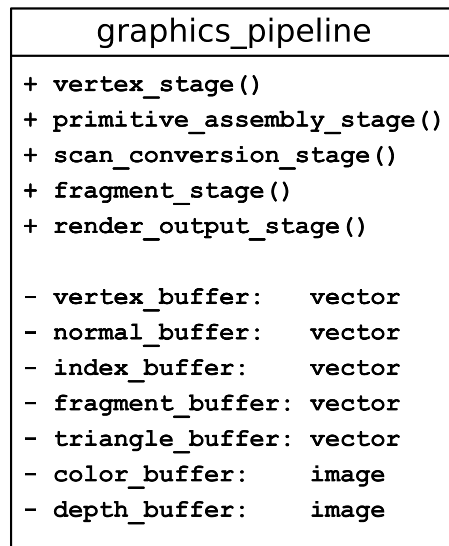
Exercises for the Lecture: "Architecture and Programming Models for GPUs and Coprocessors"
Special Exercise Sheet

Priv.-Doz. Dr. rer. nat. Stefan Zellmann

SS 2021

Special Exercise Rasterization

We want to comprehend the rasterization algorithm by implementing it as a CPU program using the file `rasterization.cpp` from the template. The template implements the logical pipeline of the algorithm "Rasterization" from the lecture. The following diagram illustrates the class `graphics_pipeline` (simplified) that is the central data type. The public interface represents the various phases of the "rasterization pipeline".



The user first fills the input buffers for vertices, indices, and normals and then consecutively calls the stages of the rasterization pipeline. The triangle and fragment buffers *aren't* filled by the user but by the pipeline. The triangle buffer will be filled during the *primitive assembly phase*. Fragments will be generated and are available in the buffer after the *scan conversion phase*. The class further stores state, e.g., for alpha blending, depth test, or backface culling mode. After the pipeline was executed successfully, the output buffers are filled with image data (color and depth). Dedicated functions (`read_color_buffer()`) are used to retrieve the result from the pipeline.

The *vertex stage* and *fragment stage* are special in that they are programmable. To emulate that, the user can pass "vertex shader" and "fragment shader" in the form of C++ lambda functions to the pipeline entry points `vertex_stage()` and `fragment_stage()`. Each of those will be passed a single vertex or fragment when the pipeline is executed. Lambda captures can be used to pass "uniform variables" that are global to the shaders. Vertices are defined by their

position and normal attributes. Those can be changed (e.g., transformed) in the vertex shader. In the fragment shader, the attributes are later available in an interpolated form.

```
vertex
+ pos:    vec3
+ normal: vec3
```

```
fragment
+ x:            int
+ y:            int
+ primitive_id: int

+ z:            float
+ n:            vec3

+ color:        vec4
+ depth:        float
```

In the fragment shader, the fragment properties `x`, `y`, and `primitive_id` can be used that represent the fragment in window coordinates. `z` and `n` are the depth and normal attributes that were interpolated during the scan conversion phase and are accessible from within the fragment shader. *Output attributes* in the fragment shader are `color` and `depth`.

a.) The rasterization pipeline is being executed in `main()`. Implement a simple vertex shader by passing it to the function `graphics_pipeline.vertex_stage()`. The vertex shader transforms the vertices to *normalized device coordinates* (NDC). For that you can use the uniform variables `view_matrix` and `proj_matrix` (also see the appendix on using linear algebra functions).
Furthermore, we also transform normal vectors that are stored as vertex attributes. Normal vectors aren't transformed to NDC but to eye/camera coordinates. They're also transformed specially—by multiplying by the inverse transform of the modelview matrix.

b.) Implement Pineda's algorithm that we discussed in the lecture in the private member function `rasterize()`. That function is called by `graphics_pipeline.scan_conversion_stage()` on execution and receives one triangle in window coordinates:

triangle
+ v1: vec2
+ v2: vec2
+ v3: vec2
+ z1: float
+ z2: float
+ z3: float
+ n1: vec3
+ n2: vec3
+ n3: vec3

The vertex positions in window coordinates `v1`, `v2`, and `v3` represent 2.5D positions with subpixel accuracy. Depth and normal attributes are also available and need to be interpolated with correct perspective: `z1`, `z2` and `z3`; `n1`, `n2` and `n3`.

