Tracy Profiler

The user manual

Bartosz Taudul <wolf.pld@gmail.com>

December 30, 2018

`https://bitbucket.org/wolfpld/tracy`

Contents

1 A quick look at Tracy

Tracy is a real-time, nanosecond resolution *frame profiler* that can be used for remote or embedded telemetry of applications. It can profile CPU (C++, Lua), GPU (OpenGL, Vulkan) and memory. It also can monitor locks held by threads and show where contention does happen.

In contrast with *statistical profilers* (such as VTune, perf or Very Sleepy), Tracy does require manual markup of the source code. In return, it allows frame-by-frame inspection of the program execution. You will be able to see exactly which functions are called, how much time is spent in them, and how do they interact with each other in a multi-threaded environment. This feat is by-design impossible to achieve in statistical profilers, which work by periodically sampling the *program counter* register to see which part of the code is executing.

Even though Tracy is a *frame* profiler, with the emphasis on analysis of *frame time* in real-time applications, it does work with utilities that do not employ the concept of a frame. There's nothing that would prohibit profiling of, for example, a compression tool, or an event-driven UI application.

The close analogues of Tracy are: RAD Telemetry, Brofiler, microprofile.

Now let's take a close look at the marketing blurb.

1.1 Real-time

This claim can be described in the following ways:

1. The profiled application is not slowed down by profiling[1]. The act of recording a profiling event has virtually zero cost – it only takes ~8 ns. Even on low-power mobile devices there's no perceptible impact on execution speed.

2. The profiler itself works in real-time, without the need to process collected data in a complex way. Actually, it is quite inefficient in the way it works, as the data it presents is calculated anew each frame. And yet it can run at 60 frames per second.

3. The profiler has full functionality when the profiled application is running and the data is captured. You may interact with your application and then immediately switch to the profiler, when a performance drop occurs.

1.2 Nanosecond resolution

It is hard to imagine how long a nanosecond is. One good analogy is to compare it with a measure of length. Let's say that one second is one meter (the average doorknob is on the height of one meter).

One millisecond ($\frac{1}{1000}$ of a second) would be then the length of a millimeter. The average size of a red ant or the width of a pencil is 5 or 6 mm. A modern game running at 60 frames per second has only 16 ms to update the game world and render the entire scene.

One microsecond ($\frac{1}{1000}$ of a millisecond) in our comparison equals to one micron. The diameter of a typical bacterium ranges from 1 to 10 microns. The diameter of a red blood cell, or width of strand of spider web silk is about 7 μm.

And finally, one nanosecond ($\frac{1}{1000}$ of a microsecond) would be one nanometer. The modern microprocessor transistor gate, the width of DNA helix, or the thickness of a cell membrane are in the range of 5 nm. In one ns the light can travel only 30 cm.

Tracy can achieve single-digit nanosecond measurement resolution, due to usage of hardware timing mechanisms on the x86 and ARM architectures[2]. Other profilers may rely on the timers provided by operating system, which do have significantly reduced resolution (about 300 ns – 1 μs). This is enough to hide the subtle impact of cache access optimization, etc.

[1]See section 1.5 for a benchmark.

[2]In both 32 and 64 bit variants. On x86 Tracy requires a modern version of the `rdtscp` instruction (Sandy Bridge and later). On ARM-based systems Tracy will try to use the timer register (~40 ns resolution). If it fails (due to kernel configuration), Tracy falls back to system provided timer, which can range in resolution from 250 ns to 1 μs.

1.2.1 Timer accuracy

You may wonder why it is important to have a high resolution timer[3]. After all, you only want to profile functions that have long execution times, and not some short-lived procedures, that have no impact on the application's run time.

It is wrong to think so. Optimizing a function to execute in 430 ns, instead of 535 ns (note that there is only a 100 ns difference) results in 14 ms savings if the function is executed 18000 times[4]. It may not seem like a big number, but this is how much time there is to render a complete frame in a 60 FPS game.

You also need to understand how timer precision is reflected in measurement errors. Take a look at figure 1. There you can see three discrete timer tick events, which increase the value reported by the timer by 300 ns. You can also see four readings of time ranges, marked A_1, A_2; B_1, B_2; C_1, C_2 and D_1, D_2.

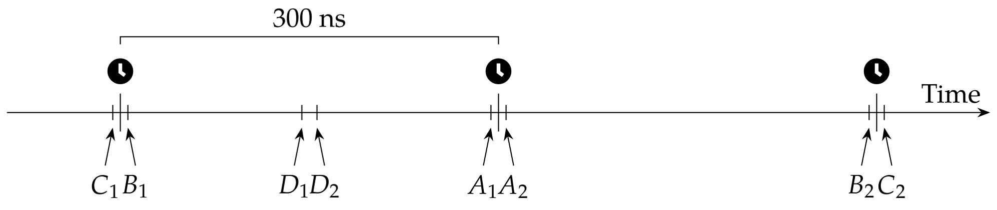

Figure 1: *Low precision (300 ns) timer. Discrete timer ticks are indicated by the* 🕒 *icon.*

Now let's take a look at the timer readings.

- The *A* and *D* ranges both take a very short amount of time (10 ns), but the *A* range is reported as 300 ns, and the *D* range is reported as 0 ns.
- The *B* range takes a considerable amount of time (590 ns), but according to the timer readings, it took the same time (300 ns) as the short lived *A* range.
- The *C* range (610 ns) is only 20 ns longer than the *B* range, but it is reported as 900 ns, a 600 ns difference!

Now you can see why it is important to use a high precision timer. While there is no escape from the measurement errors, their impact can be reduced by increasing the timer accuracy.

1.3 Frame profiler

Tracy is aimed at understanding the inner workings of a tight loop of a game (or an interactive application). That's why it slices the execution time of a program using the *frame*[5] as a basic work-unit[6]. The most interesting frames are the ones that took longer than the allocated time, producing visible hitches in the on-screen animation. Tracy allows inspection of such misbehavior.

1.4 Remote or embedded telemetry

Tracy uses the client-server model to enable a wide range of use-cases (see figure 2). For example, a game on a mobile phone may be profiled over the wireless connection, with the profiler running on a desktop computer. Or you can run the client and server on the same computer, using a localhost connection. It is also possible to embed the visualization front-end in the profiled application, making the profiling self-contained[7].

[3]Interestingly, the `std::chrono::high_resolution_clock` is not really a high resolution clock.

[4]This is a real optimization case. The values are median function run times and do not reflect the real execution time, which explains the discrepancy in the total reported time.

[5]A frame is used to describe a single image displayed on the screen by the game (or any other program), preferably 60 times per second to achieve smooth animation.

[6]Frame usage is not required. See section 3.2 for more information.

[7]See section 2.2.1 for guidelines.

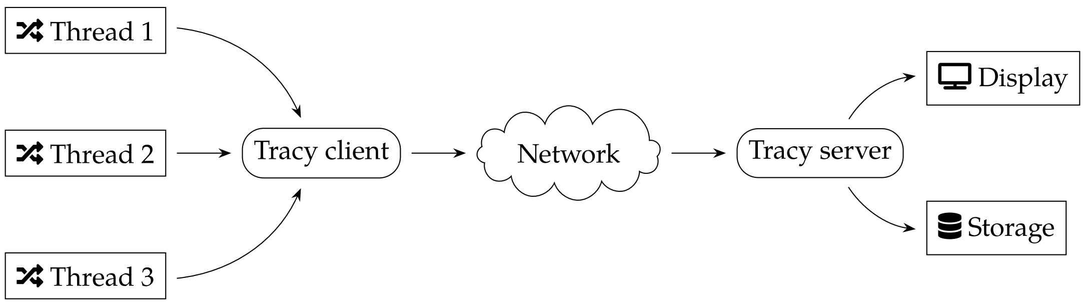

Figure 2: *Client-server model.*

In the Tracy terminology, the profiled application is the *client* and the profiler itself is the *server*. It was named this way because the client is a thin layer that just collects events and sends them for processing and long-term storage on the server. The fact that the server needs to connect to the client to begin the profiling session may be a bit confusing at first.

1.5 Performance impact

To check how much slowdown is introduced by using Tracy, let's profile an example application. For this purpose we will use etcpak[8]. Let's use an 8192×8192 pixels test image as input data and instrument everything down to the 4×4 pixel block compression function (that's 4 million blocks to compress).

The resulting timing information can be seen in table 1. As can be seen, the cost of a single-zone capture (consisting of the zone begin and zone end events) is ~15 ns.

Output	**Zones**	**Clean run**	**Profiling run**	**Difference**
ETC1	4,194,568	0.94 s	1.003 s	+0.063 s
ETC2 + mip-maps	5,592,822	1.034 s	1.119 s	+0.085 s

Table 1: *Zone capture time cost.*

It should be noted that Tracy has a constant initialization cost, needed to perform timer calibration. This cost was subtracted from the profiling run times, as it is irrelevant to the single-zone capture time.

