

Beginner’s Guide to SDL

Building Games with SDL and C++

11/11/11

nyguerrillagirl@brainycode.com

Copyright © 2011 brainycode.com

Permission is granted to copy and distribute an electronic version of this document.

Permission is NOT granted for commercial use.

Dedication to those learning C++	6
Chapter 0: Introduction	7
Purpose.....	7
Prerequisite	8
What is out there now on SDL?	8
Why SDL?	9
A Bit of History	9
The Components of SDL	10
Chapter 1 – Installing SDL	14
Obtaining copies of all the libraries	14
Installing to work with Code::Blocks	17
Using the Code::Blocks SDL wizard.....	28
Creating your own SDL Custom Template.....	28
Installing to work with WxDev-C++	29
Creating a WxDev-Cpp SDL template file	39
Installing and Testing SDL_Image	41
Installing and Testing SDL_tff	44
Installing and Testing SDL_mixer.....	47
Installing and Testing SDL_net	48
Summary.....	48
Chapter 2 - Getting started with SDL	49
Initializing SDL	49
Initializing and Closing SDL Subsystems	53
The Video Component.....	58
SDL Video Structures	59
Making Improvements to our Video Programs.....	64
How the display screen is organized.....	66
Understanding how to write to the Display	67
Drawing a Line	78
A Little History on Drawing Lines	78
A Simple Algorithm – Slope-Intercept Algorithm.....	79
A Simple Algorithm #2 – Using Symmetry.....	83
SDL_Rect.....	84

Clipping	87
SDL_VideoInfo.....	89
Loading Images.....	91
Moving a “Ball” around on the screen.....	94
Double Buffering and Page Flipping	101
Double Buffering	101
Page Flipping	102
Displaying Other Types of Images	104
Alpha Blending	109
Other Topics.....	114
Summary	114
Review Questions	115
Programming Exercises	116
Circle-Drawing Algorithms	117
Chapter 3 - Sprites, A Simple View.....	123
Chapter 4 - Processing Events	124
Keyboard Events.....	126
Joystick Events.....	130
System Events.....	131
Mouse Events.....	131
Chapter 5 – How to organize a game	131
Creating a Game Template	131
Sample Games	131
Chapter 6 – Creating Pong	131
Using SDL_TTF	131
SDL Audio	131
Some Audio Basics	132
Using SDL primitive functions	134
SDL Joystick.....	136
Chapter 7 – Creating MindSweeper.....	136
Chapter 8 – Creating Breakout.....	136
Chapter 9 – Creating Tetris.....	136
Chapter 10 – SDL Threads and Timers	136

Chapter 11 – Building a multiplayer online game	136
SDL_NET	136
SDL_MIXER	136
Chapter 12 – Building a Platform Game.....	136
Why I love Crisis Mountain!	136
Why I love Mario!!	136
Chapter 13 – Other libraries and tools to build games.....	137
Chapter 14 – What comes next?	137
Last Chapter	137
Bibliography	138
Appendix A: Places to visit on the Web	139
Appendix B – Microsoft Visual C++ 2010 Express	140
Appendix C – Pong, Breakout and MindSweeper	141
Pong	141
Breakout.....	141
MindSweeper	142
Appendix D – Unzipping files	145
Appendix E – Structs	150
What are structs?.....	150
Why use structs?	150
In summary	154
Things you can do with structs	155
Things you can't do with structs	156
Using typedef with structs	156
Appendix F – Pointers	158

Dedication to those learning C++

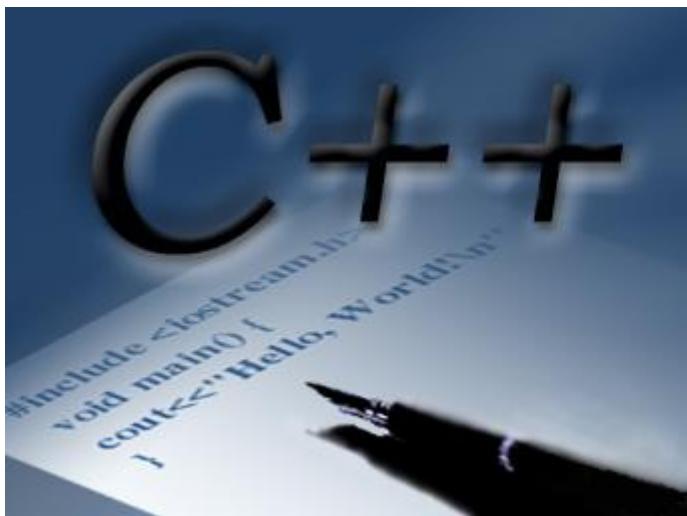


Figure 1 - C++ image

I wrote this with the students of Ocean County College¹ in mind. I wanted to create something that complemented the material they were learning in their second C++ course. The C++ course series at the college is partitioned into three courses:

- ⊕ Course 1 – covers simple data types, user-defined data types, string, operators, input, output, control structures, the ifStatement, switchStatement, loops using forStatement, whileStatements, and functions.
- ⊕ Course 2 – covers arrays, strings, vectors, structs, classes, pointers, overloading and virtual functions
- ⊕ Course 3 – covers over templates, exception handling, recursion, linked lists, stacks, and queues

The book used in the course is a good one – “C++ Programming: From Analysis to Program Design” by D.S. Malik. The only problem I saw with the book is that many students were looking for problems to solve that they could relate to – games. I originally wrote a set of notes on Windows Console Programming that demonstrated how to build Pong, Breakout and Minesweeper with just the Windows Console. I did not realize until recently that a better way to stimulate thinking and usage of the C++ constructs we were learning in the second course was to use a tool as simple and as powerful as SDL. SDL provides the capability for students to learn the key concepts of classes, pointers, and arrays and actually enjoy the programs they build since most can relate to the elements of a game. I don’t see these notes as a replacement of the material in the book. In fact, a good C++ student should read the material, look over the exercises and do as many programs as they can fit into their schedule.

The second goal is to explain in some detail the new C++ constructs starting in Course #2. So if you are more experienced and you don’t need an explanation of classes or pointers or other features I may think someone just learning C++ may not know then feel free to skip over those sections.

Have fun and build a fun game!

¹ These notes began at the time the students at OCC were required to learn C++, since that time they have moved to using Java as the programming language to start the program. I still highly recommend C++ for future game programmers.

Chapter 0: Introduction



Figure 2 - SDL Logo (<http://www.libsdl.org/>)

Purpose

This book discusses how to build 2D games using free compiler IDEs and graphics libraries. The specific graphics tool we address in these notes is SDL which is an acronym for Simple Directmedia Layer graphics library. A key thing to note about SDL is that it is cross-platform. A program written using SDL can be re-compiled to run on various operating systems platforms such as Windows, Mac, and many UNIX variants with little to no changes to the code.

The SDL library provides functions that make it easy to build 2D type games. A 2D game is a game that takes place on a static playing field (e.g. Pong, Donkey Kong) or scrolls (e.g. Mario Bros.). The graphics are simple due to the initial limitation of the game systems and computers that were used when they were first designed. The action and game play can be as enjoyable as any modern 3D game made today. A testimony to the fun and success of 2D games is their resurgence and popularity on Xbox Live Arcade and Wii Virtual Console.

The exercises and programs we build will be created using the free C++ compilers and IDE such as WxDev-C++ or

Code::Blocks. In addition the SDL library and all additional supporting libraries designed to work with SDL will be utilized. We will build some simple games – Pong, Breakout, Minesweeper and Tetris. In addition, at the end we build a platform scrolling game to demonstrate some more advance game techniques.



Figure 3 - Nintendo Donkey Kong

We will be using Windows² version of tools and libraries in our discussion. But, since all the tools and libraries have cross-platform versions users preferring to use a different operating system should have no problem following along.

Prerequisite



Figure 4 - Getting things together

In order for these notes to make sense and be worth your investment in time and energy is for you to have at least one semester of C++ under your belt. There are many useful websites if you are learning C++ on your own. I recommend <http://www.steveheller.com/cppad/Output/dialogTOC.html>. Of course, the best way to learn your first programming language is to take a course at your local college. The only downside to that suggestion is that the typical college text book tends to be over \$100. There will be plenty of exercises as we move along. I highly recommend doing all the exercises in order to build up a working knowledge of C++ and SDL.

All the programs will be explained in detail and you are encouraged to complete each one as a working model.

What is out there now on SDL?

There are few books on learning SDL only one directs itself to Windows environment, the book, “Focus on SDL” by Ernest Pazera. The book was published in 2003 and is currently out of print. It can only be obtained used over the Internet. I think the book should have been at least double its current size, in fact; I felt it was pared down to fit into the “Focus On” series which consists of small books that take a look at a topic. In any case, I found it a great source of information but wish the author had more examples and would have built a game from beginning to end in order to solidify all the new concepts being presented. There are other books that cover SDL “Linux Game Programming” by Mark “Nurgle” Collins, et al. The book is of course Linux focused but the SDL material will work on Windows as well since of course it is cross-platform. The book presents a rather well designed code structure for games (you can’t ever go wrong using it!). This book is also out of print.³ The other two book I found with several chapters dedicated to SDL are both by the same author – Erik Yuzwa. “Game Programming in C++: Start to Finish” and “Learn C++ by Making Games”. I think both books are great additions to anyone learning to program in C++ and also interested in making games in the process. The only difference between them and this book is that this book is FREE and I hope a valuable resource.

There are several websites you should be able to find on the Internet. One website has a rather good set of tutorials - <http://www.sdl tutorials.com>. In fact, I use its class layout as a basis for the game programs later in this book. Another website you should visit is <http://www.libsdl.org/> to obtain the latest version of SDL and more information on the library. The key advantage to using this book is that it presents a detailed

² We are of course referring to Microsoft Windows. All code has been tested on Windows XP, Windows Vista and Windows 7 version of Windows.

³ My favorite place to purchase out of print books is amazon.com. You can always find 3rd party merchants to purchase new or used copies of books, of course the rarer the book is the more expensive it will be! I think all out of print books should find themselves in an electronics book cemetery.

explanation of the SDL functions and gently guides you in using more advanced C++ constructs and additionally presents the mechanics for building a 2D game.

Why SDL?



Figure 5 - Commander Keen

I started to investigate SDL in order to understand its use in a game named Klone Keen. The game is a free version of the game engine that runs the game Commander Keen by Id Software. The game Commander Keen was released as shareware⁴ in a series of games in the early 1990's. The game was notable for being able to replicate “the side-scrolling action of the Nintendo Entertainment System Super Mario Bros. games in DOS.” You will need the original data files in order to run Klone Keen. The game made use of several free libraries one of which was SDL for sound, video and keyboard processing. In my quest to learn SDL I purchased several books that referenced or used SDL and checked out all the online resources. I felt that SDL was the perfect library for students to learn in order to be able to build programs that were more interesting than the typical programming assignments we usually give students in their second C++ course. The goal I had was to create a comprehensive book that was not restricted in size by any publishing constraints and would present several game variations in order to addressing different programming techniques that are useful in the creation of 2D games. The greatest difference is my intention to make this book available for free.

I started to investigate SDL in order to understand its use in a game named Klone Keen. The game is a free version of the game engine that runs the game Commander Keen by Id Software. The game Commander Keen was released as shareware⁴ in a series of games in the early 1990's. The game was notable for being able to replicate “the side-scrolling action of the Nintendo Entertainment System Super Mario Bros. games in DOS.” You will need the original data files in order to run Klone Keen. The game made use of several free libraries one of

which was SDL for sound, video and keyboard processing. In my quest to learn SDL I purchased several books that referenced or used SDL and checked out all the online resources. I felt that SDL was the perfect library for students to learn in order to be able to build programs that were more interesting than the typical programming assignments we usually give students in their second C++ course. The goal I had was to create a comprehensive book that was not restricted in size by any publishing constraints and would present several game variations in order to addressing different programming techniques that are useful in the creation of 2D games. The greatest difference is my intention to make this book available for free.

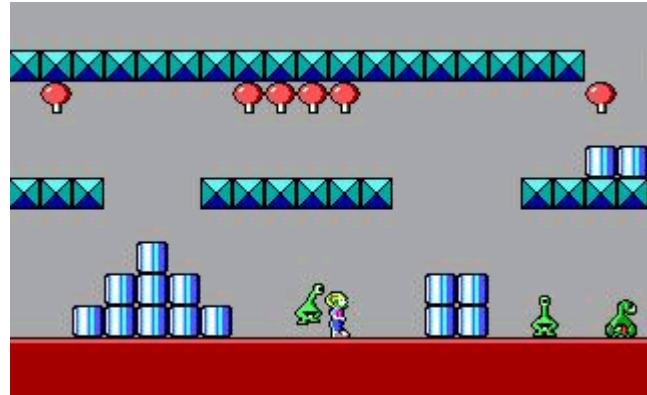


Figure 6 - A level in Commander Keen

A Bit of History

SDL was created by Sam Lantinga and released to the public in 1998. He has worked on many game projects and has been with Blizzard Entertainment since 2009⁵. You can check out his resume online at <http://www.devolution.com>. I was impressed to see that he made major contributions to my favorite game of all time – World of Warcraft.

Lantinga was inspired to create the SDL in order to support his efforts in porting applications from Windows to the Mac. The library was created to facilitate creating and porting applications, esp. games to different platforms with minimal code changes. Today, many applications use various



⁴ Shareware is “try before you buy” software. Programmers release the first couple of users to play and evaluate, many users would purchase the entire game especially if the

⁵ This is true as of April 2010

components of SDL to perform functions such as keyboard handling or joystick support and complement it with OpenGL⁶ or other libraries.

SDL was written in C but can be used with C++ (as we plan on doing here), Perl, C, Python , etc.

Figure 7 - WOW

The Components of SDL

SDL is composed of eight subsystems:

- ✚ Video – This subsystem deals with the things you see on the screen. The windows, colors, and sprites that make up the visual components of a game. This component is only intended to support simple graphics operation and the user is assumed to be sophisticated enough to be able to create additional functions to draw lines, circles, etc. In our case, we will see how we can augment this library with additional free libraries written and released by others.
- ✚ Event Handling – This subsystem handles how the user interacts with the game. The user generates events whenever they minimize the window, move the mouse or press a key. You will find this component of SDL useful to employ even if you plan on moving on to build 3D games with other libraries.
- ✚ Joystick Handling – This subsystem deals with the various complexities involved in today’s joysticks. Each joystick has a different number of buttons, dials, switches, joysticks and wheels. This component provides a set of functions to manage a joystick.
- ✚ File I/O – This subsystem handles the reading in of bmp files. I should note that since SDL is typically used to create a multimedia application such as a game it has a wonderful feature that programmers can use cout and cerr to write messages and the messages get automatically saved into the data files – stdout.txt and stderr.txt. This is a great ready-to-use logging system.
- ✚ Audio – This subsystem supports sound you may want to add to your game.
- ✚ CDROM – This neat subsystem supports access and communication to a users CD ROM device.
- ✚ Timers – This subsystem provides functions to set and control when a sequence of events or processes takes place in a periodic fashion. It is similar to you alarm clock which you set everyday to wake you up at a certain time. You might set timers in a game program to update the screen, move the monsters, etc.
- ✚ Threading – SDL provides functions to handle establish and control more than one thread of execution. A “thread of execution” is a sub process that exists within a large process that runs independently. This feature provides the programmer the capability to do more than one thing at a time. Using threads adds a layer of complexity to your program since you will need to manage communication and access to resources.



Figure 8 - A souped up joystick!

⁶ OpenGL (Open Graphics Library) is another cross-platform graphics library used to build 2D and 3D applications. I plan on covering in my planned second book on building games.

There are additional libraries that complement SDL and the next chapter (Installing and Testing) will have you install them all in addition to SDL. The libraries are:

- ✚ **SDL_image** - This library has support for image formats other than bmp, such as PNG, GIF, JPEG and many others.
- ✚ **SDL_mixer** – This library provides support for playing music and sound samples from a greater set of formats, e.g. WAV, MOD, MP3, etc. “It supports any number of simultaneously playing channels of 16 bit stereo audio, plus a single channel of music, mixed by the popular MikMod MOD, Timidity MIDI, Ogg Vorbis, and SMPEG MP3 libraries.”
- ✚ **SDL_net** – This library provides a “small cross-platform networking” set of functions. It includes a chat client and server application.
- ✚ **SDL_ttf** – This library provides you the capability to use TrueType fonts. We will use this to display text on our game screens.



Figure 9 - Image of sample game created with SDL

These libraries serve to provide a layer or wrapper around operating system functionality. The libraries provide a set of common functions that hide the complexity and differences between platforms such as Macintosh and Windows.

“On Microsoft Windows, SDL uses a GDI backend by default.” GDI stands for Graphics Device Interface and it is an API that comes with Windows to provide programming support for the representation of graphical objects to be shown or displayed on computer monitors or printers. GDI is not known to be fast or the best way of building games on a Windows

platform. The other two popular options are DirectX and OpenGL. The current version of SDL was designed to use DirectX⁷. You may be wondering why we would choose to discuss SDL on Windows rather than just directly learn the latest DirectX. There are several reasons:

- ✚ I like the idea that SDL is cross-platform. It means students can use Windows in class or at home but those that prefer other operating systems platforms such as Linux can do all the programming examples on them. There is nothing in here that is Windows specific other than the detailed installation instructions that assume you will be installing all the software in Windows operating systems.
- ✚ Students don’t have to learn WinMain just yet. The entry point for your SDL windows application remains the main function:

```
int main(int argc, char* argv[])
{
}
```

⁷ DirectX is an API created by Microsoft for handling tasks related to multimedia, especially game programming.

The use of main avoids all the explanations that would be required if we had to get into the details on how to build a typical program to run under Windows.

- ✚ I can't say enough on how I really appreciate the fact that you can use cout and cerr for output and view the results in a file after the program executes or as more often happens – does not work as expected. It helps in debugging and troubleshooting the code. TIP: If you get into a situation where “nothing” happens then go to the directory where the *.exe file resides and search for the stderr.txt file for error messages.
- ✚ Someone learning C++ with the intention of building their own games can set up and start coding pretty quickly with these free tools.



Figure 10 - An image of nothing?

I am going to assume you know the following C++ concepts:

- ✚ Data Types
- ✚ Input/Output
- ✚ Arithmetic Operators
- ✚ Control Structures
 - if, if..else,
 - switch
 - while
 - for
- ✚ Functions

The topics above cover the knowledge the typical computer science student learns in their first programming course.

Along the way, you will also use the following additional topics:

- ✚ Enumeration Types
- ✚ Using typedef
- ✚ Pointers
- ✚ Arrays
- ✚ Records (struct)
- ✚ Classes

The topics above are covered in an Appendix in enough detail to understand these notes. Of course the best thing to do is the read the corresponding chapter in any good C++ book if you need more information and exercises or start reading these notes when you start working on the second half of your C++ course(s). All the problems and exercises in these notes relate to building the games we have in mind, in

#include <stdio.h>
int main(void)
{
 printf("Hello World!\n");
 return 0;
}

C/C++

addition the programs will help you to understand how useful the computer constructs above are in developing fun programs. But, there is more....

In addition to discussing how to use SDL and the helper libraries you will need to create games. We will also cover topics probably not covered in enough detail in your programming course:

- ✚ how to build and use your own .h files
- ✚ how to build and use your own libraries
- ✚ how a program evolves and develops in order to make it easier to change and expand
- ✚ how to use the joystick and mouse

The good news for novice programmers is that these topics are covered without getting into the ugly details of Windows Programming!

Chapter 1 – Installing SDL

Obtaining copies of all the libraries

You will need to obtain copies of SDL, SDL_image, SDL_mixer, SDL_net and SDL_ttf. Visit the following pages and download the Windows V6 version of each file referenced:

- ⊕ **SDL** - <http://www.libsdl.org/download-1.2.php>
 - Download the file `SDL-devel-1.2.14.mingw32.tar.gz`
 - Unzip⁸ (see Appendix D for instructions) the file to your C:\ drive.
 - You will have a directory similar to `SDL-1.2.14` (whatever the latest version of SDL you downloaded to your computer)
- ⊕ **SDL_image** - http://www.libsdl.org/projects/SDL_image/
 - Download the file `SDL_image-devel-1.2.10-VC`
 - Unzip the file to your C:\ drive
 - You will have a directory similar to `SDL_image-1.2.10` with two folders – \include and \lib.
- ⊕ **SDL_mixer** - http://www.libsdl.org/projects/SDL_mixer/
 - Download the file `SDL_mixer-devel-1.2.11-VC`
 - Unzip the file to your C:\ drive
 - You will have a directory similar to `SDL_mixer-1.2.11` with two folders - \include and \lib.
- ⊕ **SDL_net** – http://www.libsdl.org/projects/SDL_net/
 - Download the file `SDL_net-devel-1.2.7-VC8`
 - Unzip the file to your C:\ drive
 - You will have a directory similar to `SDL_net-1.2.7` with two folders - \include and \lib
- ⊕ **SDL_ttf** – http://www.libsdl.org/projects/SDL_ttf/
 - Download the file `SDL_ttf-devel-2.0.9-VC8`
 - Unzip the file to your C:\ drive
 - You will have directory similar to `SDL_ttf-2.0.9`
 - You will also need to obtain a font library, go to <http://www.freetype.org/> or the FreeType project on SourceForge <http://sourceforge.net/projects/freetype/>.

To save time and to use all the tools and libraries referenced in these notes you can simply visit <http://www.brainycode.com> and follow the “Learning SDL – A Beginner’s Guide” link on that website. The page will have links you can use to download the same exact versions. If you encounter a problem then I encourage you to visit the SDL forum on brainycode.com and post a question. I will try to respond and help out.

⁸ I use the term “unzip” to mean uncompress or open the files in a compressed archive. The file may be a *.zip file but it may be *.tar or *.gz format. The term is not intended to imply the file will always be in *.zip format.

The first thing to do is to go to <http://www.libsdl.org> and download the latest copy of the SDL library. SDL supports many different platforms – Windows (our preference), Linux, MacOS, and many more. Download the file SDL-1.2.14-win32.zip for Windows. I unzipped under the C:\ directory to get:

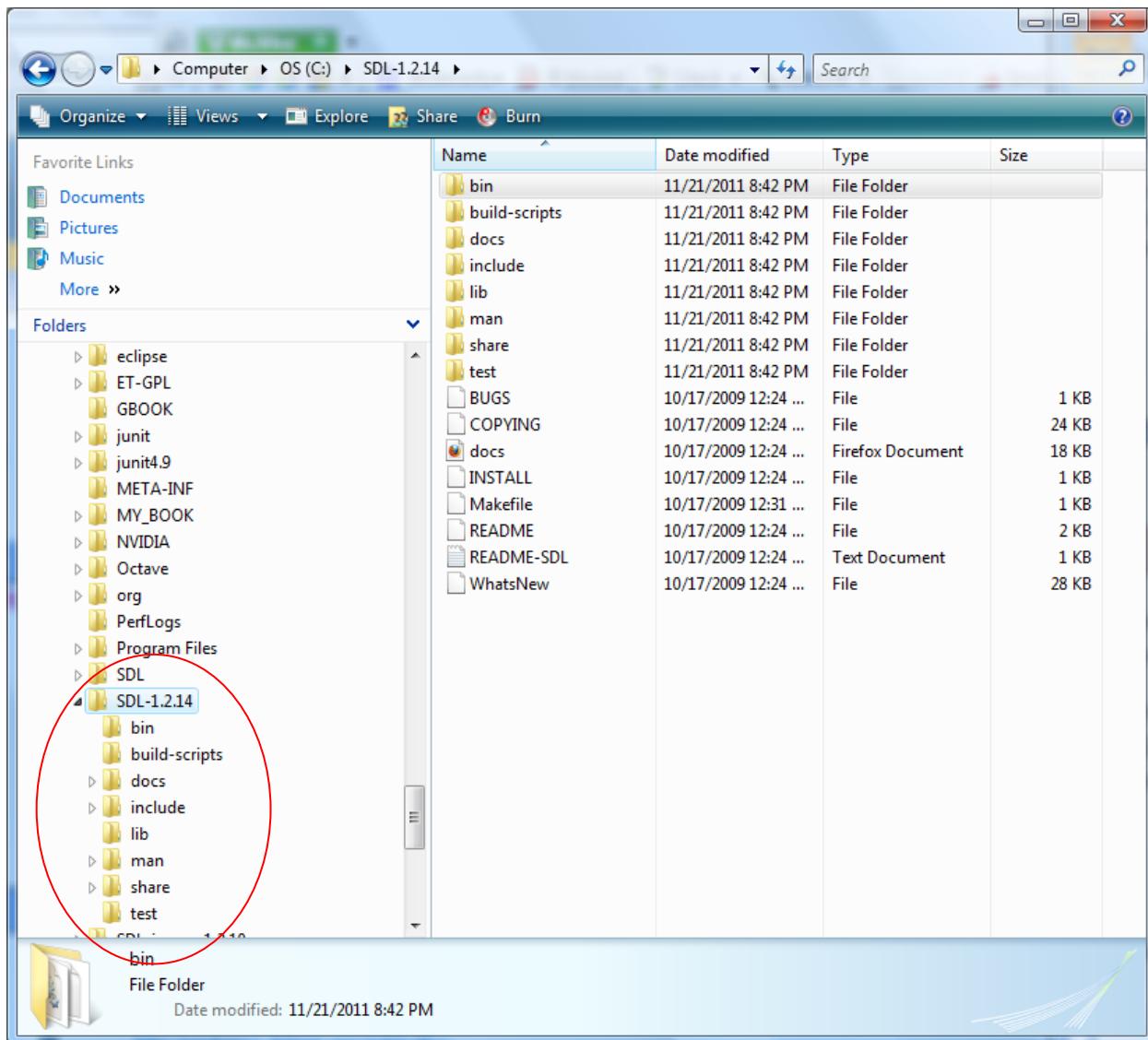


Figure 12 - Installation of SDL 1.2.14

You will need to either direct or copy the following files:

- ⊕ **SDL.dll** – This file is located in the \bin subdirectory. All your SDL exe applications will need to be able to access this file. If you place it in C:\Windows then the operating system will have no problem locating it. I favor using the wxDev-Cpp IDE since all I have to do is copy this file to the IDE’s bin directory and I have no problem running any program I am working on from within the

IDE. If you give your SDL game program to your best friend to play don’t forget to include this file.

- ⊕ **SDL.h** – This file is located in the \include\SDL directory. This needs to be accessed when your program is being compiled. I found the easy way is to copy the entire SDL subdirectory under the IDE’s include directory. (This works with no extra effort with wxDev-Cpp but you will have to set this include directory using Code::Blocks)
- ⊕ ***.a** – There are two library files you will need to have the IDE find in order to complete the link process of your SDL application – libSDL.dll.a and libSDLmain.a. I found the easiest thing to do is to copy into the IDE’s \lib directory.

You must now decide what Windows IDE you will be using. I recommend one of the following:

⊕ Code::Blocks

This IDE is available at

<http://www.codeblocks.org/>. It is free and operates across many operating systems – Windows, Linux and Mac. The IDE uses wxWidgets, which is a C++ library which is –yes – cross-platform GUI library. You can create a workspace that combines one or more projects. Each project is a collection of files making up your application. You can easily import your Dev-C++ or MSVC project files.

It also has a fully functional debugger that allows you to set code breakpoints, the call stack and disassembly of the code. You can even view the

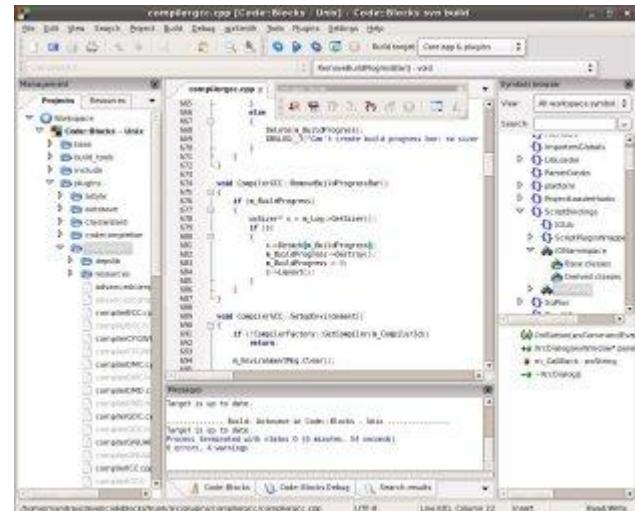


Figure 13 - CODE::BLOCKS IDE

CPU registers. The reason I highly recommend it is because it is supported and can be extended through the use of plug-ins.

<http://www.codeblocks.org/screenshots>

⊕ WxDev-C++

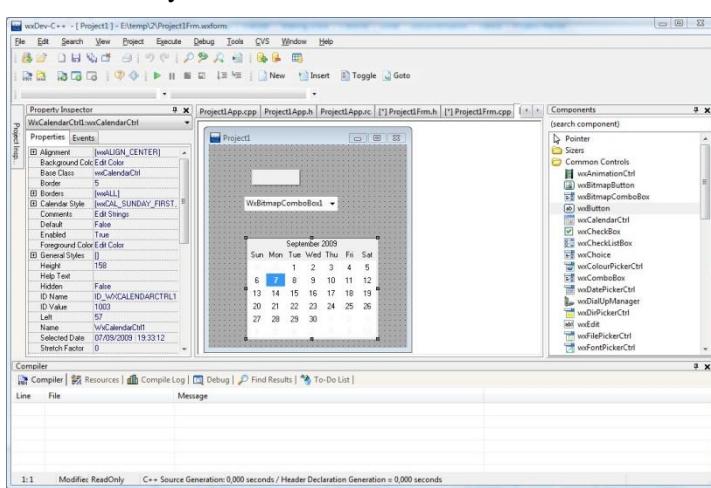


Figure 14 - WxDev-C++ IDE

wx-Dev-C++ is an extension of Dev-C++. It is similar to Code::Blocks in that it uses wxWidgets and is a free open-source IDE. It has a form designer, integrated debugging and nice editor features. Unlike Dev-C++ this product is currently supported and grows. You can pick up a copy at <http://wxdsgn.sourceforge.net/>.

I prefer to use WxDev-C++ throughout these notes but will try to switch over to Code::Blocks (if I see the latest version works well on Vista). The next sections show how to install and test the SDL libraries on both IDEs.

Regardless of which IDE you choose to use we will direct the IDE to search the include files under the SDL /include directory.

This will mean that when we create our C++ programs we will add the following include statement:

```
#include "SDL\SDL.h"
```

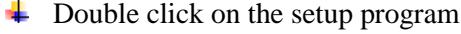
or

```
#include "SDL\SDL_ttf.h"
```

Installing to work with Code::Blocks

Steps for getting SDL program to compile and execute using Code::Blocks.

1. Obtain an install the latest version of Code::Blocks C++ IDE from <http://www.codeblocks.org/>
 - a. Note: I installed under the directory C:\CodeBlocks since some Windows operating systems create unnecessary⁹ problems when trying to change or add files under C:\Program Files.
 - b. Note: Install the version with MingGW, codeblocks-8.02mingw-setup.exe



- Double click on the setup program

⁹ I hate having to click through dialog boxes to install files under C:\Program Files under Windows Vista.



Figure 15 - Code::Blocks setup Wizard screen

- ⊕ Select Next>

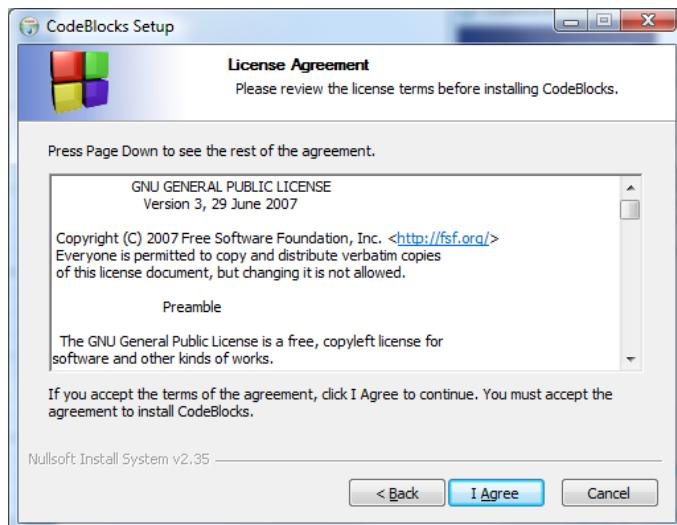


Figure 16 - GNU License Agreement

- ⊕ Most people don't read this screen. The GNU GENERAL PUBLIC LICENSE is quite different than your typical agreement. Read it. Then click on "I Agree".

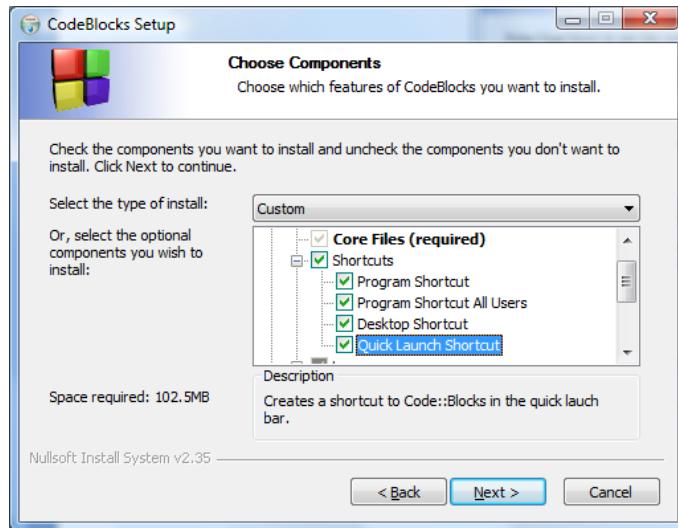


Figure 17 - Choose Components Dialog Box

- ➊ I clicked on the Default Installation node in order to select more shortcuts. Select what makes sense and click on “Next>”

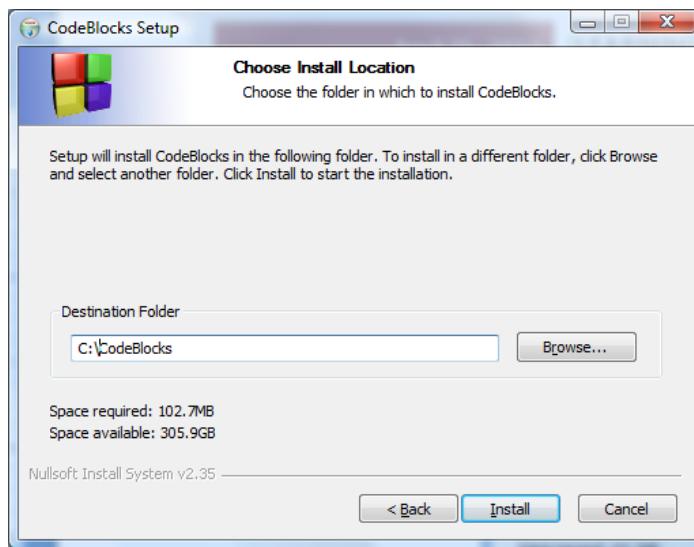


Figure 18 - Choose Install Location Dialog Box

- ➋ I decided to save under C:\CodeBlocks rather than in the default directory C:\Program Files\CodeBlocks. Click on “Install”.

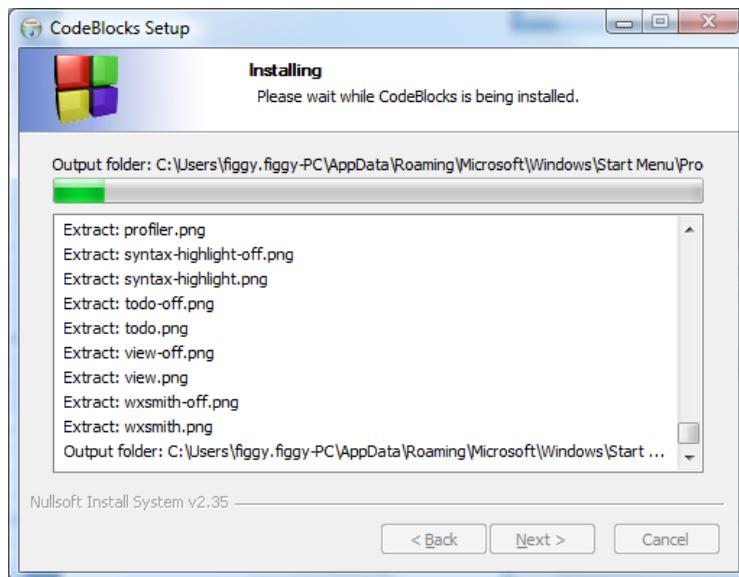


Figure 19 - Installing progress dialog box

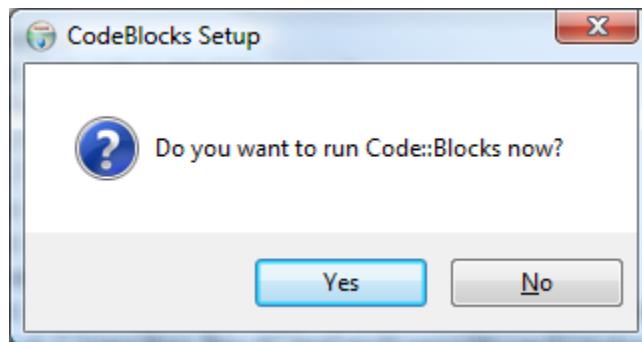


Figure 20 - Final Code::Blocks dialog

- ✚ I selected “Yes” in order to see how the IDE starts up and Finish installing. Make sure you close out and set the default compiler you want Code::Blocks to use.

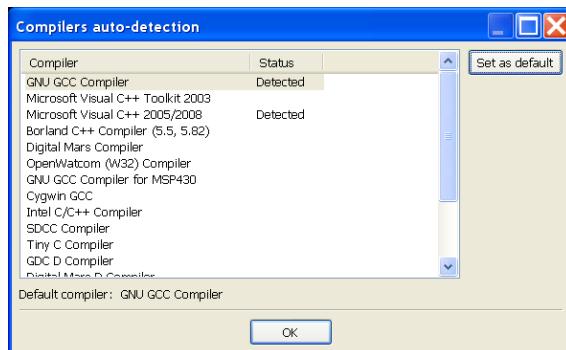


Figure 21 - Set GNU GCC Compiler as the default compiler

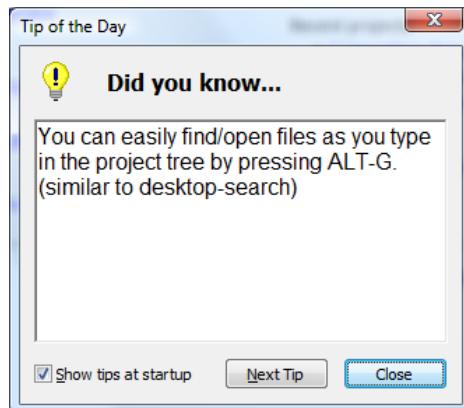


Figure 22 - Tip of the Day message

I recommend letting the IDE startup with the “Tip of the Day”.

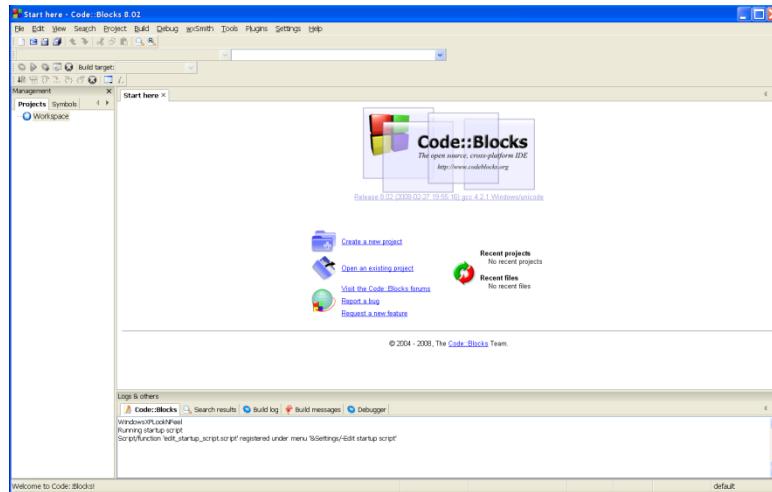


Figure 23 - Code::Blocks start up screen

2. Copy SDL.dll under C:\SDL-1.2.14\bin to Code::Blocks bin directory.
 - a. This will allow the IDE to find the SDL.dll when you execute from the IDE, otherwise you will always have to copy the SDL.dll into the same directory as the exe file.

We have now completed all the steps that need only happen one time.

