SDL_gpu: it begins with a triangle.

29 min read · May 15, 2025

Hamdy Elzanqali Follow

You may think of GPUs as *fast* black magic, their APIs as spells, and graphics programmers as wizards, and in part you are correct, because a technology too advanced is nothing short of magic. So, in lack of better a better introduction, I will try my best to introduce you to this witchcraft.

What is SDL GPU all about?

To draw anything on the screen, you need to talk to the GPU, and to talk to the GPU, you need a graphics API. OpenGL is too old, and while it's still enough for many use cases today, it's slowly getting replaced by more **modern** APIs that are faster, has newer features, better designed, better supported or frankly the **only** supported, etc. On Linux and Android, you probably want to use **Vulkan**. On windows, you may want **Directx12**. On MacOS and iOS, you *need* to use **Metal**. What about consoles? This gets impractical rather quickly. While you can learn all of these APIs with all their differences and quirks, which is a huge undertaking, you would prefer to use a **wrapper** or an API that encapsulate all of these APIs into one place. Fortunately, all modern APIs work in a similar way and share a lot of their concepts and ideas which makes such abstractions possible. Here is where SDL_gpu shines, it's a "thin layer" above Vulkan / Metal / Directx12, and more to come in the future. The goal is to write-once-run-everywhere with minimal efforts. While they say it's only a thin layer, I find that a little misleading because they did a wonderful job abstracting away a lot of these annoyances and only left the parts you would really want to focus on.

A quick overview...

It may seem complicated at first. Afterall, it's not a simple `drawTriangle()`.

However, I promise once you understand why everything is the way it is, it will start to make sense. A million triangles take as much effort as drawing a single one, so please be patient. This guide also assumes you are at least familiar with C++, but it should be easily ported to C or your language of choice.

Note: This is a C++ tutorial, if you want to use C, here is a slightly modified version of the full code that works with C. You can refer back to that file when things break.

The flow will something like this:
1. Create a **Device,** which is requesting access to a compatible GPU.
2. Create **Buffers,** which are containers of data on the GPU.
3. Create a **Graphics Pipeline** which tells the GPU how to use these buffers.
4. Acquire a **Command Buffer** to start issuing tasks to the GPU.
5. Fill the buffers with data using **Transfer Buffer** in a **Copy Pass.**
6. Acquire the **Swapchain** texture, or simply the window to draw onto.
7. Finally, issue the **Draw Call** in a **Render Pass.**

Let's start with a blank window using the new callbacks system in SDL3.

```
#define SDL_MAIN_USE_CALLBACKS
#include <SDL3/SDL_main.h>
#include <SDL3/SDL.h>

SDL_Window* window;

SDL_AppResult SDL_AppInit(void **appstate, int argc, char **argv)
{
    // create a window
    window = SDL_CreateWindow("Hello, Triangle!", 960, 540, SDL_WINDOW_RESIZ

    return SDL_APP_CONTINUE;
}

SDL_AppResult SDL_AppIterate(void *appstate)
{
    return SDL_APP_CONTINUE;
}

SDL_AppResult SDL_AppEvent(void *appstate, SDL_Event *event)
{
    // close the window on request
    if (event->type == SDL_EVENT_WINDOW_CLOSE_REQUESTED)
```

```
    {
        return SDL_APP_SUCCESS;
    }

    return SDL_APP_CONTINUE;
}

void SDL_AppQuit(void *appstate, SDL_AppResult result)
{
    // destroy the window
    SDL_DestroyWindow(window);
}
```

The GPU device.

Before you start drawing anything, the first step is to get access to a GPU device or driver. Fortunately, SDL does a great job abstracting that into a simple a simple function:

```
// signature
SDL_GPUDevice* SDL_CreateGPUDevice(
  SDL_GPUShaderFormat format_flags,
  bool debug_mode,
  const char *name
);

// example:
// create a device for either VULKAN or METAL with debugging enabled and cho
SDL_GPUDevice* device = SDL_CreateGPUDevice(SDL_GPU_SHADERFORMAT_SPIRV | SDL
```

- format_flags: it specifies the shader formats that you plan to provide later when creating your graphics pipeline. For example, if you want to use the Vulkan backend, you have to supply shaders in Vulkan's `.spv` format. Also, `.dxil` for Directx12, `.msl` for Metal, etc. This means you will have to write separate shaders for each backend. SDL_gpu does not provide a unified shader format for all backends (yet?). However, there is an official approach to support every backend with either only `.spv` or `.hlsl` using SDL_shadercross which can convert your shaders on the fly or as a build time step. We will only use the Vulkan backend with the SPIRV `.spv` shader format for now. We will set it to `SDL_GPU_SHADERFORMAT_SPIRV`. Later, we will integrate SDL_shadercross and reuse our SPIRV shader for all platforms.

- debug_mode: enables debugging layer for the given backend. You will have to install the Vulkan SDK to get Vulkan error messages or the `Vulkan Validation Layers`, also you can install other validations layers for other backends like Directx12, and so on. We will keep it off for now.
- name: this refers to the name of the driver you want to use, e.g. `vulkan`, `metal`, `direct3d12`, or `NULL` to let SDL handle that automatically for us. We will set it to `NULL`.

If no device could be created for given parameters, the function will return `NULL`. That can happen on platforms/devices which are too old or does not support the backend you choose. Once you have your device ready, you need to tell SDL that you want to use this `window` for this `device`. Which is as simple as:

```
SDL_ClaimWindowForGPUDevice(device, window);
```

Your code should look something like this:

```
...
SDL_Window* window;
SDL_GPUDevice* device;

SDL_AppResult SDL_AppInit(void **appstate, int argc, char **argv)
{
    // create a window
    window = SDL_CreateWindow("Hello, Triangle!", 960, 540, SDL_WINDOW_RESIZ

    // create the device
    device = SDL_CreateGPUDevice(SDL_GPU_SHADERFORMAT_SPIRV, false, NULL);
    SDL_ClaimWindowForGPUDevice(device, window);

    return SDL_APP_CONTINUE;
}
...
```

You should `destroy` your GPU device after you are done with the application.

```
...
void SDL_AppQuit(void *appstate, SDL_AppResult result)
{
    // destroy the GPU device
    SDL_DestroyGPUDevice(device);

    // destroy the window
    SDL_DestroyWindow(window);
}
...
```

Testing the waters.

Currently, the window is a black rectangle which isn't interesting enough. To make sure everything is working as expected, when will skip a few steps ahead and clear the screen with a color. This means you should first understand what a **Command Buffer**, a **Swapchain Texture**, **Color Targets**, and **Render Passes** mean. I promise, it's not as complicated as it sounds.

What are command buffers?

A buffer is simply a container of data. It can be an array of positions, colors, etc. A command buffer is simply a "buffer" or an array of "commands". It's a list of tasks that you expect the GPU to execute in order. For instance, instead of asking the GPU to do something, wait for the request to arrive, which is a slow operation, wait for it to execute and finish, then return back to your app, you can instead pack all these commands into a single buffer that is sent once to the GPU by the end of your frame. Command buffers also allow you to do multithreading, so you may split different tasks like updating buffers data on a thread and drawing geometry on another. Don't worry if you don't understand any of that, we only need a command buffer to send instructions to the GPU.

To `acquire` a command buffer, all you need is to simply call:

```
SDL_GPUCommandBuffer* commandBuffer = SDL_AcquireGPUCommandBuffer(device);
```

For each command buffer you create, you **must** `submit` it when you are done with

it. Otherwise, you can end with memory leaks. If you skip the frame earlier, don't forget to submit the command buffer. Once it's submitted, the GPU will start executing your instructions, if any. A typical frame usually looks like this:

```
...
// I want to do something on the GPU.
SDL_GPUCommandBuffer* commandBuffer = SDL_AcquireGPUCommandBuffer(device);

// update buffers here
// draw here
// other stuff

// Do it now.
SDL_SubmitGPUCommandBuffer(commandBuffer);
...
```

Color Targets.

When you want to draw anything, you must first tell the GPU *where* to draw it, what to do with previous target's content, and how to deal with the new data. Here we want to `clear` the `window` with a `color` then `store` the resulting pixels. Another example may be something like updating an *offscreen* `texture` without clearing its previous content. How would any of that translate to SDL_gpu? The answer is `SDL_GPUColorTargetInfo` . This struct lets you *describe* how you want the GPU to start a `Render Pass` . Looking into the SDL Wiki, which is a great place to learn SDL, you will find that this struct has many properties, but for the sake of simplicity, we will only focus on what we need now:

```
SDL_GPUColorTargetInfo colorTargetInfo{};

// discard previous content and clear to a color
colorTargetInfo.clear_color = {255/255.0f, 219/255.0f, 187/255.0f, 255/255.0
colorTargetInfo.load_op = SDL_GPU_LOADOP_CLEAR; // or SDL_GPU_LOADOP_LOAD to
                                                // keep the previous content

// store the content to the texture
colorTargetInfo.store_op = SDL_GPU_STOREOP_STORE;

// where are we going to store the result?
colorTargetInfo.texture = texture; // we will set this later to the
                                   // window's swapchain texture
```

The window's Swapchain Texture.

