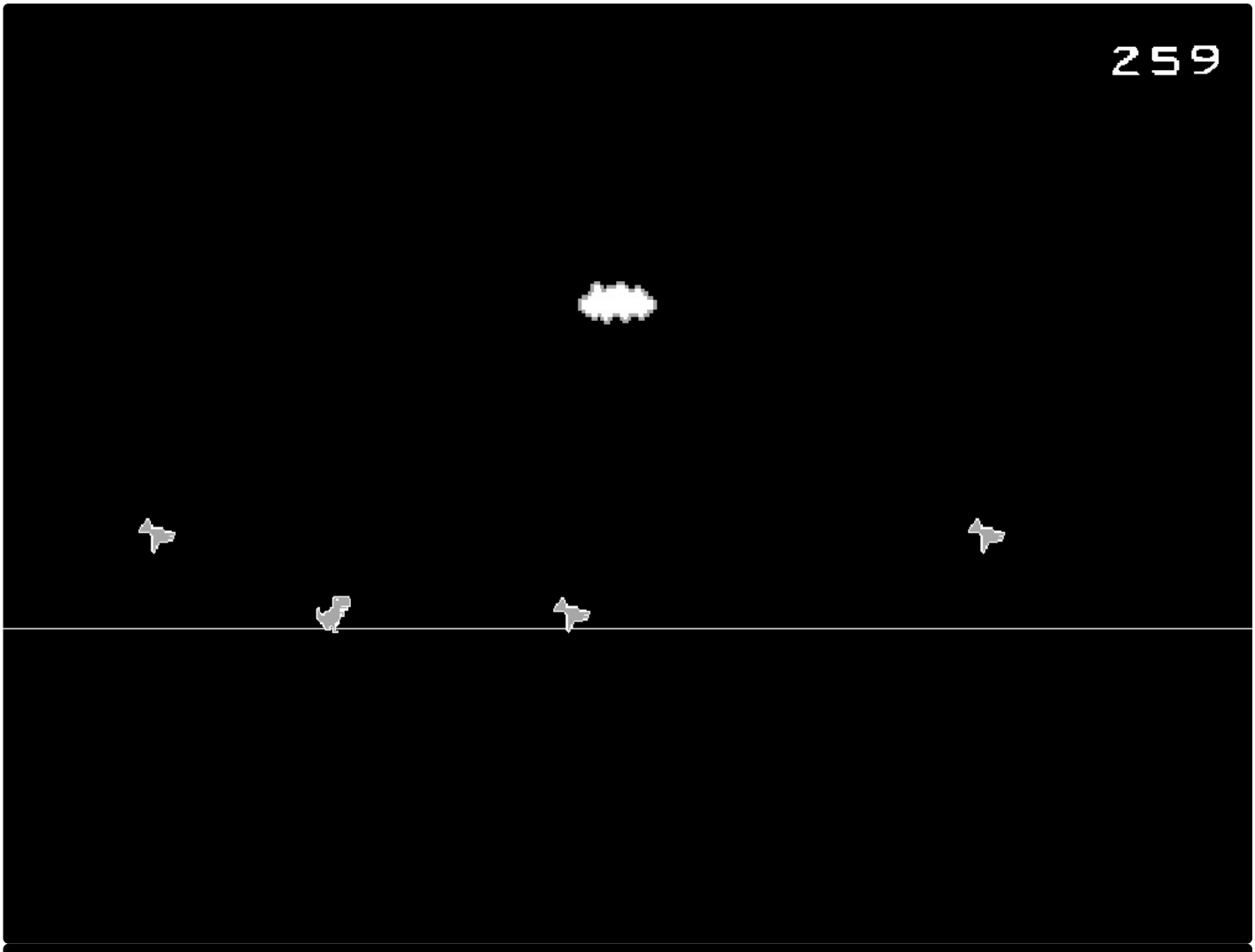