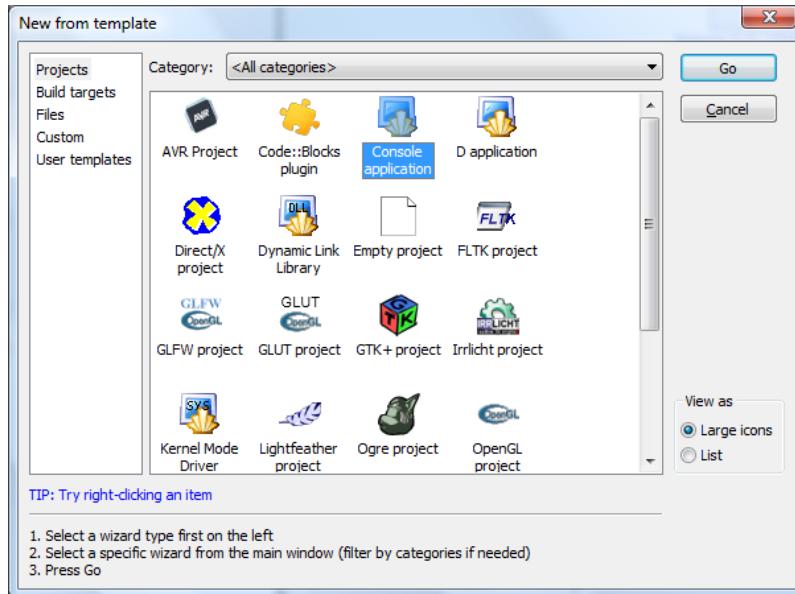


Figure 24 - Creating a Console Application

3. Create a C++ Project to test simple SDL application. Select File | New Project. Click on “Console Project” icon in the dialog box that appears.

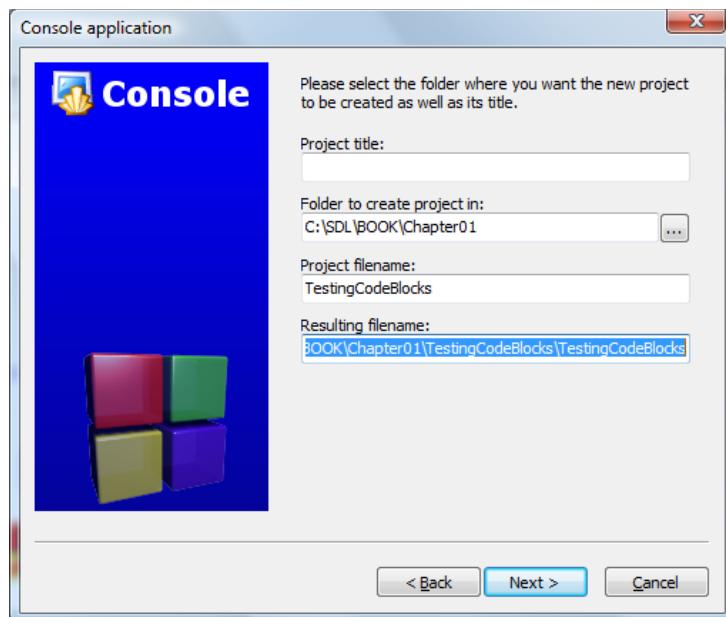


Figure 25 - TestingCodeBlocks Project

I decided to give my project title the name “TestingCodeBlocks”. Select a folder to insert the new project. I usually create a folder with the same name as the project.

4. Take the defaults and click on “Finish”.

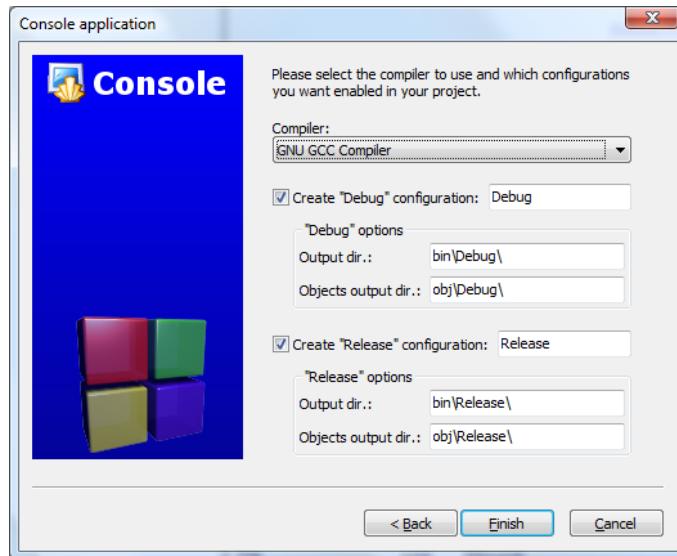


Figure 26 - Creating Console Applications

5. In order to locate the SDL header files you will need to locate
6. Set up your project's specific libraries. Select “Project”, choose “Build options”, select the main project (TestingCodeBlocks). Click on the “Linker settings” tab. Click on “Add” button. Add mingw32 first followed by the two SDL libraries. Add “-lmingw32 -lSDLmain -lSDL” to the “Other linker options”. Click on OK

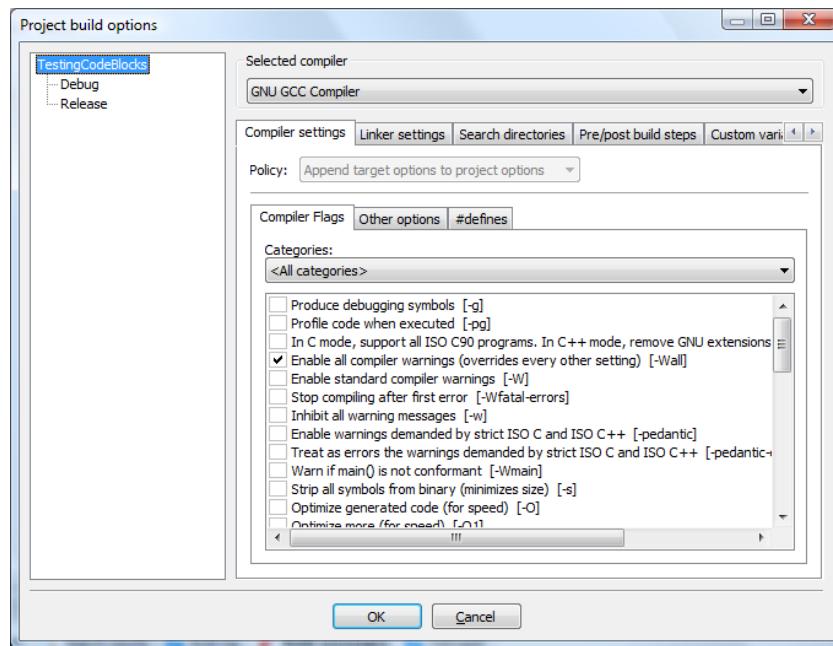


Figure 27 - Selecting the main "TestingCodeBlocks"

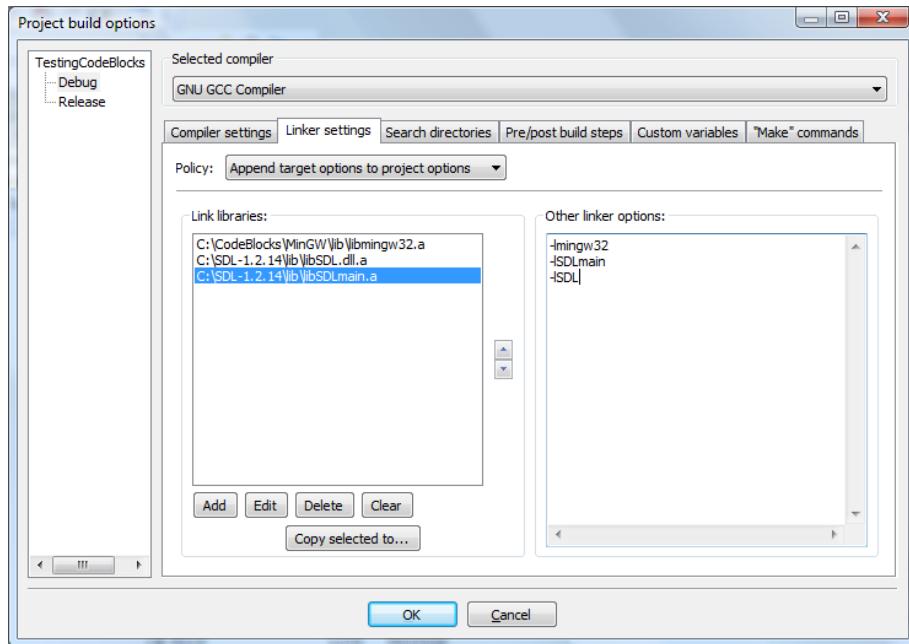


Figure 28 - Adding linker options

7. Replace the code in the default main.cpp with the sample SDL code.

Table 1 - TestingCodeBlocks (main.cpp)

```
#include <cstdlib>
#include <iostream>
#include "SDL\SDL.h"

using namespace std;
    // screen dimensions
const int SCREEN_WIDTH=640;
const int SCREEN_HEIGHT=480;

//display surface
SDL_Surface* pDisplaySurface = NULL;

//event structure
SDL_Event event;

int main(int argc, char *argv[])
{
    //initialize SDL
    if (SDL_Init(SDL_INIT_VIDEO)==-1) {
        cerr << "Could not initialize SDL!" << SDL_GetError() << endl;
        exit(1);
    }
}
```

```

} else {
    cout << "SDL initialized properly!" << endl;
}
//create windowed environment
pDisplaySurface =
    SDL_SetVideoMode(SCREEN_WIDTH, SCREEN_HEIGHT, 0, SDL_ANYFORMAT);

//error check
if (pDisplaySurface == NULL) {
    //report error
    cerr << "Could not set up display surface!" << SDL_GetError()
        << endl;
    exit(1);
}

// set caption
SDL_WM_SetCaption("Template", NULL);

//repeat forever
for(;;)
{
    //wait for an event
    if(SDL_PollEvent(&event)==0) {
        // DO OUR THING . . .

        //update the screen
        SDL_UpdateRect(pDisplaySurface,0,0,0,0);
    } else {
        //event occurred, check for quit
        if(event.type==SDL_QUIT) break;
    }
}
SDL_FreeSurface(pDisplaySurface);
SDL_Quit();
//normal termination
cout << "Terminating normally." << endl;
return EXIT_SUCCESS;
}

```

If you try to build the program (Build | Build) you will see many errors. The main reason is the inability to find the file “SDL\SDL.h”.

8. Open Project | Build Options and click on “Search directories” tab. Click on “Add” and navigate to “C:\SDL-1.2.14\include”. Click OK

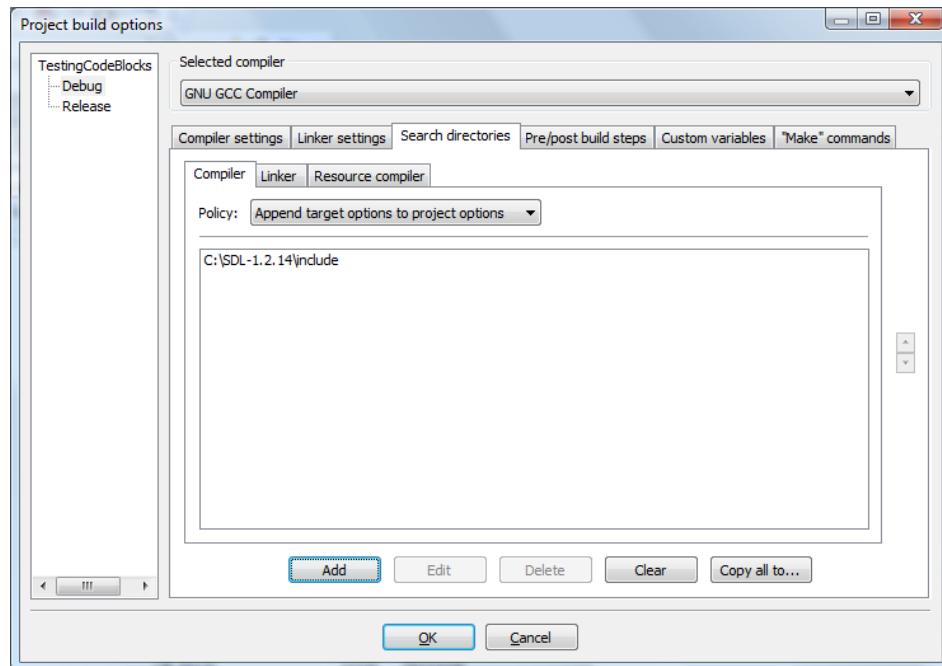


Figure 29 - Adding SDL include file to compiler "search directory" list

If you try to build the program again it will fail with the message “...cannot find –lSDLmain”. In the same screen as shown in Figure 29 click on “Linker” sub-tab and navigate to SDL\lib directory.

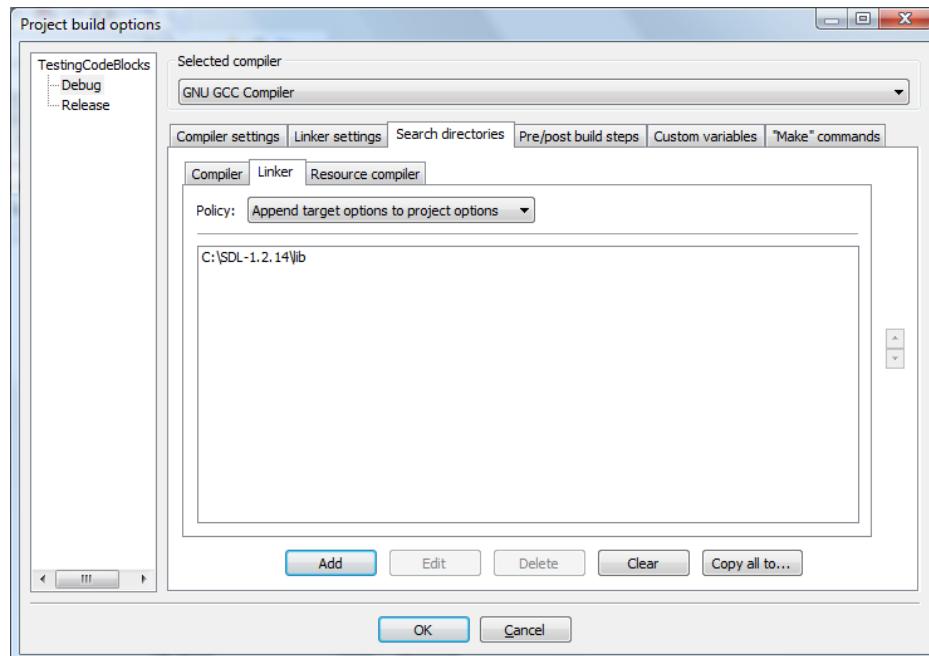


Figure 30 - Add SDL lib to Linker "search directory"

The program should now successfully build with the Build log showing:

```
----- Build: Debug in TestingCodeBlocks -----
```

```
Linking console executable: bin\Debug\TestingCodeBlocks.exe
Output size is 831.02 KB
Process terminated with status 0 (0 minutes, 0 seconds)
0 errors, 0 warnings
```

The program is a very simple SDL program that:

- ✚ Initializes SDL for video
- ✚ Creates a window
- ✚ The for loop constantly looks for events but the only one it is handling is the event that gets generated when you close the window or when you quit (SDL_QUIT) the application window.

8. Compile and execute

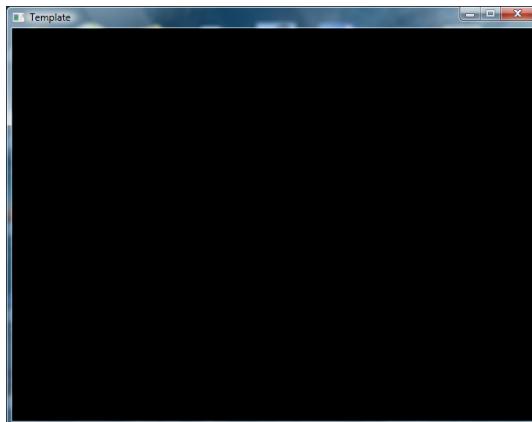


Figure 31 - Test SDL Program Window

If you followed the directions you should see a window similar to the above. If you see the following message:

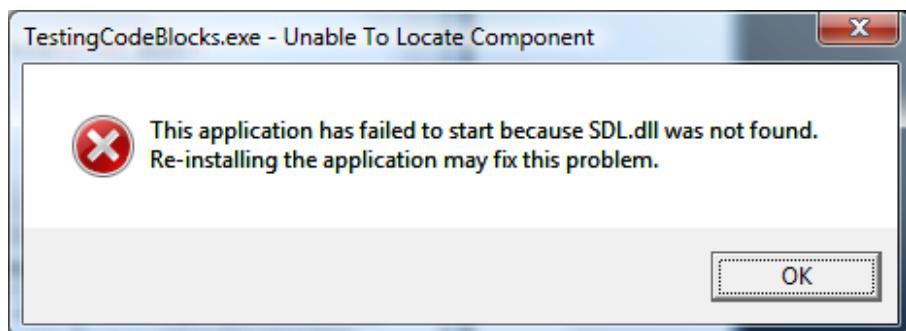


Figure 32 - Execution error

Copy the file C:\SDL-1.2.14\bin\SDL.dll into C:\Windows\ you

What is a wizard?

Many applications have wizards, which are programming aids implemented as a set of dialog boxes that are used to automate things for you. For example, when you create programs using SDL you will find that unless you use a wizard you will always need to add the same set of include files, link the same libraries and dlls. Using a wizard to create a project specifically for SDL will get all these mundane tasks done for you at the start. In addition, you can specify a start-up main file to be used as the template for all your SDL programs. No need to add main and all the startup SDL code – it will be all done by the wizard!



Using the Code::Blocks SDL wizard

- Create a Project and select “SDL Project”

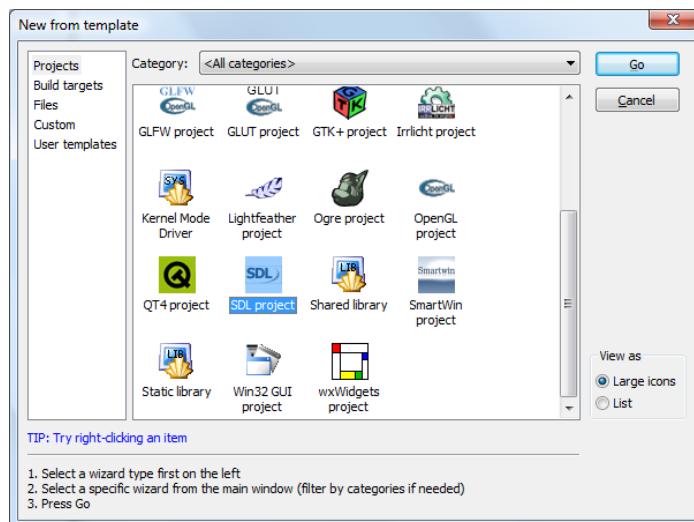


Figure 33 - Creating an SDL project

There seems to be a problem with the Code::Blocks wizard on Windows. So I will skip using it. Follow the following instructions to create your own user template.

Creating your own SDL Custom Template

- Open the last project (if it is not open).

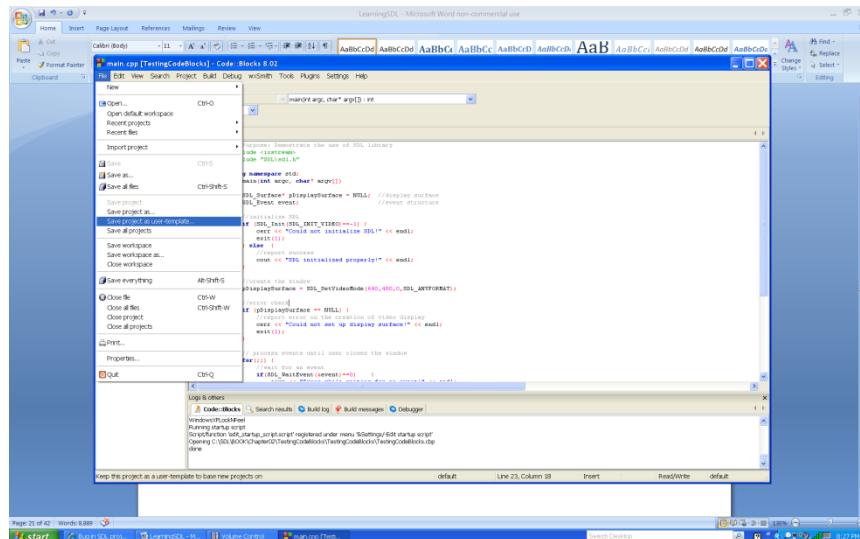


Figure 34 - Save Project as User-Template

- Select File | Save Project as User-Template

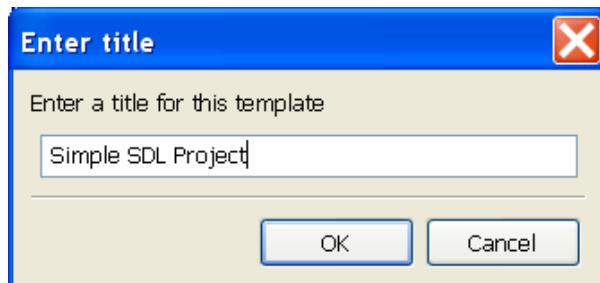


Figure 35 - Giving user-template a name

Now when you start a Simple SDL Project (something we will do for all our non-game programs) you can just start the project by selecting: File | New | From user template and select “Simple SDL Project”. This will create a starting project for a simple SDL Project. In chapter TBD we will create another user-template for our games. Click OK to create your own SDL Project.

How do we use this template?

Now when you create a new SDL program by selecting File | New Project. Select User templates and select “Simple SDL Project”.

Installing to work with WxDev-C++

- Remove any previous copy of Dev-C++.

2. Obtain and install the latest version of wxDev-C++ from <http://wxdsgn.sourceforge.net/>.
 - a. Note: I installed under the directory C:\Dev-Cpp. You can choose to save under the default directory under C:\Program Files.
 - b. Install the version wxdevcpp_7.2.0.2_full_setup
- Double click on the setup program

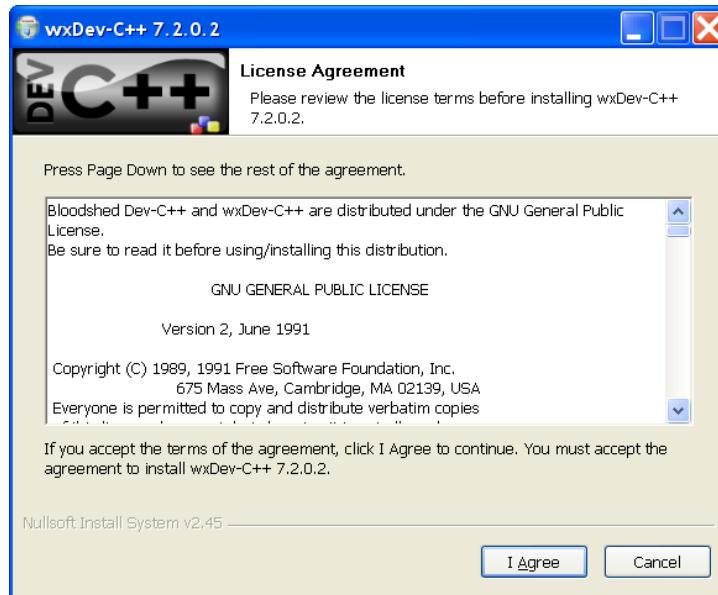


Figure 36 - WxDev-C++ License Agreement

The starting point for this IDE is the popular and free IDE Bloodshed Dev-C++.

- Click on “I Agree”

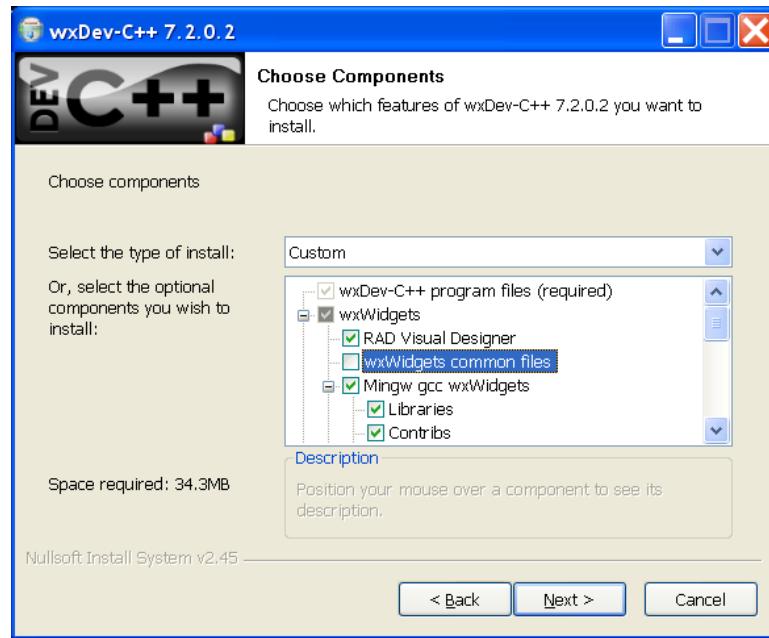


Figure 37 - Component selection screen

- Select the components that make sense. I suggest you take the defaults.

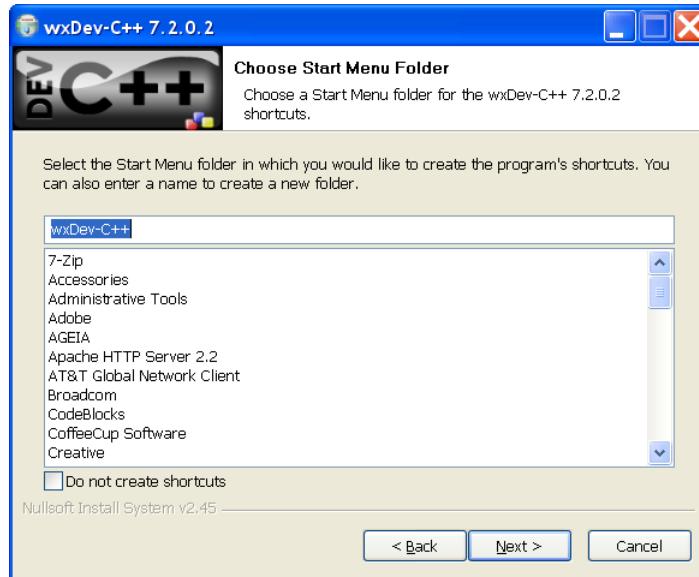


Figure 38 - Start Menu Folder name dialog box

- Click Next> to accept the default start menu folder name

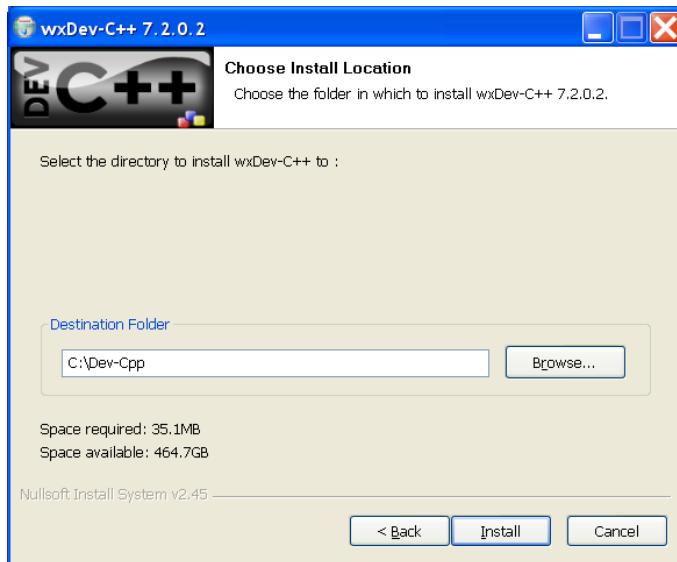


Figure 39 - Install location dialog box

- + I installed under C:\Dev-Cpp instead of the default C:\Program Files\Dev-Cpp due to problems that came up under some Windows Vista installations. I suggest you take the default if you are sure you will not encounter similar problems. Click on “Install”

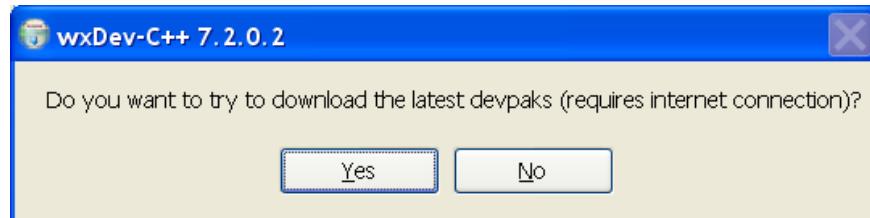


Figure 40 - message box prompt to obtain latest devpaks

- + If you prompted to download the latest devpaks, click on “Yes”.

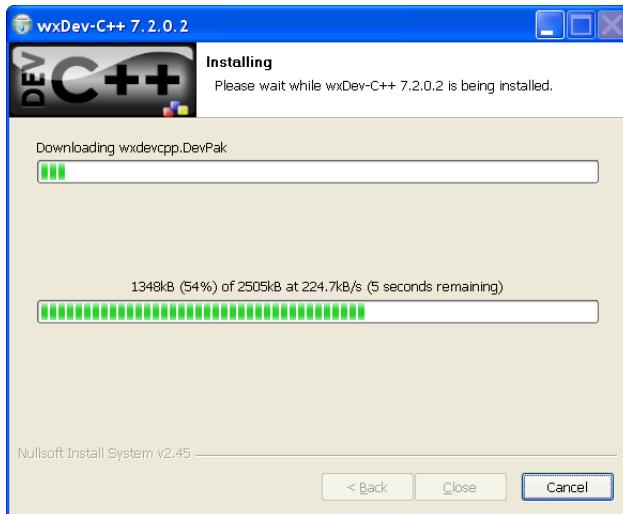


Figure 41 - Installation progress bar

- Click “Close” once the installation completes



Figure 42 - Install for all users dialog box

- Click “Yes”

- Copy the directory folder C:\SDL-1.2.14\include\SDL to a new \SDL include directory under C:\Dev-Cpp\include. This will make it easy for the compiler to locate the SDL header files.

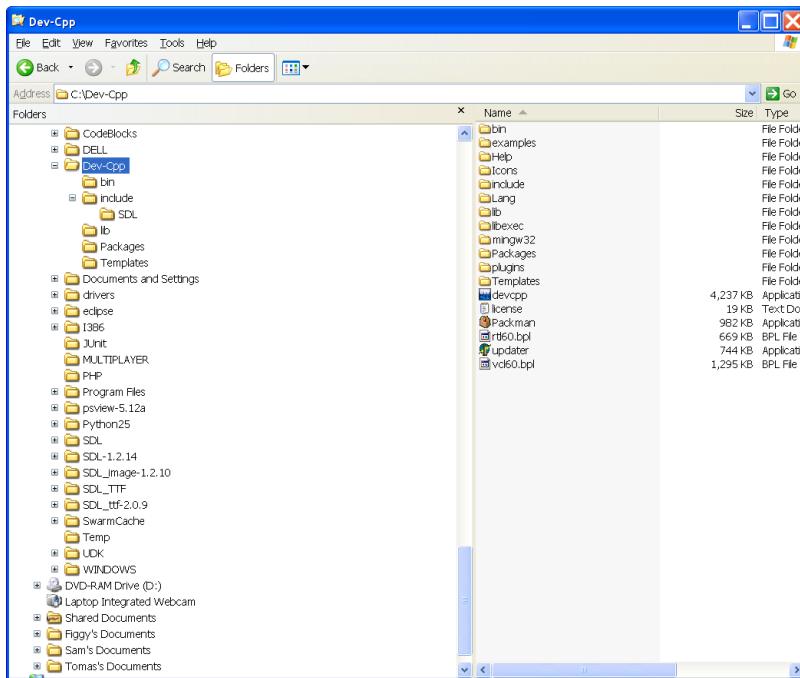


Figure 43- The result of copying /include/SDL under Dev-Cpp include directory

3. Copy the *.a files (libSDL.dll.a, libSDLmain.a) in C:\SDL-1.2.14\lib to C:\Dev-Cpp\lib directory.
4. Copy SDL.dll under C:\SDL-1.2.14\bin to Dev-Cpp bin directory.
 - a. This will allow the IDE to find the SDL.dll when you execute from the IDE, otherwise you will always have to copy the SDL.dll into the same directory as the exe file.
5. Open WxDev-C++. The next couple of screen only occur when you start wxDev-C++ for the first time.

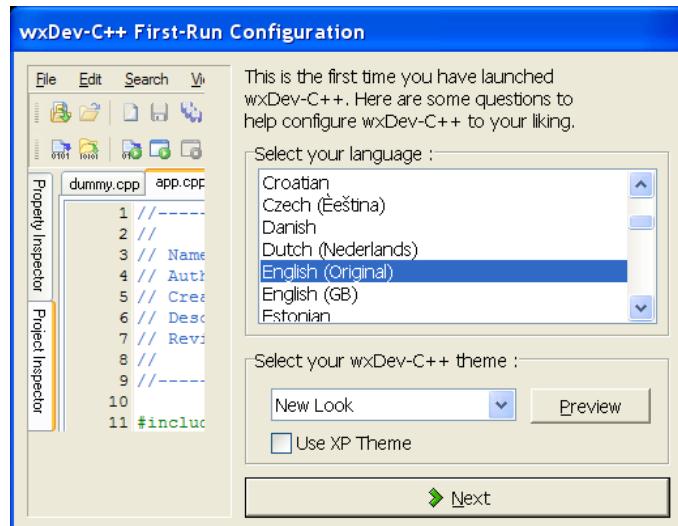


Figure 44 - Selecting language dialog box

⊕ Select the language and click > Next.

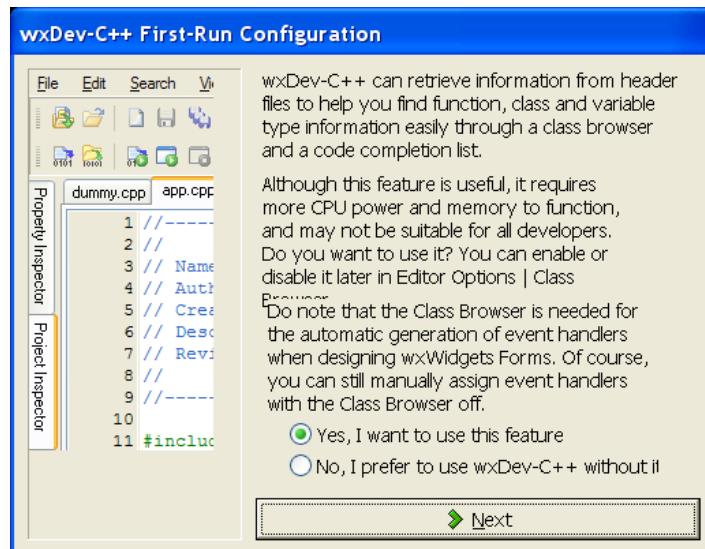


Figure 45 - Using a cool feature

⊕ Click > Next

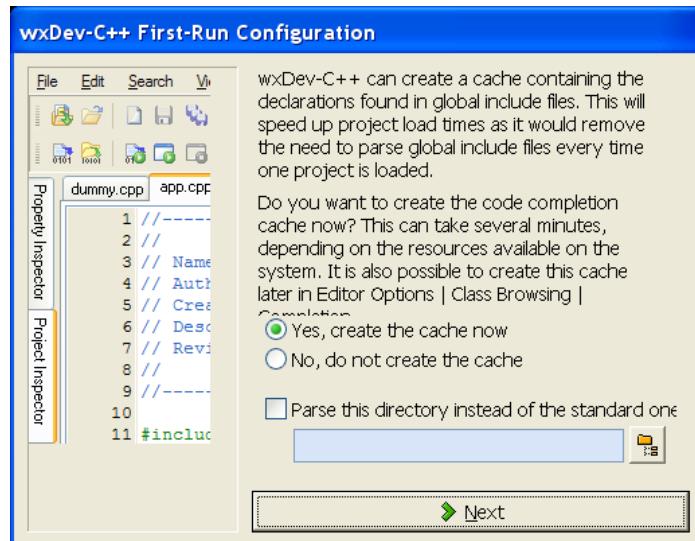


Figure 46 - Creating a cache dialog box

- Click > Next. It will take a while to parse the files.

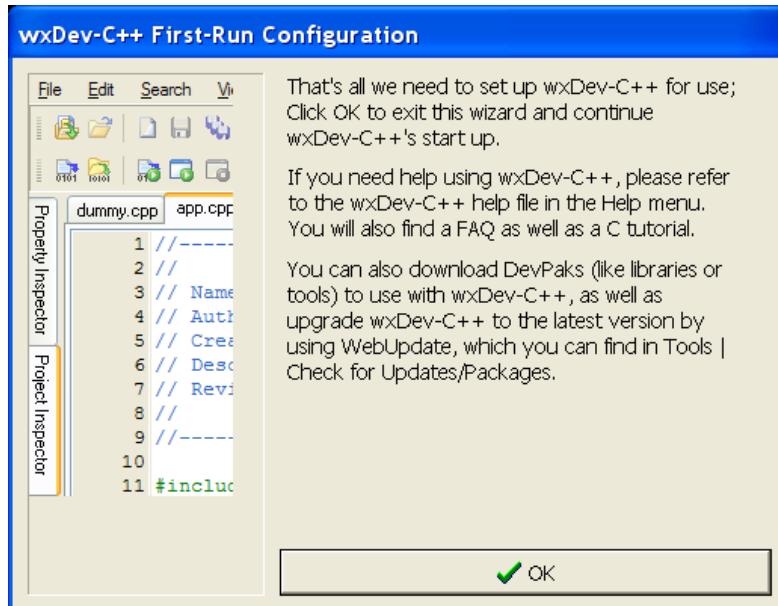


Figure 47 - Final installation dialog box

- Click OK

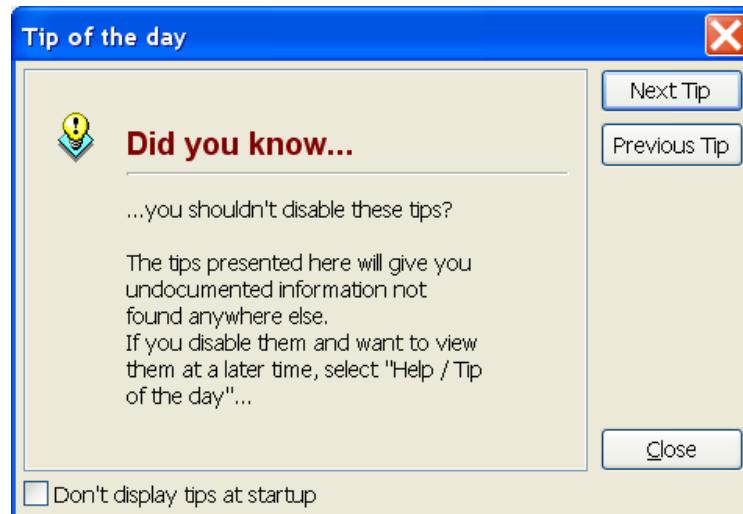


Figure 48 - Your first "tip of the day"

-  I always read the “Tip of the day” in order to learn something new about the application I am using.

