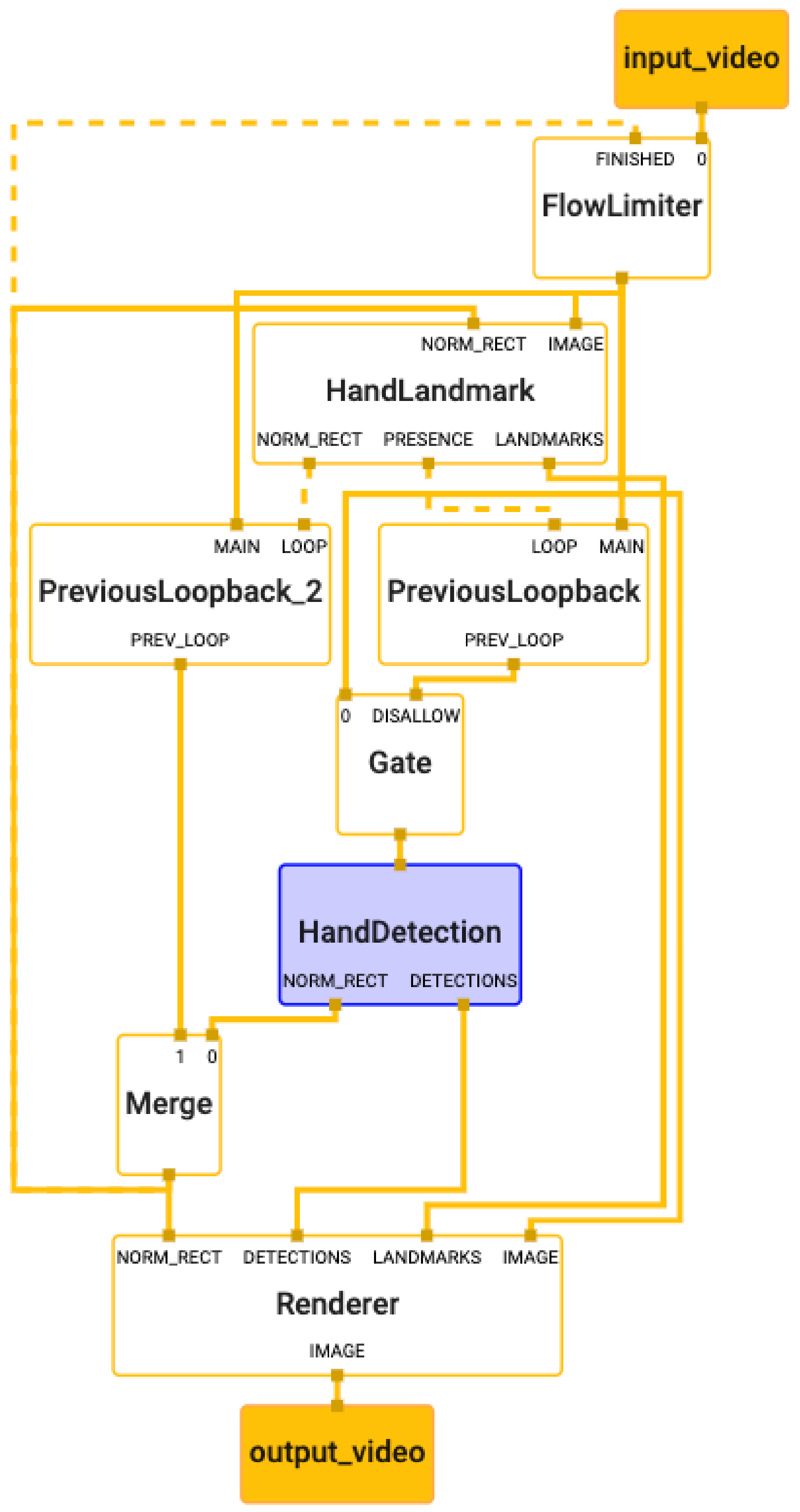
MediaPipe

Release v0.5

Aug 19, 2019

Contents

MediaPipe is a graph-based framework for building multimodal (video, audio, and sensor) applied machine learning pipelines. MediaPipe is cross-platform running on mobile devices, workstations and servers, and supports mobile GPU acceleration. With MediaPipe, an applied machine learning pipeline can be built as a graph of modular components, including, for instance, inference models and media processing functions. Sensory data such as audio and video streams enter the graph, and perceived descriptions such as object-localization and face-landmark streams exit the graph. An example graph that performs real-time hand tracking on mobile GPU is shown below.



MediaPipe is designed for machine learning (ML) practitioners, including researchers, students, and software developers, who implement production-ready ML applications, publish code accompanying research work, and build technology prototypes. The main use case for MediaPipe is rapid prototyping of applied machine learning pipelines with inference models and other reusable components. MediaPipe also facilitates the deployment of machine learning technology into demos and applications on a wide variety of different hardware platforms (e.g., Android, iOS, workstations).

APIs for MediaPipe

- Calculator API in C++
- Graph Construction API in ProtoBuf
- (Coming Soon) Graph Construction API in C++
- Graph Execution API in C++
- Graph Execution API in Java (Android)
- Graph Execution API in Objective-C (iOS)

CHAPTER 1

Alpha Disclaimer

MediaPipe is currently in alpha for v0.6. We are still making breaking API changes and expect to get to stable API by v1.0. We recommend that you target a specific version of MediaPipe, and periodically bump to the latest release. That way you have control over when a breaking change affects you.

CHAPTER 2

User Documentation

2.1 Installing MediaPipe

Note: To interoperate with OpenCV, OpenCV 3.x and above are preferred. OpenCV 2.x currently works but interoperability support may be deprecated in the future.

Note: If you plan to use TensorFlow calculators and example apps, there is a known issue with gcc and g++ version 6.3 and 7.3. Please use other versions.

Choose your operating system:

- *Installing on Debian and Ubuntu*
- *Installing on CentOS*
- *Installing on macOS*
- *Installing on Windows Subsystem for Linux (WSL)*
- *Installing using Docker*

To build and run Android apps:

- *Setting up Android SDK and NDK*
- *Setting up Android Studio with MediaPipe*

To build and run iOS apps:

- Please see the separate iOS setup documentation.

2.1.1 Installing on Debian and Ubuntu

1. Checkout MediaPipe repository.

```
$ git clone https://github.com/google/mediapipe.git

# Change directory into MediaPipe root directory
$ cd mediapipe
```

2. Install Bazel (0.23 and above required).

   Option 1. Use package manager tool to install the latest version of Bazel.

   ```
   $ sudo apt-get install bazel

   # Run 'bazel version' to check version of bazel installed
   ```

   Option 2. Follow Bazel's documentation to install any version of Bazel manually.

3. Install OpenCV.

   Option 1. Use package manager tool to install the pre-compiled OpenCV libraries.

   Note: Debian 9 and Ubuntu 16.04 provide OpenCV 2.4.9. You may want to take option 2 or 3 to install OpenCV 3 or above.

   ```
   $ sudo apt-get install libopencv-core-dev libopencv-highgui-dev \
                          libopencv-imgproc-dev libopencv-video-dev
   ```

   Option 2. Run `setup_opencv.sh` to automatically build OpenCV from source and modify MediaPipe's OpenCV config.

   Option 3. Follow OpenCV's documentation to manually build OpenCV from source code.

   Note: You may need to modify `WORKSAPCE` and `opencv_linux.BUILD` to point MediaPipe to your own OpenCV libraries, e.g., if OpenCV 4 is installed in "/usr/local/", you need to update the "linux_opencv" new_local_repository rule in `WORKSAPCE` and "opencv" cc_library rule in `opencv_linux.BUILD` like the following:

   ```
   new_local_repository(
       name = "linux_opencv",
       build_file = "@//third_party:opencv_linux.BUILD",
       path = "/usr/local",
   )

   cc_library(
     name = "opencv",
     srcs = glob(
         [
             "lib/libopencv_core.so*",
             "lib/libopencv_highgui.so*",
             "lib/libopencv_imgcodecs.so*",
             "lib/libopencv_imgproc.so*",
             "lib/libopencv_video.so*",
             "lib/libopencv_videoio.so*",

         ],
     ),
     hdrs = glob(["include/opencv4/**/*.h*"]),
     includes = ["include/opencv4/"],
     linkstatic = 1,
     visibility = ["//visibility:public"],
   )
   ```

4. Run the Hello World desktop example.

```
$ export GLOG_logtostderr=1
# Need bazel flag 'MEDIAPIPE_DISABLE_GPU=1' as desktop GPU is currently not␣
↪supported
$ bazel run --define MEDIAPIPE_DISABLE_GPU=1 \
    mediapipe/examples/desktop/hello_world:hello_world

# Should print:
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
```

2.1.2 Installing on CentOS

1. Checkout MediaPipe repository.

```
$ git clone https://github.com/google/mediapipe.git

# Change directory into MediaPipe root directory
$ cd mediapipe
```

2. Install Bazel (0.23 and above required).

   Follow Bazel's documentation to install Bazel manually.

3. Install OpenCV.

   Option 1. Use package manager tool to install the pre-compiled version.

   Note: yum installs OpenCV 2.4.5, which may have an opencv/gstreamer issue.

```
$ sudo yum install opencv-devel
```

   Option 2. Build OpenCV from source code.

   Note: You may need to modify `WORKSAPCE` and `opencv_linux.BUILD` to point MediaPipe to your own OpenCV libraries, e.g., if OpenCV 4 is installed in "/usr/local/", you need to update the "linux_opencv" new_local_repository rule in `WORKSAPCE` and "opencv" cc_library rule in `opencv_linux.BUILD` like the following:

```
new_local_repository(
    name = "linux_opencv",
    build_file = "@//third_party:opencv_linux.BUILD",
    path = "/usr/local",
)

cc_library(
  name = "opencv",
  srcs = glob(
      [
```

(continues on next page)

(continued from previous page)

```
            "lib/libopencv_core.so*",
            "lib/libopencv_highgui.so*",
            "lib/libopencv_imgcodecs.so*",
            "lib/libopencv_imgproc.so*",
            "lib/libopencv_video.so*",
            "lib/libopencv_videoio.so*",

        ],
    ),
    hdrs = glob(["include/opencv4/**/*.h*"]),
    includes = ["include/opencv4/"],
    linkstatic = 1,
    visibility = ["//visibility:public"],
)
```

4. Run the Hello World desktop example.

```
$ export GLOG_logtostderr=1
# Need bazel flag 'MEDIAPIPE_DISABLE_GPU=1' as desktop GPU is currently not␣
↪supported
$ bazel run --define MEDIAPIPE_DISABLE_GPU=1 \
    mediapipe/examples/desktop/hello_world:hello_world

# Should print:
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
```

2.1.3 Installing on macOS

1. Prework:
    - Install Homebrew.
    - Install Xcode and its Command Line Tools.
2. Checkout MediaPipe repository.

```
$ git clone https://github.com/google/mediapipe.git

$ cd mediapipe
```

3. Install Bazel (0.23 and above required).

   Option 1. Use package manager tool to install the latest version of Bazel.

```
$ brew install bazel

# Run 'bazel version' to check version of bazel installed
```

Option 2. Follow Bazel's documentation to install any version of Bazel manually.

4. Install OpenCV.

   Option 1. Use HomeBrew package manager tool to install the pre-compiled OpenCV libraries.

   ```
   $ brew install opencv
   ```

   Option 2. Use MacPorts package manager tool to install the OpenCV libraries.

   ```
   $ port install opencv
   ```

   Note: when using MacPorts, please edit the `WORKSAPCE` and `opencv_linux.BUILD` files like the following:

   ```
   new_local_repository(
     name = "macos_opencv",
     build_file = "@//third_party:opencv_macos.BUILD",
     path = "/opt",
   )

   cc_library(
     name = "opencv",
     srcs = glob(
       [
           "local/lib/libopencv_core.dylib",
           "local/lib/libopencv_highgui.dylib",
           "local/lib/libopencv_imgcodecs.dylib",
           "local/lib/libopencv_imgproc.dylib",
           "local/lib/libopencv_video.dylib",
           "local/lib/libopencv_videoio.dylib",
       ],
     ),
     hdrs = glob(["local/include/opencv2/**/*.h*"]),
     includes = ["local/include/"],
     linkstatic = 1,
     visibility = ["//visibility:public"],
   )
   ```

5. Run the Hello World desktop example.

   ```
   $ export GLOG_logtostderr=1
   # Need bazel flag 'MEDIAPIPE_DISABLE_GPU=1' as desktop GPU is currently not␣
   ↪supported
   $ bazel run --define MEDIAPIPE_DISABLE_GPU=1 \
       mediapipe/examples/desktop/hello_world:hello_world

   # Should print:
   # Hello World!
   # Hello World!
   # Hello World!
   # Hello World!
   # Hello World!
   # Hello World!
   # Hello World!
   # Hello World!
   # Hello World!
   # Hello World!
   ```

2.1.4 Installing on Windows Subsystem for Linux (WSL)

1. Follow the instruction to install Windows Sysystem for Linux (Ubuntu).
2. Install Windows ADB and start the ADB server in Windows.

   Note: Window's and WSL's adb versions must be the same version, e.g., if WSL has ADB 1.0.39, you need to download the corresponding Windows ADB from here.
3. Launch WSL.

