Shaders For Scientific Visualisation

Thomas Mathieson

2576219m

Proposal

Motivation

Current data visualisation libraries can be overly prescriptive in how data can be visualised and interacted with. Programmable shaders on the other hand can allow for extremely performant rendering of large amounts of data and unmatchable flexibility in how scientific data is displayed and can be interacted with. We aim to expose the power and flexibility of programmable shaders for data visualisation in python.

Aims

The project aims to create a python library to allow users to write custom shaders for arbitrary data visualisation. Since a lot of data science is done in Jupyter Notebooks, we aim to integrate our library into Jupyter as a custom Jupyter widget providing a canvas onto which shaders can render content. The library should help users reduce boilerplate shader code by providing shader templates and a standard library of commonly used functions for scientific visualisation. Additionally, the widget should allow for user interactivity and tools to export rendered visualisations should be provided.

Progress

- Rendering Engine
  - OpenGL based rendering engine based on moderngl
  - Multiple-process rendering with custom RPC communication layer to allow multiple independent render widgets to exist at once
  - RenderDoc API integration
- Jupyter Widget
  - PNG/JPG streaming from rendering backend to Jupyter
  - Public PyPi and NPM package
  - Jupyter Lab/Google Colab/VSCode support
  - Mouse interaction
  - Widget status bar indicating connection status, play/stop buttons, RenderDoc capture button, etc...
- Shader Preprocessor
  - Shader template preprocessor based on pcpp
  - Support for user defined shader templates
  - Shader templates can be parametrised with template arguments which are injected into the processed shader by the preprocessor.

- Data Input
  - Shader uniforms can be set at runtime
  - Custom vertex buffers can be set from NumPy arrays
  - NumPy arrays can be converted to textures (which can also be updated at runtime)
    - Automatic texture format deduction based on NumPy array format
- Shader Library
  - Shader preprocessor templates for a variety of use cases from basic full screen pixel shaders to a signed distance field renderer.
  - Built in shader uniforms for time, mouse position, etc...

Problems and risks

Problems

Good integration with Jupyter is one of the key goals of the project; and as such we've had to work around some of its complexities. Since the Jupyter kernel is global to the notebook, we can't just start a render loop in the main thread when we want to render continuously as it would block the rest of the notebook. Additionally, to support multiple widgets rendering at once we need to be able to create multiple OpenGL renderers at once. To solve this, we had to split the renderer into its own process and develop an RPC system to allow canvases to communicate with render processes.

For the project to be compatible with Google Colab, the python library needs to be published on PyPi and the frontend component (written in TS) needs to be published on npm. These platforms have different rules for naming packages; when the project was started the name "pySSV" was chosen, which isn't compatible with npm; as such the project had to be refactored such that all references to the frontend package were renamed to "py-ssv" this required significant changes to the package build configuration which assumes that the python and npm package have the same name. The process of building plugins for Jupyter is generally poorly documented (as a result of differences between Jupyter Lab and Jupyter notebooks and many recent major changes to how plugins are handled) which caused delays due to build system issues.

Risks

In the project's current state a lot of the major systems have already been implemented and the project architecture is generally flexible enough to incorporate large changes if problems are encountered without too much effort.

It's possible that as we do more user testing we discover platform specific issues due to inconsistencies in OpenGL implementations. To mitigate this, we can take advantage of the shader preprocessor and templating system to integrate workarounds for specific platforms conditionally in shader templates such that end users need not know about them.

Plan

- Week 1: Improve the library of shader templates and basic shader functions. Write more example shaders.
- Week 2: Improve video streaming; implement MJPEG or H264 based streaming from the renderer to the widget.
- Week 3-5: Start user testing. Prepare some guided user exercises, feedback forms, and find testers.
- Week 6: Implement user feedback.
- Week 7: Add functionality related to image/video export; and if time allows standalone render canvas windows.
- Week 8: Finish any work that couldn't be completed from previous weeks. Work on the dissertation.
- Week 9: Additional user testing. Work on the dissertation.
- Week 10: Work on the dissertation.