Game Technology

Lecture 2 – 24.10.2017
Timing & Basic Game Mechanics

Dipl. Inform. Robert Konrad
Polona Caserman, M.Sc.

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

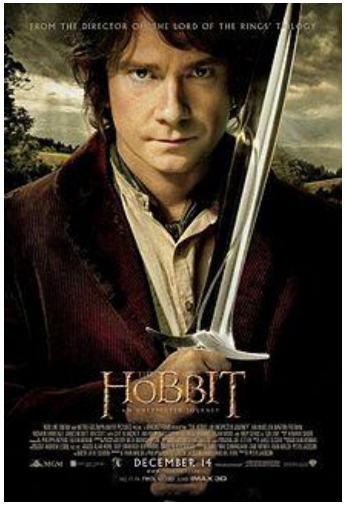
Timing

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?

- The frame rate determines how fast the game can react
- Virtual Reality
  - HTC Vive: 90 Hz
  - Oculus Rift: 90 Hz



The Hobbit, 2014
Filmed at 48 fps

Lag

Non-instantaneous reaction of the game to user input

- Controller → gaming machine (e.g. wireless)
- Computing reaction
- Rendering a frame
- Showing the frame
- Networked multiplayer: Network delay

https://www.youtube.com/watch?v=_fNp37zFn9Q

Lag

Human reaction time to

- light: As low as 190 ms
- auditory stimuli: As low as 160 ms
- But: includes time to decide and activate muscles

Impact depends on game

- Strategy-games, MMORPG: Higher lag acceptable
- First-person shooter, rhythm games, …: Lag as low as possible

VR

- Very important for immersion and comfort
- 50 ms responsive, but lagging
- 20 ms mostly unnoticeable

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

Easy Motion Blur algorithm example

Subdivide time, render multiple frames for each monitor frame

Send averaged frame to the monitor

“Temporal Anti-Aliasing”

Source: http://wallup.net/vehicle-car-wheels-formula-1-motion-blur-road-circuits-blurred-ferrari-f1/

Vertical Sync

Monitors typically operate at framerates of 60 Hz

Picture is transferred during a designated timeslot (vblank)

Signaled by vsync event

Game has to wait for that timeslot after image calculations are done, or else…

- Tearing
- Display of different images intermixed

Buffering

TECHNISCHE UNIVERSITÄT DARMSTADT

General premise

- Keep the memory area of the screen untouched while the image is read
- Ideal solution: Always have the image ready while it is being read
- Hard to achieve in complex games
  - Requires predictable performance

Double Buffering

- Render image to off-screen buffer
- Wait for vblank signal
- Change active buffer
  - Change pointer to active memory (page flipping)
  - Copy to another memory region
- Repeat

Triple Buffering

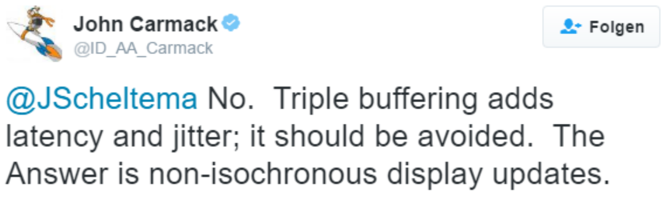
TECHNISCHE UNIVERSITÄT DARMSTADT

Additional buffer to avoid waiting time

- With Double Buffering, we have to wait for vsync until we can continue drawing
- In the worst case, CPU & GPU stalled when we could do other calculations

\+ With three buffers, one can always be free for writing

- Frames can be skipped
- Frametime becomes hard to predict
- More memory required
- More latency



Depends on implementation

- Swap-chains
  - Keep a linked list of buffers that loops around
  - On each vblank, continue to next buffer, never skip buffers
- Dropping buffers
  - On vblank, choose the buffer that has the most recently finished image in it

General framerate considerations

Steady framerate is the most important goal

- In a game played on a 2D monitor → stutters
- In a game played in VR → users throw up

Careful with peaks

- Stall because of loading data
  → Load everything at level load
  → Load async
- Several objects are updated in the same frame
  → Load balancing, spread out over several frames

