Metamorphic Shader Fusion for Testing Graphics Shader Compilers

Dongwei Xiao, Zhibo Liu, and Shuai Wang*
The Hong Kong University of Science and Technology, Hong Kong, China
{dxiaoad, zliudc, shuaiw}@cse.ust.hk

2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) | 978-1-6654-5701-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE48619.2023.00201

***Abstract*—Computer graphics are powered by graphics APIs (e.g., OpenGL, Direct3D) and their associated shader compilers, which render high-quality images by compiling and optimizing user-written high-level *shader programs* into GPU machine code. Graphics rendering is extensively used in production scenarios like virtual reality (VR), gaming, autonomous driving, and robotics. Despite the development by industrial manufacturers such as Intel, Nvidia, and AMD, shader compilers — like traditional software — may produce ill-rendered outputs. In turn, these errors may result in negative results, from poor user experience in entertainment to accidents in driving assistance systems.**

This paper introduces FSHADER, a metamorphic testing (MT) framework designed specifically for shader compilers to uncover erroneous compilations and optimizations. FSHADER tests shader compilers by mutating input shader programs via four carefully-designed metamorphic relations (MRs). In particular, FSHADER *fuses* two shader programs via an MR and checks the visual consistency between the image rendered from the fused shader program with the output of fusing individually rendered images. Our study of 12 shader compilers covers five mainstream GPU vendors, including Intel, AMD, Nvidia, ARM, and Apple. We successfully uncover over 16K error-triggering inputs that generate incorrect rendering outputs. We manually locate and characterize buggy optimization places, and developers have confirmed representative bugs.

I. INTRODUCTION

Recent years have witted a growing demand for 2D and 3D graphics in production scenarios, including gaming, virtual reality, automated driving assistance systems, and robotics. It is challenging to develop the desired graphics rendering effects while also conforming to the hardware characteristics of the underlying platforms, such as desktop and mobile phones.

To ease the development hurdle of graphics applications, shader compilers translate user-written shader programs that describe hardware-independent graphical effects into low-level optimized machine code specific to the target hardware backends, such as NVIDIA GPUs. The inputs of shader compilers, shader programs, are often very complex in production. For instance, in typical shader programs, vertex coordinates are first determined using transformations like projection and scaling. Shader programs then decide each pixel's color and compute various accompanying visual effects.

To date, production shader programs often comprise hundreds to thousands of lines of code and are compiled and optimized using hardware-independent and dependent optimizations. Hence, compiling and optimizing shader programs require careful consideration, posing a unique challenge to hardware vendors (shader compilers are often shipped with GPU drivers). Since shader compilers have been applied in various scenarios, from entertainment industry to security-critical sectors like driving-assistance systems [9], bugs hidden in the shader compilers, in turn, can potentially lead to poor user experience or even catastrophic accidents. Recent research has constantly uncovered bugs in production shader compilers [18].

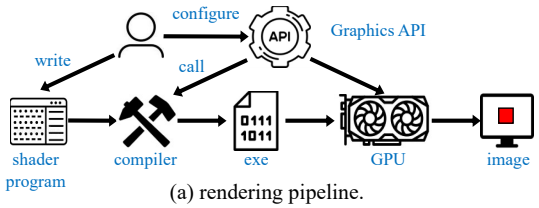
The test oracle problem is the major challenge for graphics compiler testing automation. A test oracle is the expected output of the program under test. In the context of 2D and 3D graphics, the test oracle of a shader program is hard to predict. Moreover, it has been shown that different GPU vendors output different results for some common math functions. For instance, the math function `fract`, which extracts the fractional part of a floating point number, gets different results on AMD and Apple GPU due to the inconsistent implementations across the two GPUs [11]. Additionally, the shader language definitions have ambiguities themselves. For example, the shader language of OpenGL allows a built-in function `dFdx` to return different values [23] depending on the underlying GPU implementation. The inconsistent output results across different GPU vendors, as well as the ambiguities of shader language definitions, cause the fact that even the same shader program can get distinct compilation results on different devices.

To address the test oracle problem for shader compiler testing, we propose FSHADER, a systematic and automated testing framework to uncover logic errors in shader compilers that induce incorrectly rendered outputs. FSHADER performs metamorphic testing (MT) [14], a testing scheme proved effective in combating the test oracle problem [15, 21, 47, 50, 53, 55, 59]. MT uses a set of carefully-designed metamorphic relations (MRs) to mutate test inputs and expose bugs by checking if any invariant program properties are violated. To test shader compilers, FSHADER *fuses* multiple shader programs into a new shader program following a set of novel MRs. It then checks the visual consistency between the image rendered from the *fused* shader program and the image generated by fusing rendered images from each shader program *individually*.

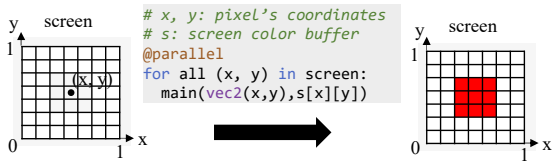
We implement FSHADER targeting three graphics APIs: OpenGL, Direct3D, and Vulkan. Our experiments cover 12 shader compilers, spanning the five leading GPU vendors (each ships its own shader compilers), including Intel, AMD, Nvidia, ARM, and Apple. Real-world shader compilers supplied on commercial devices exhibit a high engineering quality. FSHADER generates 12,000 fused shader programs to test

* Corresponding author

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00201



(a) rendering pipeline.



(b) working procedure of the shader program.

```
// geometry encoding
vec2 square[4] = vec2[](
  vec2(0, 0),
  vec2(1, 1),
  vec2(1, 0),
  vec2(0, 1)
); // four vertices
```

```
// transform geometry
for (int i=0;i<4;i++){
  square[i] *= 0.5;
  square[i] += 0.25;
}
// scale down by half
// then move to center
```

```
// assign color to each pixel
void main(in vec2 coord, out vec3 color){
  if (inside_square(coord))
    color.rgb = vec3(1, 0, 0); // red
  else
    color.rgb = vec3(1, 1, 1); // white
}
```

```
// coordinates inside square?
bool inside_square(vec2 coord){
  return coord.x > square[0].x
      && coord.x < square[1].x
      && coord.y > square[2].y
      && coord.y < square[3].y;
}
```

(c) a sample shader program, including geometry encoding, transformations, and a main function to assign color to each pixel.

Fig. 1: Graphics rendering pipeline.

shader compilers, and during approximately 20 days of testing, we detected 16,944 inputs that resulted in erroneously rendered images. While the discovered error-triggering inputs do not directly crash the shader compilers, they can lead to incorrect rendering results. We also found over 300 inputs that triggered crashes, indicating severe security issues like buffer overflow. We submitted all of our findings to the developers of shader compilers. As of this writing, representative cases have been confirmed. In sum, we make the following contributions:

- We target a crucial yet underexplored need to test shader compilers. Our novel testing oracle compares the visual consistency of images rendered from *fused* shader programs with the output of fusing each *separately* rendered image.
- Our testing framework, named FSHADER, features four MRs to fuse shader programs. We also incorporate various design principles and optimizations to mutate shader programs in an effective and systematic manner.
- We test industry-leading shader compilers on different commercial hardware platforms and find over 16K error-triggering inputs. The flagged defects are critical and can largely change the visual appearance of rendered images from human's perspective. Representative findings have been promptly confirmed by the vendors.

We release and will maintain the codebase of FSHADER at [1] to boost future research.

II. PRELIMINARY

Graphics APIs. *Rendering* is the process of generating 2D or 3D images by means of computer graphics programs. GPUs are typically the primary hardware for this task. To render graphical effects, graphics programmers interface with the GPU and instruct it on what operations to perform. Instead of writing GPU machine code directly, programmers use *graphics APIs* to convert high-level graphics descriptions in shader programs with user-provided input configurations (as in Fig. 1(a)) into GPU machine code. Graphics APIs provide a convenient approach for developers to render visual effects. OpenGL, Direct3D, and Vulkan are the three most popular graphics APIs for production usage. OpenGL and Vulkan are open-standard graphics APIs available on multiple platforms, including Windows, macOS, and Linux. Direct3D is exclusive to the Windows platform. OpenGL, Direct3D, and Vulkan are API specifications, whose implementations are on the shoulders of GPU vendors. GPU vendors such as AMD, Intel, and Nvidia may implement these APIs differently.

