Modern computer graphics introduction, with SDL3

Introduction

This lesson will introduce you to the basics of modern computer graphics. Classic modern GPU API (Vulkan, Molken and DirectX12) use a lot of concepts and requires thousands lines of code even to draw a simple triangle. This tend to create confusion in the mind of students and make further learning difficult.

SDL3 innovated with a SDL_gpu.h API, which aim to simplify and abstract the use of modern GPU API. In function of the OS you're working with, it will call Vulkan, DirectX or Molken (Vulkan to Metal). SDL_gpu.h will retain the general architecture of modern GPU APIs and expose usual GPU operations, while hiding the most complicated aspects.

This course is thus a stepping stone to further learning of modern GPU APIs. It will train your brain to reason with modern GPU concepts, which will allow you not to be lost when you will code with Vulkan, DirectX or Metal,

A basic framework

In order experiment with SDL3, you'll need a basic engine structure. Here is a suggestion of what you could use.

CMake and dependencies

We will use CMake to download and install all dependencies.

```
CMakeLists.txt

cmake_minimum_required(VERSION 3.20)
project(graphics-SDL3)
set(CMAKE_CXX_STANDARD 20)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

cmake_policy(SET CMP0135 NEW)

# SDL3
set(SDL3_VERSION 3.1.6)
find_package(SDL3 ${SDL3_VERSION} QUIET) # QUIET or REQUIRED
if (NOT SDL3_FOUND) # If there's none, fetch and build SDL3
    include(FetchContent)
    if (UNIX)
        FetchContent_Declare(
                SDL3
                URL https://github.com/libsdl-org/SDL/archive/refs/tags/preview-
${SDL3_VERSION}.tar.gz
        )
        FetchContent_MakeAvailable(SDL3)
    elseif (WIN32)
        FetchContent_Declare(
                SDL3
                URL https://github.com/libsdl-org/SDL/archive/refs/tags/preview-
${SDL3_VERSION}.zip
        )
        FetchContent_MakeAvailable(SDL3)
    endif()
endif()

file( GLOB graphics-with-SDL3_SOURCES *.cpp )
add_executable(${PROJECT_NAME} ${graphics-with-SDL3_SOURCES})

target_include_directories(${PROJECT_NAME} PUBLIC ${SDL3_INCLUDE_DIRS})
target_link_libraries(${PROJECT_NAME} SDL3::SDL3)
```

This code basically download and install the SDL3 version you’ll specify.

Content and compiled shaders

In order to use shaders whichever the OS you are using, you will need compiled shaders for each OS. As a matter of fact, modern GPU APIs compile shaders before runtime in order to use them.

We will use already compiled shaders. Download the provided Content.zip file and unzip it in the root of your project. You also need to copy ti in your Cmake build directory.

Basic classes

Window

This class will manage the window.

`Window.hpp`

```
#ifndef WINDOW_HPP
#define WINDOW_HPP
#include <SDL3/SDL_video.h>

class Window {
public:
    SDL_Window* sdlWindow;
    void Init();
    void Close() const;

    int width { 640 };
    int height { 480 };
};

#endif //WINDOW_HPP
```

`Window.cpp`

```
#include "Window.hpp"

#include <string>
#include <SDL3/SDL.h>
using std::string;

void Window::Init() {
    SDL_Init(SDL_INIT_VIDEO);
    const string title { "Computer Graphics with SDL3" };
    sdlWindow = SDL_CreateWindow(title.c_str(), width, height, SDL_WINDOW_RESIZABLE);
}

void Window::Close() const {
    SDL_DestroyWindow(sdlWindow);
    SDL_Quit();
}
```

Time

This class will manage delta time.

