Sony Computer Entertainment Europe
Research & Development Division

Pitfalls of Object Oriented Programming

Tony Albrecht – Technical Consultant
Developer Services
What I will be covering

• A quick look at Object Oriented (OO) programming
• A common example
• Optimisation of that example
• Summary
Object Oriented (OO) Programming

- **What is OO programming?**
  - a programming paradigm that uses "objects" – data structures consisting of datafields and methods together with their interactions – to design applications and computer programs. (Wikipedia)

- **Includes features such as**
  - Data abstraction
  - Encapsulation
  - Polymorphism
  - Inheritance
What’s OOP for?

• OO programming allows you to think about problems in terms of objects and their interactions.

• Each object is (ideally) self contained
  – Contains its own code and data.
  – Defines an interface to its code and data.

• Each object can be perceived as a ‘black box’.
Objects

• If objects are self contained then they can be
  – Reused.
  – Maintained without side effects.
  – Used without understanding internal implementation/representation.

• This is good, yes?
Are Objects Good?

• Well, yes
• And no.
• First some history.
A Brief History of C++

C++ development started

1979 2009
A Brief History of C++

1979 1983 2009

Named “C++”
A Brief History of C++

1979 1985 2009

First Commercial release
A Brief History of C++

Release of v2.0

1979  1989  2009
A Brief History of C++

Release of v2.0

1979

1989

2009

Added
• multiple inheritance,
• abstract classes,
• static member functions,
• const member functions
• protected members.
A Brief History of C++

1979 1998 2009

Standardised
A Brief History of C++

Updated 2009

1979  2003  2009
A Brief History of C++

1979

2009

C++0x

?
So what has changed since 1979?

- Many more features have been added to C++
- CPUs have become much faster.
- Transition to multiple cores
- Memory has become faster.

http://www.vintagecomputing.com
CPU performance

Performance

Year

CPU/Memory performance

![Graph showing the performance of processor and memory over time, with a significant gap between the two lines, indicating the increasing disparity in performance between the two components.](image)

- Processor
- Processor-Memory Performance Gap
- Memory


Performance Scale: 1, 10, 100, 1,000, 10,000, 100,000
What has changed since 1979?

• One of the biggest changes is that memory access speeds are far slower (relatively)
  – 1980: RAM latency ~ 1 cycle
  – 2009: RAM latency ~ 400+ cycles

• What can you do in 400 cycles?
What has this to do with OO?

- OO classes encapsulate code and data.
- So, an instantiated object will generally contain all data associated with it.
My Claim

• With modern HW (particularly consoles), excessive encapsulation is BAD.

• Data flow should be fundamental to your design (Data Oriented Design)
Consider a **simple** OO Scene Tree

- **Base Object class**
  - Contains general data
- **Node**
  - Container class
- **Modifier**
  - Updates transforms
- **Drawable/Cube**
  - Renders objects
Object

• Each object
  – Maintains bounding sphere for culling
  – Has transform (local and world)
  – Dirty flag (optimisation)
  – Pointer to Parent

```plaintext
class Object
{
  // <methods removed for clarity>
  Matrix4 m_Transform;
  Matrix4 m_WorldTransform;
  BoundingSphere m_BoundingSphere;
  BoundingSphere m_WorldBoundingSphere;
  char* m_Name;
  bool m_DIRTY;
  Object* m_Parent;
};
```
Objects

Class Definition

Memory Layout

Each square is 4 bytes

```c++
class object {
   // <methods removed for clarity>
   Matrix4 m_Transform;
   Matrix4 m_WorldTransform;
   BoundingSphere m_BoundingSphere;
   BoundingSphere m_WorldBoundingSphere;
   char* m_Name;
   bool m_Dirty;
   Object* m_Parent;
};
```
Nodes

• Each Node is an object, plus
  – Has a container of other objects
  – Has a visibility flag.

```cpp
class Node : public Object {
    // methods removed for clarity
    std::vector<Object*> m_objects;
    bool m_IsVisible;
};
```
Nodes

Class Definition

```cpp
class Node : public Object {
    // methods removed for clarity
    std::vector<Object*> m_Objects;
    bool m_IsVisible;
};
```

