

# 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

---

### Abstract

We present a case study of rewriting the video game Pingus from C++ to the structured synchronous reactive language CÉU. CÉU supports reactive control-flow primitives that eliminate callbacks and let programmers write code in direct and sequential style. Structured reactivity helps describing complex control-flow relationships in the game logic more concisely. We show gains in productivity for six behaviors in Pingus through a qualitative analysis of the proposed implementations in CÉU in comparison to the originals in C++. We also categorize the behaviors in four recurrent control-flow patterns that likely apply to other games.

*Keywords:* C++, CÉU, Control Flow, Event-Driven Programming, Game Logic, Synchronous Reactive Programming

---

### 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 code-base<sup>1</sup> 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 only

---

<sup>1</sup>Official Pingus repository: [github.com/Pingus/pingus/](https://github.com/Pingus/pingus/)



Figure 1: Pingus gameplay.

accounts for 10% of the CPU budget [20]. 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 use 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*”.<sup>2</sup>

In this work, we advocate structured synchronous reactive programming as a more productive alternative for game logic development. We present a case study of rewriting Pingus from C++ to the programming language CÉU [19, 18]. CÉU has the following general characteristics:

- *Reactive*: code executes in reactions to events.

<sup>2</sup>“Callbacks as our Generations’ *goto* Statement”: [tirania.org/blog/archive/2013/Aug-15.html](http://tirania.org/blog/archive/2013/Aug-15.html)

- 28 • *Structured*: code uses structured control-flow mechanisms, such as `spawn`  
29 and `await` (to create and suspend an activity).
- 30 • *Synchronous*: event reactions never overlap and run atomically and to com-  
31 pletion on each activity. There is no implicit preemption or real parallelism,  
32 resulting in deterministic execution.

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

38 Our case study shows gains in productivity for six selected behaviors in the  
39 game logic of Pingus rewritten in CÉU. We present an in-depth qualitative anal-  
40 ysis of the proposed solutions in comparison to the original implementations in  
41 C++. We also identify four control-flow patterns that likely apply to other games:  
42 *Finite State Machines*, *Continuation Passing*, *Dispatching Hierarchies*, and *Lifes-*  
43 *pan Hierarchies*. A control-flow pattern is a recurring technique to describe ex-  
44 ecution dependency and/or explicit ordering between statements. We focus on a  
45 qualitative analysis for the programming techniques that we applied during the  
46 rewriting process. Not all techniques result in reduction of *locs* (especially con-  
47 sidering the verbose syntax of CÉU), but have other effects such as eliminating  
48 shared variables and dependencies between classes. We employed a *live code*  
49 *rewrite*, i.e., starting from the original codebase in C++, we reimplemented it  
50 piece-by-piece in CÉU without breaking the game compilation and execution.

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

## 56 2. The Pingus Codebase and Rewriting Process

57 In Pingus, the game logic accounts for almost half the size of the codebase<sup>3</sup>:  
58 18.173 from 39.362 *locs* (46%) spread across 272 files. However, about half of the

---

<sup>3</sup>We used *SLOCCount* to count only non-blank, non-comment lines in the codebase: [www.dwheeler.com/sloccount/](http://www.dwheeler.com/sloccount/)

	Path	Ceu	C++	Ceu/C++	Description
1	game/	2064	2268	0.91	the main gameplay
2	./	710	679	1.05	main functionality
3	objs/	470	478	0.98	world objects (tiles, traps, e
4	pingu/	884	1111	0.80	pingu behaviors
5	./	343	458	0.75	main functionality
6	actions/	541	653	0.83	pingu actions (bomber, cli
7	worldmap/	468	493	0.95	campaign worldmap
8	screens/	1109	1328	0.84	menus <b>and</b> screens
9	option/	347	357	0.97	option menu
10	others/	762	971	0.78	other menus <b>and</b> screens
11	misc/	56	46	1.22	miscellaneous functionality
		-----	-----	-----	
		3697	4135	0.89	

Listing 1: The Pingus codebase directory tree.

game logic relates to non-reactive code, such as dealing with configurations and options, saved games and serialization, maps and level descriptions, string formatting, collision detection, graph algorithms, etc. This part remains unchanged and relies on the seamless integration between C  U and C/C++ [19]: the type systems are equivalent and the integration happens at the source code level. This enables accessing data and calling C/C++ from C  U and vice-versa. Therefore, we only rewrote 9.186 *locs* spread across 126 files<sup>4</sup>. In order to only consider relevant 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++<sup>4</sup>. We did the same with the implementation in C  U, resulting in 3.697 dense *locs*<sup>4</sup>. Figure ?? summarizes the effective game logic codebase in the two implementations.

Although the analysis in this work is mostly qualitative, the rows with lower ratio numbers in Figure ?? do correlate with the parts of the game logic that we consider more susceptible to structured reactive programming. For instance, the *Pingu* behavior (row 4, *ratio* 0.80) contains complex animations that are affected by timers, game rules, and user interaction. In contrast, the *Option screen* (row 9, *ratio* 0.97) is a simple UI grid with trivial mouse interactions.

The rewriting process consisted of identifying sets of callbacks in C++ imple-

<sup>4</sup> Effective codebase: [github.com/an000/p/tree/master/](https://github.com/an000/p/tree/master/)

---

```

input void A;
input void B;
var int x = 1;
par/and do
    await A;
    x = x + 1;
with
    await B;
    x = x * 2;
end

```

---

[a] Accesses to  $x$  are never concurrent.

---

```

1 input void A;
2 // (empty line)
3 var int y = 1;
4 par/and do
5     await A;
6     y = y + 1;
7 with
8     await A;
9     y = y * 2;
10 end

```

---

[b] Accesses to  $y$  are concurrent but still deterministic.

Figure 2: Shared-memory concurrency in C  U: example [a] is safe because the trails access  $x$  atomically in different reactions; example [b] is unsafe because both trails access  $y$  in the same reaction.

menting control flow in the game and translating them to C  U 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. As a general rewriting rule, we identify control-flow behaviors in the C++ codebase by looking for class state members with identifiers resembling verbs, statuses, and counters (e.g., `pressed`, `particle_thrown`, `mode`, and `delay_count`). Good chances are that such variables encode some form of control-flow progression that cross multiple callback invocations.

### 3. Control-Flow Patterns & Case Studies

During the rewriting process, we have identified four abstract cause/effect control-flow patterns which likely apply to other games as well:

1. *Finite State Machines*: Event occurrences lead to transitions between states and trigger actions comprising the behavior of a game entity.
2. *Continuation Passing*: The completion of a long-lasting activity in the game may carry a continuation, i.e., some action to execute next.

- 96 3. *Dispatching Hierarchies*: Entities form a dispatching hierarchy in which a  
97 container that receives a stimulus automatically forwards it to its managed  
98 children.  
99 4. *Lifespan Hierarchies*: Entities form a lifespan hierarchy in which a termi-  
100 nating container entity automatically destroys its managed children.