The new thing

G-Sync (nVidia)

Freesync (AMD)

Dynamic monitor framerate

- Send picture -> monitor updates “shortly” after

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

- Prevalent in the Pre-Pentium times
- E.g. Wing Commander

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

50/60 Hz versions

TV Standards

- Japan & US: NTSC – 60Hz
- Europe: PAL – 50 Hz

If games were not coded with this in mind

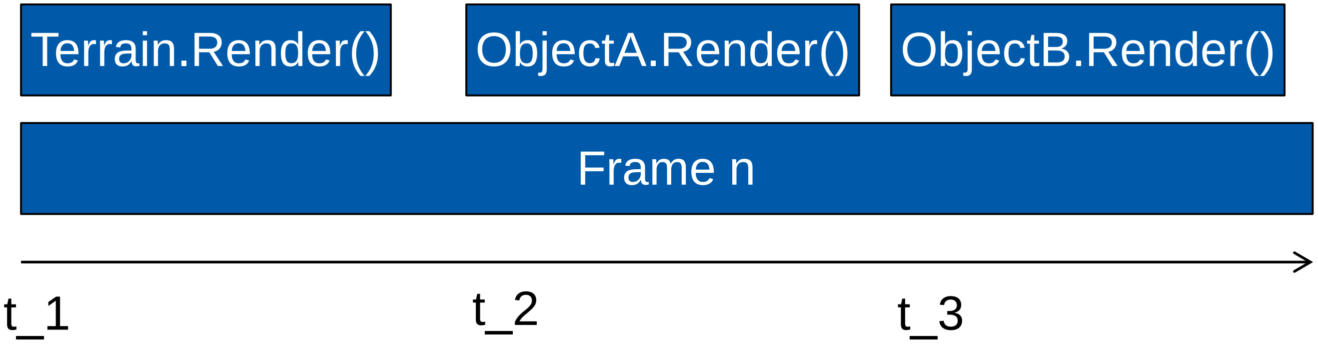
- Gameplay depends on refresh rate
- Sound speed depends on refresh rate

https://www.youtube.com/watch?v=WEuiBmCloas

Timing

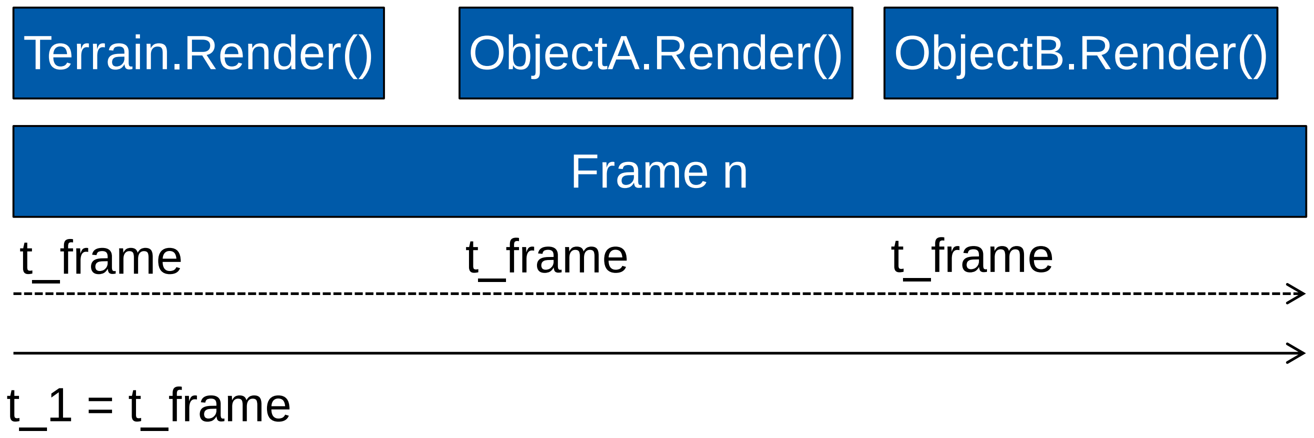
Which time to use?