Rendering Pipeline Overview. Fig. 1(a) provides a simplified view of the rendering pipeline. The user writes shader programs to describe the desired rendering effects. The graphics API would then call the underlying shader compilers shipped by the GPU vendors to compile the shader programs into GPU machine code. The compiled GPU executable is then loaded into the GPU to perform rendering with user configurations (e.g., size of display window) as extra inputs.

To clarify, though the graphics rendering pipeline and the rendered outputs (e.g., an image) are influenced by both input shader programs and user configurations, user configurations are considerably less flexible and constrained. In practice, users build rendering effects mostly through shader programs. A non-trivial shader program typically contains hundreds or thousands of lines of code (LOC). Compiling and optimizing shader programs is thus intensive, making compilers the most error-prone component in the rendering pipeline. Given the importance of shader compilers in rendering, our testing approach aims to expose hidden logical bugs in shader compilers by mutating the shader programs as inputs to shader compilers.

Shader Programs. We illustrate how a shader program renders a pixel in Fig. 1(b). Holistically, the GPU will iterate over all pixels on the screen *in parallel*. The shader program will be executed for each pixel to compute its color, using the pixel's coordinates as input (denoted by `x,y` in Fig. 1(b)), and setting the pixel's color in the screen buffer (the `s` in Fig. 1(b)).

Fig. 1(c) presents a simplified shader program to render a red square. This program is written in GLSL, the shader language of OpenGL. In the first code snippet, the shader program represents the range of the red square by storing coordinates of the four square vertices in an array. A coordinate is represented as a 2D vector (`vec2`). The coordinates are often further processed with linear transformation to achieve the desired size and placing location for the rendered objects. In the second code snippet, the square is down-sized by half (`square[i]*=0.5`), and then moved to the screen center (`square[i]+=0.25`). Then, in the main function (third code snippet), the shader program yields different colors depending on the currently-rendered pixel's coordinates. In our case, pixels inside the square are

colorized in red while others are in white. `inside_square` decides if the currently-rendered pixel is inside the square or not by comparing the pixel coordinates against four vertices of the square. This way, the square is rendered by calling the shader program on each pixel.

The shader programs are written in different programming languages provided by different graphics API specifications. For instance, OpenGL uses a C-like language called GLSL, while Direct3D and Vulkan provide HLSL and SPIR-V as their shader languages. FSHADER primarily mutates programs written in GLSL. To test shader programs written in other languages, we use translators to convert GLSL to HLSL for Direct3D and SPIRV for Vulkan.

Shader Compilers. Production graphics APIs employ shader compilers to compile and optimize shader programs into GPU executables prior to rendering each pixel. Due to the unique characteristics of rendering pipelines and GPU computation primitives, shader compilers possess a series of complex optimizations that are absent from conventional C/C++ compilers. For instance, GPU has two kinds of Arithmetic-Logic Units (ALUs), namely the Scalar ALU and the Vector ALU, whose data access patterns diverge [17]. As such, the compiler must perform instruction dispatch optimizations to decide whether to issue instructions on scalar or vector units. It is thus demanding and challenging to test the correctness of complex compilation/optimization processes in modern shader compilers.

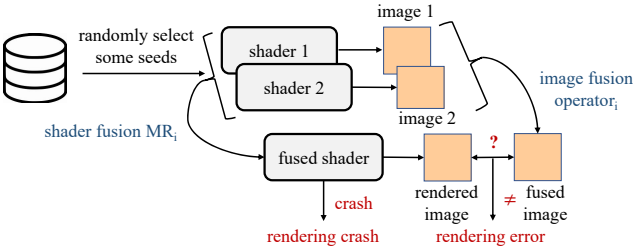


Fig. 2: Pipeline overview.

III. Overview

Overview of FSHADER. FSHADER fuses shader programs and asserts that the rendering output of the fused shader programs exhibits the same visual appearance as directly fusing the independently rendered images. A violation of such assertion indicates a potential bug in shader compilers. Fig. 2 depicts the FSHADER workflow. FSHADER features a set of carefully designed MRs (see Section IV-A). To test a shader compiler, we first collect a set of real-world shader programs as the seeds. Given two seed shader programs P_1, P_2, we fuse them into P_f using a randomly chosen MR MR_i. Meanwhile, we feed these two programs separately to the tested shader compilers and collect two output images I_1, I_2, respectively. Let the image $\hat{I}_f$ be the output of fusing I_1, I_2 following the strategy specified in MR_i, we assert that $\hat{I}_f$ should be visually identical to I_f, where I_f is the rendering output of P_f. Inconsistency between $\hat{I}_f$ and I_f implies bugs in shader compilers.

Study Scope. To clarify, FSHADER is designed to test shader compilers, which are essential in the rendering pipeline using production graphics APIs like OpenGL, Direct3D, and Vulkan. Typically, shader compilers are implemented by GPU vendors and are shipped with GPU drivers; therefore, shader compilers, even associated with the same graphics API, can have distinct implementations on different GPUs. FSHADER can expose the following defects in shader compilers:

- **Rendering errors** represent visual inconsistency when we employ the MRs in FSHADER to compare the visual consistency of rendered image I_f with the fused image $\hat{I}_f$. Inconsistency implies certain stealthy logic bugs hidden in tested shader compilers.
- **Rendering crashes** represent obvious errors, e.g., crashes or hangs during either the shader compilation or compiled shader execution phase. Memory-related errors when compiling specific shader programs might cause such crashes.

This work primarily aims to uncover *rendering errors* that result in incorrect rendering outputs. Generally, rendering errors can emit erroneous images silently. Following conventions in software testing literature, the rest of this paper will frequently refer to such defects as *"logic bugs"*. Note that our primary focus is not *rendering crashes*, given that fuzzing or regression testing may likely find such failures during in-house development. Nevertheless, we still record and report all our discovered rendering crashes (379 in total).

One may speculate if other components of the rendering pipeline may also contribute to the rendering errors; however, we believe the likelihood is low because the other components of graphics APIs, except shader compilers, are typically simple and offer mostly fixed functionality. Indeed, when manually inspecting rendering errors found by FSHADER, we are able to identify erroneous binary code fragments in the shader executable for all 900 studied error-triggering shaders.

Comparison with Existing Works. To the best of our knowledge, the only existing shader compiler testing work, GLFUZZ [18], is also based on MT. The key difference is that GLFUZZ inserts merely dead code into shaders, such as `if` statements with always-false conditions. Since dead code is never executed and imposes no effect on the rendering output, GLFUZZ asserts that rendered images from mutated shader programs should be identical to those from seeds. We compare FSHADER with GLFUZZ in the following three aspects.

① GLFUZZ and FSHADER detect *non-overlapping* defects, because GLFUZZ's dead code insertion strategy does not alter the seed shader program's semantics, and its oracle is based on the invariance between rendered images before/after mutation. In contrast, FSHADER deliberately fuses several shader programs into a single one (thereby changing the "semantics" of the seeds), and asserts that the rendered image is *visually identical* to that of directly fusing images rendered from individual seeds. We thus deem GLFUZZ and FSHADER to be *conceptually different*, and their error-triggering inputs should not overlap. For instance, FSHADER may fuse two images, I_1 and I_2, by placing I_1 in front of certain objects in I_2 (see details in Sec. IV-A). Such transformations will never occur in GLFUZZ. To some extent, this explains that though GLFUZZ was proposed in 2017 and acquired by Google [20], recent

shader compilers on production GPUs still contain numerous problems, as demonstrated by our analysis.

② In terms of the study scope, it is reasonable to assume that dead code inserted by GLFUZZ will mainly stress the shader compiler's optimization phases, particularly the standard dead code elimination pass. FSHADER can be more *comprehensive* to stress the compilation and optimization phases of shader compilers, as well as other stages in the rendering pipeline.

③ Moreover, we believe that FSHADER should be desirable in practical usage. In real-world programming, developers are unlikely to include a massive amount of dead code, such as thousands of (nested) `if` statements whose conditions are always false (as GLFUZZ may generate). Errors exposed by GLFUZZ may *not* necessarily reflect defects developers may encounter in daily usage. In contrast, developing shader programs typically involves combining several shader programs. Instead of writing a shader program from scratch, developers frequently reuse, extend, and combine online-available shader programs.[1] To draw a ball over grass, for instance, developers will likely find a shader program rendering grass first, then mix it with another shader program rendering a ball. MRs designed in FSHADER fuse shader programs from different angles, better reflecting how sample shader programs may be leveraged in real-world scenarios. We thus deem FSHADER to be more capable than GLFUZZ of exposing common bugs that users may encounter in daily usage. Nevertheless, we also admit that the unrealistic program transformations of GLFUZZ, such as dead code insertion, are still useful, as the corresponding hidden compiler bug patterns could also lead to unexpected output in daily program development.

