VU Entwurf und Programmierung einer Rendering-Engine

Low Level Optimizations and Insights
Motivation

● Rendering Engine Goals include
  ○ High performance
  ○ Easy to use, high level programming abstraction

● Abstraction and Efficiency often conflicting goals

● Programming languages provide abstraction mechanisms including
  ○ Procedures, methods, virtual functions

● Abstraction mechanisms have costs

● Dropping abstraction mechanisms is not productive
  ○ Avoiding abstraction mechanisms makes software unmaintainable

● Declared goal:
  ○ Understand basic mechanisms of modern computer architecture and compiler technology in order to estimate cost and evaluate tradeoffs.
Scope of this lecture

● What not to expect
  ○ In-depth analysis of abstraction mechanisms with conclusions which can simply be applied to any problem in order to evaluate tradeoffs in the design and implementation of rendering engine modules.

● What to expect
  ○ Basic understanding of programming language abstraction mechanisms
  ○ which serves as basis for benchmarks for specific use cases

● In real world scenarios, theoretical and practical performance need to be evaluated for each usage scenario!
Baseline - What can we do per frame

- Especially for games but also for simulations, frame time is crucial
  - For VR rendering applications, low latency and high framerates are inevitable
- For defining performance criteria, it is necessary to estimate approximate costs for some instructions/tasks.
- Often computing theoretical throughput is useful when thinking about high level performance goals…. 
- How long takes a loop till 1 000 000, i5-4690K @ 3.5GHz?
  - Just measured 2ms = 500fps
  - 10 million: 18ms = 55fps
  - 100 million: 163ms
The cost of programming language abstraction....

- Micro benchmark
- How many of those per second?
  - Approx 500 mio on my machine
- In the next experiment, we will use
  - Static function calls
  - Method calls (obj on stack)
  - Method calls (obj in heap)
  - Method calls, virtual calls

```c
void mad(float m, float a) {
    X = X * m + a;
    Y = Y * m + a;
    Z = Z * m + a;
}
```
Reference solution

- Class defined in same file, full optimization
- VC++ 2015, whole program optimization
- 100 000 000 invocations: 219ms
  - 3.3 cycles per vector mad
  - Without loop overhead: 2.933 cycles per vector mad
- Observation
  - In same file, with proper optimization, there is no method call overhead.

```cpp
void V3f::mad(float m, float a)
{
    X = X * m + a;
    Y = Y * m + a;
    Z = Z * m + a;
}
```
Move vector class into separate file

- Without whole program analysis, severe slowdown
- Cannot count on whole program analysis if dynamically linked…

- 385ms total
- 5.36 cycles per vector mad
- Factor 1.82 slowdown
- Drawback of ahead of time compilation

```c
for (int i = 0; i < cnt; i++)
{
    stackVar.mad(m, a);

    00007FF6D3F517D0  movaps      xmm2,xmm7
    00007FF6D3F517D3  lea         rcx,[rbp-68h]
    00007FF6D3F517D7  movaps      xmm1,xmm8
    00007FF6D3F517DB  call        V3f::mad (07FF6D3F51AA0h)
    00007FF6D3F517E0  sub         rdi,1
    00007FF6D3F517E4  jne         main+2F0h (07FF6D3F517D0h)

    //global+=step;
}
```
Using functions

- Pure function call overhead
- All inlined: 195ms
  - 2.9 cycles per vec MAD
- functionMadCopy: 581ms
  - 8.7 cycles per vec MAD
- functionMadRef: 357ms
  - 5.31 cycles per vec MAD

```c
V3f functionMadCopy(V3f vec, float m, float a)
{
    Inline impl
}

void functionMadRef(V3f &vec, float m, float a)
{
    Inline impl
}
```
Method call overview (OOP)

- Reference was: 195ms
- Heap variable, method call: 363ms
- Stack variable, method call: 359ms
- Virtual method: 357ms
  - Additional fetch is free in optimal scenario
- Approx 5.5 cycles for all variants.
- Note that this is the best case (!)
  - All virtual function tables are in cache
  - How about multiple inheritance?
  - Virtual functions cannot be inlined easily (though JVM is doing this…)

```c
for (int i = 0; i < cnt; i++)
{
    vec->mad(m, a);

    00007FF72C0316B0  mov         rax,qword ptr [rbx]
    00007FF72C0316B3  movaps      xmm2,xmm7
    00007FF72C0316B6  movaps      xmm1,xmm8
    00007FF72C0316BA  mov         rcx,rbx
    00007FF72C0316BD  call        qword ptr [rax+10h]
    00007FF72C0316C0  sub         rdi,1
    00007FF72C0316C4  jne        main+1D0h
}
```