Memory Layout

<table>
<thead>
<tr>
<th>vtbl</th>
<th>Matrix4</th>
<th>Matrix4</th>
<th>Matrix4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vector3</th>
<th>Vector3</th>
<th>Vector3</th>
<th>char*</th>
<th>bool</th>
<th>Object*</th>
<th>vector</th>
<th>bool</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: The diagram represents the memory layout of the `Node` class, showing the fields and their types.
Consider the following code…

- Update the world transform and world space bounding sphere for each object.

```cpp
const BoundingSphere& Node::GetWorldBoundingSphere(const Matrix4& parentTransform) {
    m_worldTransform = parentTransform * m_Transform;
    for (std::vector<Object*>::const_iterator itr = m_objects.begin();
        itr != m_objects.end();
        ++itr) {
        m_worldBoundingSphere.ExpandBy((*itr)->GetWorldBoundingSphere(m_worldTransform));
    }
    return m_worldBoundingSphere;
}
```
Consider the following code...

- Leaf nodes (objects) return transformed bounding spheres

```cpp
virtual const BoundingSphere& GetWorldBoundingSphere(const Matrix4& parentTransform) {
    if (m_dirty)
        m_worldBoundingSphere = m_boundingSphere.Transform(parentTransform);
    return m_worldBoundingSphere;
}
```
Consider the following code...

- Leaf nodes (objects) return transformed bounding spheres.

What's wrong with this code?

```cpp
virtual const BoundingSphere& GetWorldBoundingSphere(const Matrix4& parentTransform) {
    if(mDirty)
        mWorldBoundingSphere = mBoundingSphere.Transform(parentTransform);
    return mWorldBoundingSphere;
}
```
Consider the following code...

- Leaf nodes (objects) return transformed bounding spheres.

If `m_Dirty` = `false` then we get branch misprediction which costs 23 or 24 cycles.

```cpp
virtual const BoundingBoxSphere& GetWorldBoundingSphere(const Matrix4& parentTransform) {
    if(m_Dirty)
        m_WorldBoundingSphere = m_BoundingSphere.Transform(parentTransform);
    return m_WorldBoundingSphere;
}
```
Consider the following code...

- Leaf nodes (objects) return transformed bounding spheres.

  Calculation of the world bounding sphere takes only 12 cycles.

```cpp
virtual const BoundingSphere& GetWorldBoundingSphere() const
{
    if(m_Dirty)
        m_WorldBoundingSphere = m_BoundingSphere.Transform(parentTransform);
    return m_WorldBoundingSphere;
}
```
Consider the following code…

- Leaf nodes (objects) return transformed bounding spheres.

So using a dirty flag here is actually slower than not using one (in the case where it is false).

```cpp
virtual const BoundingBox& GetWorldBoundingBox() const
{
    if (m_dirty)
        m_WorldBoundingBox = m_BoundingBox.Transform(parentTransform);
    return m_WorldBoundingBox;
}
```
Let's illustrate cache usage

Each cache line is 128 bytes

Main Memory

L2 Cache

m_worldTransform = parentTransform * m_Transform;
Cache usage

parentTransform is already in the cache (somewhere)

\[ m_{\text{worldTransform}} = \text{parentTransform} \times m_{\text{Transform}}; \]
Cache usage

Assume this is a 128-byte boundary (start of cacheline)

m_worldTransform = parentTransform * m_Transform;
Cache usage

Load m_Transform into cache

\[
m_{\text{worldTransform}} = \text{parentTransform} \times m_{\text{Transform}};
\]
**Cache usage**

Main Memory

```
m_worldTransform = parentTransform * m_Transform;
```

L2 Cache

m_WorldTransform is stored via cache (write-back)
Cache usage

Main Memory

L2 Cache

Next it loads m_Objects
Cache usage

Main Memory

Then a pointer is pulled from somewhere else (Memory managed by std::vector)

```cpp
for (std::vector<object*>::const_iterator itr = m_objects.begin();
     itr != m_objects.end(); ++itr)
{
    m_worldBoundingSphere.ExpandBy((*itr)->GetWorldBoundingSphere(m_worldTransform));
}
```
### Cache usage

