VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV RADIOELEKTRONIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF RADIO ELECTRONICS

AUGMENTED REALITY APPLICATIONS IN EMBEDDED NAVIGATION DEVICES

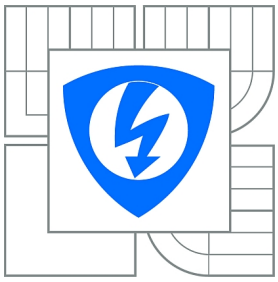
SEMESTRÁLNÍ PRÁCE
SEMESTRAL THESIS

AUTOR PRÁCE MARTIN JAROŠ
AUTHOR

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV RADIOELEKTRONIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF RADIO ELECTRONICS

AUGMENTED REALITY APPLICATIONS IN EMBEDDED NAVIGATION DEVICES

AUGMENTED REALITY APPLICATIONS IN EMBEDDED NAVIGATION DEVICES

SEMESTRÁLNÍ PRÁCE

SEMESTRAL THESIS

AUTOR PRÁCE — MARTIN JAROŠ

AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. TOMÁŠ FRÝZA, Ph.D.

SUPERVISOR

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav radioelektroniky

Semestrální práce

bakalářský studijní obor
Elektronika a sdělovací technika

Student: Martin Jaroš ***ID:*** 146847
Ročník: 3 ***Akademický rok:*** 2013/2014

NÁZEV TÉMATU:

Augmented reality applications in embedded navigation devices

POKYNY PRO VYPRACOVÁNÍ:

BB2E: Analyze the hardware possibilities of the OMAP platform and design an application to effectively combine captured video data and rendered virtual scene based on navigational data from GPS and INS sensors. Design and create a functional prototype.

BBCE: Examine practical use cases of the proposed navigation device, design applicable user interface.

DOPORUČENÁ LITERATURA:

[1] BIMBER, O.; RASKAR, R. Spatial augmented reality: merging real and virtual worlds. Wellesley: A K Peters, 2005, 369 p. ISBN 15-688-1230-2.

[2] Texas Instruments. OMAP 4460 Multimedia Device [online]. 2012 - [cit. 8. listopadu 2012]. Available: http://www.ti.com/product/omap4460.

Termín zadání: 23.9.2013 ***Termín odevzdání:*** 20.12.2013

Vedoucí práce: doc. Ing. Tomáš Frýza, Ph.D.
Konzultanti semestrální práce:

doc. Ing. Tomáš Kratochvíl, Ph.D.
Předseda oborové rady

UPOZORNĚNÍ:

ABSTRAKT

Tato práce se zabývá aplikací rozšířené reality v oblasti navigačních zařízení. Popisuje možnosti zpracování videa za účelem projekce virtuální scény pomocí dat získaných satelitním a inerciálním navigačním sybsystémem. Práce klade důraz na využití moderních hardwarových prostředků pro zpracování videa v mikroprocesorech pomocí grafických akcelerátorů. Součástí je návrh aplikace a realizace prototypu.

KLÍČOVÁ SLOVA

Rozšířená realita, satelitní navigace, inerciální měřící jednotka

ABSTRACT

This work deals with application of augmented reality in navigation devices. It describes possibilities of video processing, rendering a virtual scene by using data measured by satellite and inertial navigation subsystems. Special care is taken into account for use of modern graphic accelerator hardware available in microprocessors. Design of the application is supplemented with prototype realization.

KEYWORDS

Augmented reality, satellite navigation, inertial measurement unit

JAROŠ, M. Augmented reality applications in embedded navigation devices. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2014. 30 s. Vedoucí semestrální práce doc. Ing. Tomáš Frýza, Ph.D..

PROHLÁŠENÍ

Prohlašuji, že svou semestrální práci na téma Augmented reality applications in embedded navigation devices jsem vypracoval samostatně pod vedením vedoucího semestrální práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené semestrální práce dále prohlašuji, že v souvislosti s vytvořením této semestrální práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a~jsem si plně vědom následků porušení ustanovení § 11 a následujících zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

(podpis autora)

PODĚKOVÁNÍ

Děkuji vedoucímu semestrální práce doc. Ing. Tomáši Frýzovi, Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé semestrální práce.

Brno ....

(podpis autora)

Contents

List of Figures

List of Tables

Preface

Augmented reality technologies have been around for a while [1], however their implementation in embedded systems were not possible until recently. Navigation applications require a broad spectrum of functionality such as video processing, accelerated graphical rendering and of course the navigation itself. With public access to the global satellite navigation systems such as GPS or GLONASS, precise position determination is possible worldwide. Advances in sensor technologies offer many small and effective devices, such as gyroscopes, compasses, accelerometers or barometers. The augmented reality navigation provides the future of spatial navigation systems, delivering all necessary information to the user in the most clear way by projecting it directly to the visible world. Users do not have to search for the information in specialized devices or multifunction displays, they will just see it floating around, related to where they look. Applications include automotive, aeronautical or personal navigation.

This work provides foundation on all aspects of designing such system. The first chapter specifies requirements and limitations for the project. The second chapter deals with the application design, it is divided per each individual subsystem. The third chapter specifies hardware details and platform realization.

1 Augmented reality

1.1 Project overview

Main goal of this project is to develop a device capable of rendering a real time overlay over the captured video frame. There are three external inputs, the image sensor capable of video capture feeds real-time images to the system. The GPS receiver delivers positional information and inertial sensors supplement it with spatial orientation. Expected setup is that the image and inertial sensors are on a single rack able to freely rotate around, while the GPS receiver is static, relative to the whole moving platform (vehicle for example). The overlay consists of fixed kinematic data such as speed or altitude, reference indicators such a horizon line and dynamic location markers. These will be spatially aligned with real locations visible in the video thus providing navigation information. They work in a way that wherever the camera is pointed to, specific landmarks will label currently visible locations. Complex external navigation system may be also connected. This allows integration with already existing systems, such as moving map systems, PDAs or other specialized hardware. These provides user with classic route navigation and map projection, while this project gives spatial extension to further improve total situational awareness. The analogy are the head up and head down displays, each delivering specific set of information. This project focuses on visual enhancement instead of a full featured navigation device. Overview on of the project design is in the following figure.

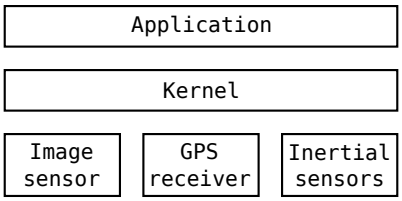


Figure 1: Project overview

1.2 Hardware limitations

Application is designed to run in embedded environments, where power management is very important. While many platforms features multiple symmetrical processor cores - CPUs, application should focus on lowest per-core usage as possible. This can be done by delegating specific tasks to specialized hardware. CPU is specialized in execution of single thread, integer and bitwise operations, with many branches in its code. With vector and floating point extensions they are also very efficient in computation of difficult mathematical algorithms. However they do not perform well in simple calculations over large amounts of data, where mass parallelization is possible. This is the case in graphics where special graphics processors - GPUs have been deployed. GPU consists of high number (hundreds) of simple cores, which are able to perform operations over large blocks of data. They scale efficiently with the amount of data needed to be processed due to parallelization, however they have problems

with nonlinear code with branches. While CPUs have long pipelines for branch optimizations, GPUs cannot employ those, any branch in their code will be extremely inefficient and should be avoided. Graphics (2.3) chapter focuses on this area. There are also available specialized subsystems designed and optimized for a single purpose. For example video accelerators, capable of video encoding and decoding, image capture systems or peripheral drivers. They will be mentioned in specific chapters.

Developing an application for an embedded device faces a problem, as there are big differences between these devices it is hard to support the hardware and make the application portable. In order to reuse code and reduce application size, libraries are generally used to create an intermediary layer between application and the hardware. However, to provide enough abstraction some sort of operating system has to be used. Operating systems may be real-time, giving applications full control, behaving just like large libraries. This is favorable approach in embedded systems as it allows precise timings. There are many such systems specially tailored for embedded applications like FreeRTOS [2] or proprietary VxWorks [3]. On the other hand, as recent processors improved greatly in power, efficiency and capabilities, it is possible and quite feasible to run a full featured system like Linux or proprietary Windows CE [4]. Linux kernel is highly portable and configurable, although it does restrict applications from real-time use (Linux RT patches also exist for real-time applications), as all hardware dependent modules which requires full control over the hardware are part of the kernel itself, application does not need to run in real-time at all. Other advantages are free, open and well documented sources, highly standardized and POSIX compliant application interface, large amount of drivers with good manufacturer support. While its disadvantages are very large code base and steep learning curve, which may slow the initial development. Nevertheless Linux kernel has been chosen for the project, more details about its interfaces are in the Linux kernel (2.1) chapter. While the application is designed to be highly portable depending only on the kernel itself, several devices has been chosen as the reference, they are listed in the hardware (3) chapter.