   Note: All the following steps will be executed in WSL. The Windows directory of the Linux Subsystem can be found in C:\Users\YourUsername\AppData\Local\Packages\CanonicalGroupLimited.UbuntuonWindows_SomeID\LocalState\roo
4. Install the needed packages.

```
username@DESKTOP-TMVLBJ1:~$ sudo apt-get update && sudo apt-get install -y --no-
↪install-recommends build-essential git python zip adb openjdk-8-jdk
```

5. Install Bazel (0.23 and above required).

```
username@DESKTOP-TMVLBJ1:~$ curl -sLO --retry 5 --retry-max-time 10 \
https://storage.googleapis.com/bazel/0.27.0/release/bazel-0.27.0-installer-linux-
↪x86_64.sh && \
sudo mkdir -p /usr/local/bazel/0.27.0 && \
chmod 755 bazel-0.27.0-installer-linux-x86_64.sh && \
sudo ./bazel-0.27.0-installer-linux-x86_64.sh --prefix=/usr/local/bazel/0.27.0 &&␣
↪\
source /usr/local/bazel/0.27.0/lib/bazel/bin/bazel-complete.bash

username@DESKTOP-TMVLBJ1:~$ /usr/local/bazel/0.27.0/lib/bazel/bin/bazel version &&
↪ \
alias bazel='/usr/local/bazel/0.27.0/lib/bazel/bin/bazel'
```

6. Checkout MediaPipe repository.

```
username@DESKTOP-TMVLBJ1:~$ git clone https://github.com/google/mediapipe.git

username@DESKTOP-TMVLBJ1:~$ cd mediapipe
```

7. Install OpenCV.

   Option 1. Use package manager tool to install the pre-compiled OpenCV libraries.

```
username@DESKTOP-TMVLBJ1:~/mediapipe$ sudo apt-get install libopencv-core-dev␣
↪libopencv-highgui-dev \
                       libopencv-imgproc-dev libopencv-video-dev
```

   Option 2. Run `setup_opencv.sh` to automatically build OpenCV from source and modify MediaPipe's OpenCV config.

   Option 3. Follow OpenCV's documentation to manually build OpenCV from source code.

   Note: You may need to modify `WORKSAPCE` and `opencv_linux.BUILD` to point MediaPipe to your own OpenCV libraries, e.g., if OpenCV 4 is installed in "/usr/local/", you need to update the "linux_opencv" new_local_repository rule in `WORKSAPCE` and "opencv" cc_library rule in `opencv_linux.BUILD` like the following:

```
new_local_repository(
    name = "linux_opencv",
```

(continues on next page)

(continued from previous page)

```
    build_file = "@//third_party:opencv_linux.BUILD",
    path = "/usr/local",
)

cc_library(
  name = "opencv",
  srcs = glob(
      [
          "lib/libopencv_core.so*",
          "lib/libopencv_highgui.so*",
          "lib/libopencv_imgcodecs.so*",
          "lib/libopencv_imgproc.so*",
          "lib/libopencv_video.so*",
          "lib/libopencv_videoio.so*",

      ],
  ),
  hdrs = glob(["include/opencv4/**/*.h*"]),
  includes = ["include/opencv4/"],
  linkstatic = 1,
  visibility = ["//visibility:public"],
)
```

8. Run the Hello World desktop example.

```
username@DESKTOP-TMVLBJ1:~/mediapipe$ export GLOG_logtostderr=1

# Need bazel flag 'MEDIAPIPE_DISABLE_GPU=1' as desktop GPU is currently not␣
↪supported
username@DESKTOP-TMVLBJ1:~/mediapipe$ bazel run --define MEDIAPIPE_DISABLE_GPU=1 \
    mediapipe/examples/desktop/hello_world:hello_world

# Should print:
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
```

2.1.5 Installing using Docker

This will use a Docker image that will isolate mediapipe's installation from the rest of the system.

1. Install Docker on your host sytem.
2. Build a docker image with tag "mediapipe".

```
$ git clone https://github.com/google/mediapipe.git
$ cd mediapipe
$ docker build --tag=mediapipe .
```

(continues on next page)

(continued from previous page)

```
# Should print:
# Sending build context to Docker daemon  147.8MB
# Step 1/9 : FROM ubuntu:latest
# latest: Pulling from library/ubuntu
# 6abc03819f3e: Pull complete
# 05731e63f211: Pull complete
# ........
# See http://bazel.build/docs/getting-started.html to start a new project!
# Removing intermediate container 82901b5e79fa
# ---> f5d5f402071b
# Step 9/9 : COPY . /mediapipe/
# ---> a95c212089c5
# Successfully built a95c212089c5
# Successfully tagged mediapipe:latest
```

3. Run the Hello World desktop example.

```
$ docker run -it --name mediapipe mediapipe:latest

root@bca08b91ff63:/mediapipe# GLOG_logtostderr=1 bazel run --define MEDIAPIPE_
↪DISABLE_GPU=1 mediapipe/examples/desktop/hello_world:hello_world

# Should print:
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
# Hello World!
```

2.1.6 Setting up Android SDK and NDK

Requirements:

- Android SDK release 28.0.3 and above.
- Android NDK r17c and above.

MediaPipe recommends setting up Android SDK and NDK via Android Studio, and see *next section* for Android Studio setup. However, if you prefer using MediaPipe without Android Studio, please run `setup_android_sdk_and_ndk.sh` to download and setup Android SDK and NDK before building any Android example apps.

If Android SDK and NDK are already installed (e.g., by Android Studio), set $ANDROID_HOME and $ANDROID_NDK_HOME to point to the installed SDK and NDK.

```
export ANDROID_HOME=<path to the Android SDK>
export ANDROID_NDK_HOME=<path to the Android NDK>
```

Please verify all the necessary packages are installed.

- Android SDK Platform API Level 28 or 29

- Android SDK Build-Tools 28 or 29
- Android SDK Platform-Tools 28 or 29
- Android SDK Tools 26.1.1
- Android NDK 17c or above

2.1.7 Setting up Android Studio with MediaPipe

The steps below use Android Studio to build and install a MediaPipe example app.

1. Install and launch Android Studio.
2. Select `Configure` | `SDK Manager` | `SDK Platforms`.
   - Verify that Android SDK Platform API Level 28 or 29 is installed.
   - Take note of the Android SDK Location, e.g., `/usr/local/home/Android/Sdk`.
3. Select `Configure` | `SDK Manager` | `SDK Tools`.
   - Verify that Android SDK Build-Tools 28 or 29 is installed.
   - Verify that Android SDK Platform-Tools 28 or 29 is installed.
   - Verify that Android SDK Tools 26.1.1 is installed.
   - Verify that Android NDK 17c or above is installed.
   - Take note of the Android NDK Location, e.g., `/usr/local/home/Android/Sdk/ndk-bundle`.
4. Set environment variables `$ANDROID_HOME` and `$ANDROID_NDK_HOME` to point to the installed SDK and NDK.

```
export ANDROID_HOME=/usr/local/home/Android/Sdk
export ANDROID_NDK_HOME=/usr/local/home/Android/Sdk/ndk-bundle
```

5. Select `Configure` | `Plugins` install `Bazel`.
6. Select `Import Bazel Project`.
   - Select `Workspace`: `/path/to/mediapipe`.
   - Select `Generate from BUILD file`: `/path/to/mediapipe/BUILD`.
   - Select `Finish`.
7. Connect an Android device to the workstation.
8. Select `Run...` | `Edit Configurations...`.
   - Enter Target Expression: `//mediapipe/examples/android/src/java/com/google/mediapipe/apps/facedetectioncpu`
   - Enter Bazel command: `mobile-install`
   - Enter Bazel flags: `-c opt --config=android_arm64` select `Run`

2.2 MediaPipe Concepts

2.2.1 The basics

Packet

The basic data flow unit. A packet consists of a numeric timestamp and a shared pointer to an **immutable** payload. The payload can be of any C++ type, and the payload's type is also referred to as the type of the packet. Packets are value classes and can be copied cheaply. Each copy shares ownership of the payload, with reference-counting semantics. Each copy has its own timestamp. Details.

Graph

MediaPipe processing takes place inside a graph, which defines packet flow paths between **nodes**. A graph can have any number of inputs and outputs, and data flow can branch and merge. Generally data flows forward, but backward loops are possible.

Nodes

Nodes produce and/or consume packets, and they are where the bulk of the graph's work takes place. They are also known as "calculators", for historical reasons. Each node's interface defines a number of input and output **ports**, identified by a tag and/or an index.

Streams

A stream is a connection between two nodes that carries a sequence of packets, whose timestamps must be monotonically increasing.

Side packets

A side packet connection between nodes carries a single packet (with unspecified timestamp). It can be used to provide some data that will remain constant, whereas a stream represents a flow of data that changes over time.

Packet Ports

A port has an associated type; packets transiting through the port must be of that type. An output stream port can be connected to any number of input stream ports of the same type; each consumer receives a separate copy of the output packets, and has its own queue, so it can consume them at its own pace. Similarly, a side packet output port can be connected to as many side packet input ports as desired.

A port can be required, meaning that a connection must be made for the graph to be valid, or optional, meaning it may remain unconnected.

Note: even if a stream connection is required, the stream may not carry a packet for all timestamps.

2.2.2 Input and output

Data flow can originate from **source nodes**, which have no input streams and produce packets spontaneously (e.g. by reading from a file); or from **graph input streams**, which let an application feed packets into a graph.

Similarly, there are **sink nodes** that receive data and write it to various destinations (e.g. a file, a memory buffer, etc.), and an application can also receive output from the graph using **callbacks**.

2.2.3 Runtime behavior

Graph lifetime

Once a graph has been initialized, it can be **started** to begin processing data, and can process a stream of packets until each stream is closed or the graph is **canceled**. Then the graph can be destroyed or **started** again.

Node lifetime

There are three main lifetime methods the framework will call on a node:

- Open: called once, before the other methods. When it is called, all input side packets required by the node will be available.
- Process: called multiple times, when a new set of inputs is available, according to the node's input policy.
- Close: called once, at the end.

In addition, each calculator can define constructor and destructor, which are useful for creating and deallocating resources that are independent of the processed data.

Input policies

The default input policy is deterministic collation of packets by timestamp. A node receives all inputs for the same timestamp at the same time, in an invocation of its Process method; and successive input sets are received in their timestamp order. This can require delaying the processing of some packets until a packet with the same timestamp is received on all input streams, or until it can be guaranteed that a packet with that timestamp will not be arriving on the streams that have not received it.

Other policies are also available, implemented using a separate kind of component known as an InputStreamHandler.

See *scheduling* for more details.

2.3 Building MediaPipe Calculators

2.3.1 Example calculator

This section discusses the implementation of `PacketClonerCalculator`, which does a relatively simple job, and is used in many calculator graphs. `PacketClonerCalculator` simply produces a copy of its most recent input packets on demand.

`PacketClonerCalculator` is useful when the timestamps of arriving data packets are not aligned perfectly. Suppose we have a room with a microphone, light sensor and a video camera that is collecting sensory data. Each of the sensors operates independently and collects data intermittently. Suppose that the output of each sensor is:

- microphone = loudness in decibels of sound in the room (Integer)
- light sensor = brightness of room (Integer)
- video camera = RGB image frame of room (ImageFrame)

Our simple perception pipeline is designed to process sensory data from these 3 sensors such that at any time when we have image frame data from the camera that is synchronized with the last collected microphone loudness data and light sensor brightness data. To do this with MediaPipe, our perception pipeline has 3 input streams:

- room_mic_signal - Each packet of data in this input stream is integer data representing how loud audio is in a room with timestamp.
- room_lightening_sensor - Each packet of data in this input stream is integer data representing how bright is the room illuminated with timestamp.
- room_video_tick_signal - Each packet of data in this input stream is imageframe of video data representing video collected from camera in the room with timestamp.

Below is the implementation of the `PacketClonerCalculator`. You can see the `GetContract()`, `Open()`, and `Process()` methods as well as the instance variable `current_` which holds the most recent input packets.

```
// This takes packets from N+1 streams, A_1, A_2, ..., A_N, B.
// For every packet that appears in B, outputs the most recent packet from each
// of the A_i on a separate stream.

#include <vector>

#include "absl/strings/str_cat.h"
#include "mediapipe/framework/calculator_framework.h"

namespace mediapipe {

// For every packet received on the last stream, output the latest packet
// obtained on all other streams. Therefore, if the last stream outputs at a
// higher rate than the others, this effectively clones the packets from the
// other streams to match the last.
//
// Example config:
// node {
//   calculator: "PacketClonerCalculator"
//   input_stream: "first_base_signal"
//   input_stream: "second_base_signal"
//   input_stream: "tick_signal"
//   output_stream: "cloned_first_base_signal"
//   output_stream: "cloned_second_base_signal"
// }
//
class PacketClonerCalculator : public CalculatorBase {
 public:
  static ::mediapipe::Status GetContract(CalculatorContract* cc) {
    const int tick_signal_index = cc->Inputs().NumEntries() - 1;
    // cc->Inputs().NumEntries() returns the number of input streams
    // for the PacketClonerCalculator
    for (int i = 0; i < tick_signal_index; ++i) {
      cc->Inputs().Index(i).SetAny();
      // cc->Inputs().Index(i) returns the input stream pointer by index
      cc->Outputs().Index(i).SetSameAs(&cc->Inputs().Index(i));
    }
    cc->Inputs().Index(tick_signal_index).SetAny();
    return ::mediapipe::OkStatus();
  }

  ::mediapipe::Status Open(CalculatorContext* cc) final {
    tick_signal_index_ = cc->Inputs().NumEntries() - 1;
```

(continues on next page)

(continued from previous page)

```
    current_.resize(tick_signal_index_);
    // Pass along the header for each stream if present.
    for (int i = 0; i < tick_signal_index_; ++i) {
      if (!cc->Inputs().Index(i).Header().IsEmpty()) {
        cc->Outputs().Index(i).SetHeader(cc->Inputs().Index(i).Header());
        // Sets the output stream of index i header to be the same as
        // the header for the input stream of index i
      }
    }
    return ::mediapipe::OkStatus();
  }

  ::mediapipe::Status Process(CalculatorContext* cc) final {
    // Store input signals.
    for (int i = 0; i < tick_signal_index_; ++i) {
      if (!cc->Inputs().Index(i).Value().IsEmpty()) {
        current_[i] = cc->Inputs().Index(i).Value();
      }
    }

    // Output if the tick signal is non-empty.
    if (!cc->Inputs().Index(tick_signal_index_).Value().IsEmpty()) {
      for (int i = 0; i < tick_signal_index_; ++i) {
        if (!current_[i].IsEmpty()) {
          cc->Outputs().Index(i).AddPacket(
              current_[i].At(cc->InputTimestamp()));
          // Add a packet to output stream of index i a packet from inputstream i
          // with timestamp common to all present inputs
          //
        } else {
          cc->Outputs().Index(i).SetNextTimestampBound(
              cc->InputTimestamp().NextAllowedInStream());
          // if current_[i], 1 packet buffer for input stream i is empty, we will set
          // next allowed timestamp for input stream i to be current timestamp + 1
        }
      }
    }
    return ::mediapipe::OkStatus();
  }

 private:
  std::vector<Packet> current_;
  int tick_signal_index_;
};

REGISTER_CALCULATOR(PacketClonerCalculator);
}  // namespace mediapipe
```

Typically, a calculator has only a .cc file. No .h is required, because mediapipe uses registration to make calculators known to it. After you have defined your calculator class, register it with a macro invocation REGISTER_CALCULATOR(calculator_class_name).

