



Computer Graphics

Computer Graphics Topics 1

- **Introduction**
 - What is computer graphics, “hello world” in OpenGL, ...
- **Review of Fundamentals (Linear Algebra)**
- **Drawing Geometric Objects**
- **Vertices, lines, polygons, normal vectors, ...**
- **Coordinate systems and transformations**
- **World space vs. object space, projection, clipping, ...**
- **Color models**
- **Shading**
- **Z-buffer, shading models, light sources, ...**

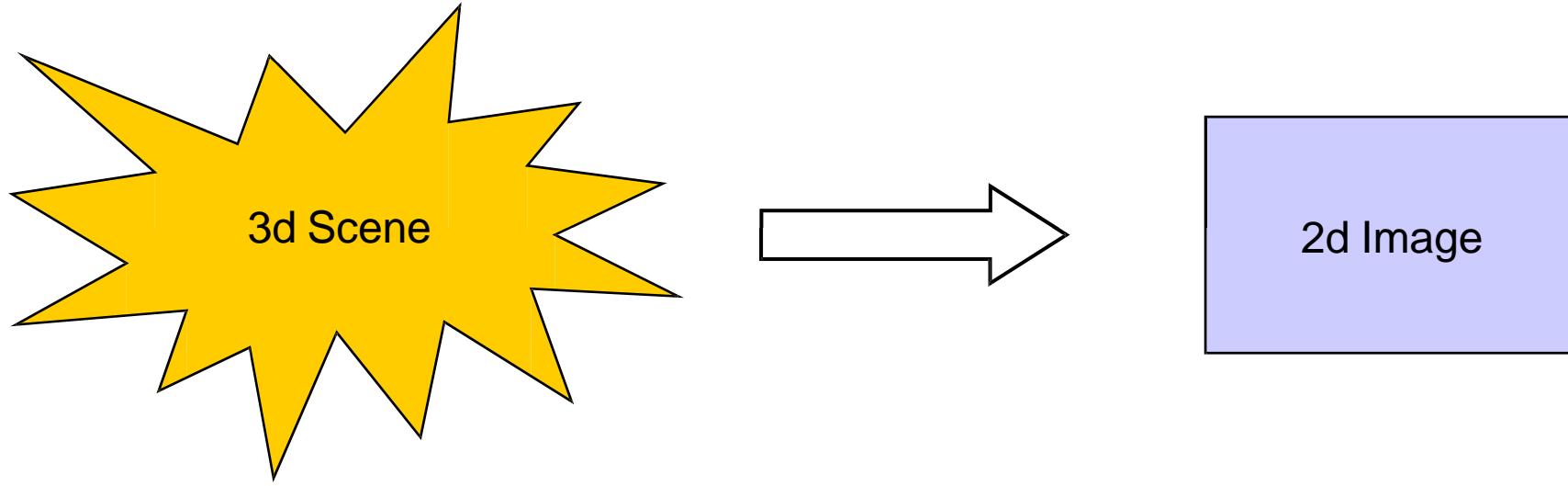
Computer Graphics Topics 2

- **Textures**
 - Images in OpenGL
- **Texture mapping, ...**
- **Animations, ...**
- **Interaction, ...**

What is Computer Graphics?

- Visual Representation on a computer
- includes many “obvious” things (2D graphics, GUI, etc.)
- 3d modelling and rendering
- other relevant fields: display devices, physics, mathematics, ...
- 2D graphics (many well-known standard operations):
 - text
 - graphical user interface (GUI)
 - image editing (Photoshop)
 - drawing and presentation (PowerPoint)

Rendering Pipeline 1



Rendering Pipeline 2

- a pipeline consists of several stages
- similar to an assembly line
- stages perform in parallel
- but each stage has to wait (stall) for the slowest stage to finish
- therefore, the speed of the pipeline is determined by the slowest stage

... in the Context of Rendering

- CPU limited
- GPU limited
 - bandwidth limited
 - fill rate limited
- no simple answers exist
 - development of HW is way too dynamic
 - broad spectrum of different applications

Real-Time vs. Non-Real-Time

- **Real-Time Rendering Pipeline**

- quite standardized
- defined by APIs like OpenGL or DirectX
- several stages commonly performed in HW

- **Non-Real-Time Rendering**

- not standardized
- no prevalent APIs
- mostly software implementations

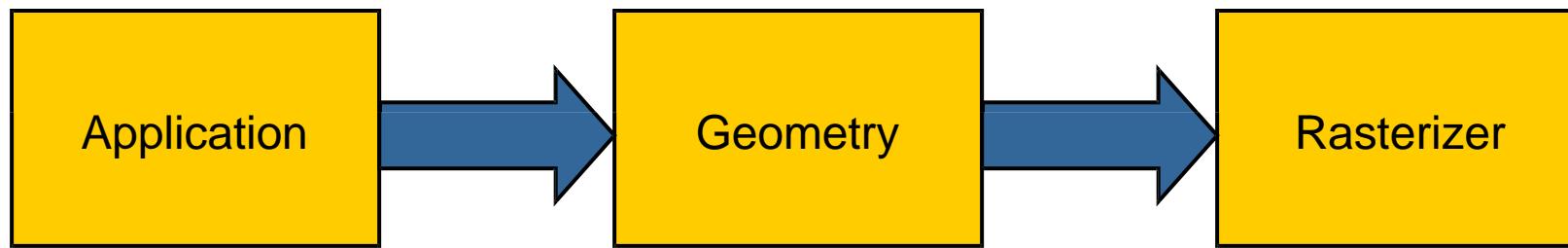
Real-Time Rendering Pipeline

- We take a detailed look at the Real-Time Rendering Pipeline, since it is quite standardized.
- **Rendering Speed**
 - update speed of images
 - determined by slowest stage (**bottleneck**)
 - expressed in frames per second (**fps**)
 - or Hz (1/second, frequency)

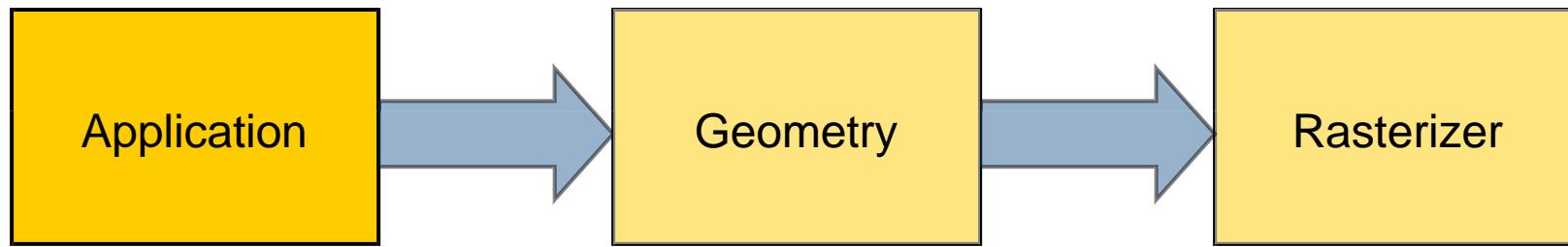
Pipeline Stages

Conceptual Stages

- basic construction of the rendering pipeline
- consists of 3 conceptual stages
- each stage may be a pipeline in itself



Application Stage 1



Application Stage 2

- developer has full control
- implemented in software
- common tasks are
 - rendering loop
 - data management & scene graphs (models, textures, ...)
 - user interaction
 - acceleration algorithms
 - networking
 - game logic & AI
 - animation & collision detection & physics
 - ...

Rendering Loop

- interacts with operating system / user
 - windowing system messages
 - input devices
 - window messages
 - ...
- acts as scheduler
 - allots time slots for rendering and all other tasks
 - rule-of-thumb: render first, then CPU-based stuff
 - because GPU can render asynchronously, while CPU already performs other tasks!

Data Management

- **responsible for loading**
 - **models, textures, other resources ...**
- **from slower media to faster media**
 - **often not all data will fit into (V)RAM**
 - **caching hierarchy**
 - Network -> optical drives? -> hard disk (SSD) -> RAM -> VRAM
- **multi-threading / asynchronous jobs**
 - **synchronization issues**
 - **very important for multi-core systems**

User Interaction

- **input devices**
 - mouse, keyboard, joystick
 - other devices
 - network as input device (multi-user apps)
- **important to create a well-defined interface**
 - handle input at a well-defined instant of time and not every time an event occurs – why?

Acceleration Algorithms

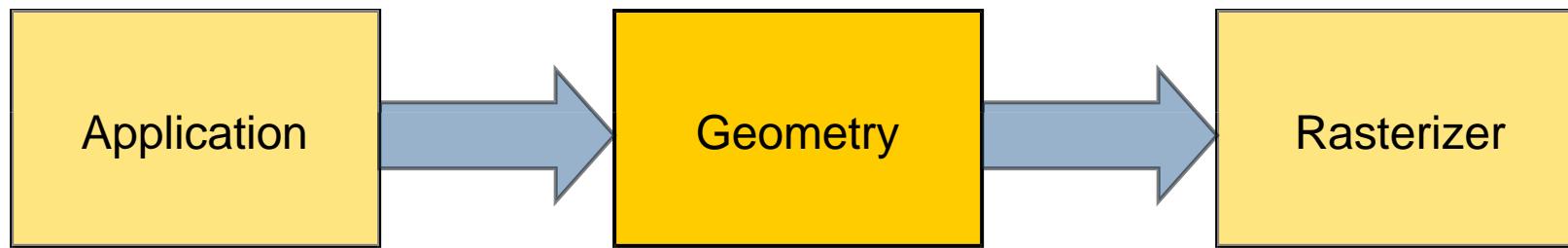
- **Culling mechanisms:**
 - often its cheaper to check if something needs to be done, than to actually do it
 - but: this is a moving target, due to never-ending advances in HW & SW design
- **Examples:**
 - visibility culling
 - levels of detail hierarchies
 - lazy evaluation approaches
 - ...

Networking

- this topic would justify its own course(s)
- multi-user applications range from
 - few users – e.g. collaborative design applications
 - dozens of users – e.g. games (sports, shooters, ...)
 - thousand of users – e.g. MMPrPG (Massive Multi Player Role Playing Games)
- it must be ensured that everyone sees the same state
- very hard to achieve when communicating over unreliable networks (e.g. internet)

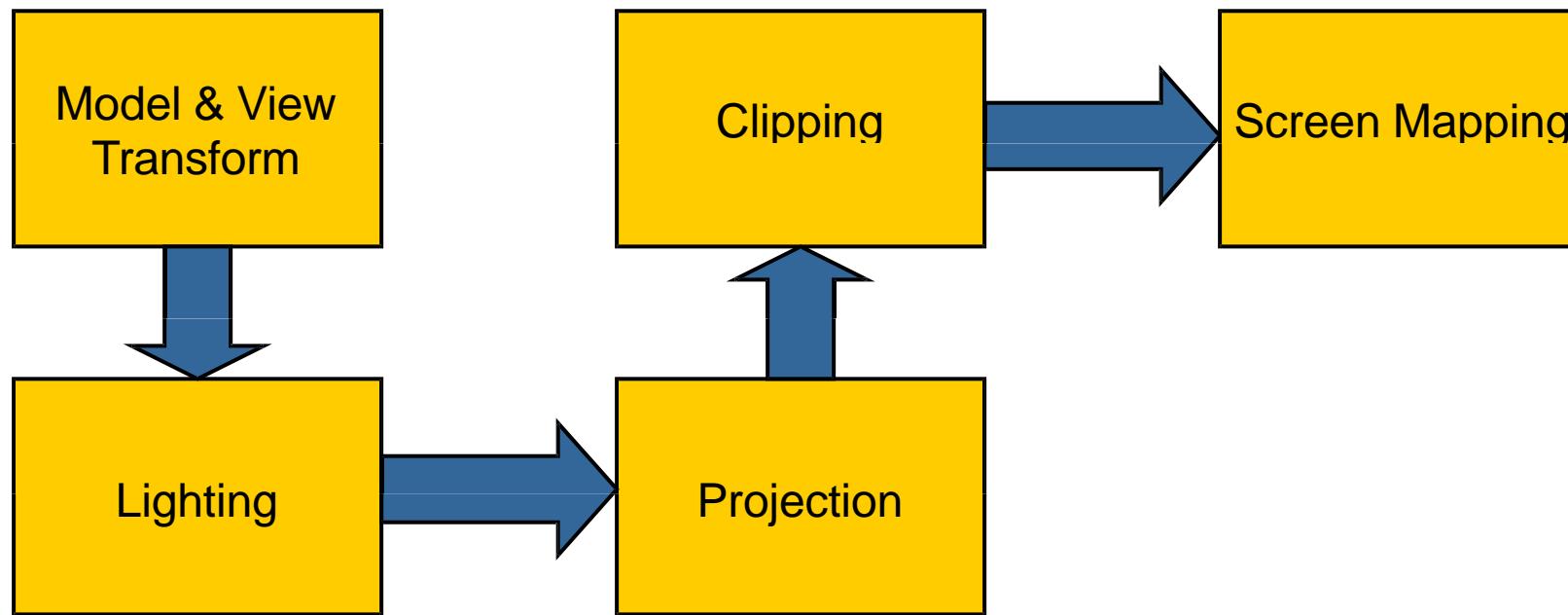
- rendering is just one small part of a computer graphics application
- in many applications the state of a simulated environment has to be updated each frame
 - e.g. behaviour and actions of NPCs
 - animation of models
 - physics (GPU)
 - interactions between players and environment
 - collision detection
 - user actions may change the environment

Geometry Stage 1



Geometry Stage 2

- today mostly implemented in hardware (PC, Consoles)
 - and on mobile devices
- consists of several functional stages



Model and View Transform

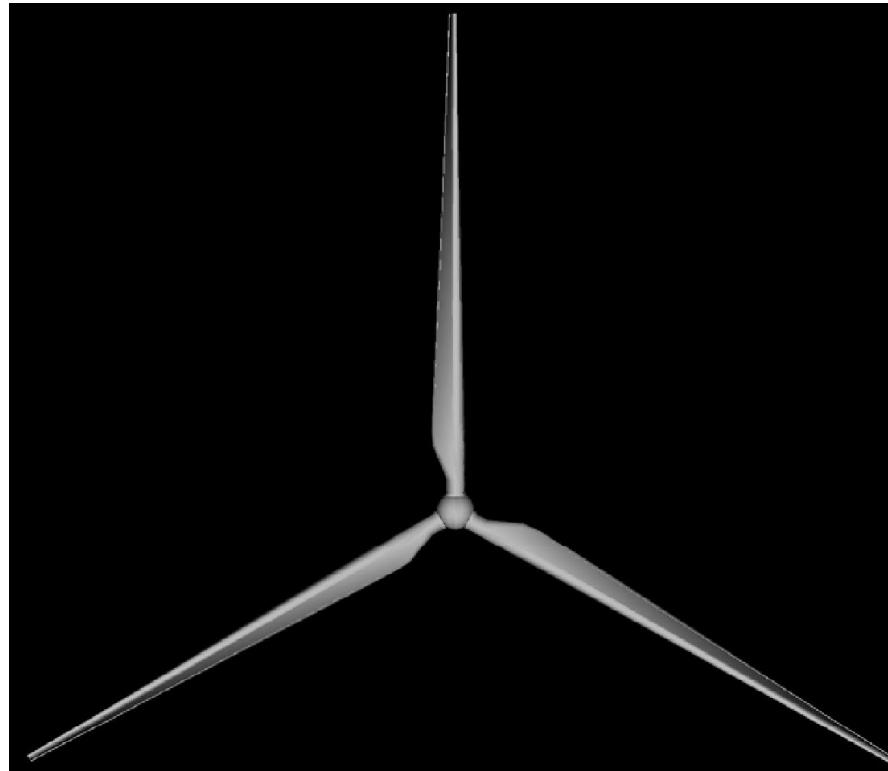
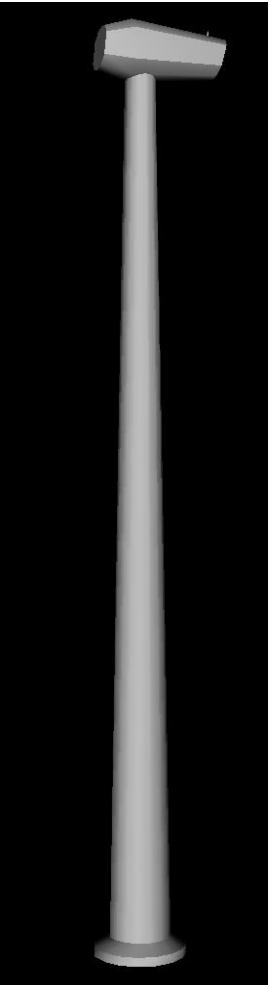
- on its way to the screen, each model (e.g. a chair) is transformed into several different spaces



Model Space

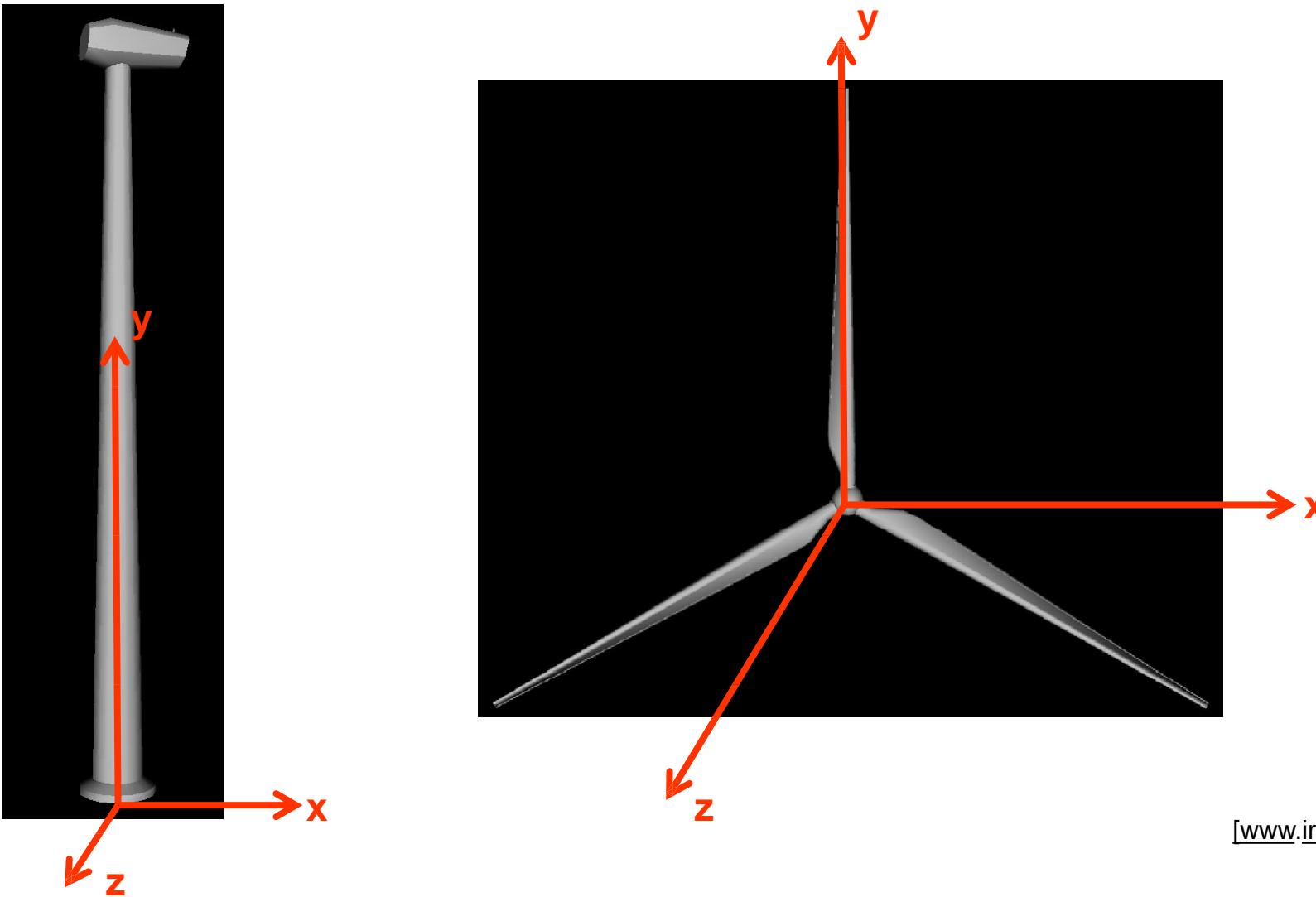
- originally, a model resides in its own model space
- this is the space in which it has been modelled
- that means, it has not been transformed at all

Model Space Example 1



[www.irtc.org]

Model Space Example 2



[www.irtc.org]

World Space

- each model is associated with a model transform, so that it can be positioned and oriented
- possible to have multiple model transforms to place several copies (instances) in the same scene
- after applying the model transform, the model is in world space

World Space Example



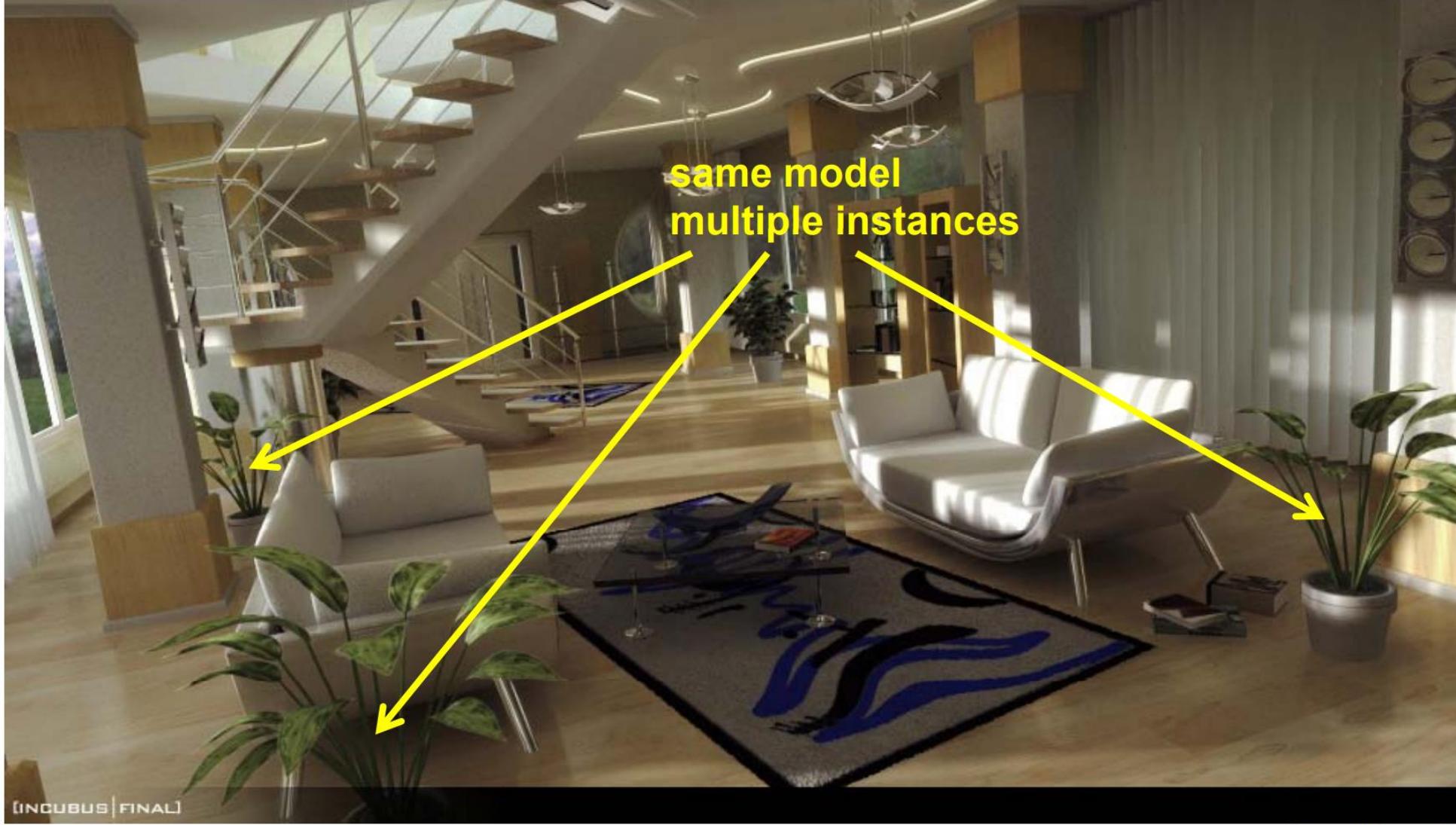
**Same model
multiple
instances!**

[VRVis]

Do-It-Yourself



Do-It-Yourself



Do-It-Yourself

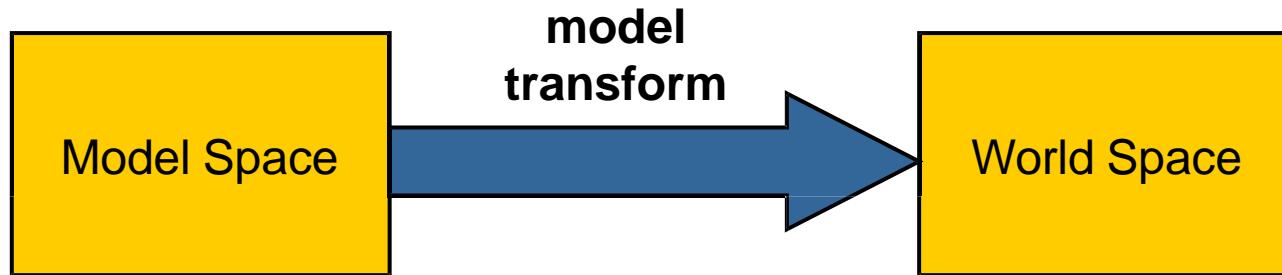


Do-It-Yourself



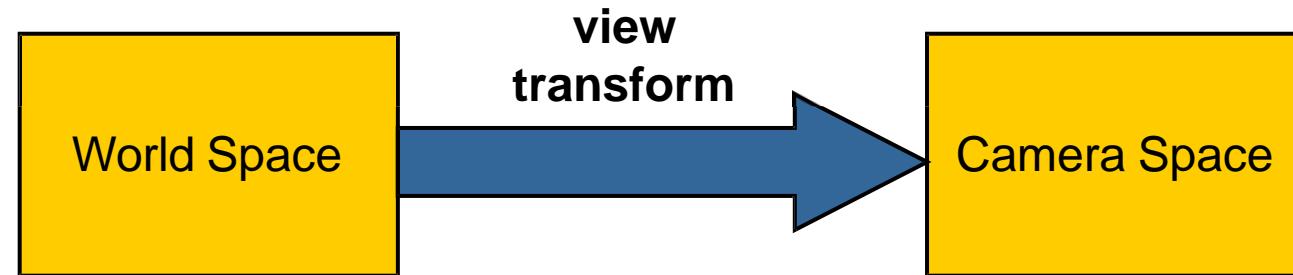
World Space

- the world space is unique (there is only one)
 - not always true in apps dealing with extremely large extents of space
- all models now exist in this same space
- the camera is also defined in world space



Camera Space

- next, the camera and all models are transformed by the view transform
- its purpose is to place the camera at the origin, and aim it in a predefined direction (e.g. to look along the negative z-axis, in OpenGL)



Lighting and Shading

- shading
 - vertices are associated with colors and textures
 - values are interpolated between vertices
 - Gouraud shading
- lighting
 - effect of light sources on models is calculated
 - lighting equations, simple (but fast) approximations of real-world interaction of light with surfaces

Projection

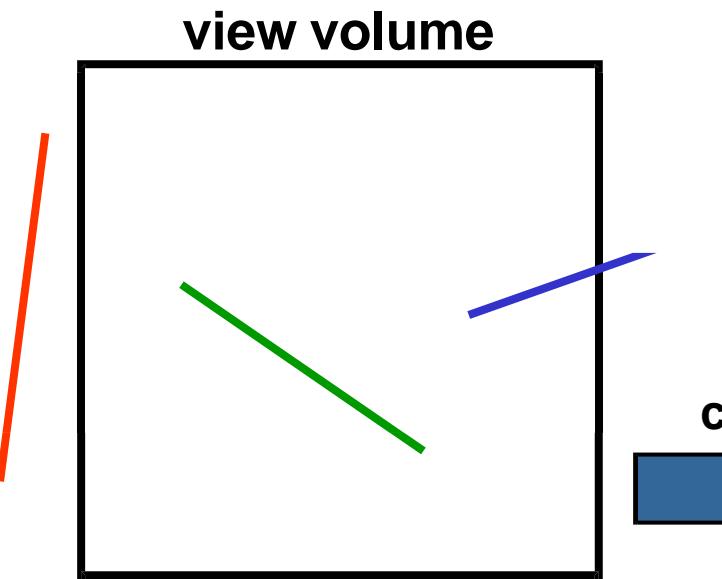
- next, the **viewing volume** (the part of the scene which is in the cameras field-of-view) is transformed into a unit cube (in OpenGL)
 - extreme points at $(-1,-1,-1)$ and $(1,1,1)$
 - also called canonical view volume
 - coords are called Normalized Device Coordinates (NDC)
- there are essentially two methods
 - orthographic and perspective projection
 - (see slides “Lecture 2 - Projections”)

Clipping 1

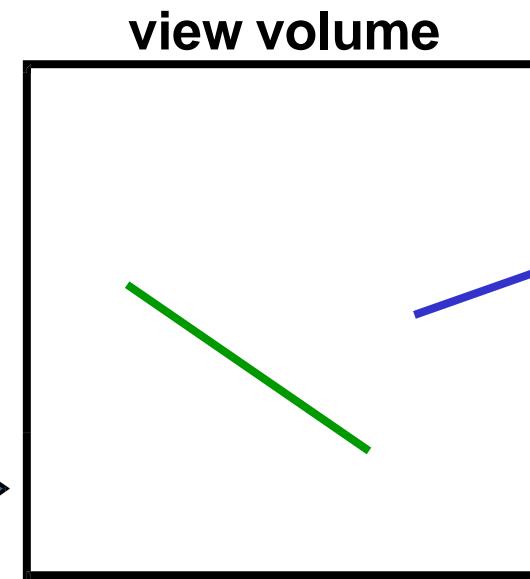
- only primitives wholly or partly inside the view volume need to be passed on to the rasterizer stage
- primitives totally outside are dropped
- primitives fully inside are passed on
- partly visible primitives require clipping

Clipping 2

BEFORE
clipping



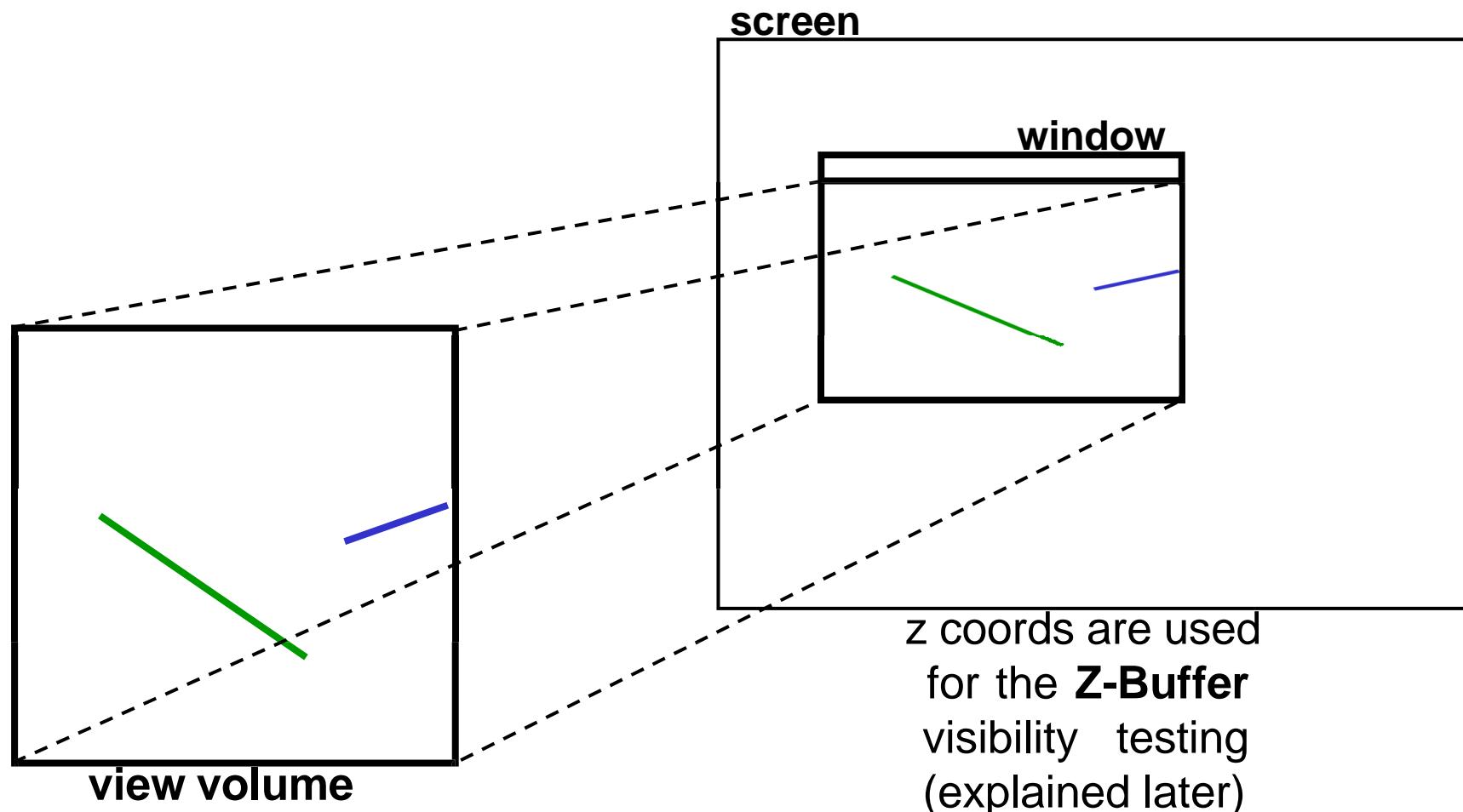
AFTER
clipping



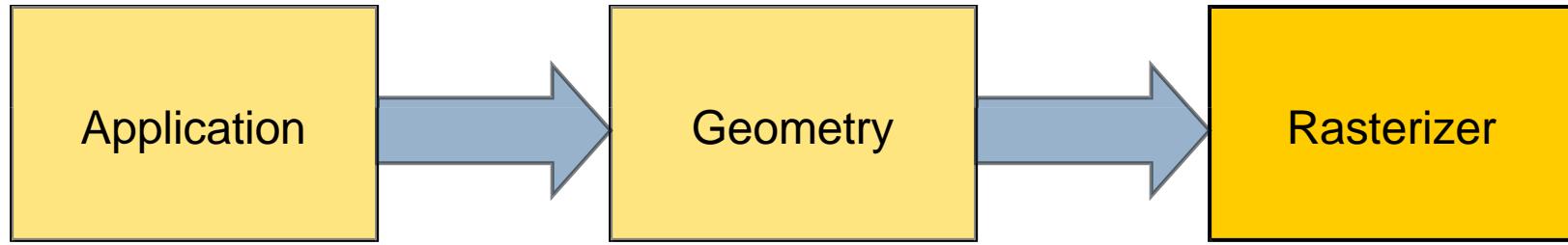
clipping

Screen Mapping

- x,y coords (NDC) are transformed to screen space (pixel coords)



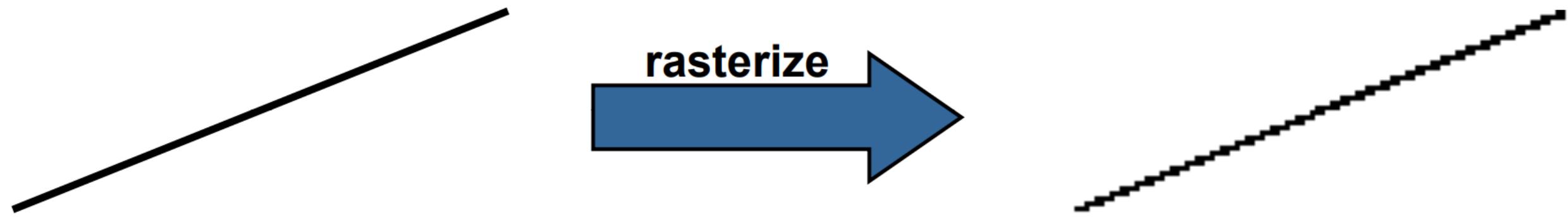
Rasterizer Stage 1



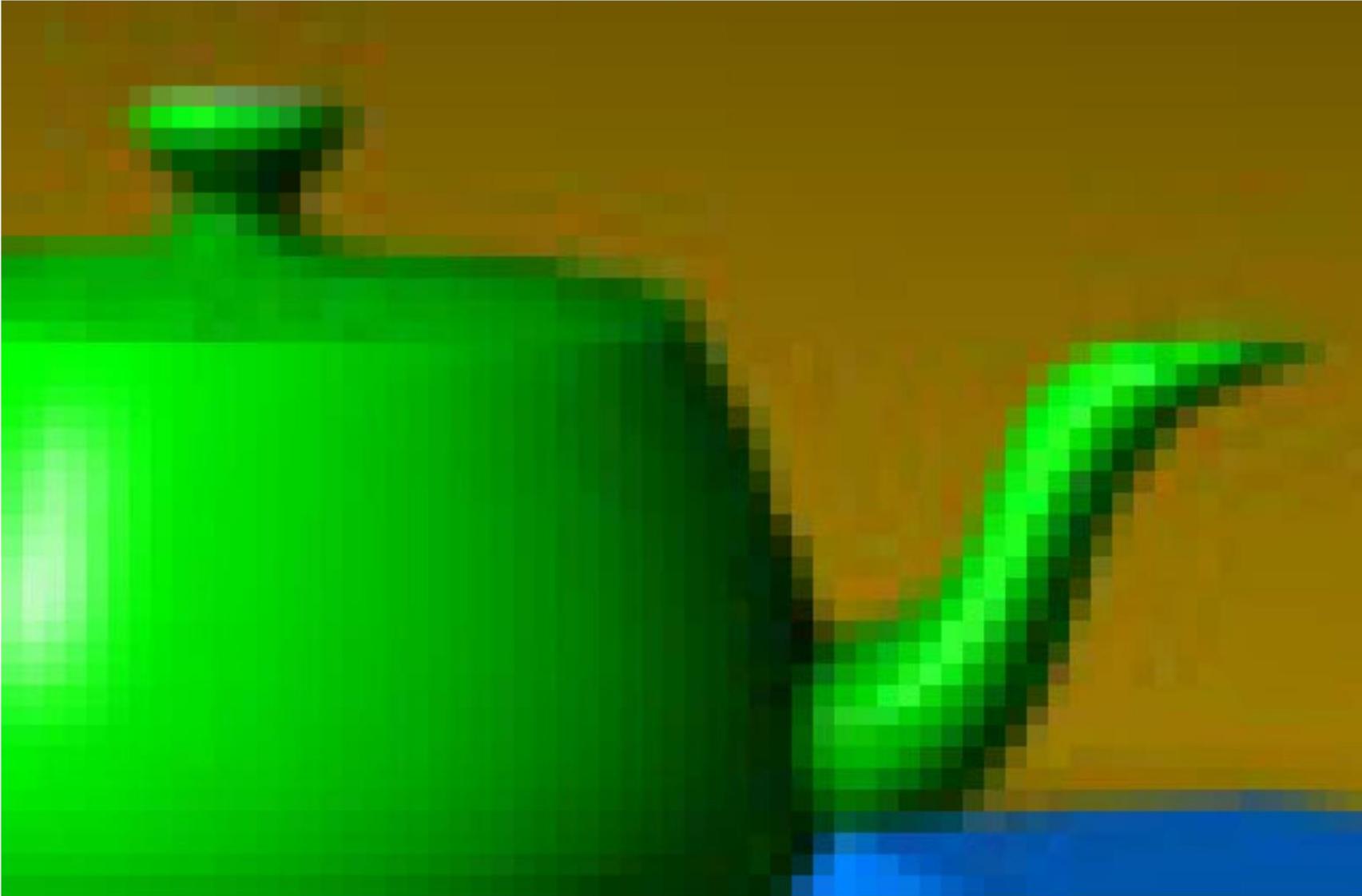
Rasterizer Stage 2

- **input**
 - transformed and projected vertices, colors, and texture coordinates
 - vertex x,y-coords in screen space
 - vertex z-coord (depth) in range [-1,1]
- **task**
 - rasterize primitives
 - assign correct colors to the pixels

Rasterize 1



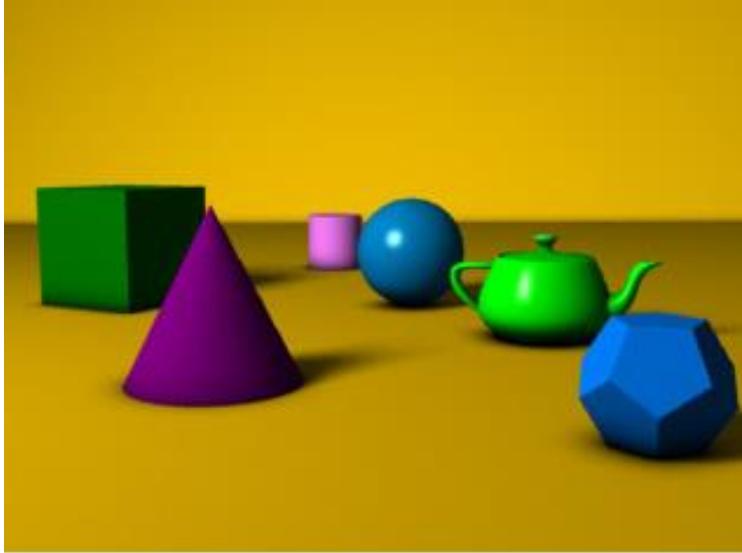
Rasterize 2



Z-Buffer 1

- used to resolve visibility
- Z-Buffer is same size as color buffer
 - stores z-value for each pixel
- when a primitive is rendered to a certain pixel
 - z-value of that pixel on primitive is calculated
 - if new z is smaller than z in Z-Buffer
 - then new pixel is closer than previous pixel at this position -> update color buffer and z-buffer
 - if new z is greater
 - then nothing is done, since new pixel is not visible

Z-Buffer 2



[<http://en.wikipedia.org/wiki/Z-buffer>]

A simple 3d scene



Z-buffer representation

The background image shows an aerial view of the FH Technikum Wien campus at night. The buildings are illuminated from within, and the surrounding city skyline is visible under a dark sky.

FH

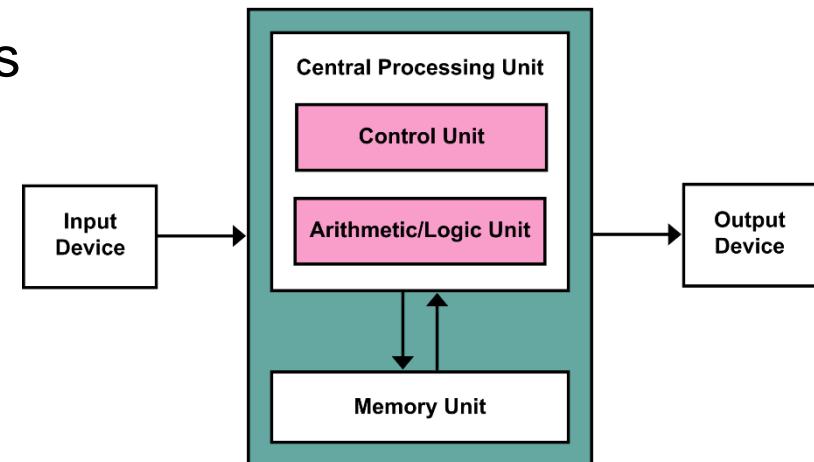
University of
Applied Sciences

TECHNIKUM
WIEN

Hardware

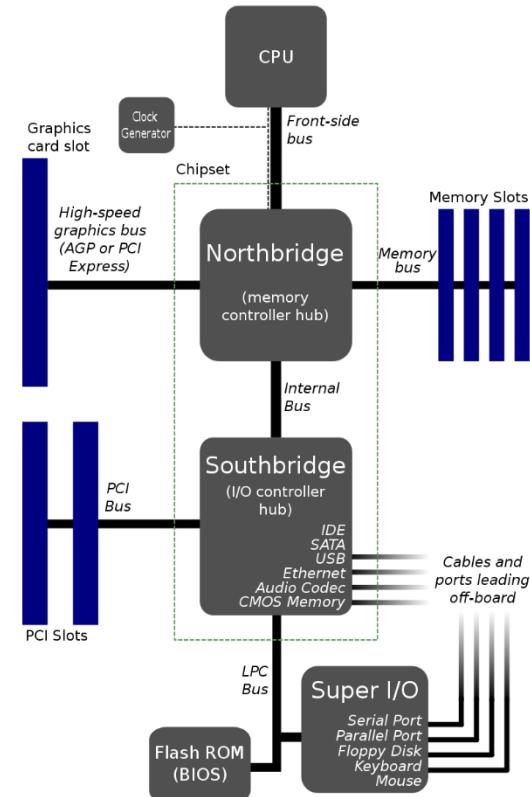
Central processing unit

- Main processor
- Primarily von-Neumann design
- Usually multiple cores (AMD Ryzen 7 5800X3D → 8)
- Control unit: directs the operation, directs the flow of the CPU and other devices
- Arithmetic logic unit: perform integer arithmetic and bitwise logic operations
- Address generation unit: calculate memory addresses required for fetching data from the memory
- Cache: smaller, faster memory, closer to a processor core



Motherboard

- CPU sockets
- Memory slots
- Chipset (interface CPU/memory/bus)
- Non-volatile memory (BIOS)
- Clock generator (synchronization)
- Slots for expansion cards
- Power connectors
- Connectors for hard drives, SSDs, ...
→ NVMe & SATA



Graphics card

- Compute Units
- Clock
- Memory
- Power/Heat
- Software
 - Drivers
 - AI (DLSS, Radeon Super Resolution, ...)

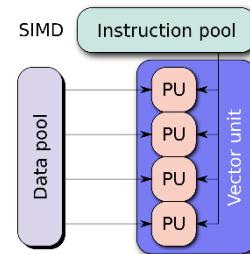


CPU

- Generalist component
- Main processing of a computer
- About 1-64 cores
- Tasks: (usually) serial
- Great at processing one big task at a time

GPU

- Specialized component
- Graphics and video rendering
- Thousands of cores
- Tasks: parallel
- Great at processing multiple smaller tasks at a time





University of
Applied Sciences

TECHNIKUM WIEN

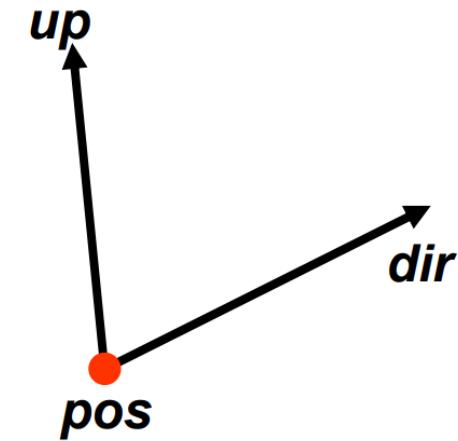
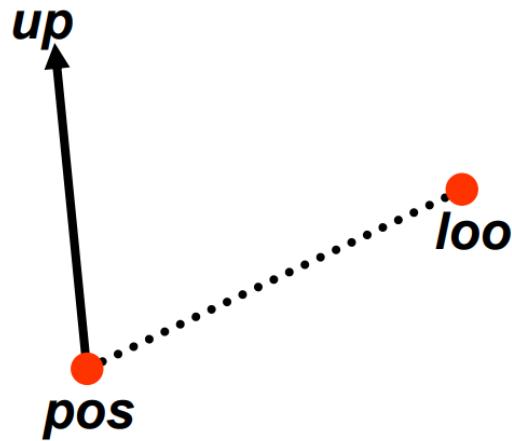
Viewing & Projections

Overview

- Camera Definition
- View Transform
- Orthogonal Projection
- Perspective Projection
- Field of View
- Special Projections

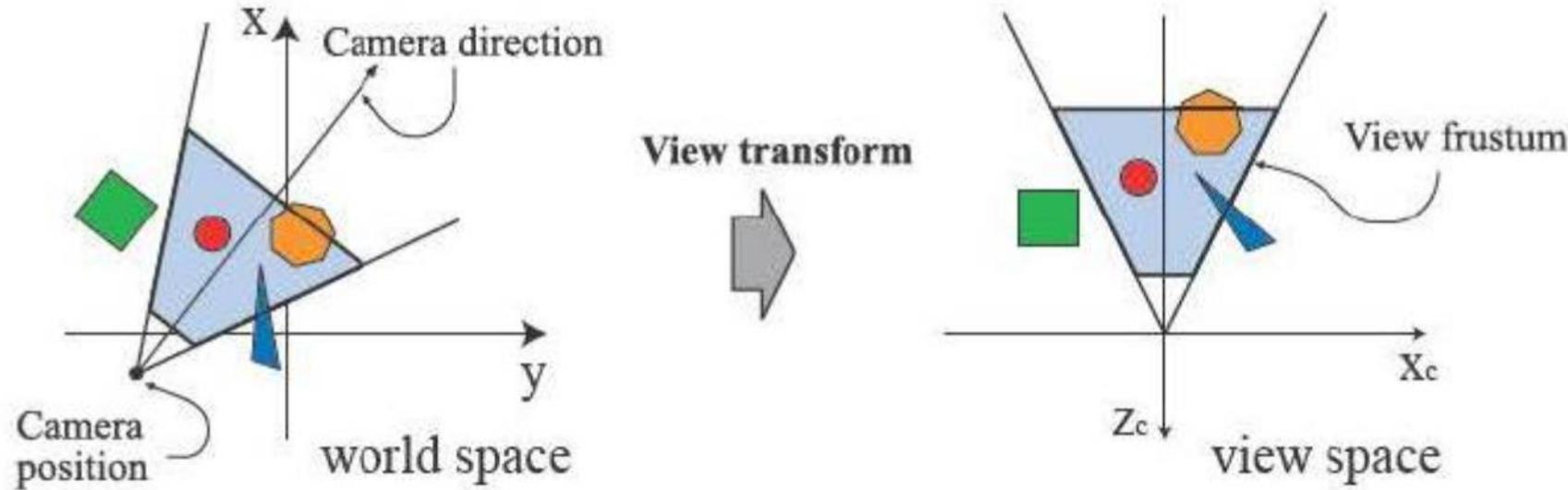
Camera Definition 2

- position
- “look at” point (or viewing direction)
- up vector



View Transform 1

- purpose
 - place camera at origin
 - look along negative z-axis
 - y is up

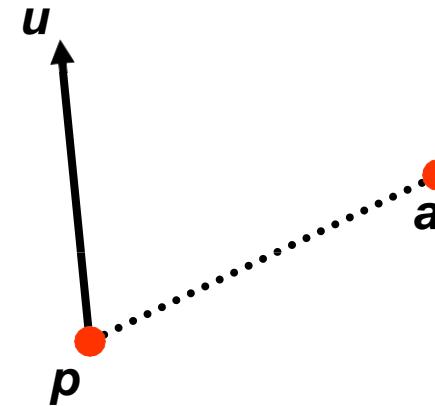


[Real-Time Rendering, Figure 2.4]

View Transform 2

- How?

