The new power equation p10

The e-waste paradox in a hyperconnected world p26

Health trackers: The pebble or the pillow? p52

Fun with graphics programming using SDL3 and SDL_GPU p56

₹125

VOL. 38 | ISSUE 5 | May 2025

www.pcquest.com

PCQUEST

TECH TODAY & TOMORROW

CyberMedia

Fighting AI with AI: A cybersecurity blueprint for 2025

BARRIERS TO SUSTAINABLE TECH

Special Subscription offer on page 50

76 pages including cover

PCQUEST
UNDERSTAND • CHOOSE • IMPLEMENT
www.pcquest.com

For subscription queries contact:
subscriptions@cybermedia.co.in
9289870545

Send all your tech questions to:
pcquest@cybermedia.co.in

http://twitter.com/pcquest
http://facebook.com/pcquest
http://linked.in/pcquest
https://instagram.com/pcquestindia/

EXPLORE

COVER STORY
SUSTAINABLE TECHNOLOGY

Why the smartest tech might be the most unsustainable

COVER STORY
SUSTAINABLE TECHNOLOGY

p10
The new power equation

p13
Building a sustainable backbone for India's digital workforce

CORPORATE OFFICE: **Cyber House, B-35, Sec-32, Gurugram (NCR Delhi) 122003. India**
email us pcquest@cybermedia.co.in
call us +91-124-423-7517**, Ext. No.:** 347**, Mobile** +91-9953150474, +91- 7993574118

OUR OFFICES

BENGALURU: Address: 205-207, Sree Complex (Opp. RBANMS Ground), # 73, St John's Road, Bangaluru - 560 042. Tel: +91 (80) 4302 8412, Fax: +91 (80) 2530 7971

MUMBAI: Address: INS Tower, Office No. 326, Bandra Kurla Complex Road, G Block BKC, Bandra East, Mumbai – 400051. Mobile: +91 9969424024

DELHI: Address: Cyber House, B-35, Sec 32, Gurugram, NCR Delhi-122003.
Tel: 0124-4237517, Mobile: 9953150474

Printed and published by Pradeep Gupta on behalf of CyberMedia (India) Ltd, printed at printed at M/s Archna Printers, D-127, Okhla Industrial Area, Phase-1, New Delhi, published from D-74, Panchsheel Enclave, New Delhi-110017. Editor: Sunil Rajguru. Distributed in India by IBH Books & Magazines Dist. Pvt. Ltd, Mumbai.

EXPLORE

COVER STORY
SUSTAINABLE TECHNOLOGY

p16 Legacy tech, modern problems

p19 When old meets bold

p21 Rewiring the e-waste problem

p26 The e-waste paradox in a hyperconnected world

p29 Code that cares

p32 Where tech meets responsibility

p35 Not all code is created equal

p38 Green innovation: Why eco-friendly tech is no longer optional

AI

p42 The rise of intelligent automation

CYBERSECURITY

p44 Fighting AI with AI: A cybersecurity blueprint for 2025

CYBERSECURITY

p47 Prevention is better than data recovery: 7 cybersecurity priorities for India in 2025

HEALTHTECH

p52 Health trackers: The pebble or the pillow?

