Welcome!
Today's Agenda:

- OOP Performance Pitfalls
- DOD Concepts
- DOD or OO?
"Floating point code is (typically) undeterministic"

```c
float v0 = 1;
float v1 = 1;
float v2 = 1;
float v3 = 1;
float v4 = 1;
float v5 = 1;
float v6 = 1;
float v7 = 1;
for (int i = 0; i < 200000; i++) {
    v0 *= 1.00001f;
    v1 *= 1.00001f;
    v2 *= 1.00001f;
    v3 *= 1.00001f;
    v4 *= 1.00001f;
    v5 *= 1.00001f;
    v6 *= 1.00001f;
    v7 *= 1.00001f;
    fld1
    fld st(0)
    fld st(1)
    fld st(2)
    fld st(3)
    fld st(4)
    fld st(5)
    fld st(6)
    fmul st(7), st; fxch st(7); fstp [v0]
    fxch st(5); fmul st, st(6)
    fxch st(4); fmul st, st(6)
    fxch st(3); fmul st, st(6)
    fxch st(2); fmul st, st(6)
    fxch st(1); fmul st, st(6)
    fxch st(5); fmul st, st(6)
    fld [v7] fmul st, st(7) fstp [v7]
```
Fact Checking

“Doubles are slower than floats (4x)”

This statement is **mostly true**. The real story, CPU (win32, x64):

- **A float** takes 32-bit in memory, but gets promoted to 80 bits in an FPU register.
- **A double** takes 64-bit in memory, but gets promoted to 80 bits in an FPU register.
- **A long double** takes 64-bit in memory, but gets promoted to 80 bits in an FPU register.

Calculation time on 80-bit FPU registers does not depend on the source of the data. HOWEVER: the fp registers are rarely used anymore...

The real story, GPU (Nvidia, AMD):  
- **Titan V**: FP64 = 1/2 * FP32 (6900 vs 13800 GFLOPS)
- **Titan X Pascal**: FP64 = 1/32 * FP32 (350 vs 11300 GFLOPS) (same for all 10xx)
- **Radeon RX Vega 64**: FP64 = 1/16 * FP32 (790 vs 12700 GFLOPS)
- **Radeon HD 7990**: FP64 = 1/4 * FP32 (1946 vs 7782)

**FP16 (GPU only):**  
- **GTX 1080Ti**: FP16 = 1/64 * FP32 (ouch)
- **Radeon RX Vega 64**: FP16 = 2 * FP32 (!)
Today’s Agenda:

- OOP Performance Pitfalls
- DOD Concepts
- DOD or OO?
“Death by a Thousand Cuts”

Object Oriented Programming:

- Objects
  - Data
  - Methods
  - Instances
“Death by a Thousand Cuts”

Object Oriented Programming:

- Objects
  - Data
  - Methods
  - Instances

Cost of a virtual function call:

1. Virtual Function Table
2. No inlining

... 

Calling such a function:

1. Read pointer to VFT of base class
2. Add function offset
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.
“Death by a Thousand Cuts”

Characteristics of OO:

- Virtual calls
- Scattered individual objects
"Death by a Thousand Cuts"

The problem is growing with time.
“Death by a Thousand Cuts”

Dealing with “bandwidth starvation”:

Caching

Continuous memory access (full cache lines)

Large array continuous memory access

(caches ‘read ahead’)
“Death by a Thousand 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.

“Most programs are made faster if we improve their memory access patterns.”
(this will be more true every year)

“You cannot be fast without knowing how data is touched.”
“Death by a Thousand Cuts”

Parallel processing typically requires synchronization.

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

“Death by a Thousand Cuts”

Parallel processing requires coherent program flow.

