Blis: Better Language for Image Stuff

Project Proposal

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 8, 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 Parts of the language and what they do

- Data types: Integers (int), floating point numbers (float), single and multidimensional arrays, matrices (up to 4x4) (matrix), strings (string) – for file I/O and writing
- Built-in cos, sin, exp, and log functions
- Loops (for and while), if and else statements
- User-defined functions
- Comments
- Buffer/Image opaque types for data on the GPU
  - Represent data stored on the GPU
  - Only way to interact is through uploading/downloading to arrays
- Shaders
  - Functions annotated for use on the GPU
  - Arguments represent resources
- Pipeline built-in type for actual rendering
  - Can "bind" shaders, buffers and images to the pipeline for use as inputs/outputs

4 Sample Code

```
/* Draws a spinning triangle */

var transform: mat3;

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

/* This creates a pipeline, which includes everything necessary
 * to draw some triangles to the framebuffer(s).
 */
Pipeline {
    /* 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.
     */
    vertex_shader: fn @vertex (color: vec3, pos: vec3) ->
        (color: vec4, position: vec4) {
        /* 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.
         */
        return (vec4(color, /* alpha */ 1),
                vec4(transform * pos, /* w coordinate */ 1));
    },

    /* The fragment shader is responsible for determining
     * the color of the "fragment" (potential pixel).
     * "color" is a varying input, matched by name with
     * "color" from the vertex shader.
     */
    fragment_shader: fn @fragment (color: vec4) -> (framebuffer: vec4) {
        return color;
    },

    /* 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.
     */
     color: Buffer<vec3>,

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

    /* 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>
} 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 = make_window(1024, 768, "My␣Window");

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

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