Week 2

Game Loop Programming

Instructor: Hooman Salamat

2019-01-14

Objectives

- Explore an example game loop
- Review and apply the concepts of frames, frame rates and fixed time steps
- Create and display sprites to the game window

Pragma

- The pragma directive is used to access compiler-specific preprocessor extensions. A common use of #pragma is the #pragma once directive, which asks the compiler to include a header file only a single time, no matter how many times it has been imported:
- Syntax: #pragma once

In this example, using #pragma once is equivalent to

```
#ifndef _FILE_NAME_H_
#define _FILE_NAME_H_
/* code */
#endif // #ifndef _FILE_NAME_H_
```

Pragma comment

- #pragma comment is a compiler directive which indicates Visual C++ to leave a comment in the generated object file. The comment can then be read by the linker when it processes object files.
- #pragma comment(lib, libname) tells the linker to add the 'libname' library to the list of library dependencies, as if you had added it in the project properties at Linker->Input->Additional dependencies
- #pragma comment(lib,"opengl32.lib")

- #pragma comment(lib,"glu32.lib")

Using Pragma to link SFML lib files

```
#ifdef _DEBUG
#pragma comment(lib,"sfml-system-d.lib")
#pragma comment(lib,"sfml-window-d.lib")
#pragma comment(lib,"sfml-graphics-d.lib")
#pragma comment(lib,"sfml-audio-d.lib")
#pragma comment(lib,"sfml-network-d.lib")
#else
#pragma comment(lib,"sfml-system.lib")
#pragma comment(lib,"sfml-window.lib")
#pragma comment(lib,"sfml-graphics.lib")
#pragma comment(lib,"sfml-audio.lib")
#pragma comment(lib,"sfml-network.lib")
#endif
```

Friend function

- A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class.

```
class Box {
double width;

public:
double length;
friend void printWidth(Box box);
void setWidth(double wid);
};
```

Default Constructor

- Class myClass {}
- the compiler provides you a default zero argument constructor, along with a destructor, a copy constructor, and a copy assignment operator.

Implicit Conversion

```
#include <iostream>
using namespace std;
class complexNumbers {
double real, img;
public:
//default constructor
complexNumbers() : real(0), img(0) { }

//copy constructor
complexNumbers(const complexNumbers& c) { real = c.real; img = c.img; }

//implicit conversion
complexNumbers(double r, double i = 0.0) { real = r; img = i; }


//friend function
friend void display(complexNumbers cx);
};
void display(complexNumbers cx) {
cout << "Real Part: " << cx.real << " Imag Part: " << cx.img << endl;
}
int main() {
complexNumbers one(1);
complexNumbers five = 5;  //copy assignment operator
display(one);
display(five);
display(300);  //⍰ Implicit conversion
return 0;
}
```

