Modern computer graphics introduction: A moving quad

Introduction

Now, we will send data to allow the quad to rotate. This data will be sent as a uniform to the shader. It will use a very simple and immediate sending style to the shader, similar to what you would expect with Push Constants in Vulkan.

We will also have a progressive color change, so that we can send data to the fragment shader too.

This lesson is thought as an introduction to data passing in modern graphics programming, so the architecture used is not at all relevant for a real game engine.

A simple matrix class

In order to be able to rotate the quad, we need to be able to create a rotation and a translation matrix. We will create a simple matrix class for this purpose.

Mat4.hpp

```
//
// Created by gaetz on 24/09/2024.
//

#ifndef MAT4_HPP
#define MAT4_HPP

class Mat4 {
public:
    Mat4();
    Mat4 (float m0_, float m4_, float m8_, float m12_,
          float m1_, float m5_, float m9_, float m13_,
          float m2_, float m6_, float m10_, float m14_,
          float m3_, float m7_, float m11_, float m15_):
            m0 { m0_ }, m4 { m4_ }, m8 { m8_ }, m12 { m12_ },
            m1 { m1_ }, m5 { m5_ }, m9 { m9_ }, m13 { m13_ },
            m2 { m2_ }, m6 { m6_ }, m10 { m10_ }, m14 { m14_ },
            m3 { m3_ }, m7 { m7_ }, m11 { m11_ }, m15 { m15_ } {}

    float m0, m4, m8, m12;  // Matrix first row (4 components)
    float m1, m5, m9, m13;  // Matrix second row (4 components)
    float m2, m6, m10, m14; // Matrix third row (4 components)
    float m3, m7, m11, m15; // Matrix fourth row (4 components)

    static const Mat4 Identity;

    static Mat4 CreateRotationMatrix(float axisX, float axisY, float axisZ, float angle);
    static Mat4 CreateRotationZ(float angle);
    static Mat4 CreateTranslation(float x, float y, float z);
    Mat4 operator*(const Mat4& other) const;
    static Mat4 CreateOrthographicOffCenter(float left, float right, float bottom, float
        top, float zNearPlane, float zFarPlane);
    static Mat4 CreatePerspectiveFieldOfView(float fieldOfView, float aspectRatio,
                                       float nearPlaneDistance, float farPlaneDistance);
};

#endif //GMATH_MAT4_HPP
```

