Structured Synchronous Reactive Programming for Game Development
Case Study: On Rewriting Pingus from C++ to CÉU

Francisco Sant'Anna
Departamento de Informática e Ciência da Computação, UERJ
francisco@ime.uerj.br

Figure 1: Pingus gameplay.

ABSTRACT

TODO.

Keywords: TODO, TODO, TODO.

1 INTRODUCTION

Pingus is an open-source puzzle-platform video game based on Lemmings. The objective of the game is to guide a group of penguins through a number of obstacles towards a designated exit (Figure 1). Pingus is developed in standard object-oriented C++, the *lingua franca* of game development [12]. The codebase[1] is about 40.000 lines of code (LoCs), divided into the engine, level editor, auxiliary libraries, and the game logic itself.

According to Tim Sweeney (of Unreal Engine fame), about half the complexity in game development resides in *simulation* (aka *game logic*), but which accounts for only 10% of the CPU budget [21]. The high development costs contrasting with the low impact on performance appeals for alternatives with productivity in mind, especially considering that it is the game logic that varies the most between projects. Sweeney states that "will gladly sacrifice 10% of our performance for 10% higher productivity".

Object-oriented games rely on the *observer pattern* [12] to handle events from the environment (e.g., key presses and timers) and also as a notification mechanism between entities in the game logic. The observers are short-lived callbacks that have to execute as fast as possible to keep the game reactive to incoming events in real time. For this reason, callbacks cannot contain long-lasting locals and loops, which are elementary capabilities of classical structured programming [10, 17, 2]. In this sense, callbacks actually disrupt structured programming, becoming "our generation's `goto`".[2]

CÉU [19, 18] is a programming language that offers a concurrent and expressive alternative to C/C++ with the characteristics that follow:

- *Reactive*: code only executes in reactions to events.
- *Structured*: programs use structured control mechanisms, such as `await` (to suspend a line of execution), and `par` (to combine multiple lines of execution).
- *Synchronous*: reactions run atomically and to completion on each line of execution, i.e., there's no implicit preemption or real parallelism.

Structured reactive programming eliminates callbacks, letting programmers write code in direct and sequential style and recover from the inversion of control imposed by the observer pattern [10]. CÉU supports logical parallelism with a resource-efficient implementation in terms of memory and CPU usage [19]. The runtime is single threaded and does not rely on garbage collection.

In this work, we advocate structured synchronous reactive programming as an expressive and productive alternative for game logic development. We present a case study of rewriting Pingus from C++ to CÉU with the contributions as follows:

- Applying idiomatic code in CÉU as alternative solutions for six selected behaviors in the game logic.
- Presenting an in-depth qualitative analysis of the proposed solutions in comparison to the original implementations in C++.
- Identifying four recurrent control-flow patterns that likely apply to other games: *Finite State Machines*, *Continuation Passing*, *Dispatching Hierarchies*, and *Lifespan Hierarchies*.

A control-flow pattern is a recurring technique to describe dependencies and explicit orders between statements (or groups of statements) in a program. For instance, consider how a key press stimulus propagates through the game entities and also what happens with them if the stimulus causes the end of the game. CÉU supports primitives that help describing complex control-flow relationships in the game logic more concisely. The rewriting process consisted of identifying sets of callbacks implementing control flow in the game and translating them to CÉU using appropriate structured constructs. As an example, a double mouse click is characterized by a first click, followed by a maximum amount of time, followed by a second click. This behavior depends on different events (clicks and timers) which have to occur in a particular order. In C++, the implementation involves callbacks crossing reactions to successive events which manipulate state variables explicitly. We only present six cases distributed in the four patterns due to space constraints, but we actually documented in detail a total of eleven cases in six

[1] Pingus repository: `github.com/Pingus/pingus/`

[2] "Callbacks as our Generations' Go To Statement": `tirania.org/blog/archive/2013/Aug-15.html`

patterns (we omit the sixth pattern *Signaling Mechanisms* entirely). This work focuses on a qualitative analysis for the programming techniques that we applied during the rewriting process. Not all techniques result in reduction in LoCs (especially considering the verbose syntax of CÉU), but have other effects such as eliminating shared variables and dependencies between classes.

The rest of the paper is organized as follows: Section 2 gives an overview of the Pingus codebases in C++ and CÉU and describes our approach to identify and rewrite the control flow in the game. Section 3 discusses six case studies in detail which are categorized in four control-flow patterns. Section 4 discusses related work. Section 5 concludes the paper.

2 THE PINGUS CODEBASE

In Pingus, the game logic also accounts for almost half the size of the codebase: 18.173 from 39.362 LoCs (46%) spread across 272 files. However, about half of the game logic relates to non-reactive code, such as configurations and options, saved games and serialization, maps and levels descriptions, string formatting, collision detection, graph algorithms, etc. This part remains unchanged and relies on the integration between CÉU and C/C++. Therefore, we rewrote 9.186 LoCs spread across 126 files[3]. In order to only consider effective code in the analysis, we then removed all headers, declarations, trivial getters & setters, and other innocuous statements, resulting 4.135 dense LoCs spread across 70 implementation files originally written in C++[4]. We did the same with the implementation in CÉU, resulting in 3.697 dense LoCs[5]. Figure 2 summarizes the effective codebase in the two implementations.

