

```
ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * ddn;
        cos2t = 1.0f - nnt * ddn;
        D, N );
    }
    at a = nt - nc, b = nt + nc;
    at Tr = 1 - (R0 + (1 - R0) * ddn);
    (Fr) R = (D * nnt - N * (ddn *
    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, &L, &light,
    e.x + radiance.y + radiance.z) > 0) && (depth <
    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
    (survive)
    ;
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
}
```

/INFOMOV/

Optimization & Vectorization

J. Bikker - Sep-Nov 2018 - Lecture 8: "Data-Oriented Design"

Welcome!



Today's Agenda:

- OOP Performance Pitfalls
- DOD Concepts
- Practical DOD
- DOD or OO?



OOP

“Death by a Thousand Paper Cuts”

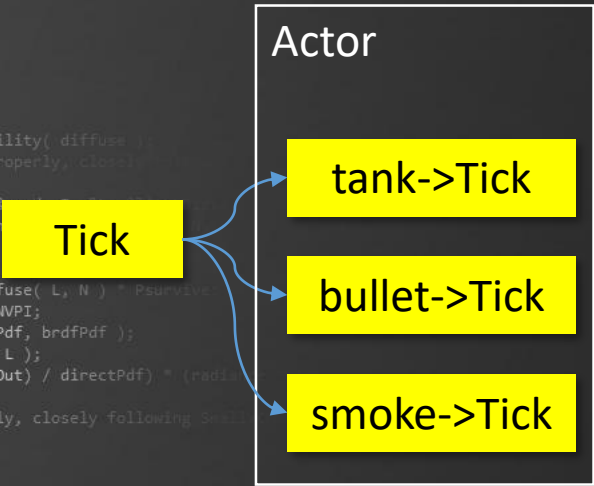
Object Oriented Programming:

- Objects
 - Data
 - Methods
 - Instances

```

...
& (depth < MAXDEPTH)
...
= inside ? 1.0 : 0.0;
nt = nt / nc; ddn = ddn * nc;
cos2t = 1.0f - nnt * nnt;
D, N );
...
at a = nt - nc; b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * cos2t);
Tr) R = (D * nnt - N * (ddn
...
E * diffuse;
= true;
...
refl + refr)) && (depth < MAXDEPTH)
...
D, N );
refl * E * diffuse;
= true;
...
MAXDEPTH)
survive = SurvivalProbability( diffuse, i);
estimation - doing it properly, closely following
df;
radiance = SampleLight( &L, N );
e.x + radiance.y + radiance.z;
...
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurface;
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;

```



Cost of a virtual function call:

1. Virtual Function Table
2. No inlining
- ...

Calling such a function:

- cache miss** 1. Read pointer to VFT of base class
- cache miss** 2. Add function offset
- branch** 3. Read function address from VFT
- 4. Load address in PC (jump)

But, that isn't realistic, right?

It is, if we use OO for what it was designed for: operating on heterogeneous objects.



OOP

“Death by a Thousand Paper Cuts”

Characteristics of OO:

- Virtual calls
- Scattered individual objects

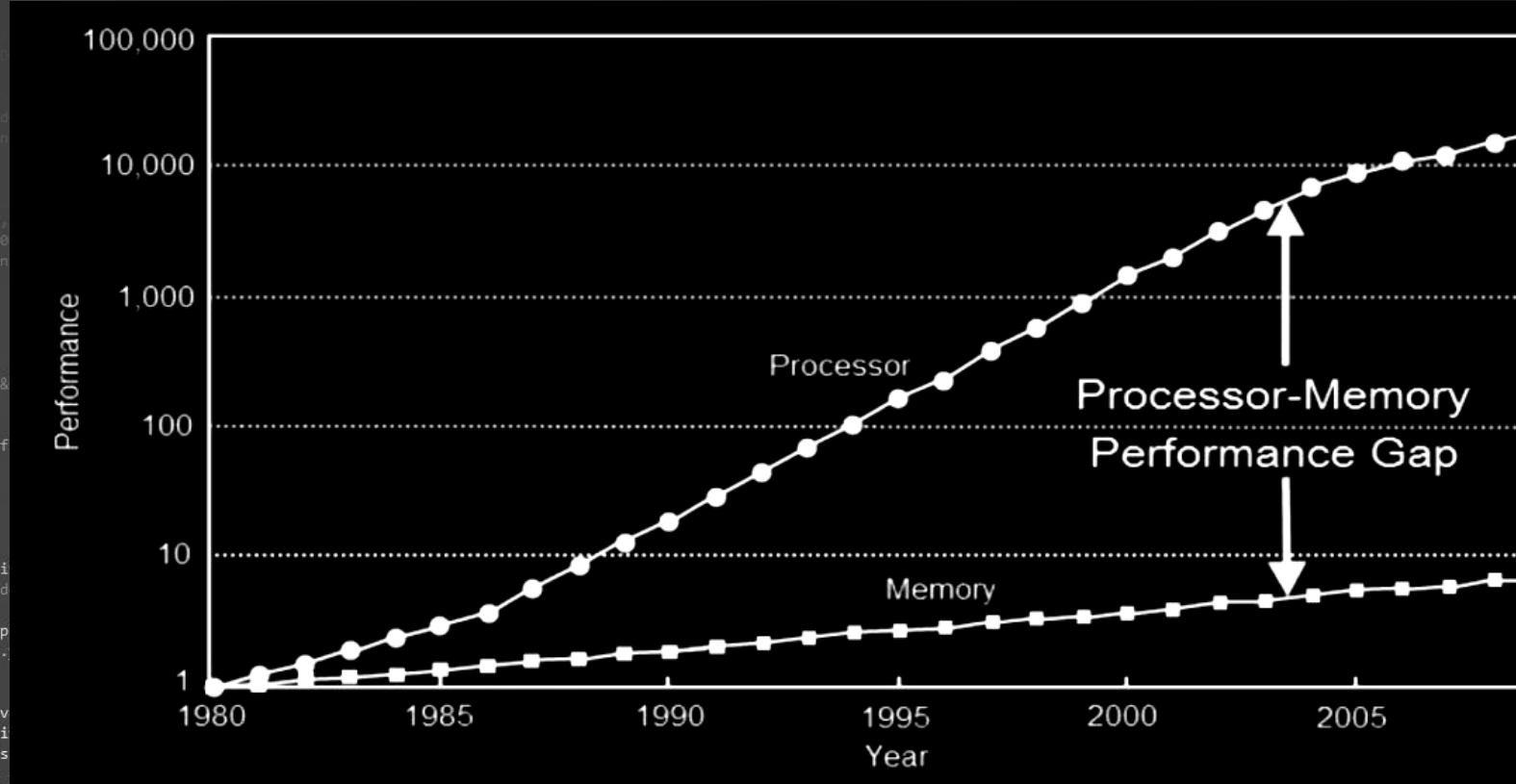
```

ics
& (depth < MAXDEPTH)
c = inside ? 1 : 0;
nt = nt / nc; ddn = ddn * ddn;
s2t = 1.0f - nnt * nnt;
D, N );
0);
at a = nt - nc, b = nt + nc;
at Tr = 1 - (R0 + (1 - R0) * s2t);
Tr) R = (D * nnt - N * (ddn *
E * diffuse;
= true;
efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;
MAXDEPTH)
survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light;
e.x + radiance.y + radiance.z) > 0) && (depth <
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;
    
```