101 We describe six representative game behaviors in detail distributed in the four  
102 patterns and analyze their implementations in C++ and CÉU.<sup>5</sup>

### 103 3.1. *Finite State Machines*

104 Event occurrences lead to transitions between states and trigger actions com-  
105 prising the behavior of a game entity.

#### 106 3.1.1. *Case Study: Detecting Double-Clicks in the Armageddon Button*

107 In Pingus, a double click in the *Armageddon button* at the bottom right of the  
108 screen literally explodes all pingus.<sup>6</sup>

109 Figure 3.a shows the C++ implementation for the class `ArmageddonButton`  
110 with methods for rendering the button and handling mouse and timer events. The  
111 code in the figure focus on the double click detection and hides unrelated parts  
112 with `<...>`. The methods `update` (ln 14–26) and `on_click` (ln 28–34) are ex-  
113 amples of *short-lived callbacks*, which are pieces of code that execute atomically  
114 in reaction to external input events. The callback `on_click` reacts to mouse clicks  
115 detected by the base class `RectComponent` (ln 2), while the callback `update`  
116 continuously reacts to the passage of time, frame by frame. The class first initial-  
117 izes the variable `pressed` (ln 3) to track the first click (ln 32). It also initializes  
118 the variable `press_time` (ln 4) to count the time since the first click (ln 16–17).  
119 If another click occurs within 1 second, the class signals the double click to the  
120 application (ln 29–30). Otherwise, the `pressed` and `press_time` state variables  
121 are reset (ln 18–21). Figure 4 illustrates how we can model the double-click be-  
122 havior in C++ as a state machine. The circles represent the state of the variable  
123 `pressed`, and the arrows represent the callbacks manipulating it. Note in Fig-  
124 ure 3.a how the accesses to the state variables are spread across the entire class:  
125 the distance between the initialization of `pressed` (ln 3) and the last access to it  
126 (ln 32) is over 40 lines in the original file. Arguably, this dispersion of code across

---

<sup>5</sup>Due to space constraints, we omit three other game behaviors and a fifth pattern *Signaling Mechanisms*.

<sup>6</sup>Double click animation: [github.com/an000/p/#1](https://github.com/an000/p/#1)

127 methods makes the understanding and maintenance of the double-click behavior  
128 more difficult. Also, even though the state variables are private, unrelated methods  
129 such as `draw`, which is defined in middle of the class (ln 10–12), can potentially  
130 access them.

131 CÉU supports structured constructs to deal with events, aiming to eradicate  
132 explicit manipulation of state variables for control-flow purposes. In Figure 3.b,  
133 the loop to detect double clicks (ln 4–10) awaits the first click (ln 5) and then,  
134 while watching 1 second (ln 6–9), awaits the second click (ln 7). If the second  
135 click occurs within 1 second, the `break` terminates the loop (ln 8) and the `emit`  
136 in sequence signals the double click to the application (ln 12). Otherwise, the  
137 `watching` block as a whole aborts after 1 second and the loop restarts, falling  
138 back to the first click `await` (ln 5). Double click detection in CÉU does not rely  
139 on state variables and is entirely self-contained in the `loop` body. Also, those 7  
140 lines of code *only* detect the double click, leaving the actual effect (ln 12) as well  
141 as all unrelated code (such as redrawing the button) to happen outside the loop.

142 The `await` allows for nested control-flow statements to suspend execution  
143 while retaining all enclosing state alive, such as local variables and next statement  
144 to execute. Then, a subsequent reaction to an event resumes execution normally.  
145 In contrast, method callbacks have a single entry point at the top level of the class,  
146 in which only instance members remain active between invocations. In particular,  
147 locals and loops cannot persist across invocations.

### 148 3.1.2. Case Study: The Bomber Action Animation Sequence

149 The player may assign actions to specific pingus, as illustrated in Figure 5. The  
150 *Bomber action* explodes the clicked pingu, throwing particles around and also de-  
151 stroying the terrain under its radius.<sup>7</sup> We can model the explosion animation with  
152 a sequential state machine (Figure 6) with effects associated to specific frames as  
153 follows<sup>8</sup>:

- 154 1. 0th frame: plays a "Oh no!" sound.
- 155 2. 10th frame: plays a "Bomb!" sound.
- 156 3. 13th frame: throws particles, destroys the terrain, and shows an explosion  
157 sprite.
- 158 4. Game tick: hides the explosion sprite.
- 159 5. Last frame: kills the pingu.

---

<sup>7</sup>Bomber action animation: [github.com/an000/p/#2](https://github.com/an000/p/#2)

<sup>8</sup>State machine animation: [github.com/an000/p/#3](https://github.com/an000/p/#3)

160 In C++, the class `Bomber` in Figure 8.a defines the callbacks `draw` and `update`  
 161 to manage the state machine of Figure 6. The class first defines one state variable  
 162 for each effect to perform (ln 4–7). The “Oh no!” sound plays as soon as the object  
 163 starts in *state-1* (ln 11). The `update` callback (ln 14–38) first updates the pingu  
 164 animation and movement on every frame, regardless of its current state (ln 15–  
 165 16). When the animation reaches the 10th frame, it switches to *state-2* and plays  
 166 the “Bomb!” sound (ln 18–22). The state variable `sound_played` is required  
 167 because the sprite frame doesn’t necessarily advance on every `update` invocation  
 168 (e.g., `update` may execute twice during the 10th frame). The same reasoning and  
 169 technique applies to *state-3* (ln 24–32 and 41–46). The explosion sprite appears in  
 170 a single frame in *state-4* (ln 45). Finally, the pingu dies after the animation frames  
 171 terminate (ln 34–37). Note that a single numeric state variable would suffice to  
 172 track the states as in Figure 6, but the original developers probably chose to en-  
 173 code each state in an independent boolean variable to rearrange and experiment  
 174 with them during development. Still, due to the short-lived nature of callbacks,  
 175 state variables are unavoidable and are actually the essence of object-oriented pro-  
 176 gramming (i.e., methods with mutable state). Like the double click detection in  
 177 C++, note that the state machine is encoded across 3 different methods, each in-  
 178 termixing code with unrelated functionality (e.g., changing frames, moving, and  
 179 redrawing).

