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

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

Figure 1: Pingus gameplay.

ABSTRACT

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 demonstrate gains in productivity for six behaviors in the game logic of Pingus through a qualitative analysis of the proposed solutions in CÉU in comparison to the original implementations 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 codebase[1] is about 40.000 lines of code (LoCs), divided into the engine, level editor, auxiliary libraries, and the game logic itself.

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

Object-oriented games rely on the *observer pattern* [12] to handle events from the environment (e.g., key presses and timers) and also as a notification mechanism between entities in the game logic. The observers are short-lived callbacks that have to execute as fast as possible to keep the game reactive to incoming events in real time. For this reason, callbacks cannot 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*".[2]

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], which has the following characteristics:

- *Reactive*: code only executes in reactions to events.
- *Structured*: programs use structured control-flow mechanisms, such as `spawn` and `await` (to create and suspend an activity).
- *Synchronous*: reactions run atomically and to completion on each activity. There is no implicit preemption or real parallelism, resulting in deterministic execution.

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

Our case study demonstrates gains in productivity for six selected behaviors in the game logic of Pingus through idiomatic code in CÉU. We present an in-depth qualitative analysis of the proposed solutions in comparison to the original implementations in C++. We also identify four recurrent control-flow patterns that likely apply to other games: *Finite State Machines*, *Continuation Passing*, *Dispatching Hierarchies*, and *Lifespan Hierarchies*. A control-flow pattern is a recurring technique to describe dependencies and explicit orders between statements (or groups of statements) execution in a program.

The rewriting process consisted of identifying sets of callbacks implementing control flow in the game and translating them to CÉU using appropriate structured constructs. As an example, a double mouse click is characterized by a first click, followed by a maximum amount of time, followed by a second click. This behavior depends on different events (clicks and timers) which have to occur in a particular order. In C++, the implementation involves callbacks crossing reactions to successive events which manipulate state variables explicitly. We focus on a qualitative analysis for the programming techniques that we applied during the rewriting process. Not all techniques result in reduction in LoCs (especially considering

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

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

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

Figure 2: The Pingus codebase directory tree.

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

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

2 The Pingus Codebase

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

Although our analysis is mostly qualitative, the lines in Figure 2 with lower ratio numbers correlate to the parts of the game logic that we consider more susceptible to structured reactive programming. For instance, the *Pingu* behavior (*ratio 0.80*) contains complex animations that are affected by timers, game rules, and user interaction. In contrast, the *Option screen* (*ratio 0.97*) is a simple UI grid with trivial mouse interactions.

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

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

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

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

3 Control-Flow Patterns & Case Studies

During the course of the rewriting process, we have identified four abstract 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.
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 analyse their implementations in C++ and CÉU.[6]

3.1 Finite State Machines

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

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

In Pingus, a double click in the *Armageddon button* at the bottom right of the screen literally explodes all pingus.[7]

Figure 3.a shows the C++ implementation for the class `ArmageddonButton` with methods for rendering the button and handling mouse and timer events. The code focus on the double click detection and hides unrelated parts with `<...>`. The methods `update` (ln. 14–26) and `on_click` (ln. 28–34) are examples of *short-lived callbacks*, which are pieces of code that execute atomically in reaction to external input events. The callback `on_click` reacts to mouse clicks detected by the base class `RectComponent` (ln. 2), while the callback `update` continuously reacts to the passage of time, frame by frame. The class first initializes the variable `pressed` (ln. 3) to track the first click (ln. 32). It also initializes the variable

[6]Due to space constraints, we omit a fifth pattern *Signaling Mechanisms* and three other game behaviors.

