rics & (depth < Modes

: = inside ? 1 + 1,0 ht = nt / nc, ddn bs2t = 1.0f - nnt D, N); D)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 Tr) R = (D = nnt - N = (dd

= * diffuse = true;

efl + refr)) && (depth < MAXDEPIN

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse)
estimation - doing it properly, close |
f;

radiance = SampleLight(&rand, I, &L, &ll e.x + radiance.y + radiance.z) > 0) && (dl.)

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) (real

andom walk - done properly, closely following Small /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 2019 - Lecture 13: "Snippets"

Welcome!



ics (depth < Modean

: = inside ? 1 + . . ht = nt / nc, ddn os2t = 1.0f - nnt 2, N); 3)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - Rc Fr) R = (D = nnt - N = (dd)

= * diffuse = true;

efl + refr)) && (depth < MAXDEDIII

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly closed if; radiance = SampleLight(&rand, I, &L, &Light) e.x + radiance.y + radiance.z) > 0) && (dobb

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) * (rad

andom walk - done properly, closely following Sec. /ive)

; t3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); Sion = true:

Today's Agenda:

- Self-modifying code
- Multi-threading (1)
- Multi-threading (2)
- Experiments





Self-modifying

ics ⊾(depth < NOCS

:= inside 7 1 1 1 0 bt = nt / nc, ddn 4 0s2t = 1.0f - nnt 4 0, N); >)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 Fr) R = (D = nnt - N = (1

= * diffuse; = true;

efl + refr)) && (depth < MAXDEPTH

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuent estimation - doing it properly if; radiance = SampleLight(&rand, I, &L, & 2.x + radiance.y + radiance.z) > 0) &&

w = true; at brdfPdf = EvaluateDiffuse(L, N) * Ps at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely following Sov /ive)

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

....

Fast Polygons on Limited Hardware

Typical span rendering code:

for(int i = 0; i < len; i++)
{
 *a++ = texture[u,v];
 u += du;
 v += dv;</pre>

How do we make this faster? Every cycle counts...

<u>Loop unrolling</u> Two pixels at a time





Self-modifying

Fast Polygons on Limited Hardware

), N); refl * E * diffuse;

AXDEPTH)

survive = SurvivalProbability(diffu radiance = SampleLight(&rand, I, &L e.x + radiance.y + radiance.z) > 0)

v = true;

at brdfPdf = EvaluateDiffuse(L, N) at3 factor = diffuse * INVPI;

at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L);

E * ((weight * cosThetaOut) / directPdf) *

/ive)

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

How about...

switch (len)

case	8:	*a++	=	<pre>tex[u,v];</pre>	u+=du;	v+=dv;
case	7:	*a++	=	<pre>tex[u,v];</pre>	u+=du;	v+=dv;
case	6:	*a++	=	<pre>tex[u,v];</pre>	u+=du;	v+=dv;
case	5:	*a++	=	<pre>tex[u,v];</pre>	u+=du;	v+=dv;
case	4:	*a++	=	<pre>tex[u,v];</pre>	u+=du;	v+=dv;
case	3:	*a++	=	<pre>tex[u,v];</pre>	u+=du;	v+=dv;
case	2:	*a++	=	<pre>tex[u,v];</pre>	u+=du;	v+=dv;
case	1:	*a++	=	<pre>tex[u,v];</pre>	u+=du;	v+=dv;





Self-modifying

at a = nt

), N);

= true;

(AXDEPTH)

v = true;

/ive)

if;

efl + refr)) && (dept)

survive = SurvivalProbability(diff

radiance = SampleLight(&rand, I, 8 e.x + radiance.y + radiance.z) > 0)

at brdfPdf = EvaluateDiffuse(L, N) * i
at3 factor = diffuse * INVPI;
at weight = Mis2(directPdf, brdfPdf);
at cosThetaOut = dot(N, L);
E * ((weight * cosThetaOut) / directPdf

andom walk - done properly, closely foll

refl * E * diffuse;

Fast Polygons on Limited Hardware

What if a massive unroll isn't an option, but we have only 4 registers?

```
for( int i = 0; i < len; i++ )
{
    *a++ = texture[u,v];</pre>
```

```
u += du, v += dv;
```

Registers: { i, a, u, v, du, dv, len }.

Idea: just before entering the loop,

replace 'len' by the correct constant *in the code*;
replace du and dv by the correct constant.

Our code is now *self-modifying*.

, H33 brdf = SampleDiffuse(diffuse, N, r1, r2, SR, Spdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:



Self-modifying

Self-modifying Code

Good reasons for **not** writing SMC:

- the CPU pipeline (mind every potential (future) target)
- L1 instruction cache (handles reads only)
- code readability

Good reasons for writing SMC:

- code readability
 - genetic code optimization





6

w = true; at brdfPdf = EvaluateDiffuse(L, N) * Psui at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

survive = SurvivalProbability(diff

radiance = SampleLight(&rand, I, &L e.x + radiance.y + radiance.z) > 0) &

at a = nt

), N);

(AXDEPTH)

f:

refl * E * diffuse;

andom walk - done properly, closely following Sec. /ive)

; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, 8pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Self-modifying

Hardware Evolution*

Experiment:

- take 100 FPGA's, load them with random 'programs', max 100 logic gates
 - test each chip's ability to differentiate between two audio tones
 - use the best candidates to produce the next generation.

