Writeup

Patrick LaFontaine

May 11, 2021

1 Some kind of introduction

Graphics bugs are notoriously difficult in both identifying that a program actually has a bug and in debugging what the issue is. One class of graphics bugs are related to data transfer at the boundary between the CPU and GPU. More specifically, this library statically checks that: all of the data needed for a Webgpu pipeline to run is bound on the GPU, that this data is set to the correct location/slot/index in the pipeline, the data is of the correct type expected by the shader, and

2 An example bug in psuedo-code

Take for example the following vertex and fragment shader in the glsl-like macro style that I am proposing:

```
my_shader! {vertex = {
    [[vertex in] vec4] a_position;
    [[vertex in] vec4] vertexColor;

    [group1 [uniform in] mat4] u_view;
    [group2 [uniform in] mat4] u_proj;
    [group3 [uniform in] mat4] u_model;

    [[out] vec3] fragmentColor;
    [[out] vec4] gl_Position;

    {{
        void main() {
            fragmentColor = vertexColor;
            gl_Position = u_proj * u_view * u_model * a_position;
        }
    }}
}}
```

```
my_shader! {fragment = {
    [[in] vec3] fragmentColor;
    [[out] vec4] color;
    {{
        void main() {
            color = vec4(fragmentColor, 1.0);
        }
    }}
}}
```

The bulk of the work happens in the Vertex shader(cite figure) which iterates over two vertex buffers containing a position value and a color value. There are 3 uniform matrices: the view, projection, and model matrices which are declared as uniform and are organized in their own groups.

3 How my library/dsl prevents this bug

3.1 Why is this a real bug/issue? Examples????

4 Background: Rust, WebGPU, GLSL, Compile-time/Macro programming

5 The details

This library provides four parts: a macro that which used to declare a shader, a macro that takes in shaders and produces a "Context" type, methods for binding data to the GPU, and sub-libraries that are specific to running a compute or graphics pipeline.

5.1 The shader macro

Currently shaders are implemented in a modified GLSL-like syntax using the my_shader! macro. This macro takes in a name which the shader will be set to, a list of parameters, and the main body of the shader which is valid GLSL. For each parameter, it takes a group name if it is part of a bind group, a list of qualifiers, a GLSL type and the parameter name.

The name of the shader can then be used later as a macro to call the shader when you want to use it. This is useful as at compile time, a shader can be declared using my_shader! in one file and then used in a different file.

5.2 The Context and set functions

One or more shaders are passed to the eager_binding! macro to create what I call the Context type. A Context is parameterized on the type of the pipeline it is used for and all of its parameters. Each parameter is represented as either

Bound, or Unbound depending on whether the Context contains that parameter. Initially, all parameters are set as Unbound like Context<ComputePass, Unbound, Unbound>. A parameter can be switched from Unbound to Bound by calling the appropriate Context::set_<parameter_name(s)> method where <parameter_name(s)> is one of more names of parameters that are underscore seperated. An actual set method may be called like let context1 = context.set_a_position(&mut rpass, &vertex_position);.

5.3 Sending data to the GPU (BindGroups/Vertex/Indices)

Most data that is used in a pipeline is either data stored in vertex buffers or in bind groups depending on how the parameters are declared in the shader. Vertex buffers are stored in a struct called Vertex which is parameterized on a single WgpuType. Likewise, bind groups are created and stored by the BindGroupN struct where N is the number of types the bind group is parameterized on. N is typically limited by the hardware to be between $1 \leq N \leq 4$ or $1 \leq N \leq 8$. The WgpuType is a trait that is implemented on Rust types which have an equivalent type in GLSL. In addition to the standard GLSL type, additional information needs to be stored so three wrapper structs are included: BufferData, SamplerData, and TextureData. Each of these structs only wraps the data that is passed in and is parameterized on layout specific information.

Index Buffer data is a special case as it is limited to the Vec<u8> rust type or a list of unsigned 8-bit integers. This buffer is stored in the Indices and it is used for graphics_run_indecies.

5.4 Graphics/Compute Header Library

This library contains two helper libraries for writing Webgpu graphics pipelines and compute pipelines. The bulk of these libraries are in provided a function to compile a shader(s) into a pipeline (graphics_compile and compute_compile) and then functions to run the pipeline when it has been fully bound (graphics_run, graphics_run_indecies, compute_run). These functions wrap the typical WebGPU boilerplate involved in using GPU pipelines while maintaining as much expressiveness as possible.

6 Future work

In terms of future work, more time should be spent ensuring this library correctly handles all graphics pipelines with more complex examples like a ray tracer. It would be interesting to see if this way of binding variables to a typed context can be applied to more domains or with other graphics libraries besides WebGPU.

7 Related Work

There is some related work in inline assembly programming where the user provides a block of inline assembly, typically for C/C++ programs, which the

compiler will treat as a black box. In "Interface Compliance of Inline Assembly"(https://arxiv.org/pdf/2102.07485.pdf), their work statically checks the interface of inline assembly with C/C++ code that calls it to make sure it has properly set up the environment for the assembly code to run without bugs.