Mat4.cpp

```cpp
//
// Created by gaetz on 24/09/2024.
//

#include "Mat4.hpp"
#ifdef __APPLE__
    #include <cmath>
#else
    #include <cstdlib>
#endif

    const Mat4 Mat4::Identity { 1.0f, 0.0f, 0.0f, 0.0f,
                                0.0f, 1.0f, 0.0f, 0.0f,
                                0.0f, 0.0f, 1.0f, 0.0f,
                                0.0f, 0.0f, 0.0f, 1.0f };


Mat4 Mat4::CreateRotationMatrix(float x, float y, float z, float angle) {
    Mat4 result {};
    float lengthSquared = x*x + y*y + z*z;

    if ((lengthSquared != 1.0f) && (lengthSquared != 0.0f))
    {
        float iLength = 1.0f / sqrtf(lengthSquared);
        x *= iLength;
        y *= iLength;
        z *= iLength;
    }

    float sinRes = sinf(angle);
    float cosRes = cosf(angle);
    float t = 1.0f - cosRes;

    result.m0 = x*x*t + cosRes;
    result.m1 = y*x*t + z * sinRes;
    result.m2 = z*x*t - y * sinRes;
    result.m3 = 0.0f;

    result.m4 = x*y*t - z * sinRes;
    result.m5 = y*y*t + cosRes;
    result.m6 = z*y*t + x * sinRes;
    result.m7 = 0.0f;

    result.m8 = x*z*t + y * sinRes;
    result.m9 = y*z*t - x * sinRes;
    result.m10 = z*z*t + cosRes;
    result.m0 = 0.0f;

    result.m4 = 0.0f;
    result.m8 = 0.0f;
    result.m12 = 0.0f;
    result.m15 = 1.0f;

    return result;
}

Mat4 Mat4::CreateRotationZ(float radians) {
        return Mat4 {
                cosf(radians), sinf(radians), 0, 0,
                -sinf(radians), cosf(radians), 0, 0,
                0,  0, 1, 0,
                0,  0, 0, 1
```

```
        };
}


Mat4 Mat4::CreateTranslation(float x, float y, float z)
{
    return Mat4 {
            1, 0, 0, 0,
            0, 1, 0, 0,
            0, 0, 1, 0,
            x, y, z, 1
    };
}

Mat4 Mat4::CreateOrthographicOffCenter(
        float left,
        float right,
        float bottom,
        float top,
        float zNearPlane,
        float zFarPlane
) {
    return Mat4 {
            2.0f / (right - left), 0, 0, 0,
            0, 2.0f / (top - bottom), 0, 0,
            0, 0, 1.0f / (zNearPlane - zFarPlane), 0,
            (left + right) / (left - right), (top + bottom) / (bottom - top), zNearPlane /
        (zNearPlane - zFarPlane), 1
    };
}

Mat4 Mat4::CreatePerspectiveFieldOfView(
        float fieldOfView,
        float aspectRatio,
        float nearPlaneDistance,
        float farPlaneDistance
) {
    float num = 1.0f / ((float) tanf(fieldOfView * 0.5f));
    return Mat4 {
            num / aspectRatio, 0, 0, 0,
            0, num, 0, 0,
            0, 0, farPlaneDistance / (nearPlaneDistance - farPlaneDistance), -1,
            0, 0, (nearPlaneDistance * farPlaneDistance) / (nearPlaneDistance -
        farPlaneDistance), 0
    };
}

Mat4 Mat4::operator*(const Mat4& other) const {
    Mat4 result;

    result.m0 = (
            (m0 * other.m0) +
            (m4 * other.m1) +
            (m8 * other.m2) +
            (m12 * other.m3)
    );
    result.m4 = (
            (m0 * other.m4) +
            (m4 * other.m5) +
            (m8 * other.m6) +
            (m12 * other.m7)
    );
    result.m8 = (
            (m0 * other.m8) +
```

```
        (m4 * other.m9) +
        (m8 * other.m10) +
        (m12 * other.m11)
);
result.m12 = (
        (m0 * other.m12) +
        (m4 * other.m13) +
        (m8 * other.m14) +
        (m12 * other.m15)
);
result.m1 = (
        (m1 * other.m0) +
        (m5 * other.m1) +
        (m9 * other.m2) +
        (m13 * other.m3)
);
result.m5 = (
        (m1 * other.m4) +
        (m5 * other.m5) +
        (m9 * other.m6) +
        (m13 * other.m7)
);
result.m9 = (
        (m1 * other.m8) +
        (m5 * other.m9) +
        (m9 * other.m10) +
        (m13 * other.m11)
);
result.m13 = (
        (m1 * other.m12) +
        (m5 * other.m13) +
        (m9 * other.m14) +
        (m13 * other.m15)
);
result.m2 = (
        (m2 * other.m0) +
        (m6 * other.m1) +
        (m10 * other.m2) +
        (m14 * other.m3)
);
result.m6 = (
        (m2 * other.m4) +
        (m6 * other.m5) +
        (m10 * other.m6) +
        (m14 * other.m7)
);
result.m10 = (
        (m2 * other.m8) +
        (m6 * other.m9) +
        (m10 * other.m10) +
        (m14 * other.m11)
);
result.m14 = (
        (m2 * other.m12) +
        (m6 * other.m13) +
        (m10 * other.m14) +
        (m14 * other.m15)
);
result.m3 = (
        (m3 * other.m0) +
        (m7 * other.m1) +
        (m11 * other.m2) +
        (m15 * other.m3)
);
```

```
    result.m7 = (
            (m3 * other.m4) +
            (m7 * other.m5) +
            (m11 * other.m6) +
            (m15 * other.m7)
    );
    result.m11 = (
            (m3 * other.m8) +
            (m7 * other.m9) +
            (m11 * other.m10) +
            (m15 * other.m11)
    );
    result.m15 = (
            (m3 * other.m12) +
            (m7 * other.m13) +
            (m11 * other.m14) +
            (m15 * other.m15)
    );

    return result;
}

Mat4::Mat4() {
    m0 = 0.0f;
    m1 = 0.0f;
    m2 = 0.0f;
    m3 = 0.0f;
    m4 = 0.0f;
    m5 = 0.0f;
    m6 = 0.0f;
    m7 = 0.0f;
    m8 = 0.0f;
    m9 = 0.0f;
    m10 = 0.0f;
    m0 = 0.0f;
    m4 = 0.0f;
    m8 = 0.0f;
    m12 = 0.0f;
    m15 = 0.0f;
}
```

Setting up the rotating quad scene

This scene is very similar to the previous textured quad. Then the code is nearly identical.