Below is a trivial MediaPipe graph that has 3 input streams, 1 node (PacketClonerCalculator) and 3 output streams.

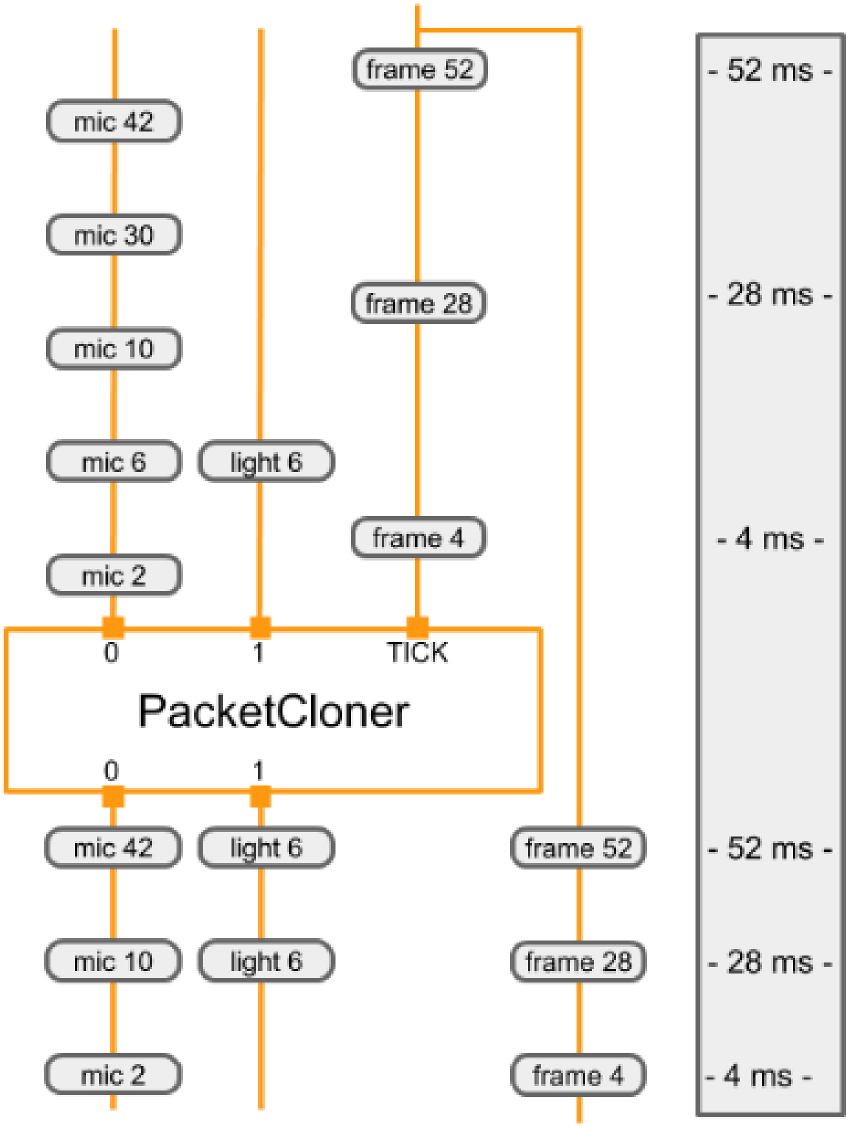
```
input_stream: "room_mic_signal"
input_stream: "room_lighting_sensor"
input_stream: "room_video_tick_signal"
```

(continues on next page)

(continued from previous page)

```
node {
   calculator: "PacketClonerCalculator"
   input_stream: "room_mic_signal"
   input_stream: "room_lighting_sensor"
   input_stream: "room_video_tick_signal"
   output_stream: "cloned_room_mic_signal"
   output_stream: "cloned_lighting_sensor"
   output_stream: "cloned_video_tick_signal"
 }
```

The diagram below shows how the `PacketClonerCalculator` defines its output packets based on its series of input packets.



| Graph using PacketClonerCalculator | |:–:| | *Each time it receives a packet on its TICK input stream, the PacketClonerCalculator outputs the most recent packet from each of its input streams. The sequence of output packets is determined by the sequene of input packets and their timestamps. The timestamps are shows along the right side of the diagram.* |

2.4 Examples

Below are code samples on how to run MediaPipe on both mobile and desktop. We currently support MediaPipe APIs on mobile for Android only but will add support for Objective-C shortly.

2.4.1 Mobile

Hello World! on Android

Hello World! on Android should be the first mobile Android example users go through in detail. It teaches the following:

- Introduction of a simple MediaPipe graph running on mobile GPUs for Sobel edge detection.
- Building a simple baseline Android application that displays "Hello World!".
- Adding camera preview support into the baseline application using the Android CameraX API.
- Incorporating the Sobel edge detection graph to process the live camera preview and display the processed video in real-time.

Hello World! on iOS

Hello World! on iOS is the iOS version of Sobel edge detection example.

Object Detection with GPU

Object Detection with GPU illustrates how to use MediaPipe with a TFLite model for object detection in a GPU-accelerated pipeline.

- Android
- iOS

Object Detection with CPU

Object Detection with CPU illustrates using the same TFLite model in a CPU-based pipeline. This example highlights how graphs can be easily adapted to run on CPU v.s. GPU.

Face Detection with GPU

Face Detection with GPU illustrates how to use MediaPipe with a TFLite model for face detection in a GPU-accelerated pipeline. The selfie face detection TFLite model is based on "BlazeFace: Sub-millisecond Neural Face Detection on Mobile GPUs", and model details are described in the model card.

- Android
- iOS

Hand Detection with GPU

Hand Detection with GPU illustrates how to use MediaPipe with a TFLite model for hand detection in a GPU-accelerated pipeline.

- Android
- iOS

Hand Tracking with GPU

Hand Tracking with GPU illustrates how to use MediaPipe with a TFLite model for hand tracking in a GPU-accelerated pipeline.

- Android
- iOS

Hair Segmentation with GPU

Hair Segmentation on GPU illustrates how to use MediaPipe with a TFLite model for hair segmentation in a GPU-accelerated pipeline. The selfie hair segmentation TFLite model is based on "Real-time Hair segmentation and recoloring on Mobile GPUs", and model details are described in the model card.

- Android

2.4.2 Desktop

Hello World for C++

Hello World for C++ shows how to run a simple graph using the MediaPipe C++ APIs.

Preparing Data Sets with MediaSequence

Preparing Data Sets with MediaSequence shows how to use MediaPipe for media processing to prepare video data sets for training a TensorFlow model.

Object Detection on Desktop

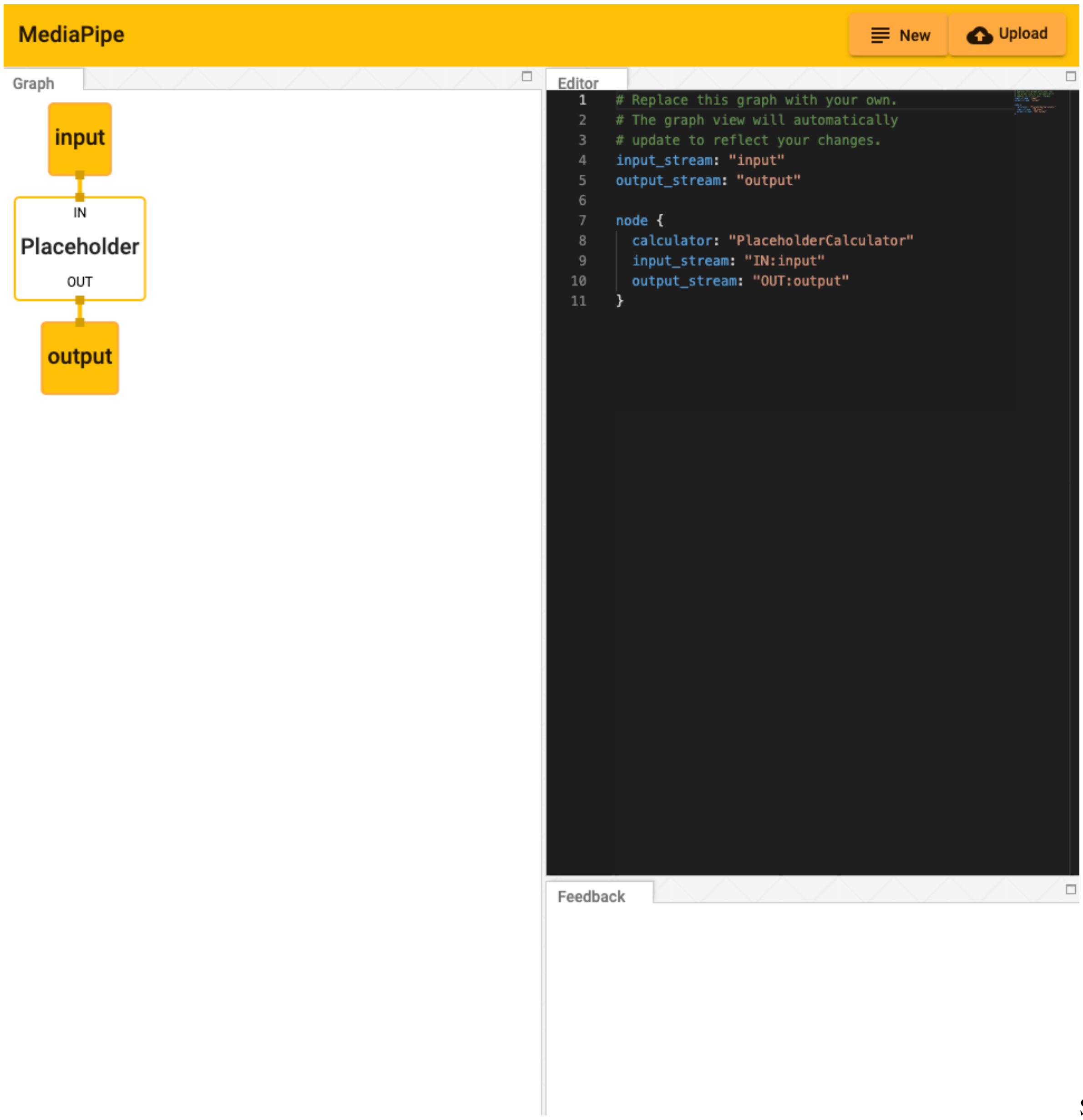
Object Detection on Desktop shows how to run object detection models (TensorFlow and TFLite) using the MediaPipe C++ APIs.

2.5 Visualizing MediaPipe Graphs

- *Working within the Editor*
- *Understanding the Graph*
- *Visualizing Subgraphs*

To help users understand the structure of their calculator graphs and to understand the overall behavior of their machine learning inference pipelines, we have built the MediaPipe Visualizer that is available online.

- A graph view allows users to see a connected calculator graph as expressed through a graph configuration that is pasted into the graph editor or uploaded. The user can visualize and troubleshoot a graph they have created.



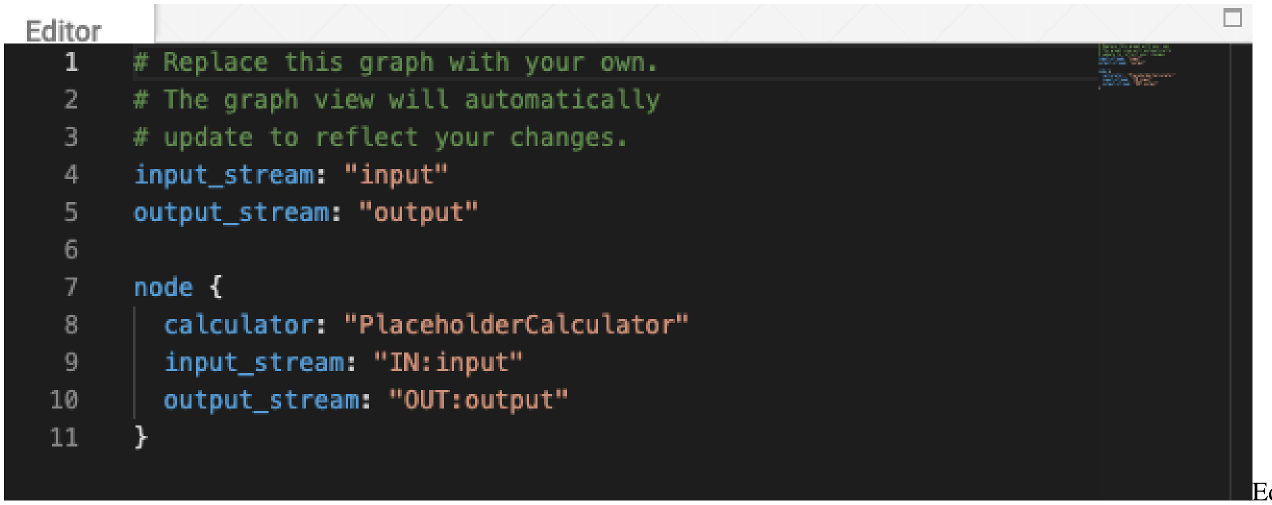
```
# Replace this graph with your own.
# The graph view will automatically
# update to reflect your changes.
input_stream: "input"
output_stream: "output"

node {
  calculator: "PlaceholderCalculator"
  input_stream: "IN:input"
  output_stream: "OUT:output"
}
```

Startup screen

2.5.1 Working within the Editor

Getting Started:

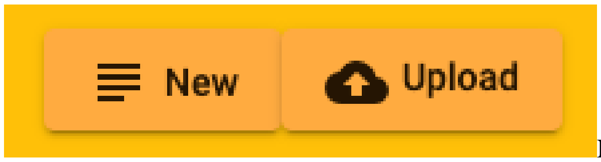
The graph can be modified by adding and editing code in the Editor view.



```
# Replace this graph with your own.
# The graph view will automatically
# update to reflect your changes.
input_stream: "input"
output_stream: "output"

node {
  calculator: "PlaceholderCalculator"
  input_stream: "IN:input"
  output_stream: "OUT:output"
}
```

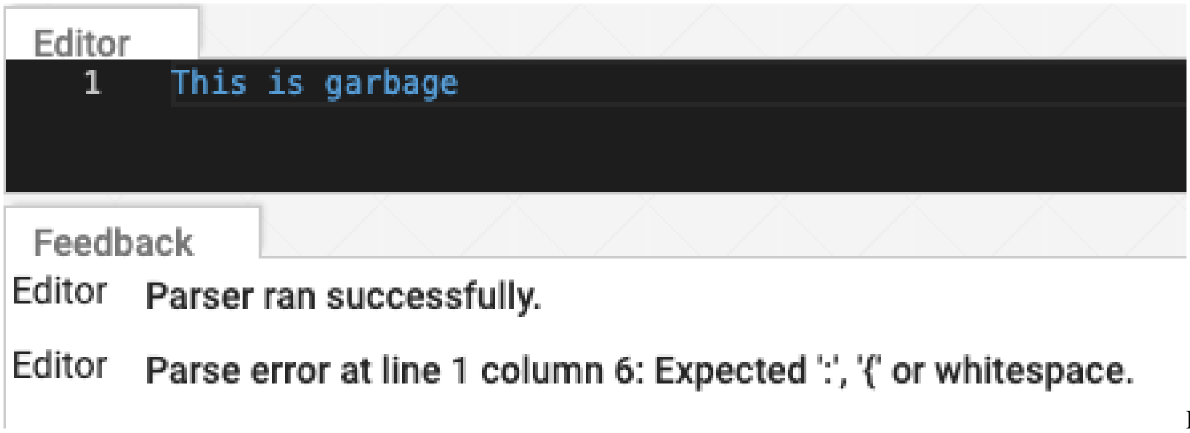
Editor UI

- Pressing the "New" button in the upper right corner will clear any existing code in the Editor window.



New Button

- Pressing the "Upload" button will prompt the user to select a local PBTXT file, which will everwrite the current code within the editor.
- Alternatively, code can be pasted directly into the editor window.
- Errors and informational messages will appear in the Feedback window.



```
This is garbage
```

Editor Parser ran successfully.

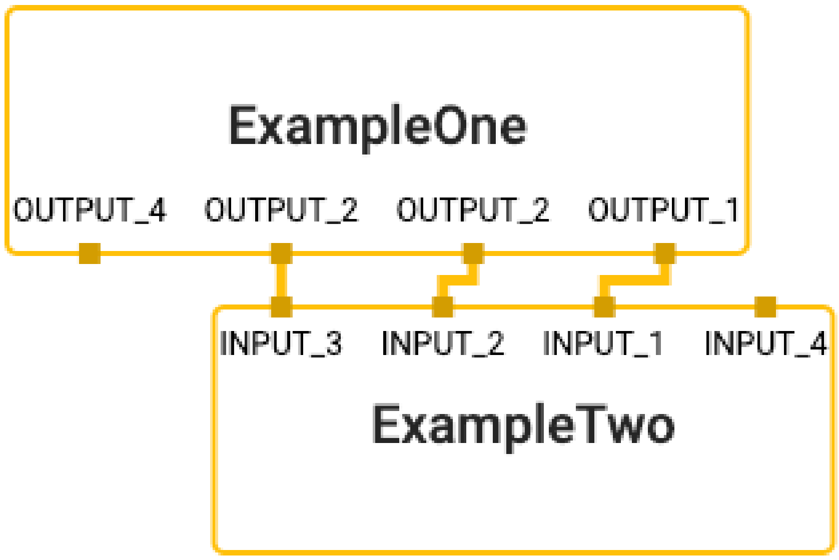
Editor Parse error at line 1 column 6: Expected ':', '{' or whitespace.

Error Msg

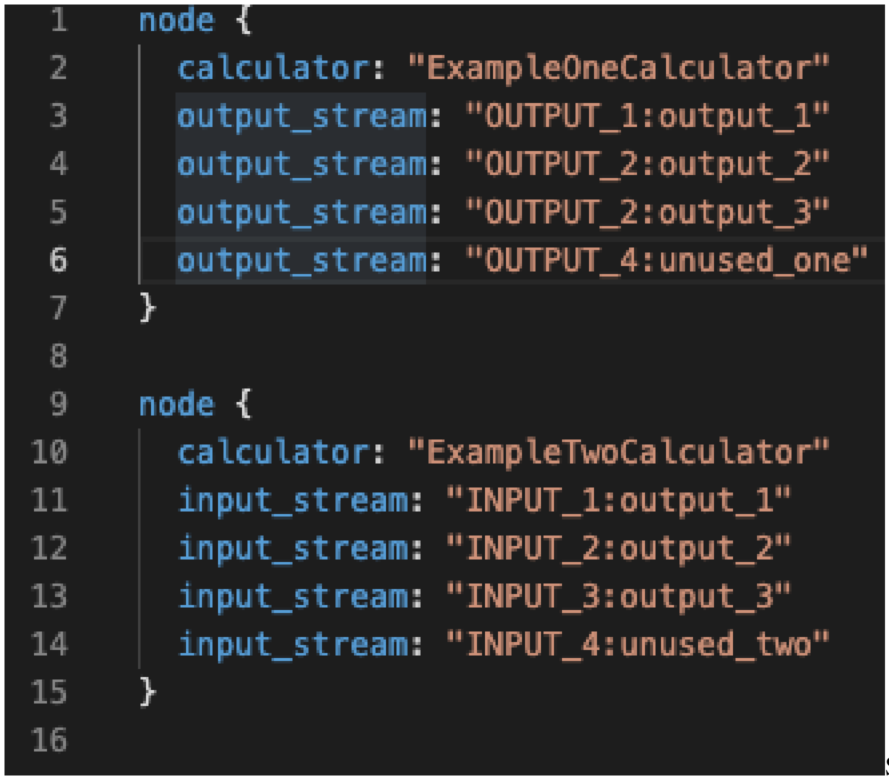
2.5.2 Understanding the Graph

The visualizer graph shows the connections between calculator nodes.

- Streams exit from the bottom of the calculator producing the stream and enter the top of any calculator receiving the stream. (Notice the use of the key, "input_stream" and "output_stream").



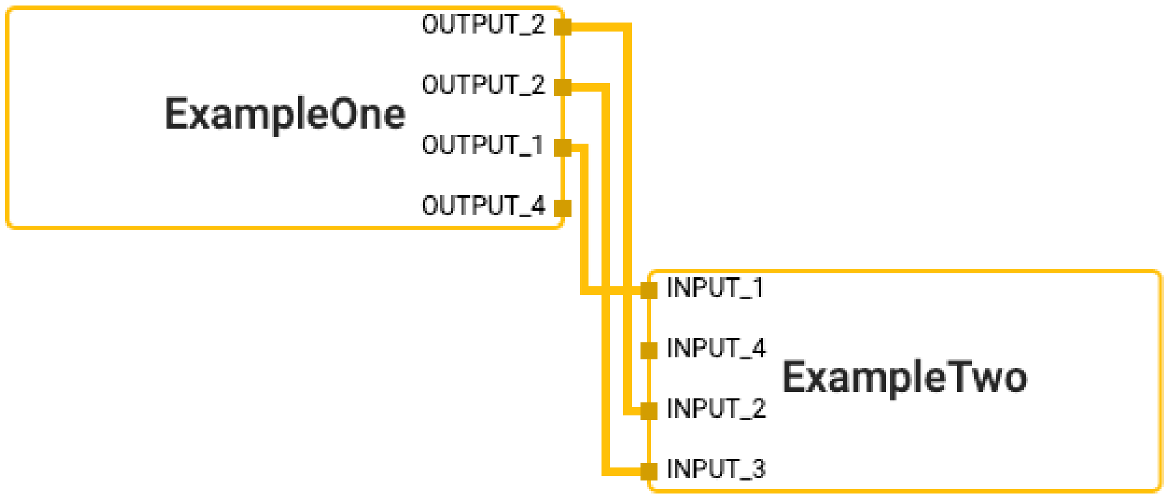
Stream UI



```
node {
  calculator: "ExampleOneCalculator"
  output_stream: "OUTPUT_1:output_1"
  output_stream: "OUTPUT_2:output_2"
  output_stream: "OUTPUT_2:output_3"
  output_stream: "OUTPUT_4:unused_one"
}

node {
  calculator: "ExampleTwoCalculator"
  input_stream: "INPUT_1:output_1"
  input_stream: "INPUT_2:output_2"
  input_stream: "INPUT_3:output_3"
  input_stream: "INPUT_4:unused_two"
}
```

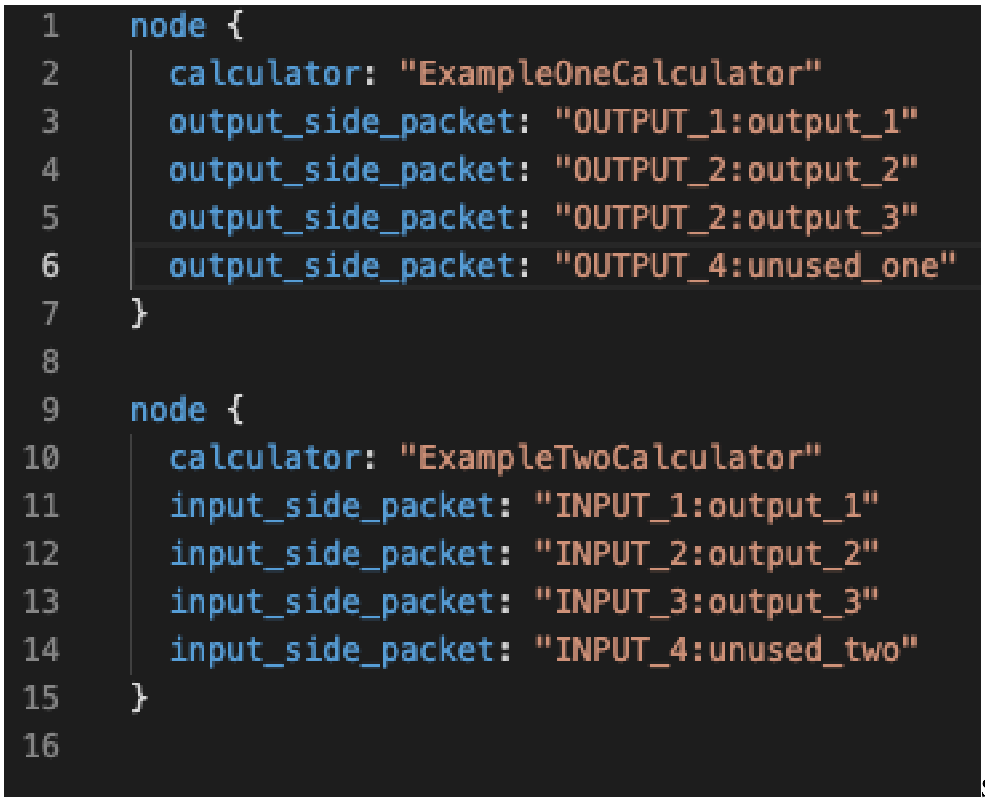
Stream_code

- Sidepackets work the same, except that they exit a node on the right and enter on the left. (Notice the use of the key, “input_side_packet” and “output_side_packet”).



