Ray Tracing for Games

Dr. Jacco Bikker - IGAD/BUAS, Breda, January 31

Welcome!

3

Agenda:

- State of Affairs
- Optimization Introduction
- Profiling
- Low-level: Rules of Engagement
- The Cache

Ray Tracing for Games

Wednesday
13:00 – 17:00

course intro
LH2
template
Whitted
refactoring
RT-centric games

LAB 1

Thursday
09:00 – 14:00

advanced Whitted
audio, AI & physics
faster Whitted
Heaven7

LAB 2

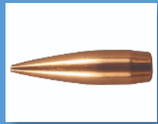
work @ home

End result day 2:

A solid Whitted-style ray tracer, as a basis for subsequent work.

Friday
09:00 – 17:00

optimization
profiling, rules of engagement
threading



LAB 3

SIMD
applied SIMD
SIMD triangle
SIMD AABB

LAB 4

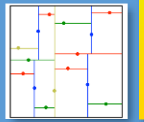
End result day 3:

A 5x faster tracer.

GLOBAL GAME JAM

Monday
09:00 – 17:00

acceleration
grid, BVH, kD-tree
SAH
binning



LAB 5

refitting
top-level BVH
threaded building

LAB 6

End result day 4:

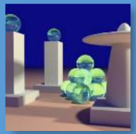
A real-time tracer.

Tuesday
09:00 – 17:00

Monte-Carlo
Cook-style
glossy, AA
area lights, DOF

LAB 7

path tracing



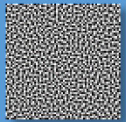
LAB 8

End result day 5:

Cook or Kajiya.

Thursday
09:00 – 17:00

random numbers
stratification
blue noise



LAB 9

importance sampling
next event estimation

LAB 10

End result day 6:

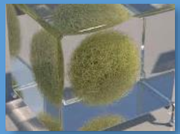
Efficiency.

Friday
09:00 – 17:00

future work

LAB 11

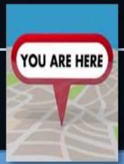
path guiding



LAB 10

End result day 6:

Great product.



Towards the End product:

1. Path tracer that produces pretty images in a few minutes.

   You have a clean ray tracer with a solid API, you implemented most or all of the Whitted-style features.

2. Real-time ray tracer on the CPU or GPU.

   You have finished the functionality that you want to run in real-time, graphics-wise. Animation is still to be done.

3. Raytraced game or demo.

   You have a minimalistic game engine that allows you to build the game, while the engine is being completed.

4. RTX port of an existing game.

   You have basic output from the original game engine using the custom ray tracing renderer.

Wednesday
13:00 – 17:00

course intro
LH2
template
Whitted
refactoring
RT-centric games

LAB 1

Thursday
09:00 – 14:00

advanced Whitted
audio, AI & physics
faster Whitted
Heaven7

LAB 2

work @ home

End result day 2:

A solid Whitted-style ray tracer, as a basis for subsequent work.

Friday
09:00 – 17:00

optimization
profiling, rules of engagement
threading

LAB 3

SIMD
applied SIMD
SIMD triangle
SIMD AABB

LAB 4

End result day 3:

A 5x faster tracer.

GLOBAL GAME JAM

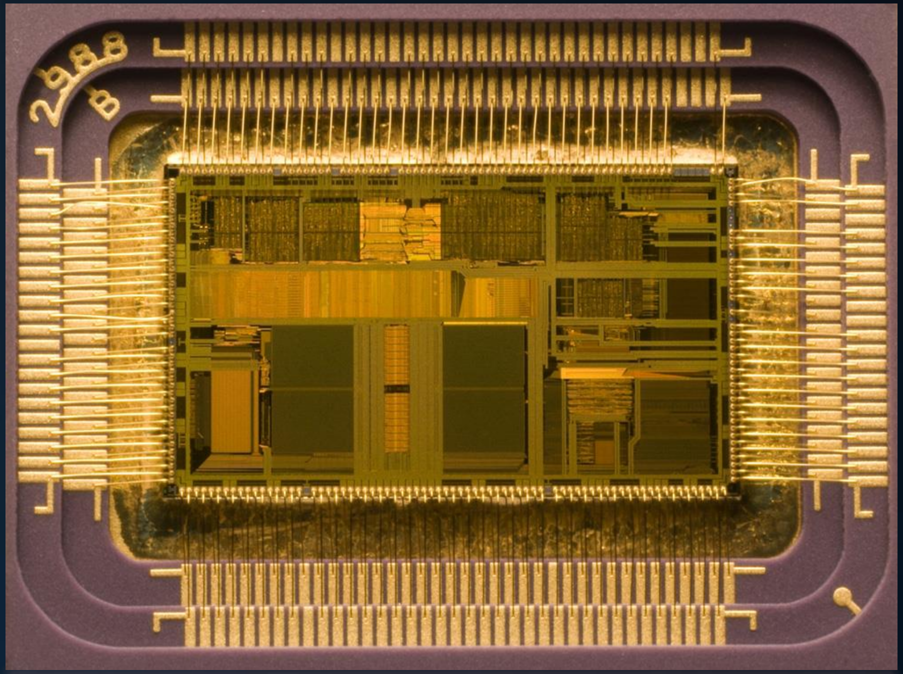
Agenda:

- State of Affairs
- Optimization Introduction
- Profiling
- Low-level: Rules of Engagement
- The Cache

What is optimization?

Think like a CPU

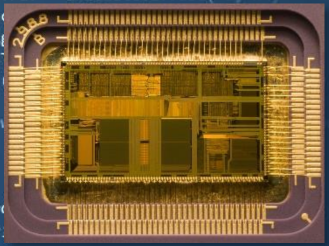
- Instruction pipelines
- Latencies
- Dependencies
- Bandwidth
- Cycles
- Floating point versus integer
- SIMD



What is optimization?

Work smarter, not harder: algorithm scalability

- Big O
- Research: not reinventing the wheel
- Data characteristics & algorithm choice
- STL, Boost: Trust No One
- As accurate as necessary (but not more)
- Balancing accuracy, speed and memory

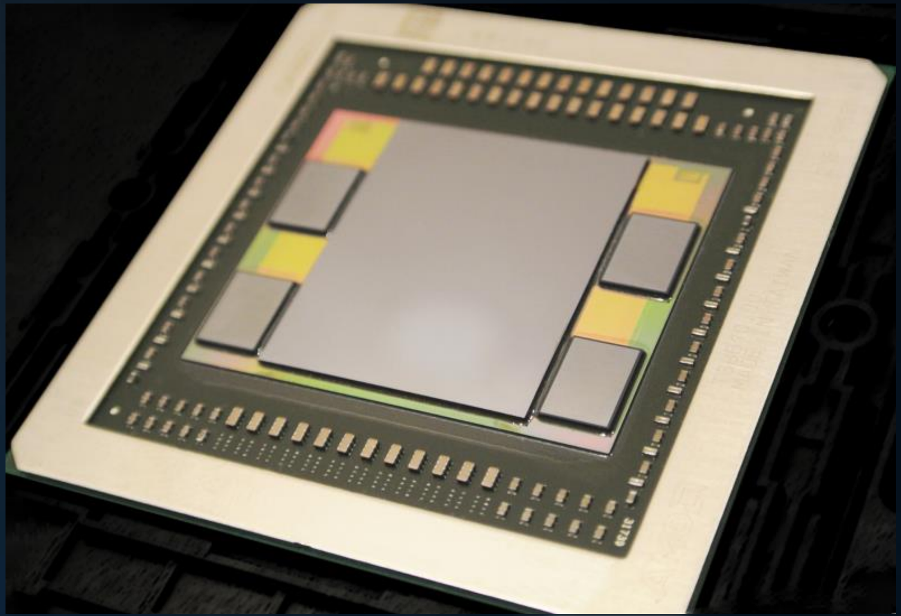


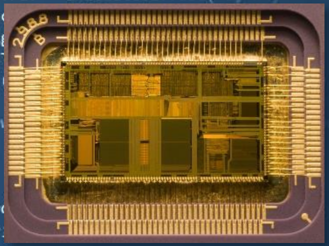
- Instruction pipelines
- Latencies
- Dependencies
- Bandwidth
- Cycles
- Floating point versus integer
- SIMD

What is optimization?