- Especially problematic if objects query the time, e.g. for simulation of motion



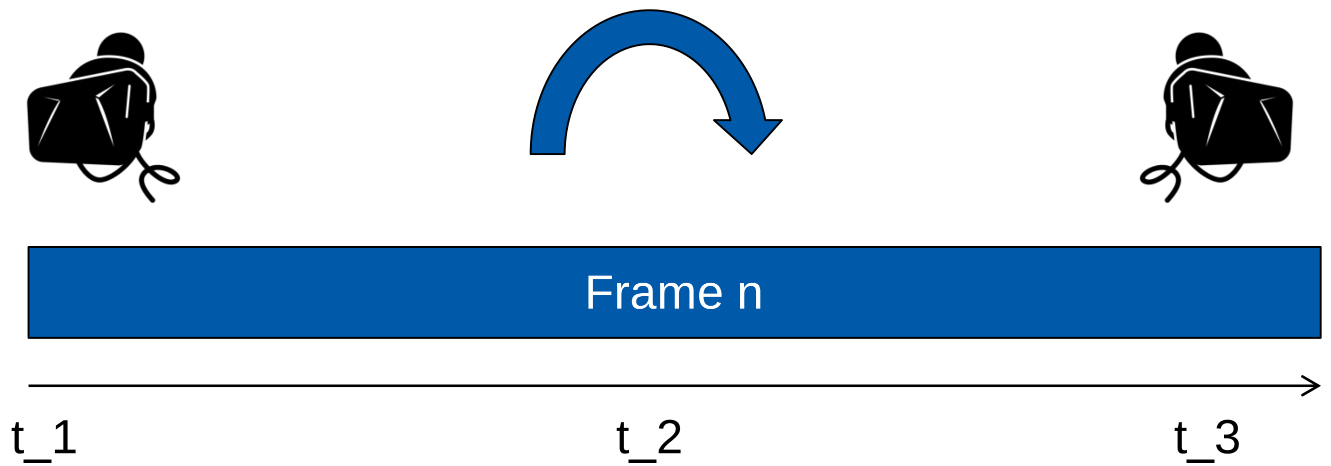
Virtual frame time

Calculate a time that is used throughout the frame



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

What is the frame time?

Ideally - the time when the next frame is displayed to the user

Can only be guessed if performance is unpredictable

- Typically the average of last n frame times

High framerates make the problem more difficult

Triple buffering makes the problem more difficult

G-Sync/Freesync makes the problem more difficult

Debugging Timing Issues

Create a simple test scene

- For example move a box with constant speed

Use a HDMI capture device

- Can usually capture at static 60fps

Use your mobile phone

- More often than not includes a high framerate mode
- Can be used to debug G-Sync/freesync

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)

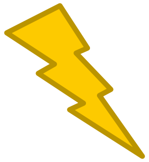
Multithreading

TECHNISCHE UNIVERSITÄT DARMSTADT

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...
  // ...or inside the function...
  DoAnotherThing();
}
```

Multithreading Problems

TECHNISCHE UNIVERSITÄT DARMSTADT

Communication between threads

Critical Sections

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

Second thread

