

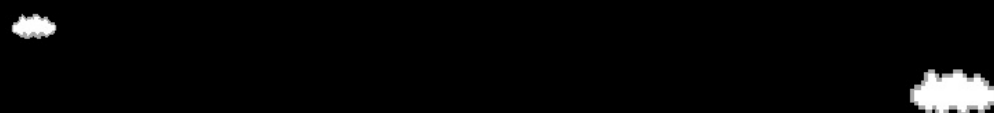
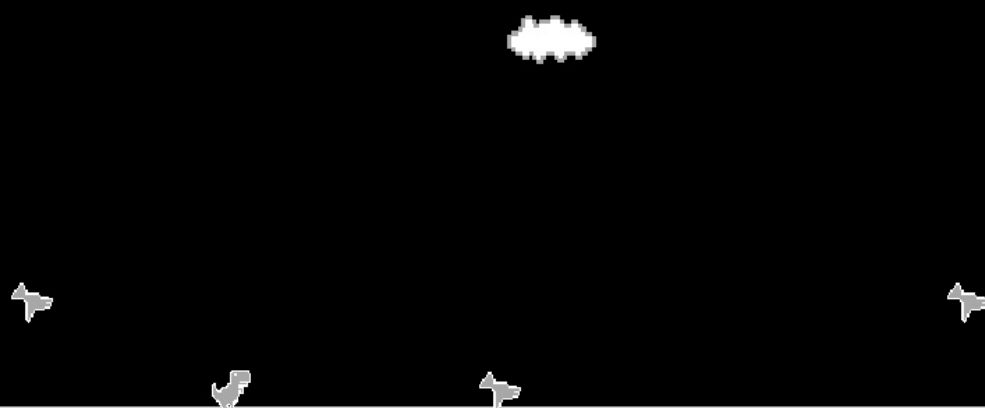
debugassaurus

debugassaurus is an endless runner inspired by the Google Chrome Dino game. The player must survive as long as possible while avoiding cacti and pterodactyls.

Our version adds play/pause functionality and a high score board that records the seed used for each score. Players can replay specific runs by using a hidden feature: in the main menu, pressing a number key reveals a seed input field.

Screenshots





PAUSED

ENTER TO RESUME

ESC TO QUIT



HIGHSCORES

1	TIAGO	2671	
	31-05-25	SEED-0	
2	TIAGO	1527	
	30-05-25	SEED-1748605815	
3	PLAYER2	1040	
	31-05-25	SEED-2203260705	
4	PLAYER1	377	
	31-05-25	SEED-123456789	
5	POKI	256	
	30-05-25	SEED-1748605780	

PRESS ENTER TO CONTINUE

267

GAME OVER

ENTER NAME

PRESS ENTER TO EXIT

debugassaurv5

play

highscores

exit

SEED 1234567890

Video link

To watch our demo video [click here](#)

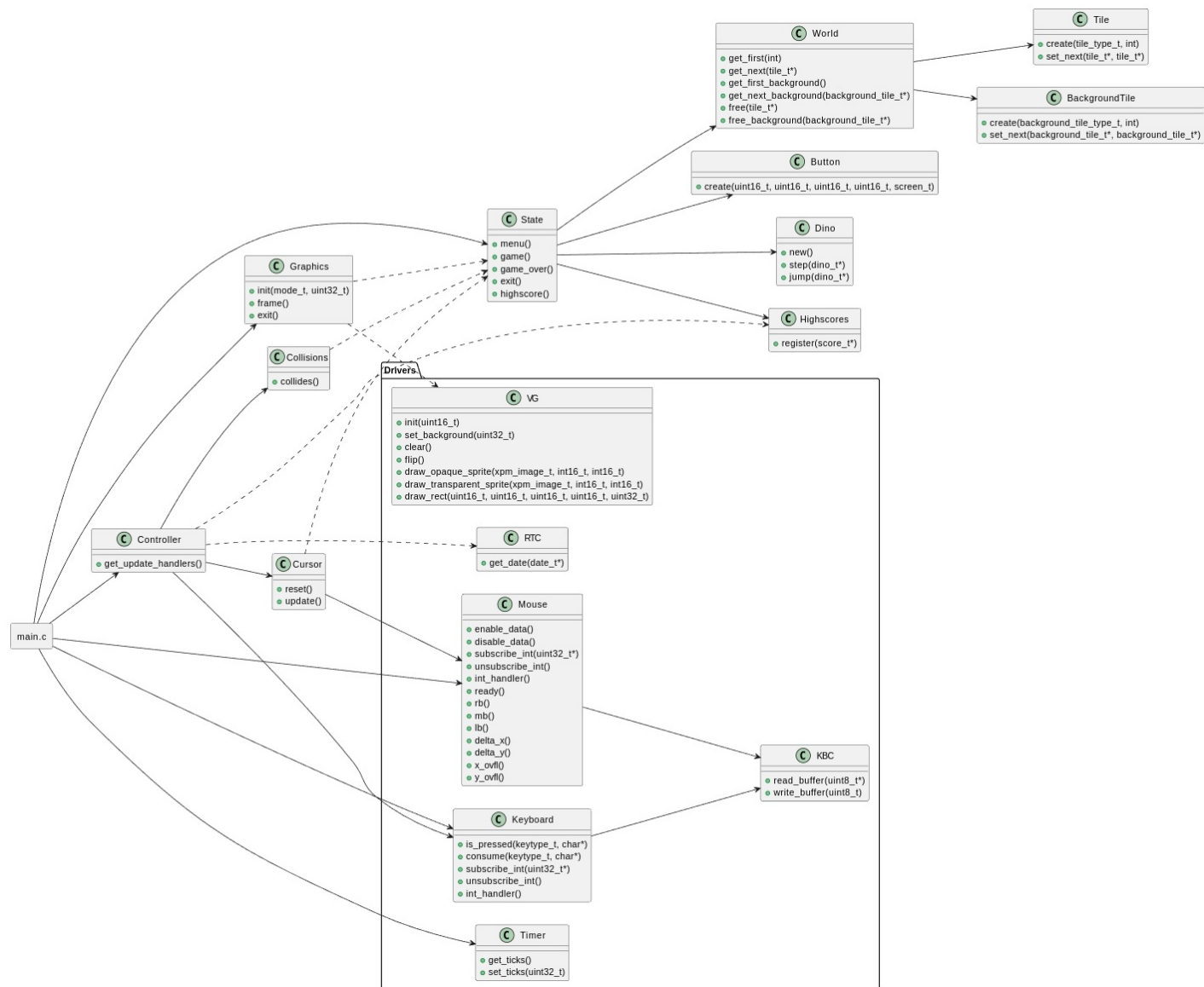
Running

```
cd src  
make clean && make && lcom_run proj
```