Outcome (generation 4000): one chip capable of the intended task.

survive = SurvivalProbability(diffuse)
estimation - doing it properly, closed
ff;
radiance = SampleLight(&rand, I, &L, &l
e.x + radiance.y + radiance.z) > 0) && ()

= true:

), N);

= true;

AXDEPTH)

refl * E * diffuse;

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

3.

andom walk - done properly, closely followi /ive)

; t3 brdf = SampleDiffuse(diffuse, N, r1, urvive; pdf; n = E * brdf * (dot(N, R) / pdf); view true; Observations:

- The chip used only 37 logic gates, of which 5 disconnected from the rest. The 5 disconnected gates where vital to the function of the chip.
- The program could not be transferred to another chip.

*: On the Origin of Circuits, Alan Bellows, 2007, <u>https://www.damninteresting.com/on-the-origin-of-circuits</u> **: Evolved antenna, Wikipedia.



NASA's evolved antenna**



Compiler Flags*

Experiment:

Self-modifying

ics & (depth < Mo⊙s

: = inside 7 1 ; ... ht = nt / nc, ddn os2t = 1.0f - nnt ? 0, N); 3)

nt a = nt - nc, b = nt nt Tr = 1 - (R0 + (1 - 60 ir) R = (D ⁼ nnt - N - (d)

= * diffuse = true;

• :fl + refr)) && (depth < MAXDEPID

), N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly, closed H; radiance = SampleLight(&rand, I, &L, slight e.x + radiance.y + radiance.z) > 0) && cool

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) (1000)

andom walk - done properly, closely following Soli /ive)

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

execute, as the fitness function."

"...we propose a genetic algorithm to determine the combination of

flags, that could be used, to generate efficient executable in terms

of time. The input population to the genetic algorithm is the set of

compiler flags that can be used to compile a program and the best

derived over generations, based on the time taken to compile and

chromosome corresponding to the **best combination of flags** is





Self-modifying

Compiler Flags*



AXDEPTH)

survive = Surv estimation -Hf; radiance = Sam

e.x + radiance.y + radiance.z) > 0) 88 (0)

w = true; at brdfPdf = EvaluateDiffuse(L, N) = 1 at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L);

E * ((weight * cosThetaOut) / directPdf) * (rad

andom walk - done properly, closely following Scil. /ive)

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





ics (depth < Modean

: = inside ? 1 + . . ht = nt / nc, ddn os2t = 1.0f - nnt 2, N); 3)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - Rc Fr) R = (D = nnt - N = (dd)

= * diffuse = true;

efl + refr)) && (depth < MAXDEDIII

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly closed if; radiance = SampleLight(&rand, I, &L, &Light) e.x + radiance.y + radiance.z) > 0) && (dobb

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) * (rad

andom walk - done properly, closely following Sec. /ive)

; t3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); Sion = true:

Today's Agenda:

- Self-modifying code
- Multi-threading (1)
- Multi-threading (2)
- Experiments





Multi-threading

at Tr = 1

), N); refl * E * diffuse; = true;

(AXDEPTH)

survive = SurvivalProbability(diff. f: radiance = SampleLight(&rand, I, &L e.x + radiance.y + radiance.z) > 0)

v = true; at brdfPdf = EvaluateDiffuse(L, N) at3 factor = diffuse * INVPI at weight = Mis2(directPdf, brdfPdf) at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely follow /ive)

at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, & urvive; pdf; 1 = E * brdf * (dot(N, R) / pdf); sion = tru

A Brief History of Many Cores

Once upon a time...

Then, in 2005: Intel's Core 2 Duo (April 22). (Also 2005: AMD Athlon 64 X2. April 21.)

2007: Intel Core 2 Quad

2010: AMD Phenom II X6







Multi-threading

tics & (depth < Mo⊙S

t = inside ? | ; | ; ht = nt / nc, ddh → 552t = 1.0f - nnt → 2, N); 3)

at a = nt - nc, b = 00 at Tr = 1 - (R0 + (1 - 00 Fr) R = (D = nnt - 0

= * diffuse; = true;

efl + refr)) && (depth < MAXDEPIN

), N); refl * E * diffuse; = true;

(AXDEPTH)

survive = SurvivalProbability(different estimation - doing it properly, closed df; radiance = SampleLight(&rand, I, &L, e.x + radiance.y + radiance.z) > 0) & w = true; at brdfPdf = EvaluateDiffuse(L, N) at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / direct

andom walk - done properly, closely fo /ive)

; t33 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true;

A Brief History of Many Cores

Once upon a time...

Then, in 2005: Intel's Core 2 Duo *(April 22). (Also 2005: AMD Athlon 64 X2. April 21.)*

2007: Intel Core 2 Quad

2010: AMD Phenom II X6

Today...