DEVELOPER

p56 Fun with graphics programming using SDL3 and SDL_GPU

DEVELOPER

p65 The rise of strategic game development

REVIEWS

p69 **Alogic Yoga 3-in-1**

p70 **Tecsox Alpha earbuds**

p71 **Sennheiser TeamConnect Bar M**

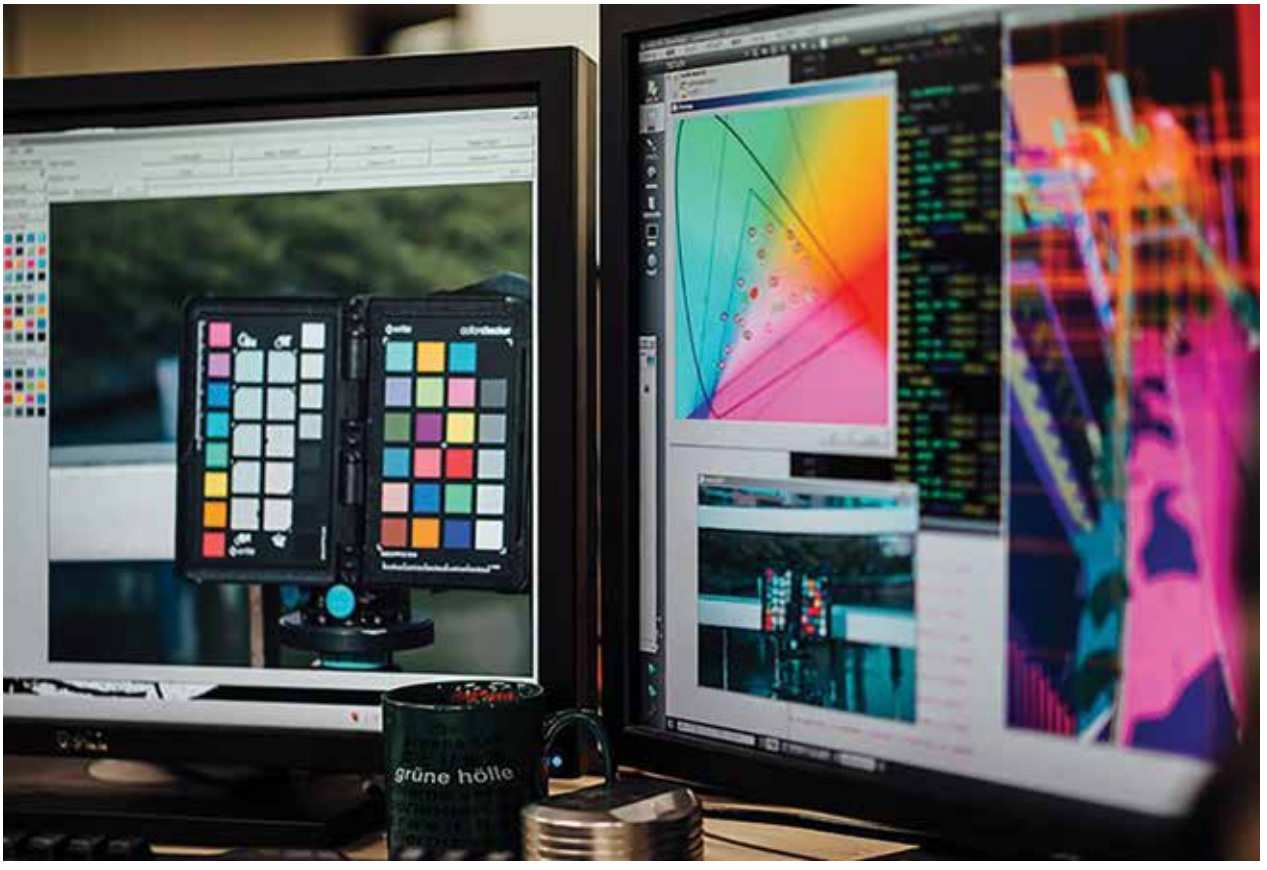
p73 **Looka AI logo tool review**

TECHTOON

My invention is simple - all the hot air that Boss produces in his speeches goes up, turns this turbine, which produces electricity!

Fun with graphics programming using SDL3 and SDL_GPU

Sankrant Chaubey
pcquest@cybermedia.co.in



From zero to triangle in no time—this hands-on journey into SDL3 and SDL_GPU strips away the boilerplate and brings color to the canvas. Say goodbye to complex APIs and hello to clean, modern graphics. One render pass at a time

Modern graphics programming can be intimidating, with complex APIs like Vulkan and DirectX requiring extensive boilerplate code just to display a simple triangle. Fortunately, SDL3 introduces a new abstraction layer called SDL_GPU, simplifying the process while still granting access to powerful graphics features.

Let's explore how to get started with the SDL_gpu module by building a program that renders a colorful triangle.

Understanding SDL_gpu

SDL_GPU is a graphics abstraction layer in SDL3. It provides a unified interface for backend APIs like Vulkan, DirectX, Metal, and OpenGL. You can write your code once and run it on multiple platforms with minimal changes.

