OpenGL

Rendering of meshes and models

Advanced Graphics Programming

VAOs, VBOs, and IBOs

Definitions

VBO (Vertex buffer object)

- Buffers in the GPU that contain the actual geometry data (positions, normals…)

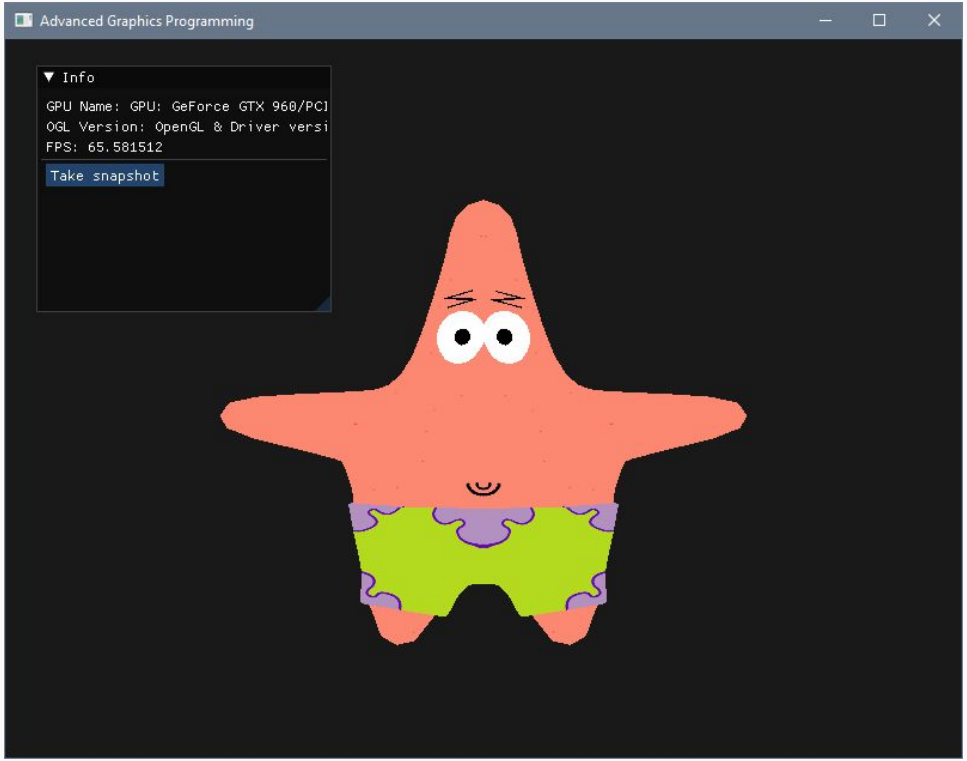
IBO / EBO (Index / element buffer object)

- Buffers in the GPU with indices describing the order in which to render the VBO.

VAO (Vertex attribute object)

