LINKÖPINGS UNIVERSITET

TDP005 Projekt: Objektorienterat system

Introduktion till SFML

Författare

Filip Strömbäck; Dag Jönsson

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2026
Version 1.302

30 oktober 2026

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 `https://www.sfml-dev.org/documentation/3.0.2/group__graphics.html`. 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 Ubuntu baserade distros, kör `sudo apt-get install libsfml-dev`.

På IDA:s system finns SFML3 tillgängligt via moduler. Ni kan ladda in modulen ni behöver genom att köra:

```
module init-add prog/sfml/3.0.2 # Inkludera automatiskt vid nästa inloggning
module add prog/sfml/3.0.2 # Inkludera NU!
```

Notera att IDA:s system har SFML 3.0.2 och 3.1.0 tillgängligt. Era system kan eventuellt installera senare versioner. Se till att ni minst har version 3.0.2, och högst 3.1.0. Undvik att använda funktioner som introducerades i senare versioner.

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/3.0.2/classsf_1_1RenderWindow.html`

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>
#include <SFML/Window.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 *graphics*, *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++ -std=c++17 -g -Wall -Wextra sfml.cpp \
    -lsfml-graphics -lsfml-window -lsfml-system -o sfml
```

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 (window.isOpen()) {
  while (auto e{window.pollEvent()}) {
    sf::Event event{e.value()};

    if (/* Är det en sf::Event::Closed? */) {
      window.close();
    }
  }
  // ... resterande kod ...
}
```

Medlemsfunktionen `pollEvent` som finns som i `sf::Window` kommer returnera en Event[1] om det finns något att hantera. Ni kan hämta ut det via `e.value()` och arbeta vidare med det genom att spara det i en ny sf::Event variabel. Om kön var tom kommer villkoret i while-loopen att utvärderas till `false`.

Se `http://www.sfml-dev.org/documentation/3.0.2/classsf_1_1Event.html` för att se vilka typer av händelser som finns i SFML, och hur man avgör vilken typ av Event som har skett.

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

Se till att vi kan stänga fönstret genom att trycka på krysset i hörnet av fönstret.

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, eventuellt är det svart. 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

[1] Sanning med modifikation, egentligen är det en `std::optional`, men vi kan ignorera det med given kod. Om ni vill fördjupa er lite i det kan ni börja här: `https://en.cppreference.com/cpp/utility/optional`.

buffern som visas på skärmen när vi är klara med att rita. Detta kallas för dubbelbuffring och används för att eliminera 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 detta bör vara en del av huvudloopen, och sker efter att vi har hanterat händelser. 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/3.0.2/classsf_1_1RenderTarget.html` för att se den relevanta dokumentationen.

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

Se till att få fönstret att fyllas med en annan färg än svart.

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. *shape*) och sedan rita ut den. 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/3.0.2/group__graphics.html` 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 rita upp en figur används medlemmen `draw` på det fönster man vill rita i.

Se till att skapa en figur och rita ut det i ert fönster.

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. Ett problem som ofta uppkommer i spel är att man 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:

```
while (window.isOpen()) {
  // ... händelsehanteringen ...

  if (/* Kolla om högerpil eller 'D' är nertryckt just nu */) {
    /* Flytta er figur till höger */
  }

  if (/* Kolla om nedpil eller 'S' är nertryckt just nu */) {
```

```
      /* Flytta er figur neråt */
    }
    /* Resterande riktningar */

    // ... kod för att rita er figur ...
  }
```

För att klura ut hur man läser av tangentbordet kan ni kika på dokumentationen för `sf::Keyboard` här `https://www.sfml-dev.org/documentation/3.0.2/namespacesf_1_1Keyboard.html`

Man kan såklart titta efter händelser när tangenter trycks ner och släpps, kika på händelserna `sf::Event::KeyPressed` och `sf::Event::KeyReleased` på `http://www.sfml-dev.org/documentation/3.0.2/classsf_1_1Event.html` för detaljer.

Båda sätt kommer vara intressanta för er, så det är bra om ni övar på båda.

Se till att användaren kan styra figuren på skärmen med tangentbordet.