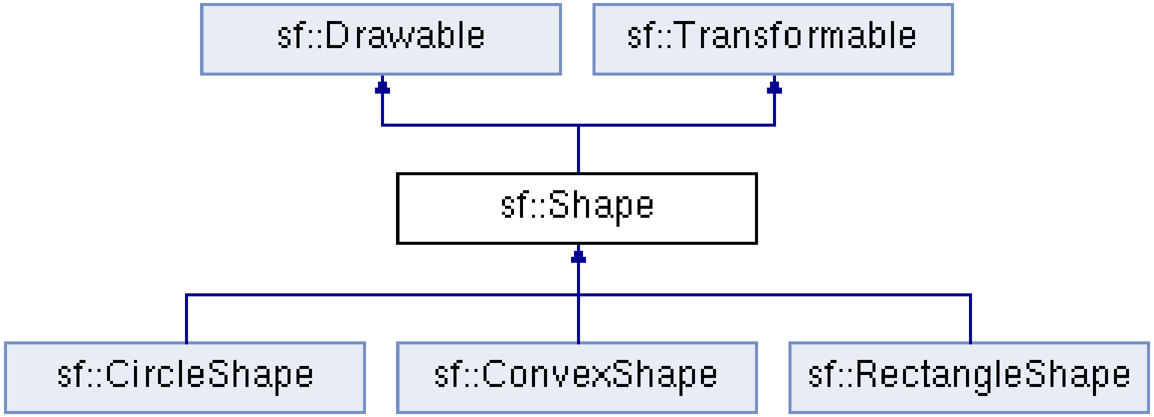
Explicit constructor

- The explicit function specifier controls unwanted implicit type conversions. It can only be used in declarations of constructors within a class declaration.

```
explicit complexNumbers2(double r, double i = 0.0)
{ real = r; img = i; }
complexNumbers2 one(1);
//complexNumbers2 five = 5; //not allowed using explicit
display2(one);
//display2(300);  // Implicit conversion not allowed
```

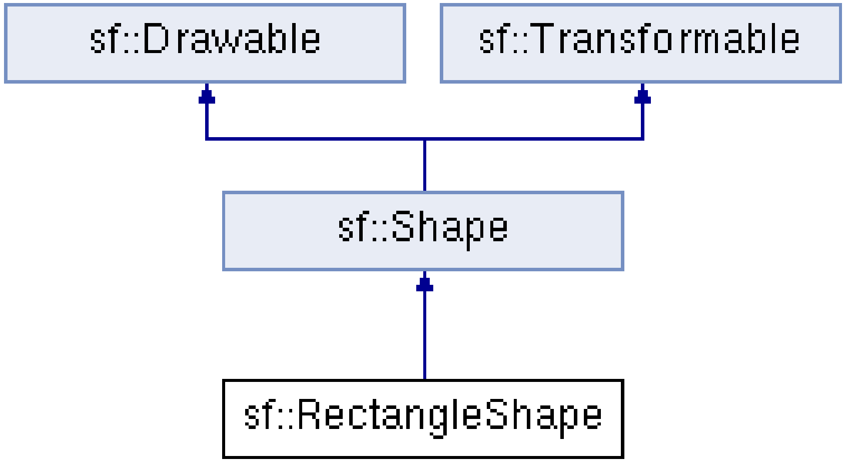
Sf::Shape

- Base class (abstract) for textured shapes with outline



Sf::RectangleShape

- RectangleShape (const Vector2f &size=Vector2f(0, 0))



Sf::ConvexShape

- Creates a convex polygon.

```
sf::ConvexShape mPolygon;

mPolygon.setPointCount(3);
mPolygon.setPoint(0, sf::Vector2f(0, 0));
mPolygon.setPoint(1, sf::Vector2f(0, 50));
mPolygon.setPoint(2, sf::Vector2f(50, 50));
mPolygon.setFillColor(sf::Color::Magenta);
mPolygon.setOutlineColor(sf::Color::Red);
mPolygon.setOutlineThickness(5);
mPolygon.setPosition(400, 400);
```

Sf::Image

- Class for loading, manipulating and saving images.

```
sf::Image mImage;

mImage.create(20, 20, sf::Color::Yellow);
sf::Color color = mIcon.getPixel(0, 0);
color.a = 0; //make the top-left pixel transparent
color.r = 0;   //set the r = 0 (rgb) from the color
mIcon.setPixel(0, 0, color);
// Save the image to a file
if (!mImage.saveToFile("Media/Textures/result.png"))
cout << "Cannot save my image to Textures folder " <<
endl;
```

How to set an Icon for the window

```
sf::Image mIcon;
mIcon.loadFromFile("Media/Textures/icon.png");
mWindow.setIcon(mIcon.getSize().x, mIcon.getSize().y,
mIcon.getPixelsPtr());
```