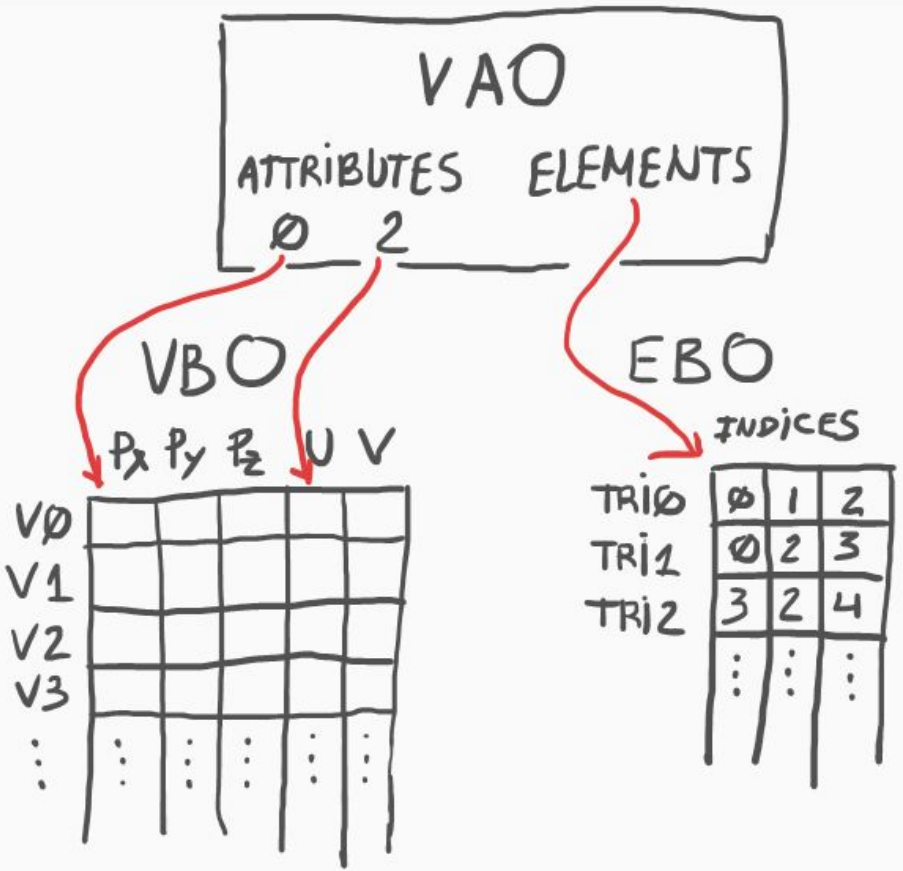
- State object that contains the association between the vertex attributes required by a vertex shader and the corresponding vertex data in a VBO.
    - Shorter: it is a link between vertex shader attributes and the data in a vertex buffer object

A simple shader to render model albedo




```
#if defined(VERTEX) ////////////////////////////////////////////////////

layout(location = 0) in vec3 aPosition;
//layout(location = 1) in vec3 aNormal;
layout(location = 2) in vec2 aTexCoord;
//layout(location = 3) in vec3 aTangent;
//layout(location = 4) in vec3 aBitangent;


out vec2 vTexCoord;

void main()
{
    vTexCoord = aTexCoord;

    // We will usually not define the clipping scale manually...
    // it is usually computed by the projection matrix. Because
    // we are not passing uniform transforms yet, we increase
    // the clipping scale so that Patrick fits the screen.
    float clippingScale = 5.0;

    gl_Position = vec4(aPosition, clippingScale);

    // Patrick looks away from the camera by default, so I flip it here.
    gl_Position.z = -gl_Position.z;
}

#elif defined(FRAGMENT) ////////////////////////////////////////////////

in vec2 vTexCoord;

uniform sampler2D uTexture;

layout(location = 0) out vec4 oColor;

void main()
{
    oColor = texture(uTexture, vTexCoord);
}

#endif
```

A simple shader to render model albedo




```glsl
#if defined(VERTEX) ////////////////////////////////////////////////

layout(location = 0) in vec3 aPosition;
//layout(location = 1) in vec3 aNormal;
layout(location = 2) in vec2 aTexCoord;
//layout(location = 3) in vec3 aTangent;
//layout(location = 4) in vec3 aBitangent;


out vec2 vTexCoord;

void main()
{
    vTexCoord = aTexCoord;

    // We will usually not define the clipping scale manually...
    // it is usually computed by the projection matrix. Because
    // we are not passing uniform transforms yet, we increase
    // the clipping scale so that Patrick fits the screen.
    float clippingScale = 5.0;

    gl_Position = vec4(aPosition, clippingScale);

    // Patrick looks away from the camera by default, so I flip it here.
    gl_Position.z = -gl_Position.z;
}

#elif defined(FRAGMENT) ////////////////////////////////////////////

in vec2 vTexCoord;

uniform sampler2D uTexture;

layout(location = 0) out vec4 oColor;

void main()
{
    oColor = texture(uTexture, vTexCoord);
}

#endif
```