Memory hierarchy: caches

- Cache architecture
- Cache lines
- Hits, misses and collisions
- Eviction policies
- Prefetching
- Cache-oblivious
- Data-centric programming





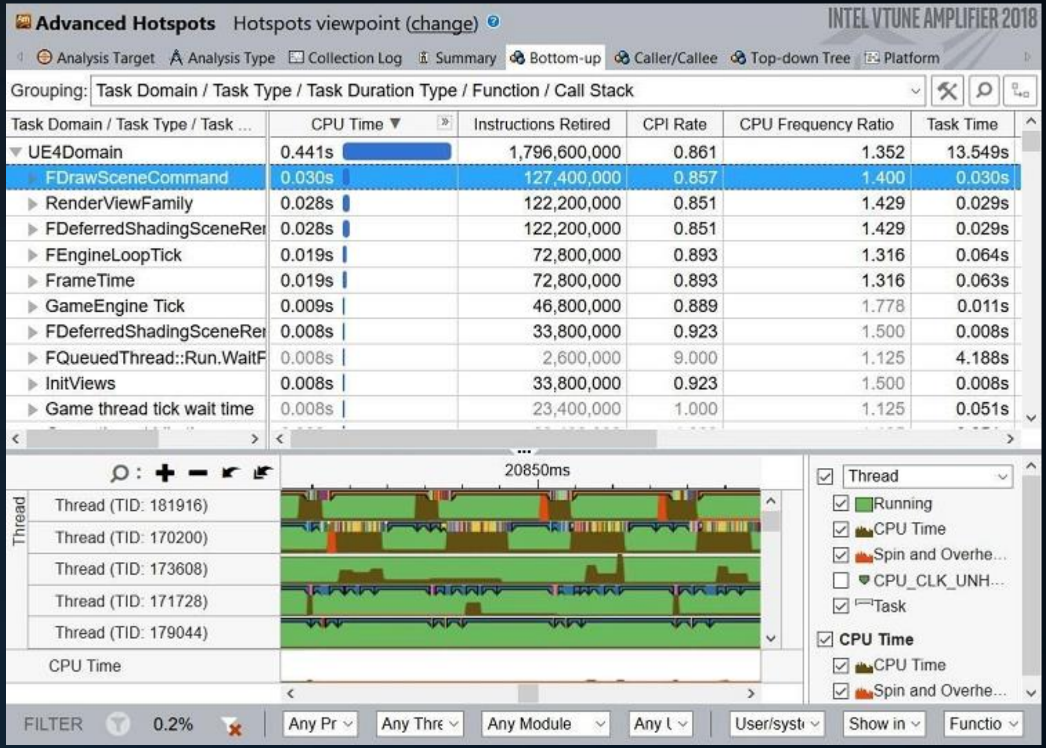
- Instruction pipelines
- Latencies
- Dependencies
- Bandwidth
- Cycles
- Floating point versus integer
- SIMD

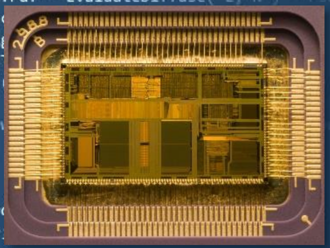
- Big O
- Research: not reinventing the wheel
- Data characteristics & algorithm choice
- STL: Trust No One
- As accurate as necessary (but not more)
- Balancing accuracy, speed and memory

What is optimization?

Don't assume, measure

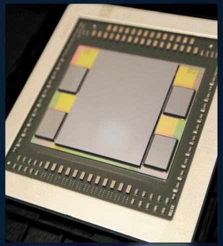
- Profilers
- Interpreting profiling data
- Instrumentation
- Bottlenecks
- Steering optimization effort





- Instruction pipelines
- Latencies
- Dependencies
- Bandwidth
- Cycles
- Floating point versus integer
- SIMD

- Big O
- Research: not reinventing the wheel
- Data characteristics & algorithm choice
- STL: Trust No One
- As accurate as necessary (but not more)
- Balancing accuracy, speed and memory



- Cache architecture
- Cache lines
- Hits, misses and collisions
- Eviction policies
- Prefetching
- Cache-oblivious
- Data-centric programming

What is optimization? – Project Management

Keeping code maintainable

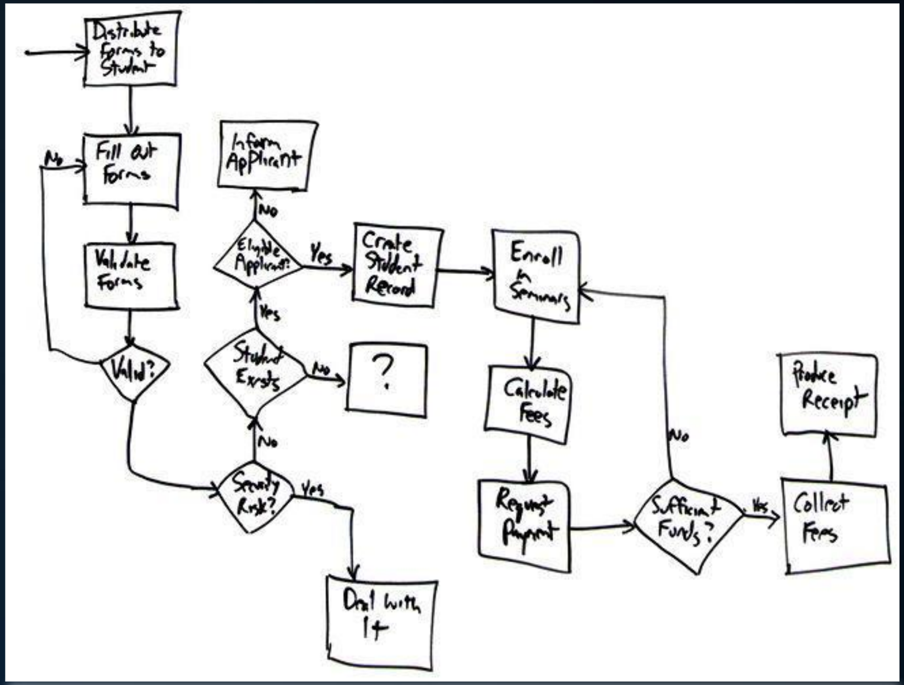
- Pareto principle / 80-20 rule: roughly 80% of the effects are caused by 20% of the causes.
- 1% of the code takes 99% of the time.

“The curse of premature optimization”

- Optimization, rule 1: “Don’t do it”.
- Rule 2 (for experts only!), “Don’t do it yet”.

Optimization as a deliberate process

- *Get predictable gains using a consistent approach.*



What is optimization?

"Perceived Performance"

1. Wait for user input
2. Respond to user input *as quickly as possible*
3. Execute requested operation.

Agenda:

- State of Affairs
- Optimization Introduction
- Profiling
- Low-level: Rules of Engagement
- The Cache

Consistent Approach

(0.) Determine optimization requirements
1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize / vectorize / use GPGPU
6. Profile again.
7. Apply low level optimizations to hotspots
8. Repeat step 6 and 7 until time runs out
9. Report.

Consistent Approach

(0.) Determine optimization requirements

- Target hardware (or range of hardware)
- Target performance
- Time available for optimization
- Constraints related to maintainability / portability
- …

1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize / vectorize / use GPGPU
6. Profile again.
7. Apply low level optimizations to hotspots
8. Repeat steps 6 and 7 until time runs out
9. Report.

From here on, we will assume that:

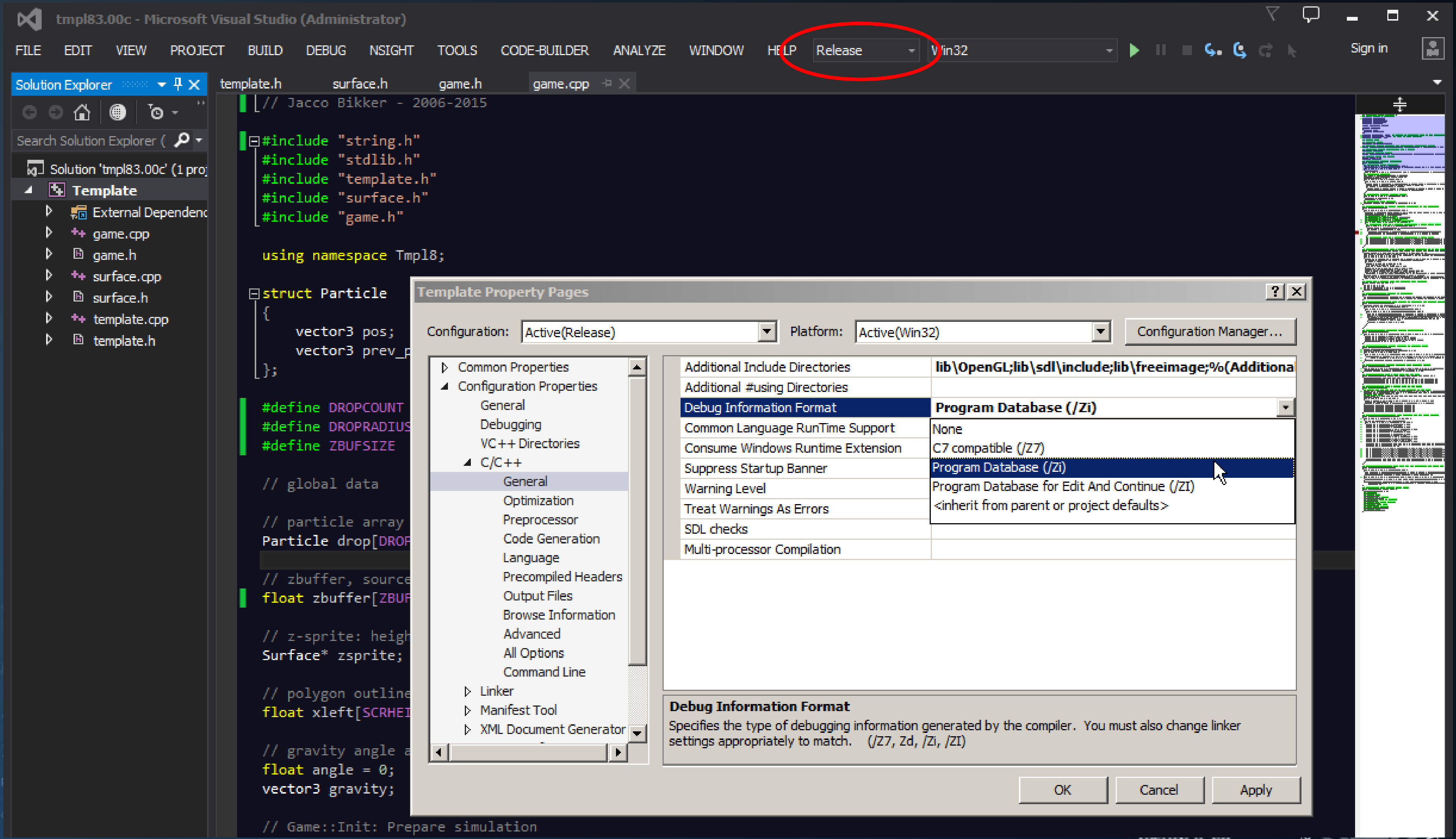
- the code is 'done' (feature complete);
- a speed improvement is required;
- we have a finite amount of time for this.