Multi-threading

A Brief History of Many Cores

Once upon a time...

at Tr = 1 - (R0

), N); refl * E * diffuse;

AXDEPTH)

survive = SurvivalProbability(di radiance = SampleLight(&rand, e.x + radiance.y + radiance.z) >

v = true: at brdfPdf = EvaluateDiffuse(at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely follo /ive)

at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, A urvive; pdf; 1 = E * brdf * (dot(N, R) / pdf); sion = tru

Then, in 2005: Intel's Core 2 Duo *(April 22)*. (Also 2005: AMD Athlon 64 X2. April 21.)

2007: Intel Core 2 Quad

2010: AMD Phenom II X6

2017: Threadripper 1950X (16 cores, 32 threads) 2018: Threadripper 2950X 2019: Epyc 7742, 64 cores, 128 threads (\$6,950)





Multi-threading

Threads / Scalability



sion = true:

Multi-threading

Optimizing for Multiple Cores

What we did before:

Profile.

Goal:

- Understand the hardware.
- 3.

-), N); refl * E * diffuse; = true;

at a = nt - nc,

- (AXDEPTH)
- survive = SurvivalProbability(diff. if; radiance = SampleLight(&rand, I, & e.x + radiance.y + radiance.z) > 0)
- v = true; at brdfPdf = EvaluateDiffuse(L, N) at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf) at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely follo /ive)

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

1.33 2 core 2 sequential core 1 core 2 1.604 core 3

Trust No One.

- It's fast enough when it scales linearly with the number of cores.
 - It's fast enough when the parallelizable code scales linearly with the number of cores.
- It's fast enough if there is no sequential code.





Multi-threading

Hardware Review

- at a = nt

-), N); refl * E * diffuse; = true;

(AXDEPTH)

- survive = SurvivalProbability(diff f: radiance = SampleLight(&rand, I e.x + radiance.y + radiance.z) >
- v = true: at brdfPdf = EvaluateDiffuse(L, at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)
- andom walk done properly, closely f /ive)
- at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, & urvive; pdf; 1 = E * brdf * (dot(N, R) / pdf); sion = true:

We have:

- Four physical cores
- Each running two threads
- L1 cache: 32Kb, 4 cycles latency
- L2 cache: 256Kb, 10 cycles latency
- A large shared L3 cache.

Observation:

If our code solely requires data from L1 and L2, this processor should do work split over four threads exactly four times faster.

(Is that true? Any conditions?)



- Work must stay on core
- No I/O, sleep



Multi-threading

ics ≰j(depth < NotDa

: = inside ? 1 () 0 nt = nt / nc, ddn () ps2t = 1.0f - nnt () p, N); ∂)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - Rc Fr) R = (D = nnt - N = (dd)

= * diffuse; = true;

. :fl + refr)) && (depth < MAX

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse
estimation - doing it properly, closed
if;
radiance = SampleLight(&rand, I, &L, &light()
e.x + radiance.y + radiance.z) > 0) &&

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) (cost)

andom walk - done properly, closely following Source /ive)

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

Simultaneous Multi-Threading (SMT)

(Also known as hyperthreading)

Pipelines grow wider and deeper:

- Wider: to execute multiple instructions in parallel in a single cycle.
- Deeper: to reduce the complexity of each pipeline stage, which allows for a higher frequency.

	E				
	E				
	Е				
		E			
		E			
		Е			
			E		
			Е		
			Е		
				E	
				E	
				E	



Multi-threading

Superscalar Pipeline

nics ≹ (depth < Modes

: = inside ? | ht = nt / nc, ddn bs2t = 1.0f - nnt 2, N); 2)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0) Fr) R = (D = nnt - N = (300)

= * diffuse; = true;

.

), N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly if; radiance = SampleLight(&rand, I, &L, &lig 2.x + radiance.y + radiance.z) > 0) && ()

v = true;

at brdfPdf = EvaluateDiffuse(L, N) * Ps at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely followin ${\cal L}$. /ive)

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

	E				
	E				
	E		b ru		
		E			
		E			
		E		e T.ª.	
			E		
			E		
			E		
				Е	
				E	
				Е	

fldz

xor ecx, ecx fld dword ptr [4520h] mov edx, 28929227h fld dword ptr [452Ch] 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, OEh dec esi jne tobetimed+1Fh



Multi-threading

Superscalar Pipeline

Nehalem (i7): *six* wide.

