

# Structured Synchronous Reactive Programming for Game Development

## Case Study: On Rewriting Pingus from C++ to CÉU



Figure 1. Pingus gameplay.

**Abstract**—We present a qualitative 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 most games.

**Keywords**—Control Flow ; Event-Driven Programming ; Game Logic ; Synchronous Reactive Programming ;

### I. 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” [13]. The codebase<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 accounts for

10% of the CPU budget [22]. 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* [13] 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 [2], [11], [18]. In this sense, callbacks actually disrupt structured programming, becoming “our generation’s *goto*”.

In this work, we advocate structured synchronous reactive programming as a more productive alternative for game logic development. We present a qualitative case study of rewriting Pingus from C++ to CÉU.

CÉU [19], [20] is a Esterel-based [4] programming language that originally targets embedded soft real-time systems. It aims to offer a concurrent, safe, and expressive alternative to C with the characteristics that follow:

- **Reactive:** code only executes in reactions to events.
- **Structured:** programs use structured control mechanisms, such as `spawn` and `await` (to create and suspend activities).
- **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 lets developers write code in direct style, recovering from the inversion of control imposed by event-driven execution [2], [11], [18]. CÉU supports logical parallelism with a resource-efficient implementation in terms of memory and CPU usage. The runtime is single threaded and does not rely on garbage collection for memory management [20]. Existing work in the context of embedded sensor networks evaluates the expressiveness of CÉU in comparison to event-driven code in C and attests a reduction in source code size (around 25%) with a small increase in memory usage (around 5–10%) [20]. CÉU has also been used in the context of multimedia systems [21] and games [19].

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

Our case study shows gains in productivity for six selected behaviors in the game logic of Pingus rewritten in C  U. We present an in-depth qualitative analysis of the proposed solutions in comparison to the original implementations in C++. Not all techniques result in reduction of *locs* (especially considering the verbose syntax of C  U), but have other effects such as eliminating shared variables and dependencies between classes. We also identify four control-flow patterns that likely apply to most games: *Finite State Machines*, *Continuation Passing*, *Dispatching Hierarchies*, and *Lifespan Hierarchies*. A control-flow pattern is a recurring technique to describe execution dependency and/or explicit ordering between statements.

We employed a *live code rewrite*, i.e., starting from the original codebase in C++, we reimplemented it piece-by-piece in C  U without breaking the game compilation and execution. This approach shows the feasibility of a partial and gradual translation between the languages.

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

## II. THE PINGUS CODEBASE AND REWRITING PROCESS

In Pingus, the game logic accounts for almost half the size of the codebase<sup>2</sup>: 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 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++ [20]: 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>3</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 condensed *locs* spread across 70 implementation files originally written in C++<sup>3</sup>. We did the same with the implementation in C  U, resulting in 3.697 condensed *locs*<sup>3</sup>. Figure 2 summarizes the effective game logic codebase in the two implementations.

Although the analysis in this work is qualitative, the rows with lower ratio numbers in Figure 2 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++ implementing 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. Not all state follows these conventions, but they help finding classes that are heavy on control flow quickly at the beginning of the process.

## III. CONTROL-FLOW PATTERNS & CASE STUDIES

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

- 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.
- 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.

We describe six representative game behaviors in detail distributed in the four patterns and analyze their implementations in C++ and C  U.

### A. Finite State Machines

Event occurrences lead to transitions between states and trigger actions comprising the behavior of a game entity.

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.<sup>4</sup>

Figure 3.a shows the C++ implementation for the class `ArmageddonButton` with methods for rendering the button

<sup>2</sup>We used *SLOCCount* to count only non-blank, non-comment lines in the codebase: [www.dwheeler.com/sloccount/](http://www.dwheeler.com/sloccount/)

<sup>3</sup> Effective codebase: [github.com/an000/p/tree/master/](https://github.com/an000/p/tree/master/)

<sup>4</sup>Double click animation: [github.com/an000/p/#1](https://github.com/an000/p/#1)