[7]Double click animation: `github.com/an000/p/#1`

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

[a] Implementation in C++

```
1  do
2      var RectComponent r = <...>;
3      <...>
4      loop do
5          await r.on_click;
6          watching 1s do
7              await r.on_click;
8              break;
9          end
10     end
11     <...>
12     emit game.armageddon;
13 end
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34 .
```

[b] Implementation in CÉU

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

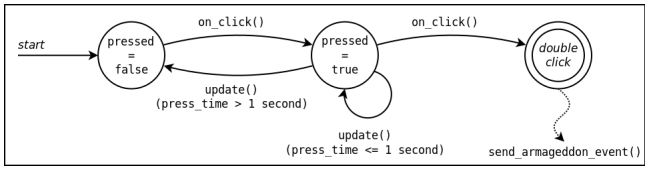

Figure 4: State machine for detecting double-clicks in the *Armageddon button*.

`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 variables in the class, while the arrows represent the callbacks manipulating state. Note in Figure 3 how the accesses to the state variables are spread across the entire class. For instance, the distance between the initialization of `pressed` (ln. 3) and the last access to it (ln. 32) is over 40 lines in the original file. Arguably, this dispersion of code across methods makes the understanding and maintenance of the double-click behavior more difficult. Also, even though the state variables are private, unrelated methods such as `draw`, which is defined in middle of the class (ln. 10–12), can potentially access them.

CÉU 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 CÉU does not rely on state variables and is entirely self-contained in the `loop` body. Also, these 7 lines of code *only* detect the double click, leaving the actual effect to happen outside the loop (ln. 12), as well as all unrelated code such as redrawing the button.

3.1.2 Case Study: The *Bomber Action* Animation Sequence

The player may assign actions to specific pingus, as illustrated in Figure 5. The *Bomber action* explodes the clicked pingu, throwing particles around and also destroying the terrain under its radius.[8] We can model the explosion animation with a sequential state machine (Figure 6) with effects associated to specific frames as follows[9]:

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

[8] Bomber action animation: `github.com/an000/p/#2`
[9] State machine animation: `github.com/an000/p/#3`

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

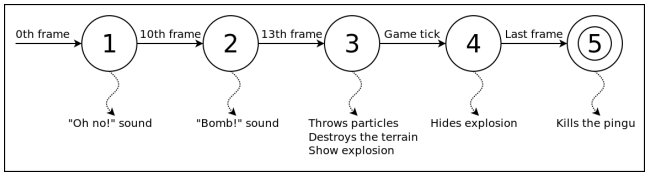

Figure 6: State machine for the *Bomber animation* sequence.

5. Last frame: kills the pingu.

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

The equivalent code in CÉU for the *Bomber action* in Figure 8.b does not rely on state variables and reflects the sequential state machine implicitly, using `await` statements to separate the effects in direct style. The `Bomber` is a `code/await` abstraction of CÉU, which is similar to a coroutine or fiber [2]: a subroutine that retains runtime state, such as local variables and the program counter, across reactions to events (i.e., across `await` statements). The pingu movement and sprite animation are isolated in two other `code/await` abstractions and execute in separate through the `spawn` primitive (ln. 4–5). In CÉU, if multiple abstractions react to the same event, the scheduler employs lexical order to preserve determinism, i.e., the `spawn` that appears first in the source code executes first [18]. The event `game.update` (ln. 12,16,24) is analogous to the `update` callback of C++ and occurs on every game frame. The code tracks the animation aliveness (ln. 7–27) and, on termination, performs the last bomber effect, killing the pingu (ln. 30). As soon as the animation starts, the code performs the first effect (ln. 9). The intermediate effects are performed when the corresponding conditions occur (ln. 12,16,24). The `do-end` block (ln. 19–25), restricts the lifespan of the single-frame explosion sprite (ln. 21): after the next game tick (ln. 24), the block terminates and automatically destroys the spawned abstraction (removing it from the screen). In contrast with the implementation in C++, all effects occur in a contiguous chunk of code (ln. 7–30), which handles no extra functionality.

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: Pingus actions in CÉU and C++.

3.1.3 Summary & Pattern Uses in Pingus

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

- They encode all states with direct sequential code, eliminating 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 [12]. However, this approach is not fundamentally different from Pingus' use of `switch` or `if` branches for each possible state.

