Welcome!
Today's Agenda:

- Introduction
- Course Formalities
- High Level Overview
- Profiling
Why?

*Some problems require the supercomputer of the future.*

- Anything that depends on Moore’s Law and time to become feasible.
Introduction

Why?

Games want to raise the bar:

- More, better, faster. Also: be scalable.
Introduction

Why?

Some software needs to run on pretty weak hardware.

- Limited CPU, limited RAM (limited controls).
Introduction

Why?

*Some software should not use 90% of your CPU.*

- Leave room for other applications, be invisible.
Introduction

Why?

*Sometimes the cheapest / lowest power CPU is the best.*

- What is the lowest end CPU this will still run on? Can we go lower?
Introduction

Why?

Some things are done so frequently, they must be efficient.

- Memory manager
- Garbage collector
- JIT compiler
- Compilers in general
- Image processing in Photoshop
- OS startup / resume
- ...
- ...

…
What is optimization?

Part of it is:

- INFOB3CC - Concurrency
- INFONW - Computerarchitectuur en netwerken
- INFOB3TC - Talen en compilers

And of course: any course that deals with improving existing algorithms.

Specific purpose of INFOMOV:

- To gain understanding of performance aspects of the hardware we use;
- To gain an intuition for what affects performance;
- To learn to apply a structured process to improve performance.
What is optimization?

Working with the hardware: *think like a CPU*

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

What is optimization?

Work smarter, not harder: algorithm scalability

- 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
Introduction

What is optimization?

Memory hierarchy: caches

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

Don’t assume, measure

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

What is optimization?

“Perceived Performance”

1. Wait for user input
2. Respond to user input as quickly as possible
3. Execute requested operation.
At the end of this course:

**You will know how to speed up critical code by a factor 2x to 10x (and more).**

- You will be able to do this to virtually any program*.
- Your understanding of higher level optimization approaches will increase.
- You will be able to apply these principles to new / alien hardware.
- You will have a more intimate relationship with your computer.

In other words:

We will talk a lot about the ‘C’ in O(N).

---

*disclaimer: ‘that has not been optimized by an expert’.
Today’s Agenda:

- Introduction
- Course Formalities
- High Level Overview
- Profiling
INFOMOV – Lecture 1 – “Introduction”

Formalities

Lecturer

Dr. Jacco Bikker
j.bikker@uu.nl
Room 4.24 BBG
 Formalities

Course Layout

8 weeks + exam week:

- 2 lectures per week (for exceptions: see website)

Assessment:

- 3 assignments (20% each, individual or pairs);
- 1 final assignment (40%, individual or pairs);
- 1 final theory exam.
Prerequisites

C++

English

Hardware / software

You'll need access to a computer with a CPU that supports SSE2 and OpenCL. Obtaining VTune (Intel CPU) or CodeXL (AMD CPU) is beneficial (VTune is free to try for 30 days). Visual Studio 2013, 2015 or 2017 will be needed.

Other tools will be free.
Formalities

Literature

No book!
But that doesn’t mean you won’t be reading.

Main documents:

(also see his website: http://agner.org)


You are encouraged to do research into specific topics of interest yourself, and to report on this in class.
Formalities

Audience

Any computer science student
(with a slight bias towards games)

Make sure you get as much as possible out of this course. This automatically includes a free pass.
Today’s Agenda:

- Introduction
- Course Formalities
- High Level Overview
- Profiling
Overview

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
6. Use GPGPU
7. Profile again.
8. Apply low level optimizations to hotspots
9. Repeat steps 7 and 8 until time runs out
Overview

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
6. Use GPGPU
7. Profile again.
8. Apply low level optimizations to hotspots
9. Repeat steps 7 and 8 until time runs out

From here on, we will assume that:

- the code is ‘done’ (feature complete);
- a speed improvement is desired;
- we have a finite amount of time for this.
Overview

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
6. Use GPGPU
7. Profile again.
8. Apply low level optimizations to hotspots
9. Repeat steps 7 and 8 until time runs out
Consistent Approach