A texture, much like a buffer, is just a container that contains the image pixel data in a certain format. A **Swapchain** is simply a collection of textures that are "swapped" on the window surface one after the other. So, you can be updating a frame while the previous frame is being displayed on the screen then once the new frame is ready, you can smoothly swap the old one with the one you have just finished rendering. In SDL_gpu, you can easily acquire the next swapchain texture by:

```
SDL_GPUTexture* swapchainTexture;
Uint32 width, height;
SDL_WaitAndAcquireGPUSwapchainTexture(commandBuffer, window, &texture, &widt
```

This will fill the `swapchainTexture` with the swapchain texture and update the `width` and `height` variables. The width and height variables filled from the swapchain are more *reliable* than just using the window size. This may come in handy later.

Note that I used `SDL_WaitAndAcquireGPUSwapchainTexture()` instead of `SDL_AcquireGPUSwapchainTexture()`, because the latter, if not used correctly, may result in memory leaks on some backends. This essentially blocks the thread until a swapchain texture is available.

One last thing, the swapchain texture returned can still be `NULL` in certain situations like the window being minimized. So, you must always check before attempting to use it.

Render Passes.

A command buffer consists of **passes**, these passes can be a **Copy Pass** where you upload data to GPU like updating a buffer or uploading a texture, a **Render Pass** where you actually draw something onto a color target, or a **Compute Pass** where you can pass heavy calculation to be done on the GPU using compute shaders. To use a pass, you first `begin` the pass, do your thing, then `end` it later. You can't be inside of two passes at the same time, so you if you begin a copy pass, you must first end it before you can begin a render pass.

A render pass is where all your draw commands go, and it requires a color target to use. To begin a render pass:

```
// signature
SDL_GPURenderPass* SDL_BeginGPURenderPass(
    SDL_GPUCommandBuffer *command_buffer,
    const SDL_GPUColorTargetInfo *color_target_infos,
    Uint32 num_color_targets,
    const SDL_GPUDepthStencilTargetInfo *depth_stencil_target_info
);

// example

// create a color target
SDL_GPUColorTargetInfo colorTargetInfo{};
colorTargetInfo.clear_color = {240/255.0f, 240/255.0f, 240/255.0f, 255/255.0
colorTargetInfo.load_op = SDL_GPU_LOADOP_CLEAR;
colorTargetInfo.store_op = SDL_GPU_STOREOP_STORE;
colorTargetInfo.texture = swapchainTexture;

// begin a render pass
SDL_GPURenderPass* renderPass = SDL_BeginGPURenderPass(commandBuffer, &color

// draw something

// end the render pass
SDL_EndGPURenderPass(renderPass);
```

- color_target_infos: it's an array of the struct `SDL_GPUColorTargetInfo`, this allows you to render to multiple color targets at the same time. Using `&colorTargetInfo` as a pointer just treats the struct as an array that contains a single element.
- num_color_targets: it specifies the size of the previous array.
- depth_stencil_target_info: it's used to describe the `depth` and `stencil` operations which are useful in the context of 3d applications. We will keep it `NULL` for now.

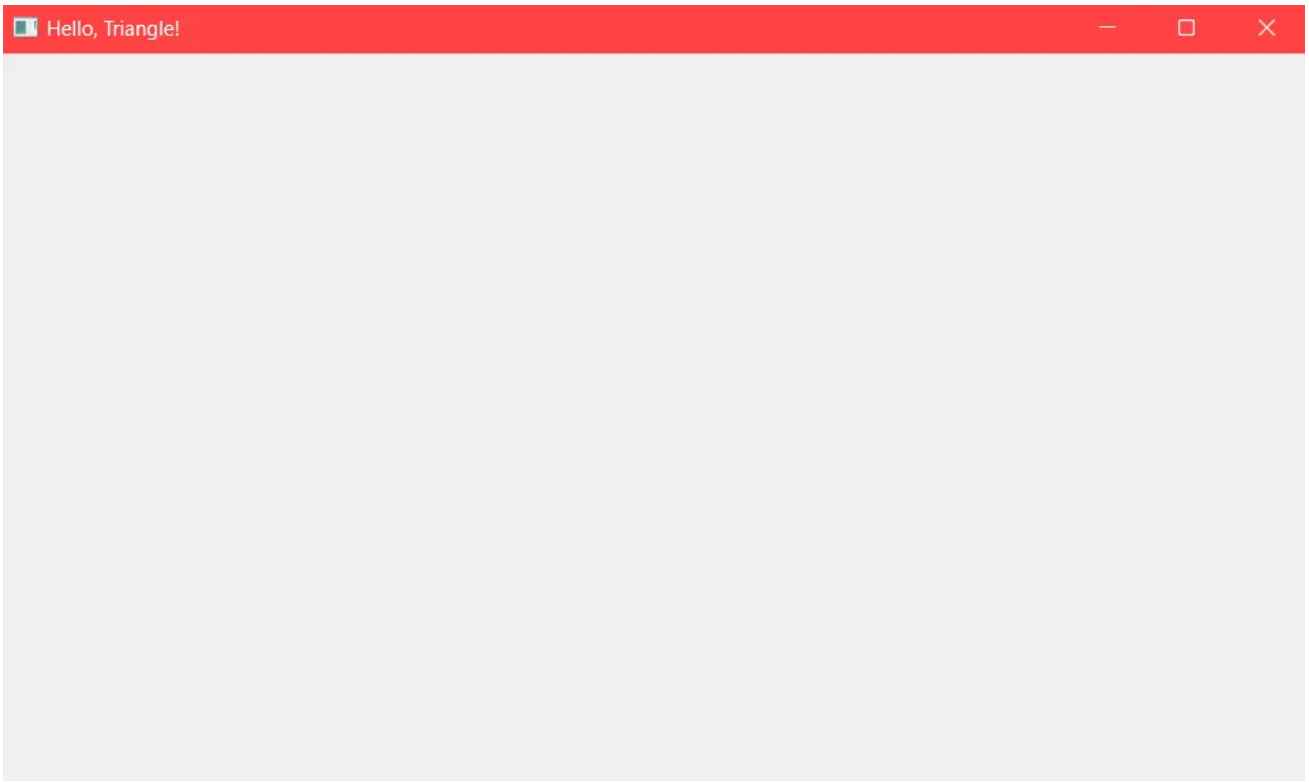
Let's put all that together.

If you did everything correctly so far, you should end up with something similar to this.

```
...
SDL_AppResult SDL_AppIterate(void *appstate)
{
    // acquire the command buffer
    SDL_GPUCommandBuffer* commandBuffer = SDL_AcquireGPUCommandBuffer(device

    // get the swapchain texture
    SDL_GPUTexture* swapchainTexture;
    Uint32 width, height;
    SDL_WaitAndAcquireGPUSwapchainTexture(commandBuffer, window, &swapchainT

    // end the frame early if a swapchain texture is not available
    if (swapchainTexture == NULL)
    {
        // you must always submit the command buffer
        SDL_SubmitGPUCommandBuffer(commandBuffer);
        return SDL_APP_CONTINUE;
    }

    // create the color target
    SDL_GPUColorTargetInfo colorTargetInfo{};
    colorTargetInfo.clear_color = {240/255.0f, 240/255.0f, 240/255.0f, 255/2
    colorTargetInfo.load_op = SDL_GPU_LOADOP_CLEAR;
    colorTargetInfo.store_op = SDL_GPU_STOREOP_STORE;
    colorTargetInfo.texture = swapchainTexture;

    // begin a render pass
    SDL_GPURenderPass* renderPass = SDL_BeginGPURenderPass(commandBuffer, &c
```

```
        // draw something

        // end the render pass
        SDL_EndGPURenderPass(renderPass);

        // submit the command buffer
        SDL_SubmitGPUCommandBuffer(commandBuffer);

        return SDL_APP_CONTINUE;
    }
    ...
```

and voila! You've got your first SDL_gpu program running and it does, well, nothing but clear the window to a light greyish color. To recap, you have created a GPU device, a color target with the window's swapchain texture, then began a simple render pass that clears the screen.



So, how do we draw anything?

How does floats and integers end up as something on the screen? This is done by making a GPU pipeline. The steps needed can be summarized in the following points:

- You first create GPU buffers that hold data and upload the data to these GPU buffers. You can create **Vertex Buffers** that contain positions, colors, textures

UV, etc. You can upload **Textures** that you want to be available to use in shaders. You can also create other types of buffers like **Uniform Buffers, Storage Buffers,** and **Instance Data Buffers**. You will need to update all these as the state of your app changes.