Key advantages of SDL_GPU include:

- Cross-platform compatibility (Windows, macOS, Linux)
- Simplified pipeline setup
- Automatic handling of platform-specific details
- Support for other multimedia outcomes

This tutorial covers:

1. Installing and setting up SDL3
2. Understanding basic SDL3 concepts
3. Creating your first window and renderer
4. Working with SDL_gpu
5. Implementing basic shader programming

Getting Started

Before writing code, ensure you have CMake, a C compiler (MSVS on Windows, gcc/clang on Linux/macOS), and a shader compiler appropriate for your platform.

Installation on Linux

```
git clone https://github.com/libsdl-org/SDL &&cd SDL
mkdir build &&cd build
cmake -DCMAKE_BUILD_TYPE=Release -G Ninja ..
sudo ninja install
```

The shared library and headers are installed in /usr/local/lib64 and /usr/local/include respectively by default, but you can change it if you want.

Installation on Windows

Make sure Visual Studio is installed. The repository comes with a Visual Studio solution. Just open and build it in Release mode. Copy the resulting lib and include it in a valid destination, then link these libs.

Test Program

Let's write a quick test program to check if everything works by creating a window and a renderer. This simple program listens for events—such as the window being closed (SDL_EVENT_QUIT)—and continuously renders frames in the while loop.

```
#include <SDL3/SDL.h>
#include <stdbool.h>
static SDL_Window *window = NULL;
static SDL_Renderer *renderer = NULL;
intmain()
{
        if (!SDL_Init(SDL_INIT_VIDEO))
        {
        return-1;
        }

        if (!SDL_CreateWindowAndRenderer("Test", 640, 480, 0, &window, &renderer))
        {
        return-1;
        }
        bool quit = false;
        while (!quit)
        {
        SDL_Event event;
        while (SDL_PollEvent(&event))
        {
        if (event.type == SDL_EVENT_QUIT)
```

```
        {
        quit = true;
        }
        SDL_RenderClear(renderer);
        SDL_RenderPresent(renderer);
        }
        }
        SDL_DestroyRenderer(renderer);
        SDL_DestroyWindow(window);
        SDL_Quit();
        return0;
}
```

On Linux, you can compile it by running

```
gcc -o test test.c -I/usr/local/include -L/usr/local/lib64 -lSDL3
```

On running this program, you will see an empty window.
In the next section, we'll start exploring SDL_gpu.

SDL_gpu

Let's begin by defining a state structure, which will contain all the required components. To start, we need a window and a GPU device.

```
typedefstruct
{
       SDL_Window* window;
       SDL_GPUDevice* device;
} GameState;
```

We'll start by initializing all the components required to set up the window and the render pipeline.

Init Function

```
static inlineint
Init(GameState* state)
{
       if (!SDL_Init(SDL_INIT_VIDEO))
       {
           SDL_Log("SDL_Init Failed");
       return-1;
       }
     // Initialize here
       return0;
}
```

The first step is to initialize the SDL_GPUDevice (device) and the SDL_Window. We'll tell the GPU device to use the window.

```
// Inside the init procedure
       ...
       state->device = SDL_CreateGPUDevice(SDL_GPU_SHADERFORMAT_SPIRV, true, 0);
       if (state->device == NULL)
       {
```

```
        SDL_Log("GPU device creation failed: %s", SDL_GetError());
        return-1;
        }

        SDL_Log("Internally using: %s", SDL_GetGPUDeviceDriver(state->device));

        state->window = SDL_CreateWindow("Test GPU", 640, 480, 0);
        if (state->window == NULL)
        {
        SDL_Log("Window creation failed: %s", SDL_GetError());
        return-1;
        }

        if (!SDL_ClaimWindowForGPUDevice(state->device, state->window))
        {
        SDL_Log("Graphics device cannot use the window: %s", SDL_GetError());
        return-1;
        }
        ...
        return0;
```

Other shader formats that can be used are

- SDL3/SDL_GPU_SHADERFORMAT_DXIL (For Microsoft Windows devices)
- SDL3/SDL_GPU_SHADERFORMAT_INVALID
- SDL3/SDL_GPU_SHADERFORMAT_METALLIB (For Apple devices)
- SDL3/SDL_GPU_SHADERFORMAT_MSL
- SDL3/SDL_GPU_SHADERFORMAT_PRIVATE

After initializing the window and the device, we can start writing the function that gets called every frame. This function acquires the command buffer and the swapchain from the device we created earlier.