2 Application

Application is divided into four subsystems, each being standalone component. The video subsystem (2.2) is responsible for enumeration and control of the video architecture and its devices. It provides the application with raw video buffers and means to configure its format. It is designed to support high range of devices from embedded image sensors to external cameras, while using single application interface and common image format. Depending on the hardware, high definition video output is expected. Video subsystem is optimized for synchronous operation with the graphics subsystem (2.3). Graphics subsystem utilizes platform interfaces for its graphic accelerator units to provide optimized video processing and rendering. It is hardware independent through common library support to run on most embedded systems. Its goal is to provide application with efficient methods for rendering primitives, video frames and vector fonts with object oriented interface. These methods combined will create the scene overlay over the source video in real time. Graphic output should be high definition digital, maintaining source quality. Data needed for the overlay creation are provided by satellite (2.5) and inertial (2.4) subsystems. They are both designed for asynchronous operation. The satellite navigation provides application with positional and kinematic data. It is responsible for communication with external navigation systems such as GPS receivers and all needed calculations. Its interfaces allows application to access required information asynchronously as needed by the rendering loop. The inertial measurement subsystem utilizes sensors needed for spatial orientation not provided by satellite navigation. As there are many such sensors, common interface is provided. Subsystem handles all initialization, control, data acquisition and required calculations. Its internal state machine provides application the requested data on demand. Application is designed to be modular and highly configurable. All constants used throughout the implementation are defined with a default value and modifiable through the configuration file. This includes for example video setup, device selection or rendering parameters.

2.1 Linux kernel

Programs running in Linux are divided into two groups, kernel-space and user-space. Only kernel and its runtime modules are allowed to execute in kernel-space, they have physical memory access and use CPU in real-time. All other programs runs as processes in user-space, they have virtual memory access, which means their memory addresses are translated to the physical addresses in the background. In Linux each process runs in a sandbox, isolated from the rest of the system. Processes access memory unique to them, they cannot access memory assigned for other processes nor memory managed by the kernel. They may communicate with the outside environment by several means:

- Arguments and environment variables
- Standard input, output and error output
- Virtual File System
- Signals

- Sockets
- Memory mapping

Each process is ran with several arguments in a specific environment with three default file descriptors. For example running

```
VARIABLE=value ./executable argument1 argument2 <input 1>output 2>error
```

will execute `executable` with environment variable `VARIABLE` of value `value` with two arguments `argument1` and `argument2`. Standard input will be read from file `input` while regular output will be written to file `output` and error output to file `error`. This process may further communicate by accessing files in the Virtual File System, kernel may expose useful process information for example via `procfs` file-system usually mounted at `/proc`. Other types of communication are signals (which may be sent between processes or by kernel) and network sockets. With internal network loop-back device, network style inter process communication is possible using standard protocols (UDP, TCP, ...). Memory mapping is a way to request access to some part of the physical memory.

Process execution is not real-time, but they are assigned restricted processor time by the kernel. They may run in numerous threads, each thread has preemptively scheduled execution. Threads share memory within a process, memory access to these shared resources must done with care to avoid race conditions and data corruption. Kernel provides *mutex* objects to lock threads and avoid simultaneous memory access. Each shared resource should be attached to a *mutex*, which is locked during access to this resource. Thread must not lock *mutex* while still holding lock to this or any other *mutex* in order to avoid dead-locking. Source code on how to use threads is in the threads example appendix.

Linux kernel has monolithic structure, so all device drivers resides in the kernel-space. From application point of view, this means that all peripheral access must be done through the standard library and Virtual File System. Individual devices are accessible as device files defined by major and minor number typically located at `/dev`. These files could be created automatically by kernel (`devtmpfs` file-system), by daemon (`udev(8)`), or manually by `mknod(1)`. Complete kernel device model is exported as `sysfs` file-system and typically mounted at `/sys`.

Function name	**Access type**	**Typical usage**
`select()`, `poll()`	event	Synchronization, multiplexing, event handling
`ioctl()`	structure	Configuration, register access
`read()`, `write()`	stream	Raw data buffers, byte streams
`mmap()`	block	High throughput data transfers

Table 1: Available functions for working with device file descriptors

For example, assume a generic peripheral device connected by the I^2C bus. First, to tell kernel there is such a device, the `sysfs` file-system may be used

```
echo $DEVICE_NAME $DEVICE_ADDRESS > /sys/bus/i2c/devices/i2c-1/new_device
```

This should create a special file in `/dev`, which should be opened by `open()` to get a file descriptor for this device. Device driver may export some *ioctl* requests, each request is defined by a number and a structure passed between the application and the kernel. Driver should define requests for controlling the device, maybe accessing its internal registers and configuring a data stream. Each request is called by

```
ioctl(fd, REQNUM, &data);
```

where `fd` is the file descriptor, `REQNUM` is the request number defined in the driver header and `data` is the structure passed to the kernel. This request will be synchronously processed by the kernel and the result stored in the `data` structure. Assume this devices has been configured to stream an integer value every second to the application. To synchronize with this timing application may use

```
struct pollfd fds = {fd, POLLIN};
poll(&fds, 1, -1);
```

which will block infinitely until there is a value ready to be read. To actually read it,

```
int buffer[1];
ssize_t num = read(fd, buffer, sizeof(buffer));
```

will copy this value to the buffer. Copying causes performance issues if there are very large amounts of data. To access this data directly without copying them, application has to map physical memory used by the driver. This allows for example direct access to a DMA channel, it should be noted that this memory may still be needed by kernel, so there should be some kind of dynamic access restriction, possibly via *ioctl* requests (this would be driver specific).

2.2 Video subsystem

Video support in Linux kernel is maintained by the LinuxTV project, it implements the `videodev2` kernel module and defines the *V4L2* interface. Modules are part of the mainline kernel at `drivers/media/video/*` with header `linux/videodev2.h`. The core module is enabled by the `VIDEO_V4L2` configuration option, specific device drivers should be enabled by their respective options. V4L2 is the latest revision and is the most widespread video interface throughout Linux, drives are available from most hardware manufactures and usually mainlined or available as patches. The Linux Media Infrastructure API [5] is a well documented interface shared by all devices. It provides abstraction layer for various device implementations, separating the platform details from the applications. Each video device has its device file and is controlled via *ioctl* calls. For streaming, standard I/O functions are supported, but the memory mapping is preferred, this allows passing only pointers between the application and the kernel, instead of unnecessary copying the data around. Available *ioctl* calls are:

Name	Description
VIDIOC_QUERYCAP	Query device capabilities
VIDIOC_G_FMT	Get the data format
VIDIOC_S_FMT	Set the data format
VIDIOC_REQBUFS	Initiate memory mapping
VIDIOC_QUERYBUF	Query the status of a buffer
VIDIOC_QBUF	Enqueue buffer to the kernel
VIDEOC_DQBUF	Dequeue buffer from the kernel
VIDIOC_STREAMON	Start streaming
VIDIOC_STREAMOFF	Stop streaming

Table 2: V4L2 ioctl calls defined in `linux/videodev2.h`

Application sets the format first, then requests and maps buffers from the kernel. Buffers are exchanged between the kernel and the application. When the buffer is enqueued, it will be available for the kernel to capture data to it. When the buffer is dequeued, kernel will not access the buffer and application may read the data. After all buffers are enqueued, application starts the stream. Polling is used to wait for the kernel until it fills the buffer, buffer should not be accessed simultaneously by the kernel and the application. After processing the buffer, application should return it back to the kernel queue. Note that buffers should be properly unmapped by the application after stopping the stream. The video capture process is described in the following diagram.

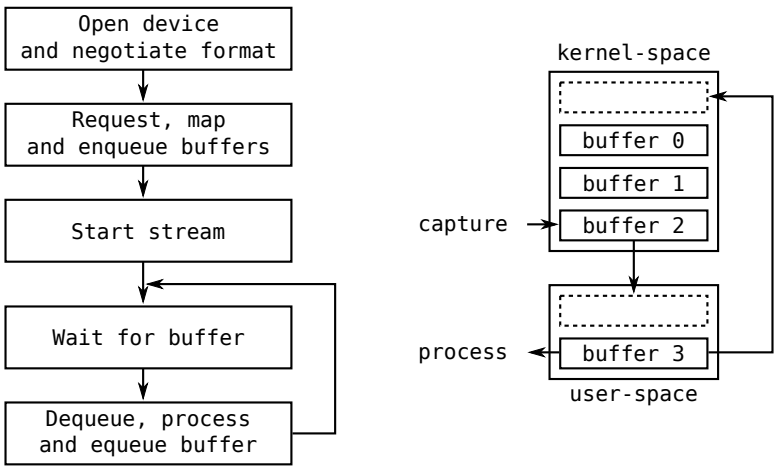


Figure 2: V4L2 capture

Source code for simple video capture is in video capture example appendix. The image format