Font and Text

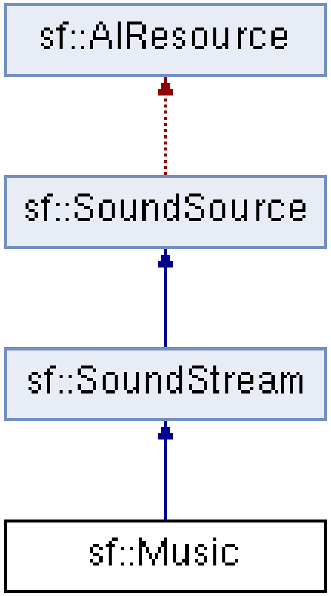
- Create a graphical text to display

```
sf::Font mFont;
sf::Text mText;
if
(!mFont.loadFromFile("Media/Sansation.ttf"))
return;


mText.setString("Hello SFML");
mText.setFont(mFont);
mText.setPosition(5.f, 5.f);
mText.setCharacterSize(50);
mText.setFillColor(sf::Color::Black);
```

Sf::Music

- Streams music played from an audio file



Play the Music

- Streamed music played from an audio file
- Musics are sounds that are streamed rather than completely loaded in memory
- A music is played in its own thread in order not block the rest of the program
- Supported audio formats: ogg, wav, flac, aiff, au, raw, paf, svx, nist, voc, ircam, w64, mat4, mat5, pvf, htk, sds, avr, sd2, caf, wve, mpc2k, rf64

Music Parameters

```cpp
#include <SFML/Audio.hpp>
sf::Music mMusic;
mMusic.openFromFile("Media/Textures/nice_music.ogg");
//change some parameters
mMusic.setPosition(0, 1, 10); //change its 3D position
mMusic.setPitch(2); //increase the pitch
mMusic.setVolume(50); //reduce the volume
mMusic.setLoop(true); //make it loop
mMusic.setAttenuation(100);
mMusic.play();
```

Game Loop

- The run() function you saw in the example from last week, and below, is known as the main loop or game loop

```
void Game::run()
{
   while (mWindow.isOpen())
   {
      processEvents();
      update();
      render();
   }
}
```

Game Loop (cont'd.)

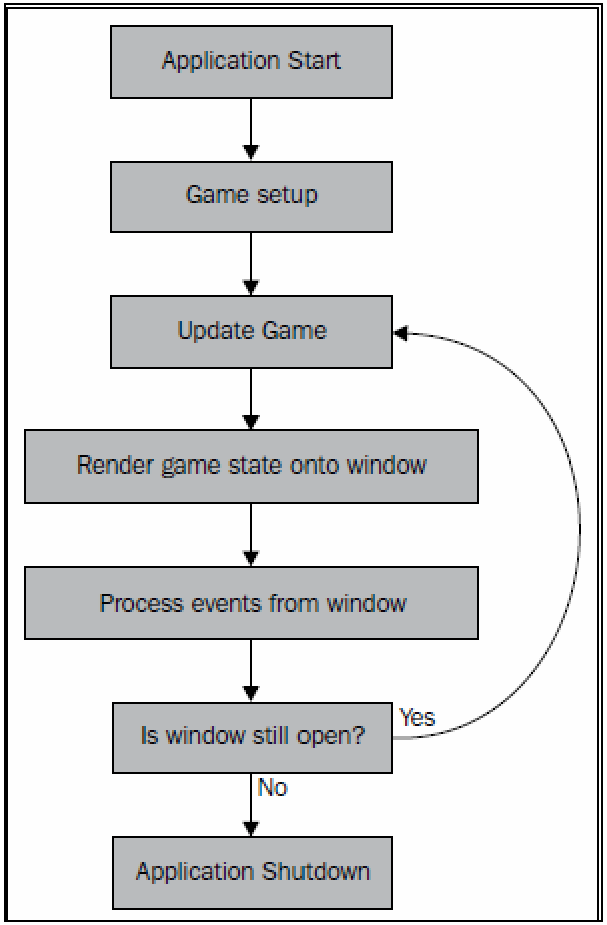
- It processes all the components in the game and continues to do so until the application is terminated
- The processing of events, updating of all game assets and then rendering them to the destination output is a standard loop for games

Game Loop (cont'd.)

