Blis: Better Language for Image Stuff
Language Reference Manual
Programming Languages and Translators, Spring 2017

Abbott, Connor (cwa2112)	`[System Architect]`
Pan, Wendy (wp2213)	`[Manager]`
Qinami, Klint (kq2129)	`[Language Guru]`
Vaccaro, Jason (jhv2111)	`[Tester]`

February 23, 2017

1 Introduction

Blis is a programming language for writing hardware-accelerated 3D rendering programs. It gives the programmer fine-grained control of the graphics pipeline while abstracting away the burdensome details of OpenGL. The Blis philosophy is that OpenGL provides more power and flexibility than some graphics programmers will ever need. These OpenGL programmers are forced to write boilerplate code and painfully long programs to accomplish the simplest of tasks. By exposing only the bare essentials of the graphics pipeline to the programmer, Blis decreases the number of decisions that a graphics developer has to make. The result is sleek, novice-friendly code.

With Blis, you can write real-time 3D rendering programs such as 3D model viewers. You can write shadow maps for rendering dynamic shadows and use render-to-texture techniques to produce a variety of effects. In short, the idea is that you can write programs that manipulate Blis' simplified graphics pipeline.

In particular, writing vertex and fragment shaders is now more convenient. Rather than having to write shader code in a separate shader language (GLSL), you can use Blis to write both tasks that run on the GPU and those that run on the CPU. Consequently, shaders can reuse code from other parts of the program. Uniforms, attributes, and textures are registered with shaders by simply passing them as arguments into user-defined shader functions. Furthermore, loading shaders from the CPU to GPU is easily accomplished by uploading to a built-in pipeline object. See the "Parts of the Language" section below for more information about this pipeline object, which is a central feature of Blis.

Blis has two backends: one for compiling source code to LLVM and another for compiling to GLSL. The generated LLVM code links to the OpenGL library in order to make calls that access the GPU.

2 Sorts of programs to be written

Rendering is a major subtopic of 3D computer graphics, with many active researchers developing novel techniques for light transport modeling. These researchers would like to test their new ray tracing, ray casting, and rasterization developments by actual experimentation. These new methods would be translated into rendering programs written in our language. Because these programs would be written at a higher level of abstraction than OpenGL, they would resemble mathematical thinking and derivations. Our language would also facilitate users to make larger tweaks to their programs with fewer lines of code, allowing for a more effective use of their time in experimentation. Minimizing this transaction cost between research and code should allow for more development of rendering software overall.

3 Lexical Conventions

For the most part, tokens in Blis are similar to C. The main difference is the extra keywords added to support vector and matrix types. There are four kinds of tokens: identifiers, keywords, literals, and expression operators like `(`, `)`, and `*`. Blis is a free-format language, so comments and whitespace are ignored except to separate tokens.

3.1 Comments

Blis has C-style comments, beginning with `/*` and ending with `*/` which do not nest. C++ style comments, beginning with `//` and extending to the end of the line, are also supported.

3.2 Identifiers

An identifier consists of an alphabetic character or underscore followed by a sequence of letters, digits, and underscores.

3.3 Keywords

The following identifiers are reserved as keywords, and may not be used as identifiers:

- `int`
- `bool`
- `break`
- `char`
- `struct`
- `continue`
- `float`
- `return`
- `if`

- `else`
- `for`
- `while`
- `true`
- `false`
- `out`
- `inout`
- `vec2`
- `vec3`
- `vec4`
- `mat2x2`
- `mat2x3`
- `mat2x4`
- `mat3x2`
- `mat3x3`
- `mat3x4`
- `mat4x2`
- `mat4x3`
- `mat4x4`
- `bvec2`
- `bvec3`
- `bvec4`
- `bmat2x2`
- `bmat2x3`
- `bmat2x4`
- `bmat3x2`
- `bmat3x3`
- `bmat3x4`
- `bmat4x2`
- `bmat4x3`
- `bmat4x4`
- `RGBA8`
- `RGB8`
- `RG8`
- `R8`
- `@vertex`
- `@fragment`

3.4 Literals

3.4.1 Integer Literals

Integer literals consist of a sequence of one or more of the digits 0-9. They will be interpreted as a decimal number.