**Main Memory**

<table>
<thead>
<tr>
<th>vtbl</th>
<th>Matrix4</th>
<th>Matrix4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vector3</th>
<th>radius</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector3</td>
<td>radius</td>
</tr>
<tr>
<td>char*</td>
<td>bool</td>
</tr>
<tr>
<td>bool</td>
<td></td>
</tr>
</tbody>
</table>

**L2 Cache**

<table>
<thead>
<tr>
<th>vtbl</th>
<th>Matrix4</th>
<th>Matrix4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vector3</th>
<th>radius</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector3</td>
<td>radius</td>
</tr>
<tr>
<td>char*</td>
<td>bool</td>
</tr>
<tr>
<td>bool</td>
<td></td>
</tr>
</tbody>
</table>

### Code Snippet

```cpp
for (std::vector<object*>::iterator itr = m_objects.begin();
     itr != m_objects.end();
    ++itr)
{
  m_worldBoundingSphere.ExpandBy((*itr)->GetWorldBoundingSphere(m_worldTransform));
}
```
Cache usage

Main Memory

L2 Cache

Look up virtual function

```cpp
for (std::vector<object*>::iterator itr = m_objects.begin();
    itr != m_objects.end();
    ++itr)
{
    m_worldBoundingSphere.ExpandBy((*itr)->GetWorldBoundingSphere(m_worldTransform));
}
Cache usage

Main Memory

L2 Cache

Then branch to that code (load in instructions)

```cpp
for(std::vector<object*>::iterator itr = m_objects.begin();
    itr != m_objects.end(); ++itr)
{
    m_worldBoundingSphere.ExpandBy((*itr)->GetWorldBoundingSphere(m_worldTransform));
}
```
New code checks dirty flag then sets world bounding sphere

```cpp
const BoundingSphere& GetWorldBoundingSphere(const Matrix4& parentTransform) {  
    if (m_dirty)  
        m_worldBoundingSphere = mBoundingSphere.Transform(parentTransform);  
    return m_worldBoundingSphere;  
}
Node’s World Bounding Sphere is then Expanded.

```cpp
for (std::vector<Object*>::const_iterator itr = m_objects.begin();
    itr != m_objects.end();
    ++itr)
{
    m_WorldBoundingSphere.ExpandBy((*itr)->GetWorldBoundingSphere(m_WorldTransform));
}
```
Then the next Object is processed.
Cache usage

First object costs at least 7 cache misses
Subsequent objects cost at least 2 cache misses each
The Test

• 11,111 nodes/objects in a tree 5 levels deep
• Every node being transformed
• Hierarchical culling of tree
• Render method is empty
Performance

This is the time taken just to traverse the tree!
Why is it so slow?

~22ms
Look at GetWorldBoundingSphere()
Samples can be a little misleading at the source code level.
if(!m_Dirty) comparison
Stalls due to the load 2 instructions earlier
Similarly with the matrix multiply
Some rough calculations

Branch Mispredictions: 50,421 @ 23 cycles each  \( \approx 0.36 \text{ms} \)
Some rough calculations

Branch Mispredictions: 50,421 @ 23 cycles each ~ = 0.36ms
L2 Cache misses: 36,345 @ 400 cycles each ~ = 4.54ms
• From Tuner, ~ 3 L2 cache misses per object
  – These cache misses are mostly sequential (more than 1 fetch from main memory can happen at once)
  – Code/cache miss/code/cache miss/code...
Slow memory is the problem here

- How can we fix it?
- And still keep the same functionality and interface?
The first step

• Use homogenous, sequential sets of data
# Homogeneous Sequential Data

<table>
<thead>
<tr>
<th>World transforms</th>
<th>Local Transforms</th>
<th>World Bounding Spheres</th>
<th>Local Bounding Spheres</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="#">Matrix4</a></td>
<td><a href="#">Matrix4</a></td>
<td><a href="#">Vector3</a></td>
<td><a href="#">Vector3</a></td>
</tr>
<tr>
<td><a href="#">Matrix4</a></td>
<td><a href="#">Matrix4</a></td>
<td><a href="#">Vector3</a></td>
<td><a href="#">Vector3</a></td>
</tr>
<tr>
<td><a href="#">Matrix4</a></td>
<td><a href="#">Matrix4</a></td>
<td><a href="#">Vector3</a></td>
<td><a href="#">Vector3</a></td>
</tr>
<tr>
<td><a href="#">Matrix4</a></td>
<td><a href="#">Matrix4</a></td>
<td><a href="#">Vector3</a></td>
<td><a href="#">Vector3</a></td>
</tr>
</tbody>
</table>