is specified using the little-endian four-character code (FOURCC). V4L2 defines several formats and provides `v4l2_fourcc()` macro to create a format code from four characters. As described later in the graphics subsystem (2.3) chapter, graphics uses natively the RGB4 format. This format is defined as a single plane with one sample per pixel and four bytes per sample. These bytes represents red, green and blue channel values respectively. Image size is therefore $width \cdot height \cdot 4$ bytes. Many image sensors however support YUV color-space, for example the YU12 format. This one is defined as three planes, the first plane with one luminance sample per pixel and the second and third plane with one chroma sample per four pixels (2 pixels per row, interleaved). Each sample has one byte, this format is also referenced as YUV 4:2:0 and its image size is $width \cdot height \cdot 1.5$ bytes. The luminance and chroma of a pixel is defined as

$$E_Y = W_R \cdot E_R + (1 - W_R - W_B) \cdot E_G + W_B \cdot E_B,$$

$$E_{C_r} = \frac{0.5(E_R - E_Y)}{1 - W_R},$$

$$E_{C_b} = \frac{0.5(E_B - E_Y)}{1 - W_B},$$

where E_R, E_G, E_B are normalized color values and W_R, W_B are their weights. ITU-R Rec. BT.601 [6] defines weights as 0.299 and 0.114 respectively, it also defines how they are quantized

$$Y = 219E_Y + 16,$$

$$C_r = 224E_{C_r} + 128,$$

$$C_b = 224E_{C_b} + 128.$$

To calculate R, G, B values from Y, Cr, Cb values, inverse formulas must be used

$$E_Y = \frac{Y - 16}{219},$$

$$E_{C_r} = \frac{C_r - 128}{224},$$

$$E_{C_b} = \frac{C_b - 128}{224},$$

$$E_R = E_Y + 2E_{C_r}(1 - W_R),$$

$$E_G = E_Y - 2E_{C_r}\frac{W_R - {W_R}^2}{W_G} - 2E_{C_b}\frac{W_B - {W_B}^2}{W_G},$$

$$E_B = E_Y + 2E_{C_b}(1 - W_B).$$

It should be noted that not all devices may use the BT.601 recommendation, V4L2 refers to it as `V4L2_COLORSPACE_SMPTE170M` in the `VIDIOC_S_FMT` request structure. Implementation of the YUV to RGB color-space conversion is most efficient on graphics accelerators, such example is included in colorspace conversion example appendix. It is written in GLSL for fragment processor, see graphics subsystem (2.3) chapter for further description.

There is a kernel module `v4l2loopback` which creates a video loop-back device, similar to network loop-back, allowing piping two video applications together. This is very useful

not only for testing, but also for implementation of intermediate decoders. GStreamer is a powerful multimedia framework widespread in Linux distributions, composed of a core infrastructure and hundreds of plug-ins. This command will create synthetic RGB4 video stream for the application, useful for testing

```
modprobe v4l2loopback
gst-launch videotestsrc pattern=solid-color foreground-color=0xE0F0E0 ! \
"video/x-raw,format=RGBx,width=800,height=600,framerate=20/1" \
!  v4l2sink device=/dev/video0
```

Texas Instruments distributes a meta package [7] for their OMAP platform featuring all required modules and DSP firmware. This includes kernel modules for *SysLink* inter-chip communication library, *Distributed Codec Engine* library and *ducati* plug-in for GStreamer. With the meta-package installed, it is very easy and efficient to implement mainstream encoded video formats. For example following command will create GStreamer pipeline to receive video payload over a network socket from an IP camera, decode it and push it to the loop-back device for the application. MPEG-4 AVC (H.264) decoder of the IVA 3 is used in this example.

```
modprobe v4l2loopback
gst-launch udpsrc port=5004 caps=\
"application/x-rtp,media=video,payload=96,clock-rate=90000,encoding-name=H264" \
!  rtph264depay !  h264parse !  ducatih264dec !  v4l2sink device=/dev/video0
```

On OMAP4460 this would consume only about 15% of the CPU time as the decoding is done by the IVA 3 video accelerator in parallel to the CPU which only passes pointers around and handles synchronization. Output format is NV12 which is similar to YU12 format described earlier, but there is only one chroma plane with two-byte samples, first byte being the U channel and the second byte the V channel, sampling is same 4:2:0. The YUV to RGB color space conversion must take place here, preferably implemented on the GPU as described above.

Cortex-A9 cores on the OMAP4460 also have the NEON co-processor, capable of vector floating point math. Although not very supported by the GCC C compiler, there are many assembly written libraries implementing coders with the NEON acceleration. For example the *libjpeg-turbo* library is implementing the *libjpeg* interface. It is useful for USB cameras, as the USB throughput is not high enough for raw high definition video, but is sufficient with JPEG coding (as most USB cameras supports JPEG, but does not support H.264). 1080p JPEG stream decoded with this library via its GStreamer plug-in will consume about 90% of the single CPU core time (note that there are two CPU cores available). However, comparable to the AVC, JPEG encoding will cause visible quality degradation in the raw stream (video looks grainy).

2.3 Graphics subsystem

Graphical output in the Linux Kernel is accessible as a framebuffer device `/dev/fb`. This allows directly writing to a display memory from the application. OMAP platform further extends

this interface to support its DSS2 architecture for multiplexing graphical and video systems with display outputs. There is a framebuffer device connected to the PowerVR SGX graphical accelerator with control interface at `/sys/class/graphics/fbX`, where `X` is the framebuffer number. The OMAP DSS subsystem is exported at `/sys/devices/platform/omapdss` as

- `/sys/devices/platform/omapdss/overlayX/`
- `/sys/devices/platform/omapdss/managerX/`
- `/sys/devices/platform/omapdss/displayX/`

where overlays may read from FB or V4L2 devices, combined in the manager and then linked to a physical display. Input sources can be blended by color keying to create the output image, this is useful for rendering graphical overlays. Individual displays (HDMI, LCD) are also configured by this interface.

2.3.1 OpenGL ES 2.0 API

There is a standardized library for interfacing with graphical accelerators maintained by Khronos group called OpenGL. Its recent version targeted for embedded systems is OpenGL ES 2.0 [8] [9], implemented by majority of hardware developers. It is also supported by the multi-platform Mesa3D library, so it will run also on desktop computer, either emulated by CPU or partially accelerated depending on available hardware. OpenGL ES 2.0 is implemented in two parts, the kernel module and user-space libraries *EGL* and *GLESv2*. Its implementation will be platform specific, however the application interface is the same. Functions names in the API are prefixed with *gl* and suffixed by argument types, used for overloading. EGL library is used for initialization of the OpenGL context, following function are available in the API:

Function	**Description**
`eglGetDisplay()`	Select display
`eglInitialize()`	Initialize display
`eglBindAPI()`	Select OpenGL API version
`eglChooseConfig()`	Select configuration options
`eglCreateWindowSurface()`	Create drawable surface (bound to native window)
`eglCreateContext()`	Create OpenGL context
`eglMakeCurrent()`	Activate context and surface

Table 3: EGL function for OpenGL ES initialization

Graphical pipeline is described in the diagram below. The pipeline is programmable, the

program runs on the GPU, while OpenGL API is used for communication with the application running on CPU.

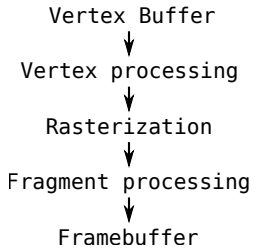


Figure 3: OpenGL ES pipeline

Programs are compiled from source and written in the GLSL language, each program consists of vertex and fragment shader. Following function are available in the API:

Function	**Description**
`glCreateShader()`	Create shader
`glShaderSource()`	Load shader source
`glCompileShader()`	Compile shader from loaded source
`glDeleteShader()`	Delete shader
`glShaderBinary()`	Load shader from binary data
`glCreateProgram()`	Create program
`glAttachShader()`	Add shader to program
`glLinkProgram()`	Link shaders in program
`glUseProgram()`	Switch between multiple programs
`glDeleteProgram()`	Delete program
`glGetUniformLocation()`	Access uniform variable defined in shader
`glGetAttribLocation()`	Access attribute variable defined in shader

Table 4: OpenGL functions for working with shader programs

The vertex shader processes geometry defined as array of verticies, it is executed per vertex. Result verticies are rasterized into fragments and then fragment shader is executed per fragment. Result fragments are then written into the framebuffer, each shader execution is done in parallel. Following figure shows how data are processed by the shader program.