Computer Vision (CV) Model Testing. CV has achieved substantial success in the era of deep learning. The software engineering community has recently proposed many research works on testing CV models and applications [41]. We clarify that testing CV models and shader compilers are two distinct and irrelevant topics. Unlike testing CV models, which mutates *images* whose outputs are often prediction labels, testing shader compilers mutates *shader programs* whose outputs are images.

IV. DESIGN OF FSHADER

We first introduce each MR in Sec. IV-A, and then present a three-step approach to fusing two shader programs in Sec. IV-B.

A. Design of MRs

FSHADER defines four MRs to fuse shaders; we present a sample usage for each MR in Fig. 3. The fused shader programs are usually much more complex, thus effectively stressing the entire rendering pipeline and exposing rendering errors. In the rest of this section, we elaborate on each MR.

Addition & Subtraction. As shown in Fig. 3(a), the first MR, MR_+, adds two images to generate a "fused" image. MR_+ accepts two images of identical dimensions and produces as output a third image of the same size in which the value of each pixel's RGB color is the average of the values of the corresponding pixel in both input images. The process is as:

$$I_f(i,j) = (I_1(i,j) + I_2(i,j))/2 \quad (1)$$

where I_f is the output image, I_1, I_2 are the two input images, and i, j refer to the x-, y-coordinates of each pixel. To fuse two shader programs, MR_+ averages the output of each shader program. Recall that, as introduced in Sec. II, the output of executing a shader is the color (in RGB form) of one pixel. MR_+ concatenates two shader programs P_1, P_2 by post-processing each pixel with the following conversion:

```
Pf(i, j) = (P1(i, j) + P2(i, j)) / 2
```

where P_f denotes the fused shader, and $P_f(i,j)$ denotes the color for the pixel at coordinate i, j. Thus, the image rendered from P_f should exhibit visual consistency with I_f. Violating such visual consistency implies a rendering error.

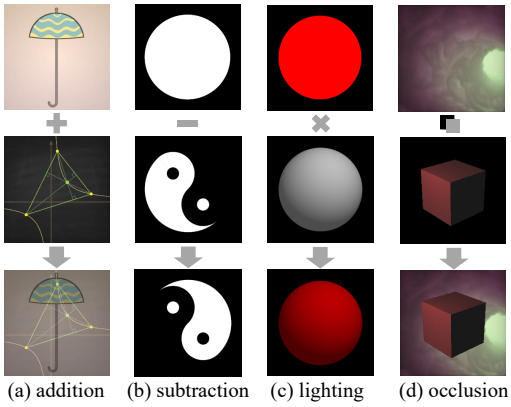
MR_- denotes another scheme to fuse images by differentiating the output color of each pixel over two shader programs. The closer the colors of the two images are, the darker the fused image will be. For instance, in Fig. 3(b), the white part of Taichi symbol in the middle image has the same color as the white circle in the upper image. Thereby, the white portion of Taichi would turn black after subtraction, creating a reversed Taichi in the bottom image. This scheme is implemented by subtracting one image from another across each pixel to produce the output image. A pixel's RGB value cannot be negative. Thus, we take the absolute value of the subtraction result. MR_- over two rendered images is therefore formulated as:

$$I_f(i,j) = |I_1(i,j) - I_2(i,j)| \quad (2)$$

Accordingly, to fuse two shaders for the effect of image differencing, we subtract the output pixel color of one shader from another, and take its absolute value via `abs`:

```
Pf(i, j) = abs(P1(i, j) - P2(i, j))
```

Image Lighting. The third MR, MR_l, is on the basis of image lighting. Simply speaking, the color of an object we see in real life is the light *reflected* from the object. Hence, the color of an object is related to both the *light* in the environment and the *reflectance factor* of the object (or *albedo*).



(a) addition (b) subtraction (c) lighting (d) occlusion

Fig. 3: Overview of proposed MRs.

[1]Our seeds are gathered from programs tagged as "hot" at shadertoy.com. We find that 71 out of the 100 hottest shaders reuse at least one existing shaders from others.

To understand this phenomenon, a piece of white paper under the sun looks to be white not because the paper is white by itself, but because the sunlight contains all colors of light, and the paper reflects all components in the sunlight. If the same piece of white paper is under blue light, it would appear to be blue since there is only blue light for it to reflect. The lighting model is formulated as follows:

```
perceived_color =  albedo x light_factor
```

We propose to imitate this lighting model to fuse two images. To do so, we take an image (e.g., the upper image in Fig. 3(c)) as the object, a 3D red ball.[2] Then, we take the middle image as the distribution of the light factor, which varies according to the object's surface. The part closer to white means more light is concentrated; the part closer to black means otherwise. Following the above lighting model, we multiply the image which functions as the object itself (the upper one) with the image describing the lighting factor (the middle one), and obtain the final appearance of the object that we perceive (the lower image). In Fig. 3(c), multiplying an image of a pure red ball with an image of the light distribution produces an image of a scarlet ball with a lighter red in its top right, and darker red in the bottom left.

In terms of fusing two shader programs in accordance with the lighting model, the core code snippet is:

```
Pf(i, j) = P1(i, j) * to_gray(P2(i, j))
```

where $P_1(i,j)$ denotes the output pixel color of the shader program whose rendered image serves the target object, and $P_2(i,j)$ is the output from another shader program whose output image servers as the lighting factor. We first convert the colors of the second image into gray-scale via `to_gray`, as we assume lighting factors are aligned across three primary colors (red, green, and blue) in daylight. Employing different lighting factors for each of the three primary colors is also feasible; we leave it as future work. Currently, the `to_gray` function simply averages the color values in the RGB channels.

Image Occlusion. MR_o explores another common phenomenon, namely image occlusion, by layering two images where parts of one image (e.g., an object) are extracted and placed on top of the other image. Considering Fig. 3(d), where we place the red cube in the middle image over the cloud (the upper image), whose result is shown in the lowest image.

Fusing two shader programs for the occlusion effect is not easy despite the straightforward concept. The key challenge is to somehow locate and extract the geometry shape of objects in one input shader program, and then place the extracted objects upon rendered content of the other shader. Proposing an automated and general way (without human intervention) is challenging, if at all possible: a simple cube's geometry can be encoded in multiple ways, let alone irregular shapes like clouds (i.e., the background image in Fig. 3(d)).[3]

[2]To clarify, the 3D ball looks like a 2D circle in the first image of Fig. 3(c) because all pixels of the 3D ball reflects the red light equally.

[3]For instance, some programmers define a cube using coordinates of the center point and the length of edge, while others define a cube by enumerating the coordinates of its six vertices.

```
1 if (outside_cube())
2   color = black
3 else
4   color = draw_cube()
```

Fig. 4: Extracted code snippet.

We implement occlusion based on an observation of real-world shader implementation: shader writers typically use `if` statements to draw distinct regions of the image [25, 27, 32, 38, 58], such as different objects, or an object and the background. Consider the middle image in Fig. 3(d), its shader program contains an `if` statement in Fig. 4, which selectively paints the cube or the black color, depending on the current coordinates. The programmer defines `outside_cube` to decide if the currently rendered pixel is outside the cube.

```
if (condition)
  Pf(i, j) = P1(i, j)
else
  Pf(i, j) = P2(i, j)
```

Fig. 5: Code for occlusion.

Let shader P_1 render the front object, whereas shader P_2 renders the object at the back. To implement MR_o, we randomly select an `if` statement from P_1, and insert P_2 into the `else` branch, as in Fig. 5. By such construction, when the condition is evaluated to be true for the current pixel under rendering, the pixel color is assigned to be the color from P_1, thus rendering the foreground; otherwise, the rendered pixel color is from P_2, hence showing the pixels from the background. To implement the occlusion effect for the Fig. 3(d) case, we instrument the `if` statement in Fig. 4 by replacing the "black" rendering at line 2 with the pixel color output by the shader program of "cloud."

