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

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

Figure 1: Pingus gameplay.

ABSTRACT

TODO.

Keywords: TODO, TODO, TODO.

1 INTRODUCTION

Pingus is an open-source puzzle-platform video game based on Lemmings. The objective of the game is to guide a group of penguins through a number of obstacles towards a designated exit (Figure 1). Pingus is developed in standard object-oriented C++, the *lingua franca* of game development [3]. 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 [7] (Figure 2). 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* [3] to handle events from the environment (e.g., key presses and timers) and also as a notification mechanism between entities in the game logic. The observers are short-lived callbacks that have to execute as fast as possible to keep the game reactive to incoming events in real time. For this reason, callbacks cannot contain long-lasting locals and loops, which are elementary capabilities of classical structured programming [2, 4, 1]. In this sense, callbacks actually disrupt structured programming, becoming "our generation's `goto`".[2]

	Game Simulation	Numeric Computation	Shading
Languages	C++, Scripting	C++	CG, HLSL
CPU Budget	10%	90%	n/a
Lines of Code	250,000	250,000	10,000
FPU Usage	0.5 GFLOPS	5 GFLOPS	500 GFLOPS

Figure 2: Three kinds of code [7].

CÉU [6, 5] is a programming language that offers a concurrent and expressive alternative to C/C++ with the characteristics that follow:

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

Structured reactive programming eliminates callbacks, letting programmers write code in direct and sequential style and recover from the inversion of control imposed by the observer pattern [2]. CÉU supports logical parallelism with a resource-efficient implementation in terms of memory and CPU usage [6]. The runtime is single threaded and does not rely on garbage collection.

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

- Identification of five recurrent control-flow patterns that likely apply to other games.
- Idiomatic code in CÉU for six selected behaviors in the game logic.
- An in-depth qualitative analysis of the implementations in CÉU in comparison to C++.

The rest of the paper is organized as follows: Section 2 gives an overview of the Pingus codebases in C++ and CÉU. Section 3 describes the five control-flow patterns and discusses six case studies. Section 4 discusses related work. Section 5 concludes the paper.

2 THE PINGUS CODEBASE

In Pingus, the game logic also accounts for almost half the size of the codebase: 18.173 from 39.362 LoCs (46%) spread across

[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 3: The Pingus codebase directory tree.

272 files. However, about half of the game logic relates to non-reactive code, such as configurations and options, saved games and serialization, maps and levels descriptions, string formatting, collision detection, graph algorithms, etc. This part remains unchanged and relies on the seamless integration between CÉU and C/C++ [6]. Therefore, we rewrote 9.186 LoCs spread across 126 files[3]. In order to only consider effective code in the analysis, we then removed all headers, declarations, trivial getters & setters, and other innocuous statements, resulting 4.135 dense LoCs spread across 70 implementation files originally written in C++[4]. We did the same with the implementation in CÉU, resulting in 3.697 dense LoCs[5]. Figure 3 summarizes the effective codebase in the two implementations.

This work focuses on a qualitative analysis for the programming techniques that we applied during the rewriting process. Not all techniques result in reduction in LoCs (especially considering the verbose syntax of CÉU), but have other effects such as eliminating shared variables and dependencies between classes. Nonetheless, the lines with lower ratio numbers above correlate to the parts of the game logic that we consider more susceptible to structured reactive programming. For instance, the *Pingu* behavior (*ratio 0.80*) contains complex animations that are affected by timers, game rules, and user interaction. In contrast, the *Option screen* (*ratio 0.97*) is a simple UI grid with trivial mouse interactions.

3 Control-Flow Patterns & Case Studies

The rewriting process consisted of identifying sets of callbacks implementing *control-flow behaviors* 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 can identify control-flow behaviors in C++ by looking for class members with identifiers resembling verbs, statuses, and counters (e.g., `pressed`, `particle_thrown`, `mode`, and `delay_count`). Good chances are that variables with these "suspicious names" encode some form of control-flow progression that cross multiple callback invocations.