- You also need **Vertex shaders**, **Fragment shaders**, and finally a **Graphics pipeline** that uses these shaders. The graphics pipeline tells the GPU how to interpret the data provided in buffers and textures. It also allows you to create some awesome effects.

- You finally make your **draw call** after setting the correct state. You can choose which pipeline, which textures, which buffers, what offset, and how many vertices you want the GPU to draw.

Creating and updating buffers.

The first thing we need is to create a **Vertex Buffer** that will be later used as an input for the **vertex shader**. This buffer will contain two things, 3 floats for the position, and 4 floats for the color. Our triangle can be represented as:

```
// the vertex input layout
struct Vertex
{
    float x, y, z;      //vec3 position
    float r, g, b, a;   //vec4 color
};

// a list of vertices
static Vertex vertices[]
{
    {0.0f, 0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f},     // top vertex
    {-0.5f, -0.5f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f},   // bottom left vertex
    {0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f}     // bottom right vertex
};
```

You may notice how the positions of the vertices are being in the [-1, 1] range. This is because it's in **NDC** or **Normalized Device Coordinates**. Converting them to the window coordinates is out of the scope of this tutorial. Just remember that (0, 0) is in the center, (1, 1) is top right, and (-1, -1) is bottom left. Now, we want to send this list to the GPU. The first problem is *where* to upload this data to. For

that, we will create a new GPU buffer.

```
// create the vertex buffer
SDL_GPUBufferCreateInfo bufferInfo{};
bufferInfo.size = sizeof(vertices);
bufferInfo.usage = SDL_GPU_BUFFERUSAGE_VERTEX;
SDL_GPUBuffer* vertexBuffer = SDL_CreateGPUBuffer(device, &bufferInfo);
```

This will create a buffer with the size of the vertices list `(3 vertices * 7 floats per vertex * 4 bytes per float)`, and it will only be used as a vertex shader input, hence it will be a `VERTEX` buffer. You can change the usage to create `storage`, `index`, and `indirect` buffers and change what is allowed to use the buffer for. We don't any of that for what are doing though. Note that creating buffers is an expensive operation, so be sure to only create them earlier in the app and reuse the buffers later instead of creating them every frame. You also have to release the buffer when it's no longer needed to free its memory. It's released like this:

```
SDL_ReleaseGPUBuffer(device, vertexBuffer);
```

The second problem is *what* the content of the buffer should be. This buffer on its own is useless. To move the data from the CPU to the GPU, a special buffer is needed which is a **Transfer Buffer**. A transfer buffer is first mapped to data on the CPU then later that data is uploaded to the GPU buffer in a **copy pass**. To create a transfer buffer:

```
// create a transfer buffer to upload to the vertex buffer
SDL_GPUTransferBufferCreateInfo transferInfo{};
transferInfo.size = sizeof(vertices);
transferInfo.usage = SDL_GPU_TRANSFERBUFFERUSAGE_UPLOAD;
SDL_GPUTransferBuffer* transferBuffer = SDL_CreateGPUTransferBuffer(device,
```

Next step is to fill the transfer buffer with the data. This is generally done like this:

```
// map the transfer buffer to a pointer
Vertex* data = (Vertex*)SDL_MapGPUTransferBuffer(device, transferBuffer, fal

data[0] = vertices[0];
data[1] = vertices[1];
data[2] = vertices[2];

// or you can copy them all in one operation
// SDL_memcpy(data, vertices, sizeof(vertices));

// unmap the pointer when you are done updating the transfer buffer
SDL_UnmapGPUTransferBuffer(device, transferBuffer);
```

Finally, the last problem is *how* should the vertex buffer be updated. We need to transfer data from the transfer buffer to the vertex buffer. To do this, we need to start a **Copy Pass**, specify what we are copying, specify where it's being copied to, and actually copy it. You can do this copy pass every frame before your render pass to dynamically update changing data, but in our case, we don't expect the triangle to change so I will only do it once at the beginning of the app.

```
// start a copy pass
SDL_GPUCommandBuffer* commandBuffer = SDL_AcquireGPUCommandBuffer(device);
SDL_GPUCopyPass* copyPass = SDL_BeginGPUCopyPass(commandBuffer);

// where is the data
SDL_GPUTransferBufferLocation location{};
location.transfer_buffer = transferBuffer;
location.offset = 0; // start from the beginning

// where to upload the data
SDL_GPUBufferRegion region{};
region.buffer = vertexBuffer;
region.size = sizeof(vertices); // size of the data in bytes
region.offset = 0; // begin writing from the first vertex

// upload the data
SDL_UploadToGPUBuffer(copyPass, &location, &region, true);

// end the copy pass
SDL_EndGPUCopyPass(copyPass);
SDL_SubmitGPUCommandBuffer(commandBuffer);
```

If you did everything correctly, your triangle should now live inside the GPU buffer and be accessible for the GPU to use. Similar to other buffers, transfer buffer should be cached, reused, and released once they are no longer needed.

```
// release buffers
SDL_ReleaseGPUBuffer(device, vertexBuffer);
SDL_ReleaseGPUTransferBuffer(device, transferBuffer);
```

Now that the GPU know about our triangle, it still does not know what to do with it. To solve that, we can create little GPU programs called shaders that tells the GPU how to interpret the vertices or output the colors. You have to provide at least one vertex shader and one fragment shader in order to draw anything.

Vertex Shaders.

The main purpose of a vertex shader is to set the position of the vertices. Typically, the vertex shader gets some inputs or `attributes` that it uses to set the

vertex position, then it can set outputs or `varyings` that are sent to the next step, which is a fragment shader in case of SDL_gpu. I will be using **GLSL Shaders** because they are easier to explain, but SDL doesn't care how you create your shaders as long as you provide the correct shader format which is `.spv` in this situation. You can use `.hlsl` or anything else you like as long as you provide the correct SPIRV file at the end. Creating a shader is an expensive operation, so only create them at the beginning of your app and cache and reuse them as needed. Let's create a `vertex.glsl` file at `shaders/vertex.glsl`. Our first vertex shader will be something like this:

```
#version 460

layout (location = 0) in vec3 a_position;
layout (location = 1) in vec4 a_color;
layout (location = 0) out vec4 v_color;

void main()
{
    gl_Position = vec4(a_position, 1.0f);
    v_color = a_color;
}
```

First the version indicates that it's using OpenGL 4.6, which does not matter because we will be converting it to Vulkan's SPIRV anyway. Next, we have multiple variables that are manually set to certain locations. We have a `vec3` (3 floats) `attribute` (input) called `a_position` that is mapped to the first vertex attribute at `location 0`. Our vertex shader also accepts another attribute for colors called `a_color` that is a `vec4` at `location 1`. This means our shader will take `in` input from our vertex buffer then map these to the correct variables. Next, we have a `varying` (output) variable called `v_color` that is a `vec4` as well at `location 0`. This will be passed to our fragment shader later. What the shader does is pretty simple, we just set the vertex position using `gl_Position` to the position passed in the vertex buffer. `vec4(a_position, 1.0f)` is the same as `vec4(a_position.x, a_position.y, a_position.z, 1.0f)`. The last "w" is `1.0f` has to do with cameras and perspective, just set it to `1.0f` and don't worry about it for now. Finally, we set the output `v_color` to the input `a_color`. Which essentially just passes the variable from the vertex buffer to the fragment shader.

To actually compile your `.glsl` shaders to the `.spv` format accepted by Vulkan. You will need to install the Vulkan SDK. You may need to add it to `environment variables` if you are on windows. If you've installed everything correctly you will have access to the command `glslc` that can compile your shader. Run the following command to compile the shader:

```
glslc -fshader-stage=vertex shaders/vertex.glsl -o shaders/vertex.spv
```

This takes in our shader at `shaders/vertex.glsl` and outputs `shaders/vertex.spv` and set the stage to be a vertex shader using `-fshader-stage=vertex`.

Now, we need to load the shader in SDL. To do that, we first load the file containing compiled shader code, then create a shader with that code. To do that we can do the following:

```
// load the vertex shader code
size_t vertexCodeSize;
void* vertexCode = SDL_LoadFile("shaders/vertex.spv", &vertexCodeSize);

// create the vertex shader
SDL_GPUShaderCreateInfo vertexInfo{};
vertexInfo.code = (Uint8*)vertexCode; //convert to an array of bytes
vertexInfo.code_size = vertexCodeSize;
vertexInfo.entrypoint = "main";
vertexInfo.format = SDL_GPU_SHADERFORMAT_SPIRV; // loading .spv shaders
vertexInfo.stage = SDL_GPU_SHADERSTAGE_VERTEX; // vertex shader
vertexInfo.num_samplers = 0;
vertexInfo.num_storage_buffers = 0;
vertexInfo.num_storage_textures = 0;
vertexInfo.num_uniform_buffers = 0;
SDL_GPUShader* vertexShader = SDL_CreateGPUShader(device, &vertexInfo);

// free the file
SDL_free(vertexCode);
```

This loads a file from disk, then create a shader using the loaded file data. We have to set the format to be `SPIRV`, the stage to be `VERTEX`, and tell SDL what

resources are used in the shaders. We don't use are textures, storage buffers, or any uniforms yet. You can set them all to zeros for now. You also have to specify the `entrypoint` which is the name of the function defined in the shader. Please note that the `"main"` entry may be problematic on `.msl` shaders. If you don't want to fill out all these information nor deal with different shader formats on different backends, `SDL_shadercross` can help you through `shader reflection`.