180 The equivalent code in CÉU for the *Bomber action* in Figure 8.b does not rely  
 181 on state variables and reflects the sequential state machine implicitly, using `await`  
 182 statements to separate the effects in direct style. The `Bomber` is a `code/await`  
 183 abstraction of CÉU, which is similar to a coroutine or fiber [2]: a subroutine that  
 184 retains runtime state, such as local variables and the program counter, across re-  
 185 actions to events (i.e., across `await` statements). The pingu movement and sprite  
 186 animation are isolated in two other `code/await` abstractions and execute in sep-  
 187 arate through the `spawn` primitive (ln 4–5). In CÉU, if multiple abstractions react  
 188 to the same event, the scheduler employs lexical order to preserve determinism,  
 189 i.e., the `spawn` that appears first in the source code reacts first [18]. The event  
 190 `game.update` (ln 12,16,24) is analogous to the `update` callback of C++ and oc-  
 191 curs on every game frame. The code tracks the animation aliveness (ln 7–27) and,  
 192 on termination, performs the last bomber effect, killing the pingu (ln 30). As soon  
 193 as the animation starts, the code performs the first effect (ln 9). The intermedi-  
 194 ate effects are performed when the corresponding conditions occur (ln 12,16,24).  
 195 The `do-end` block (ln 19–25), restricts the lifespan of the single-frame explosion  
 196 sprite (ln 21): after the next game tick (ln 24), the block terminates and automati-  
 197 cally destroys the spawned abstraction (removing it from the screen). In contrast



198 with the implementation in C++, all effects occur in a contiguous chunk of code  
199 (ln 7–30), which handles no extra functionality.

### 200 3.1.3. *Summary & Pattern Uses in Pingus*

201 The structured constructs of CÉU introduce some advantages in comparison  
202 to explicit state machines:

- 203 • They encode all states with direct sequential code, eliminating callbacks  
204 and shared state variables for control-flow purposes.
- 205 • They handle all states (and only them) in the same contiguous block, im-  
206 proving code encapsulation.

207 Object-oriented games also adopt the *state pattern* to model state machines with  
208 subclasses describing each possible state [12]. However, this approach is not fun-  
209 damentally different from Pingus’ use of `switch` or `if` branches to decode state.

210 Pingus supports 16 actions in the game. Five of them implement at least one  
211 state machine and are considerable smaller in CÉU in terms of *locs* (Figure 7).  
212 For the other 11 actions without state machines, the reduction in *locs* is negligi-  
213 ble. This asymmetry illustrates the gains in expressiveness when describing state  
214 machines in direct style.

215 Among all 65 implementation files in CÉU, we found 29 cases in 25 files that  
216 use structured mechanisms to substitute states machines. They typically mani-  
217 fest as `await` statements in sequence (e.g., ln 5,7 in Figure 3 and ln 12,16,24 in  
218 Figure 8).

### 219 3.2. *Continuation Passing*

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

#### 222 3.2.1. *Transition from Story to Credits Screen*

223 The campaign world map has clickable blue dots in the two extremes of the  
224 map road to show introductory and closing ambience stories, respectively. For  
225 introductory stories, the game returns to the world map after showing the story  
226 pages. For closing stories, the game also shows a *Credits screen* before returning  
227 to the world map.<sup>9</sup>

---

<sup>9</sup>Credits screen animation: [github.com/an000/p/#4](https://github.com/an000/p/#4)

228 In C++, the class `StoryDot` in Figure 9.a (ln 1–12) first reads the level file (ln  
 229 5) to check whether it is a closing story and should, after termination, show the  
 230 *Credits screen*. The boolean variable `show_credits` (ln 2,5,10) is passed to the  
 231 class `StoryScreen` (ln 10) and represents the screen continuation, i.e., what to  
 232 do after showing the story. The class `StoryScreen` (not shown) then forwards  
 233 the continuation even further to the auxiliary class `StoryScreenComp` (ln 16–  
 234 40). When the method `next_text` has no story pages left to display (ln 32–38), it  
 235 decides where to go next, depending on the continuation flag `show_credits` (ln  
 236 33).

237 In CÉU, the `loop` of Figure 9.b controls the flow between the screens to show  
 238 as a direct sequence of statements. We first invoke the `Worldmap` (ln 2), which  
 239 shows the map and lets the player interact with it (e.g., walking around) until a  
 240 dot is clicked. If the player selects a story dot (ln 4–9), we invoke the `Story` and  
 241 await its termination (ln 5). After showing the story, we check the returned values  
 242 (ln 6) to perhaps show the `Credits` screen (ln 8). The enclosing loop restores the  
 243 `Worldmap` and repeats the process.

244 Figure 10 illustrates the *continuation-passing style* of C++ and the *direct style*  
 245 of CÉU for screen transitions:

- 246 1. Main Loop  $\longrightarrow$  Worldmap:
  - 247 • C++ uses an explicit stack to push the `Worldmap` screen (not shown
  - 248 in Figure 12.a).
  - 249 • CÉU invokes the `Worldmap` screen expecting a return value (Figure 12.b,
  - 250 ln 2).
- 251 2. Worldmap (*blue dot click*)  $\longrightarrow$  Story:
  - 252 • C++ pushes the `Story` screen passing the continuation flag (Figure 12.a,
  - 253 ln 10).
  - 254 • CÉU stores the `Worldmap` return value and invokes the `Story` screen
  - 255 (Figure 12.b, ln 2,5).
- 256 3. Story  $\longrightarrow$  Credits:
  - 257 • C++ replaces the current `Story` screen with the `Credits` screen (Fig-
  - 258 ure 12.a, ln 34).
  - 259 • CÉU invokes the `Credits` screen after the `await Story` returns
  - 260 (Figure 12.b, ln 8).
- 261 4. Credits  $\longrightarrow$  Worldmap:

- 262 • C++ pops the `Credits` screen, going back to the `Worldmap` screen  
263 (not shown in Figure 12.a).
- 264 • CÉU uses an enclosing `loop` to restart the process (Figure 12.b, ln  
265 1–13).

266 In contrast with C++, the screens in CÉU are decoupled from each other and  
267 only the `Main Loop` touches them: the `Worldmap` has no references to `Story`,  
268 which has no references to `Credits`. Changing the screen arrangements is a  
269 matter of adjusting the main loop only.

### 270 3.2.2. *Summary & Pattern Uses in Pingus*

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

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

279 Continuation passing typically controls the overall structure of the game in  
280 C++, such as screen transitions in menus and level progressions. CÉU adopts the  
281 direct style technique in five cases involving screen transitions: the main menu,  
282 the level menu, the level set menu, the world map loop, and the gameplay loop.  
283 It also uses the same technique for the loop that switches between *pingu* actions  
284 during gameplay (e.g., *walking* to *falling* and back to *walking*).

### 285 3.3. *Dispatching Hierarchies*

286 Entities form a dispatching hierarchy in which a container that receives a stim-  
287 ulus automatically forwards it to its managed children.

#### 288 3.3.1. *Case Study: Bomber Action draw and update Dispatching*

289 In C++, the class `Bomber` in Figure 12.a declares a `sprite` member (ln 3)  
290 to handle its animation frames. The `Sprite` class is part of the game engine  
291 and knows how to update and render itself. However, the `Bomber` still has to  
292 respond to `update` and `draw` requests from the game and forward them to the

293 sprite (ln 11–13 and 15–18). To understand how the `update` callback flows from  
294 the original environment stimulus to the game down to the sprite, we need to  
295 follow a long chain of 7 method dispatches (Figure 11):

- 296 1. `ScreenManager::display` in the main game loop calls `ScreenManager::update`  
297 when starting a new frame.
- 298 2. `ScreenManager::update` calls `screen->update` for the active game  
299 screen (i.e., a `GameSession` instance, considering the screen in which the  
300 Bomber appears).
- 301 3. `GameSession::update` calls `world->update`.
- 302 4. `World::update` calls `objs->update` for each object in the world.
- 303 5. `PinguHolder::update` calls `pingu->update` for each pingu alive.
- 304 6. `Pingu::update` calls `action->update` for the active pingu action.
- 305 7. `Bomber::update` calls `sprite.update`. `Sprite::update` finally up-  
306 dates the animation frame.

