غیر این صورت ، یک ورودی جدید داخل map ایجاد کرده و آن را از فایل بارگذاری می کند .

فرمت های فونتی که پشتیبانی می شوند :

TrueType (TTF), Type 1, CFF, OpenType, SFNT, X11 PCF, Windows FNT, BDF, PFR, Type 42

فرمت هائی که اغلب رایج و مورد استفاده است عبارتند از TrueType (TTF) و OpenType (OTF)

فرمت OpenType دقیقا مثل فرمت TrueType ساخته شده اما یک مزیت کوچک دارد و آن اینکه شامل یک مجموعه ای از کاراکترهای بزرگ تر است . توجه کنید که فونت باید در پوشه در دست اقدام محل ذخیره بازی باشد یا اینکه باید مسیر ذخیره آن را به صورت کامل داخل اسم فایل بیاورید . SFML نمی تواند به صورت اتوماتیک فونت های سیستمی را مستقیما بارگذاری کند ، بنابراین باید از فایل های فونت استفاده کرد مثلا فراخوان `font.loadFromFile("Arial")` کار نمی کند چون تابع یک فایل می خواهد نه یک فونت سیستمی .

استفاده از `AssetManager` برای بارگذاری فونت ها ، ساده است :

```
sf::RenderWindow window(sf::VideoMode(640, 480), "Audio");
AssetManager manager;

sf::Text text("Look at my awesome font.", AssetManager::GetFont("awesomeFont.ttf"));
```

یادتان باشد هنگامی که از `AssetManager` استفاده می کنیم ، ابتدا باید از آن نمونه سازی کنیم .

در اینجا ، بحث ما درباره صداها ، متن و فونت ها به پایان می رسد . از این ابزارها لذت ببرید .

خلاصه

در این فصل درباره چیزهای زیادی صحبت کردیم . ما با بررسی مدل صوتی SFML شروع کردیم ، صدا ها و موسیقی را به عنوان منابع صوتی توضیح دادیم . سپس مفهوم صدای سه بعدی و شیوه های پیاده سازی آن با کمک SFML را بررسی کردیم . آخرین بخش این فصل درخصوص متن ها و فونت ها بود . ما همچنین مدیریت کننده منابع خود را به روز کردیم تا بتواند منابع جدید را مدیریت کند .

فصل آخر کتاب موضوعی پیشرفته را معرفی می کند . شیدرها موضوع بحث این فصل خواهد بود .

فصل ٦ : رندر افکت های ویژه با استفاده از شیدرها

به بخش پیشرفته کتاب خوش آمدید . این فصل آخر مفهوم شیدرها را معرفی می کند که فهم آن برای توسعه دهندگان بی تجربه سخت است و ممکن است برای باتجربه ها نیز مشکل ساز باشد . ما توضیح می دهیم که شیدرها چیستند و چگونه در SFML نمایش داده می شوند ؛ اما آنها را همراه باجزئیات و به صورت مفصل توضیح نمی دهیم . آنها به خودی خود موضوعی وسیع هستند که کتاب های زیادی درباره آنها منتشر شده است .

شیدرها ابزاری هستند که برای تولید افکت های گرافیکی جذاب ، انجام anti-aliasing ، بهینه کردن خط لوله گرافیکی و ... استفاده می شوند . آنها در اصل برنامه هائی هستند که بر روی GPU اجرا می شوند و با داده های vertex (راس) و fragment (پیکسل) کار می کنند . ما همچنین در خصوص رندرینگ مستقیم بر روی یک بافت و استفاده از آن بافت برای تولید تصویر نهائی بر روی صفحه ، صحبت خواهیم کرد .

در این فصل ، مطالب زیر پوشش داده می شود :

`sf::RenderTarget` و `sf::RenderWindow`

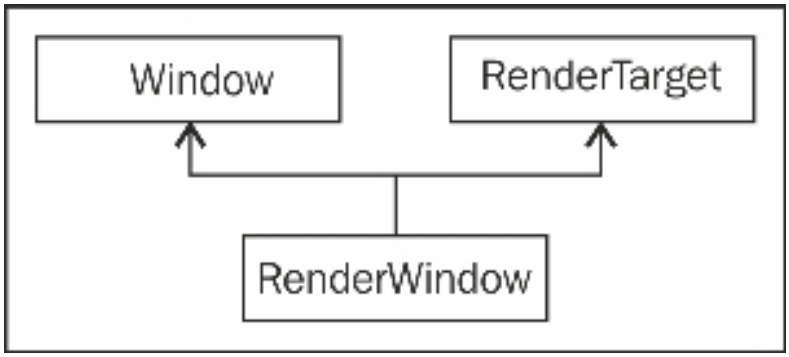
رندرینگ مستقیم به یک بافت

برنامه نویسی شیدر

sf::RenderTarget و sf::RenderWindow

ما از قبل با کلاس `RenderWindow` آشنا هستیم ، اما آن را همراه با جزئیات بررسی نکرده ایم . علت این است که لازم نیست چیز زیادی فراتر از متد `RenderWindow::draw()` برای کار کردن با آن بدانیم . اما حالا که قصد داریم درباره رندرینگ بر روی اشیاء دیگر صحبت کنیم (مثلا بافت) مهم است که بدانیم `RenderWindow` چیست .