3.4.2 Floating Point Literals

A floating-point literal consists of an integer part, a decimal, a fractional part, and `e` followed by an optionally signed exponent. Both the integer and fractional parts consist of a sequence of digits. At least one of the integer and fractional parts must be present, and at least one of the decimal point and the exponent must be present. Since Blis only supports single-precision floating point, all floating point constants are considered single-precision.

3.4.3 Character Literals

A character literal consists of a single character, or a backslash followed by one of `'`, `n`, `r`, `t`, or a sequence of decimal digits that must form a number from 0 to 255 which represents an ASCII code. `'\''`, `'\n'`, `'\r'`, and `'\t'` have the usual meanings.

3.4.4 Boolean Literals

Boolean literals consist of the two keywords `true` and `false`.

3.4.5 String Literals

String literals consist of a series of characters surrounded by `"`. All the escapes described in section 3.4.3 are supported, as well as `\"` to escape a `"`. String literals have type `char[n]`, i.e. a fixed-size array of characters where `n` is the number of characters. For example, `"Hello world"` has type `char[11]`. Unlike C, there is no extra `'\0'` inserted at the end of the string.

4 Syntax and Semantics

4.1 Basic Types

4.1.1 Integers

Blis supports integers through the `int` type, which is a 32-bit two's-complement signed integer type.

4.1.2 Characters

Blis supports characters through the `char` type, which is an 8-bit unsigned integer type.

4.1.3 Floating Point

Blis supports single-precision (32-bit) floating point numbers through the `float` type.

4.1.4 Booleans

Blis supports booleans through the `bool` type. Booleans may only ever be `true` or `false`.

4.2 Vectors and Matrices

To more easily represent position data and color data, Blis supports vectors and matrices of floating-point numbers with the `vec2`, `vec3`, and `vec4` builtin types, as well as `matNxM` where $2 \leq \texttt{N} \leq 4$ and $2 \leq \texttt{M} \leq 4$, similar to GLSL. There is no `mat1xN` or `matNx1`; use the vector types instead. Blis supports the full complement of matrix-vector multiplication operations on these types as described in section 4.7, as well as component-wise multiplication and a number of built-in functions described in section 4.12.

Blis also supports vectors and matrices of boolean type, denoted by `bvec2`, `bmat2x2`, etc., which are the result of the comparison operators `==`, `!=`, `>`, `<`, `>=`, and `<=` applied to floating-point vectors and matrices. The builtin functions `all()` and `any()` can be used to reduce these types to a single boolean value, so that e.g. `all(foo == bar)` checks that all the components of `foo` and `bar` are equal.

In addition to the special operators and built-in functions, `vecN` types have `x`, `y`, `z`, and `w` members as appropriate as if they were normal structures, and `matNxM` types have `xx`, `xy`, `yx`, etc. Thus, the syntax to access the `x` component of a `vec4` `foo` is `foo.x`. This also applies to the boolean vector and matrix types.

4.3 Packed Types

Oftentimes, when handling large amounts of data, we want to store a more compact representation of the data in order to reduce bandwidth and memory requirements. For example, color data is often stored using 8 bits per channel since a larger color depth wouldn't cause any noticeable increase in quality. Conceptually, each 8-bit channel represents a luminance value between 0 (minimum luminance) and 1 (maximum luminance), with 0 mapping to 0 and 255 mapping to 1. GPU's even include special support for packed formats like these, being able to sample from an image of packed data and convert it on the fly to floating point, or vice versa, quantizing a floating-point output to a packed format when writing to an image. To support this use-case, Blis has special packed types such as `RGBA8` which are described in the table below. These types are implicitly convertible to and from the appropriate vector type (given in the following table), but converting from the vector type to the packed type will obviously involve some loss of data. The conversion may happen on the CPU, or for operations involving the GPU, it may happen automatically through OpenGL.

