# Abstract

# Introduction

# Background

*What did other people do, and how is it relevant to what you want to do?*

# Analysis/Requirements

*What is the problem that you want to solve, and how did you arrive at it?*

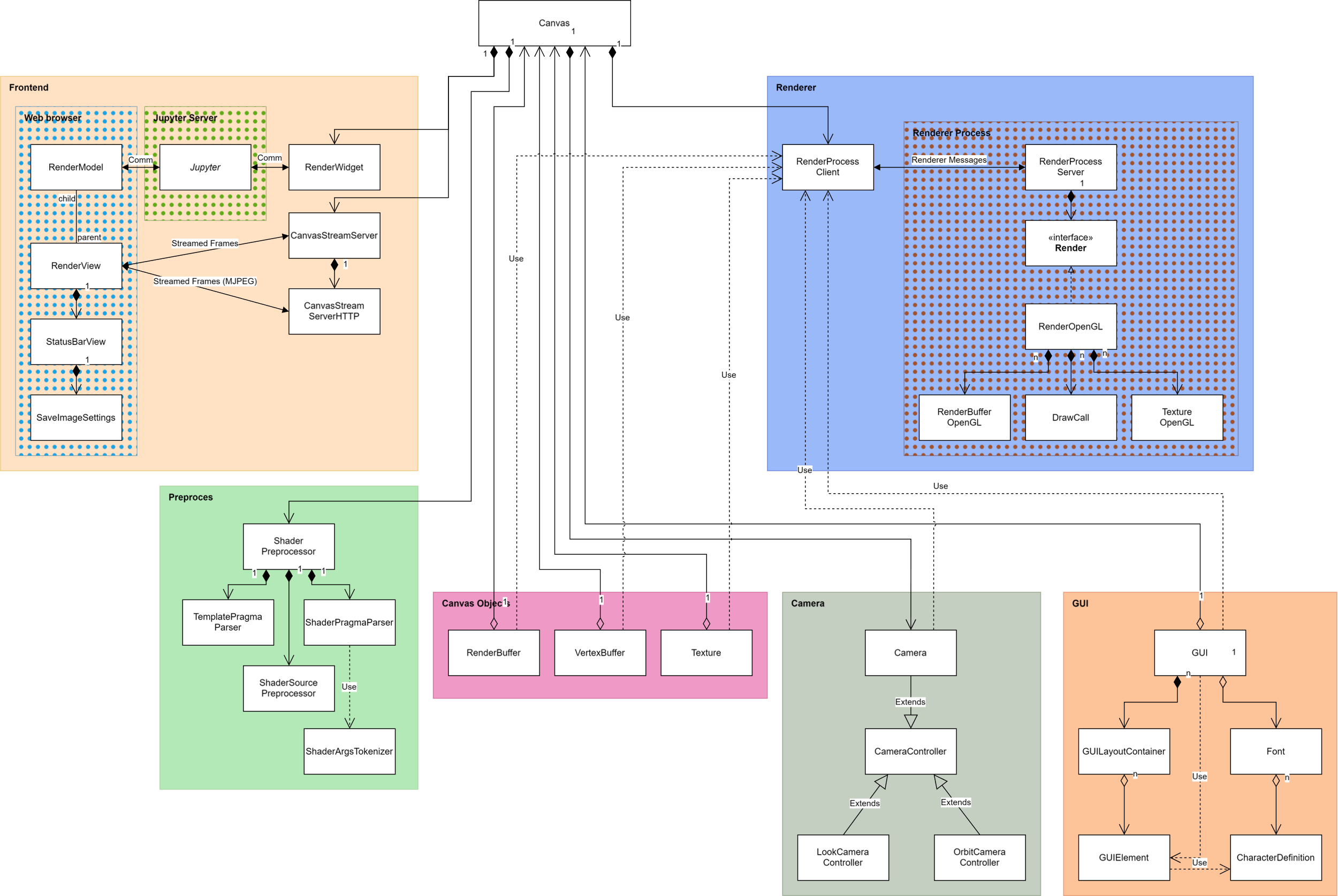
The overall goal is to allow users to create interactive visualisations using shaders in Python.