6. Open a new Project by selecting File | New | Project

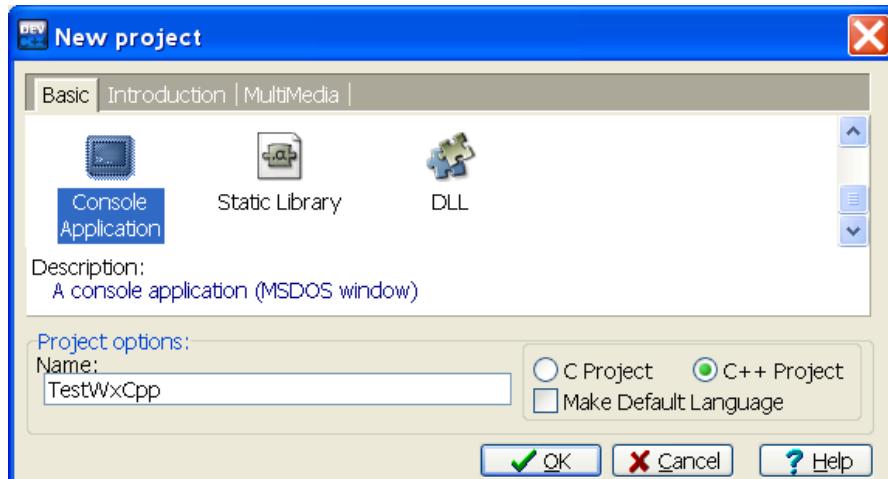


Figure 49 - New Project dialog box

7. Select “Console Application” from the list project type icons and enter a name for your Project.
8. Copy the program in **Error! Reference source not found.** into the default main.cpp program that was created for you when you selected Console program.
9. Add the following under: Project | Project Options. Select Parameters tab: In

```
-mwindows
-lmingw32
-lSDLmain
-SDL
C:\Dev-Cpp\lib\libmingw32.a
C:\Dev-Cpp\lib\libSDL.dll.a
C:\Dev-Cpp\lib\libSDLmain.a
```

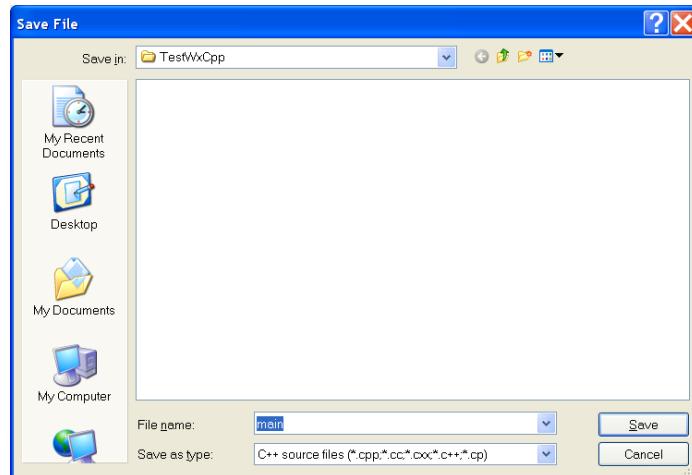


Figure 50 - Saving the "main.cpp" file

- ⊕ Save the program file main.cpp

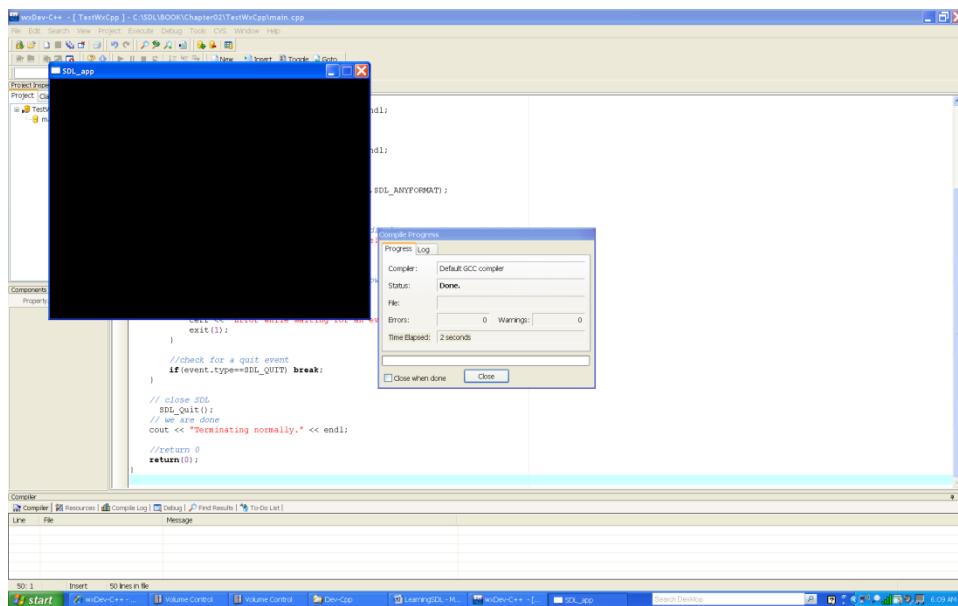


Figure 51 - Result of compiling and running the program

10. Compile and Run.

Creating a WxDev-Cpp SDL template file

You can easily create a WxDev-Cpp SDL template file to use.

- ✚ Open up the last project if it is not already opened.
- ✚ Select File | New and select Template

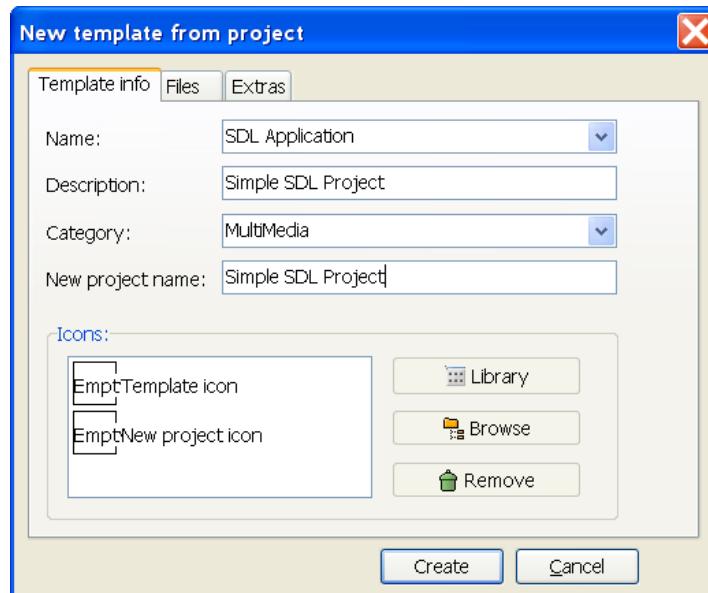


Figure 52 - New Template dialog box

- ✚ Click on the “Empty Template Icon” and then click Library

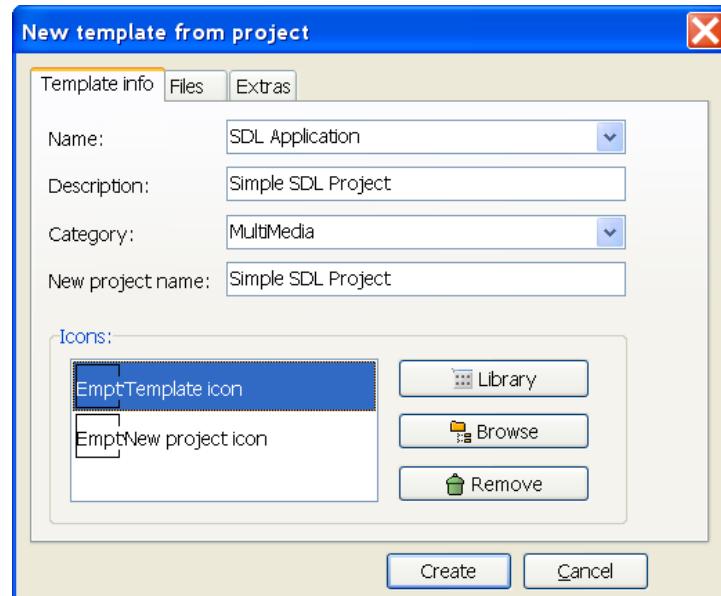


Figure 53 - Selecting template Icon

- ✚ Double click on any icon you want.
- ✚ Click on Empty New Project icon and select any icon (the same one?)

If you want more of a selection you can go online and obtain free icons to use.

- ✚ Click “Create”

You can now easily create Simple SDL Project by selecting it from the Multimedia tab when you start with File | New | Project

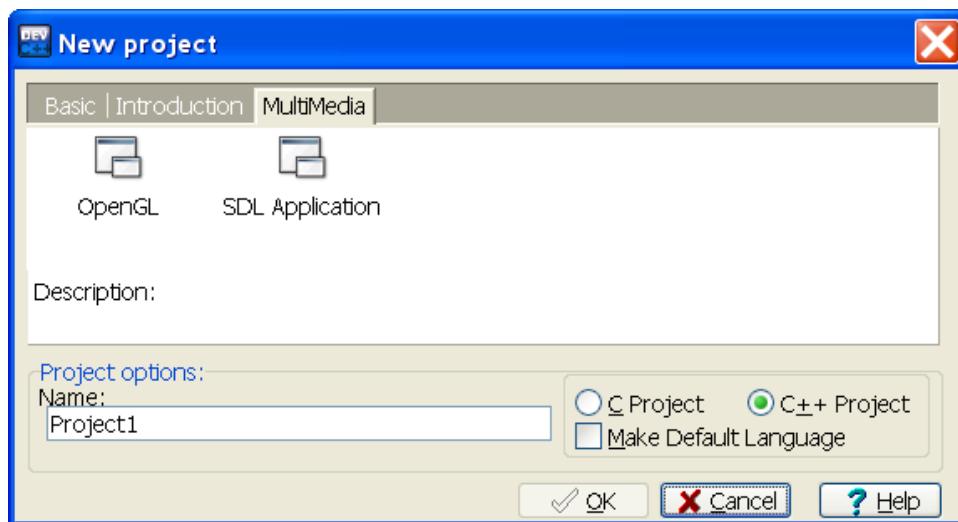


Figure 54 - Creating a Simple SDL Project

Which IDE should you use? The choice is all yours. All the code samples and screens will be from the IDE Wx-Dev++. The instructions for setting it up to use the additional libraries will be quite similar.

Installing and Testing SDL_Image

The next library to setup and test is SDL_Image.

- ⊕ Copy the file in the <SDL_Image Directory>\include\SDL_image.h to the <IDE Directory>\include\SDL directory. For me the above required that I move C:\SDL_image-1.2.10\include to C:\Dev-Cpp\include\SDL.
- ⊕ Copy the files in <SDL_Image Directory>\lib*.dll to <IDE Directory>\bin
- ⊕ Copy the file in <SDL_Image Directory>\lib\SDL_image.lib (the object file library) to <IDE Directory>\lib

To test that you can read PNG or JPEG files find some files online by going to <http://www.bing.com> and searching for an image. I found a wallpaper of Betty Boop.

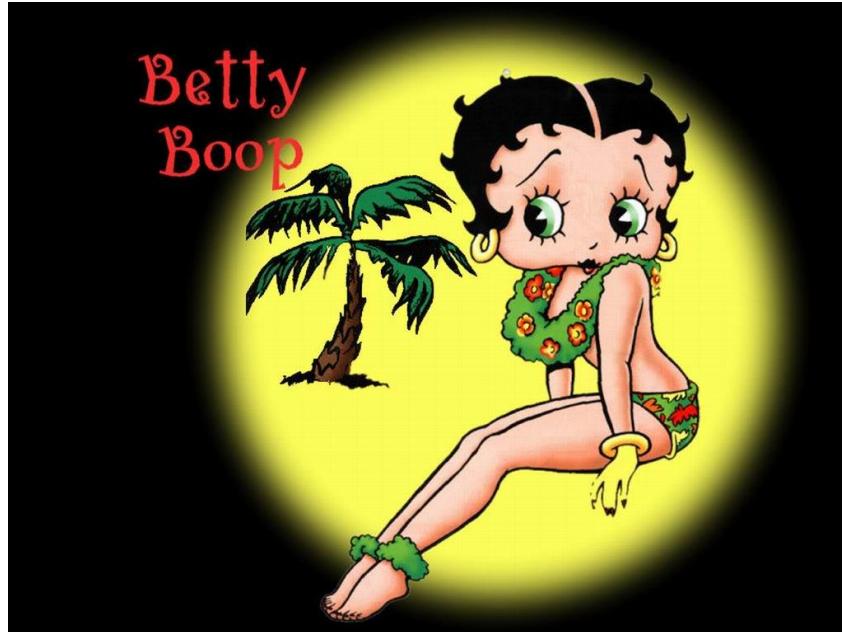


Figure 55 - My Betty Boop Wallpaper

- ⊕ Create an SDL Project “TestSDLImage”
- ⊕ Since my image is 1024 x 768 I will create a window large enough to accommodate this test image.
- ⊕ Copy the code below as your main.

Table 2 – TestSDLImage Project

```
// Purpose: Demonstrate the use of SDL Image library
#include <iostream>
```

```
#include "SDL\ sdl.h"
#include "SDL\SDL_image.h"

using namespace std;
int main(int argc, char* argv[])
{
    SDL_Surface* pDisplaySurface = NULL; //display surface
    SDL_Event event; //event structure

    //initialize SDL
    if (SDL_Init(SDL_INIT_VIDEO)==-1) {
        cerr << "Could not initialize SDL!" << endl;
        exit(1);
    } else {
        //report success
        cout << "SDL initialized properly!" << endl;
    }

    //create the window
    pDisplaySurface = SDL_SetVideoMode(1024,768,0,SDL_ANYFORMAT);

    //error check
    if (pDisplaySurface == NULL) {
        //report error on the creation of video display
        cerr << "Could not set up display surface!" << endl;
        exit(1);
    }
    // Read in the image
    SDL_Surface* pJpegimage = IMG_Load("wallpaperBettyBoop.jpg");
    if (pJpegimage == NULL) {
        // report error trying to read in image file
        cerr << "Could not read image file" << endl;
        exit(1);
    }
    // Get image ready for display on the screen
    SDL_Surface* pDisplayFormat = SDL_DisplayFormat(pJpegimage);
    // show on display screen
    SDL_Rect DestR;
    DestR.x = 0;
    DestR.y = 0;
    SDL_BlitSurface(pDisplayFormat, NULL, pDisplaySurface, &DestR);

    // process events until user closes the window
    for(;;)
    {
        //wait for an event
        if(SDL_WaitEvent(&event)==0) {
            cerr << "Error while waiting for an event!" << endl;
            exit(1);
        }
        //check for a quit event
        if(event.type==SDL_QUIT) break;
        //update the screen
        SDL_UpdateRect(pDisplaySurface,0,0,0,0);
    }
    // unload the dynamically loaded image libraries
    SDL_FreeSurface(pJpegimage);
    IMG_Quit();
}
```

```
// close SDL
SDL_Quit();
// we are done
cout << "Terminating normally." << endl;

//return 0
return(0);
}
```

- >Add the library `SDL_image.lib` to the linker parameters

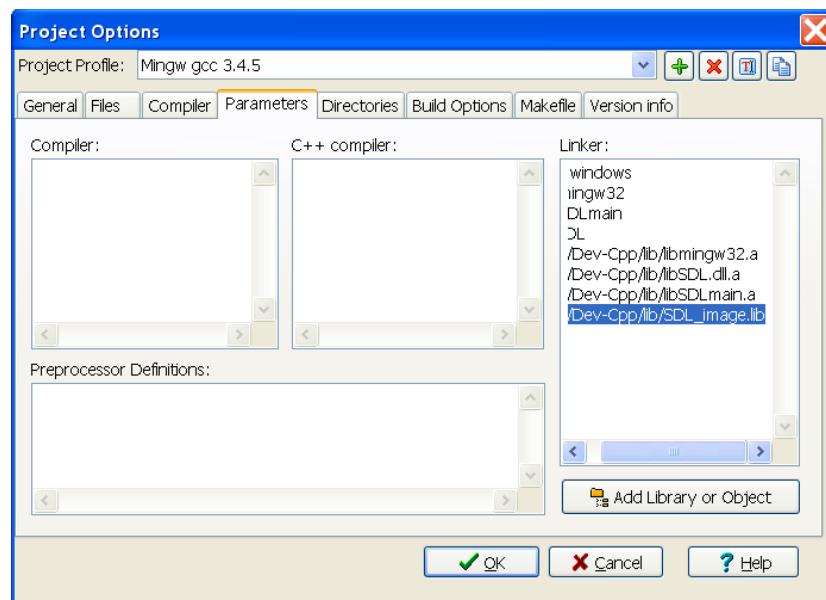


Figure 56 - Add `SDL_image` to linker parameters

We will explain how the program works later. The new code has been highlighted. If something does not work for you, I recommend you go and check the directory where the exe resides and check for an `stderr.txt` file. That file will contain any error messages if the program had problems locating your image file.

Figure 57 shows the result of executing the program.

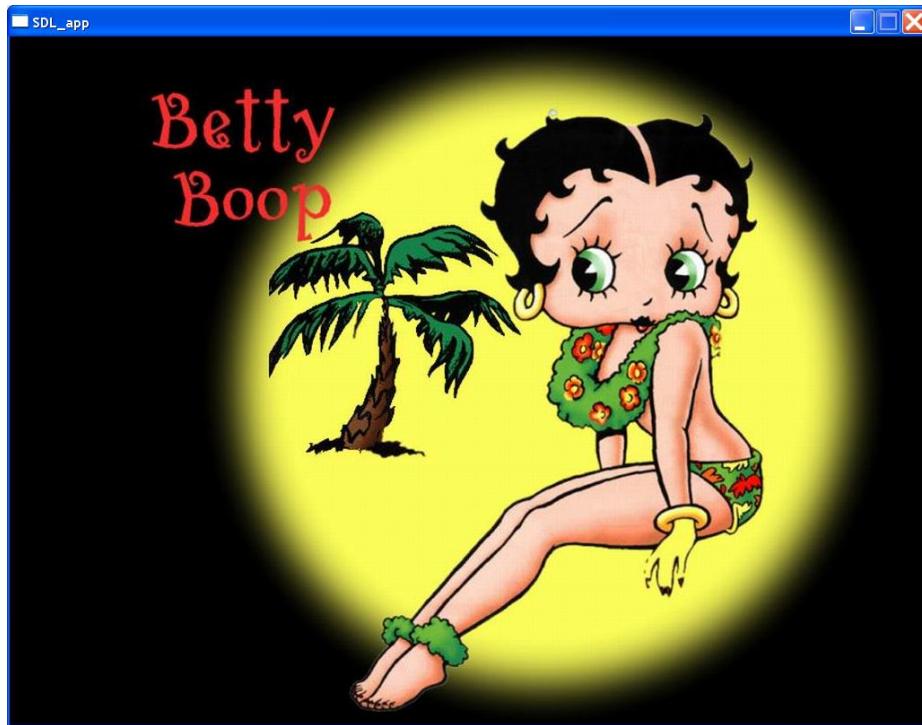


Figure 57 – Testing SDL_Image library

Installing and Testing SDL_tff

The next library to setup and test is SDL_tff.

- ⊕ Copy the file in the <SDL_tff Directory>\include\SDL_tff.h to the <IDE Directory>\include\SDL directory. For me the above required that I move C:\SDL_tff-2.0.9\include to C:\Dev-Cpp\include\SDL.
- ⊕ Copy the files in <SDL_tff Directory>\lib*.dll to <IDE Directory>\bin
- ⊕ Copy the file in <SDL_tff Directory>\lib\SDL_ttf.lib (the object file library) to <IDE Directory>\lib

I will test installation of SDL_tff by adding code to my previous program to display some text at the bottom left corner of the Betty Boop image. I will need to download a true type font for my program to use. You can go online and get a free font libraries or just copy the Arial true type font I used.

- ⊕ Create a new project “TestSDLTFF” using the main.cpp below. Copy the image file and font file to the directory where the exe will reside

Table 3 - TestSDLTff

```
// Purpose: Demonstrate the use of SDL Image library
#include <iostream>
#include "SDL\ sdl.h"
#include "SDL\SDL_image.h"
#include "SDL\SDL_ttf.h"

using namespace std;
int main(int argc, char* argv[])
{
    // Your code here
}
```

```
{
    SDL_Surface* pDisplaySurface = NULL; //display surface
    SDL_Event event; //event structure

    //initialize SDL
    if (SDL_Init(SDL_INIT_VIDEO)==-1) {
        cerr << "Could not initialize SDL!" << endl;
        exit(1);
    }

    //initialize SDL_ttf
    if(TTF_Init() == -1) {
        cerr << "TTF_Init: " << TTF_GetError() << endl;
        exit(2);
    }

    //create the window
    pDisplaySurface = SDL_SetVideoMode(1024,768,0,SDL_ANYFORMAT);

    //error check
    if (pDisplaySurface == NULL) {
        //report error on the creation of video display
        cerr << "Could not set up display surface!" << endl;
        exit(1);
    }

    // load font.ttf at size 16 into font
    TTF_Font *pfont;
    pfont=TTF_OpenFont("ARIAL.ttf", 24);
    if(!pfont) {
        cerr << "TTF_OpenFont: " << TTF_GetError() << endl;
        // handle error
    }
    // let's create white text
    SDL_Color color={255,255,255};
    SDL_Surface *ptext_surface = NULL;
    ptext_surface=TTF_RenderText_Solid(pfont,"Pass the Mojito!",color) ;
    if(ptext_surface == NULL) {
        //handle error here, perhaps print TTF_GetError at least
        cerr << "Could not create text_surface error: "
            << TTF_GetError() << endl;
        exit(3);
    }
    // Read in the image
    SDL_Surface* pJpegimage = IMG_Load("wallpaperBettyBoop.jpg");
    if (pJpegimage == NULL) {
        // report error trying to read in image file
        cerr << "Could not read image file" << endl;
        exit(1);
    }
    // Get image ready for display on the screen
    SDL_Surface* pDisplayFormat = SDL_DisplayFormat(pJpegimage);
    // show on display screen
    SDL_Rect DestR;
    DestR.x = 0;
    DestR.y = 0;
    SDL_BlitSurface(pDisplayFormat, NULL, pDisplaySurface, &DestR);
}
```

```

// print the message
DestR.y = 600;
DestR.x = 100;
SDL_BlitSurface(ptext_surface,NULL,pDisplaySurface,&DestR);
// process events until user closes the window
for(;;)
{
    //wait for an event
    if(SDL_WaitEvent(&event)==0) {
        cerr << "Error while waiting for an event!" << endl;
        exit(1);
    }
    //check for a quit event
    if(event.type==SDL_QUIT) break;
    //update the screen
    SDL_UpdateRect(pDisplaySurface,0,0,0,0);
} // end for
// free text message

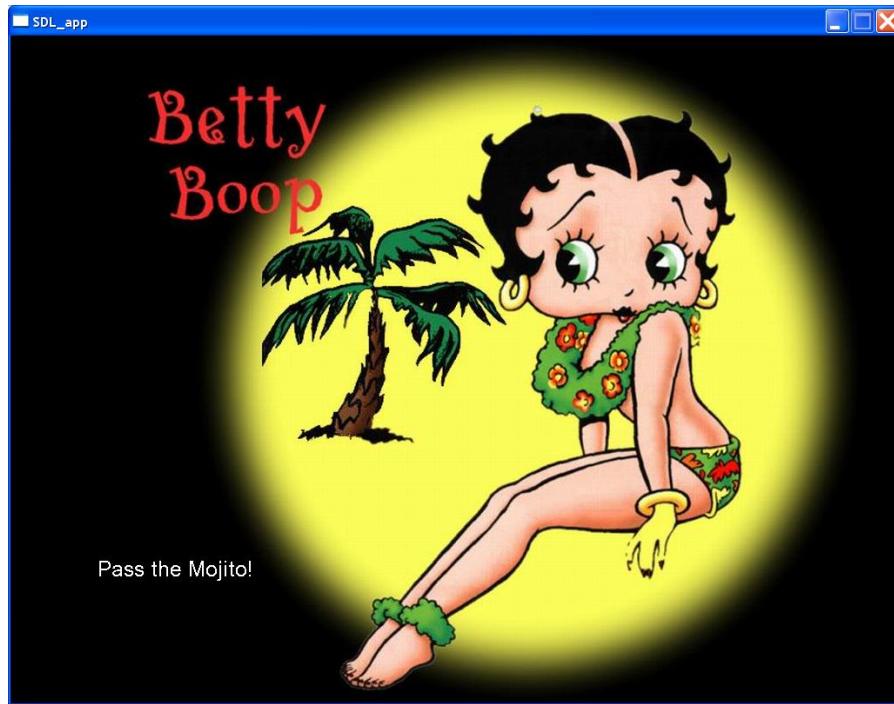
// unload the dynamically loaded image libraries
SDL_FreeSurface(pJpegimage);
IMG_Quit();
// free text message
SDL_FreeSurface(ptext_surface);
// free the font
TTF_CloseFont(pfont);
pfont=NULL; // to be safe...
// close TTF
TTF_Quit();
// close SDL
SDL_Quit();
// we are done
cout << "Terminating normally." << endl;

//return 0
return(0);
}

```

- ✚ Add the library `SDL_ttf.lib` to Project | Project Options, Parameters tab.
- ✚ Compile and Run.

The results:



[Figure 58 - Testing SDL_ttf library](#)

Installing and Testing SDL_mixer

Obtain the latest copy of SDL_mixer from http://www.libsdl.org/projects/SDL_mixer. Install the version for your system. For window users just unpack the windows file (SDL_mixer-devel-1.2.11-VC) to the C:\ drive (e.g. C:\SDL_mixer-1.2.11).

The extraction creates both an \include and \lib directory. We will just add these directories to the compile and link path. Copy the *.dll's into the bin directory, SDL_mixer object file library into the IDE's lib diretory and SDL_mixer.h into the IDE's \include\SDL directory.

- ⊕ To test create an SDL project named TestSDL_mixer
- ⊕ Copy the code below into your new main.cpp
- ⊕ Move the file Halo.mp3 and Halo.jpg to the directory containing this project

...More information on installing SDL Libraries

I discovered a wonderful website with tons of information on SDL. The page http://lazyfoo.net/SDL_tutorials/lesson03/index.php explains how to install the correct version of an SDL library for Windows, Linux and Mac OS X. For Windows Dev C++ you should download the following files:

- SDL_image_develop-1.2.4-VC6.zip (or something similar for the target library)

When you unzip the file you will get the following three parts:

- Header files (usually under /include)
- Lib files (usually under /lib)

- The *.dll files

I usually extract each library under the C: directory with a corresponding name.

The notes on this web site recommends the following steps:

- Copy the *.h files under the same directory you maintain the SDL header files (for me that under the development directory e.g. C:\Dev-Cpp\include\SDL)
- Copy the lib file (e.g. SDL_mixer.lib) to the lib directory you keep the SDL libs (again, I just copy what I need to C:\Dev-Cpp\lib)
- Copy the *.dll files to the same directory where you exe will be. (Since I usually test while in Dev-Cpp I just move the files to C:\Dev-Cpp\bin but this will not allow the program to run outside of Dev-Cpp). Note: You can always move the *.dll to C:\Windows to run from any directory.
- Under the Project | Project Options | Parameters tab enter –lmingw32 –lSDLmain –lSDL
- Add –lSDL_tff if you are using that library in your program
- Add –lSDL_mixer if you are using that library in your program

Installing and Testing SDL_net

TBD

Summary

At this point you should have selected one IDE to use to compile and build your SDL programs. I will be using wxDev-Cpp throughout these notes. If you want to set up the free Microsoft compiler Microsoft Visual C++ 2010 Express please see Appendix B. I personally did not like CODE::BLOCKS since it displayed problems in the editing window that proved irritating and troublesome. I do like Microsoft C++ IDE but you will only find cross-platform versions of the two IDEs highlighted in this chapter. You are not expected to understand any of the code you used to test the installation. I recommend you come back and take a look at the code after you have completed Chapter 2.

Chapter 2 - Getting started with SDL

This chapter presents an initial overview of the video component of SDL. We will cover:

- How to initialize SDL and the video subsystem
- How to properly release SDL resources
- How the video display is organized
- How to output pixels, draw lines, circles
- How to draw rectangles
- How to load and display bmp and other images
- How to plot pixels and lines
- How to perform simple animation

Initializing SDL

An SDL application must first use the function `SDL_Init()` to initialize the library.

Function Name: `SDL_Init()`

Format:

```
int SDL_Init(Uint32 flags);
```

Description:

The function accepts a `Uint32` which is an unsigned 32-bit value that represents a one or more pieces of information shown in Table 5. On success the function returns 0 otherwise -1.

How do we represent one or more pieces of information in one variable?

"In the early days of sailing, before there were radios and cell phones, communication at anything more than the distance a man could shout was accomplished by signal flags. Each navy and most merchant fleets had their own code of flags and often the codes changed on a regular basis. The flags could be used to deliver messages, direct fleet operations and exchange information about weather and other conditions. By the early 1880s, the British merchant fleet was able to send more than 70,000 different messages, using only 18 flags. "¹⁰



Figure 59 - The use of flags in the Navy

As programmers we use boolean variables to convey or indicate if something is true or false (e.g. `isEOF11`, `isGameOver`) or if something is on or off or there or not (e.g. `hasUserFlashLight`). A bool data type is used as in

¹⁰ http://madmariner.com/seamanship/piloting/story/SIGNAL_FLAGS_010510_SP

¹¹ EOF stands for END OF FILE, `isEOF` is a Boolean that when true means we have reached the end of the file

```
bool hasUserFlashLight = false;
```

One characteristic of the programming spirit is to try to maintain information in a concise and compact form. It is not uncommon to use an unsigned byte to represent several pieces of information of on-off values. The way it works is to use bits of the byte (or larger) to represent different information or in other words to act as “flags.” For example, suppose we were designing an application that will display a window and you wanted to capture the different styles you are willing to support in a byte value. You could set things up in the following manner:

Table 4 – Sample flags for windows

Style meaning	Value
WIN_BORDER – window has a border	Bit 0 is 1 or 00000001
WIN_CAPTION – window has caption/title	Bit 1 is 1 or 00000010
WIN_RESIZE – window can be resized	Bit 2 is 1 or 00000100
WIN_MENU – window has a system menu	Bit 3 is 1 or 00001000

The real advantage of using this method is that the values you are maintaining are similar (all relate to the same object or function – a window) and yet all are mutually exclusive, that is, one or more can be “on” at the same time. So if someone wanted to create a window with a border and caption the byte value would hold 00000011. To make setting and checking the byte value you can create the following #define statements:

```
#define WIN_BORDER      0x01
#define WIN_CAPTION     0x02
#define WIN_RESIZE       0x04
#define WIN_MENU         0x08
```

You can then ‘or’ or | a combination of these values when setting a window style for example:

WIN_BORDER | WIN_RESIZE will set the value to 00000101 which will indicate that we want the window to have a border and be re-sizable. It is also easy to determine if a particular flag is set or not by just using the ‘and’ & operation. For example:

```
if ( windowFlag & WIN_RESIZE) {
    // Re-size the window
}
```

If the user did not create the window for re-sizing (suppose they selected only WIN_BORDER and WIN_CAPTION) then the ‘&’ would be 00000011 & 00000100 would create the value 0 which will then bypass the ifstatement. Isn’t this cool!

So when you see a function argument referred to as a flag and you can set one or more values you now know that it is set up as a bits representing one of more mutually exclusive values.

The other fact to note is the use in SDL of the following data types:

Sint8 - signed 8-bit integer

Uint8 - unsigned 8-bit integer

Sint16 - signed 16-bit integer

I think you can guess at what Uint16, Sint32, and Uint32 mean. Using these variables hides the differences between machines and platforms. In addition, SDL hides issues around byte-order or big-endian and little-endian.

What is big-endian and little-endian?¹²

The terms big-endian and little-endian “introduced in 1980 by Danny Cohen in his paper ‘On Holy Wars and Plea for Peace’” In the paper Cohen references the classic novel Gulliver’s Travels to get the terms big-endian and little-endian. In the novel there is a “satirical conflicts ..between two religious sects .. some of whom prefer cracking their soft-boiled eggs from the little end, while others prefer the big end. Most computer processors represent numbers the same way inside the CPU. For example, the number 10,000 represented as a 32-bit number will appear as:

00000000 00000000 00100111 00010000

If a machine expects the integer value to be stored in memory where the “increasing numeric significance with increasing memory addresses” or as

MEMORY_ADDRESS:	100	102	103	104
INTEGER_VALUE:	00010000	00100111	00000000	00000000

The above is known as little-endian.

Little-endian was used by x86, 6502, Z80 processors (used by Intel PC based-machines, Apple II, Radio Shack TRS-80, respectively)

“It’s opposite, most-significant byte first” is called big-endian.

MEMORY_ADDRESS:	100	102	103	104
INTEGER_VALUE:	00000000	00000000	00100111	00010000

Big-endian was used by many Motorola processors (6800, 68000, and PowerPC) used by Macintosh machines before the switch to the x86 family of processors).

In order for SDL to be cross-platform it must support and successfully hide from developers this big difference between machines.

¹² This part references <http://en.wikipedia.org/wiki/Endianness>



Returning back to the `SDL_Init` function the first argument of that function flags is used to indicate which of the SDL subsystems to initialize. The table below lists all the possible flags that can be used alone or in combination when invoking `SDL_Init`.

Table 5 - `SDL_Init` initialization flags

<code>#define SDL_INIT_TIMER</code>	<code>0x00000001</code>
<code>#define SDL_INIT_AUDIO</code>	<code>0x00000010</code>
<code>#define SDL_INIT_VIDEO</code>	<code>0x00000020</code>
<code>#define SDL_INIT_CDROM</code>	<code>0x00000100</code>
<code>#define SDL_INIT_JOYSTICK</code>	<code>0x00000200</code>
<code>#define SDL_INIT_NOPARACHUTE</code>	<code>0x00100000</code> /*< Don't catch fatal signals */
<code>#define SDL_INIT_EVENTTHREAD</code>	<code>0x01000000</code> /*< Not supported on all OS's */
<code>#define SDL_INIT_EVERYTHING</code>	<code>0x0000FFFF</code>

When you initialize SDL using `SDL_Init` you have the option of using one or more of the flags above to initialize one or more subsystems. For example, if all you wanted to use was the video functions then you could use:

```
int retValue = SDL_Init(SDL_INIT_VIDEO);
```

If you wanted to use the video and JOYSTICK subsystem then you can use:

```
int retValue = SDL_Init(SDL_INIT_VIDEO | SDL_INIT_JOYSTICK);
```

Most times you will just use `SDL_INIT_EVERYTHING`. It is advisable that you check the integer return value. On success the function returns 0 otherwise -1. You can obtain the error message by calling `SDL_GetError`.

Function Name: **SDL_GetError()**

Format:

```
char* SDL_GetError();
```

Description:

The function returns a c-character string that describes the last SDL error.

All your SDL programs should start with the following code before you start using any SDL functions:

```
If (SDL_Init(SDL_INIT_EVERYTHING) == -1) {
    // darn something went wrong
    cerr << "SDL_Init failed error message: " << SDL_GetError() << endl;
    exit(1);
}
// . . . rest of program . . .
```

```
SDL_Quit();
```

Note, the use of `SDL_Quit()` to close all SDL systems. This must be at the end of your program when you are done using SDL functions. In addition, if an error were to occur you would see the creation of the file `stderr.txt` file populated with the error message.

Using `atexit()`

There is a C++ function you can use to execute functions that require no arguments. The function is `atexit()`. So instead of having to remember to insert the `SDL_Quit()` function at the end of your program you can set it up so when your program terminates (for whatever reason) the function will always get called.

Function Name: **atexit**

Format:

```
int atexit( void (* function ) (void) );
```

Description:

The function pointer provided as an argument is called when the program terminates. Note, that the function must be a void argument function. You can use `atexit` as many times as you need to ensure that all functions (usually clean-up functions) execute when the program ends.

Example Usage:

```
atexit(SDL_Quit);
```

I don't use this function in any of my examples but you may encounter it while inspecting SDL example programs.

Initializing and Closing SDL Subsystems

SDL also allows you to open and close one or more subsystems directly at the points in your program when you need them.

Function Name: **SDL_InitSubSystem**

Format:

```
SDL_InitSubSystem(Uint32 flags)
```

Description:

This function initializes one or more SDL subsystems specified in the argument flags. The return value is 0 on success, otherwise -1. You can use the function `SDL_GetError()` to obtain the last error message. The arguments will be one of the flag values show in Table 5.

You would use `SDL_InitSubSystem` if after using `SDL_Init()` at the start of your program you want to open and close the CDROM system in a small section of your program.

Example Usage:

```
If (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_JOYSTICK) == -1) {
    // darn something went wrong
    cerr << "SDL_Init failed error message: " << SDL_GetError() << endl;
    exit(1);
}
// . . . rest of program . . .
// we need to use the CDROM here
If(SDL_InitSubSystem(SDL_INIT_CDROM) == -1) {
    // something went wrong on CD...just write to log file
    cerr << "Failed to initialize CDROM " << SDL_GetError() << endl;
    exit(2);
}
// . . . do our thing with CDROM
SDL_QuitSubsystem(SDL_INIT_CDROM);
// . . . finish up the program . . .
SDL_Quit();
```

Once you are done with the subsystem you directly close it using **SDL_QuitSubsystem** function.

Function Name: **SDL_QuitSubSystem**

Format:

SDL_QuitSubSystem(Uint32 flags)

Description:

This function shuts down the one or more designated subsystems as specified in the flags argument. The argument value is one of the values shown in Table 5.

If you did not want to exit the program if a particular subsystem failed to be opened (could not get that great background music from the CDROM) you can check later in the program (when you are ready to play the music) by using the function **SDL_WasInit**.

Function Name: **SDL_WasInit**

Format:

Uint32 SDL_WasInit(Uint32 flags)

Description:

This function returns a Uint32 value a mask or flag indicating which subsystems specified in the argument flags have been initialized. For example suppose you separately initialized the CDROM and AUDIO subsystems in order to obtain and play some sounds in your game. Before you actually tried to obtain and play the music you want to check if these subsystems were properly initialized by using one of the values listed in Table 5.

Example Usage:

```

If (SDL_InitSubSystem(SDL_INIT_CDROM | SDL_INIT_AUDIO) == -1) {
    cerr << "Unable to initialize CDROM and/or AUDIO. " << SDL_GetError()
<< endl;
}

// check if we can get and play the wild tune
Uint32 retValue = SDL_WasInit(SDL_INIT_CDROM | SDL_INIT_AUDIO);
if (retValue & SDL_INIT_CDROM) {
    // we successfully initialized CDROM so let's get the music off the
    // CDROM
    .
    .
    if (retValue & SDL_INIT_AUDIO) {
        // let's play that tune
    }
}
SDL_QuitSubSystem(SDL_INIT_CDROM | SDL_INIT_AUDIO);

```

Let's create our first program to initialize and close SDL.

LAB #1: Program 2_1 – Test Initializing SDL.

- >Create a new project named Program2_1 using the template Simple SDL Project template you created in the previous chapter (see page 39)

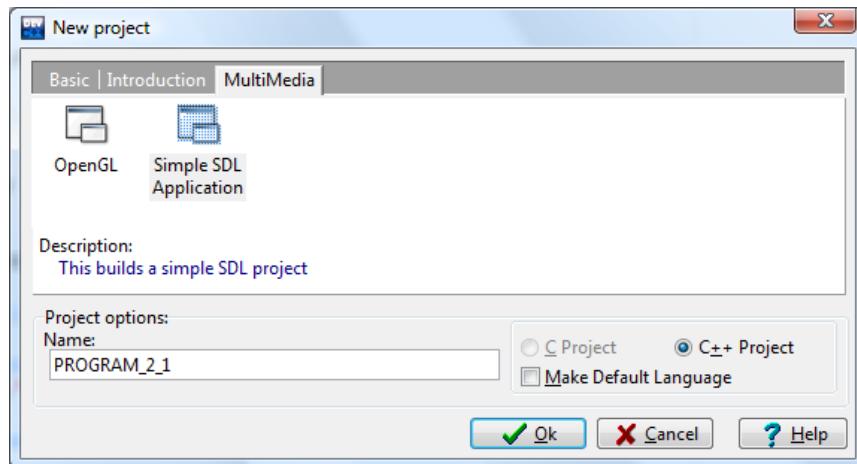


Figure 60 - Creating an SDL application

Replace the text with the following program:

Table 6 - PROGRAM 2_1

```

// Program: PROGRAM2_1
// Purpose: Initializing and closing SDL
#include <iostream>
#include "SDL\SDL.h"

using namespace std;
int main(int argc, char* argv[])

```

```
{
    //initialize SDL
    if (SDL_Init(SDL_INIT_EVERYTHING)==-1) {
        cerr << "Could not initialize SDL" << endl << SDL_GetError()
            << endl;
        exit(1);
    } else {
        //report success
        cout << "SDL_INIT_EVERYTHING worked." << endl;
    }

    cout << "Preparing to close SDL..." << endl;

    SDL_Quit();

    cout << "Terminating normally." << endl;

    return(0);
}
```

 Compile and execute

You will not see anything on the screen but if you open up the directory where the *.exe file is located you will see the file stdout.txt.

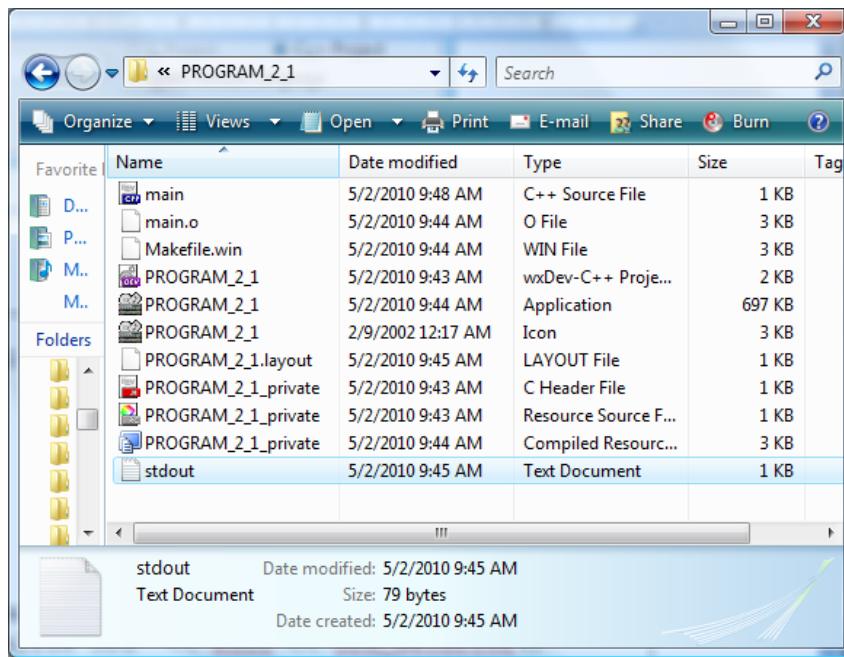


Figure 61 - Finding the stdout.txt file

Open the file and you will see the messages you generated with cout.