- It processes all the components in the game and continues to do so until the application is terminated
- The processing of events, updating of all game assets and then rendering them to the destination output is a standard loop for games
- You've heard the term **frame** or **tick** before, and that is what we call an iteration of the loop

Game Loop (cont'd.)

- The flowchart to the right illustrates the logic and different processes of the game, including the main loop



Events

- Events can be user-generated such as mouse clicks or movement or keyboard presses
- They can also be generated by the assets in the game, such as when an enemy spots the player
- We don’t have any events in our example yet, so let’s create some!

- How ‘bout moving that circle with the keyboard?

Events (cont'd.)

- For events in SFML, we use the `sf::Event` object
- We're going to use two events for this example:

`sf::Event::KeyPressed` and `sf::Event::KeyReleased`

processEvents()

```
void Game::processEvents()
{
   sf::Event event;
   while (mWindow.pollEvent(event))
   {
      switch (event.type)
      {
         case sf::Event::KeyPressed:
           handlePlayerInput(event.key.code, true);
           break;
         case sf::Event::KeyReleased:
           handlePlayerInput(event.key.code, false);
           break;
         case sf::Event::Closed:
           mWindow.close();
           break;
      }
   }
}
```

handlePlayerInput()

```
void Game::handlePlayerInput(sf::Keyboard::Key key, bool isPressed)
{
      if (key == sf::Keyboard::W)
        mIsMovingUp = isPressed;
      else if (key == sf::Keyboard::S)
        mIsMovingDown = isPressed;
      else if (key == sf::Keyboard::A)
        mIsMovingLeft = isPressed;
      else if (key == sf::Keyboard::D)
        mIsMovingRight = isPressed;
}
```

New update()

```
void Game::update()
{
    sf::Vector2f movement(0.f, 0.f);
    if (mIsMovingUp)
       movement.y -= 1.f;
    if (mIsMovingDown)
       movement.y += 1.f;
    if (mIsMovingLeft)
       movement.x -= 1.f;
    if (mIsMovingRight)
       movement.x += 1.f;

    mPlayer.move(movement);
}
```

How about mouse?

```
if (sf::Mouse::isButtonPressed(sf::Mouse::Left))
{
sf::Vector2i mousePosition =
sf::Mouse::getPosition(mWindow);
mPlayer.setPosition((float)mousePosition.x,
(float)mousePosition.y);
}
```

Vector Object

- SFML's Vector object is instantiated as:

```
sf::Vector2<float>
```

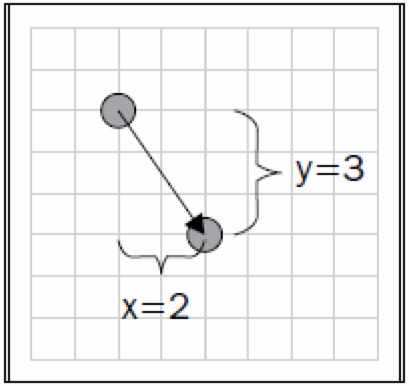
- We use the typedef for variable declarations, which is as follows:

```
sf::Vector2f myVector(0.f, 0.f);
```

- As you would expect, there are many common operations in the Vector class that we can access through a Vector object

Vector Object (cont'd.)

- In games, vectors can represent coordinates or a direction to move
- The diagram below represents a vector(2,3) and it could be a translation of 2 units to the right and 3 down:



Frame-Independence

- You might remember from Unity that we were able to move an object a certain number of units per second
  - We multiplied the speed by Time.deltaTime

- We can do this in SFML too, so that the player’s movement isn’t dependent on the framerate – or number of times the update runs per second

Frame-Independence (cont'd.)