Packed Type	Vector Type	Packed to Vector	Vector to Packed
`RGBA8`	`vec4`	$val/255$	$\lfloor \min(\max(val, 1), 0) \cdot 255 \rfloor$
`RGB8`	`vec3`	$val/255$	$\lfloor \min(\max(val, 1), 0) \cdot 255 \rfloor$
`RG8`	`vec2`	$val/255$	$\lfloor \min(\max(val, 1), 0) \cdot 255 \rfloor$
`R8`	`float`	$val/255$	$\lfloor \min(\max(val, 1), 0) \cdot 255 \rfloor$

4.4 Arrays

Blis supports one and two-dimensional arrays of fixed and variable size. An array of `foo`'s is denoted as `foo[N]`, where `N` is an integer literal, or `foo[]` for a runtime-sized array. Arrays nest left-to-right, so that `int[3][2]` is an array of size 3 of an array of size 2 of integers. Two-dimensional runtime-sized arrays are denoted by `foo[,]` or `foo[N,]` where `N` is an integer literal. If the outer dimension is omitted, then it is sized at runtime as well. The reason for including two-dimensional runtime-sized arrays explicitly, rather than using `foo[][]`, is that `foo[][]` is semantically different from `foo[,]` — a variable of the former type can have inner arrays with different sizes, while the latter is a two dimensional array where all the inner arrays must have the same size. The latter type construction, when combined with packed types, is useful for representing images whose width and height are only known at runtime; for example, a `loadImage()` routine would probably return a value of type `RGBA8[,]` or `RGB8[,]`.

Although constant values of type `foo[,]` cannot be constructed directly, values of type `foo[N][M]` can be implicitly converted to `foo[,]` as described in section 4.6. Arrays of type `foo[N]` can be constructed by giving a comma-separated list of `N` values enclosed by `[` and `]`. For example,

```
vec3[3] bar = [vec3(0., 0., 1.), vec3(0., 1., 0.), vec3(1., 0., 0.)];
```

Note that, due to implicit conversions, the following would also work:

```
vec3[] bar = [vec3(0., 0., 1.), vec3(0., 1., 0.), vec3(1., 0., 0.)];
```

although the resulting `bar` would have a different type: variable-sized array instead of fixed-size array. However, none of the elements of the list being constructed can have different types; that is, implicit type conversion doesn't happen across array members. For example, it is illegal to mix `vec4` and `RGBA8` types within a single array.

4.5 Structures

Blis supports structures, similar to C, for storing multiple values together. Structures are declared with a syntax similar to C's, for example:

```
struct {
    vec3 a;
    float[3] b;
} MyStruct;
```

However, referring to the type `MyStruct` does not require the `struct` keyword:

```
MyStruct foo;
MyStruct makeMyStruct(int a, float[3] b);
```

This gives the compiler the flexibility to use the `struct` mechanism internally to create various built-in types, like `vec3`, `window`, etc.

Declaring a structure type implicitly creates a new function with the types of the members as arguments which creates an instance of the structure, similar to GLSL. For example, you could create an instance of `MyStruct` like:

```
MyStruct foo = MyStruct(
    vec3(0., 0., 0.),
    [0., 0., 1.]
);
```

Thus, structures and functions share the same namespace.

4.6 Type Conversions

There are a few cases in Blis where one type can be an acceptable substitute for another, namely, sized vs. unsized arrays and packed types, and the type conversion happens implicitly. Generally, it is assumed that one type, known through the rest of the rules, needs to be converted to another type which is also known beforehand; in cases where this might not be true, the rule to follow is called out explicitly. A type A can be converted to another type B if:

- A and B are already the same type.
- A is a packed type and B is its corresponding vector type, or vice versa.
- A is of type `C[N][M]` or `C[N,]` and B is of type `D[N,]` or `D[,]` and C can be converted to D.
- A is of type `C[N]` and B is of type `D[N]` or `D[]` and C can be converted to D.

Note that if A and B are both struct types, then for them to be the same type, they must have the same name; that is, structure equality is by name, not structural. Also, this relation is not symmetric (for example, `float[5]` can be converted to `float[]` but not vice versa), but it is transitive, and of course it is reflexive.

There are also the usual explicit type conversions. Similar to GLSL, they use the function call syntax. Thus, you can do:

```
float foo = float(42);
float bar = float(true); // returns 1.0
int bar = int(false); // returns 0
```

The full complement of 6 conversions amongst `float`, `int`, and `bool` is provided with the usual meaning, except for converting to `bool` which can be done through comparing with 0. Also, `char` can be be converted to `int` and vice versa, with the latter reinterpreting the `int` as unsigned and taking the lowest 8 bits.