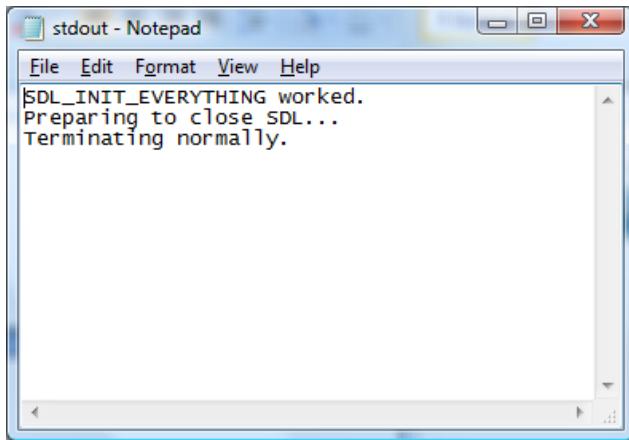


Figure 62 - Contents of stdout.txt

As you can see the program records all the messages you printed out with cout into the file stdout.txt.

TIP: Using cout throughout your program to record important events and progress is a good way to “see” what is going on. Try not to have too many statements in the game loop since that will generate many statements.

What is a game loop?

The key component of any game is the game loop. “The game loop allows the game to run smoothly regardless of a user’s input or lack thereof.”¹³ Inside the game loop the game either responds to user input (pressing the joystick to fire a missile at the aliens coming down) or to figure out what directions and how fast to move the ghosts or monsters so they make life miserable for the hero (this is called AI for artificial intelligence). The traditional game loop is something like this:

```
While (the game is not over)
    Check and process any user input;
    Compute AI;
    Move the monsters or enemies;
    Resolve collisions;
    Draw graphics;
    Play sounds;
End While
```

¹³ http://en.wikipedia.org/wiki/Game_programming

The Video Component

Since we really want to create games we need to first learn how to start working with the video display. This is where we will have the hero shoot the monsters streaming down the screen or the heroine grapple the walls as she dodges boulders and bullets, so we will investigate this first.

The computer monitor is most often used to provide users with feedback on what is going on with the application or game that is running by displaying text and graphics on the screen. Monitors are either the liquid crystal displays (LCD) or cathode ray tube (CRT).



Most of today’s computers come with LCD monitors since they are slimmer and require less energy than the classic CRT monitor. There are usually several things users care about when they purchase a monitor - the screen size and aspect ratio. Typically the aspect ratio is 4:3 which means the width to length ratio is 4 to 3. The screen is usually slightly wider 15, 17, 19 inches or more. Resolution refers to the number of individual dots of color (pixels) that the screen can display. Resolution is expressed as number on the horizontal axis times the number on the vertical axis. Many older games used 320x240¹⁴ and for many of our SDL based games we will use the resolution 640x480. You read this as 640 pixels across and 480 pixels down.

“The combination of the display modes supported by your graphics adapter and the color capability of your monitor determine how many colors it displays. For example, a display operates in SuperVGA (SVGA) mode can display up to 16,777,216 (usually rounded to 16.8 million) colors because it can process a 24-bit long description of a pixel. The number of bits used to describe a pixel is known as its bit depth.”¹⁵

When your video adapter card supports 24-bit depth, then 8-bits is used to describe each of the primary colors – red, green and blue.

Table 7 - Chart from HowStuffHowWorks

Bit-Depth	Number of Colors
1	2 (monochrome)
2	4 (CGA)
4	16 (EGA)
8	256 (VGA)
16	65,536 (High Color, XGA)
24	16,777,216 (True Color, SVGA)
32	16,777,216

¹⁴ This screen mode was known as Mode X. Its primary advantage was that the pixels were square.

¹⁵ From HowStuffWorks - <http://computer.howstuffworks.com/monitor4.htm>

	(True Color + Alpha Channel)
--	------------------------------

Most of today's monitors and video cards usually use bit-depth 24 or 32.

What is a pixel?

The video display or monitor is composed of thousands (or millions) of pixels. A pixel is short for picture element, it represents a single point in a graphic image. A pixel can be constructed by one or more dots on the screen. As programmers we conceptualize or think of the screen as being composed of these pixels, where each pixel can be a particular color.

SDL Video Structures

There are seven key structures that you use to manage the video display, if you don't know what a *struct* is then please read Appendix E for a comprehensive introduction. We will discuss each structure in detail but for now they are:

1. **SDL_Rect** – represents a rectangular area on the screen
2. **SDL_Color** – A structure to represent a color in a platform-independent way
3. **SDL_Palette** – Used to hold a palette or the set of colors your game is using. In the olden days (1990's) you had to manage the color palette but with higher graphics system of today we rarely concern ourselves with this anymore.
4. **SDL_PixelFormat** – This structure is used to hold the details of the pixels pertaining to the user's video system.
5. **SDL_Surface** – This represents a block of pixels. We use this to represent the display surface, the image surface, etc. Your application may contain many surfaces in memory. One main surface will be the surface that represents the SDL window. A surface can hold images from files or rectangular regions from other surfaces.
6. **SDL_VideoInfo** – This structure holds the details about the user's video system
7. **SDL_Overlay** – This structure is used for data streaming

- **SDL_Color** – This structure is used to hold color information in a platform-independent way.

```
typedef struct SDL_Color {
    Uint8 r;
    Uint8 g;
    Uint8 b;
    Uint8 unused;
} SDL_Color;
```

The struct members r, g, and b stand for red, green and blue respectively. Each member can be a value from 0..255, where 0 means lack of intensity and 255 is maximum intensity for that particular color. **SDL_Color** describes a color in a format independent way. We

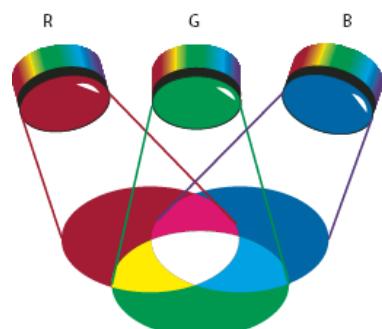


Figure 63 - RGB to make colors

usually represent a color by using a triple (r, g, b). For example, (255,0, 0) represents red, (255,255,255) is white. You will need to convert an `SDL_Color` variable to a pixel value for the display format to use by using the function `SDL_MapRGB`. That is, in order to use an `SDL_Color` variable you will need to convert it into a pixel color to be used for the video display by using `SDL_MapRGB`.

A large percentage of the visible spectrum can be represented by mixing red, green, and blue (RGB) colored light in various proportions and intensities. Where the colors overlap, they create cyan, magenta, and yellow. RGB colors are called additive colors because you create white by adding R, G, and B together—that is, all light is reflected back to the eye. Additive colors are used for lighting, television, and computer monitors. Your monitor, for example, creates color by emitting light through red, green, and blue phosphors. You can work with color values using the RGB color mode, which is based on the RGB color model. In RGB mode, each of the RGB components can use a value ranging from 0 (black) to 255 (white). For example, a bright red color might have an R value of 246, a G value of 20, and a B value of 50. When the values of all three components are equal, the result is a shade of gray. When the value of all components is 255, the result is pure white; when all components have values of 0, the result is pure black.¹⁶

In our programs we will typically create an `SDL_Color` object and set it to some color:

```
SDL_Color redColor = { 255, 0, 0 } ;
```

Then we will need to convert `redColor` into a value that can be written to the screen. We will see how to do that later when we discuss `SDL_MapRGB`.

- ⊕ **SDL_Surface** – This structure represents “areas of ‘graphical’ memory, memory that can be drawn to. The surface represents a rectangular area representing the screen or graphics area.

```
typedef struct SDL_Surface {
    Uint32 flags;                      /**< Read-only */
    SDL_PixelFormat *format;            /**< Read-only */
    int w, h;                          /**< Read-only */
    Uint16 pitch;                      /**< Read-only */
    void *pixels;                     /**< Read-write */

    /** clipping information */
    SDL_Rect clip_rect;                /**< Read-only */

    /** Reference count -- used when freeing surface */
    int refcount;                      /**< Read-mostly */

    /* -- other members that are private -- */
} SDL_Surface;
```

The `flags` is a value that indicates one or more aspects of the surface that gets established when you create the `SDL_Surface` object. The possible values are:

¹⁶ From http://help.adobe.com/en_US/Illustrator/14.0/WS714a382cdf7d304e7e07d0100196cbc5f-6293a.html

Table 8 - SDL_Surface flags

SDL_Surface flags value	
SDL_ANYFORMAT	Allow any pixel-format (display surface)
SDL_ASYNCBLIT	The surface uses asynchronous blit if possible
SDL_DOUBLEBUF	Specifies that the surface is double buffered (display surface)
SDL_HWACCEL	Use hardware acceleration blit
SDL_HWPALLETTE	Specifies that the surface has an exclusive palette
SDL_HWSURFACE	Specifies that the surface is stored in video memory
SDL_FULLSCREEN	Specifies that the surface uses the full screen (display surface)
SDL_OPENGL	Specifies that the surface has an OpenGL context (display surface)
SDL_OPENGLBLIT	Specifies that the surface supports OpenGL blitting (display surface). NOTE: This option is kept for compatibility only, and is not recommended for new code.
SDL_RESIZEABLE	Specifies that the surface is resizeable (display surface)
SDL_RLEACCEL	Specifies that accelerated colorkey blitting with RLE is being used.
SDL_SRCALPHA	Specifies that surface blit uses alpha blending
SDL_SRCCOLORKEY	Specifies that the surface uses colorkey blitting
SDL_SWSURFACE	Specifies that the surface is stored in the system memory. SDL_SWSURFACE is not actually a flag, when SDL_HWSURFACE is not set then this implies that SDL_SWSURFACE is true.
SDL_PREALLOC	Specifies that the surface uses preallocated memory

We will not get into each item above in detail but will point out the SDL functions we use specify or set one or more of these flags.

The *format* member specifies the format of the pixels stored on the surface. The field value is a pointer to an **SDL_PixelFormat** struct (more on this later).

The *w* and *h* member indicates the width and height of pixels on the surface.

The member *pitch* specifies the surface scanline in bytes. This value indicates the number of bytes you would have to add to a pixel at location x,y on the screen in order to get the pixel position immediately below it. We discuss this in more detail later.

The member *pixels* is a pointer to the actual pixels making up the surface or image.

The *clip_rect* member is actually another SDL structure **SDL_Rect**, which is a rectangular area which specifies an area on the surface that will be affected by blitting. That is, the clip area describes a subset of the display area where changes will take place and any area outside the *clip_rect* will not be changed. This help to restrict changes to only the *clip_rect* area.

The *refcount* member tracks the number of references to the **SDL_Surface** object. This int value allows you to keep track how many elements depend on or use this surface. When an object requires the **SDL_Surface** the refcount should be incremented by 1, when an object no longer requires the **SDL_Surface** then it should decrement this member by 1. When the refcount value is 0 it is safe to delete or destroy the surface.

The primary and first use of this structure in your SDL program is when we establish the video mode using the function **SDL_SetVideoMode**. **SDL_SetVideoMode** function is used to set up the video mode, that is, the width, height and bits-per-pixel or color depth.

Function Name: **SDL_SetVideoMode**

Format:

```
SDL_Surface *SDL_SetVideoMode(int width, int height, int bpp, Uint32 flags);
```

Description:

This function is used to set up the video mode display. Clients specify the width and height of the video display they want SDL to create and bpp is the bits per pixel value. If bpp is 0 then SDL uses the current display bits per pixel. The flag is a combination of one or more (or'd or |) values specified in **Error! eference source not found..**

The function returns a pointer to an **SDL_Surface** that represents our display video surface

Example Usage:

```
SDL_Surface *pDisplaySurface = NULL;  
pDisplaySurface = SDL_SetVideoMode(640, 480, 0, SDL_ANYFORMAT);
```

You should check that it really worked by checking that the pDisplaySurface != NULL, otherwise generate an error message and stop the program. See Appendix F if you need a review or need more information on pointers. In this example we are specifying that a 640x480 video display be setup. The third argument, bits per pixel is 0 which means we will use the current display default value (in this day and age this value defaults to 32). The last argument **SDL_ANYFORMAT** specifies that the current format of the display is taken as the format for the video display surface we are creating. We can add additional flags to this argument but for now we will just use **SDL_ANYFORMAT**.

Before we do our next program using this function, let's cover two additional functions we will need to use.

Function Name: **SDL_FreeSurface**

Format:

```
void SDL_FreeSurface(SDL_Surface *surface);
```

Description:

This function frees (deletes) an **SDL_Surface** surface. You are responsible for freeing ALL **SDL_Surface** variables that you create. So must make sure to do:

```
SDL_FreeSurface(pDisplaySurface);
```

after you are done with a surface.

In order to keep our next program simple and ensure that we see the display window we will add a delay statement into our first program. SDL provides the function **SDL_Delay** to manage this.

Function Name: **SDL_Delay**

Format:

```
void SDL_Delay(Uint32 ms);
```

Description:

This function waits a specified number of milliseconds before returning. If you want to wait 1 sec you would specify:

```
SDL_Delay(1000);
```

LAB #2: Program 2_2 – Create a display window

- ✚ Create a new project named Program2_2 using the template Simple SDL Project template.
- ✚ Enter the following lines into the main.cpp

Table 9 - PROGRAM2_2

```
// Program: PROGRAM2_2
// Purpose: Demonstrates creating a window and displaying for 1 second
#include <iostream>
#include "SDL\SDL.h"

using namespace std;
int main(int argc, char* argv[])
{
    //initialize SDL
    if (SDL_Init(SDL_INIT_EVERYTHING)==-1) {
        cerr << "Could not initialize SDL" << endl
            << SDL_GetError() << endl;
        exit(1);
    } else {
        //report success
        cout << "SDL_INIT_EVERYTHING worked." << endl;
    }

    // Create SDL Surface
    SDL_Surface *pDisplaySurface = NULL;
    pDisplaySurface = SDL_SetVideoMode(640, 480, 0, SDL_ANYFORMAT);
    if (pDisplaySurface == NULL ) {
        cerr << "SetVideoMode failed. " << SDL_GetError() << endl;
        exit(2);
    }
    SDL_Delay(1000);    // wait 1 second before ending the program
    cout << "free the SDL Surface..." << endl;
    SDL_FreeSurface(pDisplaySurface);

    cout << "Preparing to close SDL..." << endl;
```

```

    SDL_Quit();

    cout << "Terminating normally." << endl;

    return(0);
}

```

 Compile and execute

You will see a window quickly appear for 1 second. If you want the window to appear longer just change the value provided to `SDL_Delay`.

Making Improvements to our Video Programs

You really can't do anything with the window in the last example – not even close it! The reason is we have not written code to have the program respond to events. The next chapter gets into more details on the different types of events that a program can handle. For now we will make two improvements to the program.

1. Create consts for the `SCREEN_WIDTH` and `SCREEN_HEIGHT`
2. Add a game loop that handles the close window event

The first improvement is just plain good coding practice. Whenever I see numbers in my program (other than 0 and -1 for checks on return values) I ask myself one question – would I ever want to change it? If the answer is “yes” than I create a const in the program. The second one will allow you to decide when to close the window rather than inserting a delay in your program.

LAB #3: Program 2_3 – Making improvements

-  Create a new project named `Program2_3` using the template Simple SDL Project template.
-  Enter the following lines into the `main.cpp`

Table 10 - PROGRAM2_3

```

// Program: PROGRAM2_3
// Purpose: Demonstrates creating a window and having it wait for you to
close
#include <iostream>
#include "SDL\ sdl.h"

using namespace std;

const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;
const int BITS_PER_PIXEL = 0; // set to current display value

int main(int argc, char* argv[])
{
    //initialize SDL
    if (SDL_Init(SDL_INIT_EVERYTHING)==-1) {
        cerr << "Could not initialize SDL" << endl
}

```

```

        << SDL_GetError() << endl;
        exit(1);
    }

    // Create SDL Surface
    SDL_Surface *pDisplaySurface = NULL;
    pDisplaySurface = SDL_SetVideoMode(SCREEN_WIDTH, SCREEN_HEIGHT,
                                       BITS_PER_PIXEL, SDL_ANYFORMAT);
    if (pDisplaySurface == NULL) {
        cerr << "SetVideoMode failed. " << SDL_GetError() << endl;
        exit(2);
    }
    // Set up game loop waiting for window to close
    SDL_Event event;
    for(;;) {
        if (SDL_PollEvent(&event) == 0) {
            // no event so DO YOUR THING!
            // . . . nothing yet . . .
        } else {
            // an event ...let's check if user has closed the window
            // if so ..exit the loop ...BYE!
            if (event.type == SDL_QUIT) break;
        }
    }
    // wrap things up
    SDL_FreeSurface(pDisplaySurface);
    SDL_Quit();

    cout << "Terminating normally." << endl;

    return(0);
}

```

Let me state again the improvements we made to the program:

- Added const values
- Added a game loop
 - The game loop consists of an ifStatement that tests if an event occurred and if so to process it.

```

for(;;) {    // forever loop
    if (no event exists) {
        Do game related things - AI, collision detection, etc.
    } else {
        Check if the event was a "close window" request.
    }
} // end for loop

```

The only way to exit the game loop is for the user to close the window – doing so generates an SDL_QUIT event. All our programs in this chapter will have this form. Later we will adopt a different format that supports a more solid design involving C++ classes.

How the display screen is organized

For this discussion we will assume our screen size is 640x480, that is, there are 640 pixels across and 480 pixels down.

The direction across the screen from left-to-right is regarded as the X → direction.

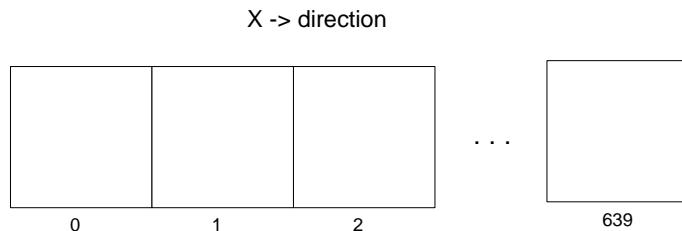


Figure 64 – 640 pixels in the x direction

Each pixel cell across is numbered from 0 to MAX_WIDTH-1 which for our example will be from 0 to 639.

This direction is referred to as the y → direction and the rows are numbered from 0 through MAX_HEIGHT-1.

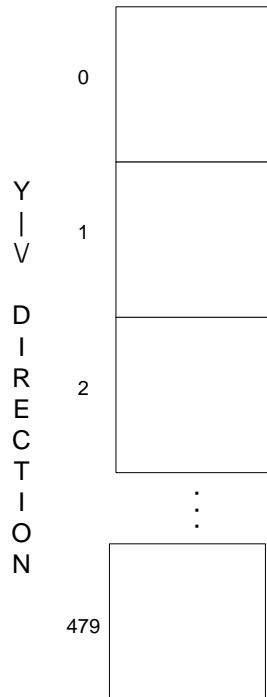


Figure 65 - Rows or y direction

In our example, the y value will be from 0 to 479.

You can view the video display as consisting of a grid of pixels.

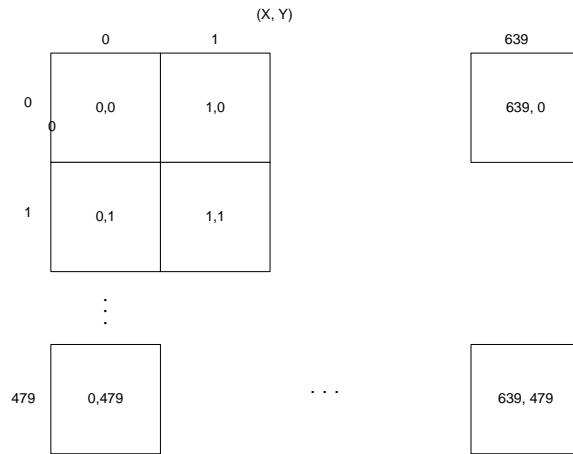


Figure 66 - Grid of pixels

Each pixel is addressable and can be set to a specific color. We usually refer to the pixel address as a tuple (x,y) the first number specifies the x location and the second the y location. The top-left most location (as shown in Figure 66) is location (0,0) and the rightmost address on that row has the address (639,0). On the second row the pixel address starts at (0,1) and goes through to (639,1). The bottom-right most pixel has the address (639, 470). It would be nice if we could just use a function like:

```
drawPixel(pDisplaySurface, x, y, color);
```

where the pDisplaySurface is the SDL_Surface we get from the function `SDL_SetVideoMode` but such a function does not exist in SDL. How to actually get this done is the topic of the next section

Understanding how to write to the Display

In this section we are going to build a program that draws a random pixel on the screen with a random color. Once we get that done we will write a general function we can call on later to just draw a pixel color on the screen. Lastly you will be challenged to come up with a general functions to draw a line¹⁷ and a circle to the screen.

What do you need to know in order to set a pixel on the screen to a particular color?

- ✚ The pixel location on the screen
 - The x location, which in our case will be a value from 0..SCREEN_WIDTH-1
 - The y location, which in our case will be a value from 0..SCREEN_HEIGHT-1
- ✚ The color – we can use `SDL_Color` and convert into a color for the display
- ✚ The video display surface

From the previous section we know that the line:

¹⁷ These functions are already available in graphics libraries that supplement SDL but it is interesting see how it is done.

```
pDisplaySurface = SDL_SetVideoMode(SCREEN_WIDTH, SCREEN_HEIGHT,
                                  BITS_PER_PIXEL, SDL_ANYFORMAT);
```

returns our display surface in a `SDL_Surface` struct.

```
typedef struct SDL_Surface {
    Uint32 flags;                      /**< Read-only */
    SDL_PixelFormat *format;           /**< Read-only */
    int w, h;                          /**< Read-only */
    Uint16 pitch;                     /**< Read-only */
    void *pixels;                     /**< Read-write */

    /** clipping information */
    SDL_Rect clip_rect;              /**< Read-only */

    /** Reference count -- used when freeing surface */
    int refcount;                    /**< Read-mostly */

    /* -- other members that are private -- */
} SDL_Surface;
```

The `SDL_Color` structure is defined as:

```
typedef struct{
    Uint8 r;
    Uint8 g;
    Uint8 b;
    Uint8 unused;
} SDL_Color;
```

We will use this structure to specify a particular color. The variables `r`, `g`, `b` will hold some value from 0..255. Here are some examples on how it works:

FFF	CCC	999	666	333	000	FFC	FF9	FF6	FF3
FFF	CCC	999	666	333	000	C00	900	600	300
99C	C00					CC9	FFC	FF9	FF6
						C33	C66	966	300
CCF	CCF	333	666	999	CCC	FFF	CC9	CC6	330
F00	F33	300	600	900	C00	F00	933	633	000
99F	CCF	99C	666	999	CCC	FFF	996	993	663
F00	F66	C33	633	933	C33	F33	600	300	333
66F	99F	66C	669	999	CCC	FFF	996	663	996
F00	F66	C33	900	966	C66	F66	633	300	666
33F	66F	339	66C	99F	CCC	FFF	CC9	CC6	CC9
F00	F33	900	C00	F33	C99	F99	966	600	999
00C	33C	336	669	99C	CCF	FFF	FF9	FFC	FF9
C00	C00	600	933	C66	F99	FCC	C99	933	CCC
33C	66C	00F	33F	66F	99F	CCF			CC9
C33	C66	F00	F33	F66	F99	FCC			996
006	336	009	339	669	99C				993
600	633	900	933	966	C99				990
003	00C	006	339	66C	99F	CCF	339	99C	CC9
300	C33	633	966	C99	FCC	FFF	9FF	CFF	9FF
00F	33F	009	00C	33F	99F	99C	006	669	999
F33	F66	933	C66	F99	FFF	CCC	6CC	9CC	9FF
00F	66F	33C	009	66F	66C	669	003	336	666
F66	F99	C66	966	FFF	CCC	999	366	699	6FF
00F	66F	33C	33F	33C	339	336	006	003	333
F99	FCC	C99	FFF	CCC	999	666	699	399	3FF
00F	33F	00F	00C	009	006	003	339	336	000
FCC	FCC	FCC	CCC	999	666	333	9CC	6CC	0FF
00C		© 2000 VisiBone			009	33C	66C	669	336
C99					9CC	CFF	9FF	6FF	3FF
									330
									0CC

Figure 67 - Hex table of colors

The numbers are in hex and a number like 99FFCC means r=99 or decimal 153, g=FF or decimal 255, b=CC or 204, which on the chart comes out to a washed out green color (I am not good at naming colors). Here is a block¹⁸ using 99FFCC or (153,255,204):

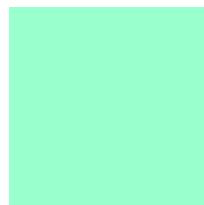


Figure 68- The color 99FFCC (153, 255, 204)

We could create an `SDL_Color` object to hold our intended color:

```
SDL_Color myColor;
myColor.r = 153;
myColor.g = 255;
myColor.b = 204;
```

or

```
SDL_Color myColor = {153, 255, 204};
```

¹⁸ I used Windows Paint program to create the block image.

SDL_Color allows us to describe a color in a format independent way, but we will need to convert it to a pixel value for a certain pixel format that can be sent to the actual video display using SDL_MapRGB function.

Function Name: **SDL_MapRGB**

Format:

```
Uint32 SDL_MapRGB(SDL_PixelFormat fmt*, Uint8 r, Uint8 g, Uint8 b);
```

Description:

This function maps the RGB color to the specified pixel format and returns the pixel value as a 32-bit int. We know that the bits per pixel (bpp also known as color depth) can be less than 32 bits, if so we can just ignore the upper-portion of the return value that does not pertain to the number of BytesPerPixel we require (or just save into a Uint16 for 16 bpp or Uint8 for 8 bpp). The troublesome format will be 24bpp since we don't have a natural datatype to express. The **SDL_PixelFormat** we will use is the one we obtained in the **SDL_Surface** structure returned from calling **SDL_SetVideoMode**, that is, **pDisplaySurface->format**.

We will need to use the function **SDL_MapRGB** in order to obtain a pixel format we can send to the screen:

```
Uint32 displayColor = SDL_MapRGB(pDisplaySurface->format, myColor.r,
                                  myColor.g, myColor.b);
```

So now the next question is how do we get this **Uint32** **displayColor** value displayed at pixel location (x,y)? In other words, we have the representation for the pixel color we want to display so where exactly is the location of (x,y)?

The **SDL_Surface** holds the area representing our surface or screen in **void *pixels**. Life would be easy if the location *(pixel+0) corresponded to location (0,0) and the location *(pixel+1) was pixel location (1,0), etc. but the location of the next pixel position in the memory area pointed to by pixels is determined by the number of bytes per pixel. How do we get that information? The information is contained right inside the **SDL_Surface** struct in the **SDL_PixelFormat *format** struct.

The **SDL_PixelFormat**:

```
/** Everything in the pixel format structure is read-only */
typedef struct SDL_PixelFormat {
    SDL_Palette *palette;
    Uint8 BitsPerPixel;
    Uint8 BytesPerPixel;
    Uint8 Rloss;
    Uint8 Gloss;
    Uint8 Bloss;
    Uint8 Aloss;
```

```

Uint8 Rshift;
Uint8 Gshift;
Uint8 Bshift;
Uint8 Ashift;
Uint32 Rmask;
Uint32 Gmask;
Uint32 Bmask;
Uint32 Amask;

/** RGB color key information */
Uint32 colorkey;
/** Alpha value information (per-surface alpha) */
Uint8 alpha;
} SDL_PixelFormat;

```

We will not cover all the fields in `SDL_PixelFormat` in this section. The key one we will need for this exercise is `BytesPerPixel`. The `BytesPerPixel` tells us the number of bytes used to represent color for each pixel in a surface. This number will usually be some number from 1 to 4. So the way to get to this information is to use the format:

```
pDisplaySurface->format->BytesPerPixel
```

We can imagine display memory as:

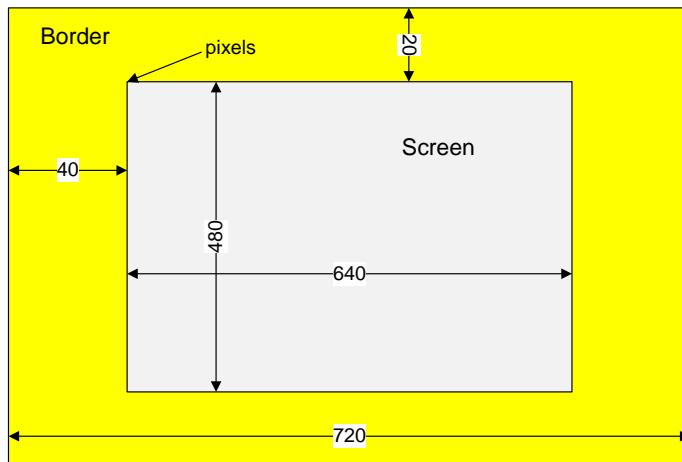


Figure 69 - Screen Memory layout

Now it would be easy to think that the formula for computing the memory address of pixel (x,y) would just be the following:

```
start_memory_of_display_memory + offset_for_starting_row(using y) + offset_in_row (using x)
```

We will use the fact that the display memory pixels start at address pointed to by `pDisplaySurface->pixels` (see Figure 69), which corresponds to the pixel at screen location $(0,0)$.

```

char* pPixelAddress;      // variable to hold hold the
                         //starting pixelAddress for the pixel at (x,y)
// initialize to the starting location of the display surface address
pPixelAddress = (char*)pDisplaySurface->pixels;

// add horizontal offset - x
pPixelAddress = pPixelAddress + (x * pDisplaySurface->format->BytesPerPixel);

// add vertical offset - y
pPixelAddress = pPixelAddress
    + (y * MAX_WIDTH * pDisplaySurface->format->BytesPerPixel);

```

Let's see if we can make sense out of the above which as seen in a more mathematical light would be expressed as:

$pPixelAddress = startingPixelAddress + x * BYTESPERPIXEL + y * MAX_WIDTH * BYTESPERPIXEL;$

Using $BYTESPERPIXEL = 3$ (24bpp) and $MAX_WIDTH=640$

for $(0,0)$ we would get

$pPixelAddress = startingPixelAddress + 0 * BYTESPERPIXEL + 0 * MAX_WIDTH * BYTESPERPIXEL;$

or just $pPixelAddress = startingPixelAddress$ (makes sense right)

Let's say we wanted the pixel address of $(1,0)$,

$pPixelAddress = startingPixelAddress + 1 * BYTESPERPIXEL + 0 * MAX_WIDTH * BYTESPERPIXEL;$

$pPixelAddress = startingPixelAddress + 3$ (again makes sense since it should be 3 bytes down)

What about the pixel address of $(0,1)$, that is the pixel right below $(0,0)$ (see Figure 66)

For $(0,1)$

$pPixelAddress = startingPixelAddress + 0 * BYTESPERPIXEL + 1 * MAX_WIDTH * BYTESPERPIXEL;$

$pPixelAddress = startingPixelAddress + 0 + 1 * 640 * 3$

$pPixelAddress = startingPixelAddress + 1920$

The above is logical but not correct. The above is assuming that if you know the address of pixel x,y on the screen that the pixel below is at (address of pixel x,y) + $MAX_WIDTH * BYTESPERPIXEL$. But, in general the offset $y * MAX_WIDTH * BYTESPERPIXEL$ will not be accurate. I tried to illustrate a

possible reason why this would be true in Figure 69¹⁹. The memory address of the pixels comprising the next scan line may not be adjacent to the end of the pixels comprising the last scan line. In our example above the offset would need to be adjusted by 80 units. To make it easier for users to compute the offset SDL holds within the `SDL_Surface` the struct variable `pitch`. The `pitch` holds the actual length of the scanline (borders and all!) so the formula to compute the `pPixelAddress` is:

```
char *pPixelAddress = (char *)pDisplaySurface->pixels
    + x * pDisplaySurface->format->BytesPerPixel
    + y *pDisplaySurface->pitch ;
```

The function we will use to copy the value we got from `SDL_MapRGB` – `displayColor` is `memcpy`:

```
memcpy(pPixelAddress, &displayColor, pDisplaySurface->format-
>BytesPerPixel)
```

memcpy

This function copies the value of `num` bytes from the location pointed by `source` directly to the memory block pointed by `destination`.

```
void * memcpy( void * destination, const void * source, size_t num);
```

The `destination` is a pointer to the array or memory where the content is to be copied. The `source` is a pointer to the source of data and `num` is the number of bytes to copy.

the above copies `BYTESPERPIXEL` bytes from `displayColor` (the source) to `pPixelAddress` (the destination).

“Graphics hardware is a shared resource. Operating systems generally require that we lock shared resources before we use them and unlock them after we are done. SDL provides `SDL_LockSurface()` and `SDL_UnlockSurface()` to lock and unlock hardware surfaces. It is possible to have a hardware surface that should not be locked and SDL provides the `SDL_MUSTLOCK()` macro so that we can tell them apart.”²⁰

Function Name: **SDL_LockSurface**

Format:

```
int SDL_LockSurface(SDL_Surface *pDisplaySurface);
```

Description:

This function is used to lock a surface so you can directly access it. Once a surface is locked the update should be done quickly and the lock released.

¹⁹ The actual memory layout may be different but the figure is meant to illustrate a point.

²⁰ From http://linuxdevcenter.com/pub/a/linux/2003/08/07/sdl_anim.html?page=2

Function Name: **SDL_UnlockSurface**

Format:

```
int SDL_UnlockSurface(SDL_Surface *pDisplaySurface);
```

Description:

This function releases a previously locked surface.

ADVICE: Try to unlock a surface as soon as you possibly can.

MACRO Name: **SDL_MUSTLOCK**

Format:

```
int SDL_MUSTLOCK(SDL_Surface *pDisplaySurface)
```

Description:

If the macro returns 0 then you can draw without having to lock and unlock the surface.

You should not make operating system or library calls between lock and unlock of a surface.

Example Usage:

```
if (SDL_MUSTLOCK(pDisplaySurface)) {
    int retValue = SDL_LockSurface(pDisplaySurface);
    if (retValue == -1) {
        cerr << "Could not lock surface. " << SDL_GetError() << endl;
        exit(1);
    }
}
// . . . update screen pixel code here
if (SDL_MUSTLOCK(pDisplaySurface)) {
    SDL_UnlockSurface(pDisplaySurface);

}
// . . . rest of program
```

Let's review what we have learned. We wanted to understand what it would take to set a pixel's color on the screen. We needed four things:

- ✚ The pixel location on the screen
 - The x location, which in our case will be a value from 0..SCREEN_WIDTH-1
 - The y location, which in our case will be a value from 0..SCREEN_HEIGHT-1
- ✚ The color – we can use `SDL_Color` and convert into a color for the display
- ✚ The video display surface

We will create a function `drawPixel` with the following signature:

```
void drawPixel (SDL_Surface* pDisplaySurface, int x, int y, SDL_Color color);
```

We will create a program to test the function. The program will do the following:

Loop

- Generate random x, y location on the screen

- Generate a random color

- Invoke `colorPixel`

Until User closes the Window

LAB #4: Program 2_4 – Plot Pixels

- ✿ Create a new project named `Program2_4` using the template Simple SDL Project template.
- ✿ Enter the following lines into the `main.cpp`

Table 11 - PROGRAM2_4

```
#include <cstdlib>
#include <iostream>
#include <time.h>
#include "SDL\SDL.h"

using namespace std;
    // screen dimensions
const int SCREEN_WIDTH=640;
const int SCREEN_HEIGHT=480;
    // COLOR RANGE
const int MAX_COLOR_VALUE = 255;
    // Function prototypes
void drawPixel (SDL_Surface *surface, int x, int y, SDL_Color color);

//display surface
SDL_Surface* pDisplaySurface = NULL;

//event structure
SDL_Event event;

int main(int argc, char *argv[])
{
    //initialize SDL
    if (SDL_Init(SDL_INIT_VIDEO)==-1) {
        cerr << "Could not initialize SDL!" << endl;
        exit(1);
    } else {
        cout << "SDL initialized properly!" << endl;
    }
    //create windowed environment
    pDisplaySurface =
        SDL_SetVideoMode(SCREEN_WIDTH,SCREEN_HEIGHT,0,SDL_ANYFORMAT);
```

```

// set caption
SDL_WM_SetCaption("Plot Pixels", NULL);
//error check
if (pDisplaySurface == NULL) {
    //report error
    cerr << "Could not set up display surface!" << endl;
    //exit the program
    exit(1);
}
// seed the random number generator
srand ( time(NULL) );

//repeat forever
for(;;) {
    //wait for an event
    if(SDL_PollEvent(&event)==0) {
        // generate random a screen position
        int x = rand() % SCREEN_WIDTH;
        int y = rand() % SCREEN_HEIGHT;
        // generate a random screen color
        SDL_Color color;
        color.r = rand() % MAX_COLOR_VALUE;
        color.g = rand() % MAX_COLOR_VALUE;
        color.b = rand() % MAX_COLOR_VALUE;
        drawPixel (pDisplaySurface, x, y, color);

        //update the screen
        SDL_UpdateRect(pDisplaySurface,0,0,0,0);
    } else {
        //event occurred, check for quit
        if(event.type==SDL_QUIT) break;
    }
}
SDL_FreeSurface(pDisplaySurface);
SDL_Quit();
//normal termination
cout << "Terminating normally." << endl;
return EXIT_SUCCESS;
}

void drawPixel (SDL_Surface *surface, int x, int y, SDL_Color color) {
    // map color to screen color
    Uint32 screenColor = SDL_MapRGB(surface->format, color.r,
                                      color.g, color.b);
    // Calculate location of pixel
    char *pPixelAddress = (char *)surface->pixels
        + x * surface->format->BytesPerPixel
        + y *surface->pitch ;

    // check and the lock the surface
    if (SDL_MUSTLOCK(surface)) {
        int retValue = SDL_LockSurface(surface);
        if (retValue == -1) {
            cerr << "Count not lock surface. " <<
                SDL_GetError() << endl;
            exit(1);
    }
}

```

```

    }

    // copy directly to memory
    memcpy(pPixelAddress, &screenColor, surface->format->BytesPerPixel);

    if (SDL_MUSTLOCK(surface)) {
        SDL_UnlockSurface(surface);
    }

}

```

- ➊ Compile and execute the program. You should see a nice star field appear as shown below.

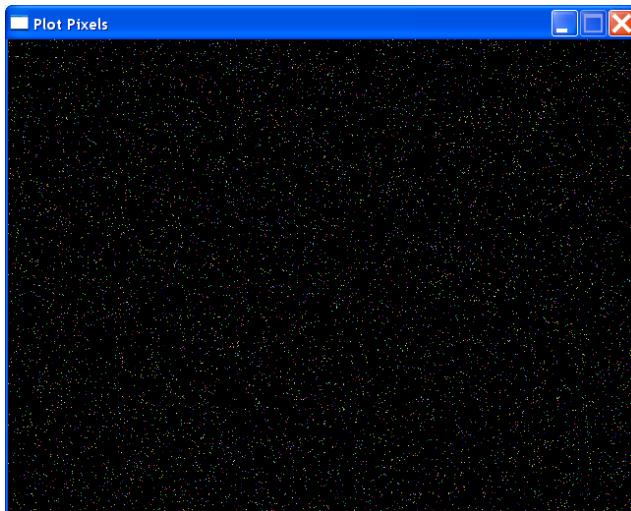


Figure 70 - Plot Pixels screen

The program uses the SDL function **SDL_UpdateRect**.

Function Name: **SDL_UpdateRect**

Format:

```
void SDL_UpdateRect(SDL_Surface *pDisplaySurface, Sint32 x, Sint32 y, Sint32 w, Sint32 h);
```

Description:

This function updates the area on the screen specified by the rectangular specified by the arguments x, y, w and h. If these values are set to 0 then the entire display screen is updated. We use this function to update the screen after we have made changes.

Another function used by the program is `SDL_WM_SetCaption`.

Function Name: **SDL_WM_SetCaption**

Format:

```
void SDL_SetCaption(const char *title, const char *icon);
```

Description:

This function is used to set the windows title (e.g. “Plot Pixels”) and icon name. I usually just use:

```
SDL_WM_SetCaption("Plot Pixels", NULL);
```

The `NULL` value just uses the default windows icon. If you wanted to set your own icon you can use:

Drawing a Line

In this section I will challenge you a bit by taking all the information we have discussed and implement a function not available in `SDL` – `drawLine`. The signature of the new function will be:

```
void drawLine(SDL_Surface *surface, int x0, int y0, int x1, int y1, SDL_Color color);
```

The new function will have six input parameters:

- A pointer to an `SDL_Surface` representing the video display
- The x location `x0` of one point of the line
- The y location `y0` of one point of the line
- The x location `x1` of the other point of the line
- The y location `y1` of the other point of the line
- The `SDL_Color` of the line

The function will draw a line from (x_0, y_0) to (x_1, y_1) using `SDL_Color` on the video display.

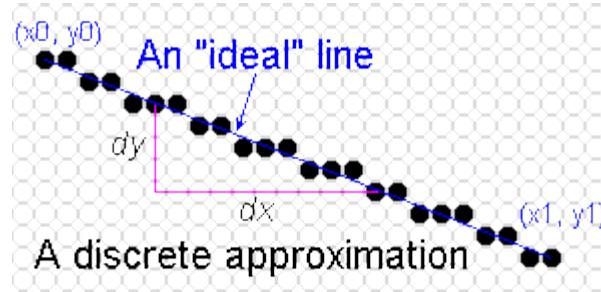
We will be examining several algorithms for drawing a line. I will provide everything but the actual function you will implement the function and test to see it matches my screen display.