1. Determine optimization requirements
2. Profile: determine hotspots
3. Analyze hotspots: determine scalability
4. Apply high level optimizations to hotspots
5. Profile again.
6. Parallelize
7. Use GPGPU
8. Profile again.
9. 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,
   - ...
10. Repeat steps 7 and 8 until time runs out
Overview

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
6. Use GPGPU
7. Profile again.
8. Apply low level optimizations to hotspots
9. Repeat steps 7 and 8 until time runs out
Assembler

In this course, we will not write assembler:

- It takes a pro to outperform the compiler
- You will be fighting the compiler
- You will have to redo the optimization for every target processor
- Maintainability will be zero.
"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%." (Donald Knuth)
“A significant improvement in performance can often be achieved by solving only the actual problem and removing extraneous functionality.” (Wikipedia)
“More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity.” (W.A. Wulff)
“Dear Charles,

In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selection amongst them (...).

One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.

Therefore, one should attend INFOMOV and learn from it.

Love, Ada.”
Today’s Agenda:

- Introduction
- Course Formalities
- High Level Overview
- Profiling
Never Assume

**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
6. Use GPGPU
7. Profile again.
8. Apply low level optimizations to hotspots
9. Repeat steps 7 and 8 until time runs out

Do you actually need to speed it up?
By how much?

Things to consider:

- You have a finite amount of time for this
- You don’t want to break anything
- You don’t want to reduce maintainability

▶ **Focus on ‘low hanging fruit’** – typically a small portion of the code.
Never Assume

Consistent Approach

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

**Don’t trust your intuition**

- Not even when optimizing your own code.
- *Especially* not when you are proficient at optimizing.

Blind changes may reduce the performance of the code.

Needless to say: *use version control.*
Never Assume

Profiling

Measuring application performance

- Using external tools
- Using timers in the code

Measurements:

- How much time is spent were? (inclusive / exclusive, cycles, percentage)
- How often is each function called?
- Low level behavior: stalls / latencies, branch mispredictions, occupation, ...
- Performance over time: lag, spikes, stutter
What if the goal is to have a 10x larger army in your RTS?

Don't just measure performance, measure *scalability*. 

Never Assume
Profiling – getting accurate results

A profiler needs information about your code: this is typically available in debug builds.

However:
Debug builds have very different performance characteristics, for many reasons. We need to profile in release mode.

Enabling debug information in release mode in Visual Studio:

- Properties >> C/C++ >> General >> Debug information format
- Properties >> Linker >> Debugging >> Generate Debug Info

Differences between a debug and release configurations

In debug:
- your code is not optimized
- debug info is added to the executable
- variables are initialized
- memory blocks are padded with guard bytes
- array bounds are checked

In release:
- code may be reordered

IMPORTANT:
It makes very little sense to optimize in debug mode.
Never Assume
Never Assume
INFOMOV – Lecture 1 – “Introduction”

Tools

Performance and Diagnostics

Recently Opened Sessions
- Template150701.vsp
  Created 4 minutes ago
- Template150701.vsp
  Created 4 minutes ago
- water150701.vsp
  Created 19 minutes ago

