

```
ics
& (depth < MAXDEPTH)
{
    inside / inside
    nt = nt / nc;
    cos2t = 1.0f - nt;
    D, N );
}
at a = nt - nc, b = nt;
at Tr = 1 - (R0 + (1 - R0) *
Tr) R = (D * nnt - N * (a
E * diffuse;
= true;
efl + refr)) && (depth < MAXDEPTH)
D, N );
efl * E * diffuse;
= true;
MAXDEPTH)
survive = SurvivalProbability( diffuse );
estimation - doing it properly, class
if;
radiance = SampleLight( &rand, I, &t, &align,
e.x + radiance.y + radiance.z) && (rand.N
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
random walk - done properly, closely following
ive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
urvive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

# /INFOMOV/

# Optimization & Vectorization

J. Bikker - Sep-Nov 2015 - Lecture 2: "Low Level"

# Welcome!



# Catching up:

- Profiling

```
ics
& (depth < MAXDEPTH)
{
    t = inside / 1.5;
    nt = nt / nc;
    cos2t = 1.0f - nnt * nnt;
    D, N );
}

at a = nt - nc; b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) *
Tr) R = (D * nnt - N * (a *
E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
D, N );
efl * E * diffuse;
= true;

MAXDEPTH)
survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &t, &align,
e.x + radiance.y + radiance.z) > 0) && (survive)
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
random walk - done properly, closely following
vive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
urvive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```



# 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
10. Report.

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

- (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
10. Report.

## *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.*

And:

```
#ifndef OPTIMIZEDCODE
    // same functionality, hopefully
#else
    // original slow code
#endif
```



# Never Assume

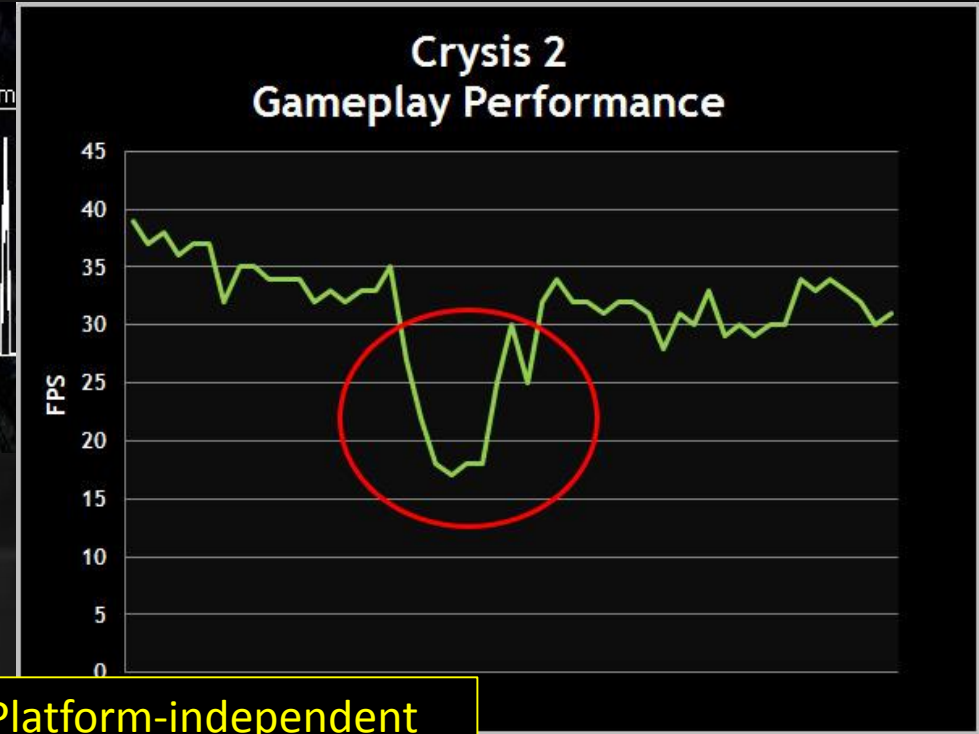
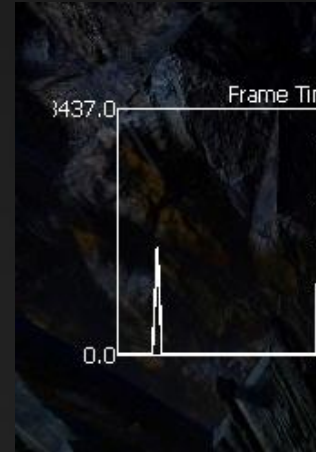
## Profiling

### 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



Platform-independent

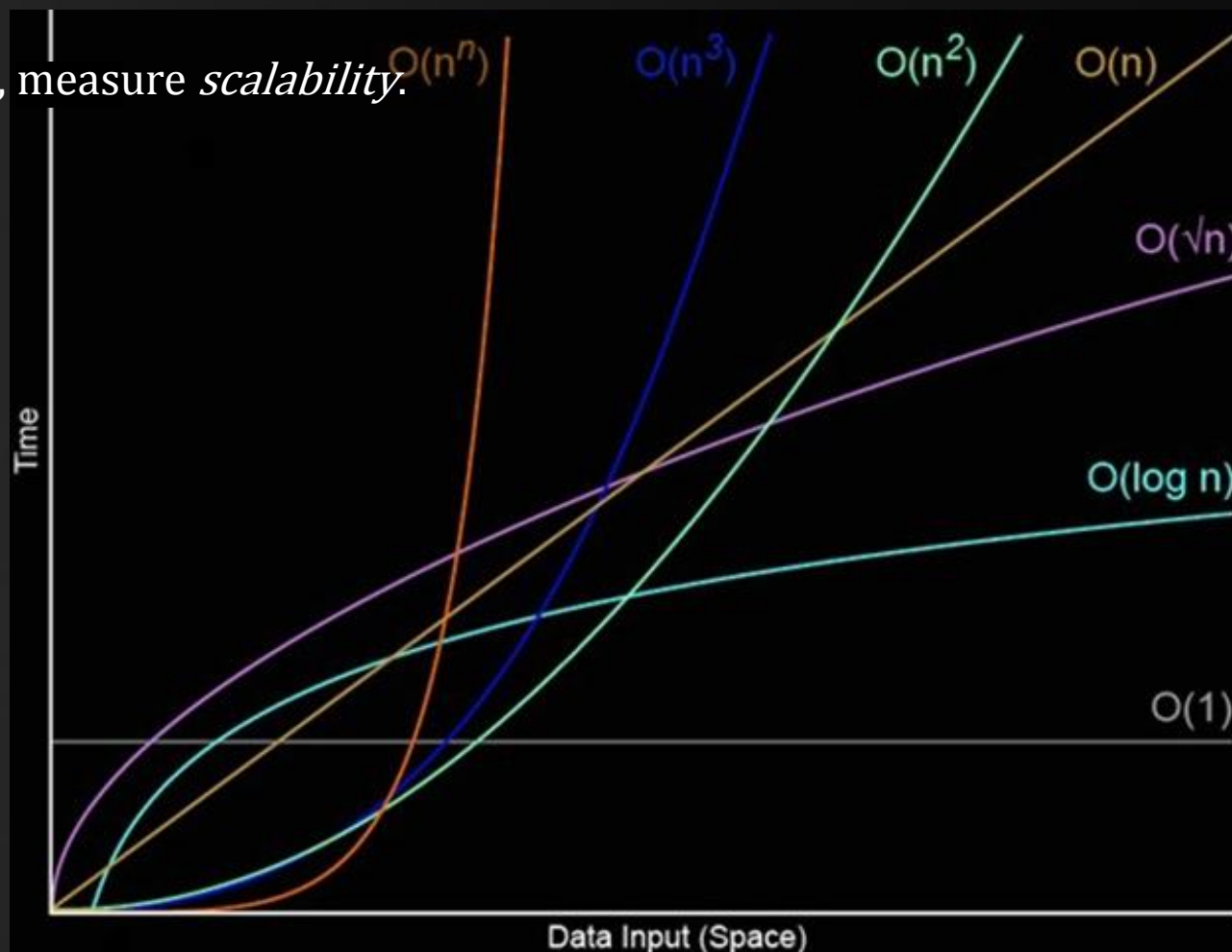
Platform-dependent



# Never Assume

What if the goal is to have a 10x larger army in your RTS?