- p ... camera position
- a ... look at point
- u ... up vector



Step 1: shift camera to origin

$$t = -p$$

View Transform 3

- Step 2: align

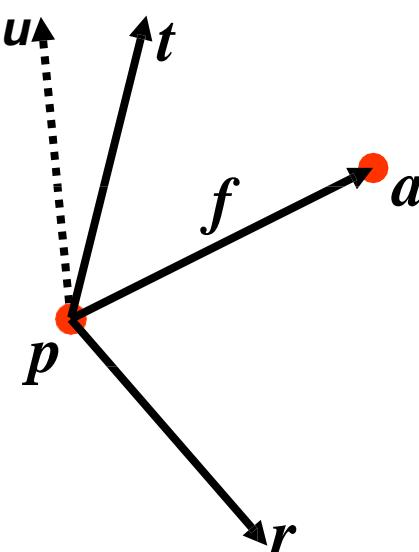
we need a orthogonal basis!
f ... forward (viewing direction)

r ... right dir (perpendicular to f and u)
t ... top (perp. to r and f)

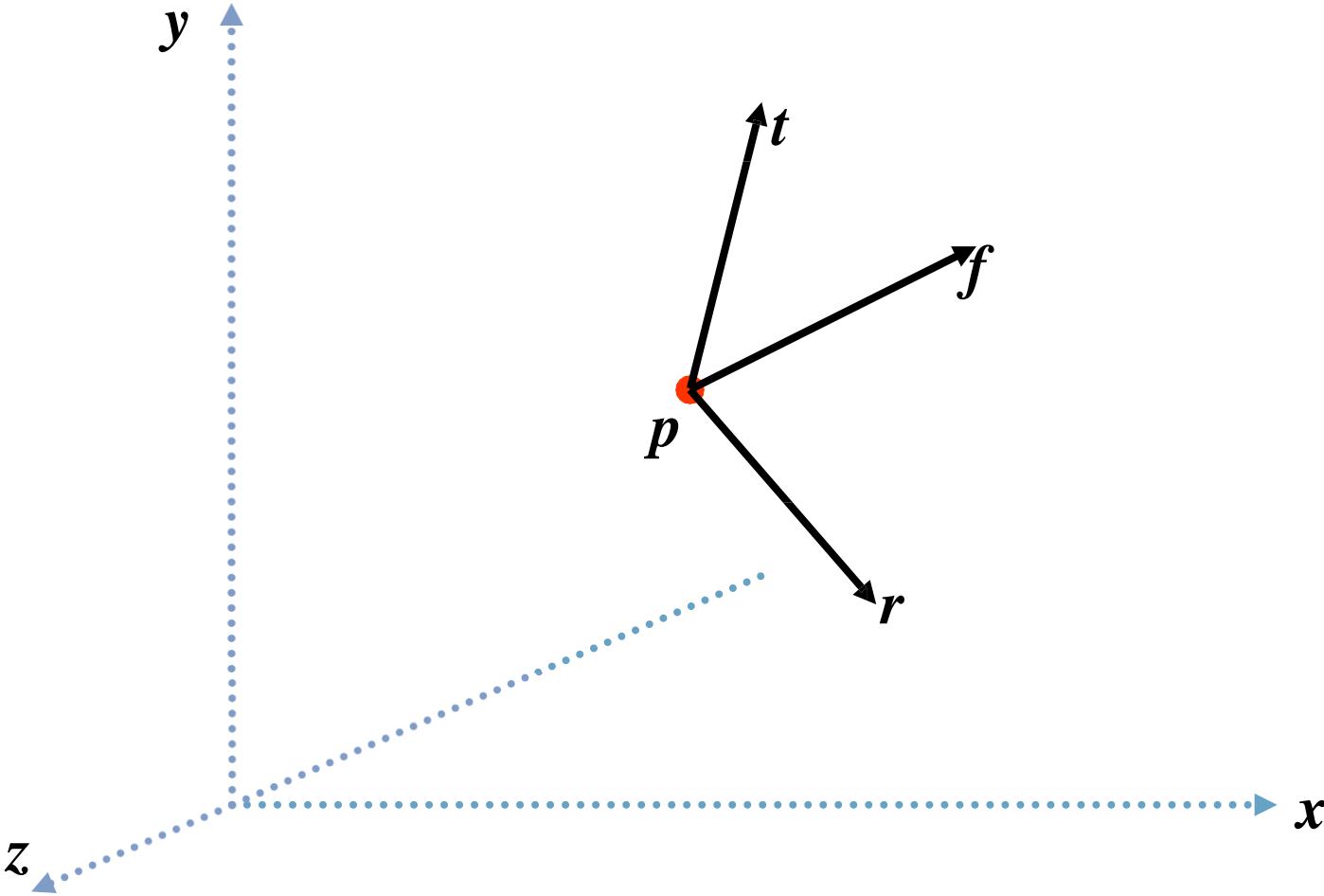
$$\mathbf{f} = \|\mathbf{a} - \mathbf{p}\|$$

$$\mathbf{r} = \|\mathbf{f} \times \mathbf{u}\|$$

$$\mathbf{t} = \|\mathbf{r} \times \mathbf{f}\|$$

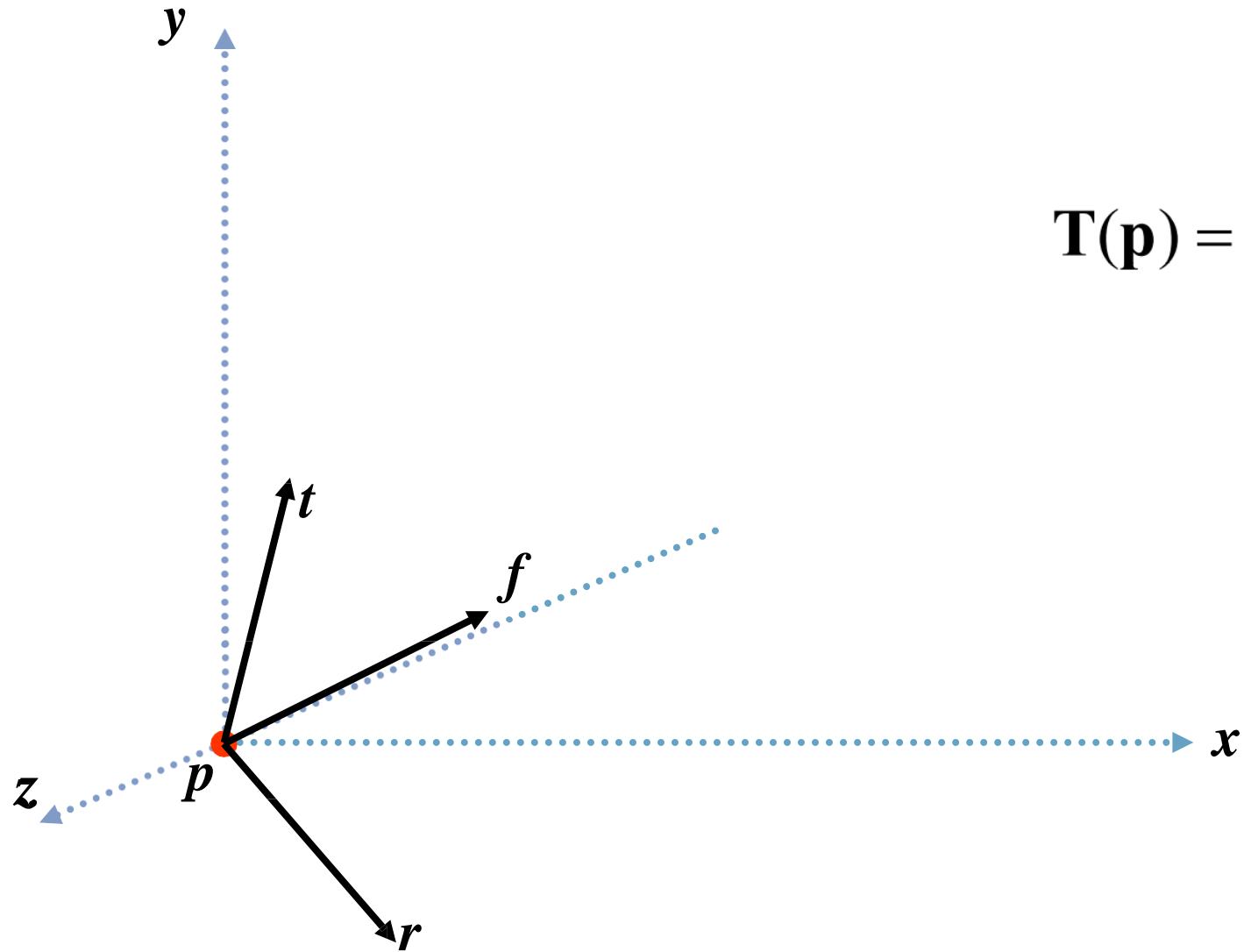


View Transform 4



View Transform 5

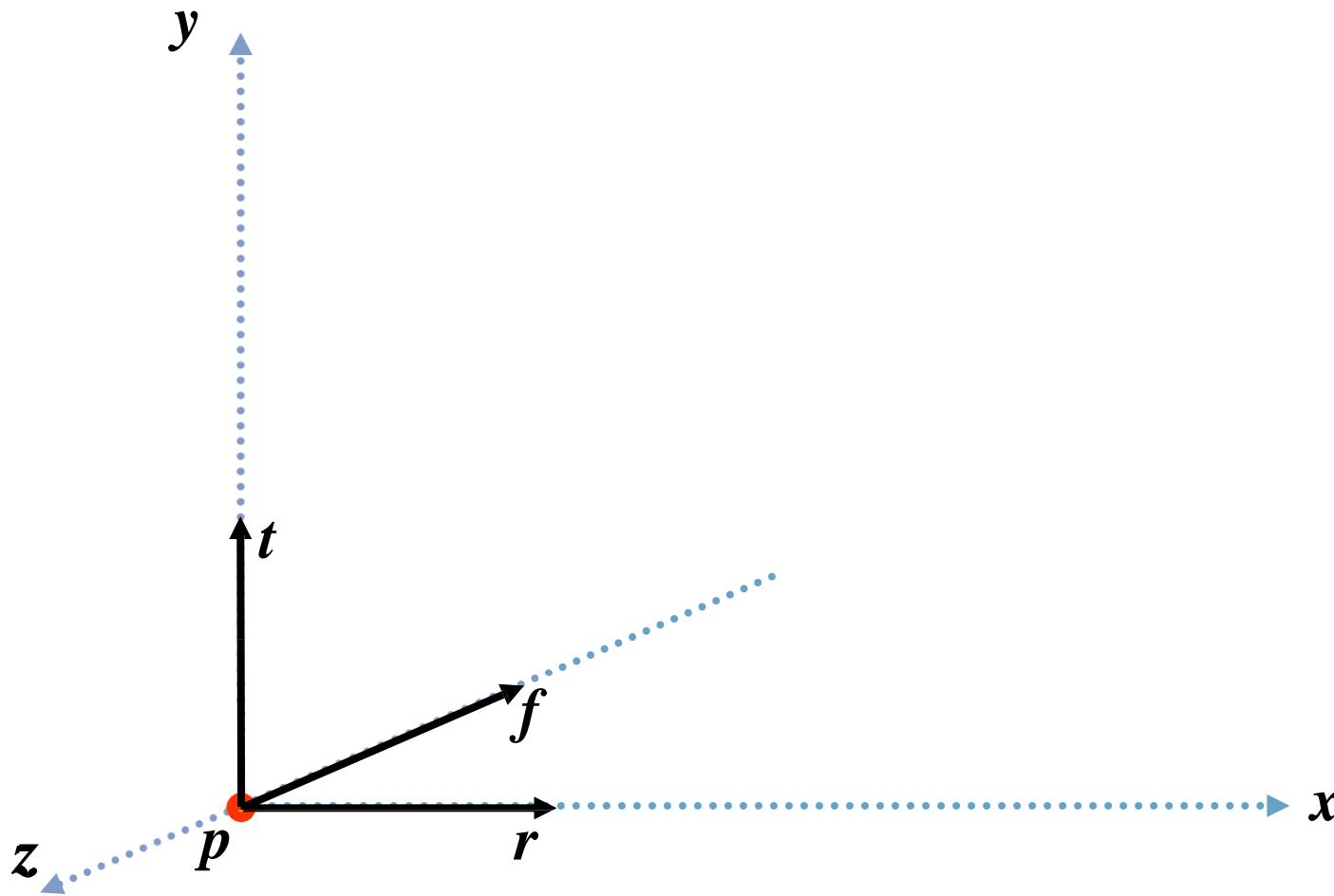
- shift



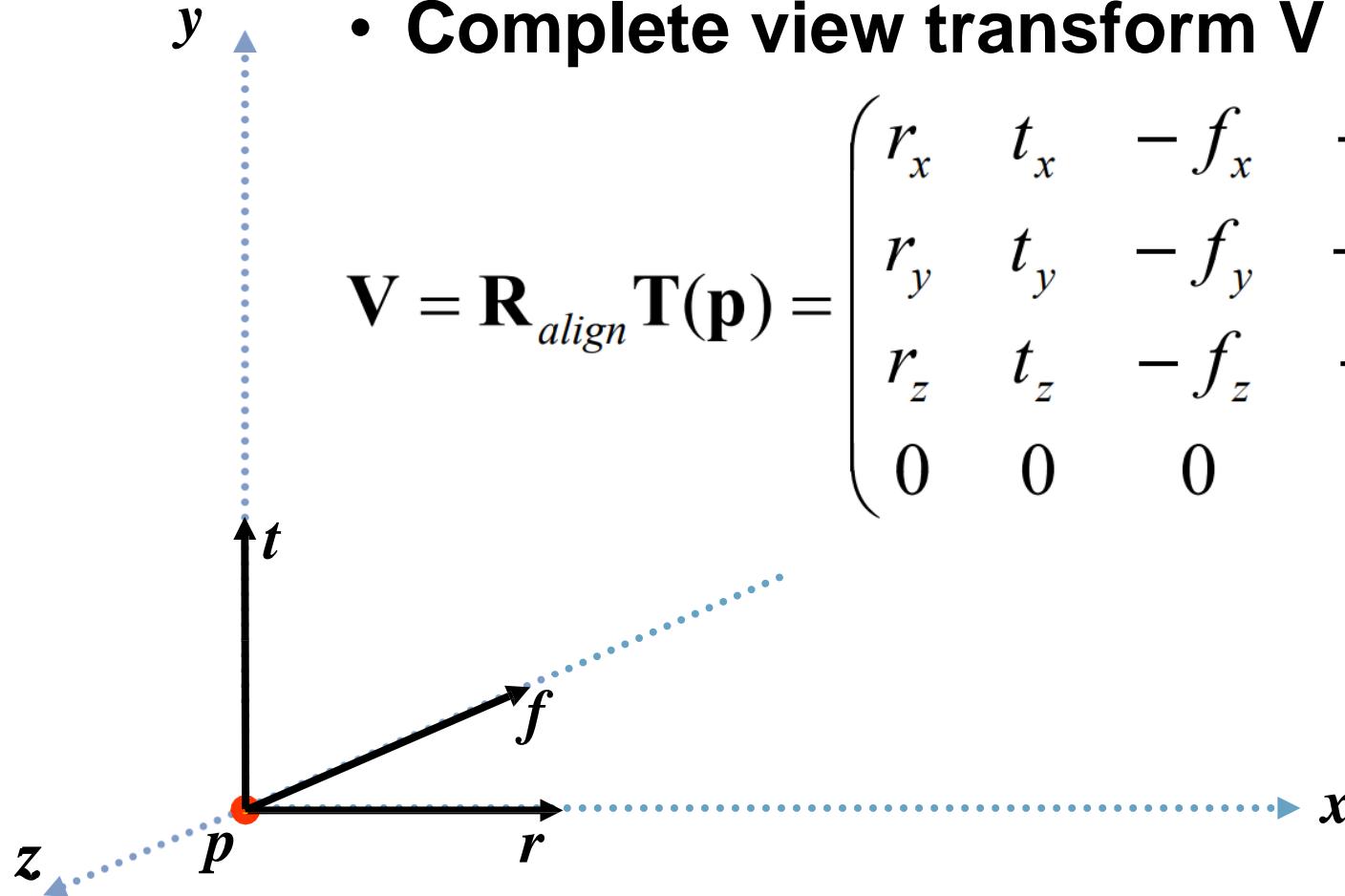
$$T(p) = \begin{pmatrix} 1 & 0 & 0 & -p_x \\ 0 & 1 & 0 & -p_y \\ 0 & 0 & 1 & -p_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

View Transform 6

- align



View Transform 7

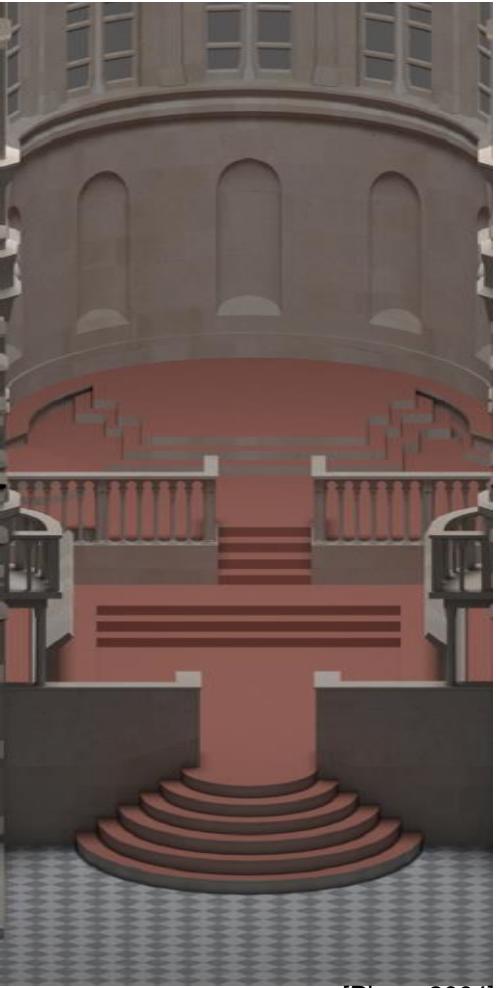


Projections

Projection

- Why?
 - 3d scene → 2d screen
- Most common
 - orthographic projection
 - perspective projection

Example



← orthographic

perspective →



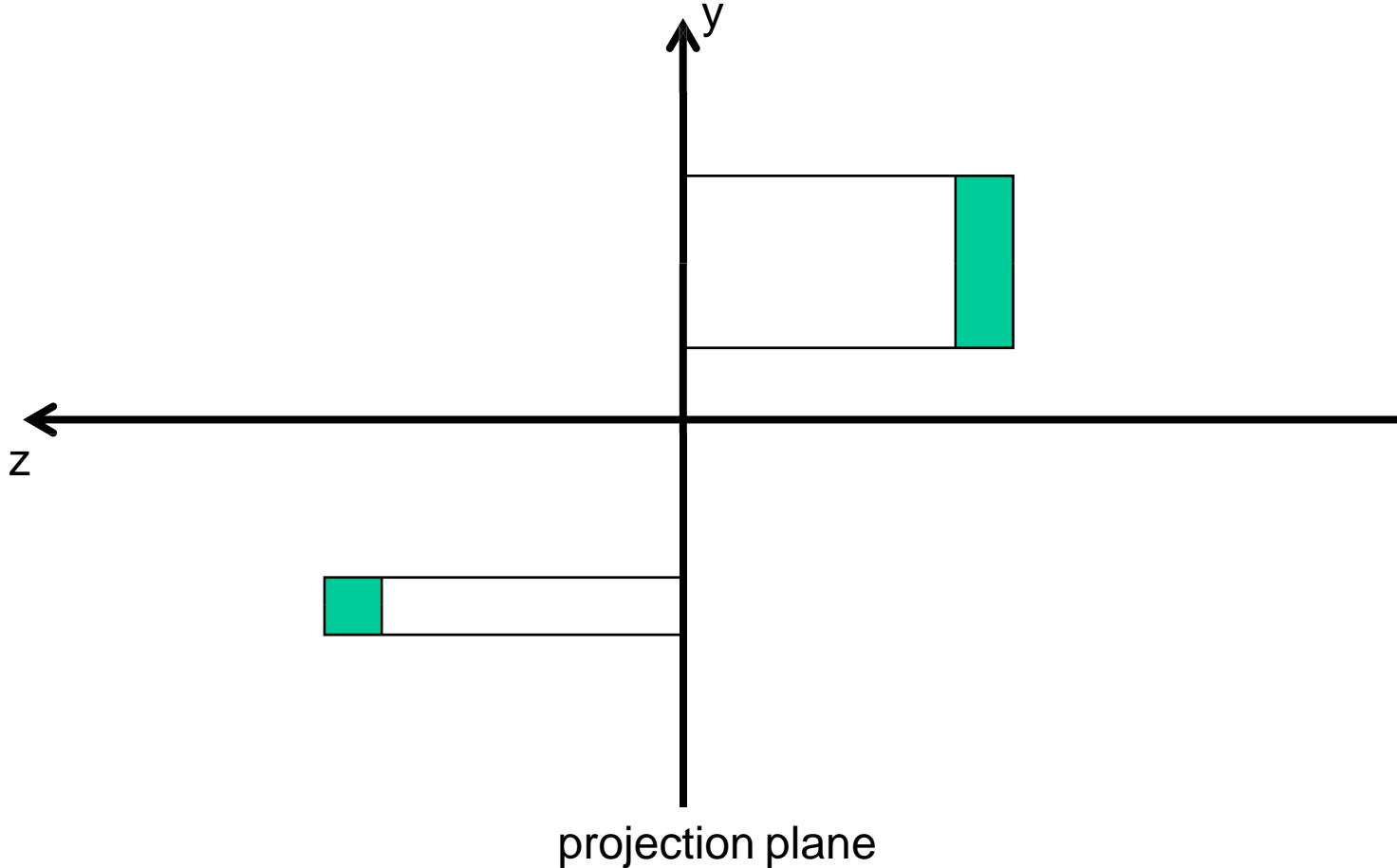
[Pharr, 2004]

Simple Orthographic Projection 1

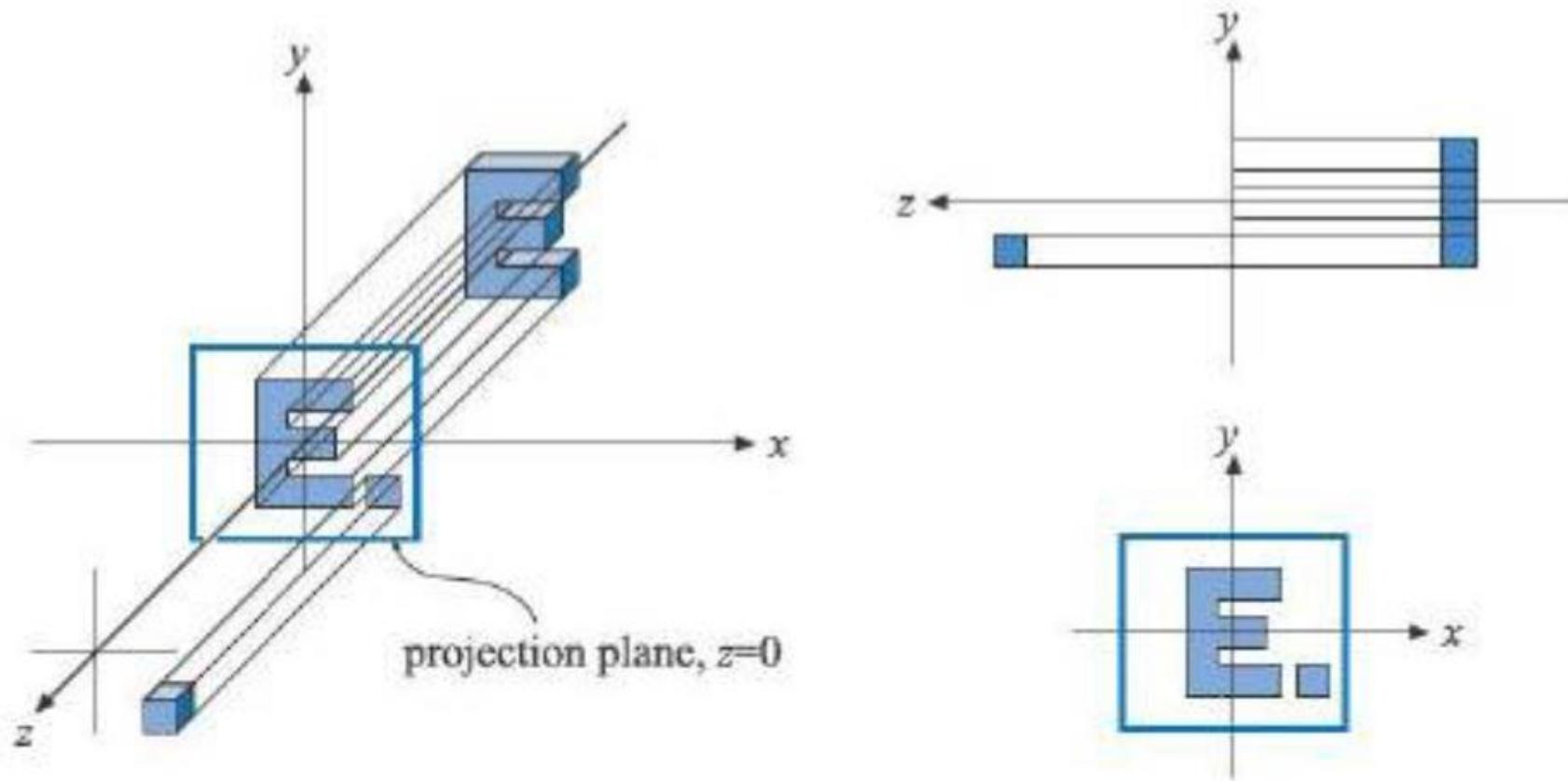
- Parallel lines remain parallel
- Matrix P_o
 - simple orthogr. projection matrix
 - leaves x- and y-components unchanged
 - z-component set to 0
 - → orthogr. projection onto plane $z = 0$

$$P_o = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Simple Orthographic Projection 2



Orthographic Projection 1



Real-Time Rendering Figure 4.17

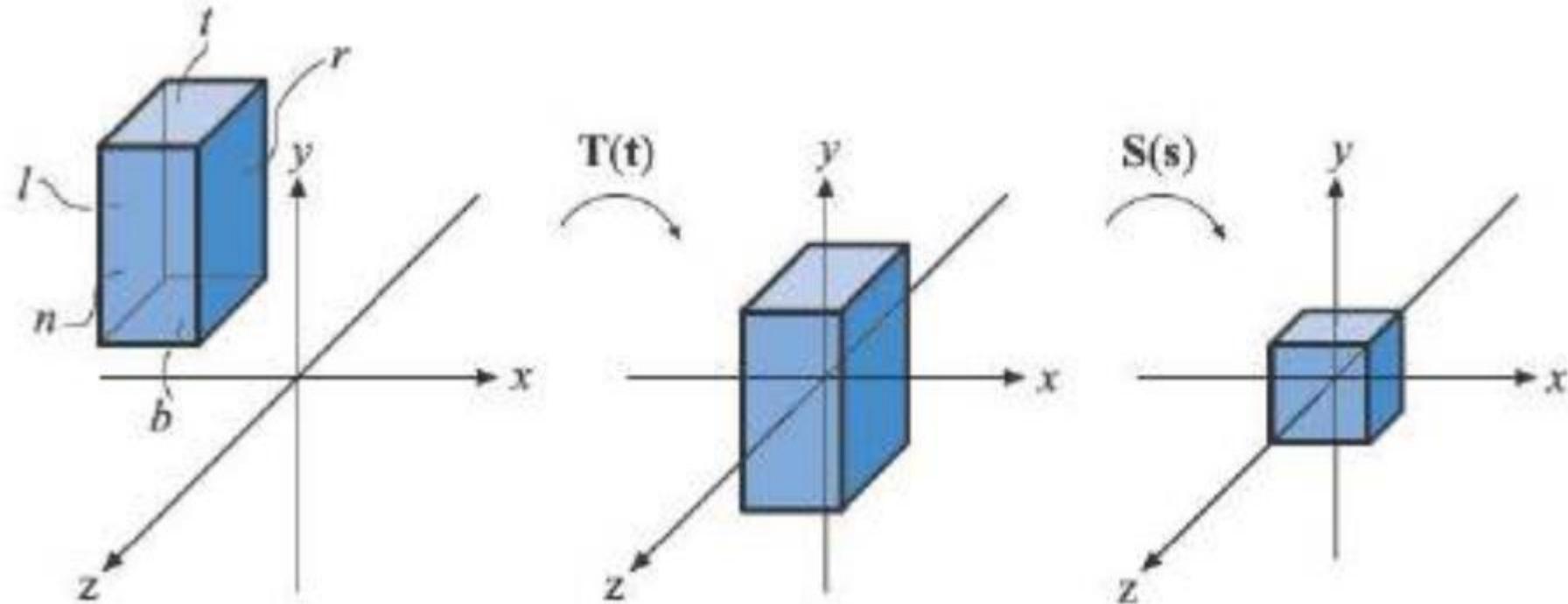
Orthographic Projection 2

- **Problem**
 - P_o projects points with positive and negative z-values onto projection plane
- **Better**
 - Restrict z-values to a certain interval
 - $n \leq z \leq f$
 - $n \dots$ near plane
 - $f \dots$ far plane

Common Orthographic Projection

- projection expressed as six-tuple
 - (l, r, b, t, n, f)
- left, right, bottom, top, near, far
- transform AABB with min corner (l, b, n) and max corner (r, t, f) to canonical view volume ...
- which is an axis-aligned box centered around the origin: min $(-1, -1, -1)$ and max $(1, 1, 1)$

Orthographic Projection 3



Real-Time Rendering Figure 4.18

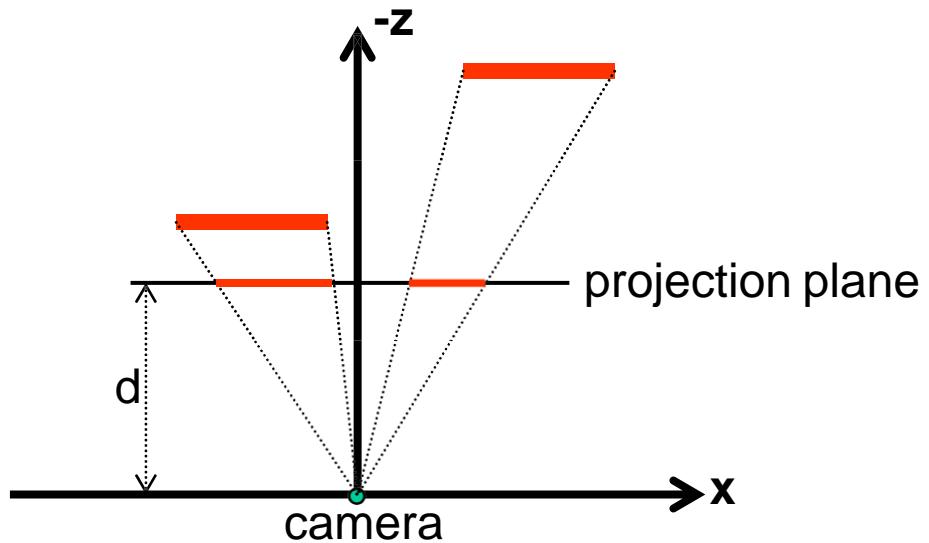
$$\mathbf{t} = \left(-\frac{r+l}{2}, -\frac{t+b}{2}, -\frac{f+n}{2} \right) \quad \mathbf{s} = \left(\frac{2}{r-l}, \frac{2}{t-b}, \frac{2}{f-n} \right)$$

Orthographic Projection 4

$$\mathbf{P}_o = \mathbf{S}(\mathbf{s})\mathbf{T}(\mathbf{t}) = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$
$$= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{l+r}{2} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{2} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Perspective Projection 1

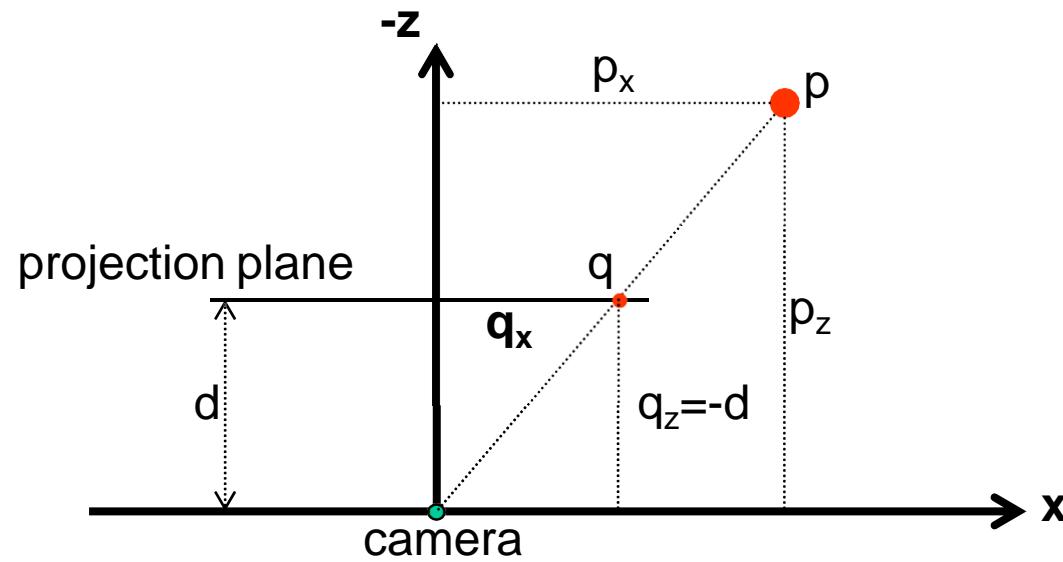
- parallel lines (generally) not parallel after projection
- objects further away appear smaller
- more closely matches how we perceive the world



Simple Perspective Projection 1

Point p is projected onto projection plane

q ... projected point

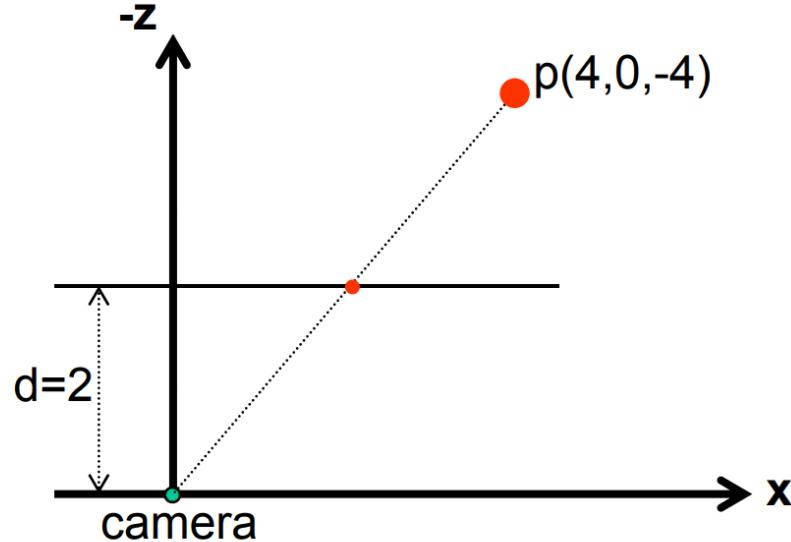


$$\frac{q_x}{-d} = \frac{p_x}{p_z} \quad \Leftrightarrow \quad q_x = -d \frac{p_x}{p_z}$$

Simple Perspective Projection 2

- Matrix formulation for 3d space

$$\mathbf{P}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix}$$

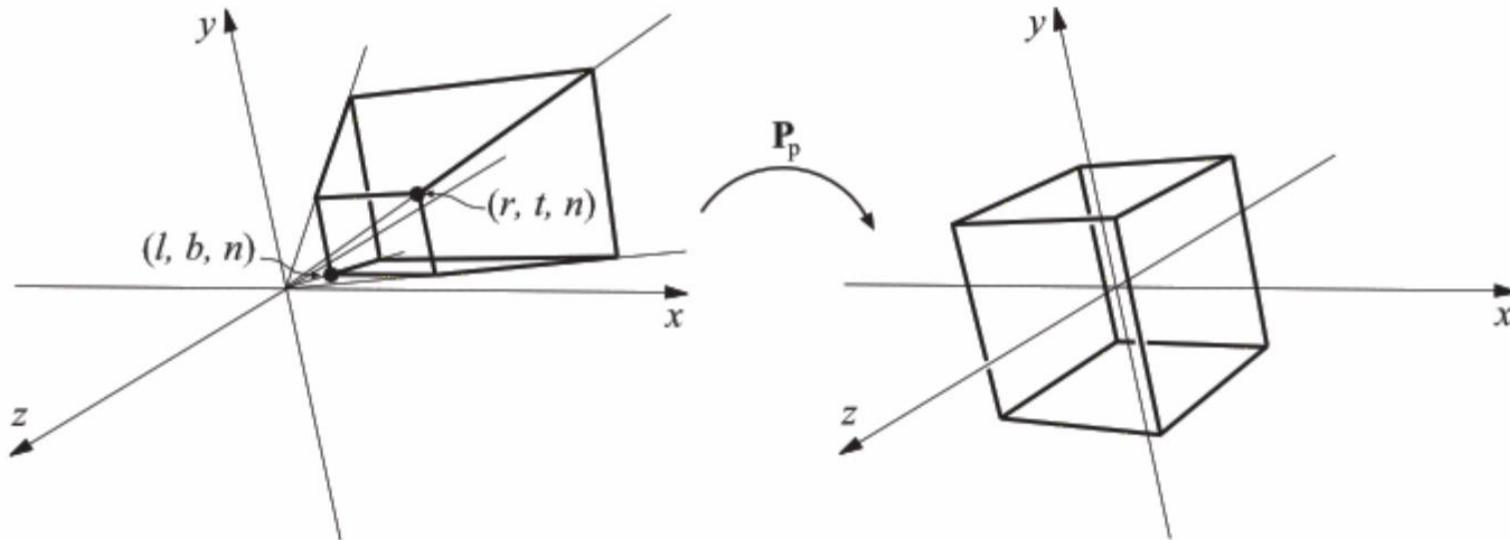


$$\mathbf{q} = \mathbf{P}_p \mathbf{p} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/2 & 0 \end{pmatrix} \begin{pmatrix} 4 \\ 0 \\ -4 \\ 1 \end{pmatrix} = \begin{pmatrix} 4 \\ 0 \\ -4 \\ 2 \end{pmatrix} \Rightarrow \begin{pmatrix} 2 \\ 0 \\ -2 \\ 1 \end{pmatrix}$$

Homogenization required!

Perspective Projection 2

- We also want clipping planes here (left, right, bottom, top, near, far)



Real-Time Rendering Figure 4.20

Viewing Frustum

Canonical View Volume

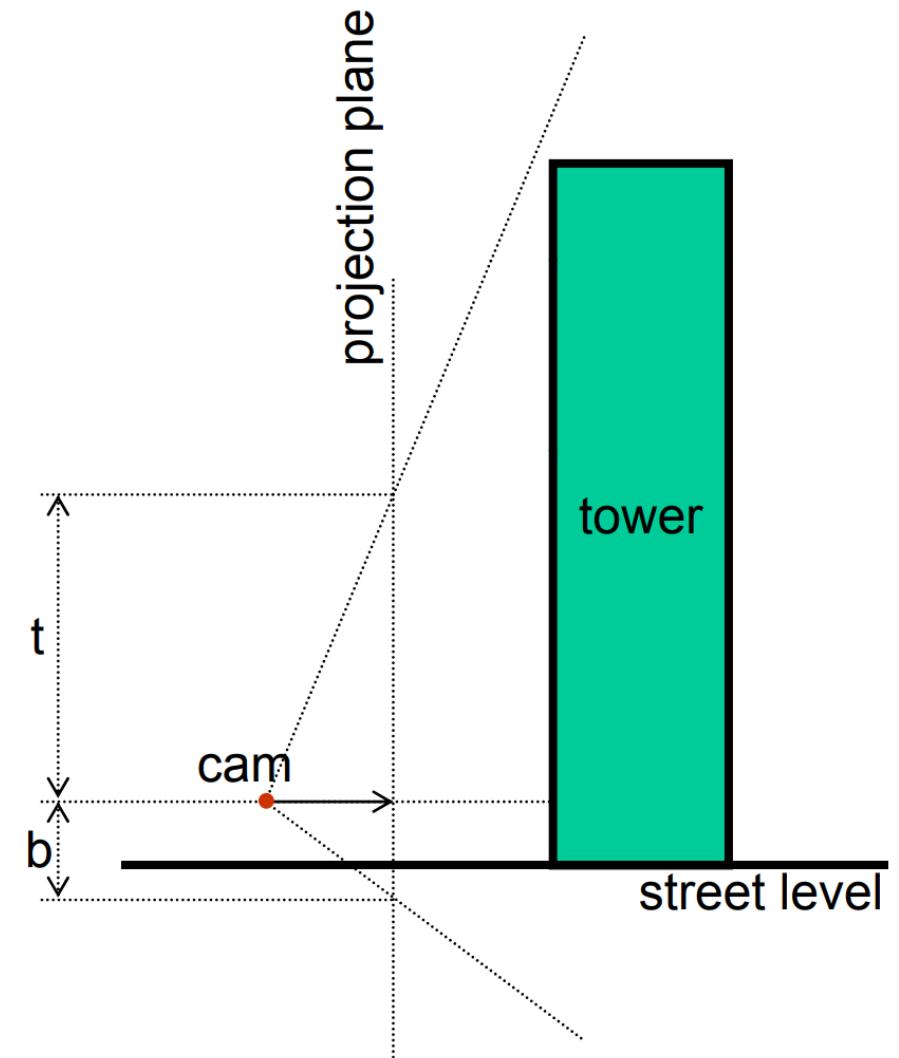
Perspective Projection 3

Asymmetric frustums with $r \neq -l$ or $t \neq -b$
are possible!

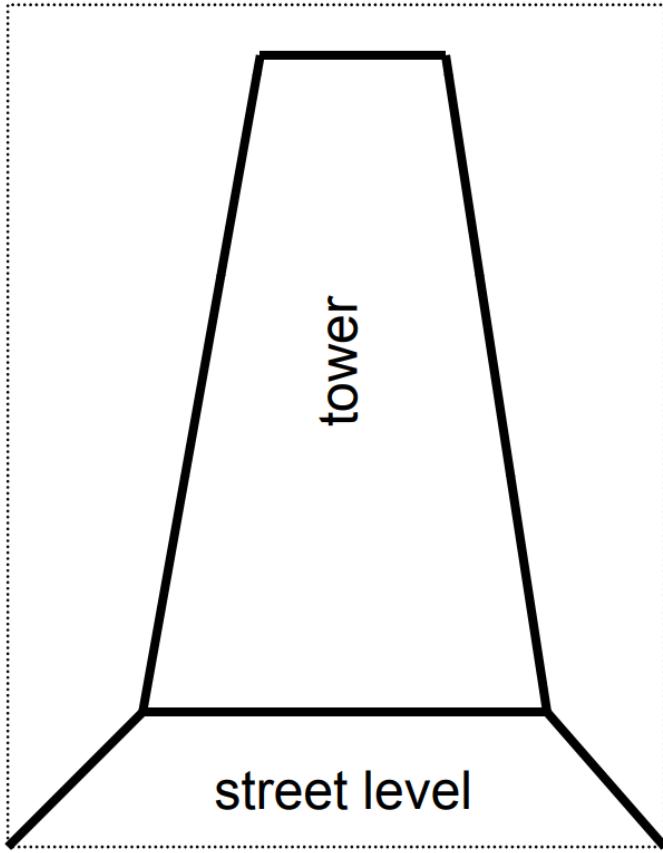
=> Off-Axis Camera

Used for

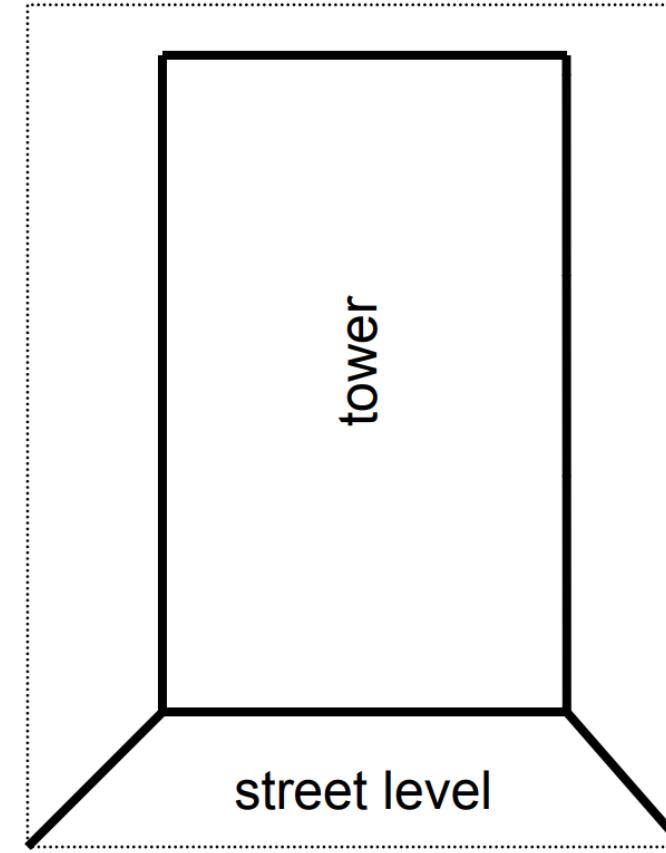
- stereo viewing, CAVEs
- architectural models



Off-Axis Projection



**symmetric
(conventional)**



asymmetric

Field of View

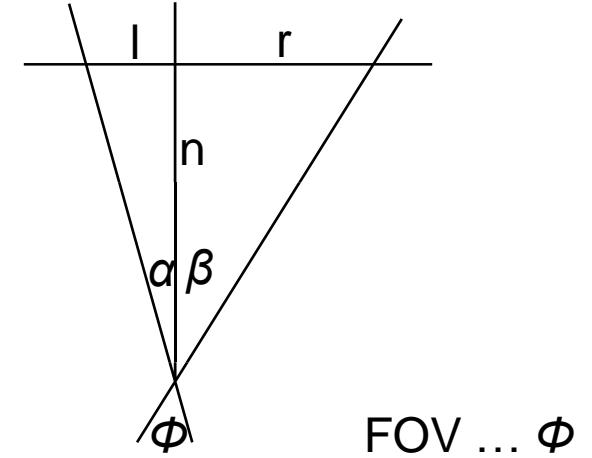


large FOV
(wide angle)



small FOV
(tele angle)

<https://digital-photography-school.com/>



$$\phi = \arctan \frac{|l|}{n} + \arctan \frac{|r|}{n}$$

Perspective Projection (final)

$$\mathbf{P}_p = \begin{pmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$z = f$: maps to +1

$z = n$: maps to -1

1. projection

2. clipping and homogenization

⇒ result:

normalized device coordinates

Perspective Projection (OpenGL)

$$\mathbf{P}_{OpenGL} = \begin{pmatrix} \frac{2n'}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n'}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f'+n'}{f'-n'} & -\frac{2fn'}{f'-n'} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

OpenGL:

near and **far values** are entered as **positive** values, with $0 < n' < f'$

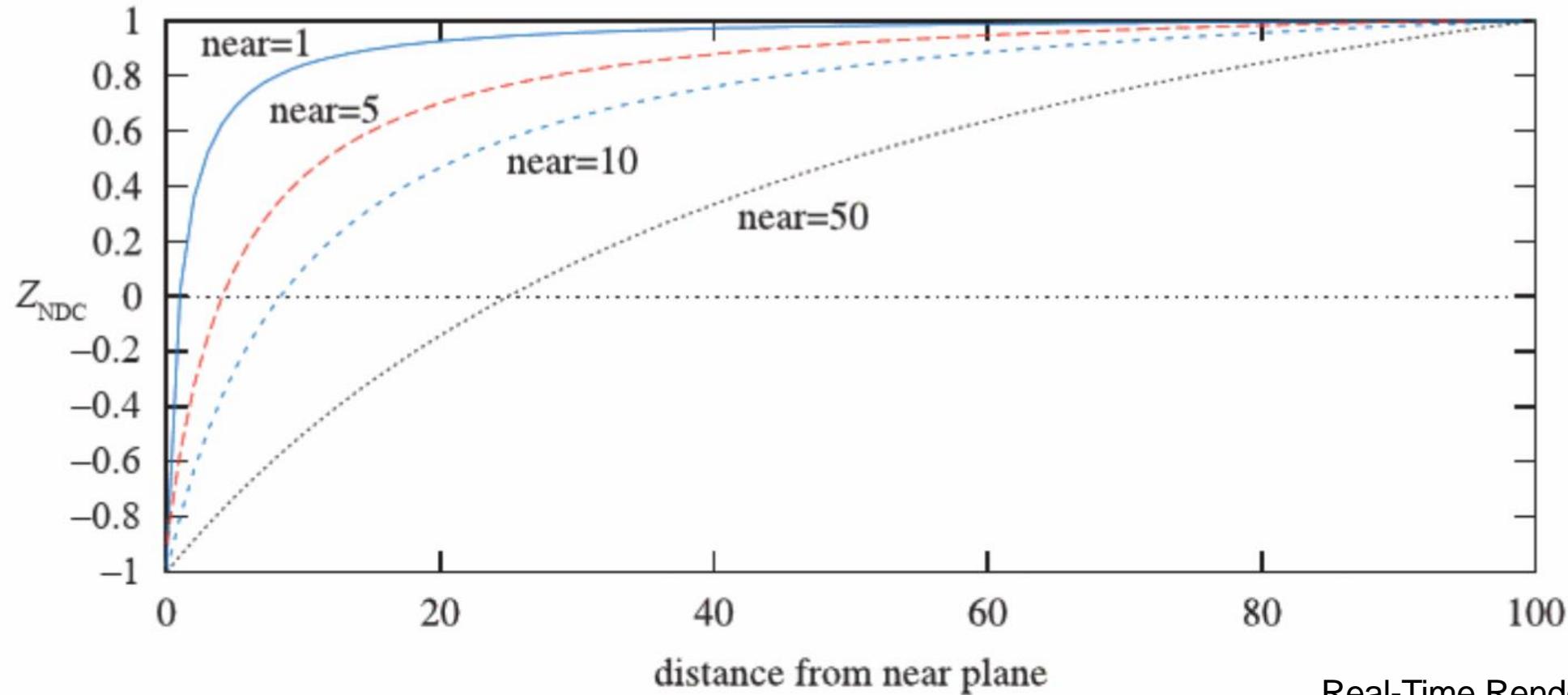
Perspective Projection (DirectX)

$$\mathbf{P}_{p[0,1]} = \begin{pmatrix} \frac{2n'}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n'}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f'}{f'-n'} & -\frac{f'n'}{f'-n'} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

DirectX:

near plane is mapped to 0 (instead of -1), $z = n$: maps to 0

Depth Values



**non-linear NDC depth diagram:
 $f - n = 100$; horizontal: distance from near plane**

Real-Time Rendering Figure 4.21

Special Projections and Cameras

(not necessarily supported by hardware)

Environment Camera



The environmental projection is nonlinear and cannot be captured by a single 4×4 matrix.

<https://pbr-book.org/>
(2018)

Depth of Field



<https://pbr-book.org/>
(2018)

Depth of field gives a greater sense of depth and scale to this part of the landscape scene. (*Scene courtesy of Laubwerk.*)

The background image shows an aerial view of the Vienna city skyline during sunset. The sky is filled with warm orange and yellow hues. In the foreground, there are several modern white buildings with many windows. To the right, there are older, more traditional buildings and some industrial structures with tall chimneys emitting smoke. The overall atmosphere is urban and architectural.

FH

University of
Applied Sciences

TECHNIKUM WIEN

Transformations

Overview

- **Basics**
 - **Vectors, Matrices, ...**
- **Transforms (2d, 3d)**
- **Homogeneous Coordinates**
- **Special Transforms**

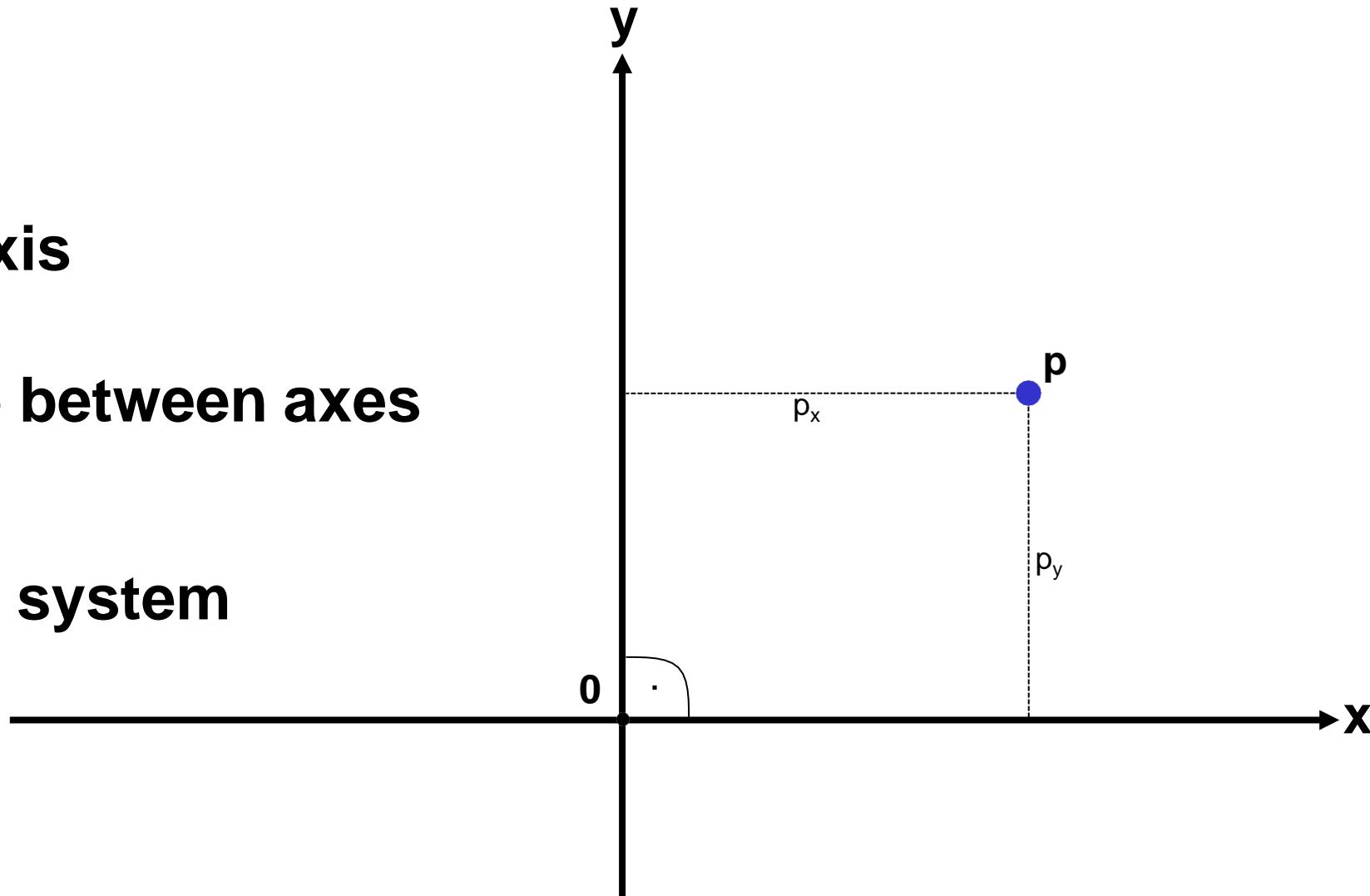
2-dim Euclidean* Space

origin ... 0

x-axis, y-axis

right angle between axes

coordinate system



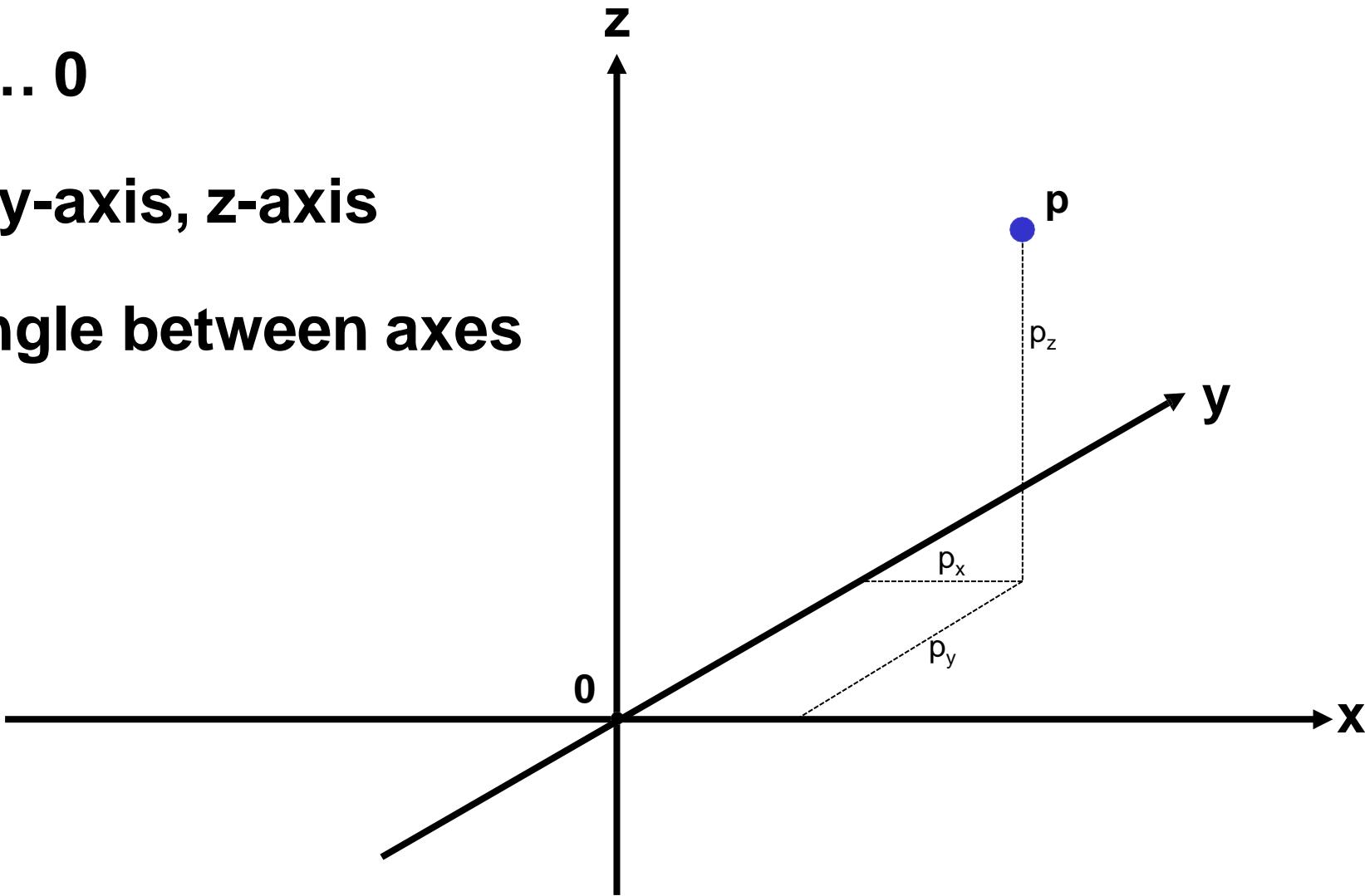
* also known as **Cartesian Space**

3-dim Euclidean Space

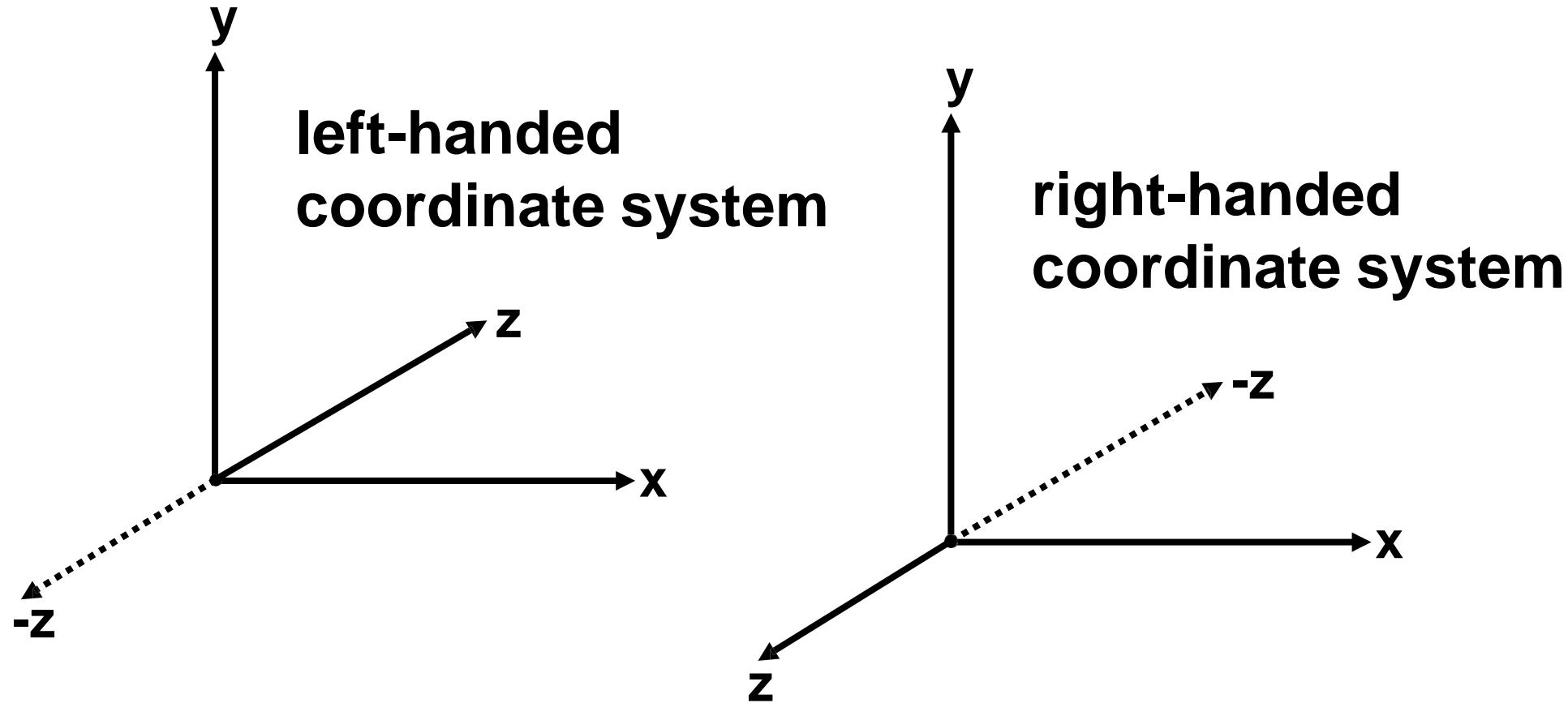
origin ... 0

x-axis, y-axis, z-axis

right angle between axes



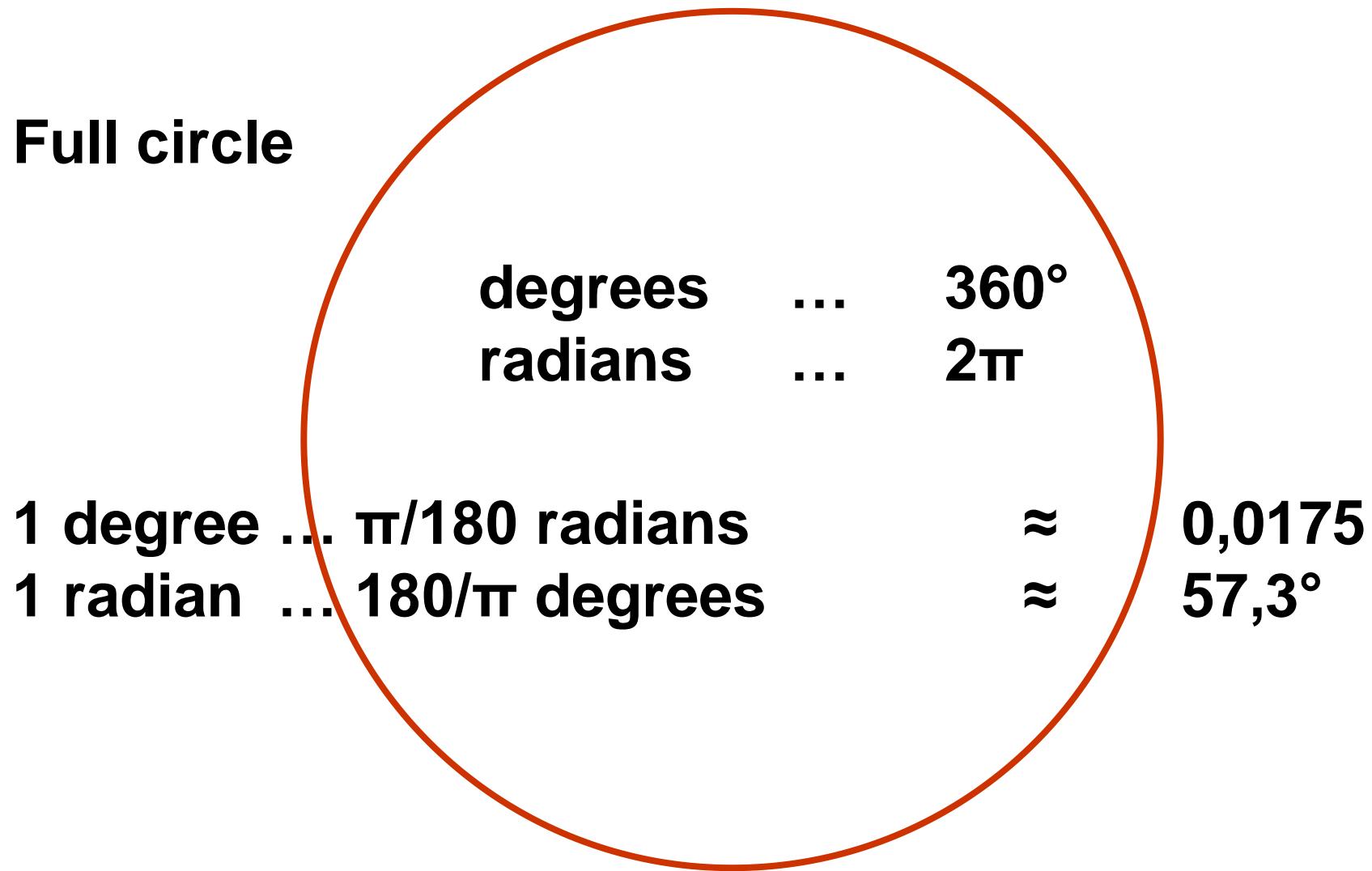
Handedness



Notation

- angle
 - lowercase Greek, e.g. $\alpha_i, \phi, \rho, \theta$
- scalar
 - lowercase italic, e.g. a, b, t, u_k, v, w_{ij}
- vector or point
 - lowercase bold, e.g. $\mathbf{a}, \mathbf{u}, \mathbf{v}_s, \mathbf{h}(\rho), \mathbf{h}_z$
- matrix
 - capital bold, e.g. $\mathbf{M}, \mathbf{T}(t), \mathbf{X}, \mathbf{R}_x(\rho)$

