Computer Laboratory: Building applications from scratch

2º L.EIC

Bruno Mendes

May 15, 2026

Outline

Defining *from scratch*

Are we really programming from a blank slate?

- In the embedded world, *from scratch* usually means *in a bare metal*, e.g. without a companion OS
- This is a significant effort that requires a lot of ceremony, such as setting up HW devices for initial usage
- This is not the environment we're working on in a virtualized x86 machine running Minix

If we go down the rabbit hole, we could even argue that *from scratch* would require controlling the HW, and there are ways to design it with code - lookup `Verilog`.

Our toolbox

From the first line of our `main` function, we have already several tools at our disposal:

- A C standard library implementation (*libc*) living in the Minix source files
- The Minix OS API, most notably the `SYS_DEVIO` calls for port-mapped I/O and `SYS_IRQCTL` for interrupt policy management
- Your device drivers developed during practical classes: a façade for the RTC, the timer, the keyboard, the mouse, and the graphics card

Tasks ahead

We'll use *from scratch* as *requiring more effort* or *having more control* compared to building a higher-level application.

- You'll need to design an event-driven core for dispatching HW events to relevant state handlers
- Rendering entities involves managing graphical memory directly, whose details you *really* want to abstract away
- The domain is novel – most real-world applications are scoped efforts, solving a specific problem (and if not, as in a large game, are split into gigantic teams)

Programming a reactive graphical application without a baggage like `GTK/Qt` or `Electron` is one kind of an experience.

The entrypoint

Delegates to the LCOM framework.

```
int main(int argc, char *argv[]) {
  lcf_set_language("EN-US");
  lcf_trace_calls("/home/lcom/labs/proj/trace.txt");
  lcf_log_output("/home/lcom/labs/proj/output.txt");
  if (lcf_start(argc, argv)) return 1;
  return lcf_cleanup();
}
```

- `lcf_start` will boot a tracing engine and a state machine responsible for asserting the sanity of some HW access operations, as seen in the labs, and then delegate to a function depending on *argv* – for the project, `lcom_run proj` will yield `proj_main_loop`
- This is a thin help layer – you may reduce it even further setting the `__LCOM_OPTIMIZED__` symbol

Compilation at scale

The C language notably lacks native extensive tooling.

- This is acceptable for a language designed in the 1970s, when most programs were relatively small and a single compilation command sufficed
- Modern C/C++ in the industry is likely compiled using `CMake`, `Bazel` or other more advanced custom build systems

Our goal with using `GNU Make` is enabling incremental compilation and tidier dependency management to speed up development.

GNU Make

- `Make` is a GNU utility originally designed for controlling the generation of executables and other non-source files of a program from the program's source files. It is also sometimes used as a simple command runner and dependency manager, although modern solutions like `mise` are usually a better fit for these tasks
- A `Makefile` may be seen as a recipe for assembling *targets*, requiring a set of *dependencies* and a sequence of build steps

```
target: dep1 dep2 dep3
    command1
    command2
```

A target is rebuilt whenever one of its dependencies changes – via last modified timestamp versus target.

Make substitutions

`Make` supports a lot of syntax, some dedicated to simple variable binding and operations over slice expressions. Its more powerful capabilities lie in the susbtitution model.

Syntax	**Meaning**
`$(VAR)`	Expand the value of a variable.
`$(SRC:.c=.o)`	Substitute every `.c` suffix in `SRC` with `.o`.
`$@`	The current target name.
`$<`	The first prerequisite.
`$^`	All prerequisites.

A Makefile

```
CC = clang
CFLAGS = -Wall
LDFLAGS = -L./lib
LDLIBS = -ltimer -lkbc -lgraphics
SOURCES = main.c player.c world.c
OBJECTS = $(SOURCES:.c=.o)
PROG = proj

$(PROG): $(OBJECTS)
        $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^ $(LDLIBS)

%.o: %.c
        $(CC) $(CFLAGS) -c $< -o $@

clean:
        rm -f $(TARGET) $(OBJECTS)
```

In LCOM, the Makefile extends over another, via an `.include` statement. Some variables are already defined and should be used instead.