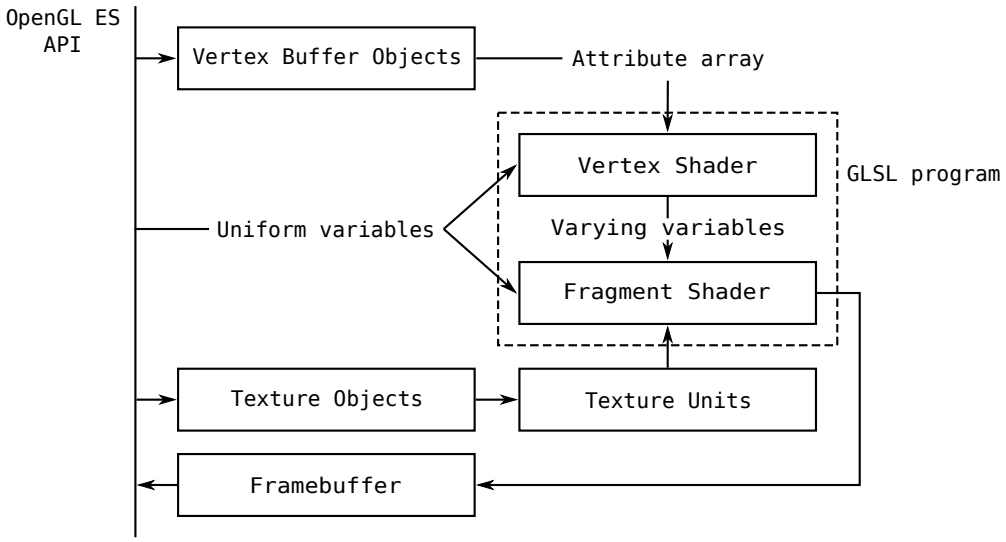


Figure 4: OpenGL ES shader program

GPU uses its own memory, it may be physically shared with the system memory, but is not directly accessible by the application. Verticies are stored in the GPU memory in vertex buffer objects (VBO), following API function are available:

Function	**Description**
`glGenBuffers()`	Create vertex buffer object
`glDeleteBuffers()`	Destroy vertex buffer object
`glBufferData()`	Load vertex data into VBO
`glBindBuffer()`	Bind VBO to attribute array
`glVertexAttribPointer()`	Specify attribute array
`glEnableVertexAttribArray()`	Enable attribute array

Table 5: OpenGL functions for working with VBOs

Shader programs have three types of variables, attributes, uniforms and varyings. Attributes are read from the attribute array which is bound to the VBO. Each vertex from the array is processed separately by each shader execution, having its value accessible by the attribute variables. Uniforms can be assigned directly by the application using `glUniform()`, they are read-only by the shader and they are shared by each execution. Varyings are used to pass variables from vertex shader to fragment shader, they are interpolated between verticies during the rasterization. Fragment shaders may use textures to to calculate fragment color, textures are stored in the GPU memory and loaded by the application. Following functions are available in the API:

Function	Description
`glGenTextures()`	Create texture object
`glTexImage2D()`	Load pixel data into texture object
`glTexSubImage2D()`	Load partial pixel data
`glTexParameter()`	Set parameter
`glBindTexture()`	Bind texture to active unit
`glActiveTexture()`	Select texture unit
`glDeleteTextures()`	Delete texture

Table 6: OpenGL functions for working with textures

Textures are bound to the texture units which may be accessed from the shader program. If the fragment shader needs to work with multiple textures, each texture needs to be loaded into a different texturing unit. The typical drawing loop is:

- select shader program
- assign uniform variables
- bind textures to texturing units
- bind VBOs to attribute arrays
- start processing with `glDrawArrays()`
- swap framebuffers and repeat

2.3.2 OpenGL ES Shading Language

Shader programs are designed to be executed in parallel, over large blocks of data. The language is similar to C, the minimal structure of the vertex shader is:

```
attribute vec4 vertex;
varying vec2 texcoord;

void main()
{
  gl_Position = vec4(vertex.xy, 0, 1);
  texcoord = vertex.zw;
}
```

and the fragment shader:

```
uniform sampler2D texture;
varying vec2 texcoord;

void main()
{
  gl_FragColor = texture2D(texture, texcoord);
}
```

This vertex shader takes one vertex attribute as a vector, where first two fields are display coordinates and the second two fields are texture coordinates. It defines varying variable to pass texture coordinates to the fragment shader. The `gl_Position` is special variable resembling the output of the vertex shader (the vertex position). The fragment shader access texturing unit passed as a uniform variable with the interpolated texture coordinates from the vertex shader. Result is written to `gl_FragColor` which is a special variable resembling the output of the fragment shader (the RGBA color). GLSL supports vector and matrix types, for example `vec4` is the four element vector and `mat3` is the 3x3 matrix. Vectors may be combined freely for example

```
vec4(vec4(0, 1, 2, 3).xy, 4, 5) * 1.5
```

will create vector (0, 1, 2, 3), take its first two fields (x, y) to create vector (0, 1, 4, 5) and then multiply by scalar resulting in vector (0, 1.5, 6, 7.5). GLSL also features standard mathematical functions such as trigonometry, exponential, geometric, vector or matrix. The `sampler2D` type refers to the texturing unit, to access its texture function `texture2D()` is used. The way by which samplers calculates the texture color is determined by the parameters set through the API. For example having two pixel texture with one black and one white pixel with setting

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

will cause linear color mapping and therefore black-white color gradient. However setting

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

will create chessboard pattern. These setting are done per texture object and reflected by texturing unit to which the texture is bound.

2.3.3 TrueType font rendering

In order to be able to render TrueType vector fonts, each glyph needs to be pre-rasterized first. The best method to achieve this is to create glyph atlas texture with all glyphs needed and then generate strings as VBOs with proper texture coordinates for each character. To

utilize unicode support, this atlas needs to be appended by newly requested characters in real-time as there are too many glyphs to be pre-rasterized. The FreeType2 library can rasterize glyphs from the TrueType font file on the fly. These glyphs are in fact alpha maps to be processed by the fragment shader:

```
gl_FragColor = color * texture2D(texture, texcoord).a;
```

or

```
gl_FragColor = vec4(color.rgb, texture2D(texture, texcoord).a);
```

if the alpha blending is enabled with

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

where `color` is the text color.

The FreeType library loads the font faces from `.ttf` files and has following API:

Function	**Description**
`FT_Init_FreeType()`	Initialize the library
`FT_New_Face()`	Load font face from file
`FT_Select_Charmap()`	Set encoding (Unicode)
`FT_Set_Pixel_Sizes()`	Set font size
`FT_Load_Char()`	Rasterize single glyph
`FT_Done_FreeType()`	Release resources

Table 7: FreeType API

The glyphs are loaded to texture with `glTexSubImage2D()`. To change the font size it is better to rasterize new glyph with `FT_Load_Char()`, then trying to scale the glyph in the fragment shader. To achieve best results, there should be 1:1 mapping between glyph pixels and OpenGL fragments and blending should be enabled. The library also supports kerning and other features to make the result text better or to create special effects. Special care needs to be taken into account about pixel alignment when rendering text. Vertex shader must ensure that each glyph verticies are aligned without pixel fractions. The typical example is the centering of text with even pixel width, which causes verticies to be aligned with 0.5 pixel offset. This causes major aliasing artifacts during rasterization and can be prevented by subtracting the fragment coordinate fractional part in the vertex shader.

2.3.4 Texture streaming extensions

Loading textures the standard way causes copying the pixel buffer to the texture memory, which is very inefficient if the texture needs to be changed often. This is typical to

streaming video through the GPU. There is a `KHR_image_base` extension for the EGL and a `OES_EGL_image` extension for the OpenGL ES defined in `EGL/eglext.h` and `GLES2/gl2ext.h` respectively. These extensions are platform specific, this text refers to the Texas Instruments implementation. The `EGLImage` offers a way to map images in the EGL API to be accessed in OpenGL as `GL_TEXTURE_EXTERNAL_OES` textures. This works as memory mapping and no copying is done whatsoever, synchronization is handled by the application. The mapping is created by

```
EGLImageKHR img =
  eglCreateImageKHR(dpy, EGL_NO_CONTEXT, EGL_RAW_VIDEO_TI, ptr, attr);
glEGLImageTargetTexture2DOES(GL_TEXTURE_EXTERNAL_OES, (GLeglImageOES)img);
glBindTexture(GL_TEXTURE_EXTERNAL_OES, myTexture);
```

where `dpy` is the active EGL display, `ptr` is pointer to the video buffer and `attr` is array of configuration options. This extension is also able to perform YUV to RGB color-space conversion in the background.

2.4 Inertial measurement subsystem

Application needs to know its spatial orientation for rendering, there are three devices which may provide such information, gyroscope, compass (magnetometer) and accelerometer. Hardware details about these devices are in the hardware (3) chapter.

2.4.1 Industrial I/O module

A relatively young kernel module `iio` has been implemented in recent kernels to provide standardized support for sensors and analog converters typically connected by I^2C bus. While many device drivers are still in staging tree, to core module is ready for production code. Subsystem provides device structure mapped in `sysfs`, typically available at `/sys/bus/iio/devices/`. Device are implemented usually on top of the `i2c-dev` driver and registered as `/sys/bus/iio/devices/iio:deviceX`, where X is the device number and the device name may be obtained by

```
cat /sys/bus/iio/devices/iio:deviceX/name
```

There are many possible channels, named by the value type they represents. To read an immediate value, for example from an ADC channel 1

```
cat /sys/bus/iio/devices/iio:deviceX/in_voltage1_raw
cat /sys/bus/iio/devices/iio:deviceX/in_voltage_scale
```

where the result value in volts is $raw \cdot scale$. However, being easy, this is not efficient, buffers have been implemented to stream measured data to the application. Buffer uses device file named after the `iio` device, e.g. `/dev/iio:deviceX`. To stream data through the buffer, driver needs to have control over the timing, triggers have been implemented for this purpose.