2 First steps

Tracy requires compiler support for C++11, Thread Local Storage and a way to workaround static initialization order fiasco. There are no other requirements. The following platforms are confirmed to be working (this is not a complete list):

- Windows (x86, x64)
- Linux (x86, x64, ARM, ARM64)
- Android (ARM, x86)
- FreeBSD (x64)
- Cygwin (x64)
- WSL (x64)

[8]`https://bitbucket.org/wolfpld/etcpak`

- OSX (x64)[9]

The following compilers are supported:

- MSVC
- gcc
- clang

2.1 Initial client setup

The recommended way to integrate Tracy into an application is to create a git submodule in the repository (assuming that git is used for version control). This way it is very easy to update Tracy to newly released versions.

If that's not an option, copy all files from the `tracy/client` and `tracy/common` directories, along with the source files in Tracy's root directory to your project. Next, add the `tracy/TracyClient.cpp` source file to the IDE project and/or makefile. That's all. Tracy is now integrated into the application.

In the default configuration Tracy is disabled. This way you don't have to worry that the production builds will perform collection of the profiling data. You will probably want to create a separate build configuration, with the `TRACY_ENABLE` define, which enables profiling.

Finally, on Unix make sure that the application is linked with libraries `libpthread` and `libdl`.

2.1.1 Short-lived applications

In case you want to profile a short-lived program (for example, a compression utility that finishes its work in one second), add the `TRACY_NO_EXIT` define to the build configuration. With this option enabled, Tracy will not exit until an incoming connection is made, even if the application has already finished executing. This mode of operation can also be turned on by setting the `TRACY_NO_EXIT` environment variable to 1.

2.1.2 On-demand profiling

By default Tracy will begin profiling even before the program enters the `main` function. If you don't want to perform a full capture of application life-time, you may define the `TRACY_ON_DEMAND` macro, which will enable profiling only when there's an established connection with the server.

It should be noted, that if on-demand profiling is *disabled* (which is the default), then the recorded events will be stored in the system memory until a server connection is made and the data can be uploaded[10]. Depending on the amount of the things profiled, the requirements for event storage can easily grow up to a couple of gigabytes. Since this data is cleared after the initial connection is made, you won't be able to perform a second connection to a client, unless the on-demand mode is used.

Caveats

The client with on-demand profiling enabled needs to perform additional bookkeeping, in order to present a coherent application state to the profiler. This incurs additional time cost for each profiling event.

[9] In the Apple world everything has to be *think different*. Support for Thread Local Storage is only available since Xcode 8 and not before iOS 9. There's no way to handle static initialization order fiasco, so you will have to make your own workarounds. Zone filtering described in section 3.3.3 may be of help.

[10] This memory is never released, but it is reused for collection of further events.

2.1.3 Setup for multi-DLL projects

In projects that consist of multiple DLLs/shared objects things are a bit different. Compiling `TracyClient.cpp` into every DLL is not an option because this would result in several instances of Tracy objects lying around in the process. We rather need to pass the instances of them to the different DLLs to be reused there.

For that you need a *main DLL* to which your executable and the other DLLs link. If that doesn't exist you have to create one explicitly for Tracy. Link the executable and all DLLs which you want to profile to this DLL.

You should compile the main library with the `tracy/TracyClient.cpp` source file and then add the `tracy/TracyClientDLL.cpp` file to the source files lists of the executable and the other DLLs.

2.2 Running the server

The easiest way to get going is to build the data analyzer, available in the `profiler` directory. With it you can connect to localhost or remote clients and view the collected data right away.

If you prefer to inspect the data only after a trace has been performed, you may use the command line utility in the `capture` directory. It will save a data dump that may be later opened in the graphical viewer application.

Note that ideally you should use the same version of the Tracy profiler on both client and server. The network protocol may change, in which case you won't be able to make a connection.

See section 4 for more information about performing captures.

2.2.1 Embedding the server in profiled application

While not officially supported, it is possible to embed the server in your application, the same one which is running the client part of Tracy. This is left up for you to figure out.

Note that most libraries bundled with Tracy are modified in some way and contained in the `tracy` namespace. The one exception is Dear ImGui, which can be freely replaced.

Be aware that while the Tracy client uses its own separate memory allocator, the server part of Tracy will use global memory allocation facilities, shared with the rest of your application. This will affect both the memory usage statistics and Tracy memory profiling.

The following defines may be of interest:

- `TRACY_FILESELECTOR` – controls whether a system load/save dialog is compiled in. If it's left out, the saved traces will be named `trace.tracy`.
- `TRACY_NO_STATISTICS` – Tracy will perform statistical data collection on the fly, if this macro is *not* defined. This allows extended analysis of the trace (for example, you can perform a live search for matching zones) at a small CPU processing cost and a considerable memory usage increase (at least 10 bytes per zone).
- `TRACY_EXTENDED_FONT` – use this define, if you have loaded extra symbol ranges in your font and added icons[11]. Otherwise, some characters will be replaced with an ASCII compatible version. For example, the micro (μ) symbol will be replaced with `u`, and ⚠ icon will be replaced with `/!\`.
- `TRACY_ROOT_WINDOW` – the main profiler view will occupy whole window if this macro is defined. Additional setup is required for this to work. If you are embedding the server into your application you probably do *not* want this.

[11] See the `profiler` utility source code for reference.

2.3 Naming threads

Remember to set thread names for proper identification of threads. You should use the functions exposed in the `tracy/common/TracySystem.hpp` header to do so.

Be aware that even if you already have thread naming functionality implemented, some platforms[12] do not have adequate system-level capabilities (or none at all), in which case Tracy uses its own internal thread name storage.

2.4 Crash handling

On selected platforms[13] Tracy will intercept application crashes[14]. This serves two purposes. First, the client application will be able to send the remaining profiling data to the server. Second, the server will receive a crash report with information about the crash reason, call stack at the time of crash, etc.

This is an automatic process and it doesn't require user interaction.

Caveats

On MSVC the debugger has priority over the application in handling exceptions. If you want to finish the profiler data collection with the debugger hooked-up, select the *continue* option in the debugger pop-up dialog.

3 Client markup

With the aforementioned steps you will be able to connect to the profiled program, but there won't be any data collection performed. In order to begin profiling, Tracy requires that you manually instrument the application[15]. All the user-facing interface is contained in the `tracy/Tracy.hpp` header file.

The best way to start is to add markup to the main loop of the application, along with a few function that are called there. This will give you a rough outline of the function's time cost, which you may then further refine by instrumenting functions deeper in the call stack.

3.1 Handling text strings

When dealing with Tracy macros, you will encounter two ways of providing string data to the profiler. In both cases you should pass `const char*` pointers, but there are differences in expected life-time of the pointed data.

1. When a macro only accepts a pointer (for example: `TracyMessageL(text)`), the provided string data must be accessible at any time in program execution (*this also includes the time after exiting the* `main` *function*). The string also cannot be changed. This basically means that the only option is to use a string literal (e.g.: `TracyMessageL("Hello")`).

2. If there's a string pointer with a size parameter (for example: `TracyMessage(text, size)`), the profiler will allocate an internal temporary buffer to store the data. The pointed-to data is not used afterwards. You should be aware that allocating and copying memory involved in this operation has a small time cost.

[12]Basically everything, but the recent Windows releases.
[13]Windows, Linux and Android.
[14]For example, invalid memory accesses ('segmentation faults', 'null pointer exceptions'), divisions by zero, etc.
[15]Automatic tracing of every entered function is not feasible due to the amount of data that would generate.

3.2 Marking frames

Do I need this?

This step is optional, as some applications do not use the concept of a frame.

To slice the program's execution recording into frame-sized chunks[16], put the `FrameMark` macro after you have completed rendering the frame. Ideally that would be right after the swap buffers command.

3.2.1 Secondary frame sets

In some cases you may want to track more than one set of frames in your program. To do so, you may use the `FrameMarkNamed(name)` macro, which will create a new set of frames for each unique name you provide.

3.2.2 Discontinuous frames

Some types of frames are discontinuous by nature. For example, a physics processing step in a game loop, or an audio callback running on a separate thread. These kinds of workloads are executed periodically, with a pause between each run. Tracy can also track these kind of frames.

To mark the beginning of a discontinuous frame use the `FrameMarkStart(name)` macro. After the work is finished, use the `FrameMarkEnd(name)` macro.

Important

- Frame types *must not* be mixed. For each frame set, identified by an unique name, use either continuous or discontinuous frames only!
- You *must* issue the `FrameMarkStart` and `FrameMarkEnd` macros in proper order. Be extra careful, especially if multi-threading is involved. Note that the profiler event data is unordered between threads, so you can't start a frame in one thread and end it in another one.