Pingus supports 16 actions in the game. As Figure 7 shows, 5 of them implement at least one state machine and are considerable smaller in CÉU in terms of LoCs after eliminating the state variables. Considering the other 11 actions, the reduction in LoCs is negligible. This asymmetry in the implementation of actions illustrates the gains in expressiveness when describing state machines in direct style.

Among all 65 implementation files in CÉU, we found 29 cases in 25 files using structured mechanisms to substitute states machines. They typically manifest as `await` statements in sequence.

3.2 Continuation Passing

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

3.2.1 Transition to the *Credits Screen* from the *Story Screen*

The campaign world map has clickable blue dots in the two extremes of the progress trail for introductory and ending ambience stories, respectively. For introductory stories, the game returns to the world map after displaying the story pages. For ending stories, the game also displays a *Credits screen* before returning to the world map.[10]

In C++, the class `StoryDot` in Figure 9.a (ln. 1–12) first reads the level file (ln. 5) to check whether its an ending story and should, after termination, display the *Credits screen*. The boolean variable `credits` is passed to the class `StoryScreen` (ln. 10) and represents the screen continuation, i.e., what to do after displaying 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 pages left to display (ln. 32–38), it decides where to go next, depending on the continuation flag `credits` (ln. 33).

[10]Credits screen animation: `github.com/an000/p/#4`

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

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

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

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

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

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

[a] Implementation in C++

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

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

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

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

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

[b] Implementation in CÉU

Figure 8: The *Bomber action* sequence.

In CÉU, the `loop` of Figure 9.b controls the flow between the screens to display as a direct sequence of statements. We first invoke the `Worldmap` (ln. 2), which exhibits the map and let 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). Finally, we check the returned values (ln. 6) to perhaps display the `Credits` screen (ln. 8). The enclosing loop restores the `Worldmap` and repeats the process.

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

1. `Main Loop` ⟶ `Worldmap`:
   - C++ uses an explicit stack to push the `Worldmap` screen.
   - CÉU invokes the `Worldmap` screen expecting a return value (ln. 2).
2. `Worldmap` (*blue dot click*) ⟶ `Story`:
   - C++ pushes the `Story` screen passing the continuation flag (ln. 10).
   - CÉU stores the `Worldmap` return value and invokes the `Story` screen (ln. 2,5).
3. `Story` ⟶ `Credits`:
   - C++ replaces the current `Story` screen with the `Credits` screen (ln. 34).
   - CÉU invokes the `Credits` screen after the `await Story` returns (ln. 8).
4. `Credits` ⟶ `Worldmap`:
   - C++ pops the `Credits` screen, going back to the `Worldmap` screen. CÉU uses an enclosing `loop` to restart the process (ln. 1–13).

In contrast with C++, the screens in CÉU are completely decoupled and only the `Main Loop` touches them: the `Worldmap` has no

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

void StoryDot::on_click() {
  <...>
  push_screen(<StoryScreen>(<...>, credits));
  <...>
}

