Modern computer graphics introduction: Drawing a triangle

Introduction

Drawing a triangle is the first operation to check a graphics pipeline is working. We will use this lesson to learn about the fundamental concepts of modern GPU APIs.

Shaders and graphics pipeline

Shaders and graphics pipeline are represented modern graphics APIs.

Shaders are, as usual, small programs that run on the GPU. You would find the usual shaders: vertex, fragment, geometry, tesselation and compute.

Pipelines are a specific way to draw - the goal is to optimize rendering by drawing similar objects together. Pipelines need to be configured, then will be used to draw objects through the command buffer.

We will add them to our scene:

```
Scene02Triangle.hpp

#ifndef SCENE02TRIANGLE_HPP
#define SCENE02TRIANGLE_HPP

#include <SDL3/SDL_gpu.h>
#include "Scene.hpp"

class Scene02Triangle : 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;
    const char* basePath;
    SDL_GPUShader* vertexShader;
    SDL_GPUShader* fragmentShader;
    SDL_GPUGraphicsPipeline* fillPipeline;
    SDL_GPUGraphicsPipeline* linePipeline;
    SDL_GPUViewport smallViewport = {160, 120, 320, 240, 0.1f, 1.0f };
    SDL_Rect scissorRect = {320, 240, 320, 240 };

    bool useWireframeMode = false;
    bool useSmallViewport = false;
    bool useScissorRect = false;
};
```

In this specific case, we will use a vertex and a fragment shader. We will also use two pipelines: one for filling the triangle and another for drawing the triangle’s edges. We will also use a small viewport and a scissor rectangle, as options and to introduce those concepts.

First, we need to enable shader loading in the renderer:

Renderer.hpp

```
        SDL_GPUShader* LoadShader(
        const char* basePath,
        const char* shaderFilename,
        Uint32 samplerCount,
        Uint32 uniformBufferCount,
        Uint32 storageBufferCount,
        Uint32 storageTextureCount
    );
    void ReleaseShader(SDL_GPUShader* shader) const;
```

Renderer.cpp

```
SDL_GPUShader* Renderer::LoadShader(
    const char* basePath,
    const char* shaderFilename,
    Uint32 samplerCount,
    Uint32 uniformBufferCount,
    Uint32 storageBufferCount,
    Uint32 storageTextureCount
) {
    // Auto-detect the shader stage from the file name for convenience
    SDL_GPUShaderStage stage;
    if (SDL_strstr(shaderFilename, ".vert")) {
        stage = SDL_GPU_SHADERSTAGE_VERTEX;
    }
    else if (SDL_strstr(shaderFilename, ".frag")) {
        stage = SDL_GPU_SHADERSTAGE_FRAGMENT;
    }
    else {
        SDL_Log("Invalid shader stage!");
        return nullptr;
    }

    char fullPath[256];
    SDL_GPUShaderFormat backendFormats = SDL_GetGPUShaderFormats(device);
    SDL_GPUShaderFormat format = SDL_GPU_SHADERFORMAT_INVALID;
    const char *entrypoint;

    if (backendFormats & SDL_GPU_SHADERFORMAT_SPIRV) {
        SDL_snprintf(fullPath, sizeof(fullPath),
"%sContent/Shaders/Compiled/SPIRV/%s.spv", basePath, shaderFilename);
        format = SDL_GPU_SHADERFORMAT_SPIRV;
        entrypoint = "main";
    } else if (backendFormats & SDL_GPU_SHADERFORMAT_MSL) {
        SDL_snprintf(fullPath, sizeof(fullPath), "%sContent/Shaders/Compiled/MSL/%s.msl",
basePath, shaderFilename);
        format = SDL_GPU_SHADERFORMAT_MSL;
        entrypoint = "main0";
    } else if (backendFormats & SDL_GPU_SHADERFORMAT_DXIL) {
        SDL_snprintf(fullPath, sizeof(fullPath),
"%sContent/Shaders/Compiled/DXIL/%s.dxil", basePath, shaderFilename);
        format = SDL_GPU_SHADERFORMAT_DXIL;
        entrypoint = "main";
    } else {
        SDL_Log("%s", "Unrecognized backend shader format!");
        return nullptr;
    }

    size_t codeSize;
    void* code = SDL_LoadFile(fullPath, &codeSize);
    if (code == nullptr) {
        SDL_Log("Failed to load shader from disk! %s", fullPath);
```

```
        return nullptr;
    }

    SDL_GPUShaderCreateInfo shaderInfo = {
        .code_size = codeSize,
        .code = static_cast<Uint8*>(code),
        .entrypoint = entrypoint,
        .format = format,
        .stage = stage,
        .num_samplers = samplerCount,
        .num_storage_textures = storageTextureCount,
        .num_storage_buffers = storageBufferCount,
        .num_uniform_buffers = uniformBufferCount
    };
    SDL_GPUShader* shader = SDL_CreateGPUShader(device, &shaderInfo);
    if (shader == nullptr) {
        SDL_Log("Failed to create shader!");
        SDL_free(code);
        return nullptr;
    }

    SDL_free(code);
    return shader;
}

void Renderer::ReleaseShader(SDL_GPUShader *shader) const {
    SDL_ReleaseGPUShader(device, shader);
}
```

The LoadShader function serves principally to load any kind of compiled shader. We will see how the different parameters of the function are used throughout this course.