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



```

...
    & (depth < MAXDEPTH)
...
    inside / inside
    nt = nt / nc;
    pos2t = 1.0f - nnt;
    D, N );
    )
...
    at a = nt - nc, b = nt;
    at Tr = 1 - (R0 + (1 - R0)
    Tr) R = (D * nnt - N * (
...
    E * diffuse;
    = true;
...
    refl + refr) && (depth < MAXDEPTH)
...
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
survive = SurvivalProbability( diffuse );
estimation - doing it properly, class
if;
    radiance = SampleLight( &rand, I, &t, &align;
    e.x + radiance.y + radiance.z) && (
...
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * P;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf;
survive;
pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```



# Profiler Output

```

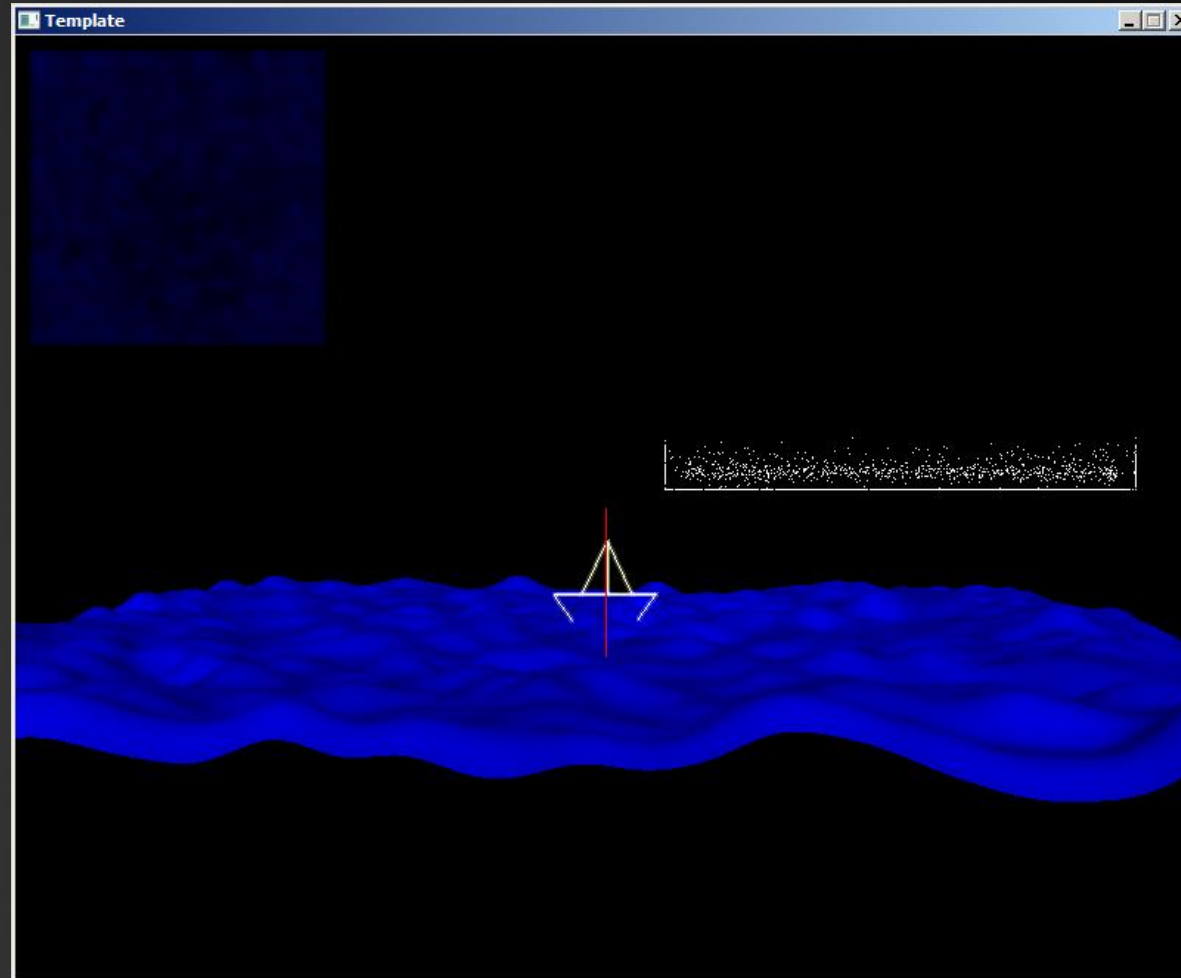
ics
& (depth < MAXDEPTH)
{
    t = inside / 1.5;
    nt = nt / nc;
    cos2t = 1.0f - nt * nt;
    D, N );
}

at a = nt - nc, b = nt * nc;
at Tr = 1 - (R0 + (1 - R0) *
Tr) R = (D * nnt - N * (a *
E * diffuse;
= true;

efl + refr) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;

MAXDEPTH)
survive = SurvivalProbability( diffuse );
estimation - doing it properly, class
if;
radiance = SampleLight( &rand, I, &t, &align;
e.x + radiance.y + radiance.z) > 0) && (survive)
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * survive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
random walk - done properly, closely following
ive)
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf;
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



# 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

The screenshot shows the Microsoft Visual Studio interface for a project named 'tmpl83.00c'. The main window displays the source file 'game.cpp' with the following code:

```

// 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;
    vector3 prev_p;
};

#define DROPCOUNT 1000
#define DROPRADIUS 100
#define ZBUFSIZE 1000

// global data
// particle array
Particle drop[DROPCOUNT];

// zbuffer, source
float zbuffer[ZBUFSIZE];

// z-sprite: height
Surface* zsprite;

// polygon outline
float xleft[SCRHEIGHT];

// gravity angle
float angle = 0;
vector3 gravity;

// Game::Init: Prepare simulation

```

The 'Template Property Pages' dialog is open, showing the following configuration:

- Configuration: Active(Release)
- Platform: Active(Win32)
- Additional Include Directories: lib\OpenGL\lib\sdl\include;lib\freeimage;% (Additional)
- Additional #using Directories: (empty)
- Debug Information Format: Program Database (/ZI)
- Common Language RunTime Support: None
- Consume Windows Runtime Extension: C7 compatible (/Z7)
- Suppress Startup Banner: Program Database (/ZI)
- Warning Level: Program Database for Edit And Continue (/ZI)
- Treat Warnings As Errors: <inherit from parent or project defaults>
- SDL checks: (empty)
- Multi-processor Compilation: (empty)

The 'Debug Information Format' section is expanded, showing the following text:

**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)



# Never Assume

The screenshot shows the Visual Studio IDE with the 'Template Property Pages' dialog box open. The dialog is titled 'Template Property Pages' and shows the configuration for 'Active(Release)' on the 'Active(Win32)' platform. The 'Generate Debug Info' property is set to 'Yes (/DEBUG)'. A red circle highlights the 'Release' dropdown in the top menu bar.

Property	Value
Generate Debug Info	Yes (/DEBUG)
Generate Program Database File	No
Strip Private Symbols	Yes (/DEBUG)
Generate Map File	<inherit from parent or project defaults>
Map File Name	
Map Exports	No
Debuggable Assembly	

Below the table, there is a section for 'Generate Debug Info' with the text: 'The /DEBUG option creates debugging information for the .exe file or DLL.'



# Tools

The screenshot shows the Microsoft Visual Studio interface. The main window displays a C++ code file named 'game.cpp'. On the left, the Solution Explorer shows a project structure with folders for 'Solution Items', 'Performance1', and 'Template', and files for 'game.cpp', 'game.h', 'surface.cpp', 'surface.h', 'template.cpp', 'template.h', 'threads.cpp', and 'threads.h'. The Performance and Diagnostics tool window is open, showing a 'Performance Wizard -- Page 1 of 3' dialog. The dialog title is 'Analysis Target' and the main heading is 'Specify the profiling method'. It lists four options:

- 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

Below the dialog, a snippet of C++ code is visible:

```

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;
    
```



# Tools

The screenshot displays the Visual Studio Profiler interface. The top pane shows the 'Function Details' view for the 'Simulate' function in 'Template.exe'. It includes a call graph with 'Calling functions' (Tick: 53.7%, Init: 13.8%), 'Current function' (Simulate: 67.4%, Function Body: 67.4%), and 'Called functions' (Bottom of Stack). The bottom pane shows the 'Function Code View' for the 'Simulate' function, with a performance overlay on the code. The code includes simulation steps for moving, applying gravity, and satisfying constraints.

