Game Technology

Lecture 2 – 24.10.2015
Timing & Basic Game Mechanics

Dr.-Ing. Florian Mehm

Prof. Dr.-Ing. Ralf Steinmetz
KOM - Multimedia Communications Lab

Overview

Timing

- Different timing options
- Animations

Basic Game Mechanics

- Game Loop
- Multithreading
- Collision

C++

- Memory management
- Strings

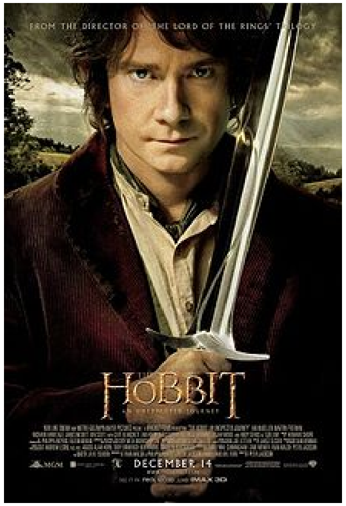
Timing

Monitors commonly run at 60 Hz

- Games should provide a new frame every ~16 ms
- Movies (used to) operate at 24 Hz (40 ms)

Why work harder than that?

- Some people have been shown to be able to distinguish up to 90 Hz images
- The frame rate determines how fast the game can react
  - Gamers want speed!
- Virtual Reality
  - HTC Vive: 90 Hz
  - Oculus Rift: 90 Hz



Timing

„"At Ubisoft for a long time we wanted to push 60 fps. I don't think it was a good idea because you don't gain that much from 60 fps and it doesn't look like the real thing. It's a bit like The Hobbit movie, it looked really weird.” Nicolas Guérin, World Level Design Director, Assassin’s Creed Unity http://www.techradar.com/news/gaming/viva-la-resoluci-n-assassin-s-creed-dev-thinks-industry-is-dropping-60-fps-standard-1268241

See also “black bars” discussion, e.g. around The Order 1886

Motion Blur

In a real camera, the filmed objects change during a frame

The movements are blurred

- Fast moving objects more
- More the longer the exposure time is

Source: Wikipedia

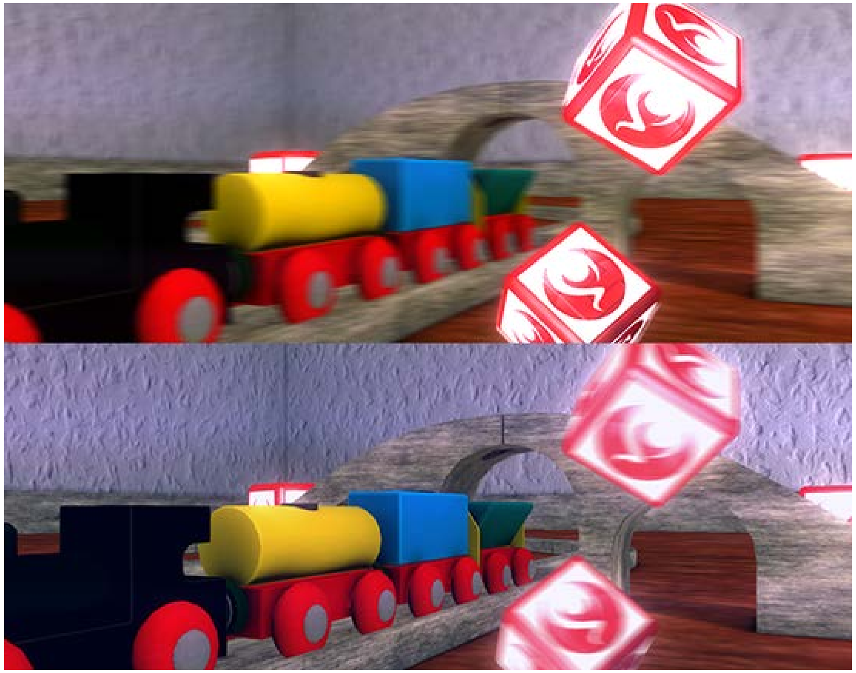
In a virtual camera, without additional measures, no blurring is present

- All objects rendered at a perfect instant in time
- Similar to the missing depth of field

Motion Blur algorithm example

Directional blur along a pixel's velocity

Introduces artifacts for fast-moving objects



Source: http://docs.unity3d.com/Manual/script-CameraMotionBlur.html

Multithreading

Cooperative Multithreading

- Often used in games

Returning

- Every (game) object is called
- Carries out its calculations…
- …and returns, saving its state

+ Synchronization easier to handle

- Can't use multiple CPU cores

Preemptive Multithreading

- Used in current operating systems

Returning

- Every process is called
- The scheduler takes control back
- State is saved for the process

+ Stalled threads don't stall the whole system

- Needs proper synchronization

- Additional costs (saving all state)

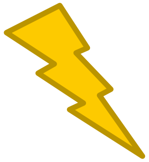
Used for whole systems (e.g. physics)

Multithreading

Cooperative Multithreading

```
while (true)
{
  DoSomething();
  yield(); // Explicitly return control
  DoAnotherThing();
}
```

```
while (true)
{
  DoSomething();
  DoAnotherThing();
}
```



Preemptive Multithreading

```
while (true)
{
  // Might be preempted here...
  DoSomething();
  // ...or here...
  DoAnotherThing();
}
```