Sidepacket

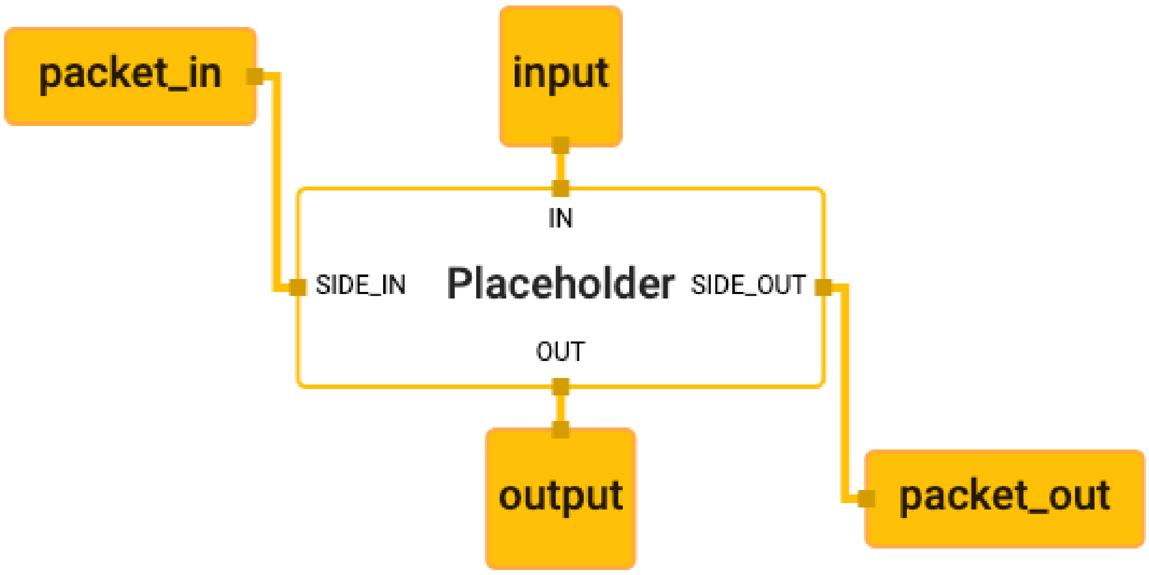
UI



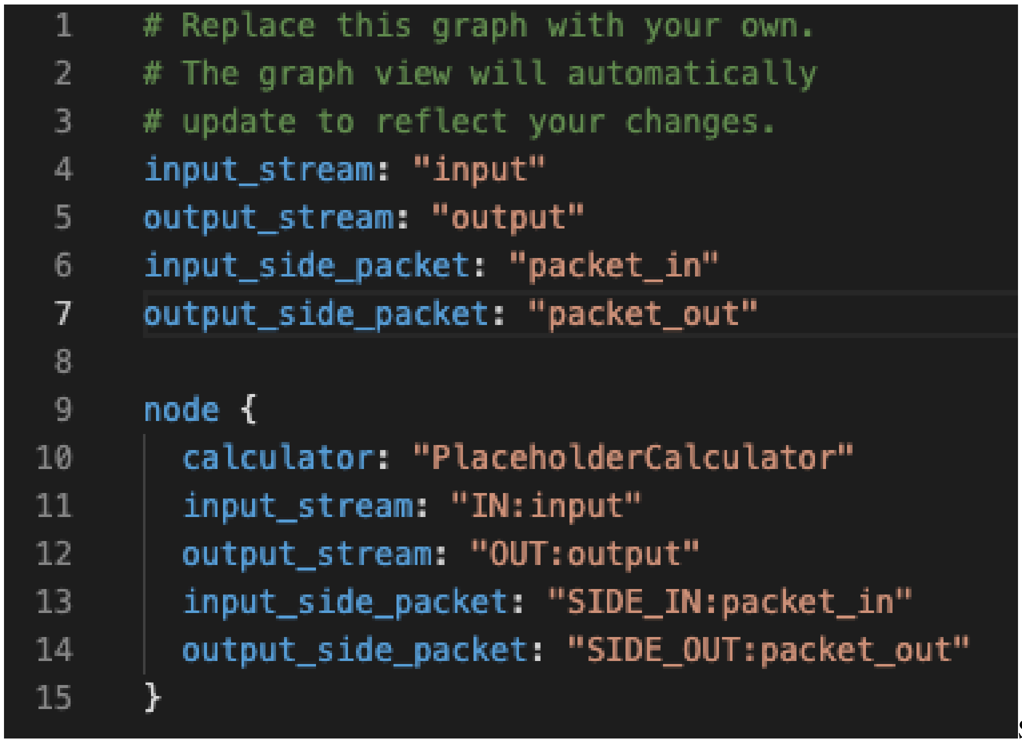
```
node {
  calculator: "ExampleOneCalculator"
  output_side_packet: "OUTPUT_1:output_1"
  output_side_packet: "OUTPUT_2:output_2"
  output_side_packet: "OUTPUT_2:output_3"
  output_side_packet: "OUTPUT_4:unused_one"
}

node {
  calculator: "ExampleTwoCalculator"
  input_side_packet: "INPUT_1:output_1"
  input_side_packet: "INPUT_2:output_2"
  input_side_packet: "INPUT_3:output_3"
  input_side_packet: "INPUT_4:unused_two"
}
```

Sidepacket_code

- There are special nodes that represent inputs and outputs to the graph and can supply either side packets or streams.



Special nodes



```
# Replace this graph with your own.
# The graph view will automatically
# update to reflect your changes.
input_stream: "input"
output_stream: "output"
input_side_packet: "packet_in"
output_side_packet: "packet_out"

node {
  calculator: "PlaceholderCalculator"
  input_stream: "IN:input"
  output_stream: "OUT:output"
  input_side_packet: "SIDE_IN:packet_in"
  output_side_packet: "SIDE_OUT:packet_out"
}
```

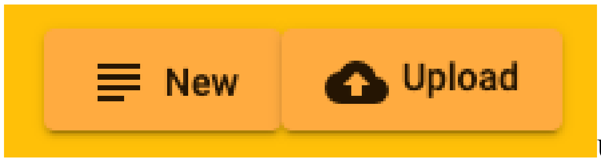
Special nodes

2.5.3 Visualizing Subgraphs

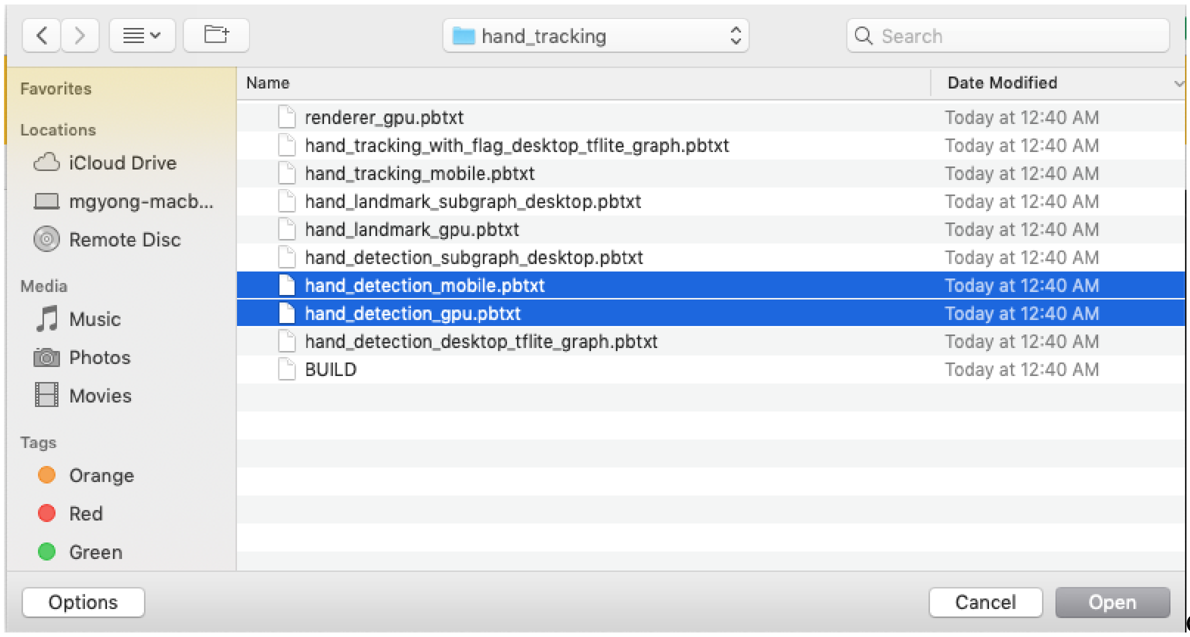
The MediaPipe visualizer can display multiple graphs in separate tabs. If a graph has a `type` field in the top level of the graph's text proto definition, and that value of graph `type` is used as a calculator name in another graph, it is considered a subgraph by the visualizer and colored appropriately where it is used. Clicking on a subgraph will navigate to the corresponding tab which holds the subgraph's definition.

For instance, there are two graphs involved in the hand detection example: the main graph (source pbtxt file) and its associated subgraph (source pbtxt file). To visualize them:

- In the MediaPipe visualizer, click on the upload graph button and select the 2 pbtxt files to visualize (main graph and its associated subgraph).



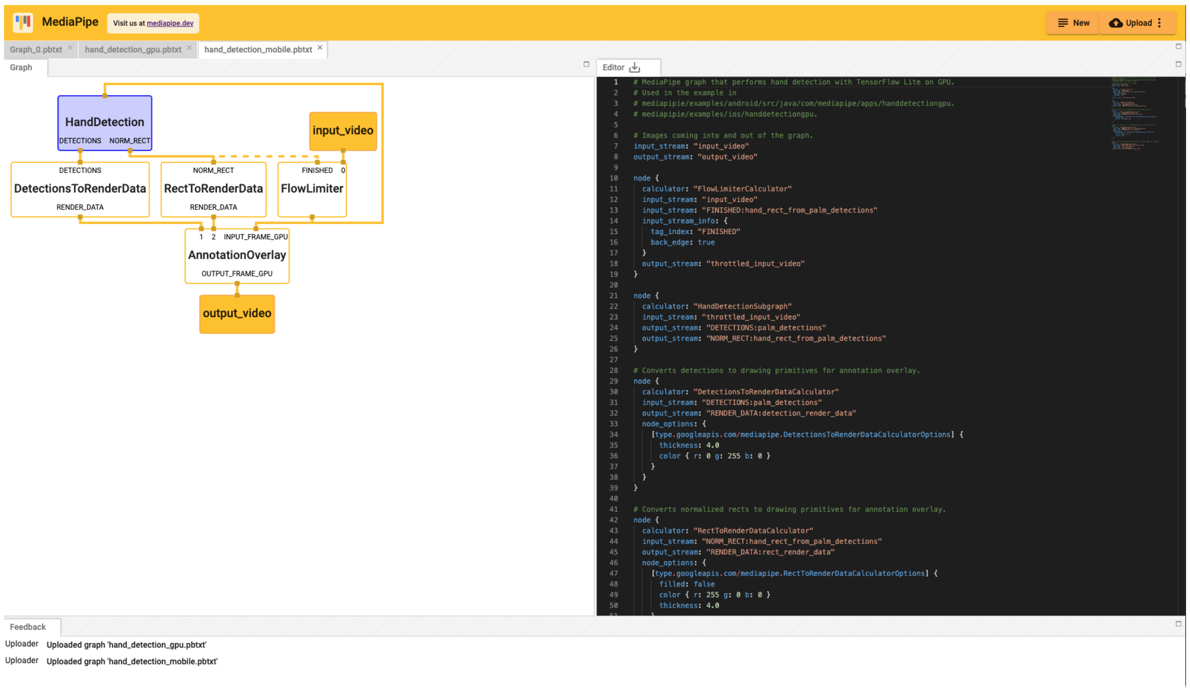
Upload graph button



Name	Date Modified
renderer_gpu.pbtxt	Today at 12:40 AM
hand_tracking_with_flag_desktop_tflite_graph.pbtxt	Today at 12:40 AM
hand_tracking_mobile.pbtxt	Today at 12:40 AM
hand_landmark_subgraph_desktop.pbtxt	Today at 12:40 AM
hand_landmark_gpu.pbtxt	Today at 12:40 AM
hand_detection_subgraph_desktop.pbtxt	Today at 12:40 AM
hand_detection_mobile.pbtxt	Today at 12:40 AM
hand_detection_gpu.pbtxt	Today at 12:40 AM
hand_detection_desktop_tflite_graph.pbtxt	Today at 12:40 AM
BUILD	Today at 12:40 AM

Choose the 2 files

- There will be 2 additional tabs. The main graph tab is `hand_detection_mobile.pbtxt`.

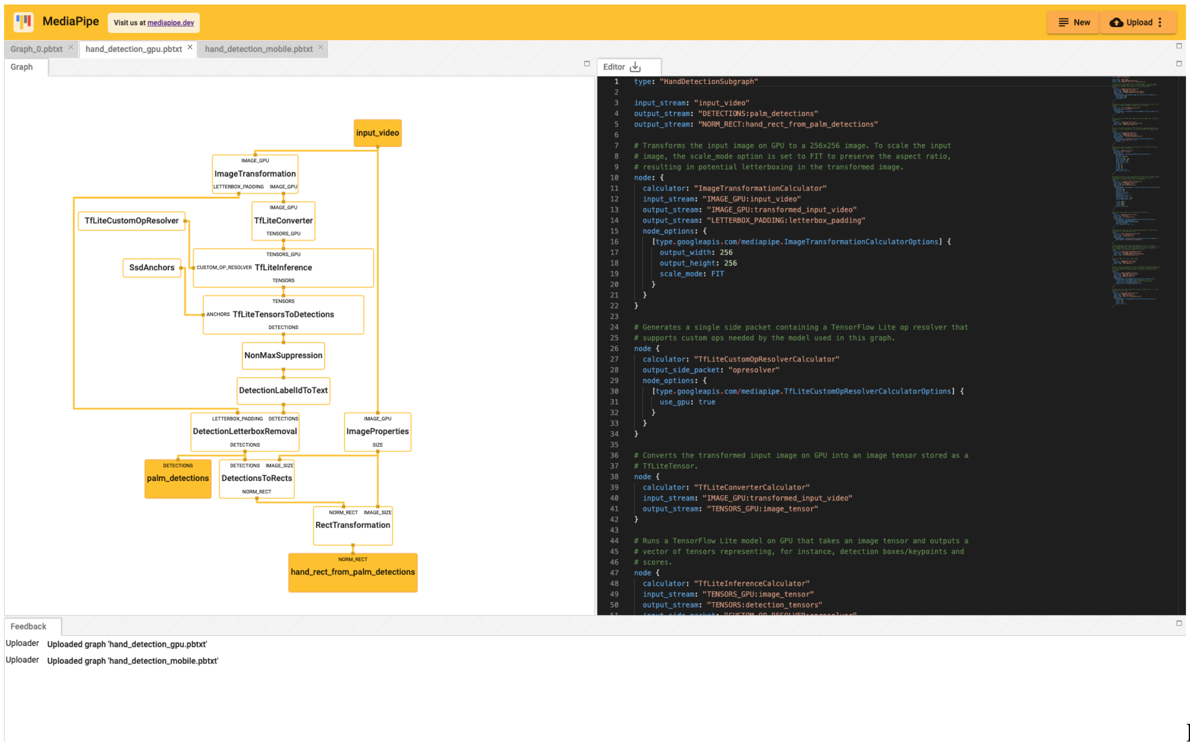


```
# MediaPipe graph that performs hand detection with TensorFlow Lite on GPU.
# Used in the example in
# mediapipie/examples/android/src/java/com/mediapipe/apps/handdetectiongpu.
# mediapipie/examples/ios/handdetectiongpu.

# Images coming into and out of the graph.
input_stream: "input_video"
output_stream: "output_video"

node {
  calculator: "FlowLimiterCalculator"
  input_stream: "input_video"
  input_stream: "FINISHED:hand_rect_from_palm_detections"
  input_stream_info: {
    tag_index: "FINISHED"
    back_edge: true
  }
  output_stream: "throttled_input_video"
}

node {
  calculator: "HandDetectionSubgraph"
  input_stream: "throttled_input_video"
  output_stream: "DETECTIONS:palm_detections"
  output_stream: "NORM_RECT:hand_rect_from_palm_detections"
}

# Converts detections to drawing primitives for annotation overlay.
node {
  calculator: "DetectionsToRenderDataCalculator"
  input_stream: "DETECTIONS:palm_detections"
  output_stream: "RENDER_DATA:detection_render_data"
  node_options: {
    [type.googleapis.com/mediapipe.DetectionsToRenderDataCalculatorOptions] {
      thickness: 4.0
      color { r: 0 g: 255 b: 0 }
    }
  }
}

# Converts normalized rects to drawing primitives for annotation overlay.
node {
  calculator: "RectToRenderDataCalculator"
  input_stream: "NORM_RECT:hand_rect_from_palm_detections"
  output_stream: "RENDER_DATA:rect_render_data"
  node_options: {
    [type.googleapis.com/mediapipe.RectToRenderDataCalculatorOptions] {
      filled: false
      color { r: 255 g: 0 b: 0 }
      thickness: 4.0
```

```
Uploader  Uploaded graph 'hand_detection_gpu.pbtxt'
Uploader  Uploaded graph 'hand_detection_mobile.pbtxt'
```