A few structs to organize this VBO / EBO / shader / VAO stuff

What attributes does the VBO contain?

```
struct VertexBufferAttribute
{
    u8 location;
    u8 componentCount;
    u8 offset;
};

struct VertexBufferLayout
{
    std::vector<VertexBufferAttribute> attributes;
    u8                                 stride;
};
```

What attributes does the shader require?

```
struct VertexShaderAttribute
{
    u8 location;
    u8 componentCount;
};

struct VertexShaderLayout
{
    std::vector<VertexShaderAttribute> attributes;
};
```

How do we relate the VBO with the shader attributes so the GPU can draw?

```
struct Vao
{
    GLuint handle;
    GLuint programHandle;
};
```

A few structs to organize this VBO / EBO / shader / VAO stuff

What attributes does the VBO contain?

```
struct VertexBufferAttribute
{
    u8 location;
    u8 componentCount;
    u8 offset;
};

struct VertexBufferLayout
{
    std::vector<VertexBufferAttribute> attributes;
    u8                                 stride;
};
```


```
// create the vertex format
VertexBufferLayout vertexBufferLayout = {};
vertexBufferLayout.attributes.push_back(VertexBufferAttribute{ 0, 3, 0 });              // 3D positions
vertexBufferLayout.attributes.push_back(VertexBufferAttribute{ 2, 2, 3*sizeof(float) }); // tex coords
vertexBufferLayout.stride = 5 * sizeof(float);

// add the submesh into the mesh
Submesh submesh = {};
submesh.vertexBufferLayout = vertexBufferLayout;
submesh.vertices.swap(vertices);
submesh.indices.swap(indices);
myMesh->submeshes.push_back(submesh);
```

A few structs to organize this VBO / EBO / shader / VAO stuff

What attributes does the shader require?

```
struct VertexShaderAttribute
{
    u8 location;
    u8 componentCount;
};

struct VertexShaderLayout
{
    std::vector<VertexShaderAttribute> attributes;
};
```

When we load a program, we should fill its vertex input layout with the attributes it requires

```
app->texturedMeshProgramIdx = LoadProgram(app, "shaders.glsl", "SHOW_TEXTURED_MESH");
Program& texturedMeshProgram = app->programs[app->texturedMeshProgramIdx];
texturedMeshProgram.vertexInputLayout.attributes.push_back({0, 3}); // position
texturedMeshProgram.vertexInputLayout.attributes.push_back({2, 2}); // texCoord
```

Vertex shader layout reflection

Fill input vertex shader layout automatically

Remember this lines?

```
app->texturedMeshProgramIdx = LoadProgram(app, "shaders.glsl", "SHOW_TEXTURED_MESH");
Program& texturedMeshProgram = app->programs[app->texturedMeshProgramIdx];
texturedMeshProgram.vertexInputLayout.attributes.push_back({0, 3}); // position
texturedMeshProgram.vertexInputLayout.attributes.push_back({2, 2}); // texCoord
```

They fill the vertex shader input layout of a program after loading it…

Having to write this every time we load a program is very tedious…

Plus is redundant because this information is already contained in the shader…

Plus is error prone!

Fill input vertex shader layout automatically

With this, you can obtain the number of attributes in a program

```
glGetProgramiv(programHandle, GL_ACTIVE_ATTRIBUTES, &attributeCount);
```

With this, given its index *i*, you can obtain each attribute name, type, etc...

```
glGetActiveAttrib(programHandle, i,
                  ARRAY_COUNT(attributeName),
                  &attributeNameLength,
                  &attributeSize,
                  &attributeType,
                  attributeName);
```

With this, given the attribute name, you can obtain each attribute location

```
attributeLocation = glGetAttribLocation(programHandle, attributeName);
```