Multithreading Problems

TECHNISCHE UNIVERSITÄT DARMSTADT

Communication between threads

Critical Sections

```
int a = 5;
if (a == 10)
{
  // Will never happen...
  print("Boo!");
}
```

Multithreading - Uses in Games

Cooperative Multithreading

- E.g. Unity‘s coroutines
- Simple enough to use without preemptive problems, but powerful enough for many purposes

Preemptive Multithreading

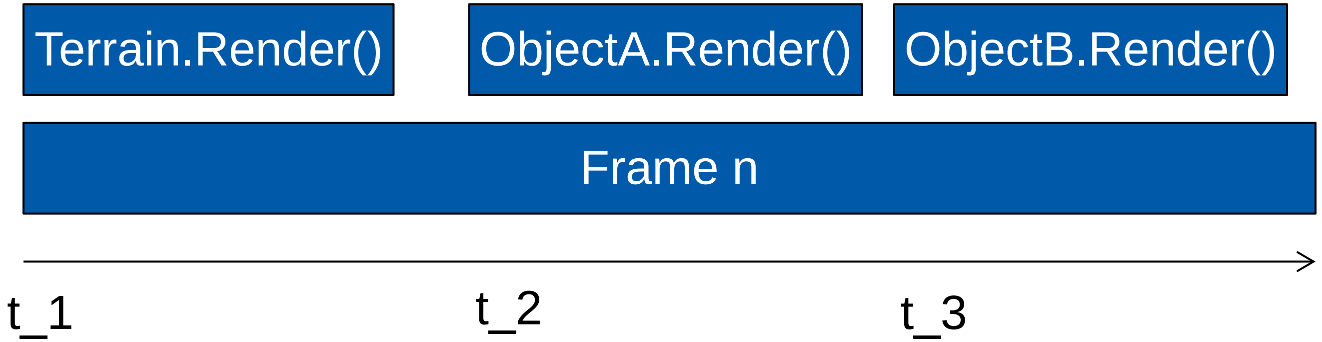
- Most often for larger systems
- For systems which take longer than a frame to compute results, e.g. AI queries
- For systems that run all the time, e.g. physics
- Can make use of multicore systems

Massively parallel execution

- General purpose computation on GPU, Compute Shaders
- Will handle this variant when we look at hardware rendering

Timing

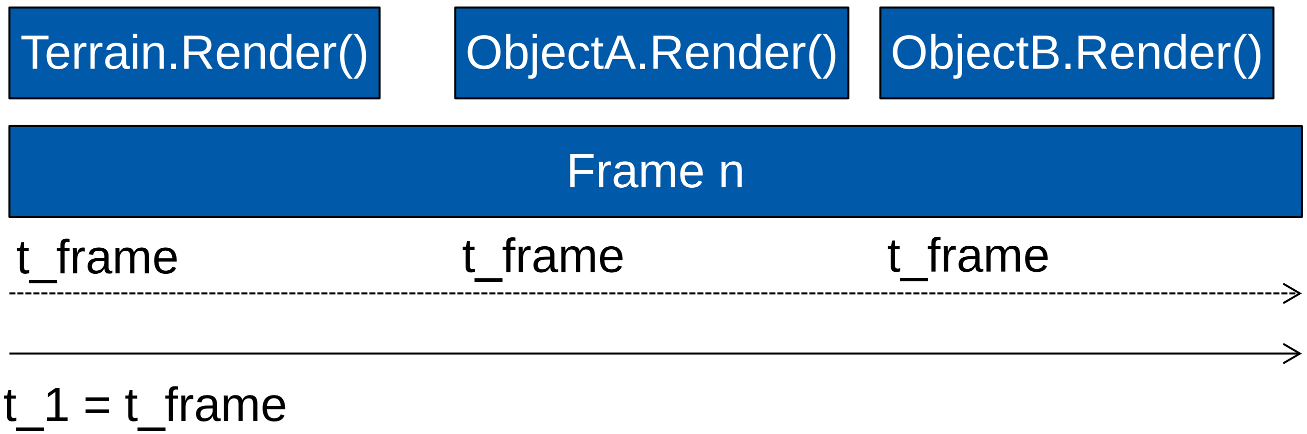
Which time to use?



