ProFUSION embedded systems

Tips and Tricks

to develop software for CE product on low-end hardware

LinuxCon Brasil
São Paulo

Gustavo Sverzut Barbieri

about me

- developer since 1991
- unicamp - computer engineering 2001-2005
- freevo - python media center 2003
- indt - embedded software 2006-2008
- profusion embedded systems - since 2008
- efl, python, ffmpeg, mplayer, systemd...

ce products

Shop2social - Android Market
Remove Bookmark
Navigation
Change Address
Bookmarks
Type
History
Cursor
Zoom
Scroll
SAMSUNG
PS3
SONY
G-SHOCK
SHOCK RESIST

ce products

- consumer electronics
- high volume - every cent counts
- well defined purpose
- target audience

consumer expectations - before

Video

Time

Setup

◀Custom ▶

Contrast 34
Bright 48
Sharpness 30
Color 36
Tint 00

consumer expectations - ipod (2001)

- raises the bar
- ease of use hits mass market
- ipod click wheel
  - . technical point-of-view: suboptimal
  - . commercial point-of-view: expensive
  - . users point-of-view: awesome!
- itunes - optimize and organize - EASY!
- music store: easy to get legal media

consumer expectations - iphone (2007)

- raises the bar, again

- introduces (mass market):

    . capacitive/glass touchscreen

    . highly responsive operating system

    . central application store and updates

    . easy mobile internet

- purpose not so well defined anymore

- impacts EVERY market: cars, planes, refrigerators...

developers expectations

- the best software architecture
- the most beautiful code
- the best algorithm
- scalable (screens, cores, ...)
- modular
- reusable

graphical designer expectations

- non-rectangular paths and shapes
- transparency, blur and other filters
- fluid animations
- ~~change design at any project stage~~
- ~~if illustrator/flash does, ce does as well~~

expectations summary

- developers and users differ widely
- designers and users tend to converge
- … developers shouldn't design a product
- … but designers are unrealistic

myths

- developers: fast feels fast
- designers: make everything themable
- users: effects are nice per-se, (ab)use them

solutions

- general:
  - . always focus on the user
  - . define your target audience
  - . define the product purpose
- technical:
  - . be responsive
  - . never block
  - . allow cancellation
  - . avoid work

be responsive

- provide user feedback as quickly as possible
- … graphics, sound, vibration
- good even if technically useless
- amiga: coprocessors
- windows: high priority mouse interruption
- touchscreens with click sound

never block

- cooperative tasks (idlers)
- threads
- processes

never block - cooperative

- cooperative tasks that preempt themselves
- best option for easy-to-segment tasks
- needs predictable task duration
- needs no locking, no race conditions
- not multi-core friendly
- easy to cancel
- integrates fine into main loops
- easy to update user interface

never block - cooperative

```
struct ctx {
   unsigned int current, end, step;
   double value;
   double *input;
};

bool sum_pow5(struct ctx *ctx) {
   unsigned int last = ctx->current + ctx->step;
   if (last > ctx->end)
      last = ctx->end;

   for (; ctx->current < last; ctx->current++)
      ctx->value += pow(ctx->input[ctx->current], 5);

   return ctx->current < ctx->end;
}
```

never block - cooperative

```
int main(int argc, char *argv[]) {
    // code...
    while (run) {
       do_something();
       if (needs_sum_pow5) {
          if (!sum_pow5(ctx)) {
             needs_sum_pow5 = false;
             printf("sum_pow5=%f\n", ctx->value);
          }
       }
    // code...
```

never block - threads

- task is preempted by kernel
- best option for hard-to-segment tasks
- good for unpredictable task duration
- good for blocking syscalls, hardware access
- may need locking, may have race conditions
- multi-core friendly
- harder to cancel
- harder to update user interface (qt, gtk, efl...)