Function	Percentage
Tick	53.7%
Init	13.8%
Simulate	67.4%
Function Body	67.4%

```

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 constrains
for ( int step = 0; step < 3; step++ )
{
    // simulation step 3a - satisfy constraints - evade other drops
    for ( int j = i + 1; j < DROPCOUNT; j++ )
    {
        float dist = length( drop[i].pos - drop[j].pos );
        if ( dist < ( DROPRADIUS * 2 ) )
        {
            vec3 direction = normalize( drop[i].pos - drop[j].pos );
            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 * 0.0001f;
    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;
}
    
```

Visual Studio Profiler



# Tools

**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\cr...
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
    for ( int j = i + 1; j < DROPCOUNT; j++ )
    {
        float dist = (drop[i].pos - drop[j].pos).Length();
        if (dist < (DROPRADIUS * 2))
        {
            vector3 direction = (drop[i].pos - drop[j].pos).Normalized;
            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 * 0.0001f;
    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;
}
    
```

Source file: d:\water\game.cpp Line 97

VerySleepy



# Tools

**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					Spin Time	Overhead Time
	Effective Time by Utilization						
	Idle	Poor	Ok	Ideal	Over		
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

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

Smoke.exe!TaskMan...managertbb.cpp:606

Thread Utilization Chart (Time: 31s to 32s)

- wWinMainCRTStartup (TID: 1048)
- \_endthreadex (TID: 4392)
- \_endthreadex (TID: 1102)
- CBatchFilter::LHBatch
- CPU 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



# Tools

MyApp - CodeXL | Profile Mode (CPU: Time-based Sampling)

CodeXL Profile Session... CPU: Sep 24, 2012 02:25:38

Process: 7736

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	Self + Children	Func Samples	Deep Samples	% of Deep Samples
Worker::doWork	4	201	46%	_Pow_int<double>	142	152	44%
SemiWorker::Calc	6	118	27%	(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%
▲ main		128	61%

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.

```

...
    & (depth < MAXDEPTH)
...
    inside / inside
    nt = nt / nc;
    cos2t = 1.0f - nt;
    D, N );
...
    at a = nt - nc, b = nt;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (a0
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
...
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
survive = SurvivalProbability( diffuse );
estimation - doing it properly, class
if;
radiance = SampleLight( @rand, I, Rt, Alignment
e.x + radiance.y + radiance.z) > 0) && (survive
...
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



# Profiler Output

Very Sleepy CS - C:\Users\Jacco\AppData\Local\Temp\B916.tmp

File View Help

Functions

Name	Exclu...	Inclusive	% Exclusive	% Inclusive	Module	Source File	Sourc...	Address
Tmpl8::Game::Simulate	3.06s	3.06s	67.89%	67.89%	Template	f:\projects\water\game.cpp	97	0x401761
Tmpl8::Game::SmoothWater	0.47s	0.47s	10.54%	10.54%	Template	f:\projects\water\game.cpp	206	0x401c20
Tmpl8::Game::RenderZSprites	0.32s	0.32s	7.18%	7.18%	Template	f:\projects\water\game.cpp	170	0x401a59
Tmpl8::Game::DrawTriangle	0.32s	0.32s	7.14%	7.14%	Template	f:\projects\water\game.cpp	119	0x402a6e
Tmpl8::Game::RenderWaterSurface	0.11s	0.44s	2.52%	9.66%	Template	f:\projects\water\game.cpp	278	0x40262e
Tmpl8::Surface::Plot	0.01s	0.01s	0.31%	0.31%	Template	f:\projects\water\surface.cpp	177	0x403910
Tmpl8::Surface::Clear	0.01s	0.01s	0.29%	0.29%	Template	f:\projects\water\surface.cpp	76	0x4036f7
[0062F8B6]	0.01s	0.01s	0.13%	0.13%	Template		0	0x62f8b6
[0062F8C0]	0.01s	0.01s	0.13%	0.13%	Template		0	0x62f8c0
Tmpl8::Game::RenderDebugInfo	0.01s	0.02s	0.13%	0.44%	Template	f:\projects\water\game.cpp	308	0x40299f
zbuffer	0.00s	0.01s	0.00%	0.13%	Template	[unknown]	0	0x4090f8
__tmainCRTStartup	0.00s	4.51s	0.00%	100.00%	Template	f:\dd\vctools\crt\crtw32\dlstuf...	618	0x405e0f
SDL_main	0.00s	4.49s	0.00%	99.58%	Template	f:\projects\water\template.cpp	252	0x404cc5
Tmpl8::Game::Tick	0.00s	3.44s	0.00%	76.32%	Template	f:\projects\water\game.cpp	320	0x402c6c
Tmpl8::Game::Init	0.00s	0.89s	0.00%	19.81%	Template	f:\projects\water\game.cpp	49	0x40146c
WinMain	0.00s	4.51s	0.00%	100.00%	Template	x:\projects\sdl\src\main\windo...	177	0x4010c7
main	0.00s	4.51s	0.00%	100.00%	Template	x:\projects\sdl\src\main\windo...	140	0x40101f

Averages Call Stacks Filters

Called From

Name	Samples	% Calls
Tmpl8::Game::Tick	2.17s	70.83%
Tmpl8::Game::Init	0.89s	29.17%

Child Calls

Name	Samples	% Calls
------	---------	---------

Source Log

```

// 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 constrains
for ( int step = 0; step < 3; step++ )
{
    // simulation step 3a - satisfy constraints - evade other drops
    for ( int j = i + 1; j < DROPCOUNT; j++ )
    {
        float dist = length( drop[i].pos - drop[j].pos );
        if ( dist < (DROPRADIUS * 2) )
        {
            vec3 direction = normalize( drop[i].pos - drop[j].pos );
            drop[i].pos += direction * (DROPRADIUS * 2 - dist) * 0.02f;
            drop[j].pos -= direction * (DROPRADIUS * 2 - dist) * 0.02f;
        }
    }
}

```

Source file: f:\projects\water\game.cpp Line 25



# Profiler Output

## Profiling – Results

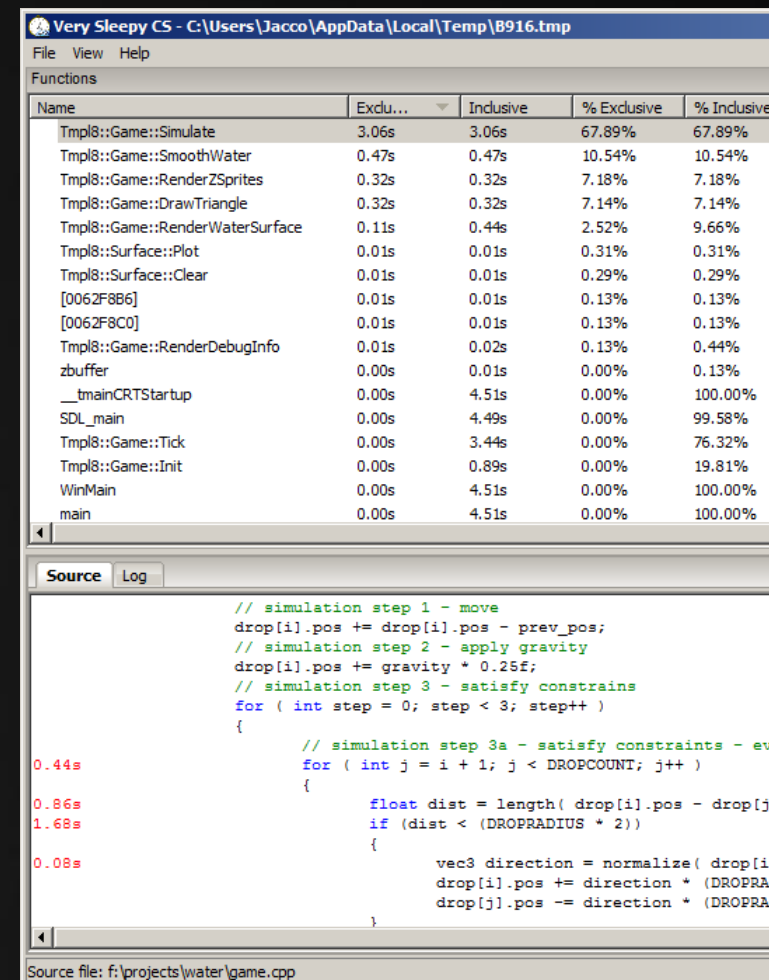
Game::Simulate	67.89%	67.89%
Game::SmoothWater	10.54%	10.54%
Game::RenderZSprites	7.18%	7.18%
Game::Tick	0.00%	76.32%

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

**Elapsed Time:** 11.571s

- Paused Time: 3.150s
- Clockticks: 3,710,005,565
- Instructions Retired: 7,236,010,854
- CPI Rate: 0.513

**Filled Pipeline Slots:**

- Retiring:** 0.774
  - General Retirement: 0.767
  - Microcode Sequencer: 0.007
- Bad Speculation:** 0.107
  - Branch Mispredict: 0.000
  - Machine Clears: 0.107

**Unfilled Pipeline Slots (Stalls):**

- Back-End Bound:** 0.070
- Memory Bound:** 0.000
- Core Bound:** 0.493
- Front-End Bound:** 0.049

**CPU Usage Histogram**

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.