Hardware timers vs. very coarse timers

Virtual frame time

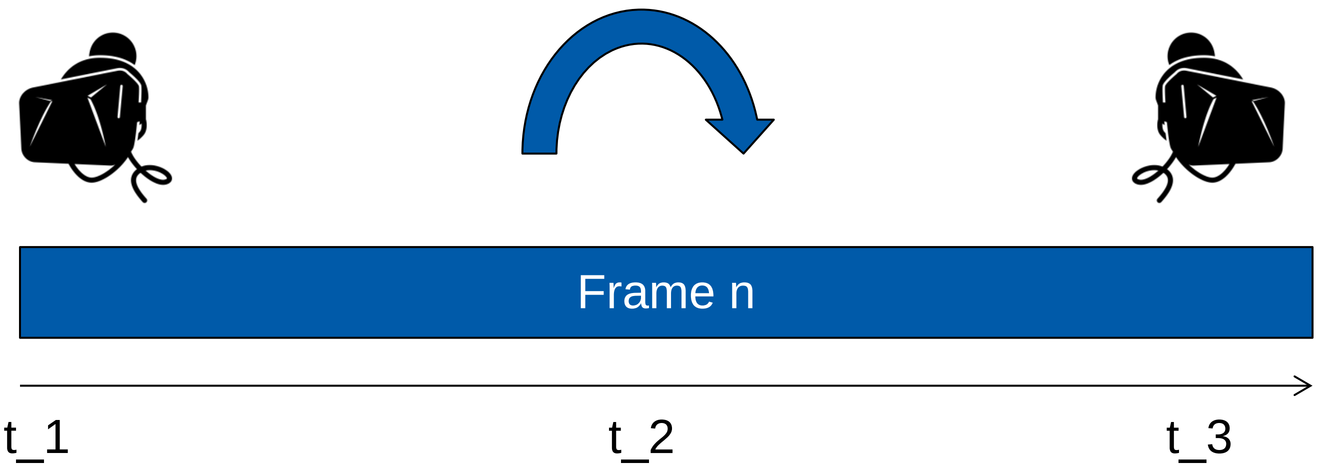
Calculate a time that is used throughout the frame



Further advantage: Can scale/pause this time

Virtual Reality Frame Time

Which head position to use?



Future positions often predicted by HMD

- E.g. using the measured acceleration, physiological models
- Can use timewarp mechanism → will look at this in a later lecture

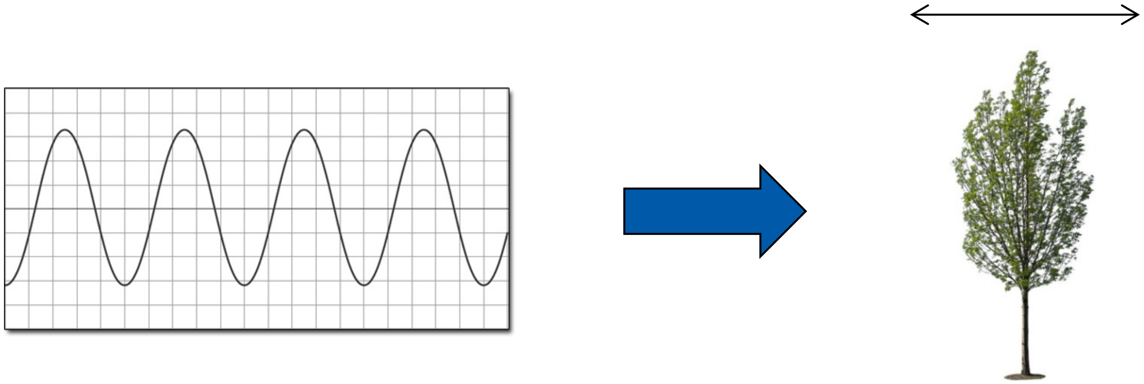
Icon by Hans Gerhard Meier, Noun Project

Procedural Animations

Calculate the state without information about the previous state

- Based solely on parameters
  - Current time
  - Configuration parameters
- Usually ranged [0-1]; later scaled to correct amount
  - Allows adding/multiplying using sine/exp/…