never block - threads

```
struct ctx {
   unsigned int count;
   double *input;
   enum { NEED, DOING, DONE, END } stage;
};
int cmp(const void *p1, const void *p2) {
   double *a = p1, *b = p2;
   return (int)(*a - *b);
}
void *th_sort(void *data) {
   struct ctx *ctx = data;
   qsort(ctx->input, ctx->count, sizeof(double), cmp);
   ctx->stage = DONE;
   return NULL;
}
```

never block - threads

```
int main(int argc, char *argv[]) {
    // code...
    while (run) {
        do_something();
        if (ctx->stage == NEED) {
            ctx->stage = DOING;
            pthread_create(&th, NULL, th_sort, ctx);
        } else if (ctx->stage == DONE) {
            pthread_join(&th);
            ctx->stage = DID;
            puts("thread sorted!");
        }
    // code...
```

never block - processes

- similar to thread
- usually for heavy-weight long running
- good for problem-prone (NFS, uninterpretable)
- different memory space - killable
- more robust
- harder to communicate - ipc/shmem
- harder to update user interface

allow cancellation

- if possible stop the task
- otherwise ignore its results
- rollback changes
- avoid partial work (leftovers)
- NEVER EVER pthread_cancel()

avoid work

- cache and pre-calculate
- offload (coprocessors or servers)
- optimizations (graphics)

avoid work - cache

- excellent for "pure" operations
- define allowed cache size (no leaks!)
- define invalidation policy (no stales!)
- optimize lookup (must be worth!)

avoid work - cache examples

- binary, validated and optimized files
- native objects retrieved from database
- decoded images, sounds and fonts

avoid work - offload

- use hw acceleration (audio, video, graphics)
- delegate work to remote servers
    . map routing
    . voice recognition (siri)
    . mail index and searching (gmail)

avoid work - graphics

- use specific painting operations
- do retained rendering
- employ occlusion

graphics - painting operations

- solid opaque fill

```
pixel_color = color;
```

- image blend with color and transparency

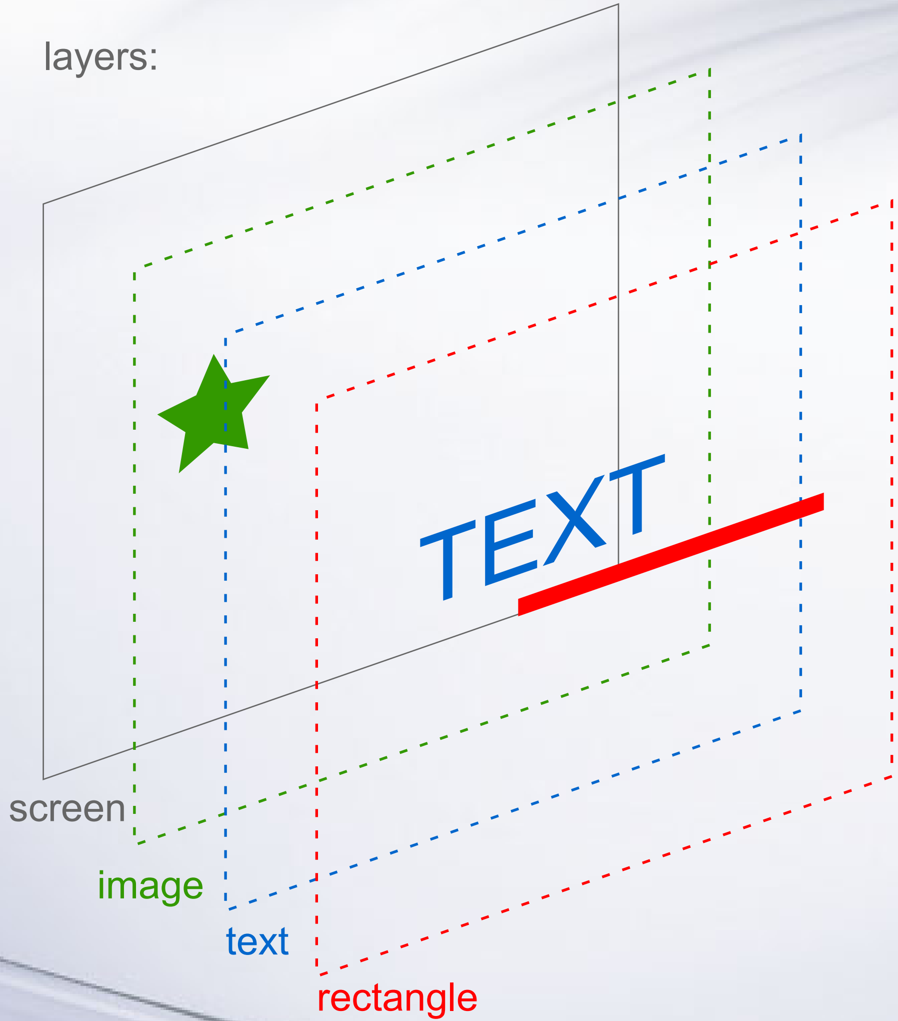
```
alpha2 = 255 - alpha1;
 pixel_color = (source1 * alpha1) / 255 +
   (((source2 * color) / 255) * alpha2) / 255;
```

- cost is very different!