Model resources

Models and materials

Within our application instance, we can maintain all the resources in an array...

```
std::vector<Texture>    textures;
std::vector<Material>   materials;
std::vector<Mesh>       meshes;
std::vector<Model>      models;
std::vector<Program>    programs;
```

Models and materials

```
std::vector<Texture>     textures;
std::vector<Material>    materials;
std::vector<Mesh>        meshes;
std::vector<Model>       models;
std::vector<Program>     programs;
```

```
struct Submesh
{
    VertexBufferLayout vertexBufferLayout;
    std::vector<float> vertices;
    std::vector<u32>   indices;
    u32                vertexOffset;
    u32                indexOffset;

    std::vector<Vao>   vaos;
};

struct Mesh
{
    std::vector<Submesh> submeshes;
    GLuint               vertexBufferHandle;
    GLuint               indexBufferHandle;
};
```

Models and materials

```
std::vector<Texture>     textures;
std::vector<Material>    materials;
std::vector<Mesh>        meshes;
std::vector<Model>       models;
std::vector<Program>     programs;
```

```
struct Material
{
    std::string name;
    vec3        albedo;
    vec3        emissive;
    f32         smoothness;
    u32         albedoTextureIdx;
    u32         emissiveTextureIdx;
    u32         specularTextureIdx;
    u32         normalsTextureIdx;
    u32         bumpTextureIdx;
};
```

Models and materials

```
std::vector<Texture>    textures;
std::vector<Material>   materials;
std::vector<Mesh>       meshes;
std::vector<Model>      models;
std::vector<Program>    programs;
```

```
struct Model
{
    u32                 meshIdx;
    std::vector<u32>    materialIdx;
};
```

How to draw our meshes

How to draw our meshes

```
Program& texturedMeshProgram = app->programs[app->texturedMeshProgramIdx];
glUseProgram(texturedMeshProgram.handle);

Model& model = app->models[app->model];
Mesh& mesh = app->meshes[model.meshIdx];

for (u32 i = 0; i < mesh.submeshes.size(); ++i)
{
    GLuint vao = FindVAO(mesh, i, texturedMeshProgram);
    glBindVertexArray(vao);

    u32 submeshMaterialIdx = model.materialIdx[i];
    Material& submeshMaterial = app->materials[submeshMaterialIdx];

    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, app->textures[submeshMaterial.albedoTextureIdx].handle);
    glUniform1i(app->texturedMeshProgram_uTexture, 0);

    Submesh& submesh = mesh.submeshes[i];
    glDrawElements(GL_TRIANGLES, submesh.indices.size(), GL_UNSIGNED_INT, (void*)(u64)submesh.indexOffset);
}
```

