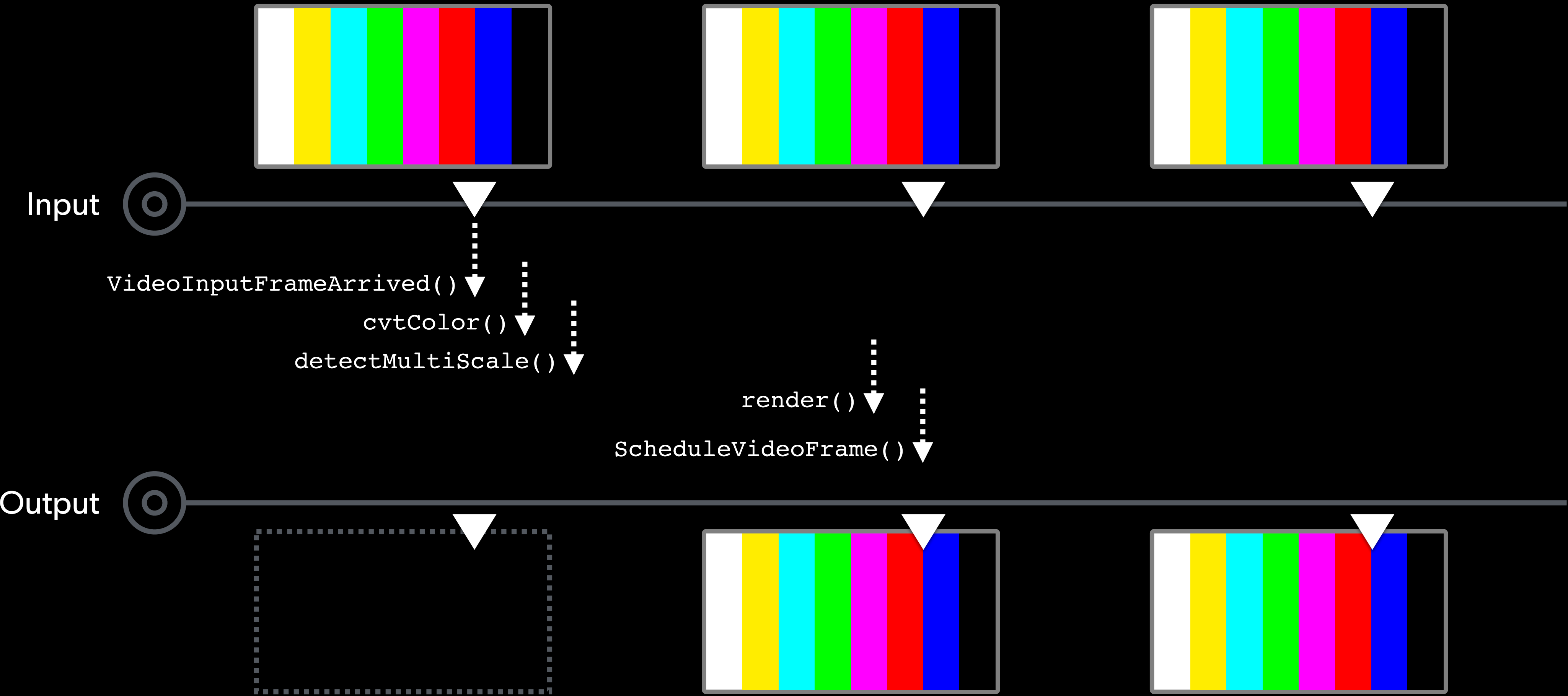


# Live Video Processing with OpenCV

# OpenCV

- Image processing, computer vision
- Cross-platform: desktop, tablet, mobile
- C++, Python, Java, C# ...
- Fast: SSE / AVX / NEON, threads, GPU