Scene08TextureQuadMoving.hpp

```
#ifndef SCENE08TEXTUREQUADMOVING_HPP
#define SCENE08TEXTUREQUADMOVING_HPP

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

using std::array;
using std::string;

typedef struct FragMultiplyUniform
{
    float r, g, b, a;
} fragMultiplyUniform;
```

```
class Scene08TextureQuadMoving : 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 {nullptr};
    SDL_GPUShader* vertexShader {nullptr};
    SDL_GPUShader* fragmentShader {nullptr};

    SDL_GPUGraphicsPipeline* pipeline {nullptr};
    SDL_GPUBuffer* vertexBuffer {nullptr};
    SDL_GPUBuffer* indexBuffer {nullptr};
    SDL_GPUTexture* texture {nullptr};
    SDL_GPUSampler* sampler {nullptr};
    float time {0};
};

#endif //SCENE08TEXTUREQUADMOVING_HPP
```

Scene08TextureQuadMoving.cpp

```
void Scene08TextureQuadMoving::Load(Renderer& renderer) {
    basePath = SDL_GetBasePath();
    vertexShader = renderer.LoadShader(basePath, "TexturedQuadWithMatrix.vert", 0, 1, 0,
        0);
    fragmentShader = renderer.LoadShader(basePath, "TexturedQuadWithMultiplyColor.frag",
        1, 1, 0, 0);

    SDL_Surface* imageData = renderer.LoadBMPImage(basePath, "ravioli.bmp", 4);
    if (imageData == nullptr) {
        SDL_Log("Could not load image data!");
    }

    // 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(PositionTextureVertex),
                .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_FLOAT2,
                .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),
           .blend_state = {
               .src_color_blendfactor = SDL_GPU_BLENDFACTOR_SRC_ALPHA,
               .dst_color_blendfactor = SDL_GPU_BLENDFACTOR_ONE_MINUS_SRC_ALPHA,
               .color_blend_op = SDL_GPU_BLENDOP_ADD,
               .src_alpha_blendfactor = SDL_GPU_BLENDFACTOR_SRC_ALPHA,
               .dst_alpha_blendfactor = SDL_GPU_BLENDFACTOR_ONE_MINUS_SRC_ALPHA,
               .alpha_blend_op = SDL_GPU_BLENDOP_ADD,
               .enable_blend = true,
           }
        }},
        .num_color_targets = 1,
    },
};
pipeline = renderer.CreateGPUGraphicsPipeline(pipelineCreateInfo);

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

// Texture sampler
sampler = renderer.CreateSampler(SDL_GPUSamplerCreateInfo {
    .min_filter = SDL_GPU_FILTER_NEAREST,
    .mag_filter = SDL_GPU_FILTER_NEAREST,
    .mipmap_mode = SDL_GPU_SAMPLERMIPMAPMODE_NEAREST,
    .address_mode_u = SDL_GPU_SAMPLERADDRESSMODE_CLAMP_TO_EDGE,
    .address_mode_v = SDL_GPU_SAMPLERADDRESSMODE_CLAMP_TO_EDGE,
    .address_mode_w = SDL_GPU_SAMPLERADDRESSMODE_CLAMP_TO_EDGE,
});


// Create the vertex buffer
SDL_GPUBufferCreateInfo vertexBufferCreateInfo = {
    .usage = SDL_GPU_BUFFERUSAGE_VERTEX,
    .size = sizeof(PositionTextureVertex) * 4
};
vertexBuffer = renderer.CreateBuffer(vertexBufferCreateInfo);
renderer.SetBufferName(vertexBuffer, "Ravioli Vertex Buffer");

// Create the index buffer
SDL_GPUBufferCreateInfo indexBufferCreateInfo = {
    .usage = SDL_GPU_BUFFERUSAGE_INDEX,
    .size = sizeof(Uint16) * 6
};
indexBuffer = renderer.CreateBuffer(indexBufferCreateInfo);

// Create texture
SDL_GPUTextureCreateInfo textureInfo {
    .type = SDL_GPU_TEXTURETYPE_2D,
    .format = SDL_GPU_TEXTUREFORMAT_R8G8B8A8_UNORM,
    .usage = SDL_GPU_TEXTUREUSAGE_SAMPLER,
    .width = static_cast<Uint32>(imageData->w),
    .height = static_cast<Uint32>(imageData->h),
    .layer_count_or_depth = 1,
    .num_levels = 1,
};
texture = renderer.CreateTexture(textureInfo);
renderer.SetTextureName(texture,"Ravioli Texture");