3.3 Marking zones

To record a zone's[17] execution time add the `ZoneScoped` macro at the beginning of the scope you want to measure. This will automatically record function name, source file name and location. Optionally you may use the `ZoneScopedC(0xRRGGBB)` macro to set a custom color for the zone. Note that the color value will be constant in the recording (don't try to parametrize it). You may also set a custom name for the zone, using the `ZoneScopedN(name)` macro. Color and name may be combined by using the `ZoneScopedNC(name, color)` macro.

Use the `ZoneText(text, size)` macro to add a custom text string that will be displayed along the zone information (for example, name of the file you are opening).

If you want to set zone name on a per-call basis, you may do so using the `ZoneName(text, size)` macro. This name won't be used in the process of grouping the zones for statistical purposes (sections 5.4 and 5.5).

Color palette

You may use named colors predefined in `common/TracyColor.hpp` (included by `Tracy.hpp`). Visual reference: `https://en.wikipedia.org/wiki/X11_color_names`.

[16]Each frame starts immediately after the previous has ended.

[17]A zone represents the life-time of a special on-stack profiler variable. Typically it would exist for the duration of a whole scope of the profiled function, but you also can measure time spent in scopes of a for-loop, or an if-branch.

3.3.1 Multiple zones in one scope

Using the ZoneScoped family of macros creates a stack variable named ___tracy_scoped_zone. If you want to measure more than one zone in the same scope, you will need to use the ZoneNamed macros, which require that you provide a name for the created variable. For example, instead of ZoneScopedN("Zone name"), you would use ZoneNamedN(variableName, "Zone name", true)[18].

The ZoneText and ZoneName macros work only for the zones created using the ZoneScoped macros. For the ZoneNamed macros, you will need to invoke the methods Text or Name of the variable you have created.

3.3.2 Variable shadowing

The following code is fully compliant with the C++ standard:

```
void Function()
{
    ZoneScoped;
    ...
    for(int i=0; i<10; i++)
    {
        ZoneScoped;
        ...
    }
}
```

This doesn't stop some compilers from dispensing *fashion advice* about variable shadowing (as both ZoneScoped calls create a variable with the same name, with the inner scope one shadowing the one in the outer scope). If you want to avoid these warnings, you will also need to use the ZoneNamed macros.

3.3.3 Filtering zones

Zone logging can be disabled on a per zone basis, by making use of the ZoneNamed macros. Each of the macros takes an active argument ('true' in the example above), which will determine whether the zone should be logged.

Note that this parameter may be a run-time variable, for example an user controlled switch to enable profiling of a specific part of code only when required. It is also useful to replace handling of the static order initialization fiasco on OSX.

If the condition is constant at compile-time, the resulting code will not contain a branch (the profiling code will either be always enabled, or won't be there at all). The following listing presents how profiling of specific application subsystems might be implemented:

```
enum SubSystems
{
    Sys_Physics     = 1 << 0,
    Sys_Rendering   = 1 << 1,
    Sys_NasalDemons = 1 << 2
}

...

// Preferably a define in the build system
#define SUBSYSTEMS Sys_Physics | Sys_NasalDemons

...

void Physics::Process()
{
    ZoneScopedN( __tracy, SUBSYSTEMS & Sys_Physics );       // always true, no runtime cost
```

[18]The last parameter is explained in section 3.3.3.

```
        ...
    }

    void Graphics::Render()
    {
        ZoneScopedN( __tracy, SUBSYSTEMS & Sys_Graphics );      // always false, no runtime cost
        ...
    }
```

3.4 Marking locks

Modern programs must use multi-threading to achieve full performance capability of the CPU. Correct execution requires claiming exclusive access to data shared between threads. When many threads want to enter the critical section at once, the application's multi-threaded performance advantage is nullified. To answer this problem, Tracy can collect and display lock interactions in threads.

To mark a lock (mutex) for event reporting, use the `TracyLockable(type, varname)` macro. Note that the lock must implement the Mutex requirement[19] (i.e. there's no support for timed mutices). For a concrete example, you would replace the line

```
std::mutex m_lock;
```

with

```
TracyLockable(std::mutex, m_lock);
```

Alternatively, you may use `TracyLockableN(type, varname, description)` to provide a custom lock name.

The standard `std::lock_guard` and `std::unique_lock` wrappers should use the `LockableBase(type)` macro for their template parameter (unless you're using C++17, with improved template argument deduction). For example:

```
std::lock_guard<LockableBase(std::mutex)> lock(m_lock);
```

To mark the location of a lock being held, use the `LockMark(varname)` macro, after you have obtained the lock. Note that the `varname` must be a lock variable (a reference is also valid). This step is optional.

Similarly, you can use `TracySharedLockable`, `TracySharedLockableN` and `SharedLockableBase` to mark locks implementing the SharedMutex requirement[20]. Note that while there's no support for timed mutices in Tracy, both `std::shared_mutex` and `std::shared_timed_mutex` may be used[21].

Caveats

Due to limits of internal bookkeeping in the profiler, each lock may be used in no more than 64 unique threads. If you have many short lived temporary threads, consider using a thread pool to limit the numbers of created threads.

3.5 Plotting data

Tracy is able to capture and draw numeric value changes over time. You may use it to analyze draw call counts, number of performed queries, etc. To report data, use the `TracyPlot(name, value)` macro.

[19]https://en.cppreference.com/w/cpp/named_req/Mutex
[20]https://en.cppreference.com/w/cpp/named_req/SharedMutex
[21]Since `std::shared_mutex` was added in C++17, using `std::shared_timed_mutex` is the only way to have shared mutex functionality in C++14.

3.6 Message log

Fast navigation in large data sets and correlating zones with what was happening in application may be difficult. To ease these issues Tracy provides a message log functionality. You can send messages (for example, your typical debug output) using the `TracyMessage(text, size)` macro. Alternatively, use `TracyMessageL(text)` for string literal messages.

3.7 Memory profiling

Tracy can monitor memory usage of your application. Knowledge about each performed memory allocation enables the following:

- Memory usage graph (like in massif, but fully interactive).
- List of active allocations at program exit (memory leaks).
- Visualization of memory map.
- Ability to rewind view of active allocations and memory map to any point of program execution.
- Information about memory statistics of each zone.
- Memory allocation hot-spot tree.

To mark memory events, use the `TracyAlloc(ptr, size)` and `TracyFree(ptr)` macros. Typically you would do that in overloads of `operator new` and `operator delete`, for example:

```
void* operator new(std::size_t count)
{
    auto ptr = malloc(count);
    TracyAlloc(ptr, count);
    return ptr;
}

void operator delete(void* ptr) noexcept
{
    TracyFree(ptr);
    free(ptr);
}
```

Important

Each tracked memory free event must also have a corresponding memory allocation event. Tracy will not be able to report correct data if this assumption is broken. If you encounter this issue, you may want to check for:

- Mismatched `malloc/new` or `free/delete`.
- Double freeing the memory.
- Untracked allocations made in external libraries, that are freed in the application.
- Places where the memory is allocated, but profiling markup is added.

This requirement is relaxed in the on-demand mode (section 2.1.2), because the memory allocation event might have happened before the connection was made.

3.8 Lua support

To profile Lua code using Tracy, include the `tracy/TracyLua.hpp` header file in your Lua wrapper and execute `tracy::LuaRegister(lua_State*)` function to add instrumentation support.

In the Lua code, add `tracy.ZoneBegin()` and `tracy.ZoneEnd()` calls to mark execution zones. You need to call the `ZoneEnd` method, because there is no automatic destruction of variables in Lua and we don't know when the garbage collection will be performed. *Double check if you have included all return paths!*

Use `tracy.ZoneBeginN(name)` if you want to set a custom zone name[22].

Use `tracy.ZoneText(text)` to set zone text.

Use `tracy.Message(text)` to send messages.

Use `tracy.ZoneName(text)` to set zone name on a per-call basis.

Lua instrumentation needs to perform additional work (including memory allocation) to store source location. This approximately doubles the data collection cost.

Even if Tracy is disabled, you still have to pay the no-op function call cost. To prevent that you may want to use the `tracy::LuaRemove(char* script)` function, which will replace instrumentation calls with white-space. This function does nothing if profiler is enabled.

3.9 GPU profiling

Tracy provides bindings for profiling OpenGL and Vulkan execution time on GPU.

Note that the CPU and GPU timers may be not synchronized. You can correct the resulting desynchronization in the profiler's options (section 5.2).

3.9.1 OpenGL

You will need to include the `tracy/TracyOpenGL.hpp` header file and declare each of your rendering contexts using the `TracyGpuContext` macro (typically you will only have one context). Tracy expects no more than one context per thread and no context migration.

To mark a GPU zone use the `TracyGpuZone(name)` macro, where `name` is a string literal name of the zone. Alternatively you may use `TracyGpuZoneC(name, color)` to specify zone color.

You also need to periodically collect the GPU events using the `TracyGpuCollect` macro. A good place to do it is after the swap buffers function call.

