Simple DirectMedia Layer (SDL)

1. What is SDL?

The Simple DirectMedia Layer (SDL) is library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D. Since SDL is a library entirely written in C, it is also a cross-platform development library, supporting Windows, Mac OS X, Linux, iOS, and Android. Given its portability and ease of use, SDL became increasingly popular over the years. It is used for video playback software, emulators, many games and also the most popular game engines. Thanks to the SDL library and its portability, most game engines can export the same game for several platforms.

2. Using SDL2 library

```
#include <SDL2/SDL.h>

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

To work with SDL you first need to initialize the various SDL subsystems you want to use. This is done through the `SDL_Init()` function, which takes a set of flags for specifying the subsystems you'd like to initialize. Those flags may be any of the following OR'd together:

Flag	Description
`SDL_INIT_TIMER`	timer subsystem
`SDL_INIT_AUDIO`	audio subsystem
`SDL_INIT_VIDEO`	video subsystem. Automatically initializes the SDL_INIT_EVENTS subsystem
`SDL_INIT_JOYSTICK`	joystick subsystem
`SDL_INIT_HAPTIC`	haptic (force feedback) subsystem
`SDL_INIT_GAMECONTROLLER`	controller subsystem. Automatically initializes the SDL_INIT_JOYSTICK subsystem
`SDL_INIT_EVENTS`	events subsystem
`SDL_INIT_EVERYTHING`	all of the above subsystems
`SDL_INIT_NOPARACHUTE`	compatibility; this flag is ignored

Table 1: `SDL_Init(Uint32)` **subsystem flags.**

For instance, if you want to initialize the video system, you could do that by doing so:

```
SDL_Init(SDL_INIT_VIDEO);
```

Also, if you'd like to initialize two or subsystems, you can do the following way:

```
SDL_Init(SDL_INIT_VIDEO|SDL_INIT_AUDIO);
```

If you want to initialize all the SDL subsystems you can do it this way:

```
SDL_Init(SDL_INIT_EVERYTHING);
```

For now, we'll just need the video subsystem but we'll add more flags as we require more features. The event handling system (specified by SDL_INIT_EVENTS in Table 1) is initialized automatically when the video system is, even if not explicitly requested by itself, while the file I/O and threading systems are initialized by default. If everything goes ok `SDL_Init()` will return 0, if not we must print out the error and quit:

```
#include <iostream>
#include <SDL2/SDL.h>

int main(int argc, char *argv[])
{
    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
        std::cout << "Video Initialization Error: " << SDL_GetError() << std::endl;
        return 1;
    }

    /* ... */
    SDL_Quit();

    return 0;
}
```

Or, if you prefer, an object oriented version of the previous one:

```
#include <exception>
#include <string>
#include "SDL2/SDL.h"

class InitError: public std::exception {
public:
    InitError();
    InitError(const std::string&);
    virtual ~InitError() throw();
    virtual const char* what() const throw();
private:
    std::string msg;
};

InitError::InitError():
exception(), msg(SDL_GetError()) {}
InitError::InitError(const std::string& m):
exception(), msg(m) {}
InitError::~InitError() throw() {}
const char* InitError::what() const throw() {
    return msg.c_str();
}


class SDL {
public:
    SDL(Uint32 flags = 0) throw(InitError);
    virtual ~SDL();
};

SDL::SDL(Uint32 flags) throw(InitError) {
```

```
    if (SDL_Init(flags) != 0)
        throw InitError();
}

SDL::~SDL() {
    SDL_Quit();
}

/* ... */

#include <iostream>

int main(int argc, char **argv) {
    try {
        SDL sdl(SDL_INIT_VIDEO|SDL_INIT_TIMER);

        /* ... */

        return 0;
    }

    catch (const InitError& err) {
        std::cerr
        << "Error while initializing SDL:  "
        << err.what() << std::endl;
    }

    return 1;
}
```

Here, SDL is a concrete class that makes a call to `SDL_Init()` inside its constructor. If any error occurs during the initialization of the subsystems specified by `flags`, an `InitError` exception is thrown. Since `InitError` is an exception (is a subclass of std::exception) it can be thrown and catched like so. Since the `sdl` variable is a concrete type, its destructor is executed automatically when it reaches the end of its scope and, thus, making a call to the `SDL_Quit()` function.

2.1. Creating a window