	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, etc)
4	pingu/	884	1111	0.80	pingu behaviors
5	./	343	458	0.75	main functionality
6	actions/	541	653	0.83	pingu actions (bomber, climber, etc)
7	worldmap/	468	493	0.95	campaign worldmap
8	screens/	1109	1328	0.84	menus and screens
9	option/	347	357	0.97	option menu
10	others/	762	971	0.78	other menus and screens
11	misc/	56	46	1.22	miscellaneous functionality
		3697	4135	0.89	

Figure 2. The Pingus codebase directory tree.

```

ArmageddonButton::ArmageddonButton(<...>):
    RectComponent(<...>),
    pressed(false); // button is not initially 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 was
            press_time = 0; // too long ago
        }
    } else {
        <...>
        press_time = 0;
    }
}

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

```

and handling mouse and timer events. The code in the figure 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. The class first initializes the variable `pressed` (ln 3) to track the first click (ln 32).

```

1 do
2     var RectComponent but = <...>;
3     <...>
4     loop do
5         await but.on_click;
6         watching ls do
7             await but.on_click;
8             break;
9         end
10    end
11    <...>
12    emit game.armageddon;
13 end

```

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

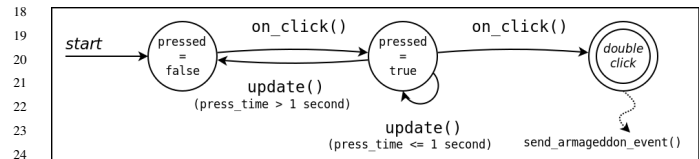


Figure 4. State machine for detecting double-clicks in the *Armageddon* button.

It also initializes the variable `press_time` (ln 4) to count the time since the first click (ln 16–17). If another click occurs within 1 second, the class signals the double click to the application (ln 29–30). Otherwise, the `pressed` and `press_time` state variables are reset (ln 18–21). Figure 4 illustrates how we can model the double-click behavior in C++ as a state machine. The circles represent the state of the variable `pressed`, and the arrows represent the callbacks manipulating it. Note in Figure 3.a how the accesses to the state variables are spread across the entire class: 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.

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 5. Pingus actions in C  U and C++ in terms of *locs* and state variables.

C  U supports structured constructs to deal with events, aiming to eradicate explicit manipulation of state variables for control-flow purposes. In Figure 3.b, the loop to detect double clicks (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 in sequence 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 C  U does not rely on state variables and is entirely self-contained in the loop body. Also, those 7 lines of code *only* detect the double click, leaving the actual effect (ln 12) as well as all unrelated code (such as redrawing the button) to happen outside the loop.

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

2) *Summary & Pattern Uses in Pingus*: In comparison to explicit state machines, the structured constructs of C  U introduce some advantages as follows:

- They encode all states with direct sequential code, eliminating callbacks and shared state variables for control-flow purposes.
- 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 [13]. However, this approach is not fundamentally different from Pingus’ use of switch or if branches to decode state.

Pingus supports 16 actions in the game. Five of them implement at least one state machine and are considerable smaller in C  U in terms of *locs* (Figure 5). For the other 11 actions without state machines, the reduction in *locs* is negligible. This asymmetry illustrates the gains in expressiveness when describing state machines in direct style.

Among all 65 implementation files in C  U, we found

```

StoryDot::StoryDot(FileReader& reader) :
    show_credits(false), // do not show by default
{
    <...>
    reader.read("credits", show_credits);
    // from file

void StoryDot::on_click() {
    <...>
    push(<StoryScreen>(show_credits));
    <...>
}

// class separator ///

StoryScreenComp::StoryScreenComp (<...>) :
    show_credits(show_credits),
    <...>
{
    <...>
}

<...> // draw and update page

void StoryScreenComp::next_text() {
    if (!displayed) {
        <...>
    } else {
        <...>
        if (!pages.empty()) {
            <...>
        } else {
            if (show_credits) {
                replace(<Credits>(<...>));
            } else {
                pop();
            }
        }
    }
}

}
%
```

29 cases in 25 files that use structured mechanisms to substitute states machines. They typically manifest as await statements in sequence (e.g., ln 5,7 in Figure 3 and ln 12,16,24 in Figure ??).