Simultaneously Utilized Logical CPUs	Usage Category
0	Idle
1-4	Poor
5-7	Ok
8-9	Ideal



# Profiler Output

So. Lin. ▲	Source	Clockticks	Instructions Retired	CPI Rate	Filled Pipeline Slots		Unfilled Pipeline Slo ...	
					Retiring	Bad Spec ...	Back-End Bound	Fr. Bo.
129								
130	// Game::Simulate: do Verlet physics simulation on the particles							
131	void Game::Simulate()							
132	{							
133	ENTER(SIMULATE);							
134	for ( int i = 0; i < DROPCOUNT; i++ )							
135	{							
136	gravity = vec3( sin( angle * PI / 180 ), cos( angle * PI / 180 ), 0 );							
137	vec3 prev_pos = drop[i].prev_pos;							
138	drop[i].prev_pos = drop[i].pos;							
139	// simulation step 1 - move							
140	drop[i].pos += drop[i].pos - prev_pos;							
141	// simulation step 2 - apply gravity							
142	drop[i].pos += gravity * 0.25f;							
143	// simulation step 3 - satisfy constrains							
144	for ( int step = 0; step < 3; step++ )	0	2,000,003	0.000	0.000	0.000	1.000	0.000
145	{							
146	// simulation step 3a - satisfy constraints - evade other drops							
147	for ( int j = i + 1; j < DROPCOUNT; j++ )	128,000,192	148,000,222	0.865	0.703	0.000	0.824	0.059
148	{							
149	float dist = length( drop[i].pos - drop[j].pos );	780,001,170	208,000,312	3.750	1.000	0.000	0.231	0.010
150	if (dist < (DROPRADIUS * 2))	1,042,001,563	2,818,004,227	0.370	0.115	0.446	0.381	0.058
151	{							
152	vec3 direction = normalize( drop[i].pos - drop[j].pos );	64,000,096	86,000,129	0.744	0.820	0.000	0.180	0.000
153	drop[i].pos += direction * (DROPRADIUS * 2 - dist) * 0.02f;	22,000,033	0		0.000	1.000	0.000	1.000
154	drop[j].pos -= direction * (DROPRADIUS * 2 - dist) * 0.02f;	6,000,009	2,000,003	3.000	0.000	0.000	1.000	0.000
155	}							
156	}							
157	// simulation step 3b - satisfy constraints - evade walls							
158	if (drop[i].pos.y > 20) drop[i].pos.y = 19.99f - drop[i].pos.z * 0.000							
159	if (drop[i].pos.x < -20) drop[i].pos.x = -19.99f + drop[i].pos.z * 0.0							
160	if (drop[i].pos.x > 20) drop[i].pos.x = 19.99f - drop[i].pos.z * 0.000							
161	if (drop[i].pos.z < -20) drop[i].pos.z = -19.99f + drop[i].pos.z * 0.0							
162	if (drop[i].pos.z > 20) drop[i].pos.z = 19.99f - drop[i].pos.z * 0.000							
163	}							
164	}							
165	LEAVE(SIMULATE);							
166	}							



# Profiler Output

## 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.



```

...
    & (depth < MAXDEPTH)
...
    t = inside / (nc + nc2);
    nt = nt / nc;
    cos2t = 1.0f - nt;
    D, N );
    )
...
    at a = nt - nc, b = nt;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (a
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
...
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
survive = SurvivalProbability( diffuse,
estimation - doing it properly, close
if;
radiance = SampleLight( @rand, 1, &t, &lig
e.x + radiance.y + radiance.z) > 0) && (ref
...
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Pear
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiant
...
random walk - done properly, closely following
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



# 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.

```

...
    & (depth < MAXDEPTH)
...
    t = inside / (1.0f - nde);
    nt = nt / ncosdd;
    cos2t = 1.0f - nnt * nnt;
    D, N );
    )
...
    at a = nt - nc, b = nt - nc;
    at Tr = 1 - (R0 + (1 - R0) * R);
    Tr) R = (D * nnt - N * (ndc
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
...
    D, N );
    -refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
survive = SurvivalProbability( diffuse );
estimation - doing it properly, class
if;
radiance = SampleLight( &rand, I, &t, &align,
e.x + radiance.y + radiance.z) > 0) && (survive
...
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiant
...
random walk - done properly, closely following
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



# Custom Profiling

Minecraft Beta 1.9 Prerelease 2 (118 fps, 2 chunk updates) Used memory: 41% (412MB) of 989MB  
 C: 979/5408, F: 1912, O: 0, E: 2517 Allocated memory: 190% (989MB)  
 E: 1/251, B: 0, I: 259  
 P: 0, T: All: 251  
 ServerChunkCache: 979 Drop: 0

X: -12.898761435882818  
 Y: 79.620000000476897  
 Z: 231.78292536730885  
 F: 3

Seed: 5130996236305320162

level 82.58%  
 textures 11.06%  
 animateTick 5.61%  
 pick 0.31%  
 ??? 0.23%  
 gameRenderer 0.07%  
 keyboard 0.03%  
 gameMode 0.02%  
 stats 0.01%  
 particles 0.01%  
 mouse 0.0%  
 gui 0.0%  
 centerChunkSource 0.0%  
 levelRenderer 0.0%

Survive

Minecraft



# Custom Profiling

```

...
    & (depth < MAXDEPTH)
...
    inside / inside;
    nt = nt / nc;
    cos2t = 1.0f - nt;
    D, N );
    )
...
    at a = nt - nc; b = nt;
    at Tr = 1 - (R0 + (1 - R0)
    Tr) R = (D * nnt - N * (a
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
...
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
    survive = SurvivalProbability( diffuse );
    estimation - doing it properly, class
    if;
    radiance = SampleLight( &rand, I, Rt, Aligned
    e.x + radiance.y + radiance.z) > 0) && (refl +
...
    w = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Radiance;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) *
...
    random walk - done properly, closely following
    (survive)
...
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```

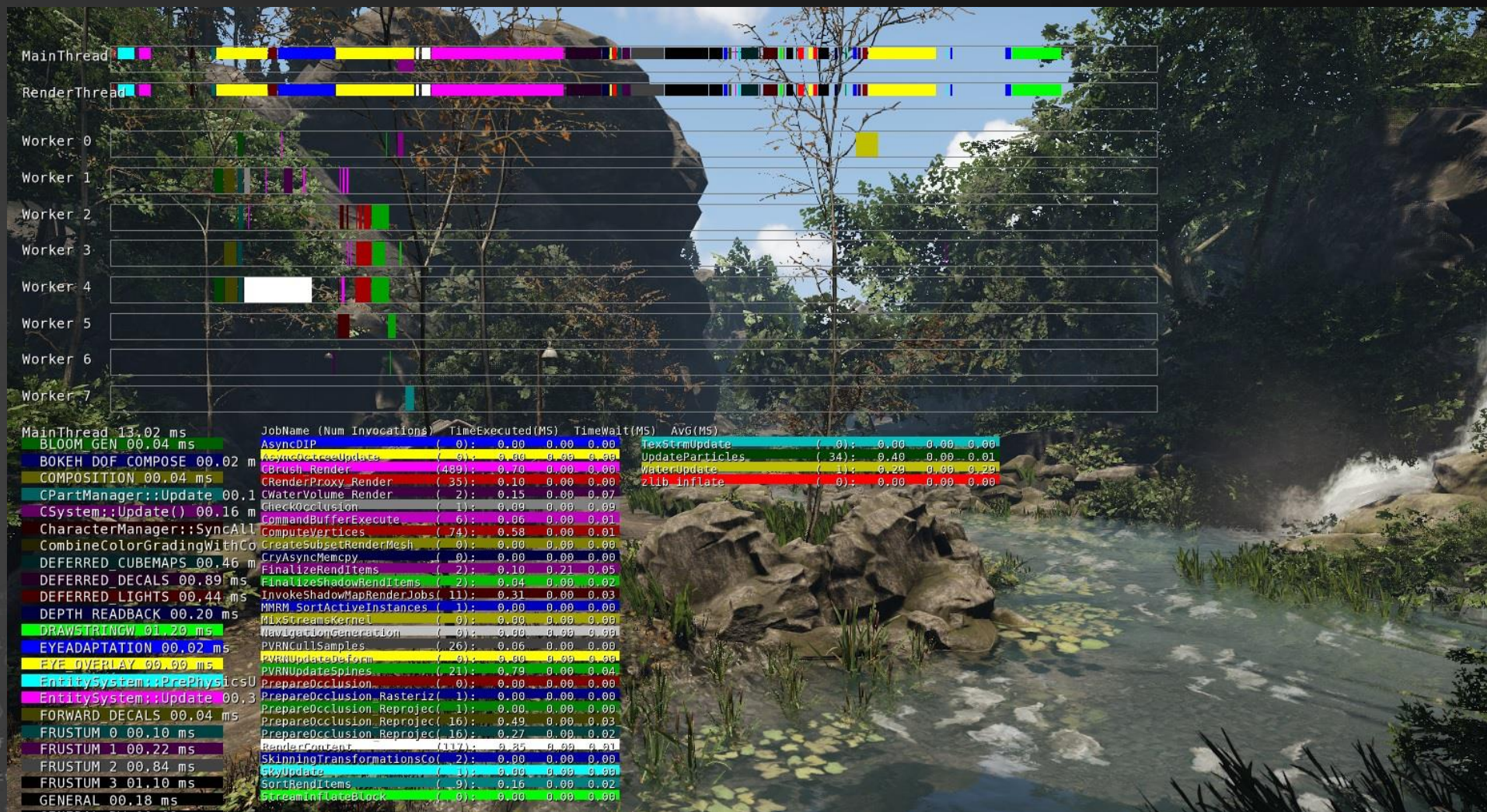
UnrealEngine 3

The screenshot shows the UnrealEngine 3 performance dashboard. At the top, it displays 'FPS: 54.7 (Avg over 20 frames)', 'Tris/Frame: 623134', 'Time: 601.9s', 'Speed: 111', 'Shaders Used (Modified)', and 'RS Used (Modified)'. The main view is a 3D scene with a 'Paused' overlay. Two charts are visible: a line chart for rendering statistics (VA, SHD, TEX, ROP) and a stacked area chart for pipeline stages (VS, GS, PS). Below these are several more charts: a bar chart for 'DrawPrim Count' and 'Avg Batch', a line chart for 'Driver Time (ms)', 'GPU Idle (ms)', 'Driver Sleeping (ms)', and 'Frame Time (ms)', and a vertical bar chart for 'ACP (MB)' and 'VID (MB)'. The bottom of the dashboard has tabs for 'Performance Dashboard', 'Debug Console', 'Frame Debugger', and 'Frame Profiler'. A large '+115' and '46' are visible in the bottom left and right corners respectively.

Copyright © 2007 Epic Games, Inc. Cary, N.C., USA. ALL RIGHTS RESERVED. Epic, Unreal, and Circle U logo are registered trademarks of Epic Games, Inc. in the United States of America and elsewhere.



# Custom Profiling



CryEngine



# Custom Profiling