They are accessible as `/sys/bus/iio/devices/triggerX`, where X is the trigger number and its name may be obtained by

```
cat /sys/bus/iio/devices/triggerX/name
```

Software trigger may be created by

```
echo 1 > /sys/bus/iio/iio_sysfs_trigger/add_trigger
```

and triggered by application

```
echo 1 > /sys/bus/iio/trigger0/trigger_now
```

Name of this trigger is `sysfstrigX`, where X is the trigger number. Hardware triggers are also implemented, both GPIO and timer based triggers. Devices may implement triggers themselves, providing for example the data ready trigger. Device triggers are generally named as `name-devX`, where `name` is device name and `X` is device number. To use trigger with the buffer use

```
echo "triggername" > /sys/bus/iio/devices/iio:deviceX/trigger/current_trigger
```

where `triggername` is the name of the trigger, for example `adc-dev0` will be the device trigger for the ADC. Data are measured in specific channels, they are defined in `/sys/bus/iio/devices/iio:device0/scan_elements`. Channels must be enabled for buffering individually, for example

```
echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_voltage1_en
echo 1 > /sys/bus/iio/devices/iio:device0/scan_elements/in_voltage2_en
```

will enable ADC channels 1 and 2. Buffer itself can be started by

```
echo 256 > /sys/bus/iio/devices/iio:deviceX/buffer/length
echo 1 > /sys/bus/iio/devices/iio:deviceX/buffer/enabled
```

this will start streaming data to the device file. Data are formatted in packets, each packed consists of per-channel values and is terminated by 8 byte time-stamp of the sample. Order of the channels in the buffer can be obtained by

```
cat /sys/bus/iio/devices/iio:device0/scan_elements/in_voltageX_index
```

which reads index of the specified channel. Data format of this channel is

```
cat /sys/bus/iio/devices/iio:device0/scan_elements/in_voltageX_type
```

which reads encoded string, for example `le:u10/16>>0`, where `le` means little-endian, `u` means unsigned, `10` is the number of relevant bits while `16` is the number of actual bits and `0` is the number of right shifts needed.

Following channels are needed by the application:

- `anglvel_x`
- `anglvel_y`
- `anglvel_z`
- `accel_x`

- `accel_y`
- `accel_z`
- `magn_x`
- `magn_y`
- `magn_z`

representing measurements from gyroscope, accelerometer and magnetometer respectively.

2.4.2 DCM algorithm

Equations needed to calculate device attitude have been derived from [10]. Gyroscope measures angular speed around device axes, it offers high differential precision and fast sampling rate, however it suffers slight zero offset error. Device attitude can be obtained simply by integrating measured angular rates, provided that initial attitude is known. The angular rate is defined as

$\overrightarrow{\omega_g} = \frac{\mathrm{d}}{\mathrm{d}t}\overrightarrow{\Phi}_{(t)}$,

so the angular displacement between last two samples is

$[\Phi_x, \Phi_y, \Phi_z] = [\omega_x, \omega_y, \omega_z] \cdot_\Delta t$.

This can be described as a rotation

$$\mathbf{R}_{gyro} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\Phi_x) & -\sin(\Phi_x) \\ 0 & \sin(\Phi_x) & \cos(\Phi_x) \end{bmatrix} \times \begin{bmatrix} \cos(\Phi_y) & 0 & \sin(\Phi_y) \\ 0 & 1 & 0 \\ -\sin(\Phi_y) & 0 & \cos(\Phi_y) \end{bmatrix} \times \begin{bmatrix} \cos(\Phi_z) & -\sin(\Phi_y) & 0 \\ \sin(\Phi_y) & \cos(\Phi_y) & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

With $_\Delta t$ close to zero a small-angle approximation may be used to simplify $\cos(x) = 1$, $\sin(x) = x$

$$\mathbf{R}_{gyro} \doteq \begin{bmatrix} 1 & -\Phi_z & \Phi_y \\ \Phi_x\Phi_y + \Phi_z & 1 - \Phi_x\Phi_y\Phi_z & -\Phi_x \\ \Phi_x\Phi_z - \Phi_y & \Phi_x + \Phi_y\Phi_z & 1 \end{bmatrix}.$$

Let us define the directional cosine matrix describing device attitude

$$\mathbf{DCM} = \begin{bmatrix} \widehat{\mathbf{I}} \cdot \widehat{\mathbf{x}} & \widehat{\mathbf{I}} \cdot \widehat{\mathbf{y}} & \widehat{\mathbf{I}} \cdot \widehat{\mathbf{z}} \\ \widehat{\mathbf{J}} \cdot \widehat{\mathbf{x}} & \widehat{\mathbf{J}} \cdot \widehat{\mathbf{y}} & \widehat{\mathbf{J}} \cdot \widehat{\mathbf{z}} \\ \widehat{\mathbf{K}} \cdot \widehat{\mathbf{x}} & \widehat{\mathbf{K}} \cdot \widehat{\mathbf{y}} & \widehat{\mathbf{K}} \cdot \widehat{\mathbf{z}} \end{bmatrix} = \begin{bmatrix} \widehat{\mathbf{I}}_{xyz} \\ \widehat{\mathbf{J}}_{xyz} \\ \widehat{\mathbf{K}}_{xyz} \end{bmatrix},$$

where $\widehat{\mathbf{I}}$ points to the north, $\widehat{\mathbf{J}}$ points to the east, $\widehat{\mathbf{K}}$ points to the ground and therefore $\widehat{\mathbf{I}} = \widehat{\mathbf{J}} \times \widehat{\mathbf{K}}$. Roll, pitch and yaw angels in this matrix are

$\gamma = -\arctan_2\left(\frac{\mathbf{DCM}_{23}}{\mathbf{DCM}_{33}}\right)$,

$\beta = \arcsin(\mathbf{DCM}_{13})$,

$\alpha = -\arctan_2\left(\frac{\mathbf{DCM}_{12}}{\mathbf{DCM}_{11}}\right)$.

DCM can be computed by applying consecutive rotations over time

$\mathbf{DCM}_{(t)} = \mathbf{R}_{gyro(t)} \times \mathbf{DCM}_{(t-1)}$.

If the sampling rate is high enough (over 1kHz at least), this method is very accurate and has good dynamics over short periods of time, but in longer runs errors integrated during processing will cause serious drift (both numerical errors and zero offset errors). To mitigate these problems accelerometer and compass has to be used to provide the initial attitude and to fix the drift over time. Accelerometer measures external mechanical forces applied to the device together with gravitational force. However precision of these devices are generally worse and they have slower sampling rates. If there are no extern forces, it will measure the gravitational vector directly, thus providing the third row of the DCM

$$\overrightarrow{\mathbf{a}}_{acc} = g\ \widehat{\mathbf{K}}_{xyz} + \frac{\overrightarrow{\mathbf{F}}}{m},$$
$$\overrightarrow{\mathbf{F}} = 0\ \rightarrow\ \widehat{\mathbf{K}}_{xyz} = \frac{\overrightarrow{\mathbf{a}}_{acc}}{\left|\overrightarrow{\mathbf{a}}_{acc}\right|}.$$

When there is an external force $\overrightarrow{\mathbf{F}}$ applied, which is not parallel and has significant magnitude relative to gravitational force $m\,g\,\widehat{\mathbf{K}}_{xyz}$, measurements will degrade rapidly reaching singularity during the free fall ($\left|\overrightarrow{\mathbf{a}}_{acc}\right| = 0$). This error may be corrected by using device speed measured by satellite navigation system with high sample rate (over 10Hz)

$$\widehat{\mathbf{K}}_{xyz} = \frac{\overrightarrow{\mathbf{a}}_{acc} - \frac{\mathrm{d}}{\mathrm{dt}}\overrightarrow{\mathbf{v}}_{GPS}}{g}.$$

Magnetometer has similar properties, it measures magnetic flux density of the field the device is within. This should ideally result in a vector pointing to the north, therefore providing the first row of the DCM

$$\widehat{\mathbf{I}}_{xyz} = \frac{\overrightarrow{\mathbf{B}}_{corr}}{\left|\overrightarrow{\mathbf{B}}_{corr}\right|}.$$

Magnetometers have even slower sampling rates and far worse precision as the Earth field is distorted by nearby metal objects. This magnetic deviation can be divided into hard-iron and soft-iron effects. Hard-iron distortion is caused by materials that produces magnetic field, that is added to the Earth magnetic field. Vector of this field can be subtracted to compensate this error

$$\overrightarrow{\mathbf{B}}_{corr1} = \overrightarrow{\mathbf{B}}_{mag} - \tfrac{1}{2}\left[\min(B_x) + \max(B_x), \min(B_y) + \max(B_y), \min(B_z) + \max(B_z)\right].$$

The soft-iron distortion is caused by soft magnetic materials, which reshapes the field in a way that is not simply additive. It may be observed as an ellipse when the device is rotated around and the measured values are plotted. Compensating for these effects is involves remapping this ellipse back to the sphere. This is computation intensive and as soft-iron effects are usually weak (up to few degrees), it may be omitted.

