ChronoProfiler

Project: AcceleRender
Author: Eric Newton
Version: 1.0
Date: 2025-10-27

Overview

ChronoProfiler is a lightweight, real-time CPU profiling system designed to measure performance of code regions ("zones") across multiple threads. The profiler is optimized for game/engine loops where profiling occurs repeatedly each frame, and profiling data is either visualized in the ProfilerUI or exported for offline analysis (JSON).

ChronoProfiler emphasizes:

- Minimal overhead during pushEventStart and pushEventEnd
- Thread-local event recording to avoid locking
- Simple RAII usage (ScopedZone, ScopedFrame)
- Ability to merge and export all recorded events per frame

The primary usage pattern is:

```
PROFILE_FRAME();
PROFILE_SCOPE("Physics update");
PROFILE_SCOPE("Vulkan submit");
```

At `endFrame()`, all event data is merged into a frame-wide list and handed off to the UI or exporter.

Goals and Non-Goals

Goals

- Measure execution time of arbitrary code blocks.
- Support profiling across multiple threads without locking overhead.
- Provide a clear event history per frame.
- Allow exporting profiling sessions to JSON.

Non-Goals

- GPU performance profiling.
- Deep statistical analysis (averages, min/max, percentiles).
- Integration with external tracing systems (e.g., Tracy, Chrome Trace) — could be added later.

Design Principles

1. **Minimal Runtime Overhead**
   Profiling must not meaningfully affect frame time.
2. **Isolation Between Threads**
   Each thread stores events independently to avoid contention.
3. **RAII-first API**
   Users should not need to call pushEventEnd() manually.
4. **Deterministic merging**
   Frame boundaries define a complete profiling dataset.

Architecture

ChronoProfiler uses static data structures and static API functions. It contains three primary components:

1. Event (struct)

Stores timing data for a single profiling zone:

- zone name

- start time (ms)
- end time (ms)
- duration (calculated as end - start)
- thread ID
- optional color/category for UI grouping

2. ScopedZone (RAII)

Used to profile a block of code. Constructor pushes a start event, destructor pushes an end event.

Example:

```
void update() {
  ScopedZone zone("Update Logic");
}
```

3. ScopedFrame (RAII)

Marks a complete frame. Constructor calls `beginFrame()`, destructor calls `endFrame()`.

Example:

```
void renderLoop() {
  while (running) {
    ScopedFrame frame;
    update();
    draw();
  }
}
```

Multi-Threading Strategy

Thread-local event buffers

Each thread maintains its own `std::vector<Event>` (threadEvents).

Purpose:

- No mutex required during zone pushes.
- Eliminates synchronization cost inside `pushEventStart` / `pushEventEnd`.

During `endFrame()`, **all thread-local event buffers are merged** into `frameEvents` using a mutex (mergeMutex).

Thread Name Mapping

Threads may be assigned human-readable names:

ChronoProfiler::setThreadName("Render Thread");

Internally stored in a `std::unordered_map<threadID, name>` protected by a mutex.

API Summary

Public API for user code:

```
static void beginFrame();
static void endFrame();

static void pushEventStart(std::string_view name,
                           uint32_t color = 0x64C8FFFF,
                           const std::string& category = "");

static void pushEventEnd();

static const std::vector<Event>& getEvents();

static void setThreadName(const std::string& name);
static std::string getThreadName(uint32_t threadId);

static void exportToJSON(const std::string& filename);
```

Private helpers:

- `nowMs()` — retrieves current time in ms using high_resolution_clock.

Internal state:

- `thread_local std::vector<Event> threadEvents`
  Thread-local buffer of events.
- `std::vector<Event> frameEvents`
  Merged result of all thread buffers after `endFrame()`.
- `std::mutex mergeMutex, threadNamesMutex`
  Protect merging + naming.
- `std::atomic<size_t> totalEventCount`
  Safety counter to avoid runaway event logging.
- `std::vector<std::vector<Event>*> allThreadBuffers`
  Registered list of thread-local buffers.

Memory Strategy

Each thread-local event store acts as a ring buffer (optional):

```
max events per thread = kMaxEventsPerThread
```

This prevents memory explosion in cases where profiling is accidentally left on in production.

JSON Export Format

`exportToJSON()` serializes the latest frame of events to a JSON file. Format example:

```
{
  "events": [
    { "name": "update", "thread": 1, "start": 0.23, "end": 1.91,
"duration": 1.68 },
```

```
    { "name": "render", "thread": 1, "start": 2.05, "end": 10.54,
"duration": 8.49 }
  ]
}
```

The ProfilerUI consumes this output to visualize timelines or aggregate data.

Example Usage

```
int main() {
  ChronoProfiler::setThreadName("Main");

  while (true) {
    PROFILE_FRAME();

    PROFILE_SCOPE("Game Update") {
      game.update();
    }

    PROFILE_SCOPE("Render Submission") {
      renderer.drawFrame();
    }
  }
}
```

Future Enhancements

- Live streaming profiling data to an external UI
- Filtering/searching events by name or category
- Visualization of nested zones

Conclusion

ChronoProfiler provides a simple, effective, zero-lock runtime CPU profiler appropriate for real-time applications. Its RAII-based API keeps instrumentation clean and readable, while thread-local event queues guarantee minimal runtime cost.

It forms the backend for ProfilerUI but is decoupled from visualization, allowing reuse in any project.