ما با مقدمه ای بر مفهوم یک render target شروع می کنیم . render target شئ ای است که می توانیم از مدل گرافیک برای ترسیم بر روی آن استفاده کنیم . یک کلاس برای اینکه به عنوان یک render target مطرح شود ، باید از `sf::RenderTarget` ارث ببرد . این کلاس ، واسط لازم در کلاس های فرزند برای ترسیم اشیاء `Drawable` ، استفاده از `Views` (یا دوربین ها) ، شیدر و ... را فراهم کرده است . کلاس `RenderWindow` یکی از این کلاس های فرزند است . ساختار درختی وراثت کلاس `RenderWindow` شبیه زیر است :



اگر به کلاس پایه Window نگاه کنیم ، می بینیم که هیچ عملکردی برای ترسیم وجود ندارد ، بلکه فقط مشخصات مربوط به پنجره ها مثل مدیریت رویدادها ، تغییر اندازه ، عنوان ، موقعیت و ... هستند . برای ترسیم چیزی در کلاس پایه Window ، باید از OpenGL استفاده کنیم . اما هر کلاسی که متدهای RenderTarget را اجرا کند ، می تواند به عنوان یک پرده نقاشی برای تمام خصوصیات مدل گرافیک ، خدمت کند . این علت مهم بودن RenderTarget است : بدون کارکردی که این کلاس فراهم کرده است ، ما نمی توانیم از بیشتر خصوصیات مدل استفاده کنیم .

بیشتر متدهای RenderWindow که قبلا درباره آنها صحبت کرده ایم ، در واقع در RenderTarget اعلام شده اند :

`draw()`, `setView()`, `clear()`, `mapPixelToCoords()`, ...

بعضی از این متدها بسته به RenderTarget به صورت متفاوتی پیاده سازی شده اند اما بیشتر آنها به یک شکل کار می کنند . قبل از ادامه بحث ، یک نکته مهم درباره متد RenderTarget::draw() وجود دارد . ما درباره متد RenderTarget::draw() در فصل های قبلی صحبت کردیم ، اما هنوز دومین پارامتر آن را (پارامتر اختیاری) که یک نمونه از sf::RenderStates است ، بررسی نکرده ایم . کلاس چهار مقدار نگهداری می کند که می تواند برای هر فراخوان ترسیم ، تعریف شود :

`sf::BlendMode`, `sf::Transform`, `Texture*`, `Shader*`

ما می توانیم قبل از پاس دادن وضعیت رندر به متد RenderTarget::draw() اینها را نصب کنیم اما بعضی اشیاء Drawable اینها را به صورت اتوماتیک نصب کرده اند (مثلا کلاس Sprite تغییر شکلش را با آنچه که داده می شود ترکیب کرده و بافت خودش را نصب می کند) . این کلاس همچنین سازنده های مجازی برای هر یک از فیلدهایش دارد ، بنابراین پاس دادن فقط یکی از این انواع کار می کند بدون اینکه مجبور باشیم شئ RenderState را به صورت صریح تولید کنیم . مثلا کد زیر به خوبی کار می کند :

```
window.draw(sprite, sf::BlendMode::BlendAdd);
```

فهم اساس کلاس RenderTarget به ما اجازه می دهد درباره یک پیاده سازی متفاوت از آن به شکل sf::RenderTexture صحبت کنیم .

رندرینگ مستقیم به یک بافت

ما قبلا بافت ها را توضیح داده ایم . ما حتی یک AssetManager ایجاد کردیم که بافت ها را بارگذاری و نگهداری می کند . اما تنها چیزی که با آنها تاکنون انجام داده ایم ، قرار دادن آنها روی یک sprite یا یک shape بوده است . بافت ها همچنین می توانند به شیوه های متفاوتی در برنامه نویسی گرافیکی مورد استفاده قرار پیرند . یکی از آنها در شیدرهاست جائی که آزادی زیادی برای استفاده از هر یک از پیکسل های بافت ، هر طوری که بخواهیم ، وجود دارد . درباره استفاده از بافت ها در شیدرها ، بعدا در این فصل صحبت خواهیم کرد . حالا یک شیوه متفاوت تولید یک بافت یعنی رندرینگ مستقیم اشیاء Drawable بر روی آن را بررسی می کنیم . در این مورد ، RenderTexture کارکردی را که می خواهیم ، فراهم کرده است .

کلاس RenderTexture از RenderTarget ارث می برد و همه کارکردهای ترسیم آن را با استفاده از شئ **framebuffer** پیاده سازی کرده است . شئ framebuffer یک شئ OpenGL است ، که اجازه رندرینگ به یک بافر خارج از صفحه می دهد . با استفاده از این تکنیک ، SFML قادر خواهد بود رفتاری را به دست آورد که بتوانیم مستقیما بر روی یک بافت ، رندرینگ را انجام دهیم . خود شئ Texture به عنوان یک فیلد درون کلاس RenderTexture نگهداری می شود و می توانیم آن را توسط فراخوان RenderTexture::getTexture() به دست آوریم .

در بسیاری از موارد ، بهتر است رندر روی یک بافت انجام شده و سپس با آن کاری انجام دهیم . مثلا رندر همه صحنه روی یک بافت ، به ما اجازه می دهد افکت های پس پردازش با شیدرها انجام دهیم .