OOP

“Death by a Thousand Paper Cuts”



Reading memory: 40 cycles @ 300Mhz



Reading memory: 600 cycles @ 3.2Ghz



OOP

“Death by a Thousand Paper Cuts”

Dealing with “bandwidth starvation”:

Caching

Continuous memory access (full cache lines)

Large array continuous memory access

(caches ‘read ahead’)

```

...
    & (depth < MAXDEPTH)
...
    if (inside ? 1 : 0)
        nt = nt / nc; ddn = ddn / nc;
        ns2t = 1.0f - nnt * nnt;
        D, N );
    )
...
    at a = nt - nc, b = nt + nc;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (ddn
...
    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, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (abs(radiance
...
    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
survive)
...
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```



OOP

“Death by a Thousand Paper Cuts”

Code performance is typically bound by memory access.

“The ideal data is in a format that we can use with the least amount of effort.”

➔ Effort = CPU-effort.

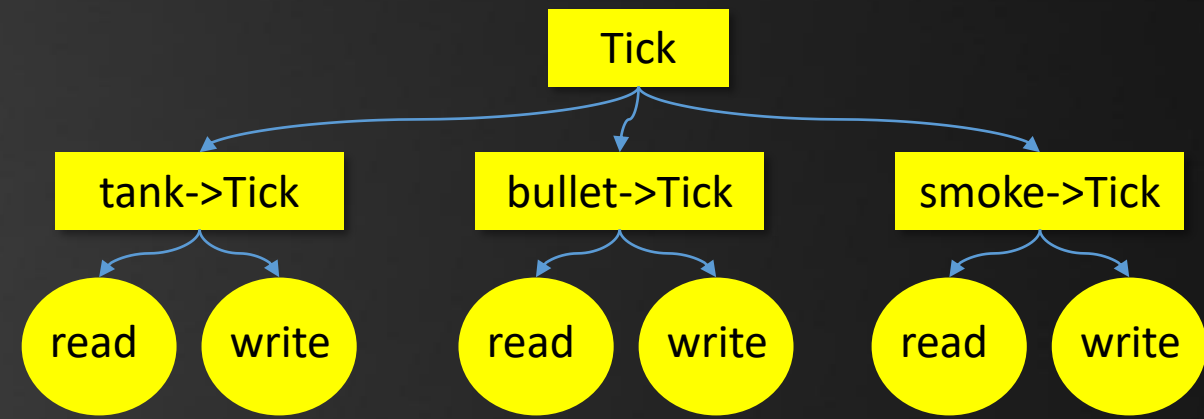
“You cannot be fast without knowing how data is touched.”



OOP

“Death by a Thousand Paper Cuts”

Parallel processing typically requires synchronization.



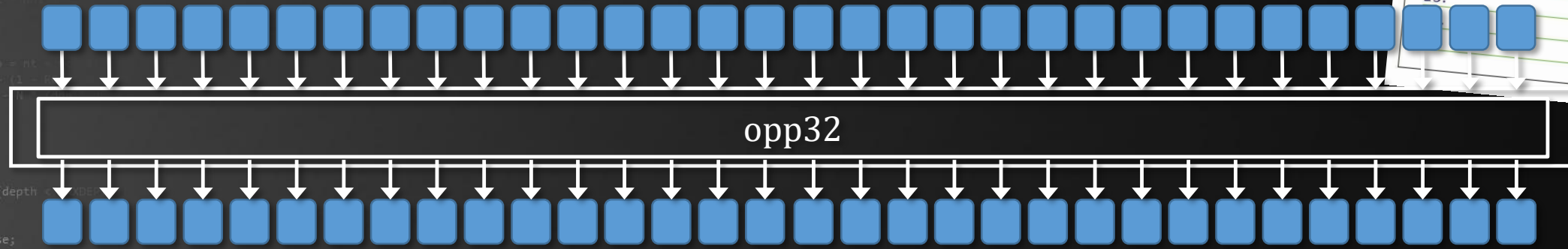
“You cannot multi-thread without knowing how data is touched.”



OOP

“Death by a Thousand Paper Cuts”

Parallel processing requires coherent program flow.

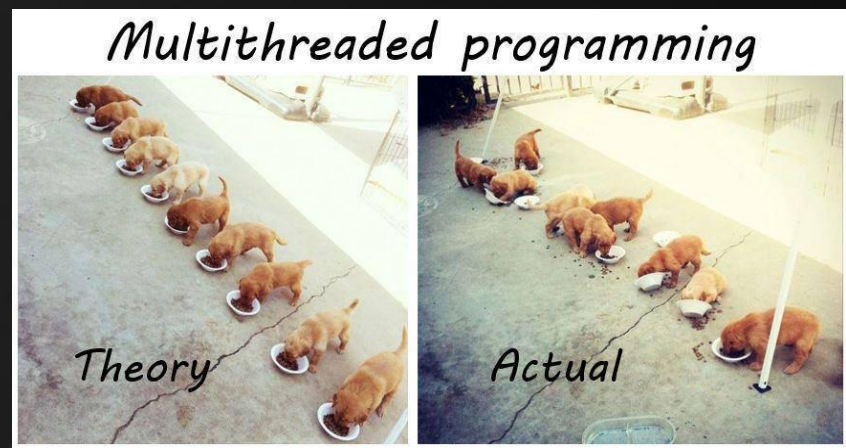