Radians and Degrees

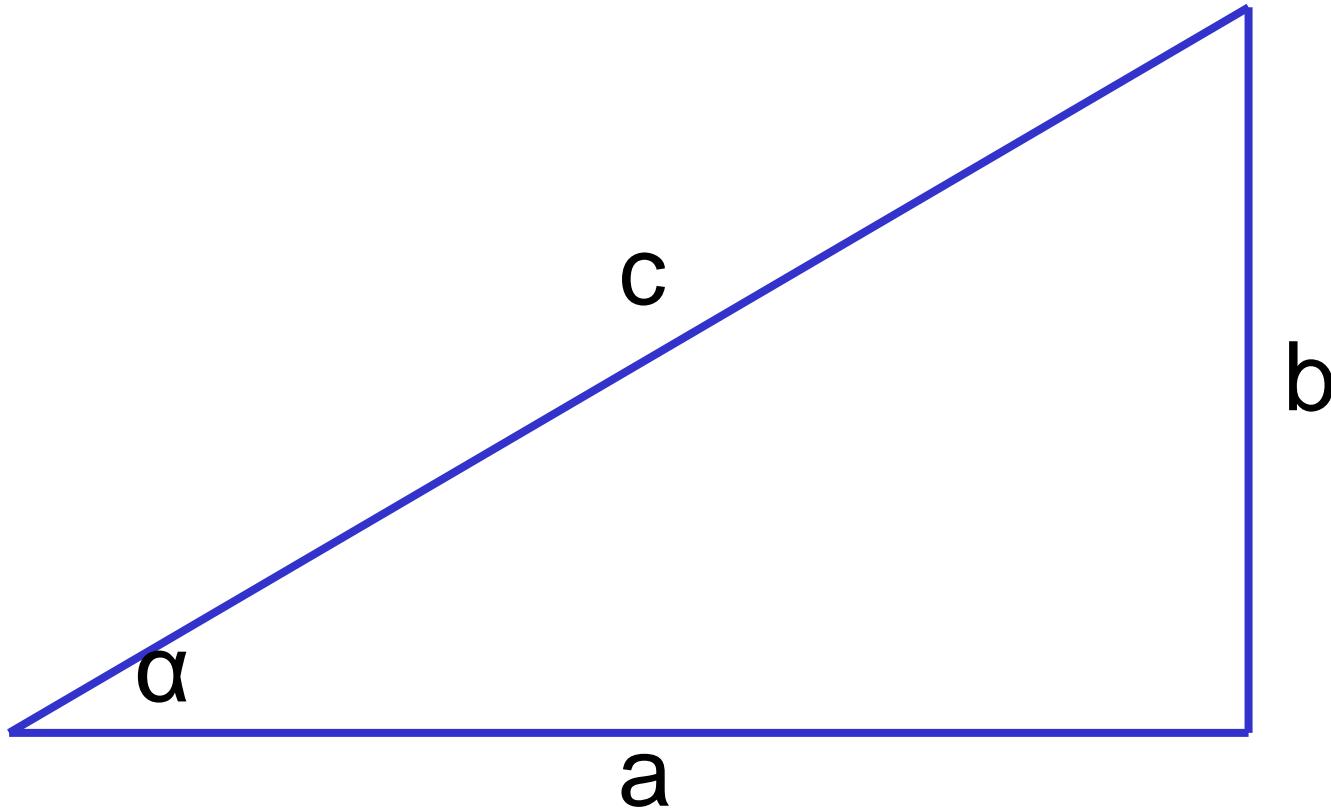


Basic Trigonometry 1

$$\sin \alpha = \frac{b}{c}$$

$$\cos \alpha = \frac{a}{c}$$

$$\tan \alpha = \frac{b}{a}$$



Basic Trigonometry 2

$$\sin(0^\circ) = 0$$

$$\sin(90^\circ) = 1$$

$$\sin(180^\circ) = 0$$

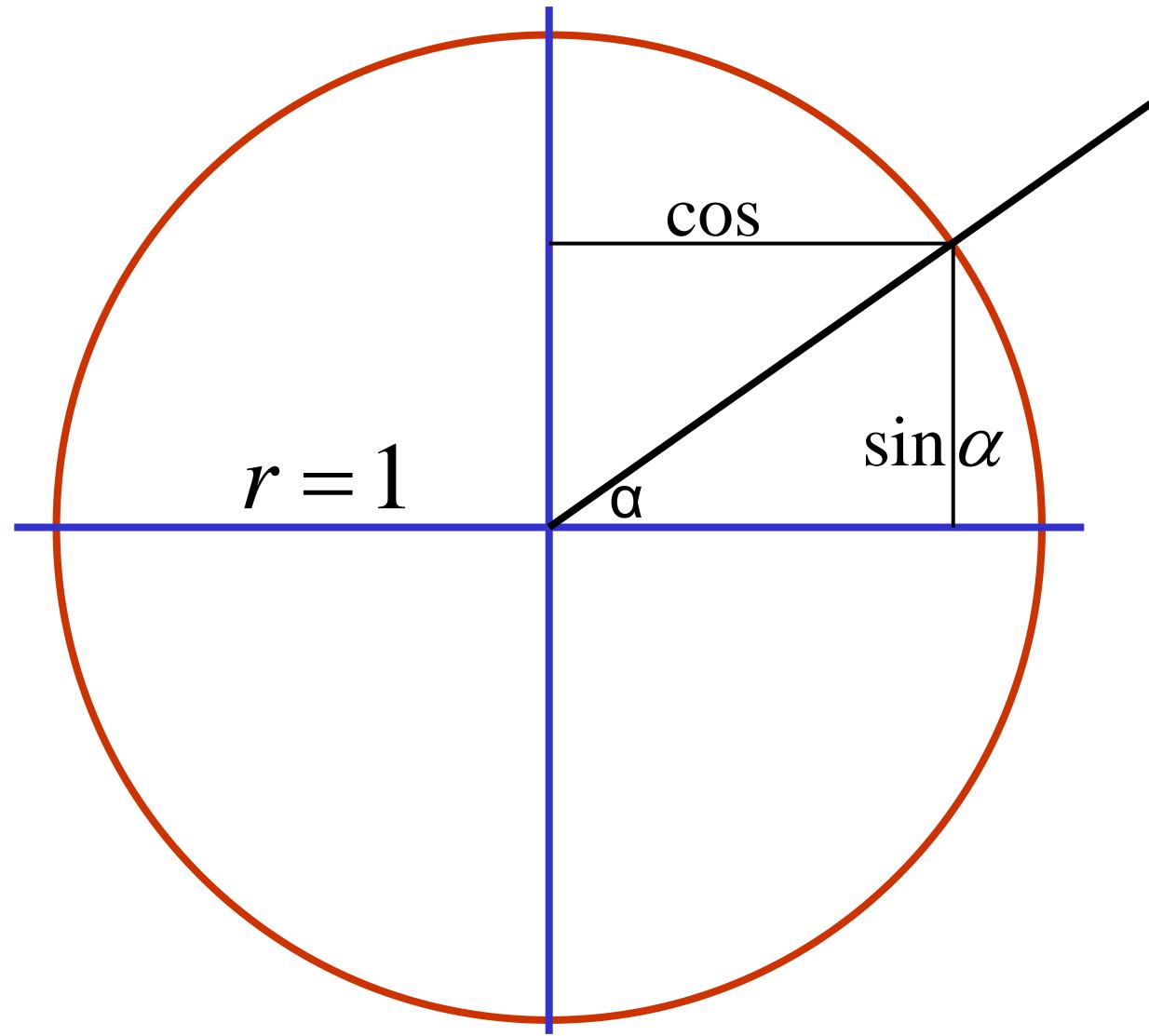
$$\sin(270^\circ) = -1$$

$$\cos(0^\circ) = 1$$

$$\cos(90^\circ) = 0$$

$$\cos(180^\circ) = -1$$

$$\cos(270^\circ) = 0$$



Do-It-Yourself – Calculate in Radians

$$\sin(\pi/2) = ?$$

$$\cos(\pi) = ?$$

$$\sin(1.5 \pi) = ?$$

$$\cos(2 \pi) = ?$$

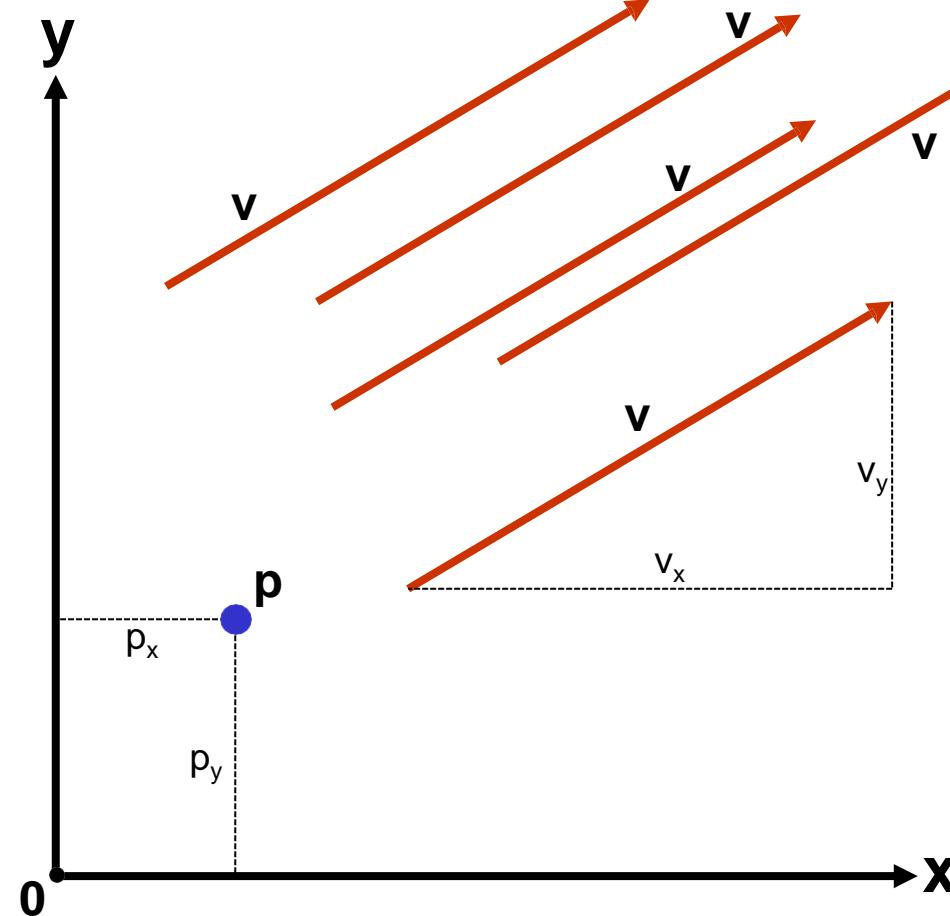
Points and Vectors

2-dim

$$v = \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

3-dim

$$v = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$



Operations 1

add / subtract
vector/vector

$$\begin{pmatrix} a_x \\ a_y \\ \vdots \end{pmatrix} \pm \begin{pmatrix} b_x \\ b_y \\ \vdots \end{pmatrix} = \begin{pmatrix} a_x \pm b_x \\ a_y \pm b_y \\ \vdots \end{pmatrix}$$

multiplication
scalar/vector

$$s \begin{pmatrix} a_x \\ a_y \\ \vdots \end{pmatrix} = \begin{pmatrix} sa_x \\ sa_y \\ \vdots \end{pmatrix}$$

Operations 2

dot product
vector/vector

$$\mathbf{a} \cdot \mathbf{b} = \begin{pmatrix} a_x \\ a_y \\ \vdots \end{pmatrix} \cdot \begin{pmatrix} b_x \\ b_y \\ \vdots \end{pmatrix} = a_x b_x + a_y b_y + \dots$$

cross product
vector/vector

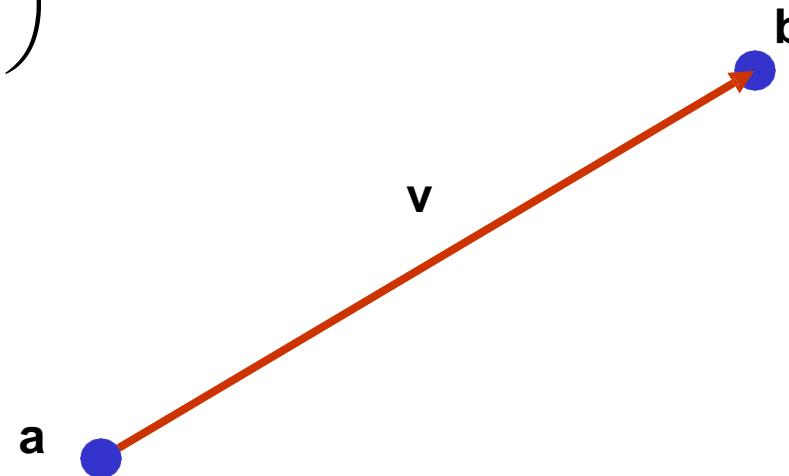
$$\mathbf{a} \times \mathbf{b} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \times \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix}$$

Vector Defined by 2 Points

How to calculate vector v from point a to point b :

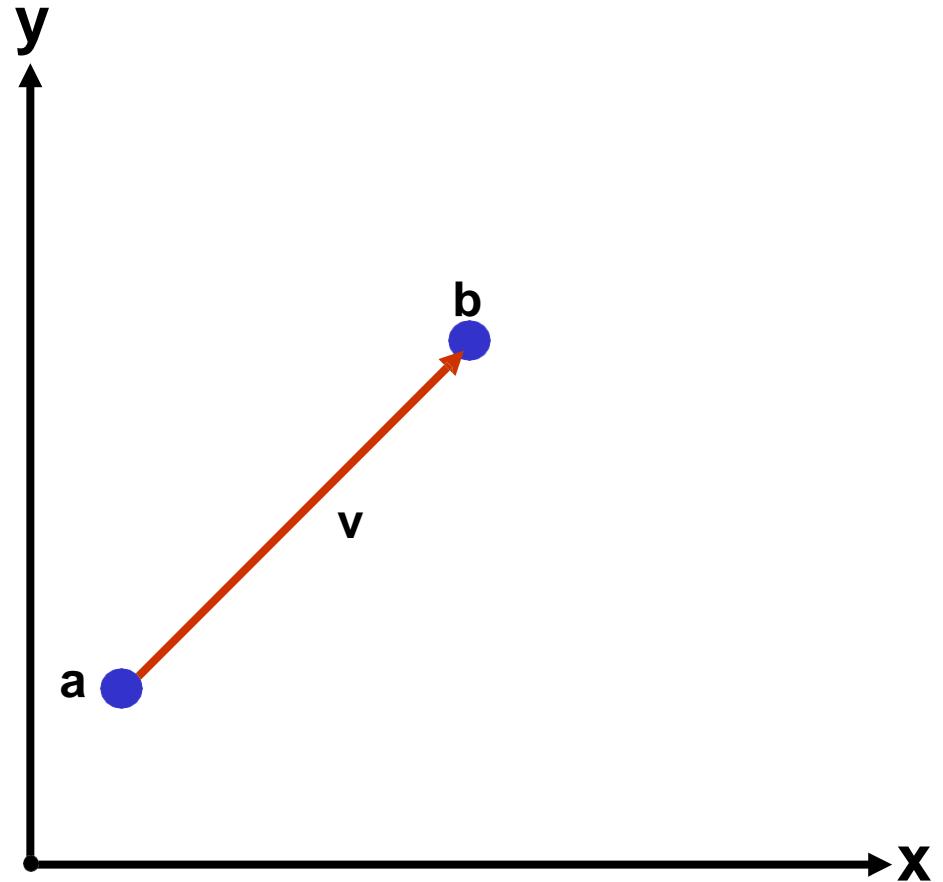
$$\mathbf{v} = \mathbf{b} - \mathbf{a} = \begin{pmatrix} b_x - a_x \\ b_y - a_y \end{pmatrix}$$

$$\mathbf{a} + \mathbf{v} = \mathbf{b}$$



Do-It-Yourself

Calculate the vector from point **a** to point **b**!



$$\mathbf{a} = (1, 2)$$

$$\mathbf{b} = (5, 6)$$

$$\mathbf{v} = ?$$

Normalization

$$\|\mathbf{v}\| = \sqrt{\mathbf{v} \cdot \mathbf{v}} = \left\| \begin{pmatrix} v_x \\ v_y \\ \vdots \end{pmatrix} \right\| = \sqrt{v_x^2 + v_y^2 + \dots}$$

vector normalization

a vector of length 1
is called **normalized vector**
or **unit vector**

$$\mathbf{v}' = \frac{\mathbf{v}}{\|\mathbf{v}\|} \quad \|\mathbf{v}'\| = 1$$

Do-It-Yourself

Which adversary (**a** or **b**) is nearer to player **p**?

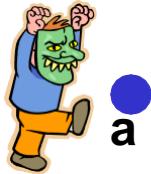
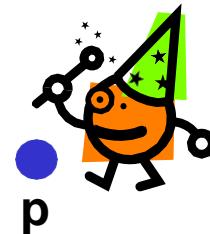
What is the exact distance?



$$\mathbf{p} = (8, 4)$$

$$\mathbf{a} = (4, 1)$$

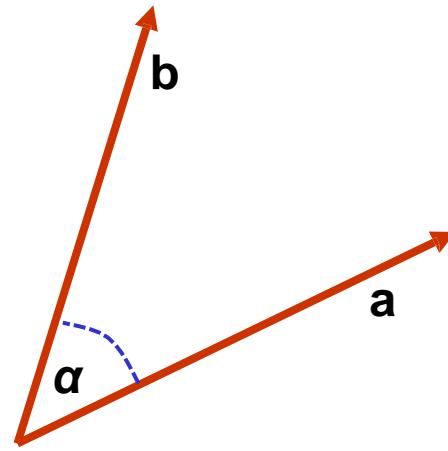
$$\mathbf{b} = (3, 7)$$



Dot Product (more closely)

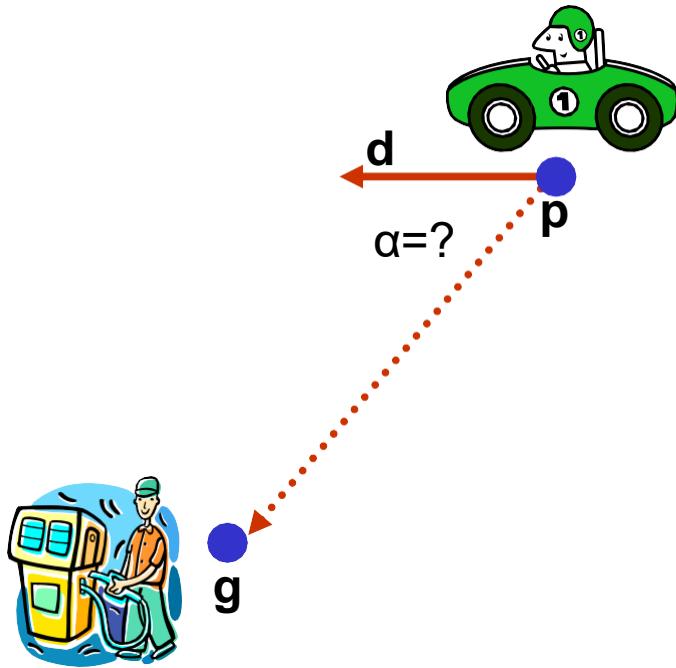
$$\mathbf{a} \cdot \mathbf{b} = \cos(\alpha)$$

where $\|\mathbf{a}\| = \|\mathbf{b}\| = 1$



Do-It-Yourself

Turn how many degrees?

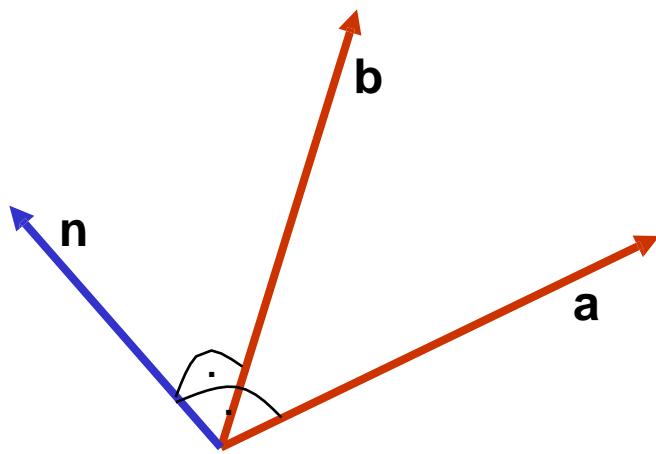


$$p = (10, 10)$$

$$d = (-1, 0)$$

$$g = (4, 3)$$

Cross Product (more closely) 1



$$\mathbf{n} = \mathbf{a} \times \mathbf{b}$$

$$\mathbf{a} \cdot \mathbf{n} = \mathbf{b} \cdot \mathbf{n} = 0$$

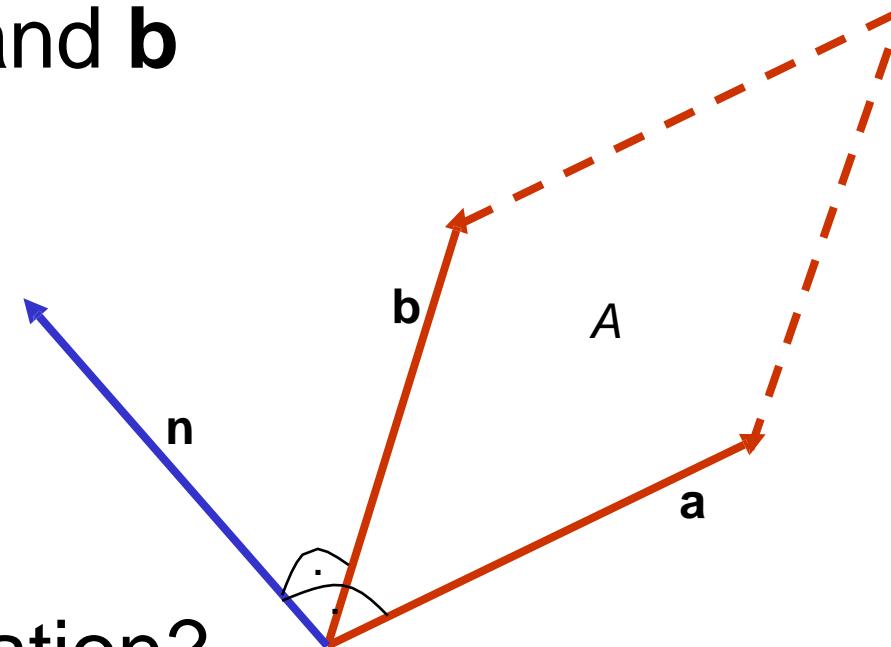
n is orthogonal to both **a** and **b**

Cross Product (more closely) 2

A ... area of parallelogram
defined by vectors \mathbf{a} and \mathbf{b}

$$A = \|\mathbf{a} \times \mathbf{b}\|$$

- geometric interpretation?
- area of triangle?



Do-It-Yourself

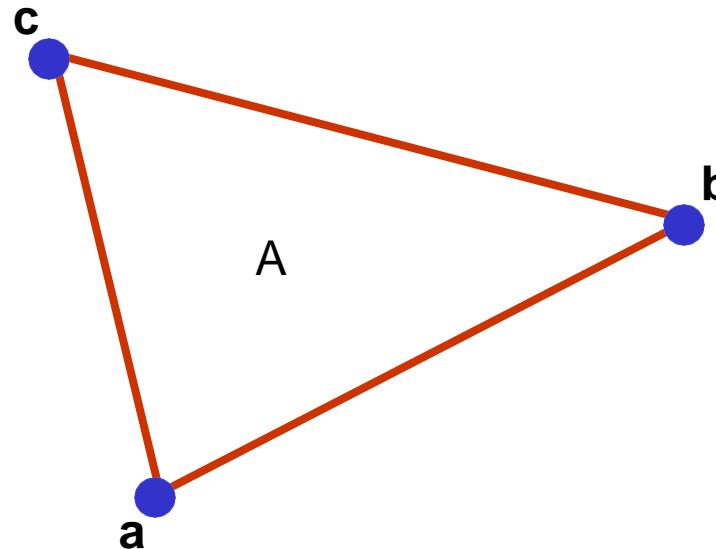
How large is this triangle? Calculate the area A of triangle **abc**!
Use the cross product!

$$\mathbf{a} = (2, 1)$$

$$\mathbf{b} = (8, 4)$$

$$\mathbf{c} = (0, 7)$$

$$A = ?$$



Matrices

2x2 Matrix

$$\mathbf{M} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

3x3 Matrix

$$\mathbf{M} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{22} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

Matrix/Vector Multiplication

$$\mathbf{Ab} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_x \\ b_y \end{pmatrix} =$$
$$= \begin{pmatrix} a_{11}b_x + a_{12}b_y \\ a_{21}b_x + a_{22}b_y \end{pmatrix} = \mathbf{r}$$

\mathbf{r}_i is dot-product of i^{th} row from \mathbf{A} and \mathbf{b}

Matrix/Vector Multiplication

$$\mathbf{Ab} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} =$$
$$= \begin{pmatrix} a_{11}b_x + a_{12}b_y + a_{13}b_z \\ a_{21}b_x + a_{22}b_y + a_{23}b_z \\ a_{31}b_x + a_{32}b_y + a_{33}b_z \end{pmatrix} = \mathbf{r}$$

\mathbf{r}_i is dot-product of i^{th} row from \mathbf{A} and \mathbf{b}

Do-It-Yourself

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 7 \\ 1 & 1 \end{pmatrix} = ?$$

Matrix/Matrix Multiplication (2x2)

$$\mathbf{AB} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} =$$
$$\begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix} = \mathbf{R}$$

\mathbf{R}_{ij} is dot-product of i^{th} row from \mathbf{A} and j^{th} column from \mathbf{B}

Matrix/Matrix Multiplication (3x3)

$$\begin{aligned}\mathbf{AB} &= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \\ &= \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{pmatrix} = \\ &= \mathbf{R}\end{aligned}$$

\mathbf{R}_{ij} is dot-product of i^{th} row from \mathbf{A} and j^{th} column from \mathbf{B}

Do-It-Yourself

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 2 & 4 \\ -1 & -2 \end{pmatrix} = ?$$

Identity Matrix

$$I = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

- elements on diagonal are 1
- all other elements are 0

$$\mathbf{IM} = \mathbf{MI} = \mathbf{M}$$

$$\mathbf{Ia} = \mathbf{aI} = \mathbf{a}$$

Do-It-Yourself

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = ?$$

Matrix Transpose

$$\mathbf{M} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots \\ a_{21} & a_{22} & a_{23} & \cdots \\ a_{31} & a_{32} & a_{33} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

$$\mathbf{M}^T = \begin{pmatrix} a_{11} & a_{21} & a_{31} & \cdots \\ a_{12} & a_{22} & a_{32} & \cdots \\ a_{13} & a_{23} & a_{33} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

rows and columns are swapped to create the transpose

We need this again when transforming normal vectors.

Inverse Matrices

The inverse of a matrix M is denoted as

$$M^{-1}$$

The product of M and its inverse yields identity

$$MM^{-1} = I$$

We need this again when transforming normal vectors and to calculate inverse transforms.

Do-It-Yourself

What is the **inverse** of the following matrix?

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

What is the **geometric interpretation** of this matrix?

Orthogonal Matrices

A $n \times n$ matrix M is orthogonal, if

$$MM^T = I$$

The **inverse** of a orthogonal matrix equals its **transpose**, therefore any orthogonal matrix is invertible.

$$M^{-1} = M^T$$

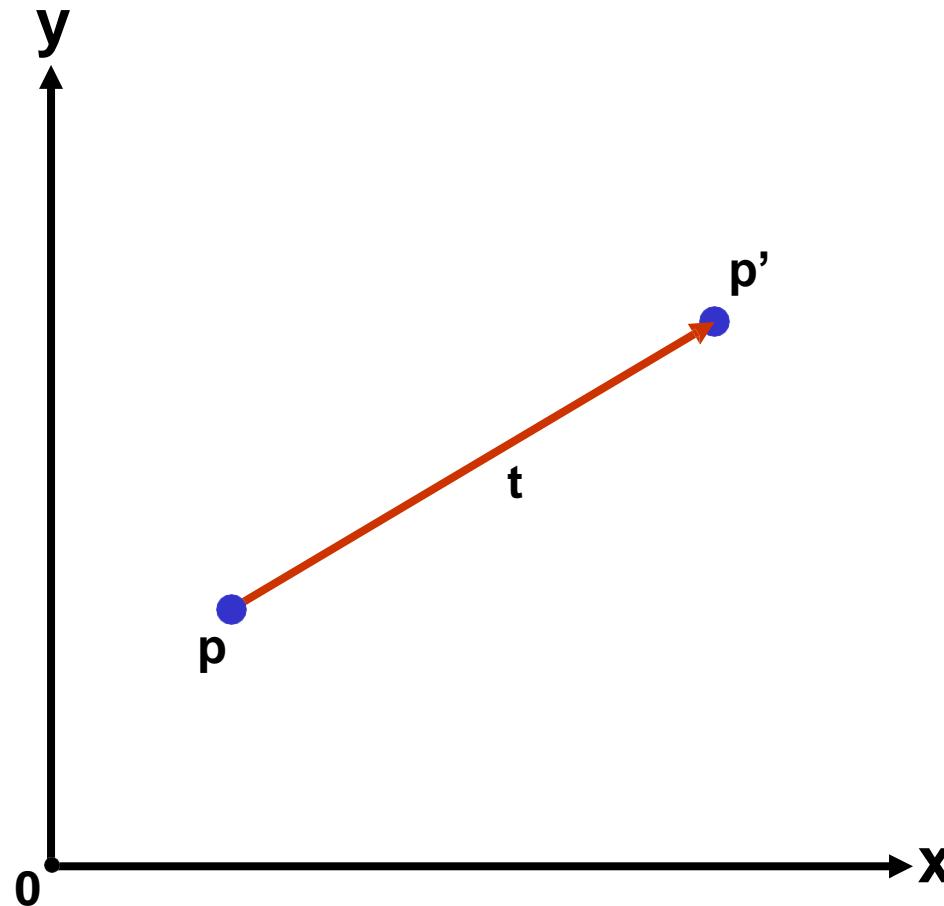
2-dim Transforms

2d Translation

$$\mathbf{p}' = \mathbf{p} + \mathbf{t}$$

$$\mathbf{p}' = \begin{pmatrix} p_x \\ p_y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} = \begin{pmatrix} p_x + t_x \\ p_y + t_y \end{pmatrix}$$

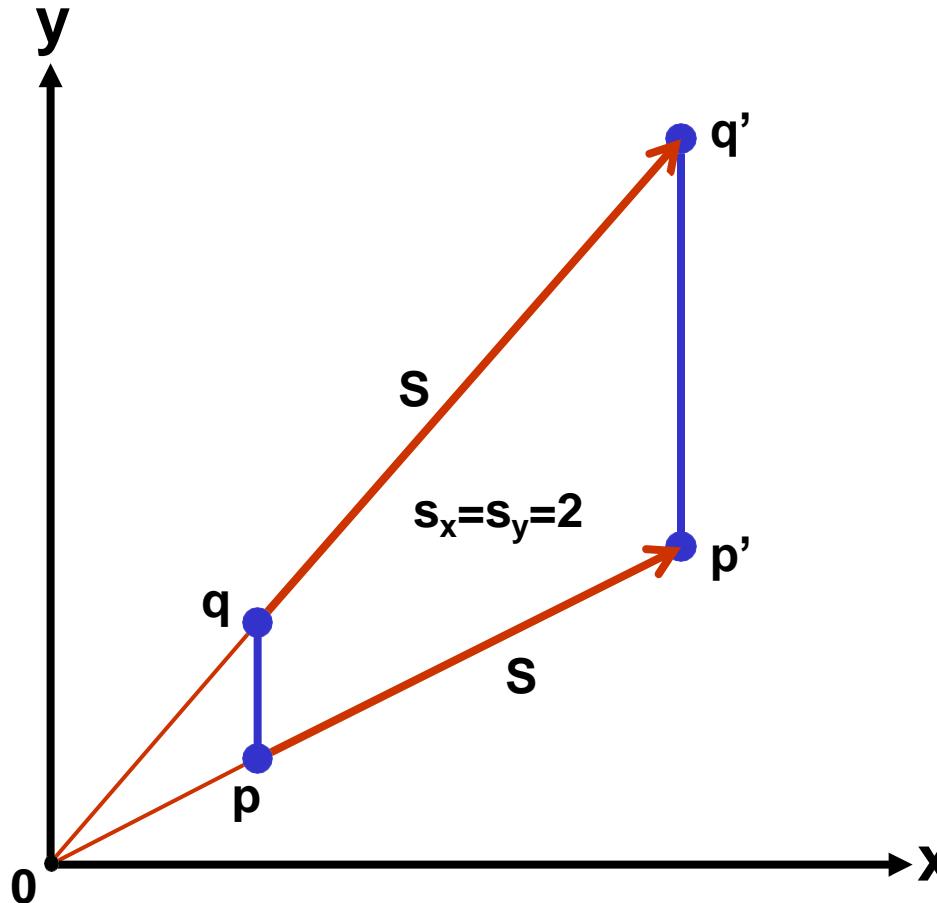
\mathbf{t} ... translation vector



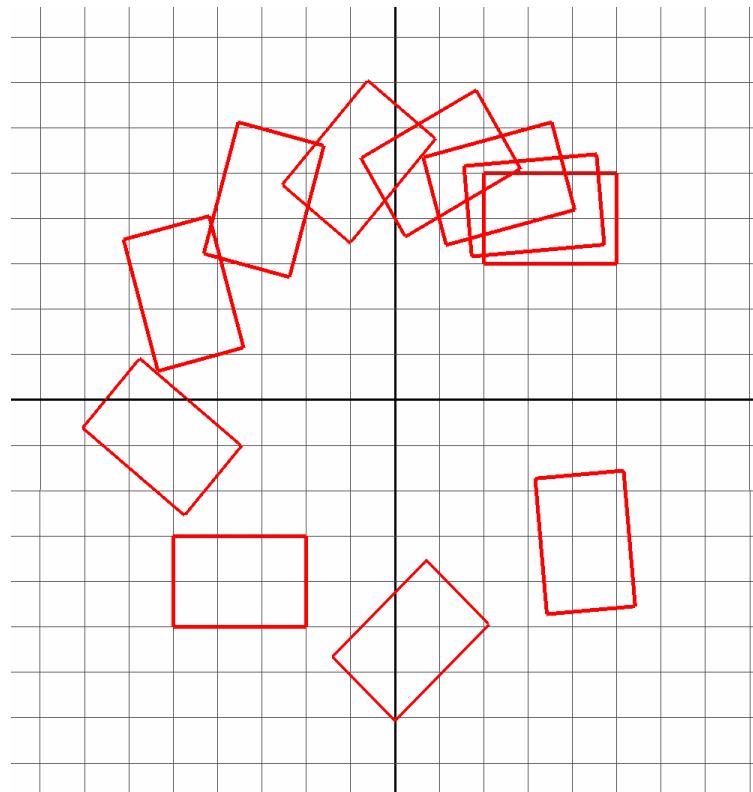
2d Scaling

$$p' = S(s)p$$
$$S(s) = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}$$

- s ... scale factors
- uniform scaling, or differential scaling
- 0 is fixed point



2d Rotation



2d Rotation

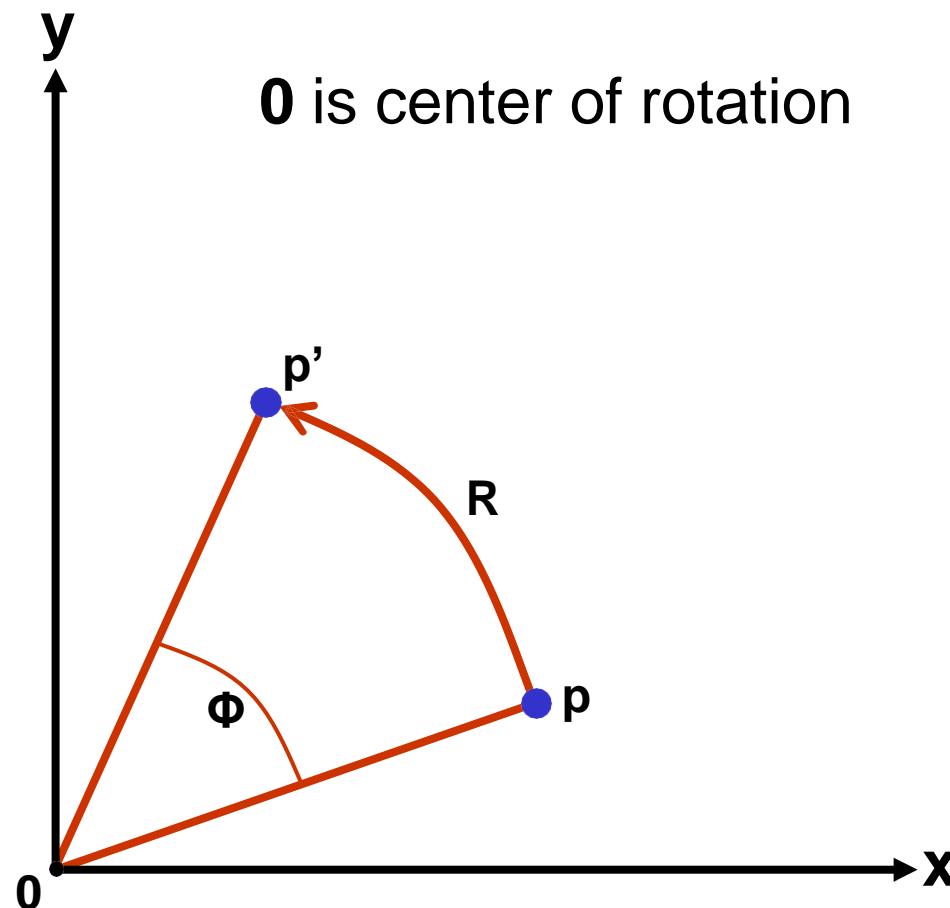
$$\mathbf{p}' = \mathbf{R}(\phi)\mathbf{p}$$

$$\mathbf{R}(\phi) = \begin{pmatrix} \cos \phi & -\sin \phi \\ \sin \phi & \cos \phi \end{pmatrix}$$

$$\mathbf{p}' = \begin{pmatrix} \cos \phi & -\sin \phi \\ \sin \phi & \cos \phi \end{pmatrix} \begin{pmatrix} p_x \\ p_y \end{pmatrix} =$$

$$= \begin{pmatrix} p_x \cos \phi - p_y \sin \phi \\ p_x \sin \phi + p_y \cos \phi \end{pmatrix}$$

ϕ ... rotation angle



Rotations

Any sequence of rotations is orthogonal,

therefore – the inverse of a rotation matrix
(or a sequence of rotations) is the same as
its transpose.

$$\mathbf{R}^{-1} = \mathbf{R}^T$$

Problem

$$\mathbf{R}(\phi) = \begin{pmatrix} \cos \phi & -\sin \phi \\ \sin \phi & \cos \phi \end{pmatrix} \quad \mathbf{S}(s) = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}$$

- But **no** 2x2 matrix for translation!
- Why is this important?

Composite Transforms

- We can combine multiple simple transforms to create a single composite transform
- by multiplying matrices

$$\mathbf{p}' = \mathbf{M}_1 \mathbf{p}$$

$$\mathbf{p}'' = \mathbf{M}_2 \mathbf{p}' \quad \rightarrow$$

$$\mathbf{p}''' = \mathbf{M}_3 \mathbf{p}''$$

$$\mathbf{M} = \mathbf{M}_3 \mathbf{M}_2 \mathbf{M}_1$$

$$\mathbf{p}''' = \mathbf{M} \mathbf{p}$$

Homogeneous Coordinates

Homogeneous Coordinates 1

homog.
point

homog.
vector

general

2-dim

$$\mathbf{p} = \begin{pmatrix} p_x \\ p_y \\ 1 \end{pmatrix}$$

$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ 0 \end{pmatrix}$$

$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ v_w \end{pmatrix}$$

3-dim

$$\mathbf{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix}$$

$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ v_w \end{pmatrix}$$

Homogeneous Coordinates 2

$$\begin{pmatrix} 3 \\ 2 \\ 5 \\ 2 \end{pmatrix}$$

Homogenization



$$\begin{pmatrix} 3/2 \\ 2/2 \\ 5/2 \\ 2/2 \end{pmatrix} = \begin{pmatrix} 1.5 \\ 1 \\ 2.5 \\ 1 \end{pmatrix}$$

is performed if $w \neq 1$

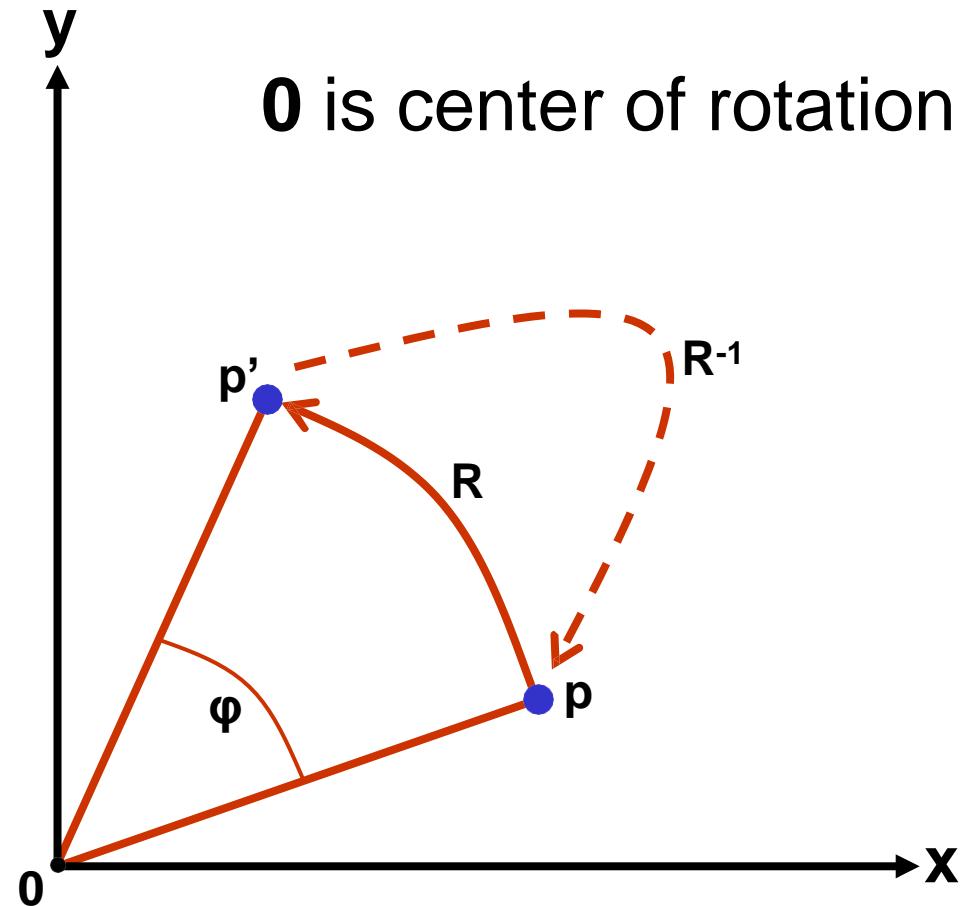
e.g., used for perspective projection
matrices, we will investigate this later

Homogeneous 2d Rotation

$$\mathbf{p}' = \mathbf{R}(\phi)\mathbf{p}$$

$$\mathbf{R}(\phi) = \begin{pmatrix} \cos \phi & -\sin \phi & 0 \\ \sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}(\phi)^{-1} = \begin{pmatrix} \cos \phi & \sin \phi & 0 \\ -\sin \phi & \cos \phi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

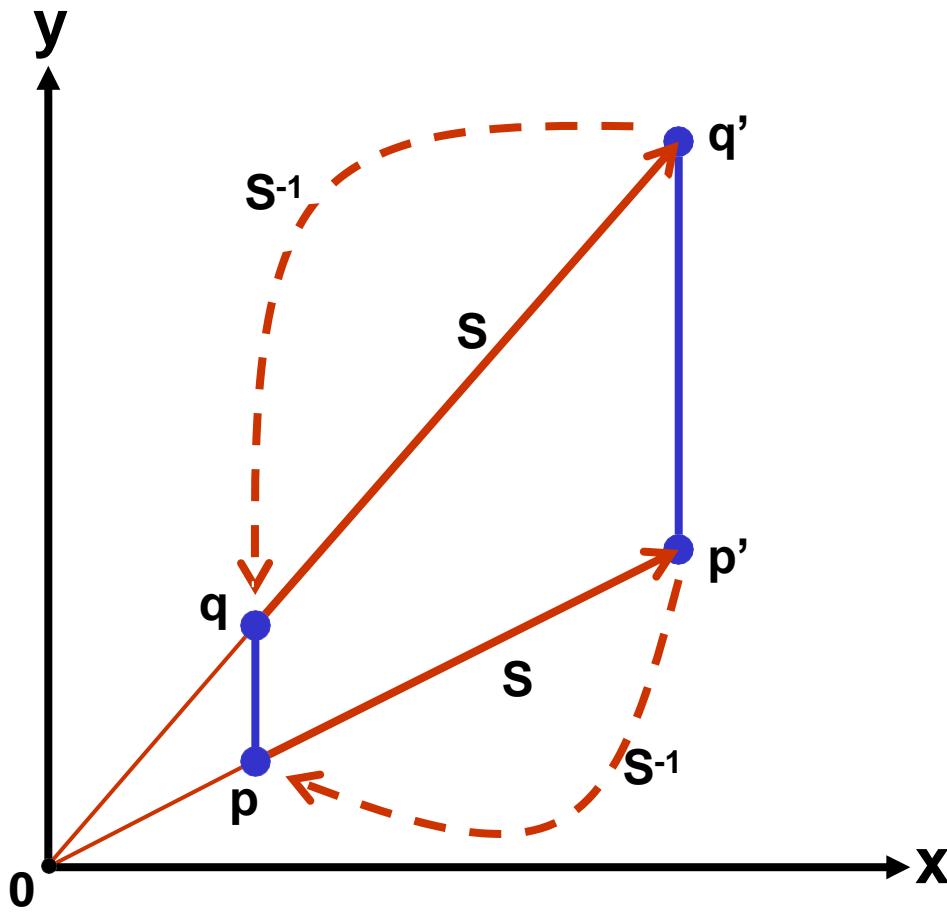


Homogeneous 2d Scaling

$$\mathbf{p}' = \mathbf{S}(s)\mathbf{p}$$

$$\mathbf{S}(s) = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{S}(s)^{-1} = \begin{pmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$



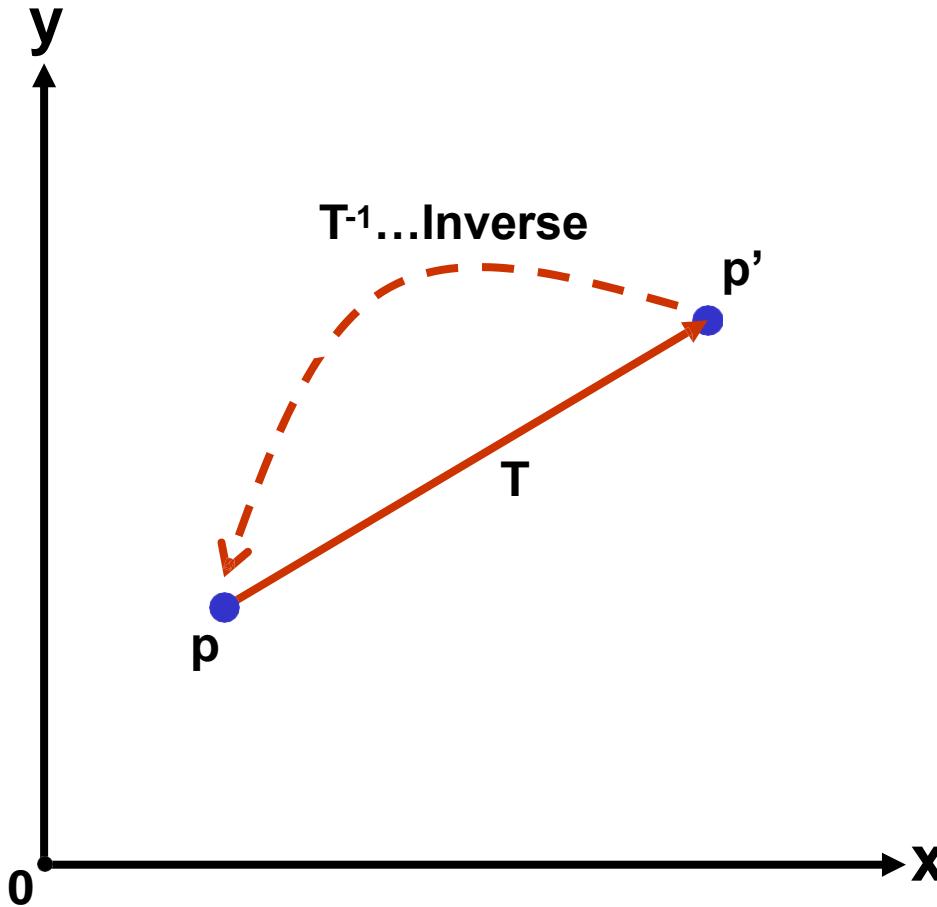
**And now ...
a matrix for translation!**

Homogeneous 2d Translation

$$\mathbf{p}' = \mathbf{T}(\mathbf{t})\mathbf{p}$$

$$\mathbf{T}(\mathbf{t}) = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{T}(\mathbf{t})^{-1} = \begin{pmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{pmatrix}$$



Example

Example 1: point $p(2,3)$, translation $t(5,-7)$

$$p' = \begin{pmatrix} 1 & 0 & 5 \\ 0 & 1 & -7 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} 1*2 + 0*3 + 5*1 \\ 0*2 + 1*3 - 7*1 \\ 0*2 + 0*3 + 1*1 \end{pmatrix} = \begin{pmatrix} 7 \\ -4 \\ 1 \end{pmatrix}$$

Example 2: vector $v(3,2)$, translation $t(7,11)$

$$v' = \begin{pmatrix} 1 & 0 & 7 \\ 0 & 1 & 11 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 3 \\ 2 \\ 0 \end{pmatrix} = \begin{pmatrix} 1*3 + 0*2 + 7*0 \\ 0*3 + 1*2 - 11*0 \\ 0*3 + 0*2 + 1*0 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 0 \end{pmatrix}$$

3-dim Transforms

3d Transforms

- 3d works like 2d (should be no surprise)
- homogeneous notation of 3d-space
 - 4d vectors - 4-th element is 1 for points, and 0 for directions
 - 4x4 matrices

3d Translation

$$T(t) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$T(t)^{-1} = \begin{pmatrix} 1 & 0 & 0 & -t_x \\ 0 & 1 & 0 & -t_y \\ 0 & 0 & 1 & -t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3d Scaling

$$\mathbf{S}(\mathbf{s}) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{S}(\mathbf{s})^{-1} = \begin{pmatrix} \frac{1}{s_x} & 0 & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 & 0 \\ 0 & 0 & \frac{1}{s_z} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3d Rotation

**Rotation
around
x-axis**

$$\mathbf{R}_x(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \phi & -\sin \phi & 0 \\ 0 & \sin \phi & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

**Rotation
around
y-axis**

$$\mathbf{R}_y(\phi) = \begin{pmatrix} \cos \phi & 0 & \sin \phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \phi & 0 & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

**Rotation
around
z-axis**

$$\mathbf{R}_z(\phi) = \begin{pmatrix} \cos \phi & -\sin \phi & 0 & 0 \\ \sin \phi & \cos \phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_x^{-1}(\phi) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \phi & \sin \phi & 0 \\ 0 & -\sin \phi & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_y^{-1}(\phi) = \begin{pmatrix} \cos \phi & 0 & -\sin \phi & 0 \\ 0 & 1 & 0 & 0 \\ \sin \phi & 0 & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_z^{-1}(\phi) = \begin{pmatrix} \cos \phi & \sin \phi & 0 & 0 \\ -\sin \phi & \cos \phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Euler Transform 1

- The Euler Transform E is an intuitive way to construct a matrix to orient yourself (i.e., the camera) or any other entity in a certain direction
- Default view direction has to be established
 - most often it lies along negative z-axis,
 - with y-axis being „up“
 - see next slide

Euler Transform 2

pitch

rotation around x-axis

yaw (or head)

rotation around y-axis

roll

rotation around z-axis

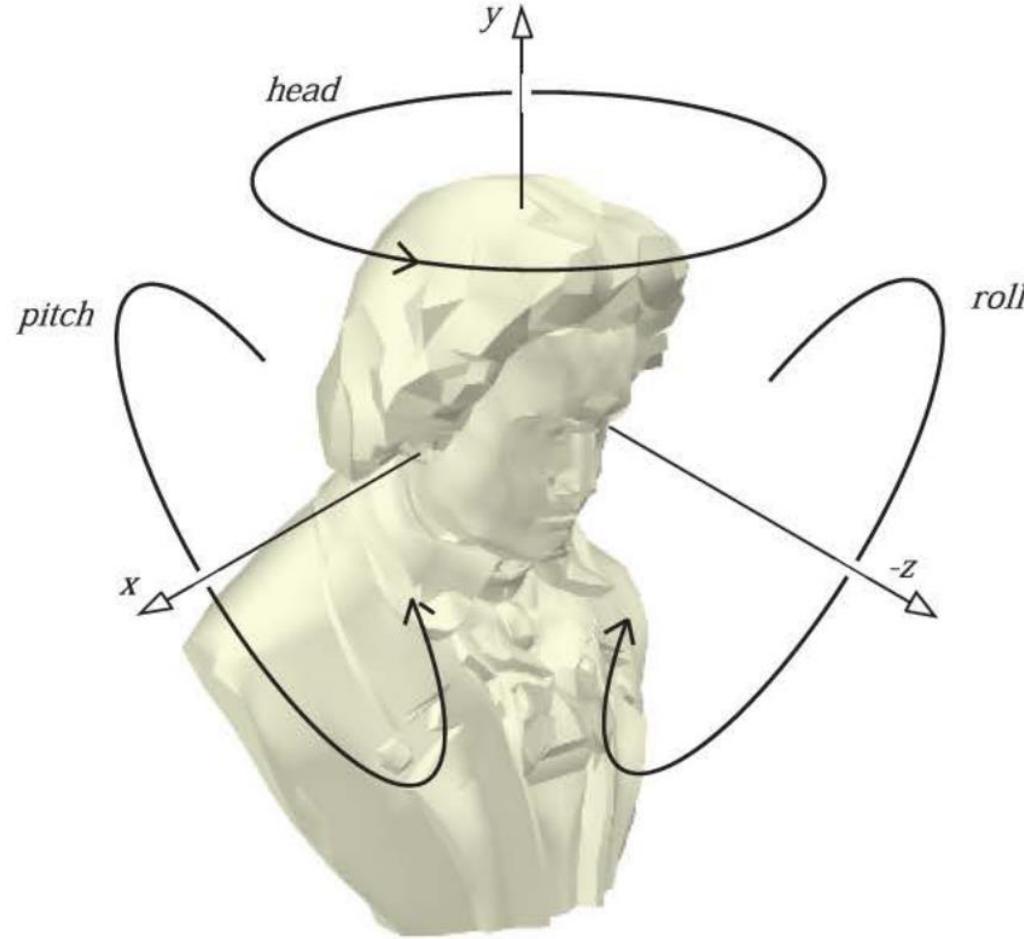


Image from [Real-Time Rendering – 4.2 Euler Transform, page 71]

Euler Transform 3

pitch

rotation around x-axis

yaw (or head)

rotation around y-axis

roll

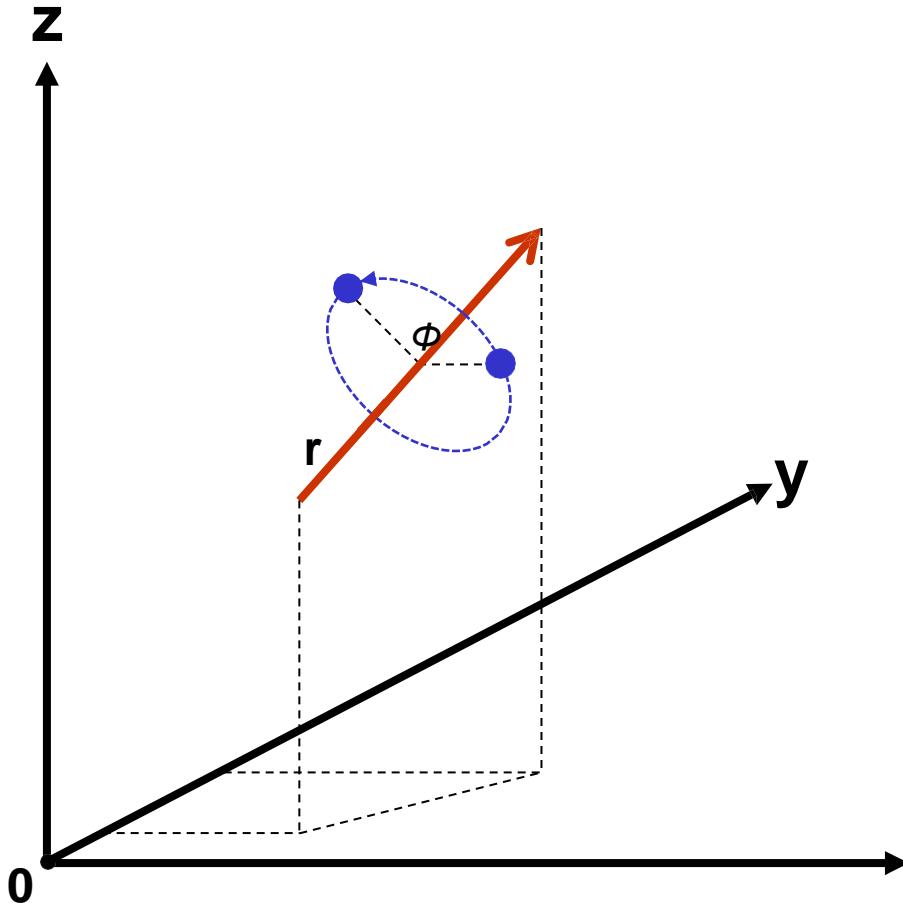
rotation around z-axis

$$\mathbf{E}(p, y, r) = \mathbf{R}_z(r) \mathbf{R}_x(p) \mathbf{R}_y(y)$$

Since \mathbf{E} is a sequence of rotations,
it clearly is orthogonal, therefore

$$\begin{aligned}\mathbf{E}^{-1} &= \mathbf{E}^T = \\ &= (\mathbf{R}_z \mathbf{R}_x \mathbf{R}_y)^T = \mathbf{R}_y^T \mathbf{R}_x^T \mathbf{R}_z^T\end{aligned}$$

Rotation Around Arbitrary Axis



normalized axis r ,
i.e. $\|\mathbf{r}\| = 1$

rotation angle ϕ

3d Rotation Around Arvitraray Axis

Rotation around normalized axis \mathbf{r} by Φ radians

$$\mathbf{R}(\mathbf{r}, \phi) =$$

$$\begin{pmatrix} \cos \phi + (1 - \cos \phi)r_x^2 & (1 - \cos \phi)r_x r_y - r_z \sin \phi & (1 - \cos \phi)r_x r_z + r_y \sin \phi & 0 \\ (1 - \cos \phi)r_x r_y + r_z \sin \phi & \cos \phi + (1 - \cos \phi)r_y^2 & (1 - \cos \phi)r_y r_z - r_x \sin \phi & 0 \\ (1 - \cos \phi)r_x r_z - r_y \sin \phi & (1 - \cos \phi)r_y r_z + r_x \sin \phi & \cos \phi + (1 - \cos \phi)r_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

[Goldman, Ronald, "Matrices and Transformations," in Andrew S. Glassner, Ed., *Graphics Gems*, Academic Press, pp. 472-475, 1990.]

