Modern computer graphics introduction: Sending data to the GPU through a vertex buffer

Introduction

Now we have drawn a triangle, we would like to set its position. As with openGL, we will use a vertex buffer to send data to the GPU.

Nevertheless, directly sending data to the GPU through a CPU-side buffer is not optimal. The recommended way is to use a specialized buffer, that will send data to GPU, where it can be read in the best possible way.

Vertex buffer

This time, our scene only contains one pipeline. We have a new vertex buffer member vartiable.

Scene03VertexBuffer.hpp

```
#ifndef SCENE03TRIANGLEVERTEXBUFFER_HPP
#define SCENE03TRIANGLEVERTEXBUFFER_HPP

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

class Scene03TriangleVertexBuffer : 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* pipeline;
    SDL_GPUBuffer* vertexBuffer;
};



#endif //SCENE03TRIANGLEVERTEXBUFFER_HPP
```

The pipeline setup is similar as before, but this time we need to setup the vertex buffer attributes.

Scene03VertexBuffer.cpp

```
void Scene03TriangleVertexBuffer::Load(Renderer& renderer) {
    basePath = SDL_GetBasePath();
    vertexShader = renderer.LoadShader(basePath, "PositionColor.vert", 0, 0, 0, 0);
    fragmentShader = renderer.LoadShader(basePath, "SolidColor.frag", 0, 0, 0, 0);

    // Create the pipeline
```

```
    SDL_GPUGraphicsPipelineCreateInfo pipelineCreateInfo = {
        .vertex_shader = vertexShader,
        .fragment_shader = fragmentShader,
        // This is set up to match the vertex shader layout!
        .vertex_input_state = SDL_GPUVertexInputState {
            .vertex_buffer_descriptions = new SDL_GPUVertexBufferDescription[1] {{
                .slot = 0,
                .pitch = sizeof(PositionColorVertex),
                .input_rate = SDL_GPU_VERTEXINPUTRATE_VERTEX,
                .instance_step_rate = 0,
            }},
            .num_vertex_buffers = 1,
            .vertex_attributes = new SDL_GPUVertexAttribute[2] {{
                .location = 0,
                .buffer_slot = 0,
                .format = SDL_GPU_VERTEXELEMENTFORMAT_FLOAT3,
                .offset = 0
            }, {
                .location = 1,
                .buffer_slot = 0,
                .format = SDL_GPU_VERTEXELEMENTFORMAT_UBYTE4_NORM,
                .offset = sizeof(float) * 3
            }},
            .num_vertex_attributes = 2,
        },
        .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,
        },
    };

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

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

As you can see, the pipeline’s vertex buffer has a description and attributes. The description is deduces from the vertex data we will send:

PositionColorVertex.hpp

```
#ifndef GRAPHICS_SDL3_POSITIONCOLORVERTEX_HPP
#define GRAPHICS_SDL3_POSITIONCOLORVERTEX_HPP

#include <SDL3/SDL_stdinc.h>

typedef struct PositionColorVertex
{
    float x, y, z;
    Uint8 r, g, b, a;
} PositionColorVertex;

#endif //GRAPHICS_SDL3_POSITIONCOLORVERTEX_HPP
```

There are 3 floats and 4 unsigned bytes. The attributes are set accordingly.

The vertex buffer is created, then we start to create the transfer buffer to ensure optimal upload to the GPU.

Scene03VertexBuffer.cpp

```
...
    // Create the vertex buffer
    SDL_GPUBufferCreateInfo vertexBufferCreateInfo = {
        .usage = SDL_GPU_BUFFERUSAGE_VERTEX,
        .size = sizeof(PositionColorVertex) * 3
    };
    vertexBuffer = renderer.CreateBuffer(vertexBufferCreateInfo);

    // To get data into the vertex buffer, we have to use a transfer buffer
    SDL_GPUTransferBufferCreateInfo transferBufferCreateInfo = {
        .usage = SDL_GPU_TRANSFERBUFFERUSAGE_UPLOAD,
        .size = sizeof(PositionColorVertex) * 3
    };
    SDL_GPUTransferBuffer* transferBuffer =
renderer.CreateTransferBuffer(transferBufferCreateInfo);

    // Map the transfer buffer and fill it with data (data is bound to the transfer
buffer)
    auto* transferData = static_cast<PositionColorVertex *>(
        renderer.MapTransferBuffer(transferBuffer, false)
    );
    transferData[0] = PositionColorVertex {  -0.5,  -0.5, 0, 255,   0,   0, 255 };
    transferData[1] = PositionColorVertex {   0.5,  -0.5, 0,   0, 255,   0, 255 };
    transferData[2] = PositionColorVertex {     0,   0.5, 0,   0,   0, 255, 255 };
    renderer.UnmapTransferBuffer(transferBuffer);