شیوه استفاده از یک بافت رندر شبیه به RenderWindow است . ابتدا باید آن را با ابعاد مناسب (عرض و ارتفاع) ایجاد کنیم و سپس با متدهای معروف ، روی آن ترسیم را انجام دهیم :

```
sf::RenderTexture rTexture;
rTexture.create(32, 32, /*Depth buffer enabled = */ false);

sf::CircleShape circle(16); //Circle radius = 16

//Render routine - clear -> draw -> display
rTexture.clear();

rTexture.draw(circle);

rTexture.display();

//RenderTexture::getTexture() gets a ref to the Texture object
sf::Sprite sprite(rTexture.getTexture());

//Use the sprite in any way we like
```

بعد از اعلام یک متغییر RenderTexture ، باید متد RenderTexture::create() را صدابزنیم تا به صورت واقعی یک بافت رندر معتبر بسازد . متد ، پارامترهای عرض و ارتفاع و یک شاخص می گیرد که آیا بافت رندر از بافر عمق استفاده می کند یا خیر . در این مورد ما فقط یک شکل دایره روی آن ترسیم می کنیم ، بنابراین بافر عمق نیاز نیست . بافر عمق به ما اجازه می دهد هر پیکسلی که قبل از پیکسل های موجود رندر شده را دور بیاندازیم . همچنین اگر مدل های سه بعدی را بر روی بافت رندر می کنیم (با استفاده از

OpenGL) احتمالا می خواهیم بافر عمق فعال باشد . در مثال ما ، شکل دایره ما شعاع 16 دارد پس بافتی با عرض و ارتفاع 32 = شعاع2x می خواهیم .

بعد از اینکه شکل دایره را ایجاد کردیم ، آن را بر روی بافت رندر می کنیم . `RenderTexture::clear()` تمام پیکسل های بافت را به رنگ داده شده تغییر می دهد . آرگومان رنگ اختیاری است چون `Color::Black` رنگ پیش فرض است . سپس با متد `RenderTexture::draw()` شکل را روی بافت ترسیم کرده و در نهایت هر چیزی را که تابه حال ترسیم شده ، نمایش می دهیم . متد `RenderTexture::display()` هر پیکسل از بافت را با هر چیزی که از آخرین بار فراخوانش ، انجام شده است ، به روز می کند .

حالا که بافت ما آماده شده ، می توانیم با گرفتن شئ `Texture` توسط `RenderTexture::getTexture()` از آن استفاده کنیم . این متد یک ارجاع به شئ بافت که به عنوان یک فیلد درون `RenderTexture` ذخیره شده است ، بازگشت می دهد . در این مورد ما از آن برای تولید یک sprite استفاده کردیم که می توانیم از این sprite بعدا هر طوری که بخواهیم ، استفاده کنیم .

شبیه شئ `Texture` ، تا زمانیکه یک شئ از بافت درونی `RenderTexture` استفاده می کند ، `RenderTexture` باید موجود و زنده باقی بماند . در غیر این صورت ، بافت درونی نابود شده و منبع از دست می رود .

صرف نظر از کارکرد ترسیم ، کلاس `RenderTexture` چند متد `Texture` مثل `RenderTexture::setSmooth()` و `RenderTexture::setRepeated()` هم دارد . هم چنین اگر بخواهیم با استفاده از OpenGL شروع به رندر بر روی یک بافت کنیم ، باید `RenderTexture::setActive()` را فرابخوانیم تا زمینه برای فراخوان های OpenGL API فعال شود . در غیر این صورت همه فراخوان های OpenGL روی محیط فعال شده قبلی (در اغلب موارد پنجره اصلی است) تاثیر خواهد گذاشت .

فعلا صحبت در خصوص اشیاء `RenderTexture` کافی است . بعدا هنگامی که فهم خوبی از شیدرها به دست آوردیم ، به اینها باز می گردیم .

برنامه نویسی شیدر

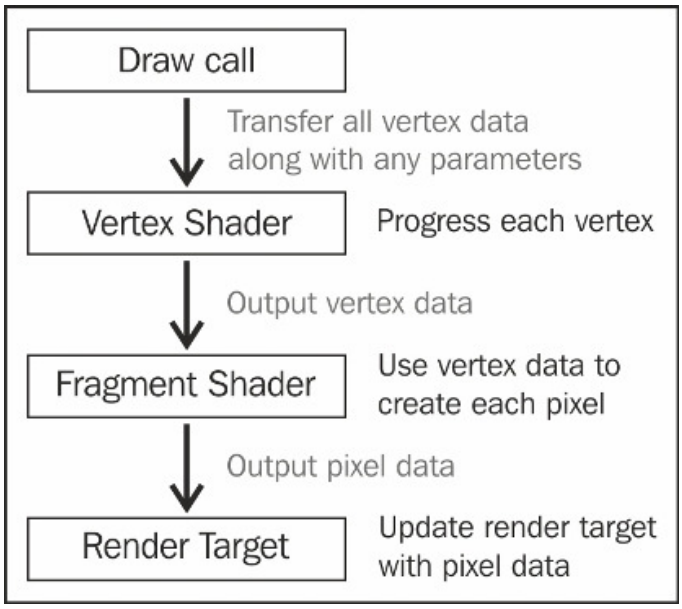
برنامه نویسی شیدر تاریخچه ای طولانی دارد و در انواع مختلف زیادی وجود دارد . این کتاب حتی برای پوشش بخش کوچکی از مباحث موجود در برنامه نویسی شیدر ، کافی نیست . در این بخش به طور مختصر می گوئیم که شیدرها چیستند و SFML درخصوص بارگذاری و استفاده از آنها ، چه پشتیبانی دارد اما وارد جزئیات نمی شویم . اگر در زمینه آموزش بیشتر آنها علاقه مند هستید ، بهتر است کتاب دیگری در زمینه زبان GLSL بگیرید یا یک راهنمای آنلاین را مطالعه کنید . پیشنهاد می کنم آموزش **GLSL 1.2** به آدرس زیر را مطالعه کنید :