```

...
    & (depth < MAXDEPTH)
...
    inside ? 1 : 0;
    nt = nt / nc;
    ns2t = 1.0f - nnt;
    D, N );
    );
...
    at a = nt - nc; b = nt;
    at Tr = 1 - (R0 + R1);
    Tr) R = (D * nnt -
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth <
...
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
    survive = SurvivalProbability( diffuse;
    estimation - doing it properly, closely
    df;
    radiance = SampleLight( &rand, I, &L, &light
    e.x + radiance.y + radiance.z) > 0) && (cos
...
    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
    vive)
...
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;

```

“You cannot multi-thread without knowing how data is touched.”



OOP

“Death by a Thousand Paper Cuts”



```

class Bot : public Enemy
{
    ...
    vec3 m_position;
    ...
    float m_mod;
    ...
    float m_aimDirection;
    ...
    virtual void updateAim( vec3 target )
    {

```

cached but not used

cached but not used

branch

cache miss

cache miss

cache miss

cache miss

cache miss



OOP

“Death by a Thousand Paper Cuts”

```

void updateAims(
    float* aimDir,
    const AimingData* aim,
    vec3 target,
    uint count
)
{
    for (uint i = 0; i < count; ++i)
    {
        aimDir[i] = dot3(aim->positions[i],target) * aim->mod[i];
    }
}

```

only reads data that is actually needed to cache

reads from linear array

writes to linear array

actual functionality is unchanged



Today's Agenda:

- OOP Performance Pitfalls
- DOD Concepts
- Practical DOD
- DOD or OO?



DOD

Data Oriented Design*

Origin: low-level game development.

Core idea: *focus software design on CPU- and cache-aware data layout.*

Take into account:

- Cache line size
- Data alignment
- Data size
- Access patterns
- Data transformations

Strive for a simple, linear access pattern as much as possible.

*: Nikos Drakos, “Data Oriented Design”, 2008. <http://www.dataorienteddesign.com/dodmain>



DOD

Bad Access Patterns: Linked List

The Perfect LinkedList™:

```

struct LLNode
{
    LLNode* next;
    int value;
};

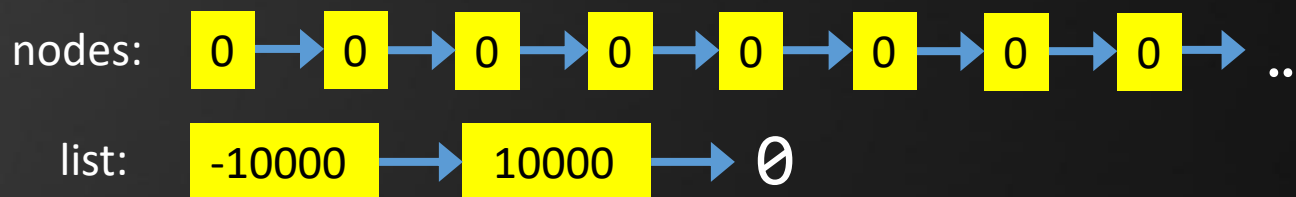
LLNode* nodes = new LLNode[...];
LLNode* pool = nodes;

for( int i = 0; i < ...; i++ )
    nodes[i].next = &nodes[i + 1];
    
```

```

LLNode* NewNode( int value )
{
    LLNode* retval = pool;
    pool = pool->next;
    retval->value = value;
    return retval;
}

list = NewNode( -10000 );
list->next = NewNode( 10000 );
list->next->next = 0;
    
```



DOD

Bad Access Patterns: Linked List

The Perfect LinkedList™, experiment:

Insert 25000 random values in the list so that we obtain a sorted sequence.

```
for( int i = 0; i < COUNT; i++ )
{
    LLNode* node = NewNode( rand() & 8191);
    LLNode* iter = list;
    while (iter->next->value < node->value)
        iter = iter->next;
    node->next = iter->next;
    iter->next = node;
}
```

```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * 2;
        ps2t = 1.0f - nnt * (nt / nc);
        D, N );
    }
}

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

E * diffuse;
= true;

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

MAXDEPTH)
survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (abs(radiance.x) < 1) && (abs(radiance.y) < 1) && (abs(radiance.z) < 1)
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.x * L.x + radiance.y * L.y + radiance.z * L.z);
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;

```



DOD

Bad Access Patterns: Linked List

KISS Array™:

```
data = new int[...];
memset( data, 0, ... * sizeof( int ) );
data[0] = -10000;
data[1] = 10000;
N = 2;
```

```
for( int i = 0; i < COUNT; i++ )
{
    int pos = 1, value = rand() & 8191;
    while (data[pos] < value) pos++;
    memcpy( data + pos + 1,
           data + pos,
           (N - pos + 1) * sizeof( int ) );
    data[pos] = value, N++;
}
```



DOD



```

...ics
& (depth < MAXDEPTH)
...
= inside ? 1 : 0;
nt = nt / nc;
s2t = 1.0f - nnt;
D, N );
}
...
at a = nt - nc, b = nt - nc;
at Tr = 1 - (R0 + (1 - R0) * nnt);
Fr) R = (D * nnt - N * (1 - nnt));
E * diffuse;
= true;
...
efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;
MAXDEPTH)
}
survive = SurvivalProbability( diffuse, N );
estimation - doing it properly, closely following the
if;
radiance = SampleLight( &rand, I, &L, &light, &N );
e.x + radiance.y + radiance.z) > 0) && (rand() < 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 the
(ive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf );
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```

```

for( int i = 0; i < COUNT; i++ )
{
    int pos = 1, value = rand() & 8191;
    while (data[pos] < value) pos++;
    memcpy( data + pos + 1, data + pos,
            (N - pos + 1) * sizeof( int ) );
    data[pos] = value, N++;
}

```



DOD

Bad Access Patterns: Linked List*

Inserting elements in an array by shifting the remainder of the array is *significantly faster* than using an optimized linked list.