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

- 308 • With a single assignment to `screen`, one can easily deactivate the current  
309 screen and redirect all dispatches to a new screen (step 2).
- 310 • The `World` class manages and dispatches events to all game entities with a  
311 common interface `WorldObj`, such as the pingus and traps (step 4).
- 312 • Since it is common to iterate only over the pingus (vs. all world objects),  
313 the container `PinguHolder` manages all pingus (step 5).
- 314 • Since a single pingu can change its actions during lifetime, the `action`  
315 member decouples them with another level of indirection (step 6).
- 316 • Sprites are part of the game engine and are reusable everywhere (e.g., UI  
317 buttons, world objects, etc.), so it is also convenient to decouple them from  
318 actions (step 7).

319 The `draw` callback also flows through a similar dispatching hierarchy until reach-  
320 ing the `Sprite` class.

321 In CÉU, the `Bomber` abstraction presented in Figure 12.b spawns a `Sprite`  
322 animation instance on its body (ln 3). The `Sprite` abstraction can react directly  
323 to external `update` and `draw` events, bypassing the program hierarchy entirely.  
324 Events in CÉU are broadcasted to the entire application in lexical order, i.e., an ab-  
325 straction that appears first in the source code (e.g., ln 3) reacts before another one

326 that appears second (e.g., ln 4). As discussed in Section 3.1.2, this rule preserves  
327 determinism and also conforms to the program static hierarchy. While (*and only*  
328 *while*) the bomber abstraction is alive, the sprite animation remains alive and re-  
329 acts to the `update` and `draw` events. The radical decoupling between the program  
330 hierarchy and reactions to events eliminates dispatching chains entirely.

### 331 3.3.2. Summary & Pattern Uses in Pingus

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

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

337 In C++, the update subsystem touches 39 files with around 100 lines of code  
338 just to forward `update` methods through the dispatching hierarchy. For the draw-  
339 ing subsystem, 50 files with around 300 lines of code. The implementation in C++  
340 also relies on dispatching hierarchy for `resize` callbacks, touching 12 files with  
341 around 100 lines of code. Most of this code is eliminated in CÉU since abstrac-  
342 tions can react directly to the environment, not depending on hierarchies spread  
343 across multiple files.

344 Note that dispatching hierarchies cross game engine code, suggesting that  
345 most games also rely heavily on this control-flow pattern. In the case of the Pin-  
346 gus engine, we rewrote 9 files with a reduction from 515 to 173 *locs* (not listed in  
347 Figure ??), mostly due to dispatching code removal.

### 348 3.4. Lifespan Hierarchies

349 Entities form a lifespan hierarchy in which a terminating container entity au-  
350 tomatically destroys its managed children.

#### 351 3.4.1. Case Study: Static Game UI Widgets

352 Figure 14 shows the game UI widgets with action buttons, score counters, and  
353 a small map, all coexisting with the game screen during its whole lifespan.

354 In C++, the widgets are created in the constructor of the class `GameSession`  
355 in Figure 13.a (ln 5–7), added to a UI container (ln 9–11), and are never removed  
356 since they must always be visible. Arguably, to better express the intent of making  
357 them coexist with the game screen, the widgets could alternatively be declared as

358 top-level automatic (non-dynamic) members. However, the class relies on a con-  
359 tainer to automate `draw` and `update` dispatching to the widgets, as discussed in  
360 Section 3.3. The container method `add` expects only dynamically allocated chil-  
361 dren because they are automatically deallocated inside the container destructor.  
362 However, the dynamic nature of containers in C++ demand extra caution from  
363 programmers:

- 364 • When containers are part of a dispatching chain, it gets even harder to know  
365 which objects are dispatched at a given moment: one has to “simulate” the  
366 program execution and track calls to `add` and `remove`.
- 367 • For objects with dynamic lifespan, calls to `add` must always have matching  
368 calls to `remove`: missing calls to `remove` lead to memory and CPU leaks  
369 (to be discussed as the *lapsed listener problem* in Section 3.4.2).

370 In CÉU, the UI entities that coexist are simply created in the same lexical  
371 block of the `Game` abstraction in Figure 13.b (ln 3–5). Since abstractions can react  
372 independently, they do not require a dispatching container. Lexical lifespan never  
373 requires containers, allocation and deallocation, or explicit references. In addition,  
374 all required memory is known at compile time, similarly to stack-allocated local  
375 variables. The *Bomber action* of Section 3.1.2 also relies on lexical scope to  
376 delimit the lifespan of the explosion sprite into a single frame (Figure 8, ln 19–  
377 25).

### 378 3.4.2. Case Study: Dynamic Pingus Lifecycle

379 A pingu is a dynamic entity created periodically and destroyed under certain  
380 conditions, such as falling from a high altitude.<sup>10</sup>

381 In C++, the class `PinguHolder` in Figure 15.a is a container that holds all  
382 alive pingus. The method `PinguHolder::create_pingu` (ln 1–6) is called pe-  
383 riodically to create a new `Pingu` and add it to the `pingus` collection (ln 3–4). The  
384 method `PinguHolder::update` (ln 8–18) checks the state of all pingus on every  
385 frame, removing those with the `dead` status (ln 12–14). Note that if the program-  
386 mer disregards the call to `remove`, a dead pingu would remain in the collection  
387 and still update on every frame (ln 11). Since the `draw` behavior for a dead pingu  
388 is innocuous, the death could go unnoticed when testing it but the program would  
389 keep consuming memory and CPU time. This problem is known as the *lapsed*

---

<sup>10</sup>Death of pingus animation: [github.com/an000/p/#5](https://github.com/an000/p/#5)

390 *listener* [12] and also occurs in languages with garbage collection: a container  
391 typically holds a strong reference to a child (sometimes the only reference to it),  
392 and the runtime cannot magically detect it as garbage. Hence, entities with dy-  
393 namic lifespan always require explicit matching `add` and `remove` calls associated  
394 to a container (ln 4,13).