Although the sections that follow compare the codebases a qualitatively, the lines with lower ratio numbers above correlate to the parts of the game logic that we consider more susceptible to structured reactive programming. For instance, the *Pingu* behavior (*ratio 0.80*) contains complex animations that are affected by timers, game rules, and user interaction. In contrast, the *Option screen* (*ratio 0.97*) is a simple UI grid with trivial mouse interactions.

As a general rewriting rule, we could identify control-flow behaviors in C++ by looking for class members with identifiers resembling verbs, statuses, and counters (e.g., `pressed`, `particle_thrown`, `mode`, and `delay_count`). Good chances are that variables with these "suspicious names" encode some form of control-flow progression that cross multiple callback invocations.

We employed *live code translation*, i.e., starting from the original codebase in C++, we reimplemented piece-by-piece without breaking the game compilation and execution. This was only possible given the seamless integration between CÉU and C/C++ [19]: the type systems are equivalent and the integration happens at the source code level. This enables trivial sharing of control (calling C/C++ from CÉU and vice-versa) and data (accessing C/C++ data from CÉU and vice-versa).

During the course of the rewriting process, we could identify more general control-flow patterns which likely apply to other games as well.

3 CONTROL-FLOW PATTERNS & CASE STUDIES

In this section, we select six representative game behaviors and describe in detail their implementations in C++ and CÉU. We also categorize these behaviors in five abstract control-flow patterns as follows:

1. *Finite State Machines*: Event occurrences map to transitions between states that trigger appropriate actions comprising the behavior of a game entity.
2. *Continuation Passing*: The completion of a long-lasting activity may carry a continuation, i.e., some action to execute next in the game.
3. *Dispatching Hierarchies*: Entities form a dispatching hierarchy in which a container that receives a stimulus automatically forwards it to its managed children.
4. *Lifespan Hierarchies*: Entities form a lifespan hierarchy in which a terminating container entity automatically destroys its managed children.

[3]Complete codebase: `github.com/an000/p/tree/master/cpp`

[4]C++ codebase: `github.com/an000/p/tree/master/all`

[5]CÉU codebase: `github.com/an000/p/tree/master/all`

3.1 Finite State Machines

Event occurrences map to transitions between states that trigger appropriate actions comprising the behavior of a game entity.

3.1.1 Case Study: Detecting Double-Clicks in the *Armageddon Button*

In Pingus, a double click in the *Armageddon button* at the bottom right of the screen literally explodes all pingus.[6]

Figure 3.a shows the C++ implementation for the class `ArmageddonButton` with methods for rendering the button and handling mouse and timer events. The code focus on the double click detection and hides unrelated parts with `<...>`. The methods `update` (ln. 14–26) and `on_click` (ln. 28–34) are examples of *short-lived callbacks*, which are pieces of code that execute atomically in reaction to external input events. The callback `on_click` reacts to mouse clicks detected by the base class `RectComponent` (ln. 2), while the callback `update` continuously reacts to the passage of time, frame by frame. Callbacks are short lived because they must react to input as fast as possible to let other callbacks execute, keeping the game with real-time responsiveness. The class first initializes the variable `pressed` to track the first click (ln. 3,32). It also initializes the variable `press_time` to count the time since the first click (ln. 4, 17). If another click occurs within 1 second, the class signals the double click to the application (ln. 30). Otherwise, the `pressed` and `press_time` state variables are reset (ln. 19–20). Figure 4 illustrates how we can model the double-click behavior in C++ as a state machine. The circles represent the state of the variables in the class, while the arrows represent the callbacks manipulating state. Note in the code how the accesses to the state variables are spread across the entire class. For instance, the distance between the initialization of `pressed` (ln. 3) and the last access to it (ln. 32) is over 40 lines in the original file. Arguably, this dispersion of code across methods makes the understanding and maintenance of the double-click behavior more difficult. Also, even though the state variables are private, unrelated methods such as `draw`, which is defined in middle of the class (ln. 10–12), can potentially access them.

CÉU provides structured constructs to deal with events, aiming to eradicate explicit manipulation of state variables for control-flow purposes. In Figure 3.b, the loop detection (ln. 4–10) awaits the first click (ln. 5) and then, while watching 1 second (ln. 6–9), awaits the second click (ln. 7). If the second click occurs within 1 second, the `break` terminates the loop (ln. 8) and the `emit` signals the double click to the application (ln. 12). Otherwise, the `watching` block as a whole aborts after 1 second and the loop restarts, falling back to the first click `await` (ln. 5). Double click detection in CÉU doesn't require state variables and is entirely self-contained in the `loop` body (ln. 4–10). Also, these 7 lines of code *only* detect the double click, leaving the actual effect to happen outside the loop (ln. 12).

```
Path             Ceu     C++   Ceu/C++    Descritpion
------------     ----    ----    ----     --------------------------------------------
game/            2064    2268    0.91     the main gameplay
  ./              710     679    1.05         main functionality
  objs/           470     478    0.98         world objects (tiles, traps, etc)
  pingu/          884    1111    0.80         pingu behaviors
    ./            343     458    0.75             main functionality
    actions/      541     653    0.83             pingu actions (bomber, climber, etc)
worldmap/         468     493    0.95     campaign worldmap
screens/         1109    1328    0.84     menus and screens
    option/       347     357    0.97         option menu
    others/       762     971    0.78         other menus and screens
misc/              56      46    1.22     miscellaneous functionality
                 ----    ----    ----
                 3697    4135    0.89
```

Figure 2: The Pingus codebase directory tree.