Once you are done with your shader, you have to `release` it.

```
SDL_ReleaseGPUShader(device, vertexShader);
```

Another thing, if you have installed the Vulkan SDK, you can now turn on the debug layer on the device creation to catch errors on the Vulkan side as they happen.

```
device = SDL_CreateGPUDevice(SDL_GPU_SHADERFORMAT_SPIRV, true, NULL);
```

Fragment Shaders.

Fragment shaders are used to output fragments (pixels) that are used later to be rendered to the color target. They may be discarded, blended, or changed in any way depending on the pipeline, depth testing, culling, etc. You can think of fragment shaders as effects that are applied on the GPU that is ran for every pixel that needs to be rendered. Similar to how we created our vertex shader, I will create another `shaders/fragment.glsl` with the following code:

```
#version 460

layout (location = 0) in vec4 v_color;
layout (location = 0) out vec4 FragColor;

void main()
{
    FragColor = v_color;
}
```

This effectively defines a fragment input or varying `v_color` at the `location 0` that is passed from the vertex shader at the same location. This also defines an output `FragColor` that saves the final color to the resulting pixel. Next, you have to compile it to `fragment.spv` using a similar command to the one we used before:

```
glslc -fshader-stage=fragment shaders/fragment.glsl -o shaders/fragment.spv
```

Note that we set `-fshader-stage=fragment` instead of setting it to `vertex`. We can create the fragment shader in SDL in the same way we created our vertex shader.

```
// create the fragment shader
size_t fragmentCodeSize;
void* fragmentCode = SDL_LoadFile("shaders/fragment.spv", &fragmentCodeSize)

// create the fragment shader
SDL_GPUShaderCreateInfo fragmentInfo{};
```

```
fragmentInfo.code = (Uint8*)fragmentCode;
fragmentInfo.code_size = fragmentCodeSize;
fragmentInfo.entrypoint = "main";
fragmentInfo.format = SDL_GPU_SHADERFORMAT_SPIRV;
fragmentInfo.stage = SDL_GPU_SHADERSTAGE_FRAGMENT; // fragment shader
fragmentInfo.num_samplers = 0;
fragmentInfo.num_storage_buffers = 0;
fragmentInfo.num_storage_textures = 0;
fragmentInfo.num_uniform_buffers = 0;

SDL_GPUShader* fragmentShader = SDL_CreateGPUShader(device, &fragmentInfo);

// free the file
SDL_free(fragmentCode);
```

Don’t forget to release the shader when you don’t need it anymore.

```
SDL_ReleaseGPUShader(device, fragmentShader);
```

The Graphics Pipeline

The graphics pipeline specifies what shaders to use, how many buffers, describes the vertex inputs, how to blend colors, and many other things that are irrelevant for what we are doing now.

First, let’s bind the shaders and set the primitive type:

```
SDL_GPUGraphicsPipelineCreateInfo pipelineInfo{};

// bind shaders
pipelineInfo.vertex_shader = vertexShader;
pipelineInfo.fragment_shader = fragmentShader;

// draw triangles
pipelineInfo.primitive_type = SDL_GPU_PRIMITIVETYPE_TRIANGLELIST;
```

The primitive types tell the GPU what vertices mean. For example, here we are drawing a list of triangles. You can also change it to draw points or lines if you want.

The next step is to define the expected buffers to use in the pipeline.

```
// describe the vertex buffers
SDL_GPUVertexBufferDescription vertexBufferDesctiptions[1];
vertexBufferDesctiptions[0].slot = 0;
vertexBufferDesctiptions[0].input_rate = SDL_GPU_VERTEXINPUTRATE_VERTEX;
vertexBufferDesctiptions[0].instance_step_rate = 0;
vertexBufferDesctiptions[0].pitch = sizeof(Vertex);

pipelineInfo.vertex_input_state.num_vertex_buffers = 1;
pipelineInfo.vertex_input_state.vertex_buffer_descriptions = vertexBufferDes
```

This means we have one vertex buffer that is set to `slot 0` which we will use later when binding the buffer just before the draw call. The buffer input rate is set to change its be per `VERTEX` and not per `INSTANCE`. This means that the select data from the buffer will be changed to the next on every vertex. So, the first vertex gets `vertices[0]`, the second vertex gets `vertices[1]`, and so on. The last `pitch` means how many bytes to jump after each cycle. Here we jump ahead the size of our defined `Vertex` struct per `VERTEX`.

The next step is to describe the layout of our vertex attributes.

```
// describe the vertex attribute
SDL_GPUVertexAttribute vertexAttributes[2];

// a_position
vertexAttributes[0].buffer_slot = 0; // fetch data from the buffer at slot 0
vertexAttributes[0].location = 0; // layout (location = 0) in shader
vertexAttributes[0].format = SDL_GPU_VERTEXELEMENTFORMAT_FLOAT3; //vec3
vertexAttributes[0].offset = 0; // start from the first byte from current bu

// a_color
vertexAttributes[1].buffer_slot = 0; // use buffer at slot 0
vertexAttributes[1].location = 1; // layout (location = 1) in shader
vertexAttributes[1].format = SDL_GPU_VERTEXELEMENTFORMAT_FLOAT4; //vec4
vertexAttributes[1].offset = sizeof(float) * 3; // 4th float from current bu

pipelineInfo.vertex_input_state.num_vertex_attributes = 2;
pipelineInfo.vertex_input_state.vertex_attributes = vertexAttributes;
```

We have to describe each vertex attribute we plan to use in our vertex shader. We first set the `buffer_slot` so we know which buffer to use, then the `location` to match the location in shader, then the `format` to match the format in the shader, and finally the `offset` to know where exactly in the buffer to look. For example, we have 2 variables `vec3 position` and `vec4 color`, so our vertex layout has the first 3 float (12 bytes) allocated for the position and the other 4 floats (16 bytes) allocated for the color. So, the first attribute starts at the first float and takes the size of 3 floats. The second starts at the 4th float and takes the size of 4 floats. First `offset` is `0` and second `offset` is `sizeof(float) * 3` which equals `12` or the 13th byte. A general rule is to just use the sum of whatever attributes that comes before. If you have a `vec3 position` then a `vec2 uv` then a `vec4 color` at last. The offset of the `vec4 color` should be `sizeof(float) * 3 + sizeof(float) * 2`.

You can in fact skip vertex buffers and vertex attributes by exclusively using storage buffers, but that's also out of the scope of what we are trying to accomplish at the moment.

The last thing we have to do is to describe the color target that this pipeline is created to work on.

```
// describe the color target
SDL_GPUColorTargetDescription colorTargetDescriptions[1];
colorTargetDescriptions[0] = {};
colorTargetDescriptions[0].format = SDL_GetGPUSwapchainTextureFormat(device,

pipelineInfo.target_info.num_color_targets = 1;
pipelineInfo.target_info.color_target_descriptions = colorTargetDescriptions
```

This assumes you are only drawing to the window, so the `format` is set to equal the format of the swapchain texture. You have to make sure that the textures you want to use match the format set in the pipeline. We don't use any textures, so we don't have to worry about this for now. `SDL_GPUColorTargetDescription` also have a `blend_state` that can be used to change how blending is applied. You can learn more on blending here. Though not really needed for this example, a typical blending setup is something like this:

```
SDL_GPUColorTargetDescription colorTargetDescriptions[1];
colorTargetDescriptions[0] = {};
colorTargetDescriptions[0].blend_state.enable_blend = true;
colorTargetDescriptions[0].blend_state.color_blend_op = SDL_GPU_BLENDOP_ADD;
colorTargetDescriptions[0].blend_state.alpha_blend_op = SDL_GPU_BLENDOP_ADD;
colorTargetDescriptions[0].blend_state.src_color_blendfactor = SDL_GPU_BLEND
colorTargetDescriptions[0].blend_state.dst_color_blendfactor = SDL_GPU_BLEND
colorTargetDescriptions[0].blend_state.src_alpha_blendfactor = SDL_GPU_BLEND
colorTargetDescriptions[0].blend_state.dst_alpha_blendfactor = SDL_GPU_BLEND
colorTargetDescriptions[0].format = SDL_GetGPUSwapchainTextureFormat(device,

pipelineInfo.target_info.num_color_targets = 1;
pipelineInfo.target_info.color_target_descriptions = colorTargetDescriptions
```