We use Render(), which should be called every cycle. In this function, we initialize the command buffer, the surface texture, and the render pass.

Rendering can happen to any defined texture (render target) or directly to the swapchain, which is a special texture that represents the window contents.

```
INTERNAL int
Render(GameState *state)
{
        SDL_GPUCommandBuffer *cmdBuffer = SDL_AcquireGPUCommandBuffer(state->device);
        if (cmdBuffer == NULL)
        {
        SDL_Log("Where is the commandbuffer? %s", SDL_GetError());
        return-1;
        }

        SDL_GPUTexture *scTexture;
        if (!SDL_AcquireGPUSwapchainTexture(cmdBuffer, state->window, &scTexture, 0, 0))
        {
        SDL_Log("Cannot aquire gpu swapchain texture %s", SDL_GetError());
        return-1;
        }
        …
```

If everything works correctly and we have a swapchain texture ready, we can define our render pass.

A render pass needs a command buffer for all the draw commands we send to the GPU, along with

information about where to draw—in the form of a color target.

```
if (scTexture != NULL)
{
// set rendertarget colour
    SDL_GPUColorTargetInfo colorTargetInfo =
    {
        .texture =  scTexture,
        .clear_color = (SDL_FColor){0.2f, 0.2f, 0.1f, 1.0f},
        .load_op = SDL_GPU_LOADOP_CLEAR,
        .store_op = SDL_GPU_STOREOP_STORE,
    };

    SDL_GPURenderPass *renderPass = SDL_BeginGPURenderPass(cmdBuffer, &colorTargetInfo, 1,
0);
       {
         // DRAW COMMANDS HERE
       }
       SDL_EndGPURenderPass(renderPass);
}
SDL_SubmitGPUCommandBuffer(cmdBuffer);
```

Once that is set up, it's good practice to clean up all the elements we had to create.

```
INTERNAL void
Destroy(GameState *state)
{
       SDL_ReleaseGPUGraphicsPipeline(state->device, state->pipeline);
       SDL_ReleaseWindowFromGPUDevice(state->device, state->window);
       SDL_DestroyWindow(state->window);
       SDL_DestroyGPUDevice(state->device);
}
```

Now all we need to do is write the main application loop. It contains a while loop and an event handler. We call Render() inside the loop.

```
intmain()
{
       GameState state = {0};
       if (Init(&state) <0)
       {
           SDL_Log("SDL+GPU Init failed: %s", SDL_GetError());
       }

       bool quit = false;
       while (!quit)
       {
           SDL_Event event;
       while (SDL_PollEvent(&event))
           {
       if (event.type == SDL_EVENT_QUIT)
         {
       quit = true;
         }
         Render(&state);
           }
       }
```

```
        SDL_Quit();
        return0;
}
```

This gives us the bare-bones window with a clear screen.

Now it's time to let the GPU know about some vertices and draw a triangle. Before that, we need to understand shaders. Shaders are small programs that run on the GPU. We compile them into shader bytecode like SPIR-V or DXIL and then point SDL3 to the compiled file. For this tutorial, we use GLSL—but any language that compiles to the supported formats works. SDL3 supports all major shader formats, including SPIR-V, DXIL, and MSL. This gives us the flexibility to explore advanced GPU languages like Slang later on.

We'll define vertex positions and output color based on the vertex index. A triangle has three indices. The vertex shader checks the index and sets the position and color at runtime. Here's what it looks like:

```
#version 460
layout(location = 0) outvec4 outColor;

void main() {
        vec2 pos;
        if (gl_VertexIndex == 0)
        {
          pos = vec2(-0.5, -0.5);
          outColor = vec4(0.5, 0.5, 0, 1);
        }
        elseif (gl_VertexIndex == 1)
        {
          pos = vec2(0.5, -0.5);
          outColor = vec4(0, 0.5, 0.5, 1);
        }
        elseif (gl_VertexIndex == 2)
        {
          pos = vec2(0.5, 0.5);
          outColor = vec4(0.5, 0, 0.5, 1);
        }
        gl_Position = vec4(pos, 0, 1);
}
```

The fragment (or pixel) shader takes the outColor passed from the vertex shader.

```
#version 460

layout(location = 0) invec4 Color;
layout(location = 0) outvec4 FragColor;

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

We compile the shaders into SPIR-V bytecode using the glslang compiler.