///

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

<...>   // draw and update page

void StoryScreenComp::next_text() {
  if (!displayed) {
    <...>
  } else {
    <...>
    if (!pages.empty()) {
      <...>
    } else {
      if (credits) {
        replace_screen(<Credits>(<...>));
      } else {
        pop_screen();
      }
    }
  }
}
```

[a] Implementation in C++

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

[b] Implementation in CÉU

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

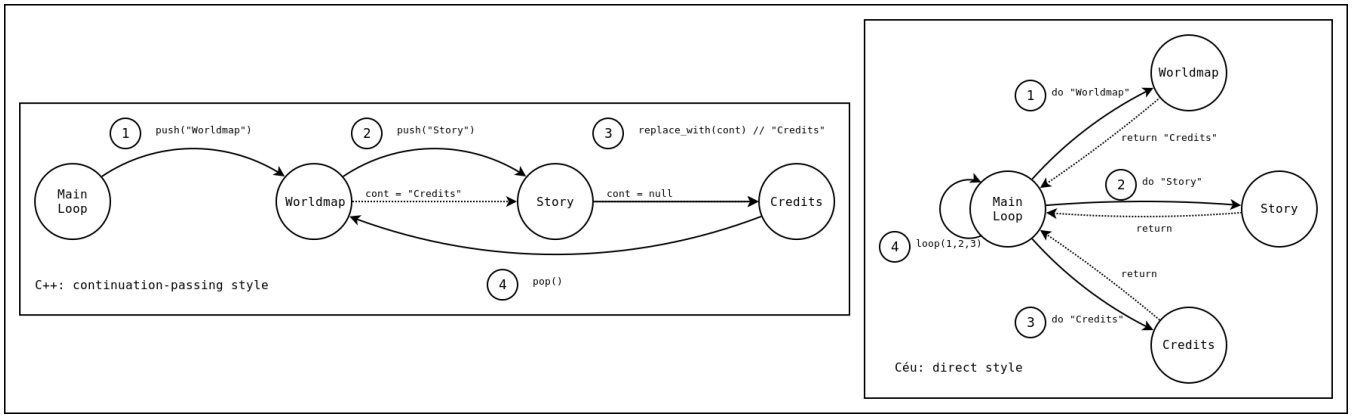

Figure 10: Continuation (C++) vs Direct (CÉU) Styles.

references to `Story`, which has no references to `Credits`.

3.2.2 Summary & Pattern Uses in Pingus

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

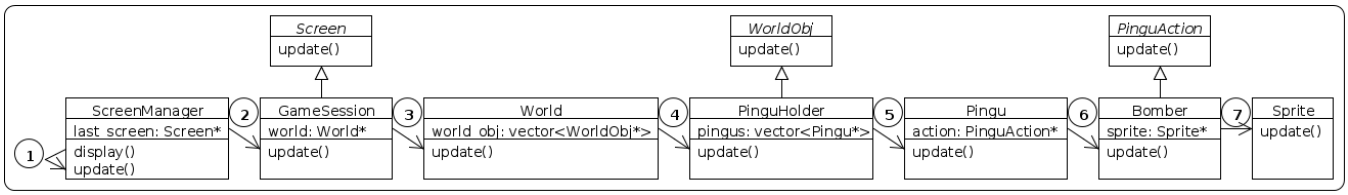

Figure 11: Dispatching chain for `update`.

```
1  class Bomber : public PinguAction {
2      <...>
3      Sprite sprite;
4  }
5
6  Bomber::Bomber (<...>) : <...> {
7      sprite.load(<...>);
8      <...>
9  }
10
11 void Bomber::update () {
12     sprite.update();
13 }
14
15 void Bomber::draw (SceneContext& gc) {
16     <...>
17     gc.color().draw(sprite, <...>);
18 }
```

[a] Implementation in C++

```
1  code/await Bomber (void) -> ActionName do
2      <...>
3      var Sprite sprite = spawn Sprite(<...>);
4      <...>
5  end
6
7
8
9
10
11
12
13
14
15
16
17
18 .
```

[b] Implementation in CÉU

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

- 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, such as screen transitions in menus and level progressions. CÉU uses the direct style techniques in five cases involving screen transitions: the main menu, the level menu, the level set menu, the world map loop, and the gameplay loop. It also uses the same technique for the loop that switches the pingu actions during gameplay (e.g., *walking* to *jumping* and back to *walking*).

3.3 Dispatching Hierarchies

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

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

In C++, the class `Bomber` in Figure 12.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 11):

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

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

- With a single assignment to `last_screen`, we can easily deactivate the current screen and redirect all dispatches to a new screen (step 2).
- The `World` class manages and dispatches events to all game entities with a common interface `WorldObj`, such as all 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).

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

In CÉU, the `Bomber` abstraction presented in Figure 12.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. External events in CÉU are broad-