hand_detection_m

- Clicking on the `HandDetection` node in purple redirects the view to the `hand_detection_gpu.pbtxt` tab.



```
type: "HandDetectionSubgraph"

input_stream: "input_video"
output_stream: "DETECTIONS:palm_detections"
output_stream: "NORM_RECT:hand_rect_from_palm_detections"

# Transforms the input image on GPU to a 256x256 image. To scale the input
# image, the scale_mode option is set to FIT to preserve the aspect ratio,
# resulting in potential letterboxing in the transformed image.
node: {
  calculator: "ImageTransformationCalculator"
  input_stream: "IMAGE_GPU:input_video"
  output_stream: "IMAGE_GPU:transformed_input_video"
  output_stream: "LETTERBOX_PADDING:letterbox_padding"
  node_options: {
    [type.googleapis.com/mediapipe.ImageTransformationCalculatorOptions] {
      output_width: 256
      output_height: 256
      scale_mode: FIT
    }
  }
}

# Generates a single side packet containing a TensorFlow Lite op resolver that
# supports custom ops needed by the model used in this graph.
node {
  calculator: "TfLiteCustomOpResolverCalculator"
  output_side_packet: "opresolver"
  node_options: {
    [type.googleapis.com/mediapipe.TfLiteCustomOpResolverCalculatorOptions] {
      use_gpu: true
    }
  }
}

# Converts the transformed input image on GPU into an image tensor stored as a
# TfLiteTensor.
node {
  calculator: "TfLiteConverterCalculator"
  input_stream: "IMAGE_GPU:transformed_input_video"
  output_stream: "TENSORS_GPU:image_tensor"
}

# Runs a TensorFlow Lite model on GPU that takes an image tensor and outputs a
# vector of tensors representing, for instance, detection boxes/keypoints and
# scores.
node {
  calculator: "TfLiteInferenceCalculator"
  input_stream: "TENSORS_GPU:image_tensor"
  output_stream: "TENSORS:detection_tensors"
```

```
Uploader   Uploaded graph 'hand_detection_gpu.pbtxt'
Uploader   Uploaded graph 'hand_detection_mobile.pbtxt'
```

Hand detection subgraph

2.6 Measuring Performance

Coming soon.

MediaPipe includes APIs for gathering aggregate performance data and event timing data for CPU and GPU operations. These API's can be found at:

- `GraphProfiler`: Accumulates for each running calculator a histogram of latencies for Process calls.
- `GraphTracer`: Records for each running calculator and each processed packet a series of timed events including the start and finish of each Process call.

Future mediapipe releases will include tools for visualizing and analysing the latency histograms and timed events captured by these API's.

2.7 Questions and Answers

- *How to convert ImageFrames and GpuBuffers*
- *How to visualize perceived results*
- *How to run calculators in parallel*
- *Output timestamps when using ImmediateInputStreamHandler*
- *How to change settings at runtime*

- *How to process real-time input streams*
- *Can I run MediaPipe on MS Windows?*

2.7.1 How to convert ImageFrames and GpuBuffers

The Calculators `ImageFrameToGpuBufferCalculator` and `GpuBufferToImageFrameCalculator` convert back and forth between packets of type `ImageFrame` and `GpuBuffer`. `ImageFrame` refers to image data in CPU memory in any of a number of bitmap image formats. `GpuBuffer` refers to image data in GPU memory. You can find more detail in the Framework Concepts section *GpuBuffer to ImageFrame converters*. You can see an example in:

- `object_detection_mobile_cpu.pbtxt`

2.7.2 How to visualize perception results

The `AnnotationOverlayCalculator` allows perception results, such as bounding boxes, arrows, and ovals, to be superimposed on the video frames aligned with the recognized objects. The results can be displayed in a diagnostic window when running on a workstation, or in a texture frame when running on device. You can see an example use of `AnnotationOverlayCalculator` in:

- `face_detection_mobile_gpu.pbtxt`.

2.7.3 How to run calculators in parallel

Within a calculator graph, MediaPipe routinely runs separate calculator nodes in parallel. MediaPipe maintains a pool of threads, and runs each calculator as soon as a thread is available and all of it's inputs are ready. Each calculator instance is only run for one set of inputs at a time, so most calculators need only to be *thread-compatible* and not *thread-safe*.

In order to enable one calculator to process multiple inputs in parallel, there are two possible approaches:

1. Define multiple calulator nodes and dispatch input packets to all nodes.
2. Make the calculator thread-safe and configure its `max_in_flight` setting.

The first approach can be followed using the calculators designed to distribute packets across other calculators, such as `RoundRobinDemuxCalculator`. A single `RoundRobinDemuxCalculator` can distribute successive packets across several identically configured `ScaleImageCalculator` nodes.

The second approach allows up to `max_in_flight` invocations of the `CalculatorBase::Process` method on the same calculator node. The output packets from `CalculatorBase::Process` are automatically ordered by timestamp before they are passed along to downstream calculators.

With either aproach, you must be aware that the calculator running in parallel cannot maintain internal state in the same way as a normal sequential calculator.

2.7.4 Output timestamps when using ImmediateInputStreamHandler

The `ImmediateInputStreamHandler` delivers each packet as soon as it arrives at an input stream. As a result, it can deliver a packet with a higher timestamp from one input stream before delivering a packet with a lower timestamp from a different input stream. If these input timestamps are both used for packets sent to one output stream, that output stream will complain that the timestamps are not monotonically increasing. In order to remedy this, the calculator must take care to output a packet only after processing is complete for its timestamp. This could be accomplished by waiting until input packets have been received from all inputstreams for that timestamp, or by ignoring a packet that arrives with a timestamp that has already been processed.

2.7.5 How to change settings at runtime

There are two main approaches to changing the settings of a calculator graph while the application is running:

1. Restart the calculator graph with modified `CalculatorGraphConfig`.
2. Send new calculator options through packets on graph input-streams.

The first approach has the advantage of leveraging `CalculatorGraphConfig` processing tools such as "subgraphs". The second approach has the advantage of allowing active calculators and packets to remain in-flight while settings change. Mediapipe contributors are currently investigating alternative approaches to achieve both of these advantages.

2.7.6 How to process realtime input streams

The mediapipe framework can be used to process data streams either online or offline. For offline processing, packets are pushed into the graph as soon as calculators are ready to process those packets. For online processing, one packet for each frame is pushed into the graph as that frame is recorded.

The MediaPipe framework requires only that successive packets be assigned monotonically increasing timestamps. By convention, realtime calculators and graphs use the recording time or the presentation time as the timestamp for each packet, with each timestamp representing microseconds since `Jan/1/1970:00:00:00`. This allows packets from various sources to be processed in a gloablly consistent order.

Normally for offline processing, every input packet is processed and processing continues as long as necessary. For online processing, it is often necessary to drop input packets in order to keep pace with the arrival of input data frames. When inputs arrive too frequently, the recommended technique for dropping packets is to use the MediaPipe calculators designed specifically for this purpose such as `FlowLimiterCalculator` and `PacketClonerCalculator`.

For online processing, it is also necessary to promptly determine when processing can proceed. MediaPipe supports this by propagating timestamp bounds between calculators. Timestamp bounds indicate timestamp intervals that will contain no input packets, and they allow calculators to begin processing for those timestamps immediately. Calculators designed for realtime processing should carefully calculate timestamp bounds in order to begin processing as promptly as possible. For example, the `MakePairCalculator` uses the `SetOffset` API to propagate timestamp bounds from input streams to output streams.

2.7.7 Can I run MediaPipe on MS Windows?

Currently MediaPipe portability supports Debian Linux, Ubuntu Linux, MacOS, Android, and iOS. The core of MediaPipe framework is a C++ library conforming to the C++11 standard, so it is relatively easy to port to additional platforms.

2.8 Troubleshooting

- *Native method not found*
- *No registered calculator found*
- *Out Of Memory error*
- *Graph hangs*

- *Calculator is scheduled infrequently*
- *Output timing is uneven*
- *CalculatorGraph lags behind inputs*

2.8.1 Native method not found

The error message:

```
java.lang.UnsatisfiedLinkError: No implementation found for void com.google.wick.Wick.
↪nativeWick
```

usually indicates that a needed native library, such as `/libwickjni.so` has not been loaded or has not been included in the dependencies of the app or cannot be found for some reason. Note that Java requires every native library to be explicitly loaded using the function `System.loadLibrary`.

2.8.2 No registered calculator found

The error message:

```
No registered object with name: OurNewCalculator; Unable to find Calculator
↪"OurNewCalculator"
```

usually indicates that `OurNewCalculator` is referenced by name in a `CalculatorGraphConfig` but that the library target for OurNewCalculator has not been linked to the application binary. When a new calculator is added to a calculator graph, that calculator must also be added as a build dependency of the applications using the calculator graph.

This error is caught at runtime because calculator graphs reference their calculators by name through the field `CalculatorGraphConfig::Node:calculator`. When the library for a calculator is linked into an application binary, the calculator is automatically registered by name through the `REGISTER_CALCULATOR` macro using the `registration.h` library. Note that `REGISTER_CALCULATOR` can register a calculator with a namespace prefix, identical to its C++ namespace. In this case, the calcultor graph must also use the same namespace prefix.

2.8.3 Out Of Memory error

Exhausting memory can be a symptom of too many packets accumulating inside a running MediaPipe graph. This can occur for a number of reasons, such as:

1. Some calculators in the graph simply can't keep pace with the arrival of packets from a realtime input stream such as a video camera.
2. Some calculators are waiting for packets that will never arrive.

For problem (1), it may be necessary to drop some old packets in older to process the more recent packets. For some hints, see: *How to process realtime input streams*.

For problem (2), it could be that one input stream is lacking packets for some reason. A device or a calculator may be misconfigured or may produce packets only sporadically. This can cause downstream calculators to wait for many packets that will never arrive, which in turn causes packets to accumulate on some of their input streams. MediaPipe addresses this sort of problem using "timestamp bounds". For some hints see: *How to process realtime input streams*.

The MediaPipe setting `CalculatorGraphConfig::max_queue_size` limits the number of packets enqueued on any input stream by throttling inputs to the graph. For realtime input streams, the number of packets queued at an input stream should almost always be zero or one. If this is not the case, you may see the following warning message:

```
Resolved a deadlock by increasing max_queue_size of input stream
```

Also, the setting `CalculatorGraphConfig::report_deadlock` can be set to cause graph run to fail and surface the deadlock as an error, such that max_queue_size to acts as a memory usage limit.

2.8.4 Graph hangs

Many applications will call `CalculatorGraph::CloseAllPacketSources` and `CalculatorGraph::WaitUntilDone` to finish or suspend execution of a MediaPipe graph. The objective here is to allow any pending calculators or packets to complete processing, and then to shutdown the graph. If all goes well, every stream in the graph will reach `Timestamp::Done`, and every calculator will reach `CalculatorBase::Close`, and then `CalculatorGraph::WaitUntilDone` will complete successfully.

If some calculators or streams cannot reach state `Timestamp::Done` or `CalculatorBase::Close`, then the method `CalculatorGraph::Cancel` can be called to terminate the graph run without waiting for all pending calculators and packets to complete.

2.8.5 Output timing is uneven

Some realtime MediaPipe graphs produce a series of video frames for viewing as a video effect or as a video diagnostic. Sometimes, a MediaPipe graph will produce these frames in clusters, for example when several output frames are extrapolated from the same cluster of input frames. If the outputs are presented as they are produced, some output frames are immediately replaced by later frames in the same cluster, which makes the results hard to see and evaluate visually. In cases like this, the output visualization can be improved by presenting the frames at even intervals in real time.

MediaPipe addresses this use case by mapping timestamps to points in real time. Each timestamp indicates a time in microseconds, and a calculator such as `LiveClockSyncCalculator` can delay the output of packets to match their timestamps. This sort of calculator adjusts the timing of outputs such that:

1. The time between outputs corresponds to the time between timestamps as closely as possible.
2. Outputs are produced with the smallest delay possible.

2.8.6 CalculatorGraph lags behind inputs

For many realtime MediaPipe graphs, low latency is an objective. MediaPipe supports "pipelined" style parallel processing in order to begin processing of each packet as early as possible. Normally the lowest possible latency is the total time required by each calculator along a "critical path" of successive calculators. The latency of the a MediaPipe graph could be worse than the ideal due to delays introduced to display frames a even intervals as described in Output timing is uneven.

If some of the calculators in the graph cannot keep pace with the realtime input streams, then latency will continue to increase, and it becomes necessary to drop some input packets. The recommended technique is to use the MediaPipe calculators designed specifically for this purpose such as `FlowLimiterCalculator` as described in *How to process realtime input streams*.

2.9 Getting help

- *Technical questions*
- *Bugs and Feature requests*

Below are the various ways to get help

2.9.1 Technical questions

For help with technical or algorithmic questions, visit Stack Overflow to find answers and support from the MediaPipe community.

2.9.2 Bugs and Feature requests

To report bugs or make feature requests, file an issue on GitHub. Please choose the appropriate repository for the project from the MediaPipe repo

2.10 Framework Concepts

- *CalculatorBase*
- *Life of a Calculator*
- *Identifying inputs and outputs*
- *Processing*
- *GraphConfig*
- *Subgraph*

Each calculator is a node of a graph. We describe how to create a new calculator, how to initialize a calculator, how to perform its calculations, input and output streams, timestamps, and options. Each node in the graph is implemented as a `Calculator`. The bulk of graph execution happens inside its calculators. A calculator may receive zero or more input streams and/or side packets and produces zero or more output streams and/or side packets.