Caveats

- GPU profiling is not supported on OSX, iOS[a].
- Android devices do work, if GPU drivers are not broken. Disjoint events are not currently handled, so some readings may be a bit spotty.
- Nvidia drivers are unable to provide consistent timing results when two OpenGL contexts are used simultaneously.
- Calling the `TracyGpuCollect` macro is a fairly slow operation (couple µs).

[a]Because Apple is unable to implement standards properly.

[22]While technically this name doesn't need to be constant, like in the `ZoneScopedN` macro, it should be, as it is used to group the zones together. This grouping is then used to display various statistics in the profiler. You may still set the per-call name using the `tracy.ZoneName` method.

3.9.2 Vulkan

Similarly, for Vulkan support you should include the `tracy/TracyVulkan.hpp` header file and initialize the Vulkan instance using the `TracyVkContext(physdev, device, queue, cmdbuf)` macro. Cleanup is performed using the `TracyVkDestroy` macro. Currently you can't track more than one instance.

The physical device, logical device, queue and command buffer must relate with each other. The queue must support graphics or compute operations. The command buffer must be in the initial state and be able to be reset. It will be rerecorded and submitted to the queue multiple times and it will be in the executable state on exit from the initialization function.

To mark a GPU zone use the `TracyVkZone(cmdbuf, name)` macro, where `name` is a string literal name of the zone. Alternatively you may use `TracyVkZoneC(cmdbuf, name, color)` to specify zone color. The provided command buffer must be in the recording state.

You also need to periodically collect the GPU events using the `TracyVkCollect(cmdbuf)` macro[23]. The provided command buffer must be in the recording state and outside of a render pass instance.

Caveats

Vulkan support is very bare at the moment. Multi-threaded submitting commands to command buffers is not supported right now.

3.9.3 Multiple zones in one scope

Putting more than one GPU zone macro in a single scope features the same issue as with the `ZoneScoped` macros, described in section 3.3.1 (but this time the variable name is `___tracy_gpu_zone`).

To solve this problem, in case of OpenGL use the `TracyGpuNamedZone` macro in place of `TracyGpuZone` (or the color variant). The same applies to Vulkan – replace `TracyVkZone` with `TracyVkNamedZone`.

Remember that you need to provide your own name for the created stack variable as the first parameter to the macros.

3.10 Collecting call stacks

Tracy can capture true calls stacks on selected platforms (Windows, Linux, Android). It can be performed by using macros with the `S` postfix, which require an additional parameter, specifying the depth of call stack to be captured. The greater the depth, the longer it will take to perform capture. Currently you can use the following macros: `ZoneScopedS`, `ZoneScopedNS`, `ZoneScopedCS`, `ZoneScopedNCS`, `TracyAllocS`, `TracyFreeS`, `TracyGpuZoneS`, `TracyGpuZoneCS`, `TracyVkZoneS`, `TracyVkZoneCS`, and the named variants.

Be aware that call stack collection is a relatively slow operation. Table 2 shows how long it took to perform a single capture of varying depth on multiple CPU architectures.

Depth	**x86**	**x64**
1	37 ns	97 ns
5	51 ns	312 ns
10	71 ns	468 ns
20	84 ns	517 ns

Table 2: *Call stack capture times.*

You can force call stack capture in the non-`S` postfixed macros by adding the `TRACY_CALLSTACK` define, set to the desired call stack capture depth. This setting doesn't affect the explicit call stack macros.

[23]It is considerably faster than the OpenGL's `TracyGpuCollect`.

Debugging symbols

To have proper call stack information, the profiled application must be compiled with debugging symbols enabled. You can achieve that in the following way:

- On MSVC open the project properties and go to *Linker→Debugging→Generate Debug Info*, where the *Generate Debug Information* option should be selected.
- On gcc or clang remember to specify the debugging information `-g` parameter during compilation and omit the strip symbols `-s` parameter. Link the executable with an additional option `-rdynamic` (or `--export-dynamic`, if you are passing parameters directly to the linker).

4 Capturing the data

After the client application has been instrumented, you will want to connect to it using a server.

4.1 Command line

You can capture a trace using a command line utility contained in the `capture` directory. To use it you will need to provide two parameters:

- `-a address` – specifies the IP address (or a domain name) of the client application.
- `-o output.tracy` – the file name of the resulting trace.

If there is no client running at the given address, the server will wait until a connection can be made. During the capture the following information will be displayed:

```
% ./capture -a 127.0.0.1 -o trace
Connecting to 127.0.0.1...
Queue delay: 9 ns
Timer resolution: 6 ns
   1.90 Mbps | Ratio:  40.8% | Real:    4.67 Mbps | Mem: 77.57 MB
```

The *queue delay* and *timer resolution* parameters are calibration results of timers used by the client. The next line is a status bar, which presents: network connection speed, connection compression ratio, the resulting uncompressed data rate and total memory usage of the utility.

4.2 Interactive profiling

If you want to look at the profile data in real-time (or load a saved trace file), you can use the data analysis utility contained in the `profiler` directory. After starting the application, you will be greeted with a welcome dialog (figure 3), presenting a bunch of useful links (📘 *User manual*, 🌐 *Homepage* and 🎥 *Tutorial*).

The client *address entry* field and the 📶 *Connect* button are used to connect to a running client. You can use the connection history button ▾ to display a list of commonly used addresses, from which you can quickly select an address. You can remove entries from this list by hovering the ↖ mouse cursor over an entry and pressing the *delete* button on the keyboard.

If you want to open a trace that you have stored on the disk, you can do so by pressing the 📂 *Open saved trace* button.

Both connecting to a client and opening a saved trace will present you with the main profiler view, which you can use to analyze the data (see section 5).

If this is a real-time capture, you will also see the connection window (figure 4), with the capture status similar to the one displayed by the command line utility. This dialog also displays the connection speed

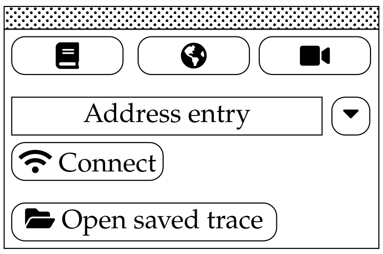

Figure 3: *Welcome dialog.*

graphed over time and the profiled application's current frames per second and frame time measurements. The circle displayed next to the bandwidth graph signals the connection status. If it's red, the connection is active. If it's gray, the client has disconnected.

You can use the *Save trace* button to save the current profile data to a file. The *Discard* button is used to discard current trace.

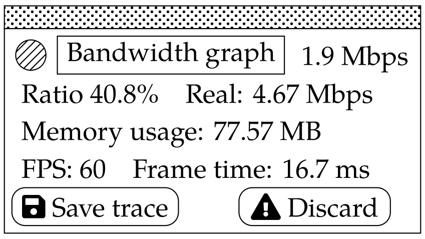

Figure 4: *Connection information window.*

4.2.1 Automatic loading or connecting

You can pass trace file name as an argument to the profiler application to open the capture, skipping the welcome dialog. You can also use the `-a address` argument to automatically connect to the given address.

4.3 Connection speed

Tracy will happily saturate a 1 Gbps network connection, as it can process up to 6 Gbps of uncompressed data. Note that at such data rates, the resulting capture will need to allocate about 1 GB of RAM per second.

4.4 Memory usage

The captured data is stored in RAM and only written to the disk, when the capture finishes. This can result in memory exhaustion when you are capturing massive amounts of profile data, or even in normal usage situations, when the capture is performed over a long stretch of time. The recommended usage pattern is to perform moderate instrumentation of the client code and limit capture time to the strict necessity.

In some cases it may be useful to perform an *on-demand* capture, as described in section 2.1.2. In such case you will be able to profile only the interesting case (e.g. behavior during loading of a level in a game), ignoring all the unneeded data.

If you truly need to capture large traces, you have two options. Either buy more RAM, or use a large swap file on a fast disk drive[24].

[24]The operating system is able to manage memory paging much better than Tracy would be ever able to.

4.5 Trace versioning

Each new release of Tracy changes the internal format of trace files. While there is a backwards compatibility layer, allowing loading of traces created by previous versions of Tracy in new releases, it won't be there forever. You are thus advised to upgrade your traces using the utility contained in the `update` directory.

To use it, you will need to provide the input file and the output file. The program will print a short summary when it finishes:

```
% ./update old.tracy new.tracy
old.tracy (0.3.0) -> new.tracy (0.4.0)
```

The new file contains the same data as the old one, but in the updated internal representation. Note that to perform an upgrade, whole trace needs to be loaded to memory.

4.5.1 Archival mode

The update utility supports optional higher level of data compression, enabled by passing the `--hc` parameter. It can reduce the trace size by 15 to 20%, at a considerable time cost (~ 17× increase of compression time).