// Set the buffer data
```

```
SDL_GPUTransferBufferCreateInfo transferBufferCreateInfo = {
    .usage = SDL_GPU_TRANSFERBUFFERUSAGE_UPLOAD,
    .size = (sizeof(PositionTextureVertex) * 4) + (sizeof(Uint16) * 6),
};
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<PositionTextureVertex *>(
    renderer.MapTransferBuffer(transferBuffer, false)
);
transferData[0] = PositionTextureVertex { -0.5f, -0.5f, 0, 0, 0 };
transferData[1] = PositionTextureVertex {  0.5f, -0.5f, 0, 1, 0 };
transferData[2] = PositionTextureVertex {  0.5f,  0.5f, 0, 1, 1 };
transferData[3] = PositionTextureVertex { -0.5f,  0.5f, 0, 0, 1 };
auto indexData = reinterpret_cast<Uint16*>(&transferData[4]);
indexData[0] = 0;
indexData[1] = 1;
indexData[2] = 2;
indexData[3] = 0;
indexData[4] = 2;
indexData[5] = 3;
renderer.UnmapTransferBuffer(transferBuffer);

// Setup texture transfer buffer
Uint32 bufferSize = imageData->w * imageData->h * 4;
SDL_GPUTransferBufferCreateInfo textureTransferBufferCreateInfo {
    .usage = SDL_GPU_TRANSFERBUFFERUSAGE_UPLOAD,
    .size = bufferSize
};
SDL_GPUTransferBuffer* textureTransferBuffer =
    renderer.CreateTransferBuffer(textureTransferBufferCreateInfo);
auto textureTransferData = static_cast<PositionTextureVertex *>(
    renderer.MapTransferBuffer(textureTransferBuffer, false)
);
std::memcpy(textureTransferData, imageData->pixels, bufferSize);
renderer.UnmapTransferBuffer(textureTransferBuffer);


renderer.BeginUploadToBuffer();
// Upload the transfer data to the vertex and index buffer
SDL_GPUTransferBufferLocation transferVertexBufferLocation {
    .transfer_buffer = transferBuffer,
    .offset = 0
};
SDL_GPUBufferRegion vertexBufferRegion {
    .buffer = vertexBuffer,
    .offset = 0,
    .size = sizeof(PositionTextureVertex) * 4
};
SDL_GPUTransferBufferLocation transferIndexBufferLocation {
    .transfer_buffer = transferBuffer,
    .offset = sizeof(PositionTextureVertex) * 4
};
SDL_GPUBufferRegion indexBufferRegion {
    .buffer = indexBuffer,
    .offset = 0,
    .size = sizeof(Uint16) * 6
};
SDL_GPUTextureTransferInfo textureBufferLocation {
    .transfer_buffer = textureTransferBuffer,
    .offset = 0
};
```

```
    SDL_GPUTextureRegion textureBufferRegion {
        .texture = texture,
        .w = static_cast<Uint32>(imageData->w),
        .h = static_cast<Uint32>(imageData->h),
        .d = 1
    };

    renderer.UploadToBuffer(transferVertexBufferLocation, vertexBufferRegion, false);
    renderer.UploadToBuffer(transferIndexBufferLocation, indexBufferRegion, false);
    renderer.UploadToTexture(textureBufferLocation, textureBufferRegion, false);
    renderer.EndUploadToBuffer(transferBuffer);
    renderer.ReleaseTransferBuffer(textureTransferBuffer);
    renderer.ReleaseSurface(imageData);
}
```

The Update function will just update the time variable.

```
bool Scene08TextureQuadMoving::Update(float dt) {
    const bool isRunning = ManageInput(inputState);
    time += dt;

    return isRunning;
}
```

The Unload function will release all the resources.

```
void Scene08TextureQuadMoving::Unload(Renderer& renderer) {
    renderer.ReleaseSampler(sampler);
    renderer.ReleaseBuffer(vertexBuffer);
    renderer.ReleaseBuffer(indexBuffer);
    renderer.ReleaseTexture(texture);
    renderer.ReleaseGraphicsPipeline(pipeline);
}
```

Computing and sending the matrix to the shader

The Draw function will compute the transform matrix and the color, then send them respectively to the vertex and fragment shaders.