- Well, we're still relying on the framerate, but the movement is spread out evenly over the frames
- So let's have a look at our new update function and see what's new…

New update ()

```
void Game::update(sf::Time deltaTime)
{
sf::Vector2f movement(0.f, 0.f);
if (mIsMovingUp)
movement.y -= PlayerSpeed;
if (mIsMovingDown)
movement.y += PlayerSpeed;
if (mIsMovingLeft)
movement.x -= PlayerSpeed;
if (mIsMovingRight)
movement.x += PlayerSpeed;
mPlayer.move(movement * deltaTime.asSeconds());
}
```

Measuring Frames

- We can measure the time each frame takes in order to figure out `deltaTime`
- We use the `sf::Clock` class
- Sf::Clock has only two methods: getElapsedTime() and restart(). Both returns the elapsed time since the clock was started and then it resets the clock to zero. getElapsedTime() can be called without calling restart()

```
sf::Clock clock;
sf::Time time = clock.getElapsedTime();
float seconds = time.asSeconds();
sf::Int32 milliseconds = time.asMilliseconds();
sf::Int64 microseconds = time.asMicroseconds();
time = clock.restart();
```

Measuring Frames (cont'd.)

```
void Game::run()
{

sf::Clock clock;
while (mWindow.isOpen())
{
sf::Time deltaTime = clock.restart();
processEvents();
update(deltaTime);
render();
}
}
```

Fixed Time Step

- Time based on a system function such as a while loop will never be constant
  - We saw this way back with HTML5 and its highly-varying frame rate
  - Unity too!
- Fortunately we can create fixed time execution using a counter and a check
  - The while loop is definitely going to execute fast enough to serve as very small time increments added to our counter variable

Fixed Time Step (cont'd.)

```
void Game::run()
{
   sf::Clock clock;
   sf::Time timeSinceLastUpdate = sf::Time::Zero;
   while (mWindow.isOpen())
   {
      processEvents();
      timeSinceLastUpdate += clock.restart();
      while (timeSinceLastUpdate > TimePerFrame)
      {
         timeSinceLastUpdate -= TimePerFrame;
         processEvents();
         update(TimePerFrame);
      }
      render();
   }
}
```

Fixed Time Step (cont'd.)

- If you want to read more on this topic, you can read the article at the following address:
  - http://gafferongames.com/game-physics/fix-your-timestep

Displaying Sprites

```
sf::Texture texture;
if (!texture.loadFromFile("path/to/file.png"))
{
  // Handle loading error
}
sf::Sprite sprite(texture);
sprite.setPosition(100.f, 100.f);
window.clear();
window.draw(sprite);
window.display();
```

Rendering

- Rendering is the process of drawing your assets to the screen
- Ideally, we'd only want our assets being drawn if they were updated somehow in the program
  - Would save on performance
- However, *real-time rendering* just draws to the screen as fast as possible

Rendering (cont'd.)

- Have you ever wondered why there is an FPS count in games? Like 30 or 60
- It's because the end user can't see blindingly-fast updates so the frame rate is limited to allow the processor to perform other tasks

Rendering (cont'd.)

- *Double buffering* is a technique of rendering that uses two virtual windows or screens, called buffers
  - Front and back buffers
- The front buffer is what the user sees
- The back buffer is what's going to be drawn next frame – will become the front buffer

Adding the Sprite

```
// Game.hpp
class Game
{
  public:
    Game();
    ...
  private:
    sf::Texture mTexture;
    sf::Sprite mPlayer;
    ...
};
```

Adding the Sprite (cont'd.)

```
// Game.cpp
Game::Game()
: ...
, mTexture()
, mPlayer()
{
  if (!mTexture.loadFromFile("Media/Textures/Eagle.png"))
  {
    // Handle loading error
  }
  mPlayer.setTexture(mTexture);
  mPlayer.setPosition(100.f, 100.f);
}
```

Appendix B – Code Editor Tips

- Zoom
  - Press **Ctrl+Mouse** wheel to increase and decrease the font size
- Box Selector
  - Select rectangular region, type a line, and have that line repeated for every line part of the region
    - Hold **Alt** while using mouse to make vertical section
    - Start typing…typed line appears on every line in region
- Generate from Usage
  - Type method invocation without first creating method
    - **Ctrl** + dot (.) automatically generates heading