## B. Continuation Passing

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

1) *Transition from Story to Credits Screen*: The campaign world map has clickable blue dots in the two extremes of the map road to show introductory and closing ambience stories, respectively. For introductory stories, the game returns to the world map after showing the story pages. For closing stories, the game also shows a *Credits screen* before returning to the world map.<sup>5</sup>

In C++, the class StoryDot in Figure 6.a (ln 1–12) first reads the level file (ln 5) to check whether it is a closing story and should, after termination, show the *Credits screen*. The boolean variable show\_credits (ln 2,5,10) is passed

<sup>5</sup>Credits screen animation: [github.com/an000/p/#4](https://github.com/an000/p/#4)

---

```

loop do
  var int ret = await Worldmap();
  if ret==CREDITS or ret==BACK then
    <...>
    var bool is_click = await Story();
    if is_click and ret==CREDITS then
      <...>
      await Credits();
    end
  else
    <...>
  end
end

```

---

Figure 6. Transition from *Story* to *Credits* screen.

to the class `StoryScreen` (ln 10) and represents the screen continuation, i.e., what to do after showing the story. The class `StoryScreen` (not shown) then forwards the continuation even further to the auxiliary class `StoryScreenComp` (ln 16–40). When the method `next_text` has no story pages left to display (ln 32–38), it decides where to go next, depending on the continuation flag `show_credits` (ln 33).

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

Figure 7 illustrates the *continuation-passing style* of C++ and the *direct style* of C  U for screen transitions:

- 1) Main Loop  $\longrightarrow$  `Worldmap`:
  - C++ uses an explicit stack to push the `Worldmap` screen (not shown in Figure 6.a).
  - C  U invokes the `Worldmap` screen expecting a return value (Figure 6.b, ln 2).
- 2) `Worldmap` (*blue dot click*)  $\longrightarrow$  `Story`:
  - C++ pushes the `Story` screen passing the continuation flag (Figure 6.a, ln 10).
  - C  U stores the `Worldmap` return value and invokes the `Story` screen (Figure 6.b, ln 2,5).
- 3) `Story`  $\longrightarrow$  `Credits`:
  - C++ replaces the current `Story` screen with the `Credits` screen (Figure 6.a, ln 34).
  - C  U invokes the `Credits` screen after the `await Story` returns (Figure 6.b, ln 8).
- 4) `Credits`  $\longrightarrow$  `Worldmap`:
  - C++ pops the `Credits` screen, going back to the `Worldmap` screen (not shown in Figure 6.a).
  - C  U uses an enclosing loop to restart the process (Figure 6.b, ln 1–13).

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

2) *Summary & Pattern Uses in Pingus*: The direct style of C  U has some advantages in comparison to the continuation-passing style of C++:

- It uses structured control flow (i.e., sequences and loops) instead of explicit data structures (e.g., stacks) and continuation variables (e.g. boolean flags).
- The activities in sequence 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 in C++, such as screen transitions in menus and level progressions. C  U adopts the direct style technique 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 between *pingu* actions during gameplay (e.g., *walking* to *falling* and back to *walking*).

### C. Dispatching Hierarchies

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

1) *Case Study: Bomber Action draw and update Dispatching*: In C++, the class `Bomber` in Figure 8.a declares a `sprite` member (ln 3) to handle its animation frames. 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 to the game down to the `sprite`, we need to follow a long chain of 7 method dispatches (Figure 9):

- 1) `ScreenManager::display` in the main game loop calls `ScreenManager::update` when starting a new frame.
- 2) `ScreenManager::update` calls `screen->update` for the active game screen (i.e., a `GameSession` instance, considering the screen in which the `Bomber` appears).
- 3) `GameSession::update` calls `world->update`.
- 4) `World::update` calls `objs->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.