Example: Simple wind animation of trees

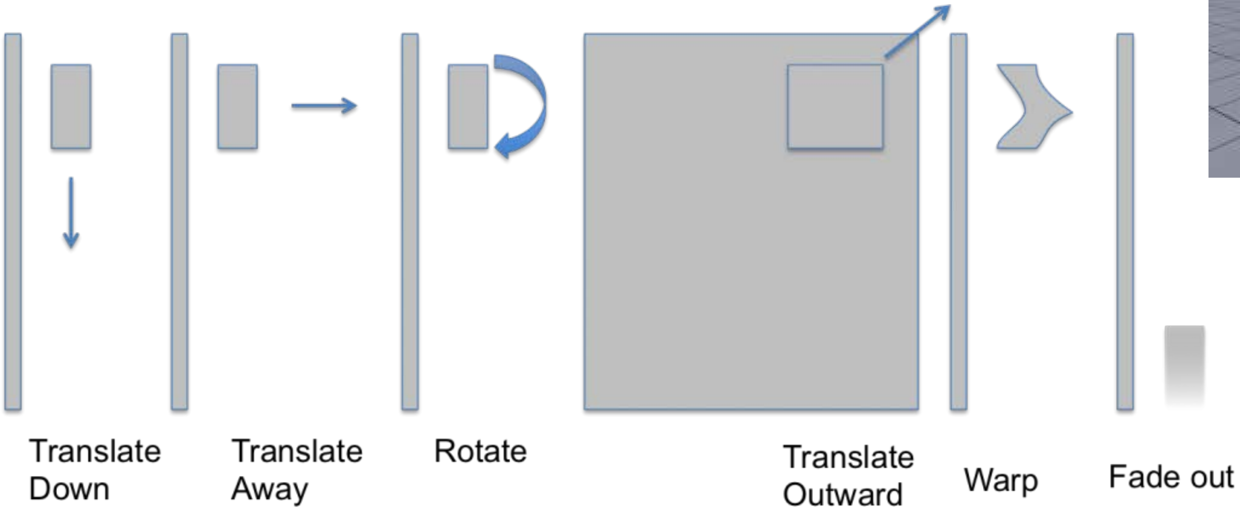


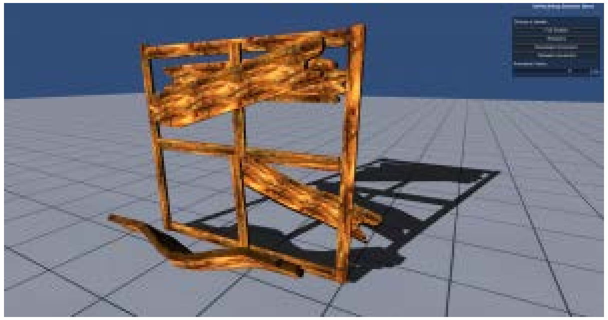
Procedural Animation Example

Original Source: "The Inner Workings of Fortnite's Shader Based Procedural Animation" (Jonathan Lindquist, Epic, GDC Talk)

Effect for "self-building structures"

Composed of several components



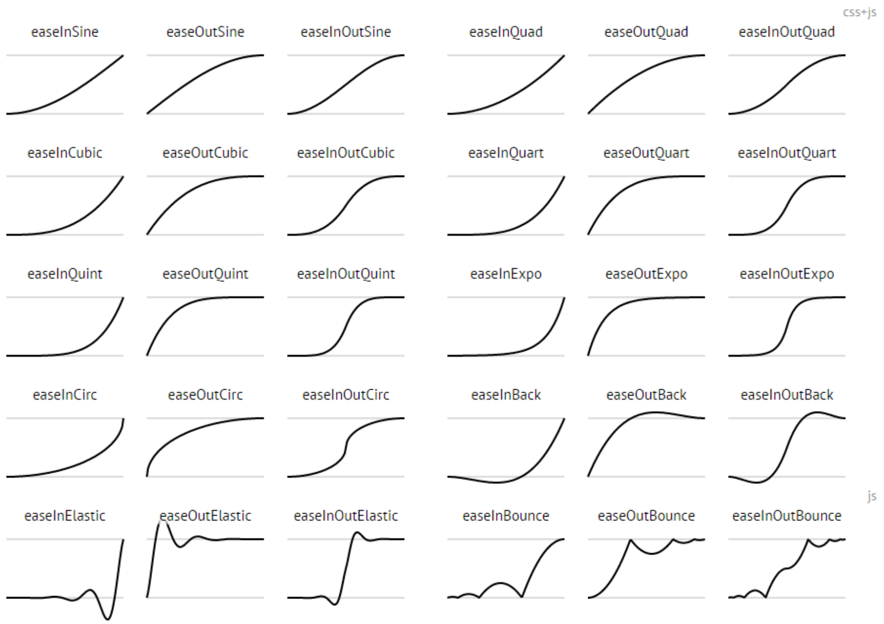


See implementation at http://mehm.net/blog/?p=1278

Procedural Animation „Mindset“

Think in procedural terms

- Input: t, e.g. in [0, 1]
- Output: Animated value f(t)



http://easings.net/

Combination of several effects

- $f(t) = g(t) * h(t)$
- …

Stretching of input parameters

- E.g. for easing

Later in shaders

- Think of equivalence to gradients

Iterative Animations

Calculated based on previous states

- Usually not from the beginning of the game
- Instead, use a window of the last frames or a running average
- Often combined with user input
- Used for animations where a “closed” form is not possible or too complicated

Example: Physical animation

- Very simple: Take the position and velocity of the last frame
- Calculate a velocity for the current frame
- Add the velocity to the object

Game Loop

Set up windowing system, OS callbacks, initialize libraries/devices, …

Do

- Read data from input devices
- Calculate new game state
- Render frame
- (Wait for Vsync)

While the game is active

Unload libraries, free memory, close window, …

Hidden Game Loop

Unity

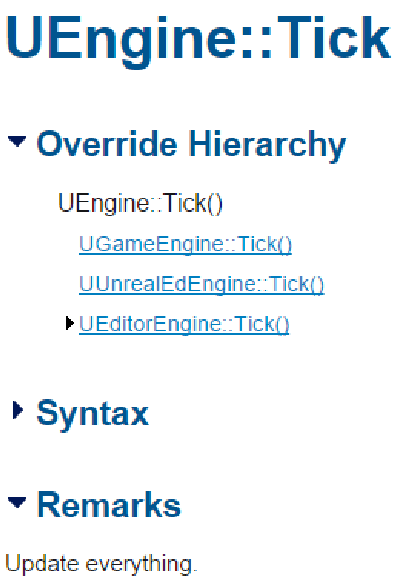
- Actual game loop implemented in C++
- Components provided by programmers compiled to .net (C#, JS, Boo)
- Update()-functions on all active components are run