3.1.2 Case Study: The *Bomber Action* Animation Sequence

In Pingus, the player may assign actions to specific pingus, as illustrated in Figure **??**. The *Bomber action* explodes the clicked pingu, throwing particles around and also destroying the terrain under its radius.[7] We can model the explosion animation with a sequential state machine (Figure 6) with actions associated to specific frames as follows[8]:

1. 0th frame: plays a "Oh no!" sound.
2. 10th frame: plays a "Bomb!" sound.
3. 13th frame: throws particles, destroys the terrain, and shows an explosion sprite.
4. Game tick: hides the explosion sprite.
5. Last frame: kills the pingu.

Figure 7 compares the implementations in C++ and CÉU.

In C++, the class `Bomber` defines the callbacks `draw` and `update` to manage the state machine described above. The class first defines one state variable for each action to perform (ln. 4–7). The "Oh no!" sound plays as soon as the object starts in *state-1* (ln. 11). The `update` callback (ln. 14–38) first updates the pingu animation and movement on every frame regardless of its current state (ln. 15–16). When the animation reaches the 10th frame, it plays the "Bomb!" sound and switches to *state-2* (ln. 18–22). The state variable `sound_played` is required because the sprite frame doesn't necessarily advance on every `update` invocation (e.g., `update` may execute twice during the 10th frame). The same reasoning and technique applies to *state-3* (ln. 24–32 and 43–44). The explosion sprite appears in a single frame in *state-4* (ln. 45). Finally, the pingu dies after the animation frames terminate (ln. 34–35). Note that a single numeric state variable suffices to track the states, but the original authors probably chose to encode each state in an independent boolean variable to rearrange and experiment with them during development. Still, due to the short-lived nature of callbacks, state variables are unavoidable and are actually the essence of object-oriented programming (i.e., *methods* + *mutable state*). Like double click detection in C++, note that the state machine is encoded across 3 different methods, each intermixing code with unrelated functionality.

The equivalent code for the *Bomber action* in CÉU doesn't require state variables and reflects the sequential state machine implicitly, using `await` statements in direct style to separate the actions. The `Bomber` is a `code/await` abstraction of CÉU, which is similar to a coroutine or fiber [2]: a subroutine that retains runtime state, such as local variables and the program counter, across reactions to events (i.e., across `await` statements). The pingu movement and sprite animation are isolated in two other `code/await` abstractions and execute in separate through the `spawn` primitive (ln. 4–5). The event `game.dt` (ln. 12,16,24) is analogous to the `update` callback of C++ and occurs on every frame. The code tracks the animation aliveness (ln. 7–27) and, on termination, performs the last bomber action (ln. 30). As soon as the animation starts, the code performs the first action (ln. 9). The intermediate actions are performed when the corresponding conditions occur (ln. 12,16,24). The `do-end` block (ln. 19–25), restricts the lifespan of the single-frame explosion sprite (ln. 21): after the next game tick (ln. 24), the block terminates and automatically destroys the spawned abstraction (removing it from the screen). In contrast with the implementation in C++, all actions happen in a contiguous chunk of code (ln. 5–30) which handles no extra functionality.

3.1.3 Summary & Uses in Pingus

The structured constructs of CÉU provide some advantages in comparison to explicit state machines:

- They encode all states with direct sequential code, eliminating shared state variables.
- They handle all states (and only them) in the same contiguous block, improving code encapsulation.

Object-oriented games also adopt the *state pattern* to model state machines, with subclasses describing each possible state [12]. However, this approach is not fundamentally different from using `switch` or `if` branches for each possible state.

Pingus supports 16 actions in the game. As Figure 8 shows, 5 of them implement at least one state machine and are considerable smaller in CÉU in terms of LoCs when eliminating the state variables. Considering the other 11 actions, the reduction in LoCs is negligible. This asymmetry in the implementation of actions illustrates the gains in expressiveness when describing state machines in direct style.

Among the 65 implementation files in CÉU, we found 29 cases in 25 files using structured mechanisms to substitute states machines.

[6] Double click animation: github.com/an000/p/blob/master/README.md#1

[7] Bomber action animation: github.com/an000/p/blob/master/README.md#2

[8] State machine animation: github.com/an000/p/blob/master/README.md#3

```
ArmageddonButton::ArmageddonButton(<...>):
    RectComponent(<...>),
    pressed(false); // button initially not pressed
    press_time(0);    // how long since 1st click?
    <...>
{
    <...>
}

void ArmageddonButton::draw (<...>) {
    <...>
}

void ArmageddonButton::update (float delta) {
    <...>
    if (pressed) {
        press_time += delta;
        if (press_time > 1.0f) {
            pressed = false; // give up, 1st click
            press_time = 0;  // was too long ago
        }
    } else {
        <...>
        press_time = 0;
    }
}

void ArmageddonButton::on_click (<...>) {
    if (pressed) {
        send_armageddon_event();
    } else {
        pressed = true;
    }
}
```

[a] Implementation in C++

```
do
    var RectComponent c = <...>;
    <...>
    loop do
        await c.component.on_click;
        watching 1s do
            await c.component.on_click;
            break;
        end
    end
    <...>
    emit game.go_armageddon;
end
.
```

[b] Implementation in CÉU

Figure 3: Detecting double-clicks in the *Armageddon button*.

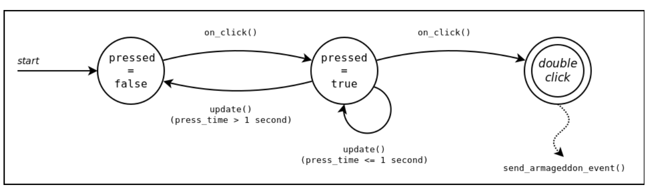

Figure 4: State machine for detecting double-clicks in the *Armageddon button*.