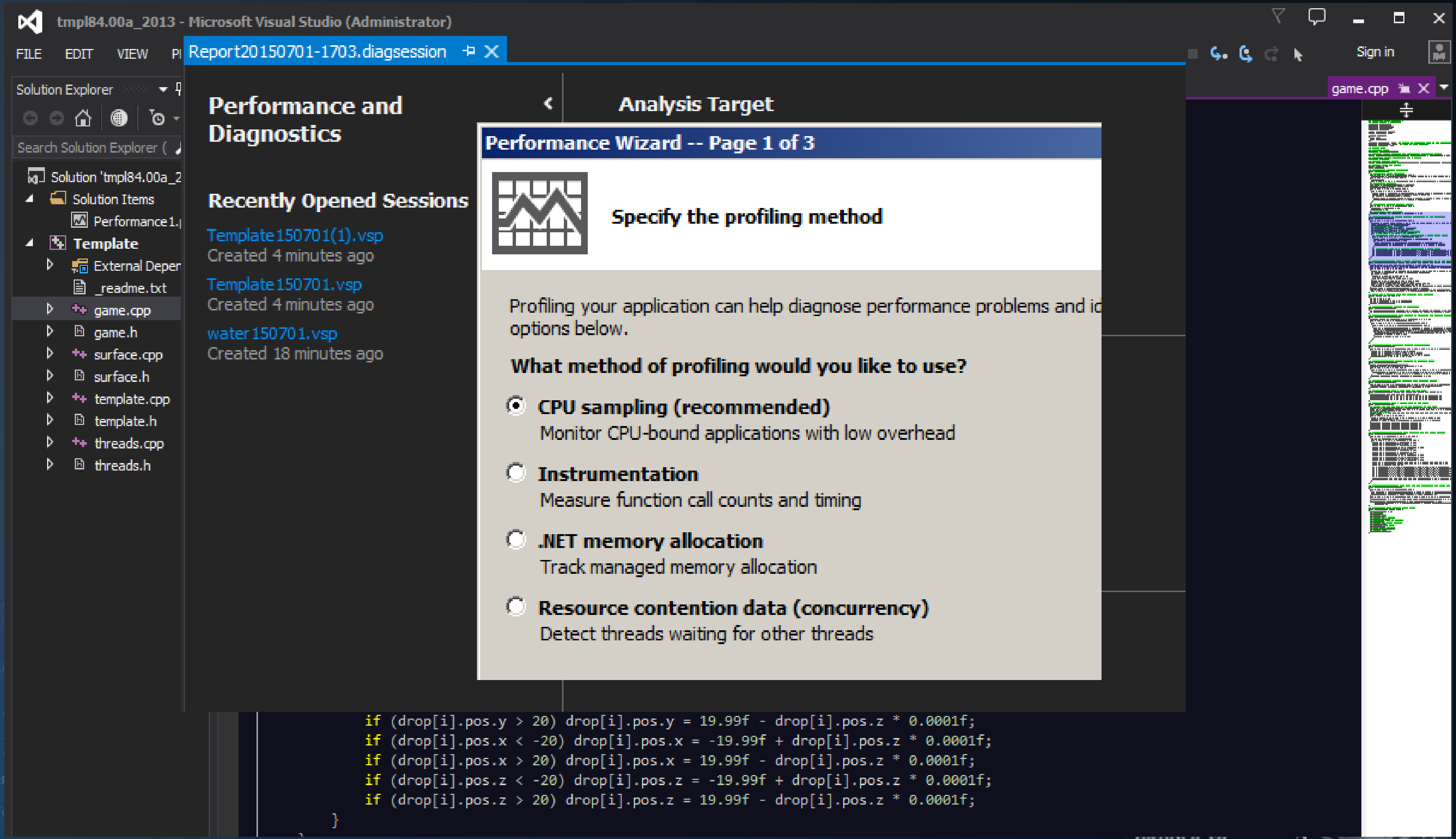
Consistent Approach

(0.) Determine optimization requirements
1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize / vectorize / use GPGPU
6. Profile again.
7. Apply low level optimizations to hotspots
8. Repeat steps 6 and 7 until time runs out
9. Report.

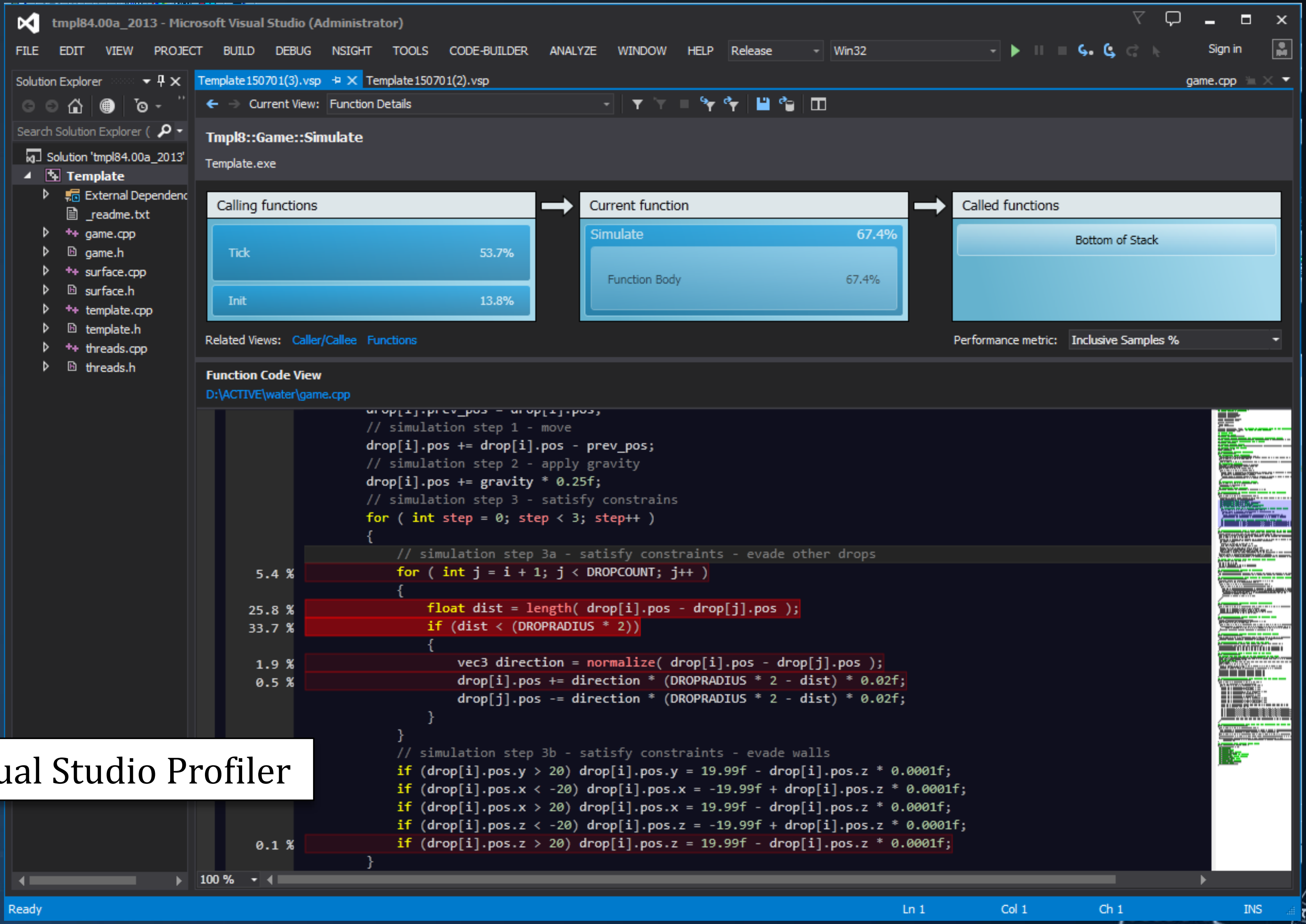
Consistent Approach

(0.) Determine optimization requirements
1. Profile: determine hotspots
2. Analyze hotspots: determine scalability
3. Apply high level optimizations to hotspots
4. Profile again.
5. Parallelize / vectorize / use GPGPU
6. Profile again.
7. Apply low level optimizations to hotspots
   - caching, data-centric programming,
   - removing superfluous functionality and precision,
   - aligning data to cache lines, vectorization,
   - checking compiler output, fixed point arithmetic,
   - …
8. Repeat steps 6 and 7 until time runs out
9. Report.