```
#include<SDL2/SDL.h>

SDL_Window* SDL_CreateWindow(const char* title, int x, int y, int width, int height,
       Uint32 flags);

void SDL_DestroyWindow(SDL_Window* window);
```

The `SDL_CreateWindow()` function creates a window with the specified title, position, dimensions, and flags. This function has the following parameters:

Flag	Description
`const char* title`	the title of the window, in UTF-8 encoding
`int x`	the x position of the window, SDL_WINDOWPOS_CENTERED, or SDL_WINDOWPOS_UNDEFINED
`int y`	the y position of the window, SDL_WINDOWPOS_CENTERED, or SDL_WINDOWPOS_UNDEFINED
`int width`	the width of the window, in screen coordinates
`int height`	the height of the window, in screen coordinates
`Uint32 flags`	0, or one or more SDL_WindowFlags OR'd together (see Table 3)

Table 2: `SDL_CreateWindow()` **function parameters.**

The `flags` parameter, can be any of the following OR'd together:

Flag	Description
`SDL_WINDOW_FULLSCREEN`	fullscreen window
`SDL_WINDOW_FULLSCREEN_DESKTOP`	fullscreen window at the current desktop resolution
`SDL_WINDOW_OPENGL`	window usable with OpenGL context
`SDL_WINDOW_HIDDEN`	window is not visible
`SDL_WINDOW_BORDERLESS`	no window decoration
`SDL_WINDOW_RESIZABLE`	window can be resized
`SDL_WINDOW_MINIMIZED`	window is minimized
`SDL_WINDOW_MAXIMIZED`	window is maximized
`SDL_WINDOW_INPUT_GRABBED`	window has grabbed input focus
`SDL_WINDOW_ALLOW_HIGHDPI`	window should be created in high-DPI mode if supported (>= SDL 2.0.1)

Table 3: `SDL_CreateWindow()` **flags.**

The `SDL_Window` is implicitly shown (`SDL_SHOWN`) if `SDL_WINDOW_HIDDEN` is not set. `SDL_WINDOW_SHOWN` may be queried later using `SDL_GetWindowFlags()`. On Apple's OS X you must set the `NSHighResolutionCapable` `Info.plist` property to YES, otherwise you will not receive a High DPI OpenGL canvas. If the window is created with the `SDL_WINDOW_ALLOW_HIGHDPI` flag, its size in pixels may differ from its size in screen coordinates on platforms with high-DPI support (e.g. iOS and Mac OS X). Use SDL_GetWindowSize() to query the client area's size in screen coordinates, and `SDL_GL_GetDrawableSize()` or `SDL_GetRendererOutputSize()` to query the drawable size in pixels. If the window is set fullscreen, the `width` and `height` parameters will not be used. However, invalid size parameters (e.g. too large) may still fail. Window size is actually limited to 16384 x 16384 for all platforms at window creation.

This function returns the window that was created or NULL in case of failure:

```
#include<SDL2/SDL.h>
#include<iostream>

int main(int argc, char *argv[])
{
    SDL_Window *window = nullptr;

    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
        std::cout << "Video Initialization Error: " << SDL_GetError() << std::endl;
        return 1;
    }

    window = SDL_CreateWindow("TAPJ", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
640, 480, SDL_WINDOW_OPENGL);

    if(window == NULL) {
        std::cout << "Window Creation Error: " << SDL_GetError() << std::endl;
```

```
        return 2;
    }
    SDL_UpdateWindowSurface(window);
    SDL_Delay(2000);

    SDL_DestroyWindow(window);
    SDL_Quit();

    return 0;
}
```

Note that before `SDL_Quit()`, we call `SDL_DestroyWindow()`. This call is to avoid resource leaks: free the allocated resources when `window` was created and then destroy it.

2.2. Working with bitmaps

```
#include <SDL2/SDL.h>

SDL_Surface* SDL_GetWindowSurface(SDL_Window* window);
SDL_Surface* SDL_LoadBMP(const char* file);
int SDL_BlitSurface(SDL_Surface* src, const SDL_Rect* srcrect, SDL_Surface* dst,
       SDL_Rect* dstrect);
int SDL_UpdateWindowSurface(SDL_Window* window);
```

For loading a bitmap and displaying it to the window, you need a **SDL Surface**. You think of a SDL surface as a canvas. Each SDL window has a SDL surface built into it: that is why you see a blank background when you create a window. With that canvas you can paint a color into it or we can load an image onto it. Also, you can paint other canvases onto other canvases, and so on.

For displaying a bitmap, first you need to loading an image onto a **surface** and then, paint it onto the window surface:

```
#include <iostream>
#include <SDL2/SDL.h>

int main(int argc, const char * argv[]) {
    SDL_Window *window = nullptr;
    SDL_Surface *windowSurface = nullptr;
    SDL_Surface *image = nullptr;

    if(SDL_Init(SDL_INIT_VIDEO) < 0) {
        std::cout << "Video Initialization Error: " << SDL_GetError() << std::endl;
        return 1;
    }

    window = SDL_CreateWindow("SDL TAPJ", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, 640, 480, SDL_WINDOW_OPENGL);
    if(window == NULL) {
        std::cout << "Window creation Error: " << SDL_GetError() << std::endl;
        return 2;
    }

    windowSurface = SDL_GetWindowSurface(window);
    image = SDL_LoadBMP("galaxy2.bmp");
    if(image != NULL)
    {
        SDL_BlitSurface(image, NULL, windowSurface, NULL);
        SDL_UpdateWindowSurface(window);
        SDL_Delay(2000);
    }