We selected 6 representative game behaviors and describe their implementations in C++ and CÉU. We also categorize these behaviors in 5 abstract control-flow patterns that likely apply to other games:

1. *Finite State Machines*: Event occurrences map to transitions between states that trigger appropriate actions comprising the behavior of a game entity.

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

3. *Dispatching Hierarchies*: Entities form a dispatching hierarchy in which a container that receives a stimulus automatically forwards it to its managed children.

4. *Lifespan Hierarchies*: Entities form a lifespan hierarchy in which a terminating container entity automatically destroys its managed children.

5. *Signaling Mechanisms*: Entities often need to communicate explicitly through signaling mechanisms, especially if there is no hierarchy relationship between them.

3.1 Finite State Machines

Case Study: Detecting double-clicks in the *Armageddon button*

In Pingus, a double click in the *Armageddon button* at the bottom right of the screen literally explodes all pingus.[6] Figure 4 compares the implementations in C++ and CÉU.

In C++, the class `ArmageddonButton` implements methods for rendering the button and handling mouse and timer events. The listing focus on the double click detection, hiding 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 button base class `RectComponent` (ln. 2), while the callback `update` continuously reacts to the passage of time, frame by frame. Callbacks are short lived because they must react to input as fast as possible to let other callbacks execute, keeping the game with real-time responsiveness. The class first initializes the variable `pressed` to track the first click (ln. 3,32). It also initializes the variable `press_time` to count the time since the first click (ln. 4, 17). If another click occurs within 1 second, the class signals the double click to the application (ln. 30). Otherwise, the `pressed` and `press_time` state variables are reset (ln.

[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`

[6]Double click animation: `github.com/an000/p/blob/master/README.md#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         server->send_armageddon_event();
31     } else {
32         pressed = true;
33     }
34 }
```

[a] Implementation in C++

```
1  do
2      var& RectComponent c = <...>;
3      <...>
4      loop do
5          await c.component.on_click;
6          watching 1s do
7              await c.component.on_click;
8              break;
9          end
10     end
11     <...>
12     emit game.go_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 4: Detecting double-clicks in the *Armageddon button*.

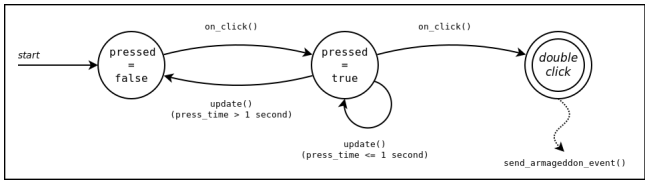

Figure 5: State machine for detecting double-clicks in the *Armageddon button*.

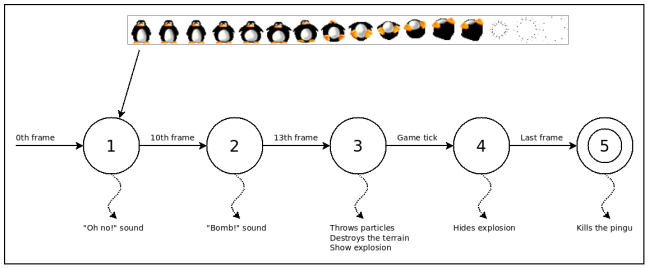

Figure 6: State machine for the *Bomber animation*.

19–20). Figure 5 illustrates how we can model the double-click behavior in C++ as a state machine. The circles represent the state of the variables in the class, while the arrows represent the callbacks manipulating state. Note in the code how the accesses to the state variables are spread across the entire class. For instance, the distance between the initialization of `pressed` (ln. 3) and the last access to it (ln. 32) is over 40 lines in the original file. Arguably, this dispersion of code across methods makes the understanding and maintenance of the double-click behavior more difficult. Also, even though the state variables are private, unrelated methods such as `draw`‘, which is defined in middle of the class (ln. 10–12), can potentially access them.