Quaternions

- another tool to represent rotations or orientations
- quaternions are a generalization of complex numbers
- better suited for calculations than matrices (e.g. smooth interpolation, ...)
- see Chapter 4.3 Quaternions for details [Real-Time Rendering]

Computation of Inverses



University of
Applied Sciences

Computation of Inverses 1

- If the matrix is a simple transform, or a sequence of simple transforms
- then “invert” parameters and matrix order
- Example:

$$\mathbf{M} = \mathbf{T}(t)\mathbf{R}(\phi)$$

$$\mathbf{M}^{-1} = \mathbf{R}(-\phi)\mathbf{T}(-t)$$

Computation of Inverses 2

- If the matrix is known to be orthogonal, then the transpose is the inverse
- Any sequence of rotations is orthogonal.

$$\mathbf{M}_{orthogonal}^{-1} = \mathbf{M}_{orthogonal}^T$$

Computation of Inverses 3

- If nothing in particular is known about M, then
 - the adjoint method
 - Cramer's rule,
 - LU decomposition, or
 - Gaussian elimination
- could be used to compute the inverse
- see [Real-Time Rendering] Section 4.1.8 for details
- this is rarely needed in „standard“ CG

Example

Example

Task

- Create a matrix, which performs the following task:
 1. uniform scale with factor 2 and center $c=(3,1,2)$
 2. translate 5 units along the x-axis

Solution

1. shift center of scale to origin
2. perform scale
3. shift back (inverse of step 1)
4. shift 5 units in x direction

Example

shift center of scale to origin

center $\mathbf{c}=(3,1,2)$

$$M_1 = \begin{pmatrix} 1 & 0 & 0 & -3 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Example

perform scale

factor is 2

$$M_2 = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Example

shift back (inverse from step 1)

center $\mathbf{c}=(3,1,2)$

$$M_3 = \begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Example

shift 5 units in x direction

translation vector
is (5,0,0)

$$M_4 = \begin{pmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Example

Now create composite matrix \mathbf{M}

$$\begin{aligned}\mathbf{M}_4 \mathbf{M}_3 \mathbf{M}_2 \mathbf{M}_1 &= \\ &= \begin{pmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -3 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \\ &= \begin{pmatrix} 2 & 0 & 0 & 2 \\ 0 & 2 & 0 & -1 \\ 0 & 0 & 2 & -2 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \mathbf{M}\end{aligned}$$

Example

Test with $p=(3,4,5)$

1. uniform scale with factor 2 and center $c=(3,1,2)$

$$c+2(p-c)=(3,7,8)$$

2. translate 5 units along the x-axis

$$(3,7,8)+(5,0,0)=(8,7,8)$$

$$\mathbf{p}' = \mathbf{Mp} = \begin{pmatrix} 2 & 0 & 0 & 2 \\ 0 & 2 & 0 & -1 \\ 0 & 0 & 2 & -2 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 3 \\ 4 \\ 5 \\ 1 \end{pmatrix} = \begin{pmatrix} 8 \\ 7 \\ 8 \\ 1 \end{pmatrix}$$

Special Transforms

Rigid-Body Transform 1

- **only orientation and location change**
- **preserves lengths and angles**
- **any rigid-body matrix X can be written as the concatenation of a translation matrix T and a rotation matrix R**

$$X = T(t)R = \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rigid-Body Transform 2

- Inverse

$$\mathbf{X}^{-1} = (\mathbf{T}(t)\mathbf{R})^{-1} =$$

$$= \mathbf{R}^{-1}\mathbf{T}(t)^{-1} =$$

$$= \mathbf{R}^T\mathbf{T}(-t)$$

Normal Vector Transform 1

- Normal vectors must be transformed by the transpose of the inverse of the matrix used to transform geometry

$$\mathbf{N} = (\mathbf{M}^{-1})^T$$

- **M ... matrix used to transform geometry**
- **N ... matrix used to transform normal vectors**
- **see next slide for illustration**

Normal Vector Transform 2

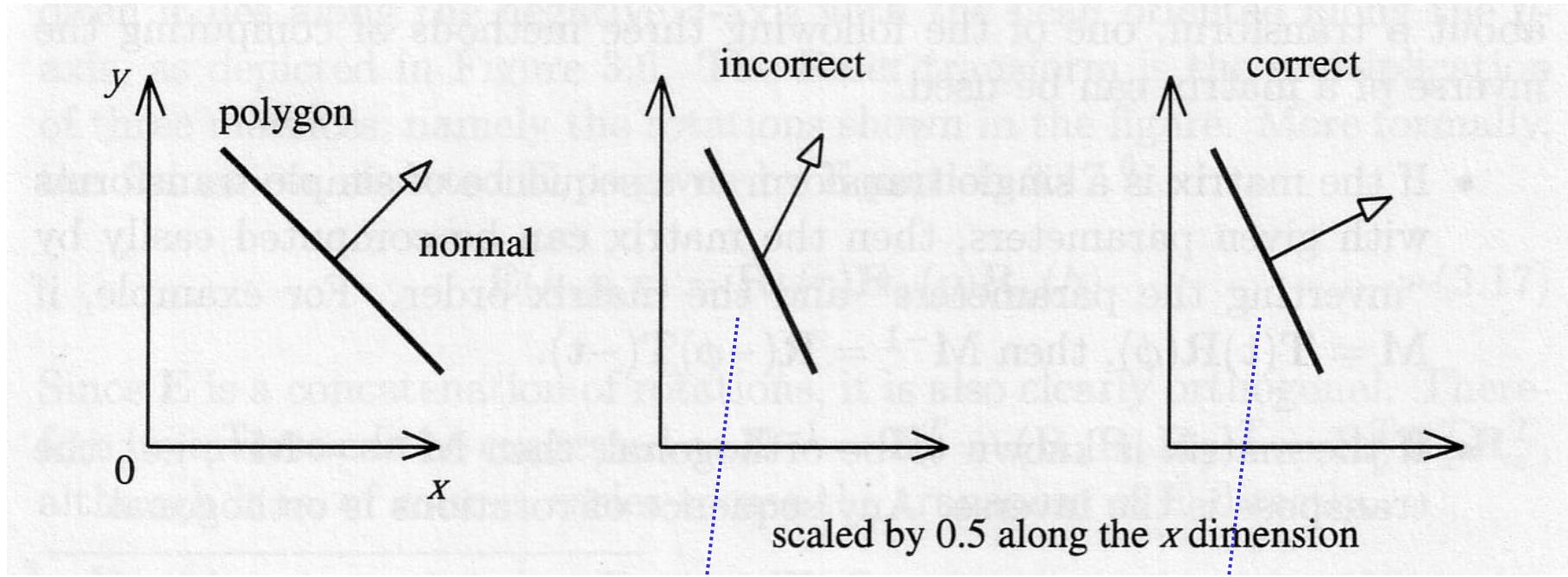


Image from [Real-Time Rendering – 4.1.7 Normal Transform]

Normal Vector Transform 3

- if M consists only of rotations and translations, then $(M^{-1})^T = M$
and nothing has to be done!
- if M contains a uniform scale s , then the transformed normal will be scaled by $1/s$
(e.g., `GL_RESCALE_NORMAL` may be used to restore unit length)
- if M contains a non-uniform scale, then the transposed inverse of M has to be created



Geometry

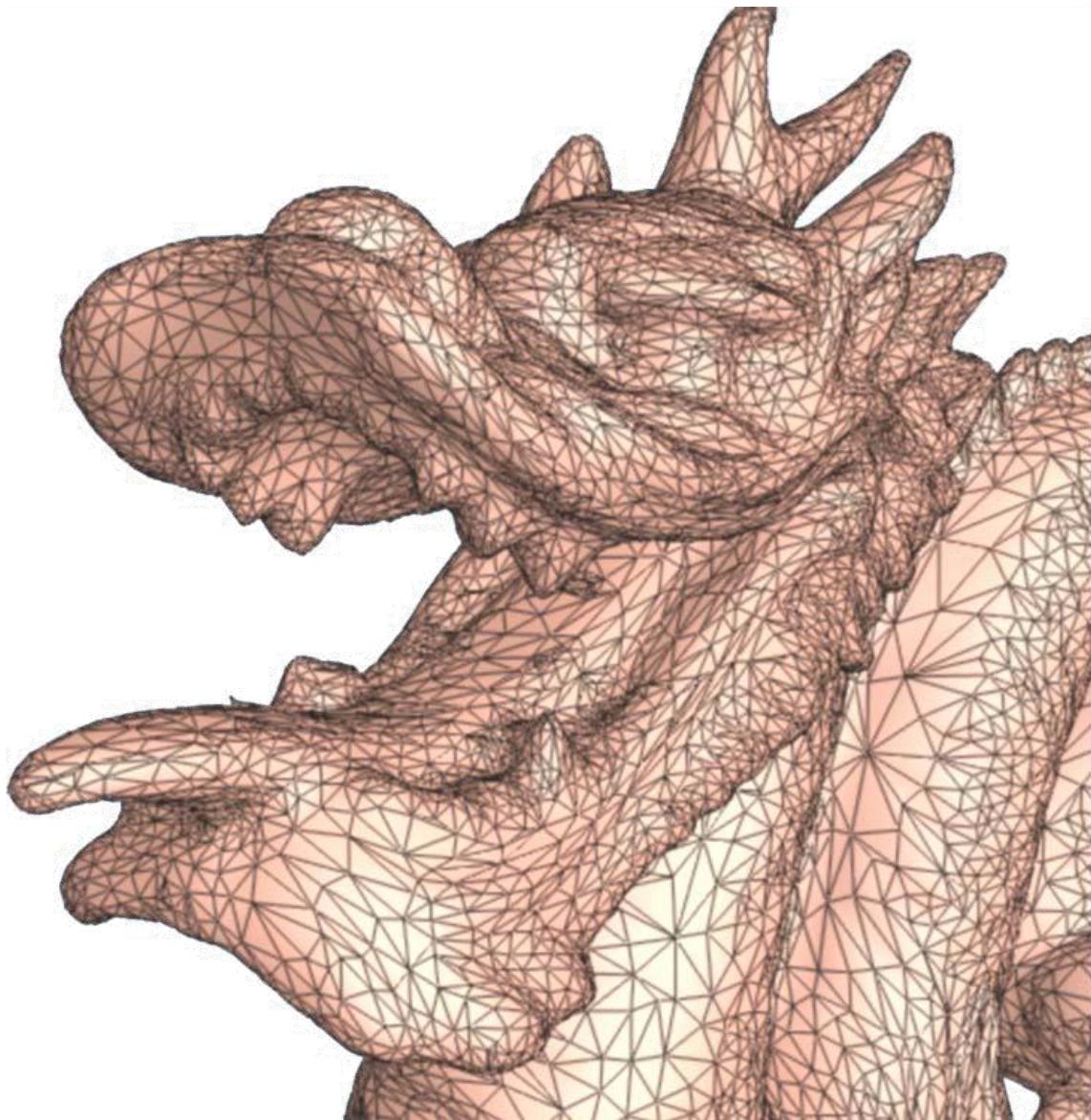
Geometry - Overview

Geometry 1

- Data structures to represent 3D surfaces
- Vertex and its properties
- Drawing primitives
- Meshes and rendering-efficient structures

Geometry 2

- Spatial data structures
- Culling techniques



Motivation

Detailed Models



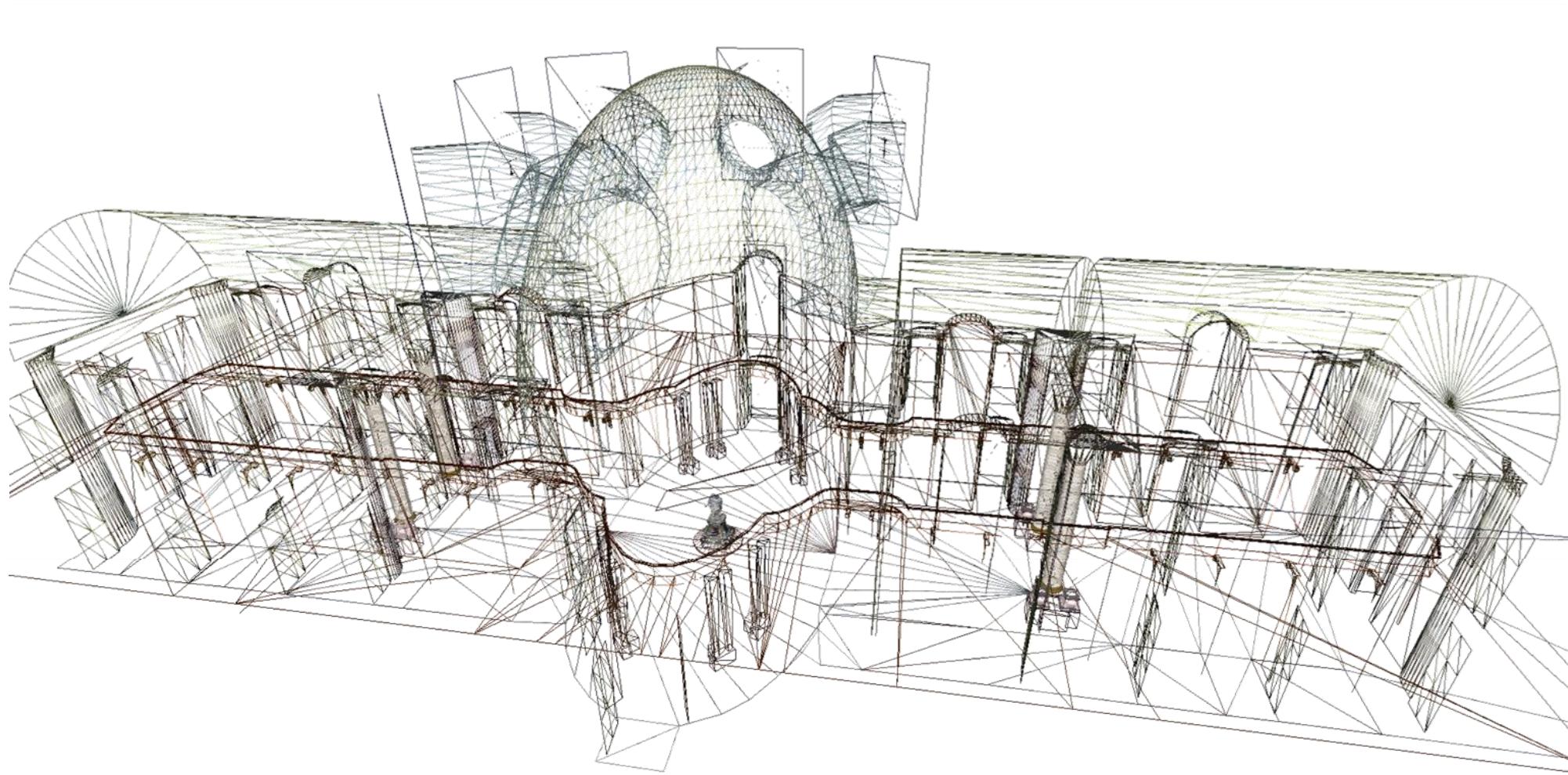
Photography [VRVis, 2006]



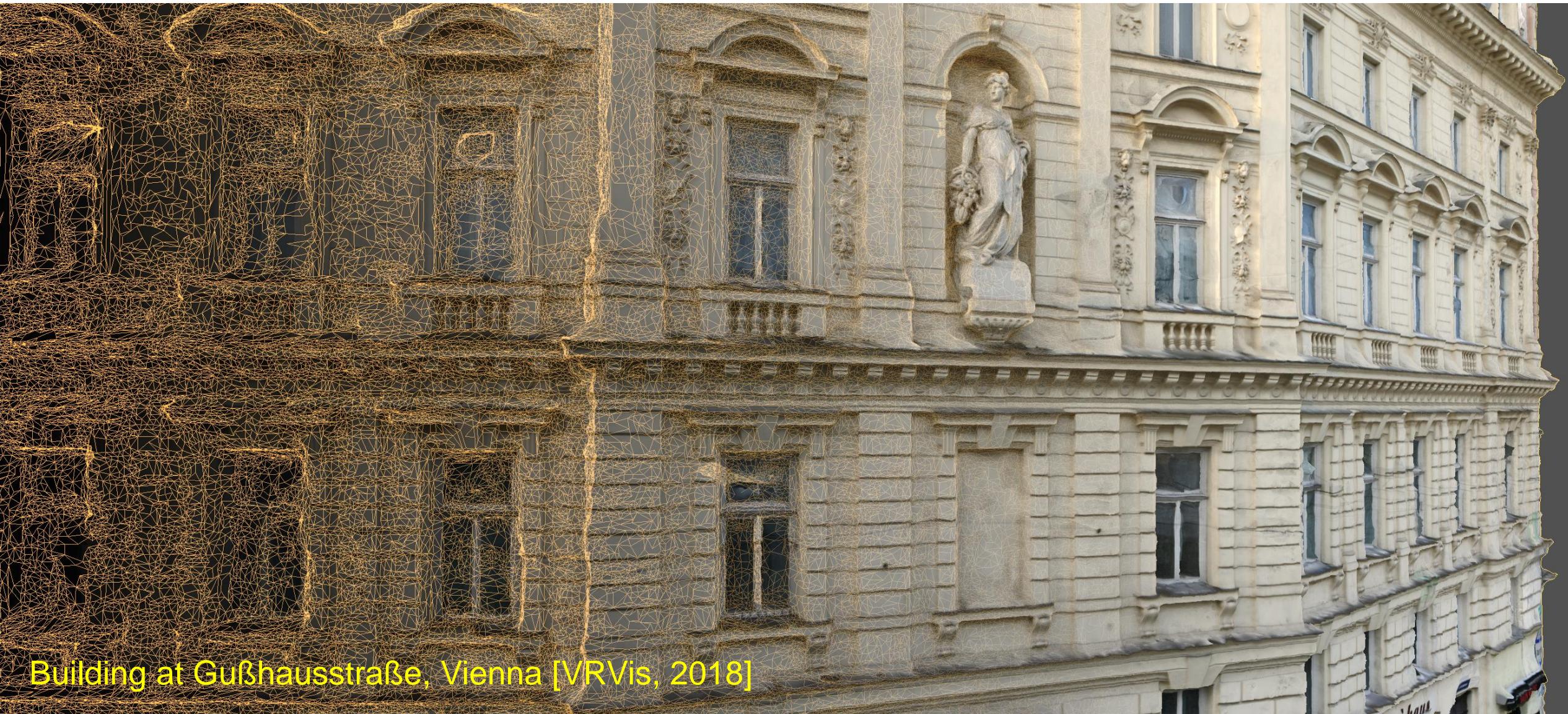
Reconstructed Geometry [VRVis, 2006]

Architecture

Prunksaal Österreichische Nationalbibliothek [VRVis, 2006]



Architecture



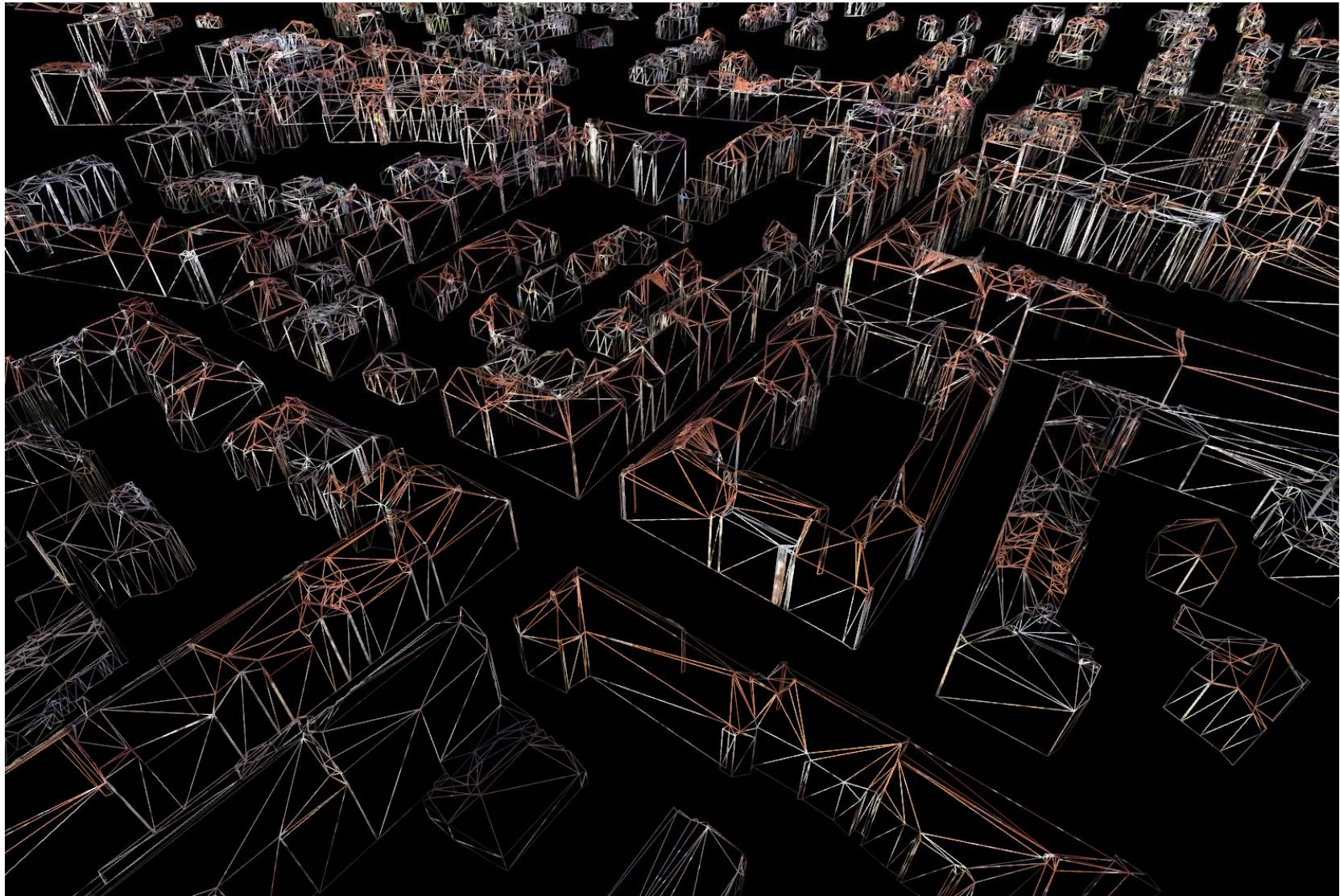
Building at Gußhausstraße, Vienna [VRVis, 2018]

City Models

Graz

Reconstruction from
aerial imagery

[VRVis, 2006]





City Models

Digital twins for decision support

- Assess impact of planned infrastructure projects
- Inform the public



GEARViewer [VRVis, 2000 - 2020]



City Models

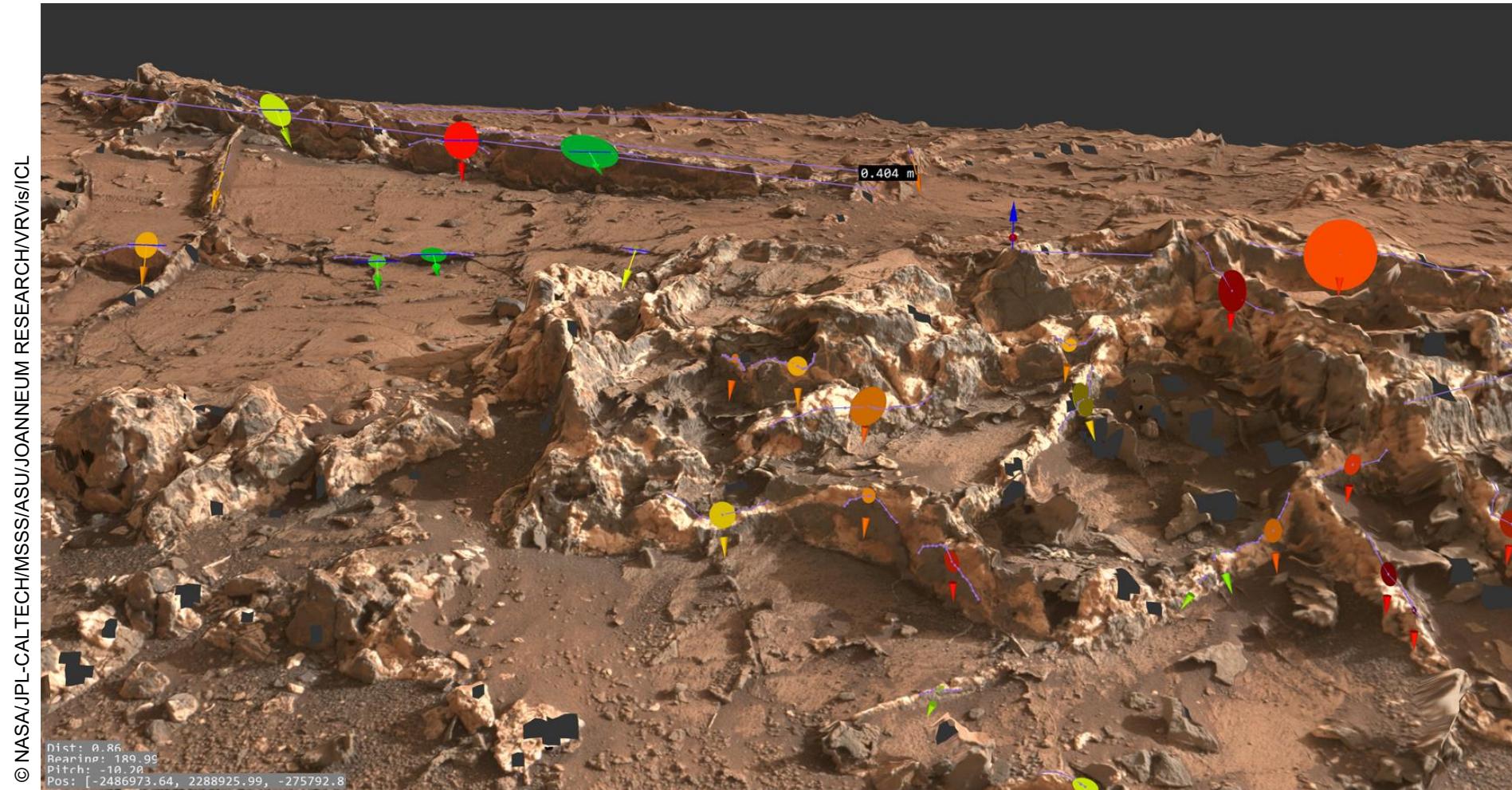
Infrastructure development in Aspern, Vienna [VRVis, 2018]



Terrain Models

Virtual exploration and geologic interpretation of Martian surface reconstructions

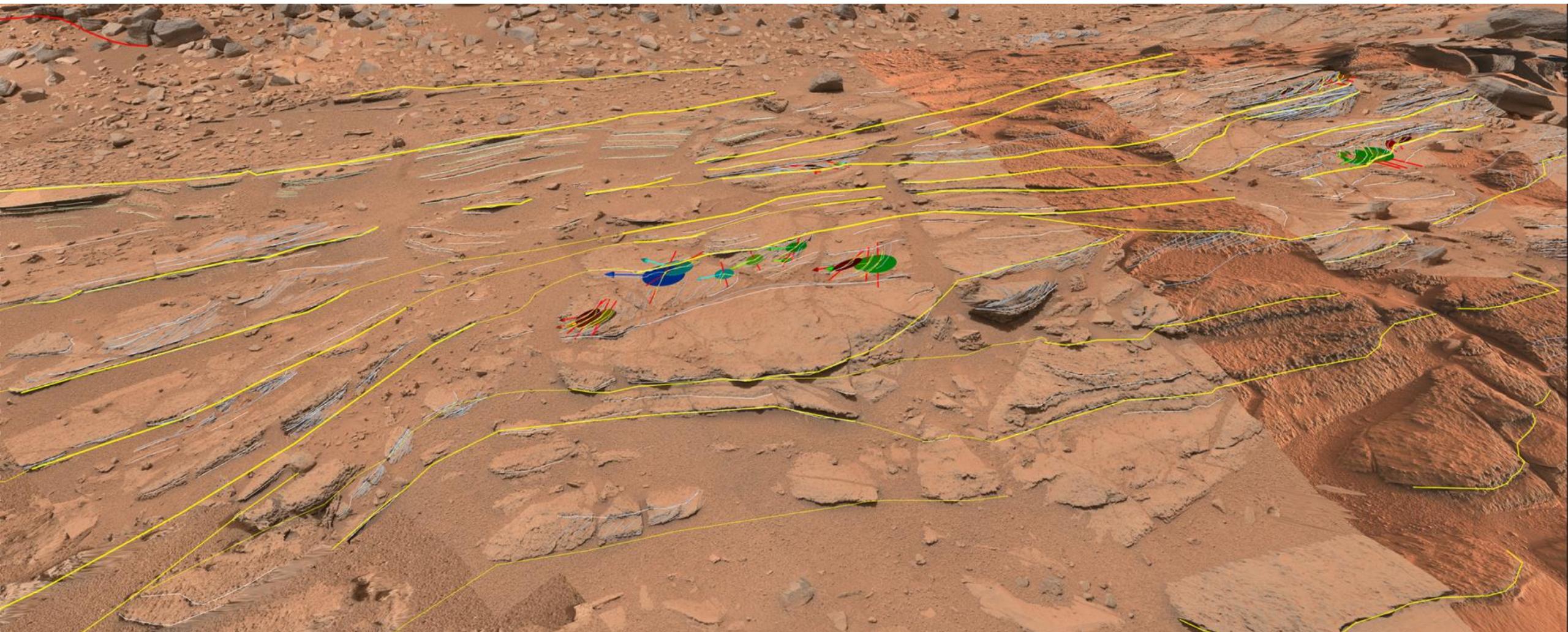
Garden City,
Gale Crater,
visited by
Curiosity rover



Terrain Models

Shaler, Gale Crater visited by Curiosity rover

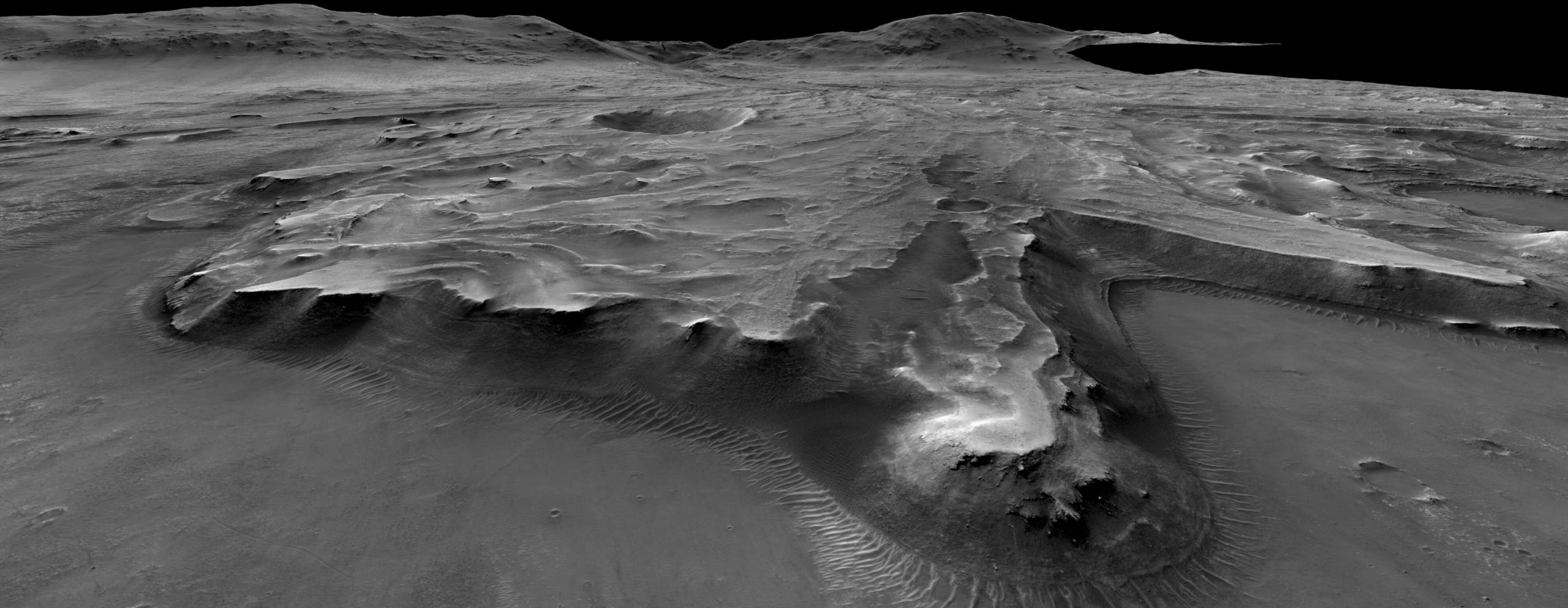
© NASA/JPL-CALTECH/MSSS/ASU/JOANNEUM RESEARCH/VRVis/ICL



Terrain Models

Jezero Crater, landing site of Perseverance rover

© NASA/JPL-CALTECH/MSSS/ASU/JOANNEUM RESEARCH/VRVis/



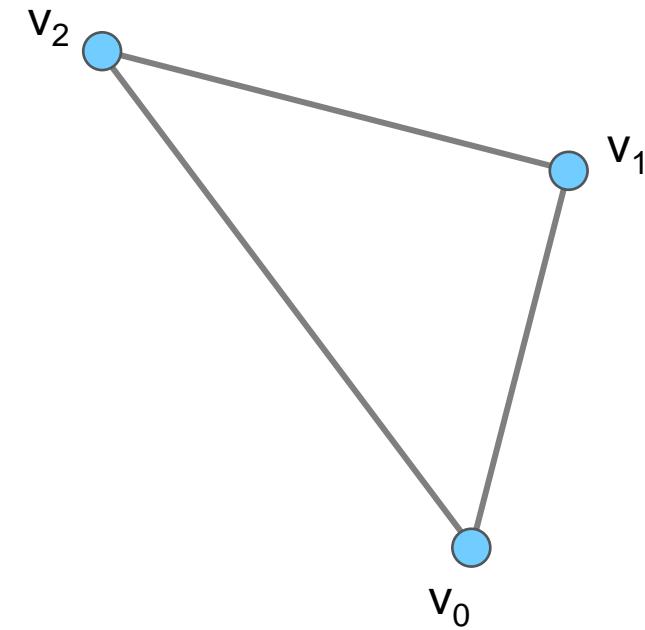
Vertices & Per-Vertex Properties

Vertex

Vertex (*pl. vertices*)

- A corner of a polygon or polyhedron (mesh)
- Per-vertex properties include

- Position
- Normal
- Colour
- texcoord0, texcoord1, ...
- Tangent
- Binormal
- ...



OpenGL Vertex Position

`glVertex[234][sifd]`

`glVertex[234][sifd]v`

- **Dimension:** 2, 3 or 4 dim
- **Data type:** short, int, float, double
- **Pointer to array (vector):** v

Example

2-dim vertex, float coordinates

```
glVertex2f(1.5f, 3.7f);
```

3-dim vertex from double array

```
double a[3] = {1.0, 2.0, 3.0};  
glVertex3dv(a);
```

Vertex Normal

`glNormal3[bsifd]`

`glNormal3[bsifd]v`

- **Dimension:** always 3 dim
- **Data type:** byte, short, int, float, double
- **Pointer to array (vector):** v

Example

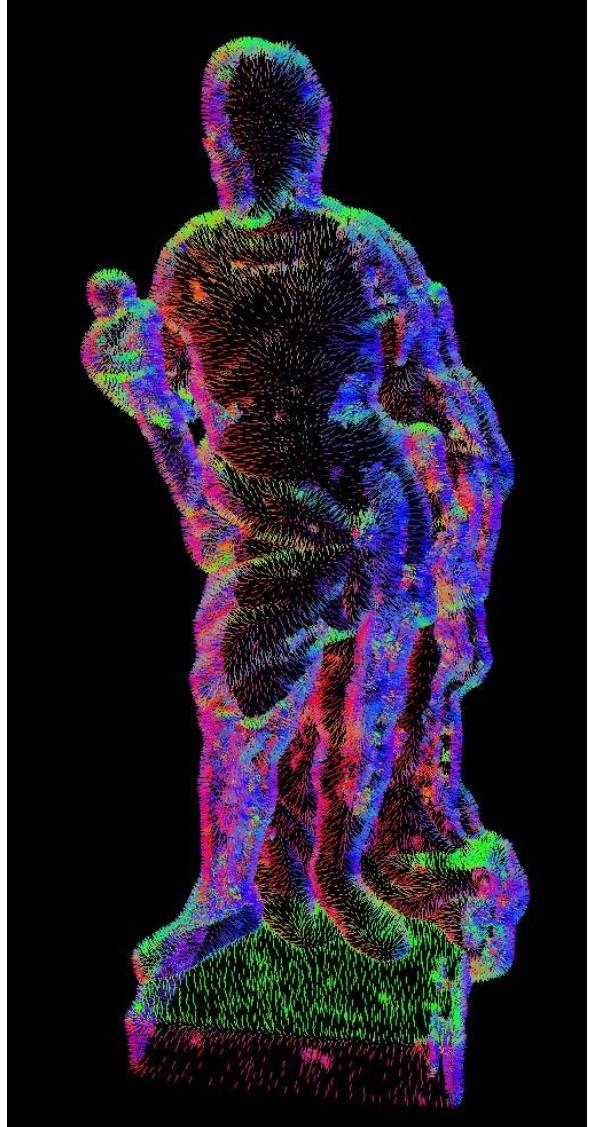
Double normal, float coordinates

```
glNormal3d(0.0, 0.0, 1.0);
```

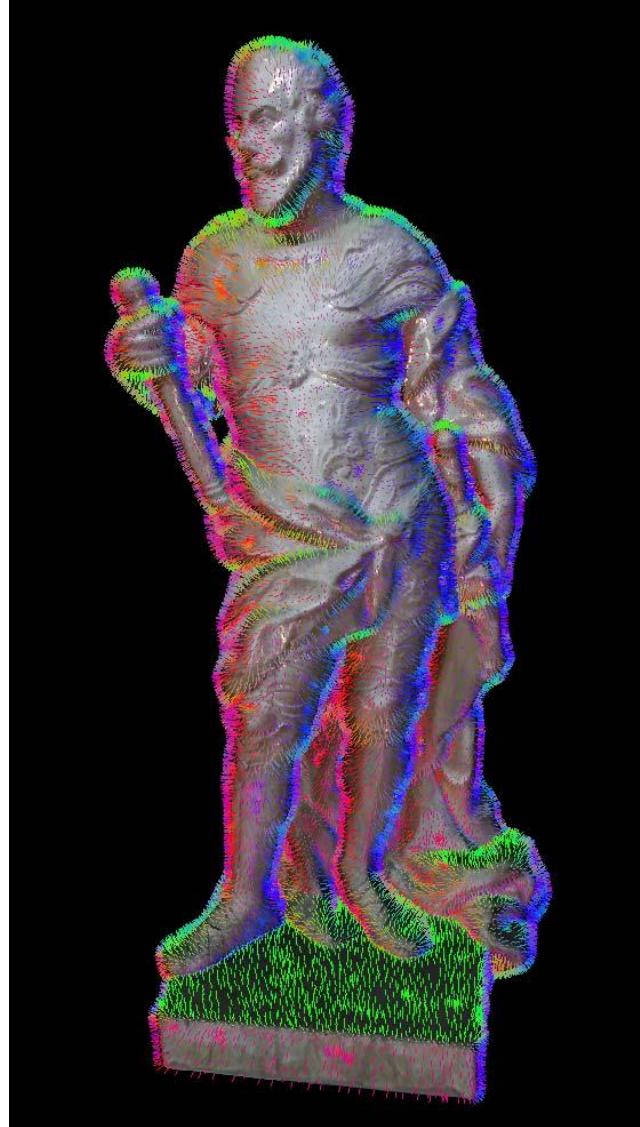
Normal from float array

```
float n[3] = {1.0f, 0.0f, 0.0f};  
glNormal3fv(n);
```

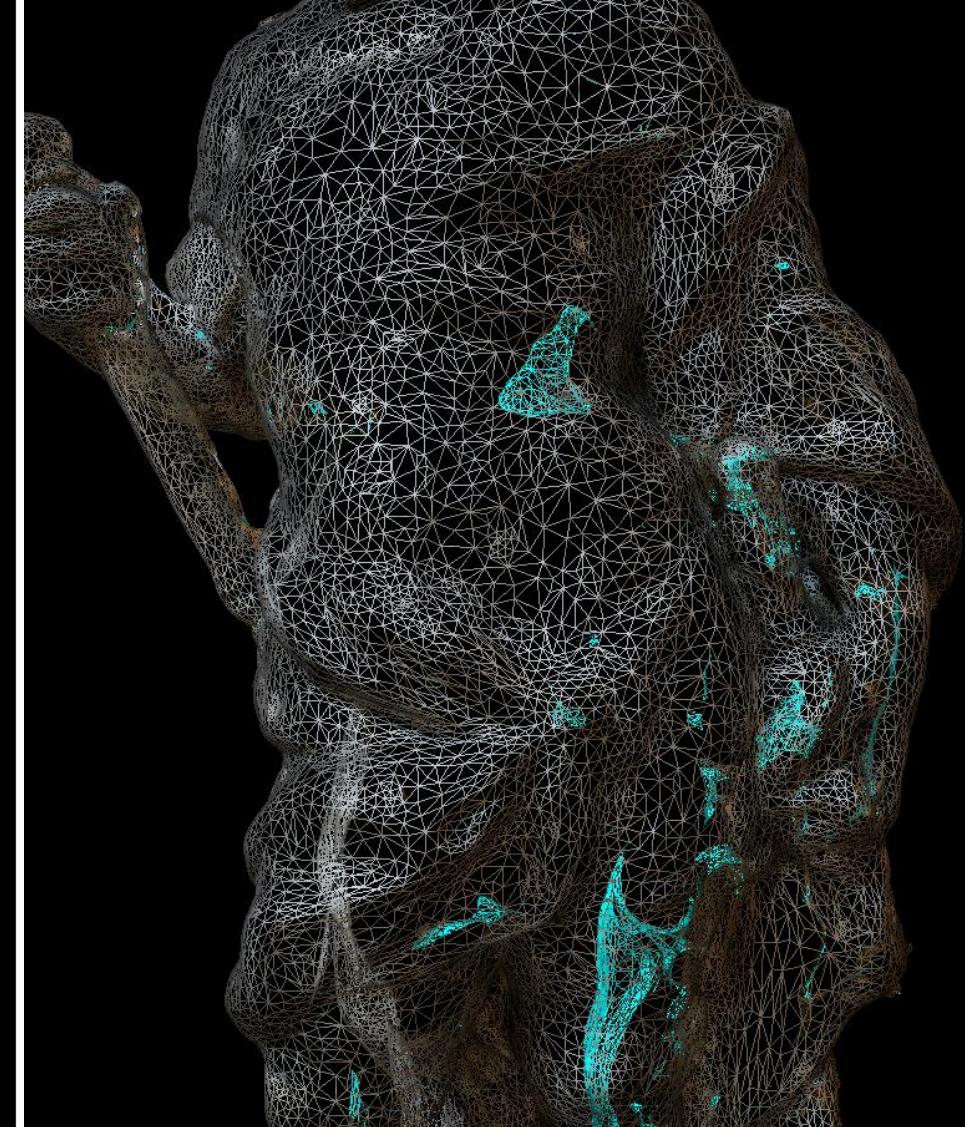
Demo



Colour-coded normal



Normals + solid



Wireframe

Vertex Colour

```
glColor[34][bsifd]  
glColor[34][bsifd]v
```

- **Dimension:** 3 (rgb), or 4 (rgba) dim
- **Data type:** byte, short, int, float, double
- **Pointer to array (vector):** v

Example

rgb colour, byte

```
glColor3b(255, 255, 0); /* brightest yellow */
```

rgba colour from float array

```
float c[4] = {0.0f, 1.0f, 0.0f, 1.0f};  
glColor4fv(c); /* brightest green */
```

alpha defaults to 1.0 if not specified

Texture Coordinates

`glTexCoord[1234][sifd]`

`glTexCoord[1234][sifd]v`

- **Dimension:** 1, 2, 3, or 4 dim
- **Data type:** short, int, float, double
- **Pointer to array (vector):** v

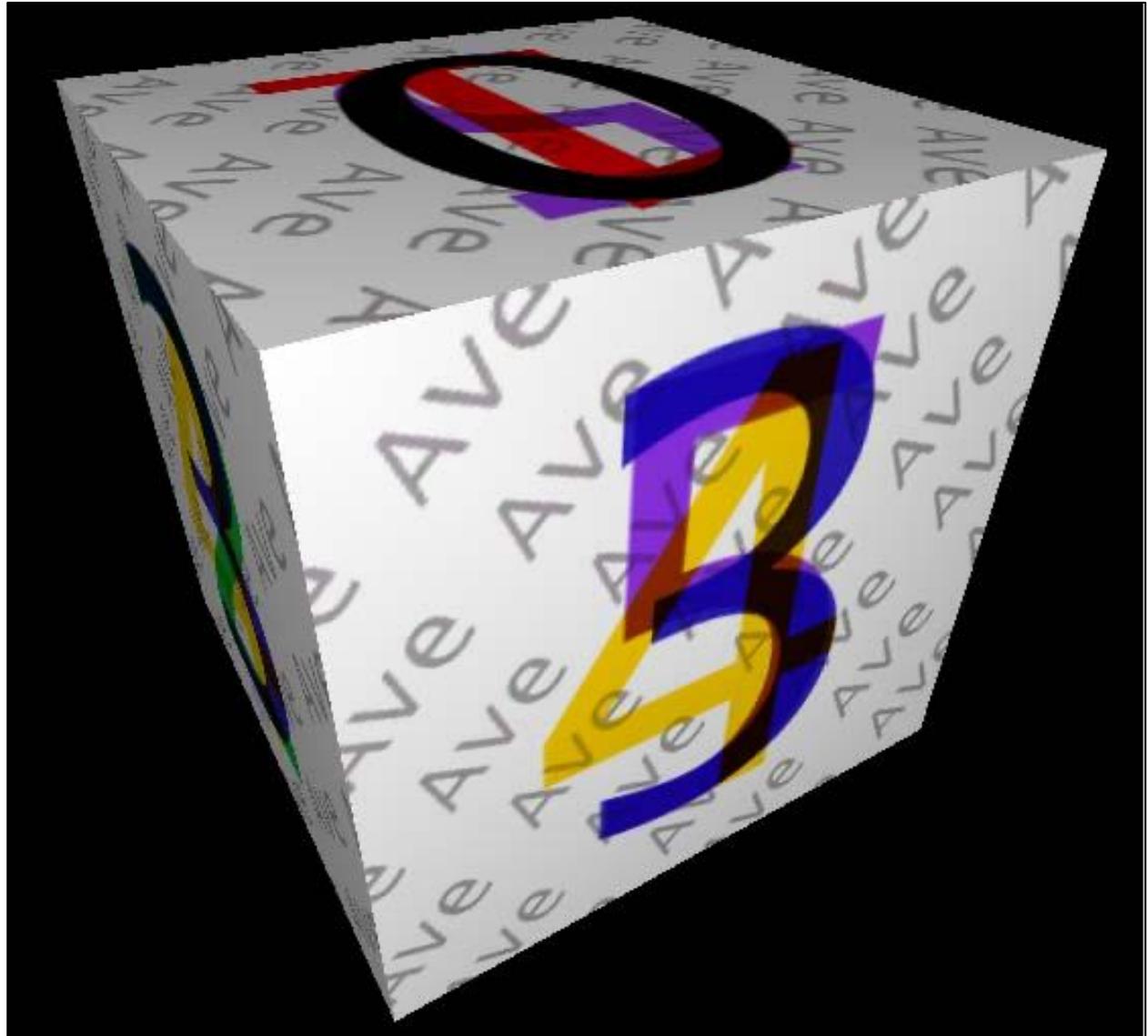
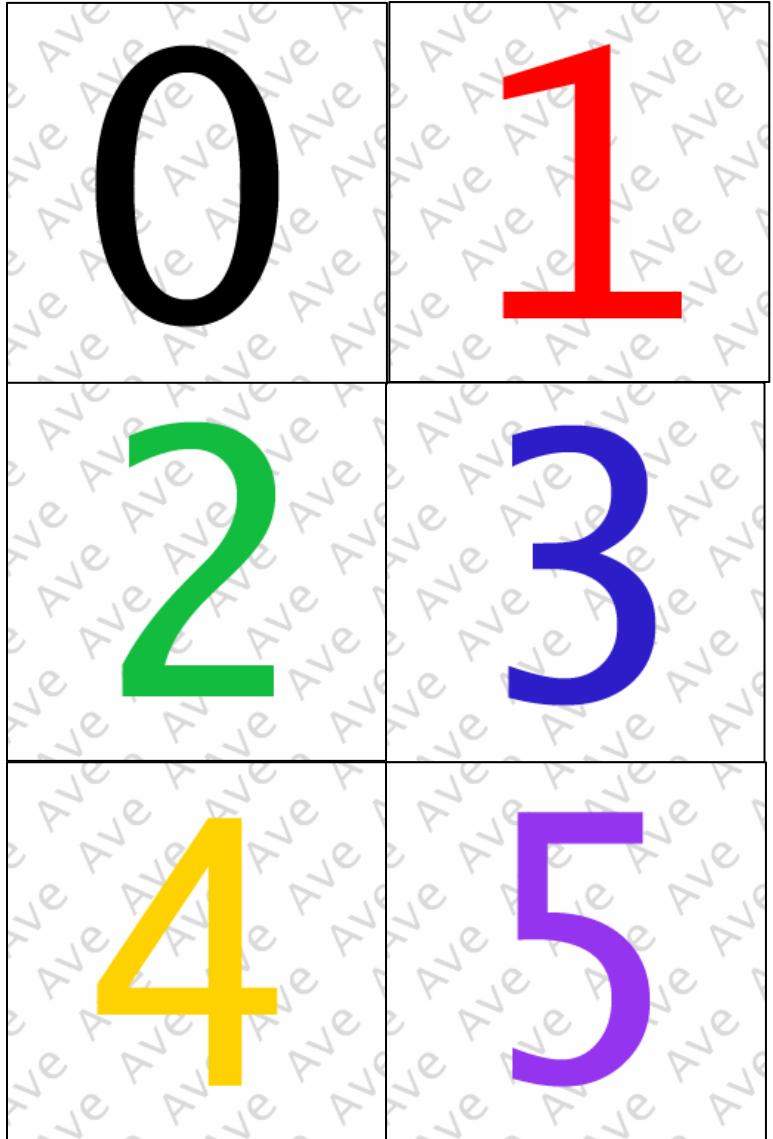
Multi-Texturing

`glMultiTexCoord[1234][sifd](texture, coords)`

`glMultiTexCoord[1234][sifd]v(texture, coords)`

- **Dimension:** 1, 2, 3, or 4 dim
- **Data type:** short, int, float, double
- **Pointer to array (vector):** v
- **Texture:** GL_TEXTURE0 - GL_TEXTUREn
(where n = GL_MAX_TEXTURE_COORDS)

Multi-Texturing



Rendering Efficiency

Specifying vertices one-by-one is **inefficient**

Solutions

- **Display lists**
- **Vertex arrays**
- **Vertex buffer objects**

Rendering Efficiency

Vertex arrays

- (see OpenGL 2.0 Spec, ‘2.8 Vertex Arrays’) for details
- Functions to set pointers to arrays of positions, normals, colours, etc.
- `glDrawArrays(mode, first, count)` to draw primitives

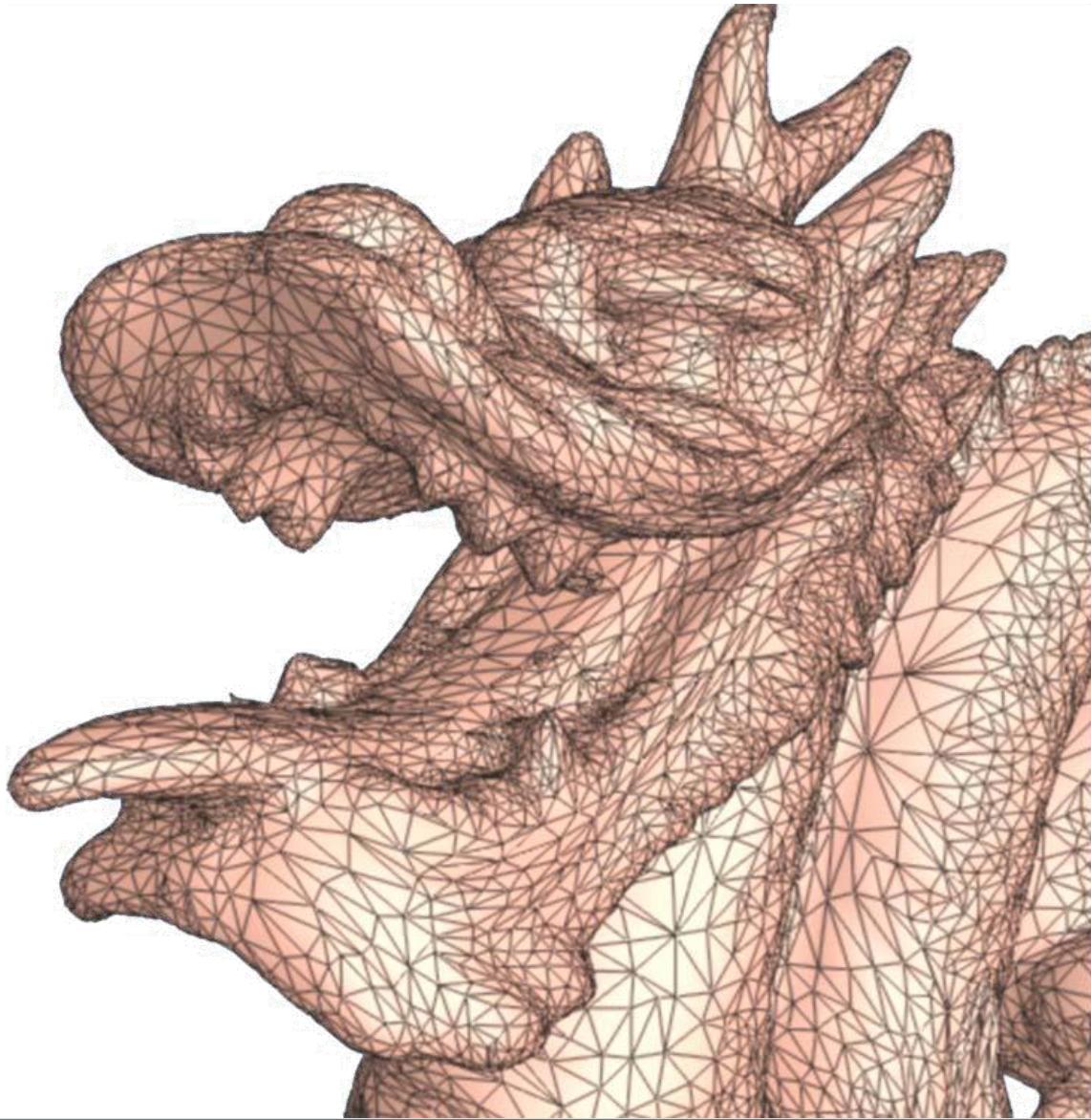
Meshes

Meshes

Mesh data structures

Two opposing design criteria

- **Efficient rendering**
- vs.
- **Efficient manipulation**



Rendering Meshes

In real-time rendering triangle meshes are used
(almost exclusively)

Stored as

- triangle strips,
- triangle sets, or
- indexed triangle sets

Efficient for rendering, but

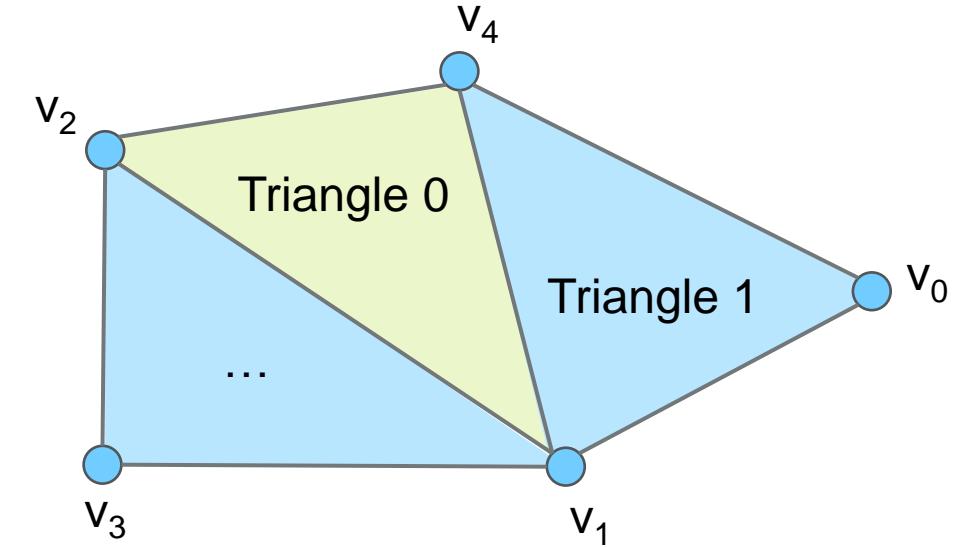
Inefficient for manipulation

Indexed Face Set

Data structure for mesh definition
and rendering

Triangle 0	Triangle 1	...
2 1 4	4 1 0	...

Pos array	Normal array	Colour array
Pos 0	Normal 0	Colour 0
Pos 1	Normal 1	Colour 1
Pos 2	Normal 2	Colour 2
Pos 3	Normal 3	Colour 3
Pos 4	Normal 4	Colour 4
...