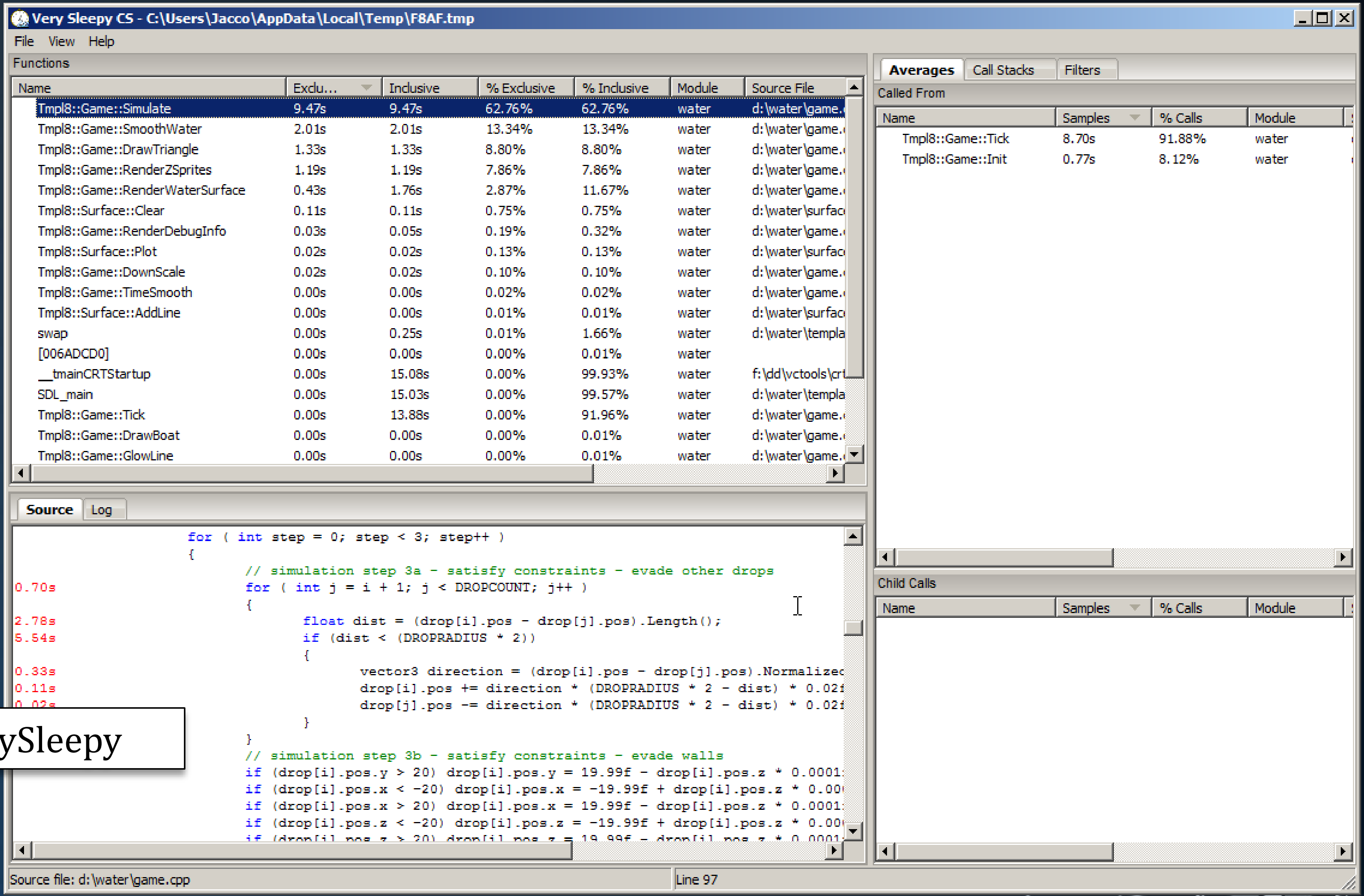

tmpl83.00c - Microsoft Visual Studio (Administrator)
FILE EDIT VIEW PROJECT BUILD DEBUG NSIGHT TOOLS CODE-BUILDER ANALYZE WINDOW HELP
Release
Win32
Sign in
Solution Explorer
Search Solution Explorer
Solution 'tmpl83.00c' (1 proj
Template
External Dependenc
game.cpp
game.h
surface.cpp
surface.h
template.cpp
template.h
template.h surface.h game.h game.cpp
// Jacco Bikker - 2006-2015
#include "string.h"
#include "stdlib.h"
#include "template.h"
#include "surface.h"
#include "game.h"
using namespace Tmpl8;
struct Particle
vector3 pos;
#define DROPCOUNT
#define DROPRADIUS
#define ZBUFSIZE
// global data
// particle array
// zbuffer, source
float zbuffer[ZBUF
// z-sprite: heigh
Surface* zsprite;
// polygon outline
// gravity angle a
float angle = 0;
vector3 gravity;
// Game::Init: Prepare simulation
Template Property Pages
Configuration: Active(Release)
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
VC++ Directories
C/C++
General
Optimization
Preprocessor
Code Generation
Language
Precompiled Headers
Output Files
Browse Information
Advanced
All Options
Command Line
Linker
Manifest Tool
XML Document Generator
Additional Include Directories
lib\OpenGL;lib\sdl\include;lib\freeimage;%(Additiona
Additional #using Directories
Debug Information Format
Program Database (/Zi)
Common Language RunTime Support
Consume Windows Runtime Extension
Suppress Startup Banner
Warning Level
Treat Warnings As Errors
SDL checks
Multi-processor Compilation
None
C7 compatible (/Z7)
Program Database (/Zi)
Program Database for Edit And Continue (/ZI)
<inherit from parent or project defaults>
Debug Information Format
Specifies the type of debugging information generated by the compiler. You must also change linker settings appropriately to match. (/Z7, Zd, /Zi, /ZI)
OK
Cancel
Apply


tmpl84.00a_2013 - Microsoft Visual Studio (Administrator)
FILE
EDIT
VIEW
Report20150701-1703.diagsession
Sign in
game.cpp
Solution Explorer
Search Solution Explorer (
Solution 'tmpl84.00a_2
Solution Items
Performance1.
Template
External Deper
_readme.txt
game.cpp
game.h
surface.cpp
surface.h
template.cpp
template.h
threads.cpp
threads.h
Performance and Diagnostics
Analysis Target
Recently Opened Sessions
Template150701(1).vsp
Created 4 minutes ago
Template150701.vsp
Created 4 minutes ago
water150701.vsp
Created 18 minutes ago
Performance Wizard -- Page 1 of 3
Specify the profiling method
Profiling your application can help diagnose performance problems and id
options below.
What method of profiling would you like to use?
CPU sampling (recommended)
Monitor CPU-bound applications with low overhead
Instrumentation
Measure function call counts and timing
.NET memory allocation
Track managed memory allocation
Resource contention data (concurrency)
Detect threads waiting for other threads
if (drop[i].pos.y > 20) drop[i].pos.y = 19.99f - drop[i].pos.z * 0.0001f;
if (drop[i].pos.x < -20) drop[i].pos.x = -19.99f + drop[i].pos.z * 0.0001f;
if (drop[i].pos.x > 20) drop[i].pos.x = 19.99f - drop[i].pos.z * 0.0001f;
if (drop[i].pos.z < -20) drop[i].pos.z = -19.99f + drop[i].pos.z * 0.0001f;
if (drop[i].pos.z > 20) drop[i].pos.z = 19.99f - drop[i].pos.z * 0.0001f;

