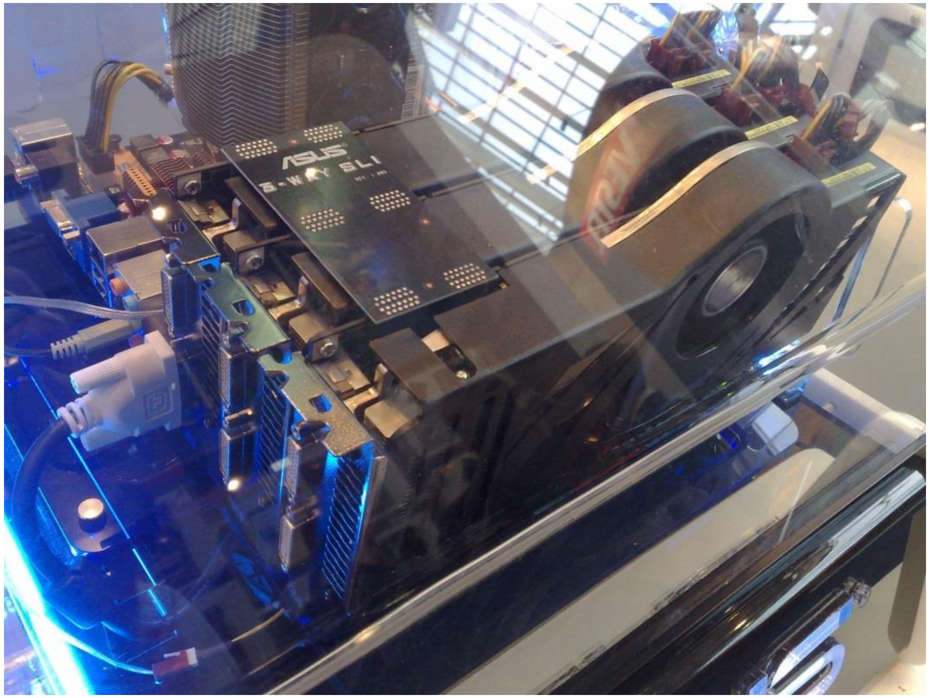
Game Technology

Lecture 5 – 28.11.2015
Hardware Rendering



Dr.-Ing. Florian Mehm

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

Organization

Date	Lecture	Topic
24.10.2015	1	Input and Output
	2	The Game Loop
	3	Software Rendering
	4	Advanced Software Rendering
28.11.2015	**5**	**Basic Hardware Rendering**
	6	**Bumps and Animations**
	7	**Physically Based Rendering**
	8	**Physics 1**
19.12.2015	9	Physics 2
	10	Procedural Content Generation
	11	Compression and Streaming
	12	Multiplayer
23.1.2016	13	Audio
	14	Artificial Intelligence
	15	Scripting

Organization

Lecture recordings

- Available on the wiki: https://wiki.ktxsoftware.com

Exercises from last block

- Exercise 1 corrected
- Will be uploaded to your git repository
- Groups which uploaded incorrectly were informed

New exercises

- 3 exercises until next block

Organization

Next block: 19.12.2015

- Sorry about the date!
- Recordings will be available soon after the block
- No exercise scheduled for winter break (but will respond to feedback during the break if you want to work)

Ludum Dare@KOM

TECHNISCHE UNIVERSITÄT DARMSTADT

Game Jams

- Game development contest
  - Vague theme (e.g. “10 seconds”)
  - Tight time constraints (e.g. 48 h)
  - Starting from scratch (design, assets, code, …)
- No excuses – just submit something…

The Head Wizards Course, 2014

Ludum Dare 30 @ TUD, 2014

A Maze Thing, 2013

10 Seconds to Apocalypse, 2013

Ludum Dare 34@TUD

- Sa., 12.12.2015, 9:00 –
  Mo., 14.12.2015 (night)
- Registration (first-come-first-serve):
  gamejam@kom.tu-darmstadt.de

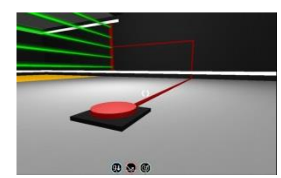
10sion, 2013

10Up Experiments: Mountain Brew, 2014

Ludum Dare

The Most Important 10 Seconds Of Your Life, 2013

As We Are, 2014



Neon Multiverse, 2014


ja!


Saskia

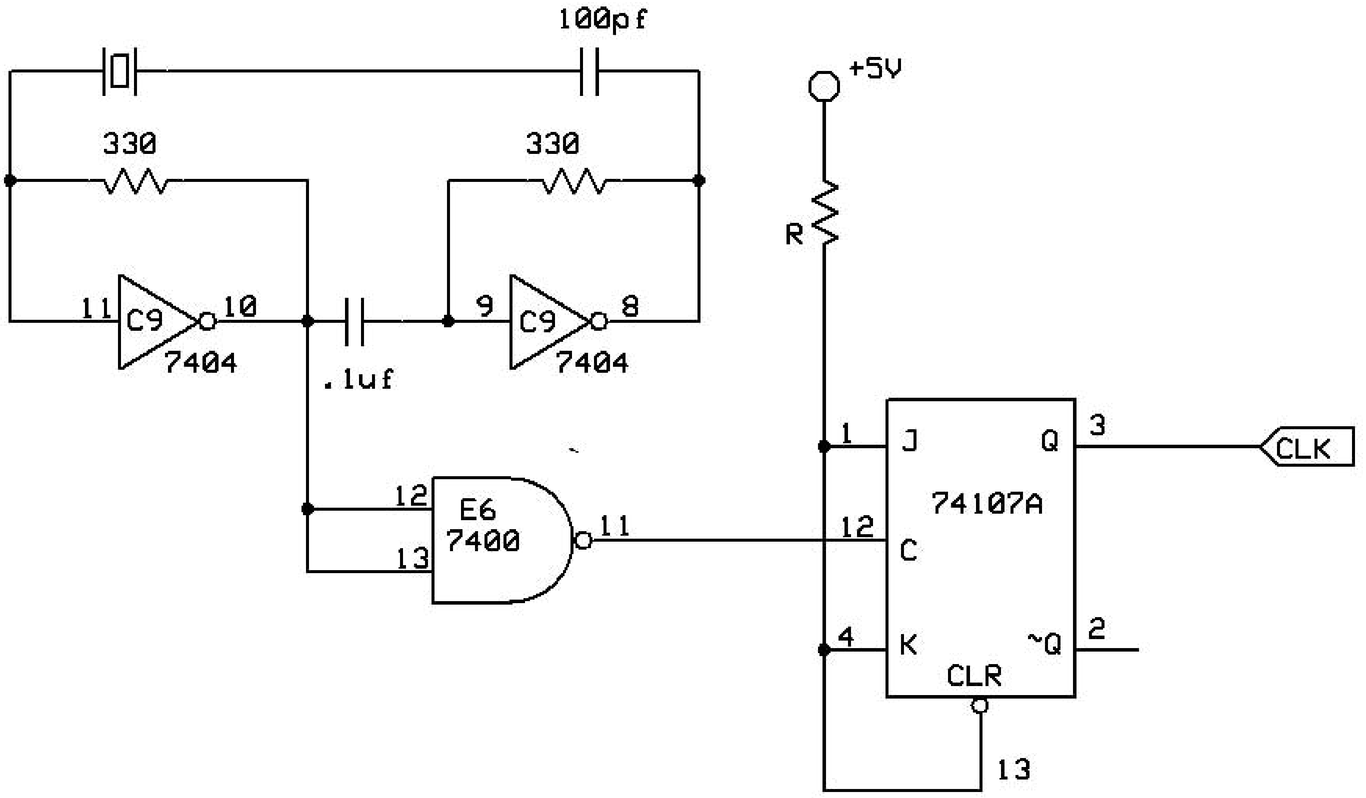

beam
DEAD PIXEL SOCIETY
INVERTIGO
CORE
CORE

TIME TO APOCALYPSE
2.912
PAUSE
Left Mouse
Shoot
Right Mouse
Grapple
Ouch, my head ...
Afterimage
ALWAYS LOOK HERE
Arrowkeys to move
(after timer ends)
AS WE ARE
Use SPACE or "2" to switch to your new player
Team2See
Look at
Pick up
YOU JUST DIED!
Mr. B
Cannibal Cage
Start Game
show inventory
take axe
Have Fun!!
Mind your Decision
Floor 1
Floor 0
Start Game
Tutorial
Credits
Quit
MONSTER
ARENA
Start
YOU ARE THE MONSTER

Pong & Computer Space

Pong (1972), Computer Space (1971)

Pong “Game Engine”



Pong (1972), Clock Generator

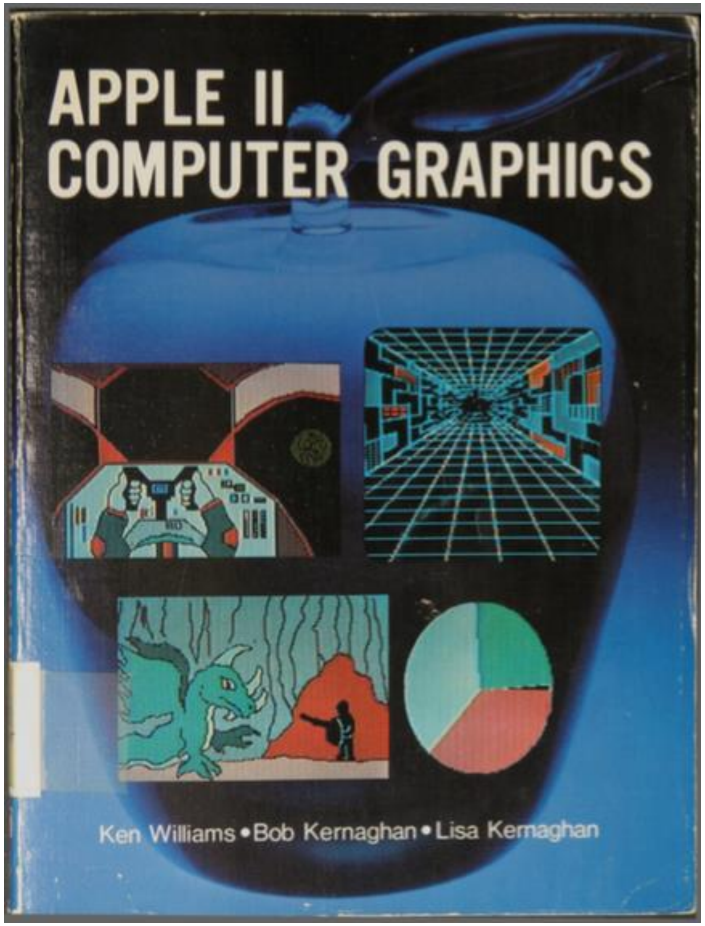
Apple 2 (1977)

Apple II

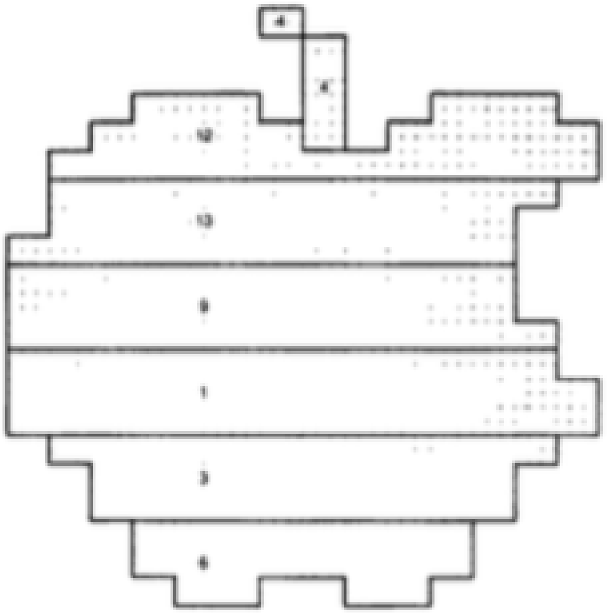
One of the first mass-produced home computer with CG capabilities