C compiler options

Both `clang` and `gcc` support a large set of common compiler and linker flags inherited from the traditional Unix toolchain model.

Flag	Purpose
`-I<dir>`	Add an additional directory to the header search path used when resolving `#include` directives.
`-L<dir>`	Add an additional directory to the library search path used by the linker.
`-l<name>`	Link against a library named `lib<name>.a` or `lib<name>.so`.
`-D<macro>`	Define a preprocessor macro before compilation begins. Equivalent to inserting a `#define`.

These flags delegate work to different stages of the compilation pipeline.

Compilation stages

A modern compiler, especially for a strong statically typed language with powerful type systems—see Scala or Haskell—may contain hundreds of compilation stages. A typical C compiler, however, usually follows a much smaller pipeline:

- **Preprocessing**: expands macros, resolves `#include` directives, and performs conditional compilation
- **Lexical Analysis**: converts the raw character stream into tokens such as identifiers, keywords, operators, and literals
- **Parsing**: builds an Abstract Syntax Tree (AST) from the token stream according to the grammar of the language
- **Semantic Analysis**: performs type checking, symbol resolution, and validates semantic correctness
- **Code Generation and Linking**: produces machine code, then links external libraries and object files into a final executable

Understanding compilation errors

Having some insight into the compilation process is key to maintaining a sane build and interpreting compilation errors as they come up.

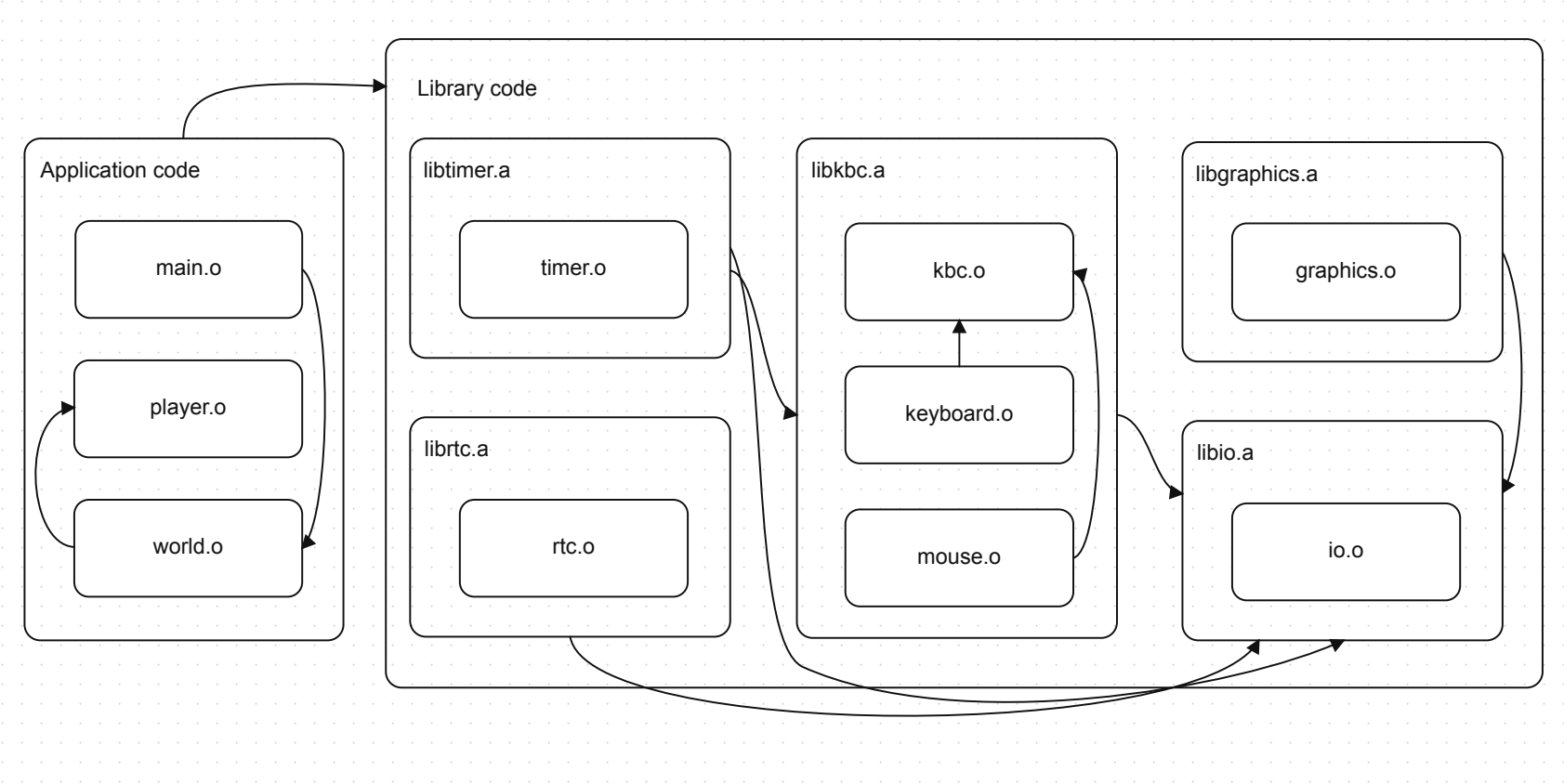
```
implicit declaration of function 'foo' is invalid
in C99
```

```
incompatible pointer types passing 'int *' to
parameter of type 'char *'
```

```
multiple definitions of 'global'
```

