Stochastic Search-Based Testing for Uniform Block Layouts

Ian Romanick *

Abstract

Software testing is a challenging problem. Testing uniform block layouts for the OpenGL Shading Language in particular is a testing problem at which the industry as a whole has failed. Hand-written tests developed by a single person or a small group of people are unlikely to discover complex cases that encounter implementation defects. This white paper describes an implementation of stochastic search-based testing for uniform block layouts.

1 Introduction

Software testing is, traditionally, a difficult problem. The problem is typically formulated in one of two ways: ensure that the software adheres to a specification, assuming there is a specification, or find any defects in the software. The first formulation is generally easier, and often leads to the development of "negative" tests. These are tests that verify correct errors are raised for classes of invalid inputs to the system. The second formulation is both more difficult and more interesting. Fully stating the problem, given a potentially infinite set of inputs, find the inputs that produce incorrect outputs. The restatement of the second formulation better illustrates the challenge.

Finding the set of inputs that produce incorrect outputs is more interesting because users of a software system do not typically provide invalid inputs to the system in day-to-day use. More often users provide differing sets of valid inputs. Different users will use a software system in different ways, and as a software system ages, trends and best-practices for the system will evolve. The input space of the software is gradually explored, by users of the system, through time. As a result, it is not uncommon for "day one" defects to remain undiscovered until many years of heavy use have passed.

When a defect is discovered by a user, two treated pieces of work occur. Software engineers analyze the inputs to determine the root cause of the failure, and test engineers analyze the inputs to isolate a minimal set that reproduces the failure. These work items can occur in either order or at the same time, and on many projects the software engineer and the test engineer are the same person.

The minimal set that reproduces the failure is often called the minimal reproducing case or the minimal test. The minimal test is extremely valuable from a software maintenance perspective for several reasons. The minimal test can be included in the software's test suite to both ensure that the defect is not reintroduced and to track when fixes are applied to maintenance versions of the software. The minimal test can also be helpful in finding the root cause of the original failure.

The previous points lead to a interesting conclusion: testing is a search problem. It is, however, an incredibly difficult search problem. Consider the set of possible inputs to a programming language compiler. Any nontrivial programming language is uncountably infinite. Given two arbitrarily close inputs, one may result in incorrect output while the other does not. In addition, a failure generated by a large, complex input may also be reproduced by a small, simple input. In terms of the original search problem, the large, complex input is not the desired solution, but the small, simple input is.

The resulting search problem does not fit well with traditional search techniques. The input space is infinite, arbitrarily close inputs may be inside and outside the search set, and not all inputs that meet the primary criteria (e.g., produces incorrect output) are in the search set. For traditional search techniques to work, it generally needs to be possible to measure two inputs to determine which is closer to the desired set.

Problems of this nature, however, are known in the search community, and techniques have been developed to solve these problems. One very well researched technique is randomized search, which may also be called stochastic optimization. There are a large variety of techniques, and [Zabinsky 2010] provides a good survey. One common theme among techniques is the use of randomness to pick either the initial search candidate, a new search candidate, or both. Additionally, randomized search algorithms will use a fitness function to evaluate candidates.

It is important to be aware of the trade off made by a randomized search algorithm. Some optimization problems may have exact solutions obtainable in exponential time. Many of these problems can be solved more quickly using randomized techniques. The quality of the solution, the execution time, or both may only be bounded with some statistical certainty. Randomized solutions to the traveling sales person problem find a solution close to the exact solution with high probability in linear time. Randomized variations of QuickSort use worst-case $O(n \log n)$ execution time with "overwhelming probability" [Mitzenmacher and Upfal 2005] to perform an exact sort.

Uniform block layouts in the OpenGL Shading Language (GLSL) is one area in dire need more robust testing across the industry. The language defines an application binary interface (ABI) called `std140`, see page 68 of [Segal and Akeley 2009]. Application developers rightfully expect that a uniform block using `std140` will have idential layout on every implementation. Applications are coded knowing that each field in a `std140` uniform block will have a specific location and organization. OpenGL implementations must adhere to this ABI. This is, however, notoriously not the case. One Twitter user proclaimed, "[Uniform block] introspection with GL: don't do it kids. Every driver has its own way of doing things. std140 layout gives no guarantees either.[1]" This sentiment is echoed on various message boards around the Internet.

