Design Document: Visualization Build Option Refactor for Kimera-VIO

Author: David Abraham
Status: In Review
Date last updated: Nov 13, 2024
Approved by:

Background & Overview

Kimera-VIO is an open-source visual-inertial odometry (VIO) library that we are adapting for deployment on a low-powered nano drone. To optimize Kimera-VIO for our application, we have modified a fork of the library to exclude its dependency on the OpenCV viz module which is solely used for visualization. These changes provide a version of Kimera-VIO for production environments and highly constrained devices i.e. it will benefit devices with strict storage limitations and simplifies the compilation process for cases where visualization tools will not be used (e.g. on a drone).

To add this build option to the library's official repository, the fork's modifications need to be productionalized. This document outlines a refactor of the main branch to achieve this. It should allow users to configure the library for scenarios where visualization is inapplicable, while maintaining Kimera's core functionality.

Goals

- **Implement Configurable Build Option**: Introduce a CMake option to enable or disable visualization at build time, and not just run-time.
- **Contribute to the mainline KimeraVIO repo**: Integrate the forked modifications into the official Kimera-VIO repository, ensuring support for this use case across future updates.
- **Add Cross-Compile Support for ARM**: Implement cross-compilation options in the CMake build system to accommodate embedded applications, such as on aarch64 development boards.

Guiding Principles

- **Maintain Core Functionality**: Ensure that removing the viz module does not affect the core visual-inertial odometry (VIO) and SLAM functionality of Kimera-VIO. The build workflow should remain consistent for existing users. There should be minimal to no software changes that break existing user functionality for pre-existing applications.
- **Minimal changes to the Kimera-VIO code base:** This will help changes to be more easily understood by maintainers and reduce the likelihood of introducing breaking changes. It

should minimize the likelihood of introducing changes that will affect future-facing work that Virtana has low visibility into.

- **Focus on Usability**: There should be clear documentation and feedback to users so that there is no ambiguity or confusion about the build settings and operating modes related to this feature.
- **Production Readiness**: Ensure that the changes are suitable to production environments i.e. adhering to best practices, appropriate code health etc.
- **Small, Digestible PRs**: Implement feature changes through small, standalone pull requests (PRs), making the code easier to review and integrate.

Proposed Solution

CMake Related Changes

CMake Flag

We will introduce a CMake option to conditionally include or exclude all visualizer source code (***src/visualizer***, ***include/kimera-vio/visualizer)*** as well as specific tests (eg. ***testVisualizer3D.cpp***). By default, this option will remain enabled. See below example for flag use (similar to KIMERA_BUILD_TESTS).

```
option(KIMERA_BUILD_VISUALIZER "Enable visualization" ON)
...
if(KIMERA_BUILD_VISUALIZER)
    add_definitions(-DKIMERA_BUILD_VISUALIZER)  # For macro access in code
    # Add source code / include tests for visualization
endif(KIMERA_BUILD_VISUALIZER)
```

Cross Compilation Support

Adding a *toolchain.cmake* file will allow users to configure the build for aarch64, with visualization disabled by default. This configuration will be invoked using the CMake argument, **-DCMAKE_TOOLCHAIN_FILE.** The intention is for users to add toolchain files for other compilers or architectures as needed.

```
# Example Toolchain file (e.g. aarch64-gnu-toolchain.cmake)
set(CMAKE_SYSTEM_PROCESSOR aarch64)
set(CMAKE_C_COMPILER aarch64-linux-gnu-gcc)
set(CMAKE_CXX_COMPILER aarch64-linux-gnu-g++)
...
# The default value will be overridden by user input if provided
set(KIMERA_BUILD_VISUALIZER OFF CACHE BOOL "Disable visualizer when cross-compiling")
```

Testing Considerations

Cross compiled Kimera-VIO tests cannot run successfully on the build machine of a different architecture. We propose that tests be built but not run when cross compiling. The below provides the advantages and disadvantages of such an approach:

- **+** Ensures tests are still compiled for the target platform, if desired, providing a quick way of validating the cross-compiled library has no issues on the target device.
- **+** Minimal changes to the CMake configuration.
- **+** Build time is not affected.
- **-** Additional user steps to transfer and run tests on the target device.

If a user who is cross-compiling wants testing to be run on the host machine, they can build Kimera-VIO for the host machine with testing enabled and then build for their target architecture; one extra command.