- Quirky hardware and software interface
- But: Gave rise to first home graphical games

Mystery House (1980)



Apple II Graphics



Apple II Graphics (Low-res mode)

```
20 REM
30 GR
40 COLOR= 4
50 PLOT 20,10
60 VLIN 11,14 AT 21
70 COLOR= 12
80 HLIN 17,19 AT 13
90 HLIN 24,26 AT 13
100 HLIN 16,20 AT 14
110 HLIN 23,27 AT 14
120 HLIN 15,27 AT 15
130 COLOR= 13
140 HLIN 15,26 AT 16
150 HLIN 15,25 AT 17
160 HLIN 14,25 AT 18
170 COLOR= 9
180 HLIN 14,25 AT 19
190 HLIN 14,25 AT 20
200 HLIN 14,26 AT 21
210 COLOR= 1
220 HLIN 14,26 AT 22
230 HLIN 14,27 AT 23
240 HLIN 14,27 AT 24
250 COLOR= 3
260 HLIN 15,26 AT 25
270 HLIN 16,25 AT 26
280 HLIN 16,25 AT 27
290 COLOR= 6
300 HLIN 17,24 AT 28
310 HLIN 17,24 AT 29
320 HLIN 18,19 AT 30
330 HLIN 22,23 AT 30
```

Atari VCS (1977)

Atari VCS

Later renamed to Atari 2600

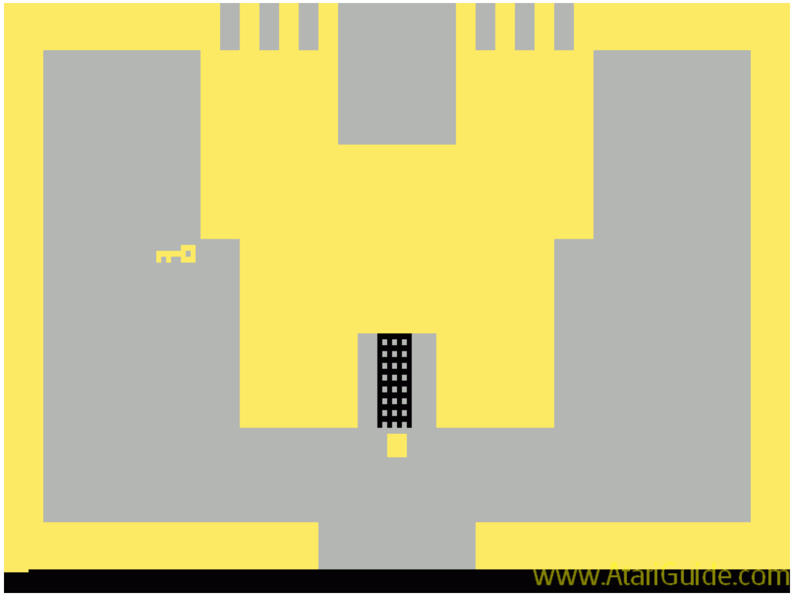
MOS Technologies 6507

- Variant of 6502: Addressable memory reduce from 64 kB to 8 kB
- ~1,19 MHz

Developers had to be very creative

- E.g. build mirrored levels
- Use the timing of the monitor to switch colors in one frame
- Use undocumented features

More info: “Racing the Beam: The Atari Video Computer System”



Adventure (1979)

Nintendo Entertainment System/Famicom (1983)

Nintendo Entertainment System/Famicom

CPU: Ricoh 2A03 (6502-base) @ 1,77 MHz (PAL) / 1,79 MHz (NTSC)

Graphics: PPU Ricoh-Chip (NTSC: RP2C02, PAL: RP2C07) @ 5,37 MHz bzw. 5,32 MHz

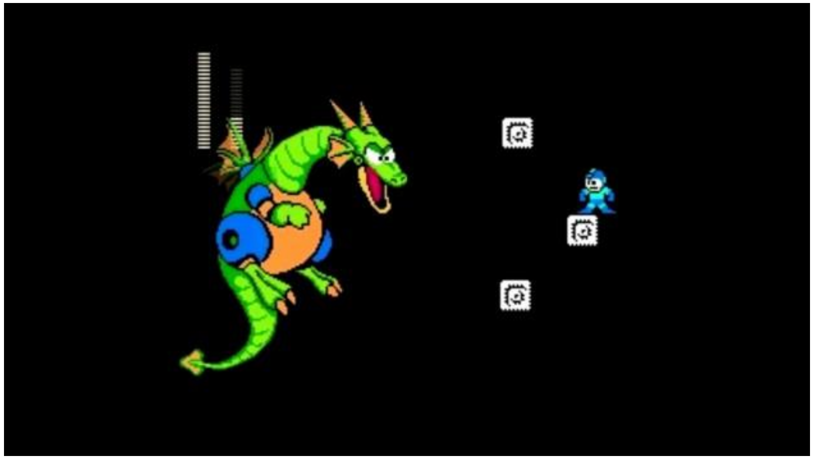
CPU: Not much difference to VCS

- But built for better handling of sprite, tiled rendering

NES quirks

Sprite flickering

- Emulated in Mega Man 9 (2008)
- Happened when too many sprites were being drawn

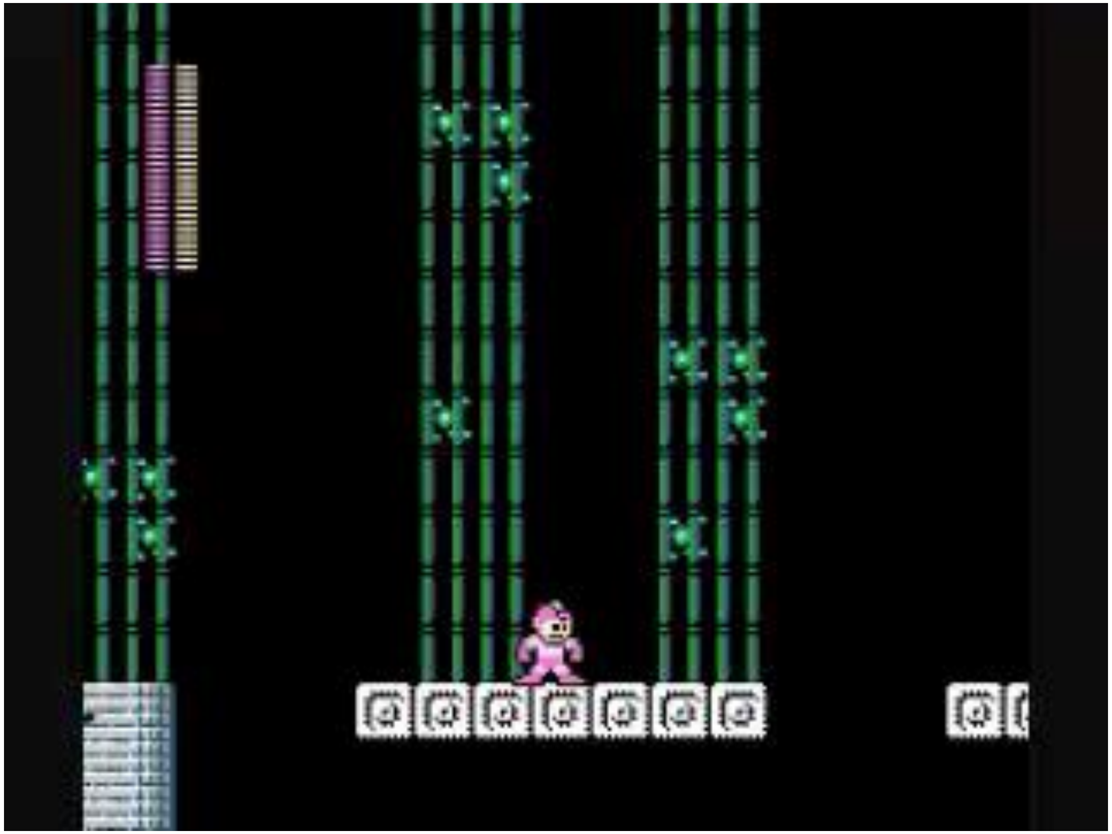


Mega Man 2 (1988)

Limited memory

- Intended for tiled backgrounds
- Sprites only small elements
- Mega Man boss fights: Black background for memory reasons

NES Quirks

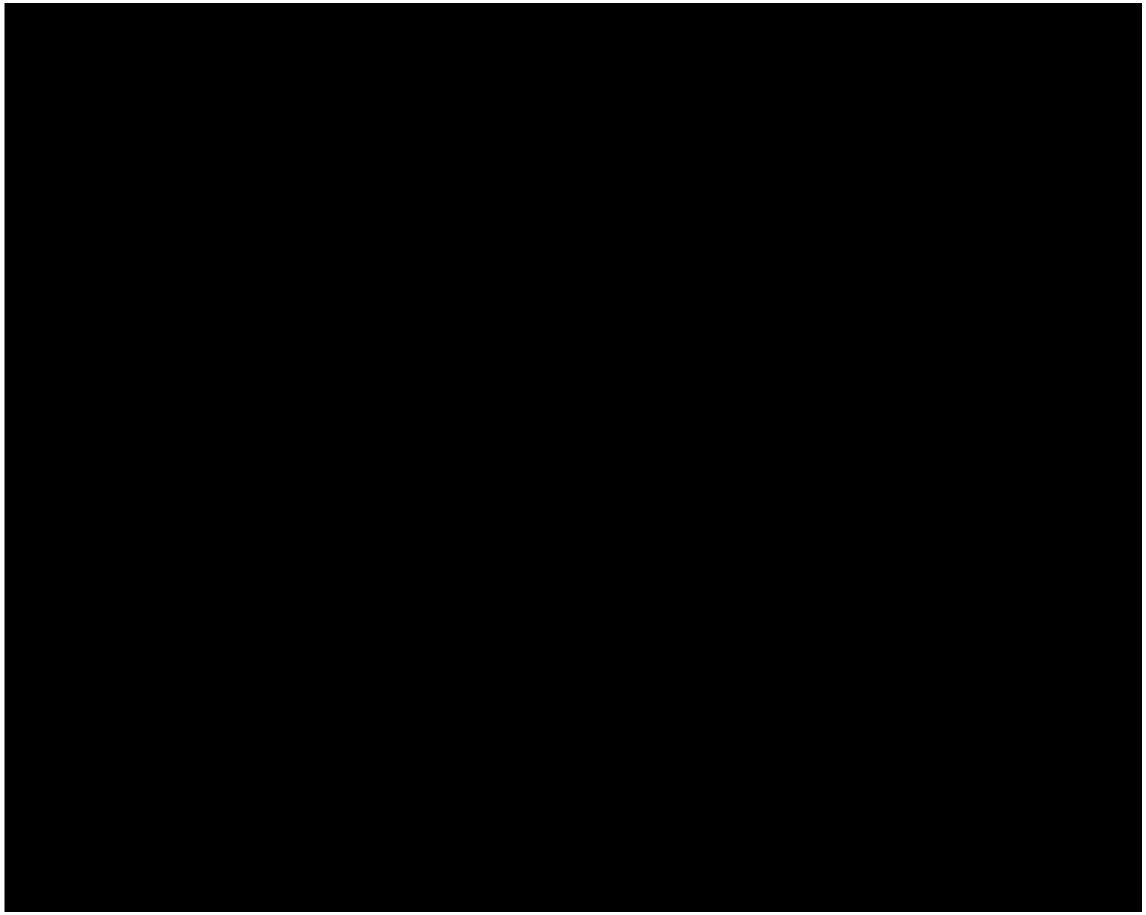


https://www.youtube.com/watch?feature=player_embedded&v=JrH5Q8gssvY

Commodore 64 (1982)

Amiga 500 (1987)