```
1  GameSession::GameSession(<...>) :
2      <...>
3  {
4      <...>       // these widgets are always active...
5      btpanel  = new ButtonPanel(<...>);
6      pcounter = new PingusCounter(<...>);
7      smallmap = new SmallMap(<...>);
8      <...>
9      uimgr->add(btpanel);  // ...but are added
10     uimgr->add(pcounter); //  dynamically to the
11     uimgr->add(smallmap); //  dispatching hierarchy
12     <...>
13 }
```

[a] Implementation in C++

```
1  code/await Game (void) do
2      <...> // other coexisting functionality
3      spawn ButtonPanel(<...>);
4      spawn PingusCounter(<...>);
5      spawn SmallMap(<...>);
6      <...> // other coexisting functionality
7  end
8
9
10
11
12
13 .
```

[b] Implementation in CÉU

Figure 13: Managing the UI widgets lifecycle.

casted to the entire application. While (*and only while*) the bomber abstraction is alive, the sprite animation remains alive. The radical decoupling between the program hierarchy and reactions to events eliminates dispatching chains entirely. Note that one can still declare a local event and restrict its visibility like a local variable.

3.3.2 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. 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 from C++ to CÉU, reducing them from 515 to 173 LoCs (not listed in Figure 2), mostly due to dispatching code removal.

3.4 Lifespan Hierarchies

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

3.4.1 Case Study: Game UI Widgets

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

In C++, the widgets are created in the constructor of the class `GameSession` in Figure 13.a (ln. 5–7), added to a UI container (ln. 9–11), and are never removed since they must always be visible. Arguably, to better express the intent of making them coexist with the game screen, the widgets could alternatively be declared as top-level automatic (non-dynamic) members. However, the class relies on a container to automate `draw` and `update` dispatching to the widgets, as discussed in Section 3.3. The container method `add` expects only dynamically allocated children because they are automatically deallocated inside the container destructor. However, the dynamic nature of containers in C++ demand extra caution from programmers:

Figure 14: UI children with static lifespan.

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

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

3.4.2 Case Study: Managing the Pingus Lifecycle

A pingu is a dynamic entity created periodically and destroyed under certain conditions, such as falling from a high altitude.[11]

In C++, the class `PinguHolder` in Figure 15.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

[11] Death of pingus animation: `github.com/an000/p/#5`