A Little History on Drawing Lines²¹

²¹ This section uses the following web site: <http://www.cs.unc.edu/~mcmillan/comp136/Lecture6/Lines.html>.

The invention of raster displays where the image is depicted using a grid of pixels (Figure 66) led to the search of good algorithms for drawing lines and polygons. The work done by researchers investigating the same issues for digital plotters was used to come up with fairly decent line plotting algorithms for monitors and printers. Jack Bresenham, an IBM researcher came up with the most popular algorithm used today. Because of the nature of a raster display we can expect to come up with an approximation to a line. The quest to find an algorithm to display a line on a raster display should meet the following criteria:

- Continuous appearance
- Uniform thickness and brightness
- Are the pixels nearest the ideal line turned on
- How fast is the line generated



A Simple Algorithm – Slope-Intercept Algorithm

The first algorithm we will investigate comes from our notion of “a line” that we learned in algebra. A line was described by the function:

$$y = mx + b$$

where m is the slope and b is the y -intercept. In our program we will have the two endpoints of a line (x_0 , y_0) and (x_1 , and y_1). You may recall that given two points of a line that you can compute the slope as:

$$m = \frac{x_1 - x_0}{y_1 - y_0}$$

The pseudo-code of the algorithm to plot a line is the following:

Given: Two points (x_0 , y_0) and (x_1 , y_1) and a color. We will first plot (x_0, y_0) and then compute the next raster point to plot computing m and b for the line given the initial two points. The main part of the algorithm adds 1 (or -1 depending if x_1 is the left or right of x_0) to x_0 and determines the corresponding y value using the formula $y = mx+b$. This continues until x_0 gets to x_1 .

```

int dx = x1 - x0;
int dy = y1 - y0;
drawPixel(surface, x0, y0, color);      // plot the first point
if (dx != 0) {
    float m = (float) dy / (float) dx; // calculate the slope
    float b = y0 - m * x0;           // compute the y-intercept
    dx = (x1 > x0) ? 1 : -1;
    while (x0 != x1) {
        x0 += dx;                  // next x value
        y0 = round(m*x0 + b); // corresponding y value
    }
}

```

```

        drawPixel(surface, x0, y0, color);
    }
}

```

Let's test it out. I have supplied the program and will leave it to you to implement the function drawLine.

LAB #5: Program 2_5 – Slope-Intercept Algorithm

- ✍ Create a new project named Program2_5 using the template Simple SDL Project template.
- ✍ Enter the following lines into the main.cpp

Table 12 - PROGRAM2_5

```

#include <cstdlib>
#include <iostream>
#include <math.h>
#include "SDL\ sdl.h"

using namespace std;
    // Function prototypes
void drawPixel(SDL_Surface *surface, int x, int y, SDL_Color color);
void drawLine(SDL_Surface *surface, int x0, int y0, int x1, int y1, SDL_Color
color);

    // screen dimensions
const int SCREEN_WIDTH=640;
const int SCREEN_HEIGHT=480;

    //display surface
SDL_Surface* pDisplaySurface = NULL;

    //event structure
SDL_Event event;

    // colors
SDL_Color COLOR_BLACK = { 0, 0, 0 };
SDL_Color COLOR_WHITE = { 255, 255, 255 };

int main(int argc, char *argv[])
{
    //initialize SDL
    if (SDL_Init(SDL_INIT_VIDEO)==-1) {
        cerr << "Could not initialize SDL!" << SDL_GetError() << endl;
        exit(1);
    } else
        cout << "SDL initialized properly!" << endl;
    //create windowed environment
    pDisplaySurface =
        SDL_SetVideoMode(SCREEN_WIDTH, SCREEN_HEIGHT, 0, SDL_ANYFORMAT);

    //error check
    if (pDisplaySurface == NULL) {
        //report error
        cerr << "Could not set up display surface!" << SDL_GetError()
    }
}

```

```

        << endl;
    exit(1);
}

// set caption
SDL_WM_SetCaption("Draw Line", NULL);

//repeat forever
for(;;) {
    //wait for an event
    if(SDL_PollEvent(&event)==0) {
        // Make the background screen white
        SDL_Rect screenRect = {0,0, SCREEN_WIDTH, SCREEN_HEIGHT};
        Uint32 color = SDL_MapRGB(pDisplaySurface->format,
                                  COLOR_WHITE.r,
                                  COLOR_WHITE.g, COLOR_WHITE.b);
        SDL_FillRect(pDisplaySurface, &screenRect, color);
        // this will show up as a widely spaced line
        drawLine(pDisplaySurface, 0,0, 100, 100, COLOR_BLACK);
        // vertical line - this will not show up at all
        drawLine(pDisplaySurface, 100,0, 100, 300, COLOR_BLACK);
        // horizontal line -
        drawLine(pDisplaySurface, 100,100, 400, 100, COLOR_BLACK);
        // this looks fine since m = 1
        drawLine(pDisplaySurface, SCREEN_WIDTH-1, SCREEN_HEIGHT-1,
                  SCREEN_WIDTH/2,SCREEN_HEIGHT/2, COLOR_BLACK);
        drawLine(pDisplaySurface, 0,0, 30, 400, COLOR_BLACK);
        //update the screen
        SDL_UpdateRect(pDisplaySurface,0,0,0,0);
    } else {
        //event occurred, check for quit
        if(event.type==SDL_QUIT) break;
    }
}
SDL_FreeSurface(pDisplaySurface);
SDL_Quit();
//normal termination
cout << "Terminating normally." << endl;
return EXIT_SUCCESS;
}

void drawPixel (SDL_Surface *surface, int x, int y, SDL_Color color) {
    // map color to screen color
    Uint32 screenColor = SDL_MapRGB(surface->format, color.r, color.g,
                                    color.b);
    // Calculate location of pixel
    char *pPixelAddress = (char *)surface->pixels
                           + x * surface->format->BytesPerPixel
                           + y *surface->pitch ;

    // check and the lock the surface
    if (SDL_MUSTLOCK(surface)) {
        int retValue = SDL_LockSurface(surface);
        if (retValue == -1) {
            cerr << "Could not lock surface. "
                << SDL_GetError() << endl;
            exit(1);
        }
    }
}
```

```

    }

    // copy directly to memory
    memcpy(pPixelAddress, &screenColor, surface->format->BytesPerPixel);

    if (SDL_MUSTLOCK(surface)) {
        SDL_UnlockSurface(surface);
    }

}

void drawLine(SDL_Surface *surface, int x0, int y0, int x1, int y1,
              SDL_Color color) {
    // . . . ADD YOUR CODE HERE . .
}

```

 Add your code for the simple line drawing algorithm

 Your screen should look like the one below:

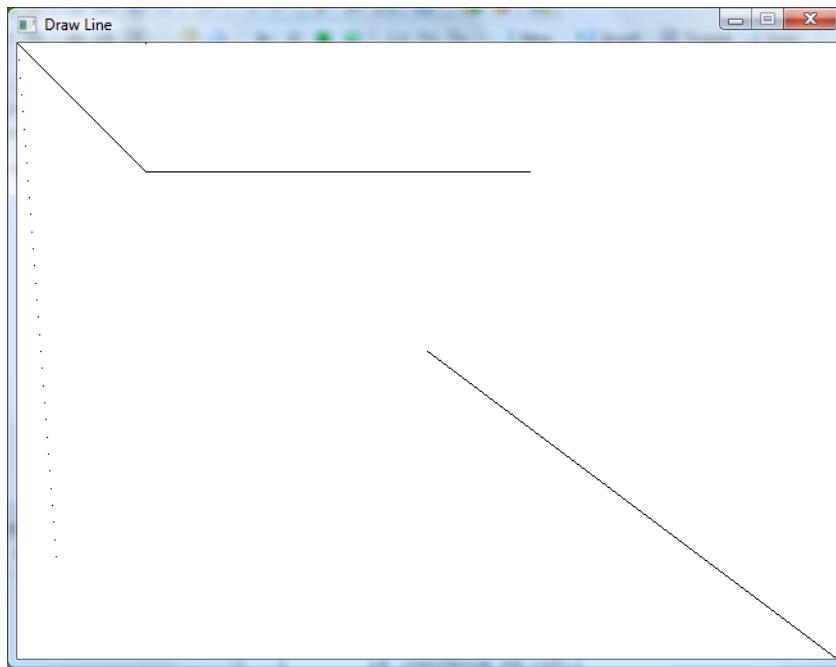
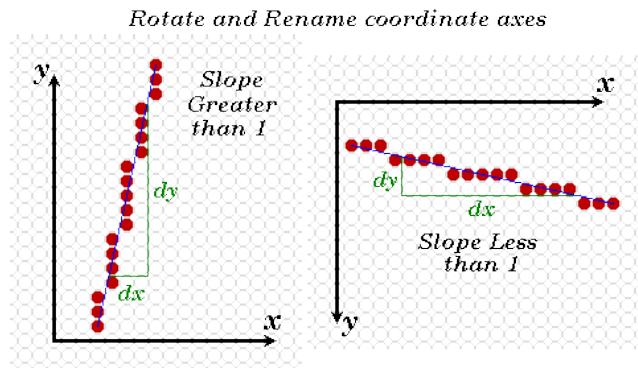


Figure 71 - Draw line algorithm #1

The program prints 5 lines but only one 3 of the lines display accurately – that is satisfying the criteria we have established for a good line drawing algorithm.. The vertical lines do not show since $x_0=x_1$ and $dx = 0$ which makes computing the slope impossible. The other line that appears more like a dotted line does not meet of the criteria of appearing continuous. This occurs when the slope > 1 .

A Simple Algorithm #2 – Using Symmetry

We will use symmetry to solve the line drawing problem when $m > 1$. The assigning of one coordinate axis the name x and the other y is arbitrary. If $m > 1$ in one orientation (e.g. $3/2$) than it is less than 1 if we switched orientations. (note for our screen drawings we actually used the second orientation!). We will modify the algorithm to switch the use of x and y when the slope > 1 .



```

int dx = x1 - x0;
int dy = y1 - y0;
drawPixel(surface, x0, y0, color);           // plot the first point
if (abs(dx) > abs(dy)) {                   // handles slope < 1
    float m = (float) dy / (float) dx; // calculate the slope
    float b = y0 - m * x0;           // compute the y-intercept
    dx = (x1 > x0) ? 1 : -1;
    while ( x0 != x1 ) {
        x0 += dx;                  // next x value
        y0 = round(m*x0 + b); // corresponding y value
        drawPixel(surface, x0, y0, color);
    }
} else {                                     // handles slope >= 1
    float m = (float) dx / (float) dy;
    float b = x0 - m * y0;
    dy = (y1 > y0) ? 1 : -1;
    while (y0 != y1) {
        y0 += dy;
        x0 = round(m*y0 + b);
        drawPixel(surface, x0, y0, color);
    }
}

```

LAB #6: Program 2_6 – Slope-Intercept Algorithm Improvement

- ⊕ Create a new project named Program2_6 using the template Simple SDL Project template.
- ⊕ Enter the same program that was used in Program 2_5
- ⊕ Enhance your previous drawLine function to handle lines with slope ≥ 1 .

You can see that all five lines are now drawn. The line going from (0,0) to (30,400) now looks more continuous but jagged! There are algorithms online (see Xiaolin Wu’s Line Algorithm) that make the line appear “straighter” by doing what is called anti-aliasing.

We really are not done since the algorithm could be greatly improved or what programmers call optimized by making some simple changes. Optimization will improve the speed in which the line is drawn. If you want to continue working on the example I invite you to check out the web for ideas and better algorithms.

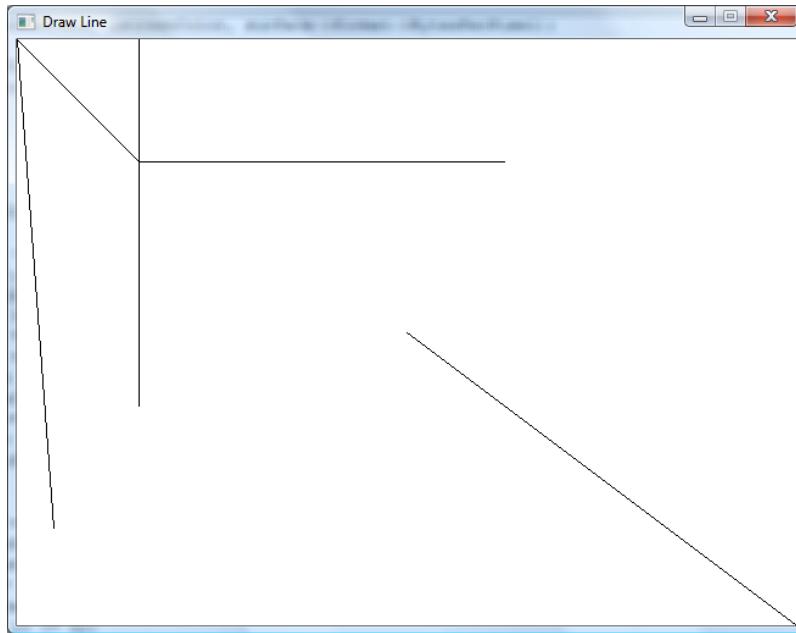


Figure 72 - Draw line algorithm #2

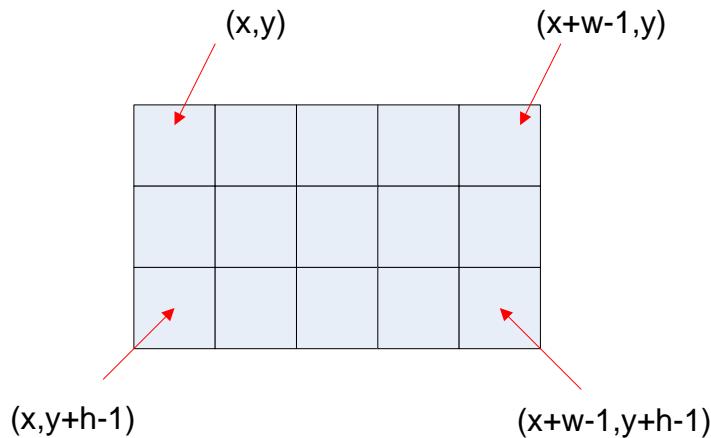
See the exercises at the end of this Chapter for ideas on how to draw circles.

SDL_Rect

The next SDL structure to learn is `SDL_Rect`. This is an essential data structure since any graphics element you want to draw on the screen has to fit within a rectangular space – yes that goes for game paddle, image of monsters and even balls!

```
typedef struct SDL_Rect {  
    Sint16 x, y;  
    Uint16 w, h;  
} SDL_Rect;
```

The structure contains four members, the upper-left position of the rectangle specified by x and y and the rectangle width (w) and height (h). Using the information in `SDL_Rect` you can compute the other edge points of the rectangle as illustrated:



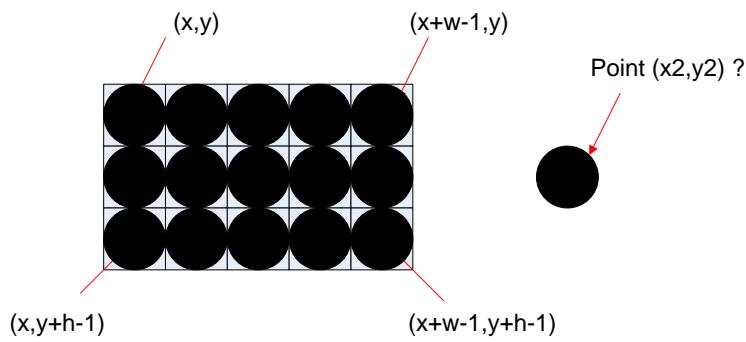
[Figure 73 - `SDL_Rect`](#)

The key figure below illustrates the actual grid positions that would be affected by creating an `SDL_Rect` starting at (10,10) with w=5 and h=3 are

```
(10,10) (11,10) (12,10) (13,10) (14,10)
(10,11) (11,11) (12,11) (13,11) (14,11)
(10,12) (11,12) (12,12) (13,12) (14,12)
```

Note, how the pixels that are part of the rectangle go from (x,y) . . . (x+w-1, y) NOT (x+w)

The way the math works the actual column x+w is not part of our rectangle nor is the row y+h. It is easy to determine if a Point P (x2, y2) is inside or outside the rectangle.



[Figure 74 - Determining if a point P\(x2, y2\) is inside or outside?](#)

To determine if $\text{Point}(x2, y2)$ is inside or outside the rectangle one need only use the following formula:

```
if ( x2 >= x && x2 < (x+w) && y2 >= y && y2 < (y+h) ) {
```

```
// inside the rectangle
} else {
    // outside the rectangle
}
```

Why do you care about something so simple? Well, we would like to be able to determine when one rectangular object has collided with another so we have to first understand how to test a simple point. We will return to this topic later.

If the previous lab exercises I actually used the `SDL_Rect` structure to make the entire screen white by using the following lines:

```
// Make the background screen white
SDL_Rect screenRect = {0,0, SCREEN_WIDTH, SCREEN_HEIGHT};
Uint32 color = SDL_MapRGB(pDisplaySurface->format, COLOR_WHITE.r,
                           COLOR_WHITE.g, COLOR_WHITE.b);
SDL_FillRect(pDisplaySurface, &screenRect, color);
```

The code creates an `SDL_Rect` variable – `screenRect` with the dimensions of the screen, then uses the `SDL_MapRGB` to obtain the actual screen color for white (using the constant we created `COLOR_WHITE` that already specified the r, g, and b values required to obtain the color white). Finally, the code used a new function `SDL_FillRect` that takes our display surface pointer, the address of our rectangular area (variable `screenRect`) and color to make in our case the entire screen white.

Function Name: **`SDL_FillRect`**

Format:

```
int SDL_FillRect(SDL_Surface *pDisplaySurface, SDL_Rect * rectArea, Uint32 color);
```

Description:

This function performs a “fast fill” of the given rectangle with the color specified. If `rectArea` = `NULL` then the entire display area is used (note, the `SDL_Surface` structure hold the w and h of the surface). The function returns 0 on success or -1 on error.

From the description above we could have made our screen white using only the following 2 lines of code:

```
// Make the background screen white
Uint32 color = SDL_MapRGB(pDisplaySurface->format, COLOR_WHITE.r,
                           COLOR_WHITE.g, COLOR_WHITE.b);
SDL_FillRect(pDisplaySurface, NULL, color);
```

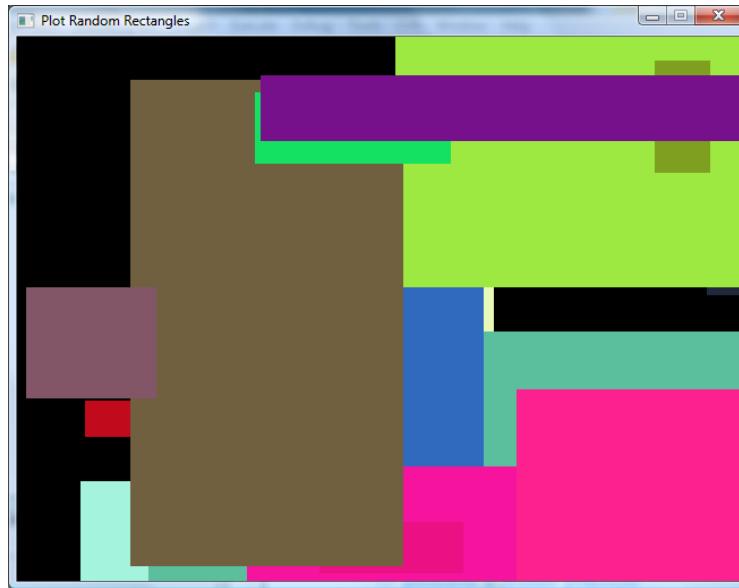


Figure 75 - Generating Random Rectangles

LAB #7: Program 2_7 – Generating Random Rectangles

- ✚ Create a new project named Program2_7 using the template Simple SDL Project template.
- ✚ Enter the same program that was used in Program 2_4
- ✚ Remove the function drawPixel and its prototype
- ✚ Add code (after generating a random screen position) to generate a random width and height
- ✚ After the code that generates a random color into an `SDL_Color` use `SDL_MapRGB` to create the actual `screenColor`.
- ✚ Add code to create an `SDL_Rect` variable named `randomRect` using the `x,y, w and h` values that were randomly generated
- ✚ Add code to call `SDL_FillRect`
- ✚ You may want to use `SDL_Delay` to delay for 300 milliseconds so you can see each rectangle being drawn as shown in the figure above.
- ✚ Compile and run the program. Your screen should look like the one above.

Clipping

Clipping is the process of dividing the surface into two areas – its visible and invisible parts. By default when we create the display surface using `SDL_SetVideoMode` we can write to the entire screen – that is, the entire screen is visible. The `SDL_Surface` has a component named `clip_rect` which defines the surface clipping rectangle. We can change the portion of the screen that gets updated by changing this clipping rectangle. For example, suppose we wanted to add a screen border to the previous program. We can use the new function `SDL_SetClipRect` to define the new rectangular area that will be “visible” or changeable by our random rectangles programs.

Function Name: **SDL_SetClipRect**

Format:

```
void SDL_SetClipRect(SDL_Surface *pDisplaySurface, SDL_Rect * rectArea);
```

Description:

This function sets the clipping rectangle for a surface (it is not limited to our display surface). Only those pixels that fall into the clipping area specified by rectArea will be displayed. If rectArea is set to NULL the clipping area will be set to the full size of the surface.

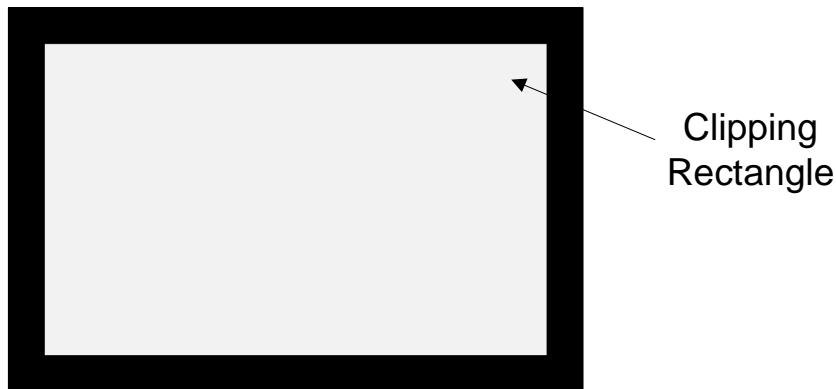


Figure 76 - Example of creating a clipping rectangle

Anything you draw to the display screen will be restricted to the clipping rectangle, that is, anything outside of the clipping area will not be drawn.

LAB #8: Program 2_8 – Generating Random Rectangles with a Border

- ⊕ Create a new project named Program2_8 using the template Simple SDL Project template.
- ⊕ Enter the same program that was used in Lab #7.
- ⊕ We will create a 20 pixel border on the top, bottom, left and right on the display by creating a clipping rectangle similar to the one shown in Figure 76.
- ⊕ Add code after creating the pDisplaySurface to set the clipping rectangle by creating an `SDL_Rect` that starts at (20,20) and has a width of `SCREEN_WIDTH-40` and height of `SCREEN_HEIGHT-40`. Invoke `SDL_SetClipRect`.
- ⊕ Compile and execute you should let the program run a while so that it fills out the screen. You should get screen display similar the one below.



Figure 77 - Setting a border using a clipping rectangle

SDL_VideoInfo

Information about the video display can be obtained by using the SDL function **SDL_GetVideoInfo**, the function returns an **SDL_VideoInfo** structure.

Function Name: **SDL_GetVideoInfo**

Format:

```
const SDL_VideoInfo* SDL_GetVideoInfo(void);
```

Description:

This function returns a READ-ONLY pointer to a structure containing information about the video hardware. If you call it before invoking **SDL_SetVideoMode** the structure member **vfmt** will contain the pixel format of the BEST video mode.

```
/** Useful for determining the video hardware capabilities */
typedef struct SDL_VideoInfo {
    Uint32 hw_available :1; /*< Flag: Can you create hardware surfaces? */
    Uint32 wm_available :1; /*< Flag: Can you talk to a window manager? */
    Uint32 UnusedBits1 :6;
    Uint32 UnusedBits2 :1;
    Uint32 blit_hw :1; /*< Flag: Accelerated blits HW --> HW */
    Uint32 blit_hw_CC :1; /*< Flag: Accelerated blits with Colorkey */
    Uint32 blit_hw_A :1; /*< Flag: Accelerated blits with Alpha */
    Uint32 blit_sw :1; /*< Flag: Accelerated blits SW --> HW */
    Uint32 blit_sw_CC :1; /*< Flag: Accelerated blits with Colorkey */
```

```

Uint32 blit_sw_A      :1; /*< Flag: Accelerated blits with Alpha */
Uint32 blit_fill       :1; /*< Flag: Accelerated color fill */
Uint32 UnusedBits3     :16;
Uint32 video_mem; /*< The total amount of video memory (in K) */
SDL_PixelFormat *vfmt; /*< Value: The format of the video surface */
int    current_w; /*< Value: The current video mode width */
int    current_h; /*< Value: The current video mode height */
} SDL_VideoInfo;

```

LAB #9: Program 2_9 – Displaying video information before and after calling `SDL_SetVideoMode`.

- ⊕ Create a new project named `Program2_9` using the template Simple SDL Project template.
- ⊕ Add the following two functions and the required prototypes:

```

void writeVideoInfo(void) {
    const SDL_VideoInfo *vInfo = SDL_GetVideoInfo();
    if (vInfo->hw_available)
        cout << "hw available, can create hardware surfaces" << endl;
    else
        cout << "hw NOT available, cannot create hardware surfaces"
              << endl;

    if (vInfo->wm_available)
        cout << "wm available, can talk to window manager" << endl;
    else
        cout << "wm NOT available, cannot talk to window manager"
              << endl;

    if (vInfo->blit_hw)
        cout << "blit_bw, supports accelerated blits HW-->HW" << endl;
    else
        cout << "blit_bw, DOES NOT support accelerated blits HW-->HW"
              << endl;

    if (vInfo->blit_hw_CC)
        cout << "blit_bw_CC, supports accelerated blits with Colorkey"
              << endl;
    else
        cout << "blit_bw_CC, DOES NOT support accelerated blits with
Colorkey" << endl;

    if (vInfo->blit_hw_A)
        cout << "blit_bw_A, supports accelerated blits with Alpha" <<
              endl;
    else
        cout << "blit_bw_A, DOES NOT supports accelerated blits with
Alpha" << endl;

    if (vInfo->blit_sw)
        cout << "blit_sw, supports accelerated blits SW-->SW" << endl;
    else
        cout << "blit_sw, DOES NOT support accelerated blits with SW-->SW"
              << endl;
}

```

```

        if (vInfo->blit_sw_CC)
            cout << "blit_sw_CC, supports accelerated blits SW-->SW with
Colorkey" << endl;
        else
            cout << "blit_sw_CC, DOES NOT support accelerated blits with SW-
->SW with Colorkey" << endl;

        if (vInfo->blit_sw_A)
            cout << "blit_sw_A, supports accelerated blits SW-->SW with
Alpha" << endl;
        else
            cout << "blit_sw_A, DOES NOT support accelerated blits with SW-->SW with Alpha" << endl;

        if (vInfo->blit_fill)
            cout << "blit_fill, supports accelerated color fill" << endl;
        else
            cout << "blit_fill, DOES NOT support accelerated color fill" <<
endl;

        cout << "Total number of video memory: " << vInfo->video_mem << endl;
        writePixelFormatInfo (vInfo->vfmt);
        cout << "Video mode width: " << vInfo->current_w << endl;;
        cout << "Video mode height: " << vInfo->current_h << endl;

    }

void writePixelFormatInfo(SDL_PixelFormat* format) {
    cout << "BitsPerPixel: " << format->BitsPerPixel << endl;
    cout << "BytesPerPixel: " << format->BytesPerPixel << endl;
}

```

- ✚ Add code in the main function to invoke writeVideoInfo
- ✚ Compile and run
- ✚ Check the file stdout in the same directory as the executable

Loading Images

SDL provides a function to load bmp images – **SDL_LoadBMP**.

Function Name: **SDL_LoadBMP**

Format:

SDL_Surface *SDL_LoadBMP(const char *file);

Description:

The function opens a file image saved as a windows bmp file (*.bmp). Note: When loading a 24-bit Windows BMP file, pixel data points are loaded as blue, green, red (not the expected red, green and blue). The function returns a pointer to an **SDL_Surface** or **NULL** if an error occurred.

There are several things to remember when using this function:

- ⊕ You will need to copy the image to the screen in order to see it
- ⊕ You must release all surfaces before exiting the program

The second point is easy to do since we already know how to use the `SDL_FreeSurface` function.

The first point requires that we use do what is referred to as blitting which stands for bit blit or bit-block (image) transfer. The operation is used to move a bitmap image from a source to a destination. The SDL function we will use is `SDL_BlitSurface`.

Function Name: **`SDL_BlitSurface`**

Format:

```
int SDL_BlitSurface(SDL_Surface *srcSurface, SDL_Rect *srcRect, SDL_Surface
*destSurface,
SDL_Rect *destRect);
```

Description:

This function performs fast blit from the surface to the destination surface. The w and h is srcRect determine the size of the copied rectangle, only the position is used in destRect. If the srcRect is NULL, then the entire srcSurface is copied. If destRect is NULL, then the destination position is (0,0).

If the function is successful it returns 0, otherwise it returns -1.

LAB #10: Program 2_10 – Displaying a ball.

- ⊕ Create a new project named Program2_10 using the template Simple SDL Project template.
- ⊕ Change the window caption to “Showing a Ball”
- ⊕ Add code to load the image `smallball1.bmp` into an `SDL_Surface` variable named `pBallImage` and test if loading was successful
- ⊕ Create a `Uint32` variable named `backgroundColor` that will be set to the color white (255,255,255) using `SDL_MapRGB`
- ⊕ Add code within the loop (if `SDL_PollEvent(&event) == 0` is true) to
 - first make the screen background white using `SDL_FillRect`
 - display the ball using `SDL_BlitSurface` to the screen
- ⊕ Add code at the bottom (before `SDL_FreeSurface`) that frees the `pBallImage`.
- ⊕ Compile and run

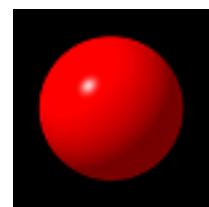


Figure 78 -
`smallball1.bmp`

You should see a screen similar to the one below:

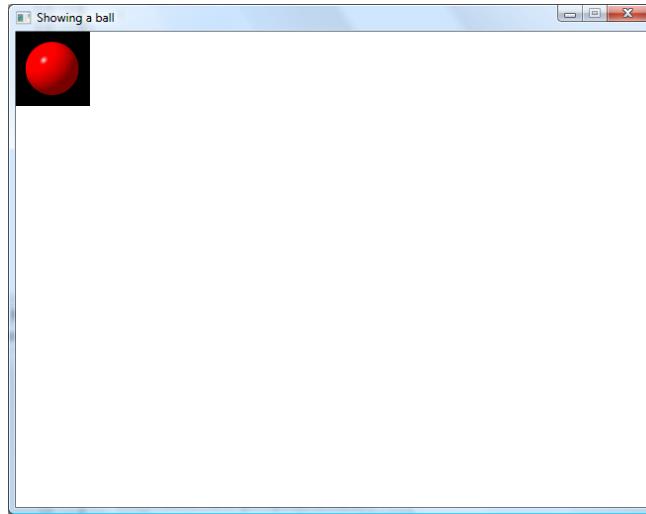


Figure 79 - Displaying the ball - Yuck!

Yuck! We really don't want to see the black portion of the ball image. This should be transparent that is invisible when drawn on the screen. This will require that you specify what the transparency color is on your image, that is, you must tell SDL what portions of your image should not be drawn to the screen. When the image gets drawn or blit to the screen the transparency colors will be ignored so that the background color shows through. The function to use to inform SDL what the transparency color for an image stored in an `SDL_Surface` is `SDL_SetColorKey`.

Function Name: **`SDL_SetColorKey`**

Format:

```
int SDL_SetColorKey(SDL_Surface *surface, Uint32 flag, Uint32 key);
```

Description:

This function sets the color key (transparent pixel) and allows you to enable or disable RLE bit acceleration – that is the flag value can be set to `SDL_SRCCOLORKEY` and/or `SDL_RLEACCEL` or both. The function returns 0 if successful or -1 if an error occurs.

RLE acceleration “is a process used to decrease the time required to draw a color keyed image to the screen.”

RLE stands for run-length encoding. RLE is a simple form of data compression (e.g. like what you do when you zip a file). The way it works is that runs of data – a sequence where the same data value appears (e.g. AAAAABBBBBAAAAAAA) are stored as a single data value and count (e.g 6A5B7A will represent the previous string).

Example Usage:

```
int retValue = SDL_SetColorKey(pImageSurface,  
    SDL_SRCCOLORKEY | or SDL_RLEACCEL, transparencyColor);
```

LAB #11: Program 2_11 – Displaying a ball using SetColorKey

- ✚ Create a new project named Program2_11 using the template Simple SDL Project template.
- ✚ Copy the code from the previous lab (LAB #10)
- ✚ Add the following lines after reading in the image:

```
// Set the transparent color  
Uint32 transparentColor = SDL_MapRGB(pDisplaySurface->format, 0, 0, 0);  
SDL_SetColorKey(pBallImage, SDL_SRCCOLORKEY, transparentColor);
```

- ✚ Compile and run

The program now looks like this:

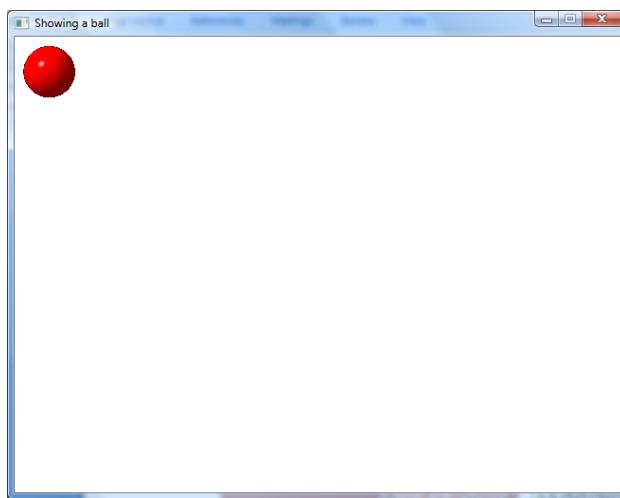


Figure 80 - Setting transparent color

Moving a “Ball” around on the screen

In the next program we will move a ball (actually a bmp image). The image will be slightly smaller ball from the image used in the previous two programs. We plan on evolving this program into a pong game. We will not worry about the details of using a more object-oriented style just yet. We will discuss a more ideal style and better way to organize our programs²² in the upcoming chapters.

²² The better style comes from the web site sdl.tutorials.com

The program moves a ball around the screen. When the ball hits one of the walls (end of the screen) the ball will bounce off and change direction.

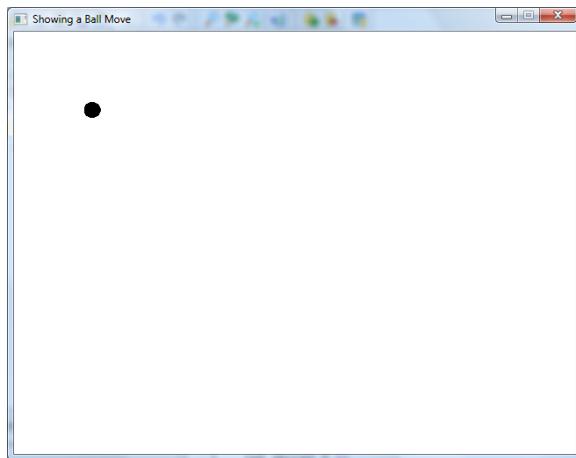


Figure 81 - The ball moving south-east on the screen

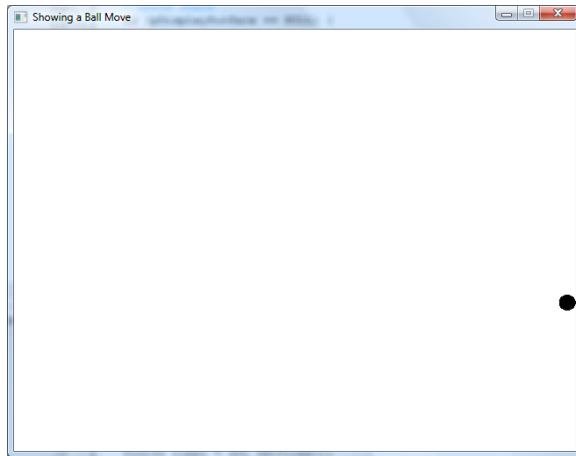


Figure 82 - The ball coming off the wall

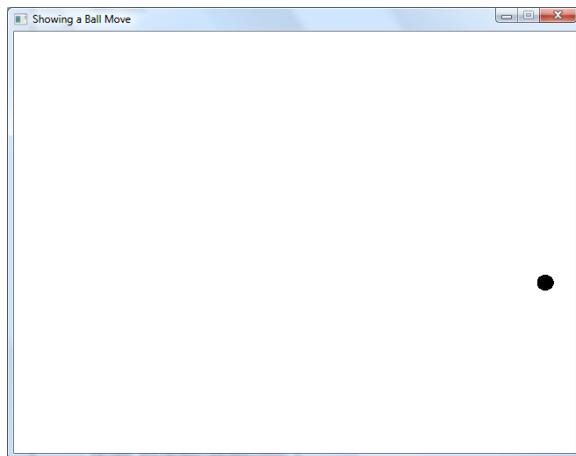


Figure 83 - The ball bounces off the wall

The program will require that we do the following:

- ⊕ Place the ball at location (0,0) on the screen
- ⊕ Set the speed in which the ball will travel on the screen
- ⊕ Detect when the ball hits the wall
 - Reverse the direction the ball is moving

You actually have all the tools to create this program. You will need to add two new constants to your program FRAMES_PER_SECOND and FRAME_RATE. The program will be drawing and erasing the ball on the screen. We don't want the screen to update so fast that we never really see the ball moving (what fun is that!). We will initially set the value to:

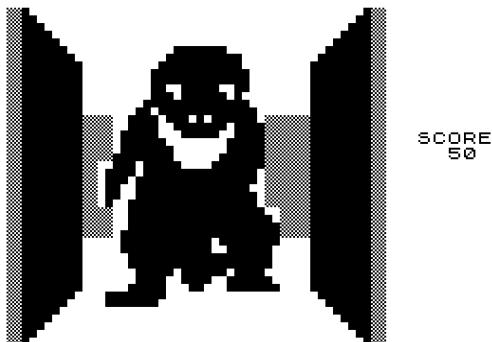
```
// refresh rate
const int FRAMES_PER_SECOND = 30;
const int FRAME_RATE = 1000 / FRAMES_PER_SECOND;
```

The value of 30 will be used to delay the actions of drawing and erasing by having the computer refresh the screen 30 times per second. If we want to refresh the screen 30 times a second then in milliseconds it will be required to update the screen every $1000 / 30$ or 33.34 milliseconds.

From Wikipedia: [Everything you wanted to know about frame rates but were afraid to ask!](#)

s

A common term that is mentioned with respect to games is *frame rate*. This is the frequency in which the image on the computer screen is being updated. The frame rate is usually expressed as frames per second (fps).



Frame rates are considered important in video games. The frame rate can make the difference between a game that is playable and one that is not. The first 3D first-person adventure game for a personal computer, [3D Monster Maze](#), had a frame rate of approximately 6 fps, and was still a success, being playable and addictive. In modern action-oriented games where players must visually track animated objects and react quickly, frame rates of between 30 to 60 fps are considered minimally acceptable by some, though this can vary significantly from game to game. Most modern action games, including popular first person shooters such as Halo 3, run around 30 frames a second, while others, such as Call of Duty 4, run at 60 frames a second. The frame

rate within most games, particularly PC games, will depend upon what is currently happening in the game.

QUESTION: If we are moving the ball every 60 milliseconds on the screen, what is the perceived frame rate?

A culture of competition has arisen among game enthusiasts with regards to frame rates, with players striving to obtain the highest fps count possible. Indeed, many benchmarks released by the marketing

departments of hardware manufacturers and published in hardware reviews focus on the fps measurement. Modern video cards, often featuring [NVIDIA](#) or [ATI](#) chipsets, can perform at over 160 fps on graphics intensive games such as [F.E.A.R.](#). One single GeForce 8800 GTX has been reported to play [F.E.A.R.](#) at up to 386 fps (at a low resolution). This does not apply to all games: some games apply a limit on the frame rate. For example, in the [Grand Theft Auto](#) series, [Grand Theft Auto III](#) and [Grand Theft Auto: Vice City](#) have a standard 30 fps ([Grand Theft Auto: San Andreas](#) runs at 25 fps) and this limit can only be removed at the cost of graphical and gameplay stability. It is also doubtful whether striving for such high frame rates are worthwhile. An average 17" monitor can reach 85 Hz, meaning that any performance reached by the game over 85 fps is discarded. For that reason it is not uncommon to limit the frame rate to the refresh rate of the monitor in a process called [vertical synchronization](#). However, many players feel that not synchronizing every frame produces sufficiently better game execution to justify some "[tearing](#)" of the images.