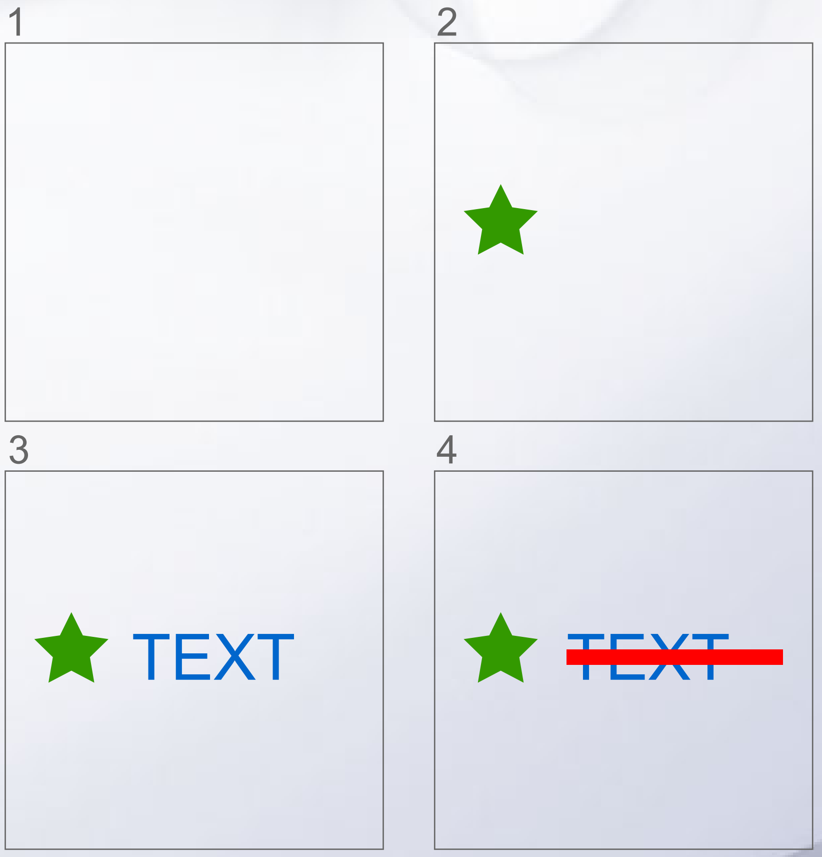
- prefer use RGB565 (16bpp) or YUV

graphics - retained rendering

layers:



painting steps:

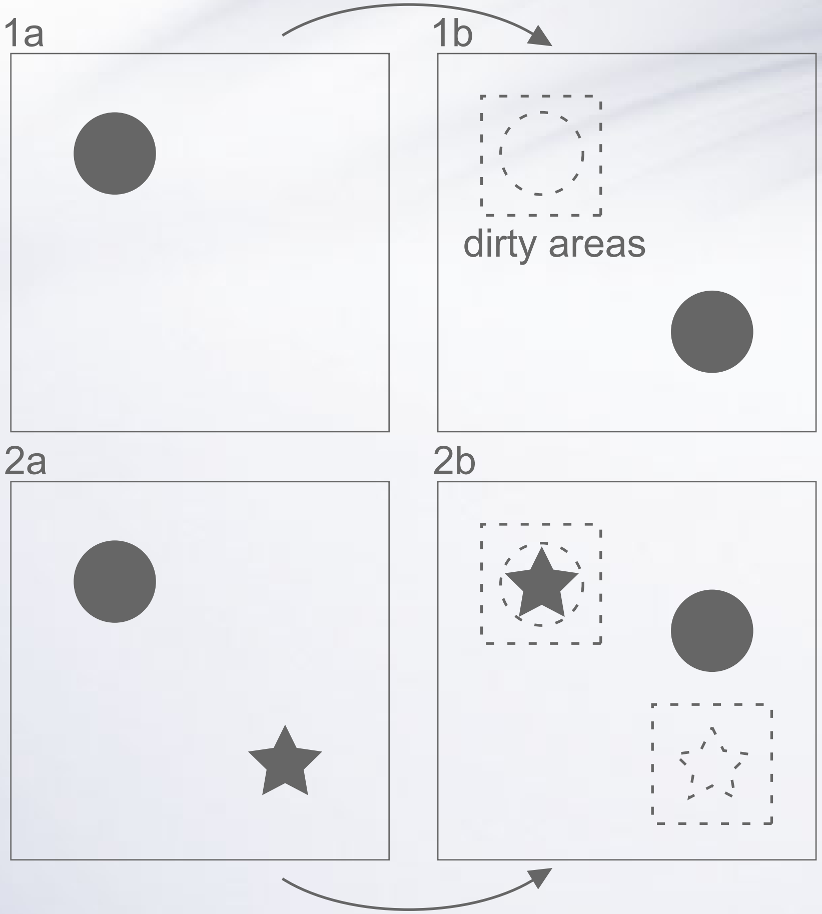


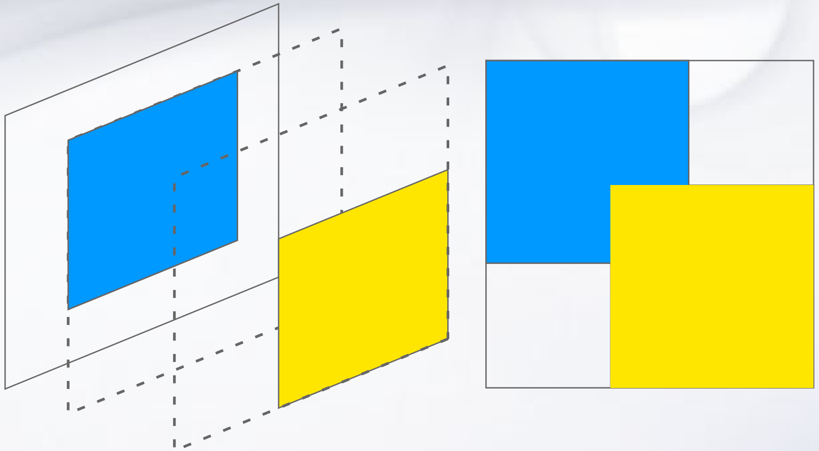
The **order** of composition of layers is important!

graphics - retained rendering

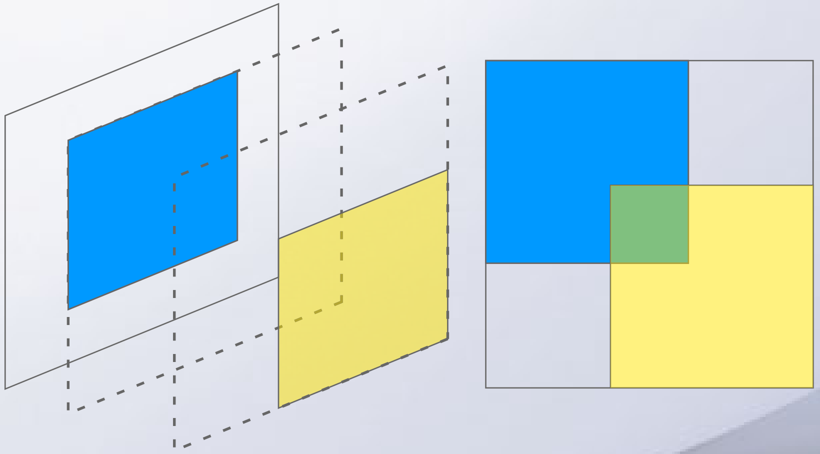
- objects are not rendered immediately
- state changes are remembered
- multiple changes != multiple rendering
- render phase will compute differences
- just visible changes should be used
- allows greater optimizations
- optimize how to know dirty regions

graphics - occlusion





Occlusion the area doesn't need to be painted blue.