Three memory operations

Three calculations (float, int, vector)

ics (depth < Model

: = inside } 1 ht = nt / nc, ddn bs2t = 1.0f - nnt D, N); 3)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - RC) Fr) R = (D = nnt - N = (dom

= * diffuse; = true;

e**fl + refr)) && (depth < MAXE**

D, N); refl * E * diffuse; = true;

AXDEPTH)

pdf;

sion = true:

1 = E * brdf * (dot(N, R) / pdf);

execution unit 1 MEM			
survive = SurvivalProbability difference of the survive = SurvivalProbability difference of the survival s			
H; radiance = SampleLight(Brand execution unit 3 MEM			
e.x + radiance.y + radiance.zexecution unit 4 CALC			
execution unit 5 CALC			
at brdfPdf = EvaluateDiffuse (Later and the second s			
nt weight = Mis2(directPdf, brdfPdf); nt cosThetaOut = dot(N, L);			
E * ((weight * cosThetaOut) / directPdf)			
andom walk - done properly, closely following ${\cal L}$ with ${\cal L}$			
; st3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, Bpdf			

fldz xor ecx, ecx fld dword ptr [4520h] mov edx, 28929227h fld dword ptr [452Ch] push esi mov esi, [0C350h] add ecx, edx mov eax, [91D2h] 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, OEh dec esi jne tobetimed+1Fh



Multi-threading

Simultaneous Multi-Threading (SMT)

(Also known as hyperthreading)

Pipelines grow wider and deeper:

- Wider, to execute multiple instructions in parallel in a single cycle.
- Deeper, to reduce the complexity of each pipeline stage, which allows for a higher frequency.

However, parallel instructions must be independent, otherwise we get bubbles.

Observation: *two threads provide twice as many independent instructions.*

(Is that true? Any conditions?)

	E				
	E				
		E			
		E			
		E			
	The second se		E		
			E		
			E		
				E	
				E	
				E	
-					

t

No dependencies between the threads



andom walk - done properly, closely followin /ive) ; ; tt3 brdf = SampleDiffuse(diffuse, N, r1, r2 urvive; pdf;

E * ((weight * cosThetaOut) / directPdf)

par; n = E * brdf * (dot(N, R) / pdf); sion = true:

at a = nt at Tr = 1 -

), N);

AXDEPTH)

v = true:

refl * E * diffuse;

survive = SurvivalProbability()

radiance = SampleLight(&rand, I e.x + radiance.y + radiance.z) >

at brdfPdf = EvaluateDiffuse(L, N at3 factor = diffuse * INVPI;

at cosThetaOut = dot(N, L);

at weight = Mis2(directPdf, brdfPdf

Multi-threading

= true:

at3 brdf = SampleDiffuse(diffuse, N, r1,

1 = E * brdf * (dot(N, R) / pdf);

), N); refl * E * diffuse; = true;

urvive; pdf;

AXDEPTH)							
, puivo - SupuivolDochobilit	execution unit 1	l MEM	fld	mov			
estimation - doing it proper	execution unit 2	2 MEM	mov	mov			
F; adiance = SampleLight(&ran	execution unit 3	3 MEM	fld				
.x + radiance.y + radiance.:	execution unit 4	4 CALC	fldz	add	xor	mul	
= true;	execution unit 5	5 CALC	xor	fld	shr	fmul	
<pre>brdTPdf = EvaluateDiffuse 3 factor = diffuse * INVPI; t weight = Mis2(directPdf, t cosThetaOut = dot(N, L)</pre>	execution unit 6	5 CALC	push	faddp			
* ((weight * cosThetaOut)	/ directPdf)		E di L				
idom walk - done properly, (ive)	losely following \mathcal{L}						

fldz	f
xor ecx, ecx	f
fld dword ptr [4520h]	m
mov edx, 28929227h	s
fld dword ptr [452Ch]	а
push esi	f
mov esi, 0C350h	X
add ecx, edx	S
mov eax, [91D2h]	m
xor edx, 17737352h	s
shr ecx, 1	d
mul eax, edx	f
fld st(1)	Х
faddp st(3), st	f
mov eax, 91D2A969h	m
shr edx, 0Eh	f
add ecx, edx	р
fmul st(1),st	m
xor edx, 17737352h	а
shr ecx, 1	m
mul eax, edx	Х
shr edx, 0Eh	s
dec esi	m

jne tobetimed+1Fh

d st(1)addp st(3), st lov eax, 91D2A969h shr edx, OEh add ecx, edx ⁻mul st(1),st or edx, 17737352h shr ecx, 1 nul eax, edx shr edx, OEh lec esi ldz kor ecx, ecx ⁻ld dword ptr [4520h] 10v edx, 28929227h ld dword ptr [452Ch] bush esi 10v esi, 0C350h dd ecx, edx lov eax, [91D2h] or edx, 17737<u>352</u>h shr ecx, 1 ul eax, edx jne tobetimed

*: Details: The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms, Thomadakis, 2011.

Three calculations (float, int, vector)

Simultaneous Multi-Threading (SMT)

Nehalem (i7) pipeline: *six* wide*.

Three memory operations

SMT: feeding the pipe from *two* threads.

All it really takes is an extra set of registers.

22

Multi-threading

rics & (depth < Motos

: = inside ? 1 ht = nt / nc, ddn ... ps2t = 1.0f - nnt ... D, N); B)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 Fr) R = (D ⁺ nnt - N = (ddn)

= * diffuse; = true;

. efl + refr)) && (depth < M

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly, df; radiance = SampleLight(&rand, I, &L, & e.x + radiance.y + radiance.z) > 0) &&

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

andom walk - done properly, closely f /ive)

```
,
t33 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, dpdf )
urvive;
.pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true:
```

Simultaneous Multi-Threading (SMT)

Hyperthreading does mean that now *two* threads are using the same L1 and L2 cache.