We can finally build our pipeline and release shader that are not needed any more.

```
// create the pipeline
SDL_GPUGraphicsPipeline* graphicsPipeline = SDL_CreateGPUGraphicsPipeline(de

// we don't need to store the shaders after creating the pipeline
SDL_ReleaseGPUShader(device, vertexShader);
SDL_ReleaseGPUShader(device, fragmentShader);
```

When you are done with the pipeline, you also have to release it.

```
// release the pipeline
SDL_ReleaseGPUGraphicsPipeline(device, graphicsPipeline);
```

Your final code so far should look something like this:

```
#define SDL_MAIN_USE_CALLBACKS
#include <SDL3/SDL_main.h>
#include <SDL3/SDL.h>
```

```
// the vertex input layout
struct Vertex
{
    float x, y, z;      //vec3 position
    float r, g, b, a;   //vec4 color
};

// a list of vertices
static Vertex vertices[]
{
    {0.0f, 0.5f, 0.0f, 1.0f, 0.0f, 0.0f, 1.0f},     // top vertex
    {-0.5f, -0.5f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f},   // bottom left vertex
    {0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f}     // bottom right vertex
};

SDL_Window* window;
SDL_GPUDevice* device;
SDL_GPUBuffer* vertexBuffer;
SDL_GPUTransferBuffer* transferBuffer;
SDL_GPUGraphicsPipeline* graphicsPipeline;

SDL_AppResult SDL_AppInit(void **appstate, int argc, char **argv)
{
    // create a window
    window = SDL_CreateWindow("Hello, Triangle!", 960, 540, SDL_WINDOW_RESIZ

    // create the device
    device = SDL_CreateGPUDevice(SDL_GPU_SHADERFORMAT_SPIRV, true, NULL);
    SDL_ClaimWindowForGPUDevice(device, window);

    // load the vertex shader code
    size_t vertexCodeSize;
    void* vertexCode = SDL_LoadFile("shaders/vertex.spv", &vertexCodeSize);

    // create the vertex shader
    SDL_GPUShaderCreateInfo vertexInfo{};
    vertexInfo.code = (Uint8*)vertexCode;
    vertexInfo.code_size = vertexCodeSize;
    vertexInfo.entrypoint = "main";
    vertexInfo.format = SDL_GPU_SHADERFORMAT_SPIRV;
    vertexInfo.stage = SDL_GPU_SHADERSTAGE_VERTEX;
    vertexInfo.num_samplers = 0;
    vertexInfo.num_storage_buffers = 0;
    vertexInfo.num_storage_textures = 0;
    vertexInfo.num_uniform_buffers = 0;

    SDL_GPUShader* vertexShader = SDL_CreateGPUShader(device, &vertexInfo);

    // free the file
    SDL_free(vertexCode);
```

```
// load the fragment shader code
size_t fragmentCodeSize;
void* fragmentCode = SDL_LoadFile("shaders/fragment.spv", &fragmentCodeS

// create the fragment shader
SDL_GPUShaderCreateInfo fragmentInfo{};
fragmentInfo.code = (Uint8*)fragmentCode;
fragmentInfo.code_size = fragmentCodeSize;
fragmentInfo.entrypoint = "main";
fragmentInfo.format = SDL_GPU_SHADERFORMAT_SPIRV;
fragmentInfo.stage = SDL_GPU_SHADERSTAGE_FRAGMENT;
fragmentInfo.num_samplers = 0;
fragmentInfo.num_storage_buffers = 0;
fragmentInfo.num_storage_textures = 0;
fragmentInfo.num_uniform_buffers = 0;

SDL_GPUShader* fragmentShader = SDL_CreateGPUShader(device, &fragmentInf

// free the file
SDL_free(fragmentCode);

// create the graphics pipeline
SDL_GPUGraphicsPipelineCreateInfo pipelineInfo{};
pipelineInfo.vertex_shader = vertexShader;
pipelineInfo.fragment_shader = fragmentShader;
pipelineInfo.primitive_type = SDL_GPU_PRIMITIVETYPE_TRIANGLELIST;

// describe the vertex buffers
SDL_GPUVertexBufferDescription vertexBufferDesctiptions[1];
vertexBufferDesctiptions[0].slot = 0;
vertexBufferDesctiptions[0].input_rate = SDL_GPU_VERTEXINPUTRATE_VERTEX;
vertexBufferDesctiptions[0].instance_step_rate = 0;
vertexBufferDesctiptions[0].pitch = sizeof(Vertex);

pipelineInfo.vertex_input_state.num_vertex_buffers = 1;
pipelineInfo.vertex_input_state.vertex_buffer_descriptions = vertexBuffe

// describe the vertex attribute
SDL_GPUVertexAttribute vertexAttributes[2];

// a_position
vertexAttributes[0].buffer_slot = 0;
vertexAttributes[0].location = 0;
vertexAttributes[0].format = SDL_GPU_VERTEXELEMENTFORMAT_FLOAT3;
vertexAttributes[0].offset = 0;

// a_color
vertexAttributes[1].buffer_slot = 0;
vertexAttributes[1].location = 1;
vertexAttributes[1].format = SDL_GPU_VERTEXELEMENTFORMAT_FLOAT4;
vertexAttributes[1].offset = sizeof(float) * 3;
```

```
pipelineInfo.vertex_input_state.num_vertex_attributes = 2;
pipelineInfo.vertex_input_state.vertex_attributes = vertexAttributes;

// describe the color target
SDL_GPUColorTargetDescription colorTargetDescriptions[1];
colorTargetDescriptions[0] = {};
colorTargetDescriptions[0].blend_state.enable_blend = true;
colorTargetDescriptions[0].blend_state.color_blend_op = SDL_GPU_BLENDOP_A
colorTargetDescriptions[0].blend_state.alpha_blend_op = SDL_GPU_BLENDOP_A
colorTargetDescriptions[0].blend_state.src_color_blendfactor = SDL_GPU_BL
colorTargetDescriptions[0].blend_state.dst_color_blendfactor = SDL_GPU_BL
colorTargetDescriptions[0].blend_state.src_alpha_blendfactor = SDL_GPU_BL
colorTargetDescriptions[0].blend_state.dst_alpha_blendfactor = SDL_GPU_BL
colorTargetDescriptions[0].format = SDL_GetGPUSwapchainTextureFormat(dev

pipelineInfo.target_info.num_color_targets = 1;
pipelineInfo.target_info.color_target_descriptions = colorTargetDescript

// create the pipeline
graphicsPipeline = SDL_CreateGPUGraphicsPipeline(device, &pipelineInfo);

// we don't need to store the shaders after creating the pipeline
SDL_ReleaseGPUShader(device, vertexShader);
SDL_ReleaseGPUShader(device, fragmentShader);

// create the vertex buffer
SDL_GPUBufferCreateInfo bufferInfo{};
bufferInfo.size = sizeof(vertices);
bufferInfo.usage = SDL_GPU_BUFFERUSAGE_VERTEX;
vertexBuffer = SDL_CreateGPUBuffer(device, &bufferInfo);

// create a transfer buffer to upload to the vertex buffer
SDL_GPUTransferBufferCreateInfo transferInfo{};
transferInfo.size = sizeof(vertices);
transferInfo.usage = SDL_GPU_TRANSFERBUFFERUSAGE_UPLOAD;
transferBuffer = SDL_CreateGPUTransferBuffer(device, &transferInfo);

// fill the transfer buffer
Vertex* data = (Vertex*)SDL_MapGPUTransferBuffer(device, transferBuffer,

SDL_memcpy(data, (void*)vertices, sizeof(vertices));

// data[0] = vertices[0];
// data[1] = vertices[1];
// data[2] = vertices[2];

SDL_UnmapGPUTransferBuffer(device, transferBuffer);

// start a copy pass
SDL_GPUCommandBuffer* commandBuffer = SDL_AcquireGPUCommandBuffer(device
SDL_GPUCopyPass* copyPass = SDL_BeginGPUCopyPass(commandBuffer);
```

```
    // where is the data
    SDL_GPUTransferBufferLocation location{};
    location.transfer_buffer = transferBuffer;
    location.offset = 0;

    // where to upload the data
    SDL_GPUBufferRegion region{};
    region.buffer = vertexBuffer;
    region.size = sizeof(vertices);
    region.offset = 0;

    // upload the data
    SDL_UploadToGPUBuffer(copyPass, &location, &region, true);

    // end the copy pass
    SDL_EndGPUCopyPass(copyPass);
    SDL_SubmitGPUCommandBuffer(commandBuffer);

    return SDL_APP_CONTINUE;
}

SDL_AppResult SDL_AppIterate(void *appstate)
{
    // acquire the command buffer
    SDL_GPUCommandBuffer* commandBuffer = SDL_AcquireGPUCommandBuffer(device

    // get the swapchain texture
    SDL_GPUTexture* swapchainTexture;
    Uint32 width, height;
    SDL_WaitAndAcquireGPUSwapchainTexture(commandBuffer, window, &swapchainT

    // end the frame early if a swapchain texture is not available
    if (swapchainTexture == NULL)
    {
        // you must always submit the command buffer
        SDL_SubmitGPUCommandBuffer(commandBuffer);
        return SDL_APP_CONTINUE;
    }

    // create the color target
    SDL_GPUColorTargetInfo colorTargetInfo{};
    colorTargetInfo.clear_color = {240/255.0f, 240/255.0f, 240/255.0f, 255/2
    colorTargetInfo.load_op = SDL_GPU_LOADOP_CLEAR;
    colorTargetInfo.store_op = SDL_GPU_STOREOP_STORE;
    colorTargetInfo.texture = swapchainTexture;

    // begin a render pass
    SDL_GPURenderPass* renderPass = SDL_BeginGPURenderPass(commandBuffer, &c

    // draw calls go here

    // end the render pass
```

```
    SDL_EndGPURenderPass(renderPass);

    // submit the command buffer
    SDL_SubmitGPUCommandBuffer(commandBuffer);

    return SDL_APP_CONTINUE;
}

SDL_AppResult SDL_AppEvent(void *appstate, SDL_Event *event)
{
    // close the window on request
    if (event->type == SDL_EVENT_WINDOW_CLOSE_REQUESTED)
    {
        return SDL_APP_SUCCESS;
    }

    return SDL_APP_CONTINUE;
}

void SDL_AppQuit(void *appstate, SDL_AppResult result)
{
    // release buffers
    SDL_ReleaseGPUBuffer(device, vertexBuffer);
    SDL_ReleaseGPUTransferBuffer(device, transferBuffer);

    // release the pipeline
    SDL_ReleaseGPUGraphicsPipeline(device, graphicsPipeline);

    // destroy the GPU device
    SDL_DestroyGPUDevice(device);

    // destroy the window
    SDL_DestroyWindow(window);
}
```