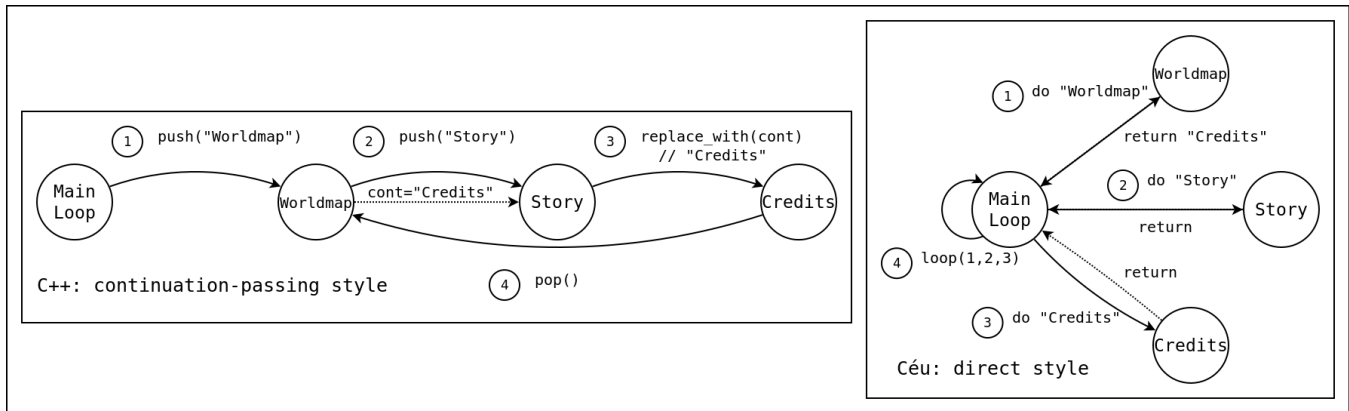


Figure 7. Continuation (C++) vs Direct (CÉU) Styles.

```

class Bomber : public Action {
    <...>
    Sprite sprite;
}

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

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

void Bomber::draw () {
    <...>
    sprite.draw();
}

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

```

Figure 8. Bomber action draw and update dispatching.

- 7) Bomber::update calls sprite.update. Sprite::update finally updates the animation frame.

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

- With a single assignment to `screen`, one can easily deactivate the current screen and redirect all dispatches to a new screen (step 2).
- The `World` class manages and dispatches events to all game entities with a common interface `WorldObj`, such as the `pingus` and `traps` (step 4).
- Since it is common to iterate only over the `pingus` (vs. all world objects), the container `PinguHolder` manages all `pingus` (step 5).

- Since a single `pingu` can change its actions during lifetime, the `action` member decouples them with another level of indirection (step 6).
- 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 (step 7).

Like `update`, the `draw` callback also flows through a similar dispatching hierarchy until reaching the `Sprite` class.

In CÉU, the `Bomber` abstraction presented in Figure 8.b spawns a `Sprite` animation instance on its body (ln 3). The `Sprite` abstraction can react directly to external `update` and `draw` events, bypassing the program hierarchy entirely. Events in CÉU are broadcasted to the entire application in lexical order, i.e., an abstraction that appears first in the source code (e.g., ln 3) reacts before another one that appears second (e.g., hidden in ln 4). This rule preserves determinism and also conforms to the program static hierarchy. While (and only while) the bomber abstraction is alive, the sprite animation remains alive and reacts to the `update` and `draw` events. The radical decoupling between the program hierarchy and reactions to events eliminates dispatching chains entirely.

2) *Summary & Pattern 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).

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.

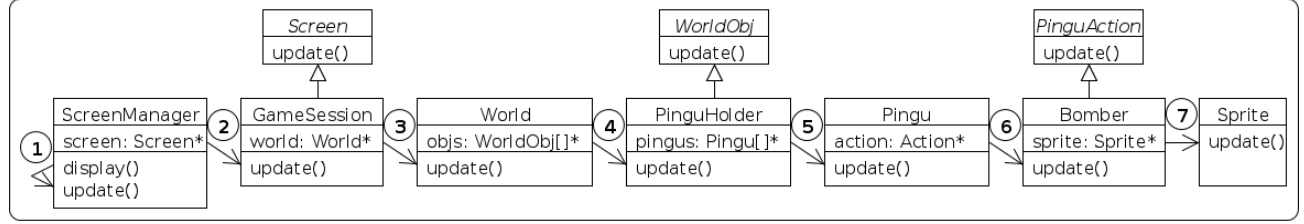


Figure 9. Dispatching chain for `update`.