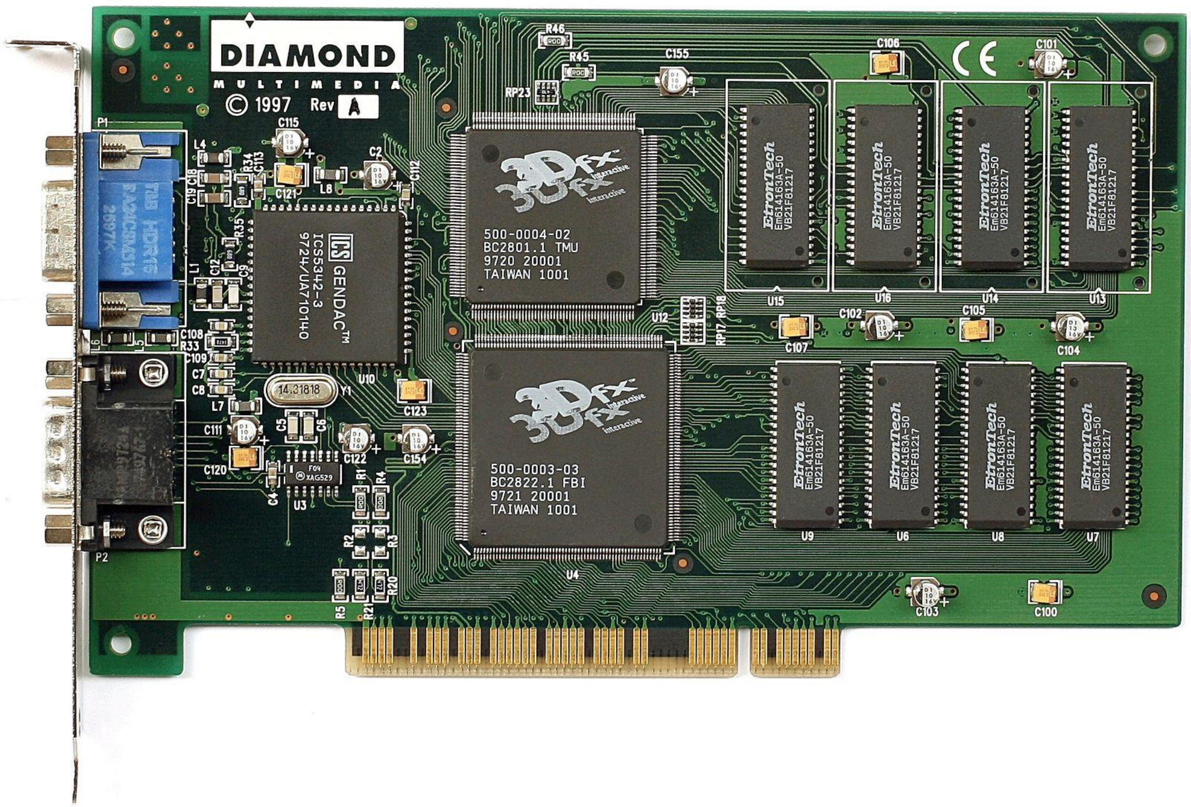
Origin (Complex), 1993



https://www.youtube.com/watch?v=MeoFaHW3nvw

IBM PC (1981)

Voodoo Graphics (1996)



Features of Voodo Graphics chip

Triangle raster engine

Linearly interpolated Gouraud-shaded rendering

Perspective-corrected (divide-per-pixel) texture-mapped rendering with iterated RGB modulation/addition

Detail and Projected Texture mapping

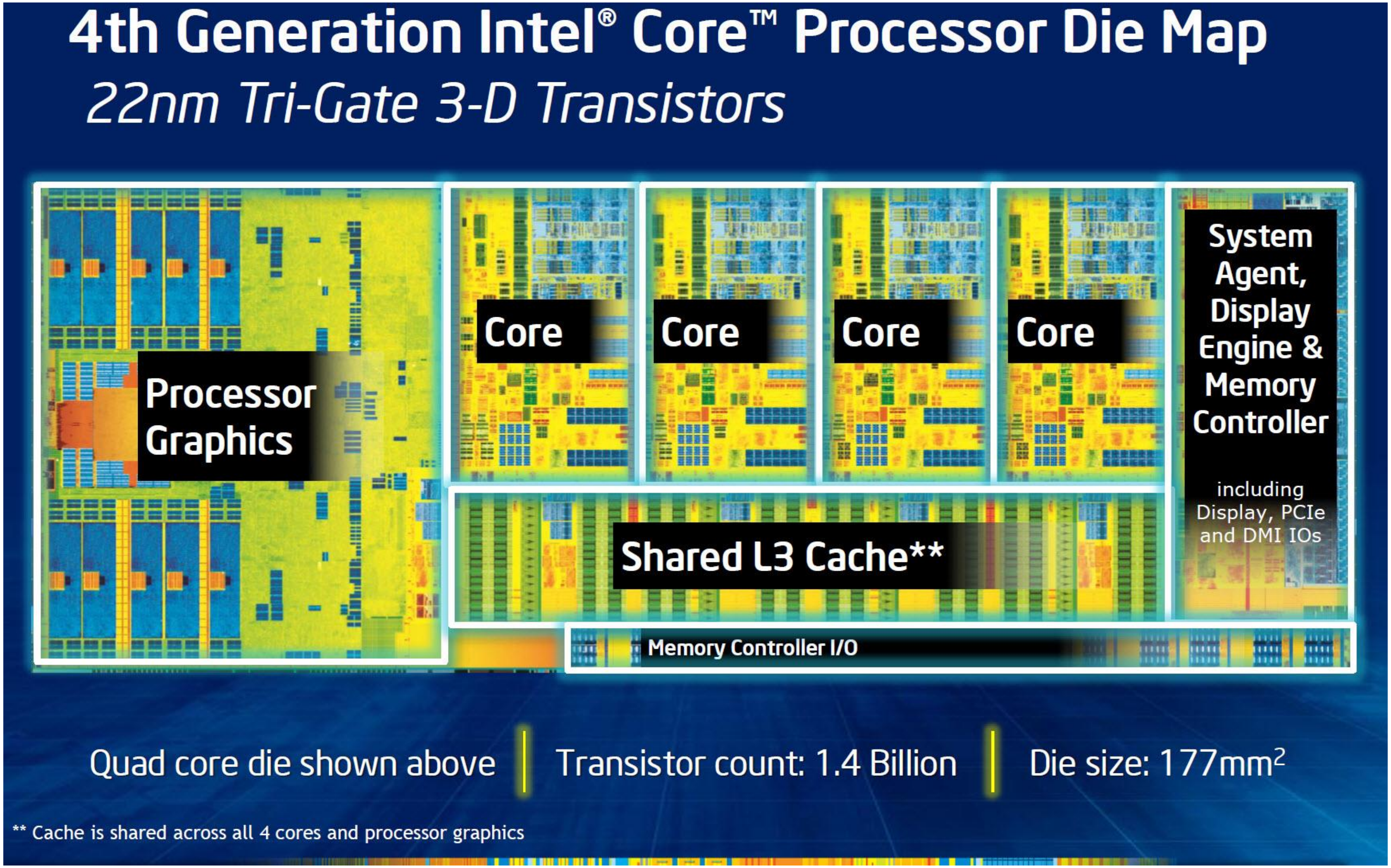
Linearly interpolated 16-bit Z-buffer rendering

Perspective-corrected 16-bit floating point W-buffer rendering (patent pending)

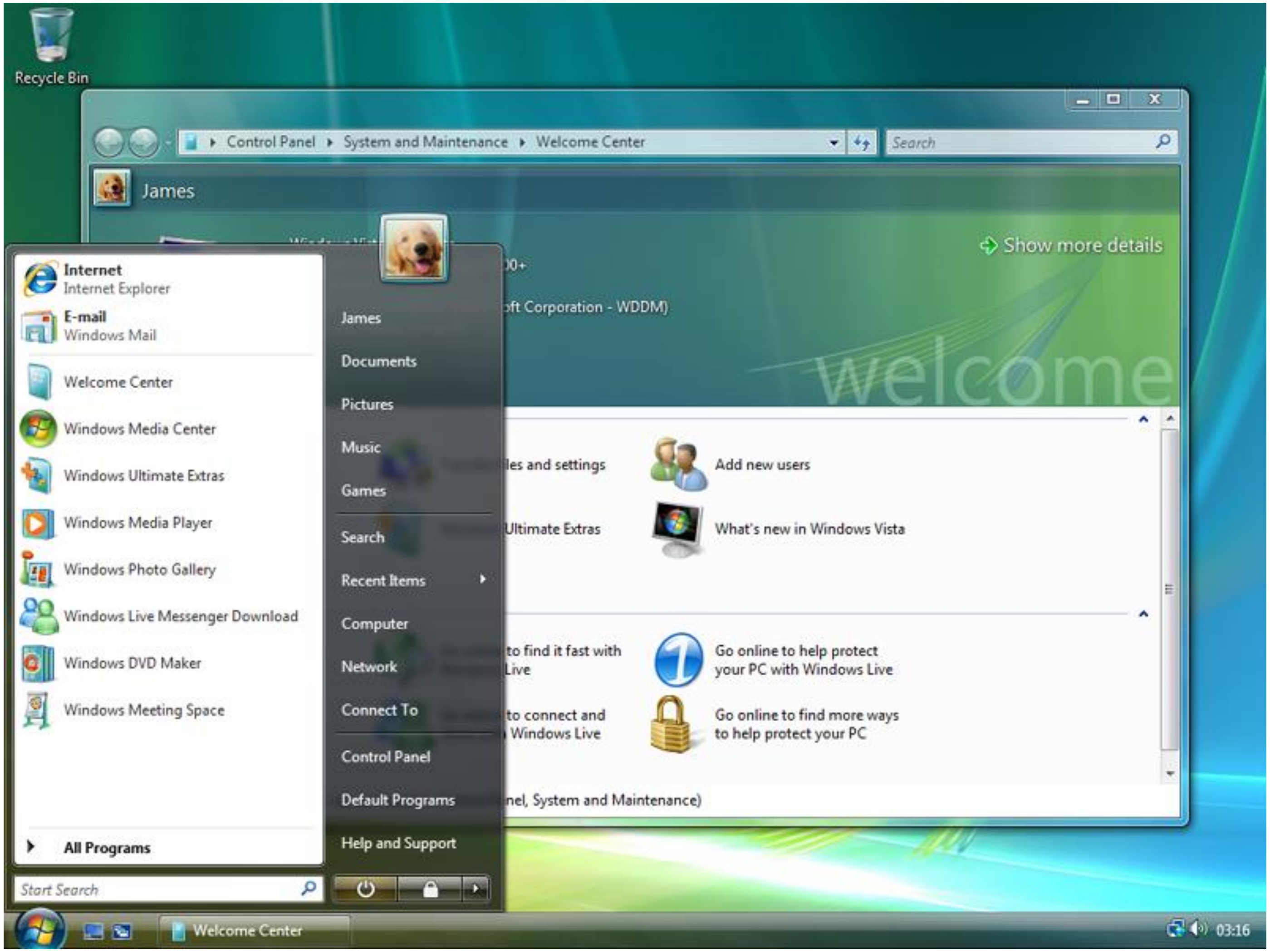
Texture filtering: point-sampling, bilinear, and trilinear filtering with mipmapping

…

Modern intel CPUs



Windows Vista (2007)



PS4

CPU vs GPU

TECHNISCHE UNIVERSITÄT DARMSTADT

CPU

- Run sequential code as fast as possible

GPU (Graphical Processing Unit)

- Massively parallel code execution
- Plus triangle rasterizer
- Plus texture sampler

GPGPU (General purpose computations on GPU)

- Programmable computing units, not directly tied to graphics anymore
- Carry out a computation massively parallelized