4.7 Operators

The operators supported by Blis, taken from GLSL, are:

Precendence	Operator Class	Operators	Associativity
1	Parenthetical Grouping	`( )`	Left to right
2	Array Subscript	`[ ]`	Left to right
	Function Call	`( )`	
	Structure Member	`.`	
	Postfix Increment/Decrement	`++`, `--`	
3	Prefix Increment/Decrement	`++`, `--`	Right to Left
	Unary Operators	`+`, `-`, `~`, `!`	
4	Multiplicative Operators	`*`, `/`, `%`	Left to Right
5	Additive Binary Operators	`+`, `-`	Left to Right
6	Bit-wise Shift	`<<`, `>>`	Left to Right
7	Relational	`<`, `>`, `<=`, `>=`	Left to Right
8	Equality	`==`, `!=`	Left to Right
9	Bitwise And	`&`	Left to Right
10	Bitwise Exclusive Or	`^`	Left to Right
11	Bitwise Inclusive Or	`|`	Left to Right
12	Logical And	`&&`	Left to Right
13	Logical Exclusive Or	`^^`	Left to Right
14	Logical Inclusive Or	`||`	Left to Right
15	Selection	`? :`	Right to Left
16	Assignment	`=`	Right to Left
	In-place Arithmetic	`+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, `|=`	
17	Sequencing	`,`	Left to Right

The in-place operators `a OP= b` are defined to be equivalent to `a = a OP b`. Evaluation proceeds left to right, and any side effects (for example, incrementing the lvalue by `++`) are evaluated immediately before returning the value.

4.7.1 Vectorized Operators

In addition to their normal operation on floating-point and boolean types, the following arithmetic operator classes can operate on vectors in a per-component manner:

- Postfix Increment/Decrement
- Prefix Increment/Decrement
- Unary Operators
- Multiplicative Operators
- Additive Binary Operators

- Relational
- Logical And
- Logical Exclusive Or

That is, for example, if `a OP b` can take an `a` of type `float` and `b` of type `bool` and produce a result of type `float`, then it can also take an `a` of type `vec4` and `b` of type `bvec4` to produce a result of type `vec4`, where `result.x = a.x OP b.x`, `result.y = a.y OP b.y`, etc. In addition, if some of the inputs are of scalar types, then they will automatically be replicated to create a vector of the required size. Thus, the following is legal:

```
vec3(1.0, 2.0, 3.0) * 2.0 // returns vec3(2.0, 4.0, 6.0)
```

but the following is not:

```
vec3(1.0, 2.0, 3.0) * vec2(1.0, 2.0)
```

In addition, all of the above operators except for `*` can operate per-component on matrices. `*` only works on matrices of floating-point type, and denotes linear algebraic matrix-matrix or matrix-vector multiplication. Multiplying a vector by a matrix is not permitted.

For any of these operators, operands of packed type are immediately expanded to their corresponding expanded type before considering whether the operation is allowed or actually doing the operation.

4.7.2 Equality

The equality operators (`==` and `!=`) are structural in Blis, meaning that they recursively compare the elements of structures and arrays. The rule for whether two values A and B can be compared is as follows:

- If A and B are vectors or matrices, then the comparison is done per-component with the implicit replication described in section 4.7.1, and returns a boolean vector of the same size as the original vector type.
- If A is a packed type and B is its corresponding vector type, or vice versa, then the vector type is truncated to the packed type and then compared componentwise. If the non-packed type is a scalar, it is also automatically replicated.
- If A and B are both packed types, then they are compared per-component. A and B must both be the same packed type for the equality to succeed.
- A is of type `C[N][M]` or `C[N,]` and B is of type `D[N,]` or `D[,]`, then each value of type C is compared with the corresponding value of type D and the result is combined (or'd for `==` and anded for `!=`).

- If A is of type `C[N]` and B is of type `D[N]` or `D[]`, then each value of type C is compared with the corresponding value of type D and the result is combined.
- If A and B are runtime-sized arrays, and they are both one-dimensional or two-dimensional, then the elements of A and B are compared and combined, except when the sizes do not match, in which case the comparison returns "not equal."
- If A and B are of the same structure type, then A and B are compared member-by-member and the result combined appropriately. If one of the comparisons returns a boolean vector, then the components of the vector are combined to produce a single boolean as well. Therefore, the return type of structure comparisons is always a single boolean.

Any situation which does not fall under one of these cases results in a type error.

4.7.3 Lvalues and Assignment

Expressions on the left-hand side of an `=` must be an *lvalue*, which must be either an identifier or one of the following:

- A member access using the `.` operator where the left-hand side is an lvalue.
- An array dereference using the `[ ]` operators where the left-hand side is an lvalue.
- An lvalue within parentheses.