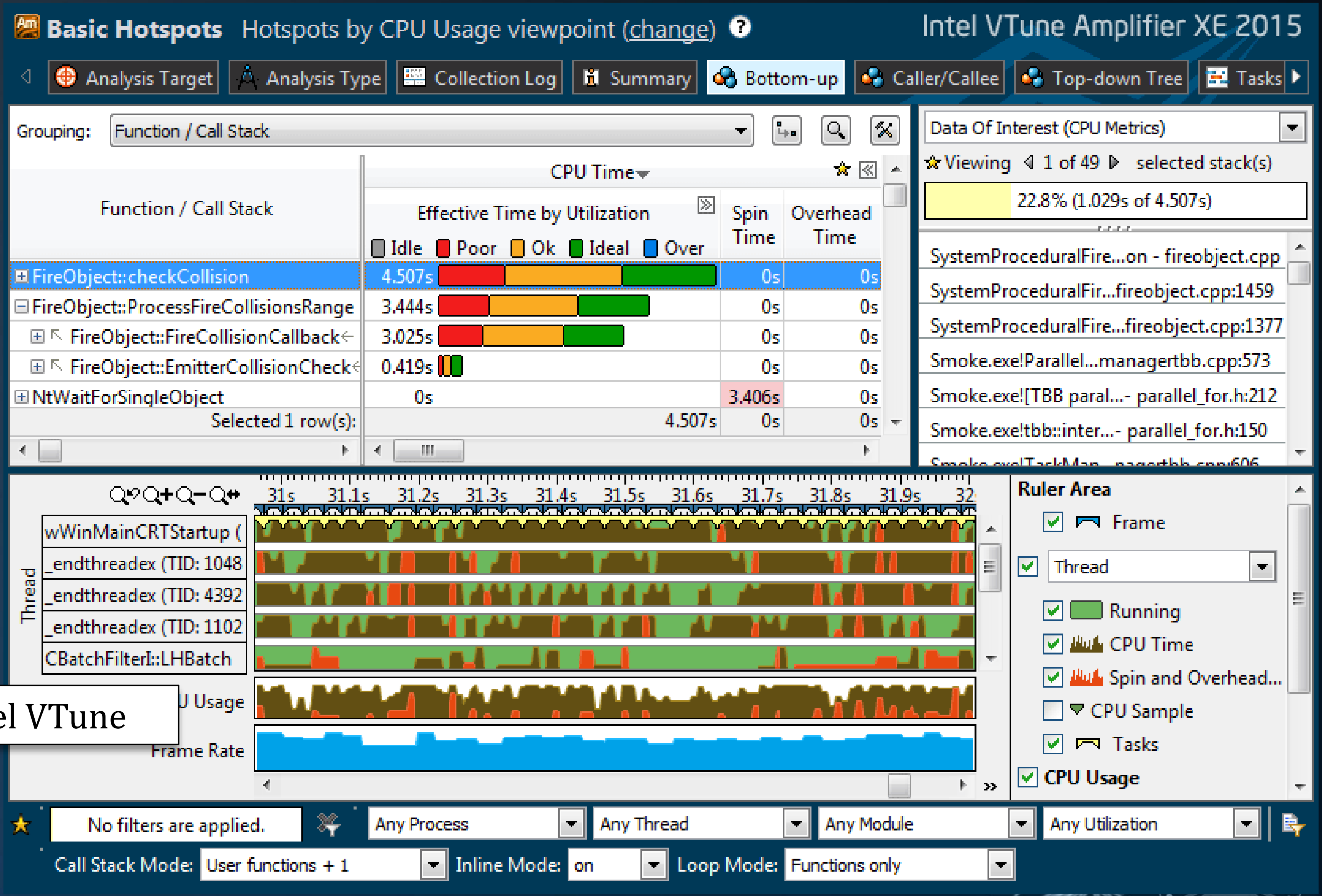
Ray Tracing for Games



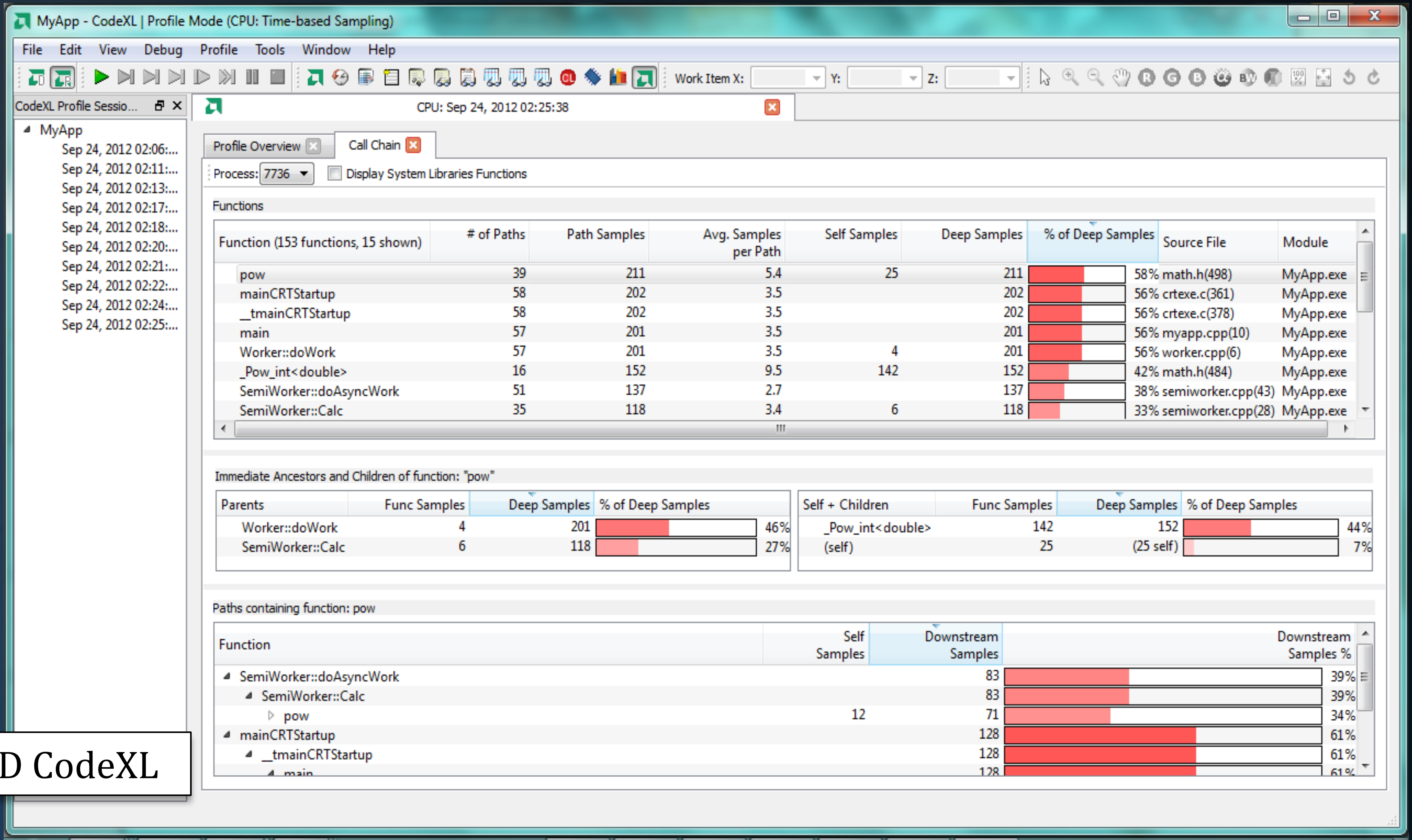
Visual Studio Profiler


Very Sleepy CS - C:\Users\Jacco\AppData\Local\Temp\F8AF.tmp
File View Help
Functions
Name | Exclu... | Inclusive | % Exclusive | % Inclusive | Module | Source File
Tmpl8::Game::Simulate 9.47s 9.47s 62.76% 62.76% water d:\water\game.
Tmpl8::Game::SmoothWater 2.01s 2.01s 13.34% 13.34% water d:\water\game.
Tmpl8::Game::DrawTriangle 1.33s 1.33s 8.80% 8.80% water d:\water\game.
Tmpl8::Game::RenderZSprites 1.19s 1.19s 7.86% 7.86% water d:\water\game.
Tmpl8::Game::RenderWaterSurface 0.43s 1.76s 2.87% 11.67% water d:\water\game.
Tmpl8::Surface::Clear 0.11s 0.11s 0.75% 0.75% water d:\water\surfac
Tmpl8::Game::RenderDebugInfo 0.03s 0.05s 0.19% 0.32% water d:\water\game.
Tmpl8::Surface::Plot 0.02s 0.02s 0.13% 0.13% water d:\water\surfac
Tmpl8::Game::DownScale 0.02s 0.02s 0.10% 0.10% water d:\water\game.
Tmpl8::Game::TimeSmooth 0.00s 0.00s 0.02% 0.02% water d:\water\game.
Tmpl8::Surface::AddLine 0.00s 0.00s 0.01% 0.01% water d:\water\surfac
swap 0.00s 0.25s 0.01% 1.66% water d:\water\templa
[006ADCD0] 0.00s 0.00s 0.00% 0.01% water
__tmainCRTStartup 0.00s 15.08s 0.00% 99.93% water f:\dd\vctools\crt
SDL_main 0.00s 15.03s 0.00% 99.57% water d:\water\templa
Tmpl8::Game::Tick 0.00s 13.88s 0.00% 91.96% water d:\water\game.
Tmpl8::Game::DrawBoat 0.00s 0.00s 0.00% 0.01% water d:\water\game.
Tmpl8::Game::GlowLine 0.00s 0.00s 0.00% 0.01% water d:\water\game.
Averages Call Stacks Filters
Called From
Name | Samples | % Calls | Module
Tmpl8::Game::Tick 8.70s 91.88% water
Tmpl8::Game::Init 0.77s 8.12% water
Child Calls
Name | Samples | % Calls | Module
Source Log
for (int step = 0; step < 3; step++)
{
// simulation step 3a - satisfy constraints - evade other drops
0.70s for (int j = i + 1; j < DROPCOUNT; j++)
{
2.78s float dist = (drop[i].pos - drop[j].pos).Length();
5.54s if (dist < (DROPRADIUS * 2))
{
0.33s vector3 direction = (drop[i].pos - drop[j].pos).Normalized
0.11s drop[i].pos += direction * (DROPRADIUS * 2 - dist) * 0.02f
drop[j].pos -= direction * (DROPRADIUS * 2 - dist) * 0.02f
}
}
// simulation step 3b - satisfy constraints - evade walls
if (drop[i].pos.y > 20) drop[i].pos.y = 19.99f - drop[i].pos.z * 0.0001
if (drop[i].pos.x < -20) drop[i].pos.x = -19.99f + drop[i].pos.z * 0.00
if (drop[i].pos.x > 20) drop[i].pos.x = 19.99f - drop[i].pos.z * 0.0001
if (drop[i].pos.z < -20) drop[i].pos.z = -19.99f + drop[i].pos.z * 0.00
Source file: d:\water\game.cpp
Line 97