GPGPU

TECHNISCHE UNIVERSITÄT DARMSTADT

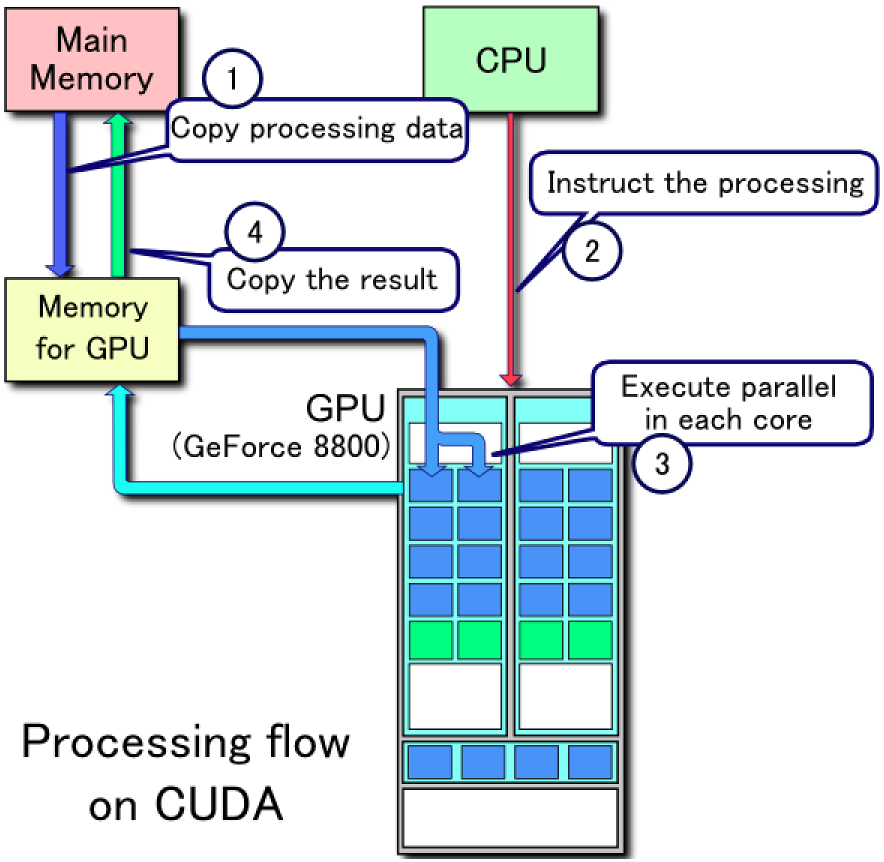
http://www.gdcvault.com/play/1022421/Ubisoft-Cloth-Simulation-Performance-Postmortem

Ideally suited for parallel tasks

- Adding many large vectors
- …

What if there are dependencies?

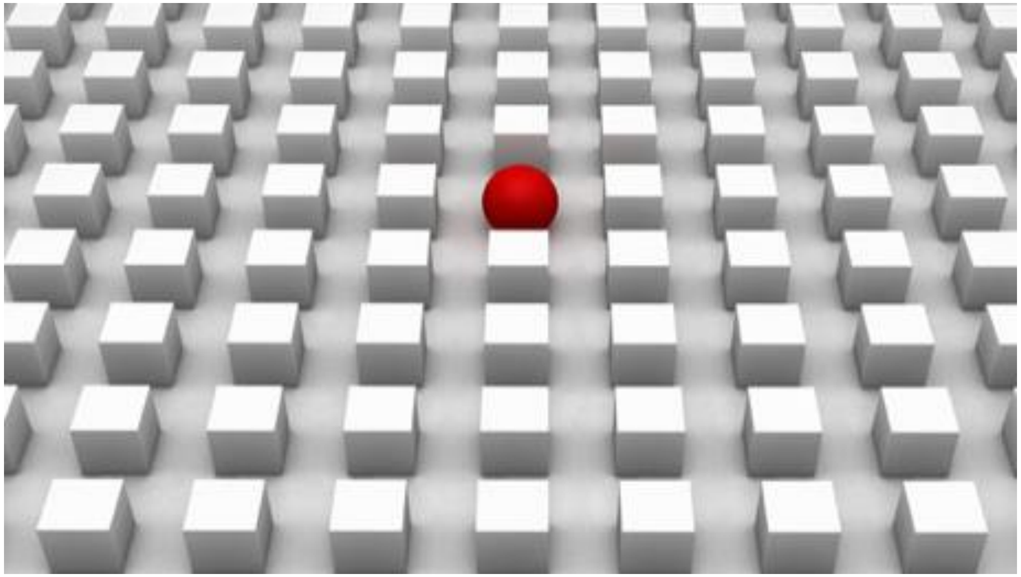
- Throw away some results
- Organize data better
- ...



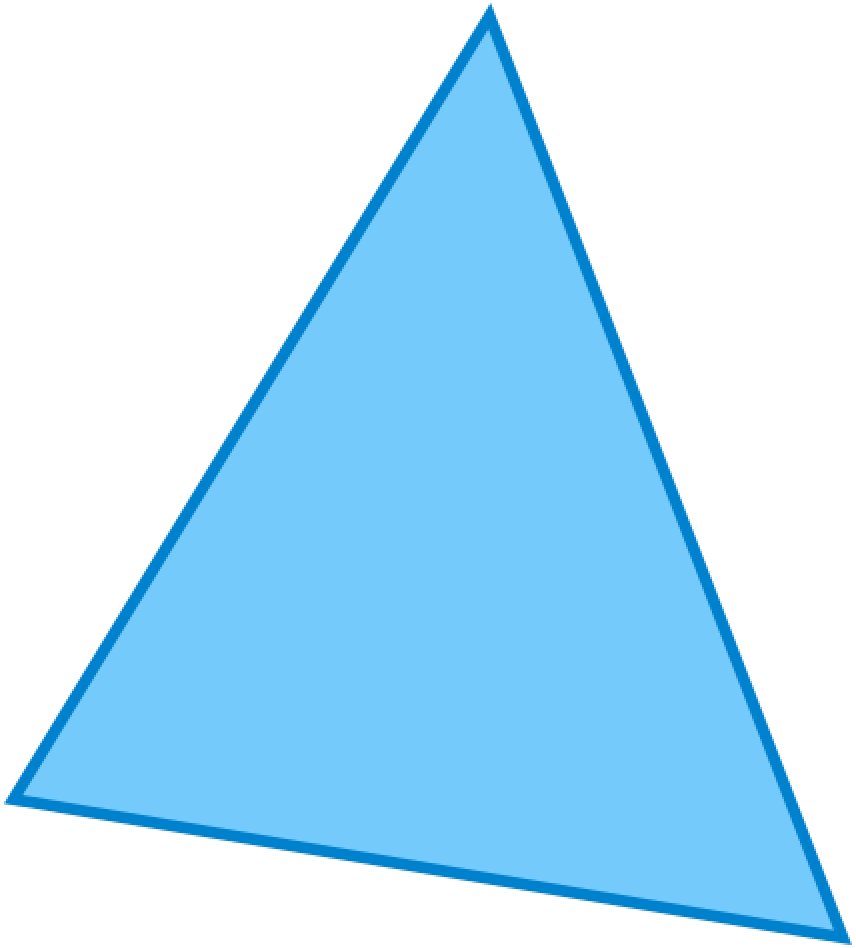
GPGPU

https://www.coursera.org/course/hetero

MOOC course “Heterogeneous Parallel Programming”
University of Illinois



Triangles

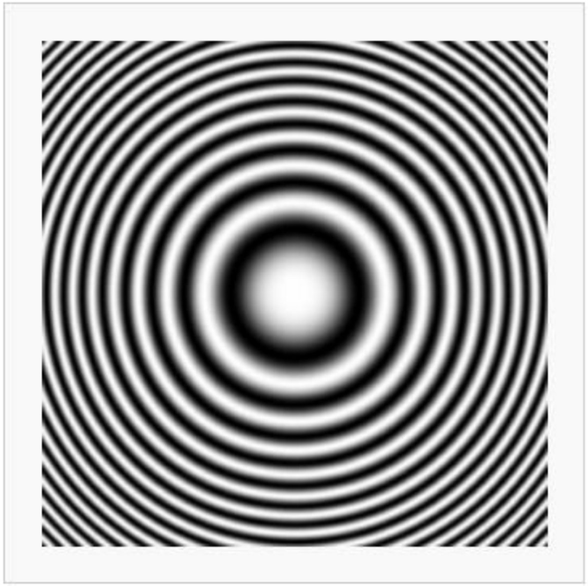
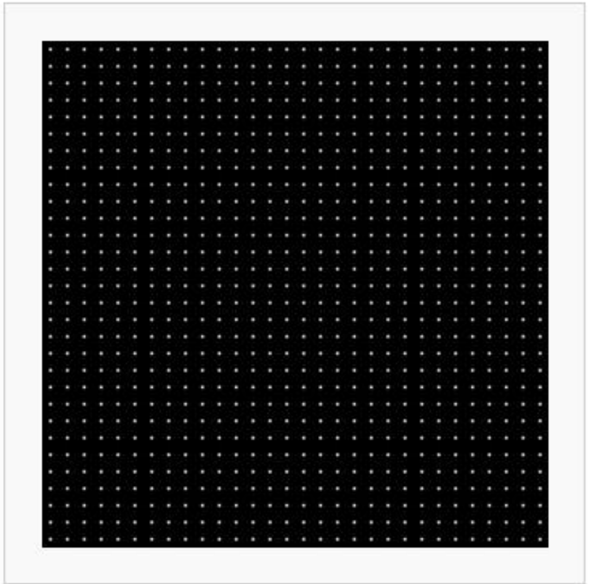
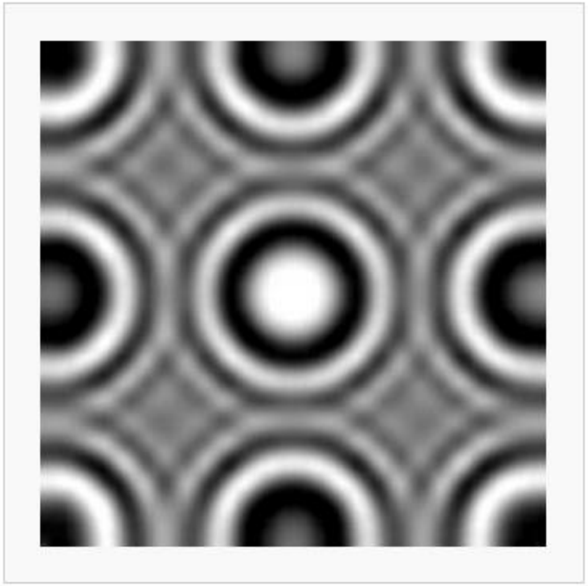


Aliasing

Aliasing

Sampling frequency is too low

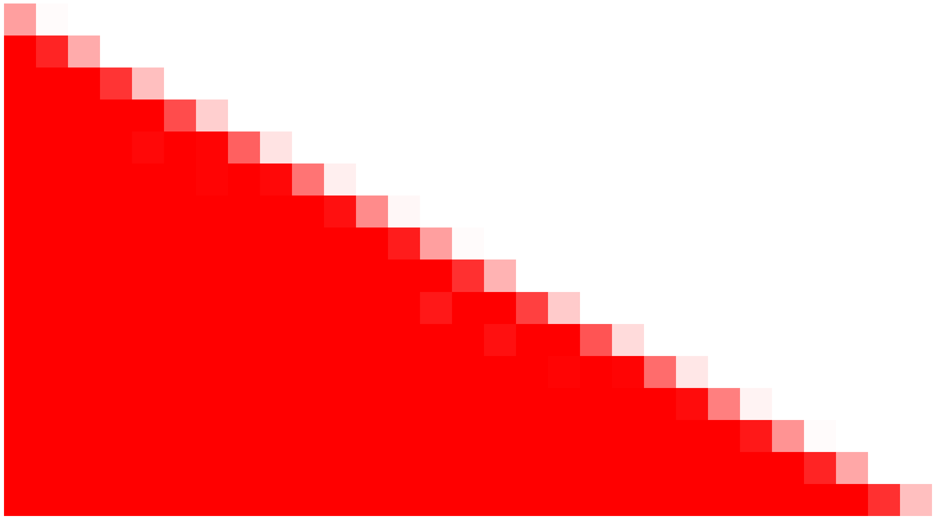
- Example: Original wave on the left
- Sample points in the middle
- Inaccurate sampled wave on the right