Time.hpp

```
#ifndef TIME_HPP
#define TIME_HPP

/*
 * Hold time related functions.
 * In charge of computing the delta time and ensure smooth game ticking.
 */
class Time {
public:
    // Compute delta time as the number of milliseconds since last frame
    float ComputeDeltaTime();

    // Wait if the game run faster than the decided FPS
    void DelayTime();

private:
    const static int FPS = 60;
    const static int frameDelay = 1000 / FPS;

    // Time in milliseconds when frame starts
    unsigned int frameStart { 0 };

    // Last frame start time in milliseconds
    unsigned int lastFrame { 0 };

    // Time it tooks to run the loop. Used to cap framerate.
    unsigned int frameTime { 0 };
};
```

Time.cpp

```
#include "Time.hpp"
#include <SDL3/SDL.h>

float Time::ComputeDeltaTime() {
    frameStart = SDL_GetTicks();
    unsigned int dt = frameStart - lastFrame;
    lastFrame = frameStart;
    return static_cast<float>(dt) / 1000.0f;
}

void Time::DelayTime() {
    frameTime = SDL_GetTicks() - frameStart;
    if (frameTime < frameDelay) {
        SDL_Delay(frameDelay - frameTime);
    }
}
```

Input state

This class will allow simple input management.

InputState.hpp

```
#ifndef INPUTSTATE_HPP
#define INPUTSTATE_HPP
```

```
enum class DirectionalKey {
    Up,
    Down,
    Left,
    Right
};

struct InputState {
    bool IsUp(DirectionalKey key) {
        switch (key) {
            case DirectionalKey::Up:
                return !up;
            case DirectionalKey::Down:
                return !down;
            case DirectionalKey::Left:
                return !left;
            case DirectionalKey::Right:
                return !right;
        }
    }

    bool IsDown(DirectionalKey key) {
        switch (key) {
            case DirectionalKey::Up:
                return up;
            case DirectionalKey::Down:
                return down;
            case DirectionalKey::Left:
                return left;
            case DirectionalKey::Right:
                return right;
        }
    }

    bool IsPressed(DirectionalKey key) {
        switch (key) {
            case DirectionalKey::Up:
                return up && !previousUp;
            case DirectionalKey::Down:
                return down && !previousDown;
            case DirectionalKey::Left:
                return left && !previousLeft;
            case DirectionalKey::Right:
                return right && !previousRight;
        }
    }

    bool IsReleased(DirectionalKey key) {
        switch (key) {
            case DirectionalKey::Up:
                return !up && previousUp;
            case DirectionalKey::Down:
                return !down && previousDown;
            case DirectionalKey::Left:
                return !left && previousLeft;
            case DirectionalKey::Right:
                return !right && previousRight;
        }
    }

    bool previousLeft { false };
    bool left { false };

    bool previousRight { false };
    bool right { false };

    bool previousUp { false };
    bool up { false };

    bool previousDown { false };
    bool down { false };
};

#endif //INPUTSTATE_HPP
```

Scene

Finally, this class will be a mother class for all our example scenes.

```
Scene.hpp

#ifndef SCENE_HPP
#define SCENE_HPP

#include <SDL3/SDL_events.h>
#include "InputState.hpp"

class Renderer;

class Scene {
public:
    virtual ~Scene() {}

    virtual void Load(Renderer& renderer) = 0;

    virtual bool Update(float dt) = 0;

    virtual void Draw(Renderer& renderer) = 0;

    virtual void Unload(Renderer& renderer) = 0;

protected:
    static bool ManageInput(InputState &inputState) {
        inputState.previousLeft = inputState.left;
        inputState.previousRight = inputState.right;
        inputState.previousUp = inputState.up;
        inputState.previousDown = inputState.down;

        SDL_Event event;
        while (SDL_PollEvent(&event))
        {
            if (event.type == SDL_EVENT_QUIT) { return false; }
            else if (event.type == SDL_EVENT_KEY_DOWN)
            {
                if (event.key.key == SDLK_ESCAPE) { return false; }
                if (event.key.key == SDLK_LEFT) { inputState.left = true; }
                if (event.key.key == SDLK_RIGHT) { inputState.right = true; }
                if (event.key.key == SDLK_UP) { inputState.up = true; }
                if (event.key.key == SDLK_DOWN) { inputState.down = true; }
            }
            else if (event.type == SDL_EVENT_KEY_UP)
            {
                if (event.key.key == SDLK_LEFT) { inputState.left = false; }
                if (event.key.key == SDLK_RIGHT) { inputState.right = false; }
                if (event.key.key == SDLK_UP) { inputState.up = false; }
                if (event.key.key == SDLK_DOWN) { inputState.down = false; }
            }
        }
        return true;
    }
};

#endif //SCENE_HPP
```