The below snippet shows the required CMakeLists.txt change that facilitates this:

```
// Kimera-VIO's Root CMakeLists.txt

if (NOT CMAKE_CROSSCOMPILING) // Only run tests if compiling natively
   include(GoogleTest)
   gtest_discover_tests(testBinary PRE_TEST)
endif()
```

Alternatives Considered
Building and running tests natively was also considered. This was ruled out after considering the following advantages and disadvantages.

- + Provides immediate feedback with quicker debugging on the build machine.
- - Leads to longer build times when cross-compiling due to compiling for two systems.
- - Host may not replicate target platform behavior for architecture-specific features.
- - Adds additional complexity to the CMake configuration.
- - The user *still* needs to transfer and run tests on the target device to be 100% certain that it is compatible with their target machine.

Further details about these alternatives are discussed in the Appendix.

Core File Changes

Excluding the visualization source code requires modifications to all visualizer references across Kimera's files. The code to be changed can be generalized below:

- Visualizer specific headers
- Kimera-VIO's visualizer types as class' constructor arguments and member variables
- OpenCV viz class references outside of the Kimera-VIO’s visualizer.

Visualization Headers

All viz-related headers will be excluded using the #ifdef preprocessor directive with ***KIMERA_BUILD_VISUALIZER***.

```
#ifdef KIMERA_BUILD_VISUALIZER
    #include "kimera-vio/visualizer/ExampleVisualizerHeader.h"
#endif
```

Handling Visualizer Objects as Class Members

All visualization references in Kimera's classes must be adapted for build configurations including and excluding visualization. The abstract class approach below will be implemented for each visualizer class utilized.

Abstraction of Visualizer Classes

Consider MeshOptimization as an example, which accepts an ***OpenCvVisualizer3D::Ptr*** object.

```
MeshOptimization(..., OpenCvVisualizer3D::Ptr visualizer = nullptr);
```

We can create an abstract class (eg. ***BaseOpenCvVisualizer3D***) with two derived classes; the real visualizer object (OpenCvVisualizer3D) and a dummy with stub implementations of any functions being referenced.

```
// BaseOpenCvVisualizer3D.h -----

class BaseOpenCvVisualizer3D { // abstract class
 public:
   BaseOpenCvVisualizer3D() = default;
   ...
   virtual void visualizePointCloud(...) = 0; //pure virtual function
};

class DummyOpenCvVisualizer3D : public BaseOpenCvVisualizer3D { // derived class
 public:
   DummyOpenCvVisualizer3D() = default;
   ...
   void visualizePointCloud(...) override {} //do nothing
};
```

For our example, ***OpenCvVisualizer3D*** will become a derived class of our base.

```
// OpenCvVisualizer3D.h/cpp -----

class OpenCvVisualizer3D : ..., public BaseOpenCvVisualizer3D { //make derived class
  public:
  ...
   void visualizePointCloud(...) override { // actual implementation }
};
```

MeshOptimization should be modified to accept the base, as shown in the example below. This will facilitate dynamic dispatch of visualizer member functions while accepting either derived class type.

```
// MeshOptimization.h -----

class MeshOptimization {
  public:
   MeshOptimization(..., BaseOpenCvVisualizer3D::Ptr visualizer = nullptr);
   ...
   BaseOpenCvVisualizer3D::Ptr visualizer_;
};

// MeshOptimization.cpp -----

MeshOptimization::MeshOptimization(..., BaseOpenCvVisualizer3D::Ptr visualizer):
...visualizer_(visualizer)

visualizer_->visualizePointCloud(...); // dynamically runs OpenCvVisualizer3D/Dummy implementations
```

This design allows class applications to instantiate the respective derived class once without having to handle this condition thereafter. Consider the following example.

```
// Example usage -----

#ifdef KIMERA_BUILD_VISUALIZER
   BaseOpenCvVisualizer3D::Ptr visualizer = std::make_shared<OpenCvVisualizer3D>(...);
#else
   BaseOpenCvVisualizer3D::Ptr visualizer = std::make_shared<DummyOpenCvVisualizer3D>();
#endif

MeshOptimization mesh_opt(..., visualizer);
```

The only caveat to this approach includes the need to update and maintain the base and dummy classes for each of the abstracted objects.