Indexed Face Set

Also supported by OpenGL and DirectX

Another example

VRML97 IndexedFaceSet Node

VRML ... Virtual Reality Modeling Language see

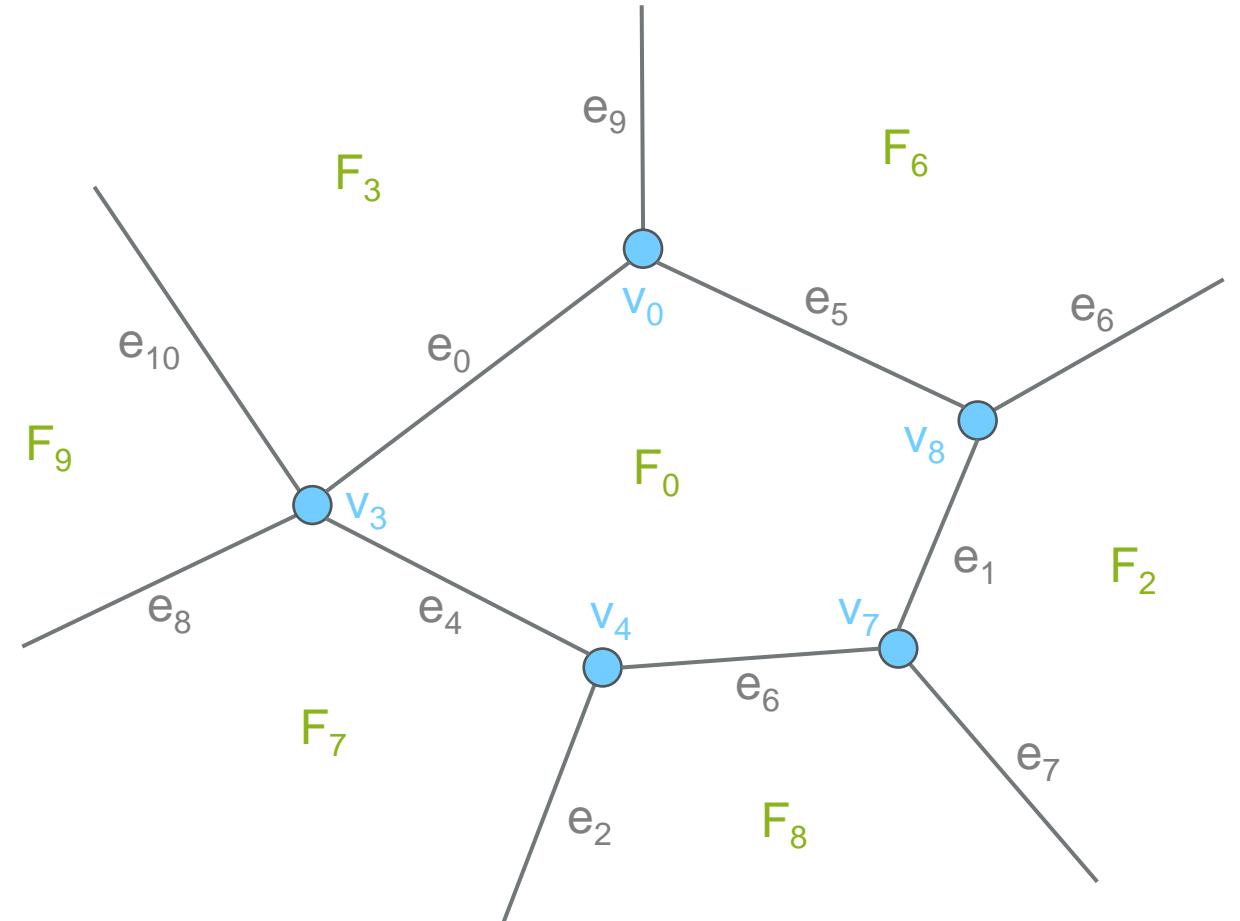
tecfa.unige.ch/guides/vrml/vrml97/spec/part1/nodesRef.html#IndexedFaceSet

Manipulating Meshes

Winged-Edge Data Structure

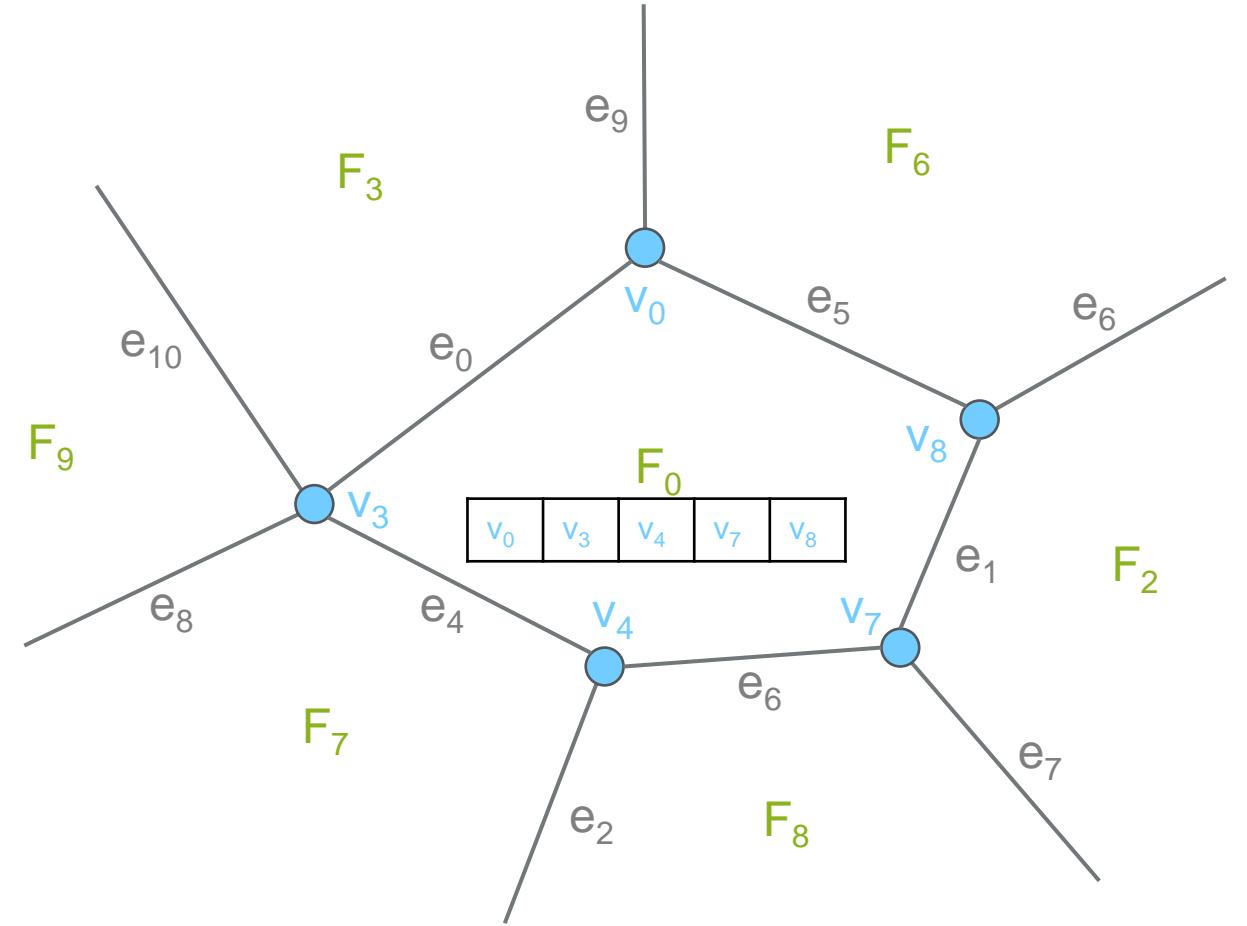
- Vertices
- Edges
- Faces

Stores pointers to neighbours



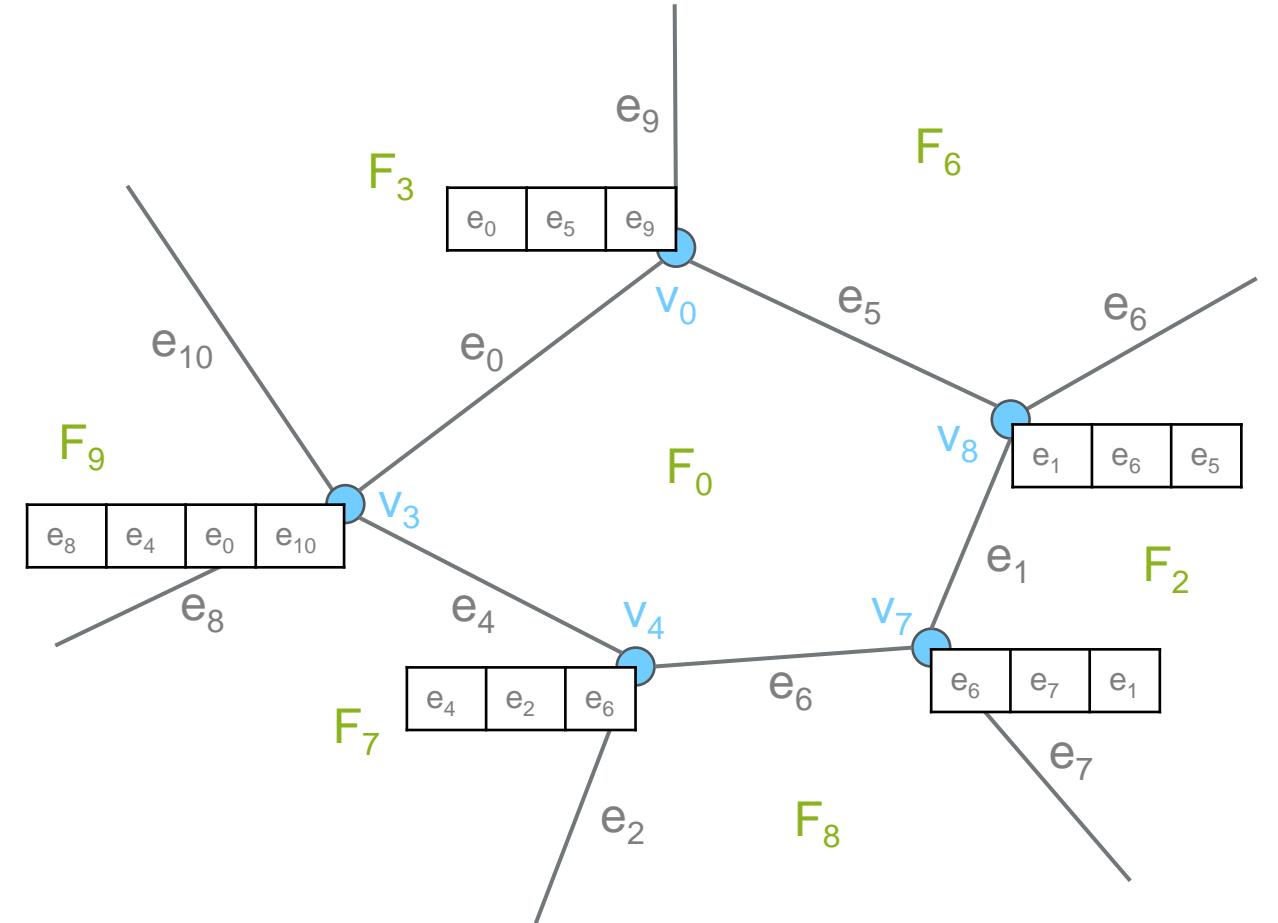
Winged-Edge

Face stores pointers to adjacent vertices



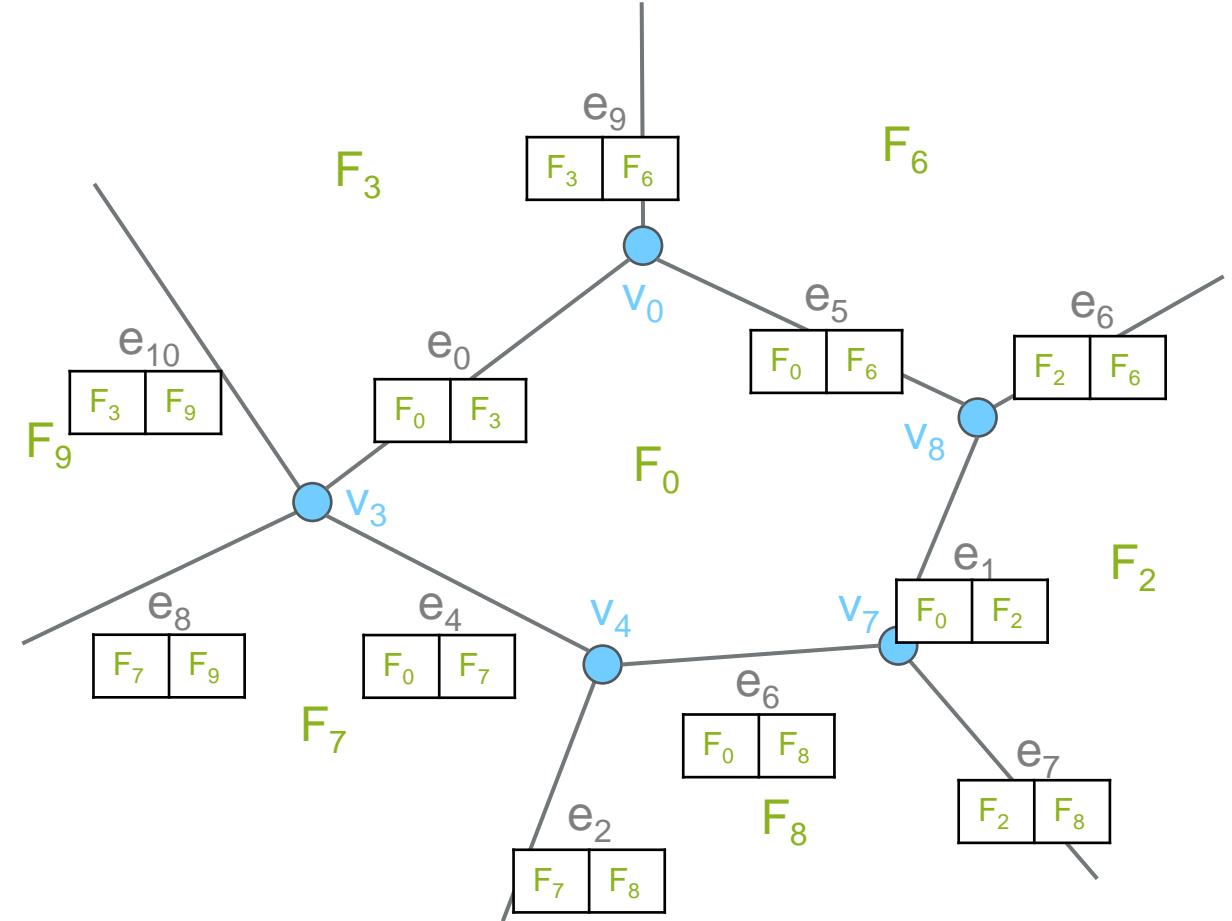
Winged-Edge

Vertex stores pointers to adjacent edges



Winged-Edge

Edge stores pointers to adjacent faces



Manipulating Meshes

Mesh can be dynamically changed

- Insertion of new vertices, edges, and faces
- Deletion of vertices, edges, and faces

Inefficient structure for rendering but
efficient for manipulation

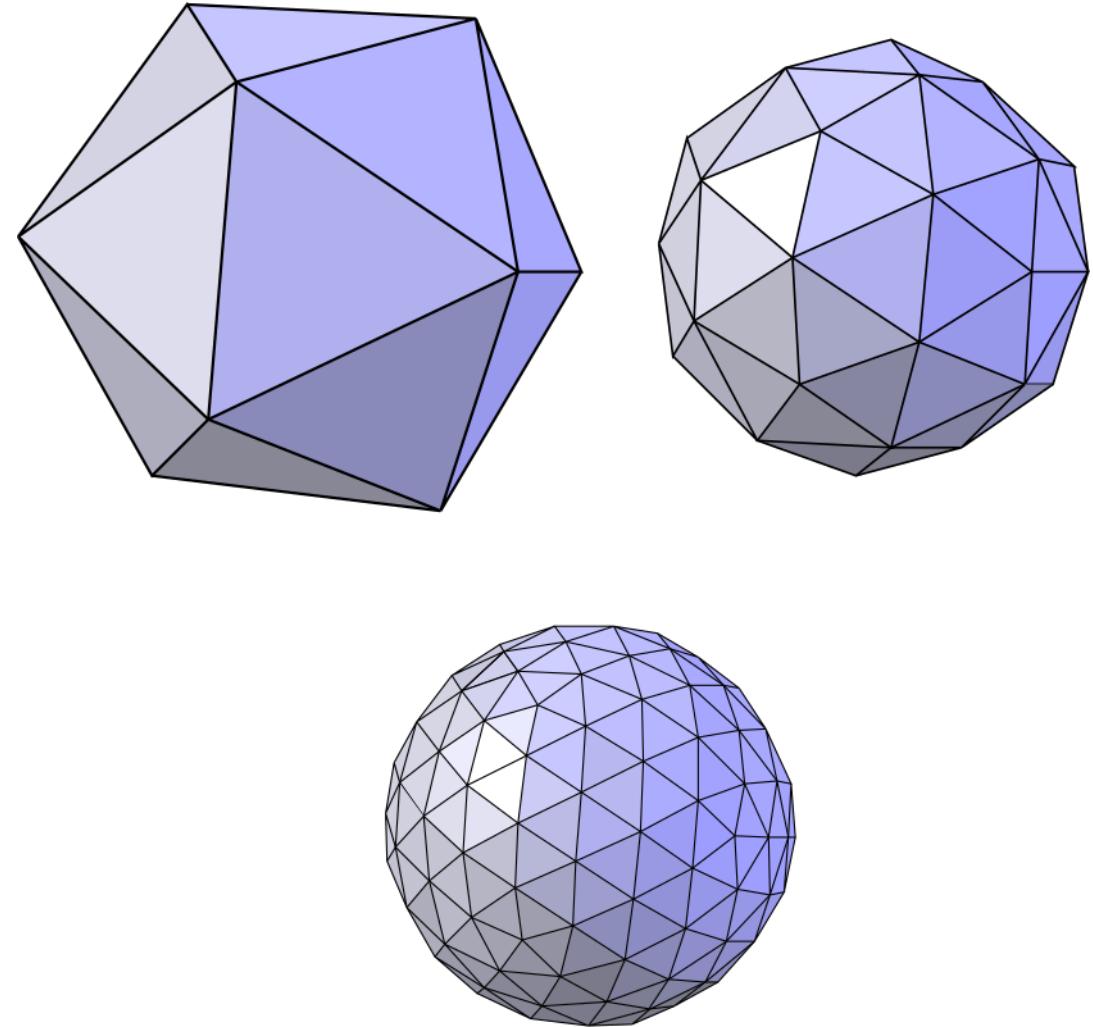
E.g. mesh editing

Manipulating Meshes

Example:

Subdivision Surfaces

In order to refine the mesh,
connectivity info is needed



© CC BY-SA 3.0 Simon Fuhrmann



Geometry 2

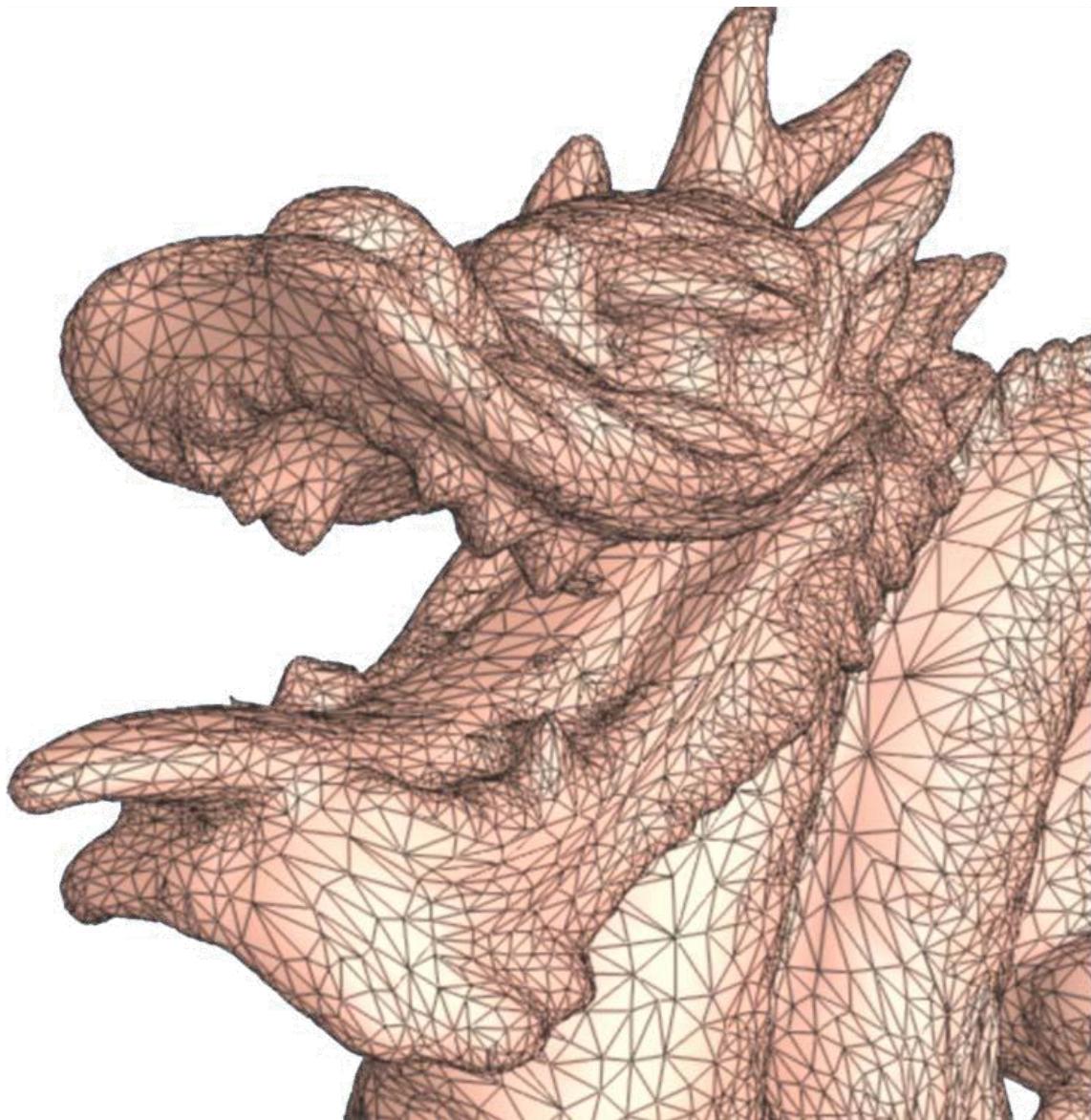
Geometry - Overview

Geometry 1

- Data structures to represent 3D surfaces
- Vertex and its properties
- Drawing primitives
- Meshes and rendering-efficient structures

Geometry 2

- Spatial data structures
- Culling techniques



Spatial Data Structures

Spatial Data Structures

What is it?

- Data structure that organizes geometry in 2D or 3D or higher
- The goal is faster processing
- Needed for most "speed-up techniques"
 - Faster real-time rendering
 - Faster intersection testing
 - Faster collision detection
 - Faster ray tracing and global illumination

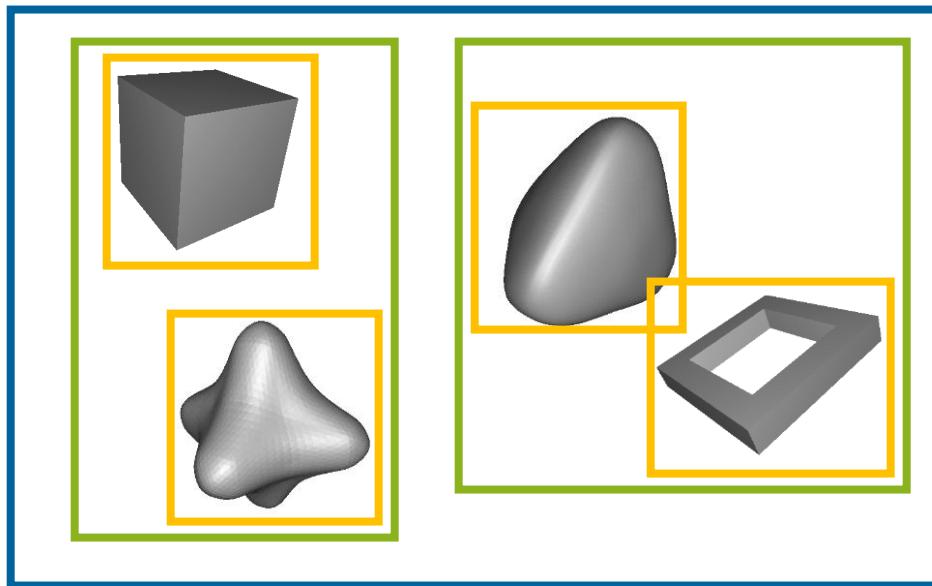
Games use them extensively

Movie production rendering tools always use them too

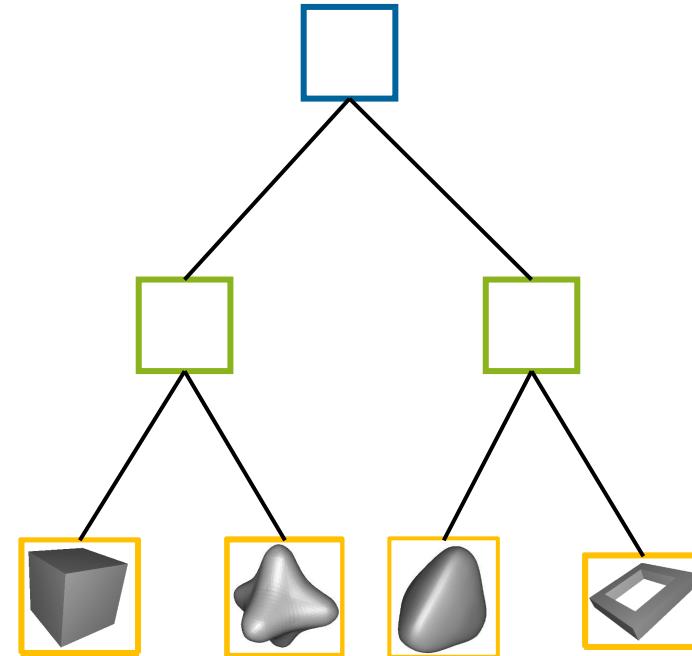
Spatial Data Structures

Organizes geometry in some hierarchy

In 2D space



Data structure



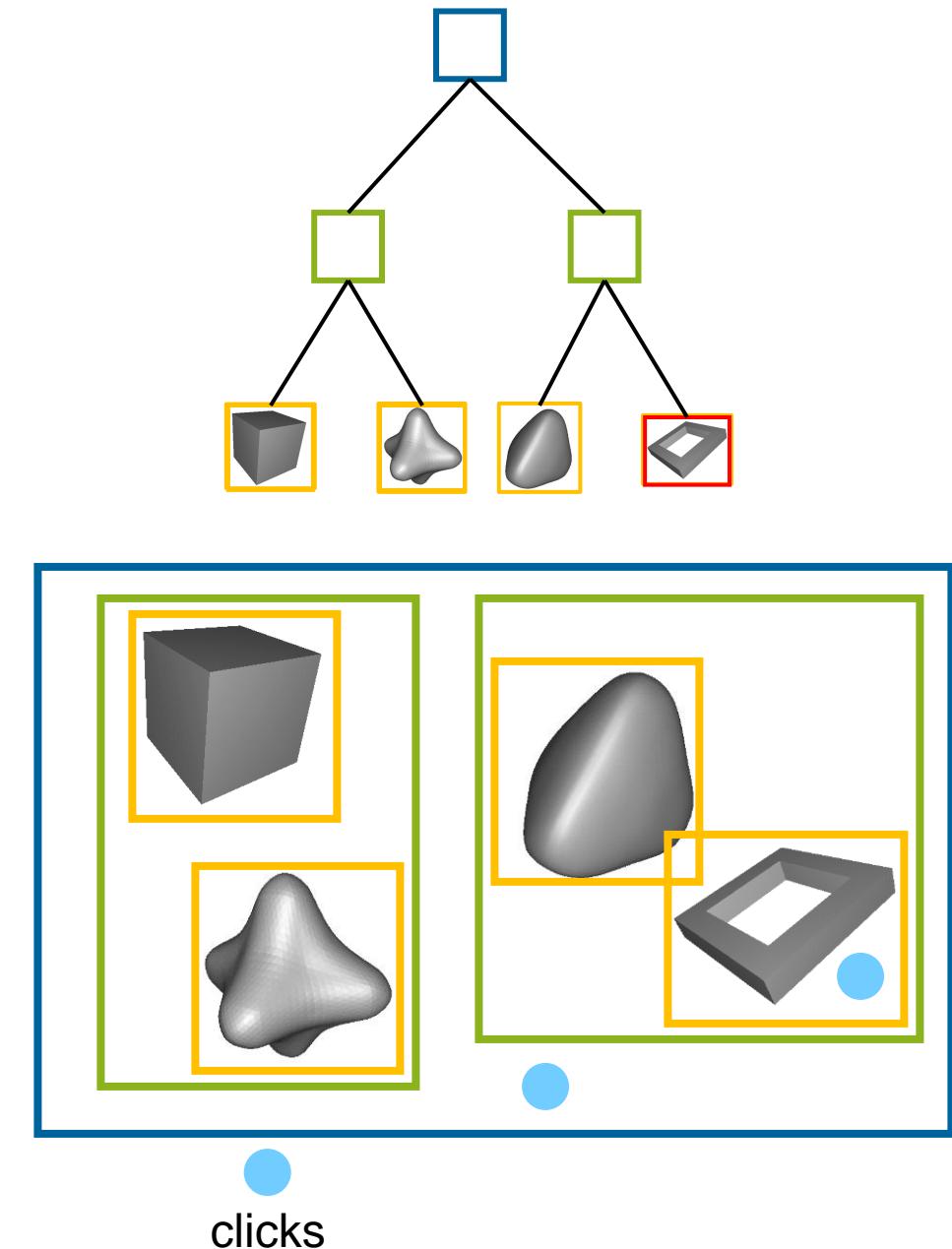
Spatial Data Structures

What is the point? An example ...

Assume we click on the screen, and want to find which object we clicked on

1. Test the root first
2. Descend recursively as needed
3. Terminate traversal when possible

In general: get $O(\log n)$ instead of $O(n)$



Bounding Volume Hierarchy (BVH)

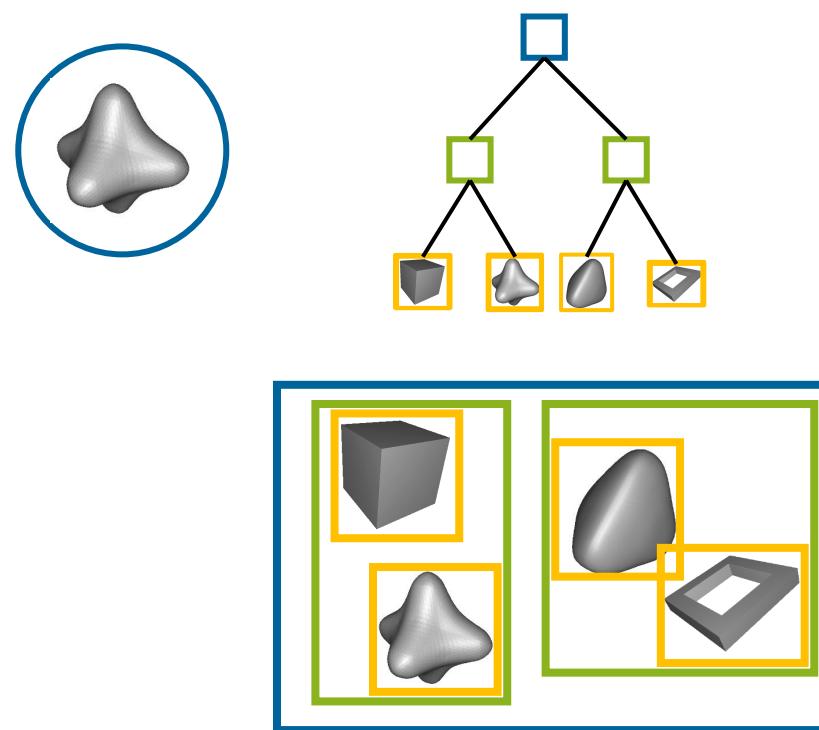
Most common bounding volumes (BVs):

- Sphere
- AABB – axis aligned bounding boxes
- OBB – oriented bounding boxes

The BV does not contribute to the rendered image, - rather encloses an object

The data structure is a tree with k branches

- Leaves hold geometry
- Internal nodes have at most k children
- Internal nodes hold BVs that enclose all geometry in its subtree



Bounding Volume Hierarchy (BVH)

Some facts about trees

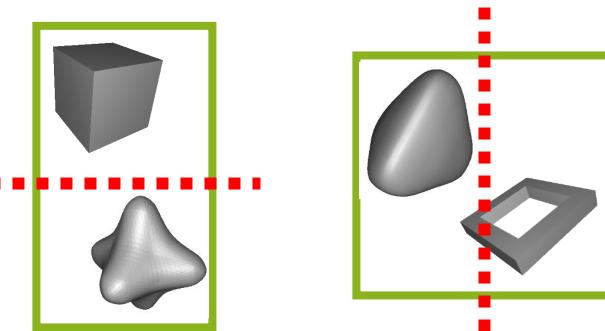
- Height of tree h is the longest path from root to leaf
- A balanced tree has all leaves at height h or $h+1$
- Height of balanced tree with n nodes: $\text{floor}(\log_k(n))$
- Binary tree ($k = 2$) is the simplest
 - $k = 4$ and $k = 8$ is quite common for computer graphics as well

Bounding Volume Hierarchy (BVH)

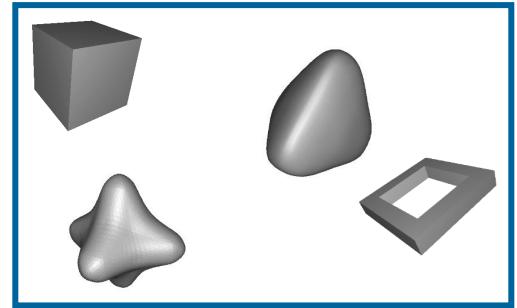
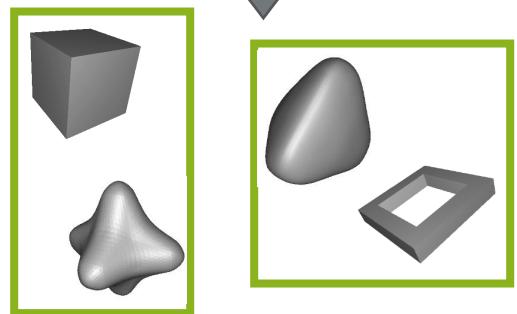
How to create a BVH?

Example: $\text{BV} = \text{AABB}$

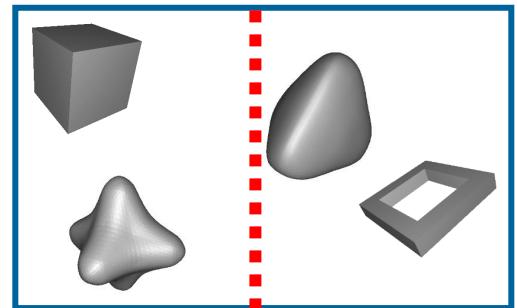
- Find minimal box, then split along longest axis
- Called **TOP-DOWN** method
- More complex for other BVs



Find minimal boxes



x is longest



Bounding Volume Hierarchy (BVH)

Stopping criteria for top-down creation

Need to stop recursion some time...

- Either when BV is empty
- Or when only one primitive (e.g. triangle) is inside BV
- Or when $< n$ primitives is inside BV
- Or when max. recursion level has been reached

Similar criteria for BSP trees and octrees

Binary Space Partitioning (BSP) Tree

Two different types:

- Axis-aligned
- Polygon-aligned

General idea:

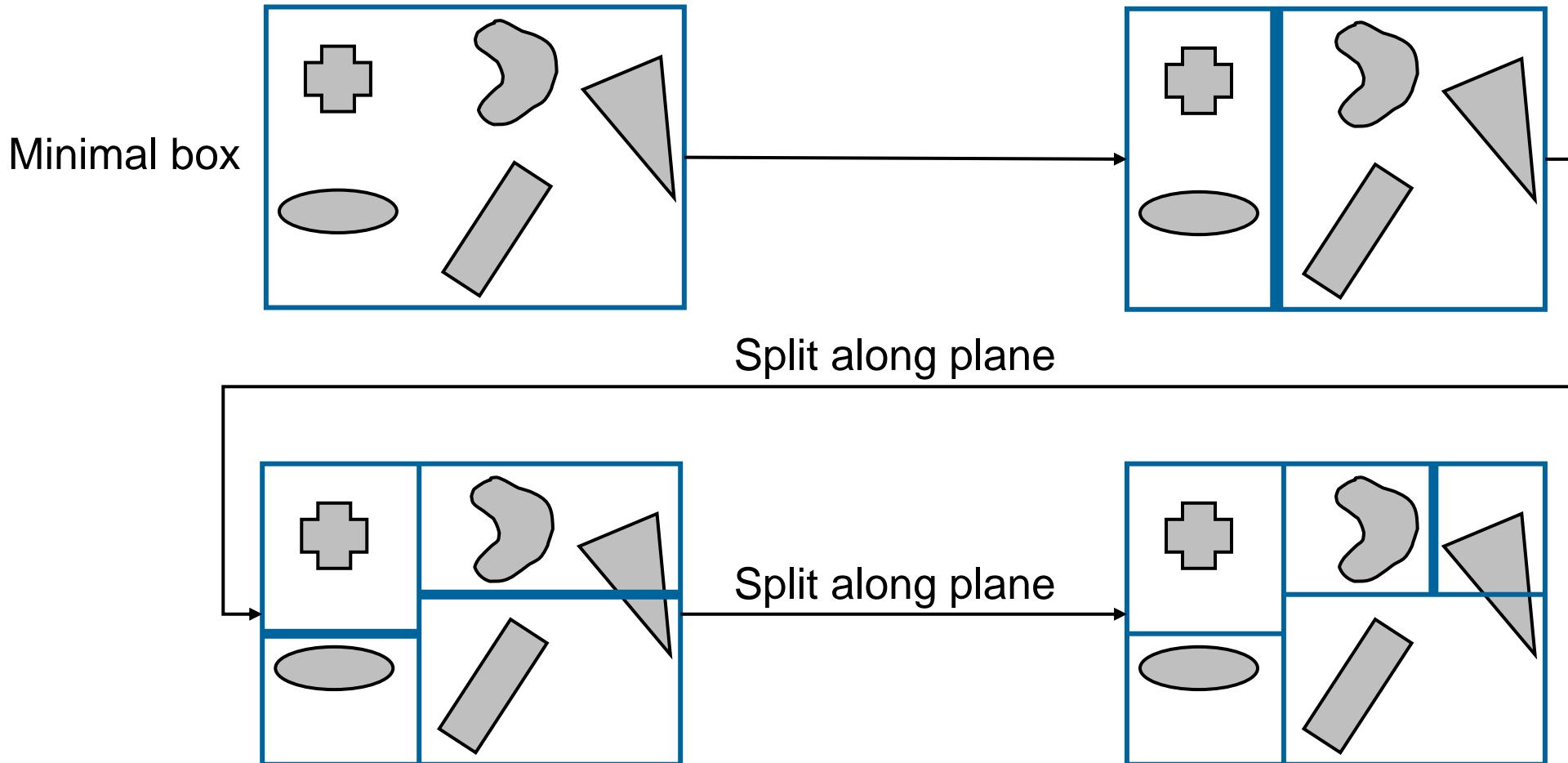
- Divide space with a plane
- Sort geometry into the sub-space it belongs
- Done recursively

If traversed in a certain way, we can get the geometry sorted along an axis

- Exact for polygon-aligned
- Approximately for axis-aligned

Axis-Aligned BSP Tree

Can only make a splitting plane along x, y, or z



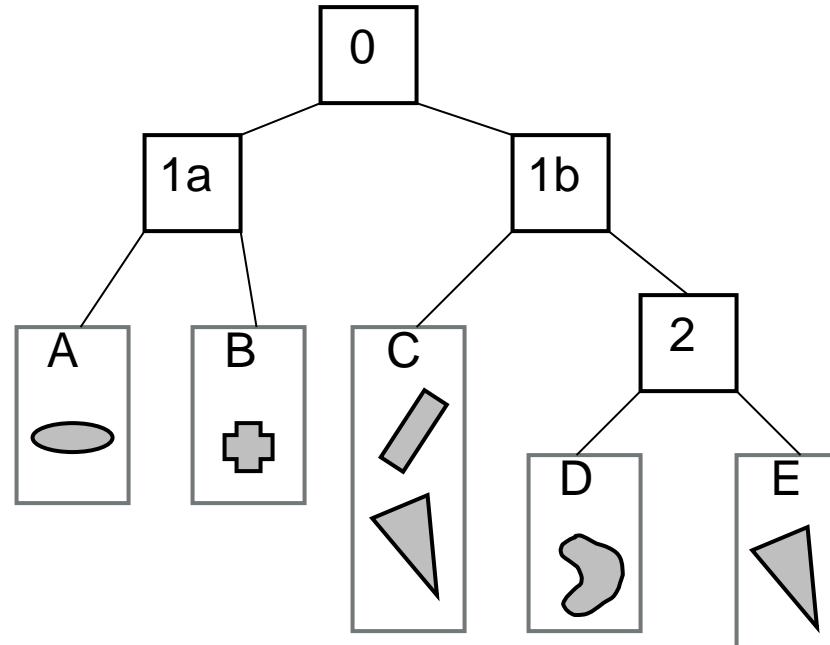
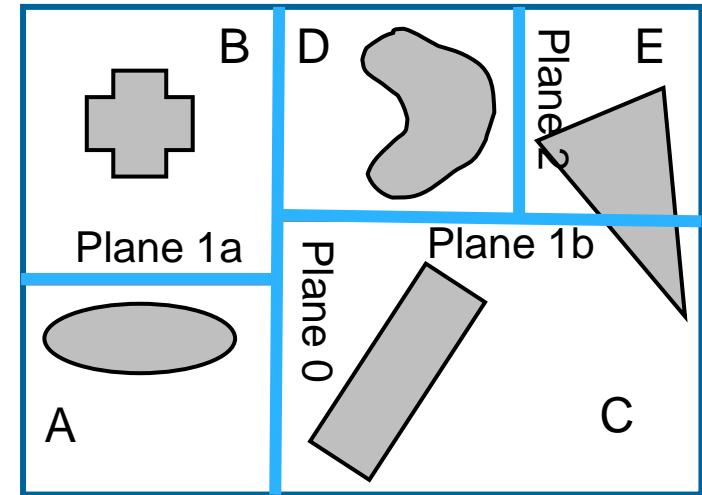
Axis-Aligned BSP Tree

Each internal node holds a divider plane

Leaves hold geometry

Differences compared to BVH

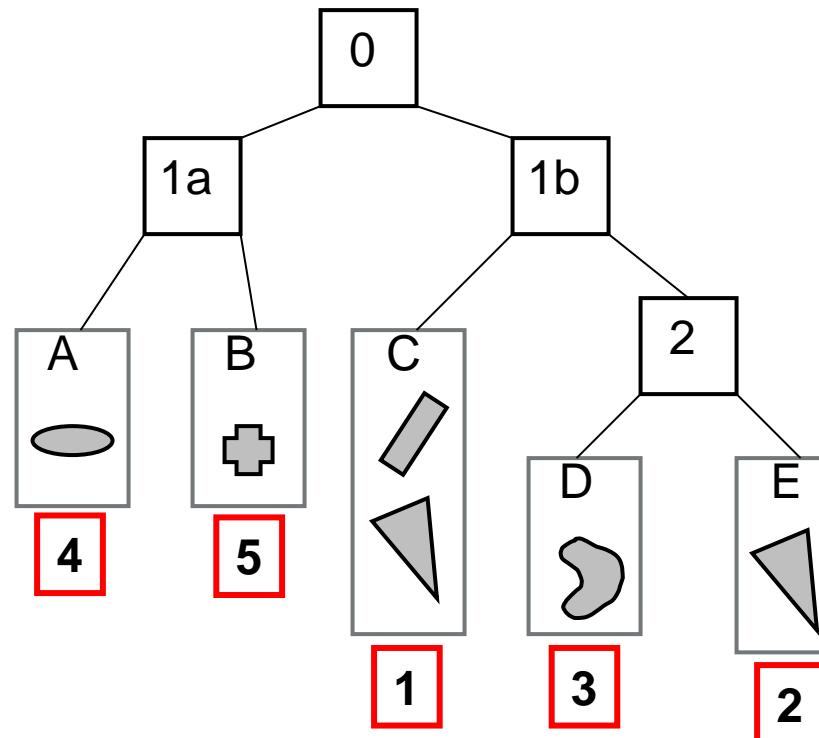
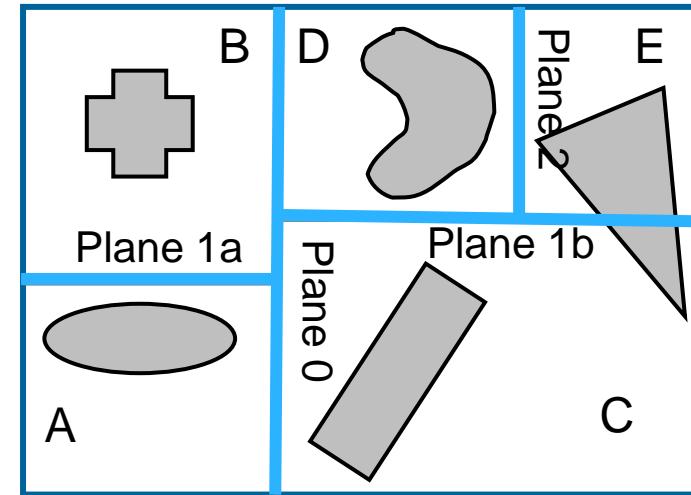
- Encloses entire space and provides sorting
- The BV hierarchy can be constructed in any way (no sort)
- BVHs can use any desirable type of BV



Axis-Aligned BSP Tree

Rough sorting

- Test the planes against the point of view
- Test recursively from root
- Continue on the "hither" side to sort front to back
- Works in the same way for polygon-aligned BSP trees, - but gives exact sorting

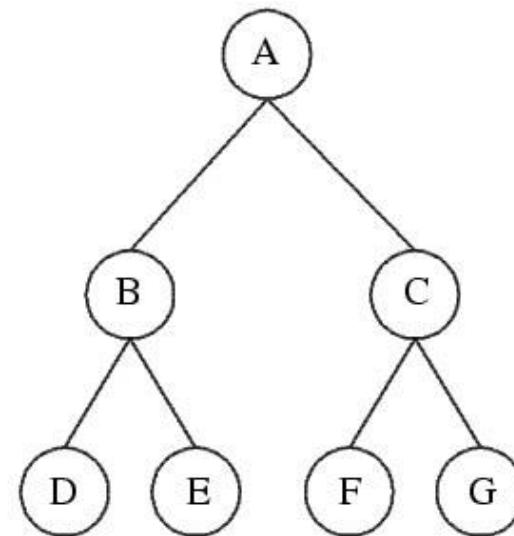
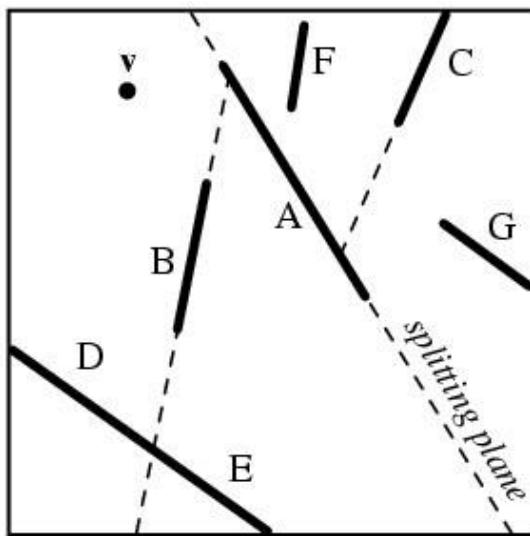


Polygon-aligned BSP Tree

Allows exact sorting

Very similar to axis-aligned BSP tree

- But the splitting plane are now located in the planes of the triangles

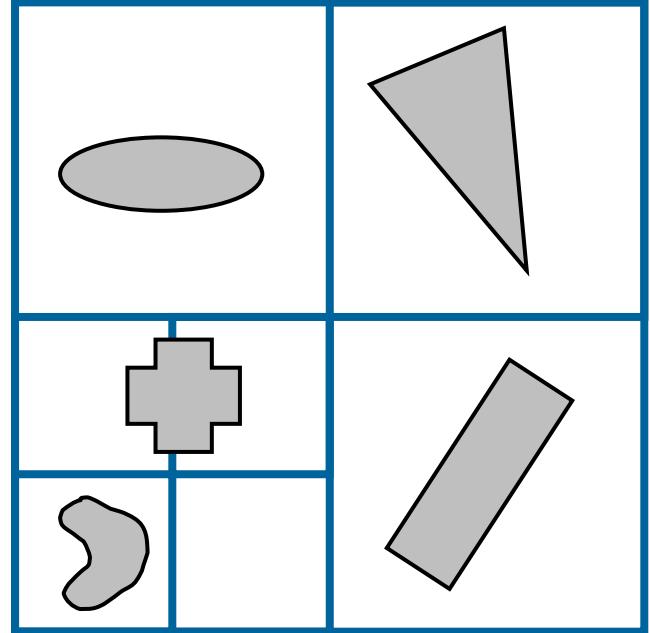
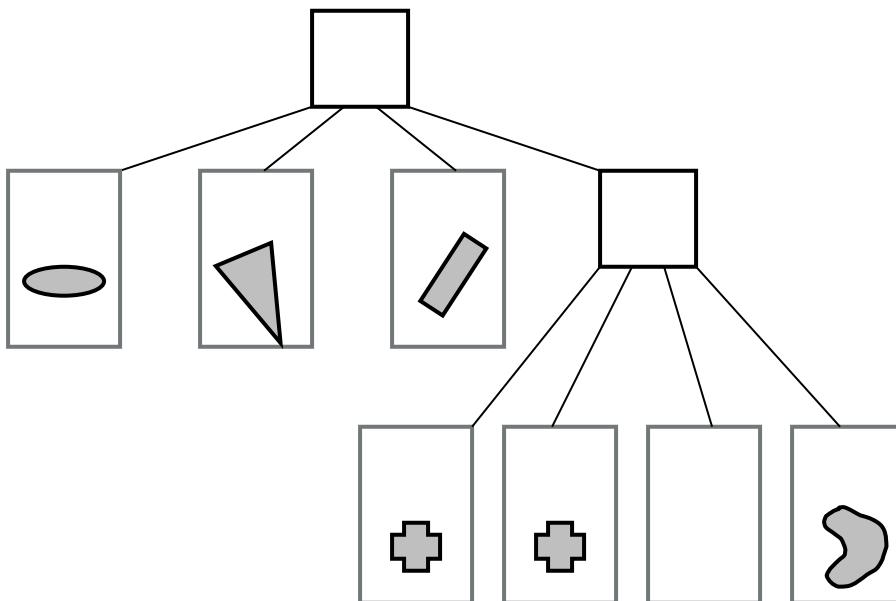


Octree

A bit similar to axis-aligned BSP trees

Will explain the quadtree, which is the 2D variant of an octree

In 3D each square (or rectangle) becomes a box, and there are 8 children



Octree

Expensive to rebuild (so are BSP trees)

Octrees can be used to

- Speed up ray tracing
- Faster picking
- Culling techniques
- Are not used that often in real-time contexts, - except loose octrees

Scene Graph

BVH is the data structure that is used most often

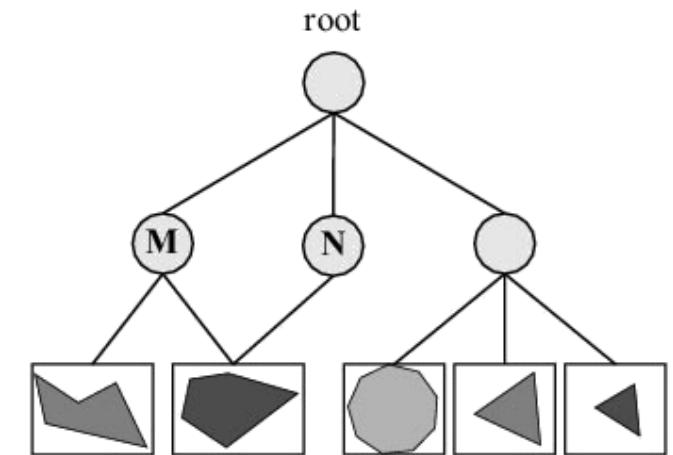
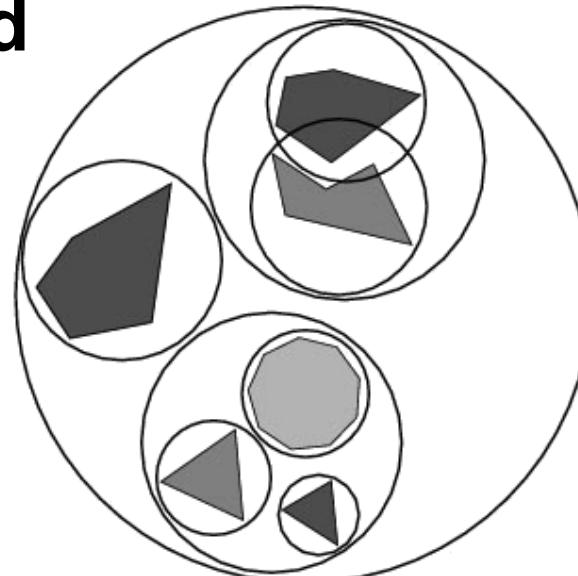
- Simple to understand
- Simple code

However, it stores just geometry

- Rendering is more than geometry

The scene graph is an extended
BVH with:

- Lights
- Textures
- Transforms
- And more



Speed-Up Techniques

Spatial data structures are used to speed up rendering and different queries

Why more speed?

- **Graphics hardware 2x faster in 6 months!**
- **Wait... then it will be fast enough!**
- **No, it wont!**
- **We will never be satisfied**
 - Screen resolution: 3840 x 2160 (UHD)
 - Realism: global illumination
 - Geometrical complexity: no upper limit!

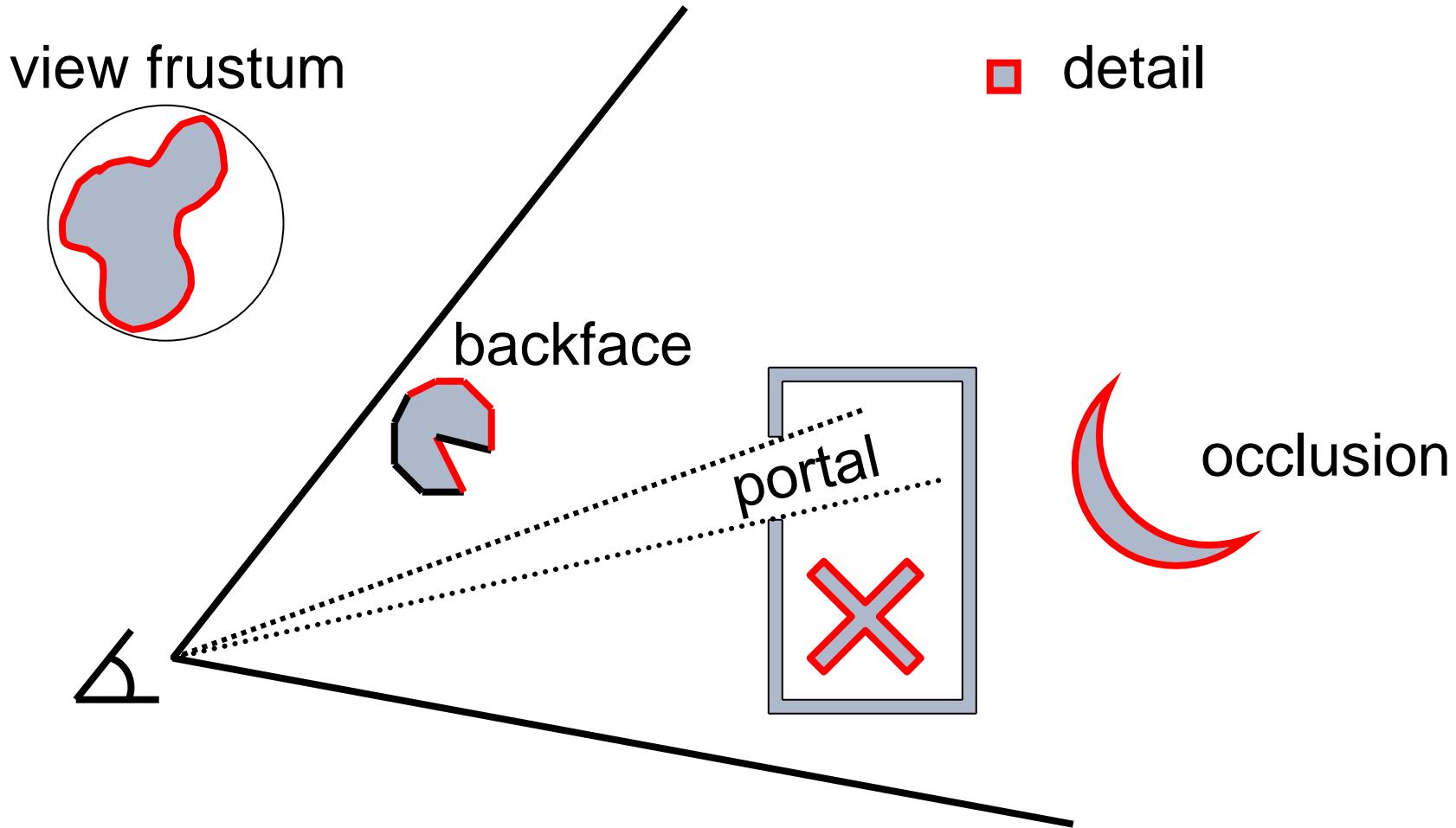
Culling

Culling

To cull” means “to select from group”

In graphics context: Do not process data that will not contribute to the final image

Different Culling Techniques



Backface Culling

Simple technique to discard polygons that faces away from the viewer

Can be used for:

- **Closed surface (example: sphere)**
- **Or whenever we know that the backfaces never should be seen (example: walls in a room)**

Two methods (screen space, eye space)

Which stages benefits?

- **Rasterizer, but also Geometry (where test is done)**

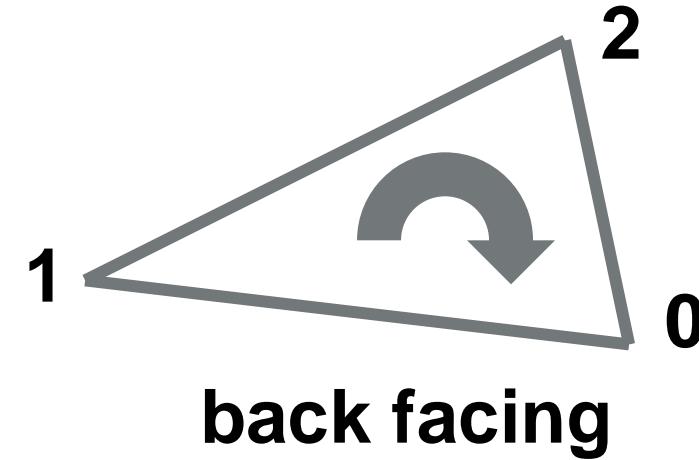
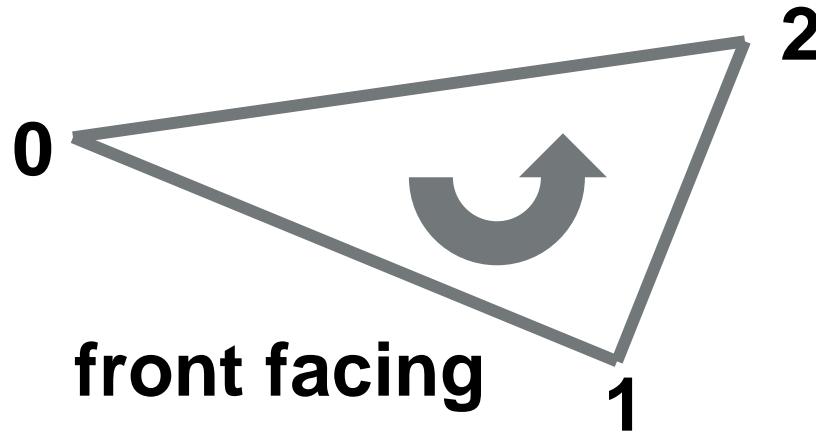
Backface Culling

Often implemented for you in the API

OpenGL: `glCullFace(GL_BACK);`

How to determine what faces away?