http://www.lighthouse3d.com/tutorials/glsl-tutorial

شیدر چیست ؟

شیدرها (Shader) در قالب های متفاوتی می آیند و بسته به API خود ، از زبان های مختلفی استفاده می کنند اما در کل ، شیدر برنامه ای است که روی GPU اجرا می شود و مثل هر برنامه دیگری ، یک شیدر نیاز به یک ورودی دارد و یک خروجی تولید می کند .

ورودی و خروجی شیدر بسته به نوع شیدر است . ما فقط انواع شیدری که توسط کلاس `Shader` موجود در SFML پشتیبانی می شوند و در زمینه sprite های دوبعدی هستند را پوشش می دهیم یعنی شیدرهای Vertex (راس) و Fragment (پیکسل) . با شکل زیر می توانیم خط لوله گرافیکی را از لحظه ای که می خواهیم چیزی را رندر کنیم تا زمان نوشتن واقعی بر روی شئ هدف ، تصور کنیم :



هر بار که متد `RenderTarget::draw()` را فرامی خوانیم ، OpenGL از بین مراحلی که در تصویر قبل شرح داده شد ، عبور می کند . ابتدا ، همه راس ها (موقعیت ، رنگ ، مختصات بافت) را در یک بسته ، بسته بندی کرده و آن را برای ذخیره در حافظه GPU می فرستد . در خصوص یک sprite ، چهار راس برای چهار گوشه مستطیل وجود دارد . بعد از دریافت رئوس در حافظه GPU ، OpenGL برنامه شیدر Vertex (راس) را روی هر یک از این رئوس اجرا می کند . سپس می توانیم هر یک از اینها را به هر شیوه ای که بخواهیم ، دستکاری کنیم . اغلب ، یک شیدر عمومی در جائی استفاده می شود که هر راس را با ماتریس های Model ، View و Projection تغییر شکل می دهد . این ماتریس ها نشان دهنده تبدیلات (موقعیت ، دوران و مقیاس) مدل ، موقعیت و جهت دوربین و تصویر دوربین هستند . بعد از تبدیل هر راس ، OpenGL محاسبه می کند که کدام قطعه باید بر روی هدف ، نقاشی شود . این فرایند را **rasterization** گویند . در مثال sprite با اندازه 16x16 و مقیاس (1, 1) و زاویه دوران صفر دقیقا 16x16 = 256 پیکسل دارد . بعد از اینکه OpenGL همه پیکسل ها (یا قطعات – fragment) را محاسبه کرد ، شیدر پیکسل ما را بر روی هر یک از آنها فرامی خواند . نتیجه این شیدر ، یک رنگ است که می تواند روی render target نسبت به آن پیکسل قرار گیرد . این مقدمه ای به موضوع شیدرها بود . در ادامه درخصوص بارگذاری و استفاده از شیدرها صحبت می کنیم .

بارگذاری شیدرها

اولین مرحله این است که مطمئن شویم GPU شیدرها را پشتیبانی می کند . OpenGL شیدرها را در ورژن 2.0 خودش معرفی کرد ، پس هنوز هم کارت های گرافیکی وجود دارد که آنها را پشتیبانی نمی کند . برای بررسی اینکه آیا شیدرها می توانند روی سیستم استفاده شوند یا خیر ، می توان از تابع `Shader::isAvailable()` استفاده کرد :

```
if (!sf::Shader::isAvailable())
    return -1; //Shaders are not supported. Abort!
```

اولین مرحله استفاده از شیدرها ، بارگذاری و کامپایل آنهاست . SFML بخش کامپایل را به صورت درونی انجام می دهد ، بنابراین ما فقط باید درباره تهیه شیدرها تلاش کنیم . شیدرها می توانند به سه شیوه بارگذاری شوند : از یک فایل ، از یک رشته متنی ، یا از یک استریم . چگونگی تولید یک برنامه شیدر توسط بارگذاری شیدرهای راس و پیکسل :

```
sf::Shader shader;
if (!shader.loadFromFile("vertShader.vert", "fragShader.frag"))
    return -1;

//Use the shader
```

چون SFML از OpenGL استفاده می کند ، شیدرها باید در زبان **OpenGL Shading Language (GLSL)** نوشته شده باشند . `Shader::load*()` هم سعی می کند شیدر را کامپایل کند . اگر مشکلی پیش آید ، یک پیغام خطا در خط فرمان نمایش داده می شود . متد هم `false` بازگشت می دهد .