I-\$

- For the average case, this will reduce data locality.
- If both threads use the same data, data locality remains the same.
 - One thread can also be used to fetch data that the other thread will need *.

*: Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors, Luk, 2001.



Multi-threading

nics & (depth < Motocr

: = inside ? 1 : 1 : 1 ht = nt / nc, ddn bs2t = 1.0f - n∩t 2, N); 3)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0) Fr) R = (D [#] nnt - N ⁻ (dd)

= * diffuse; = true;

efl + refr)) && (depth < MAXDEPIN

), N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse
estimation - doing it properly, closed
if;
radiance = SampleLight(&rand, I, &L, &light
e.x + radiance.y + radiance.z) > 0) && (dot

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) (red);

andom walk - done properly, closely following Small /ive)

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

Multiple Processors: NUMA

Two physical processors on a single mainboard:

- Each CPU has its own memory
- Each CPU can access the memory of the other CPU.

The penalty for accessing 'foreign' memory is \sim 50%.





Multi-threading

rics & (depth < Motosa

: = inside ? 1 + 1 ... ht = nt / nc, ddn bs2t = 1.0f - nnt D, N); B)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 Tr) R = (D * nnt - N * (dd

= * diffuse; = true;

. :fl + refr)) && (depth < NAXDEP⊺

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuent estimation - doing it properly, if; radiance = SampleLight(&rand, I, &L, &L e.x + radiance.y + radiance.z) > 0) &&

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

andom walk - done properly, closely followi /ive)

; at3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, 8p urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Multiple Processors: NUMA

Do we care?

- Most boards host 1 CPU.
- A quadcore still talks to memory via a single interface.

However:

Threadripper is a NUMA device.

Threadripper = 2x Zeppelin, with for each Zeppelin:

- L1, L2, L3 cache
 - A link to memory

This CPU behaves as two CPUs in a single socket.







Multi-threading

nics & (depth < Notice

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 Fr) R = (D = nnt - N - (300

= * diffuse; = true;

efl + refr)) && (depth < MAXDEPTH

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly, if; radiance = SampleLight(&rand, I, &L, &light) e.x + radiance.y + radiance.z) > 0) && doity

w = true; at brdfPdf = EvaluateDiffuse(L, N) * Psurviv at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf) * C

andom walk - done properly, closely follow /ive)

, t33 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, apdf) urvive; pdf; n = E * brdf * (dot(N, R) / pdf); sion = true:

Multiple Processors: NUMA

Threadripper & Windows:

- Threadripper hides NUMA from the OS
- Most software is not NUMA-aware.



Details: <u>https://www.extremetech.com/computing/283114-new-utility-can-double-amd-threadripper-2990wx-performance</u> <u>https://blog.michael.kuron-germany.de/2018/09/amd-ryzen-threadripper-numa-architecture-cpu-affinity-and-htcondor</u>



ics (depth < Modean

: = inside ? 1 + . . ht = nt / nc, ddn os2t = 1.0f - nnt 2, N); 3)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - Rc Fr) R = (D = nnt - N = (dd)

= * diffuse = true;

efl + refr)) && (depth < MAXDEDIII

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly closed if; radiance = SampleLight(&rand, I, &L, &Light) e.x + radiance.y + radiance.z) > 0) && (dobb

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) * (rad

andom walk - done properly, closely following Sec. /ive)

; t3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); Sion = true:

Today's Agenda:

- Self-modifying code
- Multi-threading (1)
- Multi-threading (2)
- Experiments





Trust No One

Windows

```
DWORD WINAPI myThread(LPVOID lpParameter)
{
    unsigned int& myCounter = *((unsigned int*)lpParameter);
    while(myCounter < 0xFFFFFFF) ++myCounter;
    return 0;</pre>
```

```
int main(int argc, char* argv[])
```

```
using namespace std;
unsigned int myCounter = 0;
DWORD myThreadID;
HANDLE myHandle = CreateThread(0, 0, myThread, &myCounter;, 0, &myThreadID;);
char myChar = ' ';
while(myChar != 'q') {
    cout << myCounter << endl;
    myChar = getchar();
}
CloseHandle(myHandle);
return 0;
```

```
t3 brdf = SampleDiffuse( diffuse, N, r1, r2,∫3R, 8
urvive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true:
```

efl + refr)) && (depth <

survive = SurvivalProbability(diff

radiance = SampleLight(&rand, I, &L

e.x + radiance.y + radiance.z) > 0)

at brdfPdf = EvaluateDiffuse(L, N

E * ((weight * cosThetaOut) / directPdf)

at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L);

refl * E * diffuse;

), N);

(AXDEPTH)

v = true;

