Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

3D Pipeline in C++ 20

Mirco De Zorzi

October 29, 2021

Contents

1 Assignment Description

Implement a software 3D rendering pipeline with programmable fragment shader.

Assume the pipeline receives triangles already in view coordinates, project them, rasterize the triangle, interpolate the vertex attributes, and for each fragment pass the interpolated vertex to the fragment shader. Render to a software target, a continuous region of memory (e.g. array or vector) of a parametric type target_t which holds the rendered image in row-major format and can be used to display or save the rendered image. The pipeline must be agnostic with respect to type (target_t) and dimensions of the target. The fragment shader, when producing a fragment, must provide a target_t. Implement the shader in terms of a strategy pattern, i.e., a call to a polymorphic funcion in a class held by the pipeline, but hot-swappable.

2 Solution

We initialize a variable of type `screen` defining it's size through the class template parameters `W` and `H`, respectively representing the width and height of the screen. The constructor takes 1 parameter, a functor with signature `pixel(float)` which represents th fragment shader.
Calling `screen::draw` by passing it 3 three-dimensional points will rasterise the triangle represented by the parameters to the screen buffer; calling `screen:render` will render the screen buffer to the terminal.

Follows an example of usage of the pipeline:

```
int main() {
  // define shader
  const auto shader = [](float n) -> pixel {
    short c = (n + 1.0f) * 128.0f;
    return pixel{L'\u2588', color{c, c, c}, color{c, c, c}};
  };

  screen<150, 50> screen(shader);

  const auto v1 = vec3{ 1.0, -1.0,  1.5};
  const auto v2 = vec3{ 1.0,  1.0,  1.1};
  const auto v3 = vec3{-1.0,  1.0,  1.5};
  const auto v4 = vec3{-1.0, -1.0,  1.9};

  screen.draw(v1, v2, v3);
  screen.draw(v1, v3, v4);

  screen.render();
}
```

2.1 The Screen Buffer

The screen buffer consists of:

- `shader`: a functor representing the fragment shader;
- `buf`: screen buffer, a contiguous are of memory representing the single screen pixels;
- `depth`: depth buffer, a contiguous are of memory representing the z component of each *pixel*.

```
template <int W, int H>
struct screen {
  using shader_fn_type = std::function<pixel(float)>;

  // future-proof in case we want to extend the fragment shader
  struct {
    shader_fn_type fn;

    auto operator()(float n) {
      return fn(n);
    }
  } shader;

  std::array<pixel, W * H> buf;
  std::array<float, W * H> depth;

  constexpr screen(shader_fn_type &&fn)
      requires (W > 0 && H > 0)
      : shader{fn} {
    buf.fill(pixel{L' ', color{0, 0, 0}, color{0, 0, 0}});
    depth.fill(0);
  }
```

2.2 Accessing Screen Buffer

When representing a two-dimensional space as a contiguous vector it's often useful to abstract the subscription operator to guarantee safety. The point at position (i, j) is actually at the $(wj + i)^{th}$ position of the array. Furthermore when accessing our screen buffer we must also take into consideration the screen's depth buffer. When trying to overwrite a *pixel* with one further than the previous one, the operation must fail.

There exist 3 sensible ways to archive this:

1. `char* at(...)`: A C style approach, returning a pointer to the memory if the index is in-bound, and a null pointer when out-of-bound.
2. `char& at(...) throws`: A *pre-C++17* approach, returning a reference to the memory when the index is in-bound, and throwing when out-of-bound.
3. `std::optional< std::reference_wrapper<char> > at(...)`: A *post-C++17* approach, returning an optional reference to the pointer[1].

Option 3 was chosen for the project.

```
constexpr std::optional<std::reference_wrapper<char_type>> at(int x, int y, float d) {
        assert(y * w + x < w * h);
        if (depth[y * w + x] + 1 < d) {
                return std::nullopt;
        }
        depth[y * w + x] = d;
        return std::optional{std::reference_wrapper{buf[y * w + x]}};
}
```

[1] `std::optional<T>` behaves much like Haskell's `Maybe` monad.