    SDL_FreeSurface(image);
    image = nullptr;
    SDL_DestroyWindow(window);
    window = nullptr;
```

```
    SDL_Quit();

    return 0;
}
```

When a SDL window is created with `SDL_CreateWindow()`, a new surface is created with the optimal format for the window, if necessary. This surface will be freed when the window is destroyed by calling `SDL_DestroyWindow()`. Do not free this surface.

Note: the surface **will be invalidated if the window is resized**. After resizing a window this **function must be called again to return a valid surface**.

`SDL_BlitSurface()` performs a fast surface copy to a destination surface. `SDL_UpdateWindowSurface()` copies the window surface to the screen. For further details about these and other SDL2 functions, please go to: https://wiki.libsdl.org/APIByCategory.

2.3. Game loop

The game loop consists of a loop (typically a `while`) performing all the game logic and updating the window surface, at each iteration:

```
#include<SDL2/SDL.h>
#include<iostream>

int main(int argc, char *argv[])
{
    SDL_Window *window = nullptr;
    SDL_Init(SDL_INIT_VIDEO);

    window = SDL_CreateWindow("SDL window", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, 640, 480, SDL_WINDOW_OPENGL);

    bool isRunning = true;

    while(isRunning)
    {
        /* Game Loop */

        SDL_UpdateWindowSurface(window);
    }

    SDL_DestroyWindow(window);
    SDL_Quit();

    return 0;
}
```

2.4. Event Loop

```
#include <SDL2/SDL.h>

int SDL PollEvent(SDL Event* event);
```

During each frame update, several events might occur. For example, if you pressed a key and moved the mouse, SDL will store those two events on the **event queue**. Thus, at each frame update or game loop iteration, there might be several events on the event queue. In order to handle all the events that have occurred since the last frame update, you need to retrieve them,

one by one from the event queue: **event loop**. This way, you can choose what to do when an event happens.

By calling `SDL_PollEvent()` at each frame update, it will give you the list of events that happened. For instance, if you pressed a key and moved the mouse you have two events on the event queue. With the event loop, you can get each event that is in the queue and process it. While we still have events on the event queue, we will process those events:

```
#include<SDL2/SDL.h>
#include<iostream>

int main(int argc, char *argv[])
{
    SDL_Window *window = nullptr;
    SDL_Init(SDL_INIT_VIDEO);

    window = SDL_CreateWindow("SDL window", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, 640, 480, SDL_WINDOW_OPENGL);

    bool isRunning = true;
    SDL_Event ev;

    while(isRunning)
    {
        while(SDL_PollEvent(&ev) != 0)
        {
            if(ev.type == SDL_QUIT)
                isRunning = false;
        }
        SDL_UpdateWindowSurface(window);
    }

    SDL_DestroyWindow(window);
    SDL_Quit();

    return 0;
}
```

In this specific case, `SDL_QUIT` event type means that the user has pressed the close button of the SDL window. **Note:** the code for error handling has been omitted.

2.4.1. Handling keyboard events

Besides handling the quit button, if you also want to handle keyboard events, you can do it like so:

```
#include<SDL2/SDL.h>
#include<iostream>

int main(int argc, char *argv[])
{
    // Creating and initializing variables
    SDL_Window *window = nullptr;
    SDL_Surface *windowSurface = nullptr;
    SDL_Surface *image1 = nullptr;
    SDL_Surface *image2 = nullptr;
    SDL_Surface *image3 = nullptr;
    SDL_Surface *currentImage = nullptr;
    SDL_Init(SDL_INIT_VIDEO);

    window = SDL_CreateWindow("SDL window", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, 640, 480, SDL_WINDOW_OPENGL);
    windowSurface = SDL_GetWindowSurface(window);

    image1 = SDL_LoadBMP("test.bmp");
    image2 = SDL_LoadBMP("test1.bmp");
    image3 = SDL_LoadBMP("test2.bmp");
```

```
    currentImage = image1;

    bool isRunning = true;
    SDL_Event ev;

    while(isRunning)
    {
        while(SDL_PollEvent(&ev) != 0)
        {
            // Getting the quit and the keyboard events
            if(ev.type == SDL_QUIT)
                isRunning = false;
            else if(ev.type == SDL_KEYUP)
            {
                switch(ev.key.keysym.sym)
                {
                    case SDLK_1:
                        currentImage = image1;
                        break;
                    case SDLK_2:
                        currentImage = image2;
                        break;
                    case SDLK_3:
                        currentImage = image3;
                        break;
                }
            }
        }

        // Drawing the current image to the window
        SDL_BlitSurface(currentImage, NULL, windowSurface, NULL);
        SDL_UpdateWindowSurface(window);
    }

    // Freeing the memory
    SDL_FreeSurface(image1);
    SDL_FreeSurface(image2);
    SDL_FreeSurface(image3);
    SDL_DestroyWindow(window);

    image1 = image2 = image3 = windowSurface = nullptr;
    window = nullptr;