2.10.1 CalculatorBase

A calculator is created by defining a new sub-class of the `CalculatorBase` class, implementing a number of methods, and registering the new sub-class with Mediapipe. At a minimum, a new calculator must implement the below four methods

- `GetContract()`
  - Calculator authors can specify the expected types of inputs and outputs of a calculator in GetContract(). When a graph is initialized, the framework calls a static method to verify if the packet types of the connected inputs and outputs match the information in this specification.
- `Open()`
  - After a graph starts, the framework calls `Open()`. The input side packets are available to the calculator at this point. `Open()` interprets the node configuration (see Section \ref{graph_config}) operations and prepares the calculator's per-graph-run state. This function may also write packets to calculator outputs. An error during `Open()` can terminate the graph run.
- `Process()`
  - For a calculator with inputs, the framework calls `Process()` repeatedly whenever at least one input stream has a packet available. The framework by default guarantees that all inputs have the same timestamp (see Section \ref{scheduling} for more information). Multiple `Process()` calls can be invoked

simultaneously when parallel execution is enabled. If an error occurs during `Process()`, the framework calls `Close()` and the graph run terminates.

- `Close()`
  - After all calls to `Process()` finish or when all input streams close, the framework calls `Close()`. This function is always called if `Open()` was called and succeeded and even if the graph run terminated because of an error. No inputs are available via any input streams during `Close()`, but it still has access to input side packets and therefore may write outputs. After `Close()` returns, the calculator should be considered a dead node. The calculator object is destroyed as soon as the graph finishes running.

The following are code snippets from CalculatorBase.h.

```
class CalculatorBase {
 public:
  ...

  // The subclasses of CalculatorBase must implement GetContract.
  // ...
  static ::MediaPipe::Status GetContract(CalculatorContract* cc);

  // Open is called before any Process() calls, on a freshly constructed
  // calculator.  Subclasses may override this method to perform necessary
  // setup, and possibly output Packets and/or set output streams' headers.
  // ...
  virtual ::MediaPipe::Status Open(CalculatorContext* cc) {
    return ::MediaPipe::OkStatus();
  }

  // Processes the incoming inputs. May call the methods on cc to access
  // inputs and produce outputs.
  // ...
  virtual ::MediaPipe::Status Process(CalculatorContext* cc) = 0;

  // Is called if Open() was called and succeeded.  Is called either
  // immediately after processing is complete or after a graph run has ended
  // (if an error occurred in the graph).  ...
  virtual ::MediaPipe::Status Close(CalculatorContext* cc) {
    return ::MediaPipe::OkStatus();
  }

  ...
};
```

2.10.2 Life of a calculator

During initialization of a MediaPipe graph, the framework calls a `GetContract()` static method to determine what kinds of packets are expected.

The framework constructs and destroys the entire calculator for each graph run (e.g. once per video or once per image). Expensive or large objects that remain constant across graph runs should be supplied as input side packets so the calculations are not repeated on subsequent runs.

After initialization, for each run of the graph, the following sequence occurs:

- `Open()`
- `Process()` (repeatedly)

- Close()

The framework calls Open() to initialize the calculator. Open() should interpret any options and set up the calculator's per-graph-run state. Open() may obtain input side packets and write packets to calculator outputs. If appropriate, it should call SetOffset() to reduce potential packet buffering of input streams.

If an error occurs during Open() or Process() (as indicated by one of them returning a non-Ok status), the graph run is terminated with no further calls to the calculator's methods, and the calculator is destroyed.

For a calculator with inputs, the framework calls Process() whenever at least one input has a packet available. The framework guarantees that inputs all have the same timestamp, that timestamps increase with each call to Process() and that all packets are delivered. As a consequence, some inputs may not have any packets when Process() is called. An input whose packet is missing appears to produce an empty packet (with no timestamp).

The framework calls Close() after all calls to Process(). All inputs will have been exhausted, but Close() has access to input side packets and may write outputs. After Close returns, the calculator is destroyed.

Calculators with no inputs are referred to as sources. A source calculator continues to have Process() called as long as it returns an Ok status. A source calculator indicates that it is exhausted by returning a stop status (i.e. MediaPipe::tool::StatusStop).

2.10.3 Identifying inputs and outputs

The public interface to a calculator consists of a set of input streams and output streams. In a CalculatorGraphConfiguration, the outputs from some calculators are connected to the inputs of other calculators using named streams. Stream names are normally lowercase, while input and output tags are normally UPPERCASE. In the example below, the output with tag name VIDEO is connected to the input with tag name VIDEO_IN using the stream named video_stream.

```
# Graph describing calculator SomeAudioVideoCalculator
node {
  calculator: "SomeAudioVideoCalculator"
  input_stream: "INPUT:combined_input"
  output_stream: "VIDEO:video_stream"
}
node {
  calculator: "SomeVideoCalculator"
  input_stream: "VIDEO_IN:video_stream"
  output_stream: "VIDEO_OUT:processed_video"
}
```

Input and output streams can be identified by index number, by tag name, or by a combination of tag name and index number. You can see some examples of input and output identifiers in the example below. SomeAudioVideoCalculator identifies its video output by tag and its audio outputs by the combination of tag and index. The input with tag VIDEO is connected to the stream named video_stream. The outputs with tag AUDIO and indices 0 and 1 are connected to the streams named audio_left and audio_right. SomeAudioCalculator identifies its audio inputs by index only (no tag needed).

```
# Graph describing calculator SomeAudioVideoCalculator
node {
  calculator: "SomeAudioVideoCalculator"
  input_stream: "combined_input"
  output_stream: "VIDEO:video_stream"
  output_stream: "AUDIO:0:audio_left"
  output_stream: "AUDIO:1:audio_right"
}
```

(continues on next page)

(continued from previous page)

```
node {
  calculator: "SomeAudioCalculator"
  input_stream: "audio_left"
  input_stream: "audio_right"
  output_stream: "audio_energy"
}
```

In the calculator implementation, inputs and outputs are also identified by tag name and index number. In the function below input are output are identified:

- By index number: The combined input stream is identified simply by index `0`.
- By tag name: The video output stream is identified by tag name "VIDEO".
- By tag name and index number: The output audio streams are identified by the combination of the tag name `AUDIO` and the index numbers `0` and `1`.

```
// c++ Code snippet describing the SomeAudioVideoCalculator GetContract() method
class SomeAudioVideoCalculator : public CalculatorBase {
 public:
  static ::mediapipe::Status GetContract(CalculatorContract* cc) {
    cc->Inputs().Index(0).SetAny();
    // SetAny() is used to specify that whatever the type of the
    // stream is, it's acceptable.  This does not mean that any
    // packet is acceptable.  Packets in the stream still have a
    // particular type.  SetAny() has the same effect as explicitly
    // setting the type to be the stream's type.
    cc->Outputs().Tag("VIDEO").Set<ImageFrame>();
    cc->Outputs().Get("AUDIO", 0).Set<Matrix>;
    cc->Outputs().Get("AUDIO", 1).Set<Matrix>;
    return ::mediapipe::OkStatus();
  }
```

2.10.4 Processing

`Process()` called on a non-source node must return `::mediapipe::OkStatus()` to indicate that all went well, or any other status code to signal an error

If a non-source calculator returns `tool::StatusStop()`, then this signals the graph is being cancelled early. In this case, all source calculators and graph input streams will be closed (and remaining Packets will propagate through the graph).

A source node in a graph will continue to have `Process()` called on it as long as it returns `::mediapipe::OkStatus()`. To indicate that there is no more data to be generated return `tool::StatusStop()`. Any other status indicates an error has occurred.

`Close()` returns `::mediapipe::OkStatus()` to indicate success. Any other status indicates a failure.

Here is the basic `Process()` function. It uses the `Input()` method (which can be used only if the calculator has a single input) to request its input data. It then uses `std::unique_ptr` to allocate the memory needed for the output packet, and does the calculations. When done it releases the pointer when adding it to the output stream.

```
::util::Status MyCalculator::Process() {
  const Matrix& input = Input()->Get<Matrix>();
  std::unique_ptr<Matrix> output(new Matrix(input.rows(), input.cols()));
  // do your magic here....
  //    output->row(n) =  ...
```

(continues on next page)

(continued from previous page)

```
  Output()->Add(output.release(), InputTimestamp());
  return ::mediapipe::OkStatus();
}
```

2.10.5 GraphConfig

A `GraphConfig` is a specification that describes the topology and functionality of a MediaPipe graph. In the specification, a node in the graph represents an instance of a particular calculator. All the necessary configurations of the node, such its type, inputs and outputs must be described in the specification. Description of the node can also include several optional fields, such as node-specific options, input policy and executor, discussed in *Framework Architecture*.

`GraphConfig` has several other fields to configure the global graph-level settings, eg, graph executor configs, number of threads, and maximum queue size of input streams. Several graph-level settings are useful for tuning the performance of the graph on different platforms (eg, desktop v.s. mobile). For instance, on mobile, attaching a heavy model-inference calculator to a separate executor can improve the performance of a real-time application since this enables thread locality.

Below is a trivial `GraphConfig` example where we have series of passthrough calculators :

```
# This graph named main_pass_throughcals_nosubgraph.pbtxt contains 4
# passthrough calculators.
input_stream: "in"
node {
    calculator: "PassThroughCalculator"
    input_stream: "in"
    output_stream: "out1"
}
node {
    calculator: "PassThroughCalculator"
    input_stream: "out1"
    output_stream: "out2"
}
node {
    calculator: "PassThroughCalculator"
    input_stream: "out2"
    output_stream: "out3"
}
node {
    calculator: "PassThroughCalculator"
    input_stream: "out3"
    output_stream: "out4"
}
```

2.10.6 Subgraph

To modularize a `CalculatorGraphConfig` into sub-modules and assist with re-use of perception solutions, a MediaPipe graph can be defined as a `Subgraph`. The public interface of a subgraph consists of a set of input and output streams similar to a calculator's public interface. The subgraph can then be included in an `CalculatorGraphConfig` as if it were a calculator. When a MediaPipe graph is loaded from a `CalculatorGraphConfig`, each subgraph node is replaced by the corresponding graph of calculators. As a result, the semantics and performance of the subgraph is identical to the corresponding graph of calculators.

Below is an example of how to create a subgraph named `TwoPassThroughSubgraph`.

1. Defining the subgraph.

```
# This subgraph is defined in two_pass_through_subgraph.pbtxt
# and is registered as "TwoPassThroughSubgraph"

type: "TwoPassThroughSubgraph"
input_stream: "out1"
output_stream: "out3"

node {
    calculator: "PassThroughculator"
    input_stream: "out1"
    output_stream: "out2"
}
node {
    calculator: "PassThroughculator"
    input_stream: "out2"
    output_stream: "out3"
}
```

   The public interface to the subgraph consists of:

   - Graph input streams
   - Graph output streams
   - Graph input side packets
   - Graph output side packets

2. Register the subgraph using BUILD rule `mediapipe_simple_subgraph`. The parameter `register_as` defines the component name for the new subgraph.

```
# Small section of BUILD file for registering the "TwoPassThroughSubgraph"
# subgraph for use by main graph main_pass_throughcals.pbtxt

mediapipe_simple_subgraph(
    name = "twopassthrough_subgraph",
    graph = "twopassthrough_subgraph.pbtxt",
    register_as = "TwoPassThroughSubgraph",
    deps = [
            "//mediapipe/calculators/core:pass_through_calculator",
            "//mediapipe/framework:calculator_graph",
    ],
)
```

3. Use the subgraph in the main graph.

```
# This main graph is defined in main_pass_throughcals.pbtxt
# using subgraph called "TwoPassThroughSubgraph"

input_stream: "in"
node {
    calculator: "PassThroughCalculator"
    input_stream: "in"
    output_stream: "out1"
}
node {
    calculator: "TwoPassThroughSubgraph"
    input_stream: "out1"
```

(continues on next page)

(continued from previous page)

```
    output_stream: "out3"
}
node {
    calculator: "PassThroughCalculator"
    input_stream: "out3"
    output_stream: "out4"
}
```

2.11 Running on GPUs

- *Overview*
- *OpenGL Support*
- *Life of a GPU calculator*
- *GpuBuffer to ImageFrame converters*

2.11.1 Overview

MediaPipe supports calculator nodes for GPU compute and rendering, and allows combining multiple GPU nodes, as well as mixing them with CPU based calculator nodes. There exist several GPU APIs on mobile platforms (eg, OpenGL ES, Metal and Vulkan). MediaPipe does not attempt to offer a single cross-API GPU abstraction. Individual nodes can be written using different APIs, allowing them to take advantage of platform specific features when needed.

GPU support is essential for good performance on mobile platforms, especially for real-time video. MediaPipe enables developers to write GPU compatible calculators that support the use of GPU for:

- On-device real-time processing, not just batch processing
- Video rendering and effects, not just analysis

Below are the design principles for GPU support in MediaPipe

- GPU-based calculators should be able to occur anywhere in the graph, and not necessarily be used for on-screen rendering.
- Transfer of frame data from one GPU-based calculator to another should be fast, and not incur expensive copy operations.
- Transfer of frame data between CPU and GPU should be as efficient as the platform allows.
- Because different platforms may require different techniques for best performance, the API should allow flexibility in the way things are implemented behind the scenes.
- A calculator should be allowed maximum flexibility in using the GPU for all or part of its operation, combining it with the CPU if necessary.

2.11.2 OpenGL support

MediaPipe supports OpenGL ES up to version 3.2 on Android and up to ES 3.0 on iOS. In addition, MediaPipe also supports Metal on iOS.

- MediaPipe allows graphs to run OpenGL in multiple GL contexts. For example, this can be very useful in graphs that combine a slower GPU inference path (eg, at 10 FPS) with a faster GPU rendering path (eg, at 30 FPS): since one GL context corresponds to one sequential command queue, using the same context for both tasks would reduce the rendering frame rate. One challenge MediaPipe's use of multiple contexts solves is the ability to communicate across them. An example scenario is one with an input video that is sent to both the rendering and inferences paths, and rendering needs to have access to the latest output from inference.