```
int b = 5;
a = b + 5
```

Multithreading - Uses in Games

Cooperative Multithreading

- E.g. coroutines in some languages
- Simple enough to use without preemptive problems, but powerful enough for many purposes

Preemptive Multithreading

- Most often for larger systems – seldomly in gameplay code
- 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

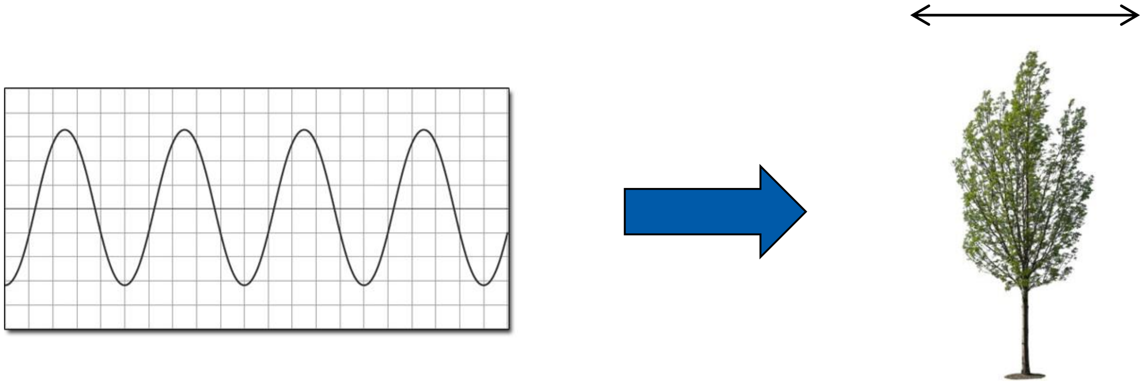
- General purpose computation on GPU, Compute Shaders

Functional Animations

Calculate the state without information about the previous state

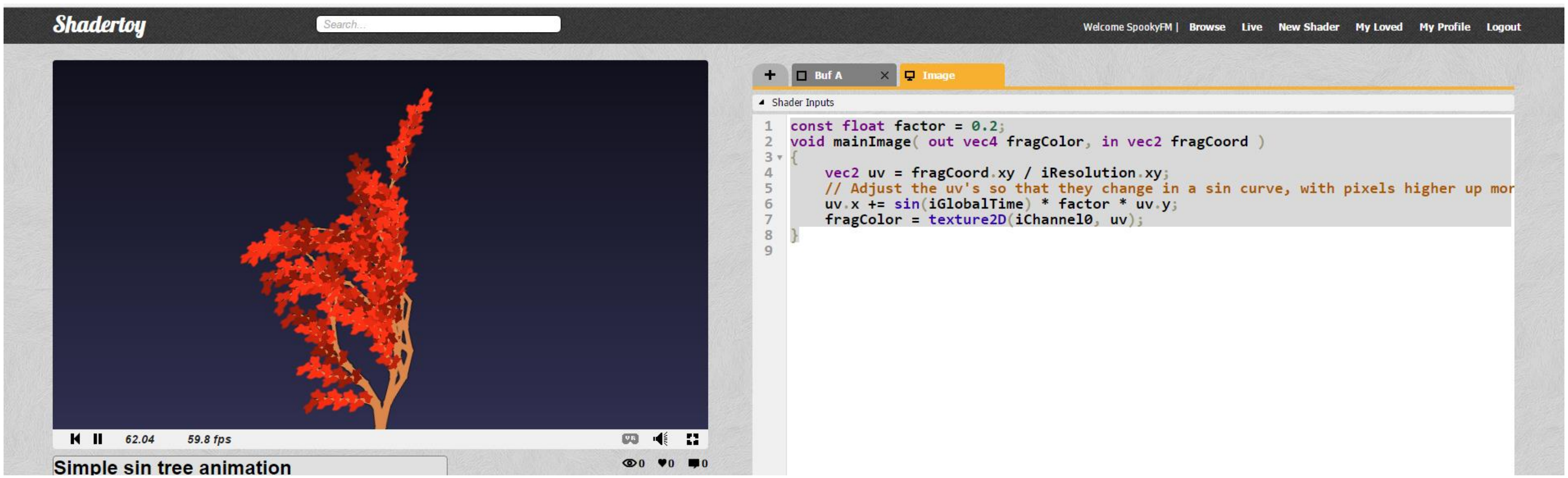
- Based solely on parameters
  - Current time
  - Configuration parameters
- Usually ranged [0-1]; later scaled to correct amount
  - Allows adding/multiplying using sine/exp/…

Example: Simple wind animation of trees



Functional Animation Example

Live at https://www.shadertoy.com/view/MtGGWG



Functional Animation „Mindset“

Think in procedural terms

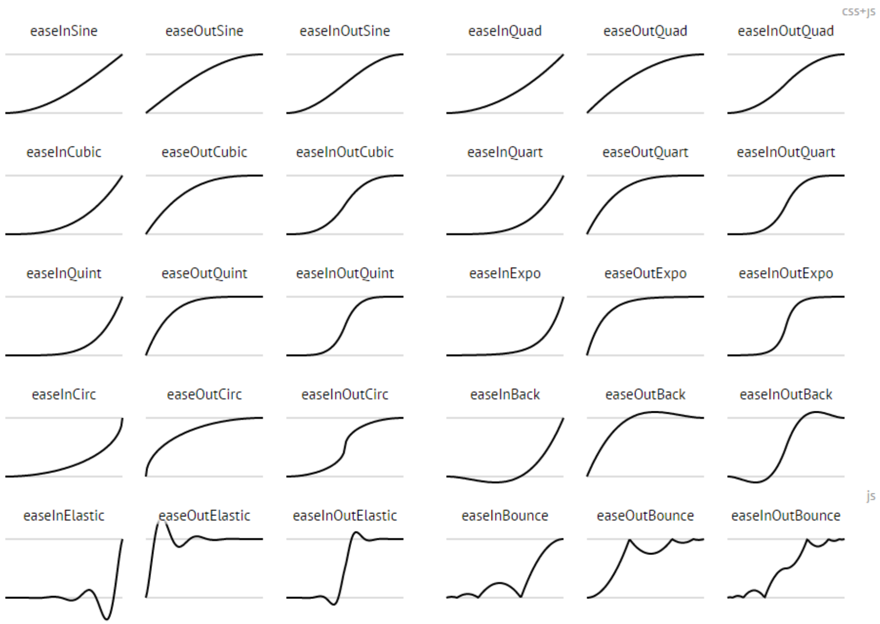
- Input: t, e.g. in [0, 1]
- Output: Animated value f(t)

Combination of several effects

- f(t) = g(t) * h(t)
- …

