Game Technology

Lecture 4 – 07.11.2014

Dipl-Inf. Robert Konrad
Dr.-Ing. Florian Mehm

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

Preliminary timetable

Lecture No.	Date	Topic
1	17.10.2014	Basic Input & Output
2	24.10.2014	Timing & Basic Game Mechanics
3	31.10.2014	Software Rendering 1
4	07.11.2014	Software Rendering 2
5	14.11.2014	Basic Hardware Rendering
6	21.11.2014	Animations
7	28.11.2014	Physically-based Rendering
8	05.12.2014	Physics 1
9	12.12.2014	Physics 2
10	19.12.2014	Scripting
11	16.01.2015	Compression & Streaming
12	23.01.2015	Multiplayer
13	30.01.2015	Audio
14	06.02.2015	Procedural Content Generation
15	13.02.2015	AI

Three Problems

- **Weird depth problems**
- **Weird textures**
- **Weird rotations**

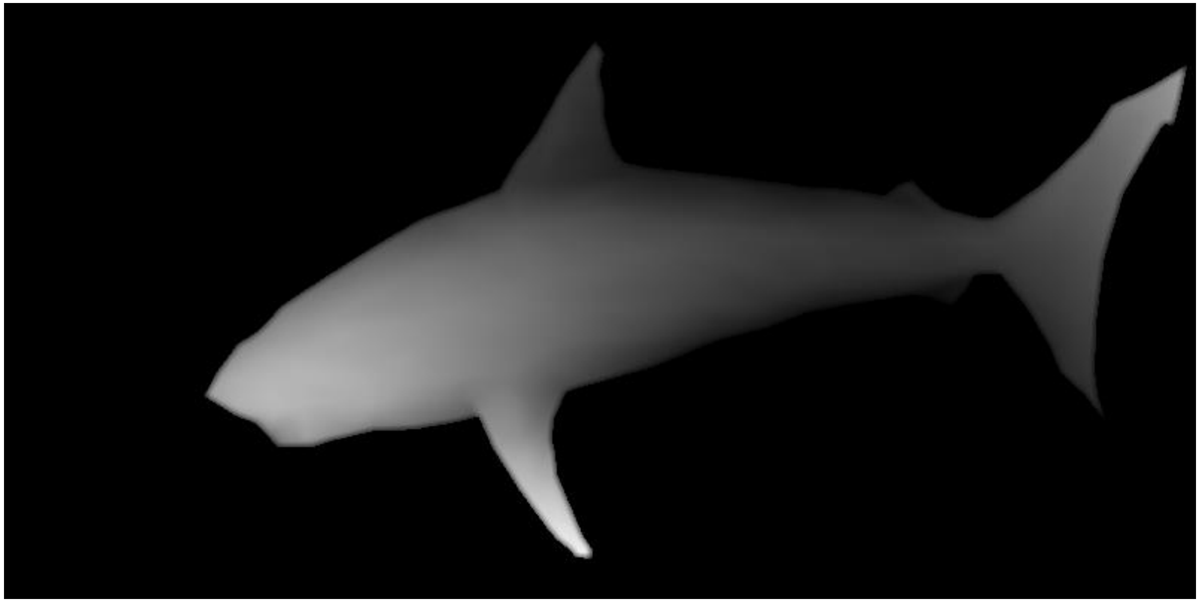
Weird Depth Problems

- **Backface culling & object sorting can not handle**
  - Overlaping geometry
  - Intersecting objects

Depth Buffer

- foreach (pixel) {

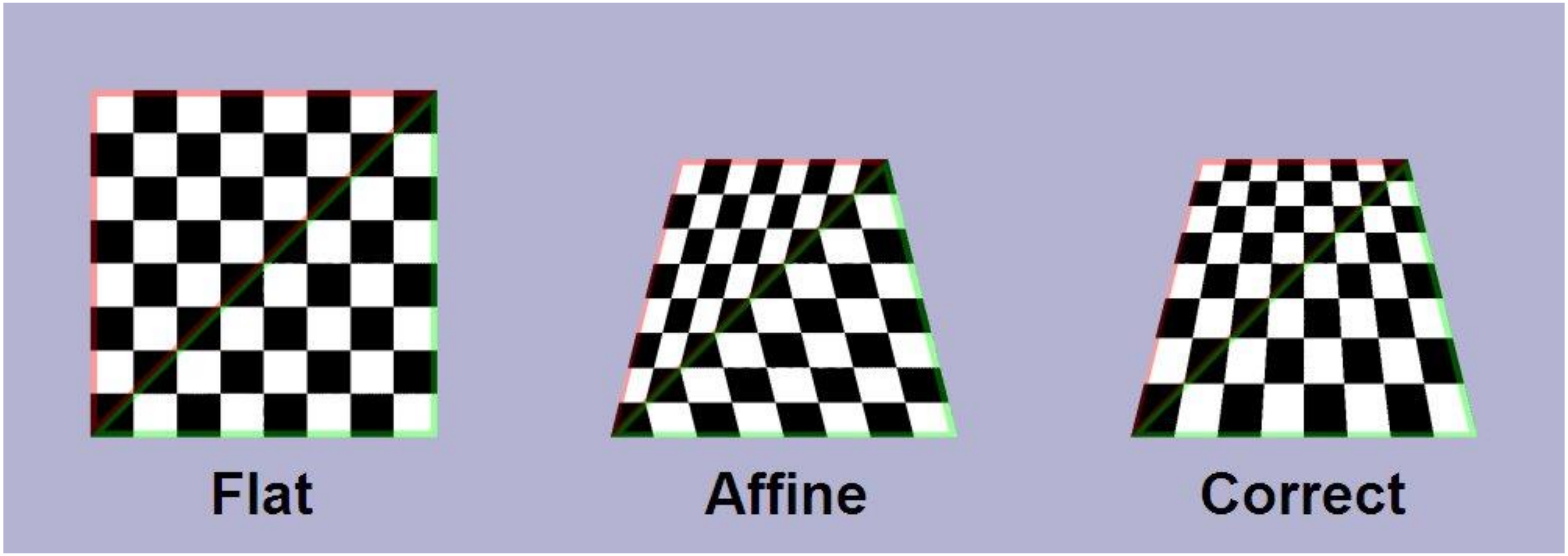
```
    if (framebuffer[pixel.x, pixel.y].z < z) continue;
    framebuffer[pixel.x, pixel.y].rgb = rgb;
    framebuffer[pixel.x, pixel.y].z = z;
}
```