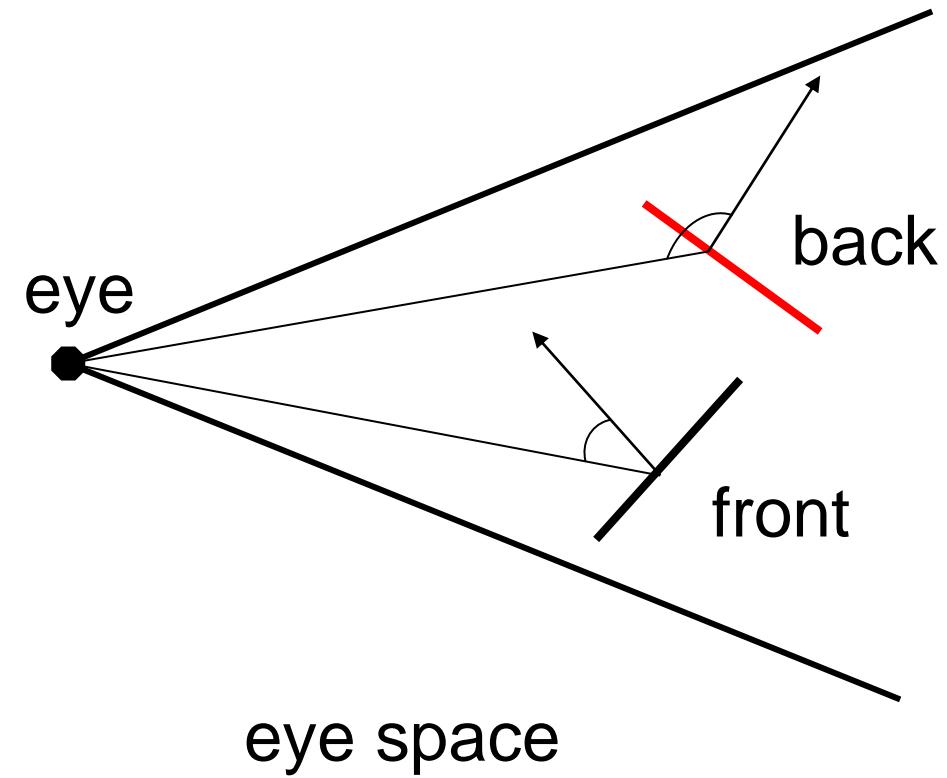
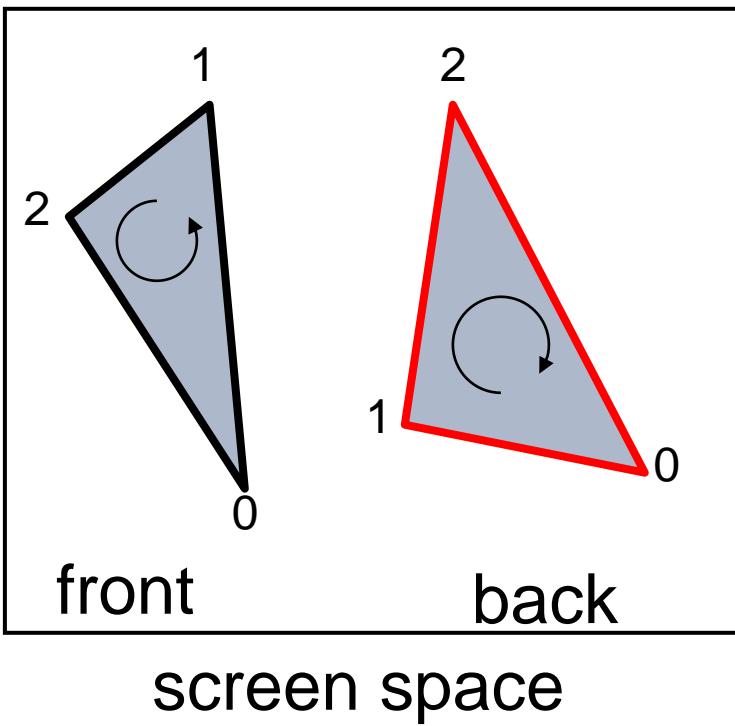
First, must have consistently oriented polygons, e.g.,
counterclockwise



Backface Culling

How to cull backfaces

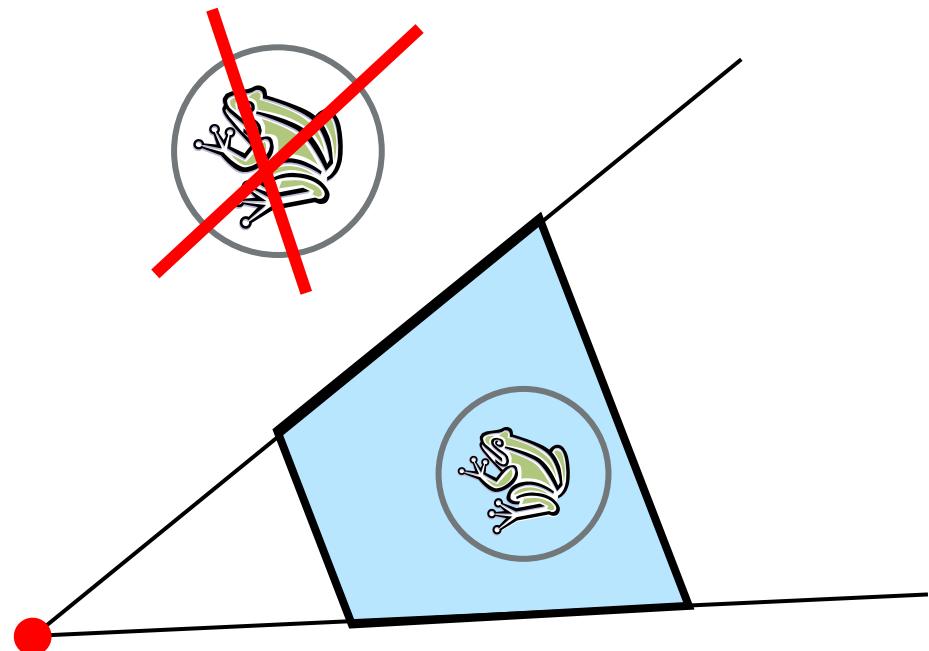
Two ways in different spaces



View-Frustum Culling

Bound every “natural” group of primitives by a simple volume (e.g., sphere, box)

If a bounding volume (BV) is outside the view frustum, then the entire contents of that BV is also outside (not visible)



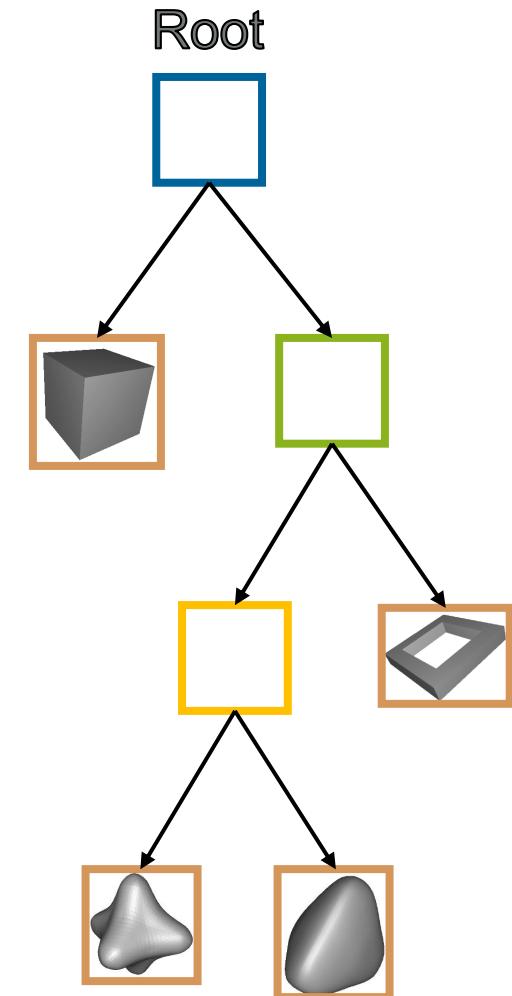
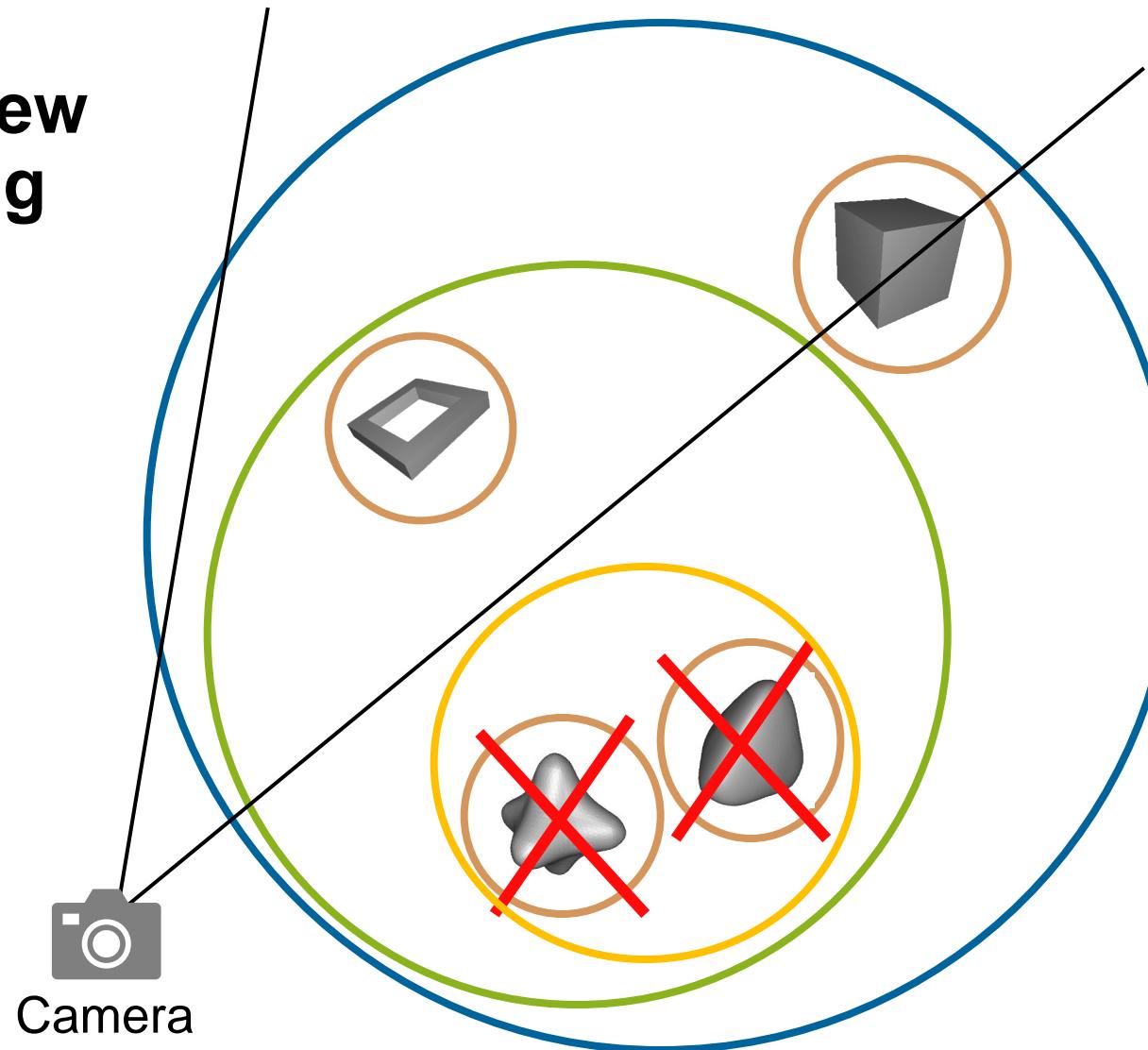
View-Frustum Culling

Can we accelerate view frustum culling further?

- Do what we always do in graphics...
- Use a hierarchical approach, e.g., a spatial data structure (BVH, BSP, scene graph)
- Which stages benefits?
 - Geometry and Rasterizer
 - Bus between CPU and Geometry

View-Frustum Culling

Example of
Hierarchical View
Frustum Culling



Portal Culling

Average: Culled 20-50% of the polys in view

Speedup: From slightly better to 10 times

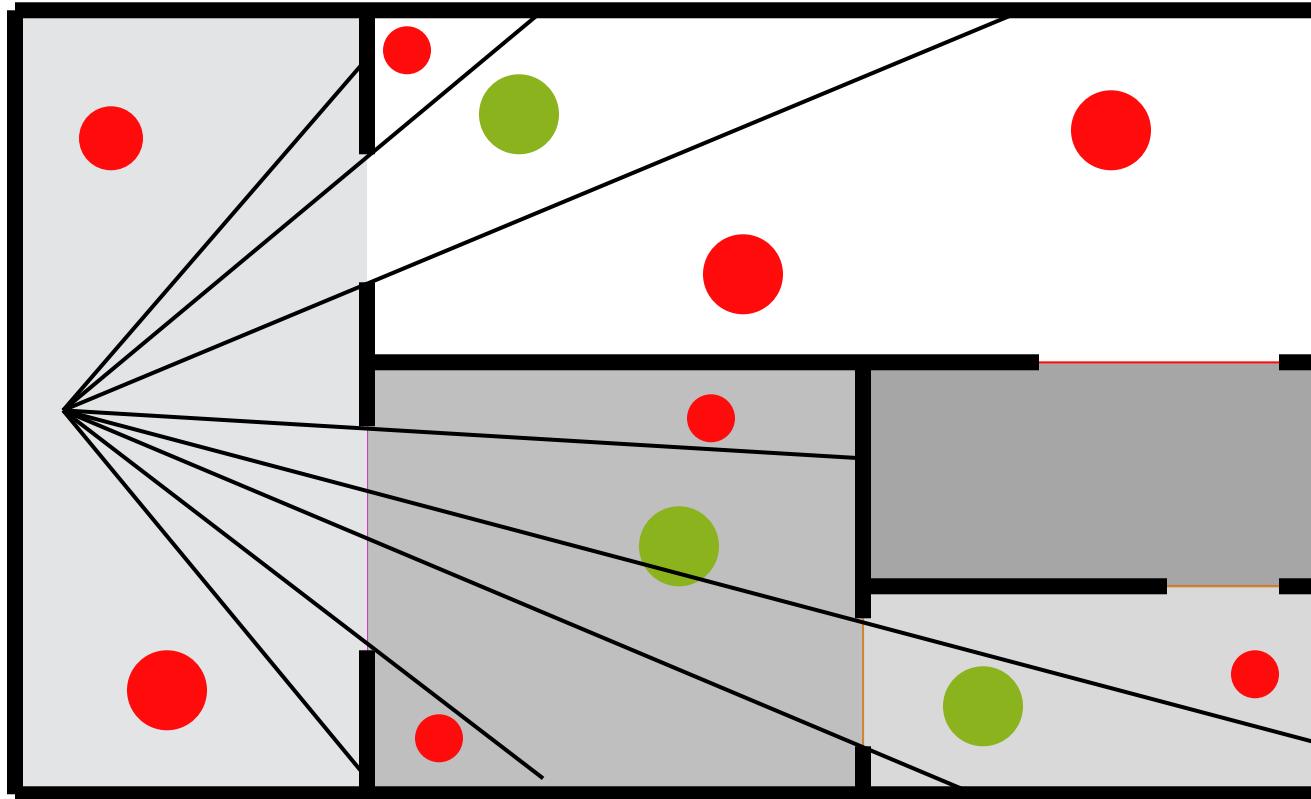


Images courtesy of David P. Luebke and Chris Georges

Portal Culling

Example

- In a building from above
- Circles are objects to be rendered



Portal Culling Algorithm

Divide into cells with portals (build graph)

```
for each frame {  
    Locate cell of viewer and init 2D AABB to whole screen;  
    while (current AABB is not empty && cell not too far away) {  
        Render current cell with View Frustum culling w.r.t. AABB;  
        Traverse to closest cells (through portals);  
        Intersection of AABB & AABB of traversed portal;  
    }  
}
```

Portal Culling Algorithm

When to exit:

- When the current AABB is empty
- When we do not have enough time to render a cell (“far away” from the viewer)

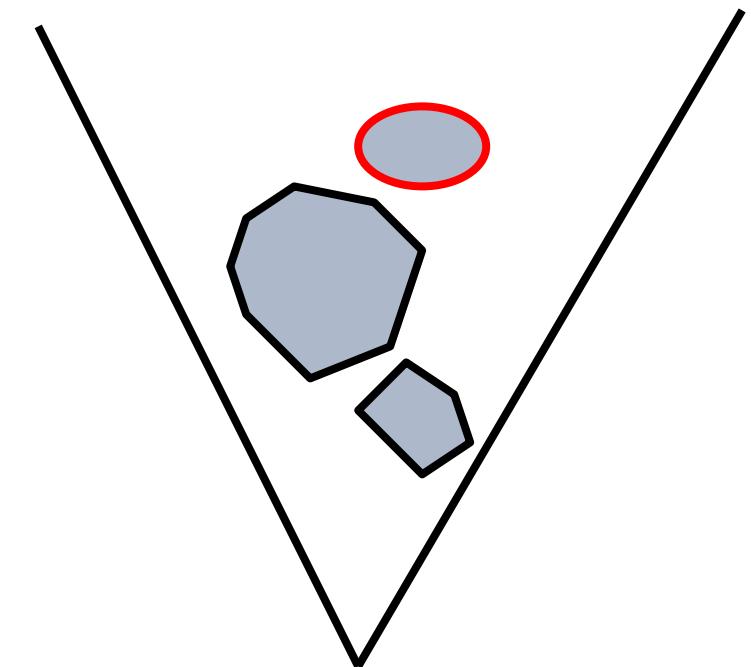
Also: Mark rendered objects

Occlusion Culling

Main idea: Objects that lies completely “behind” another set of objects can be culled

Hard problem to solve efficiently

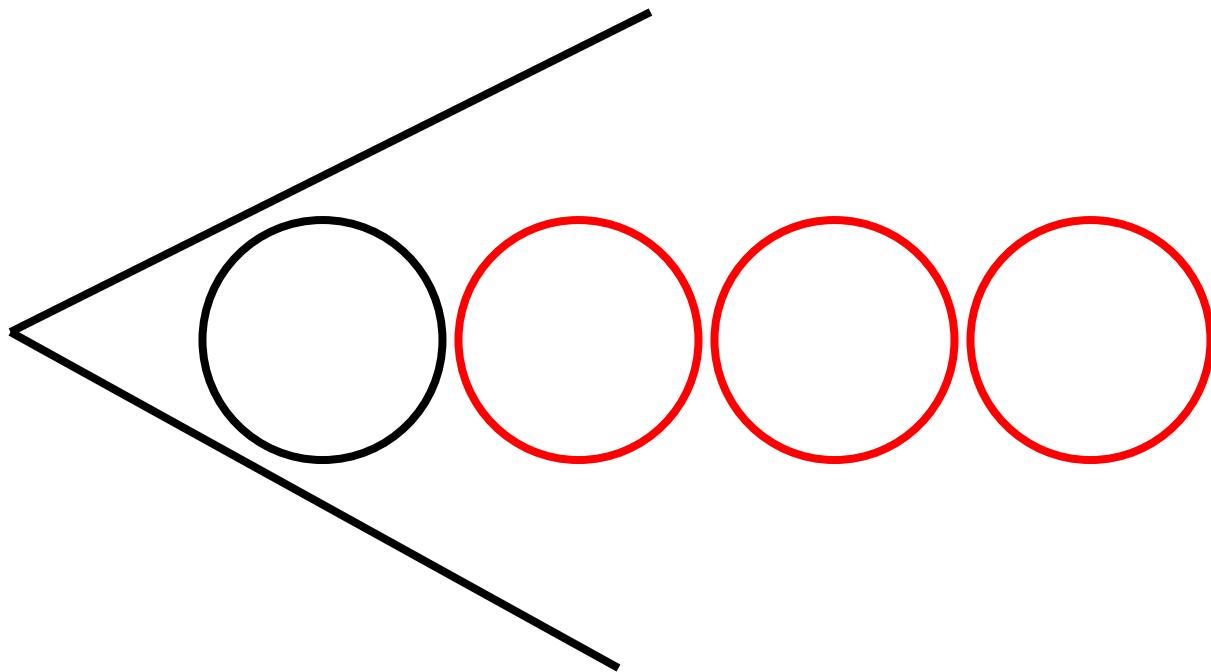
Lots of research in this area



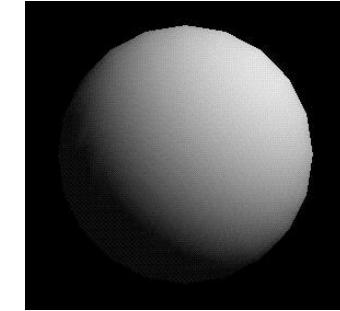
Occlusion Culling

Example

Note that “Portal Culling” is type of occlusion culling



final image



Occlusion Culling Algorithm

Use some kind of occlusion representation O_R

```
for each object obj {  
    if( not Occluded( $O_R$  ,obj) ) {  
        render(obj);  
        update( $O_R$ , obj);  
    }  
}
```

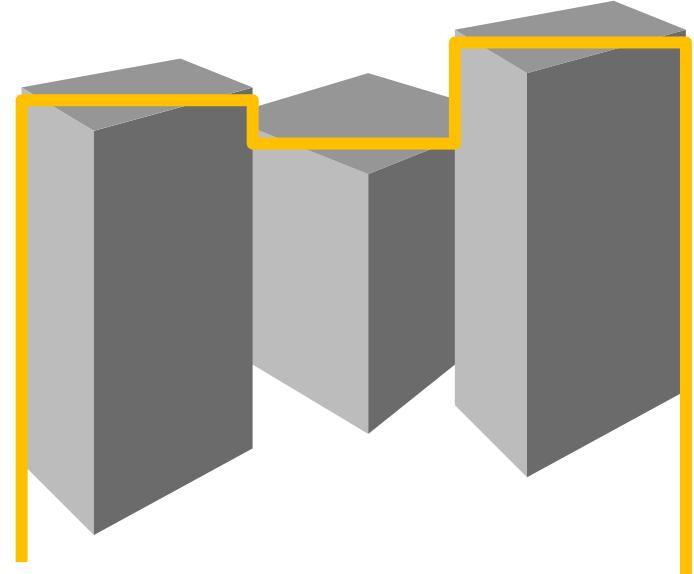
Occlusion Horizon: A simple algorithm

Target: urban scenery

- Dense occlusion
- Viewer is about 2 meters above ground

Algorithm:

- Process scene in front-to-back using a quad tree
- Maintain a piecewise constant horizon
- Cull objects against horizon
- Add visible objects' occluding power to the horizon

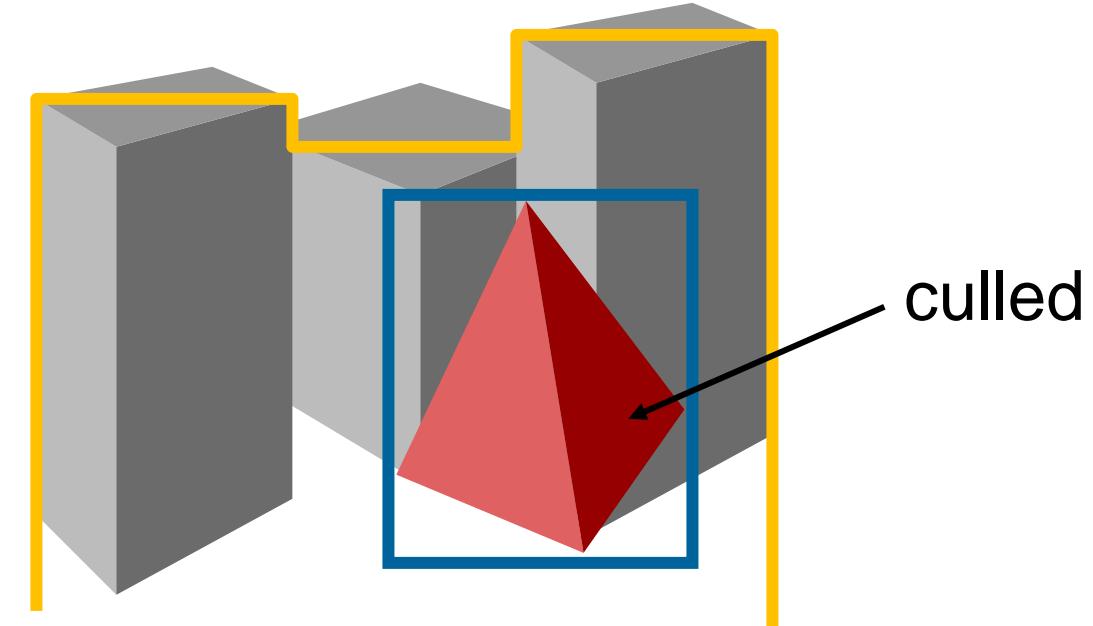


Occlusion Horizon: A simple algorithm

Occlusion testing with occlusion horizons

To process tetrahedron (which is behind grey objects):

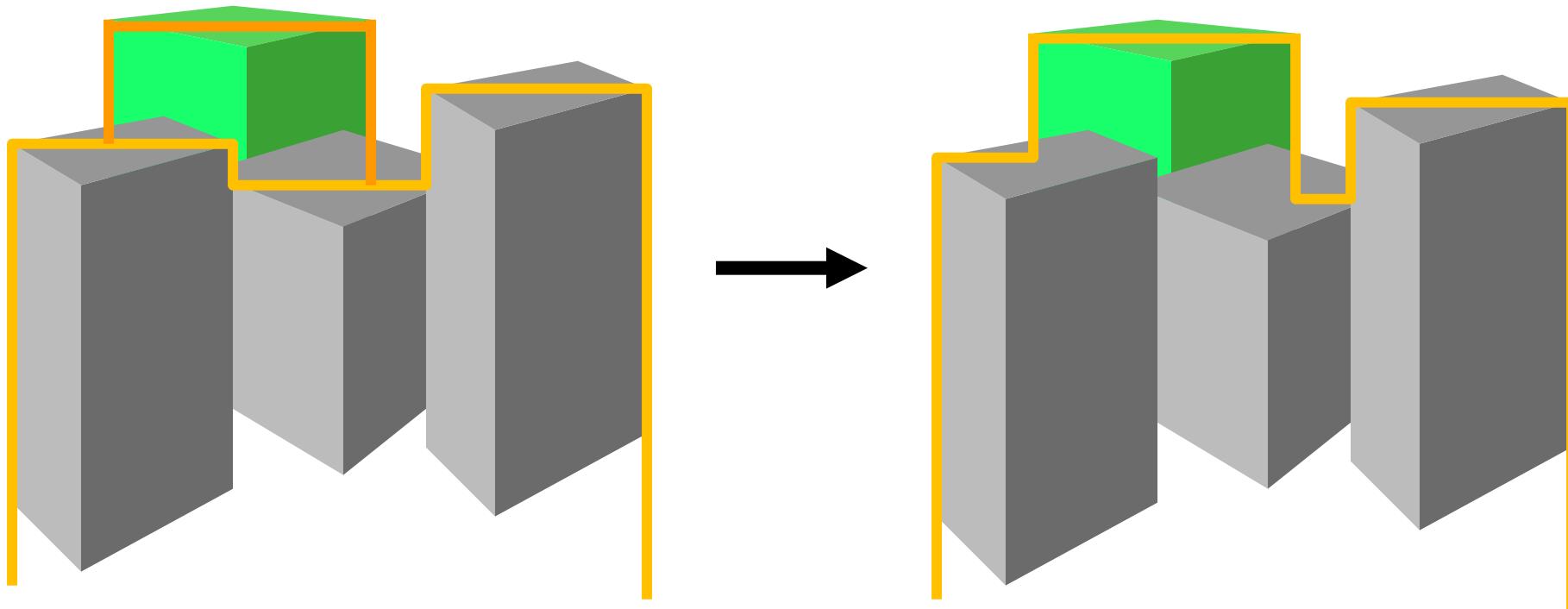
- Find axis-aligned box of projection
- Compare against occlusion horizon



Occlusion Horizon: A simple algorithm

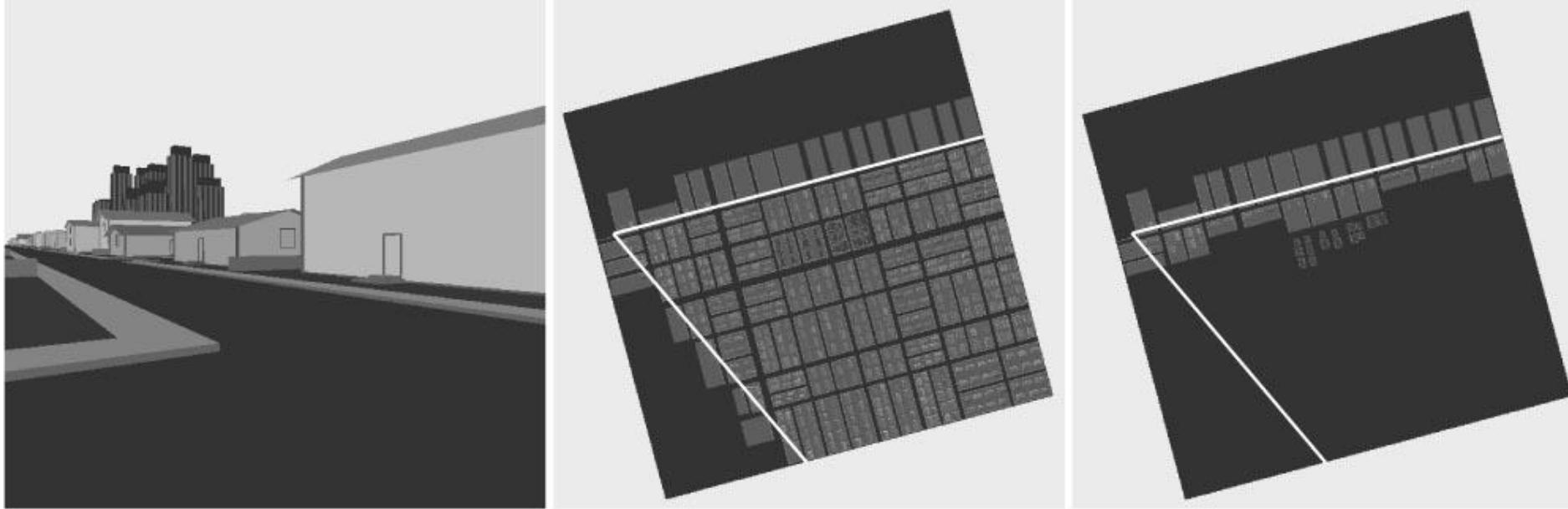
Update horizon

- When an object is considered visible:
- Add its “occluding power” to the occlusion representation



Occlusion Horizon: A simple algorithm

Example





Light and Colour

Overview

- Light in physical reality
- Rendering - simulating physical reality
- Shading
- Light and material models
- High dynamic range
- Tone mapping

Physical Reality



University of
Applied Sciences

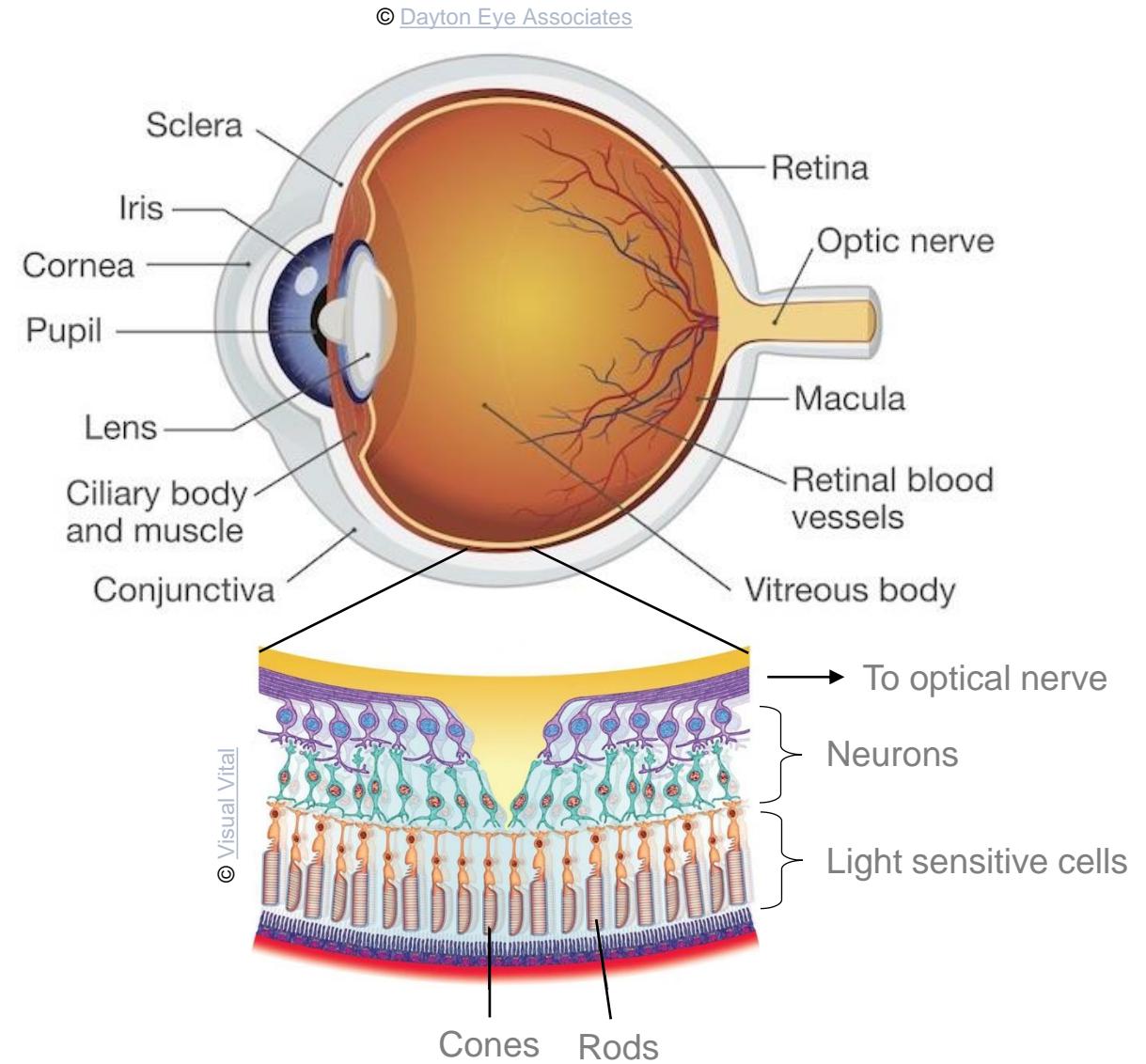
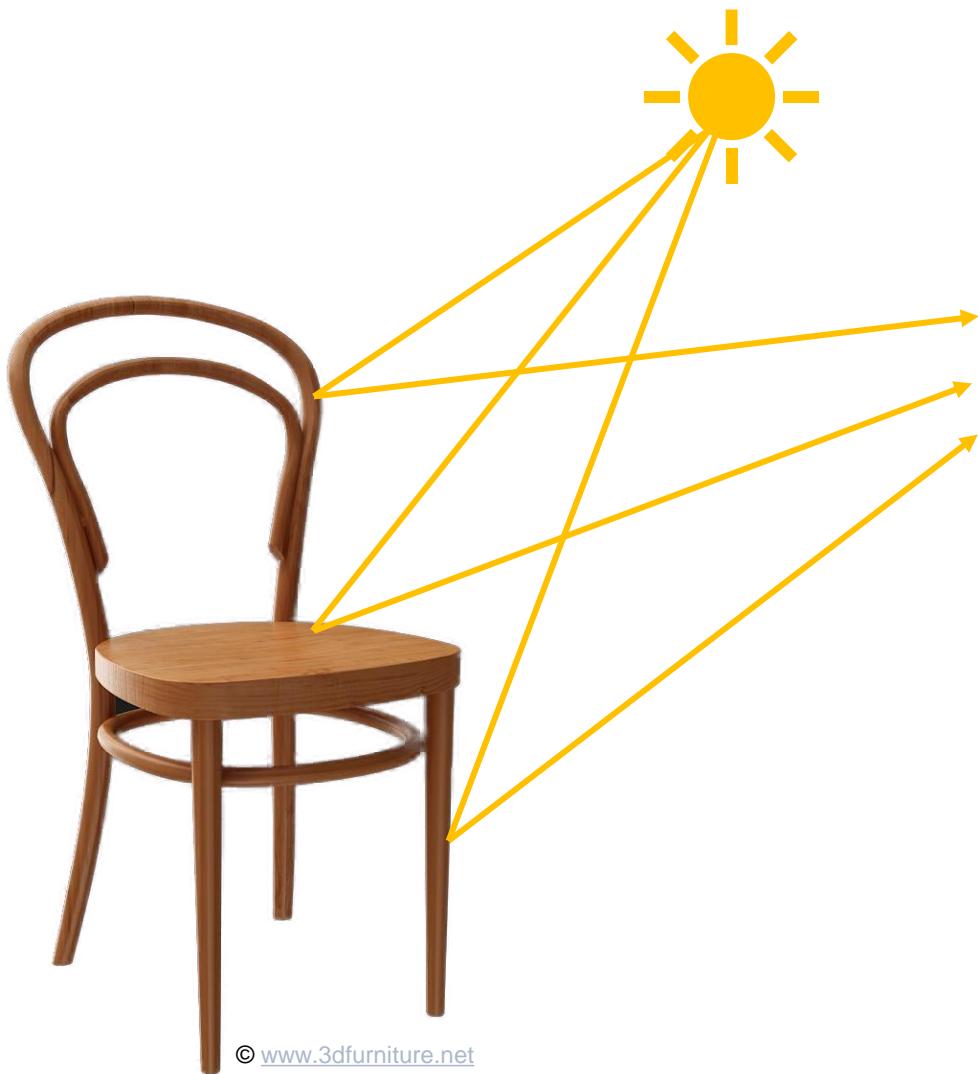
Intro

Warning: „Unconventional Intro Ahead!“

In order to understand what CG can and can't do, we take a very short and very high-level look at „How Reality Works“



Human Vision

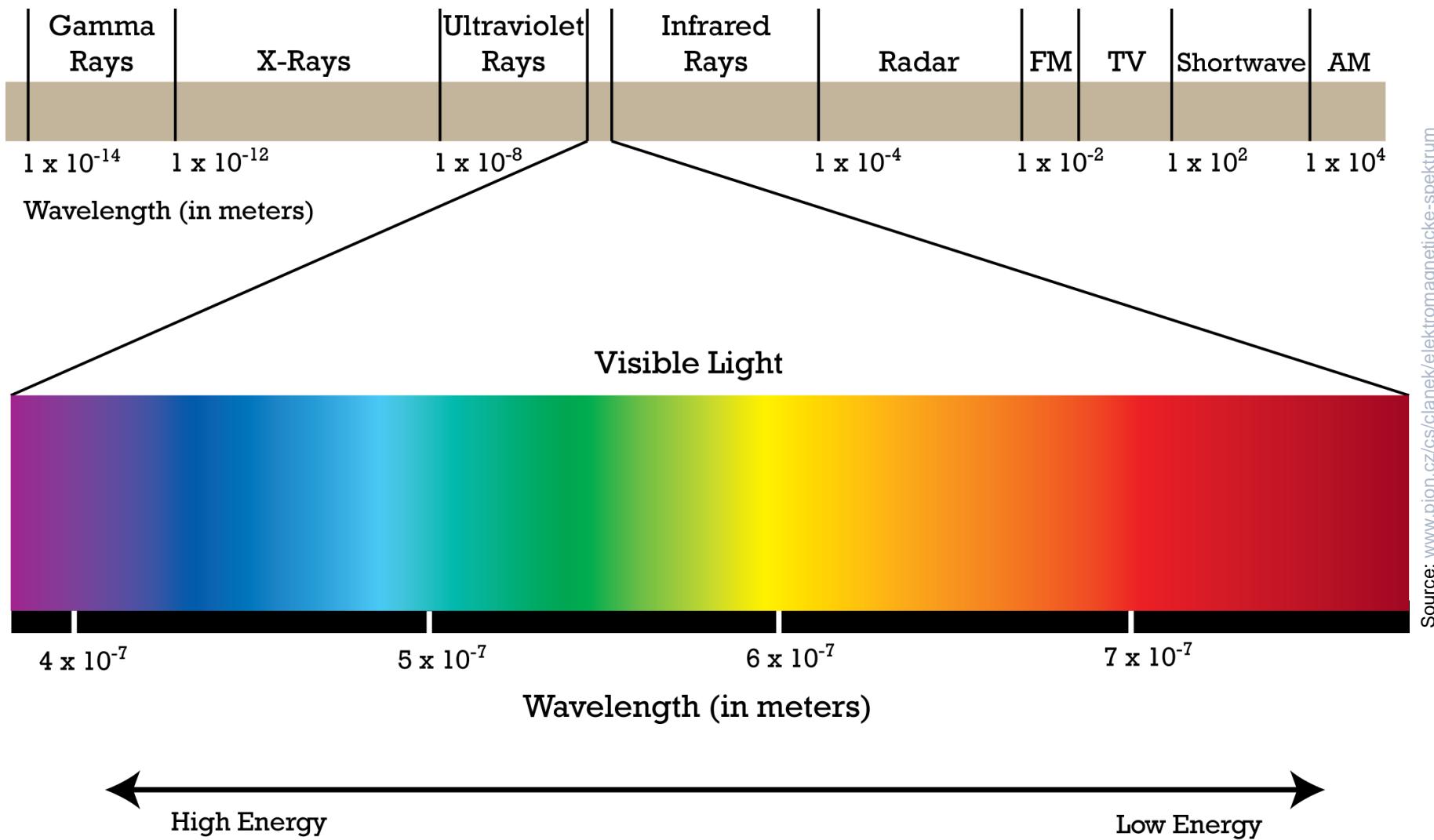


Light

What is Light?

- **Electromagnetic radiation**
- **Visible spectrum at ~ 400 – 700 nm wavelength**

Electromagnetic Spectrum



Electromagnetic Spectrum

- Smaller wavelengths

- Higher frequency
- Higher energy
- Blue colours

- Larger wavelengths

- Lower frequency
- Lower energy
- Red colours



Light-Surface Interactions

Simple Version:

- **If light hits a surface, then**
 - certain wavelengths are absorbed (creating heat)
 - other wavelengths are reflected

Example: A red shirt

- **Absorbs light at green and blue wavelengths**
- **Reflects remaining wavelengths (red)**

Human Visual System (HVS)

Retina contains light-sensitive cells

- **Rods („Stäbchen“)**
 - ~120 000 000
 - for „night vision“ – no colour, only „intensity“
- **Cones („Zäpfchen“)**
 - ~6 000 000
 - 3 different types
 - for red, green and blue wavelengths

Photons

Light can also be interpreted as particles

→ Photons

Why?

- Wave-Particle Duality – Quantum Mechanics
- Ask your physics teacher! 😊

Simulation

Simulation

Why don't we just

- design our virtual world,
 - define all the surface materials,
 - place some virtual light sources,
 - and simulate all the photons?
-
- And all photons that end up in our virtual camera result in **what we see**

Simulation

How much work is this?

Example:

- 30 W LED light (equivalent to 100W tungsten light bulb)
- Emits about 1020 photons / second
- $10^{20} = 100\ 000\ 000\ 000\ 000\ 000\ 000\ !!!$

Simulation

We have a problem!

Assuming we calculate

- 1M photon-scene intersections / second
- first order intersections only (direct illumination)
- photon-surface interactions for free
- with no memory constraints

Simulations takes 10^{14} s \approx 3 million years
→ for 1 second of a single light bulb!!!

Physically Based Rendering

We can not simulate each single photon

But we can approximate the correct solution

- Simulation based on physical laws
- Slow (minutes to even hours / image), but best possible quality
- Convergence

Different Algorithms

- Radiosity
- Ray Tracing (Path Tracing, ...)
- Photon Map

Physically Based Rendering



© PBRT

Physically Based Rendering



© PBRT

© PBRT

Physically Based Rendering



Real-Time Rendering

Interactive (whatever that means)

- Many images/sec are generated
- User acts and reacts
- Immediate feedback

Physical laws are either

- extremely simplified, or
- simply ignored

Real-Time Rendering

BUT - more and more physically-based techniques enter real-time rendering each year

- Growing performance of graphics cards
- Programmable graphics hardware
- Physically-based rendering and real-time rendering slowly converge

Real-Time Rendering

Example:
Subsurface Scattering



Before ...

... we can create such images ourselves

Before ...

... we can create such images ourselves

We have to know about something completely different ...

Before ...

... we can create such images ourselves

We have to know about something completely different ...

MATH 

Light & Material Models

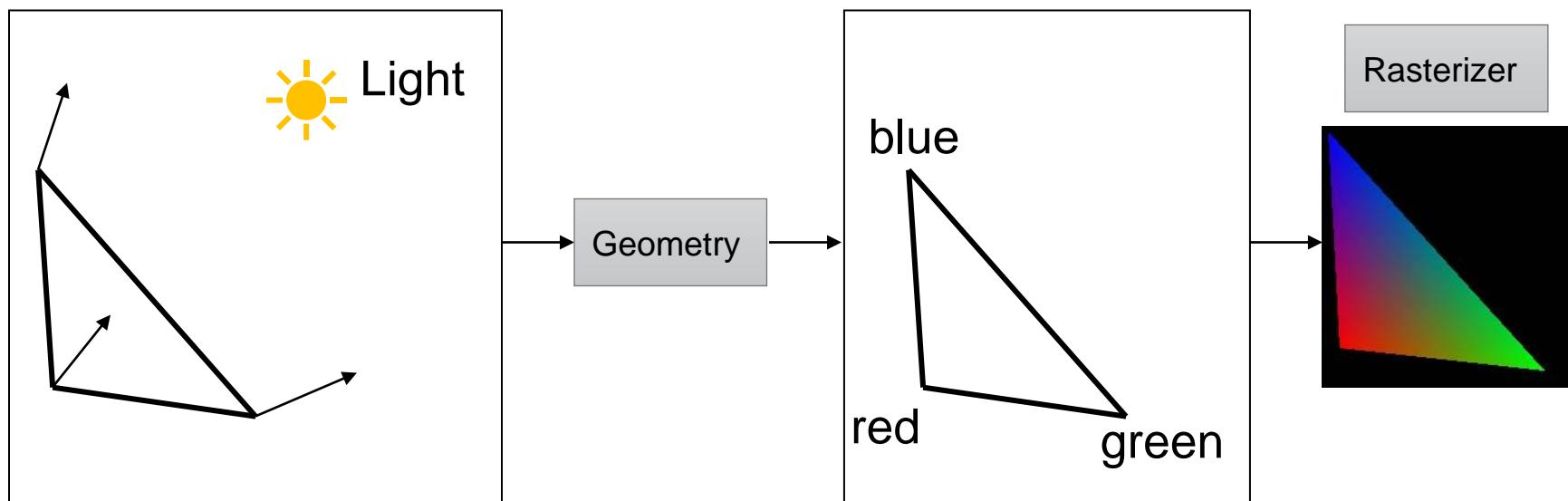


Shading

Compute lighting at vertices from

- Light sources
- Material properties
- Geometrical relationships

Then interpolate over triangle



Shading vs. Lighting

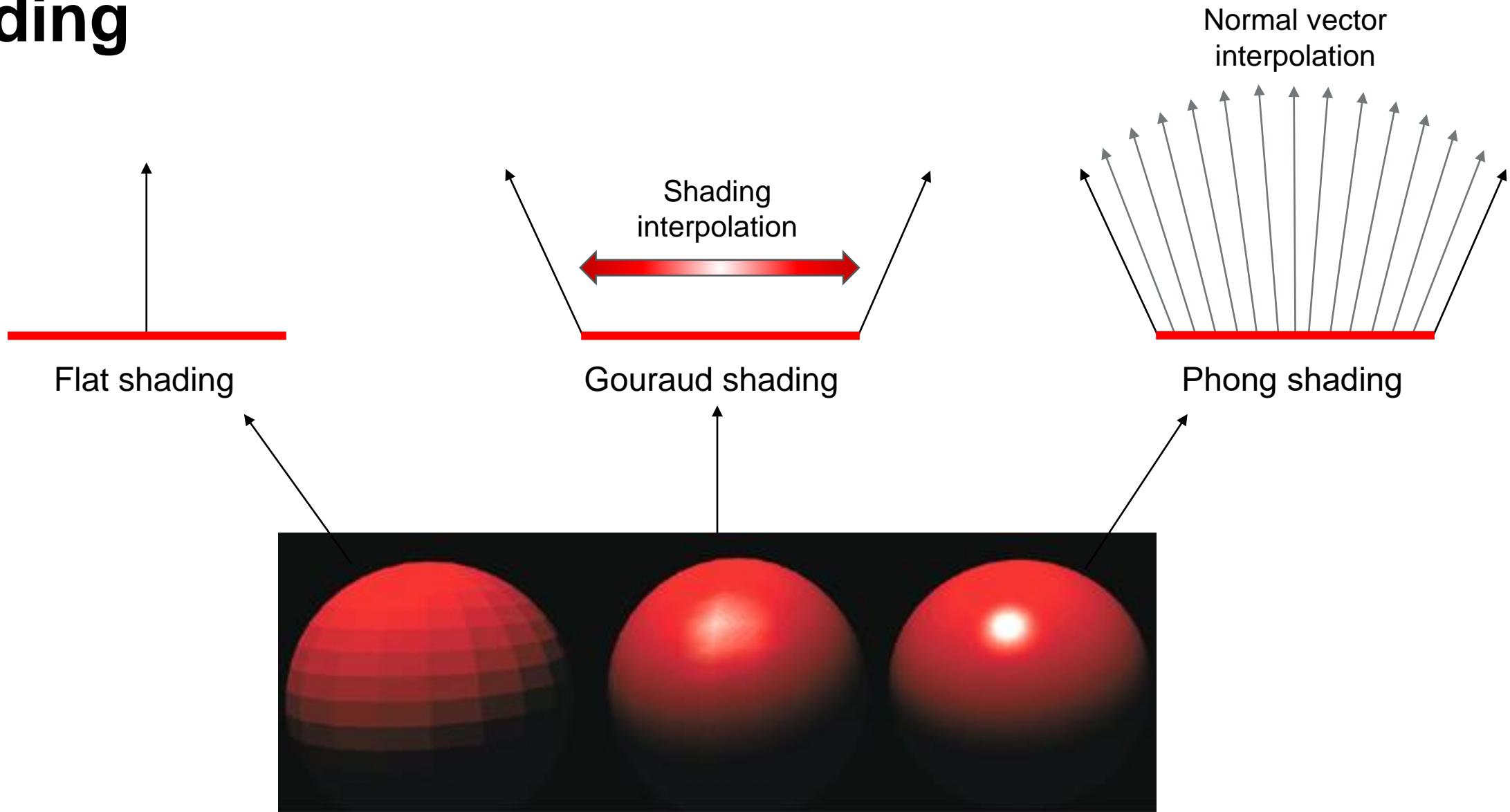
Lighting: The interaction between light and matter

Shading: Do lighting (at vertices) and determine pixel's colours from these

Three types of shading:

- Flat
- Gouraud
- Phong

Shading



Light Sources

Different types

- **Directional Light (HW support)**
- **Point Light (HW support)**
- **Spot Light (HW support)**
- **Area Light (no HW support, soft shadows)**

Surface Colour & Material

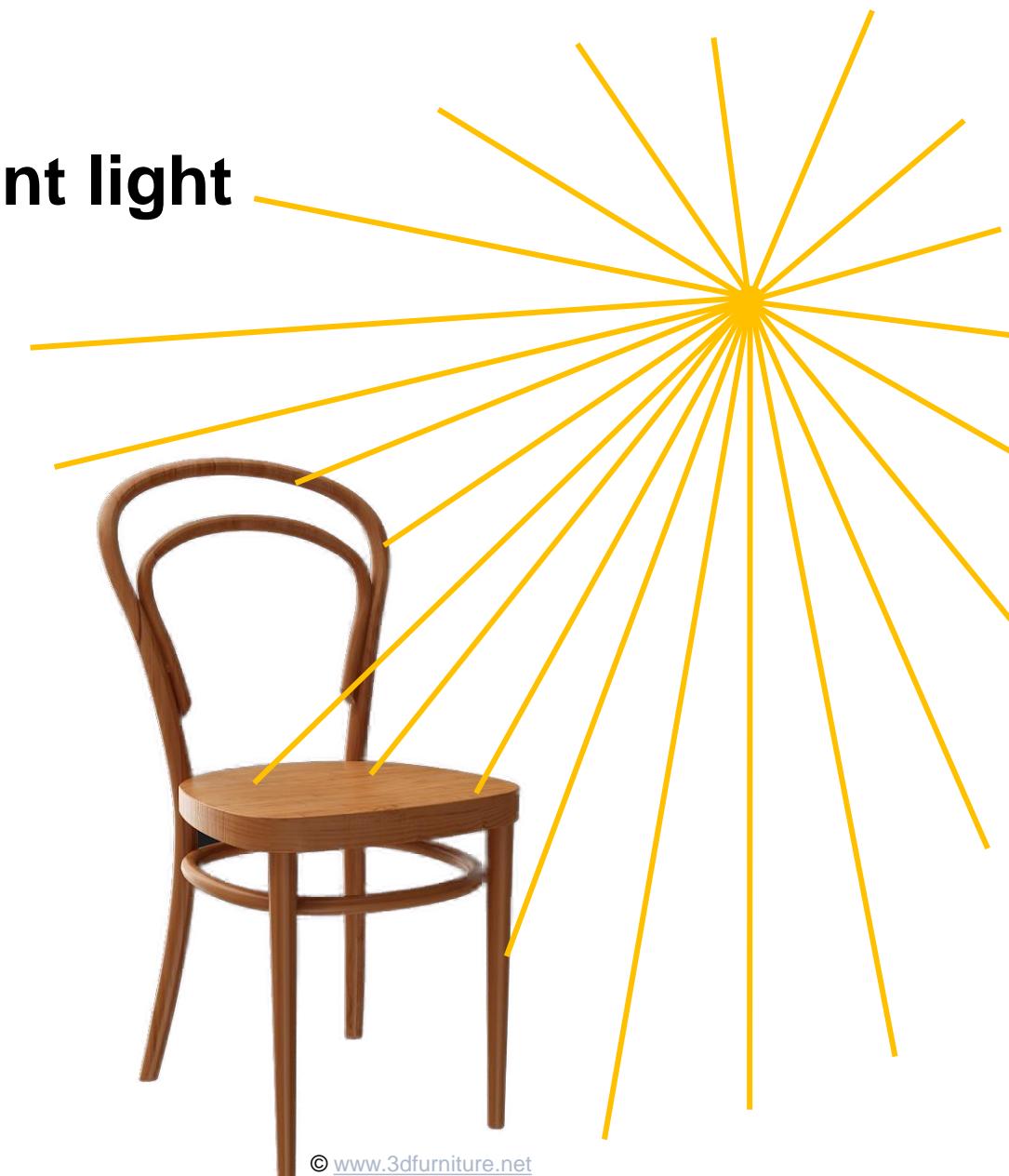
- **How light is reflected from a surface**

Light Sources

Directional light, e.g. sun

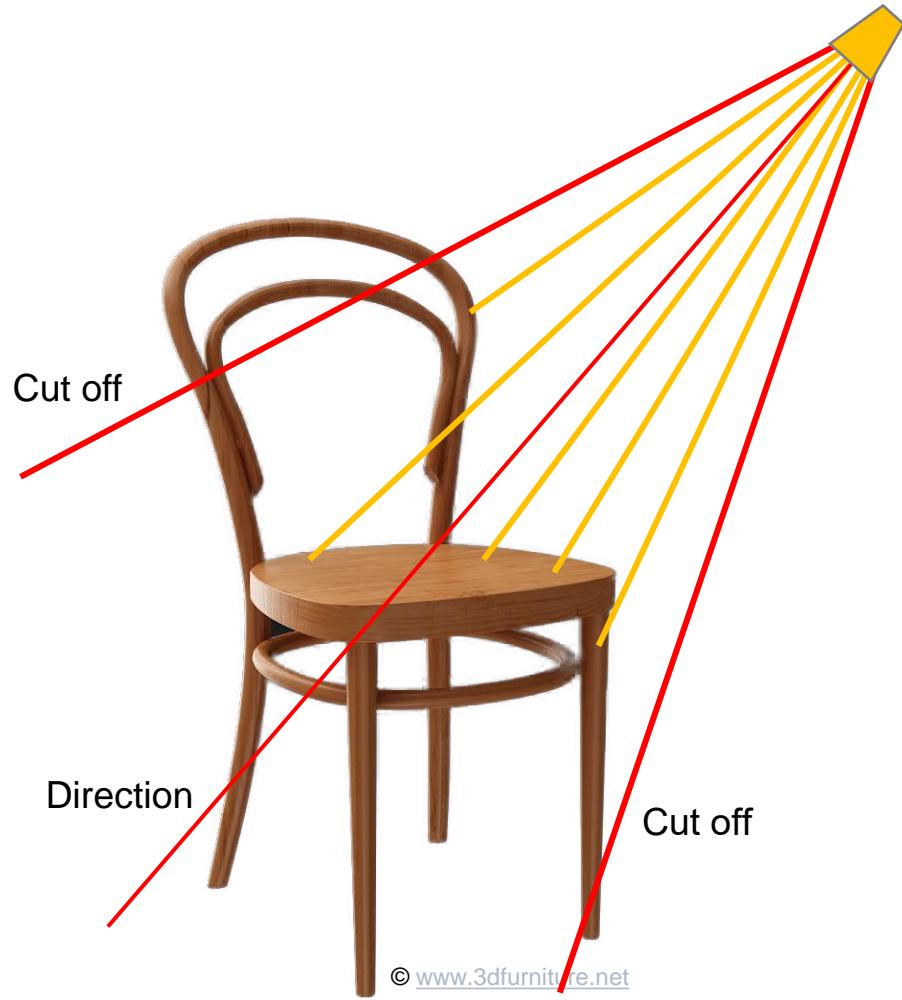


Point light



Light Sources

Spot light



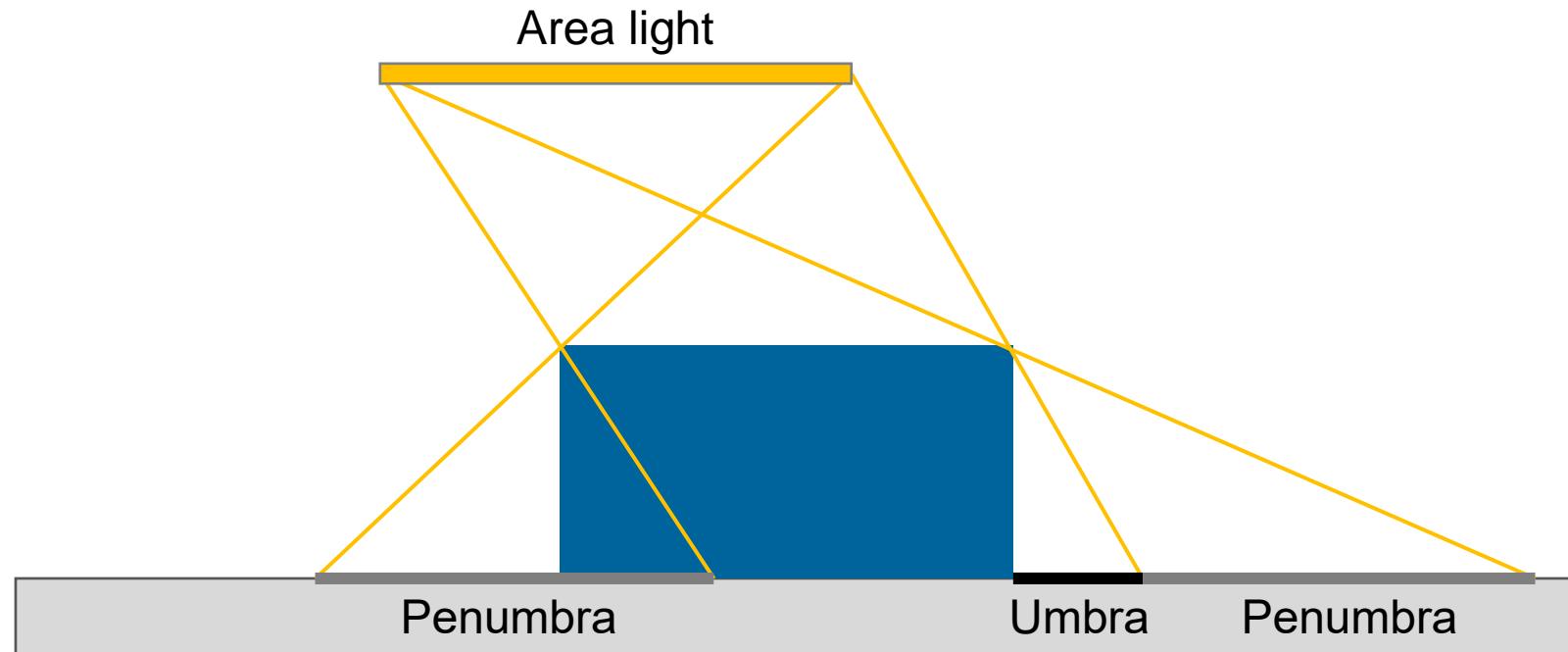
Area light



Light Sources

Area lights

- No direct support by current hardware
- But possible to simulate effect with special methods
 - (see chapter 7 “Shadows” of the Real-Time Rendering book for examples)