Stretching of input parameters

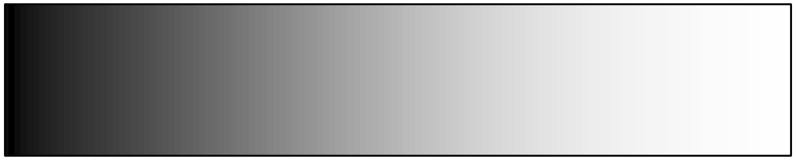
- E.g. for easing



http://easings.net/

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
- Get the new position from the old position + current speed

Game Loop

Fundamentally iterative

Typically no continuous simulation

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

Do

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

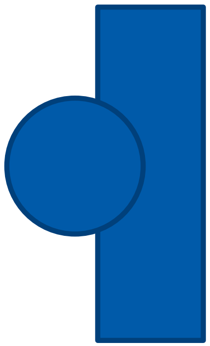
While the game is active

Close window, free memory (or don't), end process.

Collisions

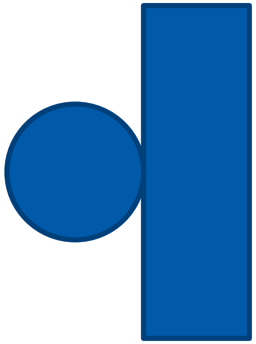
Intersection

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



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

{

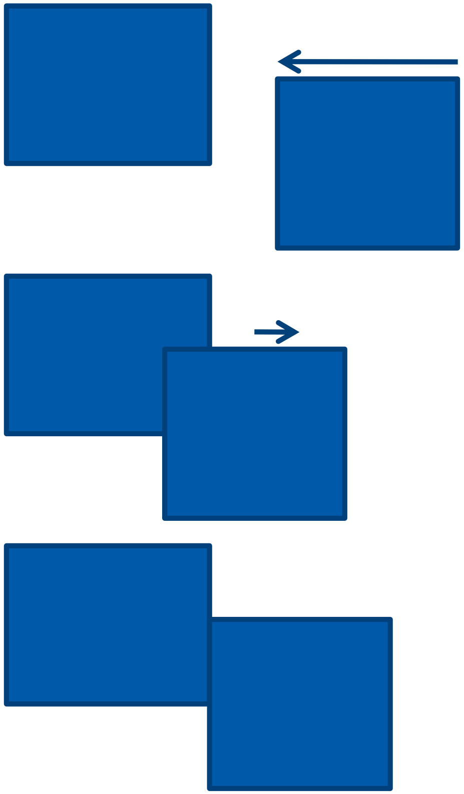
Move object

Check for collisions

If (collision detected) move back

}

}



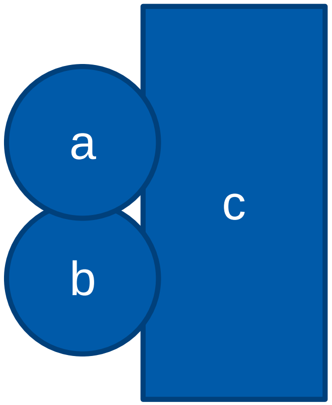
Collisions and Timing

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



Summary

Timing

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

Animations

- Functional
- Iterative

Game Loop

- Game state
- Collision detection