/ive)

if;

rics & (depth < Mo⊙er

t = inside } 1 = 1 .] ht = nt / nc, ddn = uc os2t = 1.0f - nnt = ni 0, N); ≥)

nt a = nt - nc, b = nt nt Tr = 1 - (R0 + (1 - R0 r) R = (D ⁼ nnt - N = (000

= * diffuse; = true;

. efl + refr)) && (depth < MAXDEP

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffue)
estimation - doing it properly, closed
if;
radiance = SampleLight(&rand, I, &L, &L;
e.x + radiance.y + radiance.z) > 0) && ()

w = true; at brdfPdf = EvaluateDiffuse(L, N) * Ps at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely followic /ive)

```
;

t3 brdf = SampleDiffuse( diffuse, N, r1, r2, 8

urvive;

pdf;

n = E * brdf * (dot( N, R ) / pdf);

sion = true:
```

Boost

#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)

boost::this_thread::sleep_for(boost::chrono::seconds{seconds});

void thread()

```
for (int i = 0; i < 5; ++i)
{
    wait(1);
    std::cout << i << '\n';
}</pre>
```

int main()

boost::thread t{thread};
t.join();



OpenMP

```
tics
& (depth < Mode
```

: = inside 7 1 1 1 0 ht = nt / nc, ddn 0 bs2t = 1.0f - nnt 0 D, N); ≥)

```
nt a = nt - nc, b = nt
nt Tr = 1 - (R0 + (1 - R0
r) R = (D = nnt - N = (dd)
```

```
= * diffuse;
= true;
```

```
-
efl + refr)) && (depth < MAXDE
```

```
D, N );
refl * E * diffuse;
= true;
```

```
AXDEPTH)
```

```
survive = SurvivalProbability( diffuse
estimation - doing it properly closed
f;
radiance = SampleLight( &rand, I, &L, &l
e.x + radiance.y + radiance.z) > 0) && (
```

```
w = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Ps
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf)
```

```
andom walk - done properly, closely follo
/ive)
```

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

#pragma omp parallel for for(int n = 0; n < 10; ++n) printf(" %d", n); printf(".\n");

```
float a[8], b[8];
#pragma omp simd
for( int n = 0; n < 8; ++n) a[n] += b[n];</pre>
```

struct node { node *left, *right; }; extern void process(node*); void postorder_traverse(node* p)

if (p->left)

```
#pragma omp task
    postorder_traverse(p->left);
if (p->right)
    #pragma omp task
    postorder_traverse(p->right);
#pragma omp taskwait
process(p);
```



nics & (depth < Mo⊙a⊨

: = inside } i = 1.0 ht = nt / nc, ddn - 0. bs2t = 1.0f - nnt m D, N); ð)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - RC) Fr) R = (D = nnt - N = (don

= * diffuse; = true;

. efl + refr)) && (depth < MAXDEPIN

), N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly, closed if; radiance = SampleLight(&rand, I, &L, &light e.x + radiance.y + radiance.z) > 0) && (doing)

w = true; at brdfPdf = EvaluateDiffuse(L, N) * Psur at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf) *

andom walk - done properly, closely following Sec. /ive)

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

Intel TBB

#include "tbb/task_group.h"

using namespace tbb;

int Fib(int n)

if (n<2)

return n;

else

```
int x, y;
task_group g;
g.run( [&]{x=Fib( n - 1 );} ); // spawn a task
g.run( [&]{y=Fib( n - 2 );} ); // spawn another task
g.wait(); // wait for both tasks to complete
return x + y;
```



Trust No One

Considerations

nics & (depth < Mooran

: = inside ? 1 : 1 : ht = nt / nc, ddn ps2t = 1.0f - nnt 2, N); 3)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - Rc) Fr) R = (D ^{*} nnt - N ^{**} (dd

= * diffuse; = true;

efl + refr)) && (depth < MAXOS

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse)
estimation - doing it properly, closed
if;
radiance = SampleLight(&rand, I, &L, &Light)
e.x + radiance.y + radiance.z) > 0) && (dot)

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) * (rad

andom walk - done properly, closely following Source /ive)

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

When using external tools to manage your threads, ask yourself:

- What is the overhead of creating / destroying a thread?
- Do I even know when threads are created?
- Do I know on which cores threads execute?

What if... we handled everything ourselves?



at a : at Tr [r) R = * d: = tru

-≘fl +), N refl = tr

4AXDEF survi∖ estin if; radiar ≥.x +

v = t at br at3 f at we at co E *

andom /ive)

at3 irvi pdf;

sion =

i = E * brdf * (dot(N, R) / pdf);

	worker thread 0		T0 ← L1 I-\$ ← →	1.2. ¢
	worker thread 1		T1 ↔ L1 D-\$ ↔	L2 Φ
	worker thread 2		T0 <mark>← L1 I-\$</mark> ← →	1.2 ¢
	worker thread 3		T1 ↔ L1 D-\$ ↔	LZ ₽
ffuse; e;	worker thread 4		TO 🕂 L1 I-\$	12 ¢
refr)) &&	worker thread 5		T1 ↔ L1 D-\$ ↔	LZ Ø
; E * diffu: He;				
ντH)	worker thread 6		T0 <mark>→</mark> L1 I-\$ →	12 \$
e = Survi va Nation - do	worker thread 7		T1 🕂 L1 D-\$ 🔶	
ice = Sample radiance.y	Light(&rand, I, &L, &lightCloop + radiance.z) > 0) && (dout find			
ue; HFPdf = Eva Inctor = dif ght = Mis2 ThetaOut = (weight =) walk - don rdf = Samplo	<pre>wateDiffuse(L, N) Put tasks:</pre>	threads never die threads are pinned to a core e claimed by worker threads on of a task may depend on completion of	f other tasks	1005 · 500
; * brdf * ()	Iasks ca	in produce new tasks		375/ 3011