Our dependencies

Your final binary will be the result of linking the object files resulting from the application source code generation with the ones packaged inside static libraries.

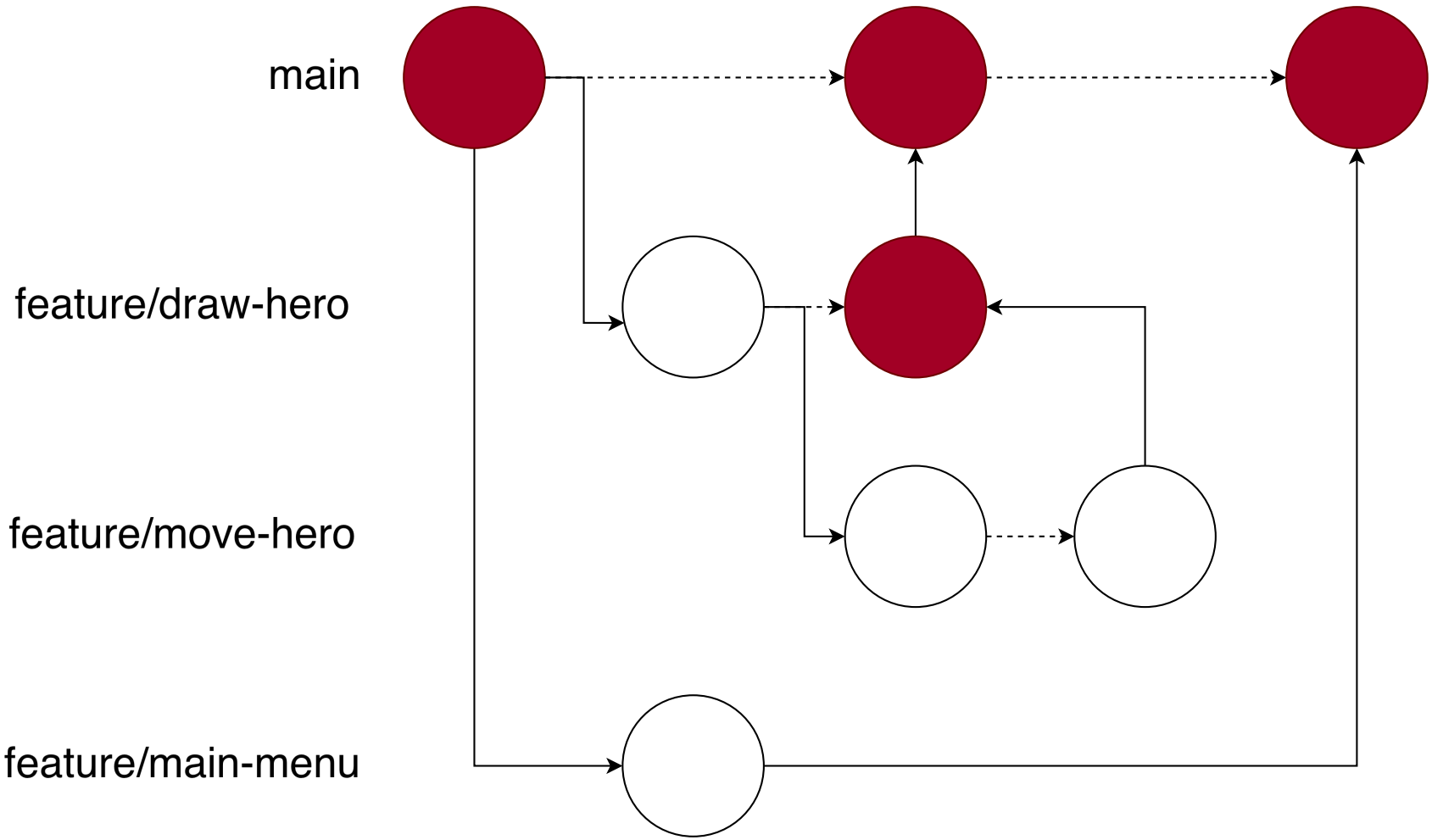


Git usage

Use Git as a true version control system designed for collaboration.

- Your main branch should be as updated as possible – rely on small feature branches, merge and delete them once done. This is usually called *trunk-based development*
- Take advantage of GitLab's collaboration tools, such as the kanban board and Merge Requests. Write feedback in the repository – you might need to trace back the *why* of some strange code in a few weeks, then you can *blame* your way through the offending commit

Git feature branches



Outline

A library's purpose

Designing our software as a composition of small, reusable functions leads to reduced coupling, easier refactoring and collaboration.

- A good function/module is self-documenting: simple, does one thing or is the composition of others that do just that
- A good function/module is well-typed and does no effect behind the developer's back

When designing large applications, one typically extracts code away from application sources once a bunch of non business logic starts to be overwhelming.

- In LCOM, as we anticipated that the device drivers developed during the labs would be used extensively in the project, we did that from the start
- We tried to keep our libraries small and non-opinionated

Exposing an API

In most programming languages, the usable artifact of a library, in the lens of the user – the application – is a set of well-established functions and types.

- Guaranteeing encapsulation of internal library logic is crucial, so that the application cannot (inadvertently or not) call code that may break the invariants of the API

```
// timer.c
unsigned long long counter = 0;
void timer_ih() { counter++; }
```

```
// app.c
extern unsigned long long counter;
counter = 42; // possibly breaks the monotonicity invariant
timer_ih();
printf("current counter: %llu\n", x);
```

Encapsulation in C

Managing the visibility of a component in C is typically achieved by using or omitting the `static` modifier for functions and variables.

```
// kbc.c
static uint8_t scancode[2]; // forbids external references
static bool ready = false;

static int read_obf(uint8_t* buf) {...}

void kbc_ih() {...}

int get_pressed_key() {
  if (!ready) return -1;
  return to_key(scancode);
}
```

```
// kbc.h
/** Returns ASCII code of the pressed key, or -1 if no new
    key was pressed. */
int get_pressed_key(void);
```

Header files

In C/C++, the public interface of a module should be *declared* in a header file. This includes relevant types, constants, and function declarations.