Light Sources

OpenGL Properties

s_{amb}

ambient intensity (colour)

crude approximation for indirect illumination

s_{diff}

diffuse intensity (colour)

light colour used for diffuse reflections

s_{spec}

specular intensity (colour)

light colour used for specular reflections

s_{pos}

light source position

A real light source only has one colour

Material

OpenGL Material Properties

m_{amb} **ambient** material colour

m_{diff} **diffuse** material colour

m_{spec} **specular** material colour

m_{shi} **shininess** parameter (scalar)

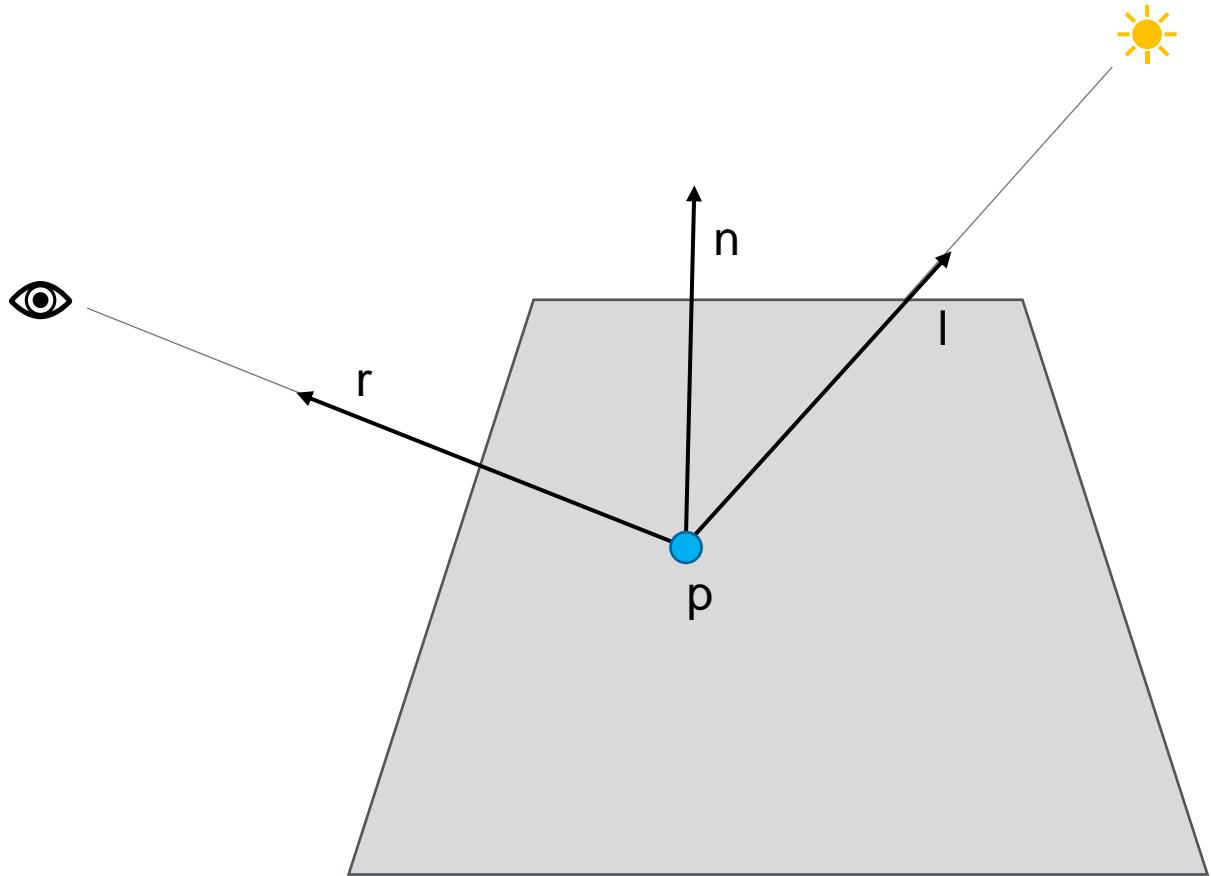
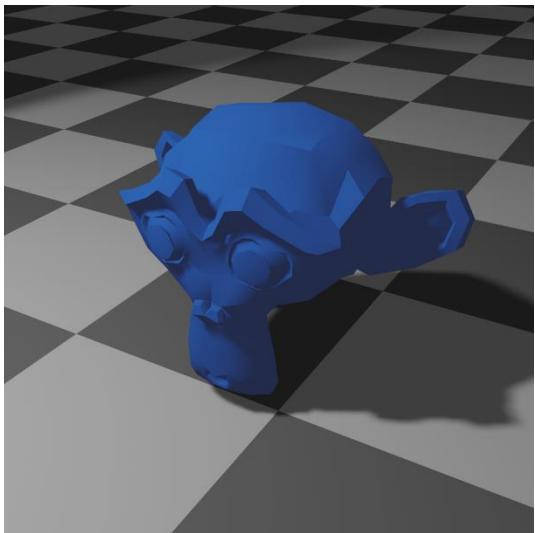
m_{emi} **emissive** material colour

Diffuse Lighting

Lambert's law

$$i_{\text{diff}} = \mathbf{n} \cdot \mathbf{l} = \cos \phi$$

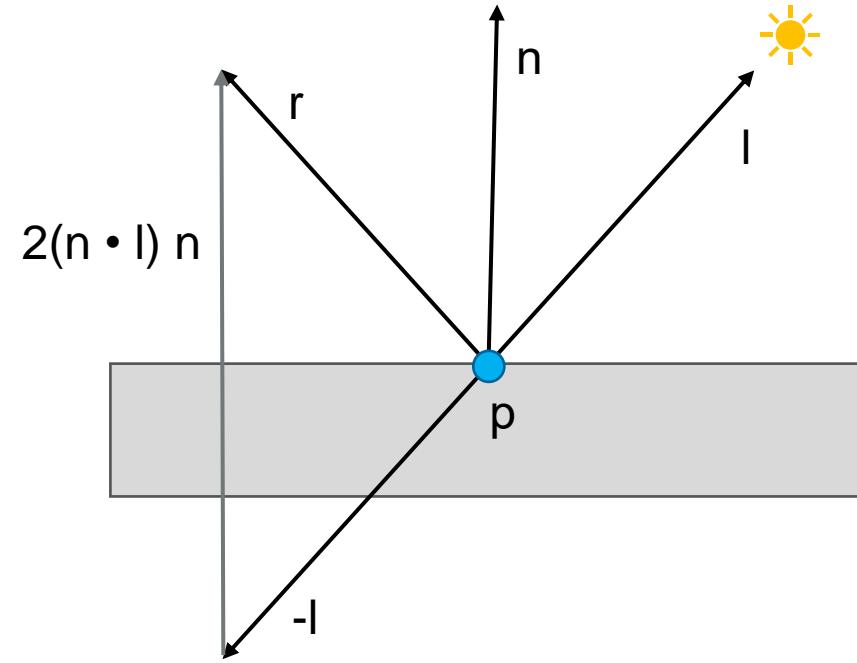
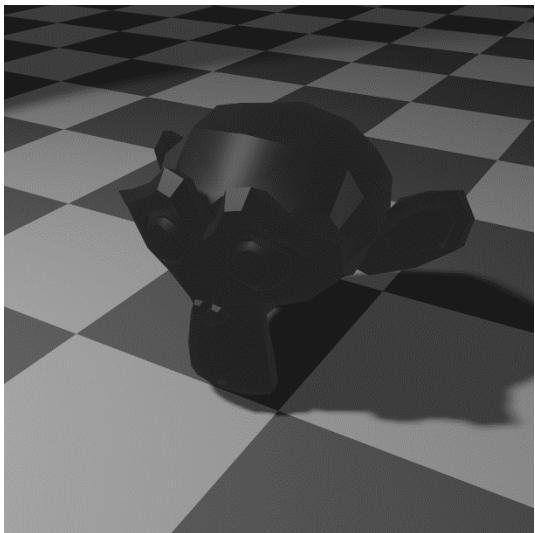
$$i_{\text{diff}} = \max(i_{\text{diff}}, 0) \mathbf{m}_{\text{diff}} \otimes \mathbf{s}_{\text{diff}}$$



Specular Lighting

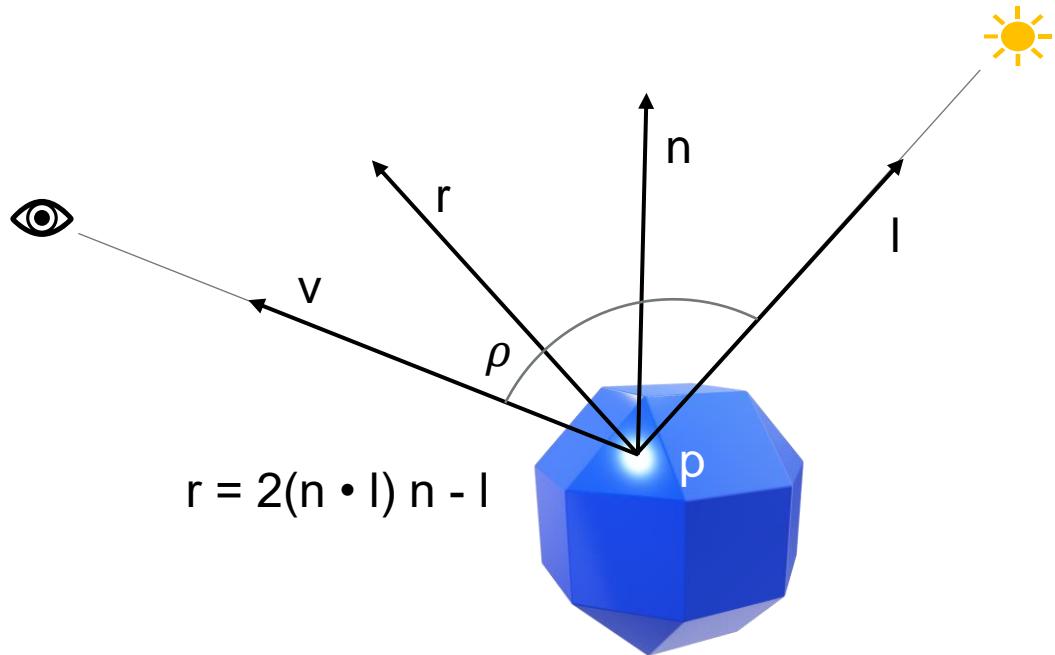
How to calculate reflection vectors

$$\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{l}) \mathbf{n} - \mathbf{l}$$



Specular Lighting

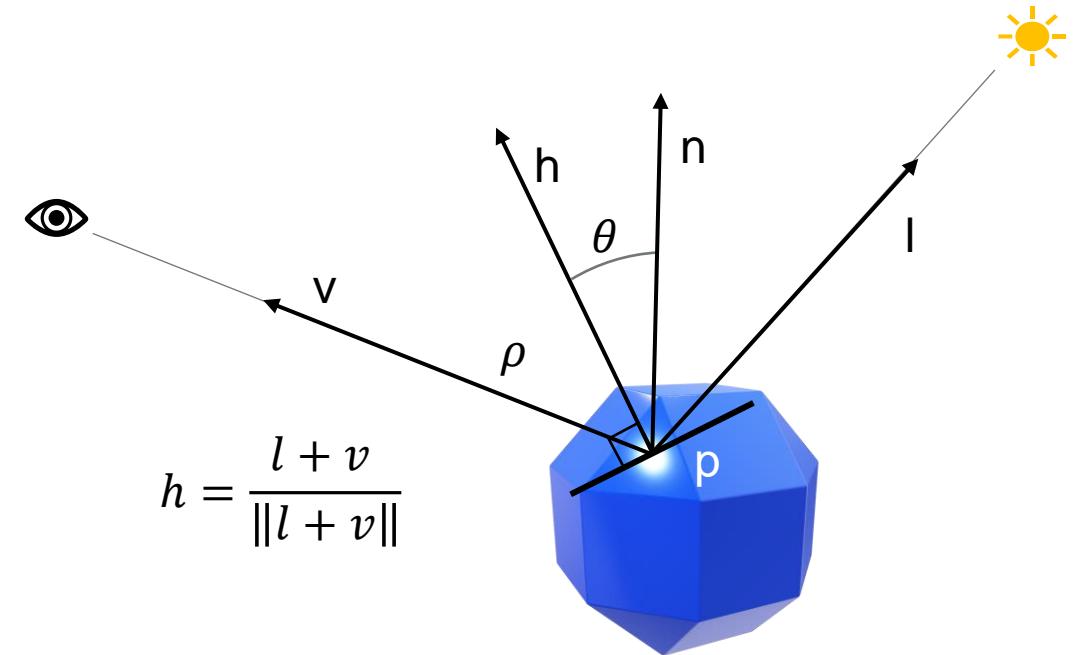
Phong Lighting Equation



$$i_{\text{spec}} = (\mathbf{r} \cdot \mathbf{v})^{m_{\text{shi}}} = (\cos \rho)^{m_{\text{shi}}}$$

$$i_{\text{spec}} = \max(i_{\text{spec}}, 0) \mathbf{m}_{\text{spec}} \otimes \mathbf{s}_{\text{spec}}$$

Blinn Lighting Equation

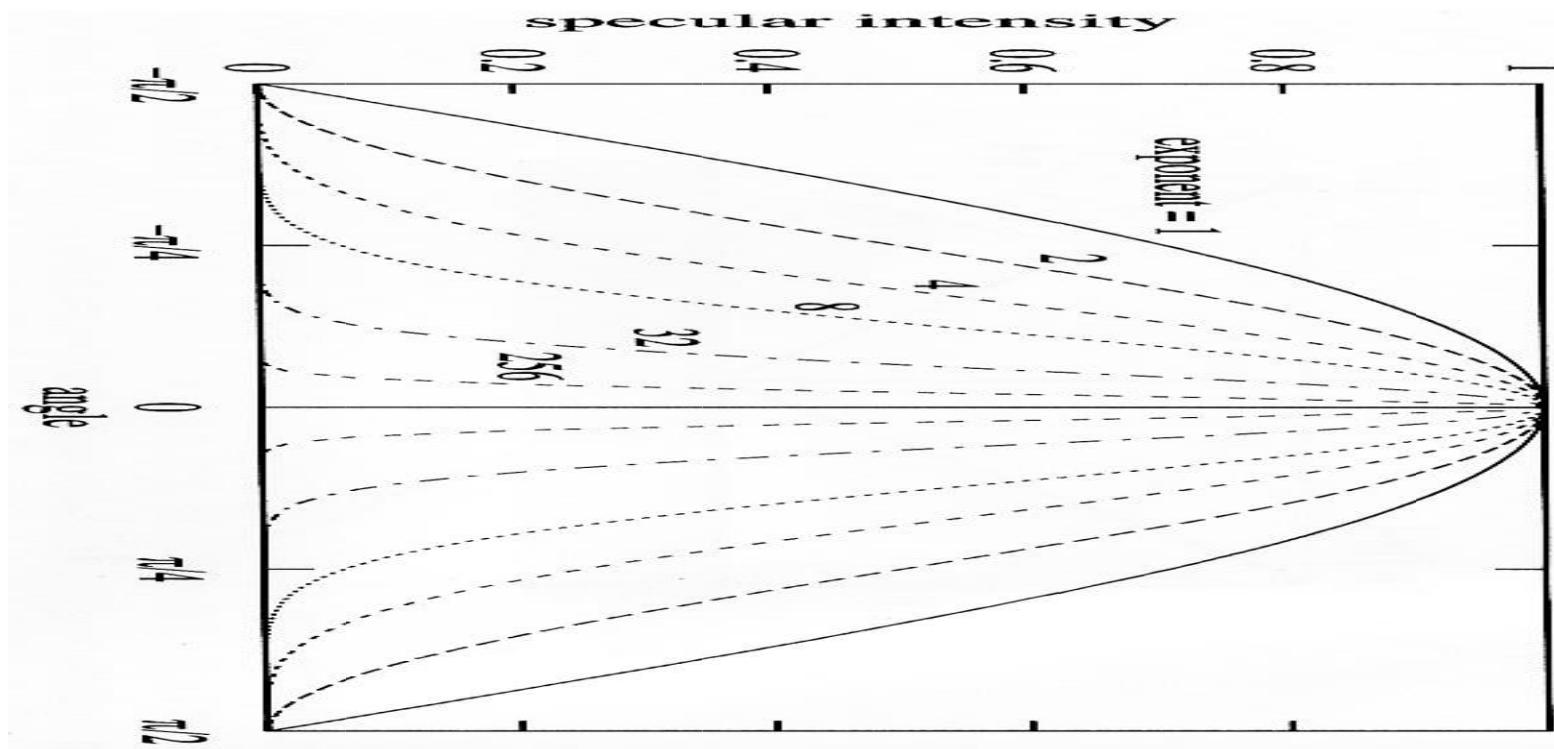


$$i_{\text{spec}} = (\mathbf{n} \cdot \mathbf{h})^{m_{\text{shi}}} = (\cos \theta)^{m_{\text{shi}}}$$

Specular Lighting

Specular intensity

- For $m_{shi} = 1$ a cosine curve is produced
- Isotropic (“symmetric”, independent of direction)

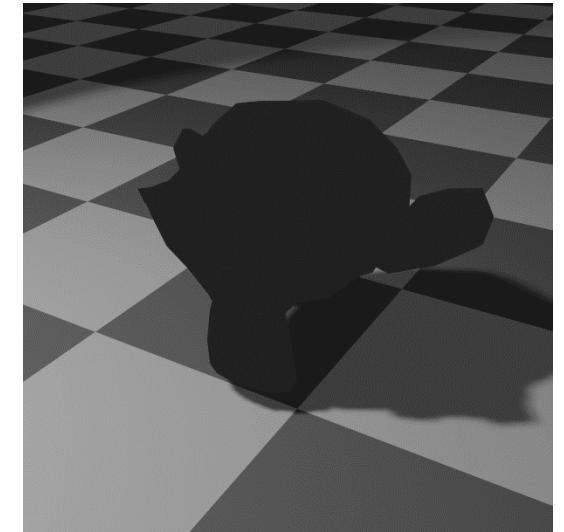


Ambient Lighting

Constant term to simulate indirect lighting

Unlit surfaces appear not completely black

$$i_{\text{amb}} = m_{\text{amb}} \otimes s_{\text{amb}}$$



NOT physically correct!!!

Just a (very) crude approximation

Scenes with too high an ambient value look

- washed-out
- low contrast

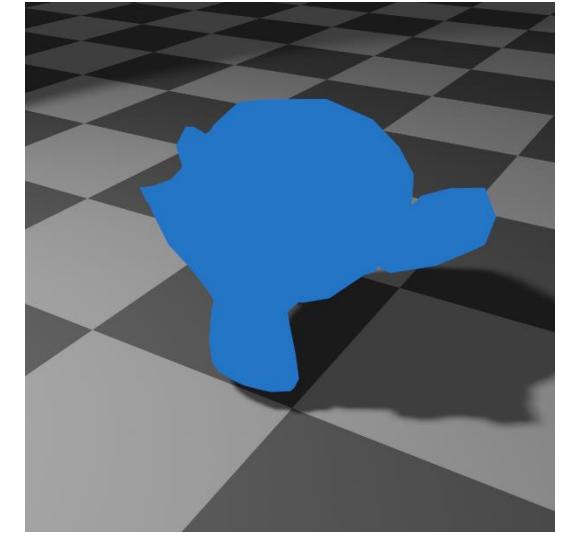
Emissive Lighting

Constant term, to simulate light emitted by object (material) itself

- Not physically correct

Unlit surfaces not completely black

Independent of any light source



$$\mathbf{i}_{\text{emi}} = \mathbf{m}_{\text{emi}}$$

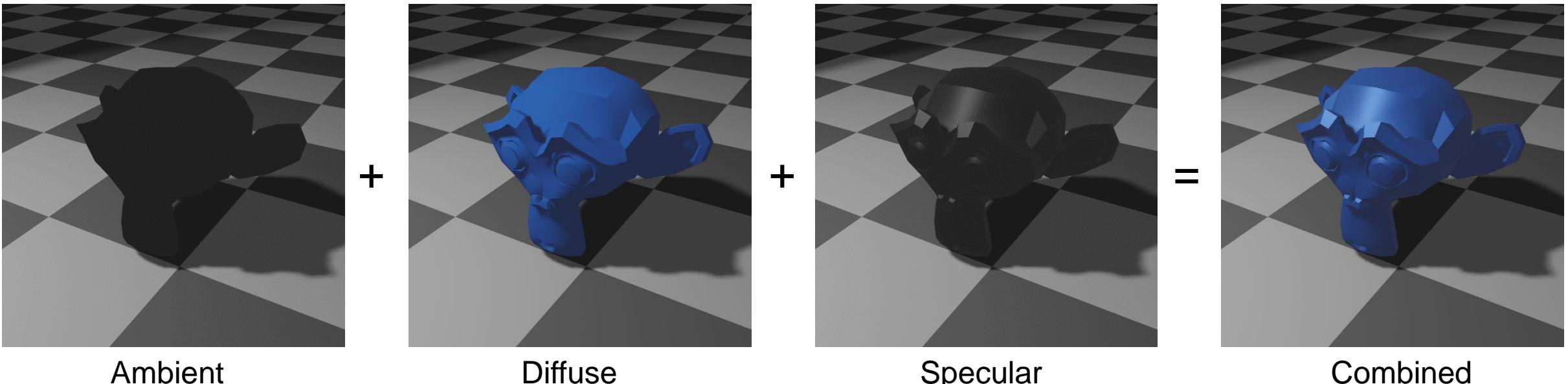
Lighting Equation

Total intensity is sum of ambient, diffuse, and specular intensity

$$i_{\text{total}} = i_{\text{amb}} + i_{\text{diff}} + i_{\text{spec}}$$

Ad-hoc approximation

- Looks ok, but has no physical meaning
- See Global Illumination for the real thing



Additional Effects

Distance Attenuation (d)

$$i_{\text{total}} = i_{\text{amb}} + d(i_{\text{diff}} + i_{\text{spec}})$$

Spotlight (c_{spot})

$$i_{\text{total}} = c_{\text{spot}} (i_{\text{amb}} + d(i_{\text{diff}} + i_{\text{spec}}))$$

Global Ambient

$$i_{\text{total}} = a_{\text{glob}} \otimes m_{\text{glob}} + m_{\text{emi}} + c_{\text{spot}} (i_{\text{amb}} + d(i_{\text{diff}} + i_{\text{spec}}))$$

Fog

Simple atmospheric effect

- A little better realism
- Help in determining distances

Colour of fog: c_f

Colour of surface: c_s

$$c_p = f c_s + (1 - f) c_f \quad f \in [0, 1]$$

How to compute f ?

Three ways:

- Linear
- Exponential
- Exponential-squared

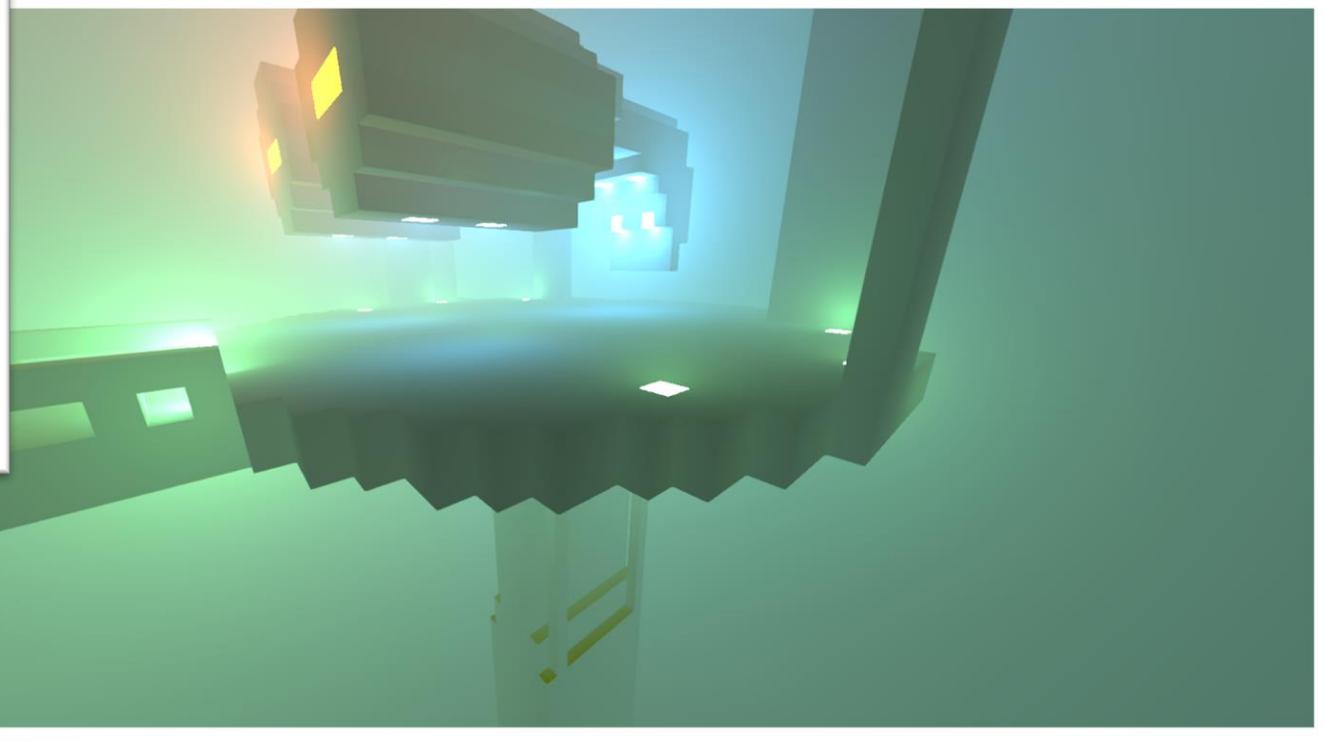
—————

$$f = \frac{z_{\text{end}} - z_p}{z_{\text{end}} - z_{\text{start}}}$$

Fog



From [GitHub](#) by AivanF



© Isaac Dykeman

Transparency

Very simple in real-time contexts

The tool: Alpha blending (mix two colours)

Alpha (α) is another component in the frame buffer, or on triangle

- Represents the opacity
- 1.0 is totally opaque
- 0.0 is totally transparent

The over operator: $c_o = \alpha c_s + (1 - \alpha) c_d$

Transparency

Need to sort the transparent objects

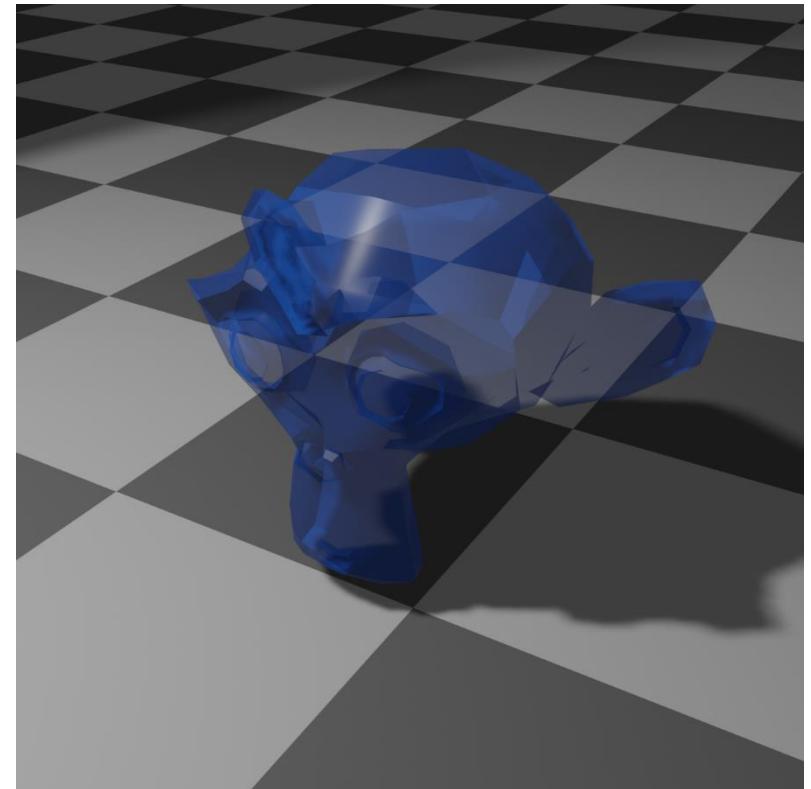
- Render back to front

Lots of different other blending modes

Can store $\text{RGB}\alpha$ in textures as well

Two ways:

- Unmultiplied
- Pre-multiplied



Multiple Light Sources

Sum of single contributions (n light sources)

$$\begin{aligned} \mathbf{i}_{\text{total}} &= \mathbf{a}_{\text{glob}} \otimes \mathbf{m}_{\text{glob}} + \mathbf{m}_{\text{emi}} + \\ &+ \sum_{k=1}^n c_{k,\text{spot}} (\mathbf{i}_{k,\text{amb}} + d_k (\mathbf{i}_{k,\text{diff}} + \mathbf{i}_{k,\text{spec}})) \end{aligned}$$

Multiple Light Sources

Sum of n light sources might exceed range [0,1]

Clamp $\begin{pmatrix} 2.5 \\ 1.5 \\ 0.5 \end{pmatrix} \Rightarrow \begin{pmatrix} 1.0 \\ 1.0 \\ 0.5 \end{pmatrix}$

Scale $\begin{pmatrix} 2.5 \\ 1.5 \\ 0.5 \end{pmatrix} \Rightarrow \begin{pmatrix} \frac{2.5}{2.5} \\ \frac{1.5}{2.5} \\ \frac{0.5}{2.5} \end{pmatrix} = \begin{pmatrix} 1.0 \\ 0.6 \\ 0.2 \end{pmatrix}$

High Dynamic Range

Luminance Levels (Example)

The HVS can adapt to lighting conditions that vary by nearly 10 orders of magnitude

Within a scene, the HVS functions over a range of about 5 orders of magnitude simultaneously (**HDR – high dynamic range**)

Displays can reproduce about 2 orders of magnitude of intensity variation
(**LDR – low dynamic range**)

Examples	Illumination in cd/m ²
Starlight	10^{-3}
Moonlight	10^{-1}
Indoor lighting	10^2
Sunlight	10^5
LCD Monitor	$10^2 - 10^3$

CIE xy Chromaticity Diagram

CIE XYZ colour space

- based on direct measurements and
- experiments on human observers as basis from which most other colour spaces are defined
- tristimulus value XYZ to describe a colour sensation
- 3-dim colour space

Each colour has 2 parts

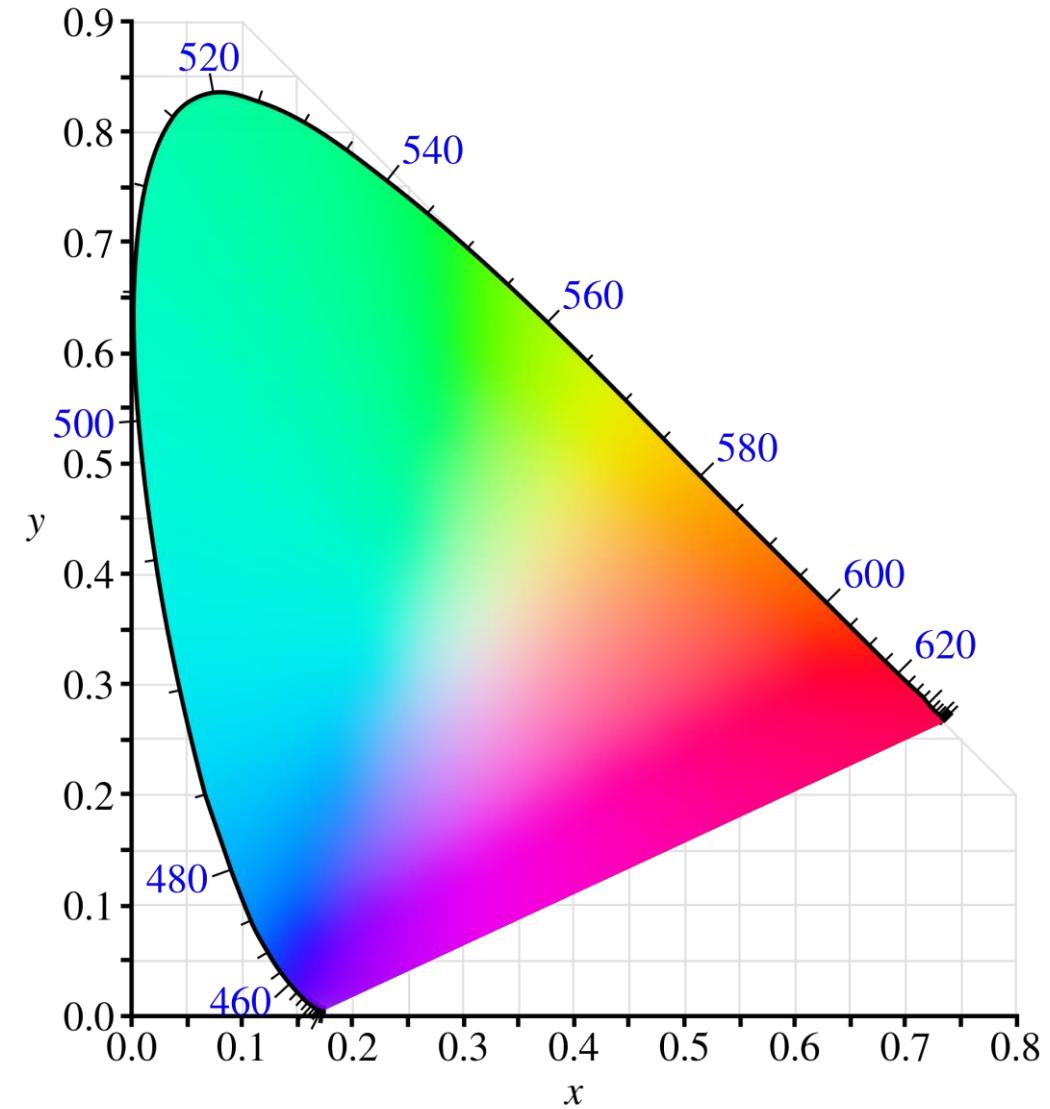
- chromaticity & brightness
- e.g. white vs. grey

Y defines brightness (by design)

Chromaticity is defined by x and y

- $x = X / (X + Y + Z)$
- $y = Y / (X + Y + Z)$

CIE: Commission Internationale de l'Eclairage



sRGB Colour Space

The standard defines 3 primaries using CIE xy space
red, green, blue

A non-linear mapping between values and the intensity of the primaries is defined to match a „typical“ LCD monitor

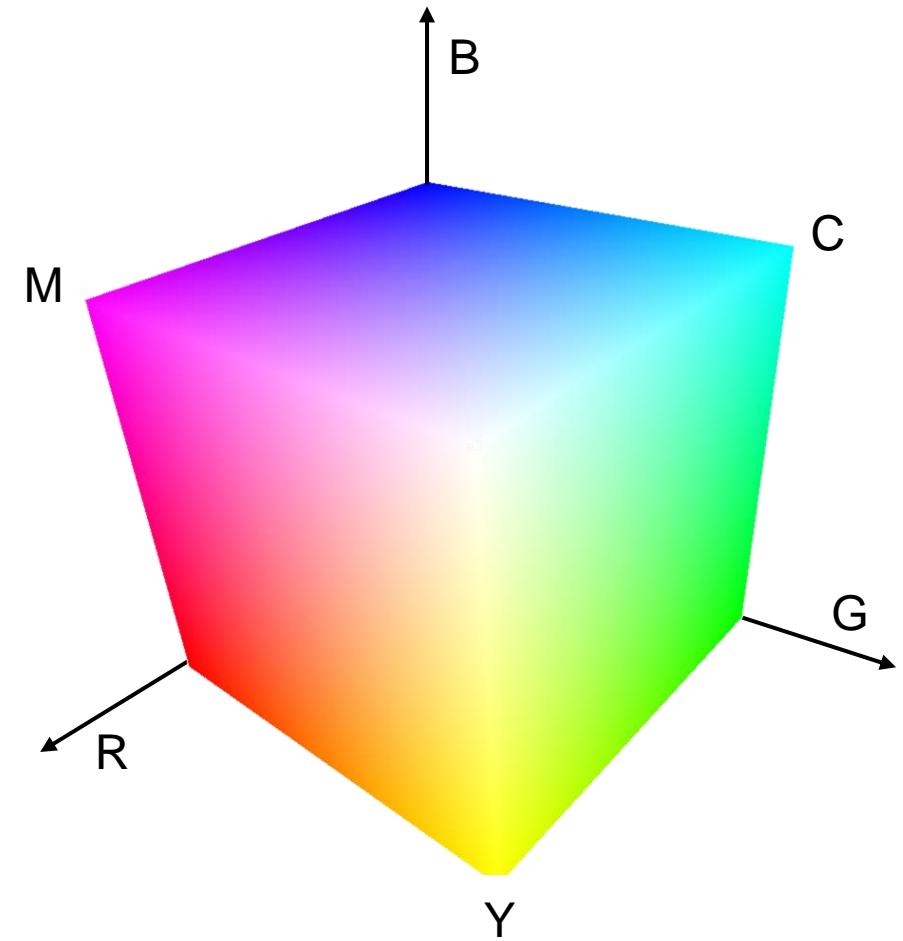
Values are defined in range [0.0, 1.0]

→ Low Dynamic Range (LDR)

As opposed to reality where no maximum intensity exists → range $[0, \infty]$

→ High Dynamic Range (HDR)

From dim starlight up to supernova



sRGB Primaries in CIE xy

[0.6400, 0.3300]

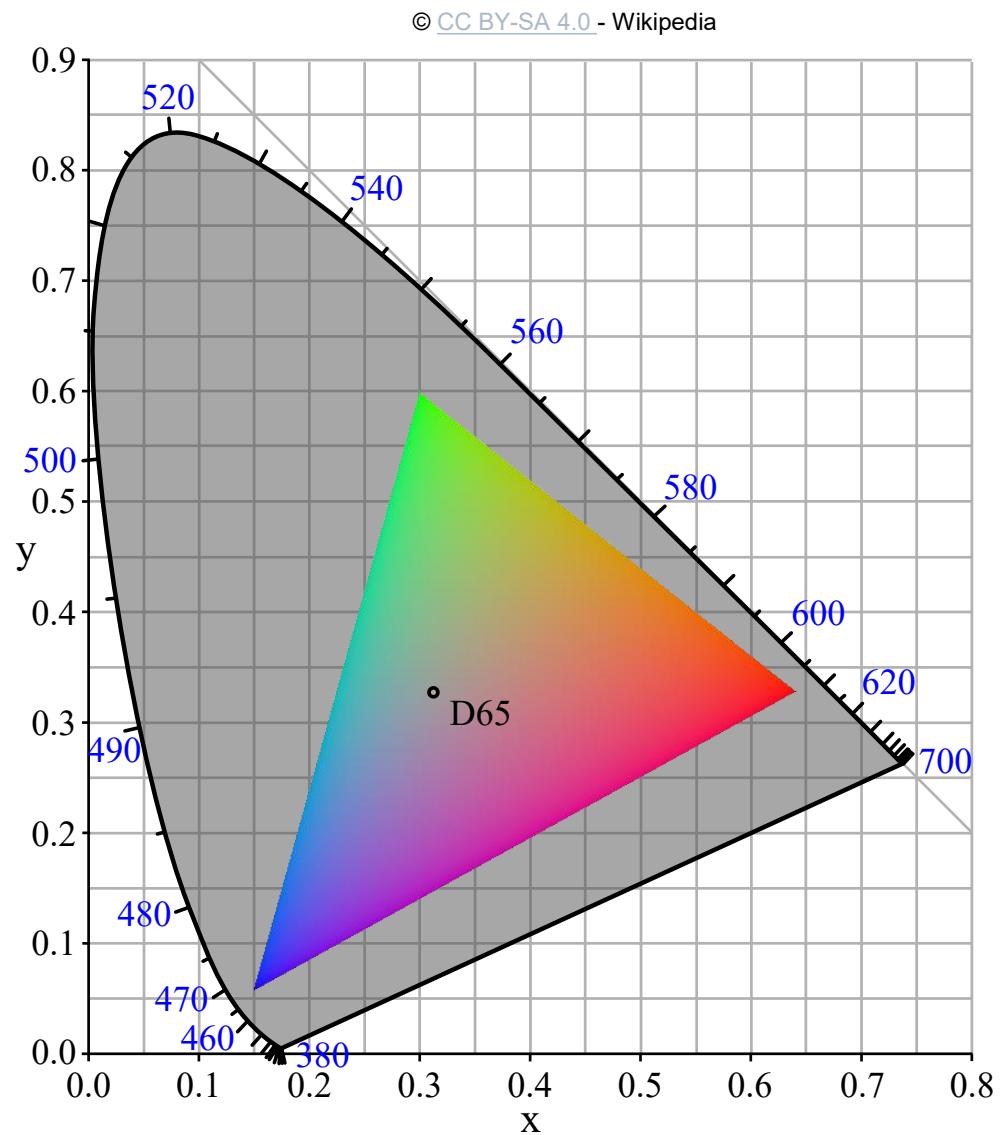
[0.3000, 0.6000]

[0.1500, 0.0600]

sRGB gamut

- Triangle defined by primaries
- Colours outside can not be specified

White point, D65 [0.3127, 0.3290]



HDR – High Dynamic Range

The current trend in CG goes towards

- Capturing and storing images and colour data in HDR formats
- Manipulating images and colour data using data types capable of faithfully representing high dynamic range
- Mapping data to LDR only in the last step to be able to display (or print) it

Mapping high dynamic range values to a displayable range is called **tone mapping**

HDR File Formats

.hdr

- **Encodings: RGBE, XYZE**
- **File format of Radiance renderer, open source**

.tiff

- **Encodings: IEEE RGB, LogLuv24, LogLuv32**
- **Public domain library (libtiff)**

.exr

- **Encodings: Half RGB**
- **Open source library (OpenEXR)**

Tone Mapping

Tone Mapping

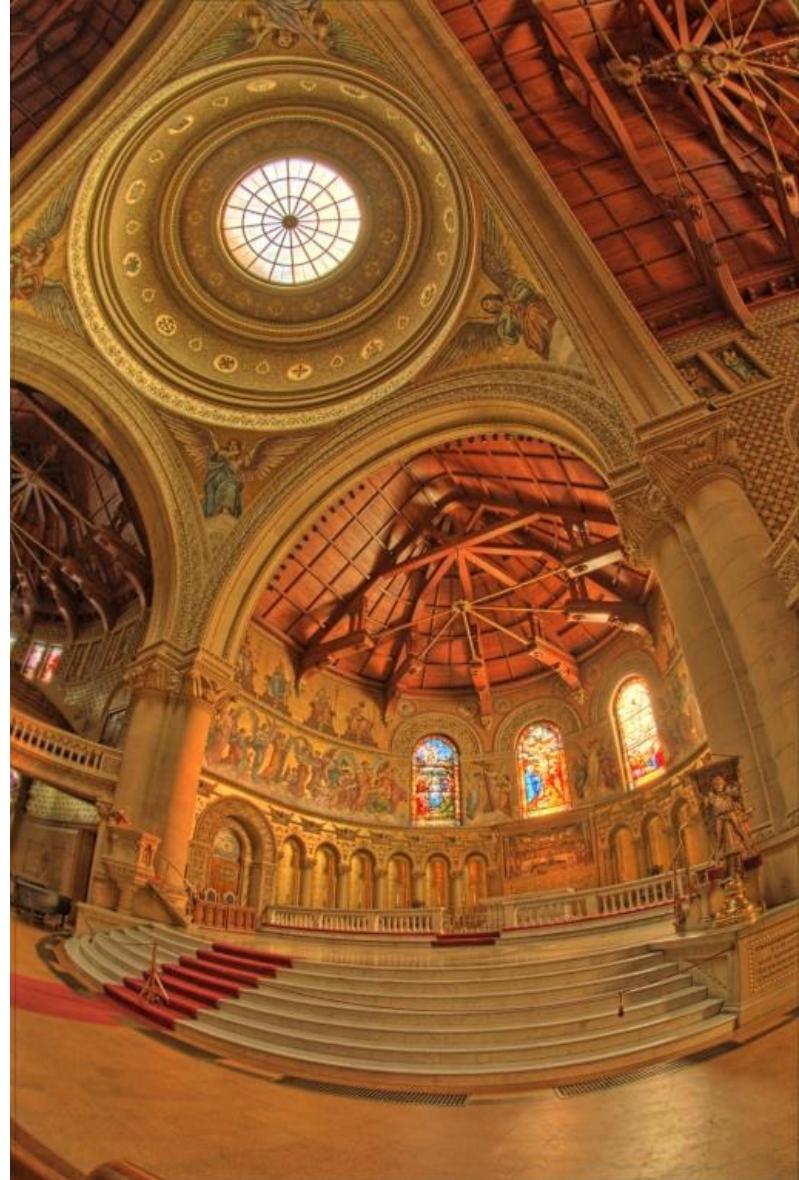
Task: Map intensities from range $[0, \infty]$ to range $[0,1]$

Different possible mapping algorithms

- Global vs local approaches

Either performed in SW as a pre-processing step

Or shader hardware for real-time tone mapping



From Gastal E.S.L. and Oliveira M.M.: [Domain Transform for Edge-Aware Image and Video Processing](#).
ACM Transactions on Graphics, Volume 30 (2011), Number 4,
Proceedings of SIGGRAPH 2011, Article 69.

Tone Mapping

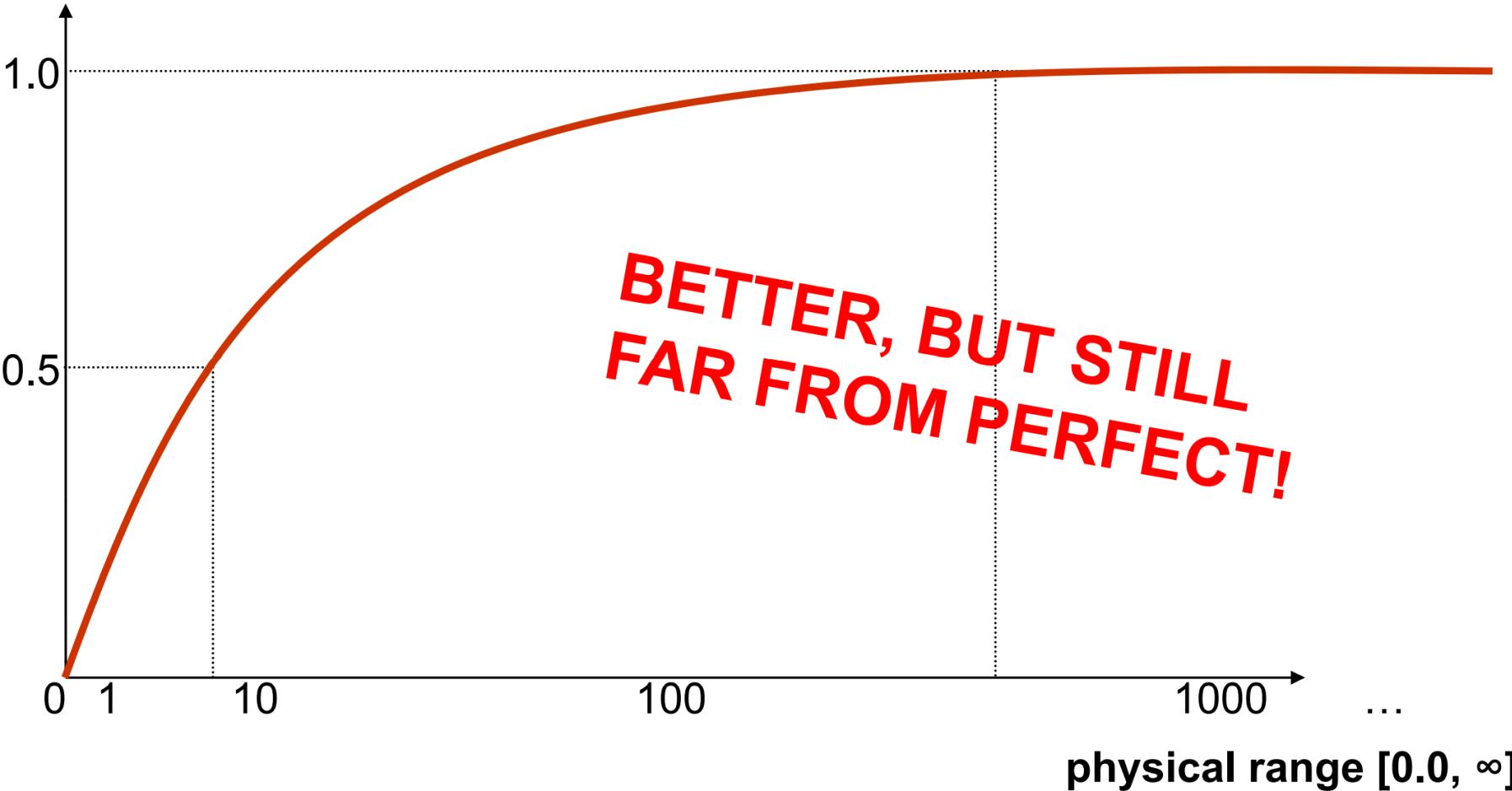
Example: Global Linear Mapping



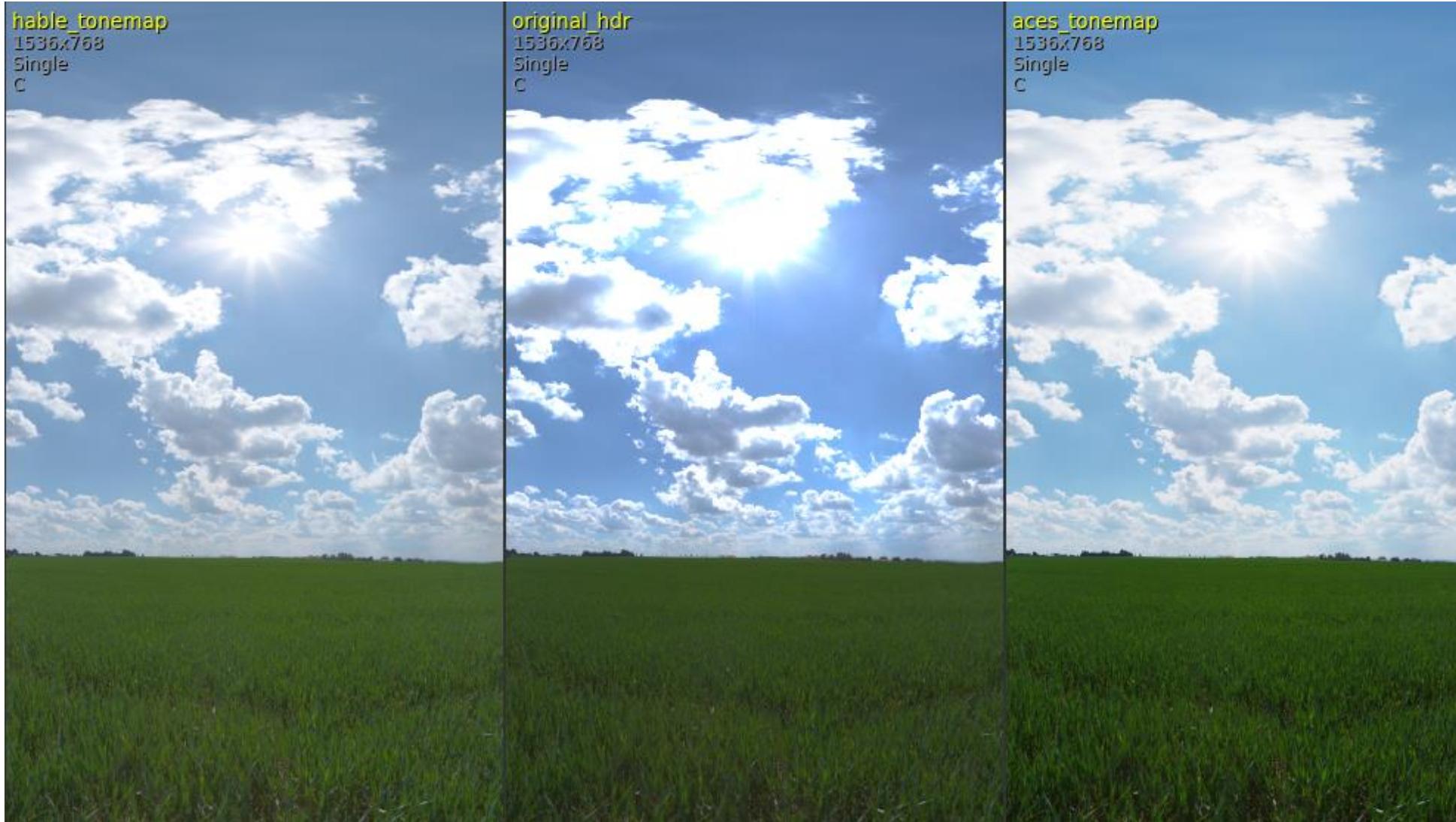
Tone Mapping

Example: Exponential Mapping

display range $[0.0, 1.0]$



Tone Mapping Example

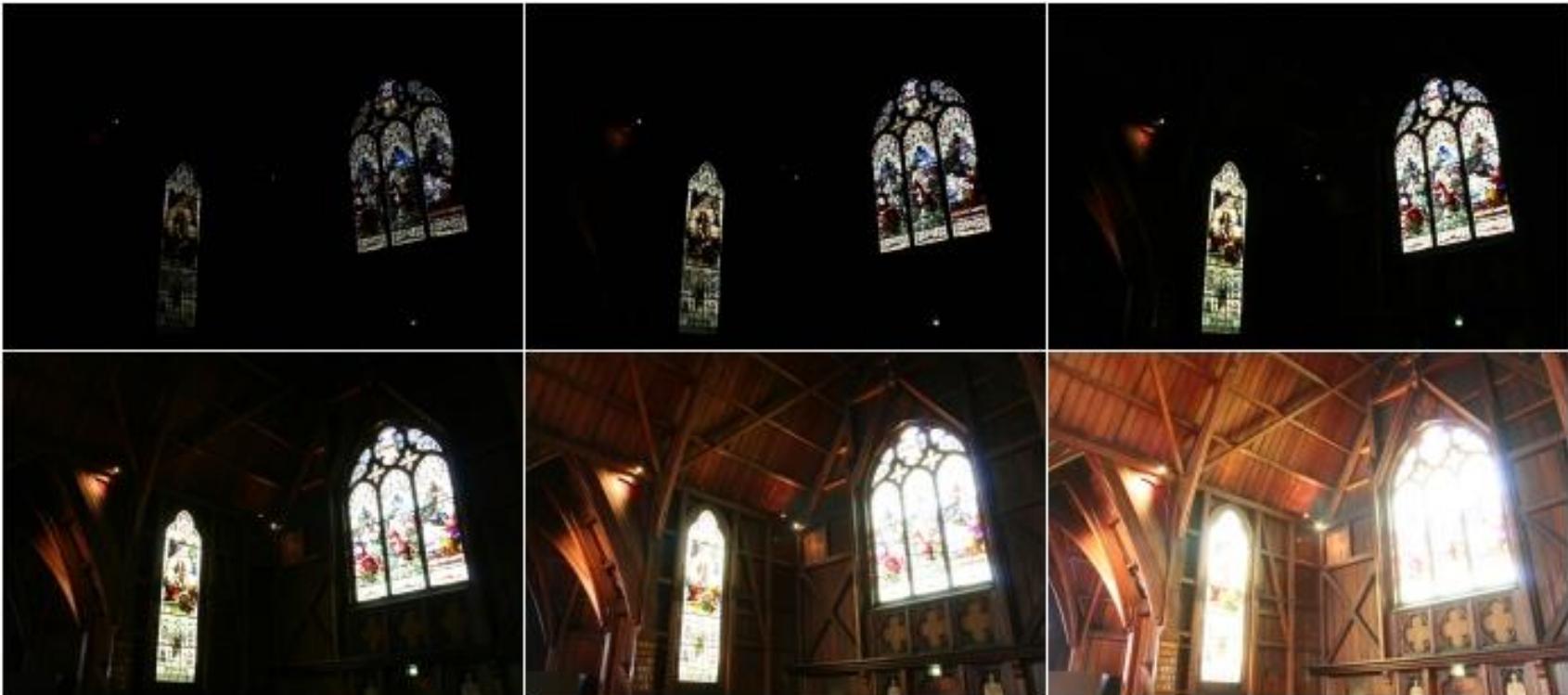


© Oskar Swierad

Tone Mapping Example

Simulate adaption of human visual system, by introducing some delay in adaption of tone mapping parameters

- Over-exposure (when moving from dark to bright environment)
- Under-exposure (when moving from bright to dark environment)



© CC BY-SA 3.0 Dean S. Pemberton

Tone Mapping Example

Combination of the previous six exposures



Tone Mapping Example

Light
blooming
effect



Tone Mapping Example

Light
blooming
effect

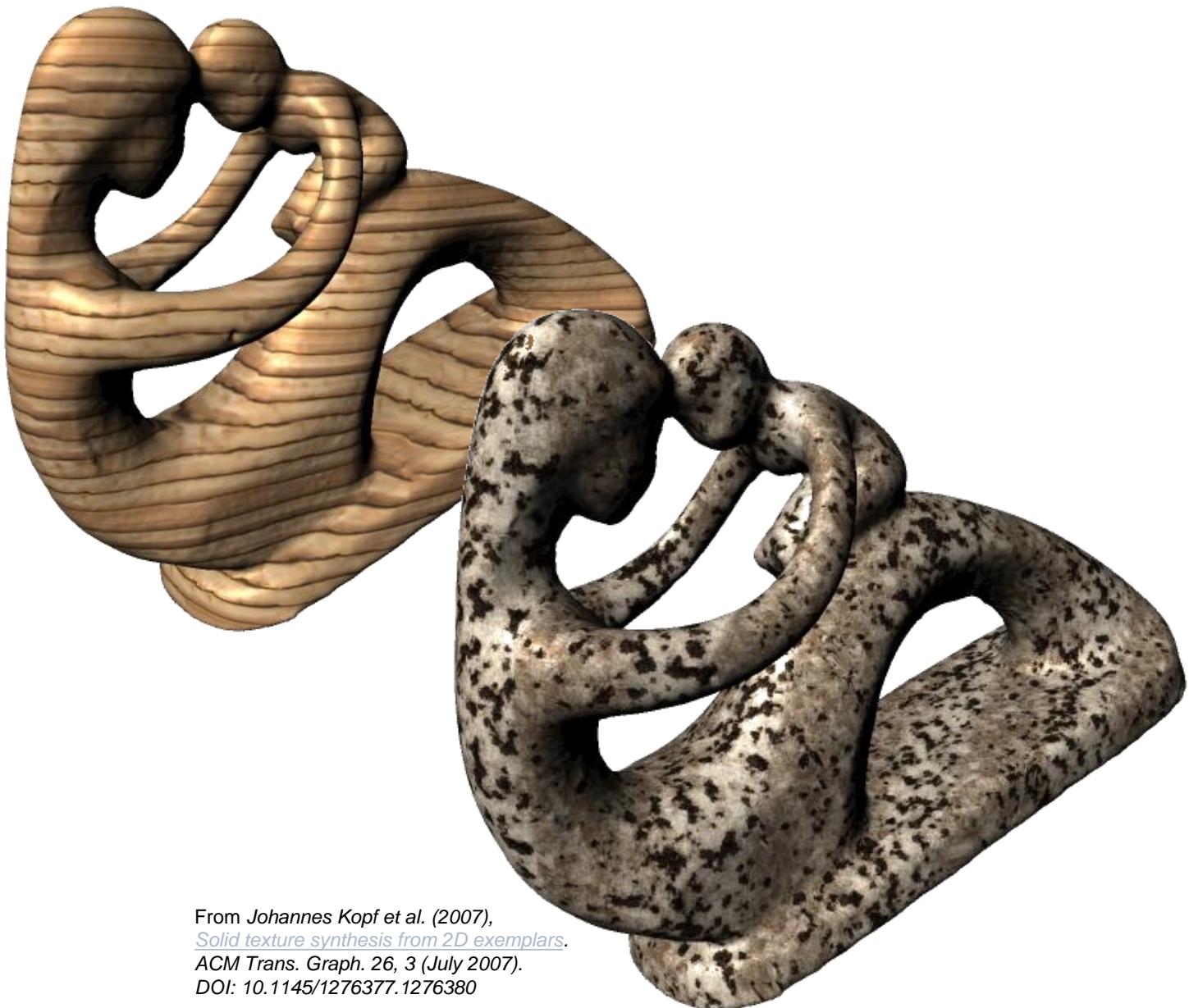




Textures

Overview

- Motivation
- Texture types
- Texture mapping
- Blending and filtering
- Textures in OpenGL



From Johannes Kopf et al. (2007),
Solid texture synthesis from 2D exemplars,
ACM Trans. Graph. 26, 3 (July 2007).
DOI: 10.1145/1276377.1276380

Texture Basics

Motivation

Greater realism

- Real objects are often multi-coloured
- They are not always smooth and regular
- Texturing adds details and surface structure

Save resources

- Imaging rendering a brick wall
- Draw every single brick?
→ That's a lot of polygons!
- Instead glue an image of a brick wall to one large polygon and draw this



Texture © [Miss Chatz](#)



What is a Texture?