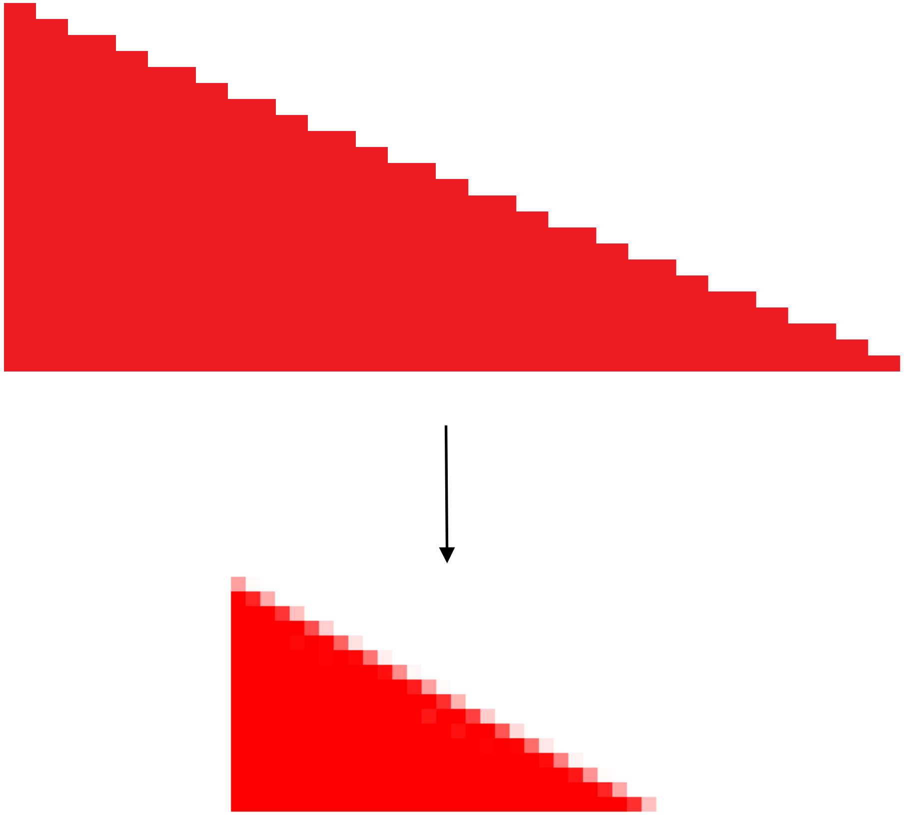
Edge Antialiasing

TECHNISCHE UNIVERSITÄT DARMSTADT

- Specifically work on edges
- Blur with the background
- Would require back-to-front rendering



Supersample Antialiasing



Multisample Antialiasing

https://www.youtube.com/watch?v=Nef6yWYu0-I

Postprocess Antialiasing

Temporal Anti-Aliasing

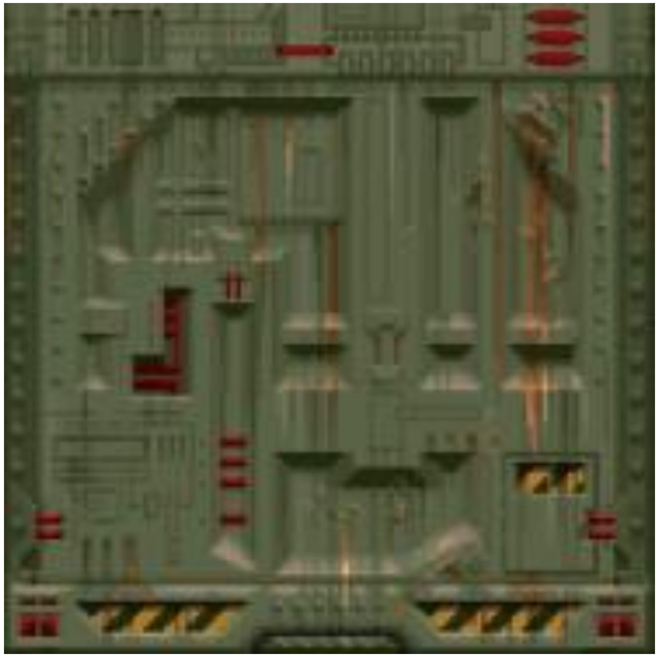
Anti-Aliasing done over several frames, to remove effects seen during motion

Textures

Basically images

Preferably $2^n * 2^n$

- Other sizes not necessarily supported
  - Expand image and fix up texture coordinates



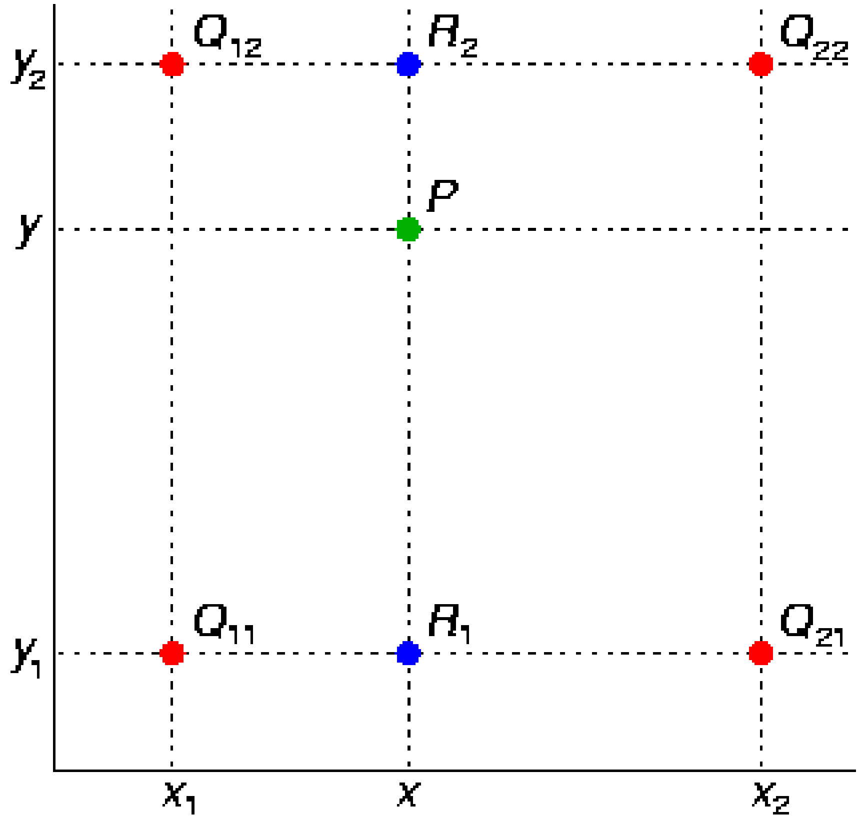
Texture Sampling

Point Filtering

Bilinear Filtering

- Interpolate four neighbouring pixels

Bilinear filtering



Mip Mapping

Example: Texture mapped to one pixel

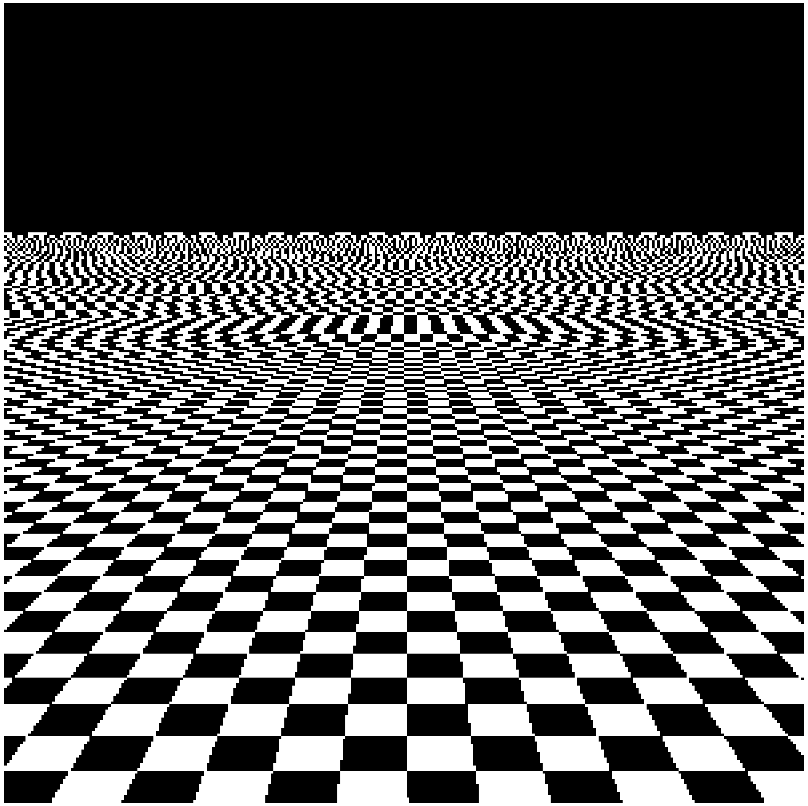
- Ideally calculate mean color value of the complete texture

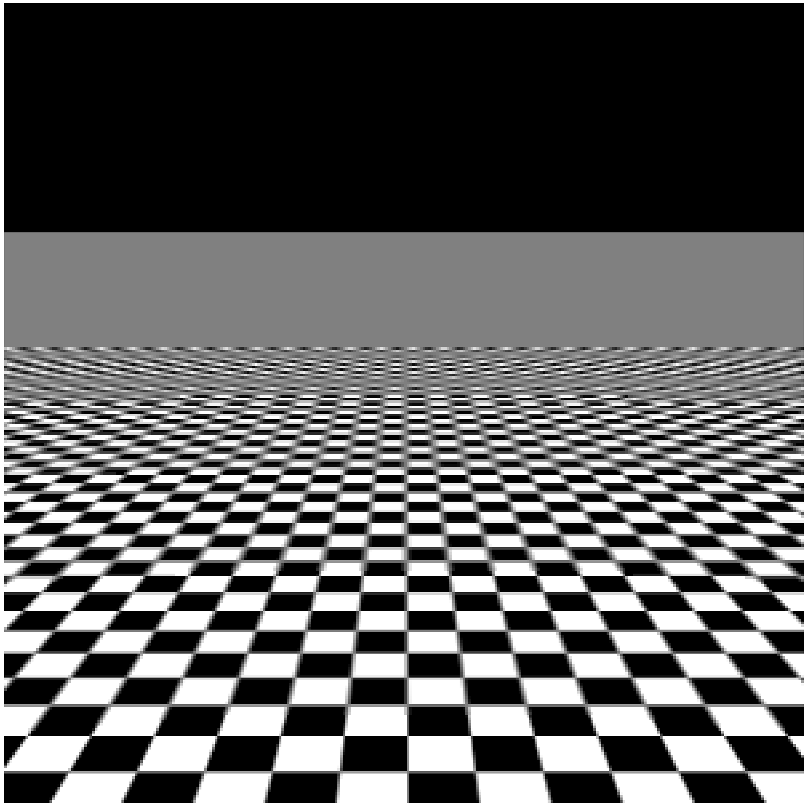
Trick: Precompute images

- Width / 2, Height / 2
- Width / 4, Height / 4
- …
- Sample from best fitting image