When assigning a value, the right-hand side must be implicitly convertible to the left-hand side as defined in section 4.6.

4.8 Statements and Control Flow

A statement in Blis can be:

- A value followed by a semicolon.
- A variable declaration, which consists of the type, followed by a comma-separated list of identifiers (names), each optionally followed by `=` and then an expression initializing the variable.
- A *block* of statements surrounded by { }
- A `break` followed by a semicolon.
- A `continue` followed by a semicolon.

- A `return` followed by an optional value and a semicolon.
- A sequence of statements enclosed in braces.
- An `if` statement with optional else clause.
- A `for` loop.
- A `while` loop.

The syntax and semantics of `if`, `while`, and `for`, `break`, `continue`, and `return` are the same as C. When the function has a non-`void` return type, the type of the expression in the `return` must be implicitly convertible to the function return type as defined in section 4.6.

4.9 Functions and Global Variables

Programs in Blis consist of forward function declarations, function definitions, and global variable declarations. Global variable declarations are the same as local variable declarations as defined in the last section. Functions in Blis are declared and defined as they are in C:

```
int my_cool_function(int a, vec3[3] b);

// ...

// formal parameters must match
int my_cool_function(int a, vec3[3] b) {
    // ...
    return 42;
}
```

The body of a function consists of a block of statements as defined in the previous section.

In addition, formal parameters may have an optional `out` or `inout` specifiers, similar to GLSL. At most one of `in` or `inout` can be added before the type of the formal parameter. Function parameters are always pass-by-value, but if `out` is specified, the reverse from the default happens: instead of the actual parameter being copied into the formal parameter at the beginning of the function, the formal parameter has an undefined value at the beginning of the function, and is copied to the actual parameter when returning from the function. `inout` is like `out`, except that the formal parameter is also initialized to the actual parameter at the beginning of the function, like normal parameters; in some sense, an `inout` parameter is a combination of the `out` and normal parameters. We follow GLSL in specifying that the order that these copies happen is undefined. In some sense, the behavior of `out` and `inout` is similar to pass-by-reference in that updates inside the function become visible outside, except that aliasing is not supported and so passing the same argument twice produces different results from what you would expect. For example, in this snippet:

```
void my_cool_function(inout int a, inout int b) {
    a = 2;
    // b still has value 1
    b = 3;
}

// ...

int c = 1;
my_cool_function(c, c);
// what is c now?
```

since the order in which `a` and `b` are copied to `c` after calling `my_cool_function` is undefined, `c` can be either 2 or 3 at the end.

4.10 Names and Scope

Similar to C, Blis has support for nested scopes. Each block of statements defines a new inner scope, which contains any bindings from the outer scope which aren't hidden by another binding from the inner scope. Variables and functions cannot be referenced before they are declared. When multiple variables are declared at the same time in a single statement, the first variable is visible when initializing the second, the second is available when initializing the first, and so on. Blis does not support separate linking, so there are no separate storage classes beyond local vs. global variables.

4.11 GPU specifics

The GPU-specific objects in Blis are *shaders*, *buffers*, *images*, *samplers*, *textures*, and *framebuffers*. There are two types of shaders, *vertex shaders* and *fragment shaders*. Vertex shaders run first, consuming buffers of `vertex attributes` to produce triangles that are then split into *fragments* that are given a color by the fragment shader which is written out to the framebuffer which collects the final result. Together, the whole process is encapsulated in the *pipeline*, which includes the vertex shader, the fragment shader, and struct members which let users bind resources like samplers and framebuffers which may be used in the process. Images and buffers are opaque structures which represent memory local to the GPU arranged in a one-dimensional or two-dimensional fashion. Images can either be bound to framebuffers or textures. Textures and samplers are the opposite of framebuffers; they are 2D arrays that can only be read by the shader. A sampler is a fixed-function piece of hardware which knows how to sample a texture at an arbitrary floating-point location by interpolating between nearby pixels in the texture using one of a few different interpolation modes. Each texture may contain one or more images, which represent different *levels of detail* or *LOD's*.