Appendix C – Visual Studio Configuration

- When Visual Studio is launched, Start Page is displayed
  - Checkbox in extreme left corner – **Show page at startup**
    - Check/uncheck **Close page after project load** to add/remove Start page tab from development environment
- Pin a project so it is always there
  - Pushpin is revealed when you move mouse over Recent projects area

Customizing

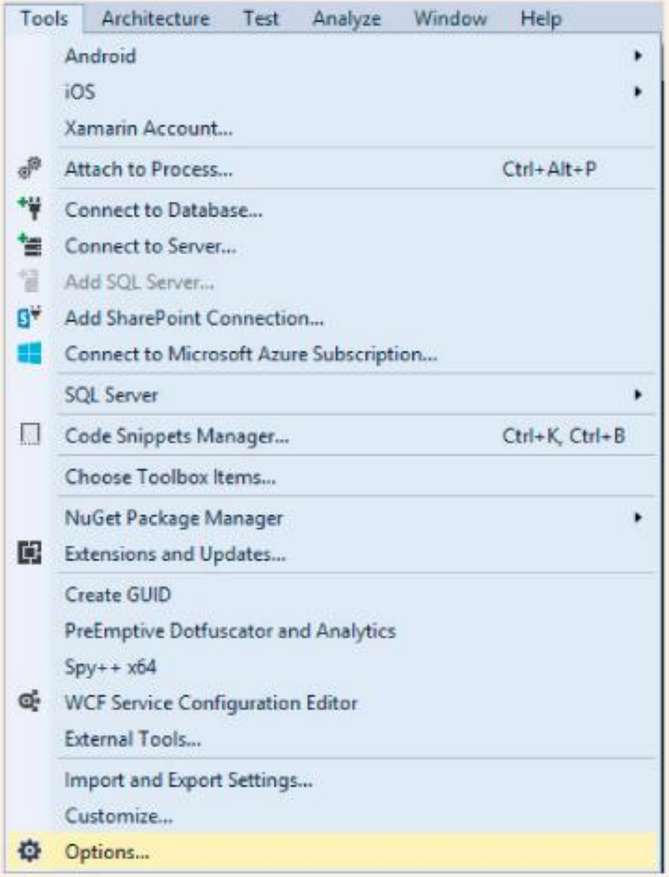
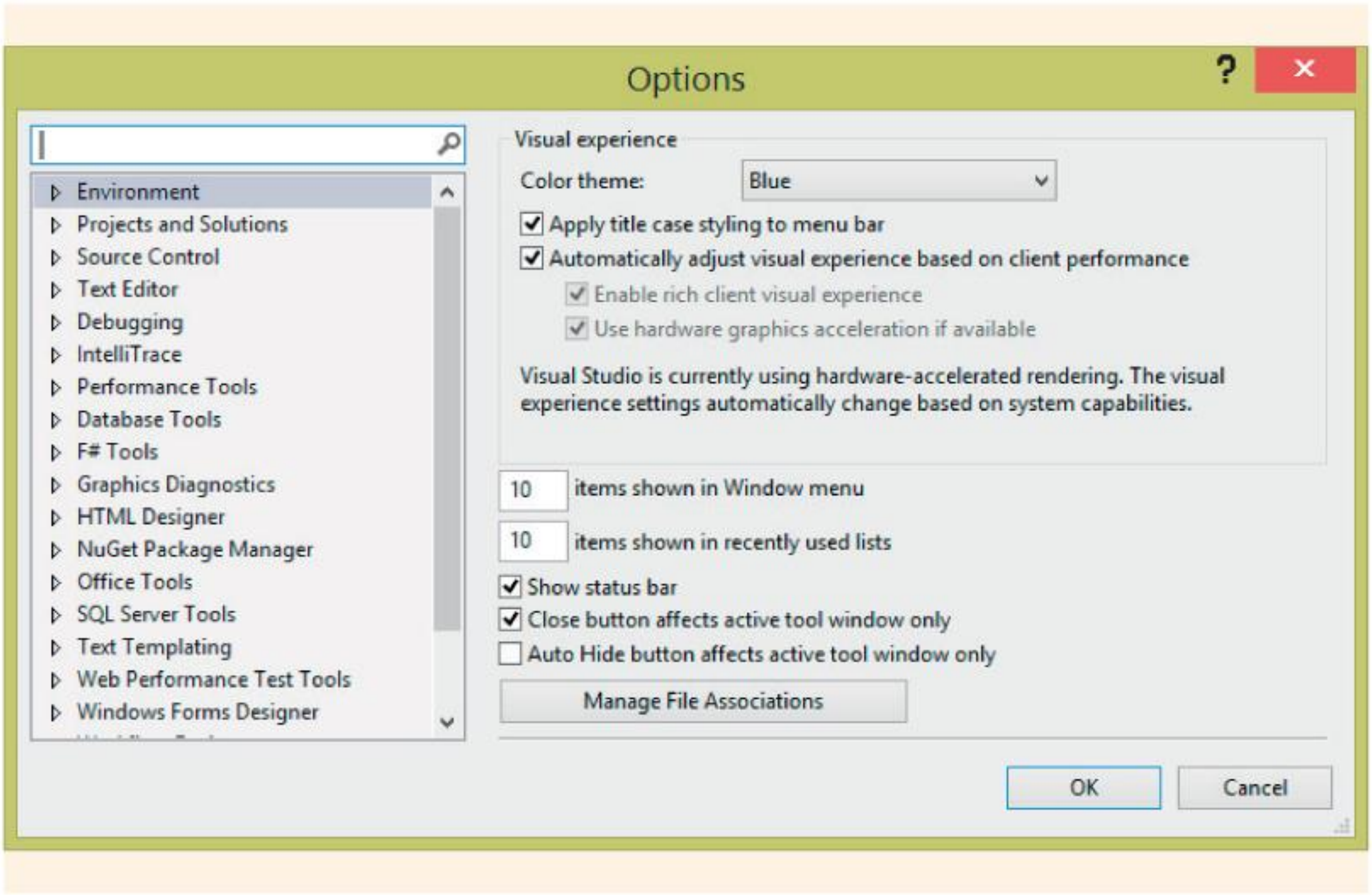