Figure 5: Assigning the *Bomber action* to a pingu.

They manifest as `await` statements in sequence or in aborting constructs such as `par/or` and `watching`.

3.2 Continuation Passing

The completion of a long-lasting activity may carry a continuation, i.e., some action to execute next in the game.

3.2.1 Case Study: Advancing Pages in the *Story screen*

The clickable *blue dots* in the campaign world map transit to ambience story screens[9]. A story is composed of multiple pages and, inside each page, the words of the story appear incrementally over time. A first click in the button `>>>` fast forwards the words to show the full page. A second click advances to the next page, until the story terminates. If the page completes before a click (due to the time elapsing), a first click advances to the next page. Figure 10 compares the implementations in C++ and CÉU.

In C++, the class `StoryScreenComponent` implements the method `next_text`, which is a callback for clicks in `>>>`. The variable 'pages' (ln. 4–5, 24–26) is a vector holding each page, but which also encodes *continuations* for the story progress: each call to `next_text` that advances the story (ln. 23–32) removes the current page (ln. 24) and sets the next action to perform (i.e., "display a new page") in the variable `current_page` (ln. 26). Figure 9 illustrates the continuation mechanism to advance pages and also a state machine for fast forwarding words (inside the dashed rectangle). The state variable `displayed` (ln. 6,15,20,21,27) switches between the behaviors "advancing text" and "advancing pages", which are both handled intermixed inside the method `next_text`.

[9]Story screen animation: `github.com/an000/p/blob/master/README.md#4`

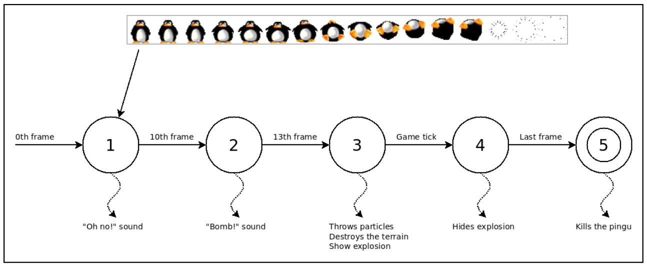

Figure 6: State machine for the *Bomber animation* sequence.

The code in CÉU uses the internal event `next_text`, which is emitted from clicks in `>>>`. The sequential navigation from page to page uses a loop in direct style (ln. 6–15) instead of explicit state variables for the continuation and state machine. While the text advances in an inner loop (hidden in ln. 9), we watch the `next_text` event that fast forwards it. The loop may also eventually terminate with the time elapsing normally. This way, we do not need a variable (such as 'displayed' in C++) to switch between the states "advancing text" and "advancing pages". The `par/or` makes the page advance logic to execute in parallel with the redrawing code (ln. 13). Whenever the page advances, the redrawing code is automatically aborted (due to the `or` modifier). The `await next_text` in sequence (ln. 11) is the condition to advance to the next page. Note that, unlike the implementation in C++, the "advancing text" behavior is not intermixed with the "advancing pages" behavior, instead, it is encapsulated inside the inner loop nested with a deeper indentation (ln. 9).

3.2.2 Summary & Uses in Pingus

The direct style of CÉU has some advantages in comparison to the continuation-passing style:

- It uses structured control flow (i.e., sequences and loops) instead of explicit data structures (e.g., stacks) or continuation variables.
- The activities are decoupled and do not hold references to one another.
- A single parent class describes the flow between the activities in a self-contained block of code.

Continuation passing typically controls the overall structure of the game, such as screen transitions in menus and level progressions. CÉU uses the direct style techniques in five cases involving screen transitions: the main menu, the level menu, the level set menu, the world map loop, and the gameplay loop. It also uses the same technique for the loop that switches the pingu actions during gameplay.

3.3 Dispatching Hierarchies

Entities form a dispatching hierarchy in which a container that receives a stimulus automatically forwards it to its managed children.

3.3.1 Case Study: *Bomber Action* `draw` and `update` Dispatching

TODO

Figure 12 compares the implementations in C++ and CÉU.

In C++, the class `Bomber` declares a `sprite` member to handle its animation frames (Figure 6). The `Sprite` class is part of the game engine and knows how to update and render itself. However, the `Bomber` still has to respond to `update` and `draw` requests from the game and forward them to the sprite (ln. 11–13 and 15–18). To understand how the `update` callback flows from the original environment stimulus from the game down to the sprite, we need to follow a long chain of 7 method dispatches (Figure 11):

1. `ScreenManager::display` in the main game loop calls `update`.
2. `ScreenManager::update` calls `last_screen->update` for the active game screen (a `GameSession` instance, considering the `Bomber`).
3. `GameSession::update` calls `world->update`.
4. `World::update` calls `obj->update` for each object in the world.
5. `PinguHolder::update` calls `pingu->update` for each pingu alive.
6. `Pingu::update` calls `action->update` for the active pingu action.
7. `Bomber::update` calls `sprite.update`. `Sprite::update` finally updates the animation frame.