395 CÉU supports `pool` declarations to hold dynamic abstraction instances. In ad-  
396 dition, the `spawn` statement supports a pool identifier to associate a new instance  
397 with a pool. The game screen in Figure 15.b spawns a new `Pingu` on every invo-  
398 cation of `Pingu.Spawn` (ln 4–7). The `spawn` statement (ln 6) specifies the pool  
399 declared at the top-level block of the game screen (ln 3). In this case, the lifespan  
400 of the new instances follows the scope of the pool (ln 1–9) instead of the enclos-  
401 ing scope of the `spawn` statement (ln 4–7), surviving the call to `Pingu.Spawn`.  
402 Since pools are also subject to lexical scope, the lifespan of all dynamically al-  
403 located pingus is constrained to the game screen. Lexical scopes handle memory  
404 and event dispatching automatically for static instances and also for pools. How-  
405 ever, the lifespan of a dynamic instance does not necessarily have to match the  
406 lifespan of its associated pool (Figure 16). In CÉU, when the execution block of  
407 a dynamic instance terminates, which characterizes its *natural termination*, the  
408 instance is automatically removed from its pool. Therefore, dynamic instances do  
409 not require any extra bookkeeping related to containers or explicit deallocation.  
410 To remove a pingu from the game in CÉU, we just need to terminate its execu-  
411 tion block according to the appropriate conditions: The `escape` statement (ln 17)  
412 aborts the execution block of the `Pingu` instance, removing it from its associated  
413 pool automatically. Hence, a dynamic instance that terminates naturally leaves no  
414 traces in the program.

### 415 3.4.3. Summary & Pattern Uses in Pingus

416 Lexical lifespan for static instances and natural termination for dynamic in-  
417 stances provide some advantages in comparison to lifespan hierarchies through  
418 containers:

- 419 • Lexical scope makes an abstraction lifespan explicit in the source code. All  
420 entities in a game have an associated lexical lifespan.
- 421 • The memory for static instances is known at compile time.
- 422 • Natural termination makes an instance innocuous and, hence, susceptible to  
423 immediate reclamation.

- 424 • Instances (static or dynamic) never require explicit manipulation of point-  
425 ers/references.

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

#### 430 4. Related Work

431 The control-flow patterns closely relate to the *GoF* behavioral patterns [8],  
432 which are discussed in the context of video games in previous work [12, 16, 3].  
433 The original Pingus in C++ uses variations of the patterns *state* (Sections 3.1 and 3.2),  
434 *visitor* (Sections 3.3 and 3.4), and *observer* (to handle events in general) as im-  
435 plementation techniques to achieve the desired higher-level control-flow patterns  
436 described in the paper. CÉU overcomes the need of behavioral patterns with sup-  
437 port, at the language level, for structured control-flow mechanisms and event-  
438 based communication via broadcast.

439 A number of domain-specific languages, frameworks, and techniques have  
440 been proposed for particular subsystems of the game logic, such as animations [13,  
441 6, 14, 15], game state and screen progression [21, 11], and behavior and AI mod-  
442 eling [9, 1]. In Pingus, the adoption of CÉU is not restricted to a specific subsys-  
443 tem. We employed CÉU at the very core of the game for event dispatching (Sec-  
444 tion 3.3) and memory management of entities (Section 3.4), eliminating parts of  
445 the original game engine. We also implemented all entity animations and behav-  
446 iors (Section 3.1), and screen transitions (Section 3.2) using the available control  
447 mechanisms of CÉU. Furthermore, CÉU is a superset of C targeting reactive sys-  
448 tems in general, not only games, and has also been successfully adopted in other  
449 domains, such as wireless sensor networks [19, 4] and multimedia systems [? ].

450 Functional reactive programming (FRP) [7] contrasts with structured synchronous  
451 reactive programming (SSRP) as a complementary programming style for reactive  
452 applications. We believe that FRP is more suitable for data-intensive applications,  
453 while SSRP, for control-intensive applications. On the one hand, FRP uses declar-  
454 ative formulas to specify continuous functions over time, such as for physics or  
455 data constraints among entities. On the other hand, describing a sequence of steps  
456 or control-flow dependencies in FRP requires to encode explicit state machines so  
457 that functions can switch behavior depending on the current state. FRP has been  
458 successfully used to implement a 3D first person shooting game from scratch,



459 but with some performance considerations [5]. Although we do not provide a  
460 performance evaluation (Pingus is not performance sensitive), previous work on  
461 CÉU shows that it is comparable to C in the context of embedded systems [19].  
462 Nonetheless, given the tight integration between CÉU and C/C++, critical parts of  
463 games can be preserved in C++ if needed.

## 464 5. Conclusion

465 We advocate *Structured Synchronous Reactive Programming* as a productive  
466 alternative for game logic development. We use the video game *Pingus* as a case  
467 study. We compare the implementation of six game behaviors in C++ and CÉU  
468 and discuss how structured reactive mechanisms can eliminate callbacks and let  
469 programmers write code in direct style. Ultimately, we rewrote about 1/4 of the  
470 whole codebase (9.186 from 39.362 lines of code) which comprises the core of  
471 the game logic that is susceptible to structured reactive programming.

472 We categorize the behaviors in four recurrent control-flow patterns: *State ma-*  
473 *chines* are the workhorses of the game logic, appearing in animations, AI be-  
474 haviors, and input handling. CÉU can encode states implicitly with sequential  
475 statements, eliminating shared state variables and improving code encapsulation.  
476 *Continuation passing* controls the overall structure of the game, such as screen  
477 transitions and level progressions. Similarly to state machines, CÉU describes  
478 the flow of the game as sequential statements in self-contained blocks, eliminat-  
479 ing explicit data structures and continuation variables. *Dispatching hierarchies*  
480 disseminate input events through the game entities and serve as a broadcast com-  
481 munication mechanism. Event broadcasting is at the core of the semantics of CÉU,  
482 allowing entities to react directly to inputs and bypass the program hierarchy en-  
483 tirely. *Lifespan hierarchies* manage the memory and visibility of game entities  
484 through class fields and containers. In CÉU, all entities have an associated lexical  
485 scope, similarly to local variables with automatic memory management.

486 Overall, we believe that most difficulties in implementing control-flow behav-  
487 ior in game logic is not inherent to this domain, but a result of accidental com-  
488 plexity due to the lack of structured abstractions and an appropriate concurrency  
489 model to develop event-based applications.

## 490 6. Acknowledgments

491 The authors would like to thank Leonardo Kaplan and Alexander Tkachov for  
492 early explorations and prototypes of the game rewrite.

