Robot vision

Autonomous robots, TME290

Ola Benderius
ola.benderius@chalmers.se

Applied artificial intelligence
Vehicle engineering and autonomous systems
Mechanics and maritime sciences
Chalmers

Course orientation

Learning outcomes, mapping of this lecture

- **Describe properties of common types of robotic hardware, including sensors, actuators, and computational nodes**
- Apply modern software development and deployment strategies connected with autonomous robots
- Set up and use equations of motion of wheeled autonomous robots
- Apply basic sensor fusion
- Set up and use computer simulations of autonomous robots
- **Apply global and local navigation of autonomous robots**
- Apply the basics of behaviour-based robotics and evolutionary robotics
- Apply methods for decision making in autonomous robots
- Discuss the potential role of autonomous robots in society, including social, ethical, and legal aspects
- Discuss technical challenges with autonomous robots in society

Introduction

Image processing is one of the main challenges when it comes to autonomous systems, but still one of the most important sources of data.

Image processing is one of the main challenges when it comes to autonomous systems, but still one of the most important sources of data.

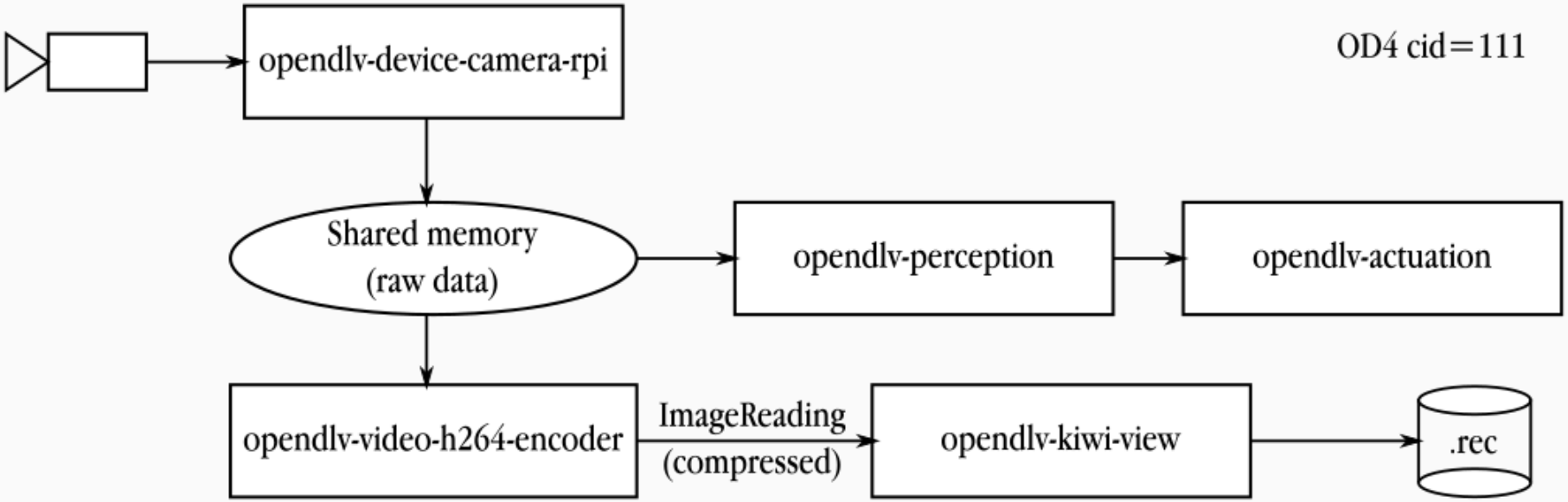
- Great data volumes (can be up to a GB/s)
- Large demands on distributed systems
- Image processing requires much computation

- An image frame does normally not fit in a single network package
- In UDP-based systems it is preferred to avoid data fragmentation (split data)
- This adds two constraints:
  - Data needs to be stored in memory for algorithms
  - Data needs to be compressed when sending over the network

- In a microservice architecture we cannot use local (program) memory
- Shared (between-programs) memory can be used, which is supported by the operating systems
- The microservice that writes to shared memory is called the *producer*
- The microservices that read that memory are called the *consumers*