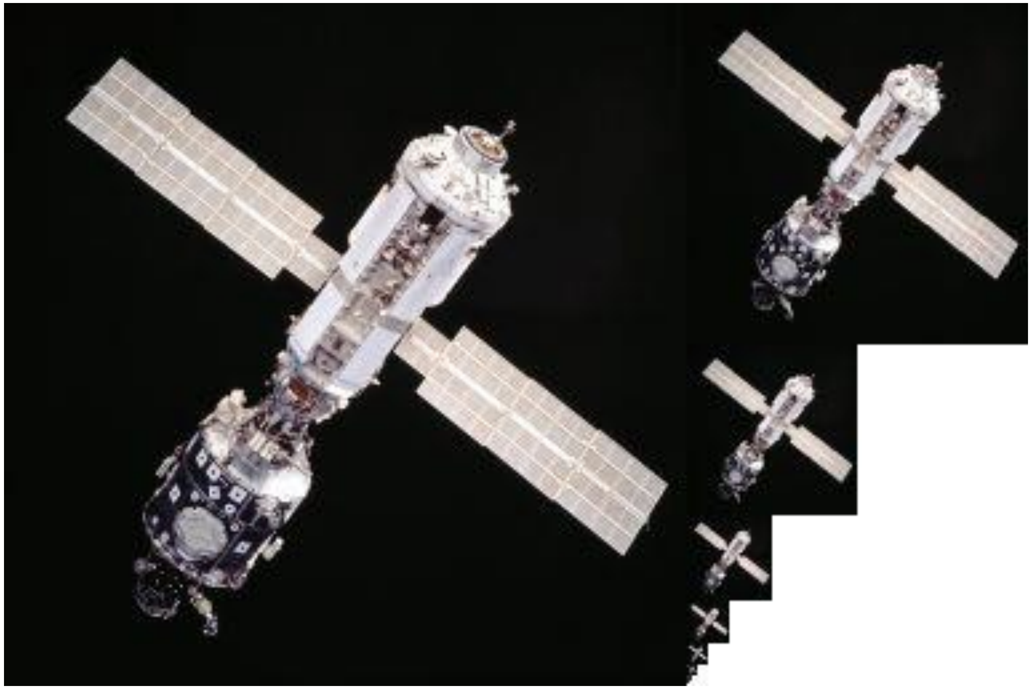
(*multultum in parvo, „much in little)*

No mip mapping





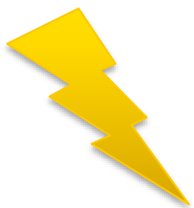
MIP Mapping



Mip Mapping

Seams between mip levels are often visible

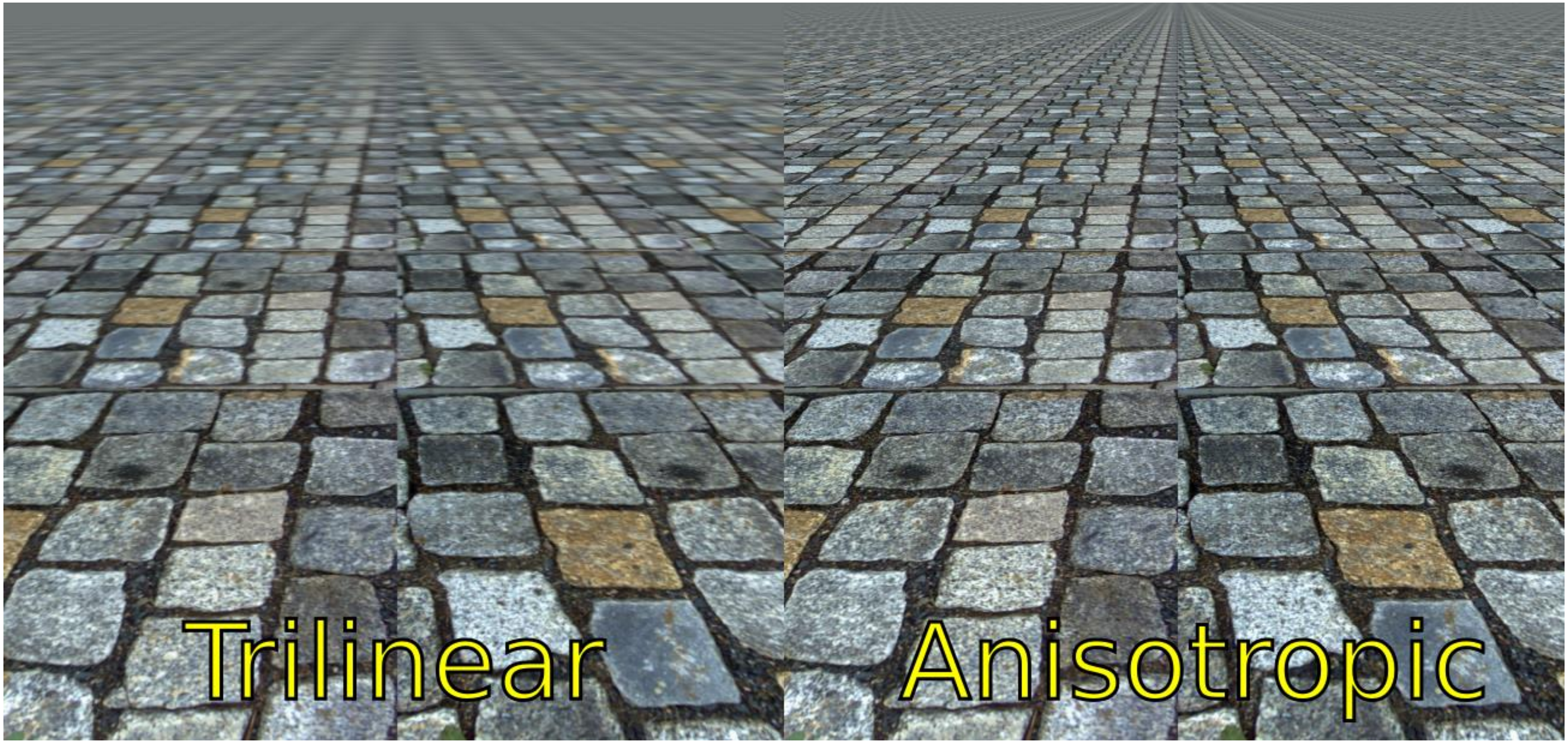
- Trilinear filtering



Perspective stretches images differently in x and y

- No optimal mip level

Anisotropic Filtering



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

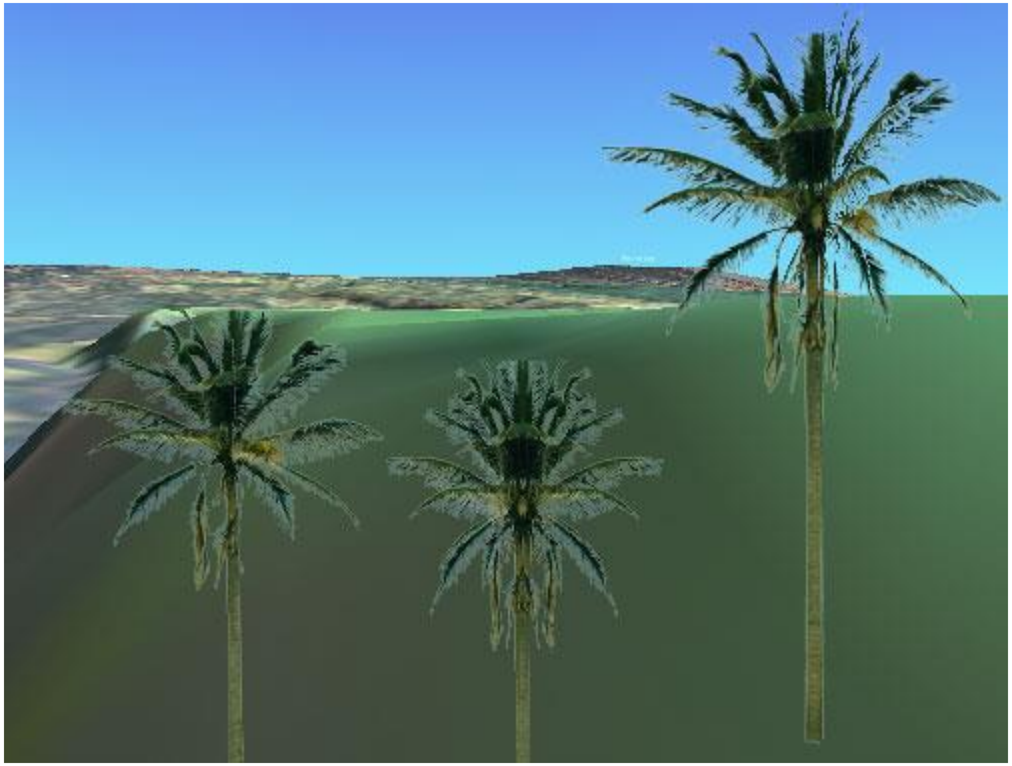
Additive blending

- 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

→ What we did in Exercise 3

Fragment Shader

Also referred to as Pixel Shader

Uses interpolated data from vertex shader

Calculates colors

→ What we did in Exercise 4

Vertex Buffer

Array of vertices

Can hold additional data per vertex

- E.g normal, animation data, ...