Main and Renderer

The main file will run the scene we are using. You will notice the renderer in it. Most graphics operations will happen in the Renderer class. Throughout the lesson, we will explain the Renderer code as it is used in the scene implementation class.

```
Main.cpp

#include <iostream>
#include <SDL3/SDL_main.h>

#include "Renderer.hpp"
#include "Scene01Clear.hpp"
#include "Scene02Triangle.hpp"
#include "Scene03TriangleVertexBuffer.hpp"
#include "Scene04TriangleCullModes.hpp"
#include "Scene05TriangleStencil.hpp"
#include "Scene06TriangleInstances.hpp"
#include "Scene07TextureQuad.hpp"
#include "Scene08TextureQuadMoving.hpp"
#include "Time.hpp"
#include "Window.hpp"

using namespace std;

int main(int argc, char **argv) {
    Window window {};
    Renderer renderer {};
    Time time {};
    window.Init();
    renderer.Init(window);

    auto scene = std::make_unique<Scene01Clear>();
    scene->Load(renderer);

    bool isRunning { true };
    while (isRunning) {
        const float dt = time.ComputeDeltaTime();

        isRunning = scene->Update(dt);
        scene->Draw(renderer);

        time.DelayTime();
    }

    scene->Unload(renderer);

    renderer.Close();
    window.Close();
    return 0;
}
```

First scene: clearing the screen

In this first scene, we will clear the screen with a color. This is the most basic operation you can do with a GPU.

Scene01Clear.hpp

```
#ifndef SCENE01CLEAR_HPP
#define SCENE01CLEAR_HPP

#include "Scene.hpp"

class Scene01Clear : public Scene {
public:
    void Load(Renderer& renderer) override;
    bool Update(float dt) override;
    void Draw(Renderer& renderer) override;
    void Unload(Renderer& renderer) override;

private:
    InputState inputState;
};


#endif //SCENE01CLEAR_HPP
```

Scene01Clear.cpp

```
#include "Scene01Clear.hpp"

#include "Renderer.hpp"
#include <SDL3/SDL_events.h>

void Scene01Clear::Load(Renderer& renderer) {

}

bool Scene01Clear::Update(float dt) {
    return ManageInput(inputState);
}

void Scene01Clear::Draw(Renderer& renderer) {
    renderer.Begin();

    renderer.End();
}

void Scene01Clear::Unload(Renderer& renderer) {

}
```

We now need to implement Init, Close, Begin and End the the Renderer.

Renderer.hpp

```
#ifndef RENDERER_HPP
#define RENDERER_HPP

#include <SDL3/SDL_gpu.h>
#include <vector>
#include <string>

using std::vector;
using std::string;

class Window;
class Renderer {
public:
    void Init(Window &window);
    void Begin(SDL_GPUDepthStencilTargetInfo* depthStencilTargetInfo = nullptr);
    void End() const;
    void Close() const;

    SDL_GPUDevice* device {nullptr};
    SDL_Window* renderWindow {nullptr};

    SDL_GPUCommandBuffer* cmdBuffer {nullptr};
    SDL_GPUTexture* swapchainTexture {nullptr};
    SDL_GPURenderPass* renderPass {nullptr};
};

#endif //RENDERER_HPP
```

We see that the renderer holds a pointer to the SDL window, which is usual, but also to a device. In modern GPU APIs, the device is the handler to the used GPU. We will need to initialize this device.