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

Case Study: The *Bomber action* sequence

The *Bomber action* explodes the clicked pingu, throwing particles around and also destroying the terrain under its radius.[7] We can model the explosion animation with a sequential state machine

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

```
1  Bomber::Bomber (Pingu* p) :
2    <...>
3    sprite(<...>),          // bomber sprite
4    sound_played(false),    // tracks state 2
5    particle_thrown(false), // tracks state 3
6    colmap_exploded(false), // tracks state 3
7    gfx_exploded(false)     // tracks state 4
8  {
9    <...>
10   // 1. plays a "Oh no!" sound.
11   get_world()->play_sound("ohno");
12 }
13
14 void Bomber::update () {
15   sprite.update();
16   <...>   // pingu movement
17
18   // 2. plays a "Bomb!" sound.
19   if (sprite.frame()==10 && !sound_played) {
20     sound_played = true;
21     get_world()->play_sound("plop");
22   }
23
24   // 3. particles, terrain, explosion sprite
25   if (sprite.frame()==13 && !particle_thrown) {
26     particle_thrown = true;
27     get_world()->get_particles()->add(...);
28   }
29   if (sprite.frame()==13 && !colmap_exploded) {
30     colmap_exploded = true;
31     get_world()->remove(bomber_radius, <...>);
32   }
33
34   // 5. kills the Pingu
35   if (sprite.is_finished ()) {
36     pingu->set_status(PS_DEAD);
37   }
38 }
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(explo_surf, <...>);
46   }
47   gc.color().draw(sprite, pingu->get_pos());
48 }
```

[a] Implementation in C++

```
1  code/await Bomber (void) -> ActionName
2  do
3    <...>
4    spawn Mover();  // movement in the background
5    var&? Sprite s = spawn Sprite(<...>);
6                     // animation in the background
7    watching s do
8      // 1. plays a "Oh no!" sound.
9      {Sound::PingusSound::play_sound("ohno")};
10
11     // 2. plays a "Bomb!" sound.
12     await game.dt until s!.sprite.frame == 10;
13     {Sound::PingusSound::play_sound("plop")};
14
15     // 3. particles, terrain, explosion sprite
16     await game.dt until s!.sprite.frame == 13;
17     spawn PinguParticles(<...>) in particles;
18     call Game_Remove({&bomber_radius}, <...>);
19     do
20       <...>
21       spawn Sprite(<...>);      // explosion
22
23       // 4. tick: hides the explosion sprite
24       await game.dt;
25     end
26     await FOREVER;
27   end
28
29   // 5. kills the pingu
30   escape DEAD;
31 end
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48 .
```

[b] Implementation in CÉU

Figure 7: The *Bomber action* sequence.

(Figure 6) with actions associated to specific frames as follows[8]:

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

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

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

In C++, the class `Bomber` defines the callbacks `draw` and `update` to manage the state machine described above: The class first defines one state variable for each action to perform (ln. 4–7). The "Oh no!" sound plays as soon as the object starts in *state-1* (ln. 11). The `update` callback (ln. 14–38) updates the pingu animation and movement on every frame regardless of its current state (ln. 15–16). When the animation reaches the 10th frame, it plays the "Bomb!" sound and switches to *state-2* (ln. 18–22). The state variable `sound_played` is required because the sprite frame doesn't necessarily advance on every `update` invocation (e.g., `update` may execute twice during the 10th frame). The same reasoning and technique applies to *state-3* (ln. 25–32 and 43–44). The explosion sprite appears in a single frame in *state-4* (ln. 45). Finally, the pingu dies after the animation frames terminate (ln. 35–37). Note that a sin-