Draw Calls

Now that you have your buffers and pipelines ready, let's ask the GPU to render something.

The first thing we need is to tell the GPU what pipeline to use. In your render pass, bind your graphics pipeline.

```
SDL_GPURenderPass* renderPass = SDL_BeginGPURenderPass(commandBuffer, &color

// bind the graphics pipeline
SDL_BindGPUGraphicsPipeline(renderPass, graphicsPipeline);
```

```
...

SDL_EndGPURenderPass(renderPass);
```

Next, we have to bind our `vertexBuffer` to the `buffer_slot 0` that we defined in the pipeline.

```
// begin a render pass
SDL_GPURenderPass* renderPass = SDL_BeginGPURenderPass(commandBuffer, &color

// bind the pipeline
SDL_BindGPUGraphicsPipeline(renderPass, graphicsPipeline);

// bind the vertex buffer
SDL_GPUBufferBinding bufferBindings[1];
bufferBindings[0].buffer = vertexBuffer; // index 0 is slot 0 in this exampl
bufferBindings[0].offset = 0; // start from the first byte

SDL_BindGPUVertexBuffers(renderPass, 0, bufferBindings, 1); // bind one buff

...

// end the render pass
SDL_EndGPURenderPass(renderPass);
```

This uses the `bufferBindings` array to bind vertex buffers starting from the slot 0 and with the array size of 1.

Everything is now ready for our first draw call. Get ready.

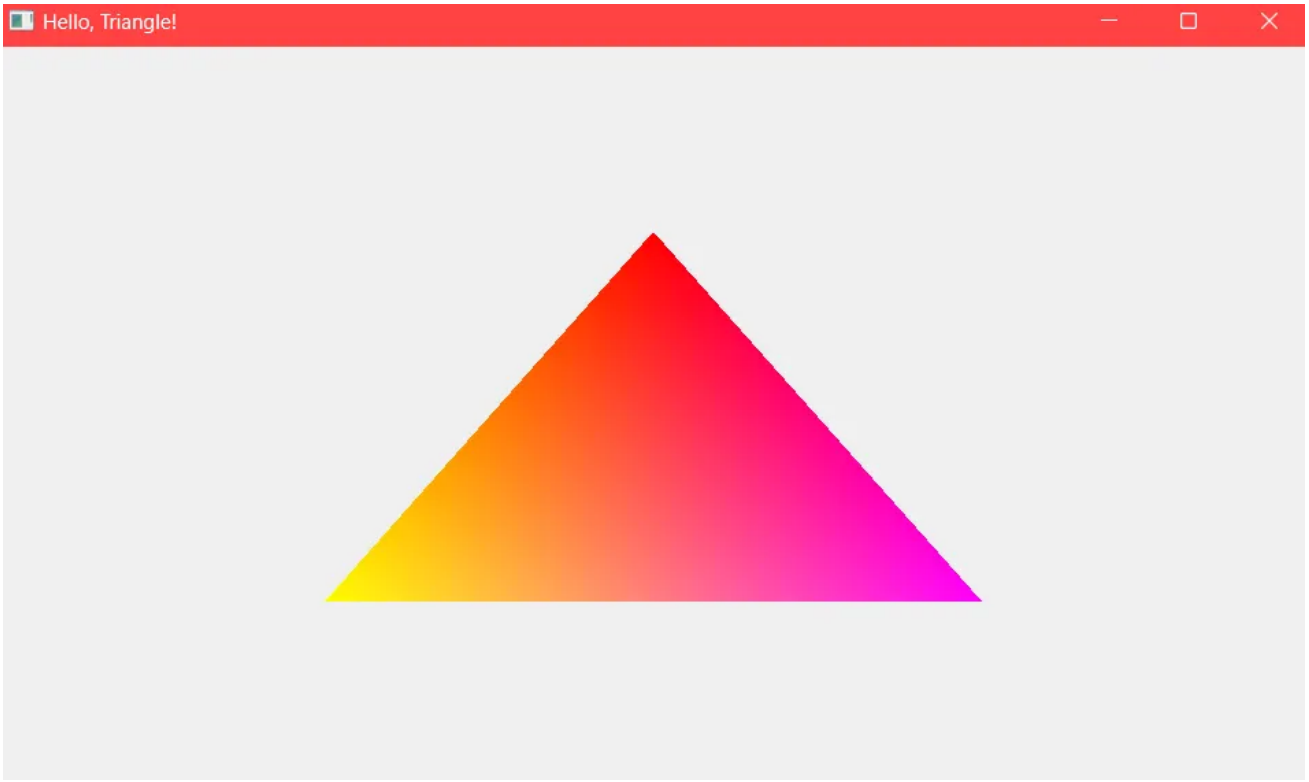
```
// begin a render pass
SDL_GPURenderPass* renderPass = SDL_BeginGPURenderPass(commandBuffer, &color

// bind the pipeline
SDL_BindGPUGraphicsPipeline(renderPass, graphicsPipeline);

// bind the vertex buffer
SDL_GPUBufferBinding bufferBindings[1];
bufferBindings[0].buffer = vertexBuffer;
bufferBindings[0].offset = 0;
```

```
SDL_BindGPUVertexBuffers(renderPass, 0, bufferBindings, 1);

// issue a draw call
SDL_DrawGPUPrimitives(renderPass, 3, 1, 0, 0);

// end the render pass
SDL_EndGPURenderPass(renderPass);
```

All this does is asks the GPU to render 3 `vertices` in one `instance` (don't worry about that now) starting from the first vertex and the first instance.

Finally, the moment you have been dying for.

Hit run!



The prettiest thing that has ever been rendered.

You did it!

Your pretty triangle should now ease all the pain you went through.

Isn't she pretty...?

I have to admit that was so much setup and boilerplate before we could even get

the simplest of geometry rendering. The good news is what we just did covers much of the basics you need learn in any graphics API. You got access to the GPU, uploaded data to the GPU, wrote some shaders that manipulates that data, created a pipeline that the GPU can use to render that data, and finally made your first draw call. The rest is just building up on what we already know. You don't have to write all that every time you need to draw something, you can easily wrap your shaders, pipelines, buffers, textures, etc. in helper functions that does all that setup behind the scenes.

Uniform Buffers

Our triangle doesn't currently do anything, which is expected. What if we wanted to pass some properties like `time` to the fragment shader like you usually see on Shadertoy? Your first thought may be something like creating a new vertex attribute `float time`. However, in this approach you will have to update this attribute for all vertices then upload the buffer every frame which is redundant, wasteful, and expensive. Uniform buffers allow you to set universal properties that can be set quickly before you make your draw call and are accessible to all vertices in that call.