Each abstract class will be written as a separate file, and all of these will be included in a new directory `Kimera-VIO/include/kimera-vio/visualizer/abstract`. The dummy classes will be written as separate files, also included in a new directory `Kimera-VIO/include/kimera-vio/visualizer/dummy_classes.`

The above approach should be translated to similar constructors such as those in the Pipeline classes, for example.

Alternatives Considered

Alternative approaches for classes that accept visualizer objects as constructor arguments were as follows:

- Make the class abstract, with two derivations that include and exclude visualizer components, respectively.
- Overload the class constructor and use the ***#ifdef*** directive to selectively exclude visualizer logic.
- Apply templates to manage visualizer object types

These options required existing and future users of these classes to implement and manage the selection logic for both build scenarios in their applications. Utilizing an abstract class for the visualizer objects keeps the classes' usage unchanged, minimizing the use of the ***#ifdef*** directive within Kimera's classes while allowing user implementations to select between the real and dummy derivations depending on their needs.

Widget-related typedefs

The `WidgetsMap` type is specified as an input argument to a number of visualizer functions that are used by other classes (e.g. `MeshOptimization`). `WidgetsMap` is defined in **`Visualizer3D-definitions.h`** as follows:

```
typedef std::unique_ptr<cv::viz::Widget3D> WidgetPtr;
typedef std::map<std::string, WidgetPtr> WidgetsMap;
```

Because this is dependent on **`cv::viz`**, it cannot be used as is when building without the visualizer. We propose to handle this as follows:

- `Visualizer3D-definitions.h` will be included ***both*** when building with and without the visualizer (CMakeLists.txt change). This is the only existing visualizer file that will be included when excluding visualization from the build.
  - The typedef definitions are the only place with cv::viz is directly used in this file.
- Use the #ifdef directive to include `opencv2/viz/widgets.hpp` when building with the visualizer and exclude it otherwise.
- Use the #ifdef directive to remove `WidgetPtr` and redefine `WidgetsMap` as std::string when building without the visualizer.
  - `WidgetPtr` is only used in `OpenCvDisplay.cpp` and in the `WidgetsMap` typedef definition right now. It will not be used when building without the visualizer.
  - `WidgetsMap` type objects are only defined in `OpenCvDisplay` and `OpenCvVisualizer3D`. Both of these classes will be replaced by placeholder classes when building without the visualizer.
    - `OpenCvDisplay` only uses WidgetsMap internally, so updating the type will have no effect when building without the visualizer.
  - Several `OpenCvVisualizer3D` functions (e.g. `visualizeGlobalFrameofReference`, `visualizePointCloud` etc.) are used by `EurocPlayground` and `MeshOptimization`.
    - For `MeshOptimization`, the `WidgetsMap` object is populated within a visualizer class, so no changes will be required to this class to handle variable initialization. This is also the case with `EurocPlayground`.

OpenCV Viz References

References to the OpenCV Viz class in several KimeraVIO classes will be addressed accordingly.

Moving cv::viz-heavy code into visualizer module

The MeshOptimization class is the only class that heavily uses cv::viz outside of the visualizer classes. These include:

1. The window_ variable

2. The draw3dMesh function
3. The spinDisplay function

All of these are related to each other and are only used within the MeshOptimization class. Because they are all used for a visualization feature, we propose moving this functionality into the existing OpenCvVisualizer3D class, as well as the corresponding abstract and placeholder classes. `window_` and `spinDisplay` should be renamed to `meshWindow_` and `meshSpinDisplay`. so it remains obvious that these are mesh-related artifacts. See below code snippet for an outline of how this can be done.

```
// Example of relocating to existing class

class BaseOpenCvVisualizer3D { // abstract class
 public:
   ...
   virtual void draw3dMesh(...) = 0; //pure virtual function
};


class DummyOpenCvVisualizer3D : public BaseOpenCvVisualizer3D { // derived class
 public:
   ...
   void draw3dMesh(...) override {} // do nothing
};

class OpenCvVisualizer3D : public BaseOpenCvVisualizer3D { // derived class (separate source code file)
 public:
   ...
   void draw3dMesh(...) override { // real implementation
       ...
       cv::viz::Mesh cv_mesh;
       ...
   }
};
```