```

...
    & (depth < MAXDEPTH)
...
    t = inside / (1.0f - r);
    nt = nt / nc;
    cos2t = 1.0f - nt * nt;
    D, N );
    )
...
    at a = nt - nc, b = nt * nc;
    at Tr = 1 - (R0 + (1 - R0) * t);
    Tr) R = (D * nnt - N * (a *
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
...
    D, N );
    -refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &t, &align, &
e.x + radiance.y + radiance.z) > 0) && (survive)
...
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Survive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



StarCraft II



# Custom Profiling

```

...
    & (depth < MAXDEPTH)
...
    c = inside / 1.5f;
    nt = nt / nc;
    cos2t = 1.0f - nt * nt;
    D, N );
}

...
    at a = nt - nc, b = nt + nc;
    at Tr = 1 - (RB + (1 - RB) *
    Tr) R = (D * nnt - N * (a
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth <
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)

survive = SurvivalProbability( diffuse,
estimation - doing it properly, class
if;
radiance = SampleLight( &rand, I, &t, Aligned
e.x + radiance.y + radiance.z) > 0) && (rand
v = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following
ive)
...
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```



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.



# 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.

```

...
    & (depth < MAXDEPTH)
...
    t = inside / (1.0f - r);
    nt = nt / nc;
    cos2t = 1.0f - nt;
    D, N );
...
    at a = nt - nc, b = nt;
    at Tr = 1 - (R0 + (1 - R0) * r);
    Tr) R = (D * nnt - N * (a *
...
    E * diffuse;
    = true;
...
    refl + refr) && (depth < MAXDEPTH)
...
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
survive = SurvivalProbability( diffuse, r);
estimation - doing it properly, class
if;
radiance = SampleLight( &rand, I, &t, &align,
e.x + radiance.y + radiance.z) && (radiance
...
    v = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Survival;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following
ive)
...
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
    survive;
    pdf;
    r1 = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```



# Preamble

Brief profiling session, digest:

- Latency & stalls: instructions causing bottlenecks on the next line
- Memory access patterns & caching
- The cost of a square root
- Hidden cost: normalize
- ‘Free calculations’:  $(\text{DROPRADIUS} * 2) * (\text{DROPRADIUS} * 2)$
- Optimization by the compiler: reordering, ...
- Optimization by the CPU: out-of-order execution, ...

```

...
    & (depth < MAXDEPTH)
...
    t = inside / (1.0f - diffuse);
    nt = nt / nc;
    cos2t = 1.0f - nt;
    D, N );
...
    at a = nt - nc, b = nt;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (a *
...
    E * diffuse;
    = true;
...
    refl + refr) && (depth < MAXDEPTH)
...
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
survive = SurvivalProbability( diffuse );
estimation - doing it properly, class
if;
radiance = SampleLight( &rand, I, &t, &align,
e.x + radiance.y + radiance.z) && (rand.N
...
    v = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following
ive)
...
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```



# Preamble

```

...ics
& (depth < MAXDEPTH)
...
c = inside / (1.0 - r);
nt = nt / nc; rdd = rdd / r;
cos2t = 1.0f - nnt * rdd;
D, N );
)
...
at a = nt - nc, b = nt * r;
at Tr = 1 - (R0 + (1 - R0) * r);
Tr) R = (D * nnt - N * (a * r + b));
...
E * diffuse;
= true;
...
efl + refr)) && (depth < MAXDEPTH)
D, N );
efl * E * diffuse;
= true;
...
MAXDEPTH)
survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, M, Alignment
e.x + radiance.y + radiance.z) > 0) && (depth <
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Survive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



**We need to go deeper**

*We need to go deeper*



# Today's Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement



```
ics
(depth < MAXDEPTH)
{
    c = inside / 1.5;
    nt = nt / nc;
    cos2t = 1.0f - nt * nt;
    D, N );
}

at a = nt - nc; b = nt + nc;
at Tr = 1 - (RB + (1 - RB) *
Tr) R = (D * nnt - N * (a *
E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)
D, N );
efl * E * diffuse;
= true;

MAXDEPTH)

survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, Rt, Alignment
e.x + radiance.y + radiance.z) > 0) && (depth <
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Survival;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
random walk - done properly, closely following
vive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

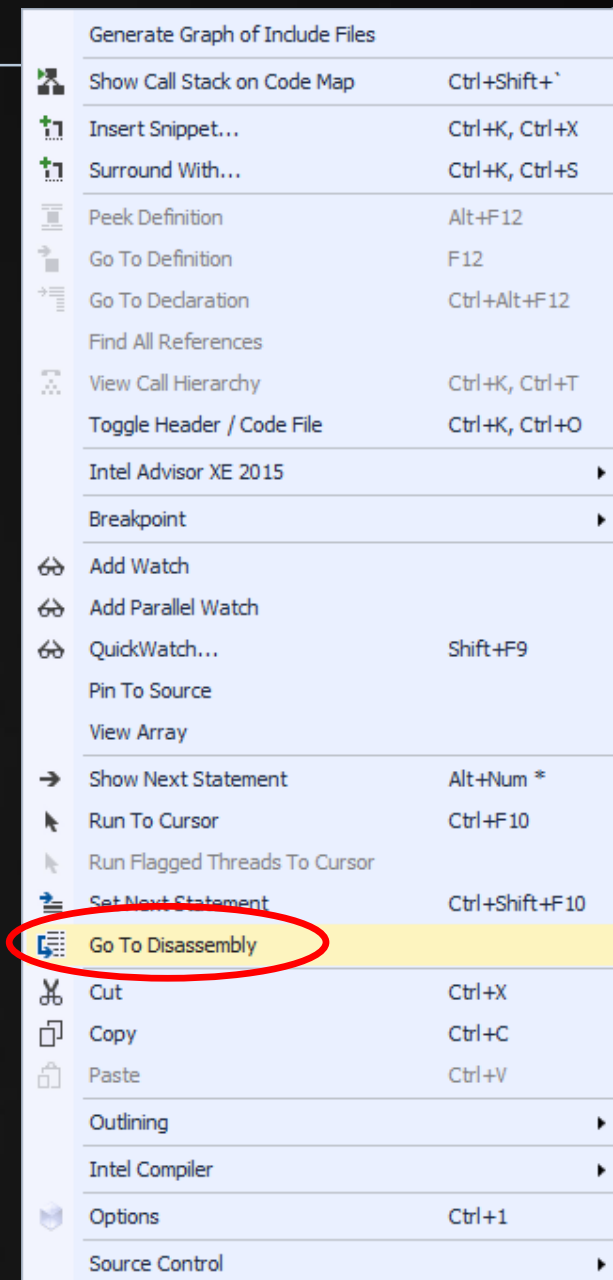




# Instruction Cost

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;
```



# Instruction Cost

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
		st0..st7	R8..R15
		XMM0..XMM7	