Then we need to create the two pipelines, in the scene's Load function. The shaders will be used in the pipelines' setups.

Scene02Triangle.cpp

```
void Scene02Triangle::Load(Renderer& renderer) {
    basePath = SDL_GetBasePath();
    vertexShader = renderer.LoadShader(basePath, "RawTriangle.vert", 0, 0, 0, 0);
    fragmentShader = renderer.LoadShader(basePath, "SolidColor.frag", 0, 0, 0, 0);

    // Create the pipelines
    SDL_GPUGraphicsPipelineCreateInfo pipelineCreateInfo = {
        .vertex_shader = vertexShader,
        .fragment_shader = fragmentShader,
        .primitive_type = SDL_GPU_PRIMITIVETYPE_TRIANGLELIST,
        .target_info = {
            .color_target_descriptions = new SDL_GPUColorTargetDescription[1] {{
                .format = SDL_GetGPUSwapchainTextureFormat(renderer.device,
renderer.renderWindow)
            }},
            .num_color_targets = 1,
        },

    };

    pipelineCreateInfo.rasterizer_state.fill_mode = SDL_GPU_FILLMODE_FILL;
    fillPipeline =  renderer.CreateGPUGraphicsPipeline(pipelineCreateInfo);
    if (fillPipeline == nullptr)
    {
        SDL_Log("Failed to create fill pipeline!");
    }
```

```
    pipelineCreateInfo.rasterizer_state.fill_mode = SDL_GPU_FILLMODE_LINE;
    linePipeline =  renderer.CreateGPUGraphicsPipeline(pipelineCreateInfo);
    if (linePipeline == nullptr)
    {
        SDL_Log("Failed to create line pipeline!");
    }

    // Clean up shader resources
    renderer.ReleaseShader(vertexShader);
    renderer.ReleaseShader(fragmentShader);

    // Finally, print instructions!
    SDL_Log("Press Left to toggle wireframe mode");
    SDL_Log("Press Down to toggle small viewport");
    SDL_Log("Press Right to toggle scissor rect");
}
```

A pipeline setup mainly consist in setting up all the pipeline create info, then running the pipeline creation function. In this case, it is a very simple call to an SDL function.

Renderer.cpp

```
SDL_GPUGraphicsPipeline *Renderer::CreateGPUGraphicsPipeline(const
SDL_GPUGraphicsPipelineCreateInfo &createInfo) const {
    return SDL_CreateGPUGraphicsPipeline(device, &createInfo);
}
```

The update function is just to toggle the options: wireframe mode, small viewport and scissor rectangle.

Scene02Triangle.cpp

```
bool Scene02Triangle::Update(float dt) {
    const bool isRunning = ManageInput(inputState);

    if (inputState.IsPressed(DirectionalKey::Left)) {
        useWireframeMode = !useWireframeMode;
    }
    if (inputState.IsPressed(DirectionalKey::Up)) {
        useSmallViewport = !useSmallViewport;
    }
    if (inputState.IsPressed(DirectionalKey::Right)) {
        useScissorRect = !useScissorRect;
    }

    return isRunning;
}
```

The draw function is where we will use the pipelines. We will also use the small viewport and scissor rectangle, if the options are enabled.

Scene02Triangle.cpp

```
void Scene02Triangle::Draw(Renderer& renderer) {
    renderer.Begin();

    renderer.BindGraphicsPipeline(useWireframeMode ? linePipeline : fillPipeline);
    if (useSmallViewport) {
        renderer.SetViewport(smallViewport);
    }
    if (useScissorRect) {
        renderer.SetScissorRect(scissorRect);
    }
    renderer.DrawPrimitives(3, 1, 0, 0);
```

```
    renderer.End();
}
```

In a similar way as with OpenGL, we need to bind the pipeline before we draw the primitives. The draw primitives function is a simple call to an SDL function.

Renderer.cpp

```
void Renderer::BindGraphicsPipeline(SDL_GPUGraphicsPipeline* pipeline) const {
    SDL_BindGPUGraphicsPipeline(renderPass, pipeline);
}

void Renderer::DrawPrimitives(int numVertices, int numInstances, int firstVertex, int
firstInstance) const {
    SDL_DrawGPUPrimitives(renderPass, numVertices, numInstances, firstVertex,
firstInstance);
}

void Renderer::SetViewport(const SDL_GPUViewport& viewport) const {
    SDL_SetGPUViewport(renderPass, &viewport);
}
void Renderer::SetScissorRect(const SDL_Rect& rect) const {
    SDL_SetGPUScissor(renderPass, &rect);
}
```

Finally, we need to release the pipelines in the scene's Unload function.

Scene02Triangle.cpp

```
void Scene02Triangle::Unload(Renderer& renderer) {
    renderer.ReleaseGraphicsPipeline(fillPipeline);
    renderer.ReleaseGraphicsPipeline(linePipeline);
}
```

Renderer.cpp

```
void Renderer::ReleaseGraphicsPipeline(SDL_GPUGraphicsPipeline* pipeline) const {
    SDL_ReleaseGPUGraphicsPipeline(device, pipeline);
}
```

Now we have a simple triangle being drawn on the screen. Use the arrow keys to toggle the wireframe mode, small viewport and scissor rectangle.

Exercises

1. Change the triangle's color.
2. Change the triangle's position.
3. Change the viewport size.
4. Change the scissor rectangle size.