    SDL_Quit();

    return 0;
}
```

Note: for better code readability, the error handling code has not been included. However, you should **always** perform error check.

2.4.2. Handling Mouse Events

Similarly, for handling both mouse events and the quit event you can do the following **event loop**:

```
while(SDL_PollEvent(&ev) != 0)
{
    // Getting the quit and the keyboard events
    if(ev.type == SDL_QUIT)
        isRunning = false;
    else if(ev.type == SDL_MOUSEMOTION)
    {
        if(ev.button.x < 200)
            currentImage = image1;
        else if(ev.button.x < 400)
            currentImage = image2;
        else
            currentImage = image3;
    }
```

```
}
```

2.4.3. Using Joystick and Gamepad controllers

In order to SDL also detect Joystick and/or Gamepad events, you must **initialize the joystick subsystem**, besides the video subsystem:

```
SDL_Init(SDL_INIT_VIDEO | SDL_INIT_JOYSTICK);

std::cout << SDL_NumJoysticks() << std::endl;
SDL_Joystick *joystick = SDL_JoystickOpen(0);
if (joystick == NULL)
    std::cout << "Unable to open Gamepad " << SDL_GetError() << std::endl;
else
{
    std::cout << "Name: " << SDL_JoystickName(joystick) << std::endl;
    std::cout << "Num Axes: " << SDL_JoystickNumAxes(joystick) << std::endl;
    std::cout << "Num Buttons: " << SDL_JoystickNumButtons(joystick) << std::endl;
    std::cout << "Num Track balls: " << SDL_JoystickNumBalls(joystick) << std::endl;
    std::cout << "Num Hats: " << SDL_JoystickNumHats(joystick) << std::endl;
}

SDL_JoystickClose(joystick);
```

For instance, if you have a PlayStation 4 controller plugged into your computer, this code has the following output:

```
1
Name: Wireless Controller
Num Axes: 6
Num Buttons: 14
Num Track balls: 0
Num Hats: 1
```

Similarly, the **event loop** for handling joystick data should be like so:

```
while (SDL_PollEvent(&ev) != 0)
{
    // Getting the quit and the keyboard events
    if (ev.type == SDL_QUIT)
        isRunning = false;
    else if (ev.type == SDL_JOYAXISMOTION)
    {
        if (ev.jaxis.which == 1)
        {
            //std::cout << ev.jaxis.value << std::endl;
            if (ev.jaxis.axis == 5)
            {
                std::cout << ev.jaxis.value << std::endl;
                if (ev.jaxis.value < -8000)
                    currentImage = image2;
                else if (ev.jaxis.value > 8000)
                    currentImage = image3;
            }
        }
    }
}
```

Also, if you want to handle joystick button events:

```
while(SDL_PollEvent(&ev) != 0)
{
    // Getting the quit and the keyboard events
    if(ev.type == SDL_QUIT)
        isRunning = false;
```

```
    else if(ev.type == SDL_JOYBUTTONDOWN)
    {
        if(ev.jbutton.which == 1)
        {
            if(ev.jbutton.button == 0)
                currentImage = image2;
            else if(ev.jbutton.button == 10)
                currentImage = image3;
        }
    }
}
```

2.4.4. The GameController API

The Game Controller API works on top of the Joystick API. However, with this one, instead of manually set and mapping each axis or button values, the GameController already maps each button and axis to the Xbox 360 buttons. This way, the same game, even in different operating systems, will behave the same way if you have a Xbox Controller.

```
SDL_GameController *controller;
int i;

SDL_Init(SDL_INIT_GAMECONTROLLER);

for (i = 0; i < SDL_NumJoysticks(); ++i) {
    if (SDL_IsGameController(i)) {
        char *mapping;
        std::cout << "Index '" << i << "' is a compatible controller, named '" <<
SDL_GameControllerNameForIndex(i) << "'" << std::endl;
        controller = SDL_GameControllerOpen(i);
        mapping = SDL_GameControllerMapping(controller);
        std::cout << "Controller " << i << " is mapped as \"" << mapping << std::endl;
        SDL_free(mapping);
    } else {
        std::cout << "Index '" << i << "' is not a compatible controller." << std::endl;
    }
}


if(controller != NULL)
    SDL_GameControllerClose(controller);
```

For instance, if you have a PlayStation 4 controller plugged into your computer, this code has the following output:

```
Index '0' is a compatible controller, named 'PS4 Controller'
Controller 0 is mapped as "4c05000000000000c405000000000000,PS4
Controller,a:b1,b:b2,back:b8,dpdown:h0.4,dpleft:h0.8,dpright:h0.2,dpup:h0.1,guide:b12,le
ftshoulder:b4,leftstick:b10,lefttrigger:a3,leftx:a0,lefty:a1,rightshoulder:b5,rightstick
:b11,righttrigger:a4,rightx:a2,righty:a5,start:b9,x:b0,y:b3,
```

The GameController API also lets you get and set mappings:

```
std::cout << SDL_GameControllerAddMapping("0,PS4 Controller,a:b11,b:b10");
SDL_GameControllerUpdate(); // Updates the controller if you're not in a loop

SDL_GameControllerGetStringForButton(SDL_CONTROLLER_BUTTON_A); // Gives you the mapping
name for the specified button

SDL_GameControllerGetStringForAxis(AXIS_GOES_HERE); // Gives you the mapping name for
the specified axis
```

Event loop using controller mappings:

```
while(SDL_PollEvent(&ev) != 0)
{
    // Getting the quit and the keyboard events
    if(ev.type == SDL_QUIT)
        isRunning = false;
    else if(ev.type == SDL_CONTROLLERBUTTONDOWN)
    {
        if(ev.cbutton.button == SDL_CONTROLLER_BUTTON_A)
            currentImage = image2;
        else if(ev.cbutton.button == SDL_CONTROLLER_BUTTON_B)
            currentImage = image3;
        }
}
```

2.5. Drawing optimized surfaces

Until now, you have been drawing raw surfaces. Raw surfaces are not optimised for blitting.

```
#include <SDL2/SDL.h>

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

The function `SDL_ConvertSurface()` copies the existing src surface into a new one that is optimised for bitting to a surface in a specified pixel format `fmt`. The `flags` parameter is unused and should be set to zero (this is a “leftover” from SDL 1.2’s API).

The following function, reads a bmp file from filePath and optimizes it for the window surface pixel format. This way, the returned surface corresponds to an optimised surface with the specified bmp file:

```
SDL_Surface *OptimizedSurface(std::string filePath, SDL_Surface *windowSurface)
{
    SDL_Surface *optimizedSurface = nullptr;
    SDL_Surface *surface = SDL_LoadBMP(filePath.c_str());
    if(surface == NULL)
        std::cout << "Error" << std::endl;
    else
    {
        optimizedSurface = SDL_ConvertSurface(surface, windowSurface->format, 0);
        if(optimizedSurface == NULL)
            std::cout << "Error" << std::endl;
    }

    SDL_FreeSurface(surface);

    return optimizedSurface;
}
```

This function can be used as follows:

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

    // Initializing and loading variables
    SDL_Window *window = nullptr;
    SDL_Surface *windowSurface = nullptr;
    SDL_Surface *currentImage = nullptr;

    SDL_Init(SDL_INIT_VIDEO);
```

```
    window = SDL_CreateWindow("SDL window", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, 640, 480, SDL_WINDOW_OPENGL);
    windowSurface = SDL_GetWindowSurface(window);
    currentImage = OptimizedSurface("galaxy2.bmp", windowSurface);

    SDL_Rect drawingRect;
    drawingRect.x = drawingRect.y = 0;
    drawingRect.w = 640;
    drawingRect.h = 480;

    bool isRunning = true;
    SDL_Event ev;

    while(isRunning)
    {
        while(SDL_PollEvent(&ev) != 0)
        {
            // Getting the events
            if(ev.type == SDL_QUIT)
                isRunning = false;
        }

        SDL_BlitScaled(currentImage, NULL, windowSurface, &drawingRect);
        SDL_UpdateWindowSurface(window);
    }

    SDL_DestroyWindow(window);
    window = nullptr;
    SDL_FreeSurface(currentImage);
    currentImage = windowSurface = nullptr;

    SDL_Quit();

    return 0;
}
```

The `SDL_BlitScaled()` function performs a scaled surface copy to a destination surface:

```
int SDL_BlitScaled(SDL_Surface* src, const SDL_Rect* srcrect, SDL_Surface* dst,
SDL_Rect* dstrect);
```

The parameter `src` corresponds to the structure to be copied from, `srcrect` is de `SDL_Rect` structure representing the rectangle area to be copied (or `NULL` to copy the entire surface), `dst` corresponds to the blit target and `dstrect` to the `SDL_Rect` structure representing the rectangle that is copied into, or `NULL` to copy into the entire surface.

2.6. SDL Texture

SDL Textures resemble SDL surfaces. However, they have slight differences such as being more optimized than surfaces. An SDL:Texture is a structure that contains an efficient, driver-specific representation of pixel data. For working with textures, you need to create a 2D rendering context for a window.