Memory Management

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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
- Allocate heap memory beforehand

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

TECHNISCHE UNIVERSITÄT DARMSTADT

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)

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

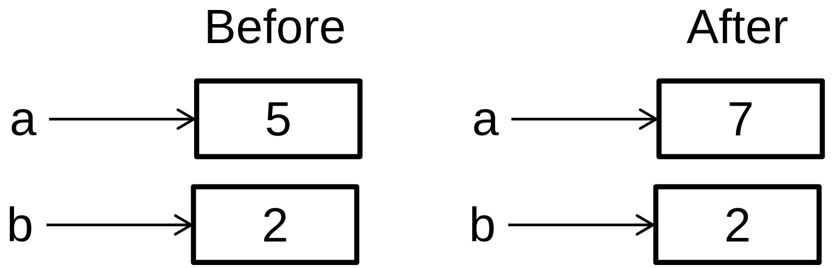
Getting the pointer to a variable

- 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

Allocated on the stack

- int array[3];

Array on the heap

- int* array = new int[3];

Deallocate using operator delete[]

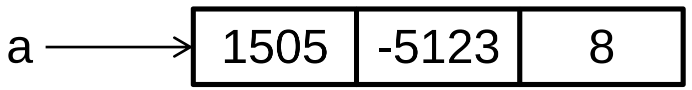
- delete[] array;

Mixing up leads to undefined behaviour

- (Also important for calling destructors)

Referencing arrays

Referenced using their first element

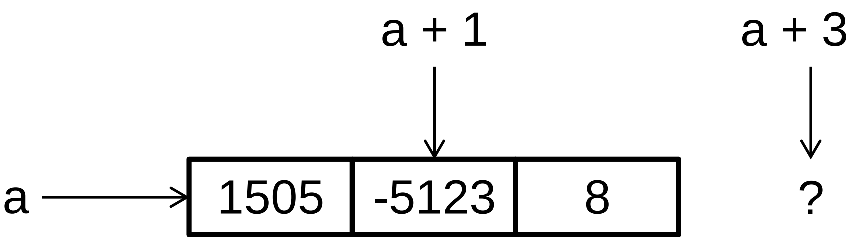


- int array[3];
- int *a = &array;
  - a points to the first element of array

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

Stack Allocator example

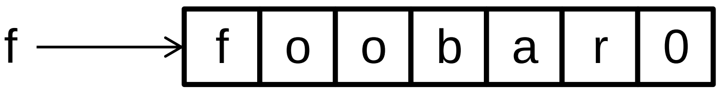
```
struct Something {
  …
};