Video processing in OpenDLV

OpenDLV video streams in live systems



OpenDLV video streams for log replay

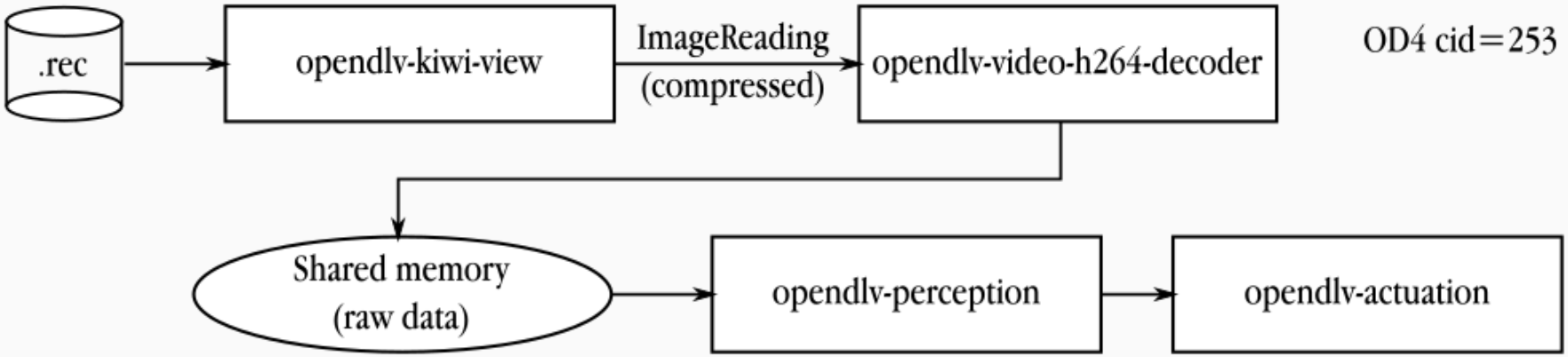


Image processing

- The image processing is in general much slower than the camera frame rate
- Since robot actuation is a direct response to perception, the processing should be as fast as possible

OpenDLV example

An example for how to do image processing in OpenDLV with opencv can be found here:
`https://github.com/chalmers-revere/opendlv-tutorial-kiwi/tree/master/opendlv-perception-helloworld-cpp`

Reading the live frame from the shared memory:

```
while (od4.isRunning()) {
  cv::Mat img;

  // Wait for a notification of a new frame.
  sharedMemory->wait();

  // Lock the shared memory.
  sharedMemory->lock();
  {
    // Copy image into cvMat structure.
    // Be aware of that any code between lock/unlock is blocking
    // the camera to provide the next frame. Thus, any
    // computationally heavy algorithms should be placed outside
    // lock/unlock.
    cv::Mat wrapped(HEIGHT, WIDTH, CV_8UC4, sharedMemory->data());
    img = wrapped.clone();
  }
  sharedMemory->unlock();

  // Do something with the frame.
  // Example: Draw a red rectangle and display image.
  cv::rectangle(img, cv::Point(50, 50), cv::Point(100, 100), cv::Scalar(0,0,255));

  // Display image.
  if (VERBOSE) {
    cv::imshow(sharedMemory->name().c_str(), img);
    cv::waitKey(1);
  }
}
```

A closer look at how to properly use the shared memory in a *consumer*:

```
1   sharedMemory->wait();
2   sharedMemory->lock();
3   {
4     cv::Mat wrapped(HEIGHT, WIDTH, CV_8UC4, sharedMemory->data());
5     img = wrapped.clone();
6   }
7   sharedMemory->unlock();
8
9   // Use img for image processing
```

A closer look at how to properly use the shared memory in a *consumer*:

```
1  sharedMemory->wait();
2  sharedMemory->lock();
3  {
4    cv::Mat wrapped(HEIGHT, WIDTH, CV_8UC4, sharedMemory->data());
5    img = wrapped.clone();
6  }
7  sharedMemory->unlock();
8
9  // Use img for image processing
```

1. Wait for new data to be written (by the producer)
2. Lock the memory to prevent the producer to write again
3. Copy the data from shared into local memory
4. Unlock the memory to allow the producer to write again
5. Continue with image processing

A closer look at how to properly use the shared memory in a *consumer*:

```
1  sharedMemory->wait();
2  sharedMemory->lock();
3  {
4    cv::Mat wrapped(HEIGHT, WIDTH, CV_8UC4, sharedMemory->data());
5    img = wrapped.clone();
6  }
7  sharedMemory->unlock();
8
9  // Use img for image processing
```

1. Wait for new data to be written (by the producer)
2. Lock the memory to prevent the producer to write again
3. Copy the data from shared into local memory
4. Unlock the memory to allow the producer to write again
5. Continue with image processing

It is important to lock the memory for as short time as possible since this affects both the producer and indirectly other consumers. Also, use a local scope when defining `wrapped` to avoid the risk of using that variable (pointer to shared memory) later.