Virtual function in VC15
Implementation of virtual functions

class Point
{
    int x, y;
    public virtual void Move(int x, int y) { }
    public virtual void Draw() { }
}

class ColoredPoint : Point
{
    int color;
    public override void Draw() { }
    public void SetColor(int color) { }
}

Load vtblptr, (obj)
Load mptr, method(vtblptr)
Call mptr
Cache is performance

Intel Haswell i7-4700 3.4GHz,

- L1 Data Cache Latency = 4 cycles for simple access via pointer
- L1 Data Cache Latency = 5 cycles for access with complex address calculation (size_t n, *p; n = p[n]).
- L2 Cache Latency = 12 cycles
- L3 Cache Latency = 36 cycles (3.4 GHz i7-4770)
- L3 Cache Latency = 43 cycles (1.6 GHz E5-2603 v3)
- L3 Cache Latency = 58 cycles (core9) - 66 cycles (core5) (3.6 GHz E5-2699 v3 - 18 cores)
- RAM Latency = 36 cycles + 57 ns (3.4 GHz i7-4770)
- RAM Latency = 62 cycles + 100 ns (3.6 GHz E5-2699 v3 dual)

Data from: http://www.7-cpu.com/cpu/Haswell.html
Native compilation vs Just in Time Compilation

Same benchmarks in .NET 4.6 (includes SIMD stuff)

Virtual member call: 5.3 cycles
Struct member call: 2.9 cycles
SIMD vectors: 2.85 cycles
SIMD vectors + cross lib static member call: 2.84 cycles
Reflection: 441 cycles
Reflection compiled to lambda: similar to virtual

Very similar overheads

- JIT compilation can inline stuff at runtime
- This simple example could not benefit From SIMD, but for LinAlg speedup can be huge!
Optimization technique: images based on native memory in .NET

• Idea: use NativePtr instead of managed arrays
• Removes bound checks
• Lots of code can be inlined (in F# functions can be marked as must inline)
• Test case: copy (byte) image (2k * 2k) and optionally apply image trafos (identity, mirror, rot90)

### Rot0
- managed: 16.10ms
- native: 10.34ms
- memcpy: 1.89ms

### Rot90
- managed: 120.65ms
- native: 44.80ms

### MirrorY
- managed: 30.25ms
- native: 10.62ms

```c
while xptr <> eX do
  let eY = xptr + sY
  while xptr <> eY do
    let eZ = xptr + sZ
    while xptr <> eZ do
      *yptr <- *xptr
      xptr <- xptr + xjZ
      yptr <- yptr + yjZ
      xptr <- xptr + xjY
      yptr <- yptr + yjY
      xptr <- xptr + xjX
      yptr <- yptr + yjX
```
Cache efficiency for image operations

Same test case: copy (byte) image and optionally apply image trafo mirror

Use different loop nesting order….

member x.CopyTo(y : NativeVolume<'b>) =
  let cXY = compare (abs info.DX) (abs info.DY)  
  let cXZ = compare (abs info.DX) (abs info.DZ)  
  let cYZ = compare (abs info.DY) (abs info.DZ)  
  if cXY >= 0  && cXZ >= 0  && cYZ >= 0
    then x.CopyToXYZ(y) 
  elif cXY <= 0 && cXZ >= 0  && cYZ >= 0
    then x.CopyToYXZ(y) 
  elif cXY <= 0 && cXZ <= 0 && cYZ >= 0
    then x.CopyToYZX(y) 
  elif cXY >= 0  && cXZ >= 0  && cYZ <= 0
    then x.CopyToXZY(y) 
  elif cXY >= 0  && cXZ <= 0 && cYZ <= 0
    then x.CopyToZXY(y) 
  else x.CopyToZYX(y)

Rot0
native: 10.34ms
native: 311.87ms (bad layout)