راه دیگر بارگذاری شیدرها ، ذخیره مستقیم آنها در داخل کد است و سپس بارگذاری آنها به عنوان رشته های متنی :

```
std::string vertShader =
    "void main() {" \
    "gl_Position = gl_Vertex;" \
    "}";
std::string fragShader =
    "void main() {" \
    "gl_FragColor = vec4(1, 0, 0, 1);" \
    "}";

sf::Shader shader;
if (!shader.loadFromMemory(vertShader, fragShader))
    return -1;
```

چون شیدرها باید از یک فایل یا از حافظه بارگذاری شوند (از فایل توصیه می شود) مهم است که آنها را چندین مرتبه بارگذاری نکنیم . ما از این شیوه قبلا در `AssetManager` استفاده کرده ایم .

AssetManager 4.0

ما تا به حال منابع زیادی را به مدیریت کننده منابع خود اضافه کرده ایم . برای شیدرها ، این کار زیاد متفاوت نیست ، به جز اینکه یک برنامه شیدر توسط دو اسم فایل تعریف شده است (*vertex* و *fragment*) *نه یکی (مثلا یک بافت)* .

اول از همه map و متد `AssetManager::GetShader()` را اضافه می کنیم :

```
class AssetManager
{
public:
    AssetManager();

    static sf::Texture& GetTexture(std::string const& filename);
    static sf::SoundBuffer& GetSoundBuffer(std::string const& filename);
    static sf::Font& GetFont(std::string const& filename);
    static sf::Shader* GetShader(
        std::string const& vsFile,
        std::string const& fsFile);

private:
    std::map<std::string, sf::Texture> m_Textures;
    std::map<std::string, sf::SoundBuffer> m_SoundBuffers;
    std::map<std::string, sf::Font> m_Fonts;
    std::map<std::string, std::unique_ptr<sf::Shader>> m_Shaders;

    //AssetManager is a singleton, so only one instance can exist at a time
    //sInstacne holds a static pointer to the single manager instance
    static AssetManager* sInstance;
};
```

توجه کنید که برای `AssetManager::GetShader()` ما یک اشاره گر بازگشت داده ایم نه یک ارجاع . این برای راحتی است چون SFML هنگامی که می خواهیم از آن استفاده کنیم ، به یک اشاره گر نیاز دارد . ما نمونه های `Shader` را به عنوان یک مقدار ذخیره نکرده ایم ، در واقع از یک اشاره گر هوشمند استفاده کرده ایم . علت این کار این است که نمونه های `Shader` نمی توانند کپی شوند (که برای کانتینر map لازم است) .

پیاده سازی متد را ببینید :

```
sf::Shader* AssetManager::GetShader(
    std::string const& vsFile,
    std::string const& fsFile)
{
    auto& shaderMap = sInstance->m_Shaders;

    //The key to be stored in the map
    auto combinedKey = vsFile + ";" + fsFile;
    auto pairFound = shaderMap.find(combinedKey);
    if (pairFound != shaderMap.end())
    {
        return pairFound->second.get();
    }
    else
    {
        //Create an element in the Shader map
        auto& shader = (shaderMap[combinedKey] = std::unique_ptr<sf::Shader>(new sf::Shader()));
        shader->loadFromFile(vsFile, fsFile);
        return shader.get();
    }
}
```

چون ما دو اسم فایل داریم ، باید یک کلید برای map ایجاد کنیم . آسانترین راه حل ، اتصال آنها با یک **;** و استفاده از آن به عنوان کلید است . بقیه کد مثل منابع دیگر است ، با این استثناء که ما به جای ارجاع از اشاره گر – به شیدر استفاده و آن را بازگشت داده ایم .

استفاده از شیدرها

حالا که شیوه ای آسان برای بارگذاری و نگهداری شیدرها داریم ، استفاده از آن آسان است :

```
auto shader = AssetManager::GetShader("vertShader.vert", "fragShader.frag");

sf::Sprite sprite(AssetManager::GetTexture("myTexture.png"));

while (window.isOpen())
{
    window.clear();

    window.draw(sprite, shader);

    window.display();
}
```

ما همچنین می توانیم به همین شیوه ، از شیدر همراه با اشیاء `RenderTexture` استفاده کنیم .

نصب uniform های شیدر

بیشتر شیدرها uniform هائی دارند که به عنوان شیوه کنترل بعضی از مشخصات شیدر عمل می کنند . یک uniform متغییری در داخل شیدر است که می تواند از خارج از شیدر نصب و تنظیم شود . می توانیم توسط `Shader::setParameter()` این uniform ها را نصب کنیم . اسم باید با اسم uniform موجود در داخل شیدر ، یکی باشد . برای مثال ، اگر بخواهیم یک uniform بردار دوبعدی را نصب کنیم ، فراخوان زیر را انجام می دهیم :

```
shader->setParameter("uSpecialVector", sf::Vector2f(3, 3));
```

کلاس shader ، متد `Shader::setParameter()` را برای تعدادی از انواع uniform ، فراهم کرده است . این انواع عباتند از :

`float`, 2D vector, 3D vector, 4D vector, `Color`, `Transform` (matrix), `Texture` (sampler)

فرض کنید می خواهیم یک شیدر ساده موج دار شدن ایجاد کنیم که sprite را در یک دوره زمانی موج دار می کند . به جای رئوس ، ما با مختصات های بافت بازی می کنیم چون ساده تر است و نسبت به استفاده از تعداد بسیار زیادی از راس ها ، کارامدتر است . شیدر راس ما به این شکل خواهد بود :

vertShader.vert

```
#version 110

//varying "out" variables to be used in the fragment shader
varying vec4 vColor;
varying vec2 vTexCoord;

void main() {
    vColor = gl_Color;
    vTexCoord = (gl_TextureMatrix[0] * gl_MultiTexCoord0).xy;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

این یک شیدر عمومی است که کار خاصی انجام نمی دهد ، فقط چند متغییر مختلف را برای شیدر پیکسل نصب می کند . این شیدر از ماتریس Model*View*Projection برای محاسبه موقعیت راس در فضای صفحه استفاده می کند . شیدر پیکسل جائی است که جادو اتفاق می افتد :

rippleShader.frag