Basic image processing and showing the results:

```
1   // Example: Draw a red rectangle and display image.
2   cv::rectangle(img, cv::Point(50, 50), cv::Point(100, 100), cv::Scalar(0,0,255));
3
4   // Display image.
5   if (VERBOSE) {
6     cv::imshow(sharedMemory->name().c_str(), img);
7     cv::waitKey(1);
8   }
```

Basic image processing and showing the results:

```
1  // Example: Draw a red rectangle and display image.
2  cv::rectangle(img, cv::Point(50, 50), cv::Point(100, 100), cv::Scalar(0,0,255));
3
4  // Display image.
5  if (VERBOSE) {
6    cv::imshow(sharedMemory->name().c_str(), img);
7    cv::waitKey(1);
8  }
```

- OpenCV images can be used not only for perception, but also visualisation
- Remember that `imshow` cannot be done on the robot since it does not support graphics, the program will crash
- If running inside a Docker container, remember to map the `DISPLAY` environment variable, the `tmp` folder, and run `xhost +` (once is enough) to allow Docker to use graphics

Basic image operations

Basic image processing using OpenCV

Colour filtering in HSV space:

```
1 cv::Mat hsv;
2 cv::cvtColor(img, hsv, cv::COLOR_BGR2HSV);
3 cv::Scalar hsvLow(110, 50, 50); // Note: H [0,180], S [0,255], V [0, 255]
4 cv::Scalar hsvHi(130, 255, 255);
5 cv::Mat blueCones;
6 cv::inRange(hsv, hsvLow, hsvHi, blueCones);
7 cv::imshow("Blue␣cones", blueCones);
```

Basic image processing using OpenCV

Colour filtering in HSV space:

```
1 cv::Mat hsv;
2 cv::cvtColor(img, hsv, cv::COLOR_BGR2HSV);
3 cv::Scalar hsvLow(110, 50, 50); // Note: H [0,180], S [0,255], V [0, 255]
4 cv::Scalar hsvHi(130, 255, 255);
5 cv::Mat blueCones;
6 cv::inRange(hsv, hsvLow, hsvHi, blueCones);
7 cv::imshow("Blue␣cones", blueCones);
```

- The HSV colour space is much easier to use compared to the RGB
  - Hue, saturation, and value
  - Separates image intensity (luma) and colour information (chroma)
  - Colour independent of brightness
- The above finds blue colours (NOTE: not optimised)
- NOTE: OpenCV at one point decided to use the range 0–180 for H (rather than the normal 0–360)

Basic image processing using OpenCV

Dilate and erode:

```
1 cv::Mat dilate;
2 uint32_t iterations = 3;
3 cv::dilate(blueCones, dilate, cv::Mat(), cv::Point(-1, -1), iterations, 1, 1);
4 cv::imshow("Dilate", dilate);
5
6 cv::Mat erode;
7 cv::erode(dilate, erode, cv::Mat(), cv::Point(-1, -1), iterations, 1, 1);
8 cv::imshow("Erode", erode);
```

Basic image processing using OpenCV

Dilate and erode:

```
1 cv::Mat dilate;
2 uint32_t iterations = 3;
3 cv::dilate(blueCones, dilate, cv::Mat(), cv::Point(-1, -1), iterations, 1, 1);
4 cv::imshow("Dilate", dilate);
5
6 cv::Mat erode;
7 cv::erode(dilate, erode, cv::Mat(), cv::Point(-1, -1), iterations, 1, 1);
8 cv::imshow("Erode", erode);
```

First note the typical OpenCV image `source` and `destination` in their function arguments (the first two arguments of each operation)

Basic image processing using OpenCV

Dilate and erode:

```
1 cv::Mat dilate;
2 uint32_t iterations = 3;
3 cv::dilate(blueCones, dilate, cv::Mat(), cv::Point(-1, -1), iterations, 1, 1);
4 cv::imshow("Dilate", dilate);
5
6 cv::Mat erode;
7 cv::erode(dilate, erode, cv::Mat(), cv::Point(-1, -1), iterations, 1, 1);
8 cv::imshow("Erode", erode);
```