VerySleepy


Basic Hotspots Hotspots by CPU Usage viewpoint (change)
Intel VTune Amplifier XE 2015
Analysis Target
Analysis Type
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Tasks
Grouping: Function / Call Stack
Function / Call Stack
CPU Time
Effective Time by Utilization
Idle Poor Ok Ideal Over
Spin Time
Overhead Time
FireObject::checkCollision 4.507s 0s 0s
FireObject::ProcessFireCollisionsRange 3.444s 0s 0s
FireObject::FireCollisionCallback 3.025s 0s 0s
FireObject::EmitterCollisionCheck 0.419s 0s 0s
NtWaitForSingleObject 0s 3.406s 0s
Selected 1 row(s): 4.507s 0s 0s
Data Of Interest (CPU Metrics)
Viewing 1 of 49 selected stack(s)
22.8% (1.029s of 4.507s)
SystemProceduralFire...on - fireobject.cpp
SystemProceduralFir...fireobject.cpp:1459
SystemProceduralFire...fireobject.cpp:1377
Smoke.exe!Parallel...managertbb.cpp:573
Smoke.exe![TBB paral...- parallel_for.h:212
Smoke.exe!tbb::inter...- parallel_for.h:150
31s 31.1s 31.2s 31.3s 31.4s 31.5s 31.6s 31.7s 31.8s 31.9s 32
Thread
wWinMainCRTStartup (
_endthreadex (TID: 1048
_endthreadex (TID: 4392
_endthreadex (TID: 1102
CBatchFilterI::LHBatch
U Usage
Frame Rate
Ruler Area
Frame
Thread
Running
CPU Time
Spin and Overhead...
CPU Sample
Tasks
CPU Usage
No filters are applied.
Any Process
Any Thread
Any Module
Any Utilization
Call Stack Mode: User functions + 1
Inline Mode: on
Loop Mode: Functions only

Intel VTune


MyApp - CodeXL | Profile Mode (CPU: Time-based Sampling)
File Edit View Debug Profile Tools Window Help
Work Item X: Y: Z:
CodeXL Profile Sessio...
MyApp
Sep 24, 2012 02:06:...
Sep 24, 2012 02:11:...
Sep 24, 2012 02:13:...
Sep 24, 2012 02:17:...
Sep 24, 2012 02:18:...
Sep 24, 2012 02:20:...
Sep 24, 2012 02:21:...
Sep 24, 2012 02:22:...
Sep 24, 2012 02:24:...
Sep 24, 2012 02:25:...
CPU: Sep 24, 2012 02:25:38
Profile Overview
Call Chain
Process: 7736
Display System Libraries Functions
Functions
Function (153 functions, 15 shown)
# of Paths
Path Samples
Avg. Samples per Path
Self Samples
Deep Samples
% of Deep Samples
Source File
Module
pow 39 211 5.4 25 211 58% math.h(498) MyApp.exe
mainCRTStartup 58 202 3.5 202 56% crtexe.c(361) MyApp.exe
__tmainCRTStartup 58 202 3.5 202 56% crtexe.c(378) MyApp.exe
main 57 201 3.5 201 56% myapp.cpp(10) MyApp.exe
Worker::doWork 57 201 3.5 4 201 56% worker.cpp(6) MyApp.exe
_Pow_int<double> 16 152 9.5 142 152 42% math.h(484) MyApp.exe
SemiWorker::doAsyncWork 51 137 2.7 137 38% semiworker.cpp(43) MyApp.exe
SemiWorker::Calc 35 118 3.4 6 118 33% semiworker.cpp(28) MyApp.exe
Immediate Ancestors and Children of function: "pow"
Parents
Func Samples
Deep Samples
% of Deep Samples
Worker::doWork 4 201 46%
SemiWorker::Calc 6 118 27%
Self + Children
Func Samples
Deep Samples
% of Deep Samples
_Pow_int<double> 142 152 44%
(self) 25 (25 self) 7%
Paths containing function: pow
Function
Self Samples
Downstream Samples
Downstream Samples %
SemiWorker::doAsyncWork 83 39%
SemiWorker::Calc 83 39%
pow 12 71 34%
mainCRTStartup 128 61%
__tmainCRTStartup 128 61%

AMD CodeXL

Agenda:

- State of Affairs
- Optimization Introduction
- Profiling
- Low-level: Rules of Engagement
- The Cache

What is the 'cost' of a multiply?

```
starttimer();
float x = 0;
for( int i = 0; i < 1000000; i++ ) x *= y;
stoptimer();
```

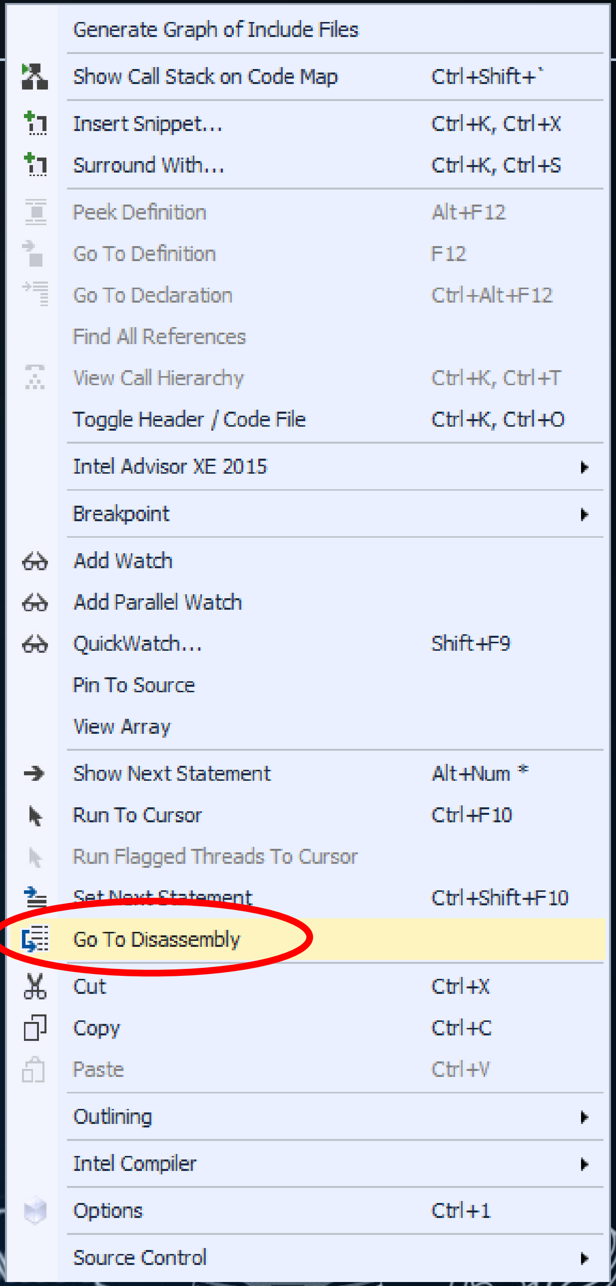
- Actual measured operations:
  - timer operations;
  - initializing 'x' and 'i';
  - comparing 'i' to 1000000 (x 1000000);
  - increasing 'i' (x 1000000);
  - jump instruction to start of loop (x 1000000).
- Compiler outsmarts us!
  - No work at all unless we use x
  - x += 1000000 * y

Better solution:

- Create an arbitrary loop
- Measure time with and without the instruction we want to time

What is the 'cost' of a multiply?

```
float x = 0, y = 0.1f;
unsigned int i = 0, j = 0x28929227;
for( int k = 0; k < ITERATIONS; k++ )
{
    // ensure we feed our line with fresh data
    x += y, y *= 1.01f;
    // integer operations to free up fp execution units
    i += j, j ^= 0x17737352, i >>= 1, j /= 28763;
    // operation to be timed
    if (with) x *= y;
    // integer operations to free up fp execution units
    i += j, j ^= 0x17737352, i >>= 1, j /= 28763;
}
dummy = x + (float)i;
```



x86 assembly in 5 minutes

Modern CPUs still run x86 machine code, based on Intel's 1978 8086 processor. The original processor was 16-bit, and had 8 'general purpose' 16-bit registers*:

AX ('accumulator register')	AH, AL (8-bit)	EAX (32-bit)	RAX (64-bit)
BX ('base register')	BH, BL	EBX	RBX
CX ('counter register')	CH, CL	ECX	RCX
DX ('data register')	DH, DL	EDX	RDX
BP ('base pointer')		EBP	RBP
SI ('source index')		ESI	RSI
DI ('destination index')		EDI	RDI
SP ('stack pointer')		ESP	RSP
			R8..R15
		st0..st7	
		XMM0..XMM7	XMM0..XMM15
			YMM0..YMM15
			ZMM0..ZMM31

* More info: http://www.swansontec.com/sregisters.html

x86 assembly in 5 minutes:

Typical assembler:

```
loop:
    mov eax, [0x1008FFA0]     // read from address into register
    shr eax, 5                // shift eax 5 bits to the right
    add eax, edx              // add registers, store in eax
    dec ecx                   // decrement ecx
    jnz loop                  // jump if not zero
    fld [esi]                 // load from address [esi] onto FPU
    fld st0                   // duplicate top float
    faddp                     // add top two values, push result
```

More on x86 assembler: http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
A bit more on floating point assembler: https://www.cs.uaf.edu/2007/fall/cs301/lecture/11_12_floating_asm.html

What is the 'cost' of a multiply?

```
float x = 0, y = 0.1f;
unsigned int i = 0, j = 0x28929227;
for( int k = 0; k < ITERATIONS; k++ )
{
    // ...
    x += y, y *= 1.01f;
    // ...
    i += j, j ^= 0x17737352, i >>= 1, j /= 28763;
    // ...
    if (with) x *= y;
    // ...
    i += j, j ^= 0x17737352, i >>= 1, j /= 28763;
}
dummy = x + (float)i;
```

```
fldz
xor ecx, ecx
fld dword ptr ds:[405290h]
mov edx, 28929227h
fld dword ptr ds:[40528Ch]
push esi
mov esi, 0C350h
```

$= 50000$

```
add ecx, edx
mov eax, 91D2A969h
xor edx, 17737352h
shr ecx, 1
mul eax, edx
fld st(1)
faddp st(3), st

mov eax, 91D2A969h
shr edx, 0Eh
add ecx, edx
fmul st(1),st
xor edx, 17737352h
shr ecx, 1
mul eax, edx
shr edx, 0Eh
dec esi
jne tobetimed<0>+1Fh
```

$= \frac{2^{46}}{28763}$ (!!)