Many Python developers use Jupyter [citation needed] Notebooks for data science tasks and interactive programming in general. Jupyter allows the creation of “Widgets” which are interactive controls/visualisations for code cells.

As the focus is on allowing the creation of shaders

# Design

*How is this problem to be approached, without reference to specific implementation details?*



# Implementation

*What did you do to implement this idea, and what technical achievements did you make?*

## Renderer

Heartbeat

Renderdoc

To be able to support multiple independent canvases running in a single Python kernel we need to be able to create multiple independent renderers. In OpenGL, only one context can be active in a given thread; in addition to this Python’s global interpreter lock prevents threads from running concurrently. As such we decided to create a separate process for each renderer. These render processes communicate with the Python kernel using a simple inter-process communication API built around Python’s multiprocessing.Queue.

When a new canvas is created it creates a new SSVRenderProcessClient which serves as the canvas’ interface to the render process. This client starts the render process which creates a new SSVRenderProcessServer. The client and server share a receive and a transmit queue for messaging. Renderer messages are Python tuples where the first item is a four-character command identifier, and the remaining items are the parameters for the command. The format of a few of the renderer commands is described below:

render = ("Rndr", target\_framerate: float, stream\_mode: str, encode\_quality: float | None)

stop = ("Stop", )

update\_uniform = ("UpdU", frame\_buffer\_uid: int | None, draw\_call\_uid: int | None, uniform\_name: str, value: Any)

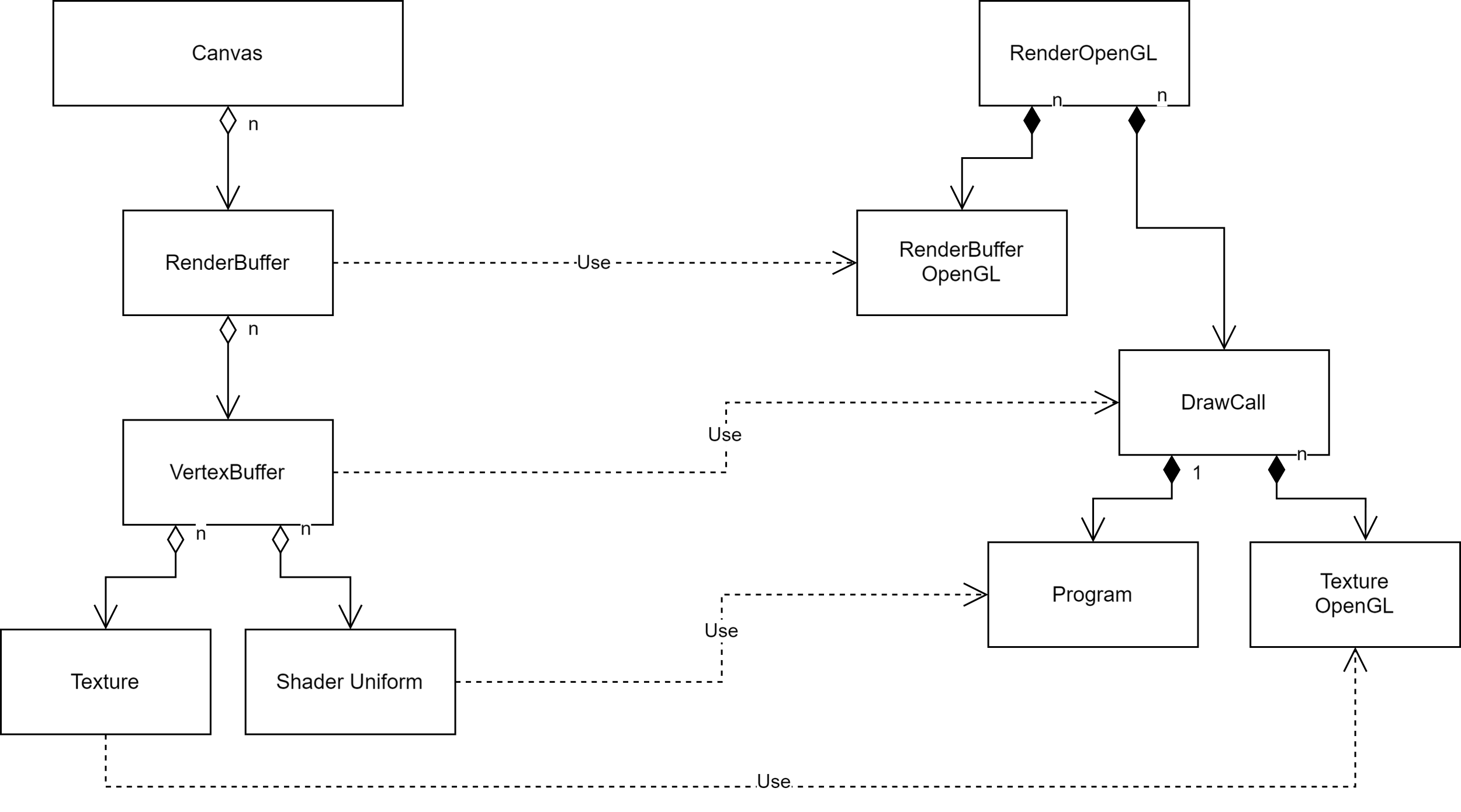
update\_vertex\_buffer = ("UpdV", frame\_buffer\_uid: int, draw\_call\_uid: int, vertex\_array: NDArray | None, index\_array: NDArray | None, vertex\_attributes: tuple[str, ...] | None)