```
#include <SDL2/SDL.h>

SDL_Renderer* SDL_CreateRenderer(SDL_Window* window, int index, Uint32 flags);
SDL_Texture* SDL_CreateTextureFromSurface(SDL_Renderer* renderer, SDL_Surface*
      surface):
```

The function `SDL_CreateRenderer()` creates a 2D rendering context for a `window`. The `window` parameter is the window where rendering is displayed, `index` is the index of the

rendering driver to initialize, or -1, to initialize the first one supporting the requested flags and `flags` is `0` or one or more `SDL_RenderFlags` OR'd together:

Flag	Description
SDL_RENDERER_SOFTWARE	the renderer is a software fallback
SDL_RENDERER_ACCELERATED	the renderer uses hardware acceleration
SDL_RENDERER_PRESENTVSYNC	present is synchronized with the refresh rate
SDL_RENDERER_TARGETTEXTURE	the renderer supports rendering to texture

Providing no flags gives priority to available SDL_RENDERER_ACCELERATED renders.

`SDL_CreateTextureFromSurface()` creates a texture from an existing surface. `Render` parameter corresponds to the rendering context and `surface` is the `SDL_Surface` structure containing the pixel data used to fill the texture.

Similarly with the function for reading bmp image files into optimized surfaces, we can use the following function to load textures from bmp files:

```
SDL_Texture *LoadTexture(std::string filePath, SDL_Renderer *renderTarget)
{
    SDL_Texture *texture = nullptr;
    SDL_Surface *surface = SDL_LoadBMP(filePath.c_str());
    if(surface == NULL)
        std::cout << "Error" << std::endl;
    else
    {
        texture = SDL_CreateTextureFromSurface(renderTarget, surface);
        if(texture == NULL)
            std::cout << "Error" << std::endl;
    }

    SDL_FreeSurface(surface);

    return texture;
}
```

This function can be used as follows:

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

    // Initializing and loading variables
    SDL_Window *window = nullptr;
    SDL_Texture *currentImage = nullptr;
    SDL_Renderer *renderTarget = nullptr;

    SDL_Init(SDL_INIT_VIDEO);
    window = SDL_CreateWindow("SDL window", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, 640, 480, SDL_WINDOW_OPENGL);
    renderTarget = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);
    currentImage = LoadTexture("galaxy2.bmp", renderTarget);

    bool isRunning = true;
    SDL_Event ev;

    while(isRunning)
    {
        while(SDL_PollEvent(&ev) != 0)
        {
            // Getting the events
```

```
            if(ev.type == SDL_QUIT)
                isRunning = false;
        }

        SDL_RenderClear(renderTarget);
        SDL_RenderCopy(renderTarget, currentImage, NULL, NULL);
        SDL_RenderPresent(renderTarget);
    }

    SDL_DestroyWindow(window);
    SDL_DestroyTexture(currentImage);
    SDL_DestroyRenderer(renderTarget);
    window = nullptr;
    currentImage = nullptr;
    renderTarget = nullptr;

    SDL_Quit();

    return 0;
}
```

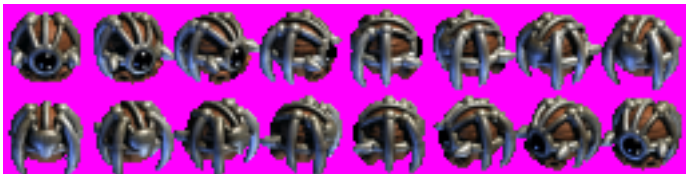
Note the following lines of code, inside the **game loop**:

```
SDL_RenderClear(renderTarget);
SDL_RenderCopy(renderTarget, currentImage, NULL, NULL);
SDL_RenderPresent(renderTarget);
```

The function `SDL_RenderClear()` clears the current rendering target, using the `renderTarget` draw color; `SDL_RenderCopy()` copies a portion of the texture to the current rendering target; and `SDL_RenderPresent()` updates the screen with any rendering performed since the previous call.

2.7. Sprite Animation

Lets consider the following image as our sprite sheet (file **drone.bmp**):



This image has 256 pixels width and 64 pixels height. Furthermore, it consists in a grid of 8 x 2 **frames**. This means that each **frame** as 32 pixel width and 32 pixel height. This way, we can do the following:

```
currentImage = LoadTexture("drone.bmp", renderTarget);
SDL_QueryTexture(currentImage, NULL, NULL, &textureWidth, &textureHeight);

frameWidth = textureWidth / 8;
frameHeight = textureHeight / 2;

playerRect.x = playerRect.y = 0;
playerRect.w = frameWidth;
playerRect.h = frameHeight;
```

The `SDL_QueryTexture()` function queries the texture containing the drone.bmp file and fills `textureWidth` and `textureHeight` with the corresponding width and height of the drone sprite sheet texture. This way, by dividing the `textureWidth` by 8 and the `textureHeight` by 2, we know the size of each frame (recall that the drone.bmp file contains a grid of 8x2 frames). By using

the following **game loop**, we have the drone sprite animation:

```
while(isRunning)
{
    while(SDL_PollEvent(&ev) != 0)
    {
        // Getting the events
        if(ev.type == SDL_QUIT)
            isRunning = false;
    }

    frameTime++;

    if(frameTime >= 15)
    {
        frameTime = 0;
        playerRect.x += frameWidth;
        if(playerRect.x >= textureWidth) {
            playerRect.x = 0;
            playerRect.y += frameHeight;
            if(playerRect.y >= textureHeight) {
                playerRect.y = 0;
            }
        }
    }

    SDL_RenderClear(renderTarget);
    SDL_RenderCopy(renderTarget, currentImage, &playerRect, &playerPosition);
    SDL_RenderPresent(renderTarget);
}
```