MirrorY
native: 10.62ms
native: 311.45ms (bad layout)

Rot90
native: 44.80ms
native: 157.16ms (bad layout)

Sparse address outer loop, dense address inner loop
Takeaways for engine developers

- Various abstraction mechanisms have different costs
- **Inlining** is crucial!
- Virtual calls are super-fast but not free (by far)
  - If they lead to cache misses, they might kill your performance
- by-value/by-ref passing matters in C++
  - Struct/Class distinction matters in C#
- JIT optimizations can help for cross module optimizations
- For image operations, C/Native memory implementation pays out
- Beware of bad loop nesting structure: **Cache efficiency**
- **Benchmark** your stuff in **real-world scenarios**
  - e.g. in our tests everything was in cache all the time!
Allocating memory

- Different mechanisms for memory management
  - Explicit allocation (malloc, free, new, delete)
  - Reference counting
  - Garbage collection

- Each mechanism has pros/cons
- For efficiency, engine developers need to really understand the memory management mechanism
- Especially in garbage collected environments the conceptual working of the GC is inevitable in order to write efficient programs!
Reminder: How GC works on a very basic level...

GC Roots:

Threads
  Stack frames
    stack variables
    stack variables
    ...
  Stack frames
    stack variables
    stack variables
    ...

Global variables

GC collects working set into new heap

"Old" Heap:
  working set

"new" Heap:
  working set

GC roots determine working set via traversal

GC does not collect garbage!

Please note that this is just the basic working. Real garbage collectors are significantly more complex!
Generational GC, basic working

- GC evacuates the working set
  - Evacuation can improve cache locality!
  - GC removes fragmentation!
  - GC can lead to unpredictable performance

- Main idea of generational GC: many objects die young

- Design
  - heap regions for different object ages (Generation 0: young objects, Generation 2: old objects)
  - Keep gen 0 small by scanning gen 0 frequently, old objects are expected to live longer and get promoted to next generation (thus checked less frequently now)
Malloc internal implementation

- OS provides two (syscall) mechanisms for allocating virtual memory
  - `sbrk` increases/shrinks process memory
  - `mmap` maps file into process memory
    - Used for memory mapped files (e.g. high-performance loading of data-structures)
    - Can also be used to assign non file-mapped Memory to process.
    - Can be used to allocate executable memory
- OS calls work on virtual memory only (not visible in physical process memory)
- C/C++ lib then assigns this memory via malloc/new
  - If memory manager runs out of virtual memory it uses sbrk or nmap
  - Bucket list/Range tree, different implementation techniques possible
  - Fragmentation possible, searching for free spot can be inefficient
- When using vulkan, you can inspire yourself by looking at memory managers
Explicit allocations vs GC

- Allocation is **expensive** in both, in C++ world and GC world
- In managed languages (with GC), allocation is **simply pointer increment** + out of memory check (memory is given out alloc by alloc)

- Explicit memory management suffers from **fragmentation**
  - No way to get around this
- Different **GC** implementations evacuate memory which works as automatic **defragmentation** technique
- Subsequent allocations in CLR are neighbours in memory
  - Cache coherence
Takeaways for engine developers

- No matter what technique, hot code should not allocate memory.
- Each method has its tradeoffs
- Recall the implementation of GC when talking about efficiency
Low level memory management in rendering engines

- Aside from CPU memory, we need to manage **GPU memory**
- Each mechanism makes sense
- When implementing memory managers, evaluate each mechanism for your requirements
  - e.g. if we cannot afford using too much memory for a short period of time, GC techniques cannot be used
  - In presence of culling, GC makes sense, since explicit allocation/deallocation is costly
More considerations for efficient code

- Today's CPUs are so called **superscalar architectures**
  - Multiple instructions are executed simultaneously
- CPU instructions are executed **out of order**
- Program blocks, which can be executed in parallel do so, but
  - Registers are renamed using **register renaming**
- For conditional jumps, **branch prediction** allows to speculatively execute the more likely branch.
- **Branch target address cache** uses hash value of jump instruction in order to estimate indirect jumps
  - In order not to stall the pipeline, i.e. the CPU pipeline continues at the jump target position.
- Result: code which looks inefficient might be efficient
- Those properties are especially interesting when implementing **virtual machines for rendering engines** -> next lecture
Further reading