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



# Instruction Cost

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](https://www.cs.uaf.edu/2007/fall/cs301/lecture/11_12_floating_asm.html)



# Instruction Cost

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 = 2^46 / 28763 (!!)
```

```

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
    
```



# Instruction Cost

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.



# Instruction Cost

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).







# Instruction Cost

## Intel Silvermont 2014

	Operands	μops	Unit	Latency	Reciprocal throughput	Remarks
<b>Arithmetic instructions</b>						
ADD SUB	r,r/i	1	IP0/1	1	1/2	
ADD SUB	r,m	1	IP0/1, Mem		1	
ADD SUB	m,r/i	1	IP0/1, Mem	6	1	
ADC SBB	r,r/i	1	IP0/1	2	2	
ADC SBB	r,m	1			2	
ADC SBB	m,r/i	1		6	2	
CMP	r,r/i	1	IP0/1	1	1/2	
CMP	m,r/i	1			1	
INC DEC	r	1	IP0/1	1	1/2	latency to flag?
NEG NOT	r	1	IP0/1	1	1/2	
INC DEC	r	1	IP0/1	1	1/2	
<b>Math</b>						
	FSCALE			27		66
AAA	FXTRACT			15	20	20
AAS	FSQRT			1	13-40	13-40
DAA	FSIN FCOS			18	40-170	40-170
DAS	FSINCOS			110	40-170	
AAD	F2XM1			9	39-90	
AAM	FYL2X			34	80-140	
MUL IMUL	FYL2XP1			61	154	
MUL IMUL	FPTAN			101	45-200	
MUL IMUL	FPATAN			63	85-190	
IMUL	r16,r16	2	IP0	4	4	
IMUL	r32,r32	1	IP0	3	1	
IMUL	r64,r64	1	IP0	5	2	
IMUL	r16,r16,i	2	IP0	4	4	
IMUL	r32,r32,i	1	IP0	3	1	
IMUL	r64,r64,i	1	IP0	5	2	
MUL IMUL	m8	3	IP0			
MUL IMUL	m16	5	IP0			
MUL IMUL	m32	4	IP0			
MUL IMUL	m64	4	IP0	14		
DIV	r/m8	9	IP0, FP0	24	19	
DIV	r/m16	12	IP0, FP0	25-29	19-23	
DIV	r/m32	12	IP0, FP0	25-39	19-31	
DIV	r/m 64	23	IP0, FP0	34-94	25-94	
IDIV	r/m8	26	IP0, FP0	24-35	25	
IDIV	r/m16	29	IP0, FP0	37-41	30-32	
IDIV	r/m32	29	IP0, FP0	29-46	29-38	
IDIV	r/m64	44	IP0, FP0	47-107	47-107	

Note: This is a low-power processor (ATOM class).



# Instruction Cost

What is the ‘cost’ of a single instruction?

The cost of a single instruction depends on a number of factors:

- The arithmetic complexity (sqrt > add);
- Whether the operands are in register or memory;
- The size of the operand (16 / 64 bit is often slightly slower);
- Whether we need the answer immediately or not (latency);
- Whether we work on signed or unsigned integers (DIV/IDIV).

On top of that, certain instructions can be executed simultaneously.



# Today's Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement

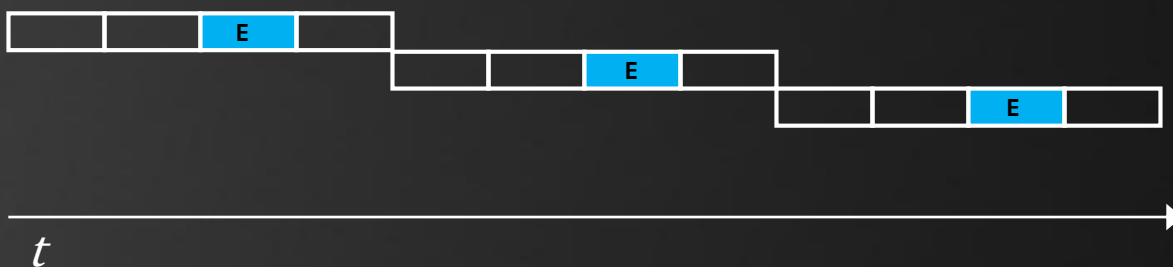


# Pipeline

## CPU Instruction Pipeline

Instruction execution is typically divided in four phases:

- |              |  |
|--------------|--|
| 1. Fetch     | Get the instruction from RAM             |
| 2. Decode    | The byte code is decoded                 |
| 3. Execute   | The instruction is executed              |
| 4. Writeback | The results are written to RAM/registers |



```

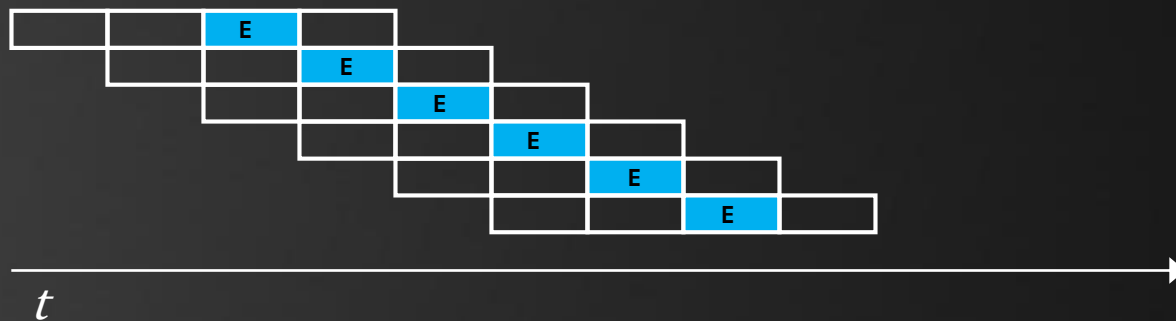
fldz
xor ecx, ecx
fld dword ptr ds:[405290h]
mov edx, 28929227h
fld dword ptr ds:[40528Ch]
push esi
mov esi, 0C350h
add ecx, edx
mov eax, 91D2A969h
xor edx, 17737352h
shr ecx, 1
mul eax, edx
fld st(1)
faddp st(3), st
mov eax, 91D2A969h
shr edx, 0Eh
add ecx, edx
fmul st(1),st
xor edx, 17737352h
shr ecx, 1
mul eax, edx
shr edx, 0Eh
dec esi
jne tobetimed<0>+1Fh
    
```



# Pipeline

## CPU Instruction Pipeline

For each of the stages, different parts of the CPU are active.  
 To use its transistors more efficiently, a modern processor overlaps these phases in a *pipeline*.



At the same clock speed, we get four times the throughput.

```

...
    & (depth < MAXDEPTH)
...
    c = inside / 10;
    nt = nt / nc;
    cos2t = 1.0f - nt;
    D, N );
...
    at a = nt - nc, b = nt;
    at Tr = 1 - (R0 + (1 - R0)
    Tr) R = (D * nnt - N * (1 -
...
    E * diffuse;
    = true;
...
    refl + refr) && (depth < MAXDEPTH)
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
survive = SurvivalProbability( diffuse,
estimation - doing it properly, class
if;
radiance = SampleLight( &rand, I, &t, &align;
e.x + radiance.y + radiance.z) > 0) && (survive)
...
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Peum;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf;
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
    
```



# Pipeline

## CPU Instruction Pipeline

Maximum clockspeed is determined by the most complex of the four stages. For higher clockspeeds, it is advantageous to increase the number of stages (thereby reducing the complexity of each individual stage).



### Stages

7	PowerPC G4e
8	Cortex-A9
10	Athlon
12	Pentium Pro/II/III, Athlon 64
14	Core 2, Apple A7/A8
14/19	Core i2/i3 Sandy Bridge
16	PowerPC G5, Core i*1 Nehalem
18	Bulldozer, Steamroller
20	Pentium 4
31	Pentium 4E Prescott

Obviously, ‘execution’ of different instructions requires different functionality.



# Pipeline

## CPU Instruction Pipeline

Different execution units for different (classes of) instructions:



Here, one execution unit handles floats;  
one handles integer;  
one handles memory operations.

Since the execution logic is typically the most complex part, we might just as well duplicate the other parts:



# Pipeline

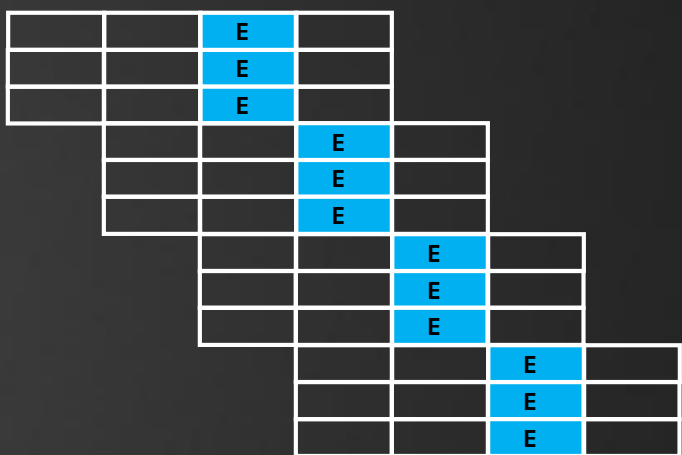
## CPU Instruction Pipeline

This leads to the superscalar processor, which can execute multiple instructions in the same clock cycle, assuming not all instructions require the same execution logic.