What is the ‘cost’ of a multiply?

Observations:

- Compiler reorganizes code
- Compiler cleverly evades division
- Loop counter *decreases*
- Presence of integer instructions affects timing
  (to the point where the mul is free)

But also:

- It is really hard to measure the cost of a line of code.

What is the ‘cost’ of a single instruction?

Cost is highly dependent on the surrounding instructions, and many other factors. However, there is a ‘cost ranking’:

<< >>	*bit shifts*
+ - & | ^	*simple arithmetic, logical operands*
*	*multiplication*
/	*division*
sqrt	
sin, cos, tan, pow, exp	

This ranking is generally true for any processor (including GPUs).

Common Opportunities in Low-level Optimization

RULE 1: Avoid Costly Operations

- Replace multiplications by bitshifts, when possible
- Replace divisions by (reciprocal) multiplications
- Avoid sin, cos, sqrt

Common Opportunities in Low-level Optimization

RULE 2: Precalculate

- Reuse (partial) results
- Adapt previous results (interpolation, reprojection, …)
- Loop hoisting
- Lookup tables

Common Opportunities in Low-level Optimization

RULE 3: Pick the Right Data Type

- Avoid byte, short, double
- Use each data type as a 32/64 bit container that can be used at will
- Avoid conversions, especially to/from float
- Blend integer and float computations
- Combine calculations on small data using larger data

Common Opportunities in Low-level Optimization

RULE 4: Avoid Conditional Branches

- if, while, ?, MIN/MAX
- Try to split loops with conditional paths into multiple unconditional loops
- Use lookup tables to prevent conditional code
- Use loop unrolling
- If all else fails: make conditional branches predictable

Common Opportunities in Low-level Optimization

RULE 5: Early Out

```
char a[] = “abcdfghijklmnopqrstuvwxyz”;
char c = ‘p’;
int position = -1;
for ( int t = 0; t < strlen( a ); t++ )
{
    if (a[t] == c)
    {
        position = t;
    }
}
```

```
char a[] = “abcdfghijklmnopqrstuvwxyz”;
char c = ‘p’;
int position = -1, len = strlen( a );
for ( int t = 0; t < len; t++ )
{
    if (a[t] == c)
    {
        position = t;
        break;
    }
}
```

Common Opportunities in Low-level Optimization

RULE 6: Use the Power of Two

- A multiplication / division by a power of two is a (cheap) bitshift
- A 2D array lookup is a multiplication too – make 'width' a power of 2
- Dividing a circle in 256 or 512 works just as well as 360 (but it's faster)
- Bitmasking (for free modulo) requires powers of 2

1-2-4-8-16-32-64-128-256-512-1024-2048-4096-8192-16384-32768-65536

Be fluent with powers of 2 (up to 2^16);
learn to go back and forth for these: 2^9 = 512 = 2^9.
Practice counting from 0..31 on one hand in binary.

Common Opportunities in Low-level Optimization

RULE 7: Do Things Simultaneously

- Use those cores
- An integer holds four bytes; use these for instruction level parallelism
- More on this later.

Common Opportunities in Low-level Optimization

1. Avoid Costly Operations
2. Precalculate
3. Pick the Right Data Type
4. Avoid Conditional Branches
5. Early Out
6. Use the Power of Two
7. Do Things Simultaneously

Agenda:

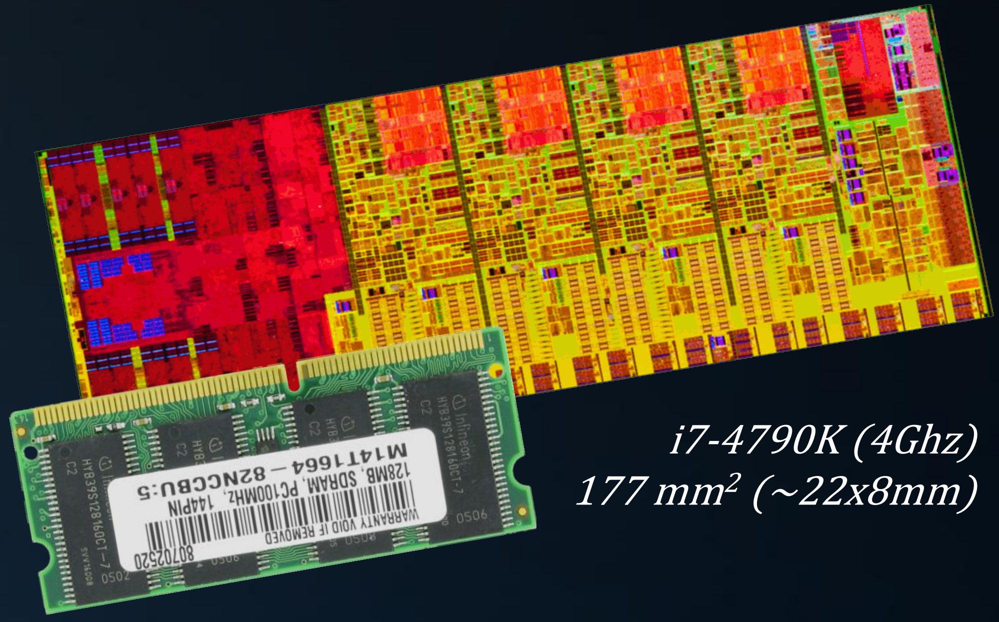
- State of Affairs
- Optimization Introduction
- Profiling
- Low-level: Rules of Engagement
- The Cache

Feeding the Beast

Let’s assume our CPU runs at 4Ghz.
What is the maximum physical distance between memory and CPU if we want to retrieve data every cycle?

Speed of light (vacuum): 299,792,458 m/s
Per cycle: ~0.075 m
➔ ~<u>3.75cm</u> back and forth.

In other words: we cannot physically query RAM fast enough to keep a CPU running at full speed.



i7-4790K (4Ghz)
177 mm^2 (~22x8mm)

Feeding the Beast

Sadly, we can't just divide by the physical distance between CPU and RAM to get the cycles required to query memory.

Factors include (stats for DDR4-3200/PC4-25600):

- RAM runs at a much lower clock speed than the CPU
  - 25600 here means: theoretical bandwidth in MB/s
  - 3200 is the number of transfers per second (1 transfer=64bit)
  - We get two transfers per cycle, so actual I/O clock speed is 1600Mhz
  - DRAM cell array clock is ~1/4th of that: 400Mhz.
- Latency between query and response: 20-24 cycles.

Feeding the Beast

Sadly, we can’t just divide by the physical distance between CPU and RAM to get the cycles required to query memory.

Additional delays may occur when:

- Other devices than the CPU access RAM;
- DRAM must be refreshed every 64ms due to leakage.

For a processor running at 2.66GHz, latency is roughly 110-140 CPU cycles.

Details in: “What Every Programmer Should Know About Memory”, chapter 2.