- When a source file *includes* a header, the preprocessor copies its contents into that translation unit. This allows the compiler to independently compile call sites before the corresponding implementation is linked
- In modern languages, this model is less necessary due to improved compiler infrastructure and abundant memory resources. Languages such as Rust, and modern C++ modules (introduced in C++20), avoid the traditional header-file mechanism

Opaque types

One of the principles of encapsulation is that the internal state of an "object" is private and can only be changed through a set of "safe" operations.

- An *opaque type* in C is a type (e.g. struct or union) whose internals are not known at usage time. Without an unknown static size, they must be used via pointers

```
// player.h
typedef struct player player; // alias to incomplete struct

/** Creates a new player with the given name. Returns NULL
    if allocation fails. */
player* player_new(char* name);

/** Applies shot damage to a player. Returns true if the
    player remains alive after the shot, false otherwise. */
bool player_register_shot(player* p);
```

Opaque types

```
// player.c
#include "player.h"

struct player {
  int hp;
  char name[64];
};

player* player_new(char* name) {
  player* new_player = malloc(sizeof player);
  if (new_player == NULL) {
    return NULL;
  }
  new_player->hp = 100;
  strncpy(new_player->name,name,63);
  new_player->name[63] = '\0';
  return new_player;
}

bool player_register_shot(player* p) {
  p->hp -= PLAYER_SHOT_PENALTY;
  return p->hp >= 0;
}
```

What should be in an API?

API design, for libraries and applications, is one of the most challenging challenges in software engineering.

- What should the caller be responsible for? Am I extracting too much or too little? Am I being too opinionated? Or is my library too liberal?
- Is this new added capability worth the maintenance effort?

It is totally fine and *expected* that you go back to the libraries sources – in LCOM, the labs – and add/remove functionality as you realize your project needs it, and *it makes sense in applications*. Iterate and recompile. Don't pollute your libraries with business logic, and don't pollute your application with low-level I/O.

Outline

Initialization

The first thing your program must do is initialize all needed HW devices and load the initial state into memory.

- Initializing HW devices is a matter of setting up interrupt handlers for the relevant IRQ lines, and making sure they are in a state where the PIC will be notified (e.g., the mouse must be in streaming mode, with data reporting enabled)
- Initializing the application state is highly domain-dependent and raises the question of how to model what your application shall be able to do

Establishing requirements

- What will the start screen look like? How should it display what my application can do?
- What must it know (i.e. keep in memory) at a time to be able to perform meaningfully?
- How will it convey new information to the user(s)?