Note that trace files (even the ones created in high compression mode) are optimized for fast decompression. You still will be able to squeeze the data using normal compression methods. For example, 7-zip can compress traces to about 25% of their uncompressed[25] size.

5 Analyzing captured data

You have instrumented your application and you have captured a profiling trace. Now you want to look at the collected data. You can do this in the application contained in the `profiler` directory.

The workflow is identical, whether you are viewing a previously saved trace, or if you're performing a live capture, as described in section 4.2.

Note that loading a saved trace will display a progress window.

5.1 Main profiler window

The main profiler window is split into three sections, as seen on figure 5: the control menu, the frame time graph and the timeline display.

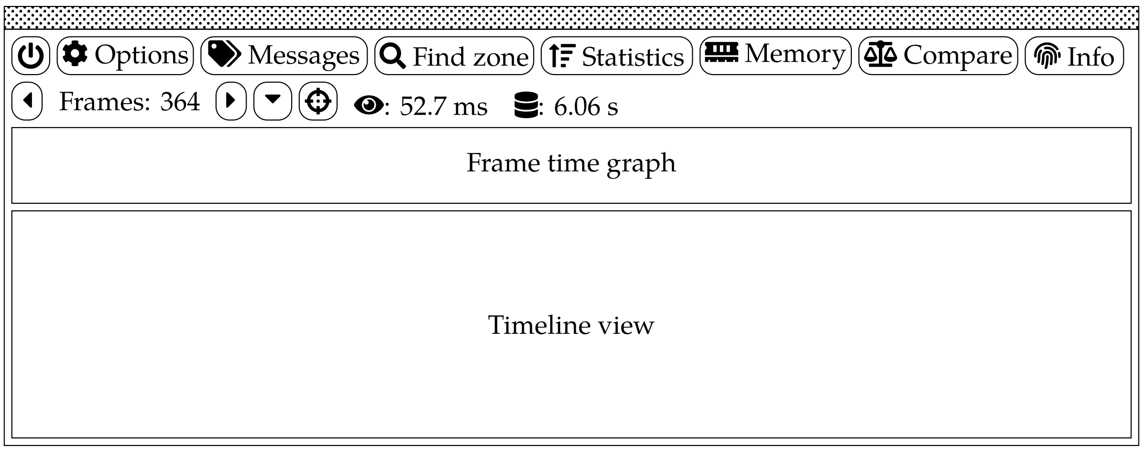

Figure 5: *Main profiler window. Note that the top line of buttons has been split into two rows in this manual.*

[25] Compressed internally.

5.1.1 Control menu

The control menu (top row of buttons) provides access to various features of the profiler. The buttons perform the following actions:

- ⏻ *Close* – This button unloads the current profiling trace and returns to the welcome menu, where another trace can be loaded. In live captures it is replaced by ⏸ *Pause*, ▶ *Resume* and ■ *Stopped* buttons.
- ⏸ *Pause* – While a live capture is in progress, the profiler will display the last three fully captured frames, so that you can see the current behavior of the program. Use this button[26] to stop the automatic updates of the timeline view (the capture will be still progressing).
- ▶ *Resume* – Use this button to resume following the most recent three frames in a live capture.
- ■ *Stopped* – Inactive button used to indicate that the client application was terminated.
- ⚙ *Options* – Toggles the settings menu (section 5.2).
- 🏷 *Messages* – Toggles the message log window (section 5.3), which displays custom messages sent by the client, as described in section 3.6.
- 🔍 *Find zone* – This buttons toggles the find zone window, which allows inspection of zone behavior statistics (section 5.5).
- ⇅ *Statistics* – Toggles the statistics window, which displays zones sorted by their total time cost (section 5.4).
- ▦ *Memory* – Various memory profiling options may be accessed here (section 5.7).
- ⚖ *Compare* – Toggles the trace compare window, which allows you to see the performance difference between two profiling runs (section 5.6).
- ◉ *Info* – Show general information about the trace (section 5.10).

The frame information block consists of four elements: the current frame set name along with the number of captured frames, the two navigational buttons ◂ and ▸, which allow you to focus the timeline view on the previous or next frame, and the frame set selection button ▾, which is used to switch to a another frame set[27]. The ⊕ *Go to frame* button allows zooming the timeline view on the specified frame. For more information about marking frames, see section 3.2.

The last two items show the 👁 view time range and ⛁ time span of the whole capture.

5.1.2 Frame time graph

The graph of currently selected frame set (figure 6) provides an outlook on the time spent in each frame, allowing you to see where the problematic frames are and to quickly navigate to them.

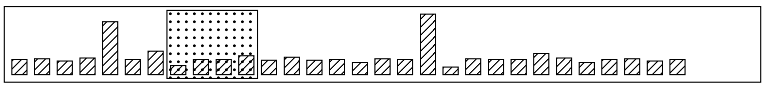

Figure 6: *Frame time graph.*

Each bar displayed on the graph represents an unique frame in the current frame set[28]. The progress of time is in the right direction. The height of the bar indicates the time spent in frame, complemented with the color information:

[26]Or perform any action on the timeline view.
[27]See section 5.1.3.2 for another way to change the active frame set.
[28]Unless the view is zoomed out and multiple frames are merged into one column.

- If the bar is *blue*, then the frame met the *best* time of 143 FPS, or 6.99 ms[29].
- If the bar is *green*, then the frame met the *good* time of 59 FPS, or 16.94 ms.
- If the bar is *yellow*, then the frame met the *bad* time of 29 FPS, or 34.48 ms.
- If the bar is *red*, then the frame didn't met any time limits.

The frames visible on the timeline are marked with a violet box drawn over them (presented as a dotted box on figure 6).

Moving the mouse cursor over the frames displayed on the graph will display tooltip with information about frame number, frame time, etc. Such tooltips are common for many UI elements in the profiler and won't be mentioned later in the manual.

The timeline view may be focused on the frames, by clicking or dragging the left mouse button on the graph. The graph may be scrolled left and right by dragging the right mouse button over the graph. The view may be zoomed in and out by using the mouse scroll. If the view is zoomed out, so that multiple frames are merged into one column, the highest frame time will be used to represent the given column.

5.1.3 Timeline view

The timeline is the most important element of the profiler UI. All the captured data is displayed there, laid out on the horizontal axis, according to the flow of time. The view is split into three parts: the time scale, the frame sets and the combined zones, locks and plots display.

Collapsed items Due to extreme differences in time scales, you will almost constantly see events that are too small to be displayed on the screen. Such events have preset minimum size (so they can be seen) and are marked with a zig-zag pattern, to indicate that you need to zoom-in to see more detail.

The zig-zag pattern can be seen applied to frame sets on figure 8, and to zones on figure 9.

5.1.3.1 Time scale

The time scale is a quick aid in determining the relation between screen space and the time it represents (figure 7).

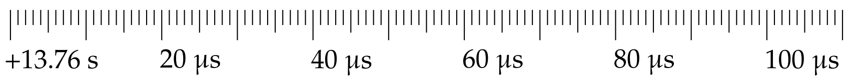

Figure 7: *Time scale.*

The leftmost value on the scale represents the time at which the timeline starts. The rest of numbers label the notches on the scale, with some numbers omitted, if there's no space to display them.

5.1.3.2 Frame sets

Frames from each frame set are displayed directly underneath the time scale. Each frame set occupies a separate row. The currently selected frame set is highlighted with bright colors, with the rest dimmed out.

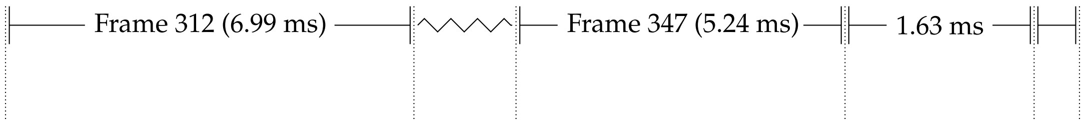

Figure 8: *Frames on the timeline.*

[29]The actual target is 144 FPS, but one frame leeway is allowed to account for timing inaccuracies.

On figure 8 we can see the fully described frames 312 and 347. The description consists of the frame name, which is *Frame* for the default frame set (section 3.2) or the name you used for the secondary name set (section 3.2.1), the frame number and the frame time. The frame 348 is too small to be fully displayed, so only the frame time is shown. The frame 349 is even smaller, with no space for any text. Moreover, frames 313 to 346 are too small to be displayed individually, so they are replaced with a zig-zag pattern, as described in section 5.1.3.

You can also see that there are frame separators, projected down to the rest of the timeline view. Note that only the separators for the currently selected frame set are displayed. You can make a frame set active by clicking the left mouse button on a frame set row you want to select (also see section 5.1.1).

Clicking the middle mouse button on a frame will zoom the view to the extent of the frame.

5.1.3.3 Zones, locks and plots display

On this combined view you will find the zones with locks and their associated threads. The plots are graphed right below.