```

...
    & (depth < MAXDEPTH)
...
    c = inside / 1.5;
    nt = nt / nc;
    cos2t = 1.0f - nt * nt;
    D, N );
)
...
    at a = nt - nc, b = nt;
    at Tr = 1 - (R0 + (1 - R0)
    Tr) R = (D * nnt - N * (D0
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
    D, N );
    -refl * E * diffuse;
    = true;
...
MAXDEPTH)
survive = SurvivalProbability( diffuse,
estimation - doing it properly, class
if;
radiance = SampleLight( &rand, I, &t, Alignment
e.x + radiance.y + radiance.z) > 0) && (survive)
...
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

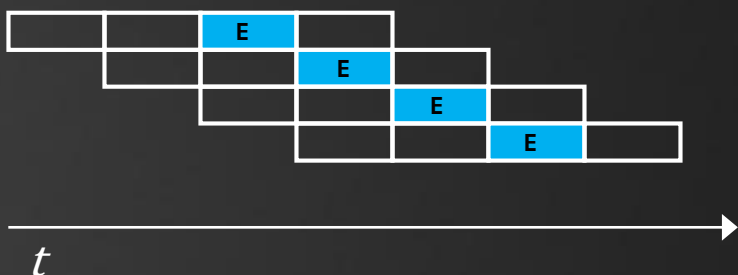
```



# Pipeline

## CPU Instruction Pipeline

Using a pipeline has consequences.  
Consider the following situation:



```
a = b * c;
d = a + 1;
```

Here, the second instruction needs the result of the first, which is available one clock tick too late. As a consequence, the pipeline stalls briefly.

```

...
    & (depth < MAXDEPTH)
...
    c = inside / 1.5;
    nt = nt / nc;
    cos2t = 1.0f - nt;
    D, N );
}
...
    at a = nt - nc, b = nt;
    at Tr = 1 - (R0 + (1 - R0)
    Tr) R = (D * nnt - N * (D0
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
survive = SurvivalProbability( diffuse,
estimation - doing it properly, class
if;
    radiance = SampleLight( &rand, 1, &t, Align
    e.x + radiance.y + radiance.z) > 0) && (refl
...
    v = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following
ive)
...
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

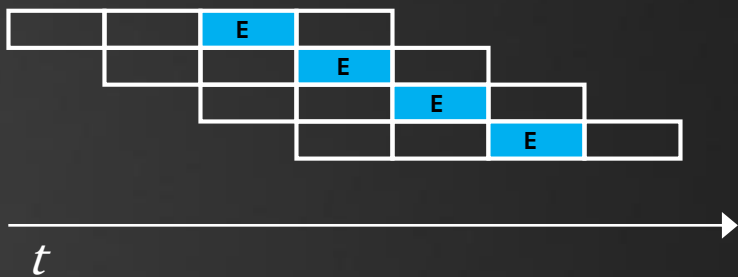
```



# Pipeline

## CPU Instruction Pipeline

Using a pipeline has consequences.  
Consider the following situation:



$a = b * c;$   
jump if a is not zero

In this scenario, a conditional jump makes it hard for the CPU to determine what to feed into the pipeline after the jump.

```

...
    & (depth < MAXDEPTH)
...
    c = inside / 1.5;
    nt = nt / nc;
    cos2t = 1.0f - nt * nt;
    D, N );
}
...
    at a = nt - nc, b = nt;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (D0
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
survive = SurvivalProbability( diffuse,
estimation - doing it properly, close
if;
    radiance = SampleLight( &rand, 1, &t, Allig
    e.x + radiance.y + radiance.z) > 0) && (sur
...
    v = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Pn
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiant
...
random walk - done properly, closely following
ive)
...
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```



# Pipeline

## CPU Instruction Pipeline - Digest

For a more elaborate explanation of the pipeline, see this document:

<http://www.lighterra.com/papers/modernmicroprocessors>

For now:

- A compiler reorganizes code to prevent latencies
- Feeding mixed code provides the compiler with sufficient opportunities for shuffling
- Branching issues need to be prevented manually



# Today's Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement



# Data Types

## Data types in C++

int  
unsigned int

Red = u4 & (255 << 16);  
Green = u4 & (255 << 8);  
Blue = u4 & 255;



Size: 32 bit (4 bytes)

Access:

union { unsigned int u4; int s4; char s[4]; };

unsigned char v = 100;

s[1] = v;

u4 = (a4 ^ (255 << 8)) | (v << 8);

u4 ^= 1 << 31;

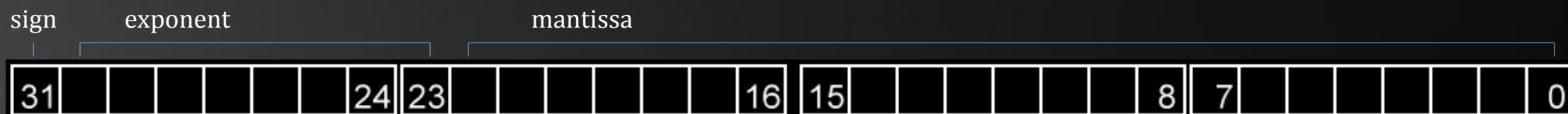
Altering sign bit of s4:  
(note: -1 = 0xffffffff)



# Data Types

## Data types in C++

### float



Size: 32 bit (4 bytes)

Exponent: 8 bit; -127 ... 128

Mantissa: 23 bit; 0 ...  $2^{23} - 1$

Value:  $\text{sign} * \text{mantissa} * 2^{\text{exponent}}$



# Data Types

## Data types in C++

```

...
    & (depth < MAXDEPTH)
...
    t = inside / 1000000;
    nt = nt / nc;
    cos2t = 1.0f - nt;
    D, N );
    )
...
    at a = nt - nc, b = nt;
    at Tr = 1 - (R0 + (1 - R0)
    Tr) R = (D * nnt - N * (a
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
...
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
survive = SurvivalProbability( diffuse,
estimation - doing it properly, class
if;
radiance = SampleLight( &rand, I, &t, &align
e.x + radiance.y + radiance.z) > 0) && (survive
...
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Pdf;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) *
...
random walk - done properly, closely following
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```

double	64 bit (8 bytes)
char, unsigned char	8 bit
short, unsigned short	16 bit
LONG	32 bit (same as int)
LONG LONG, __int64	64 bit
bool	8 bit (!)

## Padding:

```

struct Test
{
    unsigned int u;
    bool flag;
};
sizeof( Test ) is 8

```

```

struct Test2
{
    double d;
    bool flag;
};
sizeof( Test2 ) is 16

```



# Data Types

## Data types in C++ - Conversions

### Explicit:

```
float fpi = 3.141593;
int pi = (int)(1024.0f * fpi);
```

### Implicit:

```
struct Color { unsigned char a, r, g, b; };
Color bitmap[640 * 480];
for( int i = 0; i < 640 * 480; i++ )
{
    bitmap[i].r *= 0.5f;
    bitmap[i].g *= 0.5f;
    bitmap[i].b *= 0.5f;
}
```

```
// bitmap[i].r *= 0.5f;
movzx    eax,byte ptr [ecx-1]
mov      dword ptr [ebp-4],eax
fild     dword ptr [ebp-4]
fstcw    word ptr [ebp-2]
movzx    eax,word ptr [ebp-2]
or       eax,0C00h
mov      dword ptr [ebp-8],eax
fmul     st,st(1)
fldcw    word ptr [ebp-8]
fistp    dword ptr [ebp-8]
movzx    eax,byte ptr [ebp-8]
mov      byte ptr [ecx-1],al
```



# Data Types

## Data types in C++ - Conversions

Explicit:

```
float fpi = 3.141593;
int pi = (int)(1024.0f * fpi);
```

Avoiding conversion:

```
struct Color { unsigned char a, r, g, b; };
Color bitmap[640 * 480];
for( int i = 0; i < 640 * 480; i++ )
```

```
{
    bitmap[i].r >>= 1;
    bitmap[i].g >>= 1;
    bitmap[i].b >>= 1;
}
```

```
// bitmap[i].r >>= 1;
shr          byte ptr [eax-1],1
// bitmap[i].g >>= 1;
shr          byte ptr [eax],1
// bitmap[i].b >>= 1;
shr          byte ptr [eax+1],1
```



# Data Types

## Data types in C++ - Conversions

Explicit:

```
float fpi = 3.141593;
int pi = (int)(1024.0f * fpi);
```

Avoiding conversion (2):

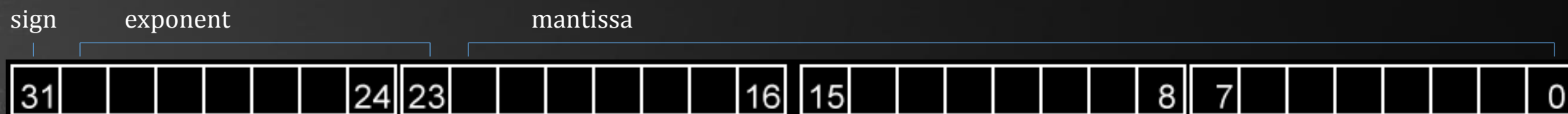
```
struct Color { union { struct { unsigned char a, r, g, b; }; int argb; }; };
Color bitmap[640 * 480];
for( int i = 0; i < 640 * 480; i++ )
{
    bitmap[i].argb = (bitmap[i].argb >> 1) & 0x7f7f7f;
}
```



# Data Types

Data types in C++ - Free interpretation

Trick: Cheaper float comparison



```
union { float v1; unsigned int u1; };
```

```
union { float v2; unsigned int u2; };
```

```
bool smaller = (v1 < v2);
```

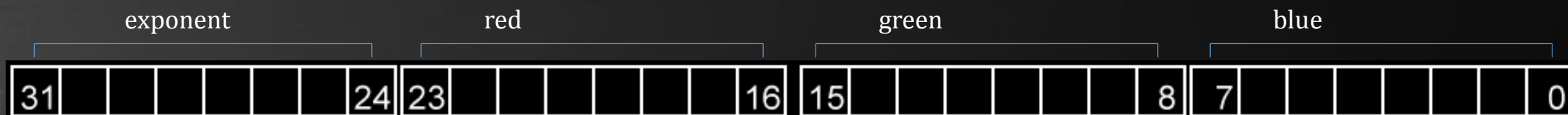
```
bool smaller = (u1 < u2); // same result, if signs of v1 and v2 are equal.
```



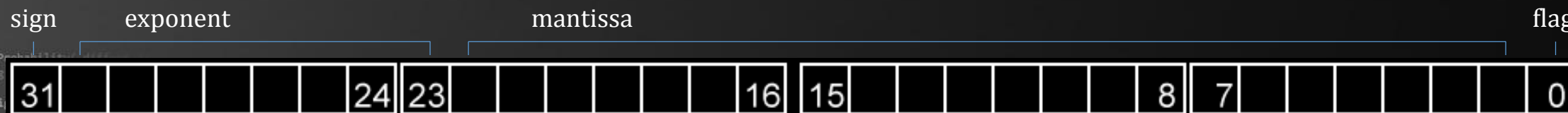
# Data Types

Data types in C++ - Rolling your own

HDR color storage



Storing a bit flag in a floating point value



# Today's Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement





# Rules of Engagement

## Common Opportunities in Low-level Optimization

### RULE 2: Precalculate

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

```

...
    & (depth < MAXDEPTH)
...
    t = inside / (1.0f - r);
    nt = nt / nc;
    cos2t = 1.0f - nt * nt;
    D, N );
    )
...
    at a = nt - nc, b = nt * nc;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (a *
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
...
    D, N );
    -refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
survive = SurvivalProbability( diffuse );
estimation - doing it properly, classically
if;
radiance = SampleLight( &rand, I, &t, &align,
e.x + radiance.y + radiance.z) > 0) && (survive)
...
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * survive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following Monte Carlo
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



# Rules of Engagement

## 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

```

...
    & (depth < MAXDEPTH)
...
    t = inside / (1.0f - ncr);
    nt = nt / ncr;
    cos2t = 1.0f - nnt;
    D, N );
...
    at a = nt - ncr, b = ncr;
    at Tr = 1 - (R0 + (1 - R0) * nnt);
    R = (D * nnt - N * (1 - nnt));
...
    E * diffuse;
    = true;
...
    refl + refr) && (depth < MAXDEPTH)
...
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
survive = SurvivalProbability( diffuse );
estimation - doing it properly, classically
if;
radiance = SampleLight( @rand, I, Rt, Alignment);
e.x + radiance.y + radiance.z) > 0) && (depth <
...
    v = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Survive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following the
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



# Rules of Engagement

## 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

```

...
    & (depth < MAXDEPTH)
...
    t = inside / (1.0f - r);
    nt = nt / nc;
    cos2t = 1.0f - nt;
    D, N );
    );
...
    at a = nt - nc, b = nt;
    at Tr = 1 - (R0 + (1 - R0));
    Tr) R = (D * nnt - N * (1 -
...
    E * diffuse;
    = true;
...
    refl + refr) && (depth < MAXDEPTH)
    D, N );
    -refl * E * diffuse;
    = true;
...
MAXDEPTH)
survive = SurvivalProbability( diffuse );
estimation - doing it properly, class
if;
radiance = SampleLight( &rand, I, &t, &align,
e.x + radiance.y + radiance.z) > 0) && (depth <
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Survive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
urvive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



# Rules of Engagement

## Common Opportunities in Low-level Optimization

### RULE 5: Early Out

```

...
    & (depth < MAXDEPTH)
...
    t = inside / inside;
    nt = nt / nc;
    cos2t = 1.0f - nt * nt;
    D, N );
    )
...
    at a = nt - nc, b = nt;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (nc
...
    E * diffuse;
    = true;
...
    efl + refr) && (depth < MAXDEPTH)
    {
    D, N );
    -efl * E * diffuse;
    = true;
...
    MAXDEPTH)
    survive = SurvivalProbability( diffuse,
    estimation - doing it properly, class
    if;
    radiance = SampleLight( &rand, I, } Alignment
    e.x + radiance.y + radiance.z) > 0) && (rand
...
    v = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
...
    random walk - done properly, closely following
    vive)
...
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```

```

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

```

```

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

```



# Rules of Engagement

## 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.

```

...
    & (depth < MAXDEPTH)
...
    inside / inside;
    nt = nt / nc;
    cos2t = 1.0f - nt;
    D, N );
    )
...
    at a = nt - nc, b = nt;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (1 -
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
...
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
    survive = SurvivalProbability( diffuse,
    estimation - doing it properly, clean
    if;
    radiance = SampleLight( &rand, 1, &t, &light
    e.x + radiance.y + radiance.z) > 0) && (refl
...
    v = true;
    at brdfPdf = EvaluateDiffuse( L, N ) * Radiance;
    at3 factor = diffuse * INVPI;
    at weight = Mis2( directPdf, brdfPdf );
    at cosThetaOut = dot( N, L );
    E * ((weight * cosThetaOut) / directPdf) * (radiance
...
    random walk - done properly, closely following the
    (vive)
...
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
    
```



# Rules of Engagement

## 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.

```

...
    & (depth < MAXDEPTH)
...
    t = inside / (1.0f - r);
    nt = nt / nc;
    cos2t = 1.0f - nt * nt;
    D, N );
    )
...
    at a = nt - nc, b = nt * nc;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (a *
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
    D, N );
    -refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
survive = SurvivalProbability( diffuse );
estimation - doing it properly, classically
if;
radiance = SampleLight( @rand, I, @t, @align, @
e.x + radiance.y + radiance.z) > 0) && (survive)
...
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Survive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following Monte Carlo
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, @R, @pdf
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



# Rules of Engagement

## 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

```

...
    & (depth < MAXDEPTH)
...
    t = inside / (inside + outside);
    nt = nt / nc;
    cos2t = 1.0f - nt * nt;
    D, N );
    )
...
    at a = nt - nc, b = nt * nc;
    at Tr = 1 - (R0 + (1 - R0) * a);
    Tr) R = (D * nnt - N * (a *
...
    E * diffuse;
    = true;
...
    refl + refr) && (depth < MAXDEPTH)
    D, N );
    -refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
survive = SurvivalProbability( diffuse );
estimation - doing it properly, class
if;
radiance = SampleLight( &rand, I, &t, &align,
e.x + radiance.y + radiance.z) > 0) && (survive)
...
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Survive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance
...
random walk - done properly, closely following
ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



# Today's Agenda:

- The Cost of a Line of Code
- CPU Architecture: Instruction Pipeline
- Data Types and Their Cost
- Rules of Engagement



# Practice

Get (from the website) project glassball.zip

Using low-level optimization, speed up this application.

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

Make sure functionality remains intact.

Target: a 10x speedup (this should be easy).