# Live Video Processing



# 4K 60fps Face Detection

- Detect face
- Extract
- Compose output frame

```
main.cpp | class Capture
Find ↕ Q:stab
// VideoInputFrameArrived is called asynchronously by the DeckLink API whenever a frame is captured
HRESULT VideoInputFrameArrived(IDeckLinkVideoInputFrame* videoFrame, IDeckLinkAudioInputPacket* audioPacket) override
{
    // Ignore empty video frames
    if (videoFrame == nullptr)
        return S_OK;

    BMDFrameFlags flags = videoFrame->GetFlags();

    // Ignore black frames
    if (flags & bmdFrameHasNoInputSource)
        return S_OK;

    // Detect dropped frames by comparing the stream time of this frame with the previously captured frame.
    // If we're too slow processing a frame (e.g. by writing to a slow disk) DeckLink will eventually drop frames
    // and there'll be a difference greater than one frame between frame arrivals.
    BMDTimeValue capturedFrameNumber;
    BMDTimeValue frameDuration;
    BMDTimeScale timeScale = 24; // FIXME
    CHECK_API(videoFrame->GetStreamTime(&capturedFrameNumber, &frameDuration, timeScale));

    const auto diff = capturedFrameNumber - m_lastArrivedFrameNumber;
    if (diff != 1)
        std::cout << "Warning: dropped " << (diff - 1) << " frames" << std::endl;

    void* data;
    videoFrame->GetBytes(&data);
    const auto width = videoFrame->GetWidth();
    const auto height = videoFrame->GetHeight();
    const auto numBytes = videoFrame->GetRowBytes() * height;

    std::string filename = "/tmp/frame" + std::to_string(m_framesCaptured);
    std::ofstream file(filename, std::ofstream::binary);

    if (!file.is_open())
        throw std::runtime_error("Could not open file: " + filename);

    const bool writeRawFile = false;
    const auto* bytes = reinterpret_cast<const char*>(data);
    if (m_framesCaptured < m_framesToCapture && writeRawFile)
```