There are three other member variables. - The command buffer gathers all the commands we want to send to the GPU. Commands are not sent in a immediate way, as in OpenGL, but are gathered in a command buffer and sent to the GPU in one go. This optimizes the communication between the CPU and the GPU. - The swapchain texture is the texture that will be displayed on the screen. The swapchain describes the buffer that will be displayed on the screen, and how is will be displayed (double buffering, triple buffering, drawing only when the texture is ready, or as soon as possible etc.). SDL_gpu.h abstracts this concept and provides a texture that will be displayed on the screen. - The render pass is a concept that describes the operations that will be done on the swapchain texture. It is possible to get several sub-passes on a single renderpass.

```
Renderer.cpp

void Renderer::Init(Window &window) {
    renderWindow = window.sdlWindow;
    device = SDL_CreateGPUDevice(
        SDL_GPU_SHADERFORMAT_SPIRV | SDL_GPU_SHADERFORMAT_DXIL | SDL_GPU_SHADERFORMAT_MSL,
        true,
        nullptr);
    SDL_ClaimWindowForGPUDevice(device, renderWindow);
}

void Renderer::Close() const {
    SDL_ReleaseWindowFromGPUDevice(device, renderWindow);
    SDL_DestroyGPUDevice(device);
}
```

Ths initialization consist in creating a device with the SDL_CreateGPUDevice function. We also claim the window for the device. The Close function release the window from the device and destroy the device.

We can now implement the Begin and End functions.

```
Renderer.cpp

void Renderer::Begin(SDL_GPUDepthStencilTargetInfo* depthStencilTargetInfo) {
    cmdBuffer = SDL_AcquireGPUCommandBuffer(device);
    if (cmdBuffer == nullptr) {
        SDL_Log("AcquireGPUCommandBuffer failed: %s", SDL_GetError());
    }

    SDL_GPUTexture* swapchainTexture;
    if (!SDL_AcquireGPUSwapchainTexture(cmdBuffer, renderWindow, &swapchainTexture,
nullptr, nullptr)) {
        SDL_Log("AcquireGPUSwapchainTexture failed: %s", SDL_GetError());
    }

    if (swapchainTexture != nullptr) {
        SDL_GPUColorTargetInfo colorTargetInfo = {};
        colorTargetInfo.texture = swapchainTexture;
        colorTargetInfo.clear_color = SDL_FColor { 0.392f, 0.584f, 0.929f, 1.0f };
        colorTargetInfo.load_op = SDL_GPU_LOADOP_CLEAR;
        colorTargetInfo.store_op = SDL_GPU_STOREOP_STORE;

        renderPass = SDL_BeginGPURenderPass(cmdBuffer, &colorTargetInfo, 1,
depthStencilTargetInfo);
    }
}

void Renderer::End() const {
    SDL_EndGPURenderPass(renderPass);
    SDL_SubmitGPUCommandBuffer(cmdBuffer);
}
```

Let’s first talk about Begin. Let’s ignore its argument, that will be null except if we are interested in stencil and depth.

We acquire a command buffer with SDL_AcquireGPUCommandBuffer. We then acquire a swapchain texture with SDL_AcquireGPUSwapchainTexture. This function will return a texture that will be displayed on the screen. We then create a color target info structure that will describe the operations that will be done on the swapchain texture. We then begin a render pass with SDL_BeginGPURenderPass.

The SDL_GPUColorTargetInfo represents the color buffer of the swapchain texture. We set the clear color to a light blue, and the load and store operations to clear and store. The load operation is the operation that will be done on the texture before the rendering, and the store operation is the operation that will be done on the texture after the rendering. In this case, we clear the texture before rendering and store it after rendering.

This way of setting up info structures before using them in a GPU operation, through a function call, is usual with Vulkan. You will get used to it with SDL too.

The End function is simple. We end the render pass and submit the command buffer. This will result in presenting the cleared color buffer on the render texture.

In our future scenes, we will add drawing instructions between Begin and End.

We can now run the program. You should see a light blue screen. If you don’t, check the error messages in the console.

Exercises

1. Change the color of the clear screen.