- An OpenGL context cannot be accessed by multiple threads at the same time. Furthermore, switching the active GL context on the same thread can be slow on some Android devices. Therefore, our approach is to have one dedicated thread per context. Each thread issues GL commands, building up a serial command queue on its context, which is then executed by the GPU asynchronously.

2.11.3 Life of a GPU calculator

This section presents the basic structure of the Process method of a GPU calculator derived from base class GlSimpleCalculator. The GPU calculator `LuminanceCalculator` is shown as an example. The method `LuminanceCalculator::GlRender` is called from `GlSimpleCalculator::Process`.

```
// Converts RGB images into luminance images, still stored in RGB format.
// See GlSimpleCalculator for inputs, outputs and input side packets.
class LuminanceCalculator : public GlSimpleCalculator {
 public:
  ::mediapipe::Status GlSetup() override;
  ::mediapipe::Status GlRender(const GlTexture& src,
                               const GlTexture& dst) override;
  ::mediapipe::Status GlTeardown() override;

 private:
  GLuint program_ = 0;
  GLint frame_;
};
REGISTER_CALCULATOR(LuminanceCalculator);

::mediapipe::Status LuminanceCalculator::GlRender(const GlTexture& src,
                                                  const GlTexture& dst) {
  static const GLfloat square_vertices[] = {
      -1.0f, -1.0f,  // bottom left
      1.0f,  -1.0f,  // bottom right
      -1.0f, 1.0f,   // top left
      1.0f,  1.0f,   // top right
  };
  static const GLfloat texture_vertices[] = {
      0.0f, 0.0f,  // bottom left
      1.0f, 0.0f,  // bottom right
      0.0f, 1.0f,  // top left
      1.0f, 1.0f,  // top right
  };

  // program
  glUseProgram(program_);
  glUniform1i(frame_, 1);

  // vertex storage
  GLuint vbo[2];
  glGenBuffers(2, vbo);
  GLuint vao;
  glGenVertexArrays(1, &vao);
  glBindVertexArray(vao);
```

(continues on next page)

(continued from previous page)

```
  // vbo 0
  glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
  glBufferData(GL_ARRAY_BUFFER, 4 * 2 * sizeof(GLfloat), square_vertices,
               GL_STATIC_DRAW);
  glEnableVertexAttribArray(ATTRIB_VERTEX);
  glVertexAttribPointer(ATTRIB_VERTEX, 2, GL_FLOAT, 0, 0, nullptr);

  // vbo 1
  glBindBuffer(GL_ARRAY_BUFFER, vbo[1]);
  glBufferData(GL_ARRAY_BUFFER, 4 * 2 * sizeof(GLfloat), texture_vertices,
               GL_STATIC_DRAW);
  glEnableVertexAttribArray(ATTRIB_TEXTURE_POSITION);
  glVertexAttribPointer(ATTRIB_TEXTURE_POSITION, 2, GL_FLOAT, 0, 0, nullptr);

  // draw
  glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

  // cleanup
  glDisableVertexAttribArray(ATTRIB_VERTEX);
  glDisableVertexAttribArray(ATTRIB_TEXTURE_POSITION);
  glBindBuffer(GL_ARRAY_BUFFER, 0);
  glBindVertexArray(0);
  glDeleteVertexArrays(1, &vao);
  glDeleteBuffers(2, vbo);

  return ::mediapipe::OkStatus();
}
```

The design principles mentioned above have resulted in the following design choices for MediaPipe GPU support:

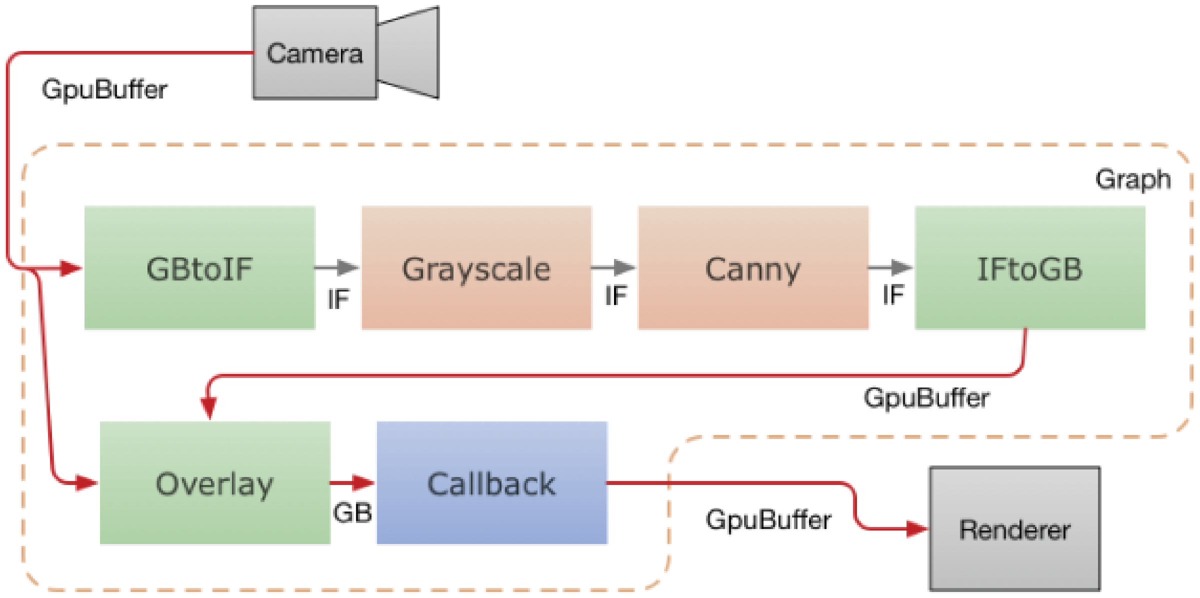
- We have a GPU data type, called `GpuBuffer`, for representing image data, optimized for GPU usage. The exact contents of this data type are opaque and platform-specific.
- A low-level API based on composition, where any calculator that wants to make use of the GPU creates and owns an instance of the `GlCalculatorHelper` class. This class offers a platform-agnostic API for managing the OpenGL context, setting up textures for inputs and outputs, etc.
- A high-level API based on subclassing, where simple calculators implementing image filters subclass from `GlSimpleCalculator` and only need to override a couple of virtual methods with their specific OpenGL code, while the superclass takes care of all the plumbing.
- Data that needs to be shared between all GPU-based calculators is provided as a external input that is implemented as a graph service and is managed by the `GlCalculatorHelper` class.
- The combination of calculator-specific helpers and a shared graph service allows us great flexibility in managing the GPU resource: we can have a separate context per calculator, share a single context, share a lock or other synchronization primitives, etc. – and all of this is managed by the helper and hidden from the individual calculators.

2.11.4 GpuBuffer to ImageFrame converters

We provide two calculators called `GpuBufferToImageFrameCalculator` and `ImageFrameToGpuBufferCalculator`. These calculators convert between `ImageFrame` and `GpuBuffer`, allowing the construction of graphs that combine GPU and CPU calculators. They are supported on both iOS and Android

When possible, these calculators use platform-specific functionality to share data between the CPU and the GPU without copying.

The below diagram shows the data flow in a mobile application that captures video from the camera, runs it through a MediaPipe graph, and renders the output on the screen in real time. The dashed line indicates which parts are inside the MediaPipe graph proper. This application runs a Canny edge-detection filter on the CPU using OpenCV, and overlays it on top of the original video using the GPU.



| How GPU calculators interact | |:–:| | *Video frames from the camera are fed into the graph as `GpuBuffer` packets. The input stream is accessed by two calculators in parallel. `GpuBufferToImageFrameCalculator` converts the buffer into an `ImageFrame`, which is then sent through a grayscale converter and a canny filter (both based on OpenCV and running on the CPU), whose output is then converted into a `GpuBuffer` again. A multi-input GPU calculator, GlOverlayCalculator, takes as input both the original `GpuBuffer` and the one coming out of the edge detector, and overlays them using a shader. The output is then sent back to the application using a callback calculator, and the application renders the image to the screen using OpenGL.* |

2.12 Framework Architecture

2.12.1 Scheduling mechanics

Data processing in a MediaPipe graph occurs inside processing nodes defined as `CalculatorBase` subclasses. The scheduling system decides when each calculator should run.

Each graph has at least one **scheduler queue**. Each scheduler queue has exactly one **executor**. Nodes are statically assigned to a queue (and therefore to an executor). By default there is one queue, whose executor is a thread pool with a number of threads based on the system's capabilities.

Each node has a scheduling state, which can be *not ready*, *ready*, or *running*. A readiness function determines whether a node is ready to run. This function is invoked at graph initialization, whenever a node finishes running, and whenever the state of a node's inputs changes.

The readiness function used depends on the type of node. A node with no stream inputs is known as a **source node**; source nodes are always ready to run, until they tell the framework they have no more data to output, at which point they are closed.

Non-source nodes are ready if they have inputs to process, and if those inputs form a valid input set according to the conditions set by the node's **input policy** (discussed below). Most nodes use the default input policy, but some nodes specify a different one.

Note: Because changing the input policy changes the guarantees the calculator's code can expect from its inputs, it is not generally possible to mix and match calculators with arbitrary input policies. Thus a calculator that uses a special input policy should be written for it, and declare it in its contract.

When a node becomes ready, a task is added to the corresponding scheduler queue, which is a priority queue. The priority function is currently fixed, and takes into account static properties of the nodes and their topological sorting within the graph. For example, nodes closer to the output side of the graph have higher priority, while source nodes have the lowest priority.

Each queue is served by an executor, which is responsible for actually running the task by invoking the calculator's code. Different executors can be provided and configured; this can be used to customize the use of execution resources, e.g. by running certain nodes on lower-priority threads.

2.12.2 Timestamp Synchronization

MediaPipe graph execution is decentralized: there is no global clock, and different nodes can process data from different timestamps at the same time. This allows higher throughput via pipelining.

However, time information is very important for many perception workflows. Nodes that receive multiple input streams generally need to coordinate them in some way. For example, an object detector may output a list of boundary rectangles from a frame, and this information may be fed into a rendering node, which should process it together with the original frame.

Therefore, one of the key responsibilities of the MediaPipe framework is to provide input synchronization for nodes. In terms of framework mechanics, the primary role of a timestamp is to serve as a **synchronization key**.

Furthermore, MediaPipe is designed to support deterministic operations, which is important in many scenarios (testing, simulation, batch processing, etc.), while allowing graph authors to relax determinism where needed to meet real-time constraints.

The two objectives of synchronization and determinism underlie several design choices. Notably, the packets pushed into a given stream must have monotonically increasing timestamps: this is not just a useful assumption for many nodes, but it is also relied upon by the synchronization logic. Each stream has a **timestamp bound**, which is the lowest possible timestamp allowed for a new packet on the stream. When a packet with timestamp `T` arrives, the bound automatically advances to `T+1`, reflecting the monotonic requirement. This allows the framework to know for certain that no more packets with timestamp lower than `T` will arrive.

2.12.3 Input policies

Synchronization is handled locally on each node, using the input policy specified by the node.

The default input policy, defined by `DefaultInputStreamHandler`, provides deterministic synchronization of inputs, with the following guarantees:

- If packets with the same timestamp are provided on multiple input streams, they will always be processed together regardless of their arrival order in real time.
- Input sets are processed in strictly ascending timestamp order.
- No packets are dropped, and the processing is fully deterministic.
- The node becomes ready to process data as soon as possible given the guarantees above.

Note: An important consequence of this is that if the calculator always uses the current input timestamp when outputting packets, the output will inherently obey the monotonically increasing timestamp requirement.

Warning: On the other hand, it is not guaranteed that an input packet will always be available for all streams.

To explain how it works, we need to introduce the definition of a settled timestamp. We say that a timestamp in a stream is *settled* if it lower than the timestamp bound. In other words, a timestamp is settled for a stream once the state of the input at that timestamp is irrevocably known: either there is a packet, or there is the certainty that a packet with that timestamp will not arrive.

Note: For this reason, MediaPipe also allows a stream producer to explicitly advance the timestamp bound farther that what the last packet implies, i.e. to provide a tighter bound. This can allow the downstream nodes to settle their inputs sooner.

A timestamp is settled across multiple streams if it is settled on each of those streams. Furthermore, if a timestamp is settled it implies that all previous timestamps are also settled. Thus settled timestamps can be processed deterministically in ascending order.

Given this definition, a calculator with the default input policy is ready if there is a timestamp which is settled across all input streams and contains a packet on at least one input stream. The input policy provides all available packets for a settled timestamp as a single *input set* to the calculator.

One consequence of this deterministic behavior is that, for nodes with multiple input streams, there can be a theoretically unbounded wait for a timestamp to be settled, and an unbounded number of packets can be buffered in the meantime. (Consider a node with two input streams, one of which keeps sending packets while the other sends nothing and does not advance the bound.)

Therefore, we also provide for custom input policies: for example, splitting the inputs in different synchronization sets defined by `SyncSetInputStreamHandler`, or avoiding synchronization altogether and processing inputs immediately as they arrive defined by `ImmediateInputStreamHandler`.

2.12.4 Flow control

There are two main flow control mechanisms. A backpressure mechanism throttles the execution of upstream nodes when the packets buffered on a stream reach a (configurable) limit defined by `CalculatorGraphConfig::max_queue_size`. This mechanism maintains deterministic behavior, and includes a deadlock avoidance system that relaxes configured limits when needed.

The second system consists of inserting special nodes which can drop packets according to real-time constraints (typically using custom input policies) defined by `FlowLimiterCalculator`. For example, a common pattern places a flow-control node at the input of a subgraph, with a loopback connection from the final output to the flow-control node. The flow-control node is thus able to keep track of how many timestamps are being processed in the downstream graph, and drop packets if this count hits a (configurable) limit; and since packets are dropped upstream, we avoid the wasted work that would result from partially processing a timestamp and then dropping packets between intermediate stages.

This calculator-based approach gives the graph author control of where packets can be dropped, and allows flexibility in adapting and customizing the graph's behavior depending on resource constraints.

2.13 License

CHAPTER 3

Indices and tables

- genindex
- modindex
- search