Why?

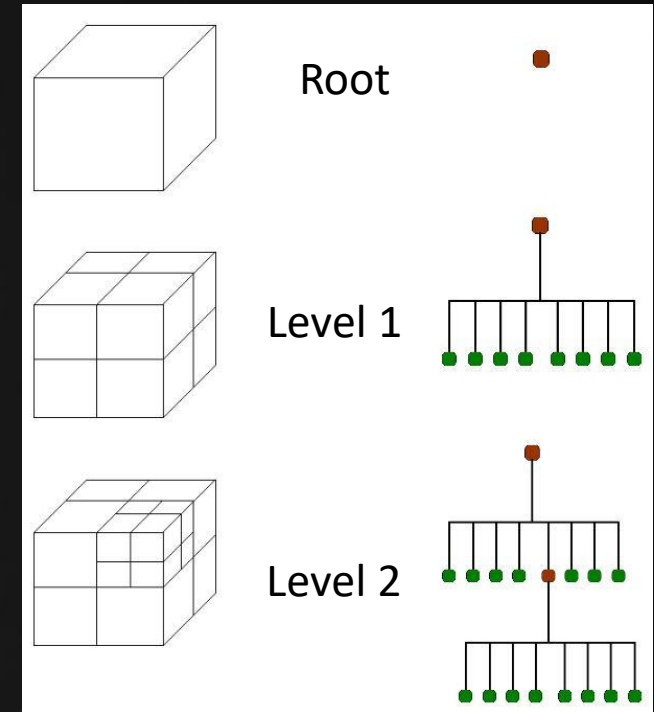
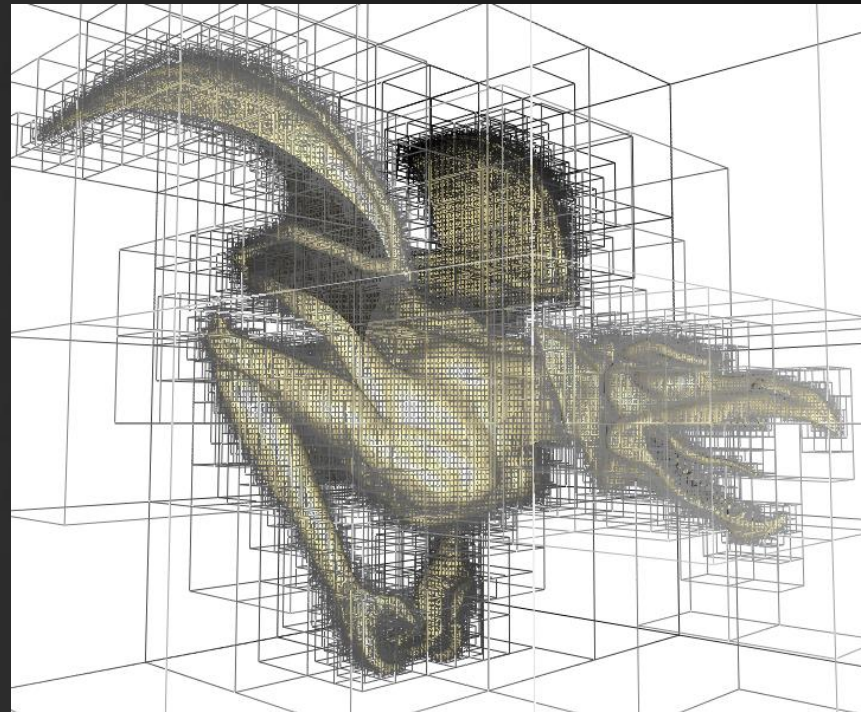
- Finding the location in the array: pure linear access
 - Shifting the remainder: pure linear access.
- ➔ Even though the amount of transferred memory is huge, this approach wins.

*: Also see: Nathan Reed, Data Oriented Hash Table, 2015.
<http://www.reedbeta.com/blog/data-oriented-hash-table>



DOD

Bad Access Patterns: Octree



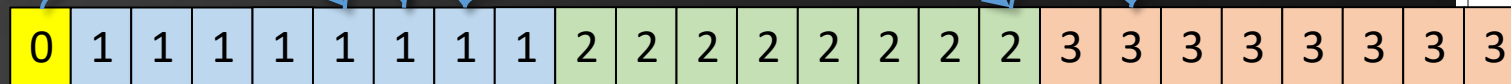
DOD

Bad Access Patterns: Octree

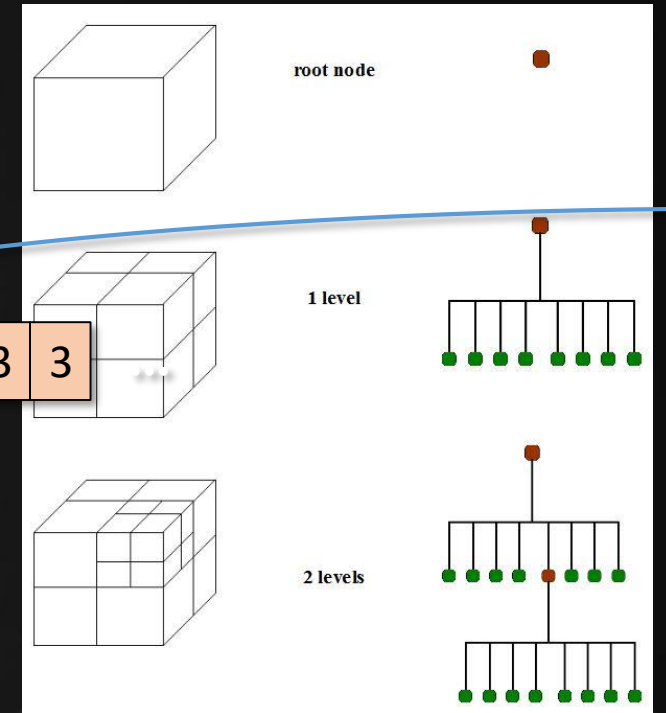
Query: find the color of a voxel visible through pixel (x,y).