Each dispatching step in the chain is necessary considering the game architecture:

- With a single assignment to `last_screen`, we can easily deactivate the current screen and redirect all dispatches to a new screen.
- The `World` class manages and dispatches events to all game entities, such as all pingus and traps, with the common interface `WorldObj`.
- Since it is common to iterate only over the pingus (vs. all world objects), the container `PinguHolder` manages all pingus.
- Since a single pingu can change between actions during lifetime, the `action` member decouples them with another level of indirection.
- Sprites are part of the game engine and are reusable everywhere (e.g., UI buttons, world objects, etc.), so it is also convenient to decouple them from actions.

The `draw` callback flows through the same dispatching hierarchy until reaching the `Sprite` class.

In CÉU, the `Bomber` action spawns a `Sprite` animation instance on its body. The `Sprite` instance (ln. 3) can react directly to external `dt` and `redraw` events (which are analogous to `update` and `redraw` callbacks, respectively), bypassing the program hierarchy entirely. While and *only while* the bomber abstraction is alive, the sprite animation is also alive. The radical decoupling between the program hierarchy and reactions to events eliminates dispatching chains entirely.

3.3.2 Summary & Uses in Pingus

Passive entities subjected to hierarchies require a dispatching architecture that makes the reasoning about the program harder:

- The full dispatching chain may go through dozens of files.
- The dispatching chain may interleave between classes specific to the game and also classes from the game engine (possibly third-party classes).

```
Bomber::Bomber (Pingu* p) :
  <...>
  sprite(<...>),           // bomber sprite
  sound_played(false),     // tracks state 2
  particle_thrown(false), // tracks state 3
  colmap_exploded(false), // tracks state 3
  gfx_exploded(false)      // tracks state 4
{
  <...>
  // 1. plays a "Oh no!" sound.
  play_sound("ohno");
}

void Bomber::update () {
  sprite.update();
  <...>   // pingu movement

  // 2. plays a "Bomb!" sound.
  if (sprite.frame()==10 && !sound_played) {
    sound_played = true;
    play_sound("plop");
  }

  // 3. particles, terrain, explosion sprite
  if (sprite.frame()==13 && !particle_thrown) {
    particle_thrown = true;
    get_world()->get_particles()->add(...);
  }
  if (sprite.frame()==13 && !colmap_exploded) {
    colmap_exploded = true;
    get_world()->remove(bomber_radius, <...>);
  }

  // 5. kills the Pingu
  if (sprite.is_finished ()) {
    pingu->set_status(PS_DEAD);
  }
}

void Bomber::draw (SceneContext& gc) {
  // 3. particles, terrain, explosion sprite
  // 4. tick: hides the explosion sprite
  if (sprite.frame()==13 && !gfx_exploded) {
    gfx_exploded = true;
    gc.color().draw(explo_surf, <...>);
  }
  gc.color().draw(sprite, pingu->get_pos());
}
```

[a] Implementation in C++

```
code/await Bomber (void) -> ActionName
do
  <...>
  spawn Mover();  // movement in the background
  var Sprite s = spawn Sprite(<...>);
                 // animation in the background
  watching s do
    // 1. plays a "Oh no!" sound.
    {play_sound("ohno")};

    // 2. plays a "Bomb!" sound.
    await game.dt until s.sprite.frame == 10;
    {play_sound("plop")};

    // 3. particles, terrain, explosion sprite
    await game.dt until s.sprite.frame == 13;
    spawn PinguParticles(<...>) in particles;
    call Game_Remove({&bomber_radius}, <...>);
    do
      <...>
      spawn Sprite(<...>);      // explosion

      // 4. tick: hides the explosion sprite
      await game.dt;
    end
    await FOREVER;
  end

  // 5. kills the pingu
  escape DEAD;
end
```

[b] Implementation in CÉU

Figure 7: The *Bomber action* sequence.

Action	Ceu	C++	Explicit State
Bomber	23	50	4 state variables
Bridger	75	100	2 state variables
Drown	6	15	1 state variable
Exiter	7	22	2 state variables
Splashed	6	19	2 state variables

Figure 8: Pingus actions in CÉU and C++.

In C++, the update subsystem touches 39 files with around 100 lines of code just to forward `update` methods through the dispatching hierarchy. For the drawing subsystem, 50 files with around 300 lines of code. The implementation in C++ also relies on dispatching hierarchy for `resize` callbacks, touching 12 files with around 100 lines of code. Most of this code is eliminated in CÉU since abstractions can react directly to the environment, not depending on hierarchies spread across multiple files.

Note that dispatching hierarchies cross game engine code, suggesting that most games use this control-flow pattern heavily. In the case of the Pingus engine, we rewrote 9 files from C++ to CÉU, reducing them from 515 to 173 LoCs, mostly due to dispatching code removal.

3.4 Lifespan Hierarchies

Entities form a lifespan hierarchy in which a terminating container entity automatically destroys its managed children.

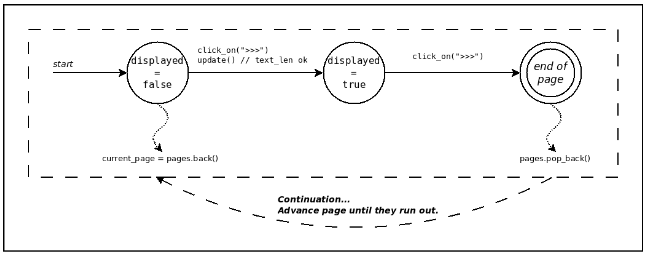

Figure 9: State machine for advancing pages in the *Story screen*.

3.4.1 Case Study: Game UI Widgets

Figure 13 shows the game UI widgets holding the buttons, pingus counter, and small map which coexist with the game screen during its whole lifespan. Figure **??** compares the implementations in C++ and CÉU.