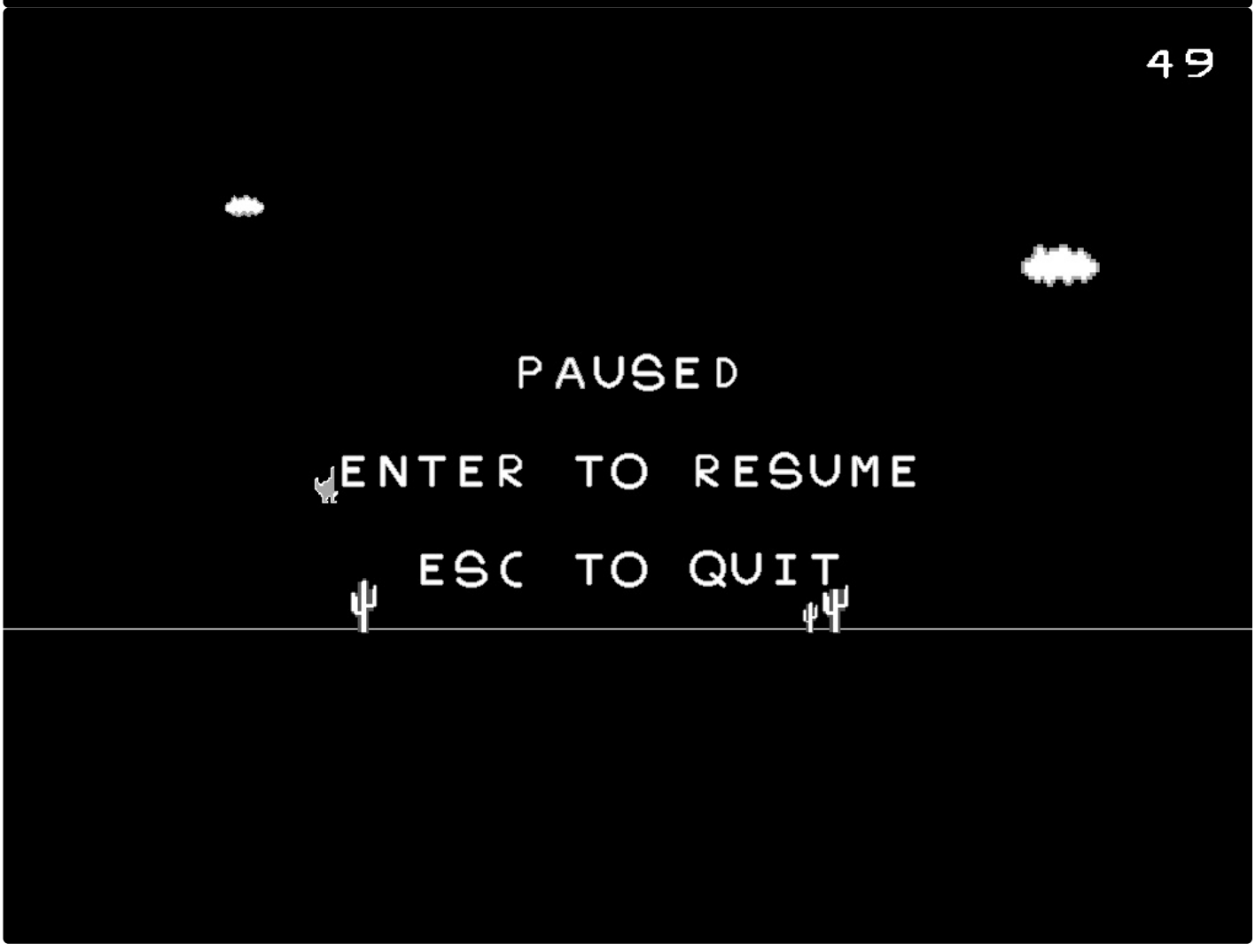
debugassaurus