Let's first define the uniform buffer layout.

```
struct UniformBuffer
{
    float time;
    // you can add other properties here
};

static UniformBuffer timeUniform{};
```

Next, we will update our fragment shader to use it.

```
#version 460

layout (location = 0) in vec4 v_color;
layout (location = 0) out vec4 FragColor;

layout(std140, set = 3, binding = 0) uniform UniformBlock {
```

```
    float time;
};

void main()
{
    float pulse = sin(time * 2.0) * 0.5 + 0.5; // range [0, 1]
    FragColor = vec4(v_color.rgb * (0.8 + pulse * 0.5), v_color.a);
}
```

Here we have a very simple pulse effect based on current time. Don't worry about the exact implementation. You may have noticed a few things though. First, we set the `set` to 3, that's because SDL has predefined sets for certain things and it happens that fragment shaders take uniforms at `set 3`. In vertex shaders, the `set 1` is used for uniforms. Check the documentation here to find out the correct set to use for different resources. The other thing is `binding` which work like a `location`. So, you can bind multiple uniforms at different slots by using different bindings. The last thing is std140 which is a standard that defines how data is laid in memory. Watch out for paddings on complex uniforms! SDL_gpu *requires* you to use std140.

Do not forget to recompile your shader.

```
glslc -fshader-stage=fragment shaders/fragment.glsl -o shaders/fragment.spv
```

You also have to update your fragment shader to know that we will be using a uniform buffer.

```
SDL_GPUShaderCreateInfo fragmentInfo{};
...
fragmentInfo.num_samplers = 0;
fragmentInfo.num_storage_buffers = 0;
fragmentInfo.num_storage_textures = 0;
fragmentInfo.num_uniform_buffers = 1; // change this to 1
...
```

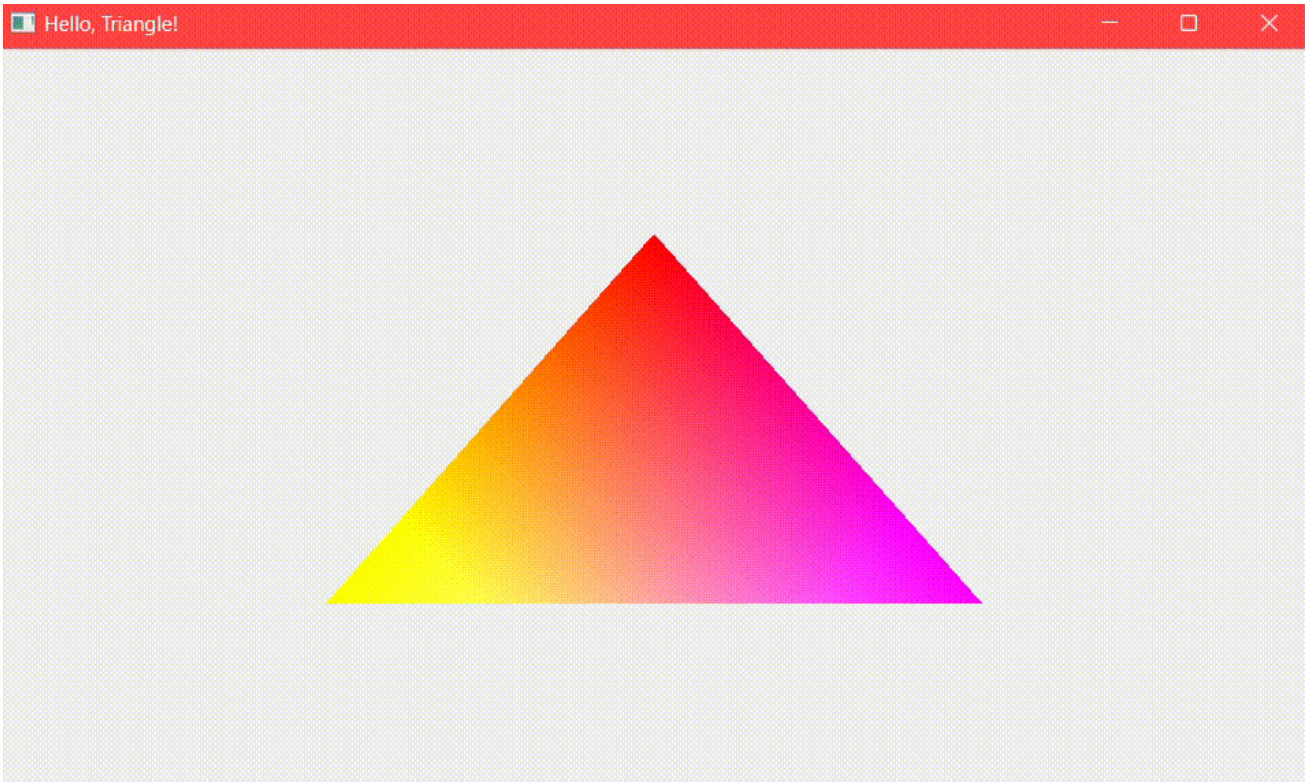
The last thing to do is to `push` the uniform to the fragment shader. Fortunately,

we don't have to create buffers, transfer buffers, start a copy pass or any of that. You can directly send them to the GPU right before your draw call. Uniforms are designed to be small and fast.

```
timeUniform.time = SDL_GetTicksNS() / 1e9f; // the time since the app starte
SDL_PushGPUFragmentUniformData(commandBuffer, 0, &timeUniform, sizeof(Unifor
```

We updated the time uniform based on how much time has passed since the start of the application in seconds. After that, we update the fragment uniform at `slot 0` or `binding 0` to have the content of our `UniformBuffer` struct.

If you did everything correctly, your triangle should have a simple pulsating effect.



Making it cross-platform!

SDL_gpu does already a good job converting your code to the Vulkan's API as well as the other supported APIs. The only issue remaining is shaders. To solve this, we will use SDL_shadercross to convert our shaders for us. We will do `online compilation` instead of `offline compilation` because it's much simpler to integrate, and while being a bit slower than loading pre-compiled shaders, it shouldn't

matter much by today's standards. If you want an example of `offline compilation`, you can have a look at the SDL_gpu examples repository. Once you have downloaded and linked SDL_shadercross to your project, you can update your project to use it to compile your shaders.

```
...

#include <SDL3_shadercross/SDL_shadercross.h>

...

device = SDL_CreateGPUDevice(SDL_ShaderCross_GetSPIRVShaderFormats(), true, N

...

SDL_ShaderCross_SPIRV_Info vertexInfo{};
vertexInfo.bytecode = (Uint8*)vertexCode;
vertexInfo.bytecode_size = vertexCodeSize;
vertexInfo.entrypoint = "main";
vertexInfo.shader_stage = SDL_SHADERCROSS_SHADERSTAGE_VERTEX;

// figure out shader metadata
SDL_ShaderCross_GraphicsShaderMetadata* vertexMetadata = SDL_ShaderCross_Ref

// cross compile to the appropriate shaderformat and create a shader object
SDL_GPUShader* vertexShader = SDL_ShaderCross_CompileGraphicsShaderFromSPIRV

// don't forget to free metadata when you no longer need it
SDL_free(vertexMetadata);

...

SDL_ShaderCross_SPIRV_Info fragmentInfo{};
fragmentInfo.bytecode = (Uint8*)fragmentCode;
fragmentInfo.bytecode_size = fragmentCodeSize;
fragmentInfo.entrypoint = "main";
fragmentInfo.shader_stage = SDL_SHADERCROSS_SHADERSTAGE_FRAGMENT;

SDL_ShaderCross_GraphicsShaderMetadata* framgentMetadata = SDL_ShaderCross_R
SDL_GPUShader* fragmentShader = SDL_ShaderCross_CompileGraphicsShaderFromSPI
SDL_free(framgentMetadata);
```

You don't have to manually set the numbers of the resources used in the shader because they are automatically set through `shader reflection` which are later stored in the `metadata` structs.

On windows, Vulkan is well supported, so you may want to disable the DXC compiler if you don't want a dependency on `dxcompiler.dll` and `dxil.dll`. However, they are properly still useful on platforms that only support Directx12 like XBox.

```
#cmake
set(SDLSHADERCROSS_DXC OFF CACHE BOOL "")
```

Alternatively, you can compile SDL_shadercross as a command line tool and use it to generate your `.spv`, `.msl`, and `.dxil` and only bundle the correct shaders on the correct platform without relying on bundling the compiler with your application. You can also set it as a build-time step that automatically compiles and update shaders as they change, but that is a bit out of the scope of this tutorial.

If you have done everything correctly, your triangle should now use your `SPIRV` binary with the correct backend on MacOS/iOS (Metal), Windows (Vulkan or DirectX12), Xbox (DirectX12), Linux/*Android**/Switch (Vulkan), PS5 (whatever the heck they use), and later the *Web** once WebGPU is added to SDL_gpu.

A note on Android.

We are in a position where Google is trying to replace OpenGLES with Vulkan, yet the Vulkan drivers are not yet up to the standards set many years ago. This results in extensions or features that are universally available on each platform to be missing on a lot of Android devices which causes SDL_gpu to fail randomly on many devices, including newer ones.