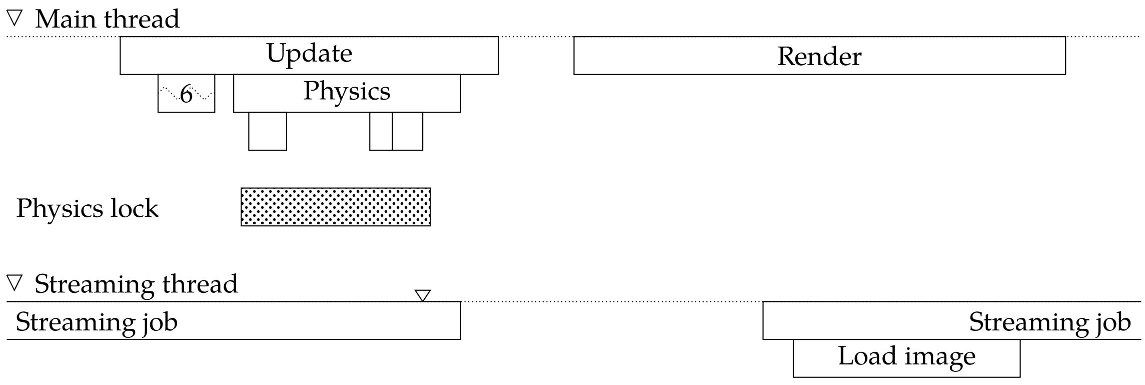

Figure 9: *Zones and locks display.*

The left hand side *index area* of the timeline view displays various labels (threads, locks), which can be categorized in the following way:

- *Light blue label* – OpenGL/Vulkan context.
- *White label* – A CPU thread. Will be replaced by a bright red label in a thread that has crashed (section 2.4).
- *Light red label* – Indicates a lock.
- *Yellow label* – Plot.

Labels accompanied by the ▾ symbol can be collapsed out of the view, to reduce visual clutter.

Zones In an example on figure 9 you can see that there are two threads: *Main thread* and *Streaming thread*[30]. We can see that the *Main thread* has two root level zones visible: *Update* and *Render*. The *Update* zone is split into further sub-zones, some of which are too small to be displayed at the current zoom level. This is indicated by drawing a zig-zag pattern over the merged zones box (section 5.1.3), with the number of collapsed zones printed in place of zone name. We can also see that the *Physics* zone acquires the *Physics lock* mutex for the most of its run time.

Meanwhile the *Streaming thread* is performing some *Streaming jobs*. The first *Streaming job* sent a message (section 3.6), which in addition to being listed in the message log is being indicated by the triangle over the

[30]By clicking on a thread name you can temporarily disable display of the zones in this thread.

thread separator. When there are multiple messages in one place, the triangle outline changes to a filled triangle.

At high zoom levels, the zones will be displayed with additional markers, as presented on figure 10. The red regions at the start and end of a zone indicate the cost associated with recording an event (*Queue delay*). The error bars show the timer inaccuracy (*Timer resolution*). Note that these markers are only *approximations*, as there are many factors that can impact the true cost of capturing a zone, for example cache effects, or CPU frequency scaling, which is unaccounted for.

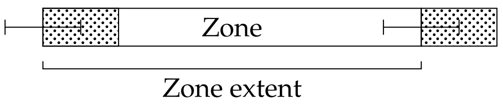

Figure 10: *Approximation of timer inaccuracies and zone collection cost.*

The GPU zones are displayed just like CPU zones, with an OpenGL/Vulkan context in place of a thread name.

Hovering the mouse pointer over a zone will highlight all other zones that have the same source location with a white outline. Clicking the left mouse button on a zone will open zone information window (section 5.11). Clicking the middle mouse button on a zone will zoom the view to the extent of the zone.

Locks Mutual exclusion zones are displayed in each thread that tries to acquire them. There are three color-coded kinds of lock event regions that may be displayed. Note that when the timeline view is zoomed out, the contention regions are always displayed over the uncontented ones.

- *Green region*[31] – The lock is being held solely by one thread and no other thread tries to access it. In case of shared locks it is possible that multiple threads hold the read lock, but no thread requires a write lock.
- *Yellow region* – The lock is being owned by this thread and some other thread also wants to acquire the lock.
- *Red region* – The thread wants to acquire the lock, but is blocked by other thread, or threads in case of shared lock.

Hovering the mouse pointer over a lock event will display important information, for example a list of threads that are currently blocking, or which are blocked by the lock. Clicking the left mouse button on a lock event or a lock label will open the lock information window, as described in section 5.14. Clicking the middle mouse button on a lock event will zoom the view to the extent of the event.

Plots The numerical data values (figure 11) are plotted right below the zones and locks. Note that the minimum and maximum values currently displayed on the plot are visible on the screen, along with the y range of the plot. The discrete data points are indicated with little rectangles. Multiple data points are indicated by a filled rectangle.

When memory profiling (section 3.7) is enabled, Tracy will automatically generate a *Memory usage* plot, which has extended capabilities. Hovering over a data point (memory allocation event) will visually display duration of the allocation. Clicking the left mouse button on the data point will open the memory allocation information window, which will display the duration of the allocation as long as the window is open.

[31]This region type is disabled by default and needs to be enabled in options (section 5.2).

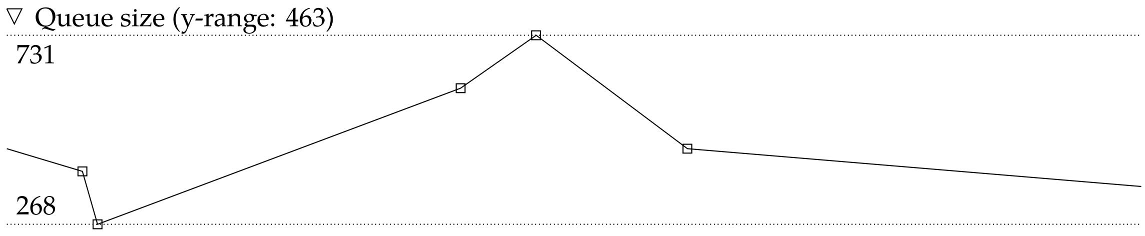

Figure 11: *Plot display.*

5.1.4 Navigating the view

Hovering the mouse pointer over the timeline view will display a vertical line that can be used to visually line-up events in multiple threads. Dragging the left mouse button will display time measurement of the selected region.

The timeline view may be scrolled both vertically and horizontally by dragging the right mouse button. Note that only the zones, locks and plots scroll vertically, while the time scale and frame sets always stay in place.

You can zoom in and out the timeline view by using the mouse scroll. You can select a range to which you want to zoom-in by dragging the middle mouse button. Dragging the middle mouse button while the *control* key is pressed will zoom-out.

5.2 Options menu

In this window you can set various trace-related options. The timeline view might sometimes become overcrowded, in which case disabling display of some profiling events can increase readability.

- *Draw GPU zones* – Allows disabling display of OpenGL/Vulkan zones. The *GPU zones* drop-down allows disabling individual GPU contexts and setting CPU/GPU drift offsets (see section 3.9 for more information).
- *Draw CPU zones* – Determines whether CPU zones are displayed. The *Namespaces* drop-down controls the display behavior of long zone names:
  - *Full* – Zone names are always fully displayed (e.g. `std::sort`).
  - *Shortened* – If there's no space for full zone name, the namespaces will be shortened to one letter (e.g. `s::sort`).
  - *None* – If there's no space for full zone name, the namespaces will be omitted (e.g. `sort`).
- *Draw locks* – Controls the display of locks. If the *Only contended* option is selected, the non-blocking regions of locks won't be displayed (see section 5.1.3.3). The *Locks* drop-down allows disabling display of locks on a per-lock basis. Clicking the right mouse button on a lock label opens the lock information window (section 5.14).
- *Draw plots* – Allows disabling display of plots. Individual plots can be disabled in the *Plots* drop-down.
- *Visible threads* – Here you can disable display of selected threads.
- *Visible frame sets* – Frame set display can be enabled or disabled here. Note that disabled frame sets are still available for selection in the frame set selection drop-down (section 5.1.1), but are marked with a dimmed font.

Disabling display of some events is especially recommended when the profiler performance drops below acceptable levels for interactive usage.

5.3 Messages window

In this window you can see all the messages that were sent by the client application, as described in section 3.6. The window is split into three columns: *time*, *thread* and *message*. Hovering the mouse cursor over a message will highlight it on the timeline view. Clicking the left mouse button on a message will center the timeline view on the selected message.

Message list can be filtered by the originating thread in the *Visible threads* drop-down.

5.4 Statistics window

Looking at the timeline view gives you a very localized outlook on things. Sometimes you want to look at the general overview of the program's behavior, for example you want to know which function takes the most of application's execution time. The statistics window provides you exactly that information.