It should also be noted that there is a rather large controversy over what is known as the "feel" of the game frame rate. It is argued that games with extremely high frame rates "feel" better and smoother than those that are just getting by. This is especially true in games such as a [first-person shooter](#). There is often a noticeable choppiness perceived in most computer rendered video, despite it being above the flicker fusion frequency (as, after all, one's eyes are not synchronized to the monitor).

This choppiness is not a perceived flicker, but a perceived gap between the object in motion and its [afterimage](#) left in the eye from the last frame. A computer samples one point in time, then nothing is sampled until the next frame is rendered, so a visible gap can be seen between the moving object and its afterimage in the eye. Many driving games have this problem, like [NASCAR 2005: Chase for the Cup](#) for Xbox, and [Gran Turismo 4](#). The polygon count in a frame may be too much to keep the game running smoothly for a second. The processing power needs to go to the polygon count and usually takes away the power from the framerate.

The reason computer rendered video has a noticeable afterimage separation problem and camera captured video does not is that a camera shutter interrupts the light two or three times for every film frame, thus exposing the film to 2 or 3 samples at different points in time. The light can also enter for the entire time the shutter is open, thus exposing the film to a continuous sample over this time. These multiple samples are naturally interpolated together on the same frame. This leads to a small amount of [motion blur](#) between one frame and the next which allows them to smoothly transition.

An example of afterimage separation can be seen when taking a quick 180 degree turn in a game in only 1 second. A still object in the game would render 60 times evenly on that 180 degree arc (at 60 Hz frame rate), and visibly this would separate the object and its afterimage by 3 degrees. A small object and its afterimage 3 degrees apart are quite noticeably separated on screen.

The solution to this problem would be to interpolate the extra frames together in the back-buffer (field [multisampling](#)), or simulate the motion blur seen by the [human eye](#) in the rendering engine. When vertical sync is enabled, video cards only output a maximum frame rate equal to the refresh rate of the monitor. All extra frames are dropped. When vertical sync is disabled, the video card is free to render frames as fast as it can, but the display of those rendered frames is still limited to the refresh rate of the monitor. For example, a card may render a game at 100 FPS on a monitor running 75 Hz refresh, but no more than 75



FPS can actually be displayed on screen.

Certain elements of a game may be more [GPU](#)-intensive than others. While a game may achieve a fairly consistent 60 fps, the frame rate may drop below that during intensive scenes. By achieving frame rates in excess of what is displayable, it makes it less likely that frame rates will drop below what is displayable during stressful scenes.

The ball needs to be adjusted or moved. The two variables the program uses to perform this is deltaX and deltaY. The two variables indicate how fast the ball is moving in the x and y direction. The larger the values the faster the ball is moving in the x or y direction.

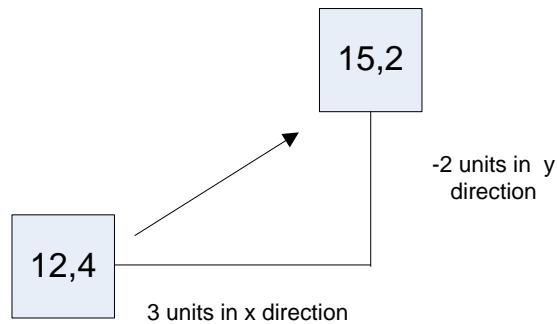


Figure 84 - moving the ball on the screen

In the figure depicted above the ball is at position 12,4²³ on the screen. The deltaX value is +3. That is the x position will be adjusted by adding +3 units to the current x position of the ball. This translates to the ball moving towards the left. The deltaY value is -2 which means it is moving up the screen that is the y position of the ball will decrease. This is how we would see the ball moves up and to the right on the screen. The problem is we have to check and adjust for when the ball hits the edge of the screen, that is, the x position value is below 0 or greater than SCREEN_WIDTH or the y position value is below 0 or greater than SCREEN_HEIGHT. In order to prevent this we will need to move the ball and make adjustments if the ball has fallen off the screen. The adjustment we will make will be to position the ball back to the screen and reverse the deltaX and deltaY value (whichever one needs adjusting) so the ball starts moving back on the screen.

So for example let's suppose the ball is at (18, 0) and the deltaX is +3 and deltaY is -2. The program will compute the next position of ball should be (21, -2). But, (21,-2) is off the screen due to the negative y coordinate value. The code will readjust the position to (21, 0) and change the deltaY value to +2. So on the next screen update for the ball the new position will be (24, 2), that is, the ball at first move up, moves off the top and get readjusted back to the screen and the deltaY changed so the ball starts moving down again. It will look to the user as if the ball bounces off the top.

We will use the `SDL_Rect` struct to hold the position of the ball.

The code to change and adjust the ball position follows:

²³ This (X,Y) value is the coordinate for the top-left position defining the rectangle the ball image occupies.

```

        // move the ball
newBallLocation = ballLocation; // copy current ball loc
newBallLocation.x += deltaX;      // move x
newBallLocation.y += deltaY;      //     then y
if (newBallLocation.x < 0) {
    // we are off to the left - adjust the ball (x)
    newBallLocation.x = 0;
    deltaX = -deltaX;
}
if (newBallLocation.x > SCREEN_WIDTH - pBallImage->w) {
    // are we off to the right - adjust the ball (x)
    newBallLocation.x = SCREEN_WIDTH - pBallImage->w;
    deltaX = -deltaX;
}
if (newBallLocation.y < 0) {
    // are we off the top - adjust the ball (y)
    newBallLocation.y = 0;
    deltaY = -deltaY;
}
if (newBallLocation.y > SCREEN_HEIGHT - pBallImage->h) {
    // are we off the bottom - adjust the ball(y)
    newBallLocation.y = SCREEN_HEIGHT - pBallImage->h;
    deltaY = -deltaY;
}
ballLocation = newBallLocation;
}

```

We save the current location of the ball in an `SDL_Rect`, `ballLocation`, where the `x` and `y` members of this struct hold the top left location of where the ball is drawn on the screen. The `newBallLocation` is also an `SDL_Rect` and every time we have waited `REFRESH_RATE` time we do the following in the above code:

- ➊ Copy the `ballLocation` into `newBallLocation`
- ➋ Adjust the `x` and `y` with `deltaX` and `deltaY`, respectively
- ➌ Check if the ball has bounced off the wall and adjust
- ➍ Copy `newBallLocation` into `ballLocation`

Compare the above list with the code above the third dot point actually accounts for most of the code to determine if the ball has moved off the left, right, top or bottom and adjust.

```

if (newBallLocation.x < 0) {
    // we off to the left - adjust the ball (x)
    newBallLocation.x = 0;
    deltaX = -deltaX;
}

```

The above checks if the `newBallLocation.x` position is off to the left (less than 0) if so then we have to get that ball back on the screen by resetting the `x` location to 0 and reversing the `deltaX`.

The check for going off the right is a bit more involved.

```
if (newBallLocation.x > SCREEN_WIDTH - pBallImage->w) {
    // are we off to the - adjust the ball (x)
    newBallLocation.x = SCREEN_WIDTH - pBallImage->w;
    deltaX = -deltaX;
}
```

We always want to see the ball on the screen so we are *not* going to wait for it to partially go off to the right. We check if the newBallLocation.x leaves enough room for the ball’s current width to display entirely on the screen, if not, we adjust the x location and reverse deltaX. The code for checking the top and bottom are similar in spirit.

LAB #12: Program 2_12 – Bouncing a ball around the screen

- ⊕ Create a new project named Program2_12 using the template Simple SDL Project template.
- ⊕ Obtain the new ball image smallball1.bmp
- ⊕ Copy the code from the previous lab (LAB #11)
- ⊕ After the screen dimension consts at the top of the program add two new const int named FRAMES_PER_SECOND and FRAME_RATE. Initialize as discussed in the last section.
- ⊕ Change the caption for the program to “Showing a Ball Move”
- ⊕ Change the SDL_LoadBMP argument to get the smallball1.bmp (black ball with a white background)
- ⊕ Change the transparent color to black (255, 255, 255)
- ⊕ After the code that creates the creates and initializes the backgroundColor to white use the SDL_FillRect function to set the entire display to the background color:

```
SDL_FillRect(pDisplaySurface, NULL, backgroundColor);
```

- ⊕ Create an SDL_Rect variable named ballLocation that will be initialized to {0,0,0,0}
- ⊕ Create the int variables deltaX and deltaY and initialize to 2
- ⊕ You now need to record the current time. How?

SDL has a function named **SDL_GetTicks()** that proves useful for managing the frame rate.

Function Name: **SDL_GetTicks()**

Format:

```
Uint32 SDL_GetTicks(void);
```

Description:

This function returns the number of milliseconds since the SDL library initialization. It will wrap around if the program runs for more than 49 days.

We will use this function to get the current time and check in the for loop to determine if the current time – the old time exceeds the FRAME_RATE (translation: check if 33.34 milliseconds has elapsed).

Create a Uint32 variable named timer before the loop that is set to the value returned by `SDL_GetTicks()`;

- ⊕ Add code in the if section
 - Create `SDL_Rect` for newBallLocation
 - Add code to get the current time (Use `SDL_GetTicks()`)
 - Add a new if statement that takes the difference between the timer variable and the current time and checks if it is greater than `FRAME_RATE`
 - The body of this new ifStatement should
 - Update the timer variable to the current time
 - Perform the ball update code discussed in this section
- ⊕ After the ifStatement make sure you have the following code to update the screen
 - Make the background white
 - Draw the ball
 - Update the screen

```
// first make the background white
SDL_FillRect(pDisplaySurface, &oldBallLocation, backgroundColor);
SDL_BlitSurface(pBallImage, NULL, pDisplaySurface, &ballLocation);

//update the screen
SDL_UpdateRect(pDisplaySurface, 0, 0, 0, 0);
```

- ⊕ Compile and run

Play around with the program by updating the `deltaX` and `deltaY` in combination with the `FRAMES_PER_SECOND`.

Double Buffering and Page Flipping

In the next program for this chapter will use an alternative method for updating the screen – double buffering. Double buffering is a technique for drawing graphics that show no flicker or tearing on the screen. The way it works is not to update the screen directly but to create an updated version of the screen in another area (a buffer) and when you have finished moving the aliens, killing or removing the debris and moving the player you then move or copy the updated screen to the video screen in one step or as quickly as possible when the video monitor is moving to reset to draw a new screen.

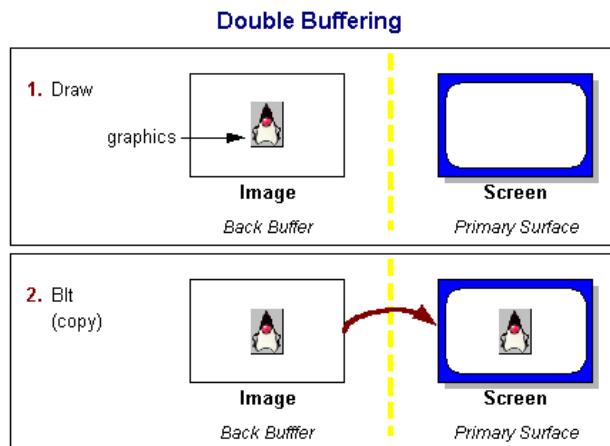


Figure 85 - Double buffering

Double Buffering

In double buffering we reserve an area in memory (RAM) that we update and then copy or what most programmers refer to as blit (bit blit or bit block) the entire memory area into video memory.

Page Flipping

In page flipping the buffer is in video ram (VRAM) so copying into video memory is not required all we have to do is switch the video pointer to point to the updated screen.

The current screen comes from the video display showing the contents of the primary surface. While it is displayed all changes are made to the back buffer. The back buffer can be in RAM (double-buffering) or VRAM (page-flipping). When the back buffer is completely updated either you blit or copy into VRAM or the video pointer is switched over to the back buffer.

When using double-buffering the memory buffer is always being updated and moved or copied into VRAM, when using page flipping the VRAM area being used for updates switches back and forth.

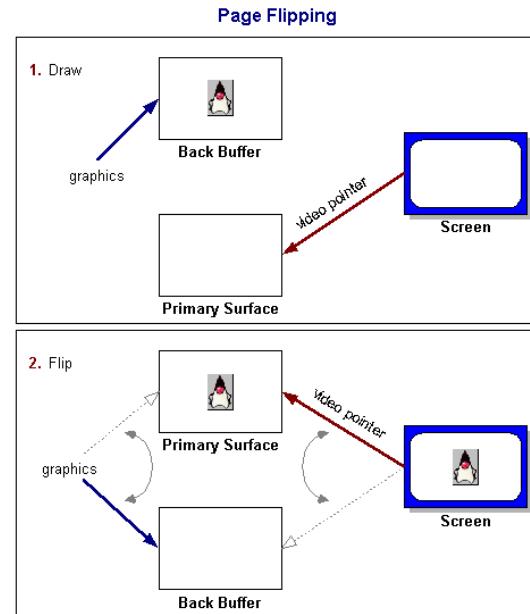


Figure 86 - Page Flipping

Images from

<http://java.sun.com/docs/books/tutorial/extr fullscreen/doublebuf.html>).



Figure 87 - Imaging illustrating "tearing"

The figure above shows where the top image is being updated but not complete and the user sees the bottom half of the previous image. Yikes!

In order to avoid tearing as seen above or flickering the update to the screen has to wait for the *best* time to update.

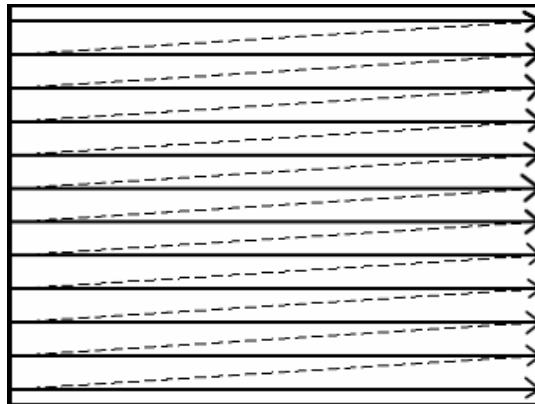


Figure 88 - Waiting for vertical re-trace

The way the video monitor gets displayed is by starting at the top-left and refreshing each scanline. As the beam moves from the one end the beginning of the next line this is called – horizontal retrace. When the beam reaches the end of the last line the beam has to move back to the top to start the screen refresh over again. The movement of the beam from bottom-right back to the top-left is called the vertical retrace. The double buffering is ideal when the screens are switched while the monitor is going through vertical retrace.

The way we do this in SDL is to use the `SDL_DOUBLEBUF` as one of the flags when you use `SDL_SetVideoMode` as in:

```
//create windowed environment
pDisplaySurface = SDL_SetVideoMode(SCREEN_WIDTH, SCREEN_HEIGHT, 0,
                                    SDL_ANYFORMAT | SDL_DOUBLEBUF );
```

Another option that you may want to use is `SDL_HWSURFACE`.

LAB #13: Program 2_13 – Bouncing a ball around the screen with double buffering

- ✚ Create a new project named Program2_13 using the template Simple SDL Project template.
- ✚ Copy the program from the previous lab.
- ✚ Change the `SDL_SetVideoMode` by adding the flag `SDL_DOUBLEBUF`
- ✚ Replace the `SDL_UpdateRect` command with `SDL_Flip` command

Function Name: **SDL_Flip()**

Format:

```
int SDL_Flip(SDL_Surface *displaySurface);
```

Description:

This function flips the screen using double-buffering if the hardware supports it. The function returns 0 on success, otherwise it returns -1 on error.

- ⊕ Compile and run the program.

Displaying Other Types of Images

SDL easily handles bmp images but you often encounter other file formats for images. The popular ones are:

- ⊕ PCX – is an older image file format developed by ZSoft Corporation for what was once a leading paint program named PC Paintbrush in the 1980's. PCX stands for “Personal Computer eXchange”. It quickly became a popular file format for images when DOS was king of the operating systems. You don't see too many images in this format (in fact I obtained the screen copy of

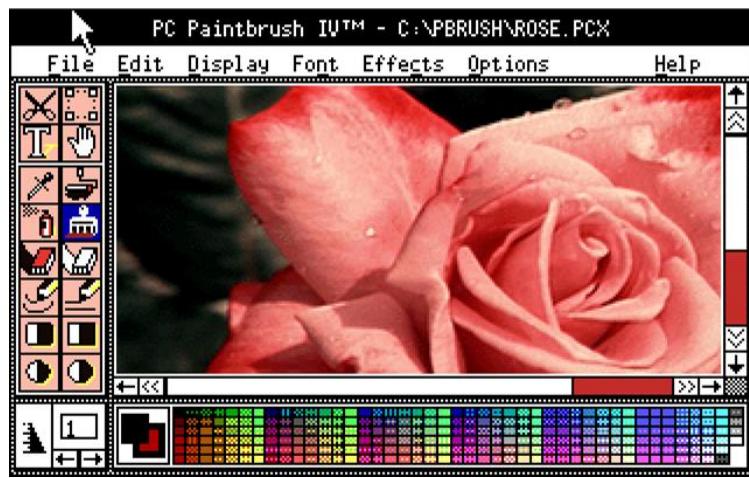


Figure 90 - DOS Picture of PC Paintbrush

the PC Paintbrush application in PNG format!). I discuss it since I had an older game engine I wanted to convert over to windows that primarily used this format for building side-scrolling games.

⊕ GIF – this is probably the most popular image-saving format. It is usually used to represent icons, cartoons and simple images for the web. It was introduced by CompuServe in 1987 and quickly became popular. “The format supports up to 8 bits per pixel allowing a single image to reference a palette of up to 256 distinct colors chosen from the 24-bit RGB color space. The color limitation makes the GIF format suitable for reproducing color photographs...” The format uses a lossless data compression format called Lempel-Ziv-Welch(LZW) to reduce the size of the file. This compression format was patented in 1985 and lead to controversy by the patent owner – Unisys over



Figure 89 - When DOS was KING!

licensing. Transparent GIFs are used to blend images into the Web page background. The color set as the “transparent” color shows through on the web page. Another popular GIF format is the Animated GIF. You see them quite often on web pages where an image is seen to move as in a flowing banner but most are just annoying.

- ⊕ PNG – stands for “Portable Network Graphics” format. It is similar to GIF format in that it contains a bitmap of indexed colors under a lossless compression but does not have the same copyright limitations. It is very popular as a web graphics format. It does not have the same

capability as GIF to animate graphics but can be used to fade an image from opaque to transparent ...more on this later.

- ❖ JPEG – stands for “Joint Photographic Experts Group” created by a similarly named group. It is primarily used for photographic images. It is a “lossy compression” format – which means you trade storage size against loss of a little information on the image. The images are usually sharp and detailed.



Figure 91 - Example jpeg image

The SDL_Image library supports the following file formats: BMP, GIF, JPEG, LBM, PCX, PNG, PNM, TGA, TIFF, XCF, XPM, and XV.

If you haven’t followed the instructions in Chapter2 in the section headed as “Installing and Testing SDL_Image” please do so now. Any program using this library will require that you add

```
#include "SDL\SDL_image.h"
```

You will need the following functions to:

- ❖ Initialize the SDL_Image library
- ❖ Load an image
- ❖ And close and release the library

Function Name: **IMG_Init()**

Format:

```
int IMG_Init(int flags);
```

Description:

This function loads support for the indicated file format in the flag argument. You can use the following flag values:

- ❖ IMG_INIT_JPG
- ❖ IMG_INIT_PNG

 **IMG_INIT_TIF**

The function returns a bitmask of all the currently initted image loaders.

Function Name: **IMG_Load()**

Format:

```
SDL_Surface * IMG_Load(const char *file);
```

Description:

This function loads the file specified in the argument. The function returns a pointer to an `SDL_Surface` representing the image obtained or `NULL` if it failed. When loading images into your SDL program it is best to do it before entering the game loop where you process events and update the screen since loading images can take some time. If the image format supports a transparent pixel this function will set the colorkey for the surface. The way to obtain the transparent color is to examine in the `colorkey` member of the `SDL_PixelFormat` member (`format`) of the `SDL_Surface`. For example, after loading an image you can do the following to set the transparent color:

```
SDL_SetColorKey(imageSurface, SDL_RLEACCEL, iamgeSurface->format->colorkey);
```

Function Name: **IMG_Quit()**

Format:

```
void IMG_Quit();
```

Description:

This function is used to close and clean up all dynamically loaded image libraries.

In order to test the typical set of functions we will be using in the `SDL_Image` library I went online to the website <http://www.spicypixel.net/2008/01/10/gfxlib-fuzed-a-free-developer-graphic-library/> to obtain some free game graphics in the library GfxLib-Fuzed.

We are going to create a program to open and display the `lev03_siberia` image with the file name `area03_mock.jpg`.

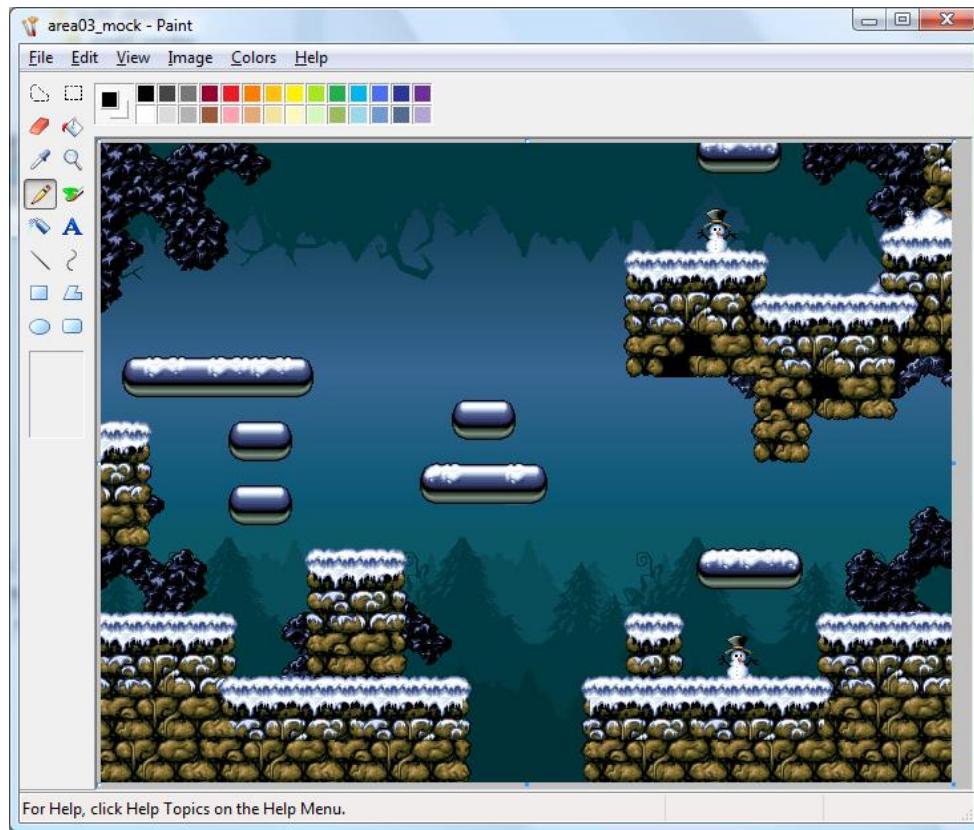


Figure 92 - jpeg image we will display.

LAB #14: Program 2_14 – Displaying a jpeg image

- ✚ Create a new project named Program2_14 using the template Simple SDL Project template.
- ✚ Add the include to use `SDL_Image.h`
- ✚ Change the window caption to “Display jpeg image”
- ✚ After the code to set the caption add code to:
 - Initialize the `SDL_Image` library for JPEG using the function `IMG_Init`
 - Load the image file `area03_mock.jpg` into the `SDL_Surface` pointer variable `pLevelImage`
 - Test to make sure the image was read in correctly, otherwise write a message to cerr and exit the program
 - Close the library using the function `IMG_Quit()`
- ✚ We will now need to convert the image into a format that will allow us to display on the screen. We will need a new `SDL` function `SDL_ConvertSurface`.

Function Name: **`SDL_DisplayFormat`**

Format:

```
SDL_Surface *SDL_DisplayFormat(SDL_Surface *surface);
```

Description:

This function takes a surface (e.g. one where we saved our jpeg image) and copies it to a new surface matching the pixel format and colors of the video display surface. If the function fails it returns NULL. The `SDL_Surface` returned can be used to display an image directly to the screen.

Add code to use the function `SDL_DisplayFormat` providing as an argument the `pLevelImage` `SDL_Surface` we got from the `IMG_Load` and save the new “display ready” `SDL_Surface` pointer into a new variable named `pLevelImageDisplay`:

```
SDL_Surface *pLevelImageDisplay = SDL_DisplayFormat(pLevelImage);
```

- ⊕ Add an if statement to make sure the `SDL_DisplayFormat` worked, exit the program with an error message if it failed.
- ⊕ Add code to free the `pLevelImage` `SDL_Surface`
- ⊕ Add code near the end of the program (before freeing the `pDisplaySurface`) to free `pLevelImageDisplay`.
- ⊕ Use `SDL_BlitSurface` to display the `SDL_Surface` to `pDisplaySurface`. Use the simple version of the function where the `SDL_Rect` arguments are `NULL`
- ⊕ Compile and execute your program. You should see the figure above but displayed in your window.

There is another useful function for converting surface data into different formats – `SDL_ConvertSurface`. The `SDL_DisplayFormat` we used in the previous lab actually calls `SDL_ConvertSurface`.

Function Name: **SDL_ConvertSurface**

Format:

```
SDL_Surface *SDL_ConvertSurface(SDL_Surface *surface, SDL_PixelFormat *fmt,  
Uint32 flags);
```

Description:

This function creates a new surface of the specified format and then copies and maps the given surface to it. If the function fails it returns NULL. The flags that can be specified are:

Table 13 - Flags for `SDL_ConvertSurface`

<code>SDL_SWSURFACE</code>	SDL creates the memory in system memory.
<code>SDL_HWSURFACE</code>	SDL creates the surface in video memory.
<code>SDL_SRCCOLORKEY</code>	Turns on color keying for blits from this surface.
<code>SDL_SRCALPHA</code>	Turns on alpha-blending

Alpha Blending

Alpha blending is a mechanism for combining a translucent foreground color with a background color, thereby producing a new blended color. The degree in which the foreground color combines ranges from 0 to 1 or in the case of SDL the value ranges from 0..255. An alpha value of 0 will make the foreground image color translucent (SDL_ALPHA_TRANSPARENT). The value 255 (SDL_ALPHA_OPAQUE) will make the image opaque – solid.

I will demonstrate how alpha works by creating an example using html.

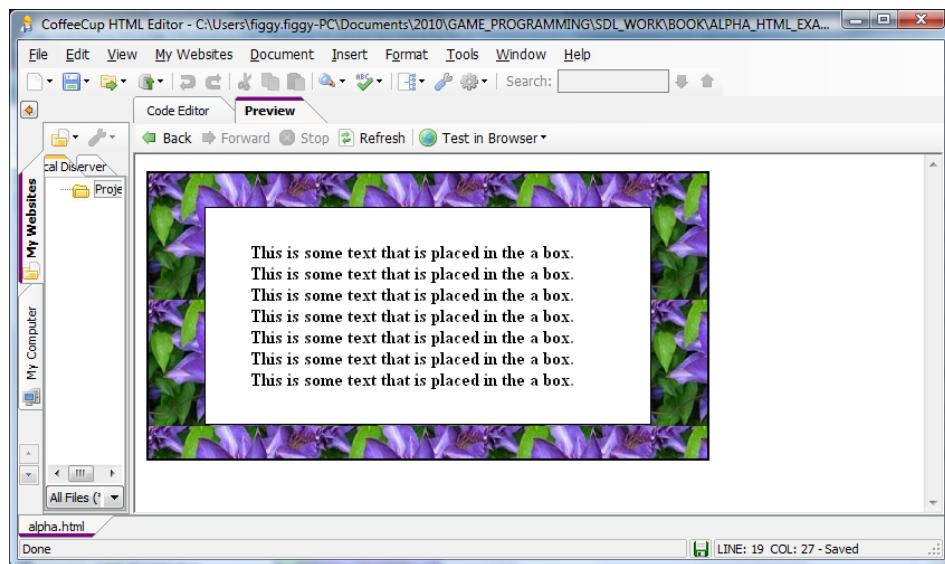


Figure 93 - text box where alpha=SDL_ALPHA_OPAQUE

Let's ignore the fact that many systems have different scales to indicate totally opaque and totally transparent. I will translate everything to SDL scale where SDL_ALPHA_OPAQUE means that the element or image is solid and SDL_ALPHA_TRANSPARENT means that the element or image is invisible. The image above has a white box where the alpha value was set to SDL_ALPHA_OPAQUE.

The next figure demonstrates the same text box but with the ALPHA value set to SDL_ALPHA_OPAQUE/2 (half the opaque value).

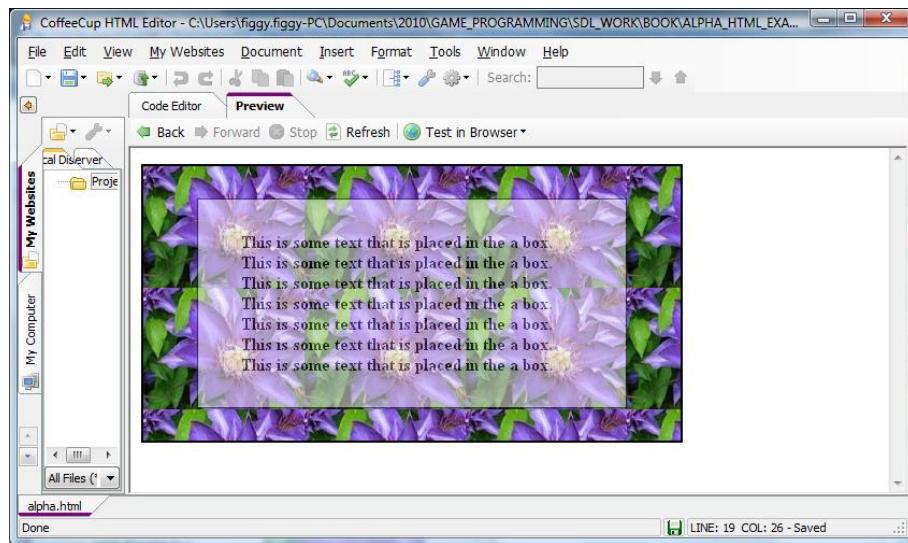


Figure 94 - text box where alpha=SDL_ALPHA_OPAQUE/2

As you can see the background image comes though that is the text box image becomes more transparent.

Question: What do you think will happen when we set the alpha value to SDL_ALPHA_TRANSPARENT to 0?

We will need to use the following two new SDL functions to create the same affect in our SDL program:

- ✚ **SDL_DisplayFormatAlpha** – converts a surface taking into consideration the alpha channel
- ✚ **SDL_SetAlpha** – used to change the alpha value of a surface

Function Name: **SDL_DisplayFormatAlpha**

Format:

```
SDL_Surface *SDL_DisplayFormatAlpha(SDL_Surface *surface);
```

Description:

This function takes a surface and returns a pointer to a new surface with the same pixel format and colors of the video display surface plus an alpha channel for fast blitting onto the display surface. Internally, it invokes the SDL function **SDL_ConvertSurface**. In order to be useful, you should set the colorkey (transparency color) and alpha value before using the function. If the function fails it will return NULL rather than a pointer to an **SDL_Surface**.

Function Name: **SDL_SetAlpha**

Format:

```
Int SDL_SetAlpha(SDL_Surface *surface, Uint32 flags, Uint8 alpha);
```

Description:

This function is used for setting the per-surface alpha value. It can be used to enable or disable alpha blending. You provide a pointer to the surface, the flags used to specify to use alpha blending (SDL_SRCALPHA) and/or RLE acceleration (SDL_RLEACCEL). The function returns 0 on success or -1 if there is an error.

LAB #15: Program 2_15 – Illustrating alpha blending

- ⊕ Create a new project named Program2_15 using the template Simple SDL Project template.
- ⊕ Copy the code from the last lab (#14)

We will be adding another image (I will call monsterImage) to the background level and slowly fade the monster image. The program will change the alpha value from SDL_ALPHA_OPAQUE to SDL_ALPHA_TRANSPARENT and back (see the next two figures)

- ⊕ Add the const int named FRAMES_PER_SECOND and FRAME_RATE after the SCREEN_WIDTH and SCREEN_HEIGHT section. Set the constant value to 30 and 1000/FRAMES_PER_SECOND for now. This will be used to trigger a change in the alpha value every 33.34 milliseconds.
- ⊕ Modify the flag for SDL_SetVideoMode to `SDL_ANYFORMAT | SDL_SRCALPHA | SDL_SRCCOLORKEY`.
- ⊕ Change the caption (if you notice from the figures below that I forgot to do that!)
- ⊕ Modify the `IMG_Init` and add `IMG_INIT_PNG` as an additional flag since our new monster image is a PNG file.
- ⊕ Add code after reading in and testing that the `area03_mock.jpg` was successfully read in to read in and test the file “`snipe.stand_right.png`”. Use the name `pMonsterImage` as the name of the `SDL_Surface` pointer.
- ⊕ Add code to set the monster image transparency color
 - First create a `Uint32` variable and set the transparencyColor to (255,0,255) using `SDL_MapRGB`
 - Use `SDL_SetColorKey` to set the transparency color



Figure 95 - Monster completely opaque



Figure 96 - Monster "ghosting" out

- ⊕ After the line that uses `SDL_DisplayFormat` to convert `pLevelImage` into an `SDL_Surface` that can be used to display the image on the screen add a new variable `Uint8` named `alphaValue` and initialize to `SDL_ALPHA_OPAQUE`.
- ⊕ Invoke the `SDL` function `SDL_SetAlpha` for the `pMonsterImage` and set to the `alphaValue`.

```
SDL_SetAlpha(pMonsterImage, SDL_SRCALPHA, alphaValue);
```

- ⊕ Use the `SDL` function `SDL_DisplayFormatAlpha` to convert the `pMonsterImage` into a surface that will be used to display the monster image on the video surface.

```
SDL_Surface *pMonsterImageDisplay = SDL_DisplayFormatAlpha(pMonsterImage);
```

- ⊕ Expand the if statement that checks that `pLevelImageDisplay` is valid and add a check for `pMonsterImageDisplay`.
- ⊕ After the `SDL_BlitSurface` code for the `pLevelImageDisplay` add two lines of code
 - The first one creates and new `SDL_Rect` named `monsterPosition` and initialize to `{64,128, 0,0}`. This will be used to place the monster at location `(64,128)` on the screen.
 - Add a new `SDL_BlitSurface` that blits the `pMonsterImageDisplay` to the surface

```
SDL_BlitSurface(pMonsterImageDisplay, NULL,
                pDisplaySurface, &monsterPosition);
```

- ⊕ Now before the for loop add a `Uint32` timer variable and initialize to the value `SDL_GetTicks()` returns
- ⊕ In the for loop (after // DO OUR THING) get the `currentTime` (see Lab #12) again using `SDL_GetTicks()`
- ⊕ Add an ifStatement that checks if `currentTime - timer` has exceeded the `FRAME_RATE`
 - If the ifStatement is true
 - Update the timer
 - Decrement the `alphaValue` by 1
 - Use `SDL_SetAlpha` to set `pMonsterImage` to the new alpha value
 - Use `SDL_DisplayFormatAlpha` again to convert the `pMonsterImage` to a surface to be used for display using `SDL_DisplayFormatAlpha`
- ⊕ After the ifStatement add code to blot the `pLevelImageDisplay` to the video display surface
- ⊕ Add code to blot the `pMonsterImageDisplay` again using the `monsterPosition` `SDL_Rect` to specify the `(x,y)` location on the screen
- ⊕ Add code after the forLoop to free the `pMonsterImage` and `pMonsterImageDisplay` surfaces.
- ⊕ Compile and Run

You should see the monster fade out ...reappear and fade out again repeatedly.

This program took a while to figure out. I started out applying `SDL_SetAlpha` on the `pMonsterImageDisplay` but the monster would not fade. I ended up have to keep the `pMonsterImage` around and changing its alpha value and re-applying the `SDL_DisplayFormatAlpha`. There may be an easier way to get this done. I am open to suggestions.

Other Topics

We will come back to additional video topics later on in this book pertaining to `SDL_Palette`, `SDL_Overlays` and `Gamma`.

Summary

We covered quite a number of topics in this chapter. You should now be able to understand how the test programs in Chapter 1 worked (go ahead check them out). We learned how to initialize `SDL` and the video display surface. We learned how to use the image library to load and display a graphics image and how to set the transparency color and alpha value (if required). In addition, we learned how to set up the game loop to check for events (e.g quit) or update the screen.

You should be familiar with the following `SDL` functions:

`SDL_Init`
`SDL_GetError`
`SDL_Quit`
`SDL_InitSubSystem`
`SDL_QuitSubSystem`
`SDL_WasInit`
`SDL_SetVideoMode`
`SDL_FreeSurface`
`SDL_Delay`
`SDL_MapRGB`
`SDL_LockSurface`
`SDL_UnlockSurface`
`SDL_MUSTLOCK`
`SDL_FillRect`
`SDL_UpdateRect`
`SDL_SetClipRect`
`SDL_GetVideoInfo`
`SDL_LoadBMP`
`SDL_BlitSurface`
`SDL_SetColorKey`
`SDL_GetTicks`
`SDL_Flip`
`IMG_Init`
`IMG_Load`
`IMG_Quit`
`SDL_DisplayFormat`
`SDL_ConvertSurface`
`SDL_DisplayFormatAlpha`
`SDL_SetAlpha`

And SDL structures:

SDL_Rect
SDL_Color
SDL_Surface
SDL_VideoInfo
SDL_PixelFormat

Review Questions

1. What function is used to initialize SDL?
2. What function is called when we want to close and return all SDL resources?
3. What flag(s) would you use to only initialize the VIDEO and JOYSTICK subsystems?
4. What function do you use if you want to open a subsystem (after already using `SDL_Init`) to use the CDROM?
5. What function is useful to determine what subsystem has already been opened?
6. What function returns the last error message from an SDL function call?
7. What is the purpose of the `SDL_Surface` struct?
8. What function call is used to obtain the display surface?
 - a. `SDL_SetVideoMode`
 - b. `SDL_SetAttribute`
 - c. `SDL_MapRGB`
 - d. `SDL_FillRect`
9. `SDL_Delay` is used to:
 - a. Put a breakpoint in your program
 - b. Put the program to sleep for a short period of time
 - c. Obtain the current time in milliseconds since SDL has been initialized
 - d. Check the operating for any events
10. A screen display of 800x600 has an x value range from:
 - a. 0..800
 - b. 0..799
 - c. 1..800
 - d. 1..799
11. A screen display of 360x200 has a y value range from:
 - a. 0..200
 - b. 0..199
 - c. 1..200
 - d. 1..199
12. What function is used to convert an (r,g,b) value into a pixel color to be used to display on the screen surface?

- a. `SDL_SetVideoMode`
 - b. `SDL_SetAttribute`
 - c. `SDL_MapRGB`
 - d. `SDL_FillRect`
13. The `SDL_LockSurface` function is used to
- a. Keep all other applications from using our graphic images
 - b. Keep all other applications from using the shared resource – display surface
 - c. Keep all other applications from running
 - d. Lock all resources to prevent simultaneous usage
14. The `SDL_SetClip` function
- a. Specifies the portion of the screen that will not get updated
 - b. Specifies the images that collide
 - c. Specifies the portion of the screen that will get updated
 - d. Specifies the rectangular area that gets overwritten
15. After reading in an image file what function should be used to display to the video display?
- a. `SDL_BlitSurface`
 - b. `SDL_FillRect`
 - c. `SDL_SetColorKey`
 - d. `SDL_SetAlpha`
16. The function to use to specify the transparency color is?
- a. `SDL_BlitSurface`
 - b. `SDL_FillRect`
 - c. `SDL_SetColorKey`
 - d. `SDL_SetAlpha`
17. The game framerate is
- a. The number of times the images move on the screen
 - b. The number of times the game loop executes per second
 - c. The number of times the user clicks a keyboard button per second
 - d. The number of times the screen display is updated per second
18. A good function to use to calculate the frame rate is
- a. `SDL_Delay`
 - b. `SDL_GetTicks`
 - c. `SDL_AddTimer`
 - d. `SDL_SetTimer`
19. A good technique to use to avoid screen tearing and flickering is:
- a. Blitting
 - b. Double buffering
 - c. Page Flipping
 - d. B and C
20. What is the best thing about using SDL?

Programming Exercises

1. Create a function called `setBackgroundColor` with the following signature:

```
void setBackgroundColor(SDL_Surface *pDisplaySurface, SDL_Color color)
```

This function will set the entire screen to the specified color. Try calling it several times with different colors (use `SDL_Delay`) to give some time between screen updates). I recommend using the technique used for setting the frame rate but set the frame rate to some a low value (e.g. 5) in order to see each color.