OpenGLES on the other hand, will *eventually* get always translated to Vulkan anyway in the near future, and it's no longer the recommended API for Android, and targeting a dying API is a bit pointless. To fix this issue, there are proposals to make SDL_gpu ignore these missing extensions as long as you promise not to use them, and you will have to give up on them anyway if you want your app to work on android whether you use SDL_gpu or not. The situation will eventually get better in the future after the fact that Google has adopted Vulkan as the default API and newer versions of Android will force better Vulkan support. **WebGPU**

may help mitigate this issue, but it is yet to be added to SDL_gpu.

That said, if you want a wider and more reliable Android support as of now, you may want to use OpenGLES which is not and will not be supported as a backend for SDL_gpu for various reasons.

What about web support?

Web is a difficult platform. You are limited to what browsers allow you to do or use. Currently, it is stuck in WebGL2/OpenGLES 3.0 era which is ancient, limited, and doesn't support a lot of the newer features needed by SDL_gpu. Fortunately, there are plans to add a **WebGPU** backend once it's more stable and streamlined. However, this may take a year or two and you will not be able to target web at all at the moment. If you mainly want to support web now, you will have to use the very limited WebGL2 API. Sorry!

Next steps.

1. Explore the SDL_gpu examples repository.
2. While this is an OpenGL tutorial, it also serves as a great introduction to graphics programming. You can try to convert the examples to the SDL_gpu equivalents as you go.
3. You can learn HLSL instead of GLSL.
4. Make something you are proud of!

You can find the complete project here.

Thanks for reading.
Cheers!

Graphics Programming | Sdl | Sdl Gpu | Programming | Beginners Guide

Follow

Written by Hamdy Elzanqali

5 followers · 0 following

I make games...

Responses (6)

Vilas Chitrakaran

What are your thoughts?

Zachery Abdallah
Jun 12, 2025

From a beginners POV this was great. Thanks for writing.

21 Reply

Pipe Ghill
Jul 19, 2025

Fantastic article, thank you!

1 Reply

iforce2d
Dec 5, 2025

Very helpful article and just what I needed, thanks!

I managed to get through the first part up to showing the static colored triangle pretty easily, but then spent about THREE HOURS trying to get it changing over time based on the uniform. This was... more

Reply

See all responses

Recommended from Medium

The Cache Cowgirl

This Tiny Golang App Pays My Rent

I never expected a 300-line Go program to become my side-income engine. But here we are: this tiny backend app, originally hacked together...

Jul 27, 2025 948 16

In Stackademic by Mobile App Developer

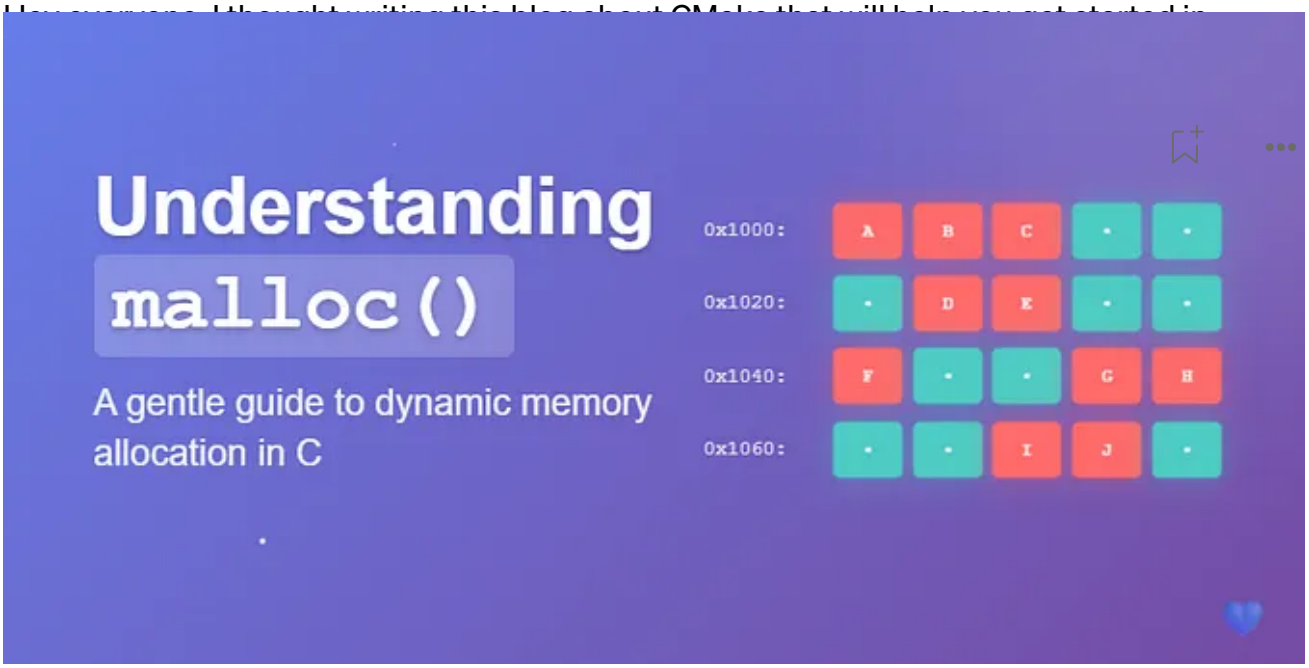
RIP Flutter? Apple's iOS 26 (Liquid Glass) Just Changed the Game — What Happened to Cross-Platform

"Flutter is dead." "React Native is done." "Apple killed cross-platform development."

Jul 14, 2025 598 28

Ragulnath M B

The Ultimate Guide to Production-Grade Projects with Modern CMake



Aniket Kumar

cHow does a malloc work?

Malloc is a standard library function that allows for dynamic memory allocation. It provides a mechanism for programs to request blocks of...

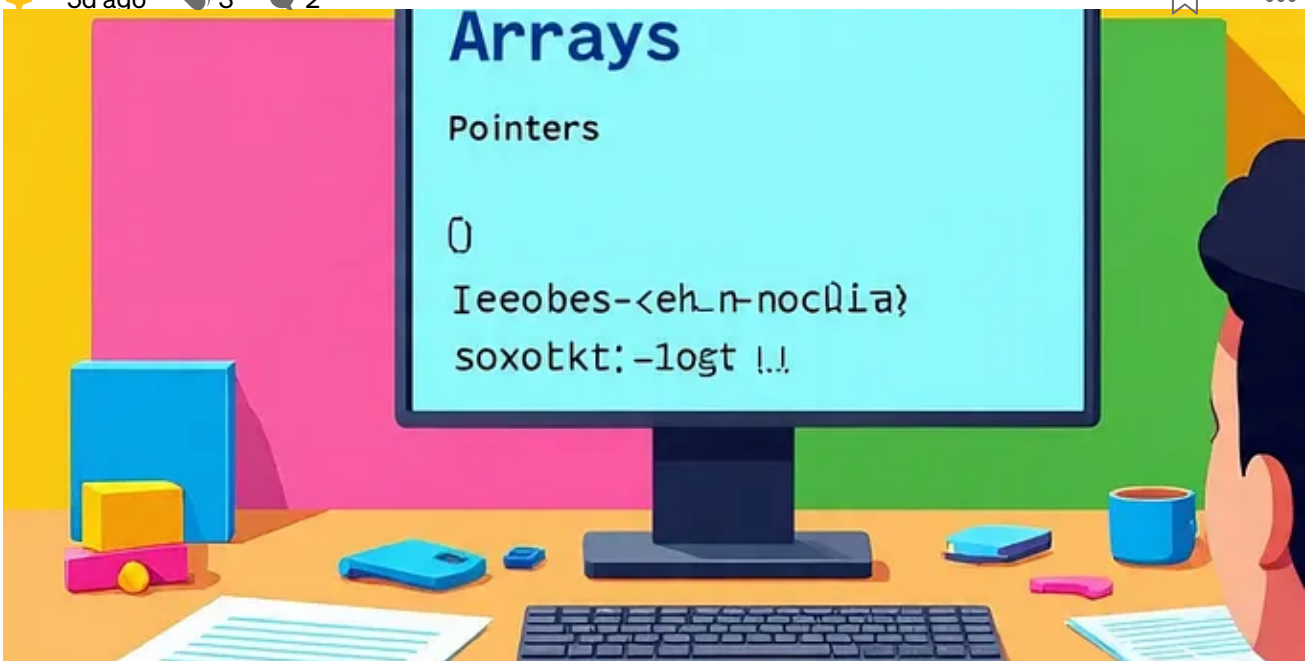
Jul 23, 2025 11

In SWE — Insights by Manish Kumar

My C++ Coding Guidelines (2026 Edition)

For High-Quality Software Development

5d ago 3 2



Aditya Bhuyan

Optimizing Memory Allocation: Zero-Length Arrays vs Pointers in C Programming

Aug 28, 2025 13 1

See more recommendations