All of the previously mentioned qualities of programming languages are present in uniform block layouts. To address this problem, randomized search techniques have been applied to uniform block testing. The remainder of this paper is divided into three sections. The method for generating tests is discussed in section 2. Section 3 presents results already obtained using the test generator. The paper concludes in section 4 with some areas of future work.

*e-mail: ian.d.romanick@intel.com or idr@freedesktop.org

[1] `https://twitter.com/paniq/status/415102722694606849`

2 Random Test Generation

Many random search techniques have already been applied to software testing, and [McMinn 2004] provides a good, though perhaps dated, survey as does [McMinn 2011]. The most applicable techniques presented there involve analyzing a specification of the possible inputs to the system to automatically generate test vectors that are likely to generate failures. Creating such a system is nontrivial. Sufficient experience with uniform block layout problems exists, and this existing experience was explicitly embedded in the test generation model.

For the purpose of testing uniform block layouts, a fairly straightforward multi-start algorithm was selected. Multi-start algorithms are separated into a global phase and a local phase. In the global phase a candidate point is selected from the entire solution space. In the local phase, the neighborhood around the candidate point selected in the global phase is explored. When the local phase terminates, a local optimal solution is produced. The best of all the local optimal solutions is finally selected as the global optimal solution.

Multiple candidate points selected in the global phase can result in the same local solution. This is a well known flaw of multi-start algorithms, and several enhancements have been proposed, including BRST [Boender et al. 1982] [Wikipedia 2014]. As will be shown in section 3, this flaw is not a particular concern for this use.

Multi-start algorithms vary on two fronts. In the global phase, algorithms vary in the model used to select initial candidates. Likewise, in the local phase, algorithms vary in the method used to explore the neighborhood around the global candidate. The variety of these methods is staggering. [Zabinsky 2010] provides some background.

2.1 Global Phase

A few areas of uniform block layout support have been observed, in shipping implementations, to be likely to contain defects.

- Row-major matrix layout qualifiers
- Padding around structures of certain sizes (e.g., less than 16 bytes)
- Nesting of structures inside other structures
- Nesting of arrays inside structures and vice versa

These areas form the basis of the model used to generate the initial candidate in the global phase. Each initial candidate contains a single uniform block with at least one special member. The special member is generated by first selecting a matrix layout, which can be one of default, row-major, or column-major, followed by a nonempty sequence of array or structure. The sequence is optionally terminated by a basic GLSL data type. For example, the sequence "row-major, structure, structure, array, structure, array, mat2x4" says the uniform block will contain a row-major decorated structure containing a structure containing an array of structures containing an array of 2-column-by-4-row matrices.

Additionally, the top level of the uniform block and each (possibly nested) structure inside the uniform block will be populated with a random selection of basic GLSL data types. Figure 1 is an example of the structures and uniform block generated by the test generator.

2.2 Local Phase

Exploration of the neighborhood around the initial candidate proceeds in two ways. Some initial exploration occurs immediately. Additional exploration of the neighborhood occurs after evaluating the initial candidates for fitness.

The initial candidate uniform block is, before any examination for fitness, modified to generate two additional uniform blocks. Each of these uniform blocks is intended to have the same layout as the original, but the layout is specified in a different manner. The initial uniform block utilizes the global, default column-major matrix layout. Each of the two modified uniform blocks inverts this default by either requesting row-major in the uniform block definition or by requesting row-major at global scope. The structures and matrices embedded in the uniform block must therefore have their explicit layouts modified to maintain the previous layout. Common modifications are from "default" to column-major and from row-major to "default."

Additionally, the initial uniform block requests `std140` layout in the uniform block definition. One of the modified uniform blocks diverges by requesting `std140` layout at global scope and by being declared as an array of blocks.

The test case as a whole is then evaluated for fitness. Unfit tests (i.e., tests that do not demonstrate a defect) are not explored further. These tests are, however, stored for later use. The neighborhood around a fit test is explored by gradually reducing the test to a minimal test.

A fit, non-minimal test case is pruned by selecting a random member and either removing it or, if the membmer is any array, minimizing it. Arrays with more than one element are converted to any array of a single element, and single element arrays are converted to non-arrays. When the last member of a structure or uniform block is removed, the structure and all uses of the structure or uniform block are removed.

During minimization, a count is kept. Each time a test is succesfully reduced (i.e., a member is removed and the test continues to demonstrate a defect), the counter is reset. Each time a test is not succesfully reduced, the counter is incremented. Once the counter reaches a predetermined value,[2] the minimization process terminates.