First note the typical OpenCV image `source` and `destination` in their function arguments (the first two arguments of each operation)

- Used on binary images (e.g. after colour filtering)
- Dilate: white areas get larger
- Erode: white areas get smaller
- Often used in series to fill in holes in white areas

Basic image processing using OpenCV

Canny edge detection:

```
1 cv::Mat canny;
2 cv::Canny(img, canny, 30, 90, 3);
3 cv::imshow("Canny␣edges", canny);
```

Basic image processing using OpenCV

Canny edge detection:

```
1 cv::Mat canny;
2 cv::Canny(img, canny, 30, 90, 3);
3 cv::imshow("Canny edges", canny);
```

- Used to find edges in an image
- Good when identifying objects or object complexity

Basic image processing using OpenCV

Hough line transform:

```
1  std::vector<cv::Vec2f> lines;
2  cv::HoughLines(canny, lines, 1, CV_PI/180, 150, 0, 0 );
3
4  // Draw the lines
5  cv::Mat hough;
6  cv::cvtColor(canny, hough, cv::COLOR_GRAY2BGR);
7  for (size_t i = 0; i < lines.size(); i++) {
8    float rho = lines[i][0];
9    float theta = lines[i][1];
10   double a = cos(theta);
11   double b = sin(theta);
12   double x0 = a * rho;
13   double y0 = b * rho;
14
15   cv::Point pt1;
16   cv::Point pt2;
17   pt1.x = cvRound(x0 + 1000 * (-b));
18   pt1.y = cvRound(y0 + 1000 * a);
19   pt2.x = cvRound(x0 - 1000 * (-b));
20   pt2.y = cvRound(y0 - 1000 * a);
21   cv::line(hough, pt1, pt2, cv::Scalar(0,0,255), 3, cv::LINE_AA);
22 }
```

Basic image processing using OpenCV

Hough line transform:

```
1  std::vector<cv::Vec2f> lines;
2  cv::HoughLines(canny, lines, 1, CV_PI/180, 150, 0, 0 );
3
4  // Draw the lines
5  cv::Mat hough;
6  cv::cvtColor(canny, hough, cv::COLOR_GRAY2BGR);
7  for (size_t i = 0; i < lines.size(); i++) {
8    float rho = lines[i][0];
9    float theta = lines[i][1];
10   double a = cos(theta);
11   double b = sin(theta);
12   double x0 = a * rho;
13   double y0 = b * rho;
14
15   cv::Point pt1;
16   cv::Point pt2;
17   pt1.x = cvRound(x0 + 1000 * (-b));
18   pt1.y = cvRound(y0 + 1000 * a);
19   pt2.x = cvRound(x0 - 1000 * (-b));
20   pt2.y = cvRound(y0 - 1000 * a);
21   cv::line(hough, pt1, pt2, cv::Scalar(0,0,255), 3, cv::LINE_AA);
22 }
```

- Used to find lines in an image

Basic image processing using OpenCV

On your own: Make sure to test the above examples on a replayed log file from Kiwi, and also take the opportunity to experiment even more!

Basic image processing using OpenCV

On your own: Make sure to test the above examples on a replayed log file from Kiwi, and also take the opportunity to experiment even more!

The OpenCV library contains most of the available image processing operations. Spend some time to find examples online.

Convolution

Convolution

- Originates from filtering techniques from signal processing
- In image processing it is implemented using a *sliding kernel*
- A central part in many image processing algorithms, as well as in *convolutional neural networks*

Convolution

- Originates from filtering techniques from signal processing
- In image processing it is implemented using a *sliding kernel*
- A central part in many image processing algorithms, as well as in *convolutional neural networks*

Convolutional neural networks

The reason why CNNs are more efficient than traditional NNs are that the tuned operators are convolution kernels rather than directly on the image pixels themselves. A kernel can be seen as a single image operation, and it is independent of image resolution.

Convolution

When convoluting matrices, the sliding kernel is flipped in both directions:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 0 \\ 0 & -2 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Convolution

When convoluting matrices, the sliding kernel is flipped in both directions:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 0 \\ 0 & -2 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

However, kernels are typically symmetrical!