Operation: ‘3DDDA’ (basically: Bresenham).

Data layout:



Color data: 32-bit (ARGB).



```

ics
& (depth < MAXDEPTH)
c = inside ? 1 : 0;
nt = nt / nc; ddn = ddn * ddn;
os2t = 1.0f - nnt * nnt;
D, N );
0);
at a = nt - nc, b = nt - nc;
at Tr = 1 - (R0 + (1 - R0) * Tr);
Tr) R = (D * nnt - N * (ddn *
E * diffuse;
= true;
efl + refr)) && (depth < MAXDEPTH);
D, N );
refl * E * diffuse;
= true;
MAXDEPTH);
survive = SurvivalProbability( diffuse, r);
estimation - doing it properly, closely following
df;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (rand < r);
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 Bresenham's
ive);
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf);
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
    
```



DOD

Bad Access Patterns: Octree

Alternative layout:

1. Tree 1: occlusion (1 bit per voxel);
2. Tree 2: color information (32 bits per voxel).

Use tree 1 to find the voxel you are looking for.

Lookup the correct voxel (incurring a single cache miss) in tree 2.

Caching in tree 1:

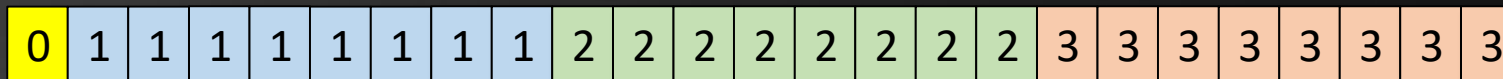
- A cache line holds $64 \cdot 8 = 512$ voxels
- Accessing the root gets several levels in L1 cache



DOD

Bad Access Patterns: Octree

Alternative layout (part 2):



Trees are typically generated by a divide-and-conquer algorithm, in a depth-first fashion.

Compact storage:

```
struct OTNode
{
    int firstChild;
    // bit 31: empty
    // bit 30: leaf
};
```

```

ics
& (depth < MAXDEPTH)

= inside ? 1 : 0;
nt = nt / nc; ddn = ddn / dnc;
s2t = 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 * (ddn

E * diffuse;
= true;

efl + refr)) && (depth < MAXDEPTH)

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

MAXDEPTH)

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

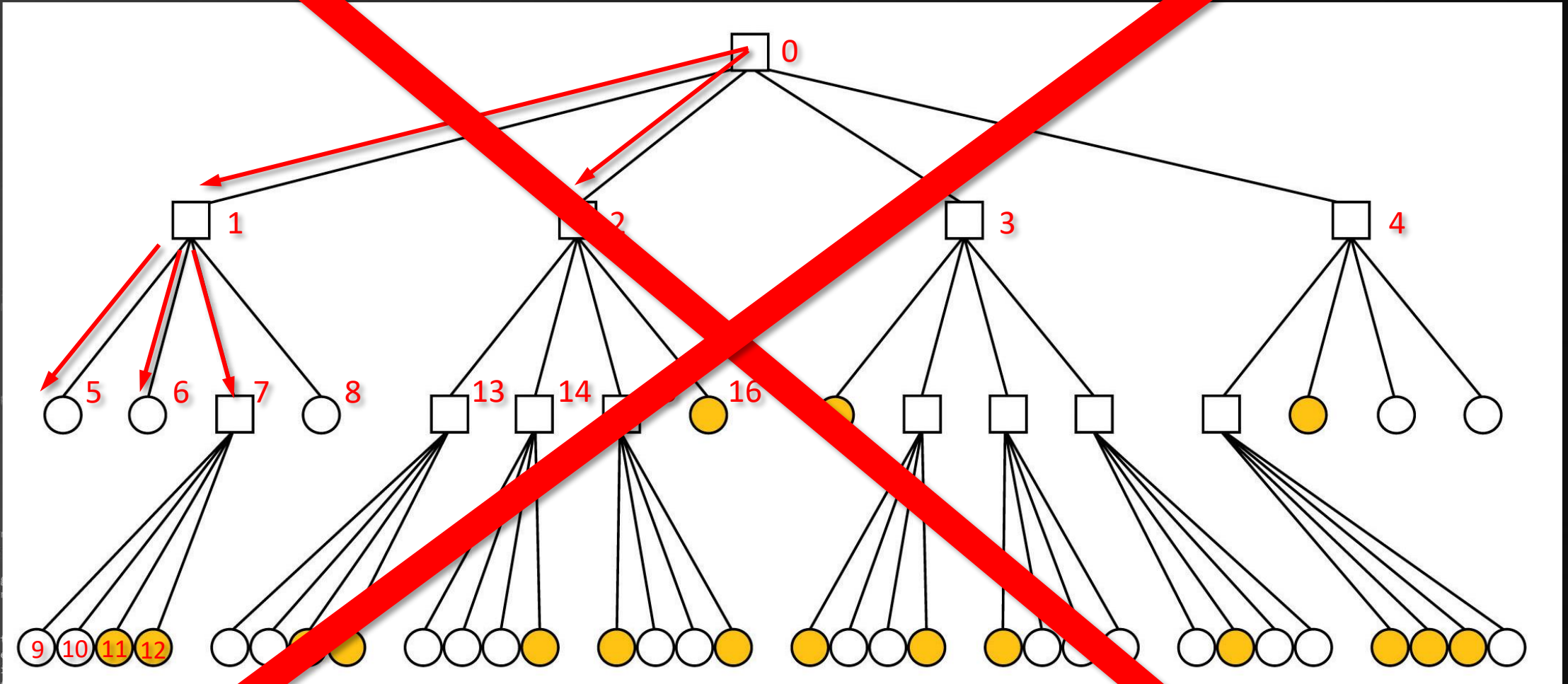
v = true;
at brdfPdf = EvaluateDiffuse( L, N ) * Psurvive;
at3 factor = diffuse * INVPI;
at weight = Mis2( directPdf, brdfPdf );
at cosThetaOut = dot( N, L );
E * ((weight * cosThetaOut) / directPdf) * (radiance

random walk - done properly, closely following
ive)

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



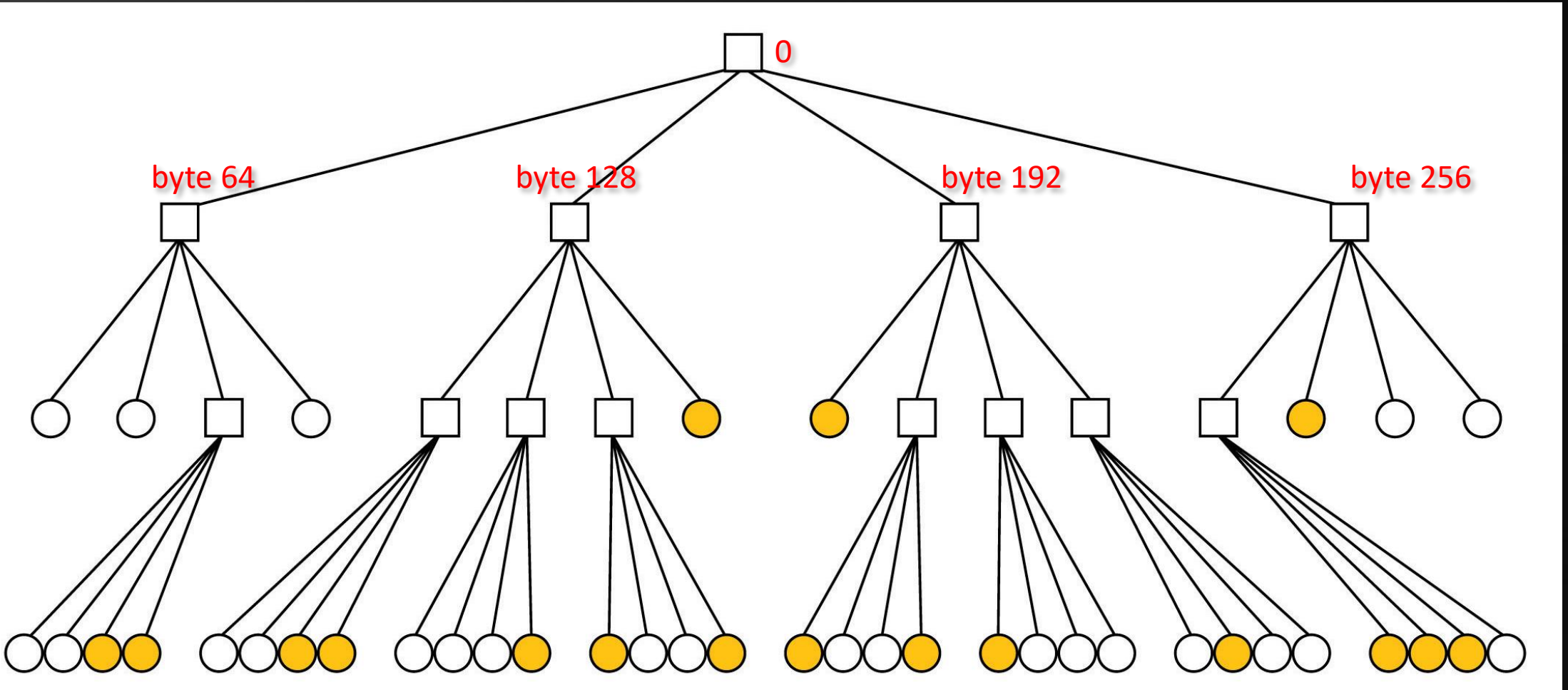
DOD