In C++, the widgets are created in the constructor of the class `GameSession` (ln. 5–7), added to a UI container (ln. 9–11), and are never removed since they must always be visible. Arguably, to better express the intent of making them coexist with the game screen, the widgets could be declared as top-level automatic (non-dynamic) members. However, the class uses a container to automate `draw` and `update` dispatching to the widgets, as discussed in Section 3.3. In turn, the container method `add` expects dynamically allocated children only because they are automatically deallocated inside the container destructor. The dynamic nature of containers in C++ demand extra caution from programmers:

- When containers are part of a dispatching chain, it gets even harder to know which objects are dispatched: one has to "simulate" the program execution and track calls to `add` and `remove`.

- For objects with dynamic lifespan, calls to `add` must always have matching calls to `remove`: missing calls to `remove` lead to memory and CPU leaks (to be discussed as the *lapsed listener* problem in Section 3.4.2).

In Cu, entities that coexist just have to be created in the same lexical block. Since abstractions can react independently, they do not require a dispatching container. Lexical lifespan never requires containers, allocation and deallocation, or explicit references. In addition, all required memory is known at compile time, similarly to stack-allocated local variables. The *Bomber action* of Section 3.1.2 also relies on lexical scope to delimit the lifespan of the explosion sprite to a single frame.

3.4.2 Case Study: Managing the Pingus Lifecycle

A pingu is a dynamic entity created periodically and destroyed under certain conditions, such as falling from a high altitude[10]. Figure 15 compares the implementations in C++ and CÉU.

In C++, the class `PinguHolder` is a container that holds all pingus alive. The method `PinguHolder::create_pingu` (ln. 1–6) is called periodically to create a new `Pingu` and add it to the `pingus` collection (ln. 3–4). The method `PinguHolder::update` (ln. 8–18) checks the state of all pingus on every frame, removing those with the dead status (ln. 12–14). Entities with dynamic lifespan in C++ require explicit `add` and `remove` calls associated to a container (ln. 4,13). Without the `erase` call above, a dead pingu would remain in the collection with updates on every frame (ln. 11). Since the `redraw` behavior for a dead pingu is innocuous, the death could go unnoticed but the program would keep consuming memory and CPU time. This problem is known as the *lapsed listener* [12] and also occurs in languages with garbage collection: A container typically holds a strong reference to a child (sometimes the only reference to it), and the runtime cannot magically detect it as garbage.

CÉU supports `pool` declarations to hold dynamic abstraction instances. Additionally, the `spawn` statement supports a pool identifier to associate the new instance with a pool. The game screen spawns a new `Pingu` on every invocation of `Pingu_Spawn`. The `spawn` statement (ln. 6) specifies the pool declared at the top-level block of the game screen (ln. 3). In this case, the lifespan of the new instances follows the scope of the pool (ln. 1–9) instead of the enclosing scope of the `spawn` statement (ln. 4–7). Since pools are also subject to lexical scope, the lifespan of all dynamically allocated pingus is constrained to the game screen. Lexical scopes handle memory and event dispatching automatically for static instances and also for pools. However, the lifespan of a dynamic instance does not necessarily have to match the lifespan of its associated pool (Figure 16). In CÉU, when the execution block of a dynamic instance terminates, which characterizes its *natural termination*, the instance is automatically removed from its pool. Therefore, dynamic instances do not require any extra bookkeeping related to containers or explicit deallocation. To remove a pingu from the game in CÉU, we just need to terminate its execution block according to the appropriate conditions: The `escape` statement (ln. 17) aborts the execution block of the `Pingu` instance, removing it from its associated pool automatically. Hence, a dynamic instance that terminates naturally leaves no traces in the program.

3.4.3 Summary & Uses in Pingus

Lexical lifespan for static instances and natural termination for dynamic instances provide some advantages in comparison to lifespan hierarchies through containers:

- Lexical scope makes an abstraction lifespan explicit in the source code.

- The memory for static instances is known at compile time.

- Natural termination makes an instance innocuous and, hence, susceptible to immediate reclamation.

- Abstraction instances (static or dynamic) never require explicit manipulation of pointers/references.

All entities in a game have an associated lifespan.

The implementation in CÉU has over 200 static instantiations spread across all 65 files. For dynamic entities, it defines 23 pools in 10 files, with almost 96 instantiations across 37 files. Pools are used to hold explosion particles, levels and level sets from files, gameplay & worldmap objects, and UI widgets.

4 RELATED WORK

The control-flow patterns closely relate to the *GoF* behavioral patterns [8], which some previous work discuss in the context of video game development [12, 16, 3]. The original game in C++ uses variations of the *state* (Sections 3.1 and 3.2), *visitor* (Sections 3.3 and 3.4), and *observer* (Section **??** and to handle input in general) patterns as implementation details to achieve the desired higher-level control-flow patterns. CÉU overcomes the need of behavioral patterns with a semantics that supports structured control-flow mechanisms and event-based communication via broadcast. As an example, the *state pattern* for the bomber animation in Section 3.1 becomes a series of blocks separated by `await` statements.