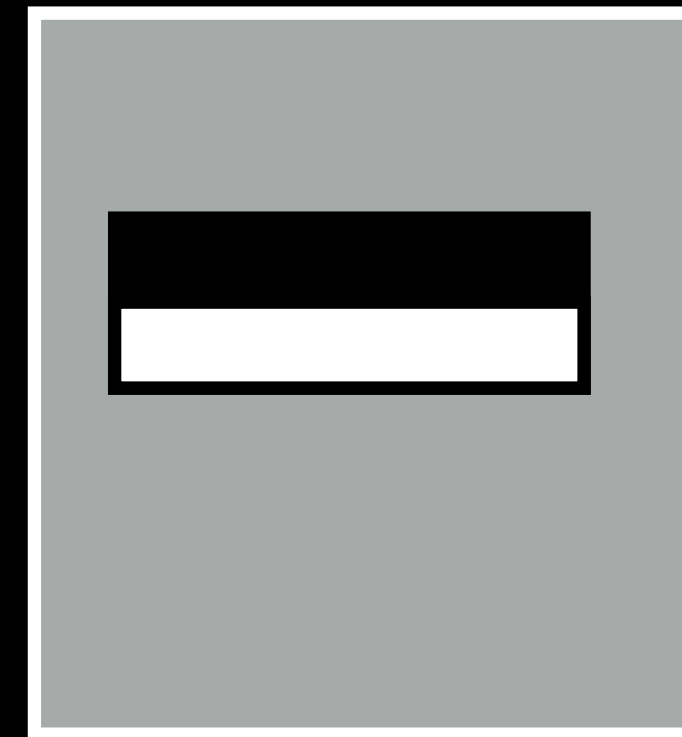




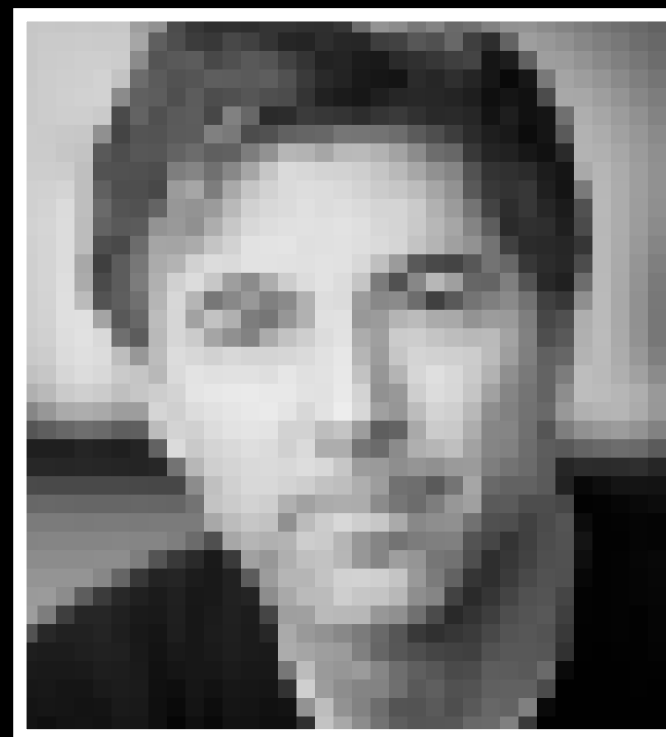
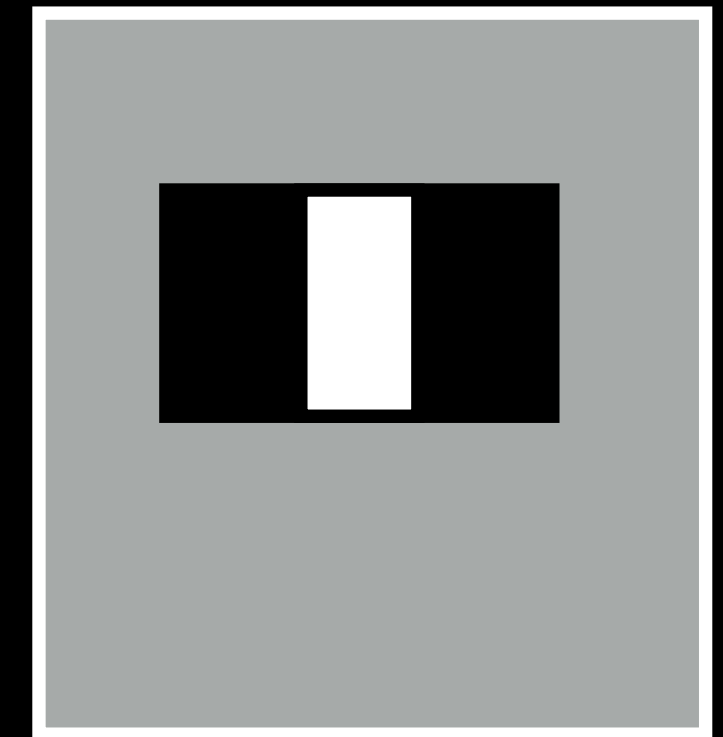
# Face Detection

- Haar Cascade Classifier
- Weak classifiers
- Grouped into stages
  - 1, 10, 25, 50 ...

Edge



Line



# Capture, Process, Playout

Setup DeckLink, Haar Classifier

```
m_deckLinkInput->EnableVideoInput(displayMode, pixelFormat, bmdVideoInputFlagDefault);
m_deckLinkOutput->EnableVideoOutput(displayMode, bmdVideoOutputFlagDefault);

m_deckLinkOutput->StartScheduledPlayback(startTime, timeScale, 1.0);
m_deckLinkInput->SetCallback(this);
m_deckLinkInput->StartStreams();
...
CascadeClassifier m_faceClassifier;
...
m_faceClassifier("data/haarcascades/haarcascade_frontalface_default.xml");
```

# Capture

```
HRESULT VideoInputFrameArrived(IDeckLinkVideoInputFrame* videoFrame, IDeckLinkAudioInputPacket*) override
{
    void*      data;
    videoFrame->GetBytes(&data);
    const auto  width  = videoFrame->GetWidth();
    const auto  height = videoFrame->GetHeight();
    if (m_pixelFormat != bmdFormat8BitYUV)
    {
        ConvertFrame(videoFrame, m_videoFrame8BitYUV);
        m_videoFrame8BitYUV->GetBytes(&data);
    }
    Mat uyvy(height, width, CV_8UC2, data);
    Mat image(height, width, CV_8UC1);
    cvtColor(uyvy, image, cv::COLOR_YUV2GRAY_UYVY);
}
```

# Process

## Detect faces

```
...  
Mat uyvy(height, width, CV_8UC2, data);  
Mat image(height, width, CV_8UC1);  
  
cvtColor(uyvy, image, cv::COLOR_YUV2GRAY_UYVY);  
  
std::vector<cv::Rect> faces;  
m_faceClassifier.detectMultiScale(image, faces, 1.1);
```