- 493 [1] Behavior trees in Unreal. [https://docs.unrealengine.com/](https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/)  
494 latest/INT/Engine/AI/BehaviorTrees/ (accessed in Jun-  
495 2017).
- 496 [2] A. Adya et al. Cooperative task management without manual stack manage-  
497 ment. In *Proceedings of ATEC'02*, pages 289–302. USENIX Association,  
498 2002.
- 499 [3] A. Ampatzoglou and A. Chatzigeorgiou. Evaluation of object-oriented de-  
500 sign patterns in game development. *Information and Software Technology*,  
501 49(5):445–454, 2007.
- 502 [4] A. Branco, F. Sant’anna, R. Ierusalimschy, N. Rodriguez, and S. Rossetto.  
503 Terra: Flexibility and safety in wireless sensor networks. *ACM Trans. Sen.*  
504 *Netw.*, 11(4):59:1–59:27, Sept. 2015.
- 505 [5] M. H. Cheong. Functional programming and 3D games. Master’s thesis,  
506 University of New South Wales, Australia, November 2005.
- 507 [6] F. Devillers and S. Donikian. A scenario language to orchestrate virtual  
508 world evolution. In *Proceedings Eurographics’03*, 2003.
- 509 [7] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of*  
510 *ICFP’97*, pages 263–273, New York, NY, 1997. ACM.
- 511 [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: ele-*  
512 *ments of reusable object-oriented languages and systems*. Addison-Wesley  
513 Reading, 1994.
- 514 [9] D. Isla. Handling Complexity in the Halo 2 AI. In *Game Developers Con-*  
515 *ference*, Mar. 2005.
- 516 [10] I. Maier, T. Rompf, and M. Odersky. Deprecating the observer pattern. Tech-  
517 nical report, 2010.
- 518 [11] W. Mallouk and E. Clua. An object-oriented approach for hierarchical state  
519 machines. In *Proceedings of SBGames’06*, pages 8–10, 2006.
- 520 [12] R. Nystrom. *Game Programming Patterns*. Genever Benning, 2014.
- 521 [13] L. Pagliosa. A new programming environment for dynamics-based anima-  
522 tion. In *Proceedings of SBGames’06*. SBC, 2006.

- 523 [14] K. Perlin and A. Goldberg. Improv: A system for scripting interactive actors  
524 in virtual worlds. In *Proceedings of SIGGRAPH'96*, pages 205–216. ACM,  
525 1996.
- 526 [15] C. W. Reynolds. Computer animation with scripts and actors. In *Proceedings*  
527 *of SIGGRAPH'82*, volume 16, pages 289–296. ACM, 1982.
- 528 [16] G. R. Roberto Figueiredo. GOF design patterns applied to the development  
529 of digital games. In *Proceedings of SBGames'15*. SBC, 2015.
- 530 [17] G. Salvaneschi et al. Rescala: Bridging between object-oriented and func-  
531 tional style in reactive applications. In *Proceedings of Modularity'13*, pages  
532 25–36. ACM, 2014.
- 533 [18] F. Sant'Anna, N. Rodriguez, and R. Ierusalimschy. Structured Synchronous  
534 Reactive Programming with Céu. In *Proceedings of Modularity'15*, 2015.
- 535 [19] F. Sant'Anna, N. Rodriguez, R. Ierusalimschy, O. Landsiedel, and P. Tsi-  
536 gas. Safe System-level Concurrency on Resource-Constrained Nodes. In  
537 *Proceedings of SenSys'13*. ACM, 2013.
- 538 [20] T. Sweeney. The next mainstream programming language: a game devel-  
539 oper's perspective. *ACM SIGPLAN Notices*, 41(1):269–269, 2006.
- 540 [21] L. Valente, A. Conci, and B. Feijó. An architecture for game state manage-  
541 ment based on state hierarchies. In *Proceedings of SBGames'06*. Citeseer,  
542 2006.

<pre> 1 ArmageddonButton::ArmageddonButton(&lt;...&gt;) : 2   RectComponent(&lt;...&gt;), 3   pressed(false); // button initially not pressed 4   press_time(0); // how long since last click 5   &lt;...&gt; 6 { 7   &lt;...&gt; 8 } 9 10 void ArmageddonButton::draw (&lt;...&gt;) 11   &lt;...&gt; 12 } 13 14 void ArmageddonButton::update (float delta) { 15   &lt;...&gt; 16   if (pressed) { 17     press_time += delta; 18     if (press_time &gt; 1.0f) { 19       pressed = false; 20       press_time = 0; 21     } 22   } else { 23     &lt;...&gt; 24     press_time = 0; 25   } 26 } 27 28 void ArmageddonButton::on_click (&lt;...&gt;) { 29   if (pressed) { 30     send_armageddon_event(); 31   } else { 32     pressed = true; 33   } 34 } </pre>	<pre> 1 ArmageddonButton::ArmageddonButton(&lt;...&gt;) : 2   RectComponent(&lt;...&gt;), var RectComponent but = &lt;...&gt;; 3   pressed(false); 4   press_time(0); 5   &lt;...&gt; 6 { 7   &lt;...&gt; 8 } 9   end 10 void ArmageddonButton::draw (&lt;...&gt;) 11   &lt;...&gt; 12   emit game.armageddon; 13 end 14 void ArmageddonButton::update (float delta) { 15   &lt;...&gt; 16   if (pressed) { 17     press_time += delta; 18     if (press_time &gt; 1.0f) { 19       pressed = false; // give up, 1st click was 20       press_time = 0; // too long ago 21     } 22   } else { 23     &lt;...&gt; 24     press_time = 0; 25   } 26 } 27 28 void ArmageddonButton::on_click (&lt;...&gt;) { 29   if (pressed) { 30     send_armageddon_event(); 31   } else { 32     pressed = true; 33   } 34 } </pre>
--	--

[a] Implementation in C++

[b] Implementation in C  U

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

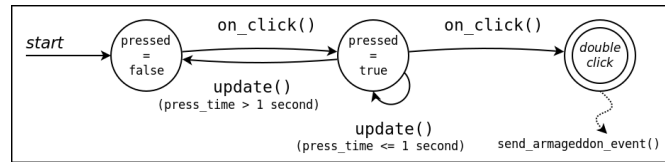


Figure 4: State machine for detecting double-clicks in the *Armageddon button*.



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

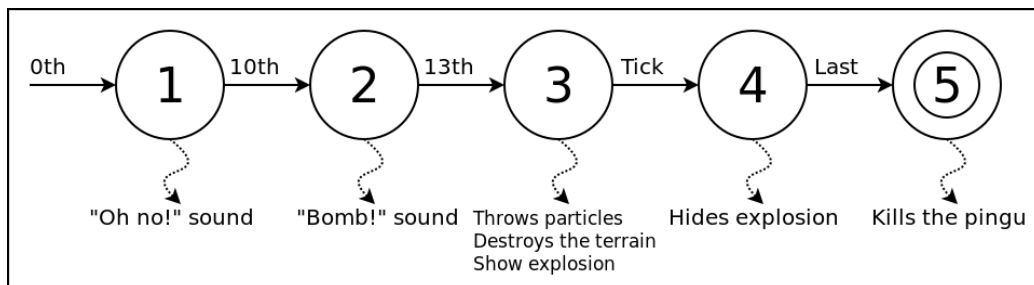


Figure 6: State machine for the *Bomber* animation sequence.