A number of domain-specific languages, frameworks, and techniques have been proposed for particular subsystems of the game logic, such as animations [13, 6, 14, 15], game state and screen

[10] Death of pingus animation: `github.com/an000/p/blob/master/README.md#5`

```
StoryScreenComponent::StoryScreenComponent (<...>) :
    <...>
{
    pages        = <...>; // vector with loaded pages
    current_page = pages.back(); // first loaded page
    displayed    = false; // if current is complete
    <...>
}

<...>   // draw page over time

void StoryScreenComponent::update (<...>) {
    <...>
    if (<all-words-appearing>) {
        displayed = true;
    }
}

void StoryScreenComponent::next_text() {
    if (!displayed) {
        displayed = true;
        <...>     // remove current page
    } else {
        pages.pop_back();
        if (!pages.empty()) { // next page
            current_page = pages.back();
            displayed    = false;
            <...>
        } else {
            <...> // terminates the story screen
        }
    }
}
```

[a] Implementation in C++

```
code/await Story (void) -> bool do
    <...>
    event void next_text; // clicks in >>>

    { pages = <...>; } // same as in C++
    loop i in [0 <- {pages.size()}[ do
        par/or do
            watching next_text do
                <...> // advance text
            end
            await next_text;
        with
            <...> // redraw _pages[i]
        end
    end
end

.
```

[b] Implementation in CÉU

Figure 10: Advancing pages in the *Story Screen*.

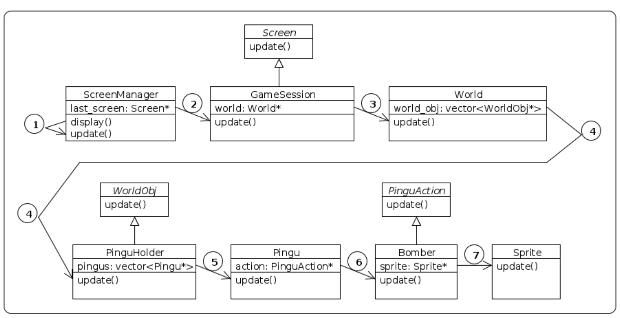

Figure 11: Dispatching chain for `update`.

progression [22, 11], and behavior and AI modeling [9, 1] In Pingus, we employed CÉU at the core of the game for event dispatching (Section 3.3) and memory management of entities (Section 3.4), eliminating parts of the original game engine. We also implemented all entity animations and behaviors (Section 3.1), screen transitions (Section 3.2), and intermodule communication (Section **??**) using the available control mechanisms. CÉU is a superset of C targeting reactive systems in general, not only games, and has also been adopted in other domains, such as wireless sensor networks [19, 4] and multimedia systems [20].

Functional reactive programming (FRP) [7] contrasts with Structured synchronous reactive programming as a complementary programming style for reactive applications. We believe that FRP is more suitable for data-intensive applications, while SSRP, for control-intensive applications. On the one hand, FRP uses declarative formulas to specify continuous functions over time, such as for physics or data constraints among entities. On the other hand, describing a sequence of steps in FRP requires to encode explicit state machines so that functions can switch behavior depending on the current state. FRP has been successfully used to implement a 3D first person shooting game from scratch, but with performance considerations [5]. We rewrote an existing game in small steps while keeping it working. Although we do not provide a performance evaluation (Pingus is not performance sensitive), previous work on CÉU shows that it is comparable to C in the context of embedded systems [19]. Nonetheless, given the tight integration between with C/C++, critical parts of games can be preserved in C++ if needed.

5 CONCLUSION

TODO: non reactive, C++ integration - TODO: OO state + methods - eliminar estados explicitos com estruturas de controle apropriadas

We promote the *structured synchronous reactive* programming model of CÉU for the development of games. We present in-depth use cases categorized in four control-flow patterns applied to *Pingus* (an open-source *Lemmings* clone) that likely apply to other games.

We show how the standard way to program games with objects and callbacks in C++ hinders structured programming techniques, such as support for sequential execution, long-lasting loops, and persisting local variables. In this sense, callbacks actually disrupt

```cpp
class Bomber : public PinguAction {
    <...>
    Sprite sprite;
}

Bomber::Bomber (<...>) : <...> {
    sprite.load(<...>);
    <...>
}

void Bomber::update () {
    sprite.update ();
}

void Bomber::draw (SceneContext& gc) {
    <...>
    gc.color().draw(sprite, <...>);
}
```

[a] Implementation in C++

```
code/await Bomber (void) -> ActionName do
    <...>
    var&? Sprite sprite = spawn Sprite(<...>);
    <...>
end













.
```

[b] Implementation in CÉU

Figure 12: *Bomber action* `draw` and `update` dispatching.

Figure 13: UI children with static lifespan.

structured programming, becoming ["our generations goto"][goto] according to Miguel de Icaza.

Overall, we believe that most difficulties in implementing control behavior in game logic is not inherent to this domain, but a result of accidental complexity due to the lack of structured abstractions and an appropriate concurrency model to handle event-based applications.

[goto]: tirania.org/blog/archive/2013/Aug-15.html

6 Acknowledgments

We would like to thank Leonardo Kaplan and Alexander Tkachov for early explorations and prototypes of the game rewrite.

References

[1] Behavior trees in unreal. `https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/` (accessed in Jun-2017).

[2] A. Adya et al. Cooperative task management without manual stack management. In *Proceedings of ATEC'02*, pages 289–302. USENIX Association, 2002.

[3] A. Ampatzoglou and A. Chatzigeorgiou. Evaluation of object-oriented design patterns in game development. *Information and Software Technology*, 49(5):445–454, 2007.

[4] A. Branco, F. Sant'anna, R. Ierusalimschy, N. Rodriguez, and S. Rossetto. Terra: Flexibility and safety in wireless sensor networks. *ACM Trans. Sen. Netw.*, 11(4):59:1–59:27, Sept. 2015.