For implementation, we randomly select one `if` statement from shader P_1. Then, we randomly choose one of the two branches in the selected `if` and replace the branch with shader P_2. Thus, when the replaced `if` branch is executed, the rendered pixel would be from P_2, otherwise from P_1.

$$I(i,j) \in \{I_1(i,j), I_2(i,j)\} \tag{3}$$

Regarding the testing oracle for the MR_o, we assert that any pixel $I(i,j)$ in the rendered image of the fused shader program should be identical to either the pixel $I_1(i,j)$ or $I_2(i,j)$. A violation of such assertion indicates a bug. The assertion is formulated in Eq. 3 above. To understand this oracle, consider all possible cases in Fig. 5: when the `if` condition is true for a pixel located at (i,j) on the image, P_2 would not be reached. Thus, the whole execution trace would be identical to the original P_1, as if P_2 were not inserted, and the output pixel color should be $I_1(i,j)$. When the `if` condition is false under pixel (i,j), even though the execution of P_1 can be tempered by P_2 [4], the final output would be from P_2 according to Fig. 5, meaning that P_1's tempered output is discarded and the output color should be identical to $I_2(i,j)$. Given these two cases, we conservatively assert that the rendered pixel color $I(i,j)$ should be identical to either $I_1(i,j)$ or $I_2(i,j)$.

Other Potential MRs. Given recent advances in CV testing, in which various ways of mutating images are proposed, one may wonder if those image mutation schemes can be borrowed to

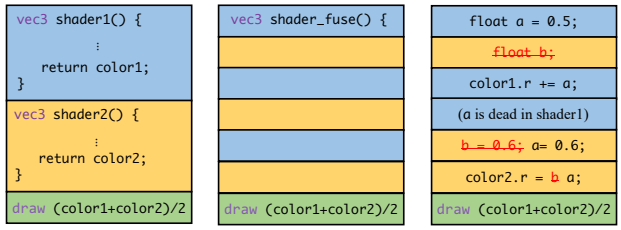
[4]We ensure that the program will exit normally by avoiding affecting loop termination conditions and arrays in P_1.

mutate shader programs. Below, we clarify that image mutations commonly seen in CV testing are not proper in our domain.

Linear transformations, like rotation, flipping, and scaling, are extensively used to mutate images in CV testing [35, 41, 53, 54]. Adopting them in mutating shader programs is technically feasible. However, these linear transformations are mainly achieved by matrix multiplications [10, 31, 34], which can be *easily* mapped to low-level GPU primitives during compilation [39, 42]. In preliminary study, we see these transformations hardly stress shader compilers. Contrarily, our MRs require shader compilers to optimize fused shader programs non-trivially, increasing the chance of exposing bugs.

Kernel-based image transformations apply a $N \times N$ kernel (e.g., $N = 5$) to transform $N \times N$ neighboring pixels at once. Kernel-based image transformations have achieved promising effects in CV testing [53], and it may impose higher challenges to shader compilers; however, their implementation over shader programs is obscure. Kernel-based implementations require access to the neighboring pixels. As introduced in Fig. 1(b), modern GPU renders every pixel in parallel, where reliable access to the neighboring pixels is generally not available.[5]

B. Fusing Shader Programs in a Three-Step Approach



(a) concatenate two shaders. (b) inline & stmt. shuffle. (c) variable reusing.

Fig. 6: Fuse and obfuscation steps.

We have explained four proposed MRs to fuse *outputs* of two shaders. Here, we further clarify how the fusion of two program bodies is implemented. Fig. 6 depicts a three-step approach to fusing two shader programs with increasing program complexity. Without losing generality, we use MR_+, image addition, to illustrate our program fusion approach in Fig. 6. For simplicity, we assume each of the two shader programs has one function, and we represent statements from two shader programs in either blue or orangle in Fig. 6.

First, we follow Fig. 6(a) to place the shader program code side by side, and place a wrapper function to fuse their output. Note that global variables and functions from two shaders have to be renamed to avoid name conflicts. Though shader compilers' implementations are obscure, our intuition is that this transformation can reasonably stress the inter-procedural optimizations of shader compilers.

Second, to complicate the fused shader and potentially stress intra-procedural optimization passes, we perform function inlining and interleave two shader programs' statements. In particular, as in Fig. 6(b), we first inline statements from the second shader into the first one. We *interleave* statements from two shaders. Notice the statement order within each individual shader program is retained to preserve original functionality. We skip loop statements, as interleaving statements in loops while retaining the original semantics is generally challenging.

Third, we implement an intra-procedural liveness analysis upon each fused shader function to determine local variables' liveness. Shader programs written in the OpenGL language do *not* use pointers, and therefore, def-use analysis is much easier than analyzing C/C++. Then, we perform *variable reuse*, reusing a recently dead variable to replace a variable from the other shader that is still live in the following statements. For instance, as in Fig. 6(c), we replace all the usages of variable b in shader2 with a from shader1. Given that a is dead in shader1 when replacing b, the replacement of two variables would not invalidate the data flow of two shader programs. Rather, we expect it will effectively complicate the data flow.

With the three-step instrumentation, two shader programs shall become much more complex than their original form, effectively stressing the compilation and (intra- and inter-procedural) optimization passes of shader compilers. Nevertheless, the functionality before instrumentation is still preserved.

V. Implementation

FSHADER is implemented in about 5K LOC [1], with 4K lines in Java for core functionality and 1K lines in Python for processing experiment data. We reuse the shader parser and some utility functions in the codebase of GLFUZZ [18].

Seed Fusion. The MRs presented in Sec. IV, by default, mutate two shader programs. Nevertheless, to further improve our testing comprehensiveness, FSHADER is implemented to fuse n shaders instead of only two.

We first prepare s sequences, each of which contains n seed shader programs randomly selected from our seed pool (see Sec. VI). When using MR_+ (same strategy for MR_- and MR_l), we conduct an iterative process, where the first and the second seed shaders are fused using MR_+, whose output is further fused with the third shader using MR_+. This way, each sequence generates $n-1$ fused shader programs for testing.

Fusing more than two shaders using MR_o, however, is not desirable, since MR_o's oracle (see Eq. 3) asserts the output pixel value to be either equal to the frond-end image or the back-end image. Thus, iteratively fusing n shaders means that each pixel in the last test input needs to compare with n pixels. This may lead to false negatives when pixels in the fused image happen to equal certain pixel values in the n set. Therefore, for MR_o, we perform pairwise fusion, i.e., we fuse the first shader with the second, then the second with the third, until all n shaders are fused pairwisely. This way, a sequence of n shaders also generate $n-1$ fused shaders for testing.

FSHADER is configured with $n = 4$, since fusing four shader programs can generate test inputs with sufficient complexity (with an average size of over 3K lines of code); the rendering pipeline is seen as slow when processing overly complex shaders. We configure s to be 1000. Given that we have four

[5]Although built-in functions like dFdx enable access to neighboring 2×2 pixels, they cannot be used when the shader program contains divergent control flow like if statements [10].

MRs in total, the total number of generated fused shader programs is thus $4s(n-1)$, which is $4 \times 1000 \times (4-1) = 12,000$.

Image Comparison. We compare each pixel value to decide if the two rendered images are the same. Note that certain small drifting in pixel values may not necessarily reflect a "bug" in the rendering process: as pointed out in [18], mutated (more complex) shaders may enable/disable certain optimization passes, such that the rendered output may be slightly different yet visually indistinguishable, as confirmed by previous work [18], compared to the original output. Thus, we leverage a threshold, such that two images are deemed visually different if their pixel value deviations are over a threshold. We use *exactly the same tactic* noted in [18] to decide the threshold. In short, we rely on a sef of non-color blind participants to decide whether pairs of images are visually different or not. We select thresholds such that all participants agree that pairs of images whose difference exceeds the selected thresholds are visually different. In the later manually debugging process of 900 defects, we confirm that all the compared images, when their deviations are above the threshold, are indeed visually different (see some examples in Fig. 13), whose defects root in compiler bugs (see Sec. VI-E).