Here is the entire code of the `main()` function:

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

    SDL_Window *window = nullptr;
    SDL_Texture *currentImage = nullptr;
    SDL_Renderer *renderTarget = nullptr;
    SDL_Rect playerRect;
    SDL_Rect playerPosition;
    playerPosition.x = playerPosition.y = 0;
    playerPosition.w = playerPosition.h = 32;
    int frameWidth, frameHeight;
    int textureWidth, textureHeight;
    int frameTime = 0;

    SDL_Init(SDL_INIT_VIDEO);

    window = SDL_CreateWindow("SDL window", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, 640, 480, SDL_WINDOW_OPENGL);

    renderTarget = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED |
SDL_RENDERER_PRESENTVSYNC);

    currentImage = LoadTexture("/Users/gustavo/Downloads/Xenon 2000 assets/graphics/
24bit/drone.bmp", renderTarget);

    SDL_QueryTexture(currentImage, NULL, NULL, &textureWidth, &textureHeight);

    frameWidth = textureWidth / 8;
    frameHeight = textureHeight / 2;

    playerRect.x = playerRect.y = 0;
    playerRect.w = frameWidth;
    playerRect.h = frameHeight;

    SDL_SetRenderDrawColor(renderTarget, 0xFF, 0, 0, 0xFF);

    bool isRunning = true;
    SDL_Event ev;
```

```
    while(isRunning)
    {
        while(SDL_PollEvent(&ev) != 0)
        {
            // Getting the events
            if(ev.type == SDL_QUIT)
                isRunning = false;
        }

        frameTime++;

        if(frameTime >= 15)
        {
            frameTime = 0;
            playerRect.x += frameWidth;
            if(playerRect.x >= textureWidth) {
                playerRect.x = 0;
                playerRect.y += frameHeight;
                if(playerRect.y >= textureHeight) {
                    playerRect.y = 0;
                }
            }
        }

        SDL_RenderClear(renderTarget);
        SDL_RenderCopy(renderTarget, currentImage, &playerRect, &playerPosition);
        SDL_RenderPresent(renderTarget);
    }

    SDL_DestroyWindow(window);
    SDL_DestroyTexture(currentImage);
    SDL_DestroyRenderer(renderTarget);
    window = nullptr;
    currentImage = nullptr;
    renderTarget = nullptr;

    SDL_Quit();

    return 0;
}
```

Note that the render target was created using an additional flag:

```
renderTarget = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED |
SDL_RENDERER_PRESENTVSYNC);
```

By using the SDL_RENDERER_PRESENTVSYNC flag, we are “locking” our game loop to be synchronised with our refresh rate. For instance, if our refresh rate is 60 frames per second (60 fps), and since we are updating the animation at intervals of 15 frames, this means that our animation is updated every quarter of second, i.e.: 250 milliseconds. However, in computers with older hardware, thus lower refresh rates, this animation will not be smooth, because in this case, the animation speed is directly related to the FSP. To tackle the problem of different refresh rates, so that the animation runs equally on different computers with different hardware, we should use **timers**.

2.8. Working with timers

To deal with timers, lets make the following changes to the previous code: first, change the type of the `frameTime` variable from `int` to `float`; then, add the following variables:

```
float frameTime = 0;
```

```
int prevTime = 0;
int currentTime = 0;
float deltaTime = 0;
```

Now, change the **game loop** to the following:

```
while(isRunning)
{
    prevTime = currentTime;
    currentTime = SDL_GetTicks();
    deltaTime = (currentTime - prevTime) / 1000.0f;

    while(SDL_PollEvent(&ev) != 0)
    {
        // Getting the events
        if(ev.type == SDL_QUIT)
            isRunning = false;
    }

    frameTime += deltaTime;

    if(frameTime >= 0.25f)
    {
        frameTime = 0;
        playerRect.x += frameWidth;
        if(playerRect.x >= textureWidth) {
            playerRect.x = 0;
            playerRect.y += frameHeight;
            if(playerRect.y >= textureHeight) {
                playerRect.y = 0;
            }
        }
    }

    SDL_RenderClear(renderTarget);
    SDL_RenderCopy(renderTarget, currentImage, &playerRect, &playerPosition);
    SDL_RenderPresent(renderTarget);
}
```

The animation is still being updated at every 250 milliseconds. However, this is not dependent of the refresh rate. In older computers, the animation will still be updated at every 250 milliseconds.

2.9. Color keying

As you noticed, bmp files don't have transparency. This means that one specific color must be set to be considered as transparent during blitting. This is done using the `SDL_SetColorKey()` function:

```
#include <SDL2/SDL.h>

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

The `surface` parameter corresponds to the surface structure to update, the `flag` must be set to `SDL_TRUE` to enable color key and `SDL_FALSE` to disable color key and the `key` parameter corresponds to the transparent pixel. By taking into account that the color in the drone.bmp that should be mapped to transparent is the RGB color (255, 0, 255): we will add the following line of code to the `LoadTexture()` function, just before the `SDL_CreateTexture()`:

```
SDL_SetColorKey(surface, SDL_TRUE, SDL_MapRGB(surface->format, 255, 0, 255));
texture = SDL_CreateTextureFromSurface(renderTarget, surface);
```

3. Final Code

```
#include<SDL2/SDL.h>
#include<iostream>


SDL_Texture *LoadTexture(std::string filePath, SDL_Renderer *renderTarget)
{
    SDL_Texture *texture = nullptr;
    SDL_Surface *surface = SDL_LoadBMP(filePath.c_str());
    if(surface == NULL)
        std::cout << "Error" << std::endl;
    else
    {
        SDL_SetColorKey(surface, SDL_TRUE, SDL_MapRGB(surface->format, 255, 0, 255));
        texture = SDL_CreateTextureFromSurface(renderTarget, surface);
        if(texture == NULL)
            std::cout << "Error" << std::endl;
    }

    SDL_FreeSurface(surface);

    return texture;
}


SDL_Surface *OptimizedSurface(std::string filePath, SDL_Surface *windowSurface)
{
    SDL_Surface *optimizedSurface = nullptr;
    SDL_Surface *surface = SDL_LoadBMP(filePath.c_str());
    if(surface == NULL)
        std::cout << "Error" << std::endl;
    else
    {
        optimizedSurface = SDL_ConvertSurface(surface, windowSurface->format, 0);
        if(optimizedSurface == NULL)
            std::cout << "Error" << std::endl;
    }

    SDL_FreeSurface(surface);

    return optimizedSurface;
}

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

    SDL_Window *window = nullptr;
    SDL_Texture *currentImage = nullptr;
    SDL_Texture *background = nullptr;
    SDL_Renderer *renderTarget = nullptr;
    SDL_Rect playerRect;
    SDL_Rect playerPosition;
    playerPosition.x = playerPosition.y = 0;
    playerPosition.w = playerPosition.h = 32;
    int frameWidth, frameHeight;
    int textureWidth, textureHeight;
    float frameTime = 0;
    int prevTime = 0;
    int currentTime = 0;
    float deltaTime = 0;

    float moveSpeed = 200.0f;
    const Uint8 *keyState;

    SDL_Init(SDL_INIT_VIDEO);

    window = SDL_CreateWindow("SDL window", SDL_WINDOWPOS_CENTERED,
SDL_WINDOWPOS_CENTERED, 640, 480, SDL_WINDOW_OPENGL);
    renderTarget = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED |
SDL_RENDERER_PRESENTVSYNC);
    currentImage = LoadTexture("drone.bmp", renderTarget);
```

```
    background = LoadTexture("galaxy2.bmp", renderTarget);

    SDL_QueryTexture(currentImage, NULL, NULL, &textureWidth, &textureHeight);

    frameWidth = textureWidth / 8;
    frameHeight = textureHeight / 2;

    playerRect.x = playerRect.y = 0;
    playerRect.w = frameWidth;
    playerRect.h = frameHeight;

    SDL_SetRenderDrawColor(renderTarget, 0xFF, 0, 0, 0xFF);

    bool isRunning = true;
    SDL_Event ev;

    while(isRunning)
    {
        prevTime = currentTime;
        currentTime = SDL_GetTicks();
        deltaTime = (currentTime - prevTime) / 1000.0f;

        while(SDL_PollEvent(&ev) != 0)
        {
            // Getting the events
            if(ev.type == SDL_QUIT)
                isRunning = false;
        }

        keyState = SDL_GetKeyboardState(NULL);
        if(keyState[SDL_SCANCODE_RIGHT])
            playerPosition.x += moveSpeed * deltaTime;
        else if(keyState[SDL_SCANCODE_LEFT])
            playerPosition.x -= moveSpeed * deltaTime;


        frameTime += deltaTime;

        if(frameTime >= 0.1f)
        {
            frameTime = 0;
            playerRect.x += frameWidth;
            if(playerRect.x >= textureWidth) {
                playerRect.x = 0;
                playerRect.y += frameHeight;
                if(playerRect.y >= textureHeight) {
                    playerRect.y = 0;
                }
            }
        }

        SDL_RenderClear(renderTarget);
        SDL_RenderCopy(renderTarget, background, NULL, NULL);
        SDL_RenderCopy(renderTarget, currentImage, &playerRect, &playerPosition);
        SDL_RenderPresent(renderTarget);
    }

    SDL_DestroyWindow(window);
    SDL_DestroyTexture(currentImage);
    SDL_DestroyTexture(background);
    SDL_DestroyRenderer(renderTarget);
    window = nullptr;
    currentImage = nullptr;
    renderTarget = nullptr;

    SDL_Quit();

    return 0;
}
```