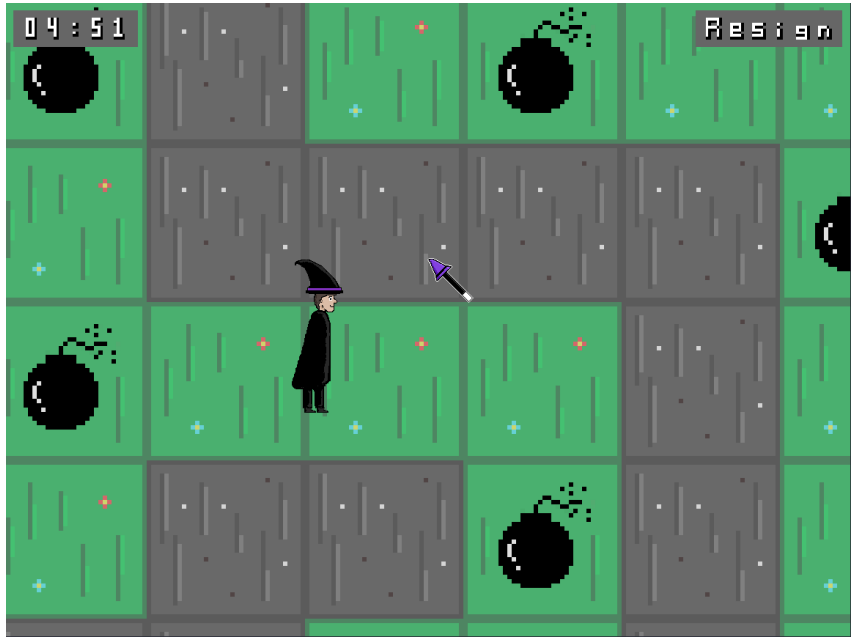


Figure 1: A Maze game is likely to have a moving map/hero, some form of obstacles and timeout.

Modelling state: a simple approach

Let your state S be a node in a state machine, and the ways to update it, yielding a new state S', the transitions.

```
enum state_t {
  START, INSTRUCTIONS, SETTINGS, IN_GAME, PAUSE, END
};

void handle_key_pressed(int key, state_t* state) {
  switch (*state) { // or model this as an array of
      fpointers
    case START: if (key == ESC_BREAKCODE) { *state = END; }
  }
}

int main() {
  state_t s = START;
  while (...) {
    driver_receive(&msg);
    if (is_kbd_int(&msg)) {
      kbd_ih();
      handle_key_pressed(get_key_pressed(), &s);
    }
  }
}
```

Modelling state: a simple approach

In this approach, if each state instance is a singleton, you can have a module per state, and hold information in static variables.

```
// start_state.c
static int selected_btn = 0;

void handle_mouse_pckt(mouse_pckt* pckt, state_t* s) {
  if (mouse_pckt.left_down) {
    switch (selected_btn) {
      case 0: *s = IN_GAME; break;
      case 1: *s = INSTRUCTIONS; break;
      default: *s = END; break;
    }
  } else {
    struct pos p = mouse_yield_pos(pckt); // in mouse.c
    selected_btn = select_btn(p);
  }
}
```

Raising the bar with tagged unions

You may leverage the type system to hold the relevant data for each state.

- A union allows you to interpret data in a memory location through different accessors, but does not yield the currently valid schema out of the box
- A *tagged union* is a struct containing a union and the currently valid schema
- This scales better, and is more testable than relying on ad-hoc state dispatchers

```
enum state_alternative {
  START, IN_GAME
};

typedef struct {
  enum state_alternative tag;
  union {
    struct {
      int selected_btn;
    } start;
    struct {
      unsigned hero_hp;
      unsigned seconds_left;
    } in_game;
  } data;
} state_t;
```

Implementing a UI

Using the VBE, you can map the memory of the graphics card to a virtual address and write to it at any point.

- You might be tempted to use some kind of `dirty` tracking in state updates and just repaint relevant entities for every changed state
- What happens if:
  - The user spams the keyboard and the mouse, and your (single-threaded) application, or the memory bus, can't keep up with so many writes?
  - You go and restore the previous entity at a certain position, but some autonomous enemy has meanwhile flown and covered it while that object pool was hidden by your hero?

Decoupling state and representation

In a highly performing, event-driven application, state and its representation are decoupled.

- You may have different ways to represent a state, for instance, dump it to a file or map to an array of pixels

The best way to keep CPU/memory usage mostly stable is to trigger a full screen repaint at a steady pace (FPS), even if between frame calculations there were more than 1 state transitions.

- How can this be implemented in our current *blocking interrupt loop* methodology?
- By which order should we "draw" our object pool?

A representable entity in memory

Using the likes of `xpm_load`, you interpret an `asset` serialized in a file in a certain format – here, XPM -, and load colors into memory, depending on the chosen graphical mode.