```
glslang tris.vert -S vert --target-env vulkan1.3 -V -o ../spirv/tris.v.spv

glslang tris.frag --target-env vulkan1.3 -V -o ../spirv/tris.p.spv
```

The new SPIR-V files will be created in ./spirv/.

To load the shader code, write a helper function that defines key shader compilation stage characteristics and GPU memory details. Normally, you'd bind a vertex buffer and send vertices to it. But here, since vertex positions are set dynamically in the vertex shader, buffer management can be skipped for now—all related values remain zero.

```
SDL_GPUShader *
CreateShaderModule(SDL_GPUDevice *device, SDL_GPUShaderStage stage, char *byteCodePath,
constchar *entryPoint)
{
        size_t byteCodeSize;
        void *byteCode = SDL_LoadFile(byteCodePath, &byteCodeSize);
        if (byteCode == NULL)
        {
        SDL_Log("Shader loading failed. Check if the shader bytecode is on the correct path
%s", byteCodePath);
        returnNULL;
        }

        SDL_GPUShaderCreateInfo shaderCreateInfo =
        {
        .code = byteCode,
        .code_size = byteCodeSize,
        .entrypoint = entryPoint,
        .format = SDL_GPU_SHADERFORMAT_SPIRV, // hehe shortcut
        .stage = stage,
        .num_samplers = 0,
        .num_uniform_buffers = 0,
        .num_storage_buffers = 0,
        .num_storage_textures = 0, // leaving all of these as 0, later may need to add them
as args
        };

        SDL_GPUShader *shader = SDL_CreateGPUShader(device, &shaderCreateInfo);
        if (shader == NULL)
        {
        SDL_Log("Failed to create shader: %s", SDL_GetError());
        returnNULL;
        }
        return shader;
}
```

You'll also need to describe a pipeline made up of the shaders. To do this, add a pipeline object to the GameState struct.

```
SDL_GPUDevice *device;
        SDL_GPUGraphicsPipeline *pipeline;
} GameState;
```

You'll set this pipeline using a helper function. In it, define shader details and how the GPU should interpret the program. This includes specifying color targets, texture formats, primitive type, and how vertices are shaded.

```
void
SetPipeline(GameState *state, SDL_GPUPrimitiveType primitiveType, SDL_GPUShader *vs, SDL_
GPUShader *ps, SDL_GPUFillMode fillMode)
{
        SDL_GPUGraphicsPipelineCreateInfo pipelineCreateInfo =
        {
        .target_info =
        {
        .num_color_targets = 1,
        .color_target_descriptions = (SDL_GPUColorTargetDescription[])
        {
        {
        .format = SDL_GetGPUSwapchainTextureFormat(state->device, state->window),
        }
        },
        },
        .primitive_type = primitiveType,
        .vertex_shader = vs,
        .fragment_shader = ps,
        .rasterizer_state =
        {
        .fill_mode = fillMode
        }
        };