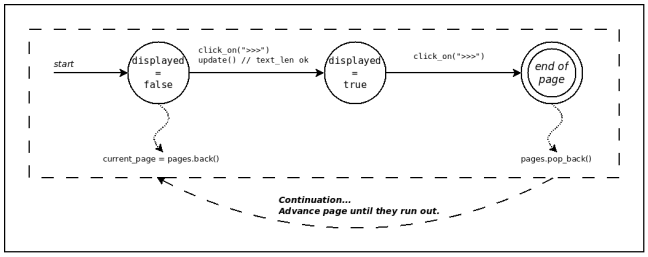

Figure 8: State machine for advancing pages in the *Story screen*.

gle numeric state variable suffices to track the states, but the original authors probably chose to encode each state in an independent boolean variable to rearrange and experiment with them during development. Still, due to the short-lived nature of callbacks, state variables are unavoidable and are actually the essence of object-oriented programming (i.e., *methods + mutable state*). Like the C++ implementation for detecting double clicks, note also that the state machine is encoded across 3 different methods, each intermixing code with unrelated functionality.

The equivalent code for the *Bomber action* in CÉU doesn't require state variables and reflects the sequential state machine implicitly, using `await` statements in direct style to separate the actions. The `Bomber` is a `code/await` abstraction of CÉU, which is similar to a coroutine [1]: 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 abstractions and execute in separate through the `spawn` primitive (ln. 4–5). The event `game.dt` (ln. 12,16,24) is analogous to the `update` callback of C++ and occurs on every frame. The code tracks the animation instance (ln. 7–27), performing the last bomber action on termination (ln. 30). As soon as the animation starts, the code performs the first action (ln. 8). The intermediate actions are performed when the corresponding conditions occur (ln. 12,16,24). The `do-end` block (ln. 19–25), restricts the lifespan of the single-frame explosion sprite (ln. 21): after the next game tick (ln. 24), the block terminates and automatically destroys the spawned abstraction (removing it from the screen). In contrast with the implementation in C++, all actions happen in a contiguous chunk of code (ln. 5–30) which handles no extra functionality.

3.2 Continuation Passing

Case Study: Advancing Pages in the *Story screen*

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

In C++, the class `StoryScreenComponent` implements the method `next_text`, which is a callback for clicks in `>>>`. The variable 'pages' (ln. 4–5, 24–26) is a vector holding each page, but which also encodes *continuations* for the story progress: each call to `next_text` that advances the story (ln. 23–32) removes the current page (ln. 24) and sets the next action to perform (i.e., "display a new page") in the variable `current_page` (ln. 26). Figure 8 illustrates the continuation mechanism to advance pages and also a state machine for fast forwarding words (inside the dashed rectangle). The state variable `displayed` (ln. 6,15,20,21,27) switches between the behaviors "advancing text" and "advancing pages", which are both handled intermixed inside the method `next_text`.

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

3.3 Dispatching Hierarchies

Case Study: *Bomber action* `draw` and `update` dispatching

TODO

Figure 11 compares the implementations in C++ and CÉU.

In C++, the class `Bomber` declares a `sprite` member to handle its animation frames (Figure 6). The `Sprite` class is part of the game engine and knows how to update and render itself. However, the `Bomber` still has to respond to `update` and `draw` requests from the game and forward them to the sprite (ln. 11–13 and 15–18). To understand how the `update` callback flows from the original environment stimulus from the game down to the sprite, we need to follow a long chain of 7 method dispatches (Figure 10):