debugassaurus is an endless runner inspired by the Google Chrome's 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


259


49
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

Video link

To watch our demo video click here

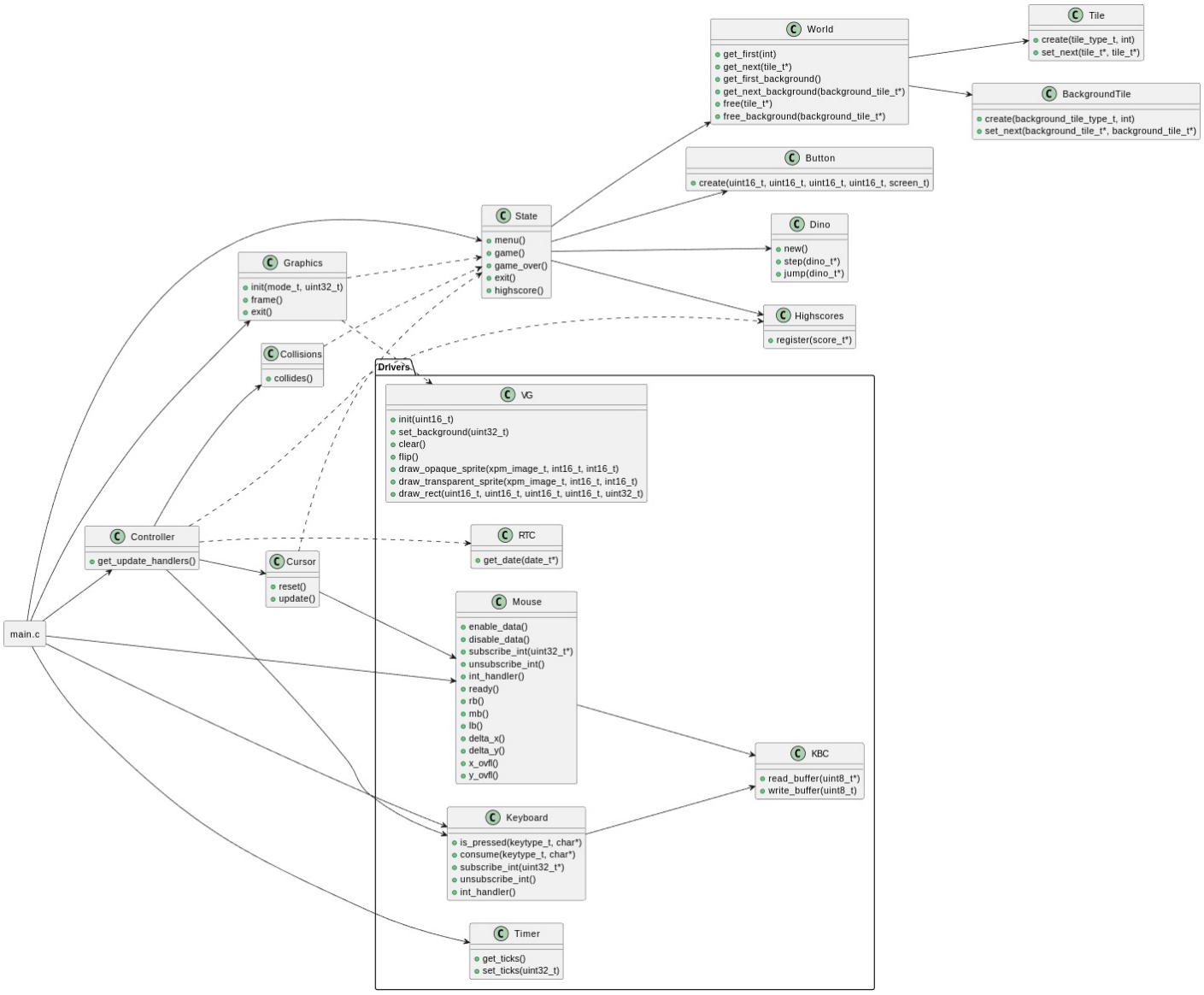
Running

```
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, setters, and screen switchers like `state_main_menu()`.

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

The graphics module handles all visual output in *debugassaurus*, translating game state into static frames with minimal overhead. Designed as the `View` in an MVC architecture, it is a *"dumb"* module - purely responsible for drawing sprites and text without any game logic.

On startup, `graphics_init()` loads every XPM sprite (dinosaur frames, obstacles, background elements, UI text, mouse cursor, etc.) and initializes the video mode. From then on, the module leverages a small set of render functions: `graphics_render_menu()`, `graphics_render_scene()`, `graphics_render_pause()`, `graphics_render_game_over()`, and `graphics_render_highscores()`. By computing only object positions and issuing draw calls, the graphics module ensures efficiency.

The `graphics_frame()` function is the single-entry point for rendering one complete frame. Called once per tick by `main.c`, it inspects the current `screen_t` in the state and dispatches to the appropriate render routine.

```
int graphics_frame() {
  screen_t screen = state_get_screen();

  if (screen == MENU) return graphics_render_menu();
  if (screen == GAME && state_is_paused()) return graphics_render_pause();
  if (screen == GAME) return graphics_render_scene();
  if (screen == GAME_OVER) return graphics_render_game_over();
  if (screen == HIGHSCORES) return graphics_render_highscores();
  return 0;
}
```

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

The collision detection system provides accurate collision detection between the dino and various obstacles in the game. It uses bounding boxes represented by the `bounding_box_t` structure, which defines rectangular collision using x and y coordinates and width and height dimensions. It also implements a distance-based bound using the `COLLISIONS_DISTANCE` constant in order to skip collision checks for obstacles too far either behind or ahead of the dino. This improves performance by reducing unnecessary calculations. The collision system integrates with the game controller by calling `collision_collides()` during the game updates in order to trigger game over conditions when the dino hits an obstacle.

Highscores

When the player loses, they can enter a name to associate with that run. If the score of the run is among the top 5 scores, it will be added to the correct position on the leaderboard, showing the name, score, date, and seed of the run.

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.

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.

Video Card: Used to display the game. We used the idexed 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.

```
vg_clear(); // Clear the second buffer
// Other vg calls that draw the game
vg_flip(); // Flip the buffers in between retraces
```

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);
```