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 7: Pingu actions in C  U and C++ in terms of *locs* and state variables.

```

1 Bomber::Bomber (Pingu* p) 1 code/await Bomber (void) -> ActionName
2 <...> 2 do
3 sprite(<...>), // bomber sprite
4 sound_played(false), // track 2
5 particle_thrown(false), // track 3
6 colmap_exploded(false), // tracks state 3 // frame animation in background
7 gfx_exploded(false) // track 4
8 { 8 // 1. plays a "Oh no!" sound.
9 <...> 9 {play_sound("ohno")};
10 // 1. plays a "Oh no!" sound. 10
11 play_sound("ohno"); 11 // 2. plays a "Bomb!" sound.
12 } 12 await game.update until sprite.frame
13 13 {play_sound("plop")};
14 void Bomber::update () { 14
15 sprite.update(); 15 // 3. particles , terrain , explosion sprite
16 <...> //pingu movement 16 await game.update until sprite.frame
17 17 spawn PinguParticles(<...>) in par
18 // 2. plays a "Bomb!" sound. 18 call Game_Remove({&bomber_radius},
19 if (sprite.frame()==10 & !sound_played) {
20 sound_played = true; 20 <...>
21 play_sound("plop"); 21 spawn Sprite(<...>); //explosion
22 } 22
23 23 //4. tick: hides the explosion sprite
24 // 3. particles , terrain , explosion sprite await game.update;
25 if (sprite.frame()==13 & !particle_thrown) {
26 particle_thrown = true; await FOREVER;
27 get_world()->get_particles()->add(...); end
28 } 28
29 if (sprite.frame()==13 & !colmap_exploded) {
30 colmap_exploded = true; escape DEAD;
31 get_world()->remove(bomber_radius, <...>); end
32 } 32
33 33
34 // 5. kills the Pingu 34
35 if (sprite.is_finished (3)) {
36 pingu->set_status(PS_DEAD);
37 } 37
38 } 38
39 39
40 void Bomber::draw (SceneContext& gc) {
41 // 3. particles , terrain , explosion sprite
42 // 4. tick: hides the explosion sprite
43 if (sprite.frame()==13 & !gfx_exploded) {
44 gfx_exploded = true;
45 gc.color().draw(explosion_surf, <...>);
46 } 46

```

<pre> 1 StoryDot::StoryDot(const FileLoader &amp; reader) : 2   show_credits(false), // do not show by default 3 { 4   &lt;...&gt; 5   reader.read("credits", show_credits, is_click = await Story()); 6 } // from file 7 8 void StoryDot::on_click() { 9   &lt;...&gt; 10  push_screen(&lt;StoryScreen&gt;(show_credits)); 11  &lt;...&gt; 12 } 13 14 /// 15 16 StoryScreenComp::StoryScreenComp (&lt;...&gt;) : 17   show_credits(show_credits), 18   &lt;...&gt; 19 { 20   &lt;...&gt; 21 } 22 23 &lt;...&gt; // draw and update page 24 25 void StoryScreenComp::next_text() { 26   if (!displayed) { 27     &lt;...&gt; 28   } else { 29     &lt;...&gt; 30     if (!pages.empty()) { 31       &lt;...&gt; 32     } else { 33       if (show_credits) { 34         replace_screen(&lt;Credits&gt;(&lt;...&gt;)); 35       } else { 36         pop_screen(); 37       } 38     } 39   } 40 } </pre>	<pre> 1 loop do 2   if ret==STORY_CREDITS or ret==STORY_ 3     &lt;...&gt; 4     reader.read("credits", show_credits, is_click = await Story()); 5     if is_click and ret==STORY_CREDITS 6       &lt;...&gt; 7       await Credits(); 8     end 9     else 10    push_screen(&lt;StoryScreen&gt;(show_credits)); 11    &lt;...&gt; 12  end 13 end 14 15 16 StoryScreenComp (&lt;...&gt;) : 17   show_credits, 18   &lt;...&gt; 19 20 21 22 23 24 25 next_text() { 26   if 27     &lt;...&gt; 28   } else { 29     &lt;...&gt; 30     if 31       &lt;...&gt; 32     } else { 33       if 34         replace_screen(&lt;Credits&gt;(&lt;...&gt;)); 35       } else { 36         pop_screen(); 37       } 38     } 39   } 40 . </pre>
---	--

[a] Implementation in C++

[b] Implementation in CÉU

Figure 9: Transition from *Story* to *Credits* screen.

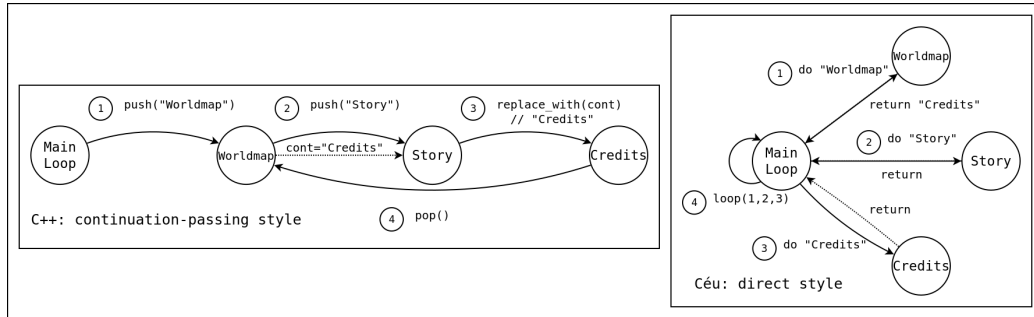


Figure 10: Continuation (C++) vs Direct (CÉU) Styles.

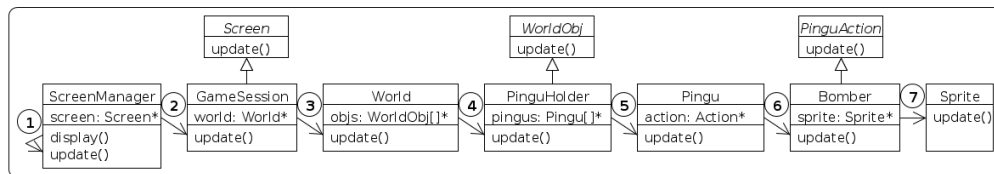


Figure 11: Dispatching chain for update.



1 <b>class</b> Bomber : <b>public</b> ActionName	1 <b>code/await</b> Bomber ( <b>void</b> ) -> ActionName
2 <...>	2 <...>
3 Sprite sprite;	3 var Sprite sprite = <b>spawn</b> Sprite(<...>)
4 }	4 <...>
5	5 <b>end</b>
6 Bomber::Bomber (<...>) : <6...> {	6 <6...> {
7 sprite.load(<...>);	7
8 <...>	8
9 }	9
10	10
11 <b>void</b> Bomber::update () {	11
12 sprite.update();	12
13 }	13
14	14
15 <b>void</b> Bomber::draw () {	15
16 <...>	16
17 sprite.draw();	17
18 }	18 .
[a] Implementation in C++	[b] Implementation in CÉU

Figure 12: Bomber action draw and update dispatching.