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

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

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

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

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

<...>   // draw page over time

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

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

[a] Implementation in C++

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

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


.
```

[b] Implementation in CÉU

Figure 9: Advancing pages in the *Story Screen*.

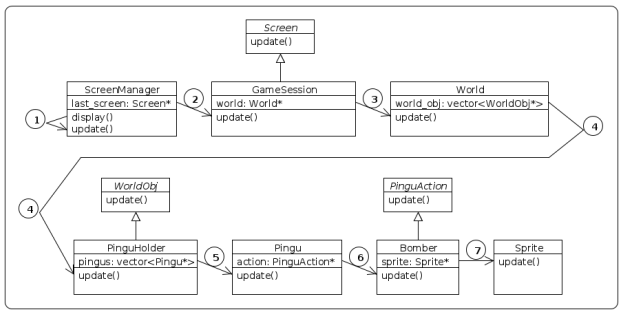

Figure 10: Dispatching chain for `update`.

- Since a single pingu can change between actions during lifetime, the `action` member decouples them with another level of indirection.
- Sprites are part of the game engine and are reusable everywhere (e.g., UI buttons, world objects, etc.), so it is also convenient to decouple them from actions.

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

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

3.4 Lifespan Hierarchies

Case Study: Managing the Pingus Lifecycle

A pingu is a dynamic entity created periodically and destroyed under certain conditions, such as falling from a high altitude[10]. Figure 12 compares the implementations in C++ and CÉU.

In C++, the class `PinguHolder` is a container that holds all pingus alive. The method `PinguHolder::create_pingu` (ln. 1–6) is called periodically to create a new `Pingu` and add it to the `pingus` collection (ln. 3–4). The method `PinguHolder::update` (ln. 8–18) checks the state of all pingus on every frame, removing those with the dead status (ln. 12–14). Entities with dynamic lifespan in C++ require explicit `add` and `remove` calls associated to a container (ln. 4,13). Without the `erase` call above, a dead pingu would remain in the collection with updates on every frame (ln. 11). Since the `redraw` behavior for a dead pingu is innocuous, the death could go unnoticed but the program would keep consuming memory and CPU time. This problem is known as the *lapsed listener* [**?**] 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.

[10]Death of pingus animation: `github.com/an000/p/blob/`

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

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

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

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

[a] Implementation in C++

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

[b] Implementation in CÉU

Figure 11: *Bomber action* `draw` and `update` dispatching.

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

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

[a] Implementation in C++

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

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

[b] Implementation in CÉU

Figure 12: Managing the pingus lifecycle.

CÉU supports `pool` declarations to hold dynamic abstraction instances. Additionally, the `spawn` statement supports a pool identifier to associate the new instance with a pool. The game screen spawns a new `Pingu` on every invocation of `Pingu_Spawn`. The `spawn` statement (ln. 6) specifies the pool declared at the top-level block of the game screen (ln. 3). In this case, the lifespan of the new instances follows the scope of the pool (ln. 1–9) instead of the enclosing scope of the `spawn` statement (ln. 4–7). Since pools are also subject to lexical scope, the lifespan of all dynamically allocated pingus is constrained to the game screen. Lexical scopes handle memory and event dispatching automatically for static instances and also for pools. However, the lifespan of a dynamic instance does not necessarily have to match the lifespan of its associated pool (Figure 13). 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.

3.5 Signaling Mechanisms

Case Study: Global Keys and the Options Menu

The *mouse grab option* restricts the mouse movement to the game window boundaries[11]. The option can be set anywhere in the game by pressing *Ctrl-G*. In addition, the *Options menu* has a check box to toggle the *mouse grab option* with mouse clicks while still responding to *Ctrl-G* presses. Figure 15 compares the implementations in C++ and CÉU.

The implementations in C++ and CÉU use a signalling mechanism to connect the key presses, the check box, and a configuration