- Time: 25000ms

# Process

Resize, detect faces

```
...  
Mat uyvy(height, width, CV_8UC2, data);  
Mat image(height, width, CV_8UC1);  
  
cvtColor(uyvy, image, cv::COLOR_YUV2GRAY_UYVY);  
  
const double scaleFactor = 1.0 / 8.0;  
cv::resize(image, image, cv::Size(), scaleFactor, scaleFactor);  
  
std::vector<cv::Rect> faces;  
m_faceClassifier.detectMultiScale(image, faces, 1.1);
```

- Time: 18ms

# Process

## Custom loop

```
...  
Mat uyvy(height, width, CV_8UC2, data);  
Mat image(height / 8, width / 8, CV_8UC1);  
  
for (int row = 0; row < image.rows; row++)  
    for (int col = 0; col < image.cols; col++)  
        image.at<uint8_t>(row, col) = uyvy.at<uint8_t>(row * 8, col * 8 + 1);  
  
std::vector<cv::Rect> faces;  
m_faceClassifier.detectMultiScale(image, faces, 1.1);
```

- Time: 40ms

# GPU Acceleration

- OpenCV 2
  - upload pixel data to GPU
  - execute kernel
  - read back pixel data
- OpenCV 3
  - Replace *Mat* with *UMat*



# Process

Resize, detect faces

```
...  
Mat uyvy(height, width, CV_8UC2, data);  
UMat image(height, width, CV_8UC1);  
  
cvtColor(uyvy, image, cv::COLOR_YUV2GRAY_UYVY);  
  
const double scaleFactor = 1.0 / 8.0;  
cv::resize(image, image, cv::Size(), scaleFactor, scaleFactor);  
  
std::vector<cv::Rect> faces;  
m_faceClassifier.detectMultiScale(image, faces, 1.1);
```

- Time: 8ms

# Process

## Using UMat

```
Mat imageCPU(height, width, CV_8UC2, data);      // OK
UMat imageUMat(height, width, CV_8UC1);          // OK
UMat imageUMat2(height, width, CV_8UC2, data);    // ERROR!

file.write(imageUMat.data, imageUMat.total() * imageUMat.elemSize()); // ERROR!

Mat imageCPU = imageUMat.getMat(ACCESS_READ);
file.write(imageCPU.data, imageCPU.total() * imageCPU.elemSize());    // OK
```

# Process

## Draw face detections

```
std::vector<cv::Rect> faces;
m_faceClassifier.detectMultiScale(image, faces, 1.1);

void*          outputData;
videoFrame->GetBytes(&outputData);
Mat outputUYVY(height, width, CV_8UC2, data);

for (size_t i = 0; i < faces.size(); i++)
{
    double scale = 1.0 / scaleFactor;
    rectangle(outputUYVY, faces[i].tl() * scale, faces[i].br() * scale, Scalar(128, 255), 8);
}
```

# Process

## Finding the best face

```
std::vector<cv::Rect> faces;
std::vector<int> numDetections;
m_faceClassifier.detectMultiScale(image, faces, numDetections, 1.1);

int bestIndex = std::distance(numDetections.begin(),
                              std::max_element(numDetections.begin(),
                                                  numDetections.end()));

cv::Rect bestFace = faces[bestIndex];
```

# Playout

```
cv::Rect originalFaceRect = cv::Rect(bestFace.tl() * scale, bestFace.br() * scale);

UMat clippedFace = uyvy(originalFaceRect & cv::Rect(0, 0, width, height));

clippedFace.copyTo(outputUYVY(cv::Rect(0, 0, clippedFace.cols, clippedFace.rows)));

m_deckLinkOutput->ScheduleVideoFrame(outputFrame, capturedFrameNumber, frameDuration, m_timeScale);
```

# Live Processing

- Track objects
- Don't iterate over pixels
- Down convert / subsample
- Use UMat

# More Information

- Blackmagic Design Developer Support website
- Blackmagic Design Software Development forum
- Developer Support email - [developer@blackmagicdesign.com](mailto:developer@blackmagicdesign.com)