    state->pipeline = SDL_CreateGPUGraphicsPipeline(state->device, &pipelineCreateInfo);
}
```

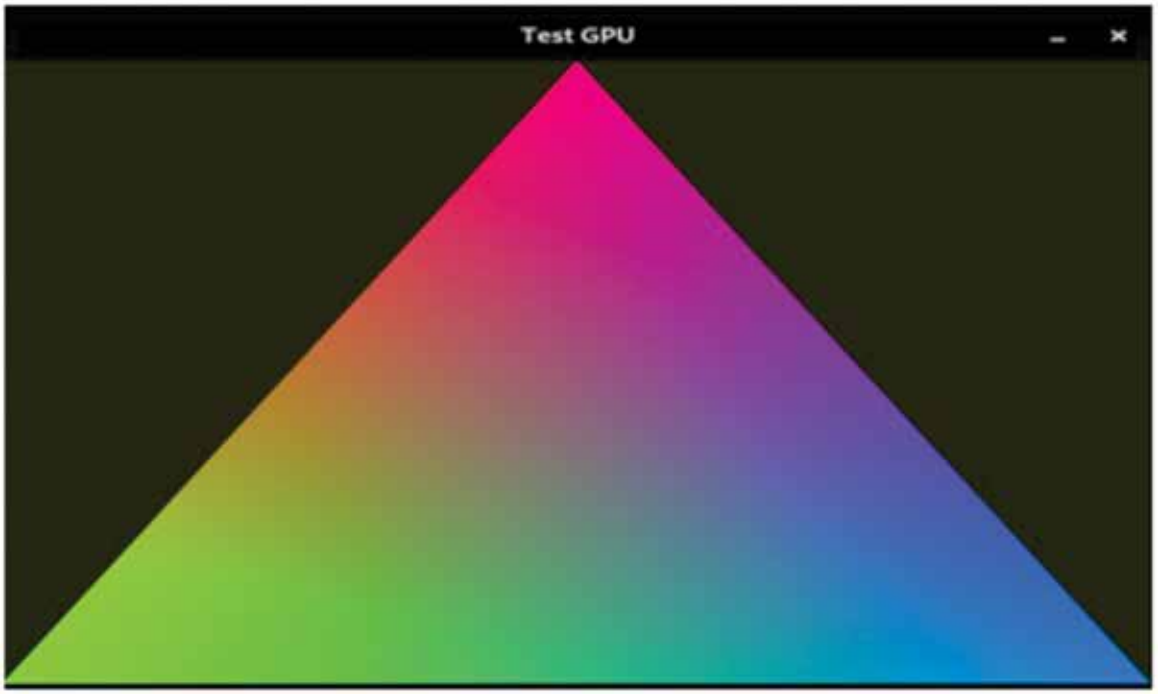
Finally, call all these during initialization. In the Init function, add the following. This step tells the GPU how the drawing should happen.

```
        SDL_GPUShader *vertexShader = CreateShaderModule(state->device, SDL_GPU_SHADERSTAGE_
VERTEX, "spirv/tris.v.spv", "main");
        if (vertexShader == NULL)
        {
        SDL_Log("Failed to create vertex shader :(");
        return-1;
        }

        SDL_GPUShader *pixelShader = CreateShaderModule(state->device, SDL_GPU_SHADERSTAGE_
FRAGMENT, "spirv/tris.p.spv", "main");
        if (vertexShader == NULL)
        {
        SDL_Log("Failed to create pixel shader :(");
        return-1;
        }

        SetPipeline(state, SDL_GPU_PRIMITIVETYPE_TRIANGLESTRIP, vertexShader, pixelShader,
SDL_GPU_FILLMODE_FILL); // Want simple rasterization
        assert(state->pipeline &&"The pipeline is null");

        SDL_ReleaseGPUShader(state->device, vertexShader);
        SDL_ReleaseGPUShader(state->device, pixelShader);
```

Now it's just a matter of adding draw commands. We're drawing a triangle with 3 vertices. Add this inside the RenderPass block in the Render function.

```
        SDL_GPURenderPass *renderPass = SDL_BeginGPURenderPass(cmdBuffer, &colorTargetInfo,
1, 0);
        {
        SDL_BindGPUGraphicsPipeline(renderPass, state->pipeline);
        SDL_DrawGPUPrimitives(renderPass, 3, 1, 0, 0);
        }
        SDL_EndGPURenderPass(renderPass);
```

Et voila!



Conclusion

SDL_GPU offers a powerful, approachable way to write modern graphics code. By abstracting platform differences, it lets you focus on visuals—not API headaches.

This guide covered:

- Window setup
- Shader creation
- Pipeline setup
- Rendering a triangle

Next steps:

1. Add textures
2. Implement complex shaders
3. Explore 3D scenes and input handling
4. Experiment with differentiable shaders

With this foundation, you're ready to build deeper, more dynamic graphics applications using SDL_GPU.

Note:This tutorial uses features from the latest SDL3 version. Make sure you're using an up-to-date release with SDL_GPU support. ■

The author is Sr. Software Engineer