Areas marked as dirty need to be painted (with background) to clean the image. REMEMBER: the ORDER MATTERS!

graphics - occlusion

- do not paint objects:
  - . outside the viewport
  - . under opaque regions
  - . obscured/forbidden regions
- optimize how to find out occlusions

general optimizations

- avoid memory allocations!
- avoid memory fragmentation
- replace copies with references
- use proper data structures
- be cpu cache-line friendly

efl - enlightenment foundation libraries

- focused on performance and low memory
- heavily optimized since 2001 (current set)
- most interesting libs:

  . eina - data types

  . eet - binary data store and load

  . evas - 2d drawing canvas

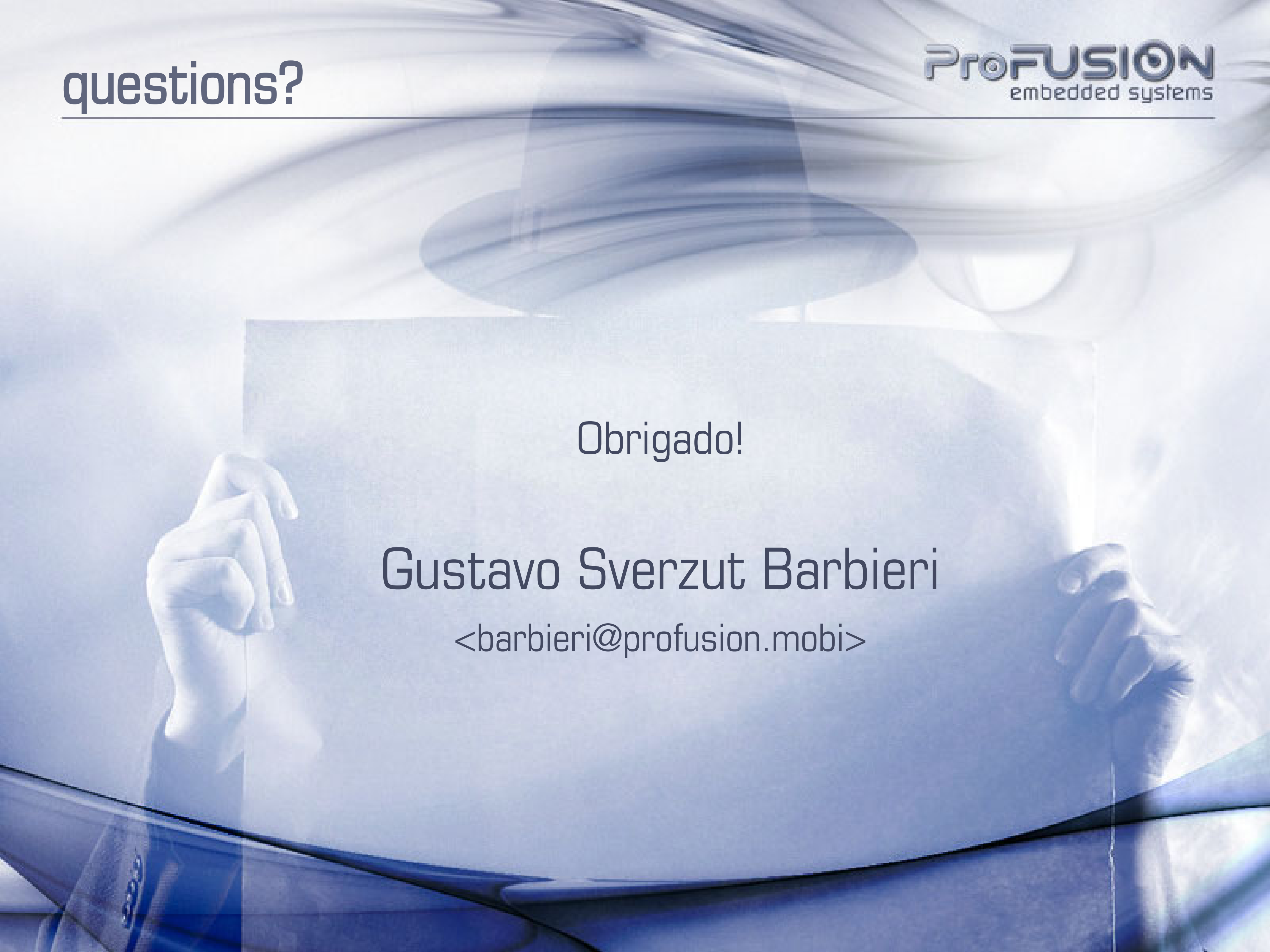
  . edje - themes, animations and layouts

  . elemetary - widget set

conclusion

- always focus on the user
- define your target audience
- define the product purpose
- be responsive and never block
- do not just optimize, avoid working at all!

questions?



Obrigado!

Gustavo Sverzut Barbieri

<barbieri@profusion.mobi>