```
ics  
& (depth < MAXDEPTH)  
nt = nt / nc, ddn  
os2t = 1.0f - nnt  
, N );  
)  
at a = nt - nc, b =  
at Tr = 1 - (R0 + (R1  
Tr) R = (D * nnt -  
E * diffuse;  
= true;  
efl + refr)) && (de  
, N );  
refl * E * diffuse;  
= true;  
MAXDEPTH)  
survive = SurvivalP  
estimation - doing  
if;  
radiance = SampleLi  
e.x + radiance.y +  
w = true;  
at brdfPdf = Evaluat  
at3 factor = diffuse  
at weight = Mis2( d  
at cosThetaOut = dot(N, e //  
E * ((weight * cosThetaOut) / directPdf) * radiance  
random walk - done properly, closely to  
(ive)  
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf  
survive;  
pdf;  
n = E * brdf * (dot( N, R ) / pdf);  
sion = true;
```



DOD



DOD

Bad Access Patterns: Octree

Alternative layout (part 2):

- Reorganize so that treelets are cacheline-aligned.

(you will waste some memory)

```

ics
& (depth < MAXDEPTH)
{
    if (inside ? 1 : 0)
    {
        nt = nt / nc; ddn = ddn * nc;
        s2t = 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 * (ddn * nnt));

E * diffuse;
= true;

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

MAXDEPTH)
survive = SurvivalProbability( diffuse );
estimation - doing it properly, closely following
if;
radiance = SampleLight( &rand, I, &L, &light);
e.x + radiance.y + radiance.z) > 0) && (depth <
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);
urvive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```



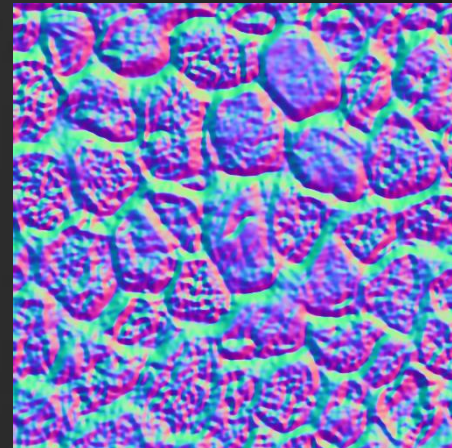
DOD

Bad Access Patterns: Textures in a Ray Tracer

Typical process for tracing a ray:

- Traverse a tree (multiple kilobytes)
- Intersect triangles in the leaf nodes (quite a few bytes)
- If a hit is found, fetch texture.

This is almost always a cache miss.