get\_context\_info = ("GtCt", query\_id: int, full: bool)

Some commands expect a value to be returned upon completion, we refer to these as queries. Queries have an extra item after the command identifier but before command parameters, this is the query ID. When a query is made, the client creates a future which it stores in a dictionary. When the query is completed the server sends back an ARes command to the client containing the query ID and the result of the query, when received, the client finds the corresponding future in it’s dictionary and resolves it using the acquired query result.

The render process server creates an instance of the render which creates the OpenGL context and executes the render commands. The OpenGL renderer implements the SSVRender interface, with the idea being that if other graphics APIs needed to be supported, one could do so by simply implementing the SSVRender interface and registering it as a new rendering backend. The render commands are relatively high level, so the OpenGL renderer is responsible for storing renderer objects such as vertex buffers, textures, and render buffers, so that these can be bound to the OpenGL pipeline when needed. The renderer also needs to perform bounds checking as needed since if it raises an uncaught exception, the render process simply crashes without informing the user; OpenGL’s error messages can also be somewhat cryptic, especially since an error might not be raised until the next frame is rendered, as such proactive bounds checking on command parameters can be really helpful to end users.

Since the renderer interface is command based, but we want to provide the user with an object-oriented way to interface with the renderer, we end up mirroring the object structure used by the renderer on the canvas. These canvas objects store a unique ID (UID) representing the corresponding renderer object and simply dispatch a render command using the render process client and it’s UID when one of its methods are called or properties are updated. This allows the user to use an object-oriented interface without having to pollute the user API with renderer specific code and keeps the renderer communication simple since we don’t need to send full objects back and forth.



## Shader Preprocessor

# DESIGN

To reduce the amount of boilerplate shader code users need to write, we decided to implement a shader templating and preprocessor system this would allow the user to pick a shader template for the type of shader they want to write and simply write the main function and allow the preprocessor to generate the necessary boilerplate. Platform specific shader code such as version and extension GLSL directives should be autogenerated. Additionally, built global uniforms and user defined uniforms can be automatically declared by the preprocessor. Shader templates themselves range in complexity from barebones templates such as the pixel shader template which defines a simple vertex shader and includes the necessary preprocessor directives to autogenerate the needed boilerplate; to the more complex signed distance function template which includes a raymarching renderer and exposes an entry point to a user defined signed distance function. Templates can also be parameterized using template arguments.

####

With GLSL being a C-derived language, it is compatible with the C preprocessor. The advantages of the C preprocessor: are that users are likely to be familiar with it, it can be extended by using custom pragma directives, and a Python library for it already exists. We used the PCPP [] preprocessor

## Render Widget

## GUI Library

## Camera

# Evaluation

*How good is your solution? How well did you solve the general problem, and what evidence do you have to support that?*

# Conclusion

*Summarise the whole project for a lazy reader who didn't read the rest (e.g. a prize-awarding committee).*