Unreal Engine

- Found in UEngine::Tick()
- Scripts provided by users can also be Blueprint

Engine core ←→ Scripts and components

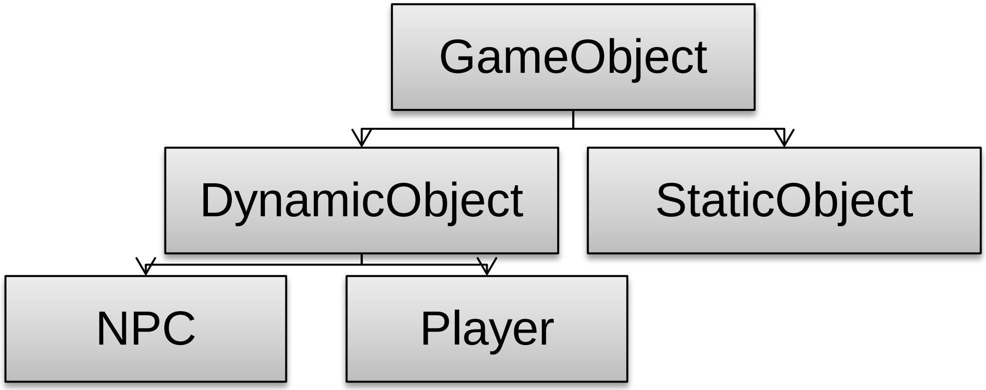
- Performance optimizations by the engine provider
- Easier to handle for programmers
- But less adaptable and transparent (→ Unity)



Game State

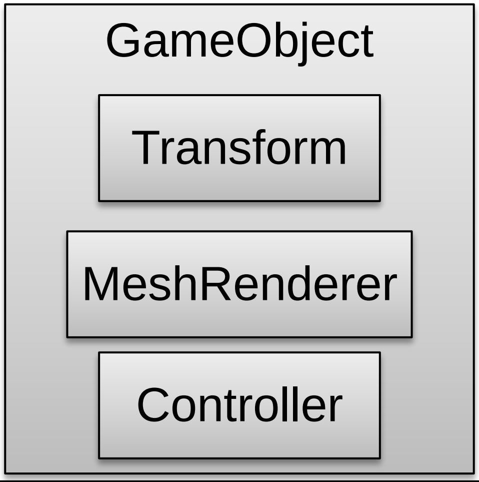
Usually handled as Game Object (or similar construct)

- Saves all relevant game state
- Handles relevant input
- Updates state each frame



Component-Based Game Objects

- Separate component for different tasks
  - Rendering
  - Position
  - Input handling
  - …
- Avoid object-oriented hierarchies



Game Objects

Hierarchies

+ More behavior defined at compile-time

+ Explicit inter-object communication

- (If not restricted): Diamond-shaped hierarchies/multiple parents

- Resistant to change

Components

+ Re-usable behaviors

+ Combinable at runtime

+ Specialization, encapsulation

- Inter-component communication

- E.g. Unity: Performance hits if other components are searched each frame

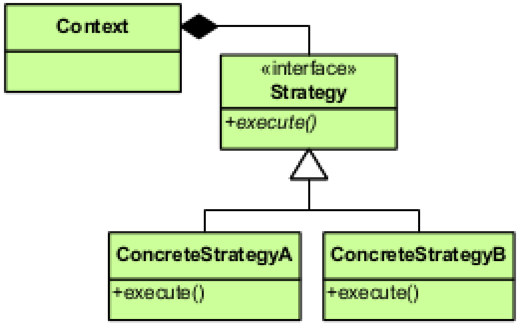
Negative Example

- „Weapons“ and „Tools“
- GDD defines a new weapon which partly acts like a tool

Best of both worlds

Class hierarchy

- Use power of virtual classes, polymorphism
- Model objects as well as needed



Encapsulate differences with strategy pattern

- Owning class handles everything that is shared between all strategies
- Defers to individual strategies for differing behaviours

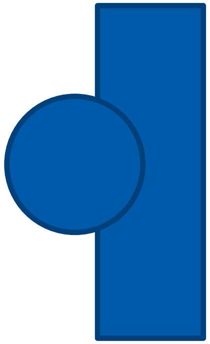
Example: Buildings in a RTS-game

- Encapsulating class handles mesh loading, animations, …
- Strategies to handle different behaviours (produced units, …)

Collisions

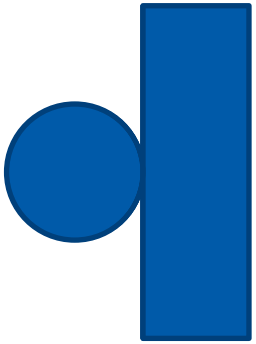
Intersection

- Objects are overlapping each other
- In reality, objects would deform/break/...
- →Unwanted state



Collision

- Objects ideally have only one contact point/edge/face
- Calculate collision response based on this state