```
#version 110

//varying attributes from the vertex shader
varying vec4 vColor;
varying vec2 vTexCoord;

//declare uniforms
uniform sampler2D uTexture;
uniform float uPositionFreq;
uniform float uSpeed;
uniform float uStrength;
uniform float uTime;

void main() {
    vec2 texCoord = vTexCoord;
    float coef = sin(gl_FragCoord.x * uPositionFreq + uSpeed * uTime);
    texCoord.y +=  coef * uStrength;
    gl_FragColor = vColor * texture2D(uTexture, texCoord);
}
```

تعداد زیادی از uniform ها در این شیدر وجود دارد تا ما بتوانیم از خارج ، با تنظیمات متفاوتی از آنها استفاده کنیم . ما یک نمونه بردار uTexture لازم داریم تا رنگ ها را از بافت sprite بگیرد . uPositionFreq تعیین می کند که افکت موج دار شدن برای پیکسل های مجاور چقدر باشد . uStrength اندازه افکت بر روی محور y را کنترل می کند و uSpeed سرعت موج دار شدن را کنترل می کند . uTime زمان سپری شده را نشان می دهد . همه کاری که باید برای تولید افکت موج دار شدن انجام دهیم این است که یک تابع coef محاسبه کنیم که بسته به موقعیت پیکسل جاری و همه uniform های ماست . ما از این مقدار برای محاسبه مختصات بافت نهائی پیکسل جاری استفاده می کنیم . از این مختصات بافت برای پیداکردن رنگ پیکسل uTextue استفاده می شود .

کدی که از این شیدر استفاده می کند به شکل زیر است :

```
auto shader = AssetManager::GetShader("vertShader.vert", "rippleShader.frag");

sf::Sprite sprite(AssetManager::GetTexture("myTexture.png"));

shader->setParameter("uTexture", *sprite.getTexture());
shader->setParameter("uPositionFreq", 0.1f);
shader->setParameter("uSpeed", 20);
shader->setParameter("uStrength", 0.03f);
```

این بخش مقداردهی اولیه است ، جائی که ما sprite و شیدر را ایجاد می کنیم . ما همه uniform ها را با مقادیر مناسب نصب می کنیم (به جز زمان که باید در هر فریم تغییر کند) . حلقه رندرینگ به شکل زیر خواهد بود :

```
sf::Clock clock;
while (window.isOpen())
{
    window.clear();

    shader->setParameter("uTime", clock.getElapsedTime().asSeconds());

    window.draw(sprite, shader);

    window.display();
}
```

توجه کنید که می توانیم قبل از اینکه هر شئ فراخوان draw را انجام دهد ، uniform ها را نصب کنیم . این یعنی با تغییر بعضی از مقادیر پارامترها ، می توانیم یک شیدر برای چندین نوع از اشیاء داشته باشیم .

تاثیر این شیدر ، موج دار شدن روی محور y است که از میان همه sprite عبور می کند :

سعی کنید از بافت های مختلفی استفاده کرده و uniform های مختلف را آزمایش کرده تا افکت های متفاوتی به دست آورید .

sf::Shader و OpenGL

اگر چیزها را مستقیما توسط OpenGL ترسیم کنیم و هنوز هم بخواهیم از کلاس Shader برای بارگذاری و مدیریت شیدرهایمان استفاده کنیم ، می توان بدون مشکل زیادی این کار را انجام داد ، فقط کافیست که شیدر را وصل کنیم . ساختار کدها برای چنین سناریوئی به شکل زیر خواهد بود :

```
//Bind the shader by passing a pointer to the function
sf::Shader::bind(shader);

/* Render objects using OpenGL here */

//Stop using shaders
sf::Shader::bind(nullptr);
```

و این همه بحث ما درباره شیدرها بود . ما فصل را با یک مثال نهائی به پایان می رسانیم که از بافت های رندر و شیدرها برای ایجاد یک افکت پیکسلی شدن استفاده می کند .

مثال آخر

ما درباره کلاس RenderTexture و شیدرها صحبت کردیم . تصادفی نبوده است که این دو کلاس در یک فصل پوشش داده شدند ، علت این است که این دو خیلی خوب با هم کا می کنند . به جای ترسیم مستقیم همه چیز بر روی RenderWindow ، می توانیم صحنه را روی RenderTexture ترسیم کرده و سپس آن بافت را همراه با یک شیدر پس پردازش روی RenderWindow رندر کنیم . این به ما اجازه تولید افکت های متفاوتی را می دهد که ، رسیدن به آن هنگامی که به صورت جداگانه شیدرها را به هر شئ اعمال می کنیم ، سخت تر است . بعضی از افکت های پس پردازش عبارتند از : تک رنگی (اعمال یک رنگ به همه بافت) ، مات کردن ، **Fast Approximate Antialiasing (FXAA)** ، پیکسلی کردن (کاهش تعداد پیکسل ها در حالیکه اندازه شان افزایش می یابد) و ...

در این بخش ما درباره چگونگی نصب RenderTexture صحبت می کنیم و مثال ساده ای از شیدر پس پردازش پیکسلی کردن می آوریم که می تواند به عنوان افکتی برای انتقال از یک صحنه استفاده شود .

نصب RenderTexture

هنگامی که می خواهیم یک افکت پس پردازش تولید کنیم ، در اغلب موارد می خواهیم بافت ما به اندازه صفحه ای که روی آن رندرینگ را انجام می دهیم ، بزرگ باشد تا کیفیت تصویر تولیدی حفظ شود . بنابراین اولین مرحله مقداردهی اولیه پنجره و RenderTexture است :

