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.

	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].

Abstract

Abstract.

Keywords: Radiosity, global illumination, constant time.

1 Introduction

Pingus[1] is an open-source clone of Lemmings[2], a puzzle-platformer video game. The objective of the game is to guide a group of penguins through a number of obstacles towards a designated exit[3].

Pingus is developed in standard object-oriented C++, the *lingua franca* of game development [3]. The codebase is about 40.000 lines of code (LoCs)[4], 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]. The game logic "models the state of the game world as interacting objects evolve over time". 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 use the *observer pattern* [3] in the game logic to handle events from the environment (e.g., key presses and timers) and also as a notification mechanism between game entities. The observers are short-lived callbacks which must execute as fast as possible and in real time to keep the game reactive to incoming events. 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`".[5][6]

CÉU [6, 5] is a programming language that aims to offer 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 the language requires no garbage collection.

Contributions: - patterns - solutions

2 Control-Flow Patterns

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

[1] Pingus: `http://pingus.seul.org/`

[2] Lemmings: `https://en.wikipedia.org/wiki/Lemmings_(video_game)`

[3] Pingus gameplay: `https://www.youtube.com/watch?v=MKrJgIFtJX0`

[4] Pingus repository: `https://github.com/Pingus/pingus/tree/7b255840c201d028fd6b19a2185ccf7df3a2cd6e/src`

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

[6] "Escape from Callback Hell": `http://elm-lang.org/learn/Escape-from-Callback-Hell.elm`

Figure 3: Double click detection.

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 9 representative game behaviors and describe their implementations in C++ and Céu. We also categorized these examples in 5 abstract C++ control-flow patterns that likely apply to other games:

1. *Finite State Machines*: State machines describe the behavior of entities by mapping event occurrences to transitions between states that trigger appropriate actions.

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

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

4. *Lifespan Hierarchies*: Entities typically 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.

2.1 Finite State Machines
Case Study: The Armageddon Button Double Click

State machines describe the behavior of entities by mapping event occurrences to transitions between states that trigger appropriate actions.

In Pingus, a double click in the *Armageddon* button at the bottom right of the screen literally explodes all pingus (Figure 3). Figure **??** 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. 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).

2.2 Continuation Passing
Case Study: Advancing Pages in the Story Screen

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

The clickable *blue dots* in the campaign world map transit to ambience story screens (Figure 8). 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.

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 au-

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

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

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

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

[a] Implementation in C++

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

[b] Implementation in CÉU

Figure 4: Shared-memory concurrency in CÉU: example [a] is safe because the trails access x atomically in different reactions; example [b] is unsafe because both trails access y in the same reaction.

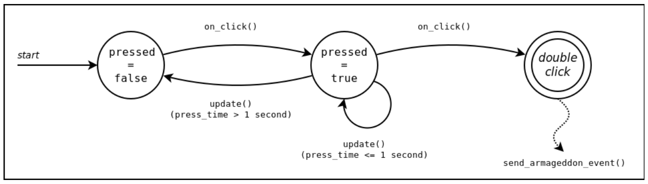

Figure 5: State machine for the *Armageddon* double click.

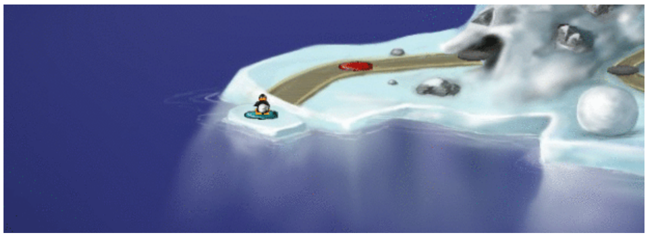

Figure 6: The Story screen.

tomatically 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).

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.

```
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 (&lt;all-words-appearing&gt;) {
        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 7: Shared-memory concurrency in Céu: example [a] is safe because the trails access x atomically in different reactions; example [b] is unsafe because both trails access y in the same reaction.

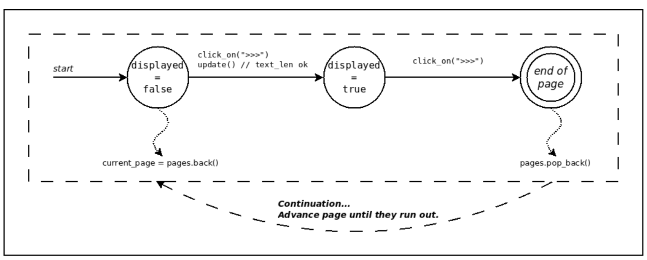

Figure 8: State machine for the Story screen.