Collision Response

- Separate bodies or
- (Stable) contact

Collisions

x times per second

{

For each object

{

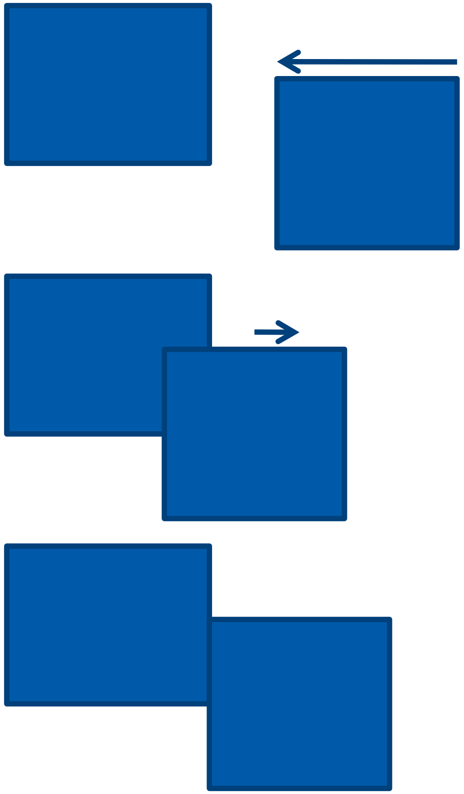
Move object

Check for collisions

If (collision detected) move back

}

}



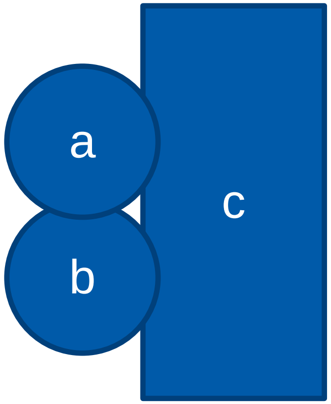
Collisions and Timing

Exact collision will almost never happen

- Due to floating point issues and discrete frame time
- Different coping strategies
  - Ignore/Keep pushing objects out of each other
    - (Smaller time steps)
  - Find the exact time when collision happened and step to this time

Collision response for multiple objects

- Often resolved one after the other
  - E.g. resolve b-c, then a-c, then a-b
- But in reality, solved all at once



Game logic timing

Separate from actual frame rate

- Keep timer for game logic
- Update in periodic time steps
- Rendering done at frame rate

Otherwise, dependent on performance of the hardware

Source: http://telkomgaming.co.za/old-versus-new-remembering-the-turbo-button/

Summary

Timing

- Use a virtual time throughout the frame
- Use smaller ticks for systems such as physics
- Motion Blur
- Multithreading

Animations

- Procedural
- Iterative

Game Loop

- Game state
- Collision detection

Memory Management

Static Memory

- Global variables
- Handled by the compiler, allocated and de-allocated automatically

Stack Memory

- Semi-automatically handled by the compiler
- Function parameters, local variables, implicit data (e.g. return addresses)

Heap Memory

- All manually allocated memory

Heap Memory

TECHNISCHE UNIVERSITÄT DARMSTADT

Allocated dynamically

- C++ handles nothing for us -> requests memory from the OS
- Can be VERY slow and unreliable

Difference to Java

- Java allocates a large block of memory at the beginning
- Allocates memory to the program during runtime
- Manages this memory
- → Can still be slow, e.g. if physical RAM is exhausted
- Garbage Collection

Custom memory management

- Utilize memory access patterns to optimize
- Avoid allocating heap memory altogether in critical sections

Heap Memory Examples

TECHNISCHE UNIVERSITÄT DARMSTADT

Managing your own memory for often-used structures

Example: Allocate enough memory for all game objects of one type

- Find typical numbers by testing or analysis
- Manage the block by yourself

Stack vs Pool-based

- Stack: Allocating and freeing using one pointer
- Pool: Manage list of free blocks

Keeps data local

- Can be better for cache efficiency

Effects of cache performance

Source: „Systems Performance: Enterprise and the Cloud”, Brendan Gregg

Table 2.2 Example Time Scale of System Latencies

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 μs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

Pointers (Example: Integer value)

TECHNISCHE UNIVERSITÄT DARMSTADT

Variable on the stack

- int foo;

Variable on the heap

- int* foo;

Passing by value (using the stack)

- void bar_val(int a, int b) { }
- Values/objects copied onto the stack

Passing by reference (using the heap)

- void bar_ref(int* a, int* b) { }
- Only a pointer copied (32/64 bits)
- Makes it possible to pass back values

Getting addresses and dereferencing pointers

TECHNISCHE UNIVERSITÄT DARMSTADT

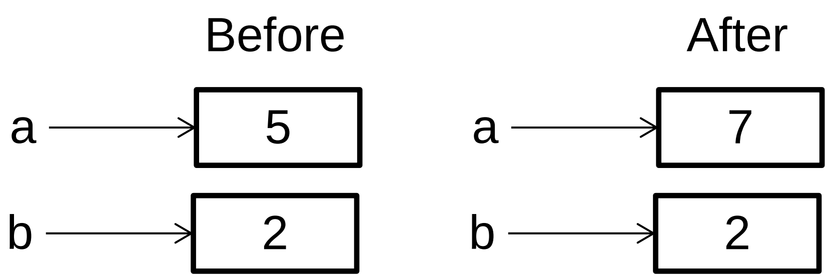
Getting the pointer to a variable

- `int a = 3;`
- `int b = 4;`
- `bar_ref(&a, &b);`

Warning: Don't take the address of a local variable and pass unless you know what you are doing → the callee might save it until it is invalid!