Specify the profiling method

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.y * 0.0001f;
Tools

Visual Studio Profiler
**Tools**

**INFOMOV – Lecture 1 – “Introduction”**

<table>
<thead>
<tr>
<th>Name</th>
<th>Reduced</th>
<th>Inclusive</th>
<th>1% Excl.</th>
<th>5% Excl.</th>
<th>Module</th>
<th>Source File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer: Game/Scene</td>
<td>9.4%</td>
<td>9.4%</td>
<td>0.7%</td>
<td>0.7%</td>
<td>water</td>
<td>d:\water\game.c</td>
</tr>
<tr>
<td>Timer: Game/Sound</td>
<td>3.3%</td>
<td>3.3%</td>
<td>8.3%</td>
<td>8.3%</td>
<td>water</td>
<td>d:\water\game.c</td>
</tr>
<tr>
<td>Timer: Game/DrawTriangle</td>
<td>1.1%</td>
<td>1.1%</td>
<td>7.6%</td>
<td>7.6%</td>
<td>water</td>
<td>d:\water\game.c</td>
</tr>
<tr>
<td>Timer: Game/RandomVertex1</td>
<td>0.4%</td>
<td>0.4%</td>
<td>1.7%</td>
<td>1.7%</td>
<td>water</td>
<td>d:\water\game.c</td>
</tr>
<tr>
<td>Debug: Surface/Clear</td>
<td>0.1%</td>
<td>0.1%</td>
<td>0.7%</td>
<td>0.7%</td>
<td>water</td>
<td>d:\water\game.c</td>
</tr>
<tr>
<td>Debug: Game/DropDebugInfo</td>
<td>0.3%</td>
<td>0.3%</td>
<td>0.1%</td>
<td>0.1%</td>
<td>water</td>
<td>d:\water\game.c</td>
</tr>
<tr>
<td>Debug: Surface/Poi</td>
<td>0.2%</td>
<td>0.2%</td>
<td>0.3%</td>
<td>0.3%</td>
<td>water</td>
<td>d:\water\game.c</td>
</tr>
<tr>
<td>Debug: Game/DropPoi</td>
<td>0.1%</td>
<td>0.1%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>water</td>
<td>d:\water\game.c</td>
</tr>
<tr>
<td>Debug: Game/Smooth</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>water</td>
<td>d:\water\game.c</td>
</tr>
<tr>
<td>Debug: Game/PoiDato</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>water</td>
<td>d:\water\game.c</td>
</tr>
<tr>
<td>Debug: Game/Smooth</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>water</td>
<td>d:\water\game.c</td>
</tr>
</tbody>
</table>

**Averages**

- **Name**: TimeSleepee
  - **Samples**: 8.7s
  - **% Cells**: 91.88% on water
  - **Module**: c:\water\game.c

---

---
Tools

INFOMOV – Lecture 1 – “Introduction”
Tools

AMD CodeXL
Never Assume

Take-away:

Never assume. Profiling *always* steers optimization.

Optimize in release mode. Enable debug info during this process. Don’t forget to turn it off before distribution.
Profiler Output

```

```
Profiler Output

```plaintext
// simulation step 1 - move
drop[1].pos = drop[1].pos - pow.pow;
// simulation step 2 - apply gravity
drop[1].pos = drop[1].pos - gravity * 0.056;
// simulation step 3 - satisfy constraints
for (-int step = 0; step < 3; step++)
{
    // simulation step 3a - satisfy constraints - move other drops
    for (int step = 1 + 1; 2 < DROPCOUNT; ++)
    {
        Stick dist = length (drop[1].pos - drop[1].pos);
        if (dist < (DROPDISTANCE + 2))
        {
            vec3 direction = normalize (drop[1].pos - drop[1].pos);
            drop[1].pos = direction + (DROPDISTANCE + 2 - dist) * 0.022;
            drop[1].pos = direction + (DROPDISTANCE + 2 - dist) * 0.022;
        }
    }
}
```
Profiling – Results

Game::Simulate 67.89% 67.89%
Game::SmoothWater 10.54% 10.54%
Game::RenderZSprites 7.18% 7.18%
Game::Tick 0.00% 76.32%

Running ~3 seconds, we spent 0.86s on this line:

float dist = length( drop[i].pos - drop[j].pos );

and 1.68s on this line:

if (dist < (DROPRADIUS * 2))
Profiler Output

Profiling – finding hotspots

The profiler allows you to quickly find the parts of your program that take most time.

But:

- Mind debug versus release;
- The profiler doesn’t tell you *why* a function is costly
- The profiler doesn’t report scalability
- There is no ‘cost over time’ information