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 also rely heavily on this control-flow pattern. In the case of the *Pingus* engine, we rewrote 9 files with a reduction from 515 to 173 *locs* (not listed in Figure 2), mostly due to dispatching code removal.

#### D. Lifespan Hierarchies

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

1) *Case Study: Dynamic Pingus Lifecycle*: A *pingu* is a dynamic entity created periodically and destroyed under certain conditions, such as falling from a high altitude.<sup>6</sup>

In C++, the class `PinguHolder` in Figure 10.a is a container that holds all alive *pingus*. 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). Note that if the programmer disregards the call to `remove`, a dead *pingu* would remain in the collection and still update on every frame (ln 11). Since the `draw` behavior for a dead *pingu* is innocuous, the death could go unnoticed when testing it but the program would keep consuming memory and CPU time. This problem is known as the *lapsed listener* [13] 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. Hence, entities with dynamic lifespan always require explicit matching `add` and `remove` calls associated to a container (ln 4,13).

CÉU supports `pool` declarations to hold dynamic abstraction instances. In addition, the `spawn` statement supports a pool identifier to associate a new instance with a pool. The game screen in Figure 10.b spawns a new *Pingu* on every invocation of `Pingu_Spawn` (ln 4–7). 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), surviving the call to `Pingu_Spawn`. 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 11). 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.

2) *Summary & Pattern 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. All entities in a game have an associated lexical lifespan.
- The memory for static instances is known at compile time.
- Natural termination makes an instance innocuous and, hence, susceptible to immediate reclamation.
- Instances (static or dynamic) never require explicit manipulation of pointers/references.

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 loaded from files, gameplay & worldmap objects, and also UI widgets.

<sup>6</sup>Death of *pingus* animation: [github.com/an000/p/#5](https://github.com/an000/p/#5)

```

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

void PinguHolder::update() {
    <...>
    while(pingu != pingus.end()) {
        (*pingu)->update();
        if ((*pingu)->status() == DEAD) {
            pingu = pingus.remove(pingu);
        }
        <...>
        ++pingu;
    }
}

```

[a] Implementation in C++

```

1  code/await Game (void) do
2      <...>
3      pool[] Pingu pingus;
4      code/await Pingu_Spawn (<...>) do
5          <...>
6          spawn Pingu(<...>) in pingus;
7          end
8          <...> // code invoking Pingu_Spawn
9      end
10
11  code/await Pingu (<...>) do
12      <...>
13      loop do
14          await game.update;
15          if Pingu_Is_Out_Of_Screen() then
16              <...>
17              escape PS_DEAD;
18          end
19      end
20  end

```

[b] Implementation in C  U

Figure 10. Managing the pingus lifecycle.

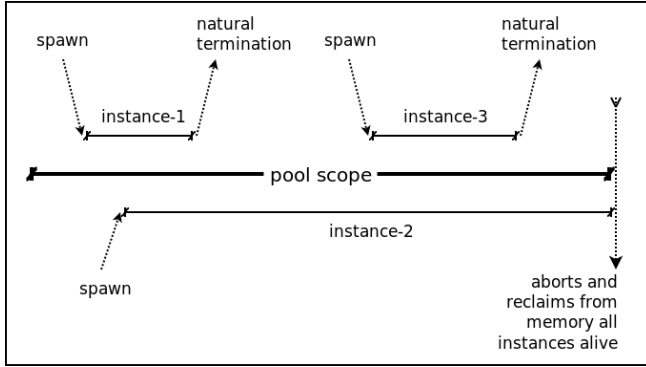


Figure 11. Lifespan of dynamic instances.

#### IV. RELATED WORK

The control-flow patterns presented in this paper closely relate to the *GoF* behavioral patterns [9], which are discussed in the context of video games in previous work [3], [13], [17]. The original *Pingus* in C++ uses variations of the patterns *state* (Sections III-A and III-B), *visitor* (Sections III-C and III-D), and *observer* (to handle events in general) as implementation techniques to achieve the desired higher-level control-flow patterns described in the paper. C  U overcomes the need of behavioral patterns with support, at the language level, for structured control-flow mechanisms and event-based communication via broadcast.