---

*Note: The image contains a table with entries for world transforms, local transforms, world bounding spheres, and local bounding spheres.*
Generating Contiguous Data

• Use custom allocators
  – Minimal impact on existing code

• Allocate contiguous
  – Nodes
  – Matrices
  – Bounding spheres
Performance

19.6ms -> 12.9ms
35% faster just by moving things around in memory!
What next?

• Process data in order
• Use implicit structure for hierarchy
  – Minimise to and fro from nodes.
• Group logic to optimally use what is already in cache.
• Remove regularly called virtuals.
We start with a parent Node

Hierarchy

Node
Hierarchy

Node

Node

Node

Which has children nodes
And they have a parent
And they have children
Hierarchy

And they all have parents
A lot of this information can be inferred
Hierarchy

Use a set of arrays, one per hierarchy level
Hierarchy

Parent has 2 children

Children have 4 children
Hierarchy

Ensure nodes and their data are contiguous in memory.
• Make the processing global rather than local
  – Pull the updates out of the objects.
    • No more virtuals
  – Easier to understand too – all code in one place.
Need to change some things…

• OO version
  – Update transform top down and expand WBS bottom up
Update
transform

Node

Node

Node

Node

Node

Node

Node

Node

Node

Node

Node

Node
Update transform and world bounding sphere
Add bounding sphere of child
Update transform and world bounding sphere
Add bounding sphere of child
Update transform and world bounding sphere
Add bounding sphere of child
• Hierarchical bounding spheres pass info up
• Transforms cascade down
• Data use and code is ‘striped’.
  – Processing is alternating
Conversion to linear

• To do this with a ‘flat’ hierarchy, break it into 2 passes
  – Update the transforms and bounding spheres (from top down)
  – Expand bounding spheres (bottom up)
Transform and BS updates

For each node at each level (top down)
{
    multiply world transform by parent’s transform
    wbs by world transform
}
Update bounding sphere hierarchies

For each node at each level (bottom up)
{
    add wbs to parent’s
cull wbs against frustum
}
Update Transform and Bounding Sphere

How many children nodes to process

```cpp
for(int j=0; j<size; j++)
{
    const int innersize = parent->m_objects.size();
    const Matrix4 *parentTransform = parent->m_worldTransform;
    j+=innersize;
    for(int k=0; k<innersize; k++, wmat++, mat++, bs++, wbs++)
    {
        *wmat = (*parentTransform)*(*mat);
        *wbs = bs->Transform(wmat);
    }
    parent++;
}
```
Update Transform and Bounding Sphere

```cpp
for(int j=0; j<size; j++)
{
    const int innersize = parent->m_objects.size();
    const Matrix4 *parentTransform = parent->m_worldTransform;

    j+=innersize;
    for(int k=0; k<innersize; k++, wmat++, mat++, bs++, wbs++)
    {
        *wmat = (*parentTransform)*(*mat);
        *wbs = bs->Transform(wmat);
    }
    parent++;
}
```

For each child, update transform and bounding sphere
Update Transform and Bounding Sphere

```cpp
for(int j=0;j<size;j++)
{
    const int innerSize = parent->m_objects.size();
    Matrix4 *parentTransform = parent->m_worldTransform;
    j+=innerSize;
    for(int k=0;k<innerSize;k++, wmat++, mat++, bs++, wbs++)
    {
        *wmat = (*parentTransform)*(*mat);
        *wbs = bs->Transform(wmat);
    }
    parent++;
}
```
So, what’s happening in the cache?