Since a child class will be instantiated for MeshOptimization as highlighted earlier, the function will be used as below.

```
// Example modification to usage in MeshOptimization.cpp

if (visualizer_) {
     ...
     visualizer_->draw3dMesh(...); // dynamically runs OpenCvVisualizer3D/Dummy implementations
     ...
   }
```

Alternatives considered

- Using #ifdef directives within the MeshOptimization class to exclude the specified functions/object.
    - This is a messy design strategy that we would prefer to only use if no other options are apparent
- Moving these artifacts into a *new* MeshVisualization class
    - The OpenCvVisualization3D class is already home to a number of visualization functions and is already used by MeshOptimization, so a new object doesn't need to be added. However, with this approach, variables would not need to be renamed.

- Keeping the existing MeshOptimizer functions as a wrapper around the functions now moved into OpenCvVisualizer3D.
    - Would ensure any code where this is used from MeshOptimization does not break, however, no examples of this could be found, so assuming this is not the intended use case.

Removing Color Dependence on cv::viz::Color

Excluding OpenCV's viz library requires an alternative for defining color such as in the example call "*cv::viz::Color::white()*". For compatibility with either build configuration, we will create a utility class in ***Kimera-VIO/include/utils*** to return scalar equivalents by color name.

```
// Example include/kimera-vio/utils/ColorUtils.h file

#include <opencv2/opencv.hpp>

class ColorUtils {
public:
   static cv::Scalar white() {
       return cv::Scalar(255, 255, 255);
   }
   ...
   // Add more colors as needed
};
```

Handling Errors with Visualization Disabled

Compile Errors with Visualizer Inclusions

When visualization is excluded from the build, any references to visualizer source headers will throw compile-time errors due to missing declarations. To improve the user's debugging experience for such a scenario, dummy versions of the visualizer headers with identical names will be included. These will contain explicit error directives, alerting the user to the reason for the build failure and how to resolve it.

```
// Example of dummy visualizer header

#error "Kimera-VIO was built without the visualizer module. Please re-compile with
KIMERA_BUILD_VISUALIZER build flag enabled."
```

Runtime Visualizer Flags

It is possible that a user will try to use the visualizer when it is not included in the build. This should not silently occur with no feedback to a user. Visualizer functionality is all tied to the "`visualize`" flag, with the exception of "`visualize_frontend_images`." `KimeraVIO.cpp` should be updated to check these two flags and display a warning message to the user, if true when Kimera-VIO isn't built with visualization.

Alternatives Considered

- **Using GFlag Validators**: Used for breaking configurations where the program should be exited. This is not the case here, since the application will still be able to run without issues to carry out VIO processes.

Proposed PR Rollout

Consider our approach to implementing this optional visualization feature.

Introduce CMake Configuration

Establishes the foundation for source code changes without altering existing functionality for users.

- **PR - Add CMake Build option**
  - Introduce CMake option, **KIMERA_BUILD_VISUALIZER**, to toggle compilation of visualization components.
  - Ensure users cannot invoke the feature prior to full implementation.

Refactor Core KimeraVIO Files

Introduces refactors that keep core VIO functionality intact while reducing the reliance on visualization components. This should encompass the majority of the work and can be split into smaller PRs. References in user guides, parameter files and comments must all reflect the new configuration option and the implications of this change where required.

- **PR - Remove color dependency on OpenCV viz**
  - Replace the usage of OpenCV's cv::viz::Color for custom utility color class to facilitate both build configurations.
- **PRs - Introduce Visualizer abstract classes and placeholder classes, update visualizer classes to be derived from the corresponding abstract class.**
  - This is a prerequisite for updating all other Kimera classes which use the visualizer.
- **PRs - Modify Kimera classes for excluded visualization**
  - Modify classes like *MeshOptimization* to accept custom visualizer interfaces as their constructor arguments.
  - Modify dependent files, class members and functions where necessary to account for this change.
  - Address all OpenCV viz direct references.
  - Modify applicable test files and example scripts that incorporate visualization.
  - This can be phased per class however, PRs can be combined depending on their overlap and scale.

Cross Compilation

Introduces a workflow for cross compiling KimeraVIO for aarch64 devices while providing a template for additional targets.

- **PR - Add Cross Compilation support**
    - Introduce toolchain file and CMake workflow for cross-compiling for aarch64 devices.