Trust No One

urvive; pdf;

1 = E * brdf * (dot(N, R) / pdf);



at3 brdf = SampleDiffuse(diffuse, N, r1, r2, 8 Tasks can produce new tasks



nics & (depth < MOD00

: = inside ? 1 ; 1 ∂ ht = nt / nc, ddn os2t = 1.0f - nnt 0, N); 3)

at a = nt - nc, b = nt - nc at Tr = 1 - (R0 + (1 - R0 Ir) R = (D = nnt - N = (d0)

= * diffuse = true;

efl + refr)) && (depth < MAXDEPIIII

D, N); refl * E * diffuse; = true;

AXDEPTH)

v = true; at brdfPdf = EvaluateDiffuse(L, N) * Psurvir at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf) * (r

andom walk - done properly, closely following Sec. /ive)

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

Fibers:

"Cooperative multithreading", no preemption

Fibers on Windows:

https://docs.microsoft.com/ en-us/windows/win32/procthread/fibers

ConvertThreadToFiber CreateFiber SwitchToFiber

Cross-platform fibers:

https://github.com/JarkkoPFC/fiber



Rules of Engagement

Multithreading & Performance

- SMT / Hyperthreading: sharing L1 & L2 cache
 - Problems similar to simply having more threads
 - However, without the extra threads we don't benefit from SMT
 - Mitigate: have the threads work on the same data
- Multiple cores
 - Threads may travel from one core to the next (mind the caches)
 - Must share bandwidth
 - Mind false sharing

NUMA

- Thread assignment now depends on what memory is used
- No longer a theoretical issue
- Libraries
 - Generally favor ease of use over performance



at a = nt

), N);

(AXDEPTH)

v = true:

/ive)

if;

efl + refr)) && (depth <)

survive = SurvivalProbability(diff)

radiance = SampleLight(&rand, I, &l

E * ((weight * cosThetaOut) / directPdf) andom walk - done properly, closely follow

e.x + radiance.y + radiance.z) > 0

at brdfPdf = EvaluateDiffuse(L, N) at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf) at cosThetaOut = dot(N, L);

refl * E * diffuse;



ics (depth < Modean

: = inside ? 1 + . . ht = nt / nc, ddn os2t = 1.0f - nnt 2, N); 3)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - Rc Fr) R = (D = nnt - N = (dd)

= * diffuse = true;

efl + refr)) && (depth < MAXDEDIII

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly closed if; radiance = SampleLight(&rand, I, &L, &Light) e.x + radiance.y + radiance.z) > 0) && (dobb

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) * (rad

andom walk - done properly, closely following Sec. /ive)

; t3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); Sion = true:

Today's Agenda:

- Self-modifying code
- Multi-threading (1)
- Multi-threading (2)
- Experiments





Experiments

nics & (depth < NOCCS

: = inside | 1 ht = nt / nc, ddn os2t = 1.0f - nnt | 1 D, N); a)

at a = nt - nc, b = nt - m at Tr = 1 - (R0 + (1 - R0) Fr) R = (D = nnt - N = (d0)

= * diffuse; = true;

efl + refr)) && (depth < MAXDEPTHIL

), N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(difference estimation - doing it properly, closed Hf; radiance = SampleLight(&rand, I, &L, &L e.x + radiance.y + radiance.z) > 0) && (

v = true; at brdfPdf = EvaluateDiffuse(L, N) * Psu at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom 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:

Trust No One

How fast does OpenMP make an 'embarrassingly parallel' application?

void Game::Tick(float deltaTime)

```
// draw one line of pixels
static int xtiles = SCRWIDTH / TILESIZE, ytiles = SCRHEIGHT / TILESIZE;
static int tileCount = xtiles * ytiles;
for( int i = 0; i < tileCount; i++ ) // #pragma omp parallel for
{</pre>
```

int tx = i % xtiles; int ty = i / xtiles; drawtile(screen, tx * TILESIZE, ty * TILESIZE);





Experiments

nics & (depth < MODE

c = inside ? 1 : 1 : ht = nt / nc, ddn ss2t = 1.0f - nmt 2, N); 2)

nt a = nt - nc, b = nt nt Tr = 1 - (R0 + (1 - R0 r) R = (D [#] nnt - N [®] (dd

= * diffuse; = true;

. :fl + refr)) && (depth < MAXDEDIII)

), N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse)
estimation - doing it properly
f;
radiance = SampleLight(&rand, I, &L, &L
e.x + radiance.y + radiance.z) > 0) && ()

w = true; at brdfPdf = EvaluateDiffuse(L, N) * Psi at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely following /ive)

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

Trust No One

How fast does OpenMP make an 'embarrassingly parallel' application?

