Flutter FragmentProgram API Support Improvements

SUMMARY

Improvements to FragmentProgram API Support.

Author: Zach Anderson (zanderso)
Go Link: flutter.dev/go/fragment-program-support
Created: 5/2022 / **Last updated:** 6/2022

WHAT PROBLEM IS THIS SOLVING?

The FragmentProgram API:

1. Isn't well documented, and requires third party tools to use at all.
2. In its current form, it won't work with Impeller.

BACKGROUND

The Flutter Engine's FragmentProgram API was added in Flutter 3.0 primarily to support the ink sparkle effect that was part of the new version of Material. The API is documented as being "in beta" with several issues still open. It accepts a subset of SPIR-V bytecode. In particular, as it is implemented today, it accepts only the subset of SPIR-V bytecode that the Engine can transpile to SkSL. The SkSL code is then passed to Skia's SkRuntimeEffect API, which compiles the SkSL at runtime into a platform specific shader. The discussion that led to this design is documented here.

Since the 3.0 release, Flutter users have started doing really cool things with the FragmentProgram API [1, 2, 3, 4]. This has spawned tutorials [1, 2], and a couple of efforts to make it easier for developers to use [1, 2]. The sudden interest in the feature has taken the Engine team a bit by surprise. It was incorrectly assumed that the API was so difficult to use, and so poorly documented, that it would not inspire much use by the community.

Concurrent with the work to add the FragmentProgram API, the team has also been

working on an experimental rendering backend called Impeller. Impeller precompiles a smaller, simpler set of shaders at Engine build time so that they won't compile while an app is running, which has been a major source of jank in Flutter. The need to have all shaders compiled ahead of time conflicts with the capabilities exposed by the FragmentProgram API. The design outlined below explains how we plan to resolve this conflict.

Glossary

- **SPIR-V** - An industry standard intermediate representation of shader programs.
- **GLSL** - The OpenGL shading language.
- **SkSL** - Skia's shading language.

OVERVIEW

The proposed solution is for the Flutter Engine to vend a shader compiler that can take GLSL (and possibly SPIR-V) as input, and generate a shader in the correct format depending on the target platform and whether Impeller is enabled. This shader compiler can then be integrated into the Flutter build in the Flutter CLI, and shaders treated just as any other asset.

Non-goals

This design will not address the issue that it will likely still be useful for Dart code to be generated for custom shaders to fill in uniforms, and present an idiomatic API to Flutter application code. Tools developed by the community like Umbra seem like a nice way to do that.

DETAILED DESIGN/DISCUSSION

Regarding the two problems mentioned above, the same solution will both make the FragmentProgram API easier to use, and make it ready to be supported in Impeller. In particular, to solve the first problem, the Engine build will vend a shader compiler (`impellerc`) that accepts GLSL (and possibly also SPIR-V). When a Flutter project includes a file ending in `.frag` in GLSL format in the asset list in its `pubspec.yaml` file, the Flutter build will use `impellerc` to generate a shader in the correct format, and include it as an Asset in the application's asset bundle.

Today, the correct format is the SPIR-V bytecode that is consumed as described above. In the future, for example with Impeller, the correct format may be dependent on the target platform and rendering backend used by the Engine. This approach solves the second problem mentioned above.

Unfortunately, this design involves a breaking change of the FragmentProgram API when Impeller is enabled. Instead of always accepting SPIR-V bytecode, when Impeller is enabled, the API will accept a format that depends on the target

platform, and which must be generated by `impellerc`.

Generally when we break something, we should be able to elucidate the benefits. In this case, the benefits are the following:

1. Less reliance on third party tools—Flutter will include everything needed to translate a GLSL shader into the format accepted by Engine APIs.
2. When Impeller is enabled, this approach will allow using custom shaders with much less jank for at least two reasons.
    a. First, there will be no need to transpile the shader to SkSL.
    b. Second, the shader won't need to be compiled from SkSL to the correct platform specific format at runtime since it will already be bundled with the application in the correct format.
3. Changes to allow Impeller to support the FragmentProgram API are also attractive because when implemented with Impeller, the feature set used by custom shaders will no longer be limited by the capabilities of the SPIR-V to SkSL transpiler. Instead, the feature set will expand to be defined by some standard version of GLSL that `impellerc` supports.
4. Users who migrate to this flow will be able to hot reload their shaders written in GLSL, even when Impeller is not enabled.

Work on the above approach is already underway on the master channel. The Engine's shader compiler, `impellerc`, is used by Impeller for its offline shader compilation. We have landed a series of changes to build it for each host platform, include it in the Engine's host artifacts, and to download and use it from the Flutter CLI. On the master channel, it is already used to compile the Material ink sparkle shader.

The remaining work is to implement the FragmentProgram API in Impeller. This will require prepending the offline-compiled platform specific shaders with metadata explaining to Impeller how e.g. uniform buffers should be filled in. However, this is expected to be straightforward, and optimistically will be finished by the end of Q2 or at the beginning of Q3. As a side note, the need to prepend platform-specific metadata implies that the FragmentProgram API will accept only shader data that has been generated by `impellerc` when Impeller is enabled.

Additionally, when the Flutter build is driving compilation of shaders, we will be able to hot reload them. This is possible because the Flutter CLI's build system can note that asset files have been modified, requiring the new assets to be sent to the running application, and a reassemble triggered, just as with the update of any other kind of asset.

TESTING PLAN

We are currently testing the above design with the tests of the Material ink sparkle shader. Going forward we will expand the scope of testing to include the existing

Engine repo unit tests of the FragmentProgram API. Currently, these are not yet using `impellerc`, but rather using an off-the-shelf one from one of the Engine's third party dependencies. This will need to change concurrent with Impeller implementing the FragmentProgram API because with Impeller enabled, the API will not accept SPIR-V.

DOCUMENTATION PLAN

At minimum, the API documentation on FragmentProgram will be expanded to include more information about the input format as it evolves, and to include information about how to bundle custom shaders with an app by including GLSL shaders in an application's asset list.

However, overall, the documentation of the FragmentProgram API, and the subset of GLSL that it supports, needs a lot of expansion, and should be featured more prominently on the website. As the maturity of the API increases, we should also consider e.g. video tutorials on YouTube, blog posts, etc.

MIGRATION PLAN

In the near term, the FragmentProgram API will not be broken. It will continue to accept SPIR-V. In the future, if we complete a migration to Impeller, the FragmentProgram API will stop accepting SPIR-V. At that point, all users will need to migrate to using `impellerc` to generate the right platform specific shaders by way of the usual Flutter build workflow as described above. Users can already begin work on the migration, and should soon see benefits to doing so since custom shaders are planned to be hot-reloadable in future releases.

Additionally, other smaller breaking changes to the FragmentProgram API are implied by this overall plan. For example, `FragmentProgram.compile` currently accepts a `spirv` named argument, which will no longer make sense. Furthermore, we may wish to employ techniques from image loading to avoid shader data entering the Dart heap, which would also require API changes. The specifics of these smaller changes will be spelled out, and migration paths communicated, as the changes are landed.