Take the following approach: Use the function `get_bounds()` to compute the triangle's bounding rectangle and iterate over each "raster position" in that rectangle. Evaluate the "Pineda edge functions" (EE) for each raster position and the three edges (`v1 - v2`), (`v2 - v3`), and (`v3 - v1`). If `backface_culling` is active, we generate *fragments* where the EEs are > 0. Generate fragments that are initialized by their raster position `(x,y)` and their `id`. Append the fragments to the `fragment_buffer` member. We also need to provide values for `z` and `n`. For that, we compute barycentric coordinates from the EEs. Those can be derived from the EEs (shown in a later exercise), they're just given by:

$$\lambda_1 = \frac{E_2(x,y)}{2A(T)}$$
$$\lambda_2 = \frac{E_3(x,y)}{2A(T)}$$
$$\lambda_3 = \frac{E_1(x,y)}{2A(T)},$$

where $A(T)$ is the triangle's area (see the already implemented function `area(triangle)`). You can use the function `lerp(T attr1, T attr2, T attr3, float bary1, float bary2)` from the linalg library (see the appendix) to interpolate attributes of, e.g., type `float` or `vec3`.

c.) Implement the *depth test* using the function `graphics_pipeline.render_output_stage()`. The function already initializes the output buffer and we now have to iterate over the (now initialized) fragment buffer and test for each fragment `f` if its depth `f.depth` is smaller than the value that is already stored at `depth_buffer(f.x, f.y)`. If so, we update the depth buffer. Also make the depth test toggleable—if the value of `depth_test` is `false`, the fragment overwrites the output no matter if it "wins" the depth test or not.

d.) The framgent shader currently just assigns the color white to each fragment. Replace that with "Lambertian shading". For that use the RGB material color (`diff`) that is passed to the shader as a uniform variable, as well as the light vector (`L`). (Recall that with the Lambertian model you weight the diffuse material color by the dot product (linalg function `dot(vec3,vec3)`) of light and normal vector.) Implement "two-sided shading": backsides are shaded, too.

e.) Finally, we adapt the pipeline to also support alpha blending. We therefore extend the render output phase (ROP). Alpha blending is the reason why we perform the ROP operations in a temporary buffer. Implement alpha blending with the over operator from the lecture, and by using pre-multiplied alpha. Only perform alpha blending if that mode was activated (one usually also deactivates depth test and backface culling then, but the pipeline should produce "some" output even if they're active).
For correct alpha blending, the geometry needs to be pre-sorted. Do that *right after the vertices are read*: Create a deep copy of the vertices. Transform all those *copied* vertices to eye coordinates. In the eye coordinate system, the camera is at the origin and pointing along "negative-z"—that coordinate system lends itself to sorting the vertex positions. We'll however not just sort vertices but triangles. For simplicity, just assume that triangles won't intersect—then sort the triangles by computing the arithmetic mean of their vertex positions. Make sure to not sort the actual *input data* but only the copy and store a mapping between the copy and the vertices from the input stream. *After* the order was determined that way, we use the mapping to sort the (untransformed) input data into visibility order.

Remark: the above procedure is redundant as we're performing this operation on a copy and the vertex pipeline will later apply the exact same transformation. This approach just illustrates how such a pre-sorting step would have to be performed on the CPU by the user while the later graphics pipeline will perform the same transformations again. In this contrived example, the procedure could of course be optimized by removing that redundancy. We don't do that—for educational reasons.

Appendix
You can use the Visionaray library for linalg functions:

`https://github.com/szellmann/visionaray`

Parts of that library are header only, you don't have to link with it. In particular, the linear algebra part of the library is header-only. Just include the file `<visionaray/math/math.h>` and those functions are available in `namespace visionaray`. You have to adjust your compiler's input path, e.g., GCC:

```
g++ rasterization.cpp -I/path/to/visionaray/include -std=c++11 -O3
```

The linalg functions are very similar to GLSL. You can find examples on their usage in the `math_example.cpp` file coming with the template. Alternatively, you can also use the library GLM which provides similar means. Linking with that might be a bit more complicated though, and some of the functions in the template would need to be ported.