```
sf::RenderWindow window(sf::VideoMode(800, 600), "Pixelation");
AssetManager m;

if (!sf::Shader::isAavailable())
    return -1; //Shaders are not supported. Abort!

sf::RenderTexture rTexture;
auto wSize = window.getSize();
rTexture.create(wSize.x, wSize.y);

//The sprite used for post-processing the texture
sf::Sprite ppSprite(rTexture.getTexture());
```

بعد از ایجاد پنجره ، `AssetManager` را هم نمونه سازی می کنیم . سپس درخصوص پشتیبانی از شیدر بررسی را انجام می دهیم و اگر این تست منفی باشد ، از برنامه خارج خواهیم شد . سپس `RenderTexture` را به اندازه پنجره ایجاد کرده و درنهایت یک sprite ایجاد می کنیم که برای رندر بافت روی پنجره استفاده می شود .

اولین مرحله تکمیل شده . دو چیز باقی مانده : اشیاء صحنه و شیدر . برای اشیاء صحنه از هرچیزی که دوست داریم می توانیم استفاده کنیم ، چون شیدر با هر چیزی کار می کند البته تا زمانیکه شفاف نباشند (در این صورت هیچ چیز را نخواهیم دید) .

برای پیکسلی کردن ، از شیدر راس مثال قبلی (`vertShader.vert`) استفاده می کنیم ولی یک شیدر پیکسل جدید داریم که در واقع افکت را تولید می کند :

pixelationShader.frag

```
#version 110

//varying attributes from the vertex shader
varying vec4 vColor;
varying vec2 vTexCoord;

//declare uniforms
uniform sampler2D uTexture;
uniform vec2 uTextureSize;
uniform float uPixelSize;

void main() {
    vec2 pixelSizeNorm = uPixelSize / uTextureSize;
    vec2 texCoord = vTexCoord - mod(vTexCoord, pixelSizeNorm);
    gl_FragColor = vColor * texture2D(uTexture, texCoord);
}
```

شیدر سه uniform نیاز دارد : بافت رندر ، اندازه بافت و اندازه پیکسل (در فضای صفحه) . اندازه بافت برای نرمال سازی اندازه پیکسل در محدوده [0...1] نیاز است . بعد از داشتن اینها ، مقدار مختصات بافت را برای پیکسل جاری تا نزدیکترین پیکسل به دست آوریم (بسته به اندازه پیکسل ها) . آخرین خط ، رنگ پیکسل را از بافت با مختصات بافت جدیدی که محاسبه شده ، نصب می کند .

بعد از نوشتن شیدر ، آن را بارگذاری کرده و مقادیری را برای uniform نصب می کنیم :

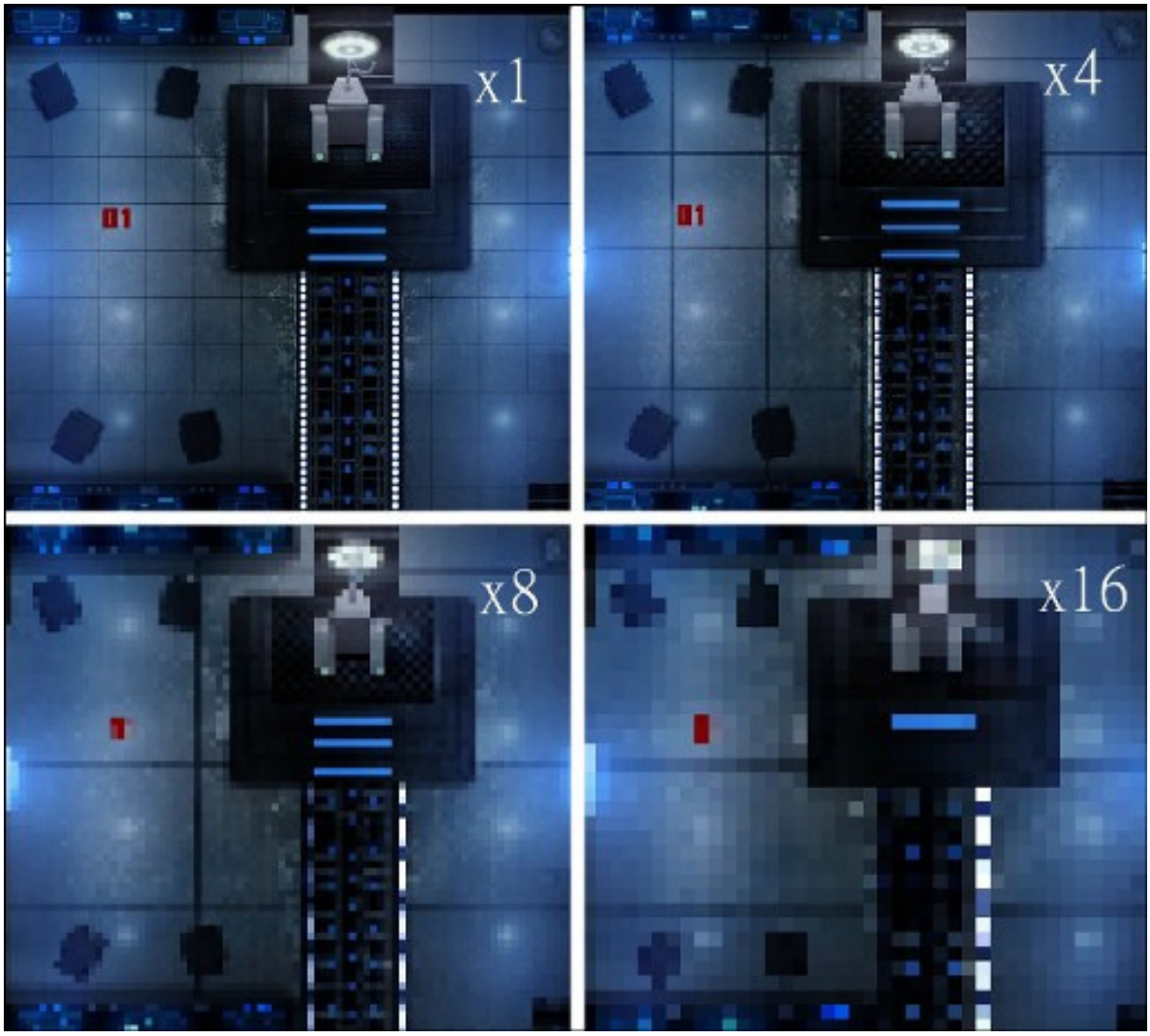
```
//The shader used for post-processing the texture
auto shader = AssetManager::GetShader("vertShader.vert", "pixelationShader.frag");