A deterministic minimization process was also considered. The deterministic process would need to remember which elements had been previously attempted to remove or minimize. The implementation is a collection of simple scripts, and remembering this information would have been difficult. The randomized process likely requires more attempts at minimization and may get stuck at a local solution. For the purposes of this project, ease of implementation was deemed more important.

After the automated minimization completes, there are still a couple simplicifcations that may be performed by hand. For example, remaining structures may be demoted to non-structures. In this case the fields of `S4` in figure 2 may be embedded directly in `S5` to produce the structure in figure 3. Additionally, local row-major or column-major declarations may be pushed up to the block level.

[2] 100 has been used thus far

```
struct S1 {
    bvec2       bv1;
    float       f1;
    uvec2       uv1;
};

struct S2 {
    mat4x2      m42_1;
};

struct S3 {
    S2          s2_1;
};

layout(std140) uniform UB {
                                // base   base   align  padded  row-   array   matrix
                                // align  off.   off.   size    major  stride  stride
    S1[1]       s1_1;           //   16     0      0      32      -      32      -
//    [0]
//    bvec2       bv1             8      0      0      8       -      -       -
//    float       f1              4      8      8      4       -      -       -
//    uvec2       uv1             8      12     16     8       -      -       -
    S3          s3_1;           //   16     32     32     64      -      -       -
//    S2          s2_1            16     32     32     64      -      -       -
//      mat4x2      m42_1         16     32     32     64     no      -      16
};
```

Figure 1: *Structures and uniform block generated by the test generator*

```
struct S4 {
    float r;
};

struct S5 {
    S4 s1;
    float g;
    float b;
    float a;
};
```

Figure 2: *Two structures*

```
struct S5 {
    float r;
    float g;
    float b;
    float a;
};
```

Figure 3: `S5` *with* `S4` *"demoted"*

2.3 Test Case Details

Test cases are implemented using the shader-runner functionality of the piglit[3] test suite. To implement these tests, shader-runner was augmented with the ability to probe the layout of uniform blocks.[4]

Each generated test has four parts:

- Declaration of all uniform blocks in the vertex shader.
- Probe of selected uniform block members in the vertex shader.
- Introspection of all API-visible data for all uniform block members.
- Declaration of data values for all uniform block members.

Shader-runner was additionally modified to allow specification of floating point values as IEEE 754 formatted hex values.[5] The values specified in the vertex shader and in the uniform block initialization are bit exact, and the value comparisons are also bit exact.

The test proceeds by compiling and linking the shaders, probing the uniform block member layouts, and running the shaders. As is typical for shader-runner tests in piglit, the shader will draw green for a passing test or red for a failing test. The colors are probed by shader-runner, and shader-runner will produce text output that is easily processed by other tools.

[3] `http://piglit.freedesktop.org/`

[4] `http://lists.freedesktop.org/archives/piglit/2014-September/012510.html`

[5] `http://lists.freedesktop.org/archives/piglit/2014-September/012516.html`

3 Results

Testing using the generator began in early September 2014 while the generator was still under development. Straight out the gate, the generator was able to detect defects in several shipping implementations. These early discoveries reinforced the community knowledge the uniform block support is generally bad across the industry. It also hampered development of the generator as it is difficult to determine whether failing test cases were due to defects in the generator or defects in the implementation.

Two closed-source OpenGL implementations, NVIDIA version 331.89 on a GTX260 and AMD version 13.20.11 on a Radeon R9 270X, and one open-source OpenGL implementation, Mesa 10.3-devel on Intel HD Graphics 4000, were tested. All tests were performed on Linux using X Windows. The following defects were discovered:

- Global layout declarations are completely ignored on NVIDIA.
- Matrix layout qualifiers on structures are not propagated to matrices inside the structures on NVIDIA.
- Members following structures that are less than 16-bytes are not properly aligned to a 16-byte boundary on NVIDIA.
- Non-square matrices with row-major layout are misreported to the API on NVIDIA. Specifically, a 4-column-by-3-row row-major matrix is reported as 3-column-by-4-row row-major.
- Block members that are not accessed by any shader stage are not reported by the API on AMD.[6]
- Arrays of two-element vectors or scalars do not have 16-byte stride on AMD.
- Use of a Boolean member as a conditional does not work on Mesa.[7]
- Row-major matrices inside arrays of structures are not accessed properly on Mesa.[8]
- Uniform block arrays (not arrays in a uniform block) crash the linker in Mesa.[9]
- Inside a structure, members following structures that are less than 16-bytes are not properly aligned to a 16-byte boundary on Mesa.[10] This is a more restricted case of the previously mentioned NVIDIA defect. The defect case on NVIDIA works properly on Mesa.
- Members following arrays of structures, where each structure is larger than 16-bytes but not a multiple of 16-bytes, are not properly aligned when the uniform block has an instance name on Mesa.[11]
- Inside a structure, members that follow a structure containing row-major matrices are not accessed properly on Mesa.[12]

[6] This defect caused almost every test case to fail, and this greatly hampered testing AMD's implementation.

[7] https://bugs.freedesktop.org/show_bug.cgi?id=83468

[8] https://bugs.freedesktop.org/show_bug.cgi?id=83506

[9] https://bugs.freedesktop.org/show_bug.cgi?id=83508

[10] https://bugs.freedesktop.org/show_bug.cgi?id=83533

[11] https://bugs.freedesktop.org/show_bug.cgi?id=83639

[12] https://bugs.freedesktop.org/show_bug.cgi?id=83741

At least a few of the defects discovered in Mesa are very specific. It seems unlikely that even a very savvy test engineer would have thought to test those cases.

Iteration of the global and local phases continues until a threshold number of minimized, fit test cases are identified. This generally results in a large number of test cases that reproduce the same defect. During development of the generator, this often led to changes in the generator itself. If the discovered defect was in the generator, the generator would be fixed. If, on the other hand, the discovered defect was a system problem in the implementation (e.g., the linker crash in Mesa), the generator would be modified to not trigger that defect.

In other cases, the defect in the implementation was fixed. The updated implementation was then expected to pass all of the tests that previously passed (the unfit test cases) and at least a subset of the of the tests that previously failed (the minimized, fit test cases). All of the newly passing tests would then be regrouped with the other unfit tests.

4 Conclusion

While still not complete, this work has already borne fruit. Many defects have been found in several OpenGL implementations, and several of these defects have already been fixed.

There are still areas for future improvement. The test generator currently only supports GLSL 3.30 with incomplete support for a couple extensions. GLSL 4.00 and GLSL 4.30 add several features that increase the complexity of the uniform block layout rules. All of these features need to be tested as well.

The test generator is also currently limited in the way that arrays are tested. The primary goal was to test the layout of uniform blocks. As such, arrays in uniform blocks are always accessed using a constant index. To better test the implementation's code generator, and to better match real-world use of uniform blocks, it should be possible to generate tests that index arrays dynamically.

Finally, the test generator needs to be better integrated with regular testing. At the time of writing, only tests that failed at one time have been integrated into the piglit test suite. Otherwise the generator has only been used by the author on a small set of platforms. This ignores the possibility that, for example, the Mesa driver for an AMD GPU may have defects that do not exist in the Mesa driver for an Intel GPU or even between different GPUs from the manufacturer.

The fundamental difficulty in integrating the generator with a test suite that is regularly run by developers is the potentially unbounded run time required to find a defect. In a defect-free implementation, the generator would run forever without discovering a fit test case.

References

BOENDER, C., RINNOOY KAN, A., TIMMER, G., AND STOUGIE, L. 1982. A stochastic method for global optimization. *Mathematical Programming 22*, 1, 125–140.

MCMINN, P. 2004. Search-based software test data generation: A survey. *Software Testing, Verification & Reliability 14*, 2 (June), 105–156.

MCMINN, P. 2011. Search-based software testing: Past, present and future. In *Proceedings of the 2011 IEEE Fourth Interna-*

tional Conference on Software Testing, Verification and Validation Workshops, IEEE Computer Society, Washington, DC, USA, ICSTW '11, 153–163.

MITZENMACHER, M., AND UPFAL, E. 2005. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press.

SEGAL, M., AND AKELEY, K. 2009. *The OpenGL Graphics System: A Specification (Version 3.2 (Core Profile))*. April.

WIKIPEDIA, 2014. BRST algorithm. `http://en.wikipedia.org/wiki/BRST_algorithm`, [Online; accessed 16-September 2014].

ZABINSKY, Z. B. 2010. *Random Search Algorithms*. John Wiley & Sons, Inc.