Here you will find a multi-column display of captured zones, which contains: the zone *name* and *location*, *total time* spent in the zone, the *count* of zone executions and the *mean time spent in the zone per call*. The view may be sorted according to the three displayed values.

By default the displayed times are inclusive, that is, they contain execution times of zone's children. If you want to view just the time spent in zone, you can enable the exclusive mode by selecting the *Show self times* option.

Clicking the left mouse button on a zone will open the individual zone statistics view in the find zone window (section 5.5).

5.5 Find zone window

The individual behavior of zones may be influenced by many factors, like CPU cache effects, access times amortized by the disk cache, thread context switching, etc. Sometimes the execution time depends on the internal data structures and their response to different inputs. In other words, it is hard to determine the true performance characteristics by looking at any single zone.

Tracy gives you the ability to display an execution time histogram of all occurrences of a zone. On this view you can see how the function behaves in general, ignoring the outliers. You can inspect how various data inputs influence the execution time and you can filter the data to eventually drill down to the individual zone calls, so that you can see the environment in which they were called.

You start by entering a search query, which will be matched against known zone names (see section 3.3 for information on the grouping of zone names). If the search found some results, you will be presented with a list of zones in the *matched source locations* drop-down. The selected zone's graph is displayed on the *histogram* drop-down and also the matching zones are highlighted on the timeline view. Clicking the right mouse button on the source file location will open the source file view window (if applicable, see section 5.13).

An example histogram is presented on figure 12. Here you can see that the majority of zone calls (by count) are clustered in the 300 ns group, closely followed by the 10 µs cluster. There are some outliers at the 1 and 10 ms marks, which can be ignored on most occasions, as these are single occurrences.

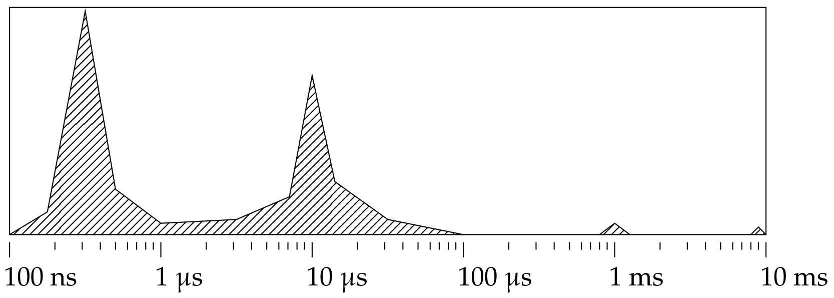

Figure 12: *Zone execution time histogram.*

The histogram is accompanied by various data statistics about displayed data, for example the *total time* of the displayed samples, or the *maximum number of counts* in histogram bins. There are three options that control how the data is presented:

- *Log values* – Switches between linear and logarithmic scale on the y axis of the graph, representing the call counts[32].
- *Log time* – Switches between linear and logarithmic scale on the x axis of the graph, representing the time bins.
- *Cumulate time* – Changes how the histogram bin values are calculated. By default the vertical bars on the graph represent the *call counts* of zones that fit in the given time bin. If this option is enabled, the bars represent the *time spent* in the zones. For example, on graph presented on figure 12 the 10 µs cluster is the dominating one, if we look at the time spent in zone, even if the 300 ns cluster has greater number of call counts.

You can drag the left mouse button over the histogram to select a time range that you want to closely look at. This will display the data in the histogram info section and it will also filter zones displayed in the *found zones* section. This is quite useful, if you want to actually look at the outliers, i.e. where did they originate from, what the program was doing at the moment, etc[33]. You can reset the selection range by pressing the right mouse button on the histogram.

The *found zones* section displays the individual zones grouped according to the following criteria:

- *Thread* – In this mode you can see which threads were executing the zone.
- *User text* – Splits the zones according to the custom user text (see section 3.3).
- *Call stacks* – Zones are grouped by the originating call stack (see section 3.10).

Each group may be sorted according to the *order* in which it appeared, the call *count*, or the total *time* spent in the group. Expanding the group view will display individual occurrences of the zone, sorted by application's time. Clicking the left mouse button on a zone will open the zone information window (section 5.11). Clicking the middle mouse button on a zone will zoom the timeline view to the zone's extent.

Clicking the left mouse button on group name will highlight the group time data on the histogram (figure 13). This function provides a quick insight about the impact of the originating thread, or input data on the zone performance. Clicking the right mouse button on the group names area will reset the group selection.

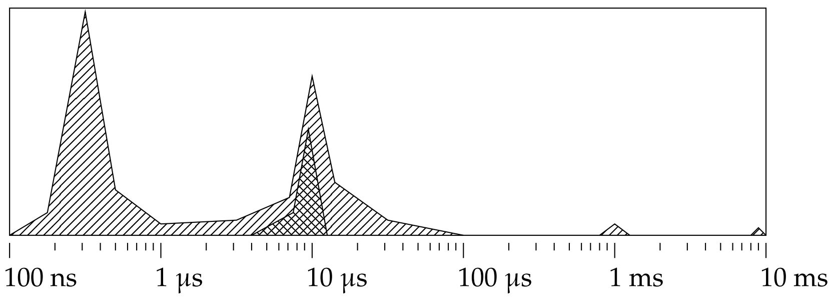

Figure 13: *Zone execution time histogram with a group highlighted.*

[32]Or time, if the *cumulate time* option is enabled.

[33]More often than not you will find out, that the application was just starting, or an access to a cold file was required and there's not much you can do to optimize that particular case.

The average and median zone times are displayed on the histogram as a red (average) and blue (median) vertical bars. When a group is selected, additional bars will indicate the average group time (orange) and median group time (green). You can disable drawing of either set of markers by clicking on the check-box next to the color legend.

Keyboard shortcut

You may press Ctrl + F to open or focus the find zone window and set the keyboard input on the search box.

5.6 Compare traces window

Comparing the performance impact of the optimization work is not an easy thing to do. Benchmarking is often inconclusive, if even possible, in case of interactive applications, where the benchmarked function might not have a visible impact on frame render time. Doing isolated micro-benchmarks loses the execution environment of the application, in which many different functions compete for limited system resources.

Tracy solves this problem by providing a compare traces functionality, very similar to the find zone window, described in section 5.5.

You would begin your work by recording a reference trace that represents the usual behavior of the program. Then, after the optimization of the code is completed, you record another trace, doing roughly what you did for the reference one. Having the optimized trace open you select the *Open second trace* option in the compare traces window and load the reference trace.

Now things start to get familiar. You search for a zone, similarly like in the find zone window, choose the one you want in the *matched source locations* drop-down, and then you look at the histogram. This time there are two overlaid graphs, one representing the current trace, and the second one representing the external (reference) trace (figure 14). You can easily see how the performance characteristics of the zone were affected by your modifications.

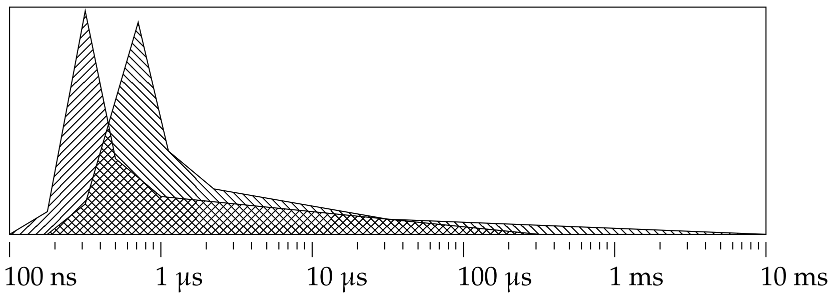

Figure 14: *Compare traces histogram.*

Note that the traces are color and symbol coded. The current trace is marked by a yellow symbol, and the external one is marked by a red symbol.

It may be difficult, if not impossible, to perform identical runs of a program. This means that the number of collected zones may differ in both traces, which would influence the displayed results. To fix this problem enable the *Normalize values* option, which will adjust the displayed results as-if both traces had the same number of recorded zones.

5.7 Memory window

The data gathered by profiling memory usage (section 3.7) can be viewed in the memory window. The top row contains statistics, such as *total allocations* count, number of *active allocations*, current *memory usage* and process *memory span*[34].

The lists of captured memory allocations are displayed in a common multi-column format thorough the profiler. The first column specifies the memory address of an allocation, or an address and an offset, if the address is not at the start of the allocation. Clicking the left mouse button on an address will open the memory allocation information window[35] (see section 5.9). Clicking the middle mouse button on an address will zoom the timeline view to memory allocation's range. The next column contains the allocation size.

The allocation's timing data is contained in two columns: *appeared at* and *duration*. Clicking the left mouse button on the first one will center the timeline view at the beginning of allocation, and likewise, clicking on the second one will center the timeline view at the end of allocation. Note that allocations that have not yet been freed will have their duration displayed in green color.

The memory event location in the code is displayed in the last four columns. The *thread* column contains the thread where the allocation was made and freed (if applicable), or an *alloc / free* pair of threads, if it was allocated in one thread and freed in another. The *zone alloc* contains the zone in which the allocation was performed[36], or - if there was no active zone in the given thread at the time of allocation. Clicking the left mouse button on the zone name will open the zone information window (section 5.11). Similarly, the *zone free* column displays the zone which freed the allocation, which may be colored yellow, if it is the same exact zone that did the allocation. Alternatively, if the zone has not yet been freed, a green *active* text is displayed. The last column contains the *alloc* and *free* call stack buttons, or their placeholders, if no call stack is available (see section 3.10 for more information). Clicking on either of the buttons will open the call stack window (section 5.12). Note that the call stack buttons that match the information window will be highlighted.

The memory window is split into the following sections:

5.7.1 Allocations

The *Allocations* pane allows you to search for the specified address usage during the whole life-time of the program. All recorded memory allocations that match the query will be displayed on a list.

5.7.2 Active allocations

The *Active allocations* pane displays a list of currently active memory allocations and their total memory usage. Here you can see where exactly your program did allocate memory it is currently using. If the application has already exited, this becomes a list of leaked memory.

5.7.3 Memory map

On the *Memory map* pane you can see the graphical representation of your program's address space. Active allocations are displayed as green lines, while the freed memory is marked as red lines. The brightness of the color indicates how much time has passed since the last memory event at the given location – the most recent events are the most vibrant.

This view may be helpful in assessing the general memory behavior of the application, or in debugging the problems resulting from address space fragmentation.

5.7.4 Call stack tree

The *Call stack tree* pane is only available, if the memory events were collecting the call stack data (section 3.10). In this view you are presented with a tree of memory allocations, starting at the call stack entry point and going up to the allocation's pinpointed place. Each level of the tree is sorted according to the number of bytes allocated in given branch.

[34]Memory span describes the address space consumed by the program. It is calculated as a difference between the maximum and minimum observed in-use memory address.

[35]While the allocation information window is opened, the address will be highlighted on the list.

[36]The actual allocation is typically a couple functions deeper in the call stack.

Each tree node consists of three elements: the function name, the source file location and the memory allocation data. The memory allocation data is either yellow *inclusive* events count (including all the children), or the cyan *exclusive* events count. There are two values that are counted: total memory size and number of allocations.

Clicking the right mouse button on the function name will open allocations list window (see section 5.8), which list all the allocations included at the current call stack tree level. Clicking the right mouse button on the source file location will open the source file view window (if applicable, see section 5.13).

Some function names may be too long to be properly displayed, with the events count data at the end. In such cases, you may press the *control* button, which will display events count tooltip.

5.7.5 Looking back at the memory history

By default the memory window displays the memory data at the current point of program execution. It is however possible to view the historical data by enabling the *Restrict time* option. This will draw a vertical violet line on the timeline view, which will act as a terminator for memory events. The memory window will use only the events lying on the left side of the terminator line (in the past), ignoring everything that's on the right side.

5.8 Allocations list window

This window displays the list of allocations included at the selected call stack tree level (see section 5.7 and 5.7.4).

5.9 Memory allocation information window

The information about the selected memory allocation is displayed in this window. It lists the allocation's address and size, along with the time, thread and zone data of the allocation and free events. Clicking the *Zoom to allocation* button will zoom the timeline view to the allocation's extent.

5.10 Trace information window

This window contains various bits of information about profiler and the current trace. For example, you can see the profiler memory usage, the number of captured zones, lock event, plot points, memory allocations, etc. There's also a section containing the selected frame set timing statistics and histogram[37].

In this window you can view the information about the machine on which the profiled application was running. This includes the operating system, the used compiler, CPU name, amount of total available RAM, etc.

Here you will also be able to see the tombstone generated during an application's crash (section 2.4). It provides you with information about the thread that has crashed, the crash reason and the crash call stack (section 5.12).

5.11 Zone information window

The zone information window displays detailed information about a single zone. There can be only one zone information window open at any time. While the window is open the zone will be highlighted on the timeline view with a green outline. The following data is presented:

- Basic source location information: function name, source file location and the thread name.
- Timing information.
- Memory events list, both summarized and a list of individual allocation/free events (see section 5.7 for more information on the memory events list).

[37]See section 5.5 for a description of the histogram. Note that there are subtle differences in the available functionality.

- Zone trace, taking into account the zone tree and call stack information (section 3.10), trying to reconstruct a combined zone + call stack trace[38]. Captured zones are displayed as normal text, while functions that were not instrumented are dimmed. Hovering the mouse pointer over a zone will highlight it on the timeline view with a red outline. Clicking the left mouse button on a zone will switch the zone info window to that zone. Clicking the middle mouse button on a zone will zoom the timeline view to the zone's extent. Clicking the right mouse button on a source file location will open the source file view window (if applicable, see section 5.13).
- Child zones list, showing how the current zone's execution time was used. All the controls from the zone trace are also available here.

The zone information window has the following controls available:

- *Zoom to zone* – Zooms the timeline view to the zone's extent.
- *Go to parent* – Switches the zone information window to display current zone's parent zone (if available).
- *Statistics* – Displays the zone general performance characteristics in the find zone window (section 5.5).
- *Call stack* – Views the current zone's call stack in the call stack window (section 5.12). The button will be highlighted, if the call stack window shows the zone's call stack. Only available if zone had captured call stack data (section 3.10).
- *Source* – Display source file view window with the zone source code (only available if applicable, see section 5.13). Button will be highlighted, if the source file is being currently displayed (but the focused source line might be different).
- *Go back* – Returns to the previously viewed zone. The viewing history is lost when the zone information window is closed, or when the type of displayed zone changes (from CPU to GPU or vice versa).

5.12 Call stack window

This window shows the frames contained in the selected call stack. Each frame is described by the function name and source file location. Clicking the left mouse button on either the function name of source file location will copy the name to the clipboard. Clicking the right mouse button on the source file location will open the source file view window (if applicable, see section 5.13).

Sometimes it may be more useful to have just the function address, instead of the source file location[39]. This can be achieved by selecting the *Show frame addresses* option.

5.13 Source file view window

In this window you can view the source code of the profiled application, to take a quick glance at the context of the function behavior you are analyzing.

Important

Source file view works on the local files you have on your disk. The traces themselves do not contain any source code! This has the following implications:

- The source files can only be viewed, if the source file location recorded in the trace matches the

[38]Reconstruction is only possible, if all zones have full call stack capture data available. In case where that's not available, an *unknown frames* entry will be present.

[39]It can pinpoint the exact assembly instruction which caused the crash.

files you have on your disk.

- **The displayed source files might not reflect the code that was profiled!** It is up to you to verify that you don't have a modified version of the code, with regards to the trace.

5.14 Lock information window

This window presents information and statistics about a lock. The lock events count represents the total number collected of wait, obtain and release events. The announce, termination and lock lifetime measure the time from the lockable construction until destruction.

Appendices

A License

B List of contributors

```
Bartosz Taudul <wolf.pld@gmail.com>
Kamil Klimek <kamil.klimek@sharkbits.com>
Bartosz Szreder <zgredder@gmail.com>
Arvid Gerstmann <dev@arvid-g.de>
Rokas Kupstys <rokups@zoho.com>
Till Rathmann <till.rathmann@gmx.de>
Sherief Farouk <sherief.personal@gmail.com>
```

C Inventory of external libraries

The following libraries are included with and used by the Tracy Profiler:

- 3-clause BSD license
  - getopt_port – `https://github.com/kimgr/getopt_port`
- 2-clause BSD license

- concurrentqueue – `https://github.com/cameron314/concurrentqueue`
- LZ4 – `https://github.com/lz4/lz4`

- Public domain
  - rpmalloc – `https://github.com/rampantpixels/rpmalloc`
  - gl3w – `https://github.com/skaslev/gl3w`
- zlib license
  - benaphore – `https://github.com/preshing/cpp11-on-multicore`
  - Native File Dialog – `https://github.com/mlabbe/nativefiledialog`
  - GLFW – `https://github.com/glfw/glfw`
  - IconFontCppHeaders – `https://github.com/juliettef/IconFontCppHeaders`
  - pdqsort – `https://github.com/orlp/pdqsort`
- MIT license
  - Dear ImGui – `https://github.com/ocornut/imgui`
  - ImGuiColorTextEdit – `https://github.com/BalazsJako/ImGuiColorTextEdit`
- Apache license 2.0
  - Arimo font – `https://fonts.google.com/specimen/Arimo`
  - Cousine font – `https://fonts.google.com/specimen/Cousine`
- Font Awesome Free License
  - Font Awesome – `https://fontawesome.com/`
- Boost Software License 1.0
  - flat_hash_map – `https://github.com/skarupke/flat_hash_map`
- FreeType License
  - FreeType – `https://www.freetype.org/`