VI. Evaluation

A. Evaluation Setup

Seed Programs. We collect shader programs written by experienced shader developers from shadertoy.com to form the test seeds. Many of the seeds programs involve advanced rendering techniques, such as ray marching. Moreover, a large number of the rendered objects, such as carton characters, city architectures, and robots, in the shaders have complex geometry structures. As a result, the shader programs in our seed corpus are usually of high quality. We download shader programs from the first 155 pages under the "Hot" tag, where each page has 12 shaders. We then manually rule out shader programs that meet any of the following criteria:

1) The shader program takes over two seconds to compile.
2) The shader program has less than 100 lines of code and does not involve advanced rendering techniques.
3) The rendered image of the shader program is of poor quality, such as containing excessive noise.
4) The shader program contains uninitialized variables.

We then randomly select 100 from the remaining shader programs as the seed pool. Then, following **Seed Fusion** mentioned in Sec. V, we generate 12,000 fused inputs.

Fusing all seed shaders takes 42 minutes on our PC with Intel i5-12400 CPU and 32GB RAM. The shader programs generated by our methods are more complex than the seeds. The average size of mutated shader programs is 3,855 lines of code (LOC), with the maximum as 30,166 LOC, while the size of seeds is 261 LOC in average.

Shader Compilers. We test shader compilers used by OpenGL, Direct3D, and Vulkan, all of which are the mainstream graphics APIs for rendering 2D/3D visual effects. FSHADER is implemented to mutate GLSL shader language used by OpenGL, and all collected seeds are written in GLSL. To test Direct3D compilers, we use a translator provided by Chrome, namely ANGLE [22], to convert GLSL shaders to Direct3D-compatible shaders. When testing the shader compilers for Vulkan, we use Glslang [24], a translator developed by The Khronos Group [3], to translate GLSL shader programs into programs in SPIR-V, a language that Vulkan accepts.

TABLE I: Details of tested GPUs.

GPU Device	Driver Version
Intel(R) UHD Graphics 730	31.0.101.3222
AMD Radeon(TM) RX 6400	22.7.1.220725
NVIDIA GeForce RTX 3060	516.94
ARM Mali-G52 MC2	v1.r26p0-01eac0
GPU on iPhone SE2	Not available

TABLE II: Combinations of GPUs and graphics APIs.

GPU	Graphics API		
	OpenGL	Vulkan	Direct3D
Intel UHD 730	✓	✓	✓
AMD RX 6400	✓	✓	✓
NVIDIA GeForce 3060	✓	✓	✓
ARM Mali-G52 MC2	✓	✓	NA
Apple GPU	✓	NA	NA

GPU Devices. Note that shader compilers are implemented by GPU vendors, and different vendors have their own implementation of shader compilers to match their own (proprietary) hardware. We choose five GPUs from five leading manufacturers in the industry, including Intel, AMD, NVIDIA, ARM, and Apple. Shader compilers implemented by each GPU vendor are often bundled with GPU drivers. We thus include the driver version of our tested devices in Table I. Testing GPUs of Intel, AMD, and NVIDIA are conducted on a personal computer with Windows 10 OS, 32GB RAM, and 1TB SSD. Testing ARM GPU is conducted on Coolpad Cool 20, a mobile phone equipped with Android 11, 4GB RAM, and 64GB storage. Apple GPU is tested on an iPhone SE2 with iOS 15.5, 4GB RAM, and 64GB storage. In Table II, we list the GPU and graphics API combinations of our tested shader compilers. Note that Android and iOS platforms do not support Direct3D, and iOS does not provide native support for Vulkan. As a result, we do not test ARM GPU for Direct3D, and Apple GPU for Direct3D and Vulkan (marked as "NA").

B. Experiment Results

Processing Time. As mentioned in Table II, we execute the generated fused shader programs on 12 combinations of GPU devices and graphics APIs. Fig. 7 reports the execution time for each setting. Vulkan could provide a performance boost compared to other graphics APIs; however, its execution time is greater than that of OpenGL due to the overhead of using a translator to convert OpenGL programs to Vulkan programs. Direct3D runs the slowest since it also requires the OpenGL

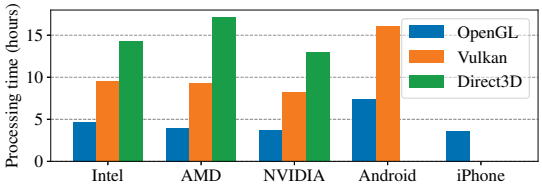


Fig. 7: Execution time.

TABLE III: #Erroneous test inputs on each tested combination.

GPU	Graphics API			Total
	OpenGL	Vulkan	Direct3D	
Intel UHD 730	1071	3350	1728	6149
AMD RX 6400	1165	2862	1792	5819
NVIDIA GeForce 3060	112	2419	1768	4299
ARM Mali-G52 MC2	109	159	NA	268
Apple GPU	409	NA	NA	409
Total	2866	8790	5288	16944

program to Direct3D program translation, and it does not enjoy the same speed benefit as Vulkan.

Summary of Findings. Table III summarizes the error-triggering inputs found in each setting. Recall in Sec. III that we record both "rendering errors", meaning erroneously rendered images, and "rendering crashes", meaning crashes that occurred during the rendering process. The "Erroneous test inputs" in Table III refers to "rendering errors". FSHADER uncovers over 16K error-triggering shaders, whose rendered images are inconsistent with the expected ones. All settings triggered considerable defects, showing that shader compiler defects are prevalent yet overlooked in production.

FSHADER uncovers most findings for Vulkan, totaling 8,790 cases across all devices. This is reasonable: Vulkan is a relatively newer graphics API than OpenGL and Direct3D [7], meaning its shader compilers may not experience as much in-house testing as the other two. Moreover, while Vulkan is designed to offer a greater performance boost than the other graphics APIs, Vulkan shaders are usually more complex, with over 1K lines of code for even an elementary rendering process [8]. Hence, traditional testing approaches, such as unit testing, can be hard to set up, leaving some hidden bugs undetected during the in-house testing stage. In contrast, our testing method is a black-box testing approach and the testing capability is thus not constrained by the code complexity of the testing target. A side note is that none of our tested shader compilers released their source code. Hence, black-box testing appears to be the only viable approach for non-staff people.

Though OpenGL and Direct3D have been introduced over a decade [2, 4], presumably with extensive internal testing by the vendors, their shader compilers nevertheless contain a substantial number of defects. We identified 2,866 and 5,288 error-triggering inputs in the OpenGL and Direct3D shader compilers, respectively. Sec. VI-E provides several examples to facilitate comprehension of our findings.

As clarified in Sec. III, FSHADER primarily targets "logic bugs", denoting the rendered images being inconsistent with the expected ones. Still, during testing, we find 379 crashes on OpenGL shader compilers, with 378 cases on Intel and one case on NVIDIA. Crashes can be due to memory-related bugs, such as memory access out-of-bounds. These findings illustrate that certain security exploitation may be launched toward the graphics pipeline; we leave attacking and hardening the graphics pipeline for future exploration.

TABLE IV: Number of deviant outputs found by each MR.

MR	MR_+	MR_-	MR_l	MR_o	Total
#Deviant Outputs	3,318	3,530	2,321	7,775	16,944

We further report the distribution of error-triggering inputs found by each MR in Table IV. Overall, it is evident that each of the MR contributes to detecting a large portion of flaws. We find that the MR_o (occlusion) is more effective than the other three MRs, in that it triggers close to 50% of errors. This can be due to the fact that the occlusion MR inserts a shader program into conditional branches of another shader, thus significantly mutating the seed program's structure. The substantially changed program structure may better stress the shader compilers and provoke more optimization passes that are not used when compiling individual seeds.

C. Comparison with GLFUZZ

Sec. I conceptually contrasts FSHADER with GLFUZZ in three aspects. Here, we launch an empirical comparison between FSHADER and GLFUZZ. We use the same 100 shader programs in Sec. VI-A as the seeds for FSHADER and GLFUZZ, and let them generate 600 mutated inputs using the their methods. All 600 mutated shader programs are then executed on all GPUs that support OpenGL and Direct3D, the two graphics APIs studied in the GLFUZZ paper. In total, eight combinations of GPU and graphics APIs are evaluated. The hyper-parameters of GLFUZZ are set to their defaults values, and all available transformations of GLFUZZ are enabled to maximize its bug-revealing ability.

TABLE V: Comparison experiment results.

	FSHADER	GLFUZZ
Mutation time	2min 32 sec	1min 14 sec
Average lines of a mutant	3,975	680
Rendering time	4.4 hours	6.5 hours
#Crashes	5	64
#Error-triggering inputs	356	40

Table V reports the comparison results. Both tools are rapid in mutating shader programs, and FSHADER generates mutants with relatively higher complexity. The majority of time (several hours) in the testing campaign is spent on executing shaders on GPU devices. To clarify, crashes often result in hanging GPUs, thereby prolonging the processing time of the GLFUZZ cases. GLFUZZ uncovers more crashes than FSHADER, whereas FSHADER finds more error-triggering inputs than GLFuzz, indicating its effectiveness in uncovering graphics rendering errors, the main focus of both tools.

To justify our statement in Sec. I that "FSHADER and GLFUZZ detect non-overlapping bugs", we study all error-triggering inputs uncovered here. We confirm that *none* of them overlap. Moreover, we examined the GLFUZZ's bug report repository at [19] and checked all bugs discovered by GLFUZZ in its paper. We confirm that none of their bugs overlap with the six bugs (see Sec. VI-E) uncovered by FSHADER.

D. Effectiveness of Three Program Transformations

Sec. IV-B presents three program transformations to stress shader compilers. This section launches an ablation study to study their effectiveness. The results are shown in Table VI.

In Table VI, "Fuse" refers to applying the three MRs without additional obfuscation (Fig. 6(a)). "Opt1" and "Opt2" refer to

TABLE VI: Results on program transformations.

Settings	# Errors uncovered
1. Fuse only	7,993
2. Fuse + Opt1	8,034
3. Fuse + Opt1 + Opt2	9,434
4. Fuse + Opt1 + Opt3	10,953
5. Fuse + Opt1 + Opt2 + Opt3	16,944
6. opt1	21

function inlining and *statement shuffling* (two transformations illustrated in Fig. 6(b)), and "Opt3" refers to *variable reuse* (illustrated in Fig. 6(c)). In general, Opt2 and Opt3 have negligible transformable statements without fusing shaders (Fuse) and inlining all the statements first (Opt1). Hence, in practice we study six ablation settings, as listed in Table VI.

Setting 6, which applies function inlining to a single shader program, only uncovers 21 error-triggering inputs. This finding justifies the necessity of fusing multiple shader programs. Setting 2 uncovers 41 more errors than setting 1, meaning that function inlining on fused programs slightly surpasses fusing programs alone. Setting 3 and 4 imply that both Opt2 and Opt3 are effective in detecting errors. The most effective setting is to apply all the transformations, as in setting 5, which uncovers the highest number of erroneous cases.

E. Characterize Shader Compiler Bugs

TABLE VII: Distribution of bugs in the Direct3D compilers.

Incorrect Optimization Place	AMD	Intel	NVIDIA	Total
`frac` function	84	73	87	244
Square expression	76	75	75	226
`round` function	52	46	49	147
`floor` function	58	50	55	163
Complex arithmetic	39	57	44	140
`If`-condition	1	1	2	4
Total	310	302	312	924

Challenges of Studying All Findings. Analyzing the root causes of shader compiler bugs faces a series of challenges:

1) Few GPU vendors allow users to retrieve compiled executables, likely due to intellectual property (IP) issues.
2) None of our tested commercial GPU drivers open-sourced their source code.
3) Debug support and toolchains are generally limited.

Overall, the rendering pipeline in production is highly complex and distinct from compiling and running conventional C programs. To the best of our knowledge, only Direct3D is shipped with a functional debugger (RenderDoc [6]), though it is not very handy to use, and certain essential debugging features are not in place.

It is generally impossible to analyze the root causes for all findings due to limited human resources. However, this should not be deemed a limit of our testing method, as vendors can better debug their code when adopting FSHADER (see Sec. VI-F for developer confirmation). Given that Direct3D has relatively mature debugging support, we randomly select 300 error-triggering inputs from each of our tested GPUs that supports Direct3D, i.e. the GPUs of NVIDIA, AMD, and Intel; we investigate totally 900 (300 each on 3 devices) erroneous cases. With debugger, we then manually localize and analyze the incorrectly compiled binary code fragment for all the 900 test cases. This step is costly; it takes an author *a month and a half* to analyze these bug-triggering inputs manually. After that, another author double-checked the bug analysis. Both authors are experts in software testing, compilers, reverse engineering, and experienced in graphics compilers.

As in Table VII, we successfully identified the buggy binary code fragment for all the 900 error-triggering inputs. The total number of root causes (buggy code fragments) uncovered is more than 900, since a test input may contain more than one buggy code fragment. We confirm that *all the 900 cases are true positives*, containing at least one piece of incorrectly compiled binary code. We attribute all buggy code fragments to six erroneously optimized code patterns. The `frac`, `round`, and `floor` are three built-in functions of shader programs. We found that the shader compilers can apply incorrect optimizations towards these functions, sometimes by wrongly replacing these function calls with a const number. "Square expression" in Table VII refers to shader compilers replacing the squaring expression (`h * h`) with a single non-squaring expression (`h`). This replacement results in a number of ill-rendered outputs. "complex arithmetic" in Table VII denotes incorrect optimizations on some more complex arithmetic computations, such as computing the polynomial of a variable. We also found a few bugs related to the wrong compilation of `if` statements, although these bugs are not as common as the other five categories, possibly due to strict conditions required to trigger the bugs (see Fig. 12 below). Since none of the GPU vendors release source code of their shader compilers, we cannot further localize root causes of such errors. Below, we present one representative code snippet for each of our six categories. The full code of error-triggering cases is at [5], enabling audiences to view all six bugs on their local machine.

```
1 int i, j; float t;
2 for (i=0;i<2;i++)
3   for (j=0;j<2;j++) {
4     t = iTime + 0.4;
5     if (frac(t)==0.0)
6       color = BLACK;
7     else
8       color = RED;
9   }
```

(a) Bug$_1$: `frac` function

```
1 float2 v; float l,h,c;
2 v[0] = 0.3;
3 v[1] = 0.4+sin(iTime);
4 l = length(v);
5 h = max(0.7 - l, 0.0);
6 c = 0.2 - h * h;
7 color.r = c / 0.16;
8 color.g = c / 0.16;
9 color.b = c / 0.16;
```

(b) Bug$_2$: `square` expression

Fig. 8: Two cases of discovered shader compilers bugs.

Bug$_1$*: Ill-Optimization for* `frac`. This bug is caused by the incorrect optimization of a built-in shader utility function, namely, `frac` [36], of Direct3D. `frac` enables extracting the decimal part of its input:

```
frac(x) = x - floor(x) // e.g., frac(3.1)=0.1
```

This bug is triggered only when fusing a shader containing `frac` with another shader containing a nested `for` loop. We hypothesize that the "synergy effect" of nested loop-related optimizations and `frac` function triggers this bug, which illustrates the subtleness of our findings. Considering the code snippet in Fig. 8a, where `iTime` is an input variable provided through calling graphics API, and is set to be 0 at runtime.

Hence, in line 4, t should be assigned as 0.4. Consequently, frac(t) yields 0.4, and frac(t)==0.0 is false. However, we find that the shader compilers incorrectly deem the output of the frac(t) as 0.0. Thus the output color is set as black.

Bug$_2$*: Incorrect Arithmetic Computation for Square.* In Fig. 8b, the compiler incorrectly optimizes a square expression. This code snippet declares variable v in float2 type (denoting a 2D float vector). The first element of v is 0.3, and the second element of v is set to 0.4, because iTime is 0.0 during runtime. At line 4, variable l stores the length of v (in Euclidean space), which is computed as $\sqrt{0.3^2 + 0.4^2} = 0.5$. Hence, the value of h should be $\max(0.7 - 0.5, 0.0) = 0.2$, and c (line 6) should be $0.2 - 0.2 \times 0.2 = 0.16$. The RGB components of the output pixel color are all $0.16/0.16 = 1.0$, denoting the color white. However, by inspecting the compiled executable, we find that the compiler incorrectly optimizes the statement c = 0.2 - h * h to be c = 0.2 - h, neglecting the square operation on h. Hence, c is $0.2 - 0.2 = 0.0$, causing the RGB components to be all zero and, consequently, the output pixel to be black.

Bug$_3$*: Ill-Optimization for* floor. floor is a commonly used built-in function that rounds a floating number down to its nearest integer. Consider Fig. 9, where the variable p is initialized with three floating numbers. Then, p * 4.0 is subsequently fed into the floor function. We find that the shader compiler incorrectly optimizes out the floor function and directly assigns u with p * 4.0. We confirm that o, r, and d are holding real numbers with non-zero decimal parts during runtime, and p * 4.0 is *not* equal to an integer. Thus, the optimization is flawed and produces a wrong value in t.

```
1 float p = o + r * d;       // o, r, d: float
2 float u = floor(p * 4.0); // floor is left out
3 float t = u * 5.0;         // Thus t is wrong
```

Fig. 9: Bug$_3$: floor function.

Bug$_4$: Ill-Compilation of round. Similarly, function round is another common utility function that returns the nearest integer of a floating number. Fig. 10 illustrates a finding of our manual study. The value of abs(t - round(t)) is used as one of the input arguments for a built-in function smoothstep, which returns an interpolation of $[0, 1]$ depending on the relation of its three inputs. We find that the compiler somehow deems t - round(t) to be always 0, thus assigning f with a wrong value. We confirm that variable t holds a floating number that is not exactly equal to an integer, and t - round(t) is not zero.

```
1 // The compiler wrongly infers t - round(t) is 0
2 f = smoothstep(0.0, 0.2, abs(t - round(t)));
```

Fig. 10: Bug$_4$: round function.

Bug$_5$*: Ill-Optimization of Complex Arithmetic Expressions.* A shader program comprises many complex arithmetic expressions of floating numbers. Shader compilers, to our knowledge, implement many passes to optimize floating number arithmetic expressions to improve rendering speed. However, such optimizations can be buggy for some complex expressions, such as the one shown in Fig. 11. The intended behavior of the statement in Fig. 11 is to compute a polynomial of p. The compiler incorrectly optimizes the expression's first item, p0 * p * p, into p0 * p (line 2), which is evidently wrong given that p is not always 1 during runtime. The second item, 3 * p1 * t1 * p * p is also turned into a wrong expression, 3 * p1 * p2 * p; with debugging, we find that the crept-in variable p2 is not at all related to p1, t1, or p.

```
1 /* Incorrectly optimized into
2 q = p0 * p + 3 * p1 * p2 * p; */
3 q = p0 * p * p * p + 3 * p1 * t1 * p * p;
```

Fig. 11: Bug$_5$: complex arithmetic expression.

Bug$_6$*: Ill-Compilation of* if*-Conditions.* When compiling a series of if statements, the shader compilers may pack several comparison operations into a single SIMD instruction. For the example in Fig. 12, the compiler packs four if conditions into the following SIMD assembly instruction:

```
eq r9.xyzw, r2.wwww, l(1.0, 2.0, 3.0, 0.0)
```

The vector r2.wwww stores four duplicate values, all of which equal to variable t in Fig. 12. This vector is compared against four numbers (1.0, 2.0, 3.0, 0.0), and each comparison result is stored in the corresponding element of register r9. It is seen that the compiler incorrectly replaces the fourth if condition at line 6, which should compare t with 4.0, with a comparison with 0.0 in the assembly code (the last element in the 4-tuple). Evidently, the if conditions in the shader source code are improperly compiled, incurring incorrect execution results.

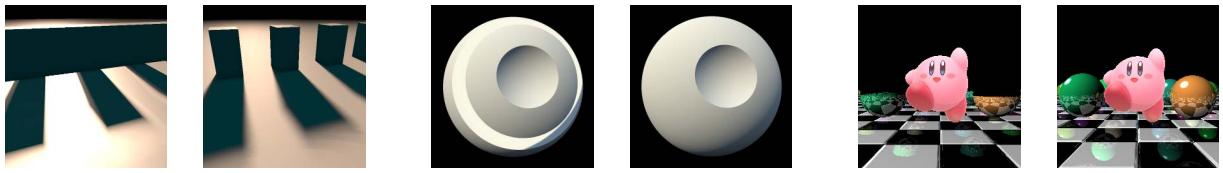
```
1 void f(float t) {
2   color = BLACK;
3   if (t == 1.0) color = RED;
4   else if (t == 2.0) color = GREEN;
5   else if (t == 3.0) color = BLUE;
6   else if (t == 4.0) color = YELLOW; // error
7 }
```

Fig. 12: Bug$_6$: if-condition

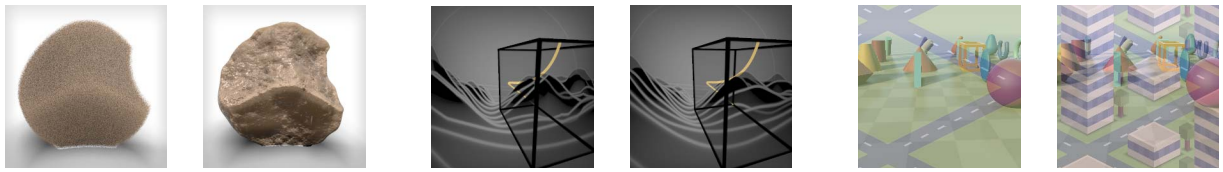
Given that shader compilers do not release source code, it is generally obscure to locate buggy code fragments in shader compilers. Nevertheless, we present samples of erroneously rendered images for each bug category in Fig. 13. Overall, though the six categories result in distinct buggy GPU assembly code, they all lead to noticeable anomalies in the rendered images. Such inconsistencies can lead to confusing visual effects and diminish user experience.

F. Confirmation with Developers

To seek prompt confirmation, we reported to developers one case for each of the six categories of discovered bugs. The chosen cases are hosted at [5]. To ease the burden of debugging and patching, we explore reducing the test cases before reporting them. However, input reduction is challenging: the error-triggering inputs have thousands lines of code and intricate control/data flows (recall we launch program transformations, as noted in Sec. IV-B). We chose two easier test cases for manual reduction, which took about four days. It is hard to reduce the other four test cases due to the intricacy of the code contexts necessary to trigger those four bugs.



(a) An erroneous case belonging to **Bug**$_1$. (b) An erroneous case belonging to **Bug**$_2$. (c) An erroneous case belonging to **Bug**$_3$.



(d) An erroneous case belonging to **Bug**$_4$. (e) An erroneous case belonging to **Bug**$_5$. (f) An erroneous case belonging to **Bug**$_6$.

Fig. 13: Examples for 6 bug categories, each showing the wrong rendering image (left) and its expected output image (right).

We reported the two reduced cases to developers, and received prompt and positive relies.[6] The developers confirmed that they can reproduce our reported bugs, and created tickets for these bugs. However, bugs are not fixed at the time of writing. The shader compilation and execution pipeline is complex, so is its underlying driver code, making it difficult to fix bugs. Also, bug fixing depends on vendor's labor resources. We found that more than 70% of the driver-related bugs (55 in total) reported on the AMD driver forum during the past two years have not yet been fixed, although many of them appear critical (e.g., related to memory overwriting).

VII. Discussion

Limitations and Threats to Validity. *Construct validity* denotes the degree to which our metrics reflect the correctness of shader compilers. We conduct dynamic testing and manual inspection. While this practical approach detects bugs, a possible threat is that our testing cannot guarantee the functional correctness of shader compilers. We clarify that FSHADER roots the same assumption as previous works that also launch dynamic testing instead of static verification [18].

A potential threat exists that FSHADER may not adapt to other shader compilers since the research focuses on three dominant graphics APIs and mutates one shader language, GLSL. We mitigate this threat to *external validity* by designing a language and platform-independent approach. As a result, our approach applies to other settings outside the current scope. We believe the proposed technique is thus general.

Obstacles in Launching Differential Testing (DT). Besides MT, another potential method for testing shader compilers is to apply DT. Though DT has been used to test conventional compilers [12, 26, 33, 40, 45, 48, 57, 60] (by detecting the inconsistency of compiled outputs), we do not deem DT suitable for testing shader compilers based on two reasons. First, graphics shader programs heavily use floating numbers, whose implementation is sensitive to both underlying software and hardware. In fact, different devices (e.g., mobile phones vs. servers) have distinct floating number implementations, likely resulting in noticeably different rendering outputs. Second, the same graphics API specification (e.g., OpenGL) may have deviated vendor implementations from device to device. For instance, NVIDIA and Intel GPUs give deviated results for OpenGL's two inherent functions, `dFdx` and `dFdy` [23].

