Welcome!

Optimization & Vectorization

J. Bikker - Sep-Nov 2018 - Lecture 8: “Data-Oriented Design”
Today’s Agenda:

- OOP Performance Pitfalls
- DOD Concepts
- Practical DOD
- DOD or OO?
“Death by a Thousand Paper 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.
OOP

“Death by a Thousand Paper Cuts”

Characteristics of OO:

- Virtual calls
- Scattered individual objects
"Death by a Thousand Paper Cuts"
“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’)
“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.”
"Death by a Thousand Paper Cuts"

Parallel processing typically requires synchronization.

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

Parallel processing requires coherent program flow.

"You cannot multi-thread without knowing how data is touched."
“Death by a Thousand Paper 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 Paper 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
Today's Agenda:

- OOP Performance Pitfalls
- DOD Concepts
- Practical DOD
- DOD or OO?
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(-10000);
list->next = newNode(10000);
list->next->next = 0;
```

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™:**

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

```c
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;
    }
    node->next = 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.

http://www.reedbeta.com/blog/data-oriented-hash-table
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).
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*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: empty
    // bit 30: leaf
};
```
DOD

![Diagram](image_url)

- byte 64
- byte 128
- byte 192
- byte 256
Bad Access Patterns: Octree

Alternative layout (part 2):

- Reorganize so that treelets are cacheline-aligned.

(you will waste some memory)
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

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

Better:

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

**AOS**

**SOA**

structure of arrays
Previously in INFOMOV

Method:

\[
X = 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \\
Y = 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \\
-----------------------------------
M = 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1
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
- Practical DOD
- DOD or OO?
**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
END of “Data-Oriented Design”

next lecture: “GPGPU (1)"