```
/**
 * testraft9.cpp
 *
 * toy example of sending data in multiple data types
 *
 * overlay
 *
 *
 * Cv::vector<vector<Point> > contours;
 * Cv::vector<Vec4i> hierarchy;
 * Cv::findContours( edgeImg, contours, hierarchy, CV_RETR_TREE , CHAIN_APPROX_SIMPL
E);
 * Cv::drawContours( colorImg, contours, -1, RGB(255,0,0),1.5,8,hierarchy,2,Point())
;
 *
 */

#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <string>
#include <iostream>
#include <cstdlib>
#include <raft>

bool terminate = false;


//template instantiation -

// T must be a class that implements clone
// change Foo into a base class, people have to derive own functions
// or assume assignment operator that is deep copy
template <typename T>
class Foo {
   protected:
   std::shared_ptr<T> mat_;
   public:
   Foo() {
      mat_ = std::make_shared<T>();
   }

   Foo<T>(const Foo<T> &f) {
      mat_ = std::make_shared<T>();
      *mat_ = f.cloneMat();
   }

   T get() const
   {
      return(*mat_);
   }
   void set(T &m)
   {
      *mat_ = m;
   }

   // precondtion: assignment operator of class T is deep copy
   T cloneMat() const
   {
       T bla = (*mat_);
       return(bla);
   }

};

template <> cv::Mat Foo<cv::Mat>::cloneMat() const
{
       return(mat_.get()->clone());
}

template <typename T, size_t n>
class OneToMany : public raft::kernel {
   static_assert(n > 0, "n must be positive");
```

```
public:
   OneToMany() {
      input.addPort<Foo<T> >("input");
      for (size_t i = 0; i < n; i++) {
         output.addPort<Foo<T> >(std::to_string(i));
      }
   }

   virtual raft::kstatus run() {
      Foo<T> src;
      //Foo<T> dst[n - 1];
      raft::signal sig_src(raft::none);

      input["input"].pop(src, &sig_src);

      for (size_t i = 1; i < n; i++) {
         output[std::to_string(i)].push(Foo<T>(src), sig_src);
      }

      output["0"].push(src, sig_src);

      if (raft::eof == sig_src) {
         return raft::stop;
      } else {
         return raft::proceed;
      }
   }
};


template <typename T>
class Source : public raft::kernel {
public:
   Source(cv::VideoCapture device) : _device(device) {
      output.addPort<Foo<T> >("output");
   }

   virtual raft::kstatus run() {
      Foo<T> f;

      T img;
      if (_device.grab() && _device.retrieve(img)) {
         f.set(img);
         output["output"].push(f);
         return raft::proceed;
      }
      return raft::stop;
   }
private:
   cv::VideoCapture _device;
};

template <typename T>
class Blur : public raft::kernel {
public:
   Blur(cv::Size kSize, cv::Point anchor=cv::Point(-1,-1), int borderType= cv::BORDE
R_DEFAULT) :
            kSize_(kSize), anchor_(anchor), borderType_(borderType)
   {
      input.addPort<Foo<T> >("input");
      output.addPort<Foo<T> >("output");
   }

   Blur(std::size_t sizeX = 5, std::size_t sizeY = 5, cv::Point anchor=cv::Point(-1,
-1), int borderType= cv::BORDER_DEFAULT ) :
      kSize_(sizeX, sizeY), anchor_(anchor), borderType_(borderType)
   {
      input.addPort<Foo<T> >("input");
      output.addPort<Foo<T> >("output");
   }
```

```
   virtual raft::kstatus run() {
      Foo<T> src;
      Foo<T> dst;
      raft::signal sig_src(raft::none);

      input["input"].pop(src, &sig_src);
      T img = src.get();
      //cv::blur(img, img, kSize_, anchor_, borderType_);
      T gray, edge;


      cv::cvtColor(img,gray, CV_BGR2GRAY);
      //cv::blur(gray, gray, kSize_);
      cv::Canny(gray, edge, 1, 300, 3);
#if 0
      std::vector<std::vector<cv::Point>> contours;
      std::vector<cv::Vec4i> hier;
      cv::findContours(edge, contours, hier, CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE, c
v::Point(0,0));

      cv::Mat result = img; //cv::Mat::zeros(edge.size(), CV_8UC3);
      for (int i = 0; i < contours.size(); i++) {
         cv::Scalar color(255,0,0);
         drawContours(result, contours, i, color, 2, 8, hier, 0, cv::Point());
      }
#endif

      dst.set(edge);
      output["output"].push(dst, sig_src);

      if (raft::eof == sig_src) {
         return raft::stop;
      } else {
         return raft::proceed;
      }
   }

private:
   cv::Size kSize_;
   cv::Point anchor_;
   int borderType_;
};



class FindContoursOutput {