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

→ One vertex can be re-used in several triangles

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 →GPGPU

No Rasterization

Additional options for data synchronization

Not yet supported everywhere

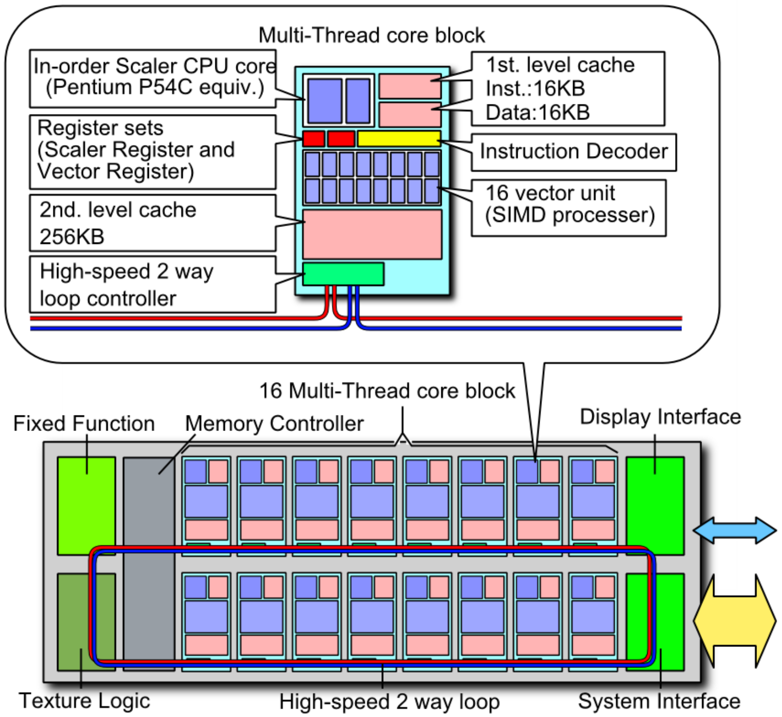
Many competing languages

- Even OpenCL and GLSL compute shaders

Triangles on Compute

Xeon Phi

- Ex project Larrabee



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

More Shaders

TECHNISCHE UNIVERSITÄT DARMSTADT

Geometry Shader

- Works on complete triangles

Tesselation Shader

- Can create new triangles

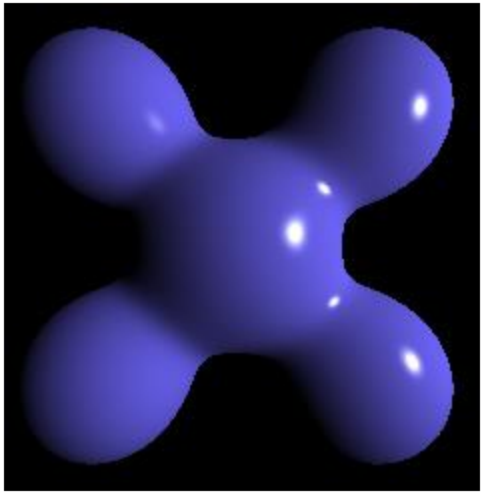
Not yet supported on all hardware

- Notably no support on iOS

Phong Lighting

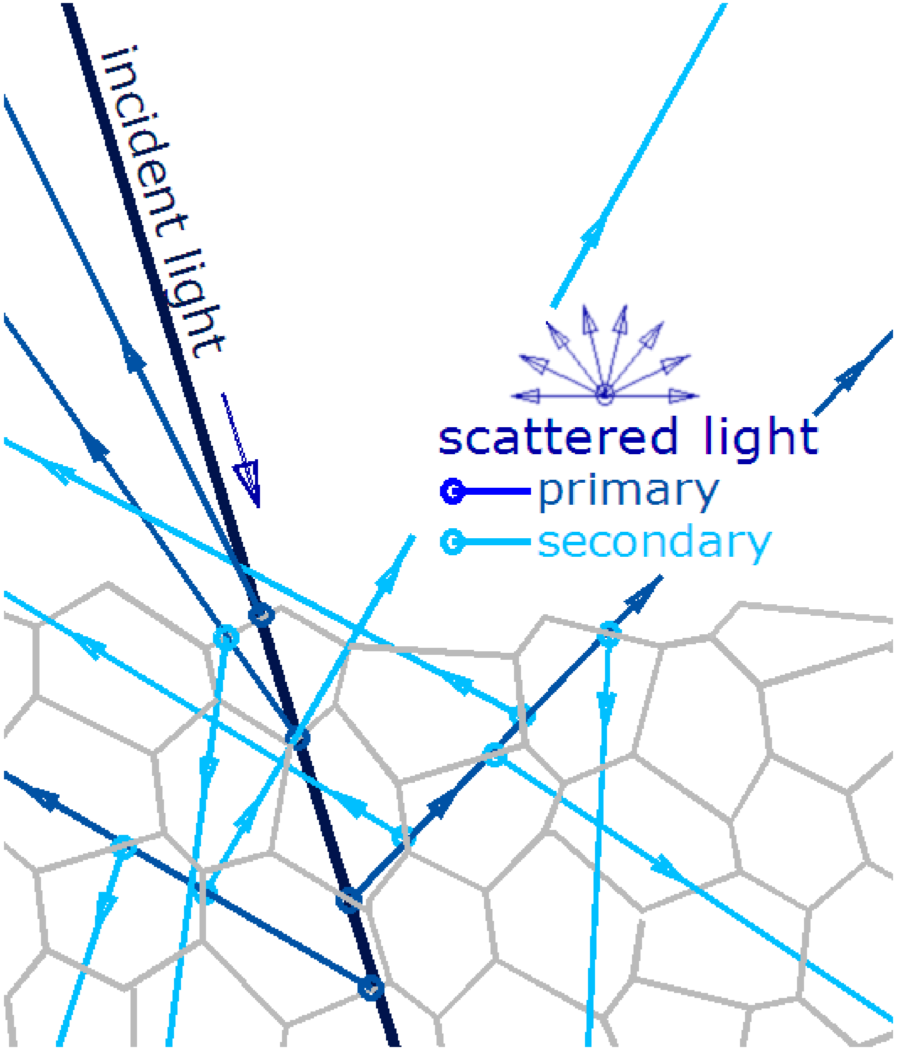
color = ambient + diffuse + specular

- Note: Light from different sources can always be added just like that



Ambient = Constant

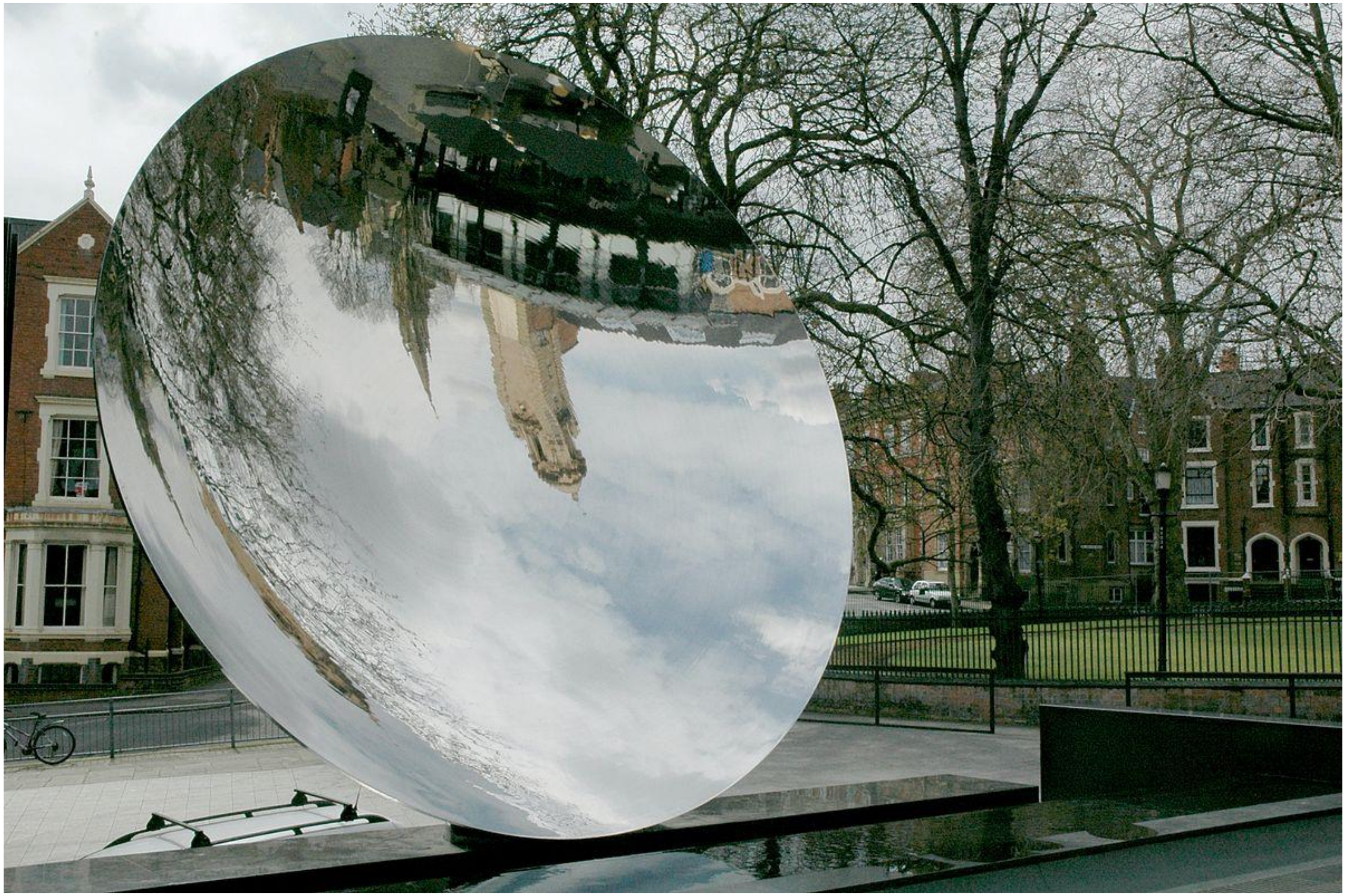
Diffuse



Diffuse

diffuse = LN (see previous lecture)

Specular



Specular

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

$$I_{specular} = I_{in} k_{specular} \left(\vec{R} \cdot \vec{V}\right)^{\mathbf{n}}$$

R: mirrored vector to the light source (reflectance vector)

V: vector to the camera

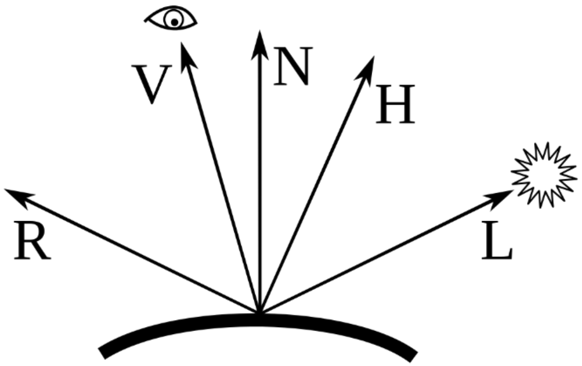
n: roughness – start at 32 and tune

Empirical model (aka basically nonsense)

Ugly for larger angles (cos → 0)

(H: Half-vector between V and L)

(N: Normal)



Blinn Phong

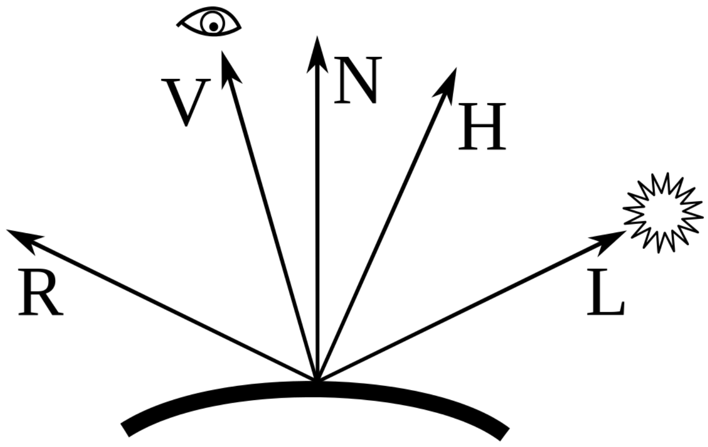
$$H = \frac{V + L}{\|V + L\|}$$

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

$$I_{specular} = I_{in} k_{specular} \cdot \left(\frac{(V + L) \cdot N}{\|(V + L)\| \cdot \|N\|} \right)$$

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

Quake (1996)

Better specular lighting

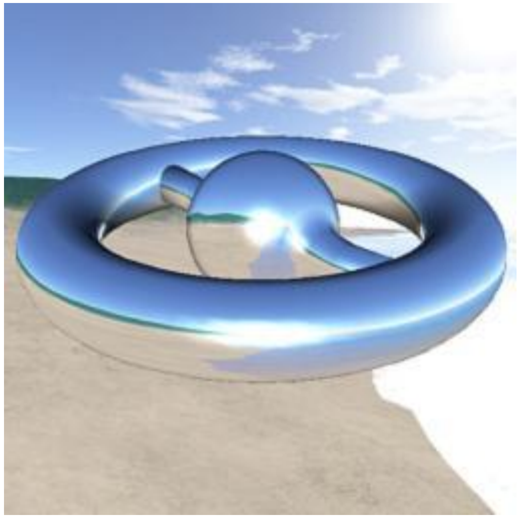
Render six orthogonal perspectives into a cube map

- 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

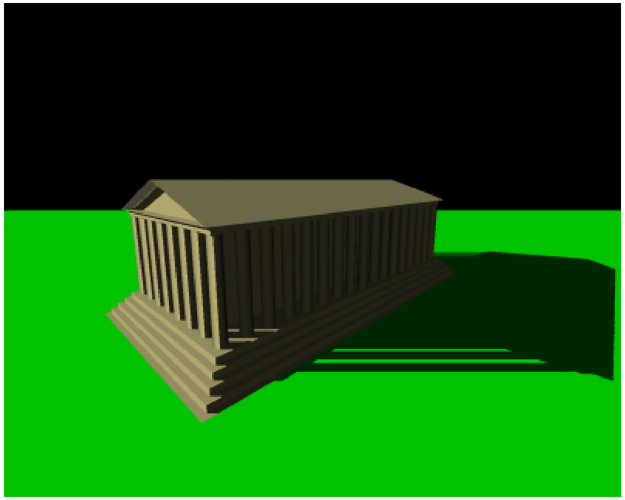
- Using camera position
- Using light position → z = distance from light