Unified L2 Cache

Parent

Children’s node data not needed

Parent Data

Children’s Data
Load parent and its transform

```
const int innerSize = parent->m_objects.size();
const Matrix4 *parentTransform = parent->m_worldTransform;
```
Load child transform and set world transform

```c
for(int k=0;k<innersize;k++, wmat++, mat++, bs++, wbs++)
{
    *wmat = (*parentTransform)*(*mat);
    *wbs = bs->Transform(wmat);
}
```
Load child BS and set WBS

```c
for(int k=0;k<innerSize;k++, wmat++, mat++, bs++, wbs++)
{
    *wmat = (*parentTransform)*(*mat);
    *wbs = bs->Transform(wmat);
}
```
Load child BS and set WBS

Next child is calculated with no extra cache misses!

for(int k=0;k<inner;
{
  *wmat = (*parentTransform)*(*mat);
  *wbs = bs->Transform(wmat);
}
Load child BS and set WBS

The next 2 children incur 2 cache misses in total

for (int k=0; k<inner; k++) {
    *wmat = (*parentTransform)*(*mat);
    *wbs = bs->Transform(wmat);
}
Because all data is linear, we can predict what memory will be needed in ~400 cycles and prefetch.
• Tuner scans show about 1.7 cache misses per node.
• But, these misses are much more frequent
  – Code/cache miss/cache miss/code
  – Less stalling
Performance

19.6 -> 12.9 -> 4.8ms
Prefetching

• Data accesses are now predictable
• Can use prefetch (dcbt) to warm the cache
  – Data streams can be tricky
  – Many reasons for stream termination
  – Easier to just use dcbt blindly
    • (look ahead x number of iterations)
Prefetching example

• Prefetch a predetermined number of iterations ahead

• Ignore incorrect prefetches

```c
for(int i=0; i<m_CurrentNumTransforms; i++)
{
    __dcbt(m_Transforms[i+20]);
    *m_Transforms[i] = (*m_Transforms[i])*(m_Transform);
}
```
Performance

19.6 -> 12.9 -> 4.8 -> 3.3ms
A Warning on Prefetching

• This example makes very heavy use of the cache

• This can affect other threads’ use of the cache
  – Multiple threads with heavy cache use may thrash the cache
The old scan

~22ms
The new scan

~16.6ms
Up close
Looking at the code (samples)

```cpp
const Node* parent = (Node*)node->m_Parent;

// iterate through all the matrices at this level, multiplying them by their parent
for(int j=0;j<size;j++)
{
    const int innerSize = parent->m_Objects.size();
    const Matrix4*parentTransform = parent->m_WorldTransform;

    j += innerSize;
    *parentTransform *= Matrix4::Identity();

    for(int k=0;k<innerSize;k++)
    {
        __dcbt(vmat+lookAhead);
        __dcbt(mat+lookAhead);
        __dcbt(bs+lookAhead);
        __dcbt(wbs+lookAhead);

        *wmat = (**parentTransform)*(*mat);
    }
    *wmat = (**node->m_Parent->m_WorldTransform)*(*mat);

    wbs = bs->Transform(vmat);
}
parent++;
```
Performance counters

Branch mispredictions: 2,867  (cf. 47,000)
L2 cache misses: 16,064         (cf 36,000)
In Summary

• Just reorganising data locations was a win
• Data + code reorganisation = dramatic improvement.
• + prefetching equals even more WIN.
OO is not necessarily EVIL

- Be careful not to design yourself into a corner
- Consider data in your design
  - Can you decouple data from objects?
  - ...code from objects?
- Be aware of what the compiler and HW are doing
Its all about the memory

• Optimise for data first, then code.
  – Memory access is probably going to be your biggest bottleneck

• Simplify systems
  – KISS
  – Easier to optimise, easier to parallelise
Homogeneity

• Keep code and data homogenous
  – Avoid introducing variations
  – Don’t test for exceptions – sort by them.

• Not everything needs to be an object
  – If you must have a pattern, then consider using Managers
Remember

• You are writing a GAME
  – You have control over the input data
  – Don’t be afraid to preformat it – drastically if need be.

• Design for specifics, not generics (generally).
Data Oriented Design Delivers

- Better performance
- Better realisation of code optimisations
- Often simpler code
- More parallelisable code
The END