Convolution, basic operation

When convoluting, the sliding kernel iterates over the target matrix, where in each position a scalar is created from the sum of all products:

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 \\ 1 & -2 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} -2 & 0 \end{bmatrix}$$

Warning

Do not mistake it for matrix multiplication!

Convolution, example 1

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} =$$

Convolution, example 1

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Convolution, example 1

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Note that the input was a 7x7 image, and the output was 5x5.

Convolution, example 1

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} =$$

Convolution, example 1

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & -2 \\ 0 & 0 & 0 & -2 & 2 \\ 0 & 0 & -2 & 2 & 0 \\ 0 & -2 & 2 & 0 & 0 \\ -2 & 2 & 0 & 0 & 0 \end{bmatrix}$$

Convolution, example 1

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & -2 \\ 0 & 0 & 0 & -2 & 2 \\ 0 & 0 & -2 & 2 & 0 \\ 0 & -2 & 2 & 0 & 0 \\ -2 & 2 & 0 & 0 & 0 \end{bmatrix}$$

By using only a 3x3 kernel we have now defined a generic edge detector. This shows how convolution is a very powerful tool for defining image operations.

Camera data

The image sensor

An image sensor is the central part of any camera.

- Just a handful of manufacturers
- A camera (as we know it) is the interface between the sensor chip and the user
- Typical sensor types: Mono (single-channel), RGB

Note

A single-channel camera should *not* be referred to as a grey scale camera.

Important aspects of cameras

- Exposure time
- Rolling or global shutter
- Sensor chip A/D converters (pixel depth: 8, 10, or 12 bit)
- Programming interface, API
- Imaging algorithms (white balancing etc.)

Raw data from RGB cameras, the Bayer pattern

For some applications it might make sense to directly use raw data from the image sensor.

Raw data from RGB cameras, the Bayer pattern

For some applications it might make sense to directly use raw data from the image sensor.

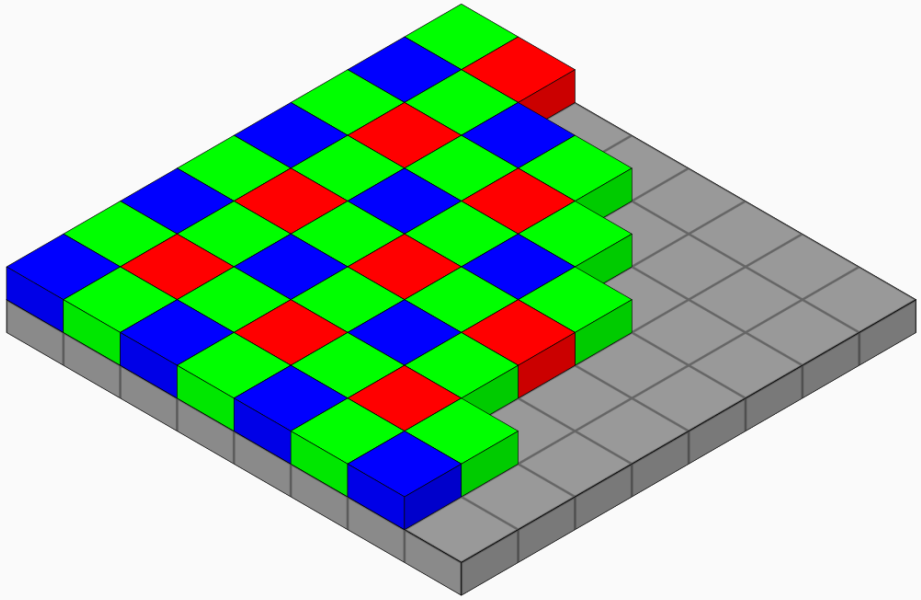


Figure 1: From Wikipedia

Raw data from RGB cameras, the Bayer pattern

For some applications it might make sense to directly use raw data from the image sensor.