Ensuring Functional Consistency

To validate that the changes made to Kimera-VIO maintains the same functionality for both build configurations, we will adopt a straightforward validation process during the rollout of pull requests:

- **Utilize Existing Unit Tests:**
    - Leverage the library's existing unit tests to confirm that core functionality remains intact for each PR.
- **Testing with the KimeraVIO example application::**
    - Perform output comparisons between the library with visualization components enabled and the library without them for a given Euroc dataset.
        - Each individual configuration should be run at a minimum of 3 times.
        - These tests should be run with and without visualization included in the build, with visualization enabled and disabled during run time (using flags).
        - Both stereo and mono configurations should be used.
        - Given the non-deterministic nature of the results from Kimera-VIO, we can conduct a statistical analysis on multiple runs of the dataset for each build configuration. This should highlight any discrepancies in results (e.g. VIO pose estimates) that arise from the changes made.

Appendix

Alternative Testing Considerations

Cross compiling Kimera-VIO tests means they will not run successfully on the build machine of a different architecture. Consider options for testing below:

Method	Description	Pros	Cons
Build Tests Natively for Host	Run tests directly on the host machine where the code is compiled.	Fast setup, no need for target device.	Tests may not reflect actual performance on target hardware.

Use an Emulator (e.g., QEMU)	Use an emulator to simulate the target environment and run the tests.	Simulates target environment, no need for real hardware.	Slower performance, complex to set up
Run Tests on Target Device	Deploy and run tests on the target device.	Tests are run in the real target environment.	Delayed feedback, requires cross-compiling and device.

Opting to build and run tests natively on the host machine offers the fastest turnaround time, avoiding the complexities of cross-compiling or setting up an emulator. Since Kimera-VIO's unit tests do not interface with hardware directly, the differences between running tests on the host and the target should be minimal.

Coupling Testing with Cross Compilation

CMake does not support multiple toolchain configurations in a single build. CMake's ***External_Project*** can be applied to compile a native and cross compiled Kimera target, each as an isolated build. When cross compiling, this native target is a prerequisite for generating the test binary to be run on the host machine. This modification at a high level will require:

- Dissecting and nesting the existing ***CMakeLists.txt*** into separate subdirectories
- Introducing a new root CMake file to invoke isolated builds
- Surfacing applicable configuration (eg. downloading gtest) to the root CMake file to reduce build time

Kimera's CMake structure can be updated to reflect the following:

```
Kimera-VIO
├── CMakeLists.txt # A (root)
├── src
│   ├── CMakeLists.txt # B
│   ├── ...
├── tests
│   ├── CMakeLists.txt # C
│   ├── ...
```

The existing ***CMakeLists.txt*** at the repo's root will be moved to the ***/src*** directory and replaced with one which invokes an independent build for each configuration. Consider an example below.

```
# A

... # general setup, option declarations etc.

if (NOT CMAKE_CROSSCOMPILING OR KIMERA_BUILD_TESTS) # for default native build or when tests enabled
   ExternalProject_Add( host-build # compile Kimera for host
                         SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/src
                         CMAKE_ARGS -DKIMERA_BUILD_TESTS=...
                         ...
                       )
endif()

if (CMAKE_CROSSCOMPILING)
   ExternalProject_Add( target-build # compile Kimera for target
                         SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/src
                         CMAKE_ARGS -DKIMERA_BUILD_TESTS=... -DCMAKE_TOOLCHAIN_FILE=...
                         ...
                       )
endif()
```

Each build will generate the main target and test binary if specified.

```
# B

... # Setup and build main Kimera binary

if(KIMERA_BUILD_TESTS)  # conditionally invoke test building
   add_subdirectory(${CMAKE_SOURCE_DIR}/../tests ${CMAKE_BINARY_DIR}/tests)
endif(KIMERA_BUILD_TESTS)
```

When cross compiling , running tests must be skipped as they will fail on the host's architecture.

```
# C

... # Setup and build Kimera test binary

if (NOT CMAKE_CROSSCOMPILING)
   include(GoogleTest)
   gtest_discover_tests(testBinary PRE_TEST)
endif()
```

While this approach does eliminate the need for manual, sequential builds, the benefits like time saved from build redundancies may not outweigh the complexity a superbuild introduces.