A number of domain-specific languages, frameworks, and techniques have been proposed for particular subsystems of the game logic, such as animations [7], [14]–[16], game state and screen progression [12], [23], and behavior and AI modeling [1], [10]. In *Pingus*, the adoption of C  U is not restricted to a specific subsystem. We employed C  U at the very core of the game for event dispatching

(Section III-C) and memory management of entities (Section III-D), eliminating parts of the original game engine. We also implemented all entity animations and behaviors (Section III-A), and screen transitions (Section III-B) using the available control mechanisms of C  U. Furthermore, C  U is a superset of C targeting reactive systems in general, not only games, and has also been successfully adopted in other domains, such as wireless sensor networks [5], [20] and multimedia systems [21].

Functional reactive programming (FRP) [8] contrasts with structured synchronous reactive programming (SSRP) 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 or control-flow dependencies 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 some performance considerations [6]. Although we do not provide a performance evaluation (*Pingus* is not performance sensitive), existing work on C  U shows that it is comparable to C in the context of embedded systems [20]. Nonetheless, given the tight integration between C  U and C/C++, critical parts of games can be preserved in C++ if needed.

#### V. CONCLUSION

We advocate *Structured Synchronous Reactive Programming* as a productive alternative for game logic development. We use the video game *Pingus* as a qualitative case study. We compare the implementation of six game behaviors in



C++ and CÉU and discuss how structured reactive mechanisms can eliminate callbacks and let programmers write code in direct style. Ultimately, we rewrote about 1/4 of the whole codebase (9.186 from 39.362 lines of code) which comprises the core of the game logic that is susceptible to structured reactive programming.

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

Overall, we most difficulties in implementing control-flow 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 develop event-based applications.

## 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] F. Boussinot and R. De Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, Sep 1991.
- [5] 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.
- [6] M. H. Cheong. Functional programming and 3D games. Master’s thesis, University of New South Wales, Australia, November 2005.
- [7] F. Devillers and S. Donikian. A scenario language to orchestrate virtual world evolution. In *Proceedings Eurographics’03*, 2003.
- [8] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of ICFP’97*, pages 263–273, New York, NY, 1997. ACM.
- [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented languages and systems*. Addison-Wesley Reading, 1994.
- [10] D. Isla. Handling Complexity in the Halo 2 AI. In *Game Developers Conference*, Mar. 2005.
- [11] I. Maier, T. Rompf, and M. Odersky. Deprecating the observer pattern. Technical report, 2010.
- [12] W. Mallouk and E. Clua. An object-oriented approach for hierarchical state machines. In *Proceedings of SBGames’06*, pages 8–10, 2006.
- [13] R. Nystrom. *Game Programming Patterns*. Genever Benning, 2014.
- [14] L. Pagliosa. A new programming environment for dynamics-based animation. In *Proceedings of SBGames’06*. SBC, 2006.
- [15] K. Perlin and A. Goldberg. Improv: A system for scripting interactive actors in virtual worlds. In *Proceedings of SIGGRAPH’96*, pages 205–216. ACM, 1996.
- [16] C. W. Reynolds. Computer animation with scripts and actors. In *Proceedings of SIGGRAPH’82*, volume 16, pages 289–296. ACM, 1982.
- [17] G. R. Roberto Figueiredo. GOF design patterns applied to the development of digital games. In *Proceedings of SBGames’15*. SBC, 2015.
- [18] 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.
- [19] F. Sant’Anna, N. Rodriguez, and R. Ierusalimschy. Structured Synchronous Reactive Programming with Céu. In *Proceedings of Modularity’15*, 2015.
- [20] F. Sant’Anna, N. Rodriguez, R. Ierusalimschy, O. Landsiedel, and P. Tsigas. Safe System-level Concurrency on Resource-Constrained Nodes. In *Proceedings of SenSys’13*. ACM, 2013.
- [21] R. Santos, G. 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.
- [22] T. Sweeney. The next mainstream programming language: a game developer’s perspective. *ACM SIGPLAN Notices*, 41(1):269–269, 2006.
- [23] L. Valente, A. Conci, and B. Feijó. An architecture for game state management based on state hierarchies. In *Proceedings of SBGames’06*. Citeseer, 2006.