Depth Buffer

- **Dead Simple**
- **Performance very bad**
  - When done in software
- **Performance OK**
  - When done in hardware
- **Does not help with partially transparent geometry**

Weird Textures



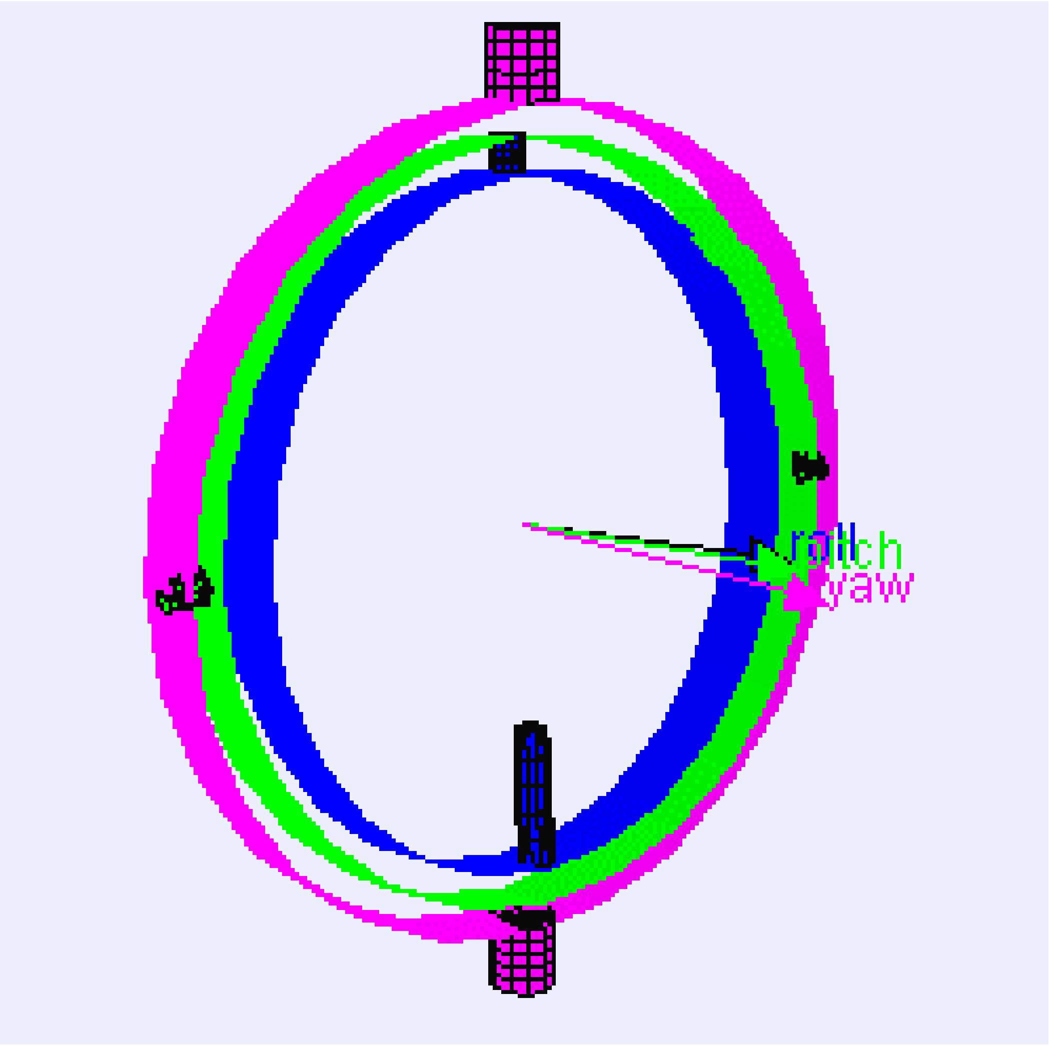
Weird Textures

Perspective Texture Correction

- **Regular interpolation:**
  - $u = (1 - \alpha)\, u0 + \alpha u1$

- **Perspective correct interpolation:**
  - $$u = \frac{(1 - \alpha)\,(u0 / z0) + \alpha\,(u1 / z1)}{(1 - \alpha)\,(1 / z0) + \alpha\,(1 / z1)}$$
  - Interpolate u / z

Weird Rotations



Dependent on order

- **Rotate around x-axis**
  Rotate around y-axis
  Rotate around z-axis

- **or**

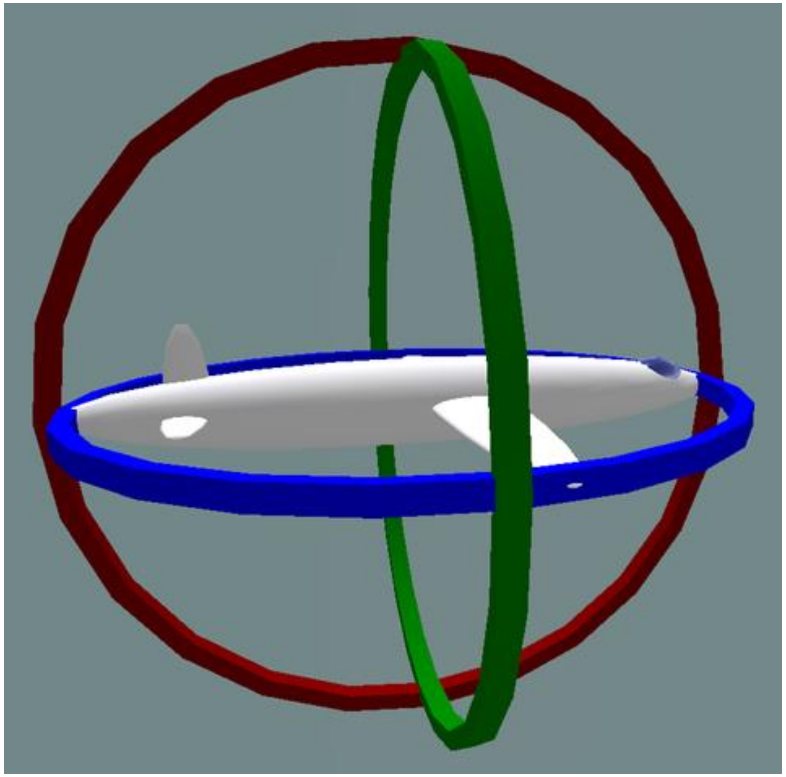
- **Rotate around z-axis**
  Rotate around y-axis
  Rotate around x-axis

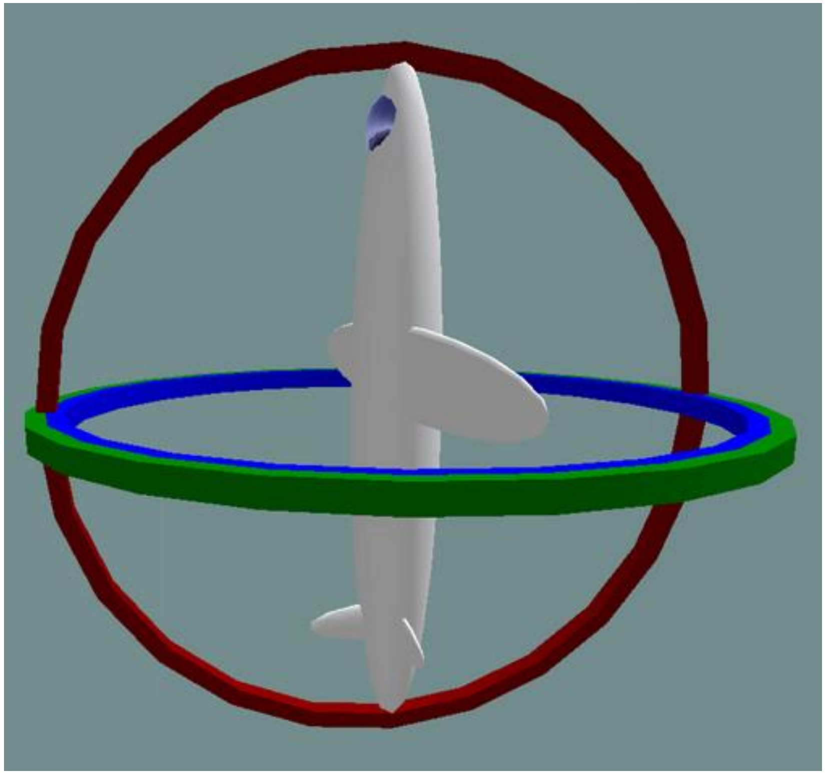
- **or**

- **…**

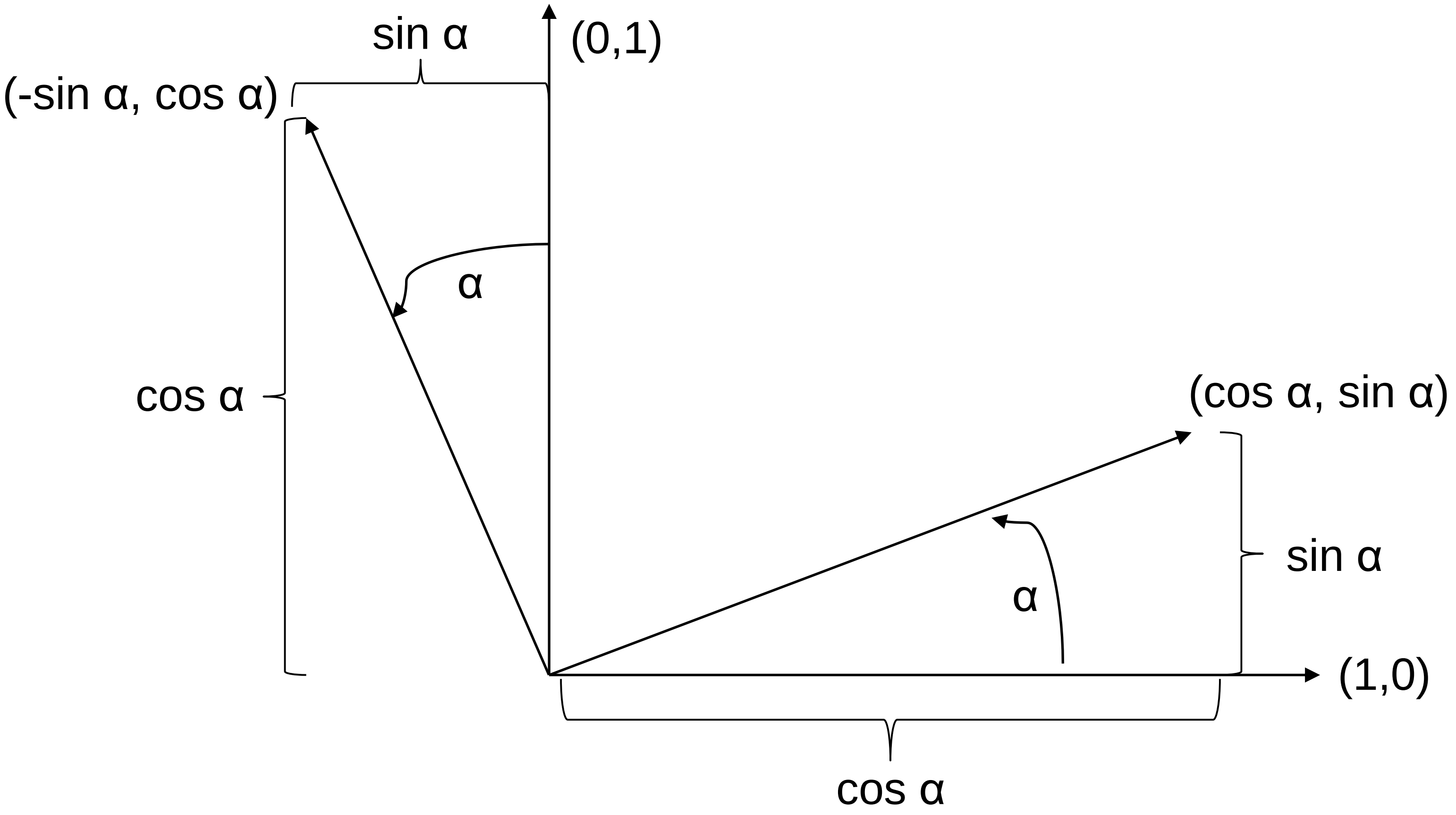
Gimbal Lock

TECHNISCHE UNIVERSITÄT DARMSTADT





Camera Rotations



Camera Rotations

- **Old Point**
  - $(x,y) = x(1,0) + y(0,1)$
- **New Point**
  - $R(x,y,\alpha) = x(\cos\alpha, \sin\alpha) + y(-\sin\alpha, \cos\alpha)$
  - $= (x\cos\alpha, x\sin\alpha) + (-y\sin\alpha, y\cos\alpha)$
  - $= (x\cos\alpha - y\sin\alpha, x\sin\alpha + y\cos\alpha)$

Camera Rotations

- **Old Point**
  - $(x,y) = x(1,0) + y(0,1)$
  - $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$

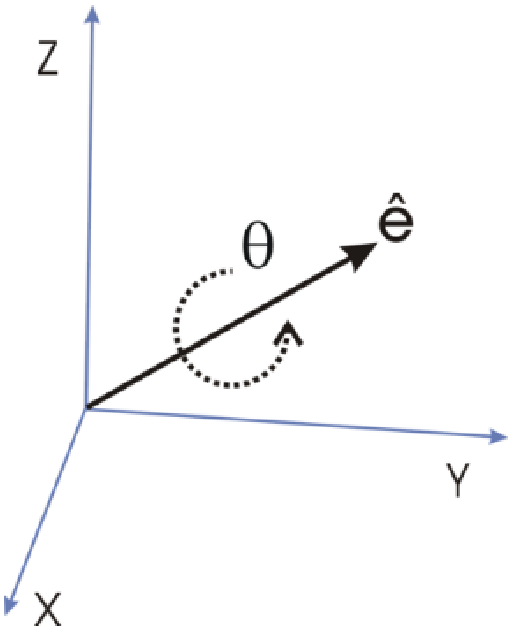
- **New Point**
  - $R(x,y,\alpha) = x(\cos\alpha, \sin\alpha) + y(-\sin\alpha, \cos\alpha)$
  - $= (x\cos\alpha, x\sin\alpha) + (-y\sin\alpha, y\cos\alpha)$
  - $= (x\cos\alpha - y\sin\alpha, x\sin\alpha + y\cos\alpha)$
  - $\begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\alpha - y\sin\alpha \\ x\sin\alpha + y\cos\alpha \end{bmatrix}$

Matrix Multiplication

$$\mathbf{AB} = \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} ax + by + cz \\ px + qy + rz \\ ux + vy + wz \end{pmatrix},$$

4 Coordinates

- **Euler's rotation theorem:**
- **Any rotation or sequence of rotations of a rigid body or coordinate system about a fixed point is equivalent to a single rotation by a given angle θ about a fixed axis (called Euler axis) that runs through the fixed point.**



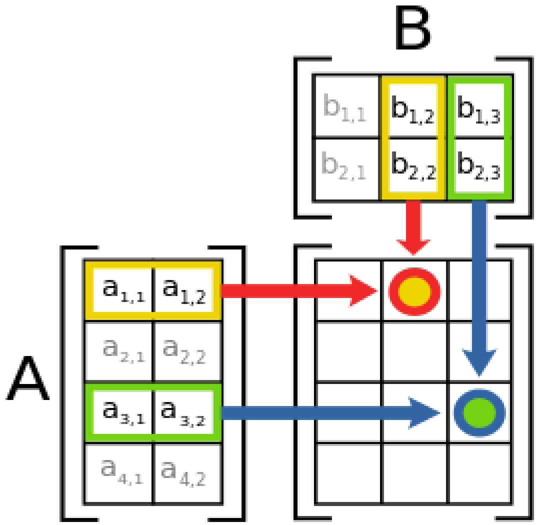
Rotation Matrix

- **u = unit vector**
- **Θ = rotation around u**

$$R = \begin{bmatrix} \cos\theta + u_x^2\left(1-\cos\theta\right) & u_x u_y\left(1-\cos\theta\right) - u_z\sin\theta & u_x u_z\left(1-\cos\theta\right) + u_y\sin\theta \\ u_y u_x\left(1-\cos\theta\right) + u_z\sin\theta & \cos\theta + u_y^2\left(1-\cos\theta\right) & u_y u_z\left(1-\cos\theta\right) - u_x\sin\theta \\ u_z u_x\left(1-\cos\theta\right) - u_y\sin\theta & u_z u_y\left(1-\cos\theta\right) + u_x\sin\theta & \cos\theta + u_z^2\left(1-\cos\theta\right) \end{bmatrix}.$$

Matrix * Matrix

- **Concatenate Rotations**
- **Save premultiplied matrices = Save calculations**



$$x_{12} = a_{11}b_{12} + a_{12}b_{22}$$
$$x_{13} = a_{11}b_{13} + a_{12}b_{23}$$
$$x_{32} = a_{31}b_{12} + a_{32}b_{22}$$
$$x_{33} = a_{31}b_{13} + a_{32}b_{23}$$

Identity Matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Affine Transformations

- „a function … which preserves points, straight lines and planes"

- Translation
- Scaling
- Shearing
- Rotation

Matrix Transformations

- **Dimension * Dimension matrices support all affine transformations**
  - But Translations

Translation Matrix

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

Homogenous Coordinates

- **(x, y, z, w)**
- **-> 3D: (x / w, y / w, z / w)**

- **3D point -> 4D: (x, y, z, 1)**
- **3D direction -> 4D: (x, y, z, 0)**

Perspective Projection

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

4x4 Matrix

x-axis	y-axis	z-axis	translation
perspective			1

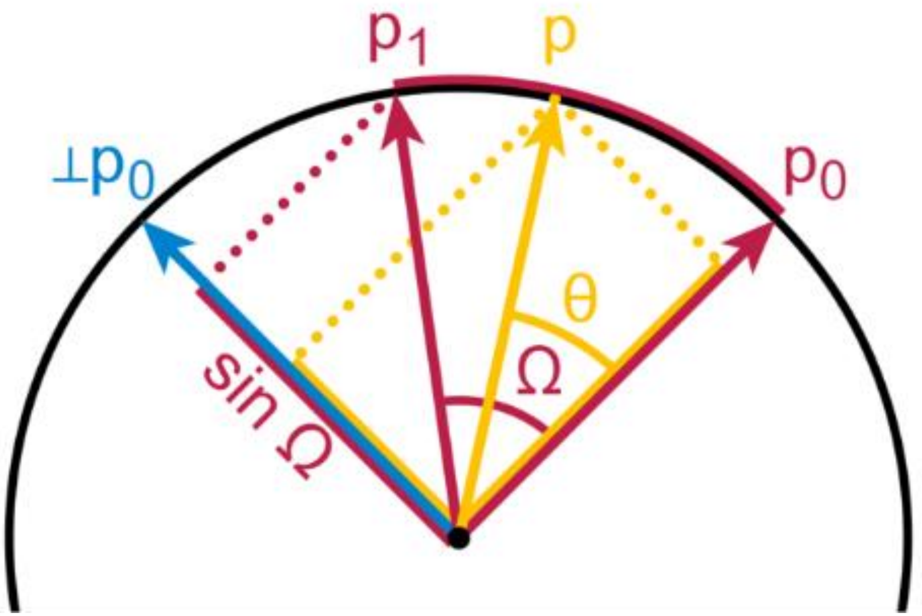
Typical Setup

- **projection * view * model * position**

- **Projection**
  - Kore::Matrix::perspectiveProjection
    - Field of view <
    - Aspect ration (width / height)
    - z near, z far
- **View**
  - Kore::Matrix::lookAt
    - Eye vector
    - At vector
    - Up vector
- **Model**
  - Translations, Rotations,…

Rotation interpolation?

- **Euler angles**
  - Easy for one rotation
  - Super weird for three rotations
- **Rotation matrices**
  - Difficult



Quaternions

TECHNISCHE UNIVERSITÄT DARMSTADT

- **4D imaginary numbers**
- **Three imaginary components**
- ***i*² = *j*² = *k*² = *ijk* = −1**

- **Can represent rotations**

$$\mathbf{q} = e^{\frac{\theta}{2}(u_x\mathbf{i}+u_y\mathbf{j}+u_z\mathbf{k})} = \cos\frac{\theta}{2} + (u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k})\sin\frac{\theta}{2}$$

- **Rotation**
  - rotated point = q * point * q^-1

Quaternions

- **[w, v] (w: real scalar, v: imaginary vector)**

- **q1*q2 = [w1w2 - v1 · v2, v1 x v2 + w1v2 +w2v1]**
  - q1*q2 != q2*q1

- **Inverse [w, v] = [w, -v] / ($w^2 + x^2 + y^2 + z^2$)**

Interpolation

- **Spherical Linear intERPolation (SLERP)**

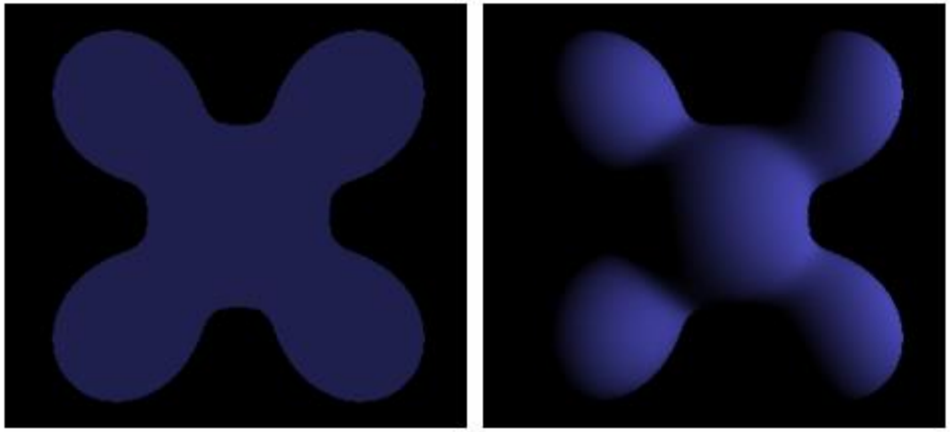
- $$\text{slerp}(q1, q2, t) = \frac{\sin(1 - t)\,\theta}{\sin\theta} q1 + \frac{\sin t\,\theta}{\sin\theta} q2$$

- $$\theta = \cos\text{\textasciicircum}{-1}\left(\frac{w1w2 + x1x2 + y1y2 + z1z2}{|q1||q2|}\right)$$

Quaternion to Matrix

$$\begin{matrix} 1 - 2*y^2 - 2*z^2 & 2*x*y - 2*z*w & 2*x*z + 2*y*w \\ 2*x*y + 2*z*w & 1 - 2*x^2 - 2*z^2 & 2*y*z - 2*x*w \\ 2*x*z - 2*y*w & 2*y*z + 2*x*w & 1 - 2*x^2 - 2*y^2 \end{matrix}$$

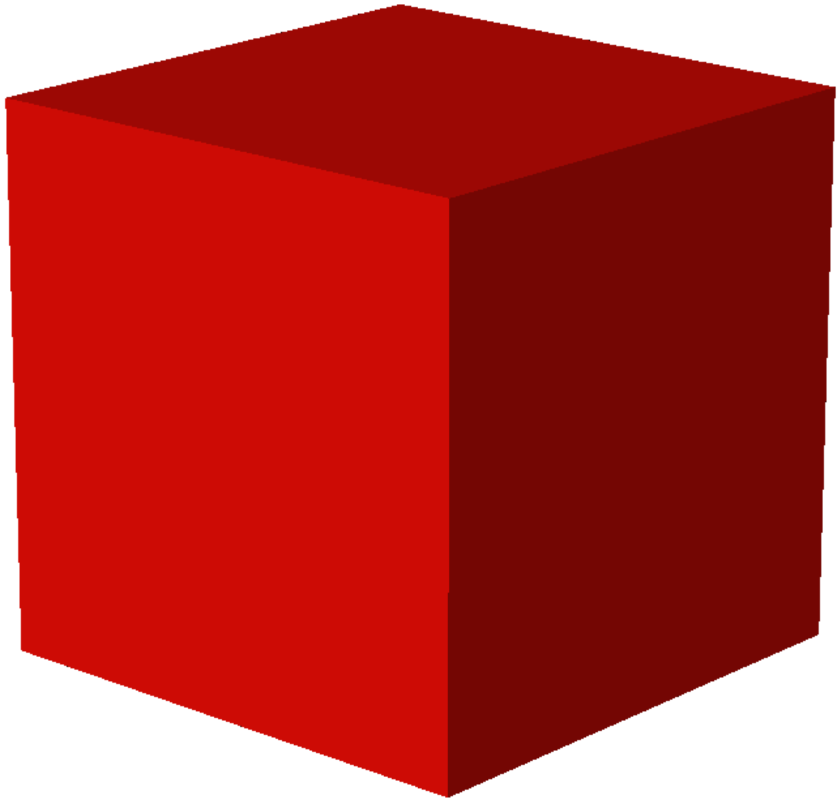
Lighting



Normals

- **Defined per vertex**
- **Direction: (x, y, z, 0)**
- **Translation * (x, y, z, 0) = (x, y, z, 0)**
- **Rotation * (x, y, z, 0) = (…)**

Vertex Splits



Super Basic Lighting

- **L*N**

- **dot product** $$\vec{a} \cdot \vec{b} = |\vec{a}|\,|\vec{b}|\ \cos\sphericalangle(\vec{a}, \vec{b}).$$ $$\vec{a} \cdot \vec{b} = a_1\,b_1 + a_2\,b_2 + a_3\,b_3.$$

- **L = Light Direction**
- **N = Normal**

Per Vertex vs per Pixel

- **Per Vertex**
  - Fast
  - Calculate lighting per vertex
    Interpolate colors

- **Per Pixel**
  - Pretty
  - Interpolate normals
    Calculate lighting per pixel

Parallel Computations

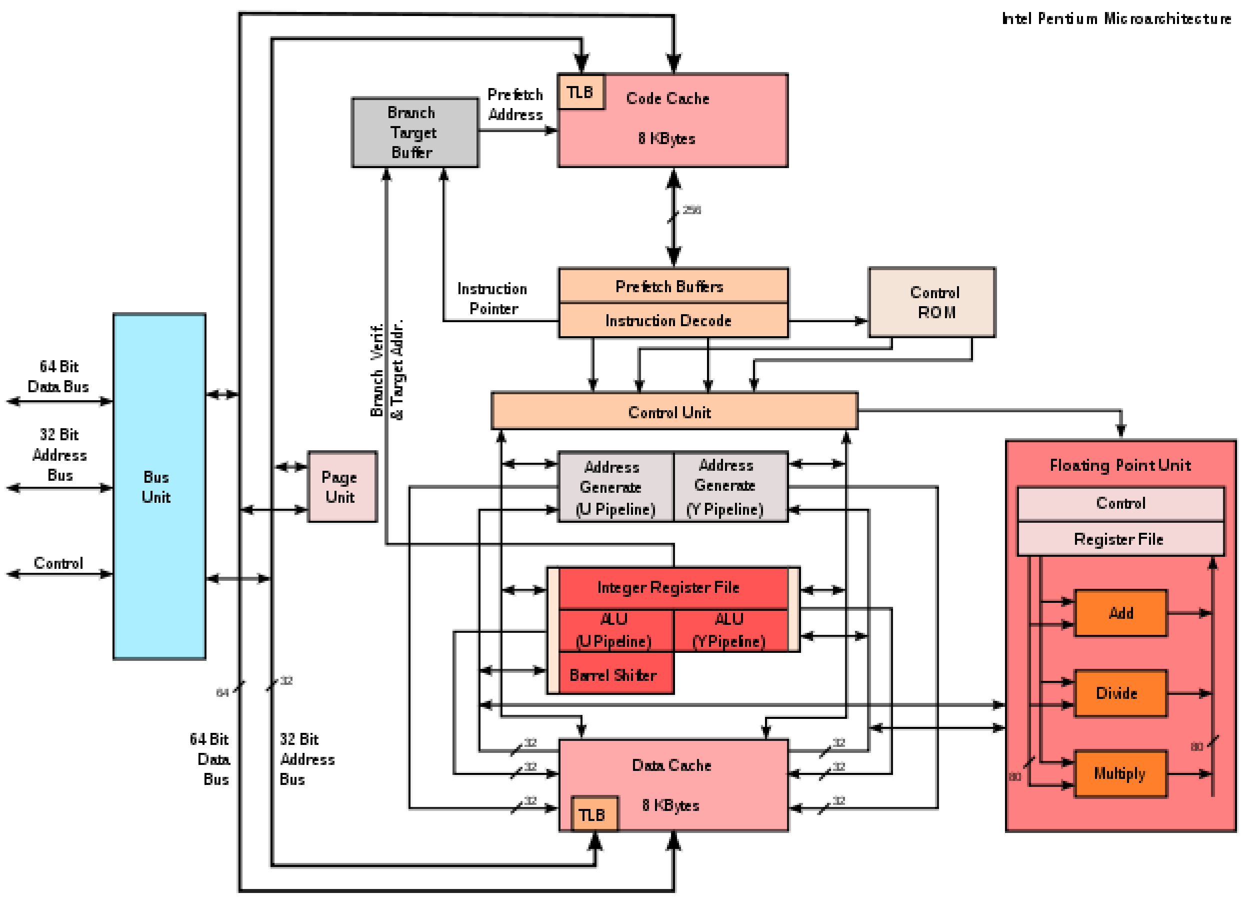
TECHNISCHE UNIVERSITÄT DARMSTADT

- **Superscalar CPUs**
- **SIMD Instructions**
- **Multithreading**

C/C++ Fail

- **No standardized support for SIMD instructions**
- **Multithreading support since 2011**

Superscalar CPUs



Superscalar Execution

- **c = a + b**
  d = a + b // can be parallelized

- **c = a + b**
  d = a + c // can not be parallelized

Superscalar Execution

- **No explicit support necessary (or even possible?)**
- **Comiler can reorder instructions**
- **Keep in mind when optimizing**
  - Profiler can show < 1 ticks per instruction

SIMD Instructions

- **SIMD – Single Instruction Multiple Data**
  - Apply same calculation to multiple values

- **Can easily be applied to Vector/Matrix math**

- **Automatic compiler optimizations – very limited**

x86

TECHNISCHE UNIVERSITÄT DARMSTADT

- **SSE – since Pentium 3 in 1999**
- **128 bit registers**
  - 4 float numbers per register
- **SSE2, SSE3, SSE4, AVX,…**
- **SSE2 supported by every x64 CPU**
- **64 bit Operating Systems use SSE instructions for all floating point calculations**

ARM

- **NEON**
- **Since Cortex-A8 (but only optional)**
- **128 bit registers**
- **…**

Intrinsics

- **#include <xmmintrin.h>**
- **__m128 value1 = _mm_set_ps(1, 2, 3, 4);**
  __m128 value2 = _mm_set_ps(5, 6, 7, 8);
  __m128 added = _mm_add_ps(value1, value2);
  float allAdded = added.m128_f32[0] + added.m128_f32[1] + added.m128_f32[2] + added.m128_f32[3];

- **Just like assembler programming**
  - (minus register numbers)

Current Situation

- **No Standard**
- **SSE and Neon – incompatible intrinsics**
- **Different compilers – ~compatible instrinsics**
- **Libraries of small functions of macros can help**
  - http://www.gamedev.net/page/resources/_/technical/general-programming/practical-cross-platform-simd-math-r3068

Multithreading

- Standard support since 2011
- OS APIs since 1980s
- Kore::Thread

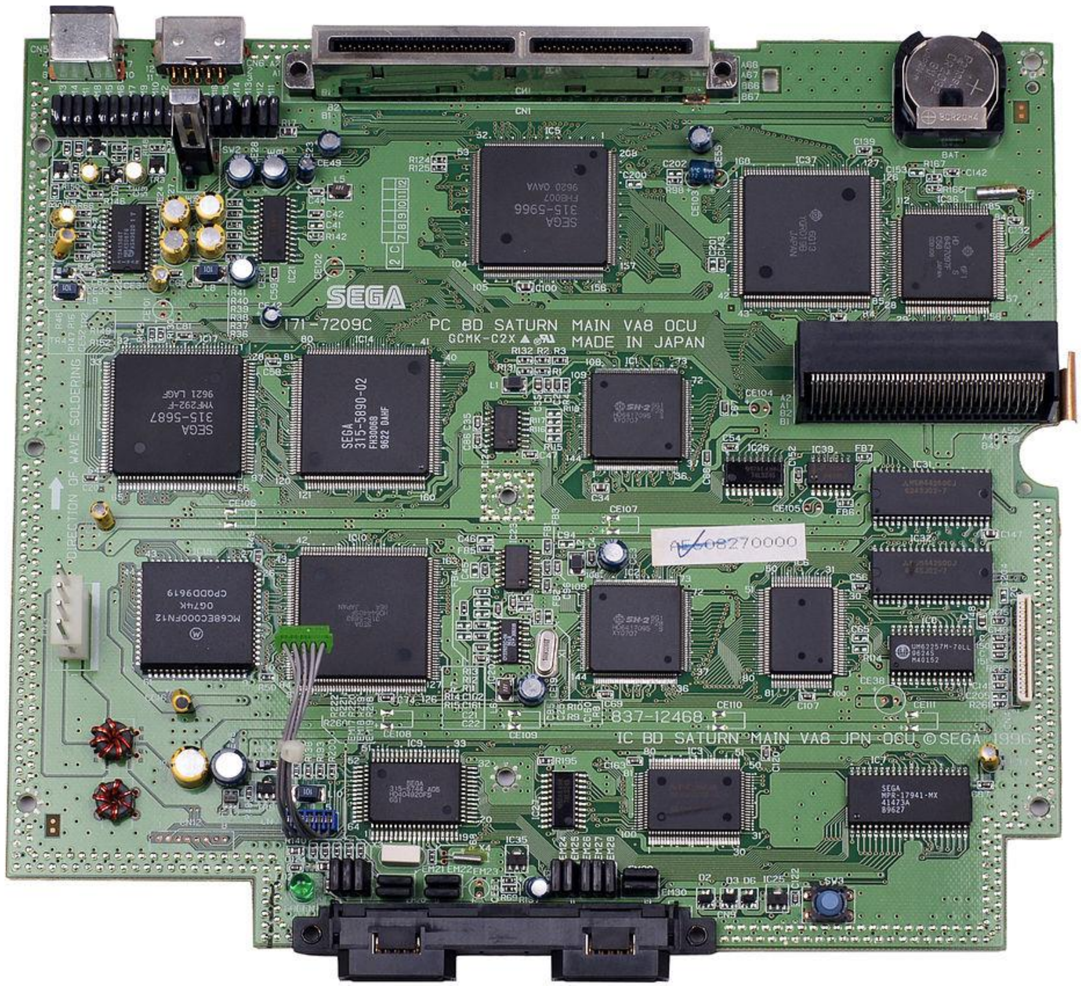
Multithreading

- **Traditionally avoided in Games**

- **Very important for multicore CPUs**

Multithreading

TECHNISCHE UNIVERSITÄT DARMSTADT



Multithreading

- **Independent execution threads**
- **Same address space**

- **Lots of problems**
- **Use for speed**
  - Number of threads = number of cores
- **Use for asynchronicity**
  - Loading data from disk
- **Never use for convenience**

Race Conditions

Race Conditions

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Race Conditions

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Race Conditions

- **Very difficult to debug**
- **Might happen very rarely**
- **Worst kind of bugs**

Fun Example

- **Android**

- **onKeyDown**
  - UI Thread
- **onDrawFrame**
  - „The renderer will be called on a separate thread"

Mutex

- **Kore::Mutex m;**
  m.Create();

- **m.Lock();**
  // access shared state
  m.Unlock();

- **Mapped to mutex in Linux**
- **Mapped to critical section in Windows**
  - Windows Mutex is used for interprocess sync

Mutex

- **Can slow down program**
  - Syscalls, cache flushes,…

- **Minimize sync points**

- **Typical design a**
  - CPU core 1 only for physics
  - CPU core 2 for everything else
  - Sync once per frame

- **Typical design b**
  - Work package objects
  - Worker threads (one for each CPU core)
  - Work package manager assigns packages to threads

Lock Free Multithreading

- **Can speed up programs**
- **Atomic operations**
- **Arcane magic**