2.3 Rendering

In this implementation each *pixel* on the screen is represented by a Unicode character, foreground and background color. To render the screen buffer we iterate through each *pixel*, calling a procedure that renders it using *ANSI* control sequences to set the correc t foreground and background colors.

```
constexpr void putpixel(pixel p) {
        std::wprintf(L"\033[48;2;%d;%d;%dm\033[38;2;%d;%d;%dm%lc\033[0;00m",
                        p.background.r, p.background.g, p.background.b,
                        p.foreground.r, p.foreground.g, p.foreground.b,
                        p.c);
}

constexpr void render() {
        for (int i = 0; i < h; i++) {
                for (int j = 0; j < w; j++) {
                        putpixel(at(j, i));
                }
                std::putwc(L'\n', stdout);
        }
}
```

2.4 Drawing a Triangle

We first derive the projection matrix through the following formula:

$$P = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{b-t} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{2nf}{n-f} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1}$$

where the varibales n, f, t, b, l, and r represent the 6 clipping planes of our camera.
By multipling the projection matrix by a 4×1 vector where the first three components represent the coordinates of our point and the fourth is the constant 1, we can derive the translated position of our point:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{b-t} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & \frac{2nf}{n-f} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2}$$

To then calculate the position in 2D space we take the x' and y' components and divide them by w'.

$$(x, y) = (\frac{x'}{w'}, \frac{y'}{w'}) \tag{3}$$

```
constexpr void draw(vec3 v1, vec3 v2, vec3 v3) {
        constexpr auto projection = get_projection(-1, 1, -1, 1, 1, 2);

        auto p1 = mat4_mul_vec4(projection, vec4{v1, 1.0});
        auto p2 = mat4_mul_vec4(projection, vec4{v2, 1.0});
        auto p3 = mat4_mul_vec4(projection, vec4{v3, 1.0});

        p1.normalize(); // equivalent to p1 = p1 / p1.w
        p2.normalize();
        p3.normalize();

        auto bb = bounding_box(p1, p2, p3, w, h);
        for (int i = bb.z; i < bb.w; i++) {
                for (int j = bb.x; j < bb.y; j++) {
                        const auto mi = to_cartesian(i, h);
                        const auto mj = to_cartesian(j, w);
                        if (inside_triangle(p1, p2, p3, mj, mi)) {
                                auto d = get_z_component(p1, p2, p3, mj, mi);
                                if (auto c = at(j, i, d)) {
                                        c->get() = shader(d);
                                }
                        }
                }
        }
}
```

To optimize our code and prevent the rasterization of triangles outside of the screen we calculate a given triangle's bounding box before iterating through each pixel in it, checking whether the pixel is contained inside of the triangle or not. If the conditions are met we normalize the screen position to cartesian coordinates in the range $[-1, 1]$ and calculate the distance between the intersection of a ray emanating from our camera hitting perpendicularly the plane passing through the points composing our triangle.

3 Disassembly

The previously described code compiled with `-O3` totals under 700 lines of assembly code[2].

4 Further Potential Improvements

4.1 `.obj` File Parsing

What follows is an example of a simple `obj` parser written in pseudo-code that could be used to load more complex three-dimensional model to render to the screen.

```
fn parseObj(path: string) -> mesh
        let file = open(path)

        let vs = []
        let ts = []

        while c.has_more
                match file.getc()
                        'f' =>
                                a, b, c = input("%d %d %d")
                                fs.append({vs[a - 1], vs[b - 1], vs[c - 1]})
                        'v' =>
                                a, b, c = input("%f %f %f")
                                vs.append({a, b, c})

                        -- skip line
                        default => while c.has_more and file.getc() != '\n'

        return mesh(ts)
```

[2]https://godbolt.org/z/3h6P4Evo6