Further more the magnetic field of the Earth itself does not point to the geographic north, but is rotated by an angle specific to the location on the Earth surface. Magnetic inclination is the vertical portion of this rotation causing magnetic vector to incline to the ground, it may be fixed by using measurements from the accelerometer to make the magnetic vector perpendicular to the gravitational vector

$\overrightarrow{\mathbf{B}}_{corr2} = \widehat{\mathbf{K}}_{xyz} \times \overrightarrow{\mathbf{B}}_{mag} \times \widehat{\mathbf{K}}_{xyz}$.

Magnetic declination (sometimes referred as magnetic variation) is the horizontal portion of this rotation and is sometimes provided by the satellite navigation systems. To correct for this error, measured values have to be rotated by the inverse angle

$$\overrightarrow{\mathbf{B}}_{corr3} = \overrightarrow{\mathbf{B}}_{mag} \begin{bmatrix} \cos(var) & \sin(var) \\ -\sin(var) & \cos(var) \end{bmatrix}.$$

By combination of the corrected results from accelerometer and magnetometer complete DCM can be calculated. Weighted average should be used, in real-time this yields

$$\mathbf{DCM}_{(t)} = W_{gyro} \left(\mathbf{R}_{gyro} \times \mathbf{DCM}_{(t-1)}\right) + (1 - W_{gyro}) \begin{bmatrix} \widehat{\mathbf{I}}_{xyz} \\ \widehat{\mathbf{K}}_{xyz} \times \widehat{\mathbf{I}}_{xyz} \\ \widehat{\mathbf{K}}_{xyz} \end{bmatrix},$$

where $\widehat{\mathbf{I}}_{xyz}$ and $\widehat{\mathbf{K}}_{xyz}$ are calculated from magnetometer and accelerometer measurements. W_{gyro} is the weight of the gyroscope measurement, it must be estimated by trial and error to mitigate its drift but not add too much noise. The DCM rows needs to be normalized after computing the average, as the equation does not ensure it.

2.5 Satellite navigation subsystem

2.5.1 Communication

Systems such GPS uses numerous satellites orbiting Earth transmitting their position and time. Receiver can measure its position and time with accuracy based on number of satellites in view. GPS receivers usually communicate via serial line, Linux kernel features TTY module originally used for teletypewriters. It handles serial lines, converters and virtual lines with devices generally called `/dev/tty*`. Serial connections (UART, RS-232) are usually named `/dev/ttySn`, where `n` is a device number. Emulated connection are named `/dev/ttyUSBn` for USB emulators or `/dev/ttyACMn` for modem emulators. There are also pseudo terminals in `/dev/pts/` used for software emulation. This devices allows standard I/O and may be controlled with functions defined in `termios.h` or by using stty(1) utility. Important physical interface settings are

- baud rate
- parity
- flow control

Usual settings are 4800 or 38 400 baud with no parity and no flow control, without proper interface configuration, no communication will occur Driver further provides line processing before passing data to the application. Important line settings are

- newline handling
- echo mode

- canonical mode
- timeouts and buffer sizes

In canonical mode, driver uses line buffer and allows line editing. Buffer is passed to the application (made readable) after new line is encountered. It may be combined with echo mode, which will automatically send received characters back. This is how standard console line works, however if application needs to be in full control, driver may be switched to raw mode with per-character buffers and echo disabled. Also specific control characters for canonical mode may be configured.

GPS receivers uses NMEA 183 standard [11]. Communication is done in sentences, each message starts with dollar sign followed by source name and sentence type, then there is comma separated list of data fields optionally terminated by a checksum. Each sentence is appended with carriage return and new line. For example in sentence

```
$GPGGA,000000,4914.266,N,01638.583,E,1,8,1,257,M,46.43,M,,*46
```

`GP` is source name (GPS receiver), `GGA` is sentence type (fix information), `*46` is a checksum and rest are data fields. Checksum is delimited by asterisk and consist of two character hexadecimal number calculated as bitwise XOR of all character between dollar and asterisk. Note that fields may be empty (as last two fields in the example) and numbers may have fractional parts.

Name	Description	Important information
GSA	Satellites (overall)	Visible satellites, dilution of precision
GSV	Satellites (detailed)	Satellite number, elevation, azimuth (per satellite)
GGA	Fix information	Time, latitude, longitude, altitude, fix quality
RMC	Position data	Time, latitude, longitude, speed, track
RMB	Navigation data	Destination, range, bearing

Table 8: Important NMEA 0183 sentences

The **GSA** sentence has following fields:

- 1st field is `A` for automatic selection of 2D or 3D fix, `M` for manual
- 2nd field is `3` for 3D fix, `2` for 2D fix or `1` for no fix
- 3-14 fields are identification numbers (PRN) of satellites in view
- 15-17 fields are dilution of precision and its horizontal and vertical parts

The **GSV** sentence has following fields:

- 1-2 fields are number of partial sentences and number of current part

- 3rd field is number of satellites in view
- 4th field is satellite number (PRN)
- 5th field is satellite elevation
- 6th field is satellite azimuth
- 7th field is satellite signal to noise ratio in decibels
- 8-11, 12-15, 16-19 fields are for other satellites info (number, elevation, azimuth, SNR)

The **GGA** sentence has following fields:

- 1st field resembles time of the fix in format `hhmmss.ff`, where `hh` are hours, `mm` are minutes, `ss` are seconds and `ff` are fractional seconds.
- 2-3 fields resembles device latitude in format `ddmm.ff,[NS]`, where `dd` are degrees, `mm` are minutes, `ff` are fractional minutes, `N` means north hemisphere and `S` means south.
- 4-5 fields resembles device longitude in format `dddmm.ff,[EW]`, where `E` means east of Greenwich and `W` means west.
- 6th field is fix type, 1 means GPS fix
- 7th field is number of satellites in view
- 8th field is horizontal dilution of precision
- 9-10 fields resembles device altitude above mean sea level, first field is the value and second field is the unit used
- 11-12 fields resembles height of geoid above WGS84 in the same fashion
- 13-14 fields are usually unused and refers to differential GPS

The **RMC** sentence has following fields:

- 1-5 fields are same as in GGA sentence
- 6th field is speed over the ground in knots
- 7th field is track angle in degrees
- 8th field resembles date in format `ddmmyy`, where `dd` is day, `mm` is month and `yy` is last two digits of year
- 9-10 fields resembles magnetic variation, first field is value in degrees, second field is `E` meaning east or `W` meaning west

The **RMB** sentence has following fields:

- 1st field is status, `A` means OK
- 2-3 fields resembles cross-track error, first field is the value in nautical miles, second field is `E` meaning east or `W` meaning west
- 4th field is origin waypoint name
- 5th field is destination waypoint name
- 6-9 fields are destination waypoint latitude and longitude with same formatting as in GGA sentence

- 10th field is range to destination in nautical miles
- 11th field is bearing to destination in degrees
- 12th field is velocity towards destination in knots
- 13th field is `A` for arrived or `V` for not arrived to destination

2.5.2 Navigation

As Earth shape is very complex, there are two layers of approximation used for computing position. Geoid is the equipotential surface, which describes mean ocean level if Earth was fully covered with water. Most recent geoid model is EGM96 which is used together with WGS84 reference ellipsoid [12]. This ellipsoid has semi-major axis of $a = 6378137$ meters and flattening $f = 1/298.257223563$. Note that ellipsoid flattening is defined as

$$f = \frac{a-b}{a},$$

where b is the semi-minor axis. The eccentricity is defined as

$$e = 2f - f^2.$$

Geodetic latitude φ is the angle between normal to the reference ellipsoid and the equator, longitude λ is the angle between normal to the reference ellipsoid and the prime meridian. Because of the flattening, the normal does not intersect ellipsoid center. Geocentric latitude ψ uses line running through the center instead of the normal,

$$\psi = \arctan[(1-e)^2 \tan(\varphi)].$$

Device position measured by GPS is defined by its geodetic latitude, longitude and altitude

$$h_{AMSL} = h_{WGS84} - h_{EGM96}$$

measured as height above mean see level, where h_{WGS84} is the height above reference ellipsoid and h_{EGM96} is the height above geoid. As GPS sensors usually send fix information at low rates, position needs to be interpolated between fixes. If current ground speed v_{GND} and course α_{TRK} are known, the interpolated position in next step will be

$$\varphi_{(t+1)} = \varphi_{(t)} + \frac{\cos(\alpha_{TRK}) \cdot v_{GND} \cdot_\Delta t}{R_{(\varphi)}},$$

$$\lambda_{(t+1)} = \lambda_{(t)} + \frac{\sin(\alpha_{TRK}) \cdot v_{GND} \cdot_\Delta t}{R_{(\varphi)} \cdot \cos(\varphi)},$$

where R is the ellipsoid radius at given latitude

$$R = \frac{\sqrt{b^4 \sin(\varphi)^2 + a^4 \cos(\varphi)^2}}{\sqrt{b^2 \sin(\varphi)^2 + a^2 \cos(\varphi)^2}}.$$