1 GameSession::GameSession(<...>)	1 <b>code/await</b> Game ( <b>void</b> ) <b>do</b>
2 <...>	2 <...> <i>// other coexisting functionality</i>
3 {	3 <b>spawn</b> ButtonPanel(<...>);
4 <...> <i>// these widgets are always spawned</i>	4 <b>spawn</b> PingusCounter(<...>);
5 btpanel = <b>new</b> ButtonPanel(<...>);	5 <b>spawn</b> SmallMap(<...>);
6 pcounter = <b>new</b> PingusCounter(<...>);	6 <i>// other coexisting functionality</i>
7 smallmap = <b>new</b> SmallMap(<...>);	7 <b>end</b> ;
8 <...>	8
9 uimgr->add(btpanel); <i>// ...but are added</i>	9
10 uimgr->add(pcounter); <i>// dynamically to the</i>	10
11 uimgr->add(smallmap); <i>// dispatching hierarchy</i>	11
12 <...>	12
13 }	13 .
[a] Implementation in C++	[b] Implementation in CÉU

Figure 13: Managing the UI widgets lifecycle.

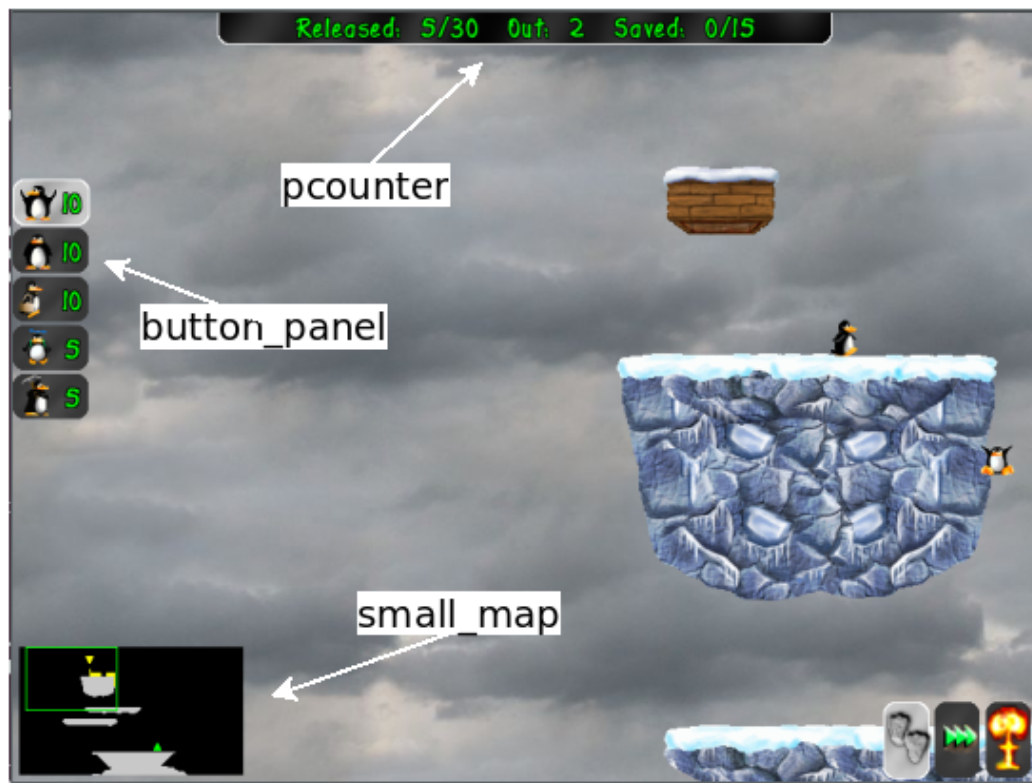


Figure 14: UI children with static lifespan.

<pre> 1 Pingu* PinguHolder::create_pingu (void) do 2     &lt;...&gt; 3     Pingu* pingu = new Pingu (&lt;pool&gt;); 4     pingus.push_back(pingu); 5     &lt;...&gt; 6 } 7 8 void PinguHolder::update() { 9     &lt;...&gt; 10    while (pingu != pingus.end()) { 11        (*pingu)-&gt;update(); 12        if ((*pingu)-&gt;get_status() == PS_DEAD) { 13            pingu = pingus.remove(pingu); 14        } 15        &lt;...&gt; 16        ++pingu; 17    } 18 } 19 20 . </pre>	<pre> 1 code/await game (void) do 2     &lt;...&gt; 3     pool[] Pingu pingus; 4     code/await Pingu_Spawn (&lt;...&gt;) do 5         &lt;...&gt; 6         spawn Pingu(&lt;...&gt;) in pingus; 7     end 8     &lt;...&gt; // code invoking Pingu_Spawn 9 end 10    end 11    code/await Pingu (&lt;...&gt;) do 12        &lt;...&gt; 13        loop do 14            await game.update; 15            if Pingu_Is_Out_Of_Screen() then 16                &lt;...&gt; 17                escape PS_DEAD; 18            end 19        end 20    end </pre>
---	--

[a] Implementation in C++

[b] Implementation in CÉU

Figure 15: Managing the pingus lifecycle.

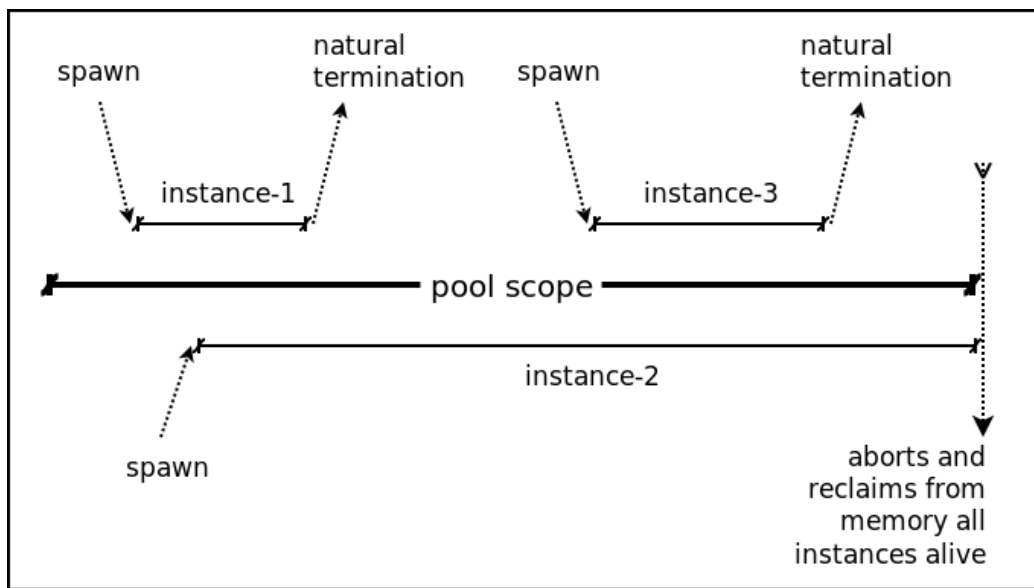


Figure 16: Lifespan of dynamic instances.