By the way, we will dray 4 rotating quads, in order to show different colors and rotations.

```
void Scene08TextureQuadMoving::Draw(Renderer& renderer) {
    renderer.Begin();

    renderer.BindGraphicsPipeline(pipeline);
    SDL_GPUBufferBinding vertexBindings { .buffer = vertexBuffer, .offset = 0 };
    renderer.BindVertexBuffers(0, vertexBindings, 1);
    SDL_GPUBufferBinding indexBindings { .buffer = indexBuffer, .offset = 0 };
    renderer.BindIndexBuffer(indexBindings, SDL_GPU_INDEXELEMENTSIZE_16BIT);
    SDL_GPUTextureSamplerBinding textureSamplerBinding { .texture = texture, .sampler =
        sampler };
    renderer.BindFragmentSamplers(0, textureSamplerBinding, 1);


    // Top-left
    Mat4 matrixUniform =
            Mat4::CreateRotationZ(time) *
            Mat4::CreateTranslation(-0.5f, -0.5f, 0);
    renderer.PushVertexUniformData(0, &matrixUniform, sizeof(matrixUniform));
    FragMultiplyUniform fragMultiplyUniform0 { 1.0f, 0.5f + SDL_sinf(time) * 0.5f, 1.0f,
        1.0f };
    renderer.PushFragmentUniformData(0, &fragMultiplyUniform0,
        sizeof(FragMultiplyUniform));
```

```
    renderer.DrawIndexedPrimitives(6, 1, 0, 0, 0);

    // Top-right
    matrixUniform =
            Mat4::CreateRotationZ((2.0f * SDL_PI_F) - time) *
            Mat4::CreateTranslation(0.5f, -0.5f, 0);
    renderer.PushVertexUniformData(0, &matrixUniform, sizeof(matrixUniform));
    FragMultiplyUniform fragMultiplyUniform1 { 1.0f, 0.5f + SDL_cosf(time) * 0.5f, 1.0f,
        1.0f };
    renderer.PushFragmentUniformData(0, &fragMultiplyUniform1,
        sizeof(FragMultiplyUniform));
    renderer.DrawIndexedPrimitives(6, 1, 0, 0, 0);

    // Bottom-left
    matrixUniform =
            Mat4::CreateRotationZ(time) *
            Mat4::CreateTranslation(-0.5f, 0.5f, 0);
    renderer.PushVertexUniformData(0, &matrixUniform, sizeof(matrixUniform));
    FragMultiplyUniform fragMultiplyUniform2 { 1.0f, 0.5f + SDL_sinf(time) * 0.2f, 1.0f,
        1.0f };
    renderer.PushFragmentUniformData(0, &fragMultiplyUniform2,
        sizeof(FragMultiplyUniform));
    renderer.DrawIndexedPrimitives(6, 1, 0, 0, 0);

    // Bottom-right
    matrixUniform =
            Mat4::CreateRotationZ(time) *
            Mat4::CreateTranslation(0.5f, 0.5f, 0);
    renderer.PushVertexUniformData(0, &matrixUniform, sizeof(matrixUniform));
    FragMultiplyUniform fragMultiplyUniform3 { 1.0f, 0.5f + SDL_cosf(time) * 1.0f, 1.0f,
        1.0f };
    renderer.PushFragmentUniformData(0, &fragMultiplyUniform3,
        sizeof(FragMultiplyUniform));
    renderer.DrawIndexedPrimitives(6, 1, 0, 0, 0);

    renderer.End();
}
```

We must add to the renderer the PushVertexUniformData and PushFragmentUniformData functions.

Renderer.cpp

```
void Renderer::PushVertexUniformData(uint32_t slot, const void* data, Uint32 size) const {
    SDL_PushGPUVertexUniformData(cmdBuffer, 0, data, size);
}

void Renderer::PushFragmentUniformData(uint32_t slot, const void* data, Uint32 size) const
        {
    SDL_PushGPUFragmentUniformData(cmdBuffer, 0, data, size);
}
```

Exercices

1. Change the rotation speed and color hues of the quads.
2. Create a Quad class in an attempt to improve this small engine architecture.
3. Create a uniform colored cube.
4. Create a rotating cube.
5. Create a rotating cube with a texture.