master/README.md#5

[11] Mouse grab animation: `github.com/an000/p/blob/master/README.md#6`

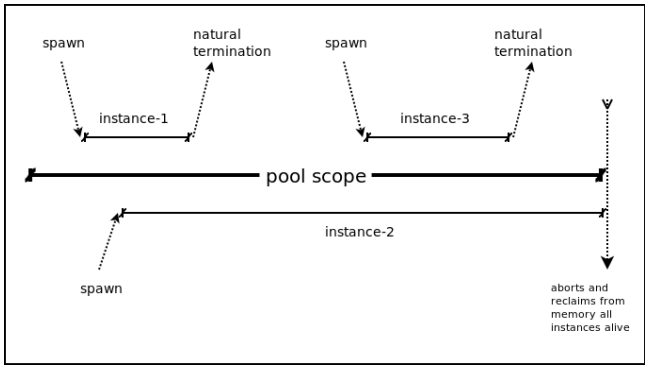

Figure 13: Lifespan of dynamic instances.

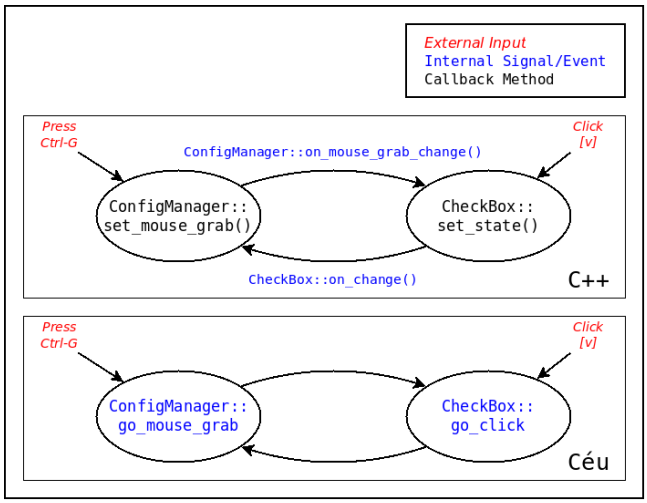

Figure 14: Mutual dependency between signals.

manager that applies the appropriate side effects in the game (i.e., restrict the mouse movement). Figure 14 illustrates how the mutual notifications create a dependency cycle between the configuration manager and the check box.

In C++, the class 'GlbEvt' detects *Ctrl-G* presses and invokes the callback `config_manager.set_grab` (ln. 5–8). The class 'CfgMgr' uses a `boost::signal` (ln. 16) to notify the application when the new configuration is applied (ln. 22). The `if` enclosing the signal emission (ln. 20–23) breaks the dependency cycle of Figure 14 and prevents an infinite execution loop. The class `ChkBox` also uses a `boost::signal` (ln. 28) to notify the application on changes (ln. 33). Again, the `if` enclosing the signal emission (ln. 32–34) breaks the dependency cycle of Figure 14 to avoid infinite execution. The class `OptMenu` creates the dependency loop by connecting the two signals. The constructor binds the signal `config_manager.on_grab_change` to the callback method `grab_box->set` (ln. 48–52), and also the signal `grab_box->on_change` to the callback method `config_manager.set_grab` (ln. 53–57). This way, every time the `CfgMgr` signals `on_grab_change` (ln. 22), the method `set` is implicitly called. The same happens between the signal `on_change` in the `ChkBox` and the method `set_grab` in the `CfgMgr` (ln. 18). Note that the signal binding to call `ChkBox::set` (ln. 50) receives a fixed value `false` as the last argument to prevent infinite execution (ln. 30). The `OptMenu` destructor (ln. 62–66) breaks the connections explicitly when the *Option screen* terminates.

In CÉU, a *Ctrl-G* key press broadcasts the internal event `config_manager.go_grab` to the application (ln. 8). The configuration manager just needs to react to `go_grab` continuously to perform the *grab* effect (ln. 25–27). The `ChkBox` exposes the event `go_click` for notifications in both directions, i.e., from the abstraction to the application and *vice versa*: The abstraction reacts to external clicks continuously (ln. 41–43) to broadcast the event `go_click` (ln. 47). It also reacts continuously to `go_click` in another line of execution (ln. 45–48), which awakes from notifications from the first line of execution or from the application. The `OptMenu` connects the two events as follows: The two loops in parallel handle the connections in opposite directions: from the configuration manager to the check box (ln. 60–62); and from the check box to the configuration manager (ln. 65–67). When the *Option screen* terminates, the connections break automatically since the body with the two loops is automatically aborted. Note that the implementation in CÉU does not check event emits to break the dependency cycle and prevent infinite execution. Due to the stack-based execution for internal events in CÉU [**?**], programs with mutually-dependent events do not create infinite execution loops.

4 Related Work

5 Conclusion

TODO: non reactive, C++ integration

We promote the *structured synchronous reactive* programming model of CÉU for the development of games. We present in-depth use cases categorized in 5 control-flow patterns applied to *Pingus* (an open-source *Lemmings* clone) that likely apply to other games.

We show how the standard way to program games with objects and callbacks in C++ hinders structured programming techniques, such as support for sequential execution, long-lasting loops, and persisting local variables. In this sense, callbacks actually disrupt structured programming, becoming ["our generations goto"][goto] according to Miguel de Icaza.

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

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

6 Acknowledgments

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

References

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

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

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

[4] 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.

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

[6] 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.

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