- This is a taxing operation: it involves reading the entire file and dynamically allocating memory for buffering the colors
- You must do this once and only once for each representable entity, and come up with a way to easily copy the interpreted colors into graphics memory once needed
- Once the entity is not needed anymore, *free* it; failure to do so in a large application is a memory leak and may result in significant performance degradation

An animated sprite

A sprite is a 2D pixel map that may be displayed anywhere in the screen.

- There shall be 1 sprite per entity type; if you have several wall and grass blocks in a map, there will be 2 sprites only drawn in a loop
- You may compose a sprite into some kind of higher abstraction to model the notion of movement and current position

Figure 2: Two sprites, animated.

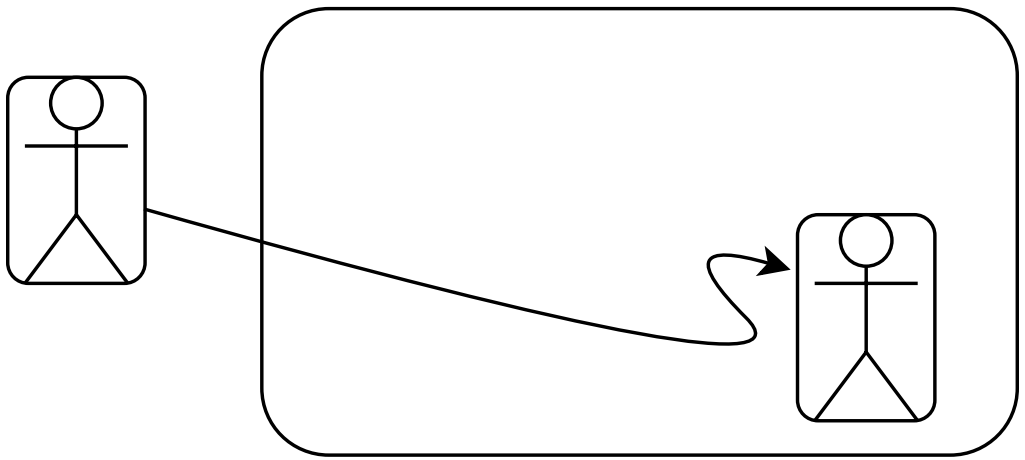
```
typedef struct {
  char* pixmap;
  uint16_t width, height;
} sprite_t;

typedef struct {
  sprite_t** sprites;
  size_t curr_sprite;
  uint16_t x, y;
  uint16_t frames_per_sprite;
  uint16_t frames_left;
} asprite_t;
```

Optimizing for performance

Screen paints are what's called *hot code* in your application – they are executed a lot of times and thus optimizing it will yield a considerable performance improvement

- A naive implementation might call some form of `draw_pixel` for each entity pixel in a loop
- Understanding the memory layout of screen buffers in the linear frame model hints at a better solution



Leveraging HW mechanisms

Even when your screen painting routine is efficient, if you draw a bunch of entities and thus take your time it is likely that the graphics card will trigger a buffer sync, usually called a *vertical screen refresh*

- If you are painting a new frame with updated state and the screen happens to be updated at the same time, you end up with visual artifacts known as *flickering* or screen tearing
- A simple solution: write to an "hidden" buffer in your application process, and turn your paint function into a bulk `memcpy` into the graphics card
- A better solution: use VBE's `0x07 (SET DISPLAY START)` to render into unused video memory and perform page flipping by switching the displayed buffer once a frame is complete

Building logic with composition

It is not hard to create helpers that based on existing primitives, i.e. entities with representable state, perform more complicated logic

- How to perform collision detection, knowing the position of two entites in the screen, and its colors?
- How to calculate the optimal path for a hero to travel themselves to a certain location, given obstacles and restrictions?
- How to model an infinite map?
- How to write text on the screen?

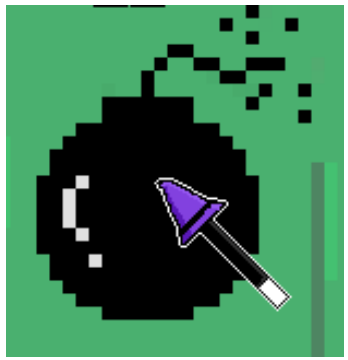


Figure 3: Collision detection.