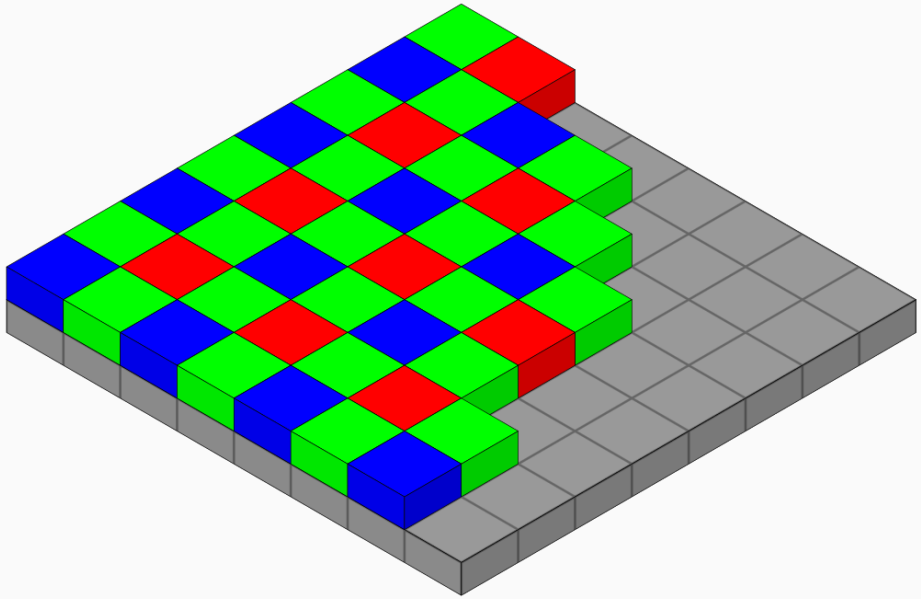


Figure 1: From Wikipedia

Each pixel has, for example, the pixel depth 8 (256 values), 10 (1024 values), or 12 (4096 values) bits.

Getting camera data into GPUs

A GPU natively work with 32 bit floats. In a GPU, images are typically represented by values between 0.0 and 1.0, so transforming from any pixel depth is easy (**pixel value** divided by **integer max value**).

Camera calibration

Some words on camera calibration

We have so far worked with images in the image coordinate system (i, j).

Some words on camera calibration

- *Intrinsic calibration* is when the image coordinates are transformed into camera world coordinates (x, y)

Some words on camera calibration

- *Intrinsic calibration* is when the image coordinates are transformed into camera world coordinates (x, y)
  - Due to image deformations from optics and camera properties (and that the image is mapped into a perfect rectangular image)

Some words on camera calibration

- *Intrinsic calibration* is when the image coordinates are transformed into camera world coordinates (x, y)
  - Due to image deformations from optics and camera properties (and that the image is mapped into a perfect rectangular image)
  - For x and y, the delta angle between each pixel is the same

Some words on camera calibration

- *Intrinsic calibration* is when the image coordinates are transformed into camera world coordinates (x, y)
  - Due to image deformations from optics and camera properties (and that the image is mapped into a perfect rectangular image)
  - For x and y, the delta angle between each pixel is the same
  - The image can no longer fit on a perfect rectangle

Some words on camera calibration

- *Intrinsic calibration* is when the image coordinates are transformed into camera world coordinates (x, y)
  - Due to image deformations from optics and camera properties (and that the image is mapped into a perfect rectangular image)
  - For x and y, the delta angle between each pixel is the same
  - The image can no longer fit on a perfect rectangle
  - Calibration is done by using a checker board, where each distance between squares are equal

Some words on camera calibration

- *Extrinsic calibration* is when (x, y) are transformed into the local frame coordinates (X, Y)

Some words on camera calibration

- *Extrinsic calibration* is when (x, y) are transformed into the local frame coordinates (X, Y)
  - Depends on how the camera is mounted in relation to the origin of the local frame

Some words on camera calibration

- *Extrinsic calibration* is when (x, y) are transformed into the local frame coordinates (X, Y)
  - Depends on how the camera is mounted in relation to the origin of the local frame
  - Can be used when relating detection from various sensors to each other

Some words on camera calibration

- *Extrinsic calibration* is when (x, y) are transformed into the local frame coordinates (X, Y)
  - Depends on how the camera is mounted in relation to the origin of the local frame
  - Can be used when relating detection from various sensors to each other
  - If assuming a flat ground surface, so called *projective transformation* can be used, where each pixel on the image can be transformed to a given X and Y on the ground surface (relative to the origin)

Important

For autonomous vehicles, the origin is most often the centre of the first axle, and X is in forward direction, and Y is in the left direction

Questions

Please post all questions on the Canvas discussion pages, in that way we can all benefit from the answers, and I can highlight important outcomes.