LINKÖPINGS UNIVERSITET

TDP005 Projekt: Objektorienterat system

Introduktion till SFML

Författare

Filip Strömbäck

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2016
Version 1.0

2016-10-04

Introduktion

SFML är ett bibliotek för att enkelt hantera diverse in- och utmatning utöver det som standardbiblioteket tillhandahåller. SFML innehåller moduler för fönsterhantering, grafik, ljud och nätverk. Fönsterhanteringsmodulen hanterar de fönster på skärmen som kan ta emot indata från användaren och där programmet kan rita saker med hjälp av grafikmodulen. Ljud- och nätverksmodulen kan också användas för att spela upp ljud och musik, respektive nätverkskommunikation. I denna introduktion kommer vi huvudsakligen titta på grafikmodulen och de delar av fönsterhanteringen som krävs för att hantera mus- och tangentbordsinmatning samt grafik.

Dokumentationen för grafikmodulen finns på `http://www.sfml-dev.org/documentation/2.4.0/group__graphics.php`. Labben kräver att ni har dokumentationen tillgänglig, eftersom vi inte kommer förklara alla klasser och funktioner ingående här. Vi visar snarare vad som finns och vad som är intressant att kika på.

För att installera SFML på Linux Mint, kör `sudo apt-get install libsfml-dev`. På IDA:s system finns SFML installerat.

1 Skapa fönster

För att komma igång med SFML börjar vi med ett enkelt program som skapar ett fönster som visas på skärmen. I och med detta ser vi till att vi kan kompilera program som använder SFML korrekt, och introducerar ett par grundläggande klasser och koncept i SFML.

Börja med att skapa en källkodsfil med en `main`-funktion i som vanligt. I `main` vill vi skapa en instans av klassen `sf::RenderWindow`. Använd konstruktorn som tar en `sf::VideoMode` och en `sf::String` som parametrar, se dokumentationen här: `http://www.sfml-dev.org/documentation/2.4.0/classsf_1_1RenderWindow.php`

Om vi inte gör något mer så kommer programmet först skapa ett fönster och sedan direkt avsluta, vilket förstör fönstret igen. Om vi inte gör något annat kommer vi alltså inte se något eftersom fönstret förstörs så snabbt. För att kunna se något lägger vi in ett anrop till `sf::sleep` för att vänta i ett par sekunder.

I slutändan bör filen innehålla följande kod:

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

int main() {
    sf::RenderWindow window{sf::VideoMode(1024, 768), "Hello World!"};

    // Vänta i 5 sekunder, så att vi hinner se fönstret.
    sf::sleep(sf::seconds(5));

    return 0;
}
```

För kunna kompilera programmet måste vi se till att länka med SFML. För att göra det, lägg till flaggorna `-lsfml-graphics -lsfml-window -lsfml-system`. Det länkar med modulerna *system*, *window* och *system*. I detta dokument klarar vi oss med dessa tre. Se till att lägga länkningsflaggorna efter alla källkodsfiler, annars får man länkningsfel. Alltså: (\ innebär att kommandot fortsätter på nästa rad)

```
g++ -g -Wall -Wextra -std=c++11 sfml.cpp \
    -lsfml-graphics -lsfml-window -lsfml-system -o sfml
```

Eftersom det blir ett ganska långt kommando gör man med fördel en makefil som man kan kompilera med! Om ni vill kompilera med CLion eller CMake, lägg till följande efter `set(SOURCE_FILES ...)`-raden i filen `CMakeLists.txt`:

```
set(SOURCE_FILES ...)

# På skolans Linux-system finns det två versioner av SFML. Se efter vilken som används:
if (DEFINED ENV{LOADEDMODULES} AND ("$ENV{LOADEDMODULES}" MATCHES ".*prog/gcc/6.1.0.*"))
    set(CMAKE_CXX_COMPILER /sw/gcc-6.1.0/bin/g++)

    if (DEFINED ENV{SFML_ROOT})
        list(APPEND CMAKE_MODULE_PATH "$ENV{SFML_ROOT}/share/SFML/cmake/Modules")
    endif()
endif()

# Välj vilka moduler i SFML som ska användas.
set(SFML_MODULES network graphics window system)

# Försök att hitta SFML genom att anropa 'FindSFML.cmake' (om det finns).
find_package(SFML 2 COMPONENTS ${SFML_MODULES})
include_directories(${SFML_INCLUDE_DIR})

# Om det inte finns hoppas vi på att SFML finns installerat ändå.
if (NOT (${SFML_FOUND} STREQUAL "TRUE"))
    set(SFML_LIBRARIES, "")
    foreach(i ${SFML_MODULES})
        list(APPEND SFML_LIBRARIES "sfml-${i}")
    endforeach(i)
endif()
```

Sedan lägger ni till följande efter `add_executable`-raden. Byt ut `EXECUTABLE` mot namnet som står som första parameter till `add_executable` sedan tidigare.