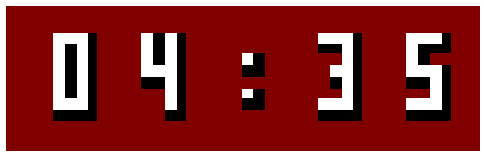


Figure 4: Text on the screen.

A Font module

```
typedef struct {
  sprite_t **tiles;
  sprite_t **zoom_tiles;
  uint16_t tile_size;
  uint32_t number_of_tiles;
} font_t;

font_t* (font_create)(uint32_t tile_size,
    char** xpm) {
  font_t *font = (font_t *) malloc(sizeof
      font_t);
  font->tile_size = tile_size;
  sprite_t* full_font = create_sprite(xpm, 0,
      0);
  font->number_of_tiles = (full_font->height
      / font->tile_size) * (full_font->width
      / font->tile_size);

  /* assemble letter tiles */
  ...

  destroy_sprite(full_font);
  return font;
}
```

Figure 5: A Start Menu.

A Button module

```
typedef enum {
  PURE_WHITE = 0xffffff,
  MILD_GREEN = 0x00008800
} button_color_t;

typedef struct {
  sprite_t *sp;
  sprite_t *hover_sp;
  char text[100];
  font_t *font;
  button_color_t back_color;
  button_color_t hover_frame_color;
} button_t;

void (button_draw)(button_t* b, bool hover){
  sprite_draw_no_checks(hover ? b->hover_sp :
       b->sp);

  draw_string(b->font, b->text, b->sp->x + (b
      ->sp->width - strlen(b->text)*b->font->
      tile_size) /2, b->sp->y + (b->sp->
      height - b->font->tile_size) /2);
}
```

Figure 6: A Start Menu.

More I/O

You may interact more with the real world to make your application more useful, depending on your intentions.

- You may have some kind of persistent state, e.g. save the date retrieved via RTC into disk once the game ends and have it recovered to show a session history
- You may *communicate* with other programs and machines
  - In the real world, the most notable use case is distributed systems – where a bunch of programs, possibly not in the same machine, work together, with clearly defined responsibilities, towards a shared goal
  - In LCOM, you will have an hint of what's this like by implementing *serial communication* via UART with another (emulated) machine

Settling on a protocol

When two or more computational actors communicate, they do so under a protocol

- A protocol is a contract, a set of structured messages they can understand – recall `lab1`

In syncing an application state, e.g. for collaborative exploration of a maze

- It is not feasible to periodically send the entire state/screen and do some kind of local merging — especially the first is a very taxing operation
- Solution: send state updates and conciliate them on the receiver; periodically send a more holistic view of the state in case messages are lost – probably not a problem in LCOM

Settling on a protocol: an example

For syncing a collaborative maze game

- `JOIN`: a request for a player to join a 2-player game
- `START`: a response to the join, starting the game
- `MOVE <x> <y>`: announcing a movement since the last message
- `SYNC <x> <y> <hp>`: announcing a partial view of their current state
- `FOUND <x> <y>`: announcing game ended; door on position

Some of these messages might occupy more than one byte, but port-mapped I/O transfers 1 byte at a time

- How to save payloads for later processing?

Message passing and queuing

In highly available systems exchanging lots of data, there is often a need to buffer messages before sending or processing them

- E.g. before sending, if the peer is unavailable, or after receiving, if no action can yet be done or busy with other work
- You have seen a simple example of this when reading the first byte of a 2-byte scancode in `lab3`

Solution: implement a queue data structure and pop from it under certain conditions

- How would you implement a generic queue data structure, say to be packaged in a `libcollections.a`, supporting all data types?

Analyzing your program with Doxygen+Graphviz