char* memory = new char[huge number];
int endOfStack = 0;

Something* thing = (Something*)&memory[endOfStack];
endOfStack += sizeof(Something);
```

Strings

TECHNISCHE UNIVERSITÄT DARMSTADT

Strings are just arrays of chars

- char* f = “foobar”;



“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)

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

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 for exercise: Cracktros

Cracktro

Intro for a cracked game

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

Sometimes 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

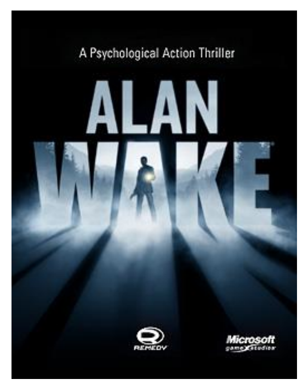
Always at the cutting edge of the hardware

- 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

Classical demo techniques

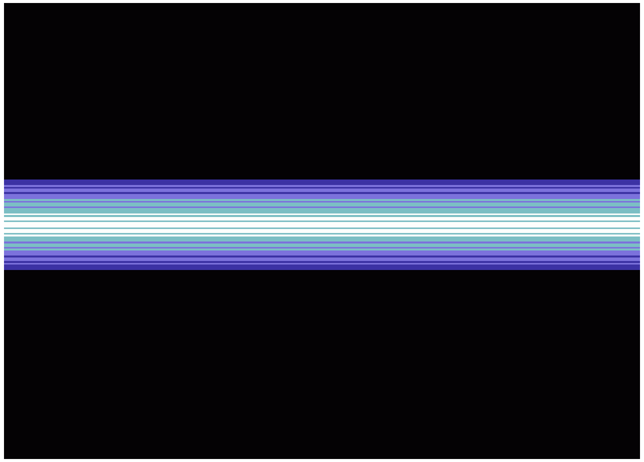
Scrolling

Moving along a sine wave

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