[5] M. H. Cheong. Functional programming and 3d games. Master's thesis, University of New South Wales, Sydney, Australia, November 2005.

[6] F. Devillers and S. Donikian. A scenario language to orchestrate virtual world evolution. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 265–275. Eurographics Association, 2003.

[7] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of ICFP'97*, pages 263–273, New York, NY, 1997. ACM.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented languages and systems*. Addison-Wesley Reading, 1994.

[9] D. Isla. Handling Complexity in the Halo 2 AI. In *Game Developers Conference*, Mar. 2005.

[10] I. Maier, T. Rompf, and M. Odersky. Deprecating the observer pattern. Technical report, 2010.

[11] W. Mallouk and E. Clua. An object-oriented approach for hierarchical state machines. In *Proceedings of SBGames'06*, pages 8–10, 2006.

[12] R. Nystrom. *Game Programming Patterns*. Genever Benning, 2014.

[13] L. L. O. P. A. Pagliosa. A new programming environment for dynamics-based animation. In *Proceedings of SBGames'06*. SBC, 2006.

[14] K. Perlin and A. Goldberg. Improv: A system for scripting interactive actors in virtual worlds. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 205–216. ACM, 1996.

[15] C. W. Reynolds. Computer animation with scripts and actors. In *ACM SIGGRAPH Computer Graphics*, volume 16, pages 289–296. ACM, 1982.

[16] G. L. R. Roberto Tenorio Figueiredo. Gof design patterns applied to the development of digital games. In *Proceedings of SBGames'15*. SBC, 2015.

[17] G. Salvaneschi et al. Rescala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of Modularity'13*, pages 25–36. ACM, 2014.

[18] F. Sant'Anna, N. Rodriguez, and R. Ierusalimschy. Structured Synchronous Reactive Programming with Céu. In *Proceedings of Modularity'15*, 2015.

[19] F. Sant'Anna, N. Rodriguez, R. Ierusalimschy, O. Landsiedel, and P. Tsigas. Safe System-level Concurrency on Resource-Constrained

```
GameSession::GameSession(<...>) :
    <...>
{
    <...>      // these widgets are always active...
    btpanel  = new ButtonPanel(<...>);
    pcounter = new PingusCounter(<...>);
    smallmap = new SmallMap(<...>);
    <...>
    uimgr->add(btpanel);  // ...but are added
    uimgr->add(pcounter); //  dynamically to the
    uimgr->add(smallmap); //  dispatching hierarchy
    <...>
}
```

[a] Implementation in C++

```
code/await Game (void) do
    <...> // other coexisting functionality
    spawn ButtonPanel(<...>);
    spawn PingusCounter(<...>);
    spawn SmallMap(<...>);
    <...> // other coexisting functionality
end
```

[b] Implementation in CÉU

Figure 14: Managing the UI widgets lifecycle.

```
Pingu* PinguHolder::create_pingu (<...>) {
    <...>
    Pingu* pingu = new Pingu (<...>);
    pingus.push_back(pingu);
    <...>
}

void PinguHolder::update() {
    <...>
    while(pingu != pingus.end()) {
        (*pingu)->update();
        if ((*pingu)->get_status() == PS_DEAD) {
            pingu = pingus.erase(pingu);
        }
        <...>
        ++pingu;
    }
}

.
```

[a] Implementation in C++

```
code/await Game (void) do
    <...>
    pool[] Pingu pingus;
    code/await Pingu_Spawn (<...>) do
        <...>
        spawn Pingu(<...>) in pingus;
    end
    <...>   // code invoking Pingu_Spawn
end

code/await Pingu (<...>) do
    <...>
    loop do
        await game.dt;
        if Pingu_Is_Out_Of_Screen() then
            <...>
            escape {PS_DEAD};
        end
    end
end
```

[b] Implementation in CÉU

Figure 15: Managing the pingus lifecycle.

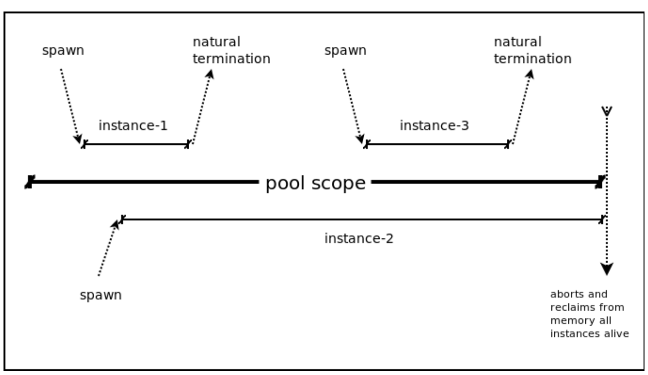

Figure 16: Lifespan of dynamic instances.

Nodes. In *Proceedings of SenSys'13*. ACM, 2013.

[20] R. C. M. Santos, G. F. Lima, F. Sant'Anna, and N. Rodriguez. Céu-Media: Local Inter-Media Synchronization Using Céu. In *Proceedings of WebMedia'16*, pages 143–150, New York, NY, USA, 2016. ACM.

[21] T. Sweeney. The next mainstream programming language: a game developer's perspective. *ACM SIGPLAN Notices*, 41(1):269–269, 2006.

[22] L. Valente, A. Conci, and B. Feijó. An architecture for game state management based on state hierarchies. In *Proceedings of SBGames'06*. Citeseer, 2006.