4.11.1 Shader Entrypoints

To tell Blis that a particular function is meant to be compiled to a *shader* that executes on the GPU, it can be given a `@fragment` or `@vertex` annotations after the declaration but before the function body. Shader entrypoints are similar to normal functions, and they can call the same functions that normal functions can, except there are a few restrictions on shader entrypoints and any functions called by them:

- Recursion is disallowed. That is, the callgraph of functions called by each shader entrypoint must not be cyclic.
- Runtime-sized arrays are not allowed.
- Any GPU-specific objects like samplers, images, etc. cannot be created.
- Global variables, as long as they do not contain any GPU-specific types, can be read, but they cannot be written.

4.11.2 Pipelines

4.12 Builtin Functions

5 Formal Grammar

6 Complete Sample Program

```
/* Draws a spinning triangle */

mat3 transform;

void generate_transform(float angle) {
    transform = mat3(cos(angle), -sin(angle), 0,
               sin(angle), cos(angle), 0,
               0, 0, 1);
}

/* The vertex shader is responsible for transforming
 * triangle vertices into screen space. "color" and
 * "pos" are per-vertex inputs, called "attributes" in
 * OpenGL. "color" is a per-vertex output that gets
 * interpolated across the triangle, called a "varying"
 * in OpenGL, while "position" is a builtin that
 * determines where the triangle is rendered.
 */
void vertex(vec3 color, vec3 pos, out vec4 color_varying, out vec4
    position) @vertex {
    /* Note that here, we're pulling in "transform", which was
     * declared as a global variable outside the pipeline. This
     * is a simple way of passing data that doesn't change per-vertex
     * to the shader. In OpenGL, this would require creating a
     * uniform, binding it, and updating it per-draw if it changes.
     */
    color_varying = vec4(color, /* alpha */ 1),
    position = vec4(transform * pos, /* w coordinate */ 1);
}

/* The fragment shader is responsible for determining
 * the color of the "fragment" (potential pixel).
 * "color_varying" is a varying input, matched by name with
 * "color_varying" from the vertex shader.
 */
void fragment(vec4 color_varying, out vec4 framebuffer) @fragment {
    framebuffer = color_varying;
}

/* This creates a pipeline, which includes everything necessary
 * to draw some triangles to the framebuffer(s).
 */
pipeline {
    /* tell the pipeline what vertex and fragment shaders to use */
    @vertex vertex,
    @fragment fragment,

    /* The "buffer" type is a way to represent data stored on the GPU.
     * Buffers cannot be accessed directly, they can only be uploaded
     * to or downloaded from. The type of the buffer, "vec3", means that
     * data is stored uncompressed in GPU memory as an array of vectors of
     * 3 floating point numbers (compare to the framebuffer below). This
     * particular buffer is used by the vertex shader as an attribute,
     * since the name matches the parameter of the function, which means
     * that assigning to this member will bind the buffer to the
     * Vertex Array Object (VAO) of the vertex shader in OpenGL terms.
     */
     Buffer<vec3> color,

     /* similar to the above */
     Buffer<vec3> pos,

    /* The framebuffer is what accumulates the final
```

```
     * color and depth. RGBA8 indicates that the pixels should
     * store their red, green, blue, and alpha as 8-bit integers
     * where 0 maps to 0.0 and 255 maps to 1.0, and is part of the
     * type.
     */
    framebuffer: Framebuffer<RGBA8>
} MyPipeline;

int main() {
    MyPipeline my_pipeline;
    /* transfer data from the CPU to the GPU */
    Buffer<vec3> color, pos;
    color.upload([vec3(1.0, 0.0, 0.0),
                  vec3(0.0, 1.0, 0.0),
                  vec3(0.0, 0.0, 1.0)]);
    pos.upload([vec3(-1.0, -1.0, 0.0),
                vec3(1.0, -1.0, 0.0),
                vec3(0.0, 1.0, 0.0)];

    /* prepare the pipeline */
    my_pipeline.color = color;
    my_pipeline.pos = pos;

    /* this creates a window and the framebuffer for it */
    Window my_window = Window(1024, 768, "My␣Window");

    /* tell the pipeline to draw to the window's framebuffer */
    my_pipeline.framebuffer = my_window.framebuffer;

    /* main loop */
    float angle = 0;
    do {
        angle += 1.0;
        generate_transform(angle);
        draw(my_pipeline);
        swap_buffers(window);
    } while (!should_close(window));

    return 0;
}
```