How to draw our meshes

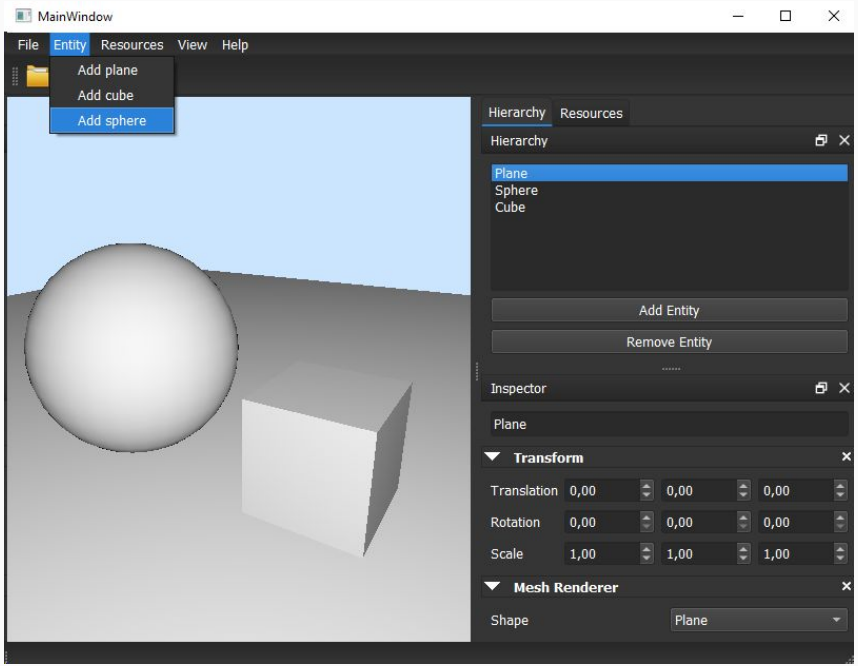
```
GLuint FindVAO(Mesh& mesh, u32 submeshIndex, const Program& program)
{
    Submesh& submesh = mesh.submeshes[submeshIndex];

    // Try finding a vao for this submesh/program
    for (u32 i = 0; i < (u32)submesh.vaos.size(); ++i)
        if (submesh.vaos[i].programHandle == program.handle)
            return submesh.vaos[i].handle;

    GLuint vaoHandle = 0;

    // Create a new vao for this submesh/program
    { ... }

    // Store it in the list of vaos for this submesh
    Vao vao = { vaoHandle, program.handle };
    submesh.vaos.push_back(vao);

    return vaoHandle;
}
```

How to draw our meshes

```
// Create a new vao for this submesh/program
{
    glGenVertexArrays(1, &vaoHandle);
    glBindVertexArray(vaoHandle);

    glBindBuffer(GL_ARRAY_BUFFER, mesh.vertexBufferHandle);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, mesh.indexBufferHandle);

    // We have to link all vertex inputs attributes to attributes in the vertex buffer
    for (u32 i = 0; i < program.vertexInputLayout.attributes.size(); ++i)
    {
        bool attributeWasLinked = false;

        for (u32 j = 0; j < submesh.vertexBufferLayout.attributes.size(); ++j)
        {
            if (program.vertexInputLayout.attributes[i].location == submesh.vertexBufferLayout.attributes[j].location)
            {
                const u32 index = submesh.vertexBufferLayout.attributes[j].location;
                const u32 ncomp = submesh.vertexBufferLayout.attributes[j].componentCount;
                const u32 offset = submesh.vertexBufferLayout.attributes[j].offset + submesh.vertexOffset; // attribute offset + vertex offset
                const u32 stride = submesh.vertexBufferLayout.stride;
                glVertexAttribPointer(index, ncomp, GL_FLOAT, GL_FALSE, stride, (void*)(u64)offset);
                glEnableVertexAttribArray(index);

                attributeWasLinked = true;
                break;
            }
        }

        assert(attributeWasLinked); // The submesh should provide an attribute for each vertex inputs
    }

    glBindVertexArray(0);
}
```

Procedural mesh creation

Built-in meshes



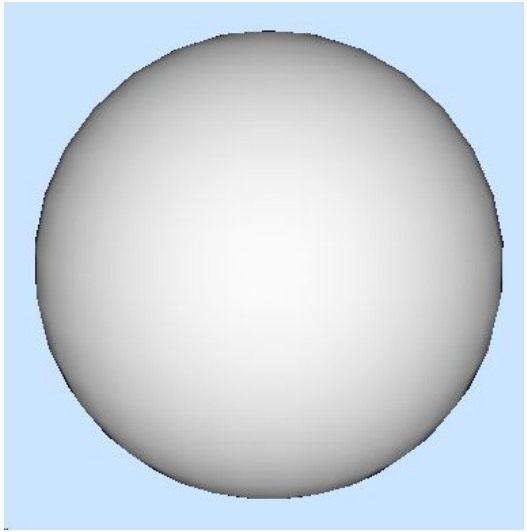
All rendering engines have built-in meshes for different purposes:

- Prototyping
- Debugging placeholders
- Rendering purposes

Try to generate some common meshes during the initialization of the application so they can be used later.