Application needs to know projections of specific landmarks as normalized horizontal and vertical coordinates (in range of -1 to 1) used for rendering

$$\begin{bmatrix} x \cdot FOV_x \\ y \cdot FOV_y \end{bmatrix} = \begin{bmatrix} \alpha_{proj} - \alpha_{dev} \\ \beta_{proj} - \beta_{dev} \end{bmatrix} \times \begin{bmatrix} \cos(\gamma_{dev}) & -\sin(\gamma_{dev}) \\ \sin(\gamma_{dev}) & \cos(\gamma_{dev}) \end{bmatrix},$$

where FOV is the field of view, *dev* means device angle (calculated by inertial subsystem (2.4)) and *proj* means projection angle (defined later on in this section). Orthodrome (great circle) is the intersection of a sphere and a plane passing though its center. However, because Earth flattening is rather small, it may be used as an approximation for a curve following Earth surface, connecting two points with shortest route. Spherical trigonometry [13] defines basis for orthodrome calculations. Heading changes along the route and its initial value is the horizontal projection angle

$$\alpha_{proj} = \arctan\left(\frac{\sin(\lambda - \lambda_0)\cos(\varphi)}{\cos(\varphi_0)\sin(\varphi) - \sin(\varphi_0)\cos(\varphi)\cos(\lambda - \lambda_0)}\right),$$

the zero index refers to the device coordinate. Angular distance between those two points is

$$\phi = \arccos(\sin(\varphi_0)\sin(\varphi) + \cos(\varphi_0)\cos(\varphi)\cos(\lambda - \lambda_0)).$$

Vertical projection angle is the angle between the horizon (perpendicular to normal) and a line directly connecting the points. Lets construct a triangle connecting the points with the center of the reference ellipsoid, assuming zero flattening, so the normals have intersection in the center ($f = 0 \to e = 0 \to \psi = \varphi$).

$$a_\Delta = h_0 + R_{(\varphi_0)},$$

$$b_\Delta = h + R_{(\varphi)},$$

$$c_\Delta = \sqrt{a_\Delta^2 + b_\Delta^2 - 2a_\Delta b_\Delta \cos(\phi)},$$

$$\beta_{proj} = \arcsin\left(\frac{b_\Delta}{c_\Delta}\sin(\phi)\right) - \tfrac{\pi}{2},$$

where h is height above reference ellipsoid. These calculations are quite complex and there is plenty of room for approximation. Over short distance, such as typical in visual ranges, orthodromes may be replaced with loxodromes (rhumb lines). Loxodrome is a curve following Earth surface, connecting two points with constant heading. It has similar path to orthodrome, if the points are relatively far from poles a close together. Simplified heading along the line is

$$\alpha_{proj} \doteq \arctan\left(\frac{\lambda\cos(\varphi) - \lambda_0\cos(\varphi_0)}{\varphi - \varphi_0}\right).$$

Note that loxodrome approximation will fail horribly near poles as the curve will run circles around the pole. Angular distance along loxodrome is

$$\phi \doteq \sqrt{(\varphi - \varphi_0)^2 + (\lambda\cos(\varphi) - \lambda_0\cos(\varphi_0))^2}$$

and a horizontal distance along the arc between those points is

$$d = \phi \cdot R_{(\varphi_0)},$$

assuming flattening difference is close to zero, so R is constant along the curve. With the approximation of local flat Earth surface perpendicular to the normal, the simplified vertical projection angle is

$$\beta_{proj} \doteq \arctan\left(\frac{h - h_0}{d}\right).$$

This approximation will fail at higher altitudes when the visibility range is high enough to make the Earth curvature observable.

2.6 Output creation

The image below shows the actual output rendered by the application, there is no video source (solid white background). Resolution was reduced to 320x240 to make the image small, this caused some aliasing artifacts normally not visible. In the top left corner is the current speed over ground as reported by the satellite navigation, while in the top right corner is the current altitude above mean sea level. Speed is displayed in kilometers per hour and the altitude in meters. In the top center is the current waypoint name and distance in kilometers, if provided by the NMEA 0183 navigation data sentence (RMB). There is a heading ruler in the bottom, it is scaled by camera field of view and oriented to current heading. Course to any visible object may be deducted directly from the ruler. There are two markers, the arrow marker point to current track angle as reported by the satellite navigation. The circle marker is the course to current waypoint if any. The dashed horizontal line in the center of the image is the horizon line, it follows the horizon as calculated by inertial subsystem by moving up or down and rotating around its center. Orientation is determined by the marginal markers, they points always to the ground. If the horizon is not visible, dashed arrow is shown instead pointing up or down to where the horizon is hidden.

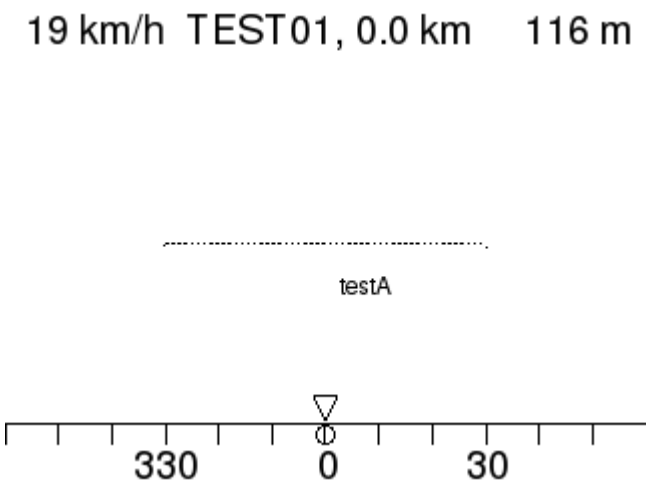


Figure 5: Visible output

Application accepts database of landmarks, following file was used in this example:

```
# Encoding ISO/IEC 8859-1
# Test landmarks (lat[rad], lon[rad], alt[m], name)
25e-5, 1e-5, 0, testA
50e-5, -1e-5, 100, testB
```

The `testA` landmark label is visible in the image, its location is centered directly above its projection. Projections are calculated in conjunction of both navigation subsystems and graphic acceleration. Large landmark database may be supplemented to provide spatial navigation references, only visible landmarks will be shown. This overlay is rendered in real time over the source video (there is none in the example), all rendering and data gathering methods are provided by the respective subsystems. Upon correct configuration, overlay elements (ruler, horizon and labels) should be aligned with the real visible places in the video frame. System allows free six degrees of freedom movement of the camera while still being able to render the overlay correctly. Waypoint management and route planning is done solely by the external navigation system, it is expected that it will provide its own user interface. This allows connection for example to PDAs or other specialized devices which delivers classic 2D moving map navigation to the user with its own controls.

3 Hardware

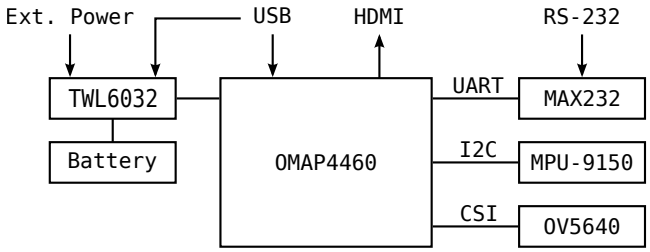


Figure 6: Proposed hardware

The diagram above specifies the proposed platform realization. The OMAP4460 [14] application processor was chosen for the project, it features two ARM Cortex-A9 general-purpose CPUs and many specialized subsystems and peripherals. Internal image co-processor allows direct connection of image sensor via its serial interface, for example the OV5640. For graphic acceleration, there is the SGX540 GPU, HDMI v1.3 video output is available. The MPU-9150 [15] is a motion tracking device composed of MPU-6050 3-axis gyroscope and accelerometer and AK8975 3-axis digital compass. It comes with a I^2C interface. TWL6032 is the power companion chip designed specifically for the OMAP platform, handling power path switching, and DC-DC voltage conversion. It can manage Li-ion battery as well as charging with extern power adapter or USB line. RS-232 connection is needed for external satellite navigation system as defined in NMEA 0183 specification, it should be connected to the UART via voltage level converter such as MAX232. USB device port allows connection to the external host, providing extra power source and generic data interface. Ethernet or mass storage gadget may be implemented over the USB device port. The embedded IVA 3 hardware video accelerator in the OMAP4460 is capable of encoding and decoding video streams simultaneously, without any load on the ARM cores, this allows media streaming through the USB port (possibly over IP network). External IP camera may also be connected. Additionally a USB host port may also be implemented for external camera connection.