Tools
Architecture
Test
Analyze
Window
Help
Android
iOS
Xamarin Account...
Attach to Process... Ctrl+Alt+P
Connect to Database...
Connect to Server...
Add SQL Server...
Add SharePoint Connection...
Connect to Microsoft Azure Subscription...
SQL Server
Code Snippets Manager... Ctrl+K, Ctrl+B
Choose Toolbox Items...
NuGet Package Manager
Extensions and Updates...
Create GUID
PreEmptive Dotfuscator and Analytics
Spy++ x64
WCF Service Configuration Editor
External Tools...
Import and Export Settings...
Customize...
Options...

FIGURE A-2 Using Tools, Options to configure Visual Studio

Environment



Visual experience

Color theme: Blue

☑ Apply title case styling to menu bar

☑ Automatically adjust visual experience based on client performance

☑ Enable rich client visual experience

☑ Use hardware graphics acceleration if available

Visual Studio is currently using hardware-accelerated rendering. The visual experience settings automatically change based on system capabilities.

10 items shown in Window menu

10 items shown in recently used lists

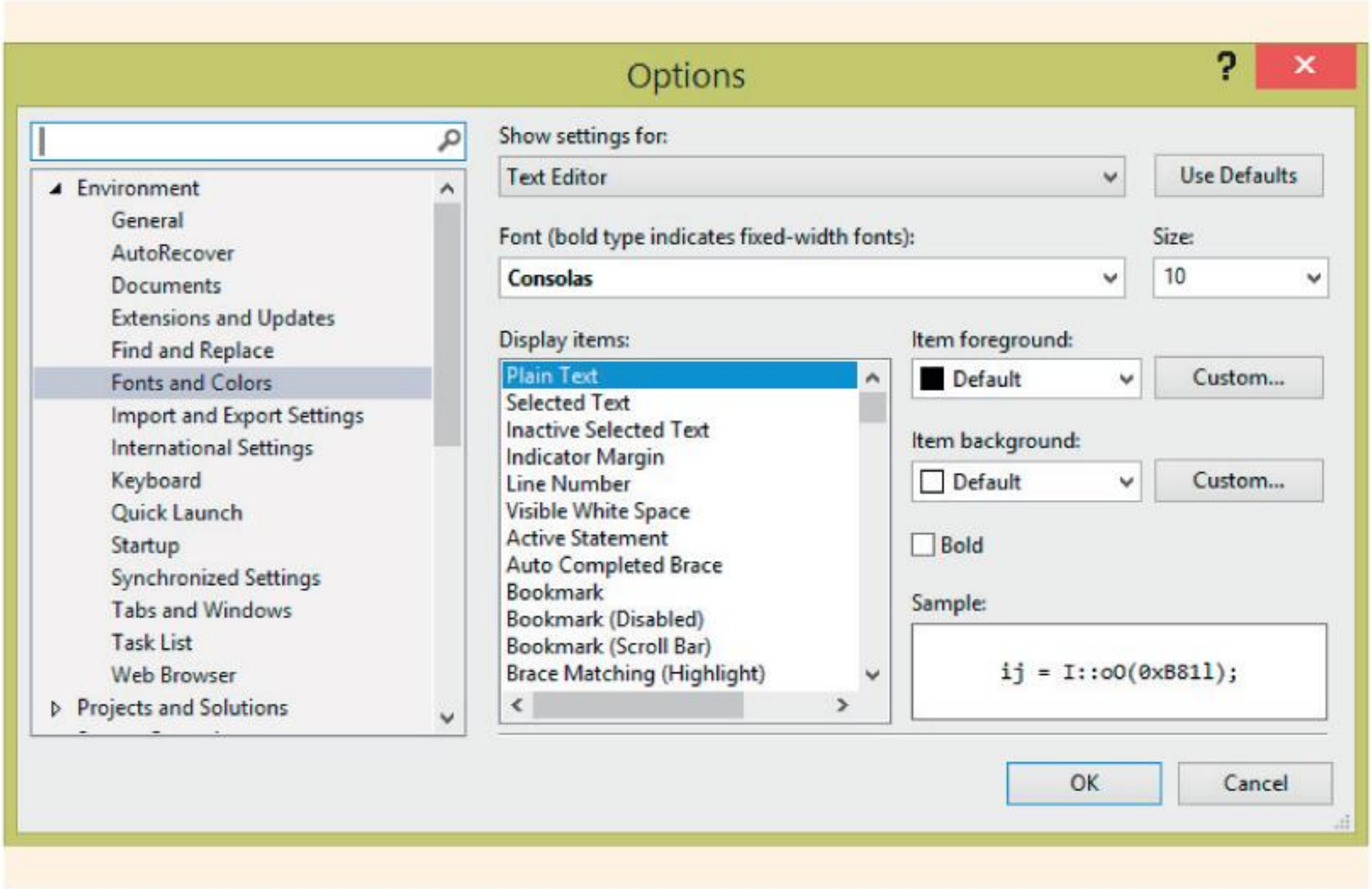
☑ Show status bar

☑ Close button affects active tool window only

□ Auto Hide button affects active tool window only

FIGURE A-3 Environment General options

Environment



Options

Show settings for: Text Editor

Font (bold type indicates fixed-width fonts): Consolas

Size: 10

Item foreground: ■ Default

Item background: □ Default

□ Bold

Sample:

```
ij = I::o0(0xB81l);
```

FIGURE A-4 Setting the fonts and colors

Environment

- **Environment**, **Startup** node lets you determine what is opened when you first start Visual Studio
  - **Show Start Page**, **Open Home Page**, **Load last loaded solution**, **Show Open Project** dialog box, **Show New Project** dialog box, or **Show empty environment**
- Set the URL for your Home page and Search page from **Environment**, **Web Browser** node