shader->setParameter("uTexture", rTexture.getTexture());
shader->setParameter("uTextureSize", static_cast<sf::Vector2f>(rTexture.getSize()));
shader->setParameter("uPixelSize", 8);

sf::Sprite sprite(AssetManager::GetTexture("myTexture.png"));
```

uniform های بافت و اندازه بافت فقط یک بار باید نصب شوند (اگر شیدر را همراه با یک بافت استفاده می کنیم) اما اندازه پیکسل می تواند هر تعدادی که دوست داریم ، تغییر کند . این محدود به یک عدد صحیح نیست ، هر مقدار مثبت `float` هم کار می کند .
برای رسیدن به یک افکت انتقال ، می توانیم آن را فریم به فریم به تدریج افزایش دهیم ، سپس صحنه جدید را بارگذاری کرده و شروع به کاهش آن کنیم .

حلقه بازی را ببینید :

```
while (window.isOpen())
{
    //Handle events
    sf::Event ev;
    while (window.pollEvent(ev)) {}

    /* Update frame here */

    //Render frame
    rTexture.clear();
    {
        /* Draw scene here */
    }
    rTexture.display();

    window.clear();
    {
        //Post processing by applying the shader to the RenderTexture
        window.draw(ppSprite, shader);
    }
    window.display();
}
```

مثل همیشه ما با مدیریت رویدادها و به روز کردن فریم شروع کرده ایم . نکته مهم در آخر حلقه است جائی که ابتدا همه صحنه را به `RenderTexture` رندر کرده و سپس sprite را با استفاده از آن بافت همراه با شیدر پس پردازش ، روی پنجره رندر می کنیم . این تمام کاری بود که باید انجام می دادیم . نتیجه این صحنه را همراه با اندازه های مختلف پیکسل ببینید :



و در ایجا سفر ما درخصوص شیدرها به پایان می رسد .

خلاصه

شیدرها و بافت های رندر ، هنگامی که به برنامه نویسی گرافیکی می آیند ، ابزارهای خیلی مفیدی خواهند بود . در این فصل درباره پیاده سازی آنها در SFML صحبت کردیم و اینکه چگونه می توانند بارگذاری شوند و برای تولید افکت های جذاب مورد استفاده قرار گیرند . در خصوص شیدرها ، چیزهای خیلی بیشتری وجود دارد چون امروزه کارت های گرافیک قدرت خیلی زیادی دارند که می توانند برای تولید صحنه های شگفت آور مورد استفاده قرار گیرند .

در اینجا سفر آغازین خود در میان سرزمین SFML را به پایان می رسانیم ، هنوز چیزهای زیادی برای سیاحت وجود دارد ، که با مطالعه کتاب های دیگر می توانید دانش و مهارت های خود را تکمیل کنید .

بازی ساز ©

مسیر ورود به دنیای مهیج بازی سازی

- برنامه نویسی بازی های کامپیوتری
- طراحی و کدنویسی بازی های مخصوص موبایل
- آموزش های کاربردی در زمینه استفاده از ابزارها و کتابخانه های بازی سازی
- معرفی و آموزش شیوه استفاده از معروفترین موتورهای بازی سازی دنیا
- برنامه نویسی به زبان های C# و C++
- و مطالب کاربردی دیگر ...

www.bazi-dev.ir ®