Launching DT may lead to considerable false positives, mainly due to the two aforementioned reasons. In contrast, our MT approach only checks the rendering consistency of each specific device, greatly alleviating the above issues. We may still encounter the floating number deviation problem, since a fused shader may be optimized differently and thus induce slightly different rendering outputs. Nevertheless, we clarify that such differences are typically negligible since they are visually indistinguishable from human beings. Moreover, by using a well-picked threshold, we are able to rule out deviant outputs brought by potentially different optimizations.

[6]The other four unreduced cases are also reported, yet we received no reply.

VIII. Related Work

MT is first proposed [14] to alleviate the absence of testing oracles. MT has been extensively used in compiler testing [13, 37, 43, 44, 51], and has led to numerous findings on testing popular mainstream compilers like GCC, LLVM, JVM, and OpenCL [29, 30, 33, 46]. The most relevant one to this research, GLFUZZ [18], launches MT to test shader compilers by inserting dead code into shader programs. MT is also used in testing deep learning compilers [56]. Similar to previous works using MT for compiler testing, FSHADER also mutates shader programs, the input of shader compilers and graphics rendering pipelines. Nevertheless, FSHADER novelly forms the MRs based on several principled *visual effects*.

Another prominent line of work on compiler testing employs differential testing (DT), which checks output consistency across multiple compilers [12, 16, 28, 48, 49, 52]; yet as aforementioned, DT may not be suitable for shader compilers testing.

IX. Conclusion

We presented a novel MT framework for shader compilers on the basis of visual consistency. Our comprehensive study uncovers six potential bugs in 12 configurations of devices and graphics APIs. This work can serve as a roadmap for users and developers to use and improve shader compilers.

Acknowledgement

We thank anonymous reviewers for their valuable feedback. We also acknowledge Huaijin Wang for his kind assistance with hardware purchase and assembly. This project is supported in part by a RGC ECS grant under the contract 26206520.

REFERENCES

[1] Fshader artifact. https://sites.google.com/view/fshader/.

[2] History of opengl. https://www.khronos.org/opengl/wiki/History_of_OpenGL, 2022.

[3] Khronos group. https://www.khronos.org/, 2022.

[4] Microsoft wiki - directx. https://microsoft.fandom.com/wiki/DirectX, 2022.

[5] Online demo of 6 bugs discovered by fshader. https://fshader2023.github.io/, 2022.

[6] Renderdoc. https://renderdoc.org/, 2022.

[7] Vulkan. https://www.vulkan.org/, 2022.

[8] Vulkan - drawing a triangle. https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Base_code, 2022.

[9] Vulkan for safety-critical industry. https://www.khronos.org/api/vulkansc, 2022.

[10] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. *Real-Time Rendering 4th Edition*. A K Peters/CRC Press, Boca Raton, FL, USA, 2018.

[11] archee. Gpu and os detector v2. https://www.shadertoy.com/view/7ssyzr, 2022.

[12] Gergö Barany. Finding missed compiler optimizations by differential testing. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 82–92, 2018.

[13] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.

[14] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong . . . , 1998.

[15] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, 51(1), jan 2018.

[16] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of jvm implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–99, 2016.

[17] Zhongliang Chen and David Kaeli. Balancing scalar and vector execution on gpu architectures. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 973–982, 2016.

[18] Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[19] Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Shader compiler bugs - graphicsfuzz. https://github.com/mc-imperial/shader-compiler-bugs, 2018.

[20] Alastair F. Donaldson, Hugues Evrard, and Paul Thomson. Putting Randomized Compiler Testing into Production (Experience Report). In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:29, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[21] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, R. P. Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *ISSTA*, 2018.

[22] Google. ANGLE: Almost native graphics layer engine. https://chromium.googlesource.com/angle/angle, 2022.

[23] Khronos Group. dfdx. https://registry.khronos.org/OpenGL-Refpages/gl4/html/dFdx.xhtml, 2022.

[24] Khronos group. GLslang. https://www.khronos.org/opengles/sdk/tools/Reference-Compiler/, 2022.

[25] JackAsser. Raymarching example. https://www.shadertoy.com/view/MsX3WN, 2013.

[26] He Jiang, Zhide Zhou, Zhilei Ren, Jingxuan Zhang, and Xiaochen Li. Ctos: Compiler testing for optimization sequences of llvm. *IEEE Transactions on Software Engineering*, 48(7):2339–2358, 2022.

[27] jlfwong. Ray marching: Part 3. https://www.shadertoy.com/view/Xtd3z7, 2016.

[28] Kota Kitaura and Nagisa Ishiura. Random testing of compilers' performance based on mixed static and dynamic code comparison. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating Test Case Design, Selection, and Evaluation*, pages 38–44, 2018.

[29] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 49(6):216–226, 2014.

[30] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices*, 50(10):386–399, 2015.

[31] Eric Lengyel. *Game engine gems 2*. CRC Press, 2011.

[32] Colin Leung. Glsl ray tracing test. https://www.shadertoy.com/view/3sc3z4, 2019.

[33] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. Many-core compiler fuzzing. *ACM SIGPLAN Notices*, 50(6):65–76, 2015.

[34] Frank Luna. *Introduction to 3D game programming with DirectX 11*. Mercury Learning and Information, 2012.

[35] Lei Ma, Felix Juefei-Xu, Jiyuan Sun, Chunyang Chen, Ting Su, Fuyuan Zhang, Minhui Xue, Bo Li, Li Li, Yang Liu, et al. DeepGauge: Comprehensive and multi-granularity testing criteria for gauging the robustness of deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018.

[36] Microsoft. frac. https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-frac, 2022.

[37] Kazuhiro Nakamura and Nagisa Ishiura. Random testing of c compilers based on test program generation by equivalence transformation. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pages 676–679. IEEE, 2016.

[38] nimitz. Protean clouds. https://www.shadertoy.com/view/3l23Rh, 2019.

[39] NVIDIA. Cutlass 2.11. https://github.com/NVIDIA/cutlass, 2022.

[40] Georg Ofenbeck, Tiark Rompf, and Markus Püschel. Randir: differential testing for embedded compilers. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, pages 21–30, 2016.

[41] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 1–18, 2017.

[42] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (gpu gems)*. Addison-Wesley Professional, 2005.

[43] Hanan Samet. Compiler testing via symbolic interpretation. In *Proceedings of the 1976 annual conference*, pages 492–497, 1976.

[44] Hanan Samet. A machine description facility for compiler testing. *IEEE Transactions on Software Engineering*, (5):343–351, 1977.

[45] Masataka Sassa and Daijiro Sudosa. Experience in testing compiler optimizers using comparison checking. In *Software Engineering Research and Practice*, pages 837–843. Citeseer, 2006.

[46] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on software engineering*, 42(9):805–824, 2016.

[47] Sergio Segura, Dave Towey, Zhi Quan Zhou, and Tsong Yueh Chen. Metamorphic testing: Testing the untestable. *IEEE Software*, 37(3):46–53, 2018.

[48] Flash Sheridan. Practical testing of a c99 compiler using output comparison. *Software: Practice and Experience*, 37(14):1475–1488, 2007.

[49] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 849–863, 2016.

[50] Liqun Sun and Zhi Quan Zhou. Metamorphic testing for machine translations: Mt4mt. In *2018 25th Australasian Software Engineering Conference (ASWEC)*, pages 96–100. IEEE, 2018.

[51] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *2010 Asia Pacific Software Engineering Conference*, pages 270–279. IEEE, 2010.

[52] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 697–709, 2022.

[53] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. ICSE '18, 2018.

[54] Jingyi Wang, Guoliang Dong, Jun Sun, Xinyu Wang, and Peixin Zhang. Adversarial sample detection for deep neural network through model mutation testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1245–1256, 2019.

[55] Shuai Wang and Zhendong Su. Metamorphic object insertion for testing object detection systems. In *ASE*, 2020.

[56] Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. Metamorphic testing of deep learning compilers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1):1–28, 2022.

[57] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[58] Zavie. Ray tracing a cone. https://www.shadertoy.com/view/MtcXWr, 2016.

[59] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. DeepRoad: GAN-based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *ASE*, 2018.

[60] Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo, and Zhaohui Wang. Automated test program generation for an industrial optimizing compiler. In *2009 ICSE Workshop on Automation of Software Test*, pages 36–43. IEEE, 2009.