DOD

Bad Access Patterns: Textures in a Ray Tracer

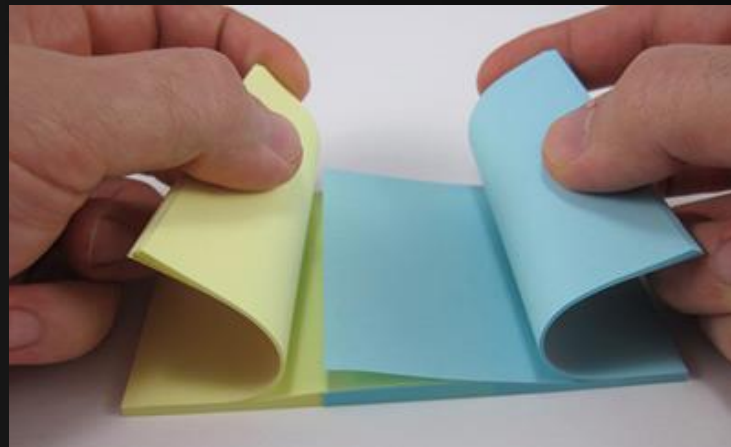
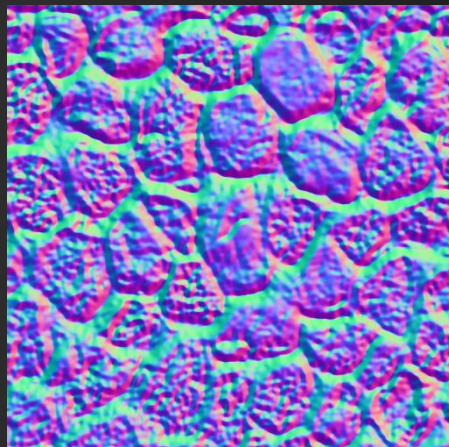
We suffer the cache miss *twice*:

- Once for the texture;
- Once for the normal map.

Note: both values are 32-bit.

```

...
    && (depth < MAXDEPTH)
...
    if (inside ? 1 : 0) {
        nt = nt / nc; ddn = ddn * ddn;
        cos2t = 1.0f - nnt * ddn;
        D, N );
    }
...
    at a = nt - nc, b = nt + nc;
    at Tr = 1 - (R0 + (1 - R0) * ddn);
    at R = (D * nnt - N * (ddn *
...
    E * diffuse;
    = true;
...
    refl + refr) && (depth < MAXDEPTH)
...
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)
...
    survive = SurvivalProbability( diffuse, r1, r2, && (depth < MAXDEPTH) );
    estimation - doing it properly, closely follow
    df;
    radiance = SampleLight( &rand, I, &L, &R, &N, &D, &N );
    e.x + radiance.y + radiance.z) > 0) && (depth <
...
    w = 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 follow
    ve);
...
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, && (depth < MAXDEPTH) );
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
    
```



DOD

Previously in INFOMOV

```

...
    & (depth < MAXDEPTH)
...
    if (inside ? 1 : 0) {
        nt = nt / nc; ddn = ddn * nc;
        nnt = nnt * nnt; nnt2t = 1.0f - nnt * nnt;
        D, N );
    }
...
    at a = nt - nc; b = nt * nc;
    at Tr = 1 - (R0 + (1 - R0) * nnt);
    (Fr) R = (D * nnt - N * (ddn * nnt2t));
...
    E * diffuse;
    = true;
...
    refl + refr)) && (depth < MAXDEPTH)
...
    D, N );
    refl * E * diffuse;
    = true;
...
MAXDEPTH)

```

```

struct Particle
{
    float x, y, z;
    float vx, vy, vz;
    float mass;
};
// size: 28 bytes

```

Better:

```

struct Particle
{
    float x, y, z;
    float vx, vy, vz;
    float mass, dummy;
};
// size: 32 bytes

```

```

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

```



DOD

Previously in INFOMOV

```

...ics
& (depth < MAXDEPTH)
...
= inside ? 1.0f : 0.0f;
nt = nt / nc; ddn = sqrt(1 - nt);
s2t = 1.0f - nnt * ddn;
D, N );
0);
...
at a = nt - nc; b = nt * nc;
at Tr = 1 - (R0 + (1 - R0) * ddn);
Tr) R = (D * nnt - N * (ddn *
...
E * diffuse;
= true;
...
efl + refr)) && (depth < MAXDEPTH)
D, N );
refl * E * diffuse;
= true;
...
MAXDEPTH)
survive = SurvivalProbability( diffuse,
estimation - doing it properly
if;
radiance = SampleLight( &rand, I, &L, &light;
e.x + radiance.y + radiance.z) > 0) && (depth <
...
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 S&S11.1
vive)
...
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf;
survive;
pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;

```

```

struct Particle
{
    float x, y, z;
    int mass;
};
Particle particle[512];

```

```

union { float x[512]; __m128 x4[128]; };
union { float y[512]; __m128 y4[128]; };
union { float z[512]; __m128 z4[128]; };
union { int mass[512]; __m128i mass4[128]; };