    // Upload the transfer data to the vertex buffer
    SDL_GPUTransferBufferLocation transferBufferLocation = {
        .transfer_buffer = transferBuffer,
        .offset = 0
    };
    SDL_GPUBufferRegion vertexBufferRegion = {
        .buffer = vertexBuffer,
        .offset = 0,
        .size = sizeof(PositionColorVertex) * 3
    };

    renderer.BeginUploadToBuffer();
    renderer.UploadToBuffer(transferBufferLocation, vertexBufferRegion, false);
    renderer.EndUploadToBuffer(transferBuffer);
}
```

The vertex buffer will contain 3 vertices, each with a position and a color.

We create a transfer buffer that will contain this data. We map the data so that when we modify it, it modifies the transfer buffer. We fill the transfer buffer with the data, then we unmap it.

To upload the data, we need to tell which transfer buffer we will use and where we want to data to land (buffer region). In this case, it will be at the beginning of the vertex buffer.

Then we can start the upload, upload the data from the transfer buffer location to the vertex buffer region, and end the upload.

We need to update the Renderer to support all those new functions. We also need to add two members to the Renderer class.

Renderer.hpp

```
    SDL_GPUCommandBuffer* uploadCmdBuf {nullptr};
    SDL_GPUCopyPass* copyPass {nullptr};
```

The uploadCmdBuffer will be used specifically to send command for uploading data to the GPU, and the copyPass will be used to copy data from the transfer buffer to the vertex buffer.

Renderer.cpp

```
SDL_GPUBuffer* Renderer::CreateBuffer(const SDL_GPUBufferCreateInfo &createInfo) const {
    return SDL_CreateGPUBuffer(device, &createInfo);
}

void Renderer::BindVertexBuffers(Uint32 firstSlot, const SDL_GPUBufferBinding &bindings,
Uint32 numBindings) const {
    SDL_BindGPUVertexBuffers(renderPass, firstSlot, &bindings, numBindings);
}

void Renderer::ReleaseBuffer(SDL_GPUBuffer* buffer) const {
    SDL_ReleaseGPUBuffer(device, buffer);
}

SDL_GPUTransferBuffer *Renderer::CreateTransferBuffer(const
SDL_GPUTransferBufferCreateInfo &createInfo) const {
    return SDL_CreateGPUTransferBuffer(device, &createInfo);
}

void *Renderer::MapTransferBuffer(SDL_GPUTransferBuffer *transferBuffer, bool cycle) const
{
    return SDL_MapGPUTransferBuffer(device, transferBuffer, cycle);
}

void Renderer::UnmapTransferBuffer(SDL_GPUTransferBuffer *transferBuffer) const {
    SDL_UnmapGPUTransferBuffer(device, transferBuffer);
}

void Renderer::ReleaseTransferBuffer(SDL_GPUTransferBuffer* transferBuffer) const {
    SDL_ReleaseGPUTransferBuffer(device, transferBuffer);
}

void Renderer::BeginUploadToBuffer() {
    uploadCmdBuf = SDL_AcquireGPUCommandBuffer(device);
    copyPass = SDL_BeginGPUCopyPass(uploadCmdBuf);
}

void Renderer::UploadToBuffer(const SDL_GPUTransferBufferLocation &source,
                              const SDL_GPUBufferRegion &destination,
                              bool cycle) const {
    SDL_UploadToGPUBuffer(copyPass, &source, &destination, cycle);
}

void Renderer::EndUploadToBuffer(SDL_GPUTransferBuffer *transferBuffer) const {
    SDL_EndGPUCopyPass(copyPass);
    SDL_SubmitGPUCommandBuffer(uploadCmdBuf);
    SDL_ReleaseGPUTransferBuffer(device, transferBuffer);
}
```

By using the upload command buffer and the copy pass, we upload the data to the GPU.

The scene’s Update function is empty, and the Drw function is similar, except that we bind the vertex buffer before drawing.

Scene03VertexBuffer.cpp

```
bool Scene03TriangleVertexBuffer::Update(float dt) {
    const bool isRunning = ManageInput(inputState);
    return isRunning;
}

void Scene03TriangleVertexBuffer::Draw(Renderer& renderer) {
    renderer.Begin();

    renderer.BindGraphicsPipeline(pipeline);
    SDL_GPUBufferBinding vertexBindings = { .buffer = vertexBuffer, .offset = 0 };
    renderer.BindVertexBuffers(0, vertexBindings, 1);
    renderer.DrawPrimitives(3, 1, 0, 0);

    renderer.End();
}
```

In the Unload function, we release the vertex buffer in addition to the pipeline.

Scene03VertexBuffer.cpp

```
void Scene03TriangleVertexBuffer::Unload(Renderer& renderer) {
    renderer.ReleaseBuffer(vertexBuffer);
    renderer.ReleaseGraphicsPipeline(pipeline);
}
```

Now, you can draw a triangle with a vertex buffer to set its position.

Exercises

1. Change the triangle’s position.
2. Change the triangle’s color.
3. Create a square with a 6 elements vertex buffer.