```
void GlbEvt::on_button_press (<...>) {
  <...>
  switch (event.keysym.sym) {
    case K_g:
      if (keys[K_LCTRL]  keys[K_RCTRL]) {
        cfgmgr.set_grab(
          !cfgmgr.get_grab());
      }
      break;
    <...>
  }
}

///

boost::signal<void(bool)> on_grab_change;

void CfgMgr::set_grab (bool v) {
  <...>
  if (v != get_grab()) {
    <...>   // the actual "grab" effect
    on_grab_change(v);
  }
}

///

boost::signal<void (bool)> on_change;

void ChkBox::set (bool on, bool sndsig) {
  <...>   // switches the check box state
  if (sndsig) {
    on_change(on);
  }
}

///

typedef std::vector<boost::connection> Conns;
Conns conns;

OptMenu::OptMenu() :
  conns(),
  grab_box(),
  <...>
{
  grab_box = new ChkBox(<...>);
  conns.push_back(
    cfgmgr.on_grab_change.connect( std::bind(
      &ChkBox::set, grab_box, <...>, false) );
    )
  );
  conns.push_back(
    grab_box->on_change.connect( std::bind(
      &CfgMgr::set_grab, &cfgmgr, <...>) );
    )
  );
  <...>

}

OptMenu::~OptMenu() {
  for (Conns::iterator i=conns.begin();
              i!=conns.end();
              ++i) {
    (*i).disconnect();
  }
}


.
```

[a] Implementation in C++

```
spawn do
  var _SDL_KeyboardEvent&& e;
  every e in SDL_KEYDOWN do
    var _u8&& keys = _SDL_GetKeyState(null);
    <...>
    if e:keysym.sym == K_g then
      if keys[K_LCTRL] or keys[K_RCTRL] then
        emit cfgmgr.go_grab(
            not {cfgmgr.get_grab()});
      end
    end
    <...>
  end
end

///

data CfgMgr with
  event bool go_grab;
end
var CfgMgr cfgmgr = val CfgMgr(_);

spawn do
  var bool v;
  every v in cfgmgr.go_grab do
    <...>   // the actual "grab" effect
  end
end

///

data IChkBox with
  var   bool is_on;
  event bool go_click;
end

code/await ChkBox (<...>) -> (var IChkBox box) -> FOREVER d
  box = val IChkBox(<...>);
  <...>
  par do
    every c.component.on_click do
      emit box.go_click(not box.is_on);
    end
  with
    loop do
      <...>   // switches the check box state
      box.is_on = await box.go_click;
    end
  end
end

///

code/await OptMenu <...> do
  <...>
  var& ChkBox b2 = <...>;
  spawn do
    par do
      var bool v;
      every v in cfgmgr.go_grab do
        emit b2.box.go_click(v);
      end
    with
      var bool v;
      every v in b2.box.go_click do
        emit cfgmgr.go_grab(v);
      end
    end
  end
  <...>
end
```

[b] Implementation in CÉU

Figure 15: TODO.