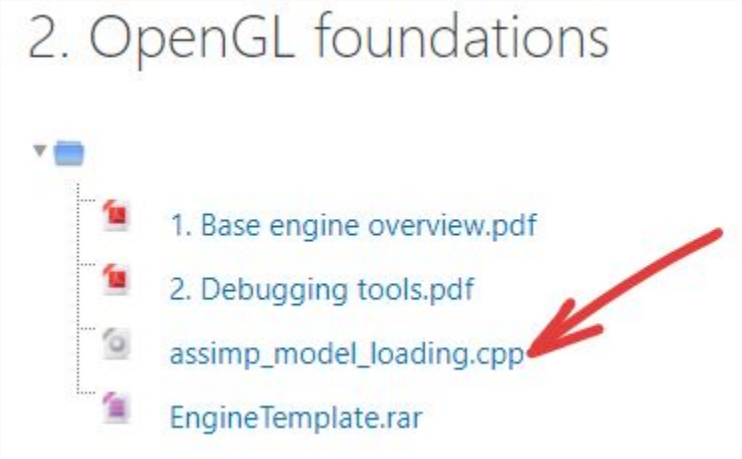
Creation of a sphere mesh

```
#define H 32
#define V 16

    static const float pi = 3.1416f;
    struct Vertex { QVector3D pos; QVector3D norm; };

    Vertex sphere[H][V + 1];
    for (int h = 0; h < H; ++h) {
        for (int v = 0; v < V + 1; ++v) {
            float nh = float(h) / H;
            float nv = float(v) / V - 0.5f;
            float angleh = 2 * pi * nh;
            float anglev = - pi * nv;
            sphere[h][v].pos.setX(sinf(angleh) * cosf(anglev));
            sphere[h][v].pos.setY(-sinf(anglev));
            sphere[h][v].pos.setZ(cosf(angleh) * cosf(anglev));
            sphere[h][v].norm = sphere[h][v].pos;
        }
    }

    unsigned int sphereIndices[H][V][6];
    for (unsigned int h = 0; h < H; ++h) {
        for (unsigned int v = 0; v < V; ++v) {
            sphereIndices[h][v][0] =  (h+0)    * (V+1) + v;
            sphereIndices[h][v][1] = ((h+1)%H) * (V+1) + v;
            sphereIndices[h][v][2] = ((h+1)%H) * (V+1) + v+1;
            sphereIndices[h][v][3] =  (h+0)    * (V+1) + v;
            sphereIndices[h][v][4] = ((h+1)%H) * (V+1) + v+1;
            sphereIndices[h][v][5] =  (h+0)    * (V+1) + v+1;
        }
    }

    VertexFormat vertexFormat;
    vertexFormat.setVertexAttribute(0, 0, 3);
    vertexFormat.setVertexAttribute(1, sizeof(QVector3D), 3);

    Mesh *mesh = createMesh();
    mesh->name = "Sphere";
    mesh->addSubMesh(vertexFormat, sphere, sizeof(sphere), &sphereIndices[0][0][0], H*V*6);
    this->sphere = mesh;
```



Loading models with Assimp

Download assimp loading functions



TODOs

1. Move the textured quad vertices to a mesh object and adapt your code to use VertexBufferLayouts, VertexInputLayouts, and VAOs
    a. No need to create a Model for this if you do not want, just a Mesh will be enough
2. Automatize the VertexShaderLayout creation when calling LoadProgram( )
3. Add the Assimp loading functions available in the Atenea campus and load / render the Patrick model with the shader at the beginning of the slides.
4. Create a few embedded submeshes (e.g. a floor-aligned plane, a cube, a sphere…) with their vertex positions and normals and store it in the same buffer where the initial textured quad is stored.

Patrick says:

Please, do your homework… I look very ugly like this :-(