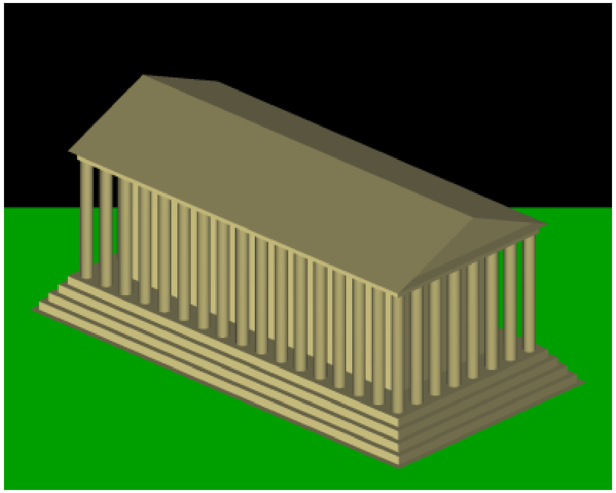
Read depth texture

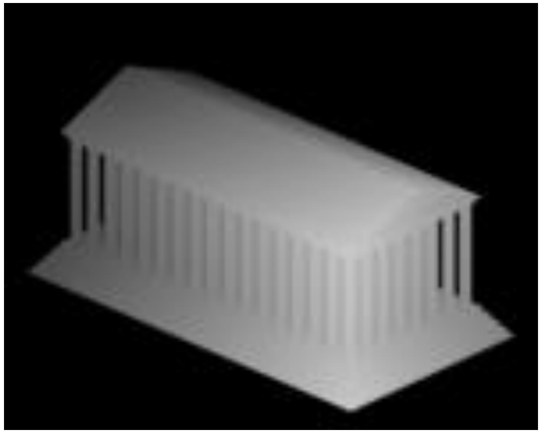
Compare depth calculated using light pos and depth from texture

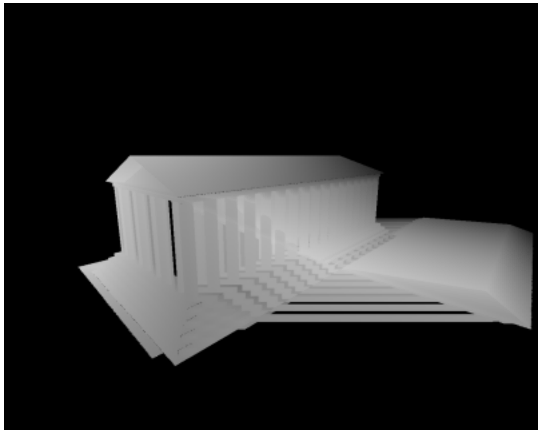
- If greater → in shadow

Shadow Mapping

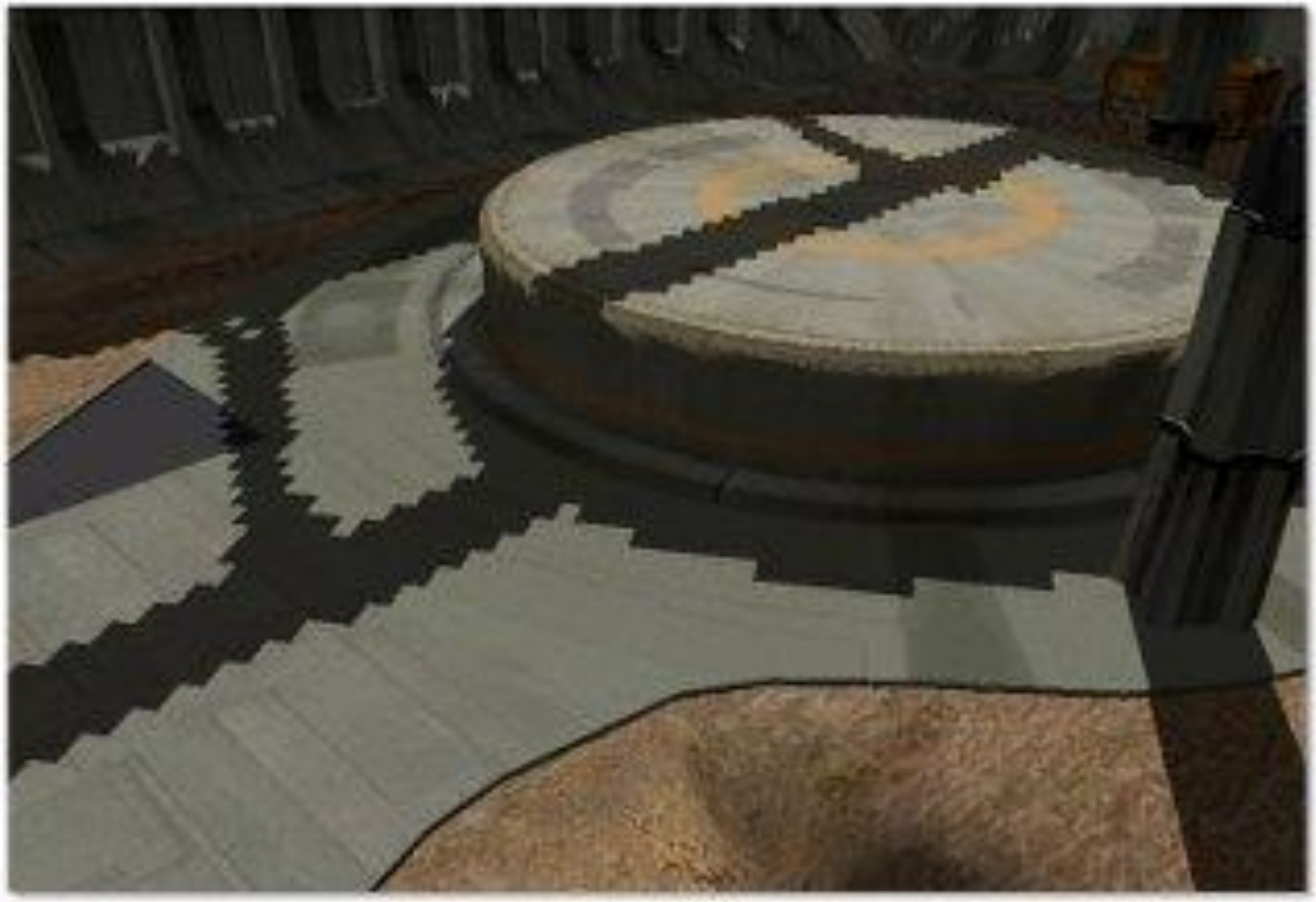




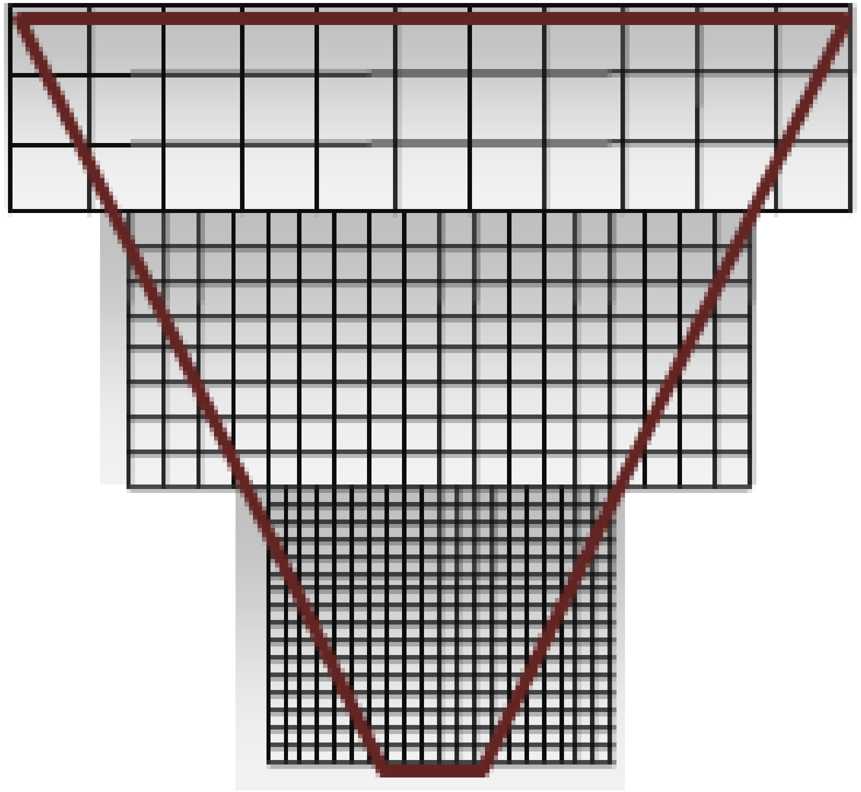




Shadow Mapping Problems



Cascaded Shadow Maps



Summary

TECHNISCHE UNIVERSITÄT DARMSTADT

What work can the GPU assist us with?

- Highly parallel calculations:
  - Graphics (each pixel, each vertex, ...)
  - General purpose tasks that can be parallelized
- Graphics-related tasks
  - Rasterization
  - Texture lookups/filtering

Techniques

- Antialiasing
- Mip-mapping
- ...

Now: How to program this?

GLSL

OpenGL Shading Language

Added to OpenGL in 2004 with OpenGL 2.0

Version 1.10

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;
}
```

Kore/Kha specialties

Kore and especially Kha are intended for cross-platform usage

Challenge 1: GSLS versions, capabilities

- Widest coverage: OpenGL ES Shading Language
- WebGL: Based on OpenGL ES
- Supported across mobile devices
- Supported on desktop devices

Challenge 2: Different shader languages

- E.g. on Windows: DirectX, HLSL
- Apple devices: Metal
- Cross-compiler krafix

Vertex Shader

Transforms vertices

Writes transformed vertex to special var

- gl_Position

Can write additional data

Fragment Shader

TECHNISCHE UNIVERSITÄT DARMSTADT

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

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

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 -g 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

TECHNISCHE UNIVERSITÄT DARMSTADT

#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

Conclusion

TECHNISCHE UNIVERSITÄT DARMSTADT

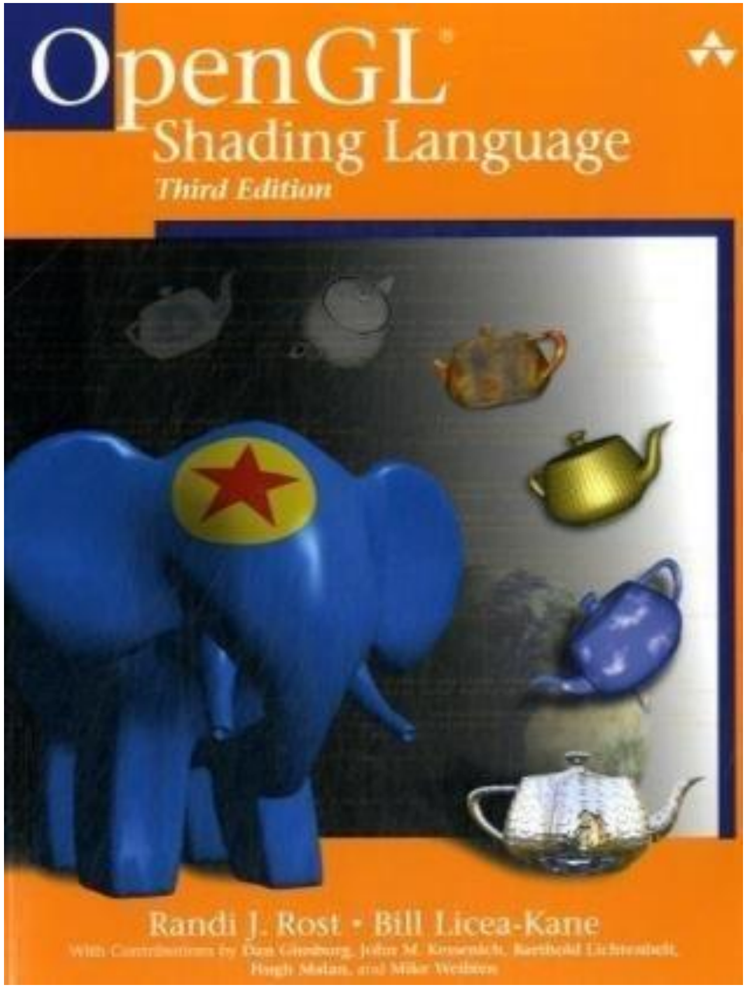
OpenGL Shading Language

Types of shaders

Input and Output

Operations

More info: „Orange Book“ (OpenGL Shading Language)



See it in action

Very nicely done "GTA V – Graphics Study"

http://www.adriancourreges.com/blog/2015/11/02/gta-v-graphics-study/