There are also many development boards already available for generic testing. The Pandaboard ES [16] features the OMAP4460 and provides all peripheral interfaces. There is also a low price board the BeagleBone Black [17] with a Sitara AM3358, while being much simpler, however most of the features needed are still available. Texas Instruments also distributes the Sensor Hub BoosterPack [18] with all needed sensors.

Application may also be fully emulated on PC in Linux environment. OpenGL ES 2.0 should run under Mesa3D natively. For video interface, GStreamer may be used together with the *v4l2loopback* module. Though serial interfaces needs to be emulated manually.

4 Conclusion

Theoretical grounds have been made for the application development. Application has been implemented with sources available at [19]. The basic functionality has been established, the graphical output works just fine even on less resourceful systems such as Sitara AM3358. Only USB camera has been tested in RGB and YUV formats. The H.264 capability showed promise for both video input and output, however limited to the OMAP4460. The MPU-9150 has been successfully implemented, however the DCM algorithm needs to be specifically adapted to its sensors. The gyroscope has little jitter, low-pass filters seems to fix this, but software filtering should be also implemented. Accelerometer and magnetometer readings were very unstable, averaging ratios over 1:10 needs to be used. Overall, the application proved the concept, leaving space for further enhancements and hardware design.

References

[1] BIMBER, Oliver and RASKAR, Ramesh. *Spatial augmented reality: merging real and virtual worlds*. Wellesley: A K Peters, 2005. ISBN 15-688-1230-2.

[2] Bit Wise. FreeRTOS. [online]. 2013. [Accessed 20 November 2013]. Available from: http://www.freertos.org/

[3] Wind River. VxWorks. [online]. 2013. [Accessed 20 November 2013]. Available from: http://www.windriver.com/products/vxworks

[4] Microsoft. Windows Embedded. [online]. 2013. [Accessed 20 November 2013]. Available from: http://www.microsoft.com/windowsembedded/en-us/windows-embedded.aspx

[5] LinuxTV Developers. Linux Media Infrastructure API. [online]. 2012. [Accessed 20 November 2013]. Available from: http://linuxtv.org/downloads/v4l-dvb-apis

[6] Consultative Committee on International Radio. Recommendation BT.601. [online]. 2011. [Accessed 20 November 2013]. Available from: http://www.itu.int/rec/R-REC-BT.601/en

[7] TI OMAP Developers. TI OMAP Trunk PPA. [online]. 2013. [Accessed 20 November 2013]. Available from: https://launchpad.net/~tiomap-dev/+archive/omap-trunk

[8] Khronos Group. OpenGL ES 2.x - for Programmable Hardware. [online]. 2013. [Accessed 20 November 2013]. Available from: http://www.khronos.org/opengles/2_X

[9] MUNSHI, Aaftab, GINSBURG, Dan and SHREINER, Dave. *OpenGLES 2.0 programming guide*. New York: Addison-Wesley Professional, 2009. ISBN 978-0-321-50279-7.

[10] JAZAR, Reza N. *Theory of applied robotics: kinematics, dynamics, and control*. 2nd ed. New York: Springer, 2010. ISBN 978-1-4419-1749-2.

[11] National Marine Electronics Association. NMEA 0183. [online]. 2008. [Accessed 20 November 2013]. Available from: http://www.nmea.org/content/nmea_standards/nmea_0183_v_410.asp

[12] NIMA. Department of Defense World Geodetic System 1984. [online]. 1997. [Accessed 20 November 2013]. Available from: http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf

[13] WEISSTEIN, Eric W. *Spherical trigonometry* [online]. MathWorld. [Accessed 20 November 2013]. Available from: http://mathworld.wolfram.com/SphericalTrigonometry.html

[14] Texas Instruments. OMAP 4460 Multimedia Device. [online]. 2012. [Accessed 20 November 2013]. Available from: http://www.ti.com/product/omap4460

[15] InvenSense. MPU-9150 Nine-axis MEMS MotionTracking™ Device. [online]. 2013. [Accessed 20 November 2013]. Available from: http://www.invensense.com/mems/gyro/mpu9150.html

[16] pandaboard.org. Pandaboard. [online]. 2013. [Accessed 20 November 2013]. Available from: http://pandaboard.org/content/resources/references

[17] beagleboard.org. BeagleBone Black. [online]. 2013. [Accessed 20 November 2013]. Available from: http://beagleboard.org/products/beaglebone%20black

[18] Texas Instruments. Sensor Hub BoosterPack. [online]. 2013. [Accessed 20 November 2013]. Available from: http://www.ti.com/tool/boostxl-senshub

[19] JAROS, Martin. *Augmented reality navigation* [online]. [Accessed 13 December 2013]. Available from: https://github.com/martinjaros/augmented-reality-navigation

List of Annexes

A Threads example

In this example two threads share standard input and output, access is restricted by *mutex* so only one thread may use the shared resource at any time.

```c
#include <stdio.h>
#include <pthread.h>

void *worker(void *arg)
{
    static int counter = 1;
    pthread_mutex_t *mutex = (pthread_mutex_t*)arg;
    char *buffer;

    // Lock mutex to restrict access to stdin and stdout
    pthread_mutex_lock(mutex);
    printf("This is worker %d, enter something: ", counter++);
    scanf("%ms", &buffer);
    pthread_mutex_unlock(mutex);

    return (void*)buffer;
}

int main()
{
    pthread_mutex_t mutex;
    pthread_t thread1, thread2;
    char *retval1, *retval2;

    // Initialize two threads with shared mutex, use default parameters
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&thread1, NULL, worker, (void*)&mutex);
    pthread_create(&thread2, NULL, worker, (void*)&mutex);

    // Wait for both threads to finish and display results
    pthread_join(thread1, (void**)&retval1);
    pthread_join(thread2, (void**)&retval2);
    printf("Thread 1 returned with `%s`.\n", retval1);
    printf("Thread 2 returned with `%s`.\n", retval2);

    pthread_mutex_destroy(&mutex);
    return 0;
}
```

B Video capture example

In this example video device is configured to capture frames using memory mapping. These frames are dumped to standard output, instead of further processing.

```
#include <fcntl.h>
#include <unistd.h>
#include <poll.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <linux/videodev2.h>

int main()
{
    // Open device
    int fd = open("/dev/video0", O_RDWR | O_NONBLOCK);

    // Set video format
    struct v4l2_format format =
    {
        .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
        .fmt =
        {
            .pix =
            {
                .width = 320,
                .height = 240,
                .pixelformat = V4L2_PIX_FMT_RGB32,
                .field = V4L2_FIELD_NONE,
                .colorspace = V4L2_COLORSPACE_SMPTE170M,
            },
        },
    };
    ioctl(fd, VIDIOC_S_FMT, &format);

    // Request buffers
    struct v4l2_requestbuffers requestbuffers =
    {
        .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
        .memory = V4L2_MEMORY_MMAP,
        .count = 4,
    };
    ioctl(fd, VIDIOC_REQBUFS, &requestbuffers);
    void *pbuffers[requestbuffers.count];
```

```
40
41      // Map and enqueue buffers
42      int i;
43      for(i = 0; i < requestbuffers.count; i++)
44      {
45          struct v4l2_buffer buffer =
46          {
47              .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
48              .memory = V4L2_MEMORY_MMAP,
49              .index = i,
50          };
51          ioctl(fd, VIDIOC_QUERYBUF, &buffer);
52          pbuffers[i] = mmap(NULL, buffer.length,
53                             PROT_READ | PROT_WRITE, MAP_SHARED,
54                             fd, buffer.m.offset);
55          ioctl(fd, VIDIOC_QBUF, &buffer);
56      }
57
58      // Start stream
59      enum v4l2_buf_type buf_type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
60      ioctl(fd, VIDIOC_STREAMON, &buf_type);
61
62      while(1)
63      {
64          // Synchronize
65          struct pollfd fds =
66          {
67              .fd = fd,
68              .events = POLLIN
69          };
70          poll(&fds, 1, -1);
71
72          // Dump buffer to stdout
73          struct v4l2_buffer buffer =
74          {
75              .type = V4L2_BUF_TYPE_VIDEO_CAPTURE,
76              .memory = V4L2_MEMORY_MMAP,
77          };
78          ioctl(fd, VIDIOC_DQBUF, &buffer);
79          write(1, pbuffers[buffer.index], buffer.bytesused);
80          ioctl(fd, VIDIOC_QBUF, &buffer);
81      }
82  }
```

C Colorspace conversion example

In this example RGB to YUV color-space conversion is implemented in fragment shader. Each input channel has its own texturing unit, texture coordinates are divided by sub-sampling factor 4:2:0.

```
uniform sampler2D texY, texU, texV;
varying vec2 texCoord;

void main()
{
    float y = texture2D(texY, texCoord).a * 1.1644 - 0.062745;
    float u = texture2D(texU, texCoord / 2).a - 0.5;
    float v = texture2D(texV, texCoord / 2).a - 0.5;

    gl_FragColor = vec4(
        y + 1.596 * v,
        y - 0.39176 * v - 0.81297 * u,
        y + 2.0172 * u,
        1.0);
}
```