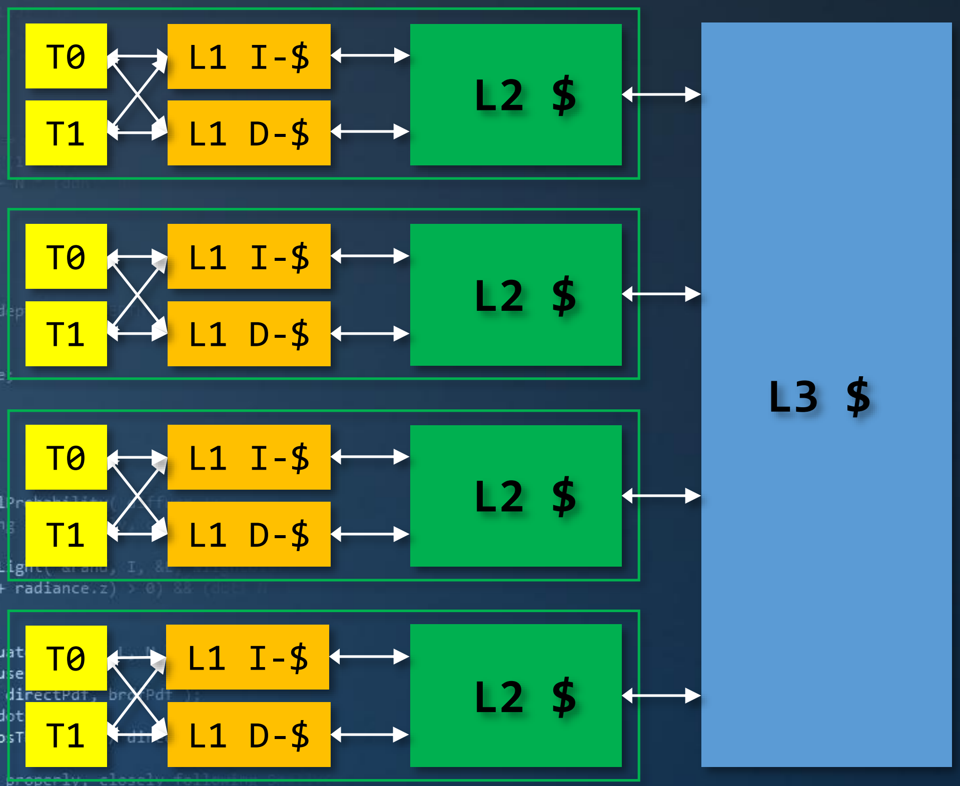
Feeding the Beast

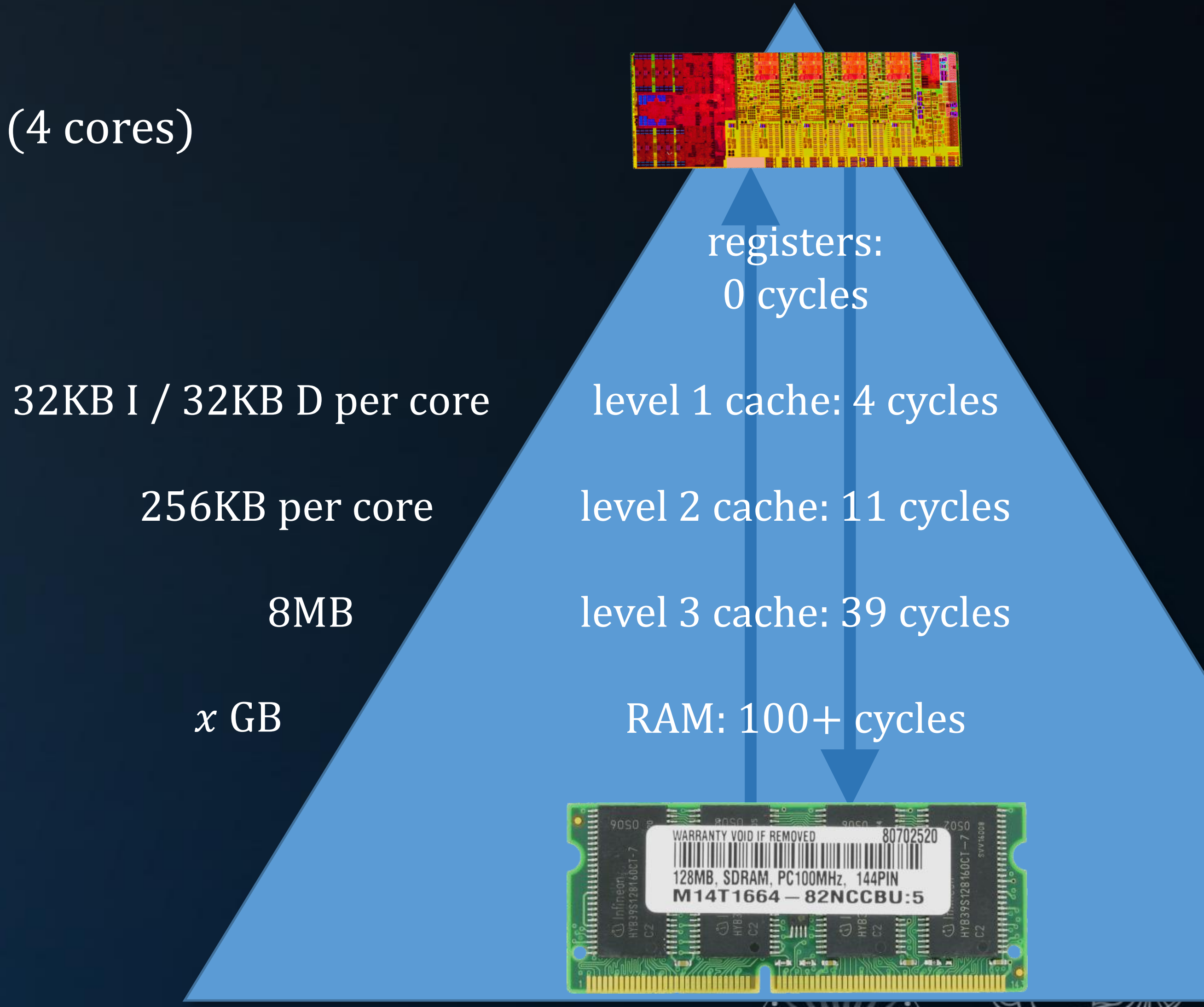
"We cannot physically query RAM fast enough to keep a CPU running at full speed."

How do we overcome this?

We keep a copy of frequently used data in fast memory, close to the CPU: the ***cache***.

The Memory Hierarchy – Core i7-9xx (4 cores)





Caches and Optimization

Considering the cost of RAM vs L1$ access, it is clear that the cache is an important factor in code optimization:

- Fast code communicates mostly with the caches
- We still need to get data into the caches
- But ideally, only once.

Therefore:

- The working set must be small;
- Or we must maximize *data locality*.

Data Locality

Wikipedia:

Temporal Locality – “If at one point in time a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future.”

Spatial Locality – “If a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future.”

* More info: http://gameprogrammingpatterns.com/data-locality.html

Data Locality

How do we increase data locality?

Linear access – Sometimes as simple as swapping for loops *

Tiling – Example of working on a small subset of the data at a time.

Streaming – Operate on/with data until done.

Reducing data size – Smaller things are closer together.

How do trees/linked lists/hash tables fit into this?

* For an elaborate example see https://www.cs.duke.edu/courses/cps104/spring11/lects/19-cache-sw2.pdf

Two more issues:

1. Cache deals in lines of 64 bytes.

2. Mind ‘false sharing’.

How to Please the Cache

Or: “how to evade RAM”

1. Keep your data in registers

Use fewer variables
Limit the scope of your variables
Pack multiple values in a single variable
Use floats and ints (they use different registers)
Compile for 64-bit (more registers)
Arrays will never go in registers
Unions will never go in registers

How to Please the Cache

Or: "how to evade RAM"

2. Keep your data local

Read sequentially
Keep data small
Use tiling / Morton order
Fetch data once, work until done (streaming)
Reuse memory locations

How to Please the Cache

Or: “how to evade RAM”

3. Respect cache line boundaries

Use padding if needed
Don’t pad for sequential access
Use aligned malloc / __declspec align
Assume 64-byte cache lines

How to Please the Cache

Or: “how to evade RAM”

4. Advanced tricks

Prefetch
Use a prefetch thread (theoretical…)
Use *streaming writes*
Separate mutable / immutable data

How to Please the Cache

Or: “how to evade RAM”

5. Be informed

Use the profiler!

Agenda:

- State of Affairs
- Optimization Introduction
- Profiling
- Low-level: Rules of Engagement
- The Cache

Ray Tracing for Games

Wednesday
13:00 – 17:00

course intro
LH2
template
Whitted
refactoring
RT-centric games

LAB 1

Thursday
09:00 – 14:00

advanced Whitted
audio, AI & physics
faster Whitted
Heaven7

LAB 2

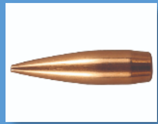
work @ home

End result day 2:

A solid Whitted-style ray tracer, as a basis for subsequent work.

Friday
09:00 – 17:00

optimization
profiling, rules of engagement
threading



LAB 3

SIMD
applied SIMD
SIMD triangle
SIMD AABB

LAB 4

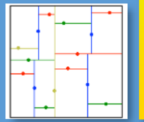
End result day 3:

A 5x faster tracer.

GLOBAL GAME JAM

Monday
09:00 – 17:00

acceleration
grid, BVH, kD-tree
SAH
binning



LAB 5

refitting
top-level BVH
threaded building

LAB 6

End result day 4:

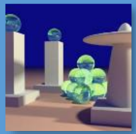
A real-time tracer.

Tuesday
09:00 – 17:00

Monte-Carlo
Cook-style
glossy, AA
area lights, DOF

LAB 7

path tracing



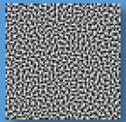
LAB 8

End result day 5:

Cook or Kajiya.

Thursday
09:00 – 17:00

random numbers
stratification
blue noise



LAB 9

importance sampling
next event estimation

LAB 10

End result day 6:

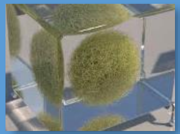
Efficiency.

Friday
09:00 – 17:00

future work

LAB 11

path guiding



LAB 10

End result day 6:

Great product.

Take some time to speed up your ray tracer.

Low level:

Don’t do more sqrt/sin/pow than you absolutely have to.
Don’t trust stl, write your own code.
Limit conditionals.
Don’t write values that you will later overwrite.
Rule of thumb: a single-threaded ray tracer for 3 spheres and a plane runs at 30+fps.

Cache / memory:

Reduce the amount of data you touch
- Don’t new/delete anything after initialization.
- Reuse the ray.

Improve data locality
- Operate on tiles.

General guidelines:

Clean code is fast code.
It can always be made faster.
Speedups do not add up, they multiply.

End of PART 3.