Dereferencing a pointer (getting to the actual value)

```
void bar_ref(int* a, int* b)
{
  *a = *a + *b;
}
```



Arrays

TECHNISCHE UNIVERSITÄT DARMSTADT

Allocated on the stack

- `int array[3];`

Array on the heap

- `int* array = new int[3];`

Deallocate using operator delete[]

- `delete[] array;`

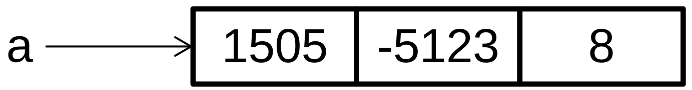
Mixing up leads to undefined behaviour

- (Also important for calling destructors)

Referencing arrays

Referenced using their first element

- int array[3];
- int *a = &array;
  - a points to the first element of array

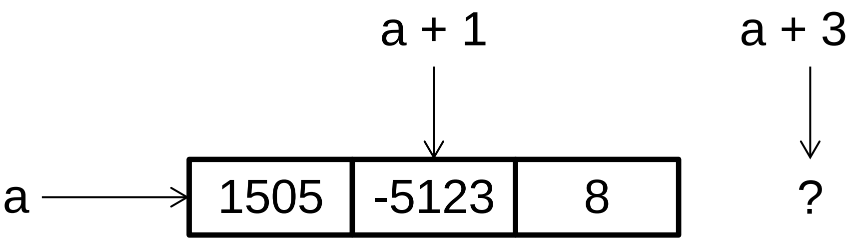


Also legal

- bar_ref(&array, &array);

Pointer arithmetics

- Pointers behave like ints
  - Addition, Subtraction, …
- Evil to operate outside the allocated memory of the array
  - No bounds checking



Strings

TECHNISCHE UNIVERSITÄT DARMSTADT

Strings are just arrays of chars

- char* f = “foobar”;

f ⟶

f	o	o	b	a	r	0

“foobar” is a 7-element array

- Zero-terminated
- Allows measuring the size in O(n) time

Encoding

- On all common systems, **sizeof(**char**)** is 8 bits
- char* can be an UTF8 string
  - every ANSI string is also a proper utf8 string
- Commonly used chars encoded in 8 bits
  - Uncommon/other languages in several 8-bit blocks
- Best practice: Use UTF8 even on systems that natively have other representations

Example UTF8 vs. UTF 16

„a“

- ANSI: 61 (Hexadecimal)
- UTF 8: 61
- UTF 16: 00 61

„ä“

- ANSI: E4
- UTF 8: C3 A4
- UTF 16: 00 E4

STL (Standard Template Library)

TECHNISCHE UNIVERSITÄT DARMSTADT

Offers template-based generic solutions for dynamic memory

Arrays: std::vector

- Adaptive size
- → Can’t keep addresses to elements in the vector, as they might be invalid upon a change in size

Strings: std::string

- Implemented as a std::vector for chars
- Comfortable functions (trim, concatenate, operator+, …)

Game studios tend to avoid these libraries

- Template overhead
- Unpredictable behaviour

STL Complexity Guarantees

Container	Insertion	Access	Erase	Find	Persistent Iterators
vector / string	Back: O(1) or O(n) Other: O(n)	O(1)	Back: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)	No
deque	Back/Front: O(1) Other: O(n)	O(1)	Back/Front: O(1) Other: O(n)	Sorted: O(log n) Other: O(n)	Pointers only
list / forward_list	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	Back/Front: O(1) With iterator: O(1) Index: O(n)	O(n)	Yes
set / map	O(log n)	-	O(log n)	O(log n)	Yes
unordered_set / unordered_map	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	O(1) or O(n)	Pointers only
priority_queue	O(log n)	O(1)	O(log n)	-	-

Source: http://john-ahlgren.blogspot.de/2013/10/stl-container-performance.html

Summary

Static, Stack and Heap Memory

- Different allocation schemes
- Different level of control for the programmer
- Choose which is the most useful

Pointers

- Allocation on the heap
- Pass by value vs. Pass by reference

Arrays

- Allocation on the heap
- Referenced by pointer to first element

Strings

- Arrays of chars
- Pointer arithmetic
- UTF8 vs. UTF 16

Side Note: Cracktros

Cracktro

Intro for a cracked game

Used to show off to other programmers, cracker groups, …

Often more impressive than the original game‘s graphics

Later split into the demo scene

Cracktro -> Demoscene

Program impressive demos and compete outside of the warez scene

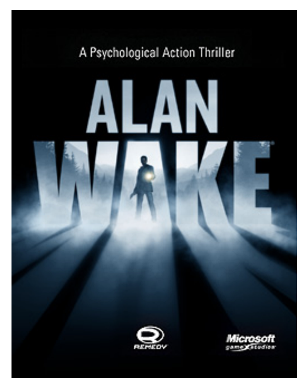
Always at the cutting edge of the hardware

- Use Assembler instead of Basic
- Find ways to exploit the hardware
- Later: Self-restricted demos (e.g. 64K demos)

Demoscene -> Game industry

- E.g. Future Crew -> Remedy

1988



2010

REMEDY

Classical demo techniques

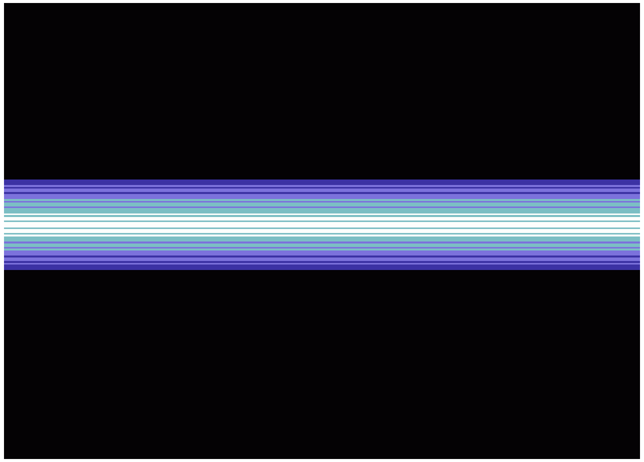
Scrolling

Moving along a sine wave

- Note: Often used a sine table for efficient computation
- Offset from other characters
- Different amplitudes
- …