```
add_executable(EXECUTABLE ${SOURCE_FILES})
target_link_libraries(EXECUTABLE ${SFML_LIBRARIES} ${SFML_DEPENDENCIES})
```

2 Huvudloop

Varje fönster i SFML har en kö av *händelser* (eng. *event*). Varje gång användaren interagerar med vårt fönster på något sätt (exempelvis trycker på en tangent, flyttar på musen etc.) så lägger SFML till en händelse i fönstrets kö. För att hantera dessa händelser så används lämpligen en så kallad händelseloop (eng. *eventloop*). Händelseloopen har till uppgift att regelbundet kontrollera om det finns några händelser i kön och hantera dem. När alla händelser är hanterade så kan händelseloopen fortsätta att hantera det som ska hända kontinuerligt (exempelvis animationer, vi kikar mer på det senare). Händelseloopen implementeras ofta så här:

```
while (!closed) {
  Event event;
  while (pollEvent(event)) {
    processEvent(event)
  }
  update(...)
}
```

Funktionen `pollEvent` finns i SFML och försöker hämta en händelse från händelsekön och spara den i `event`. Om händelsekön är tom så returnerar den `false`, annars returnerar den `true`. Se `http://www.sfml-dev.org/documentation/2.4.0/classsf_1_1Window.php`.

`processEvent` är en funktion som vi implementerar. Den har till uppgift att undersöka `event` och att agera på det om det behövs. Se `http://www.sfml-dev.org/documentation/2.4.0/classsf_1_1Event.php` för att se vilka typer av händelser som finns i SFML.

`update` är också en funktion som vi implementerar. Här kommer vi lägga logik för att rita om skärmen, uppdatera animationer och liknande. För att göra detta behöver vi tillgång till exempelvis vårt `RenderWindow`, så lämpliga parametrar behöver läggas till.

Med hjälp av detta kan vi nu modifiera vårt program så att det väntar på att användaren stänger fönstret innan det avslutas. Kika på händelsen `sf::Event::Closed`.

3 Rita till skärmen

Hittills har vi bara skapat ett fönster. Eftersom vi inte har ritat något i det så ser det ut som om det är transparent (i alla fall på Linux Mint). Därför ska vi nu kika på hur man ritar till ett fönster i SFML.

SFML fungerar likt hur många andra grafikbibliotek fungerar. När ett fönster skapas så skapar SFML två buffrar i minnet, en som innehåller den bild som visas på skärmen för närvarande (*front buffer*) och en som innehåller den bild som vi håller på att rita till för närvarande (*back buffer*), och kopieras till den buffern som visas på skärmen när vi är klara med att rita. Detta kallas för dubbelbuffring och används för att elliminera flimmer. Skulle man rita direkt till skärmen så skulle man ibland se delvis färdigritade bilder eftersom skärmen ritas om lite när den vill, och generellt inte synkroniserad med ditt program.

I ett spel vill vi typiskt rita om skärmen så ofta vi kan (i alla fall nästan), så att rita om skärmen är något vi vill göra i `update`-steget i huvudloopen ovan. Att rita till fönstret följer följande procedur:

- `clear` - fyll buffern vi ritar till med en solid färg (med svart som standard).
- rita med `draw`
- `display` - kopiera det vi ritade till skärmen.

Kika på `http://www.sfml-dev.org/documentation/2.4.0/classsf_1_1RenderTarget.php` för att se den relevanta dokumentationen.

När vi gjort detta bör vårt fönster bli svart i stället för att innehålla gammalt skräp från skärmen.

4 Grafik

Nu när vi har fått igång strukturen för att rita till skärmen är det dags att rita något mer intressant än en tom skärm. I SFML gör man detta genom att skapa en figur (eng. *shapes*) och sedan rita ut dem. En figur innehåller information om vad som ska ritas, såsom vilken kontur som ska ritas, var på skärmen den ska ritas, och vilken färg och textur den ska ha. I SFML finns det färdiga implementationer av cirklar, rektanglar och text, men det går också att skapa egna om man vill. Kika på `http://www.sfml-dev.org/documentation/2.4.0/group__graphics.php` för att se detaljer.

I allmänhet är det bra att skapa de figurer som ska användas en gång och sedan återanvända dem så länge som möjligt. Skapa alltså inte alla figurer varje gång de ska ritas ut, eftersom det ofta kostar prestanda att skapa dem. När en figur har skapats kan man flytta på dem med hjälp av `setPosition`. Det är också värt att titta på `setOrigin` för att bestämma var i figuren origo ska vara, vilket kan göra det smidigare att använda `setPosition` senare.

För att i slutändan rita upp en figur används medlemmen `draw` på det fönster man vill rita i.

5 Inmatning

Nu när vi kan rita objekt på skärmen är det dags att titta närmare på hur man kan sköta inmatning från tangentbordet för att exempelvis styra den figur vi just ritat upp.

Som tidigare nämnts kommer indata från användaren som händelser. Dessa hanterar vi i motsvarigheten till `processEvent` i vår kod. Ett problem som ofta uppkommer i spel är man ofta är intresserad av om en tangent på tangentbordet är nedtryckt snarare än när tangenterna trycks ner och släpps upp. Därför brukar det vara smidigt att göra på följande vis:

```
bool upPressed = false;
while (!closed) {
  Event event;
  while (pollEvent(event)) {
    if (/* upp-tangenten trycktes ner */) {
      upPressed = true;
    } else if (/* upp-tangenten släpptes */) {
      upPressed = false;
    }
  }

  if (upPressed)
    // flytta figuren ett par pixlar uppåt

  // Rita upp allt.
}
```

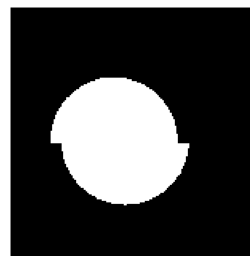
Kika på händelserna `sf::Event::KeyPressed`, `sf::Event::KeyReleased` och `sf::Keyboard` på `http://www.sfml-dev.org/documentation/2.4.0/classsf_1_1Event.php` för detaljer. Låt sedan användaren styra figuren på skärmen med tangentbordet.

6 Animation

Det som kan hända om man använder ovanstående i lösning i ett större program är att figuren kan röra sig i ojämn hastighet. Det som händer är att varje iteration i den yttre loopen tar olika tid på grund av att olika mycket arbete behöver utföras i olika iterationer. I och med detta så tar det olika lång tid mellan att figuren flyttas och därmed kan hastigheten variera.

För att lösa detta finns två saker man kan göra: dels att använda vertical synchronization eller *vsync*, och dels kan man mäta hur lång tid som har passerat mellan varje bildruta som har ritats upp.

Vsync innebär att anropet till `display` väntar på att skärmen ritas om. Detta sker ungefär 60 gånger per sekund. Att sätta på vsync innebär alltså att huvudloopen blir begränsad till 60 iterationer per sekund, alltså ca 16 millisekunder per iteration oavsett hur snabbt koden i loopen körs. Om det ibland tar mer än 16 millisekunder att rita en bildruta så kommer såklart animationer bli ojämna, men det är ofta enklare att hålla exekeveringstiden under 16 millisekunder än att hålla exekeveringstiden jämn. Att använda vsync har ytterligare en fördel: det förhindrar så kallad *tearing*. Om man inte använder vsync så kan kopieringen av den nyligen ritade bildrutan till skärmen ske under tiden som skärmen ritas om. Eftersom detta sker uppifrån och nedåt så kan det bli så att övre delen av skärmen visar den gamla bildrutan medan nedre halvan har hunnit få nästa nya bildruta. Detta syns ibland som ett horisontellt streckpå skärmen, speciellt när objekt rör sig horisontellt på skärmen. För att aktivera vsync i SFML, anropa medlemsfunktionen `setVerticalSyncEnabled` på `Window`-objektet.



Tearing av en cirkel som rör sig åt höger.

Utöver vsync, kan vi låta animationernas bero på den tid som förflutit i stället för att bero på antalet bildrutor som har ritats upp. För att åstakomma detta kan man använda `sf::Timer`, se `http://www.sfml-dev.org/documentation/2.4.0/classsf_1_1Clock.php`. Titta specifikt på `restart`-funktionen. För att låta animationen bero på förfluten tid, kan man göra följande:

```
sf::Clock clock;
float speed = 250.0f;
float x = 0;

while (!closed) {
  while (event = pollEvent()) {
    // ...
  }

  sf::Time delta = clock.restart();
  x += delta.milliseconds() * speed / 1000.0f;
  // rita ut något på position x som tidigare.
}
```

Här kan vi se att `x` ökar med `speed` pixlar per sekund i stället för ett fix antal pixlar per bildruta. Detta gör också att animationerna sker med samma hastighet på olika snabba datorer.

Implementera detta i ditt exempelprogram så att figuren som styrs med tangentbordet rör sig med en fix hastighet oberoende av antalet bildrutor per sekund.

7 Ytterligare läsning

Med SFML är det enkelt att ladda bildfiler från fil och visa dem på skärmen med hjälp av `sf::Texture` (`http://www.sfml-dev.org/documentation/2.4.0/classsf_1_1Texture.php`). Denna textur kan appliceras på figurer med hjälp av `setTexture`. Tänk bara på att se till att inte skapa fler än ett `Texture`-objekt för en textur. Föredra alltså att ladda en textur en gång och låt alla figurer som ska använda texturen dela på samma textur.