```
1  Pingu* PinguHolder::create_pingu (<...>) {
2      <...>
3      Pingu* pingu = new Pingu (<...>);
4      pingus.push_back(pingu);
5      <...>
6  }
7
8  void PinguHolder::update() {
9      <...>
10     while(pingu != pingus.end()) {
11         (*pingu)->update();
12         if ((*pingu)->get_status() == PS_DEAD) {
13             pingu = pingus.remove(pingu);
14         }
15         <...>
16         ++pingu;
17     }
18 }
19
20 .
```

[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 CÉU

Figure 15: Managing the pingus lifecycle.

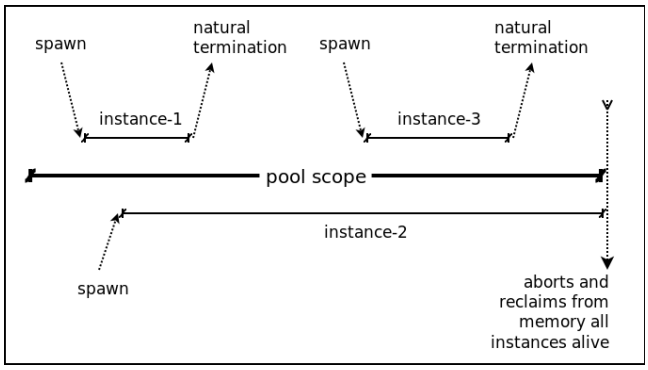

Figure 16: Lifespan of dynamic instances.

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). Without the `remove` call, a dead pingu would remain in the collection still with updates 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* [12] and also occurs in languages with garbage collection: a container typically holds a strong reference to a child (sometimes the only reference to it), and the runtime cannot magically detect it as garbage. Entities with dynamic lifespan in C++ always require explicit `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 the new instance with a pool. The game screen in Figure 15.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). Since pools are also subject to lexical scope, the lifespan of all dynamically allocated pingus is constrained to the game screen. Lexical scopes handle memory and event dispatching automatically for static instances and also for pools. However, the lifespan of a dynamic instance does not necessarily have to match the lifespan of its associated pool (Figure 16). In CÉU, when the execution block of a dynamic instance terminates, which characterizes its *natural termination*, the instance is automatically removed from its pool. Therefore, dynamic instances do not require any extra bookkeeping related to containers or explicit deallocation. To remove a pingu from the game in CÉU, we just need to terminate its execution block according to the appropriate conditions: The `escape` statement (ln. 17) aborts the execution block of the `Pingu` instance, removing it from its associated pool automatically. Hence, a dynamic instance that terminates naturally leaves no traces in the program.

3.4.3 Summary & 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.
- Abstraction 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 from files, gameplay & worldmap objects, and UI widgets.

4 Related Work

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

bomber animation in C++ in Section 3.1 becomes a series of blocks separated by `await` statements in CÉU.

A number of domain-specific languages, frameworks, and techniques have been proposed for particular subsystems of the game logic, such as animations [13, 6, 14, 15], game state and screen progression [22, 11], and behavior and AI modeling [9, 1]. In Pingus, the adoption of CÉU is not restricted to a specific subsystem. We employed CÉU at the very core of the game for event dispatching (Section 3.3) and memory management of entities (Section 3.4), eliminating parts of the original game engine. We also implemented all entity animations and behaviors (Section 3.1), and screen transitions (Section 3.2) using the available control mechanisms of CÉU. Furthermore, CÉU 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 [19, 4] and multimedia systems [20].

Functional reactive programming (FRP) [7] 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 performance considerations [5]. Instead, we rewrote an existing game and did it in small steps while keeping it working. Although we do not provide a performance evaluation (Pingus is not performance sensitive), previous work on CÉU shows that it is comparable to C in the context of embedded systems [19]. Nonetheless, given the tight integration between CÉU and C/C++, critical parts of games can be preserved in C++ if needed.

5 Conclusion

We advocate *Structured Synchronous Reactive Programming* as a productive alternative for game logic development. 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.

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 in the same contiguous block, 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 can describe the flow of the game with sequential statements in a self-contained block, eliminating explicit data structures and continuation variables (e.g., stacks and boolean flags) spread across multiple classes. *Dispatching hierarchies* spread 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. In C++, a considerable amount of interleavning code in the game logic and engine is just destinated to event dispatching. *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. This reduces explicit manipulations of pointers and references considerably.

Overall, we believe that 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 handle event-based applications.

6 Acknowledgments

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

References

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

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

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

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

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

[6] F. Devillers and S. Donikian. A scenario language to orchestrate virtual world evolution. In *Proceedings Eurographics'03*, pages 265–275, 2003.

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

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

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

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

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

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

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

[14] K. Perlin and A. Goldberg. Improv: A system for scripting interactive actors in virtual worlds. In *Proceedings of SIGGRAPH'96*, pages 205–216. ACM, 1996.

[15] C. W. Reynolds. Computer animation with scripts and actors. In *Proceedings of SIGGRAPH'82*, volume 16, pages 289–296. ACM, 1982.

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

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

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

[19] F. Sant'Anna, N. Rodriguez, R. Ierusalimschy, O. Landsiedel, and P. Tsigas. Safe System-level Concurrency on Resource-Constrained Nodes. In *Proceedings of SenSys'13*. ACM, 2013.

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

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

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