Rasterbars

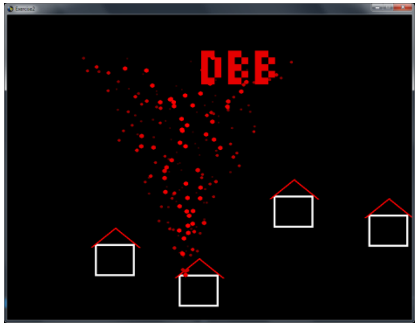
- Use an interrupt to paint lines
- Moving rasterbars along sine wave…



Good example for procedural animation

- Often impossible to store all (animation) data
- Instead, generate complex paths from simple inputs
- Simplest example: Text moving on a sine wave
- Procedural Content Generation
  - See video of .kkrieger

Examples from last year



Book Recommendations

Game Engine

„Game Engine Architecture“
Jason Gregory (Lead Programmer at Naughty Dog)

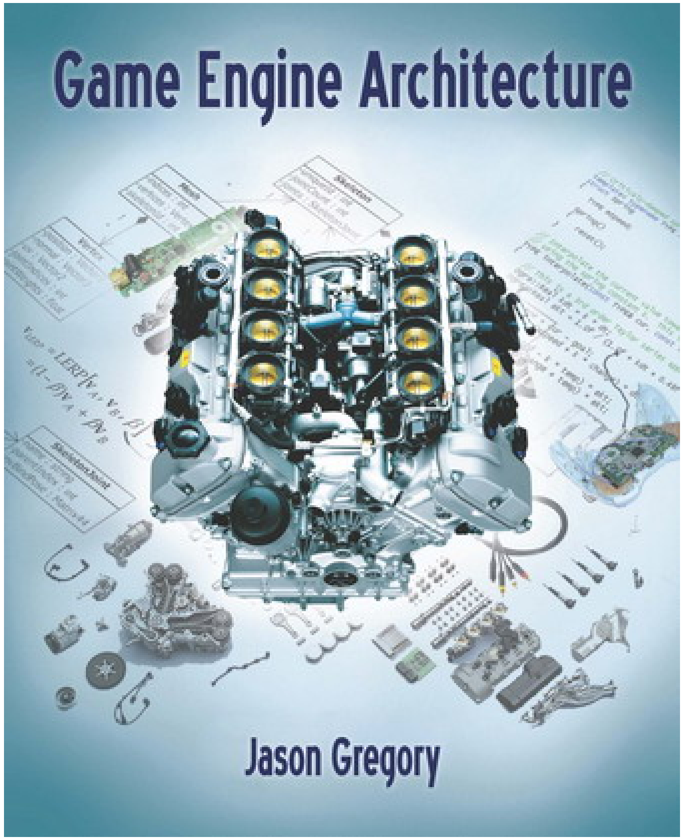
Fundamentals

- C++
- 3D Math
- Graphics, …

Practical Examples

Part of the „Semesterapparat“

- Fachlesesaal MINT in der ULB Stadtmitte, 4. Obergeschoss
- Lernzentrum Informatik



Book Recommendations

C++

„Effective C++“

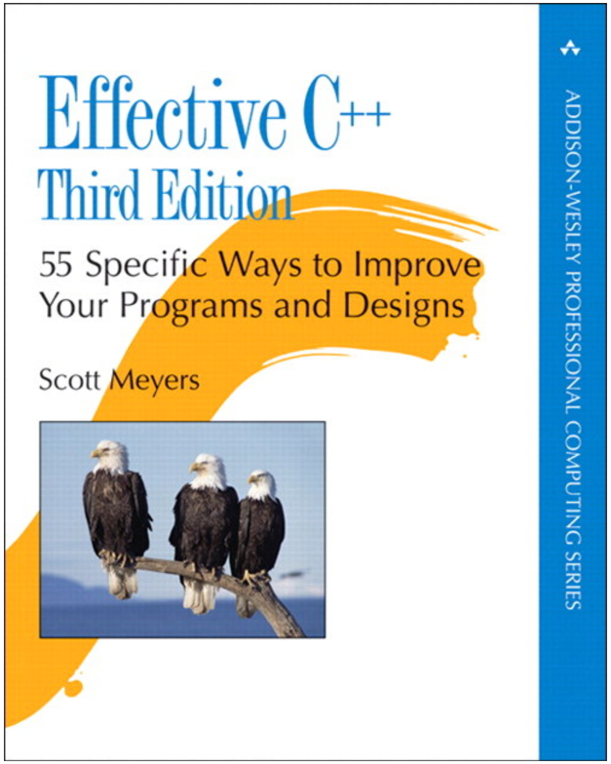
Scott Myers

Performance tips

Pitfalls/Language Details

- Functions a compiler silently adds to classes
- Good use of const, pointers, references

Performance Considerations



Book Recommendations

TECHNISCHE UNIVERSITÄT DARMSTADT

3D Graphics (next lectures)

„Real-time Rendering“

Tomas Akenine-Möller, Eric Haines

Very detailed look at graphics algorithms

Also includes further information, e.g. intersection tests and collision detection