[Circle-Drawing Algorithms²⁴](#)

Beginning with the equation of a circle:

$$(x - x_0)^2 + (y - y_0)^2 = r^2$$

We could solve for y in terms of x:

$$y = y_0 \pm \sqrt{r^2 - (x - x_0)^2}$$

And use this equation to compute the pixels of the circle. When finished we'd end up with code that looked something like the following:

```
public void circleSimple(int xCenter, int yCenter, int radius, Color c)
{
    int pix = c.getRGB();
    int x, y, r2;

    r2 = radius * radius;
    for (x = -radius; x <= radius; x++) {
        y = (int) (Math.sqrt(r2 - x*x) + 0.5);
        raster.setPixel(pix, xCenter + x, yCenter + y);
        raster.setPixel(pix, xCenter + x, yCenter - y);
    }
}
```

The above is written in Java, let's create a version in SDL (call the project `DrawCircle1`) by creating a function with the signature:

²⁴ <http://groups.csail.mit.edu/graphics/classes/6.837/F98/Lecture6/circle.html>

```
void drawCircle(SDL_Surface *surface, int xCenter, int yCenter, int radius, SDL_Color color);
```

Note: Use the drawPixel function created earlier in this chapter.

Set

```
int x = SCREEN_WIDTH / 2;  
int y = SCREEN_HEIGHT / 2;  
int radius = 100;
```

and test that it creates a circle similar to the one shown below.

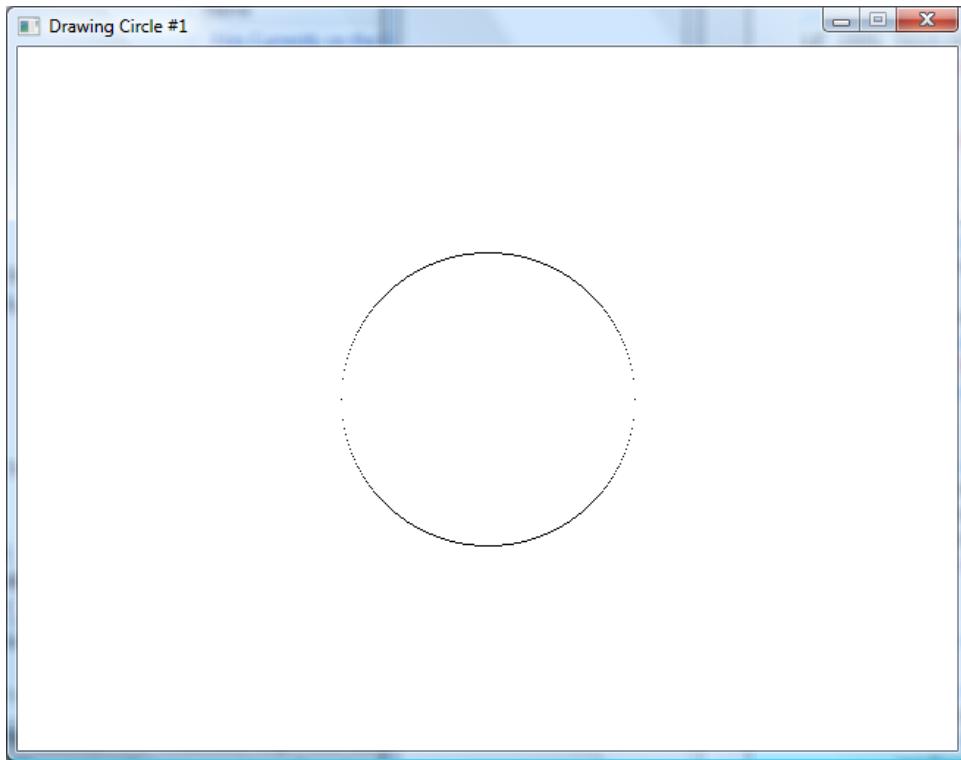


Figure 97 - Drawing a circle

The above program demonstrates the algorithm outlined in circleSimple().

As you can see the circle looks fine in areas where only one pixel is required for each column, but in areas of the circle where the local slope is greater than one the circle appears discontinuous (where have we seen this before?)

We could take the approach of computing the derivative (i.e. the local slope) of the function at each point and then make a decision whether to step in the x direction or the y direction. But, we will explore a different tact here.

A circle exhibits a great deal of symmetry. We've already exploited this somewhat by plotting two pixels for each function evaluation; one for each possible sign of the square-root function. This symmetry was

about the x-axis. The reason that a square-root function brings out this symmetry results from our predilection that the x-axis should be used as an independent variable in function evaluations while the y-axis is dependent. Thus, since a function can yield only one value for member of the domain, we are forced to make a choice between positive and negative square-roots. The net result is that our simple circle-drawing algorithm exploits *2-way symmetry* about the x-axis.

Obviously, a circle has a great deal more symmetry. Just as every point above an x-axis drawn through a circle's center has a symmetric point an equal distance from, but on the other side of the x-axis, each point also has a symmetric point on the opposite side of a y-axis drawn through the circle's center.

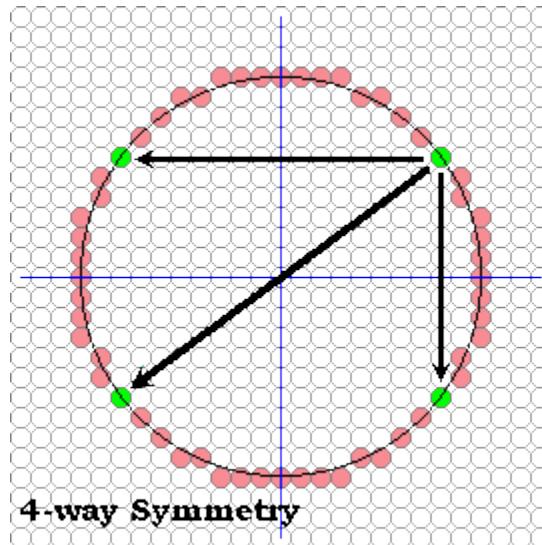


Figure 98 - The 4-way symmetry in a circle

We can quickly modify our previous algorithm to take advantage of this fact as shown below.

```
public void circleSym4(int xCenter, int yCenter, int radius, Color c)
{
    int pix = c.getRGB();
    int x, y, r2;

    r2 = radius * radius;
    raster.setPixel(pix, xCenter, yCenter + radius);
    raster.setPixel(pix, xCenter, yCenter - radius);
    for (x = 1; x <= radius; x++) {
        y = (int) (Math.sqrt(r2 - x*x) + 0.5);
        raster.setPixel(pix, xCenter + x, yCenter + y);
        raster.setPixel(pix, xCenter + x, yCenter - y);
        raster.setPixel(pix, xCenter - x, yCenter + y);
        raster.setPixel(pix, xCenter - x, yCenter - y);
    }
}
```

Create a new project (DrawCircle2) using the algorithm above.

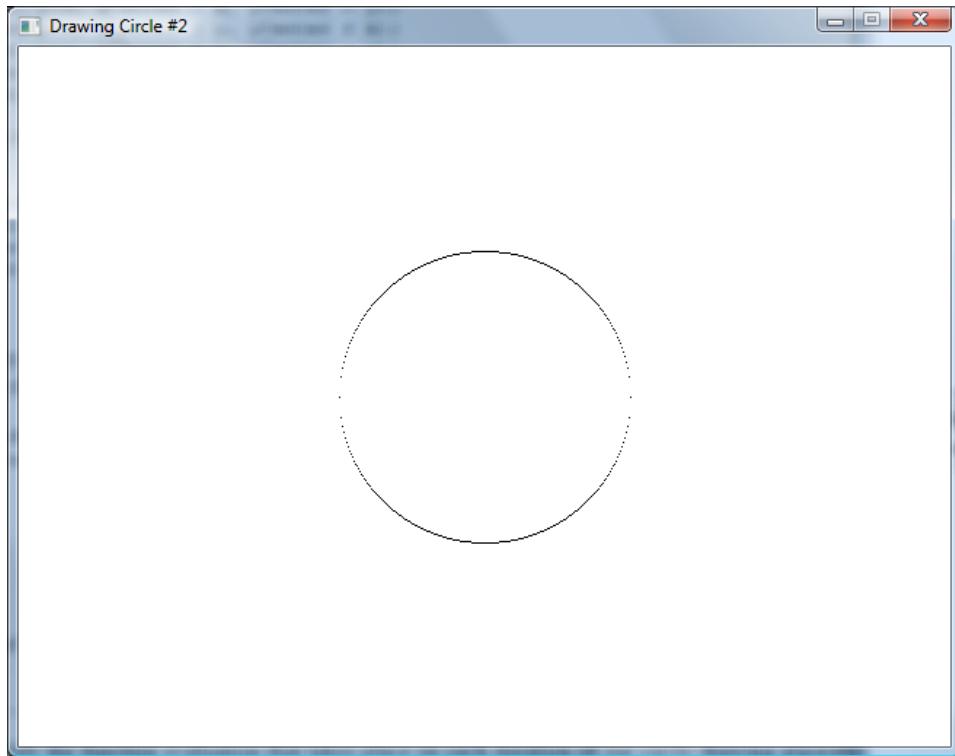


Figure 99 - DrawCircle2

This algorithm **has all the problems** of our previous algorithm, but, it gives the same result with **half as many function evaluations**. So much for "making it work first" before optimizing. But, we're on a roll so let's push this symmetry thing as far as it will take us.

Notice also that a circle exhibits symmetry about the pair of lines with slopes of one and minus one, as shown below.

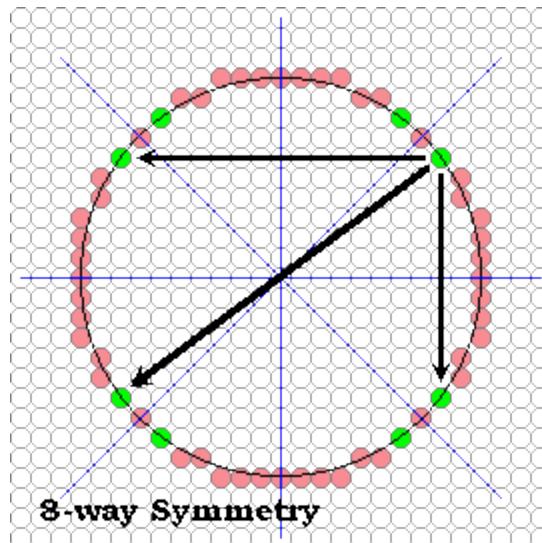


Figure 100 - 8-way symmetry

We can find any point's symmetric complement about these lines by permuting the indices. For example the point (x,y) has a complementary point (y,x) about the $\text{linex}=y$. And the total set of complements for the point (x,y) are

$$\{(x,-y), (-x,y), (-x,-y), (y,x), (y,-x), (-y,x), (-y,-x)\}$$

The following routine takes advantage of this **8-way symmetry**.

```
public void circleSym8(int xCenter, int yCenter, int radius, Color c)
{
    int pix = c.getRGB();
    int x, y, r2;

    r2 = radius * radius;
    raster.setPixel(pix, xCenter, yCenter + radius);
    raster.setPixel(pix, xCenter, yCenter - radius);
    raster.setPixel(pix, xCenter + radius, yCenter);
    raster.setPixel(pix, xCenter - radius, yCenter);

    y = radius;
    x = 1;
    y = (int) (Math.sqrt(r2 - 1) + 0.5);
    while (x < y) {
        raster.setPixel(pix, xCenter + x, yCenter + y);
        raster.setPixel(pix, xCenter + x, yCenter - y);
        raster.setPixel(pix, xCenter - x, yCenter + y);
        raster.setPixel(pix, xCenter - x, yCenter - y);
        raster.setPixel(pix, xCenter + y, yCenter + x);
        raster.setPixel(pix, xCenter + y, yCenter - x);
        raster.setPixel(pix, xCenter - y, yCenter + x);
        raster.setPixel(pix, xCenter - y, yCenter - x);
        x += 1;
    }
}
```

```

        y = (int) (Math.sqrt(r2 - x*x) + 0.5);
    }
    if (x == y) {
        raster.setPixel(pic, xCenter + x, yCenter + y);
        raster.setPixel(pic, xCenter + x, yCenter - y);
        raster.setPixel(pic, xCenter - x, yCenter + y);
        raster.setPixel(pic, xCenter - x, yCenter - y);
    }
}

```

So now we get 8 points for every function evaluation, and this routine should be approximately 4-times faster than our initial circle-drawing algorithm. What's going on with the four pixels that are set outside the loop (both at the top and bottom)? Didn't I say that every point determines 7 others?

Create a C++ SDL based project named DrawCircle3 using the algorithm above.

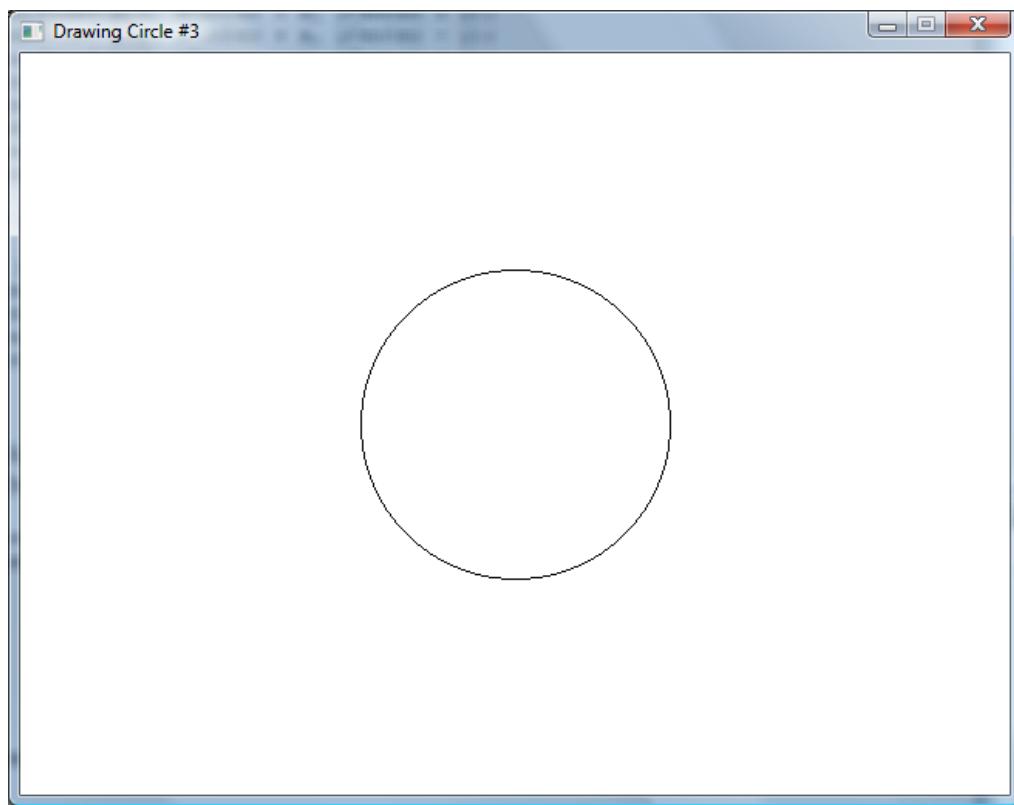


Figure 101 - Using 8-way symmetry to draw a circle

Wait a Minute! What has happened here?

It seems suddenly that our circle's appear continuous, and we added no special code to test for the slope. Symmetry has come to our rescue (actually, symmetry is also what saved us on lines... think about it).

For more information on how to make the computation faster please see the original paper (<http://groups.csail.mit.edu/graphics/classes/6.837/F98/Lecture6/circle.html>) this section was copied from.

Chapter 3 - Sprites, A Simple View

Chapter 4 - Processing Events

The key part of our program has been the forLoop which acted as our game loop

```
for(;;)      {
    //wait for an event
    if(SDL_PollEvent(&event)==0)  {
        // no event occurred so move things around
        // on the screen
        //
        // now update the screen
        SDL_Flip(pDisplaySurface);
    } else {
        //event occurred, check for quit
        if(event.type==SDL_QUIT) break;
    }
}
```

The forLoop contains an ifStatement that basically translates to:

```
If no event occurred then
    Update the screen (move monsters, fire missiles at hero, etc)
Else
    Check if the event was a user request to ‘QUIT’ application, if so, break out of the loop
End if
```

Alternative Game Loop

Another method used by many developers for the game loop above is to use the following:

```
while (SDL_PollEvent(&event) )
{
    // check for messages
    switch (event.type)
    {
        // exit if the window is closed
        case SDL_QUIT:
            done = true;
            break;

        // check for keypresses
        case SDL_KEYDOWN:
            {
                // exit if ESCAPE is pressed
                if (event.key.keysym.sym == SDLK_ESCAPE)
                    done = true;
                break;
            }
    } // end switch
} // end of message processing
```

```

// DRAWING STARTS HERE

// clear screen
SDL_FillRect(screen, 0, SDL_MapRGB(screen->format, 0, 0, 0));

// draw bitmap
SDL_BlitSurface(bmp, 0, screen, &dsrcrect);

// DRAWING ENDS HERE

// finally, update the screen :)
SDL_Flip(screen);
} // end main loop

```

The above translates into:

```

while there exists an event
    process the event
end while
update the screen

```

The are both basically the same!

The SDL function used to determine if there exists an event to process is the **SDL_PollEvent**.

Function Name: **SDL_PollEvent**

Format:

```
int SDL_PollEvent(SDL_Event *event);
```

Description:

This function checks of there are any currently pending events (keypress, mouse click, etc). It returns 1 (regarded as TRUE) if there are any pending events, otherwise 0. The event data structure is populated with the event details. The game loop we contruct always processes any pending events first before updating the screen.

Note: When the screen needs to be updated we have used the **SDL_UpdateRect** and more recently **SDL_Flip** to perform a page flip. Page flipping or double buffering allows you to prepare the next game frame and swap it out entirely so the game appears smooth and continuous to the player.

The operating system sends to your application any actions taken by the user directed to the screen or window representing your game program. The possible actions a user can take can range from pressing the ESCAPE key, firing a button on the joystick, closing the window (in order to start doing homework), or move the mouse. Each action is packaged as an event and is available for the program to process. What is an event? In SDL an event is packaged into a complex struct named **SDL_Event**. There are many

types of event packaged and sent to your program from the SDL system. These events fall into the following categories:

- ✚ Keyboard Events
- ✚ Joystick Events
- ✚ System Events
- ✚ Mouse Events

When you initialize SDL using `SDL_Init` SDL establishes an event queue. When the user performs an action that generates one of the above events it will get placed in the event queue as an `SDL_Event` struct. The struct consists of two components `Uint8` type indicating the type of event and one of the following event types:

- ✚ Keyboard Events
 - `SDL_KeyboardEvent` captures one of the following two events
 - `SDL_KEYDOWN`
 - `SDL_KEYUP`
- ✚ Joystick Events
 - `SDL_JoyAxisEvent`
 - `SDL_JoyBallEvent`
 - `SDL_JoyHatEvent`
 - `SDL_ButtonEvent`
- ✚ System Events
 - `SDL_ActiveEvent`
 - `SDL_ResizeEvent`
 - `SDL_ExposeEvent`
 - `SDL_QuitEvent`
 - `SDL_UserEvent`
 - `SDL_SywWMEvent`
- ✚ Mouse Events
 - `SDL_MouseMotionEvent`
 - `SDL_MouseButtonEvent`

Keyboard Events

There are two types of keyboard events issued when the user interacts with the keyboard – key pressed and key released. The event is captured in the `SDL_Event` struct as an `SDL_KeyboardEvent`. The definition of this struct is the following:

```
typedef struct {
    Uint8 type;
    Uint8 state;
    SDL_keysym keysym;
} SDL_KeyboardEvent;
```

The `type` value will be either be `SDL_KEYDOWN` or `SDL_KEYUP`. The `state` field pretty much reflects the same information as `type` it will be either `SDL_PRESSED` or `SDL_RELEASED`. The `SDL_keysym` contains all the details on which key(s) were pressed (or released). The `SDL_keysym` is another struct with the following format:

```
typedef struct {
    Uint8 scancode;
    SDLKey sym;
    SDLMod mod;
    Uint16 unicode;
} SDL_keysym;
```

The `scancode` hold a hardware specific scancode and will never be used in any of our games. The `sym` field holds an SDL virtual keysym. We will be primarily using this field since it will contain an `SDLKey` value.

The `SDLKey` values are one of the following:

Table 14 - SDL Keysym definitions

SDL_Key	ASCII value	Common name
<code>SDLK_BACKSPACE</code>	'\b'	backspace
<code>SDLK_TAB</code>	't'	tab
<code>SDLK_CLEAR</code>		clear
<code>SDLK_RETURN</code>	'r'	return
<code>SDLK_PAUSE</code>		pause
<code>SDLK_ESCAPE</code>	'\t'	escape
<code>SDLK_SPACE</code>	' '	space
<code>SDLK_EXCLAIM</code>	'!'	exclaim
<code>SDLK_QUOTEDBL</code>	""	quotedbl
<code>SDLK_HASH</code>	'#'	hash
<code>SDLK_DOLLAR</code>	'\$'	dollar
<code>SDLK_AMPERSAND</code>	'&'	ampersand
<code>SDLK_QUOTE</code>	""	quote
<code>SDLK_LEFTPAREN</code>	'('	left parenthesis
<code>SDLK_RIGHTPAREN</code>	')'	right parenthesis
<code>SDLK_ASTERISK</code>	'*'	asterisk
<code>SDLK_PLUS</code>	'+'	plus sign
<code>SDLK_COMMA</code>	','	comma
<code>SDLK_MINUS</code>	'-'	minus sign
<code>SDLK_PERIOD</code>	'.'	period
<code>SDLK_SLASH</code>	'/'	forward slash
<code>SDLK_0</code>	'0'	0
<code>SDLK_1</code>	'1'	1
<code>SDLK_2</code>	'2'	2
<code>SDLK_3</code>	'3'	3
<code>SDLK_4</code>	'4'	4
<code>SDLK_5</code>	'5'	5

SDL_Key	ASCII value	Common name
SDLK_6	'6'	6
SDLK_7	'7'	7
SDLK_8	'8'	8
SDLK_9	'9'	9
SDLK_COLON	:	colon
SDLK_SEMICOLON	‘;’	semicolon
SDLK_LESS	'<'	less-than sign
SDLK_EQUALS	'='	equals sign
SDLK_GREATER	'>'	greater-than sign
SDLK_QUESTION	'?'	question mark
SDLK_AT	'@'	at
SDLK_LEFTBRACKET	'['	left bracket
SDLK_BACKSLASH	'\'	backslash
SDLK_RIGHTBRACKET	']'	right bracket
SDLK_CARET	'^'	caret
SDLK_UNDERSCORE	'_'	underscore
SDLK_BACKQUOTE	'`'	grave
SDLK_a	'a'	a
SDLK_b	'b'	b
:	:	:
SDLK_z	'z'	Z
SDLK_DELETE	'^?'	Delete
SDLK_KP0		keypad 0
SDLK_KP1		keypad 1
:	:	:
SDLK_KP9		keypad 9
SDLK_KP_PERIOD	'.'	keypad period
SDLK_KP_DIVIDE	'/'	keypad divide
SDLK_KP_MULTIPLY	'*'	keypad multiply
SDLK_KP_MINUS	'-'	keypad minus
SDLK_KP_PLUS	'+'	keypad plus
SDLK_KP_ENTER	'r'	keypad enter
SDLK_KP_EQUALS	'='	keypad equals
SDLK_UP		up arrow
SDLK_DOWN		down arrow
SDLK_RIGHT		right arrow
SDLK_LEFT		left arrow
SDLK_INSERT		insert
SDLK_HOME		home
SDLK_END		end
SDLK_PAGEUP		page up
SDLK_PAGEDOWN		page down
SDLK_F1		F1
SDLK_F2		F2
:	:	:
SDLK_F15		F15
SDLK_NUMLOCK		numlock
SDLK_CAPSLOCK		capslock

SDL_Key	ASCII value	Common name
SDLK_SCROLLLOCK		scrolllock
SDLK_RSHIFT		right shift
SDLK_LSHIFT		left shift
SDLK_RCTRL		right ctrl
SDLK_LCTRL		left ctrl
SDLK_RALT		right alt
SDLK_LALT		left alt
SDLK_RMETA		right meta
SDLK_LMETA		left meta
SDLK_LSUPER		left windows key
SDLK_RSUPER		right windows key
SDLK_MODE		mode shift
SDLK_HELP		Help
SDLK_PRINT		print-screen
SDLK_SYSREQ		SysRq
SDLK_BREAK		break
SDLK_MENU		menu
SDLK_POWER		power
SDLK_EURO		euro

The program would check for keyboard events in the else part in the game loop:

```
if(SDL_PollEvent(&event)==0) {
    // NO EVENT
} else {
    //event occurred, check for quit
    if(event.type==SDL_QUIT) break;
}
```

A better design than the above is to have a switch statement that tests the value of event.type. Inside the switch statement you check for the event types you are interested in having your program handle.

```
bool endOfGame = false;
if (SDL_PollEvent(&event) == 0) {
    // Handle no event
} else {
    // Process event
    while (!endOfGame) {
        switch(event.type) {
            case SDL_KEYDOWN:
                // Handle key down
            case SDL_KEYUP:
                // Handle key up
            case SDL_QUIT:
                // Handle user quitting program
        }
    }
}
```

```

        endOfGame = true;
        break;
    default:
        // event not handling
    }
}
}

```

We now have replaced the forLoop with a whileLoop in order to have the SDL_QUIT set the flag endOfGame when the user has decided to quit. The same variable can be used to terminate the game when the game is over.

LAB #1: Program 3_1 – Handling Keyboard Events #1

- Create a new project named Program3_1 using the template Simple SDL Project template.
- Change the window caption to “Handling Keyboard Events #1”
- Create and initialize a bool variable to named endOfGame to false
- Change the forLoop to a whileLoop
- Add the switch statement as suggested above
- Compile and run
- Now

Table 15 - SDL Modifier Definitions

SDL Modifier	Meaning
KMOD_NONE	No modifiers applicable
KMOD_NUM	Numlock is down
KMOD_CAPS	Capslock is down
KMOD_LCTRL	Left Control is down
KMOD_RCTRL	Right Control is down
KMOD_RSHIFT	Right Shift is down
KMOD_LSHIFT	Left Shift is down
KMOD_RALT	Right Alt is down
KMOD_LALT	Left Alt is down
KMOD_CTRL	A Control key is down
KMOD_SHIFT	A Shift key is down
KMOD_ALT	An Alt key is down

Joystick Events

System Events

Mouse Events

Chapter 5 – How to organize a game

Creating a Game Template

Sample Games

Chapter 6 – Creating Pong

Using SDL_TTF

SDL Audio

In the beginning (80's and 90's) the only way to make sounds was to use the PC speaker. Here is a little program you can use to test your speaker (it may not work in on all platforms).

Table 16 - PC Beep

```
#include <cstdlib>
#include <iostream>

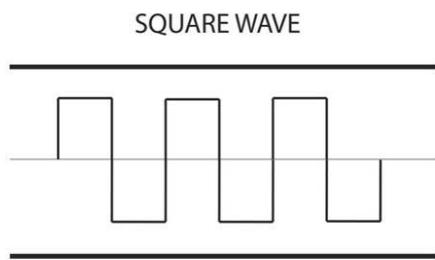
using namespace std;

int main(int argc, char *argv[])
{
    cout << "\a\aa\aa" << endl;
    cout.flush();
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

The program above beeps four times. Each '\a' corresponds to the 'alert' character which usually makes the PC beep.

Having fun with the PC Speaker!

One of my favorite games in the late 80's was Maniac Mansion²⁵. It was a graphic adventure game. The most memorable think for me and makes the point that sound matters was the background music used to set the mood as you went thought this creepy house with your friends trying to rescue your girlfriend from the evil mad scientist. It was amazing how the PC speaker was able to play the theme. Today what is even more amazing was how programmers were able to actually make the computer speaker do more than just beep – actual music without a sound card! Impossible?



recognizable.

The PC speaker is normally meant to create a square wave via only 2 levels of output. But you can actually have it act a DAC (digital to analog convertor) by timing a short pulse, that is, “going from one output level to the other and then back to the first, it is possible to drive the speaker to various output levels in between the two defined levels. This allows approximate playback of PCM²⁶ audio. This technique is called pulse-width modulation (PWM). The audio produced is poor quality but



There is a open-source C++ software for playing songs on the PC speaker named Smacky (<http://smacky.sourceforge.net/>).

Games have two types of sounds the first is sounds for events like gun shots, missles blowing or the heavy breathing of monsters. The sounds are short in duration. The second is the audio you here in the background often called background music. Audio music is used to set the atmosphere and tone for the game.

The SDL library SDL_mixer is rather low level and primitive to the other functions we have worked with. The problem stems from trying to make it cross-platform. The lowest common denominator is more primitive with respect to audio (as opposed to video). The first couple of function we will introduce will actually be avoided since it presents problems and issues we don't want to really manage when playing sounds or audio...but it helps to know the basics.

Some Audio Basics

²⁵ http://en.wikipedia.org/wiki/Maniac_Mansion

²⁶ PCM stands for pulse-code modulation a method used to digitally represent sampled analog signals. It is the standard form for digital audio in computers and various Blu-ray, CD and DVD formats.

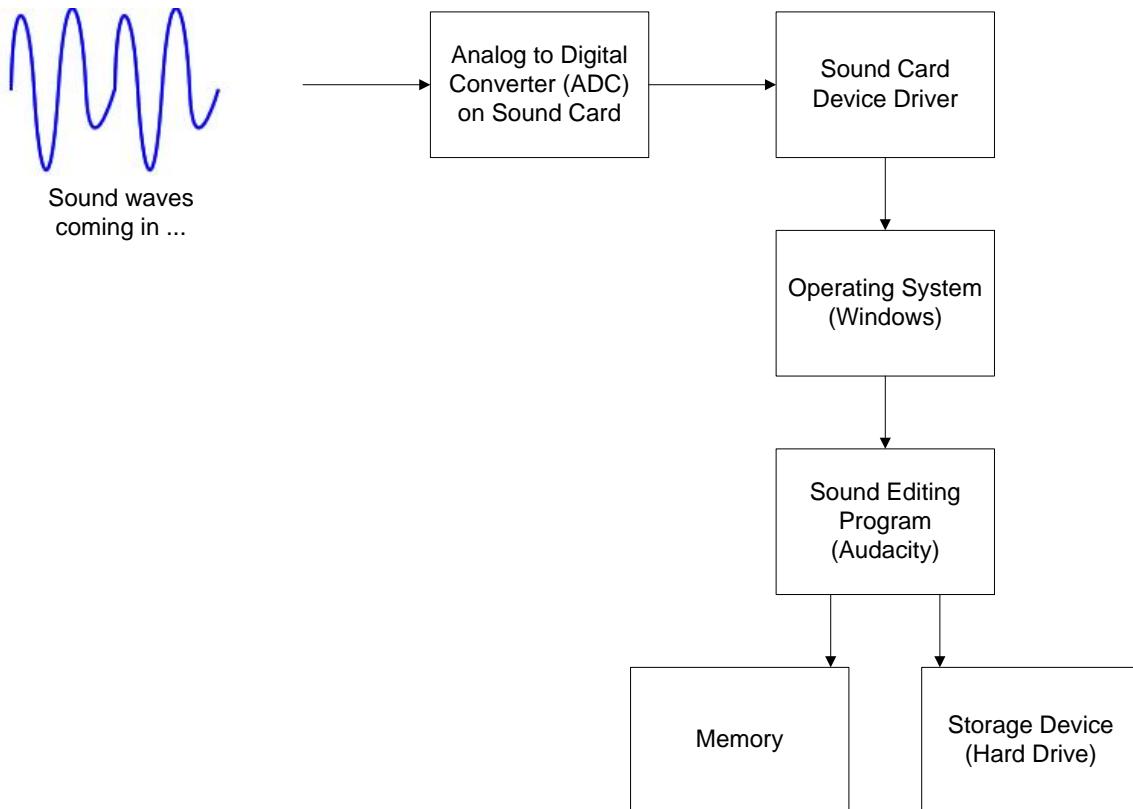


Figure 102 - How you record sounds/audio on your computer

Suppose you wanted to record sounds for your game using your computer. You would first get a sound editing program such as Audacity to record²⁷. When you are recording the sound the software is instructing the sound card to capture the input using a special hardware device called an ADC. A simple description for how the ADC works is that it periodically checks the electrical current (or voltage) coming in from the microphone and records the current (or voltage) level and translates it into a discrete number. This number is called a sample. How often (in one second) it samples or checks and creates a number is called the sample rate. That is, the sample rate is the number of times per second the ADC produces a data sample. For example if the ADC is sampling 11,025 samples per second we represent that as 11025Hz. Hz stands for “Hertz”. Typical sampling rates are 44,100Hz which gets shortened to 44.1 kHz. The notation stands for “KiloHertz” and it simply gets computed by dividing the Hz value by 1000. The value 44.1 kHz is special since it represents the sampling rate for a CD audio track.

The typical values used for sampling are 11025Hz, 22050Hz or 44100Hz. The higher the sampling rate of the sound the greater is the accuracy and quality of the sound. Another factor in the quality and depth of the sound is the number of bits used to represent a sample. Typical values are 8 bits and 16 bits. An 8-bit sample can only represent 256 values for the current (or voltage) level of the sound. A 16-bit sample can distinguish between 65,536 values. This provides the ability to add more richness and depth to the sound recording.

²⁷ I am assuming you have a microphone or some other input device and a sound card.

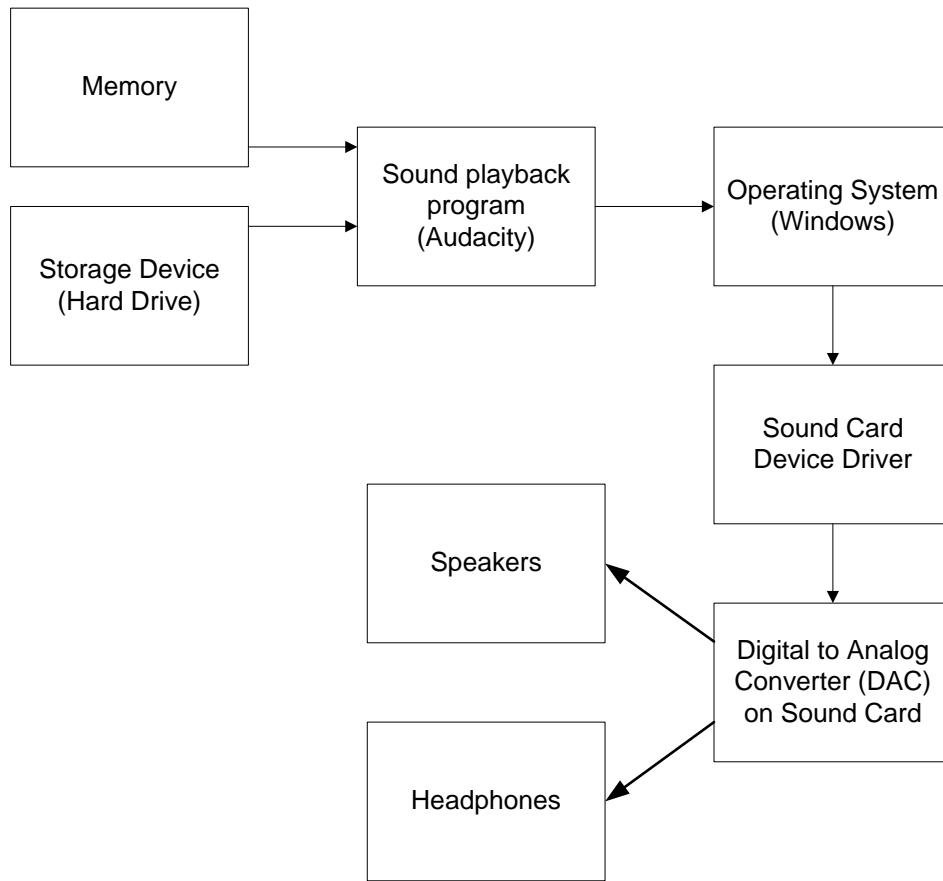


Figure 103 - Audio Out

When playing the sound or audio from a file on the hard drive or memory the samples are sent out to another hardware device that converts a digital value to analog (DAC) and sends it out to the speakers or headphones, and from there we hear sounds and music.

Using SDL primitive functions

One library that you can use is the `SDL_mixer` library. The library contains two key structures:

- ✚ `SDL_AudioSpec` – stands for audio specification, it contains information about the sound, such as format of the buffer, the number of channels, etc.