Controls

- `space` to jump. If `space` remains pressed the dino will keep jumping
- `esc` to pause

Project Structure



The project follows an MVC pattern, and is therefore divided in three main modules: state, graphics and controller. As such, most of the logic in the main loop is reduced to these simples lines:

```
state_menu();
update_handler* handlers = get_update_handlers();
while (state_get_screen() != EXIT) {
    // Handle interrupts
    // ...

    // Once every frame
    handlers[state_get_screen()]();
    graphics_frame();
}
```

State

debugassaurus has multiple screens, each represented by a `screen_t` enum value. Since each screen needs different state data - e.g., the main menu tracks buttons, the game screen tracks the world and dino - we use a union to save space. This requires care to avoid accessing invalid fields. Shared variables like the current screen and cursor position are stored outside the union.

```
struct state {
    // Common types
    screen_t screen;
    cursor_position_t cursor;
```

```

union {
    struct {
        dino_t* dino;
        tile_t* first_tile;
        background_tile_t* first_background;
        uint32_t seed;
        bool paused;
    } game;

    // ...
} data;
};

```

Externally, the state exposes getters and setters for its variables, along with functions to switch screens, such as `state_main_menu()` and `state_highscores()`.

World generation

debugassaurus presents an interesting challenge in terms of world representation, since it is an endless runner. The obstacles are represented as a linked list of `tile`s. While the `tile` class `tile` would be enough, the `world` class abstracts the world creation logic. This way, parts of the program that do not know about the state's internals can call `world_get_next(tile_t* tile)` to get the next tile in the world, and it will in fact always return a tile, as if all the tiles were pre-generated.

Graphics

...

Controller

The game is represented by a number of states (see above). The controller defines an array of update handlers, whose index corresponds to the `screen_t` enum value. These handlers are called once every frame.

Collisions

...

Highscores

...

Devices

Timer: Used for periodic rendering and keeping track of the time. Using the interrupt frequency, we were able to control the dino's speed and score.

```

// main.c

uint32_t timer_frequency = sys_hz();
uint32_t ticks_per_frame = timer_frequency / FPS;

// Setting up interrupt handlers

if (msg.m_notify.interrupts & timer_mask) {
    if (timer_get_ticks() == ticks_per_frame) {
        timer_set_ticks(0);
        // Frame logic
    }
}

```

Keyboard: Used for navigation in menus and player input during the game. A simple API was added that could be used either to check if a key was pressed or to consume a key if pressed (if the consume function is called twice, the second call will return false). Character keys are returned in a pointer passed as argument:

```
// Jump if space is pressed
if (keyboard_is_pressed(SPACE, NULL)) dino_jump(dino);

// Pause game if esc is pressed. Esc will likely remain pressed for many frames
// and we only want to detect it once. As such, we consume it.
if (keyboard_consume(ESC, NULL)) state_pause();

// Get a character input
char c;
if (keyboard_consume(CHARACTER, &c)) {
    // Do something with character
}
```

Mouse: Used for navigation in menus. Here's an excerpt from cursor.c:

```
void cursor_update() {
    if (!mouse_ready()) return;

    cursor_position_t position = state_get_cursor_position();

    position.x = cursor_update_within_range(position.x, mouse_delta_x(), vg_get_width() - 1);
    position.y = cursor_update_within_range(position.y, -mouse_delta_y(), vg_get_height() - 1);
    state_set_cursor_position(position);

    if (state_get_screen() == MENU && mouse_lb()) {
        for (/* every button */) {
            // Check if the cursor is within the button's area

            screen_t screen = button_get_screen(button);
            if (screen == MENU) state_menu();
            // ...
        }
    }
}
```

Video Card: Used to display the game. We used the indexed mode 0x105, with the reasoning that only writing one byte per pixel improves the performance of the game. Our implementation uses page flipping to avoid visual artifacts.

```
// Clear the second buffer
vg_clear();

// Other vg calls that draw the game
// ...

// Flip the buffers in between retraces
vg_flip();
```

Real Time Clock Used to get the date to be kept in a highscore. The simple API that is exposed is this:

```
typedef struct {
    uint8_t day;
    uint8_t month;
    uint8_t year;
} date_t;

int rtc_get_date(date_t *date);
```