“You cannot multi-thread without knowing how data is touched.”
"Death by a Thousand Cuts"

class Bot : public Enemy {
    
    vec3 m_position;
    float m_mod;
    float m_aimDirection;

    virtual void updateAim( vec3 target )
    {
        m_aimDirection = dot3( m_position, target ) * m_mod;
    }
}
"Death by a Thousand Cuts"

```c
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
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

Today's Agenda:

- OOP Performance Pitfalls
- DOD Concepts
- 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.**

---

Bad Access Patterns: Linked List

The Perfect LinkedList™:

```c
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( -MAXINT );
list->next = NewNode( MAXINT );
list->next->next = 0;
```

```c
nodes: 0 0 0 0 0 0 0 0 0 ...
list: -10000 10000 0
```
Bad Access Patterns: Linked List

The Perfect LinkedList™, experiment:

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

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

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++;
}
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.

Bad Access Patterns: Octree
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).
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 $64 \times 8 = 512$ voxels
- Accessing the root gets several levels in L1 cache
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:

```c
struct OTNode {
    int firstChild;
    // bit 31 set: empty
}
```
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.*
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.
Bad Access Patterns: Textures in a Ray Tracer

Interleaved texture / normal:

- One value now becomes 64-bit and contains the normal and the color.
- We still suffer a cache miss –
- But only once.
Previously in INFOMOV

```
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
```
Previously in INFOMOV

```c
struct Particle {
    float x, y, z;
    int mass;
};
```

```c
Particle particle[512];
```

**AOS**

```
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];
};
```

**SOA**

```
struct of arrays
```
Previously in INFOMOV

Method:

\[
X = 1 1 0 0 0 1 0 1 1 0 1 1 0 1
\]

\[
Y = 1 0 1 1 0 1 1 0 1 0 1 1 1 0
\]

\[
M = 1101101000111001110011111001
\]
Today's Agenda:

- OOP Performance Pitfalls
- DOD Concepts
- 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
DATA-ORIENTED DESIGN AND C++
Presenter: Mike Acton

https://www.youtube.com/watch?v=rX0ItVEVjHc
Data-Oriented Design

Richard Fabian

- Contents
- Data-Oriented Design
  - It's all about the data
  - Data is not the problem domain
  - Data and statistics
  - Data can change
  - How is data formed?
  - The framework
  - Conclusions and takeaways

http://www.dataorienteddesign.com/dodbook/
Data Oriented Design Resources

A curated list of awesome data oriented design resources.

Feel free to contribute by sending PR!

- **Data Oriented Design**
  - Presentations
  - Blog Posts
  - Videos
  - Other
  - Code Examples

**Presentations**

- A Step Towards Data Orientation (2010) - Johan Torp
- Introduction To Data Oriented Design (2010) - DICE
- Memory Optimization (2003) - Christer Ericson
- Practical Examples In Data Oriented Design (2013) - Niklas Frykholm
- Three Big Lies (2008) - Mike Acton
- Typical C++ Bullshit (2008) - Mike Acton
- Data-Oriented Design and C++ (2014) - Mike Acton
- Entity Component Systems & Data Oriented Design (2018) - Aras Pranckevičius

**Blog Posts**

[https://github.com/dbartolini/data-oriented-design](https://github.com/dbartolini/data-oriented-design)

- Adventures in data-oriented design – Part 1: Mesh data (2011) - Stefan Reinalter
Adventures in data-oriented design – Part 1: Mesh data

Let’s face it; performance on modern processors (be it PCs, consoles or mobiles) is mostly governed by memory access patterns. Still, data-oriented design is considered something new and novel, and only slowly creeping into programmers’ brains, and this really needs to change. Having co-workers fix your code and improving its performance really is no excuse for writing crappy code (from a performance point-of-view) in the first place.

This post is the first in an ongoing series about how certain things in the Molecular Engine are done in a data-oriented fashion, while still making use of OOP concepts. A common misconception about data-oriented design is that it is “G-like”, and “not OOP”, and therefore less maintainable – but that need not be the case. The concrete example we are going to look at today is how to organize your mesh data, but let’s start with the pre-requisites first.

https://blog.molecular-matters.com/2013/05/02/adventures-in-data-oriented-design-part-3a-ownership/
Today's Agenda:

- OOP Performance Pitfalls
- DOD Concepts
- DOD or OO?
END of “Data-Oriented Design”

next lecture: “GPGPU (1)”