```

typedef struct {
    int freq;          // contains the audio frequency (samples/sec)
    Uint16 format;    // the data format
    Uint8 channels;   // number of channels (1 mono, 2 stereo,
                      // 4 surround, 6 surround with center and lfe
    Uint8 silence;    // buffer silence value (calculated), avoid
                      // generating noise
    Uint16 samples;   // buffer size in samples
    Uint32 size;       // buffer size in bytes (calculated)
}
  
```

```

        // callback function for filling the audio buffer
void (*callback) (void *userdata, Uint8 *stream, int len);
                // ptr the user data which is passed to callback
void *userdata;
} SDL_AudioSpec;

```

The common values for samples per second are 11025, 22050, and 44100. The higher the sample rate the higher the quality of sound. The format can be one of the following:

Table 17 - Audio Formats

Constant	Meaning
AUDIO_U8	Each channel consists of a stream of Uint8s.
AUDIO_S8	Each channel consists of a stream of Sint8s.
AUDIO_U16LSB	Each channel consists of a stream of little endian Uint16s
AUDIO_U16MSB	Each channel consists of a stream of big endian Uint16s
AUDIO_U16	This is the same as AUDIO_U16LSB
AUDIO_U16SYS	Can be AUDIO_U16LSB or AUDIO_U16MSB depends on platform
AUDIO_S16LSB	Each channel consists of a stream of little endian Sint16s
AUDIO_S16MSB	Each channel consists of a stream of big endian Sint16s
AUDIO_16	Same as AUDIO_S16LSB
AUDIO_S16SYS	Can be AUDIO_S16LSB or AUDIO_S16MSB depends on platform

You create a callback function in order to play any audio.

- ✚ **SDL_AudioCVT** – stands for audio convert, holds information to convert sound from one format to another

```

typedef struct {
    int needed;          // set to 1 if conversion is possible
    Uint16 src_format;  // audio format of the source
    Uint16 dest_format; // audio format of the destination
    double rate_incr;   // rate conversion increment
    Uint8 *buf;          // audio buffer
    int len;             // length of the original audio buffer in bytes
    int len_cvt;         // length of the converted audio buffer in bytes
    int len_mult;        // buf must be len * len_mult bytes in size
    double len_ratio;    // final audio size is len * len_ratio
    // pointers to function needed for conversions
    void (*filters[10]) (struct SDL_AudioCVT *cvt, Uint16 format);
    int filter_index;    // index of current conversion function
} SDL_AudioCVT;

```

If you plan on using the **SDL_mixer** library to play sounds and audio the first function you use to initialize the audio subsystem is **SDL_OpenAudio**.

Function Name: **SDL_OpenAudio**

Format:

```
int SDL_OpenAudio(SDL_AudioSpec *desiredAS, SDL_AudioSpec *obtainedAS);
```

Description:

You fill in the desired valued in the `SDL_AudioSpec` data structure `desiredAS`. The function returns 0 is successful and populated `obtainedAS` with the values obtained. If it fails the value returned is -1. Further if `obtainedAS` is NULL then the audio data will be as specified in the `desiredAS`.

[SDL Joystick](#)

[Chapter 7 – Creating MindSweeper](#)

[Chapter 8 – Creating Breakout](#)

[Chapter 9 – Creating Tetris](#)

[Chapter 10 – SDL Threads and Timers](#)

[Chapter 11 – Building a multiplayer online game](#)

[SDL_NET](#)

[SDL_MIXER](#)

[Chapter 12 – Building a Platform Game](#)

[Why I love Crisis Mountain!](#)

[Why I love Mario!!](#)

Chapter 13 – Other libraries and tools to build games

Chapter 14 – What comes next?

Last Chapter

Bibliography

LaMothe, Andre. Black Art of 3D Game Programming: Writing Your Own High-Speed 3D Polygon Video Games in C. Corte Madera, CA: The Waite Group, 1995.

Pazera, Ernest. Focus on SDL. Cincinnati, Ohio: Premier Press, 2003.

Appendix A: Places to visit on the Web

1. <http://www.brainycode.com> – This is the author’s website. As this book develops – versions are placed online for review. I also updated any images or files you may want to use in order to complete the labs and exercises.
2. <http://www.libsdl.org> – This is the main Simple Directmedia Layer (SDL) web site. You can obtain the latest libraries, sample code and wiki information. I highly recommend it.
3. <http://www.sdltutorials.com/> - A great place for tutorials and information on what is going on . They even have game contests you can try your hand at.
4. <http://galaxyframeworks.com/index.html> - This is a relatively new website created by Sam Lantinga the original developer of SDL. He has many accomplishments at a relatively young age – lead engineer for Loki Entertainment, author of SDL, and had a lead software engineering role on many Blizzard games (World of Warcraft, StarCraft II, etc). My hero!
5. <http://sdl.beuc.net/sdl.wiki/FrontPage> - This website is the place to go for SDL documentation and examples. I used to it to obtain details on all SDL functions described in these notes.
6. <http://www.gamedev.net/> - A great place for game development discussions and tutorials.

Appendix B – Microsoft Visual C++ 2010 Express

Appendix C – Pong, Breakout and MindSweeper

Pong

The first arcade version of Pong (not the first time it was tried on a CRT screen) was built by Al Acorn for Atari. The game started the quarters rolling for the company and started the entire game business as we know it today.

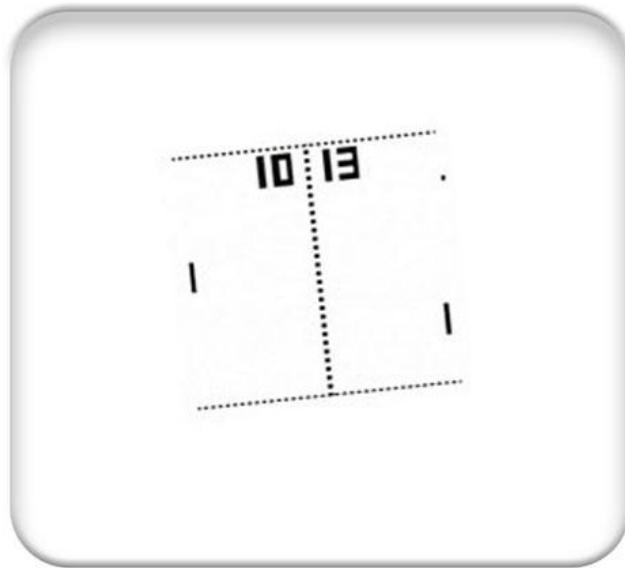


Figure 104 -The game PONG as it appears on a T-shirt

The game was a two player game where the only instructions were “Avoid missing ball for high score.”

Breakout

The story behind the original video game Breakout is one that combines hubris and the big double-cross. Nolan Bushnell, the founder of the game company Atari, came up with a variation of the game Pong (the video game that started the entire video game industry) where the player tried to clear from the top a row of bricks with a ball. You can go online and play a Flash version of the game at <http://smasher.the-game.us/>.

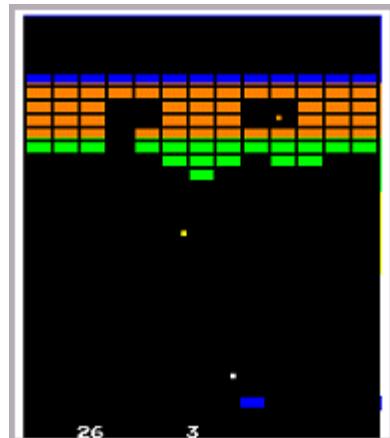


Figure 105 – Breakout

Nolan Bushnell wanted to keep the construction of a Breakout game cabinet circuitry²⁸ as cheap as possible so he challenged all his engineers to a bonus for whomever came up with the best (read cheapest) design, that is reduce the number of chips required. Steve Jobs (of Apple fame) was an Atari employee²⁹ at this time. Jobs talked his best friend Steve Wozniak (the inventor of the Apple computer) to try his hand. Jobs informed his friend Woz that the bonus money for reducing the original Breakout design was

\$750 when in fact the bonus was actually \$100 for each chip removed. Woz managed a great feat of engineering and reduced the design by 50 chips! What is more incredible is that he managed this in four days. Woz only got \$375 (half the fictitious bonus) while Jobs pocketed the rest.

The rest is of course history. The two Steve's went on to start up the company named Apple and Atari went on to make the worst³⁰ game ever – E.T.



Figure 106 - Steve Jobs and Steve Wozniak

MindSweeper

From: [http://en.wikipedia.org/wiki/Minesweeper_\(computer_game\)](http://en.wikipedia.org/wiki/Minesweeper_(computer_game))

Minesweeper is a single-player [computer game](#). The object of the game is to clear an abstract [minefield](#) without detonating a [mine](#). The game has been rewritten for nearly every [system platform](#) in use today. The most well-known version is [Minesweeper for the Windows platform](#), which comes bundled with later versions of the operating system.

²⁸ This was in time when video games were built using discrete logic circuits and not software (via a microprocessor).

²⁹ It was rumored that Jobs was spy for Bushnell since the Engineering team left Bushnell in the dark about what they were working on.

³⁰ It is rather difficult to make a game any dreadful...but many companies have come close.

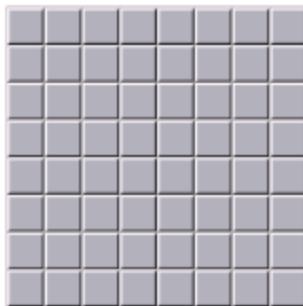


Figure 107 - Start of game

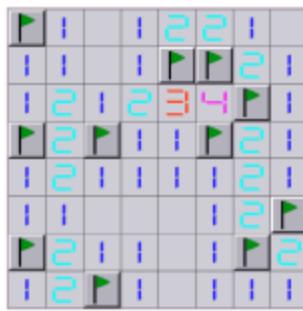


Figure 108 - Finished game

When the game is started, the player is presented by a grid of blank squares. The size of the grid is dependent on the skill level chosen by the player, with higher skill levels having larger grids. If the player clicks on a square without a mine, a digit is revealed in that square, the digit indicating the number of adjacent squares (typically, out of the possible 8) which contain mines. By using logic, players can in many instances use this information to deduce that certain other squares are mine-free (or mine-filled), and proceed to click on additional squares to clear them or mark them with flag graphics to indicate the presence of a mine.

The player can place a flag graphic on any square believed to contain a mine by right-clicking on the square. Right-clicking on a square that is flagged will change the flag graphic into a question mark to indicate that the square may or may not contain a mine. Right-clicking on a square marked with a question mark will set the square back to its original state. Squares marked with a flag cannot be cleared by left-clicking on them, though question marks can be cleared as easily as normal squares. The third question mark state is often deemed unnecessary and can be disabled so that right clicking on a flagged mine will set it back to its original state

right away so mines flagged in error can be corrected with one right-click instead of two.

In some versions of the game, middle-clicking (or clicking the left and right buttons at the same time) on a number having as many adjacent flags as the value of the number reveals all the unmarked squares neighboring the number; however, one forfeits the game should the flags be placed in error. This method is a very useful tool when trying to beat a high score. Some of those implementations also allow the player to move the mouse with the right mouse-button held down after marking mines; the player can then left-click on multiple numbered squares while dragging with the right mouse-button, in order to clear large areas in a short time. As an alternative to clicking both buttons at the same time players can also middle-click or shift-click on fully-flagged numbers.

Some implementations of minesweeper have a built in cheat option where the game will set up the board in favor of the player by never placing a mine on the first square clicked; some also change the board so the solution does not require guessing.

Appendix D – Unzipping files

There will be many files you will need to download to your PC. These files are compressed using various file compression formats. When you download a *.tar or *.zip file you will see that it contains actually many files in it. I highly recommend that you install a free utility that recognizes many different file formats. I use one recommended by my son – “7-zip File Manager”. You can obtain it at <http://www.7-zip.org/>.

You can download an exe version for Windows. It is open source and freely available under the GNI LGPL license.

After you install it when you right click on a compressed file such as `SDL-devel-1.2.14-mingw32.tar.tar` you will see an option to open the file with 7-Zip.

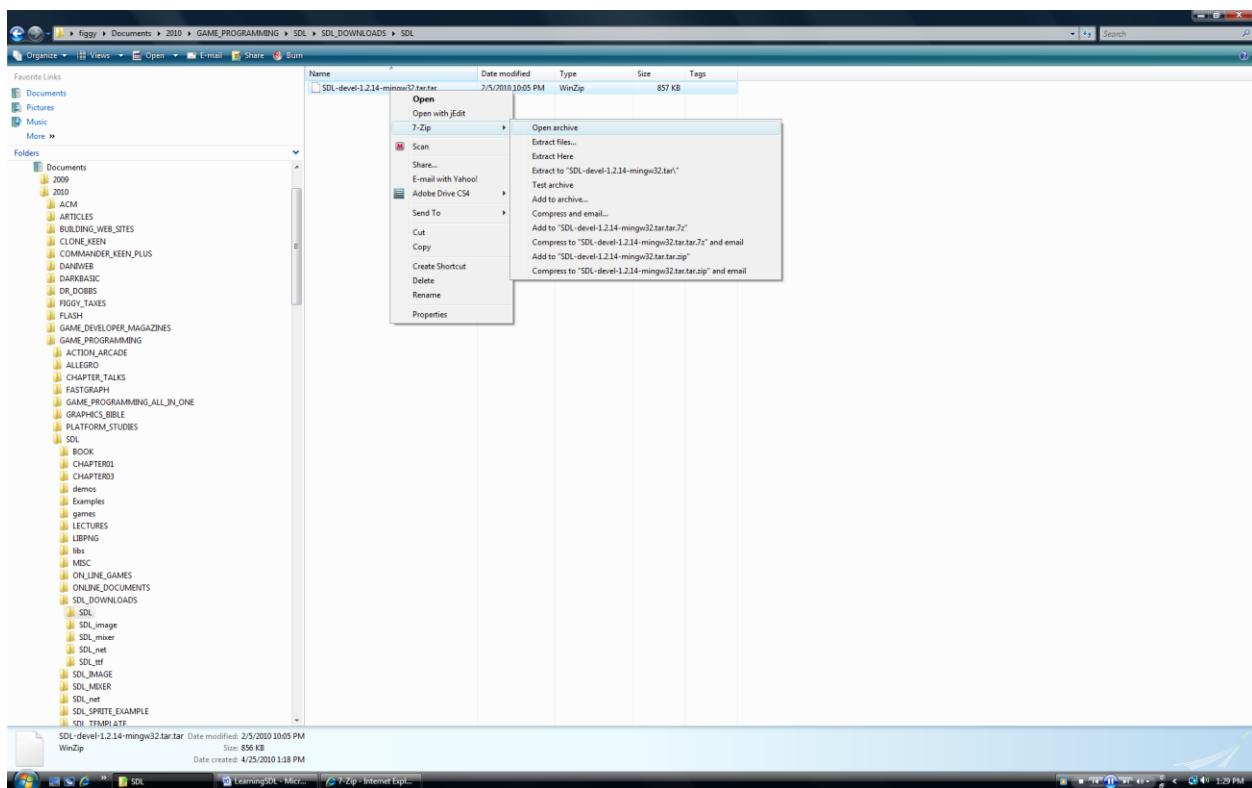


Figure 109 - Opening up a file archive with 7-zip

When you select to open up the compressed archive you will see:

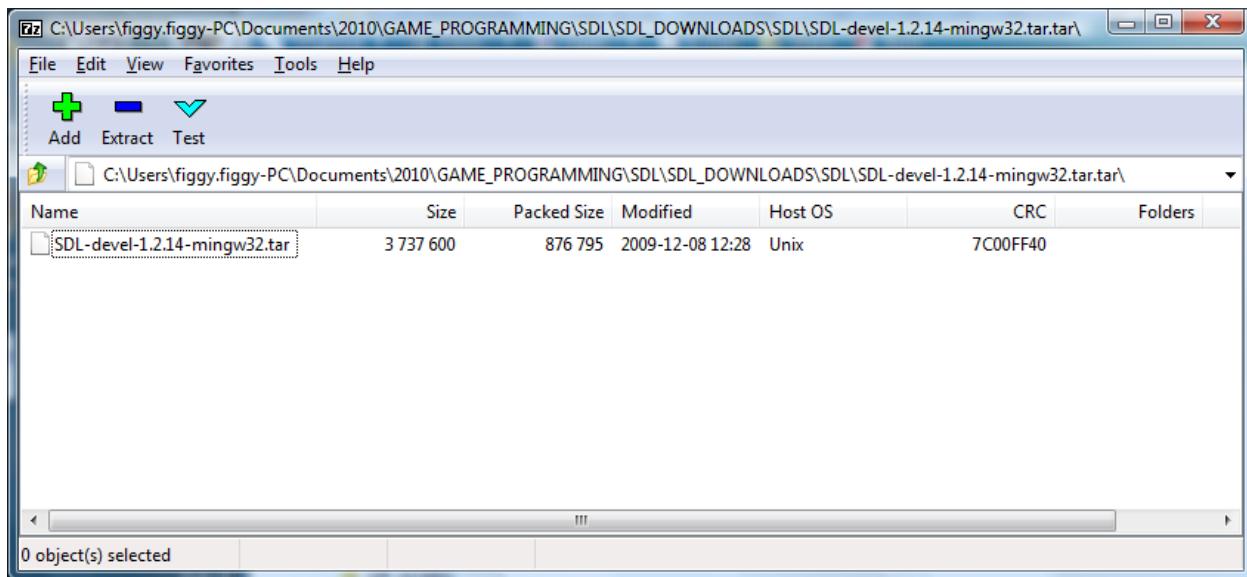


Figure 110 - Result of opening up archive

In this particular case since we don't see the folders or directory for the compressed files double click on the tar file.

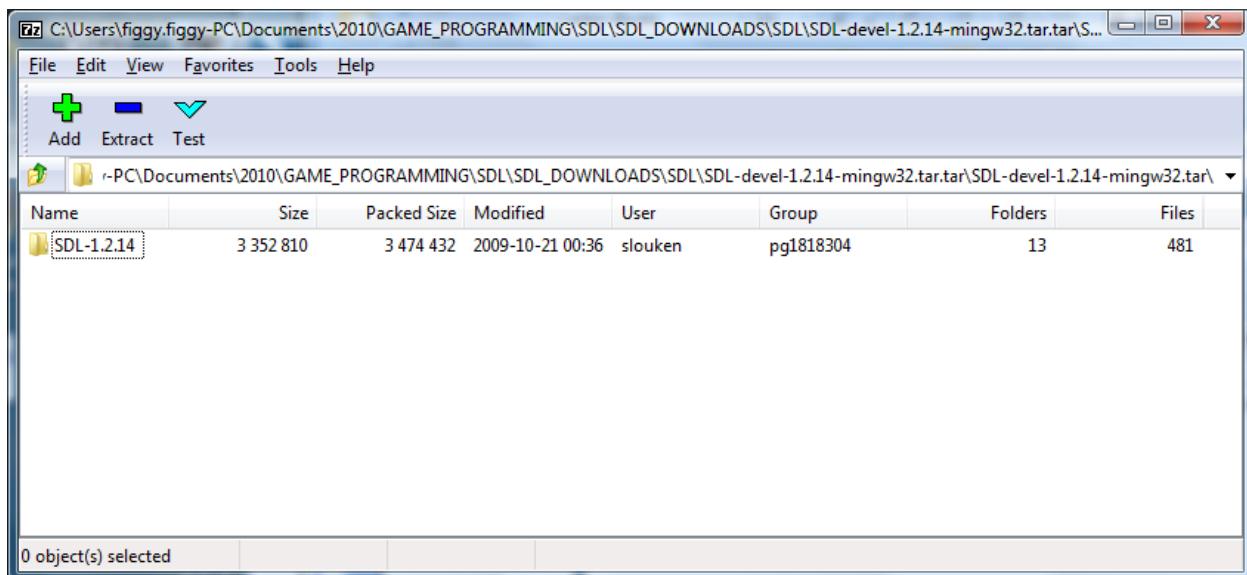


Figure 111 - SDL folder

You can continue to examine the files in the compressed folder by double clicking on the file name again. But, in our case we will extract directly to the C:\ drive. Click on “Extract”.

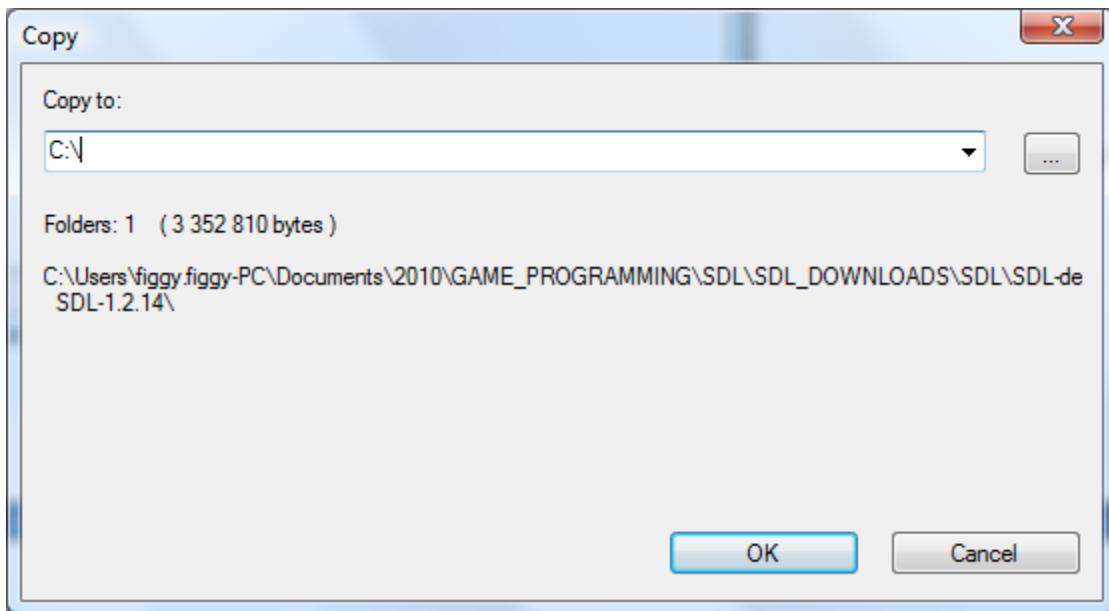


Figure 112 - Extracting the files to the C drive

You can copy to any location on your hard disk but I prefer to place into C:\ in order to easily locate the major libraries and tools I am using.

After you are done you should see a new directory under the C:\. Use Windows Explorer to see the folders that have been added under C:\SDL-1.2.14.

The key folders that you will use in your SDL development are:

- bin – contains the SDL.dll that will be needed in order to execute your SDL based programs
- docs – contains html files containing information on SDL, links to online tutorials, etc
- include/SDL – a set of *.h files that you will need to have your compiler use
- lib – libraries that you will need to move under your mingw based IDE compiler
- test – a set of test programs

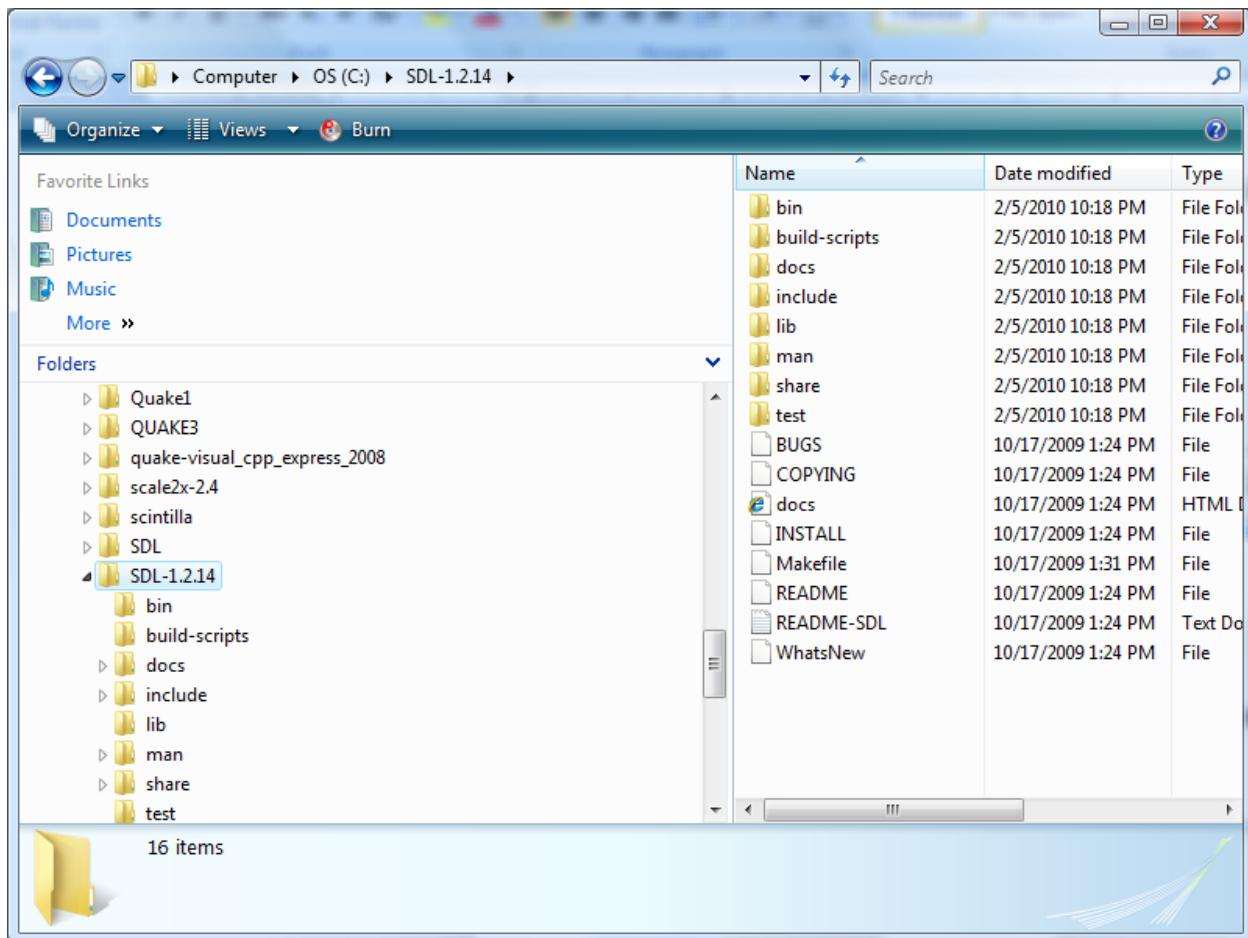


Figure 113 - SDL folders

This is the Simple DirectMedia Layer, a general API that provides low level access to audio, keyboard, mouse, joystick, 3D hardware via OpenGL, and 2D framebuffer across multiple platforms.

The current version supports Linux, Windows CE/95/98/ME/XP/Vista, BeOS, MacOS Classic, Mac OS X, FreeBSD, NetBSD, OpenBSD, BSD/OS, Solaris, IRIX, and QNX. The code contains support for Dreamcast, Atari, AIX, OSF/Tru64, RISC OS, SymbianOS, Nintendo DS, and OS/2, but these are not officially supported.

SDL is written in C, but works with C++ natively, and has bindings to several other languages, including Ada, C#, Eiffel, Erlang, Euphoria, Guile, Haskell, Java, Lisp, Lua, ML, Objective C, Pascal, Perl, PHP, Pike, Pliant, Python, Ruby, and Smalltalk.

This library is distributed under GNU LGPL version 2, which can be found in the file "COPYING". This license allows you to use SDL freely in commercial programs as long as you link with the dynamic

library.

The best way to learn how to use SDL is to check out the header files in the "include" subdirectory and the programs in the "test" subdirectory.

The header files and test programs are well commented and always up to date.

More documentation is available in HTML format in "docs/index.html", and

a documentation wiki is available online at:

<http://www.libsdl.org/cgi/docwiki.cgi>

The test programs in the "test" subdirectory are in the public domain.

Frequently asked questions are answered online:

<http://www.libsdl.org/faq.php>

If you need help with the library, or just want to discuss SDL related issues, you can join the developers mailing list:

<http://www.libsdl.org/mailing-list.php>

Enjoy!

Sam Lantinga

(slouken@libsdl.org)

Table 18 - **SDL README FILE**

Appendix E – Structs

What are structs?

Structs was how C programmers grouped different data types. Arrays (see Appendix TBD) are limited to elements of the same data type. In fact, the idea of a class evolved from the concept of a C-struct. A C++ struct can contain all the features in C++ classes

- ✚ access specifiers (public, private, protected)
- ✚ member functions
- ✚ constructors
- ✚ destructors



But for the most part we will be using them here as structures that contain different data types.

Why use structs?

For example suppose you created a program to demonstrate moving a paddle (as in the game Pong) around on the screen at first you may create the following variables:

```
int paddle_x;           // indicates the top_left x position on the screen;
int paddle_y;           // indicates the top_left y position on the screen;
int paddle_width;       // indicates the width of the paddle
int paddle_height;      // indicates the height of the paddle
int paddle_color;       // the color of the paddle
```

All the variables hold different pieces of information about the same object namely the game paddle.

You may wonder why we don't just use:

```
int x;                  // indicates the top_left x position on the screen;
int y;                  // indicates the top_left y position on the screen;
int width;              // indicates the width of the paddle
int height;             // indicates the height of the paddle
int color;              // the color of the paddle
```

One good reason is because our next program will combine the ball and paddle and the ball too will need x, y and color and fill_char so we will create similar variables:

```
int ball_x;
int ball_y;
int ball_color;
```

The problem with all these variables is that all the ones starting with the word paddle_ and ball_ are all related to the same objects in our game. We try to remind ourselves of the connection and relationship by prefixing all the variables with the same name. Suppose we wanted to create and use a function to initialize the paddle.

```
void initializePaddle( int& paddle_x, int& paddle_y, int& paddle_width,
    int& paddle_height, int& paddle_color ) {
    paddle_x = <middle of the screen>;
    paddle_y = <bottom of the screen>;
    paddle_width = PADDLE_WIDTH;
    paddle_height = PADDLE_HEIGHT;
    paddle_color = cWHITE;
}
```

You may wonder if there isn't some better way to package all these related variables. There is.

C++ provides a better way to package and organize variables that relate to the same object. C++ introduces a construct called `struct`.

A `struct` is another word for record. A struct allows us to group different data types together under one name. The `struct` will be used in our programs as a new data type. I conceptualize a struct like a special box I create with compartments. First we create a template (think cookie cutter) for the struct. This struct definition is used to create many other copies. Each copy will have its own name.

A visual example to help conceptualize the idea of a struct is to imagine we create a template box as shown below:

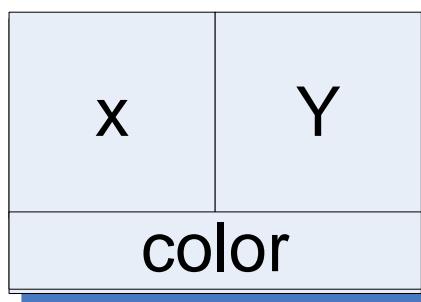


Figure 114 - visual representation of struct definition shapeStruct

The template box represents our struct definition. We may give this a name, for example, `shapeStruct`. This struct is composed of three variables `x`, `y` and `color`. We can now create two new actual variables, `paddle` and `ball` based on this `shapeStruct` we just defined.

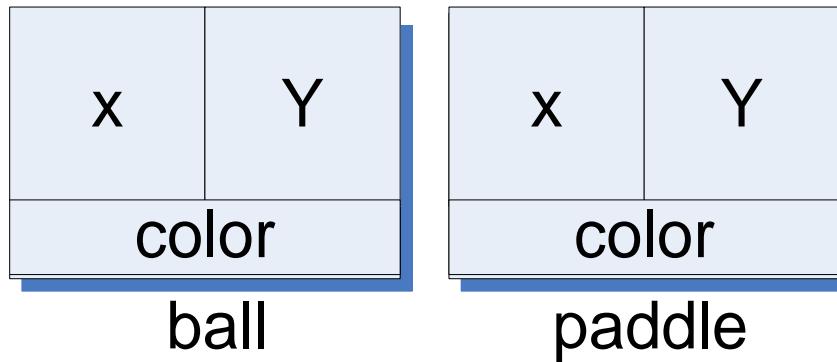


Figure 115 - Creating two structs using our template

The new variables ball and paddle each have the same three components or variables contained in them - x, y and color. We can distinguish it by using the following syntax:

```
ball.x
ball.y
ball.color
```

Same thing for paddle:

```
paddle.x
paddle.y
paddle.color
```

Note how we first use the name of our *struct* element (ball, paddle) and then specify the actual element within the *struct* by using ‘.’ And the *struct* member name (x, y, or color).

We can put things into the *struct*:

```
ball.x = 20;      // set the x position of the ball
ball.y = 5; // set the y position of the ball
ball.color = cBLUE;
```

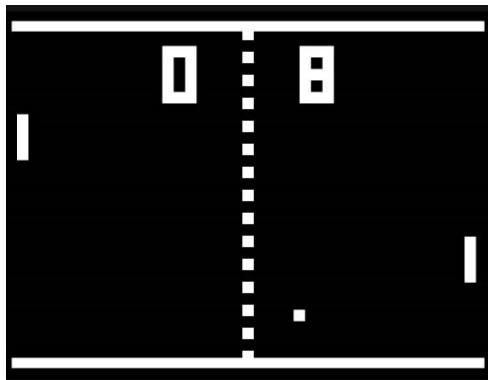
We can extract or use the contents of the record:

```
setCursorPos( ball.x, ball.y);      // set the cursor at the current location
// of the ball
```

The actual C++ syntax for defining a struct is:

```
struct <structname> {
    dataType1 memberName1;
    dataType2 memberName2;
    :
```

```
dataType3 memberNameN;
};
```



The members of a *struct* can consist of basic data types (e.g. int), other structs, or arrays. The struct definition does not reserve any memory space. You will need to create *struct* variables in order to reserve space.

For example we create a new structured data type named *paddleType*. We will use this new data type to create a variable to represent our paddle.

```
struct paddleType {
    int x;           // indicates the top_left x position on the screen;
    int y;           // indicates the top_left y position on the screen;
    int width;       // indicates the width of the paddle
    int height;      // indicates the height of the paddle
    int color;       // the color of the paddle
};
```

Now to create a paddle variable requires that we declare it:

```
paddleType paddle;
```

The variable *paddle* is a *struct*. It contains all the components we see in the definition of *paddleType*. So we now use *paddle.x*, *paddle.y*, *paddle.width*, *paddle.height*, and *paddle.color*.

The function we created earlier to initialize a paddle now can be created more concisely as:

```
void initializePaddle( paddleType& paddle ) {
    paddle.x = <middle of the screen>;
    paddle.y = <bottom of the screen>;
    paddle.width = PADDLE_WIDTH;
    paddle.height = PADDLE_HEIGHT;
    paddle.color = cWHITE;
}
```

Doesn't it look more organized? We collected all the attributes of a paddle into one component called a struct. We don't have to send around all the various pieces just the one struct variable named – *paddle*. The variable *paddle* consists or contains all the pieces for us. We pass the *paddleType* struct to the function by reference, that is, we send the address of the struct *paddle* to the function so that its members can be initialized.

You can create the variables the same time you create the struct definition by using this format:

```
struct <structname> {
    dataType1 memberName1;
    dataType2 memberName2;
    :
    dataType3 memberNameN;
} <structVariable1>, <structVariable2>, . . . , <structVariableN>;
```

This is how we could have defined the struct for paddleType and created two player paddles.

```
struct paddleType {
    int x;           // indicates the top_left x position on the screen;
    int y;           // indicates the top_left y position on the screen;
    int width;       // indicates the width of the paddle
    int height;      // indicates the height of the paddle
    int color;       // the color of the paddle
} paddlePlayer1, paddlePlayer2;
```

In summary

- ⊕ We use a *struct* to help us group related variables
- ⊕ We first create the *struct* definition
 - this does not allocate any memory
 - the struct definition MUST end with a semicolon
 - this merely creates a template for us to use
 - the components of the *struct* are called *members* of the struct
 - Example: x, y, width, height, and color are members of the struct paddleType
- ⊕ We then create *struct* variable
 - to create or declare variables based on the struct we use the format:
 - < structname ><variablename>
 - Example: paddleType paddle;
- ⊕ We then treat each struct member as a variable
 - to access struct members (the x, y and color, etc) we use the following format:
 - <variablename>.<membername>
 - Example: paddle.color

- ⊕ The struct members can have different data types, this is what makes struct a heterogeneous data type, that is, you can have a mix of int, char, double, string, etc. whatever makes sense to what you are building.

Things you can do with structs

- ⊕ You can initialize a struct when you create it

```
PaddleType myPaddle = { 0, 0, 10, 40, cBLUE };
```

- ⊕ Assign one struct variable to another

```
// create two paddleType struct variables
paddleType player1, player2;
initializePaddle( player1 ); // Initialize player 1 paddle
player2 = player1; // Assign player2 the values in player1
```

The example above creates two struct paddleTypes for player1 and player2. The initializePaddle will initialize the members of player1. The assignment statement will copy all the member values in player1 to player2. If the initialization put the value cWHITE into player1.color then after the assignment player2.color will have the same value. Neat!

- ⊕ You can create a struct that has another struct within it.

Suppose we have a struct named COORD that holds the x and y coordinate of any object that we display on the screen.

```
struct COORD {
    int x;
    int y;
};
```

We may want to re-use the struct COORD that is already defined in your own new struct that represents the paddle:

```
struct paddleType2 {
    COORD screenLocation; // the screen location of the paddle
    int width; // indicates the width of the paddle
    int height; // indicates the height of the paddle
    int color; // the color of the paddle
};
```

This new definition for a paddle will require that access to the x and y values that represent the top-left position of the paddle on the screen will need to change.

```
paddleType2 playerPaddle; // create a variable for the player
playerPaddle.screenLocation.X = 0; // set x location
playerPaddle.screenLocation.Y = 10; // set y location
```

Note the difference between how x and y values are accessed, you need to use the struct variable name (playerPaddle), the variable name that is a struct itself (screenLocation) and then the actual variable name (X) within that struct.

Things you can't do with structs

- + You can't compare one struct variable with another. So if you had the following declaration:

```
paddleType player1, player2;
:
:
// can't do this
if ( player1 == player2 ) {
```

You would need to compare the members of player1 and player2 member by member:

```
if ( player1.x == player2.x &&
     player1.y == player2.y ) {
```

- + You can't read or write into a struct variable as one entity

```
paddleType player1;
cin >> player1; // not allowed
cout << player1; // not allowed
```

Again, you will need to input/out into the members of a struct.

```
cin >> player1.x >> player1.y;
cout << "Player 1 paddle.x = " << player1.x << endl;
cout << "Player 1 paddle.y = " << player1.y << endl;
```

Using typedef with structs

A typedef allows us to associate another name or an alias with a struct. The alias is usually shorter and easier to use than the original name. The format is:

```
typedef [attributes] <datatype> <aliasName>;
```

One will typically simple uses of typedef such as:

```
typedef int km_per_hour;
typedef unsigned char Uint8;
its use in your program:
```

```
km_per_hour current_speed;
km_per_hour new_speed;
```



Figure 116 - Is there any difference?

In C you would have to use the actual keyword struct when declaring a struct variable, for example:

```
struct paddleType myPaddle;
```

to avoid that C programmers are in the habit of providing an alias for the entire struct so they can declare variables without having to specify the term struct.

A typical use of this style is the declaration of `SDL_Color` struct in the `sdl_video.h`:

```
typedef struct SDL_Color {
    Uint8 r;
    Uint8 g;
    Uint8 b;
    Uint8 unused;
} SDL_Color;
```

The above creates the alias `SDL_color` for the struct `SDL_color` structure. So in C or C++ or you would need to do to declare a variable of `SDL_color` is the follow declaration:

```
SDL_color myColor;
```

Appendix F – Pointers

When a programmer declares a variable:

```
int aNumber;
```

we don't think too much about the fact that when we use the name aNumber that we are actually referencing the memory address where our (let's say) 32-bit int is being stored:

[SDLKey](#)
[SDL_ActiveEvent](#)
[SDL_AddTimer](#)
[SDL_AudioCVT](#)
[SDL_AudioSpec](#)
[SDL_BlitSurface](#)
[SDL_BuildAudioCVT](#)
[SDL_CD](#)
[SDL_CDClose](#)
[SDL_CDEject](#)
[SDL_CDName](#)
[SDL_CDNumDrives](#)
[SDL_CDOpen](#)
[SDL_CDPause](#)
[SDL_CDPlay](#)
[SDL_CDPlayTracks](#)
[SDL_CDResume](#)
[SDL_CDStatus](#)
[SDL_CDStop](#)
[SDL_CDtrack](#)
[SDL_CloseAudio](#)
[SDL_Color](#)
[SDL_CondBroadcast](#)
[SDL_CondSignal](#)
[SDL_CondWait](#)
[SDL_CondWaitTimeout](#)
[SDL_ConvertAudio](#)
[SDL_ConvertSurface](#)
[SDL_CreateCond](#)
[SDL_CreateCursor](#)
[SDL_CreateMutex](#)
[SDL_CreateRGBSurface](#)
[SDL_CreateRGBSurfaceFrom](#)
[SDL_CreateSemaphore](#)
[SDL_CreateThread](#)
[SDL_CreateYUVOverlay](#)
[SDL_Delay](#)
[SDL_DestroyCond](#)
[SDL_DestroyMutex](#)
[SDL_DestroySemaphore](#)
[SDL_DisplayFormat](#)
[SDL_DisplayFormatAlpha](#)
[SDL_DisplayYUVOverlay](#)
[SDL_EnableKeyRepeat](#)
[SDL_EnableUNICODE](#)
[SDL_Event](#)
[SDL_EventState](#)
[SDL_FillRect](#)

Chapter 2

SDL_Flip	Chapter 2
SDL_FreeCursor	
SDL_FreeSurface	Chapter 2
SDL_FreeWAV	
SDL_FreeYUVOverlay	
SDL_GL_GetAttribute	
SDL_GL_GetProcAddress	
SDL_GL_LoadLibrary	
SDL_GL_SetAttribute	
SDL_GL_SwapBuffers	
SDL_GLAttr	
SDL_GetAppState	
SDL_GetAudioStatus	
SDL_GetClipRect	
SDL_GetCursor	
SDL_GetEventFilter	
SDL_GetGamma	
SDL_GetGammaRamp	
SDL_GetKeyName	
SDL_GetKeyState	
SDL_GetModState	
SDL_GetMouseState	
SDL_GetRGB	
SDL_GetRGBA	
SDL_GetRelativeMouseState	
SDL_GetThreadID	
SDL_GetTicks	Chapter 2
SDL_GetVideoInfo	Chapter 2
SDL_GetVideoSurface	
SDL_Init	Chapter 2
SDL_InitSubSystem	Chapter 2
SDL_JoyAxisEvent	
SDL_JoyBallEvent	
SDL_JoyButtonEvent	
SDL_JoyHatEvent	
SDL_JoystickClose	
SDL_JoystickEventState	
SDL_JoystickGetAxis	
SDL_JoystickGetBall	
SDL_JoystickGetButton	
SDL_JoystickGetHat	
SDL_JoystickIndex	
SDL_JoystickName	
SDL_JoystickNumAxes	
SDL_JoystickNumBalls	
SDL_JoystickNumButtons	
SDL_JoystickNumHats	
SDL_JoystickOpen	

SDL_JoystickOpened	
SDL_JoystickUpdate	
SDL_KeyboardEvent	
SDL_KillThread	
SDL_ListModes	Chapter 2
SDL_LoadBMP	
SDL_LoadWAV	
SDL_LockAudio	
SDL_LockSurface	Chapter 2
SDL_LockYUVOverlay	
SDL_MapRGB	Chapter 2
SDL_MapRGBA	
SDL_MixAudio	
SDL_MouseButtonEvent	
SDL_MouseMotionEvent	
SDL_NumJoysticks	
SDL_OpenAudio	
SDL_Overlay	
SDL_Palette	
SDL_PauseAudio	
SDL_PeepEvents	
SDL_PixelFormat	
SDL_PollEvent	
SDL_PumpEvents	
SDL_PushEvent	
SDL_Quit	Chapter 2
SDL_QuitEvent	
SDL_QuitSubSystem	Chapter 2
SDL_RWFromFile	
SDL_Rect	Chapter 2
SDL_RemoveTimer	
SDL_ResizeEvent	
SDL_SaveBMP	
SDL_SemPost	
SDL_SemTryWait	
SDL_SemValue	
SDL_SemWait	
SDL_SemWaitTimeout	
SDL_SetAlpha	Chapter 2
SDL_SetClipRect	Chapter 2
SDL_SetColorKey	Chapter 2
SDL_SetColors	
SDL_SetCursor	
SDL_SetEventFilter	
SDL_SetGamma	
SDL_SetGammaRamp	
SDL_SetModState	
SDL_SetPalette	

SDL_SetTimer	
SDL_SetVideoMode	Chapter 2
SDL_ShowCursor	
SDL_Surface	Chapter 2
SDL_SysWMEEvent	
SDL_ThreadID	
SDL_UnlockAudio	
SDL_UnlockSurface	
SDL_UnlockYUVOverlay	
SDL_UpdateRect	Chapter 2
SDL_UpdateRects	
SDL_UserEvent	
SDL_VideoDriverName	
SDL_VideoInfo	Chapter 2
SDL_VideoModeOK	
SDL_WM_GetCaption	
SDL_WM_GrabInput	
SDL_WM_IconifyWindow	
SDL_WM_SetCaption	Chapter 2
SDL_WM_SetIcon	
SDL_WM_ToggleFullScreen	
SDL_WaitEvent	
SDL_WaitThread	
SDL_WarpMouse	
SDL_WasInit	Chapter 2
SDL_keysym	
SDL_mutexP	
SDL_mutexV	