Can we do better?

void Game::Tick(float deltaTime)

```
// draw one line of pixels
static int xtiles = SCRWIDTH / TILESIZE, ytiles = SCRHEIGHT / TILESIZE;
static int tileCount = xtiles * ytiles;
for( int i = 0; i < tileCount; i++ ) // #pragma omp parallel for
{
    int tx = i % xtiles;</pre>
```

int ty = i / xtiles; drawtile(screen, tx * TILESIZE, ty * TILESIZE);





Experiments

Worker Threads

), N); refl * E * diffuse;

AXDEPTH)

survive = SurvivalProbability(dif if; radiance = SampleLight(&rand, I, &L, e.x + radiance.y + radiance.z) > 0) 8

v = true; at brdfPdf = EvaluateDiffuse(L, N) at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely follow /ive)

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

static DWORD threadId[THREADCOUNT]; static int params[THREADCOUNT]; static HANDLE worker[THREADCOUNT]; // spawn worker threads for(int i = 0; i < 4; i++)</pre>

params[i] = i; worker[i] = CreateThread(NULL, 0, workerthread, ¶ms[i], 0, &threadId[i]);



TRUST NO ONE

Experiments

Worker Threads

ics kj(depth < №006

: = inside ? 1 + 1 ... ht = nt / nc, ddn bs2t = 1.0f - nnt D, N); B)

at a = nt - nc, b = nt - nc at Tr = 1 - (R0 + (1 - RC Fr) R = (D ⁼ nnt - N ⁻ (dd

= * diffuse; = true;

-:fl + refr)) && (depth < MONDEPTH)

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly, closed Hf; radiance = SampleLight(&rand, I, &L, &light) e.x + radiance.y + radiance.z) > 0) && (dot)

w = true; at brdfPdf = EvaluateDiffuse(L, N) * Psur at3 factor = diffuse * INVPI; at weight = Mis2(directPdf, brdfPdf); at cosThetaOut = dot(N, L); E * ((weight * cosThetaOut) / directPdf)

andom walk - done properly, closely following S /ive)

```
;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, SR, apdf
urvive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true:
```

HANDLE goSignal[4], doneSignal[4];

volatile LONG remaining = 0;

unsigned long __stdcall workerthread(LPVOID param)

```
int threadId = *(int*)param;
while (1)
```

```
WaitForSingleObject( goSignal[threadId], INFINITE );
while (remaining > 0)
```

```
int task = (int)InterlockedDecrement( &remaining ) - 1;
if (task >= 0)
```

```
int tx = task % xtiles, ty = task / xtiles;
drawtile( theScreen, tx * TILESIZE, ty * TILESIZE );
```

SetEvent(doneSignal[threadId]);

TRUST NO ONE



Experiments

Worker Threads

nics ≹j(depth < NOCDS

: = inside ? l : l : ht = nt / nc, ddh os2t = 1.0f - nnt 0, N); 3)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - R0 Tr) R = (D ⁼ nnt - N

= * diffuse; = true;

. efl + refr)) && (depth < MAXDEPTI

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse) estimation - doing it properly closed H; radiance = SampleLight(&rand, I, &L, &Light) 2.x + radiance.y + radiance.z) > 0) && (dott)

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) * (rad

andom walk - done properly, closely following Small /ive)

; t3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); Sion = true:

remaining = tileCount;
for(int i = 0; i < 4; i++) SetEvent(goSignal[i]);
WaitForMultipleObjects(THREADCOUNT, doneSignal, true, INFINITE);</pre>



TRUST NO ONE

ics (depth < Modean

: = inside ? 1 + . . ht = nt / nc, ddn os2t = 1.0f - nnt 2, N); 3)

at a = nt - nc, b = nt at Tr = 1 - (R0 + (1 - Rc Fr) R = (D = nnt - N = (dd)

= * diffuse = true;

efl + refr)) && (depth < MAXDEDIII

D, N); refl * E * diffuse; = true;

AXDEPTH)

survive = SurvivalProbability(diffuse estimation - doing it properly closed if; radiance = SampleLight(&rand, I, &L, &Light) e.x + radiance.y + radiance.z) > 0) && (dobb

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) * (rad

andom walk - done properly, closely following Sec. /ive)

; t3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, &pdf urvive; pdf; n = E * brdf * (dot(N, R) / pdf); Sion = true:

Today's Agenda:

- Self-modifying code
- Multi-threading (1)
- Multi-threading (2)
- Experiments





/INFOMOV/

END of "Snippets"

next lecture: "Exam Practice"

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

andom walk - done properly, closely following S /ive)

, t3 brdf = SampleDiffuse(diffuse, N, r1, r2, &R, dod urvive; pdf; n = E * brdf * (dot(N, R) / pdf); rion = true:

