Game Technology

Lecture 5 – 14.11.2014

TECHNISCHE UNIVERSITÄT DARMSTADT

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

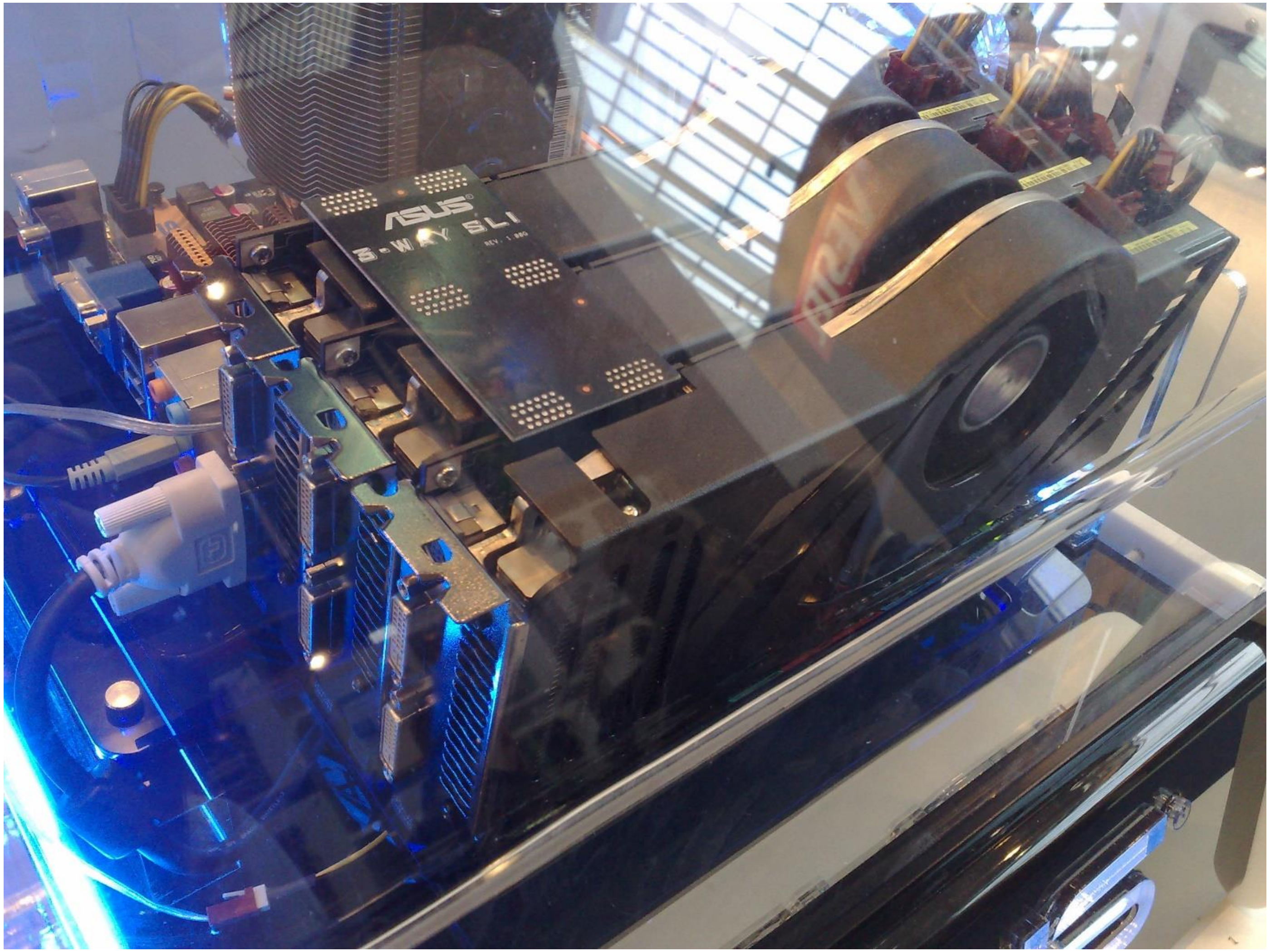
Prof. Dr.-Ing. Ralf Steinmetz
KOM - Multimedia Communications Lab

Preliminary timetable

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Dedicated Gaming Hardware



Pong & Computer Space

Apple 2

© by Marco Mioli

Atari VCS

POWER
ON
OFF
COLOR
TV TYPE
B•W
VIDEO COMPUTER SYSTEM™
GAME SELECT
GAME RESET
ATARI

NES


Nintendo
ENTERTAINMENT SYSTEM™
POWER
RESET
Nintendo®
SELECT
START
B
A

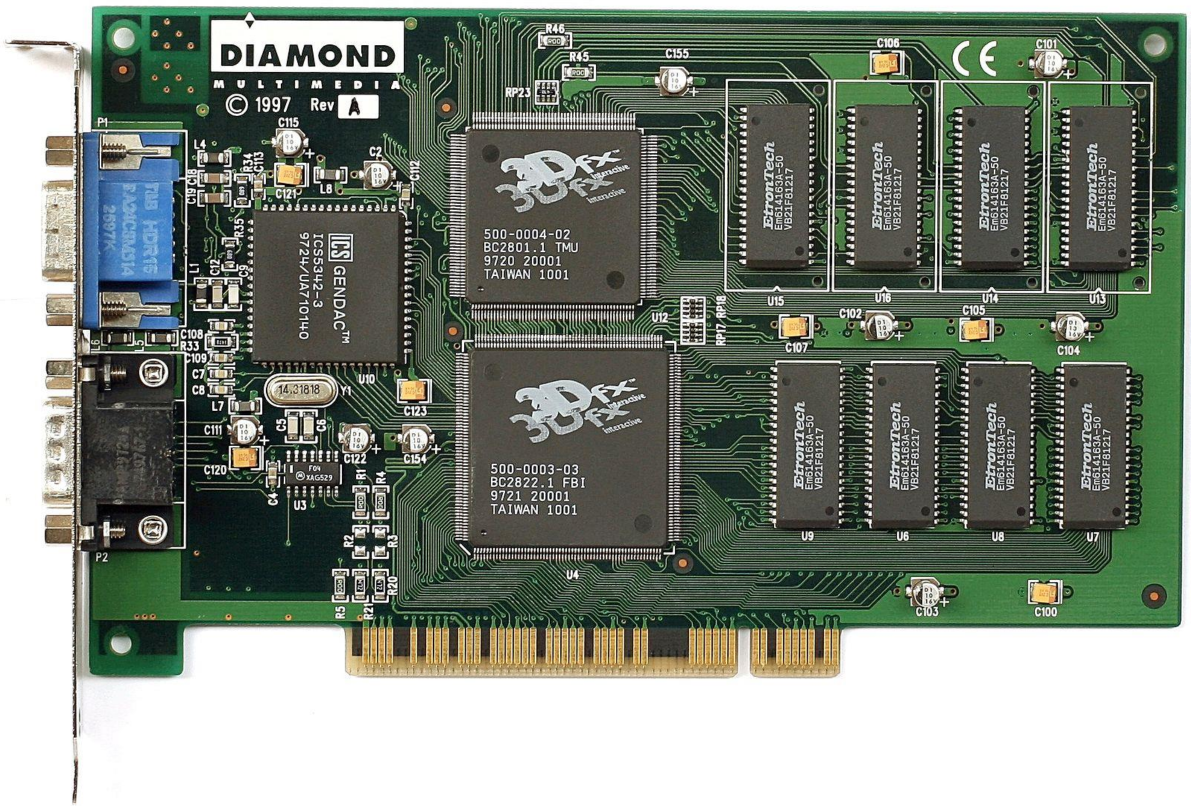
C64

Amiga

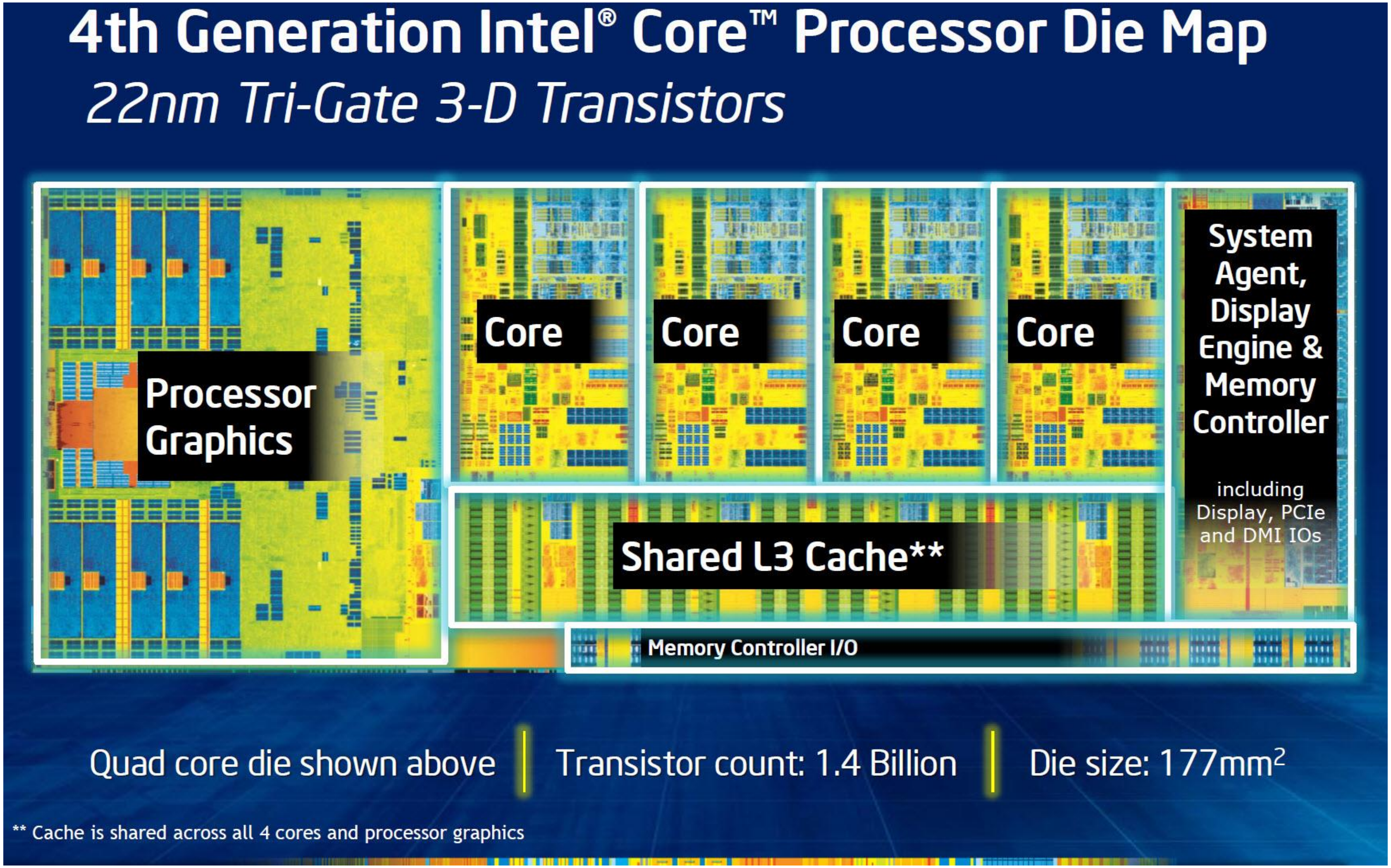
System
Shell
Utilities
Prefs
Expansion
Trashcan
RAM DISK
Workbench1.3
MergeMem
SetMap
FastMemFirst
AMIGA

IBM PC

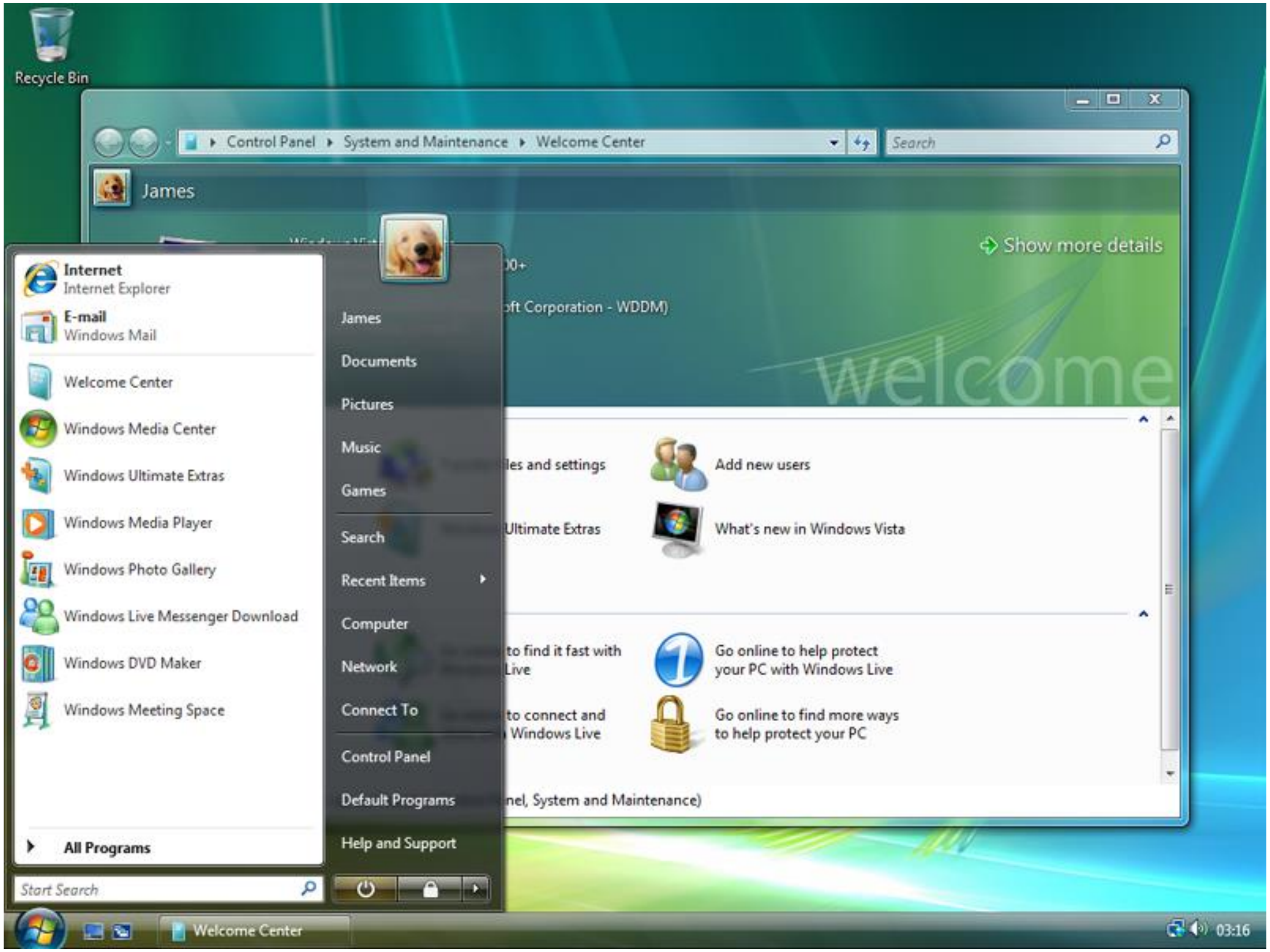
Voodoo


DIAMOND
MULTIMEDIA
© 1997 Rev A
500-0004-02
BC2801.1 TMU
9720 20001
TAIWAN 1001
500-0003-03
BC2822.1 FBI
9721 20001
TAIWAN 1001
GENDAC
EtronTech

Modern intel CPUs



Vista



Recycle Bin
Control Panel
System and Maintenance
Welcome Center
Search
James
Show more details
welcome
Internet
Internet Explorer
E-mail
Windows Mail
Welcome Center
Windows Media Center
Windows Ultimate Extras
Windows Media Player
Windows Photo Gallery
Windows Live Messenger Download
Windows DVD Maker
Windows Meeting Space
All Programs
Start Search
James
Documents
Pictures
Music
Games
Search
Recent Items
Computer
Network
Connect To
Control Panel
Default Programs
Help and Support
Add new users
What's new in Windows Vista
Go online to help protect
your PC with Windows Live
Go online to find more ways
to help protect your PC
03:16

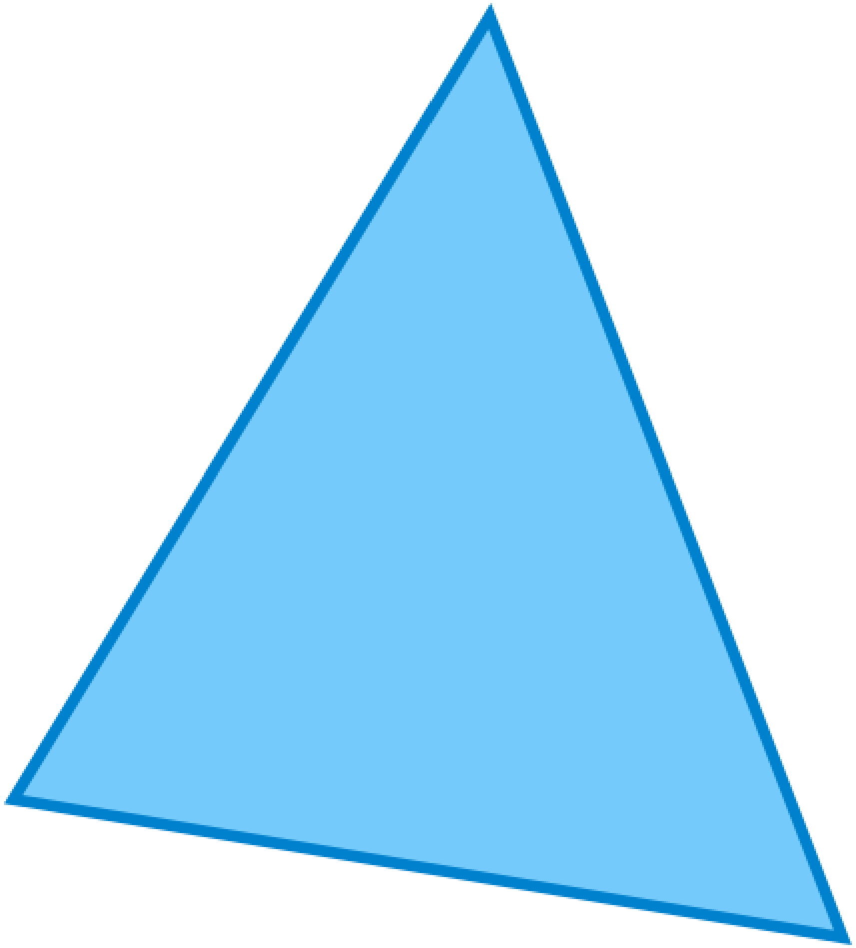
PS4

CPU vs GPU

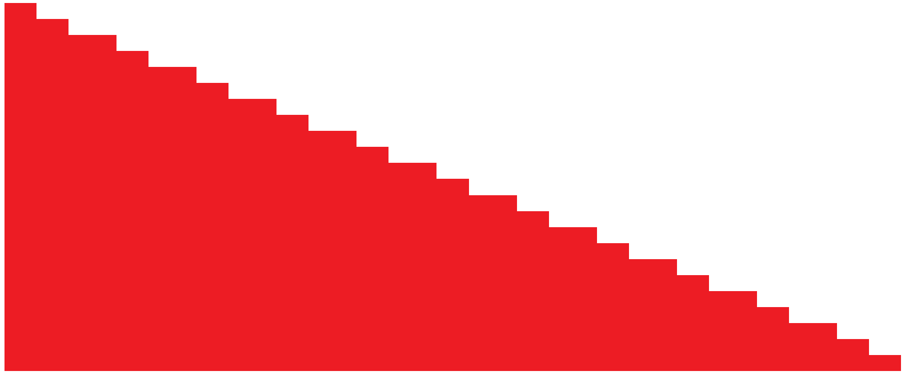
- **CPU**
  - Run sequential code as fast as possible

- **GPU (Graphical Processing Unit)**
  - Massively parallel code execution
  - Plus triangle rasterizer
  - Plus texture sampler

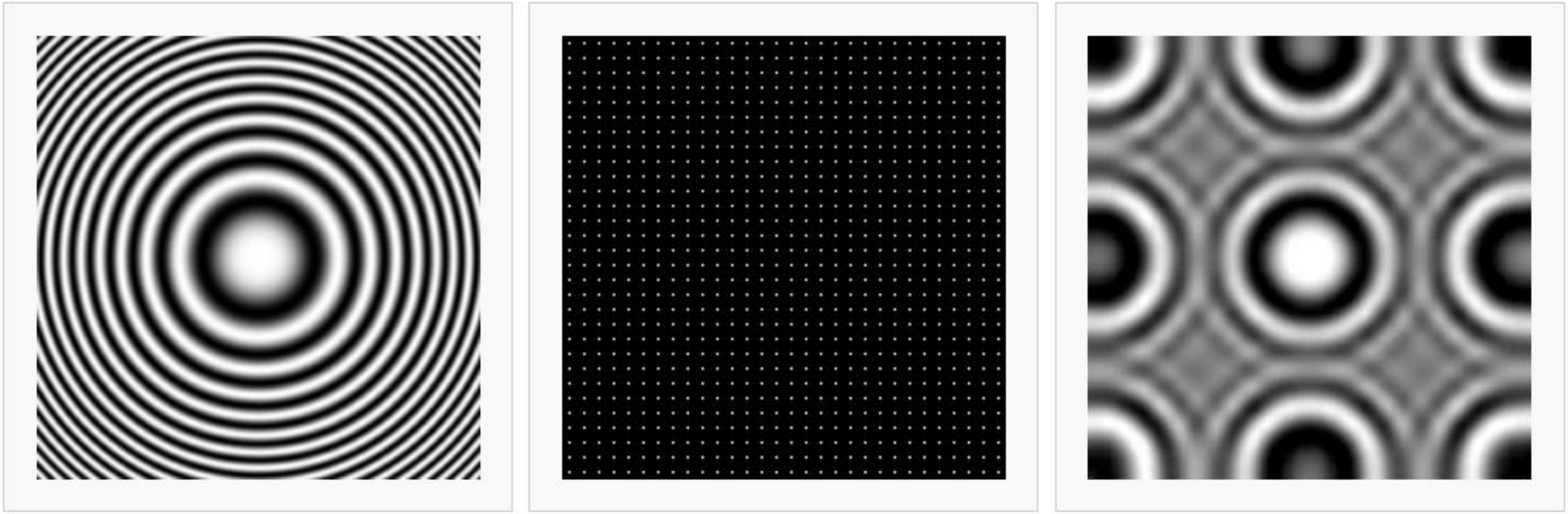
Triangles



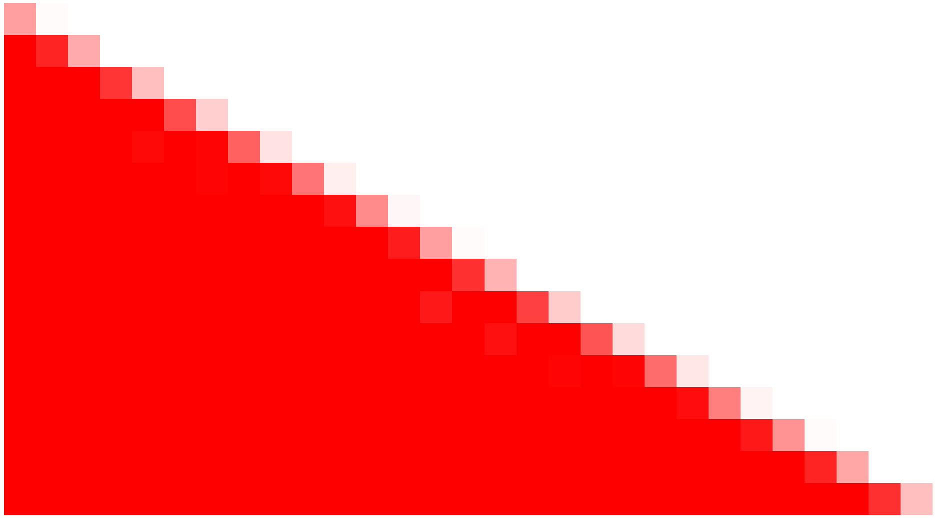
„Aliasing“



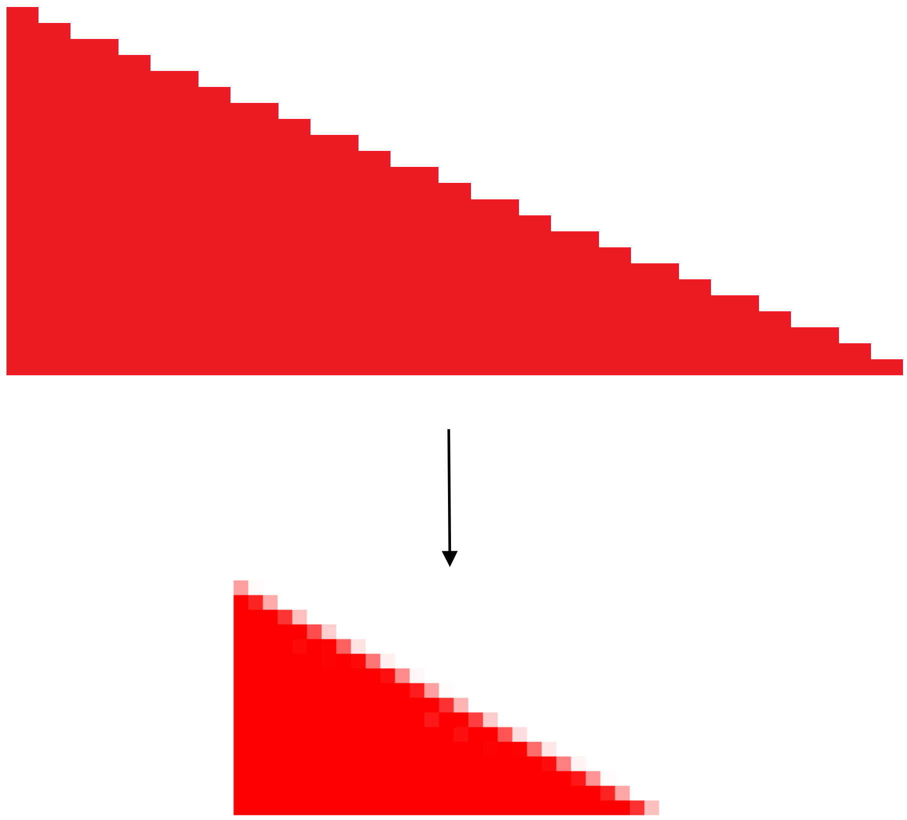
Aliasing



Edge Antialiasing



Supersample Antialiasing



Multisample Antialiasing

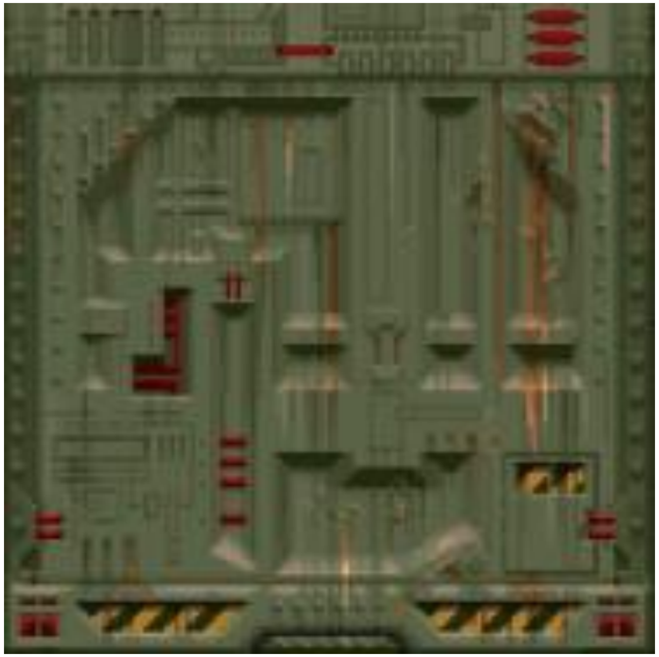
- **Optimized Supersampling**
- **More samples only at triangle edges**

Postprocess Antialiasing

Temporal Anti-Aliasing

Textures

- **Basically images**
- **Preferebly 2^n*2^n**
  - Other sizes not necessarily supported
    - Expand image and fix up texture coordinates



Texture Sampling

- **Point Filtering**
- **Bilinear Filtering**
  - Interpolate four neighbouring pixels

Mip Mapping

- **Example: Texture mapped to one pixel**
  - Ideally calculate mean color value of the complete texture

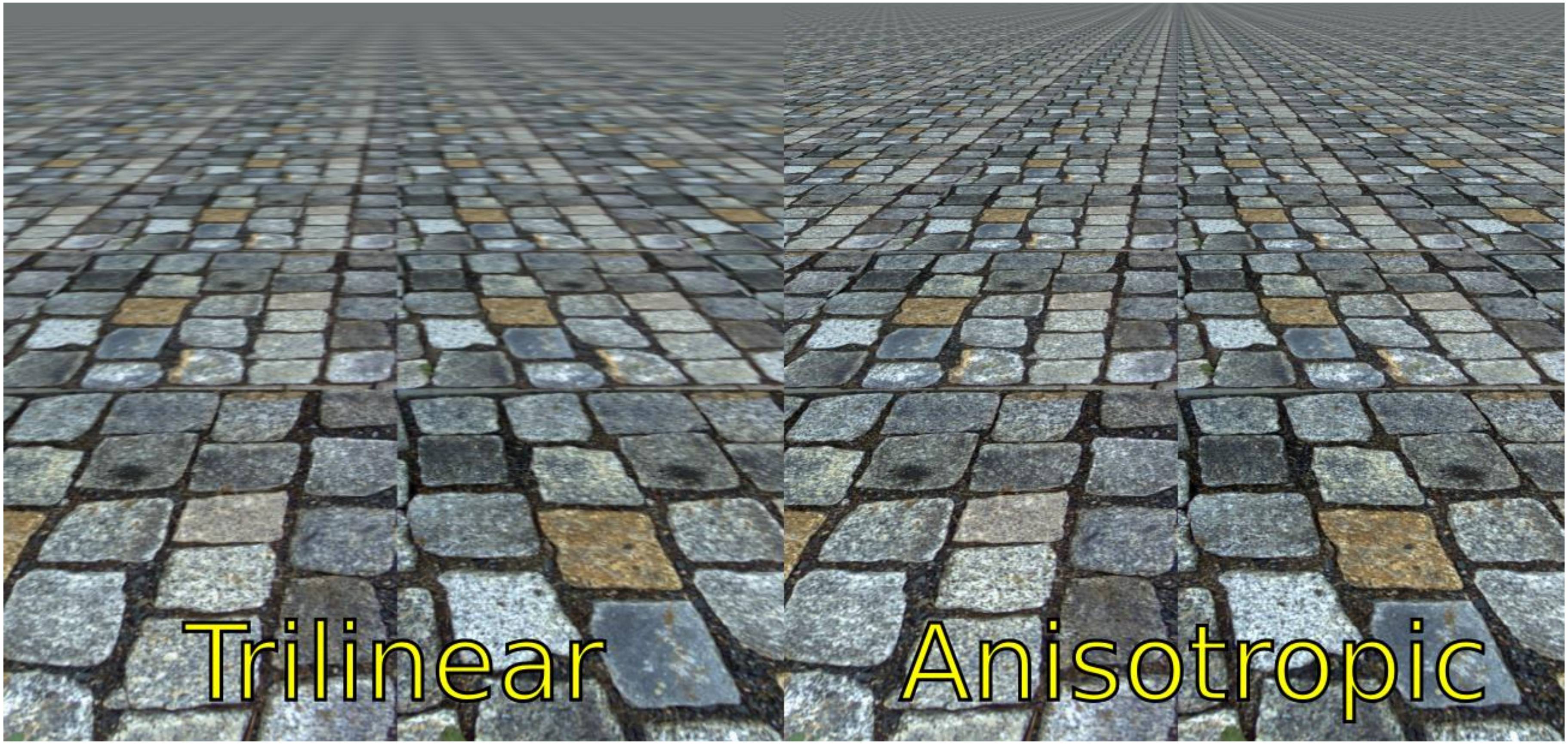
- **Trick: Precompute images**
  - Width / 2, Height / 2
  - Width / 4, Height / 4
  - …
  - Sample from best fitting image

Mip Mapping

- **Seams between mip levels are often visible**
  - Trilinear filtering

- **Perspective stretches images differently in x and y**
  - No optimal mip level

Anisotropic Filtering



Depth Buffer

- **Implemented in hardware**
- **Used automatically by the rasterizer**
- **3D apis offer simple configuration**
  - Off, allow only smaller values, allow only larger values

Alpha-Blending

- **Critical for performance**
  - Reads in previous pixels, stresses memory interface
  - Makes parallel execution more difficult

- **Fixed modes**
  - 1 * new pixel + 0 * old pixel
  - source alpha * new pixel + (1 - source alpha) * old pixel
  - …
  - (destination alpha is rarely used)

Programmable Blending

- **Render to texture**
- **Draw rendered texture**
- **Draw blended geometry**
  - Use rendered texture as input

- **Much slower**

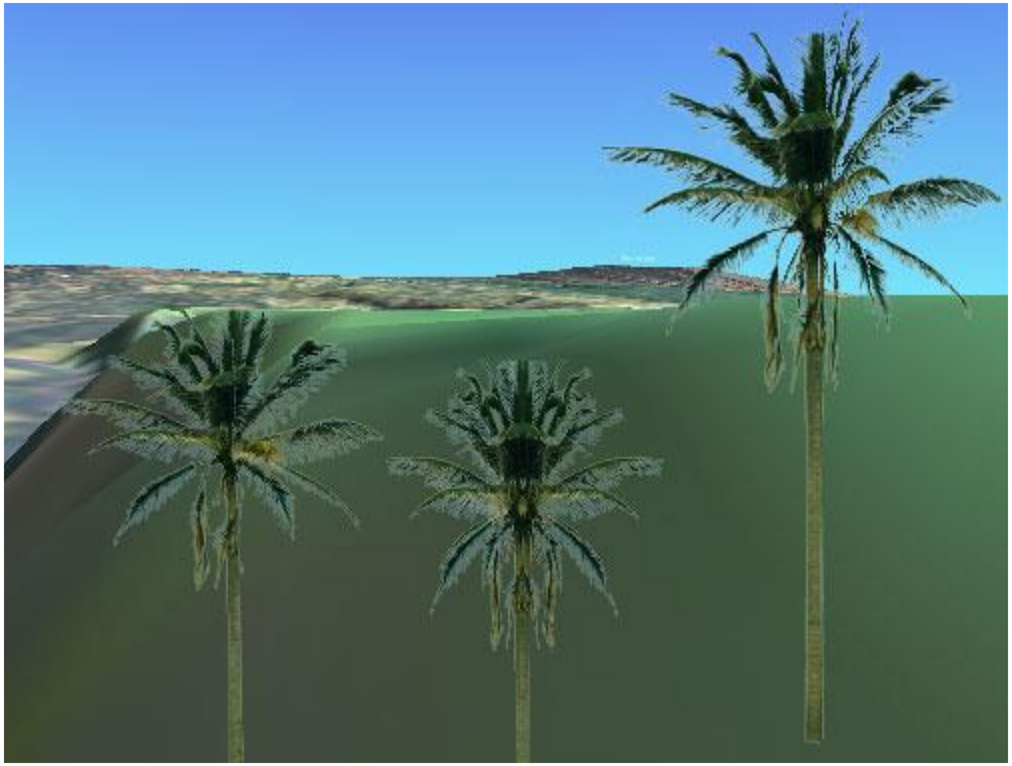
Most used blending modes

- **Standard blending**
  - source alpha * new pixel + (1 - source alpha) * old pixel

- **Additive blending**
  - source alpha * new pixel + old pixel

Texture Sampling and Transparency

- **Bilinear filtering samples rgb + alpha**
- **At alpha borders samples rgb values with alpha 0**



Premultiplied Alpha

- **Multiply rgb with alpha**
- **Fixes texture sampling (invisible pixels are multiplied with 0)**
- **Fixes sunglasses**
  - Premultiply alpha, then add something
  - Combines standard and additive blending
- **Blending mode:**
  - new pixel + (1 - source alpha) * old pixel

Vertex Shader

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Calculates vertex transformations**
- **Prepares additional data for later shader stages**
- **See Exercise 3**

Fragment Shader

- **Also called Pixel Shader**
- **Uses interpolated data from vertex shader**
- **Calculates colors**
- **See Exercise 4**

Vertex Buffer

- **Array of vertices**
- **Can hold additional data per vertex**
- **Has to assign additional data to names or registers for vertex shader**
- **Primary interface from CPU to GPU**

Index Buffer

- **Array of indices**
- **That‘s it**

Draw Calls

- Set Vertex Shader
- Set Fragment Shader
- Set IndexBuffer
- Set Vertex Buffer

- DrawIndexedTriangles()
- DrawIndexedTriangles()
- …

Implicit Work

- **Create command buffers**
- **Verify data**
- **(compile shaders)**
- **…**

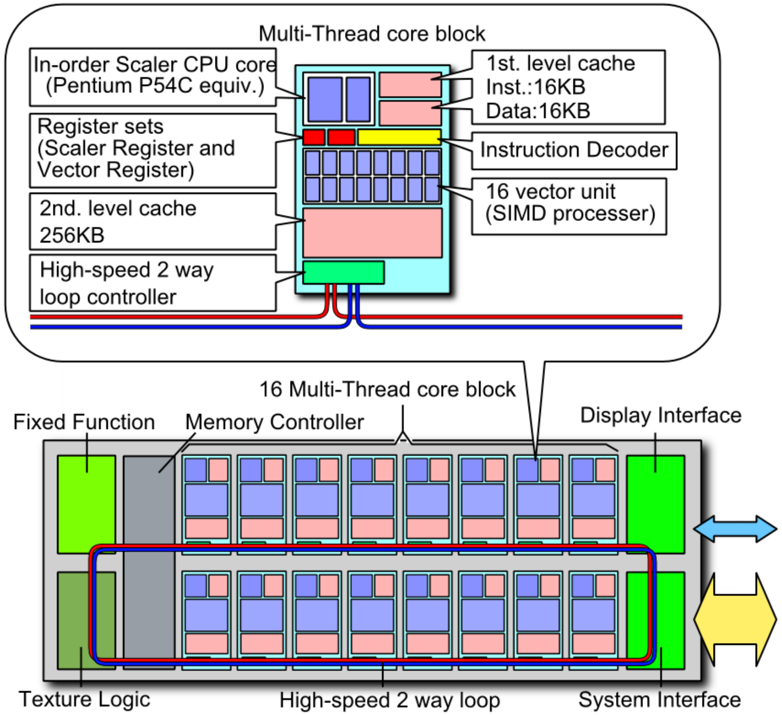
Compute Shader

TECHNISCHE UNIVERSITÄT DARMSTADT

- **No Rasterization**
- **Additional options for data synchronization**
- **Not yet supported everywhere**
- **Many competing languages**
  - Even OpenCL and GLSL compute shaders

Triangles on Compute

- **Xeon Phi**
  - Ex project Larrabee



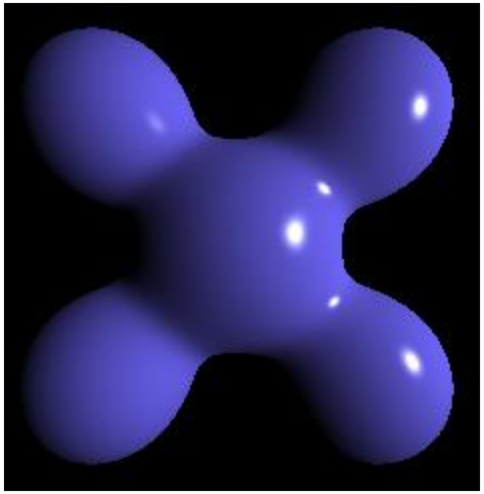
- **https://code.google.com/p/cudaraster/**
  - From nVidia

More Shaders

- **Geometry Shader**
  - Works on complete triangles
- **Tesselation Shader**
  - Can create new triangles
- **Not yet supported on all hardware**
  - Notably no support on iOS

Phong Lighting

- **color = ambient + diffuse + specular**
  - Note: Light from different sources can always be added just like that



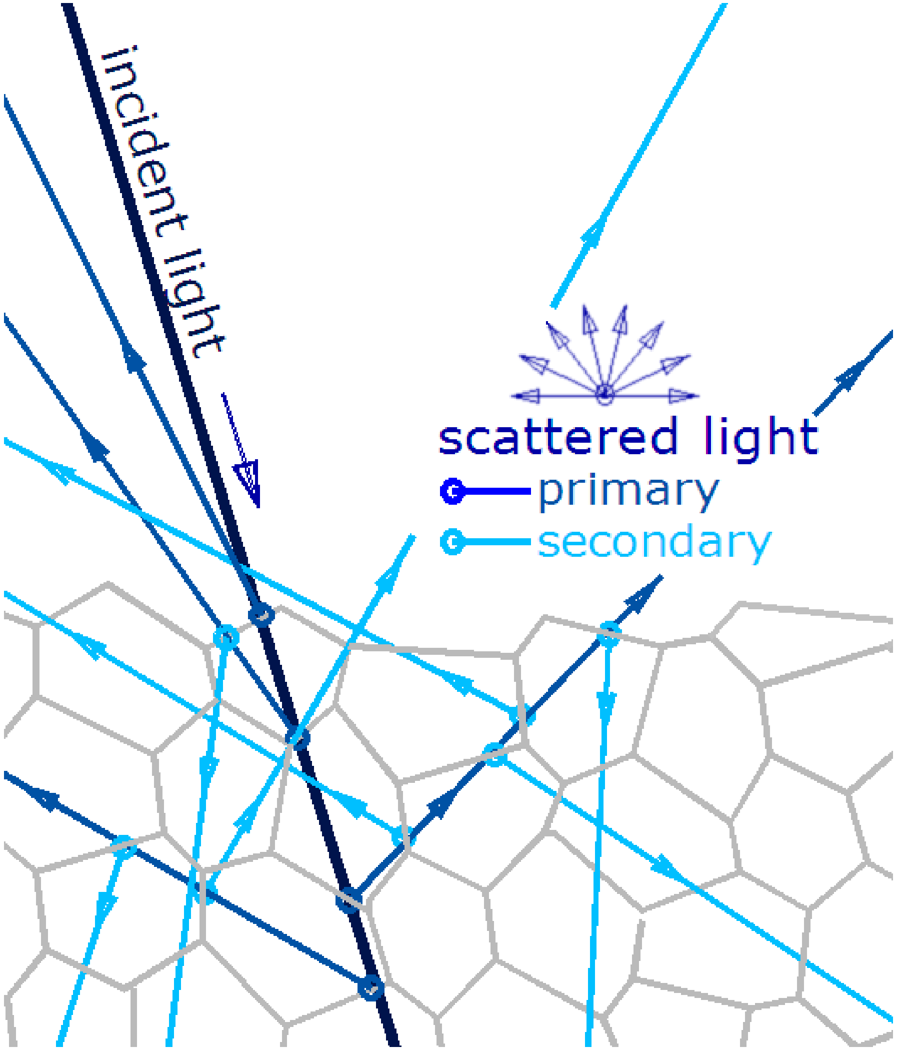
Ambient

TECHNISCHE UNIVERSITÄT DARMSTADT

Ambient

- **ambient = constant**

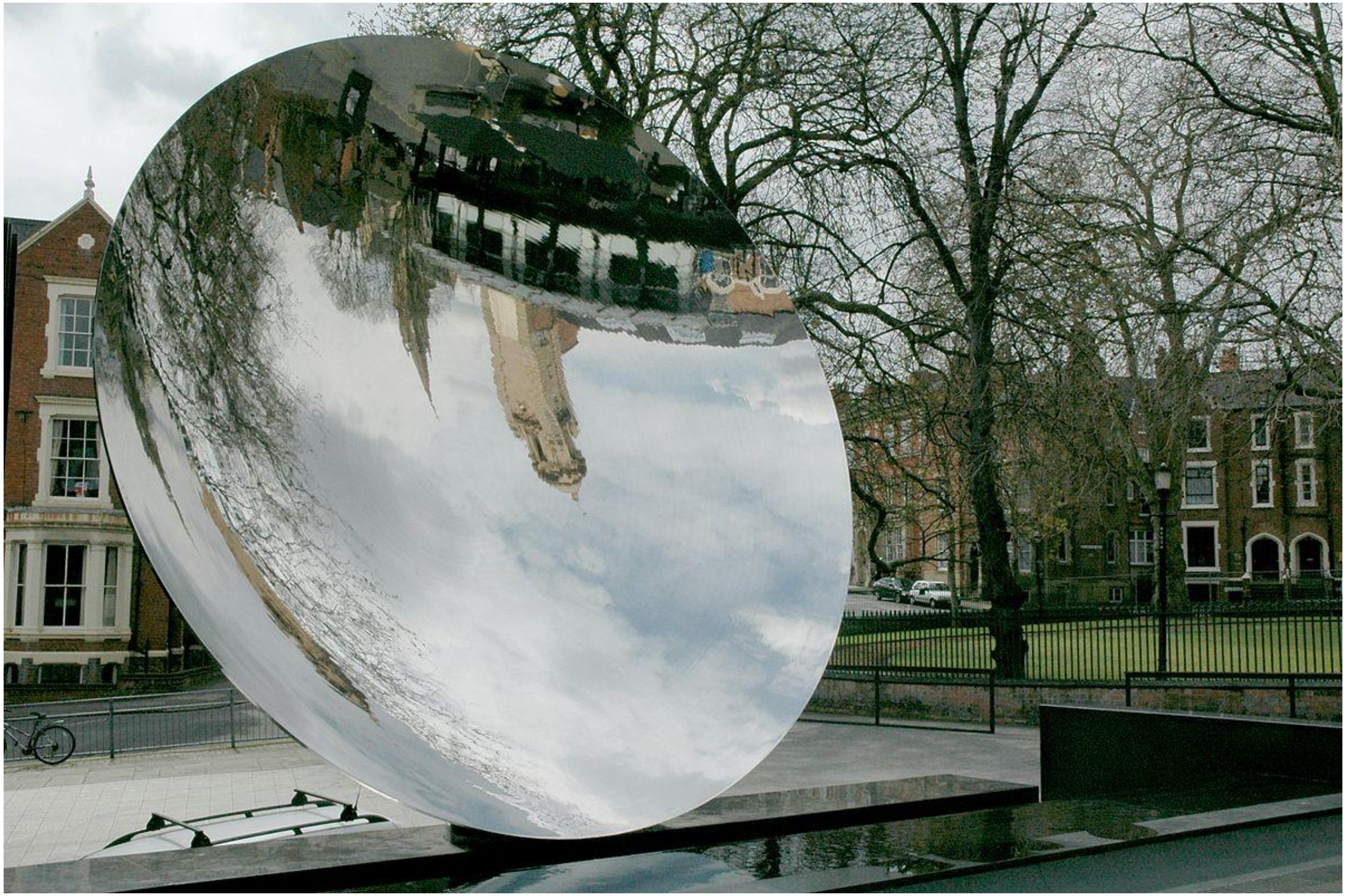
Diffuse



Diffuse

- **diffuse = LN (see previous lecture)**

Specular



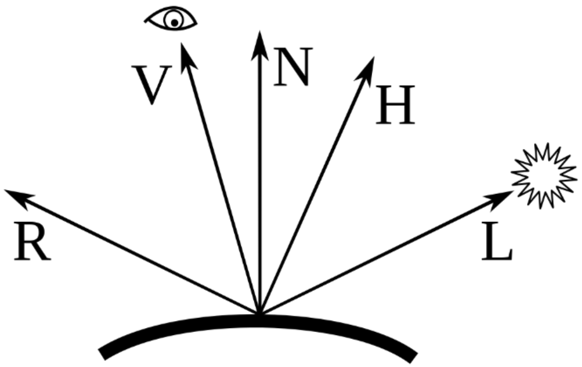
Specular

$$I_{\text{specular}} = I_{\text{in}}\, k_{\text{specular}} \cos^{n} \theta$$

$$= I_{\text{in}}\, k_{\text{specular}} (\vec{R} \cdot \vec{V})^{n}$$

- **R: mirrored vector to the light source (reflectance vector)**
- **V: vector to the camera**
- **n: roughness – start at 32 and tune**

- **Empirical model (aka basically nonsense)**
- **Ugly for larger angles (cos -> 0)**

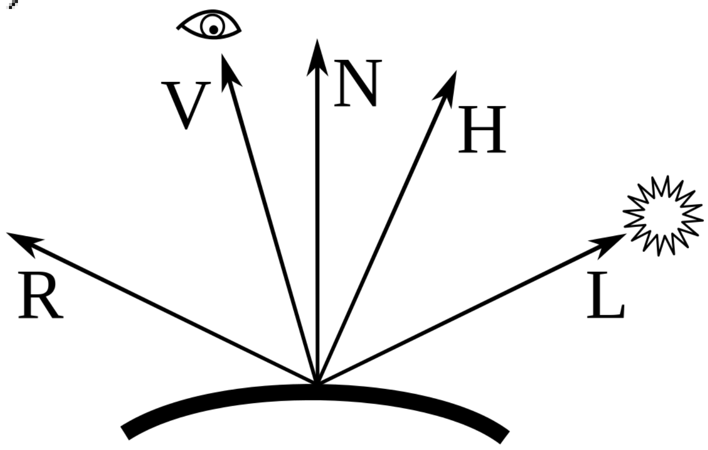


Blinn Phong

$$H = \frac{V + L}{\|V + L\|}$$

$$\begin{aligned} I_{specular} &= I_{in} \cdot k_{specular} \cdot \cos^n \theta' \\ &= I_{in} \cdot k_{specular} \cdot \left(\frac{(V+L) \cdot N}{\|(V+L)\| \cdot \|N\|} \right)^n \end{aligned}$$

- **A little faster**
- **A little nicer**



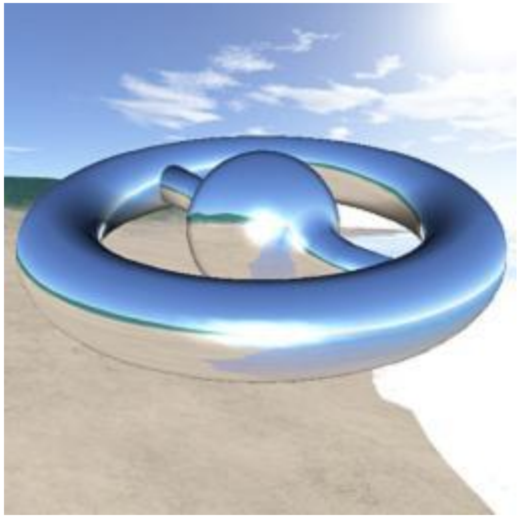
Better ambient light

- **Real ambient light is hard**
  - Light bouncing and bouncing and bouncing…
- **Ambient light tends to look very diffuse**
  - No hard borders
- **Precompute everything**
  - Put it in small textures
  - Bilinear filtering blurry stuff works wonderfully

Light Baking

Better specular lighting

- **Render six orthogonal perspectives into a cube map**
  - Camera center = center of object to be rendered
- **Sample vector into cubemap for every pixel**
- **Obviously very expensive**
- **Can not be precomputed**



Ambient, Diffuse…

- **Thinking of „Ambient“ is only an approximation**
  - Phong lighting is an approximation of an approximation

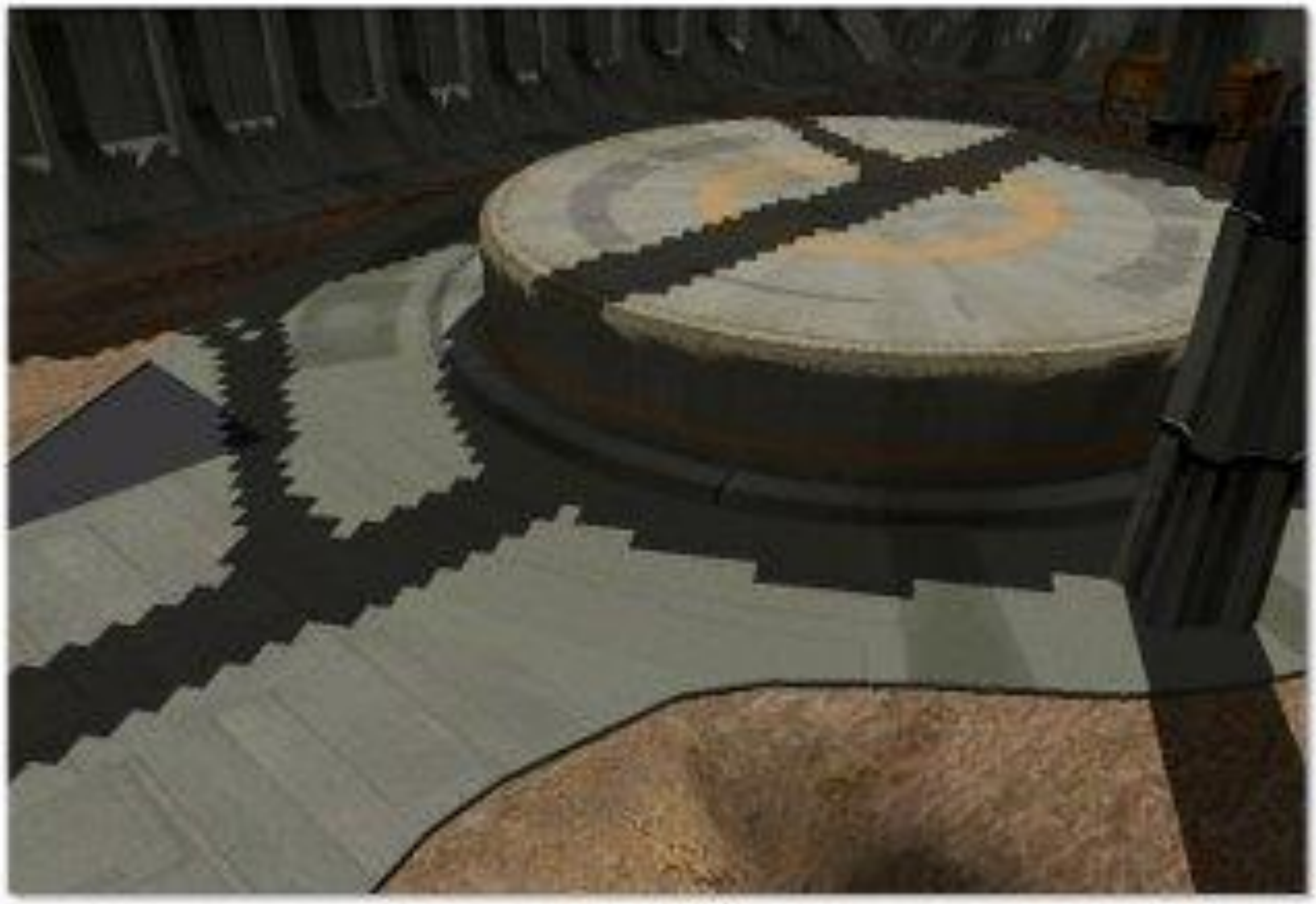
- **Light bounces around**
  - First bounce -> direct lighting (use diffuse and specular)
  - Second bounce -> hard shadows
  - More bounces -> ambient light

Shadow Mapping

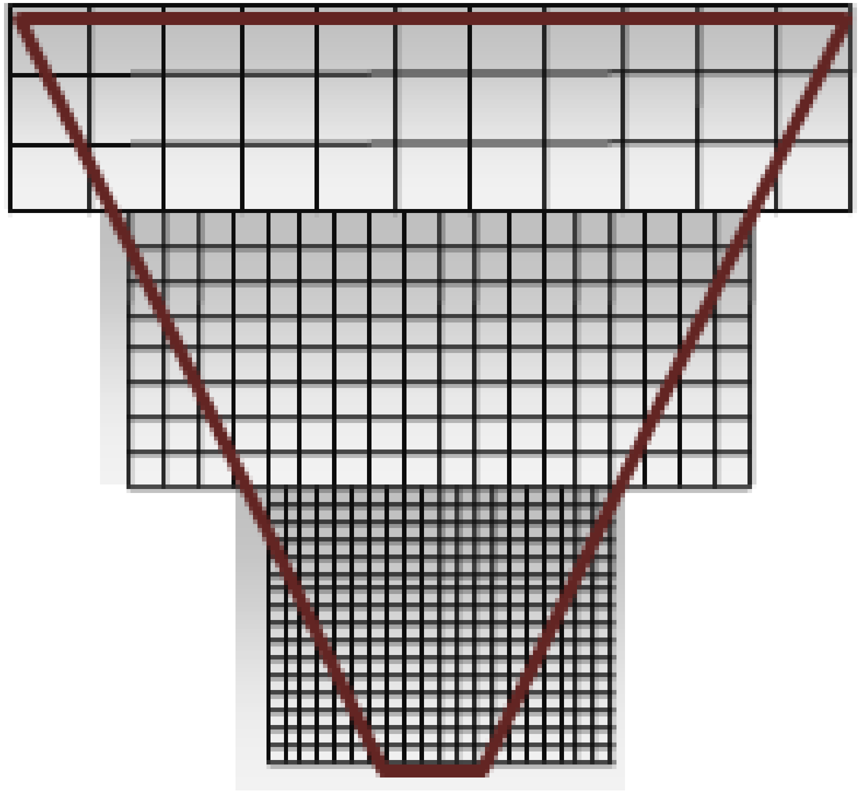
- **Set camera to light source**
- **Render depth -> each pixel value = distance from light**

- **During regular rendering**
- **Transform vertices two times**
  - Using camera position
  - Using light position -> z = distance from light
- **Read depth texture**
- **Compare depth calculated using light pos and depth from texture**
  - If greater -> in shadow

Shadow Mapping



Cascaded Shadow Maps



GLSL

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Similar to C**
- **Semiautomatic parallelization**

GLSL Example

```
uniform sampler2D tex;
varying vec2 texCoord;
varying vec4 color;

void kore() {
  vec4 texcolor = texture2D(tex, texCoord) * color;
  texcolor.rgb *= color.a;
  gl_FragColor = texcolor;
}
```

Vertex Shader

- **Tansforms vertices**

- **Writes transformed vertex to special var**
  - gl_Position

- **Can write additional data**

Fragment Shader

- **Writes final color to special var**
  - gl_FragColor

- **Can not write additional data**
  - Mostly (multi target rendering, gl_FragDepth,… - not on all hardware)

Parallelism

- **Vertex shader defines one function**
  - which is applied to lots of vertices in parallel

- **Fragment shader defines one function**
  - which is applied to lots of pixels in parallel

- **Programming model allows hardware to parallelize automatically**
  - To multiple compute cores, SIMD units or weird combinations of both

Uniforms

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Constants**
  - Do not change while shader executes
  - Can be changed between draw calls
- **uniform mat4 projectionMatrix;**
- **uniform sampler2D tex;**

Attributes

- **Vertex shader input**
- **Defined in Vertex Buffer**

- **attribute vec3 vertexPosition;**
- **attribute vec2 texPosition;**
- **attribute vec4 vertexColor;**

Varyings

- **Transfer data between shader stages**
- **Vertex shader -> interpolation -> Fragment shader**
- **Output in vertex shader, input in fragment shader**
- **varying vec2 texCoord;**

Vector types

- **vec3 position;**
- **vec4 color;**

- **Support basic arithmetic**
- **Support swizzling**
  - color.bgr
  - position.xy

Matrix types

- **mat4 projection;**
- **Supports arithmetic with vectors**

Samplers

- **To read textures**
- **uniform sampler2D tex;**
- **vec4 texcolor = texture2D(tex, texCoord);**

Special vars

- **gl_Position**
- **gl_FragColor**

- **https://www.opengl.org/wiki/Built-in_Variable_(GLSL)**
  - There are many more
  - (so many actually that they forgot one)

Precision modifiers

- **precision mediump float;**

- **Precision can be reduced**
  - Often makes sense in the fragment shader
  - And is often necessary (OpenGL ES)

GLSL versions

- **Up to version 4.5**
- **Different versions for OpenGL ES**

- **Kore uses „GLSL ES“**
  - GLSL version used by OpenGL ES 2.0 and WebGL
  - GLSL 1.1 plus some 1.2

GLSL in Kore

- **main is called kore**
  - Only difference to real GLSL
- **To make things easier in Windows use**
  - node Kore/make gfx=opengl2
  - Optionally debug Direct3D later
    - Deletes your varyings in the fragment shader when they are not used, which breaks shader linkage
- **Shader compiled automatically in Visual Studio**
  - Not in XCode or Code::Blocks
  - Optionally directly work with the files in Deployment
    - Beware: A call to koremake overwrites them

Kore Graphics

- #include <Kore/Graphics/Graphics.h>

- Straight forward api

- Set uniforms ala
- ConstantLocation loc = program->getConstantLocation(„bla“);
- Graphics::setFloat(loc, 2.0f);

- Coordinate system is (-1 to 1, -1 to 1, -1 to 1) like in OpenGL