public:
   std::vector<std::vector<cv::Point> >contours;
   std::vector<cv::Vec4i> hierarchy;
};



class FindContours : public raft::kernel {
public:
   FindContours(int mode = CV_RETR_TREE,
            int method = cv::CHAIN_APPROX_SIMPLE,
            cv::Point offset = cv::Point()):
            mode_(mode), method_(method), offset_(offset) {

      input.addPort<Foo<cv::Mat> >("input");
      output.addPort<Foo<FindContoursOutput> >("output");
   }

   virtual raft::kstatus run() {
      Foo<cv::Mat> src;
      Foo<FindContoursOutput> dst;
      raft::signal sig_src(raft::none);
      input["input"].pop(src, &sig_src);
      cv::Mat img = src.get();
      FindContoursOutput out;
      //cv::imshow("pr",img);
      //cv::waitKey(0);
```

```
      cv::findContours(img, out.contours, out.hierarchy, mode_, method_, offset_);

      dst.set(out);
      output["output"].push(dst, sig_src);

      if (raft::eof == sig_src) {
         return raft::stop;
      } else {
         return raft::proceed;
      }
   }
private:
   int mode_;
   int method_;
   cv::Point offset_;
};

//Cv::drawContours( colorImg, contours, -1, RGB(255,0,0),1.5,8,hierarchy,2,Point());

class DrawContours : public raft::kernel {
public:
   DrawContours(int contourIdx = -1,
            const cv::Scalar color = cv::Scalar(255,0,0), // meant to be red for RGB,
 but if BGR, will be blue
            int thickness = 1,
            int lineType = 8,
            int maxLevel = INT_MAX,
            cv::Point offset = cv::Point()) :
             contourIdx_(contourIdx),
             color_(color),
             thickness_(thickness),
             lineType_(lineType),
             maxLevel_(maxLevel),
             offset_(offset) {
         input.addPort<Foo<cv::Mat> >("inputImg");
         input.addPort<Foo<FindContoursOutput> >("inputContours");
         output.addPort<Foo<cv::Mat> >("output");
   }

   virtual raft::kstatus run() {
      Foo<cv::Mat> dst; // in place cv::Mat
      Foo<FindContoursOutput> src;
      raft::signal sig_src(raft::none);
      raft::signal sig_src2(raft::none);
      input["inputImg"].pop(dst,&sig_src);
      input["inputContours"].pop(src, &sig_src2);
      cv::Mat img = dst.get();
      FindContoursOutput in_ = src.get();

      for (int i = 0; i < in_.contours.size(); i++) {
         cv::drawContours(img, in_.contours, i, color_, thickness_, lineType_, in_.h
ierarchy, maxLevel_, offset_);
      }
      dst.set(img);
      output["output"].push(dst, sig_src);

      if (raft::eof == sig_src || raft::eof == sig_src2) {
         return raft::stop;
      } else {
         return raft::proceed;
      }
   }
private:
   int contourIdx_;
   const cv::Scalar color_;
   int thickness_;
   int lineType_;
   int maxLevel_;
   cv::Point offset_;
};


template <typename T>
```

```
class Display : public raft::kernel {
public:
   Display(cv::string winname) : winname_(winname) {
      input.addPort<Foo<T> >("input");
   }

   virtual raft::kstatus run() {
      Foo<T> src;
      raft::signal sig_src(raft::none);
      input["input"].pop(src, &sig_src);

      T img = src.get();

      cv::imshow(winname_, img);
      cv::waitKey(1);

      if (raft::eof == sig_src) {
         return raft::stop;
      } else {
         return raft::proceed;
      }
   }
private:
   cv::string winname_;
};


int main(int argc, char **argv) {
   Map map;

   cv::VideoCapture cap0(0);

   if (!cap0.isOpened()){
      std::cout << "cannot open video camera 0!\n";     return -1;
   }


   auto link1 = map.link(new Source<cv::Mat>(cap0), new OneToMany<cv::Mat, 2>());
   auto link2 = map.link(&(link1.getDst()), std::to_string(0), new Blur<cv::Mat>());
   auto link3 = map.link(&(link2.getDst()), new FindContours());
   auto link4 = map.link(&(link1.getDst()), std::to_string(1), new DrawContours(), "
inputImg");
   auto link5 = map.link(&(link3.getDst()), &(link4.getDst()), "inputContours");
   auto link6 = map.link(&(link5.getDst()), new Display<cv::Mat>("result"));
   //
// map.link(&(link2.getDst()), new Display<cv::Mat>("end"));
// auto link4 = map.link(&(link1.getDst()), std::to_string(1), new Display<cv::Mat>(
"original"));

   map.exe();

   return EXIT_SUCCESS;
}
```