Rasterbars

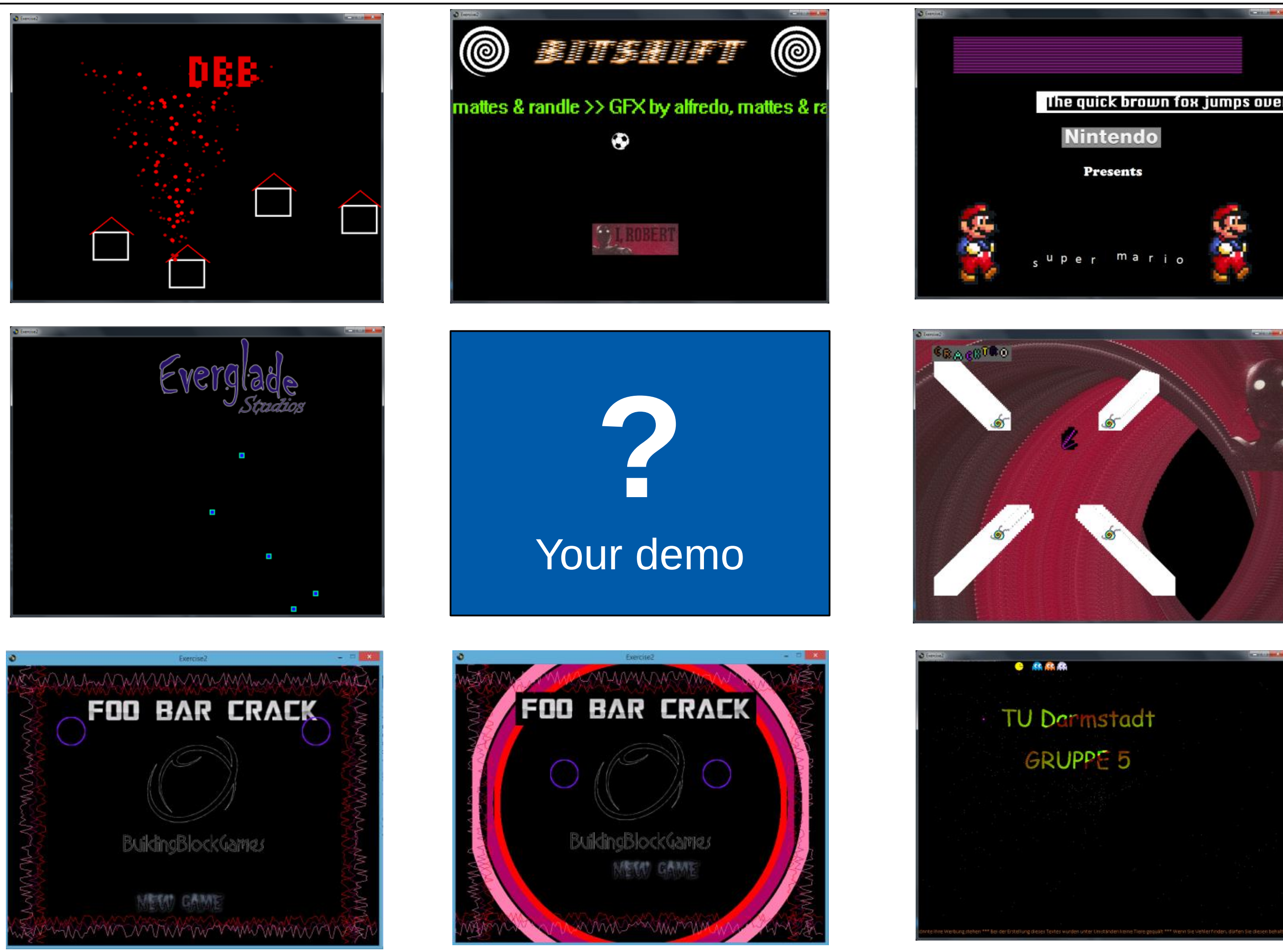
- Use an interrupt to paint lines
- Moving rasterbars along sine wave…



Good example for functional 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


DBB
BITSHIFT
mattes & randle >> GFX by alfredo, mattes & ra
The quick brown fox jumps over
Nintendo
Presents
super mario
Everglade
Studios
?
Your demo
FOO BAR CRACK
BuildingBlockGames
NEW GAME
FOO BAR CRACK
BuildingBlockGames
NEW GAME
TU Darmstadt
GRUPPE 5

Book Recommendations

TECHNISCHE UNIVERSITÄT DARMSTADT

Game Engine

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

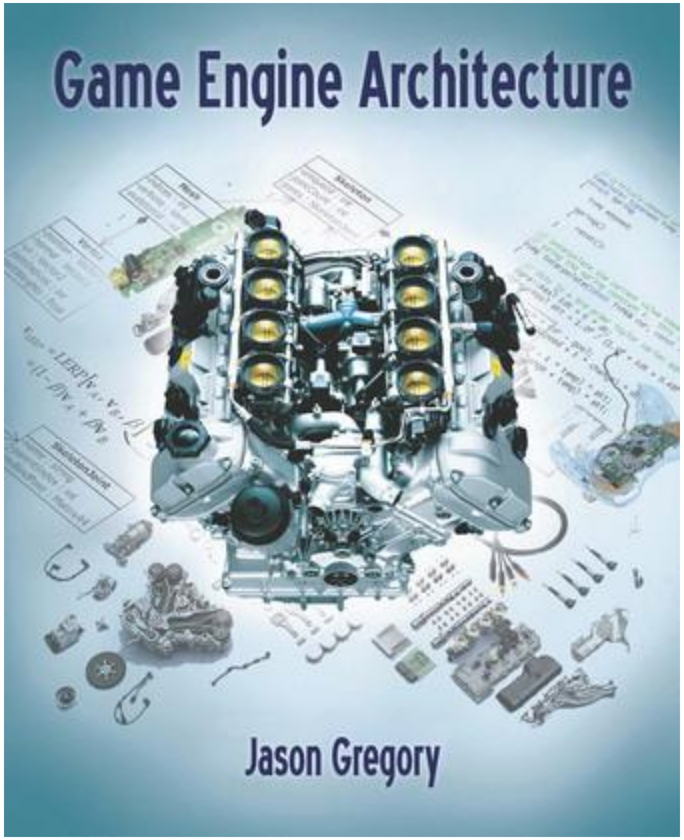
Fundamentals

- C++
- 3D Math
- Graphics, …

Practical Examples

Part of the „Semesterapparat“

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



Book Recommendations

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