```

AOS

SOA

structure of arrays

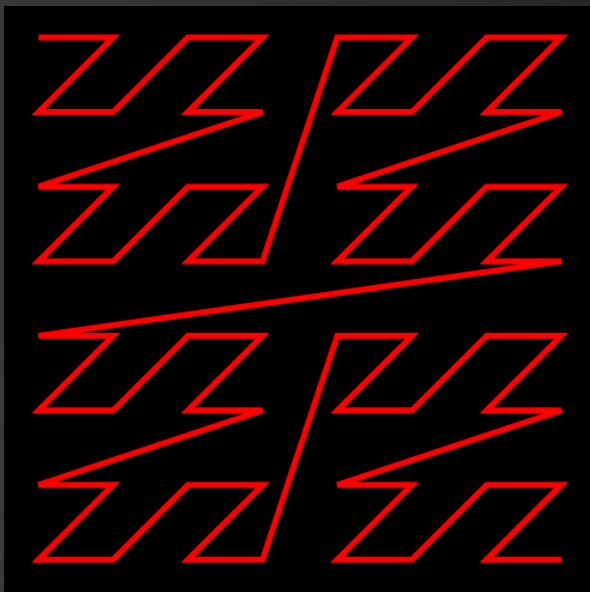


DOD

Previously in INFOMOV

```

ics
& (depth < MAXDEPTH)
    if (inside ? 1 : 0)
        nt = nt / nc; ddn = ddn * ddn;
        cos2t = 1.0f - nnt * nnt;
        D, N );
        )
        at a = nt - nc, b = nt + nc;
        at Tr = 1 - (R0 + (1 - R0) * I);
        Tr) R = (D * nnt - N * (ddn *
        E * diffuse;
        = true;
        refl + refr)) && (depth < MAXDEPTH)
        D, N );
        refl * E * diffuse;
        = true;
        MAXDEPTH)
        survive = SurvivalProbability( diffuse, I);
        estimation - doing it properly, closely following
        if;
        radiance = SampleLight( &rand, I, &L, &light;
        e.x + radiance.y + radiance.z) > 0) && (cos
        w = true;
        at brdfPdf = EvaluateDiffuse( L, N ) * Psum;
        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 90-degree
        vive)
        ;
        at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf;
        survive;
        pdf;
        n = E * brdf * (dot( N, R ) / pdf);
        sion = true;
    
```



Method:

X = 1 1 0 0 0 1 0 1 1 0 1 1 0 1

Y = 1 0 1 1 0 1 1 0 1 0 1 1 1 0

M = 1101101000111001110011111001

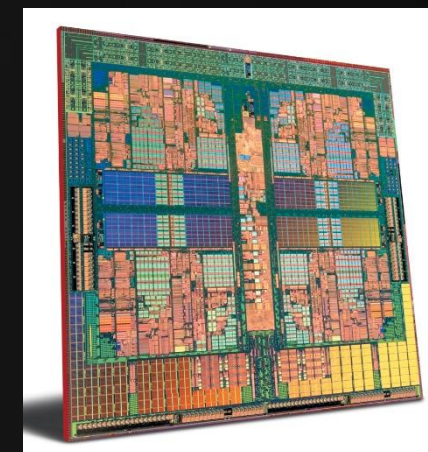
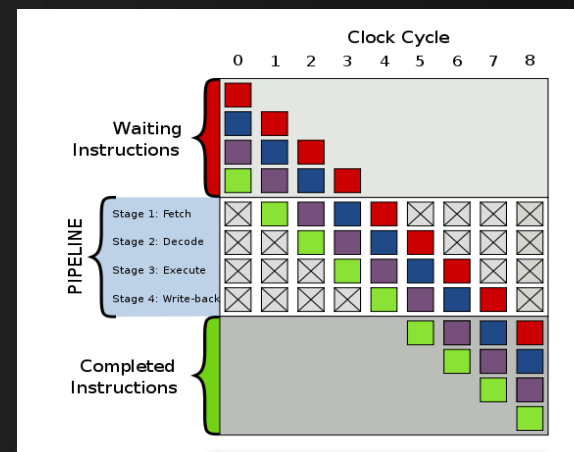
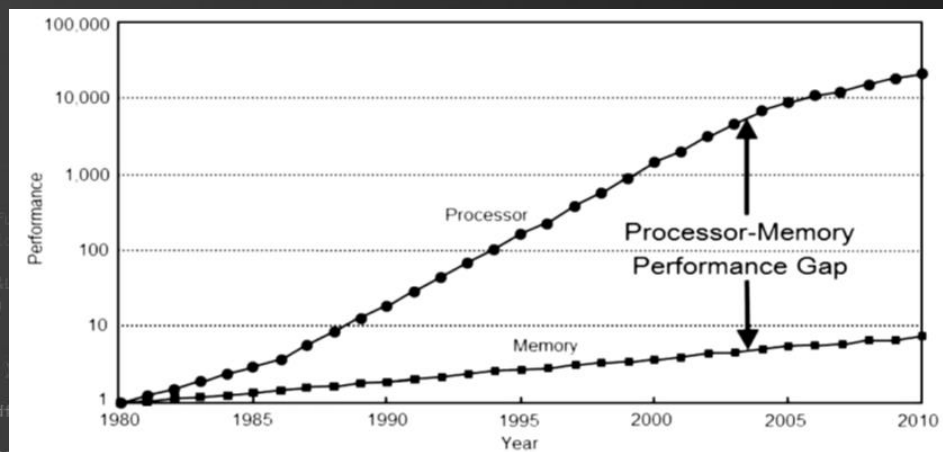


DOD

Algorithm Performance Factors

Estimating algorithm cost:

1. Algorithmic Complexity : $O(N)$, $O(N^2)$, $O(N \log N)$, ...
2. Cyclomatic Complexity* (or: Conditional Complexity)
3. Amdahl's Law / Work-Span Model
4. Cache Effectiveness



*. McCabe, A Complexity Measure, 1976.



Today's Agenda:

- OOP Performance Pitfalls
- DOD Concepts
- Practical DOD
- DOD or OO?



2B|~2B

OO = Evil, DO = Good?

10% of your code runs 90% of the time. DO is good for this 10%.

For all other code, please:

- Use STL
- Apply OO
- Program in C#
- Use event handling
- Check return values
- Focus on productivity



```

ics
& (depth < MAXDEPTH)
    c = inside ? 1 : 0;
    nt = nt / nc; ddn = ddn * ddn;
    ps2t = 1.0f - nnt * nnt;
    D, N );
    )
    at a = nt - nc; b = nt * nc;
    at Tr = 1 - (R0 + (1 - R0) *
    Tr) R = (D * nnt - N * (ddn
    E * diffuse;
    = true;
    refl + refr)) && (depth < MAXDEPTH)
    D, N );
    refl * E * diffuse;
    = true;
    MAXDEPTH)
    survive = SurvivalProbability( diffuse,
    estimation - doing it properly, closely
    df;
    radiance = SampleLight( &rand, I, &L, &light
    e.x + radiance.y + radiance.z) > 0) && (rand
    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 3rd
    ive)
    at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, $pdf
    survive;
    pdf;
    n = E * brdf * (dot( N, R ) / pdf);
    sion = true;
    
```



/INFOMOV/

END of “Data-Oriented Design”

next lecture: “GPGPU (1)”