- **Scalability analysis requires running the program with different work sets (i.e., change N in O(N)).**
- **Determining why a section takes a lot of time requires more in-depth knowledge.**
- **Solving** the performance issue requires even more in-depth knowledge.
Profiler Output
## Profiler Output

```c
// Game: Simulate a Verlet physics simulation on the particles

void Game::Simulate()
{
    // ENTER(SIMULATE);
    for (int i = 0; i < BOUNDARY; i++)
    {
        gravity = vesi2* sin(a + Pi / 180), cos(a + Pi / 180), 0);
        vec3 prev_pos = drop[i].prev_pos;
        drop[i].prev_pos = drop[i].pos;
        // simulation step 1 - move
        drop[i].pos = drop[i].pos - prev_pos;
        // simulation step 2 - apply gravity
        drop[i].pos = gravity * 0.25f;
        // simulation step 3 - satisfy constraints
        for (int step = 0; step < 3; step++)
        {
            // simulation step 8a - satisfy constraints - evade other drops
            for (int j = 1 + 1; j < BOUNDARY; j++)
            {
                float dist = (drop[i].pos - drop[j].pos);
                if (dist < (DROPRADIUS + 2))
                {
                    vec3 direction = normalize(drop[j].pos - drop[i].pos);
                    if (drop[j].pos.y > 20) drop[j].pos.y = -19.99f - drop[i].pos.y;
                    if (drop[j].pos.x < -20) drop[j].pos.x = -19.99f - drop[i].pos.x;
                    if (drop[j].pos.x > 20) drop[j].pos.x = 19.99f - drop[i].pos.x;
                }
            }
            // simulation step 8b - satisfy constraints - evade walls
            if (drop[i].pos.y > 20) drop[i].pos.y = -19.99f;
            if (drop[i].pos.x < -20) drop[i].pos.x = -19.99f;
            if (drop[i].pos.x > 20) drop[i].pos.x = 19.99f;
        }
        // simulate velocity
        if (weight == "diffuse")
        {
            fric = SAMPLEINT(1, 100, 100, 1000, 10000);
        }
        else if (weight == "stroke")
        {
            fric = SAMPLEINT(1, 100, 100, 1000, 10000);
        }
        else
        {
            fric = SAMPLEINT(1, 100, 100, 1000, 10000);
        }
        // leave
        LEAVE(SIMULATE);
    }
}
```
Take-away:

Free, vendor-agnostic profilers tell you where time is spent in your program (but not why).

Vendor-specific tools provide a wealth of information, but generally require knowledge about the hardware processes.

Stalls are generally not vendor-specific and will be similar on similar hardware.

Just timing information is often sufficient to make an educated guess towards improvements.
Custom Profiling

Generic Profiler Downsides

- No ‘performance over time’ measurements
- Requires inclusion of debug information (including source code)
- Not real-time
- Not very intuitive

Using a custom in-app profiler we can drastically improve our profiling information.
Custom Profiling
Custom Profiling

Minecraft Beta 1.8 Pre-release 2 (118 fps, 2 chunk updates)
C: 979/5408, F: 191.2, 0: 0, E: 2517
F: 1/251, B: 0, I: 250
P: 0, T: FLX: 251
Server Chunk Cache: 979 Drop: 9

Seed: 5130996236305320162

Used memory: 41x 162MB of 989MB
Allocated memory: 100x (989MB)

Minecraft
Custom Profiling

UnrealEngine 3
Custom Profiling

CryEngine
Custom Profiling

StarCraft II
Custom Profiling

StarCraft II
Custom Profiling

Take-away:

In-app profiling provides advantages over external profilers:

- You get real-time information, which is easily associated with what is going on in the app;
- You can measure statistics that are not available to the profiler;
- You can present the data in a form that is also useful to people not familiar with the intricacies of the profiler.
Considerations

Custom timers: what to measure?

- Time spent in your code
- ‘Wall clock time’
- Cycles

In what quantities?

- A millisecond is a *long time*
- Averaged / smoothed values are easier to read
- Relative performance may be better

The impact of measurements:

- Especially relevant for brief snippets of code
- Logging is expensive!
What if the CPU isn’t the problem?

A modern system consists of:

- A multicore CPU;
- A many-core GPU.

If one of them is idling, your application isn’t running at maximum performance.

Move work, even if the other processor can’t do it as efficient.
Considerations

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
6. Use GPGPU
7. Profile again.
8. Apply low level optimizations to hotspots
9. Repeat steps 7 and 8 until time runs out
And Finally:

Profiling:

Without it, no optimization – we need to *know*

How to profile: tools, custom timers, CPU + GPU

What to profile: realistically (release!), raw performance, scalability *(but also: cache misses, pipelining, branch prediction)*

Keep in mind: profiling takes time too.

Repeated profiling: things change, if you’re doing it right. Stay informed.
END of “Introduction”

next lecture: “Low Level”