6 Animation

Det som kan hända om man använder ovanstående 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.[2] Att sätta på vsync innebär alltså att huvudloopen blir begränsad till 60 iterationer per sekund, 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 exekveringstiden under 16 millisekunder än att hålla exekveringstiden 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 åstadkomma detta kan man använda `sf::Clock`, se `http://www.sfml-dev.org/documentation/3.0.2/classsf_1_1Clock.html`. Titta specifikt på `restart`-funktionen. För att låta animationen bero på förfluten tid, kan man göra följande:

```
sf::Clock clock{};
while (window.isOpen()) {
  // ... inputhanteringen ...

  float const speed{250.0f};
  float x{0};
```

[2] En skärm uppdateras vanligtvis 60 gånger per sekund, men frekvensen kan vara högre, särskilt på nyare skärmar.

```
  sf::Time const delta{clock.restart()};

  if (/* Kolla om högerpil eller 'D' trycks ner just nu */) {
    x += delta.asSeconds() * speed;
  }
  if (/* Kolla om nedpil eller 'S' trycks ner just nu */) {
    /* Flytta er figur neråt, beroende av delta */
  }
  /* Resterande riktningar, samt att faktiskt flytta er figur */

  // ... kod för att rita er figur ...
}
```

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 ert 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/3.0.2/classsf_1_1Texture.html`). 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.

8 Smidigare hantering av händelser

Som ni kanske har insett så är sättet vi har visat tidigare för att hantera händelser lite smått klumpigt. Det finns ett lite smidigare sätt, som vi skulle förespråka att ni implementerar framöver.

I `sf::Window` finns det en `handleEvents()` medlemsfunktion, som kan ta emot godtyckligt antal lambda uttryck (eller fria funktioner), där lambda uttrycket tar emot en parameter av den typen av `sf::Event` som lambdan hanterar.

```
while (window.isOpen()) {
  auto const onClose = [&w = window](sf::Event::Closed const &) {
    w.close();
  };

  auto const onKeyPressed =
      [&w = window](sf::Event::KeyPressed const &keyPressed) {
        if (keyPressed.scancode == sf::Keyboard::Scancode::Escape) {
          w.close();
        }
      };

  window.handleEvents(onClose, onKeyPressed);
  // ... resterande kod ...
}
```

Värt att notera är att lambda uttrycken fångar vårt `window` med hjälp av en *init-capture* (se `https://en.cppreference.com/cpp/language/lambda` för djupare förklaring), vilket kommer bli intressant senare när ni börjar strukturera upp ert projekt i klasser.

Dokumentationen för `sf::Window::handleEvents` finns här: `https://www.sfml-dev.org/documentation/3.0.2/classsf_1_1WindowBase.html#a0f3289445ce7103e85c56571c4bc0de9`

9 Slutmål

Vid detta läge bör ni ha en figur som ni kan flytta runt i ert fönster. Er `main` funktion bör vid detta läge börja bli ganska lång, ni har introducerat ett par variabler som behövs i olika steg i huvudloopen, allt detta gör det eventuellt lite knepigt att få en bra översikt över vad som händer; framför allt bör ni också se problem med att utöka detta exempel till något större. För att råda bot på detta ska ni nu bryta ut allt till en egen klass.

Gör detta enligt konstens alla regler, skapa en klass som håller i själva fönstret, och andra variabler, kalla den `Game`, skapa en header och en implementationsfil. Er klass bör bli likt nedan exempel. Ni kommer behöver modifiera exemplet så att det passar er lösning. Uppdatera `main` så att den endast skapar en instans av `Game` och anropar `run()`.

Slutmålet för denna lab är alltså att ni ska ha en figur som går att flytta runt i fönstret. Ni ska också ha en övergripande klass som har koll på spelets tillstånd som skapas i `main`-funktionen.

```
#include <SFML/Graphics.hpp>
class Game {
public:
  Game();

  /* Run the main game loop */
  void run();

private:
  sf::RenderWindow window{};
  // Ni kommer behöva bygga en State-maskin.
  // Ni kan skjuta upp det för denna labb men håll det i åtanke.
  State *state{};

  /* Övriga datamedlemmar */

  /* Handle events */
  void handleEvents();

  /* Updates the games state */
  void update();

  /* Draw the updated state */
  void draw();
};
```