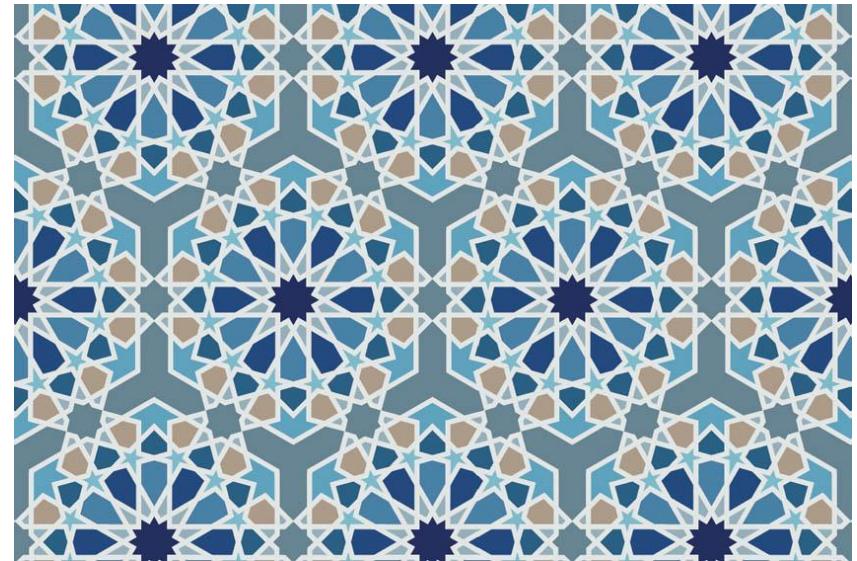
In most cases a texture is an image that is mapped to a mesh

More generally, textures are arrays of data

- Can be 1D, 2D or 3D
- For example colour data (or alpha values or ...)

A single element in a texture array is called **texel**

Whereas a pixel is one element of the frame buffer

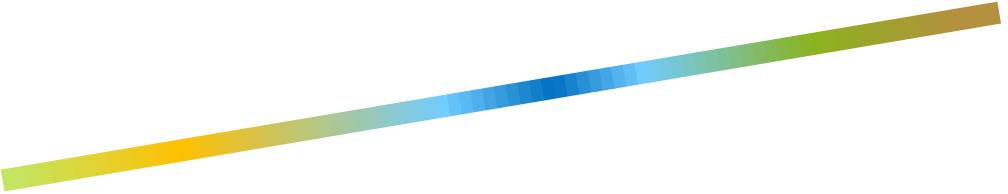


Texture © Miss Chatz

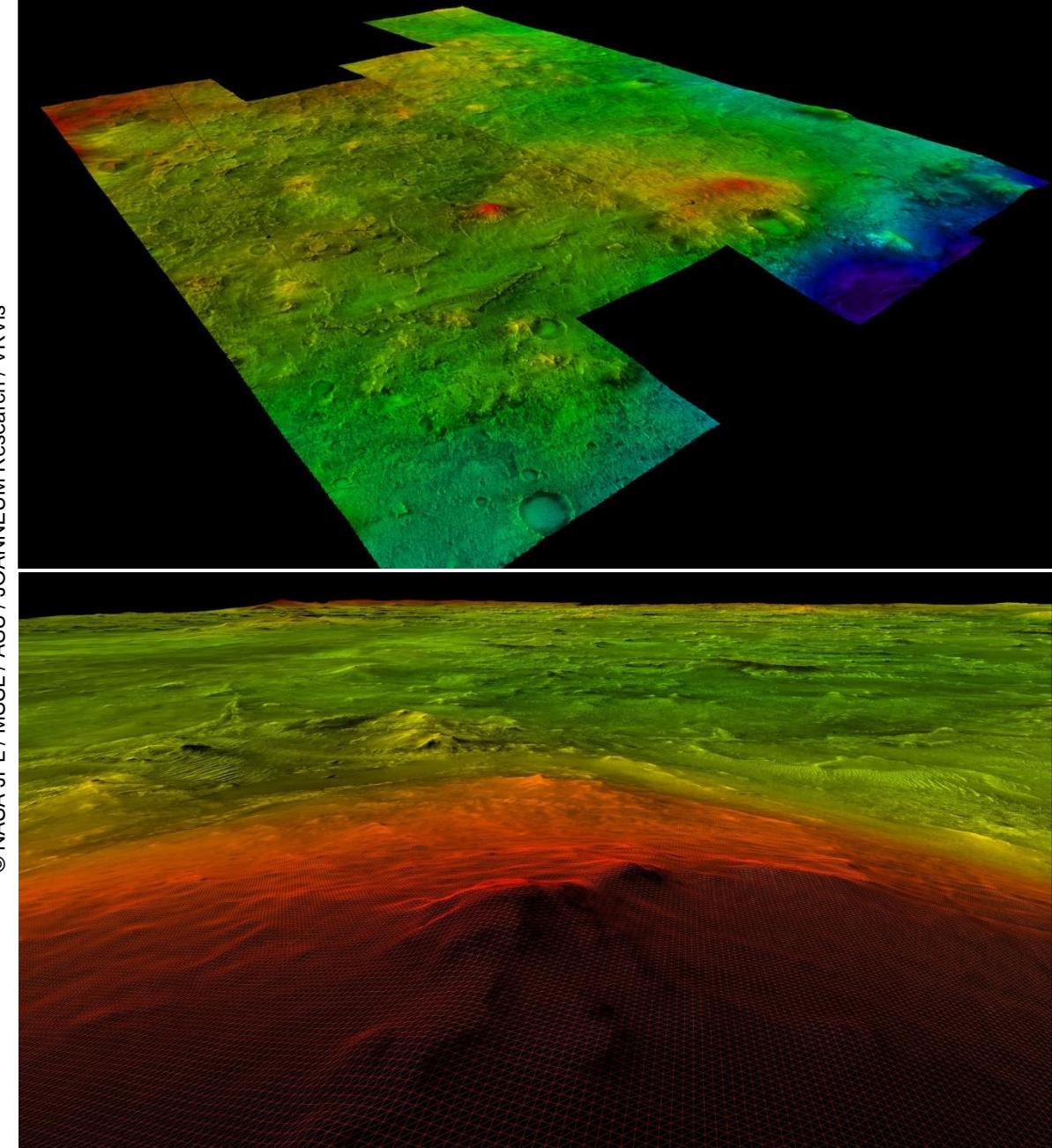


1D Textures

Used to texture lines



Or to colour terrain according to height



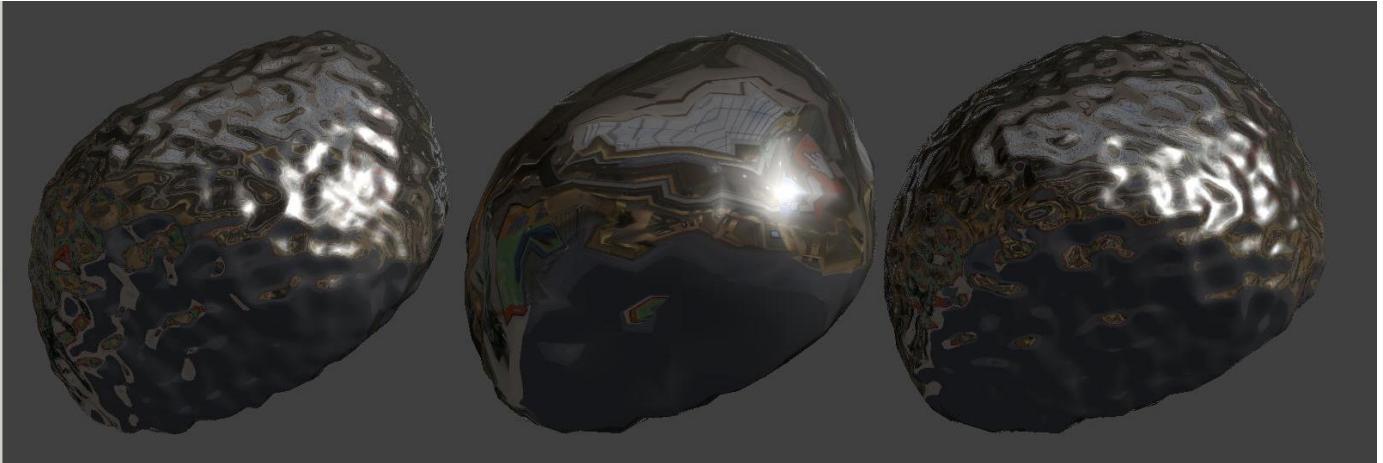
2D Textures

Most common texture

Used to texture surfaces

Also used for many special effects

- Light maps
- Normal maps
- Bump maps
- Parallax mapping
- Displacement mapping
- Environment mapping



© Eric Boissard

3D Textures

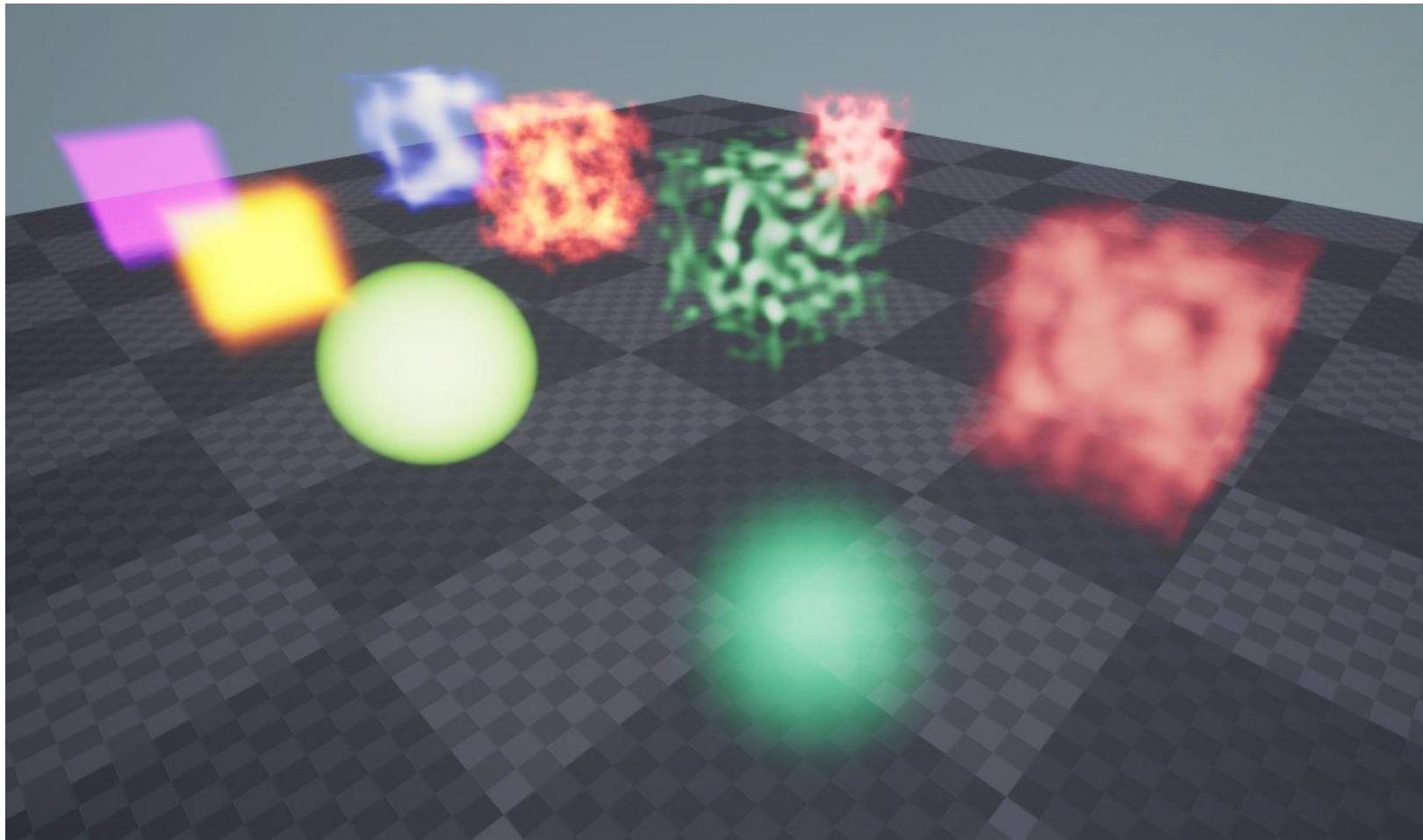
Volumetric textures

Mainly used for density of volume

Feasible on modern hardware as well

Enables new possibilities

- Can store light in a room, for example



© Polycount

Procedural Textures

Not bitmap-based

Evaluation of mathematical functions

- e.g. Perlin noise, turbulence, fractal pattern

Enables “solid textures”

- Objects appear as carved from a block of material
- E.g. marble, wood, ...



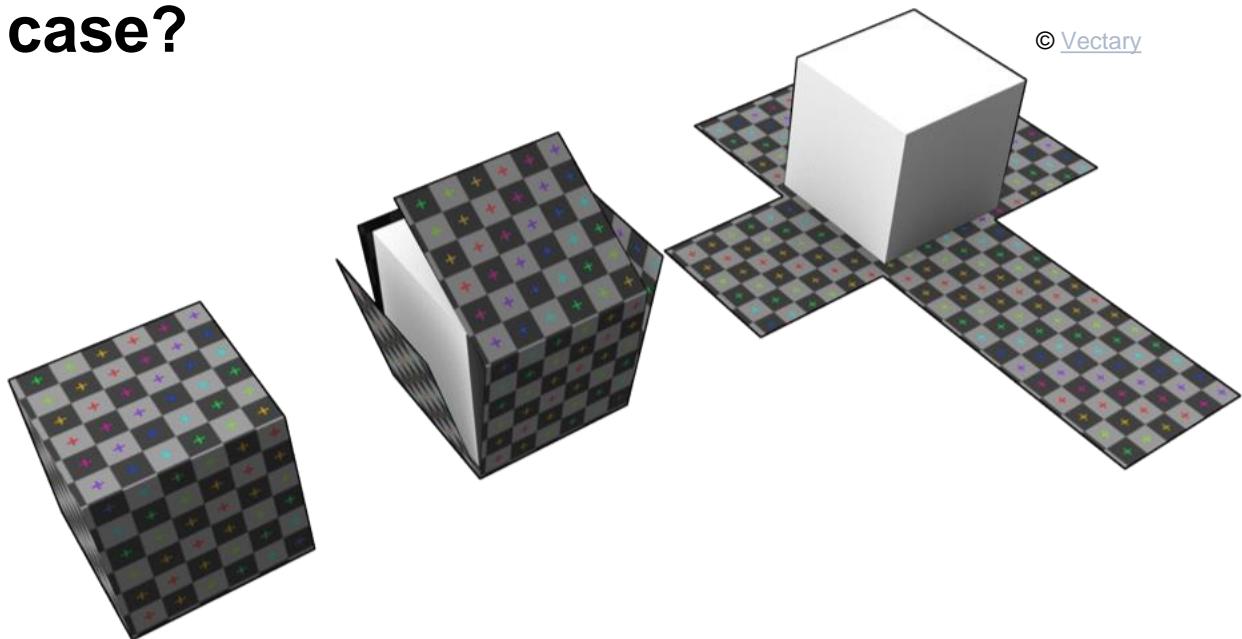
Texture Mapping

How to apply a texture to a polygon?

- Texels must somehow be matched to pixels
- Polygons can be transformed ...
- What to do with the texture in that case?

Mapping a rectangular texture
to a quad?

→ Easy 😊

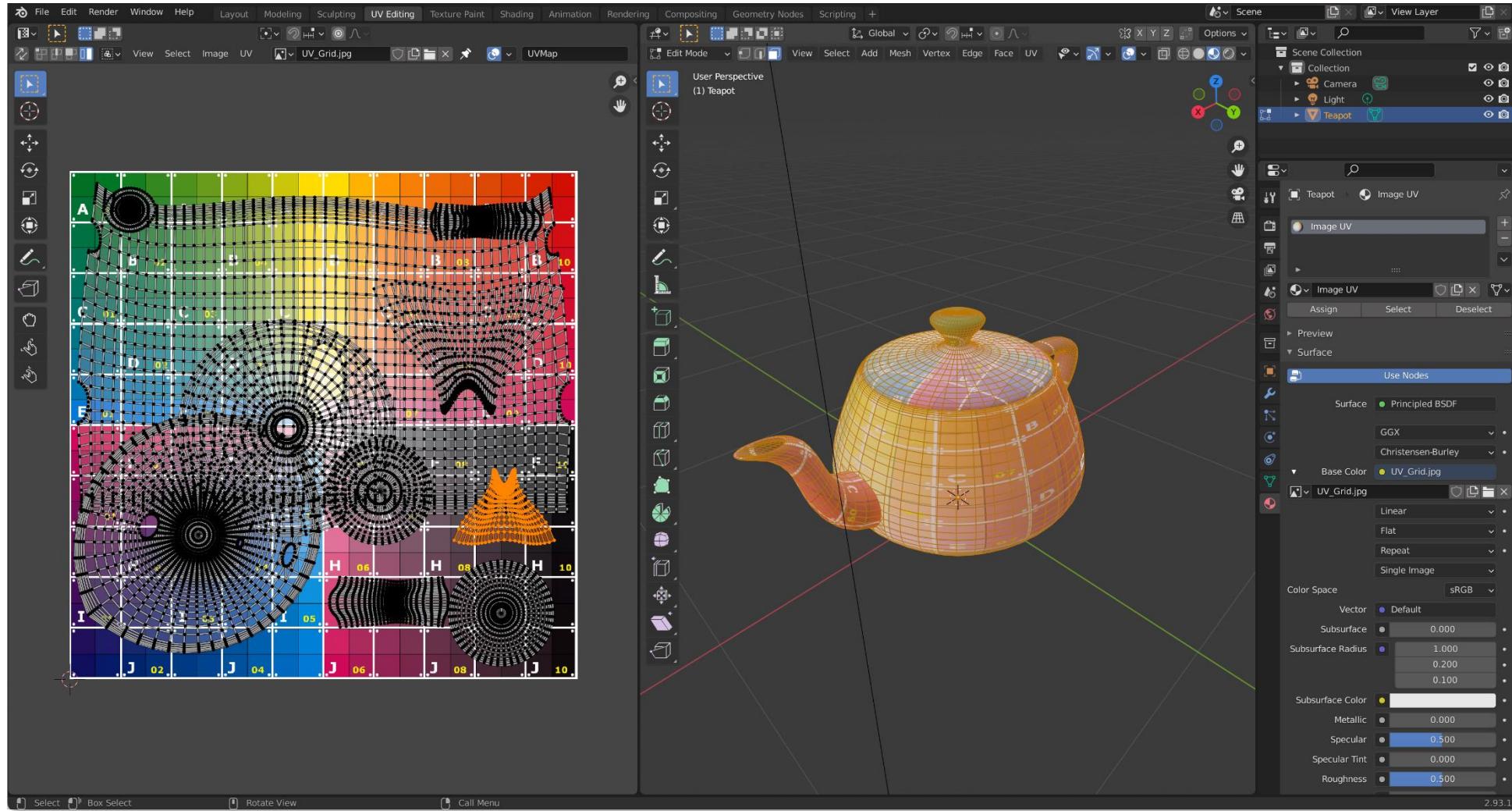


Mapping a rectangular texture
to a non-rectangular region?

→ Tricky 😬

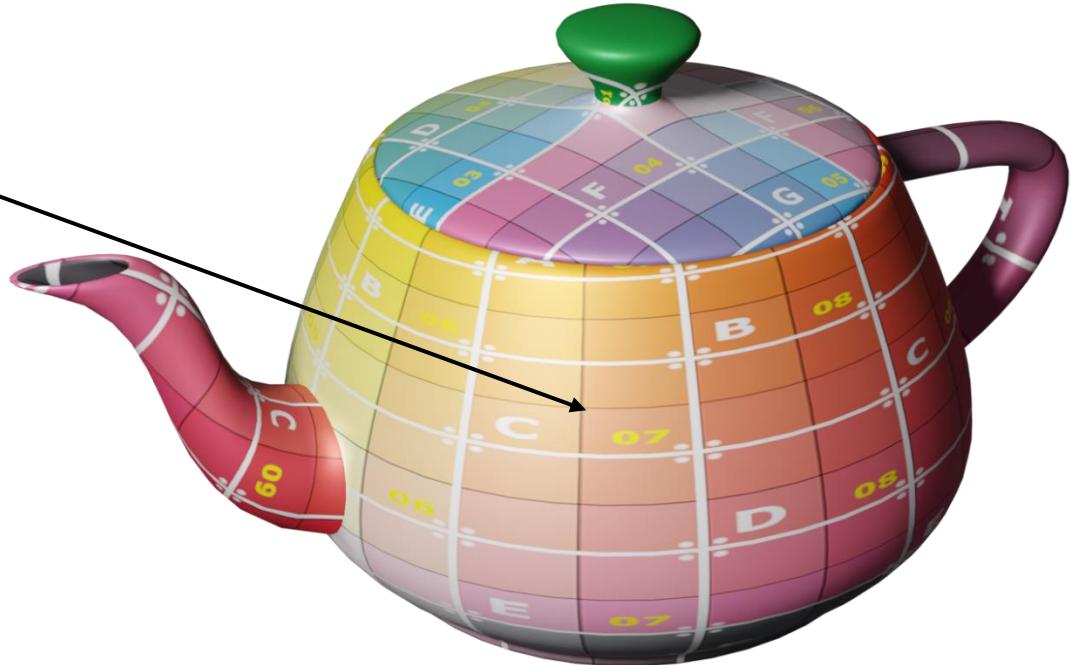
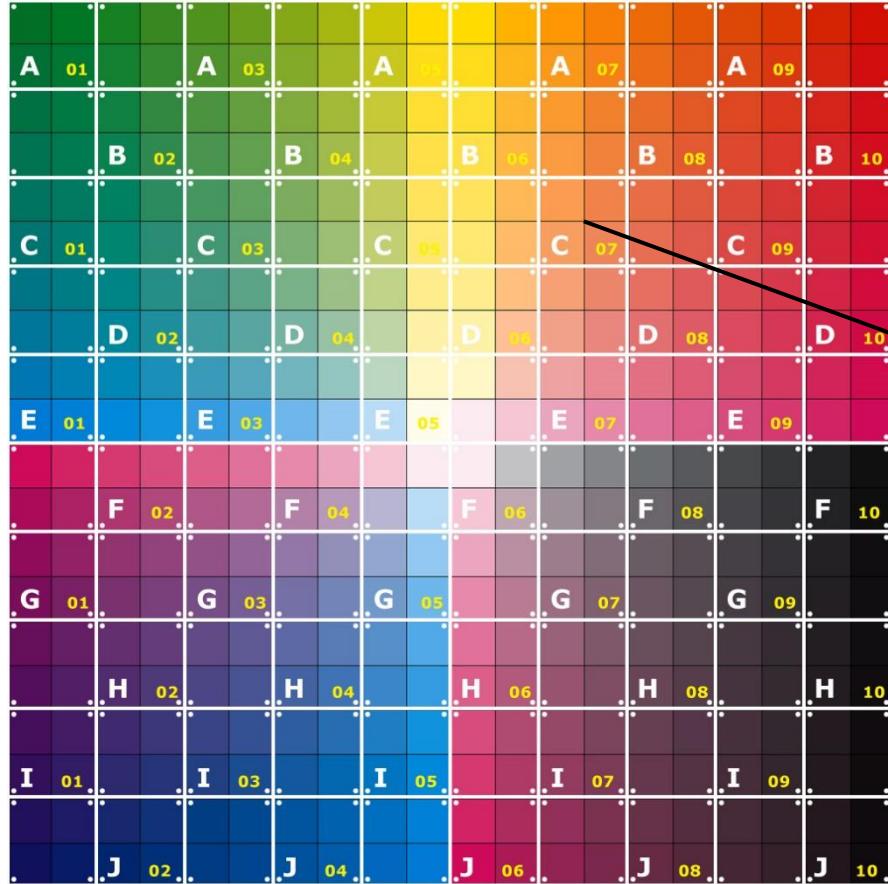
Texture Mapping

UV maps are used for more complex geometries



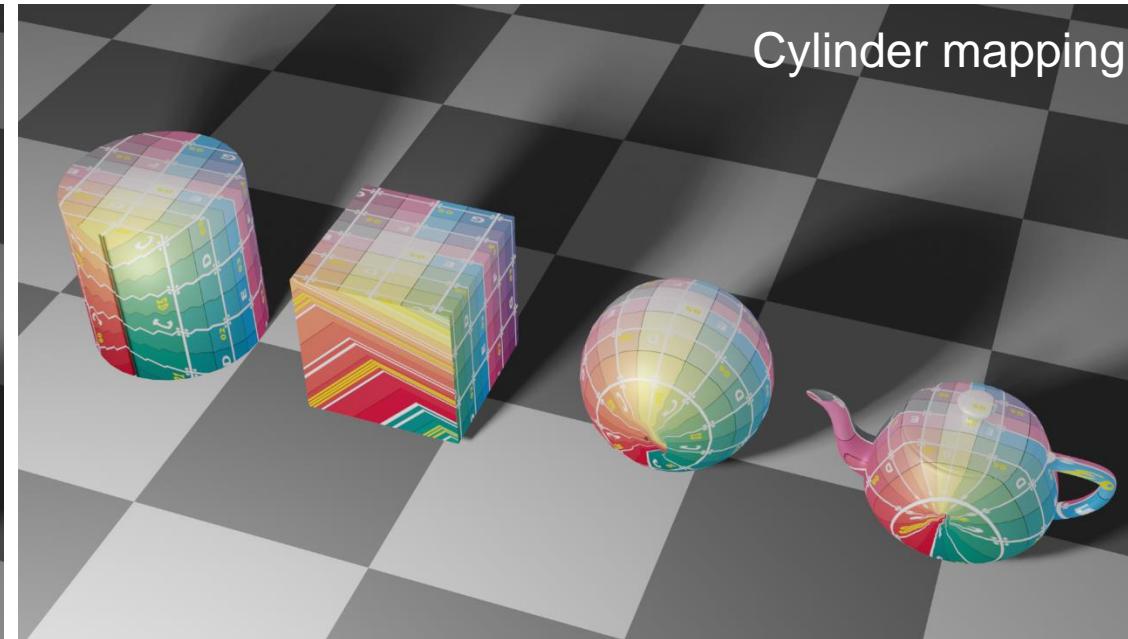
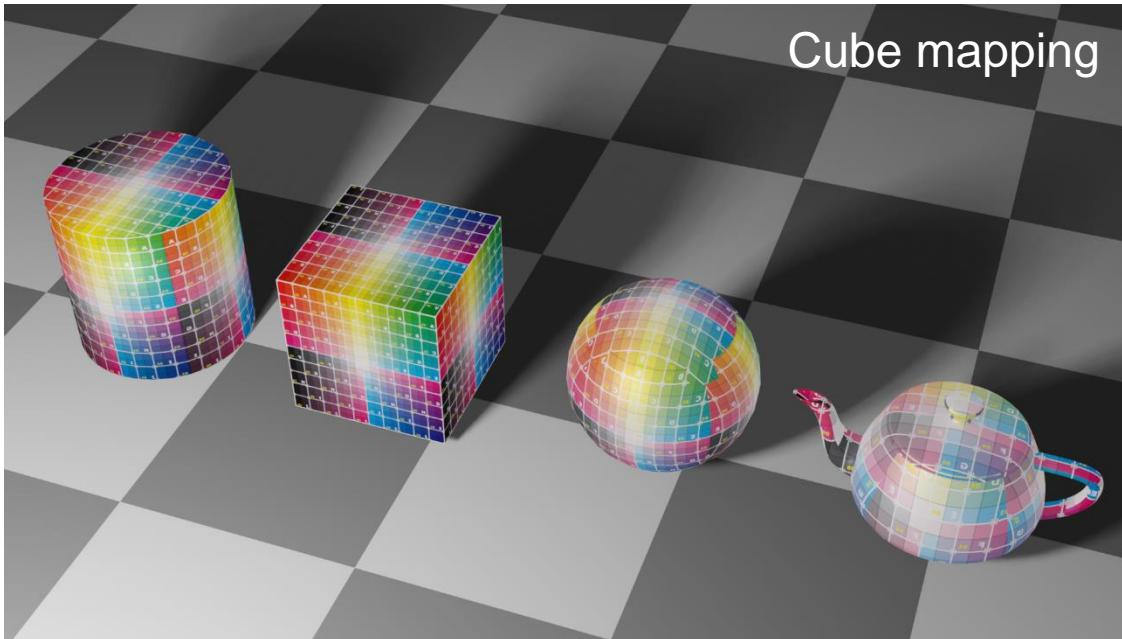
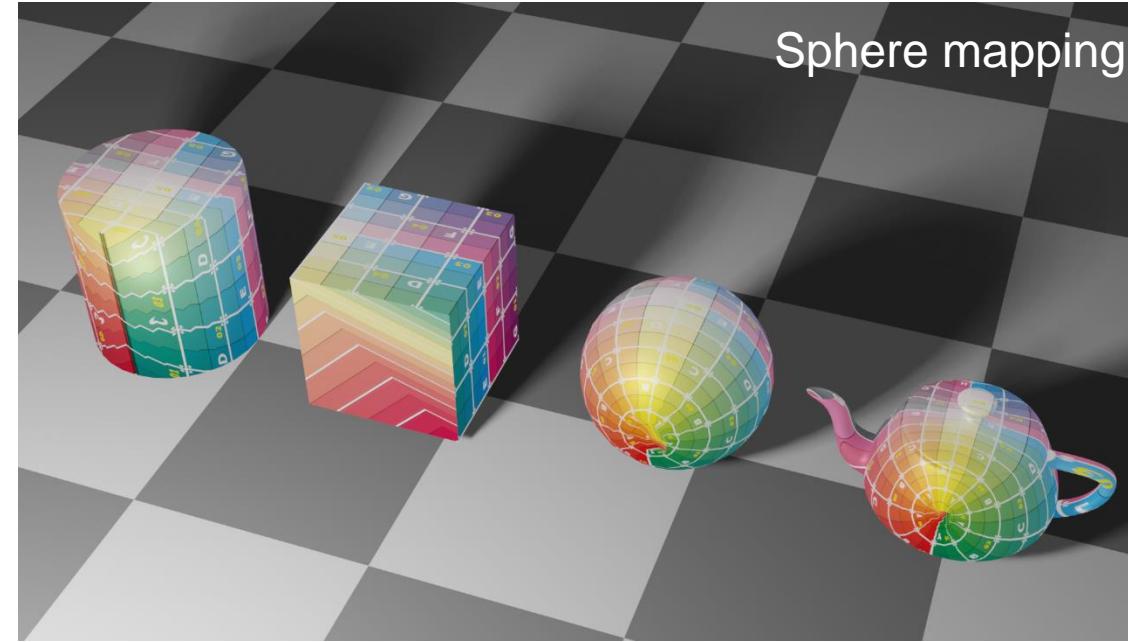
Texture Mapping

UV grid textures help to define UV maps interactively



Texture Mapping

Standard projection methods



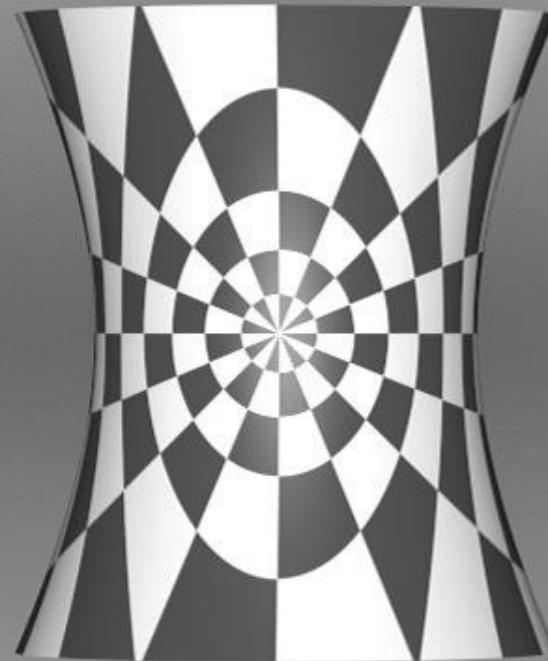
Texture Mapping

Standard projection methods

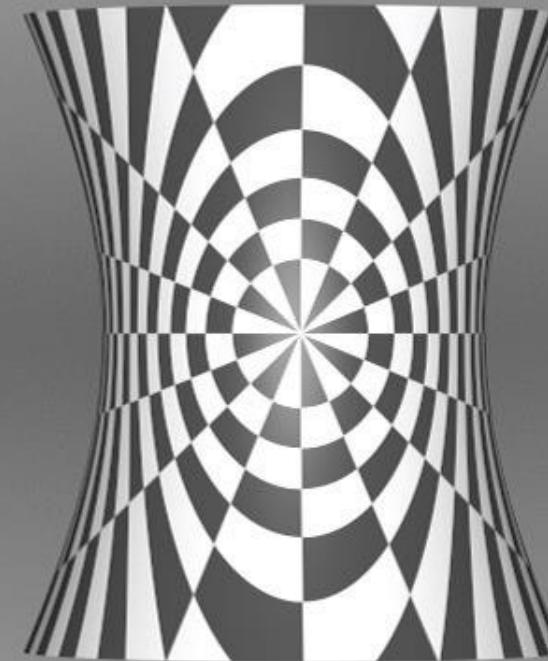
From [PBRT book](#)



manually adjusted UV map



sphere mapping



cylinder mapping



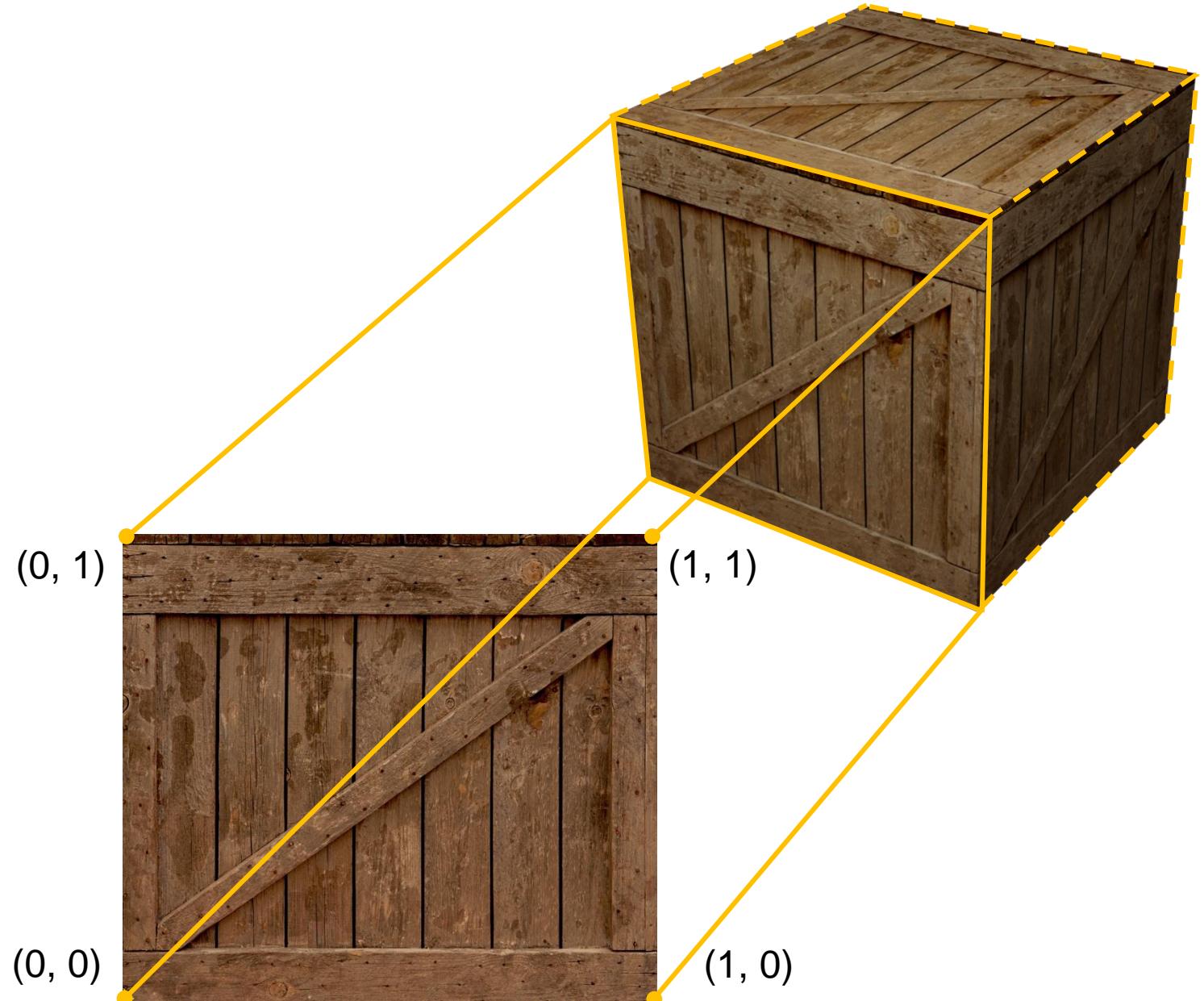
planar mapping

Texture Coordinates

Define per-vertex correspondence to texture map

Usually done with UV map

UV coordinates are stored with vertex data in indexed face sets



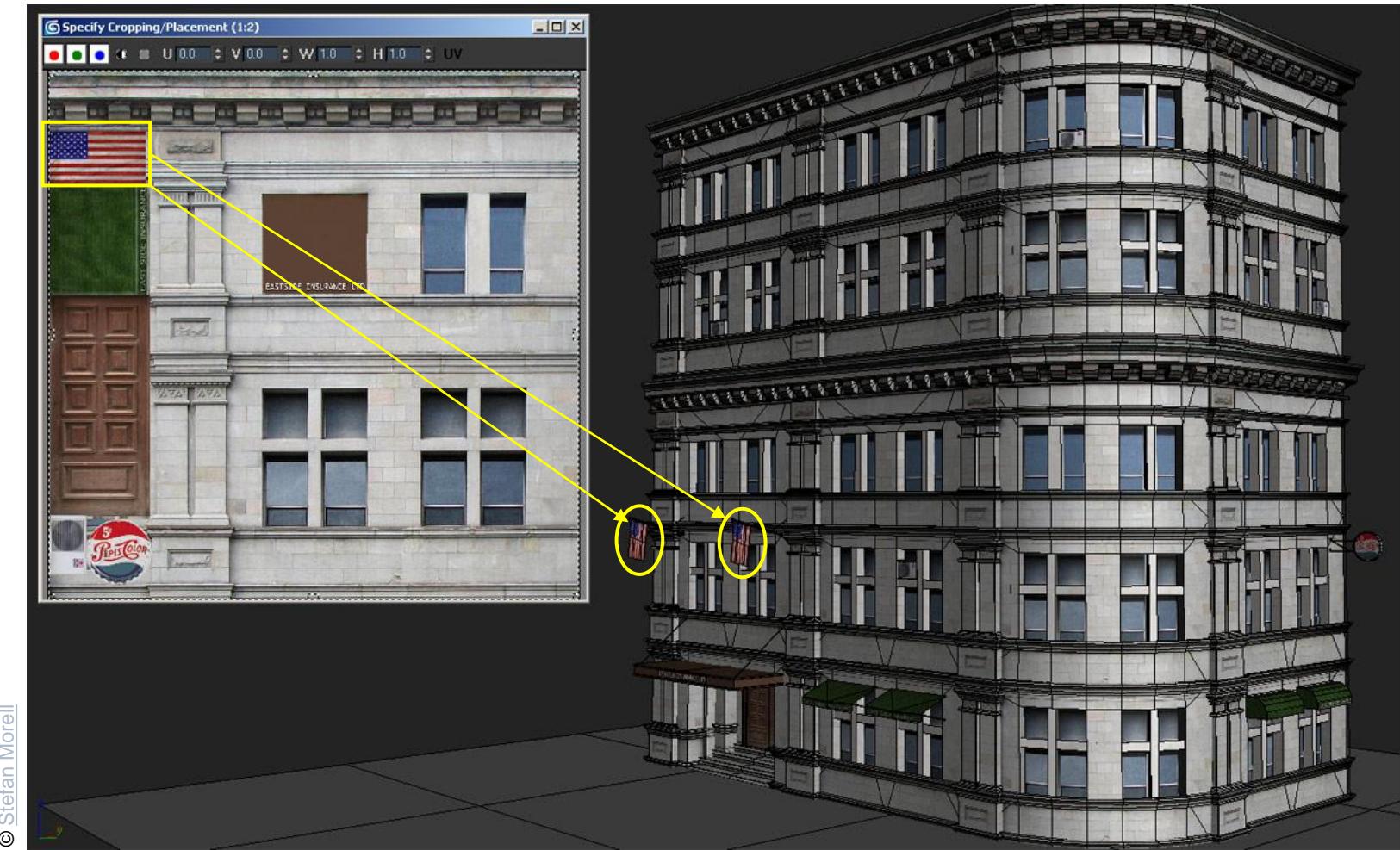
Texture Atlases

A texture atlas combines many small textures into one large texture

Texturing of objects with sub-images

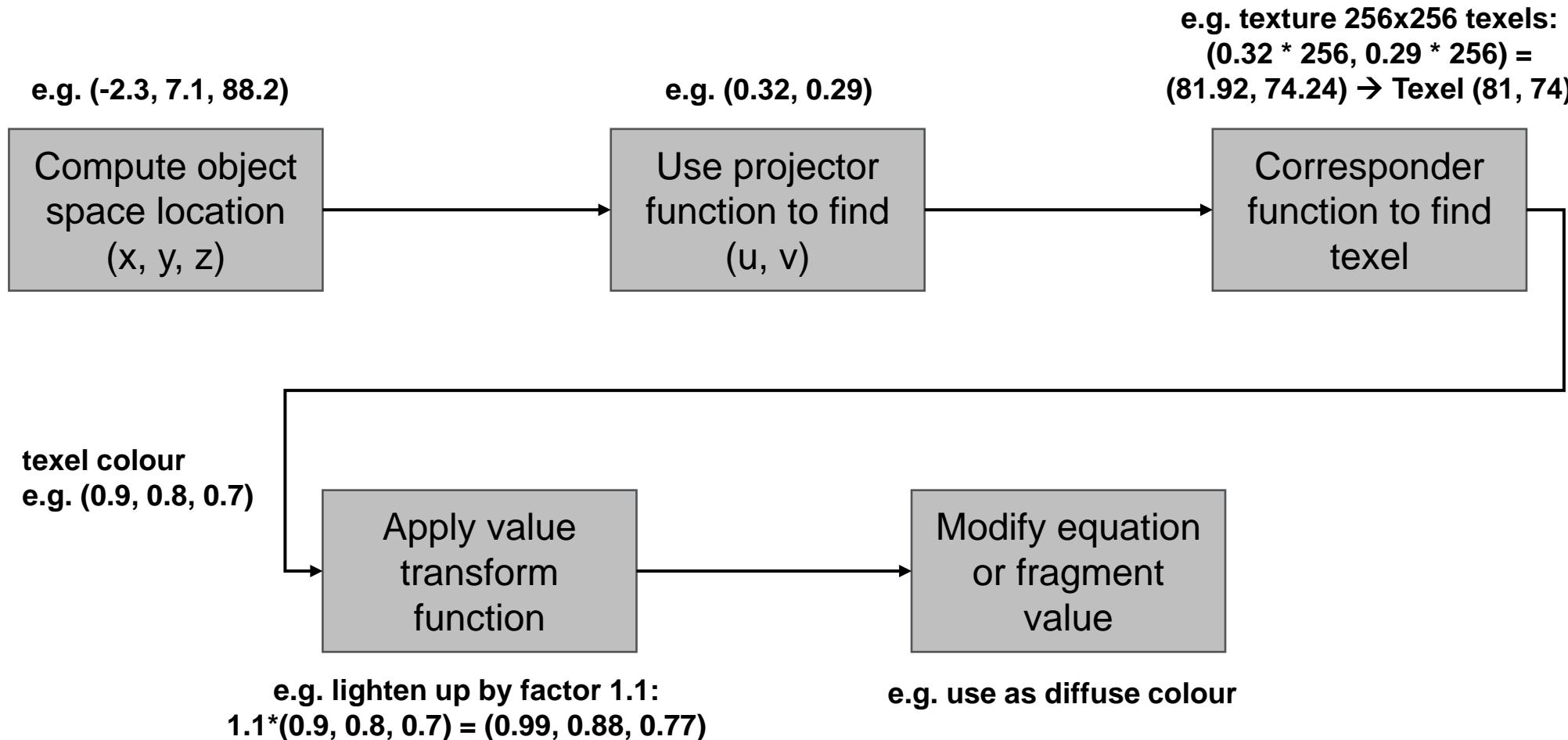
- See [Wikipedia](#) for an animated example of a small texture atlas

Multiple UV maps access same atlas



© Stefan Morell

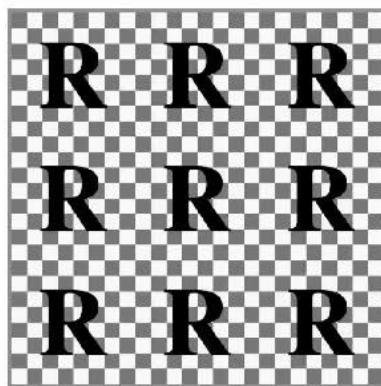
Generalized Textures



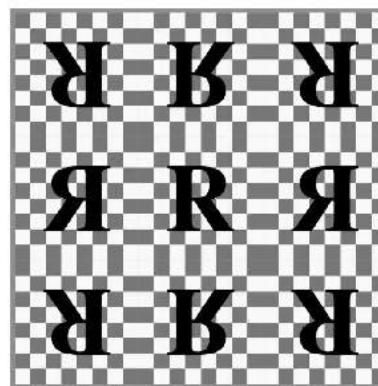
Corresponder Function

Matrix transformation to shift, scale, rotate a texture

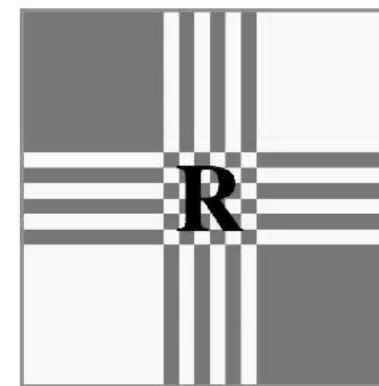
What happens outside of range [0, 1]



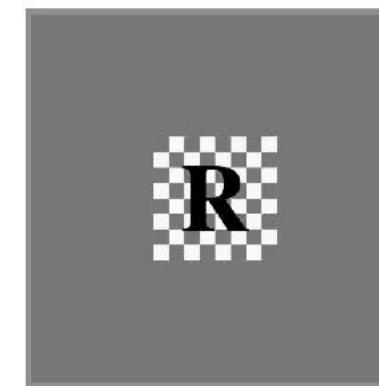
repeat



mirror



clamp



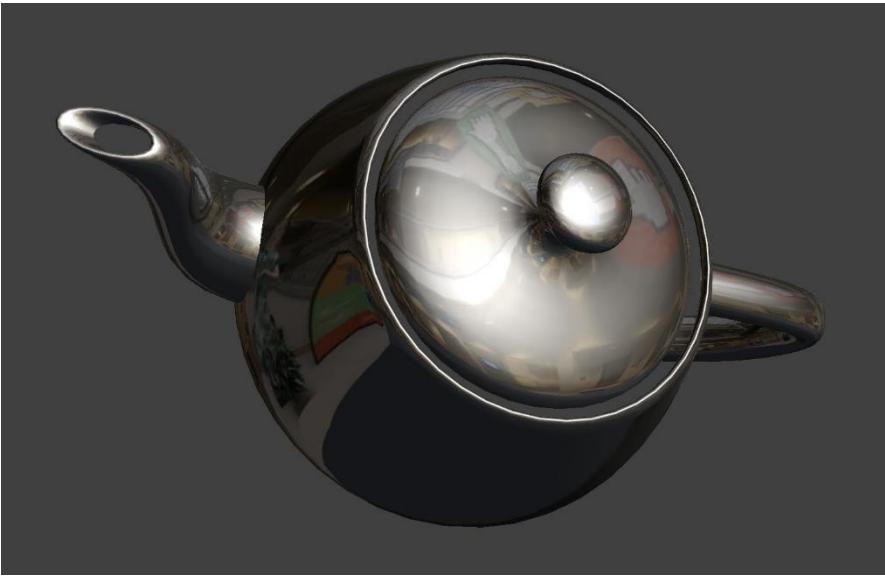
border

Environment Mapping

Assumes the environment is infinitely far away

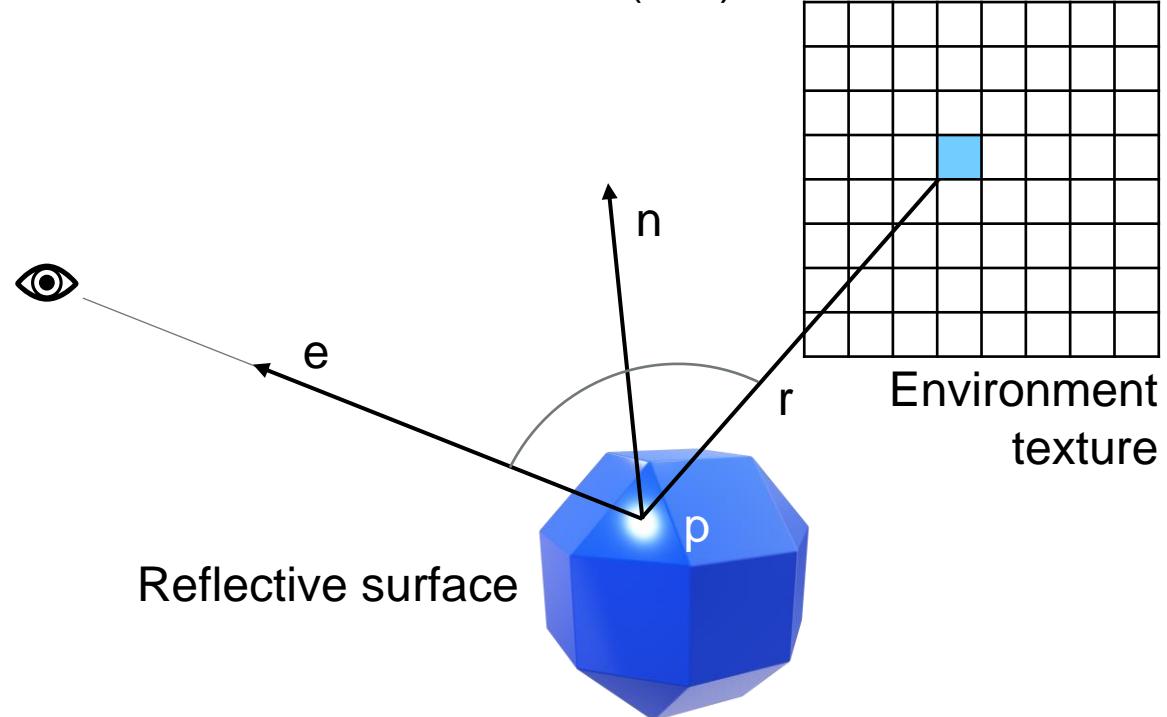
Cube mapping is the norm nowadays

- Advantages: no singularities as in sphere map
- Much less distortion
- Gives better result
- Not dependent on a view position



© Eric Boissard

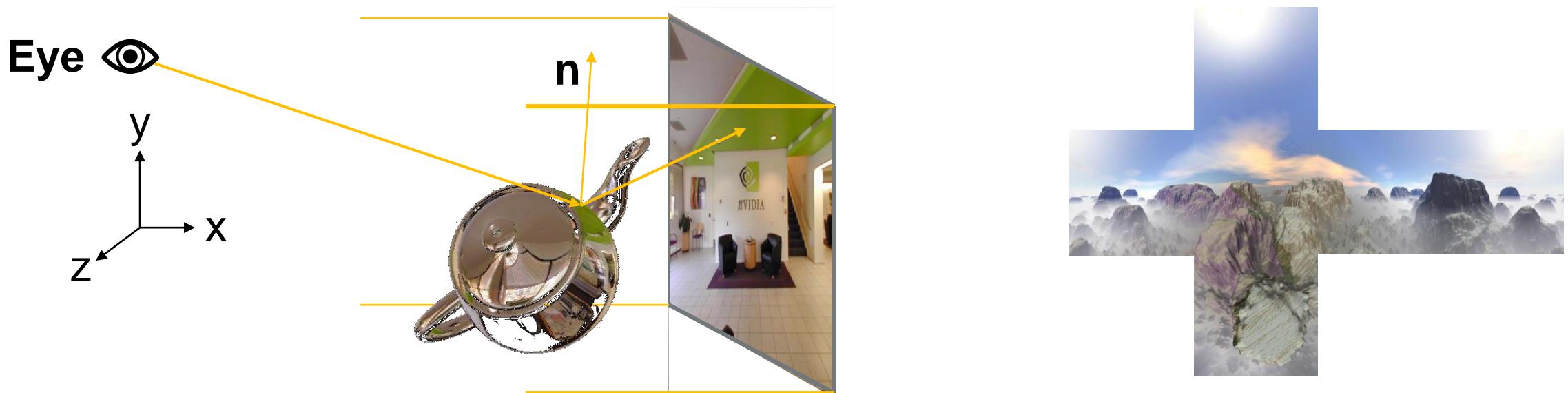
Projector function converts
reflection vector (x, y, z) to
texture (u, v)



Environment Mapping

Cube Mapping

- Simple math: compute reflection vector r
- Largest abs-value of component determines cube face
 - Example: $r = (5, -1, 2)$ gives POS_X face
- Divide r by 5 gives $(u, v) = (-1/5, 2/5)$
- If your hardware has this feature, then it does all the work

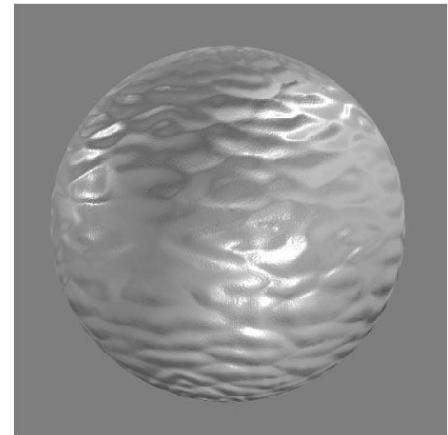
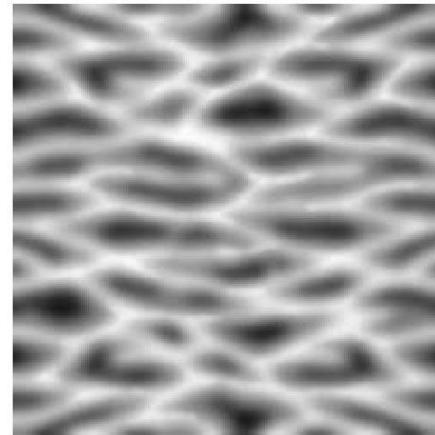


Bump Mapping

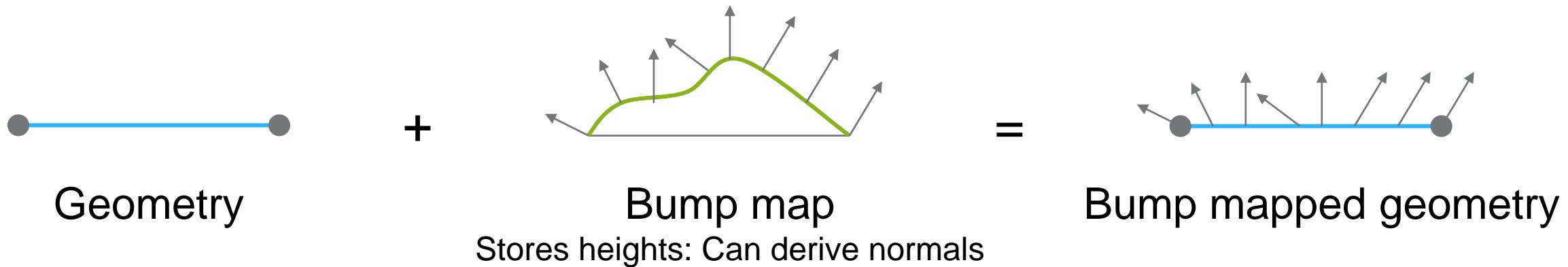
By Blinn in 1978

Inexpensive way of simulating wrinkles and bumps on geometry

- Too expensive to model these geometrically



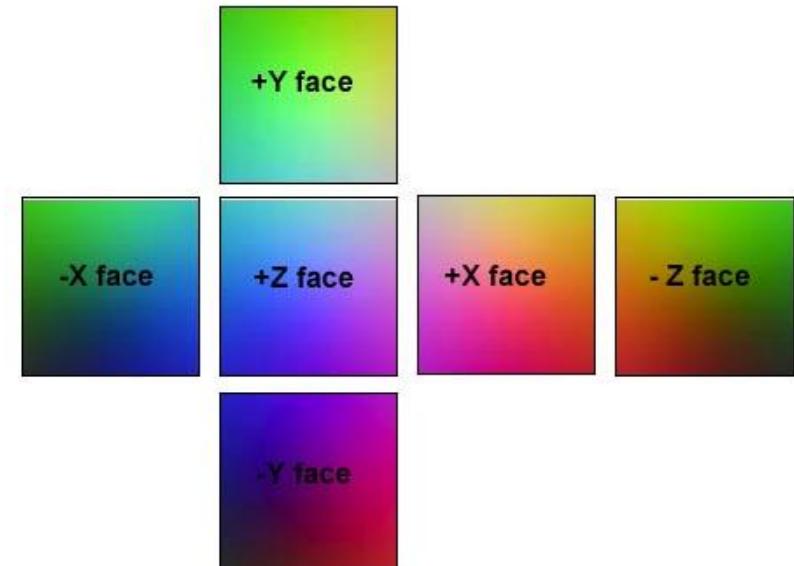
Instead let a texture modify the normal at each pixel, and then use this normal to compute lighting



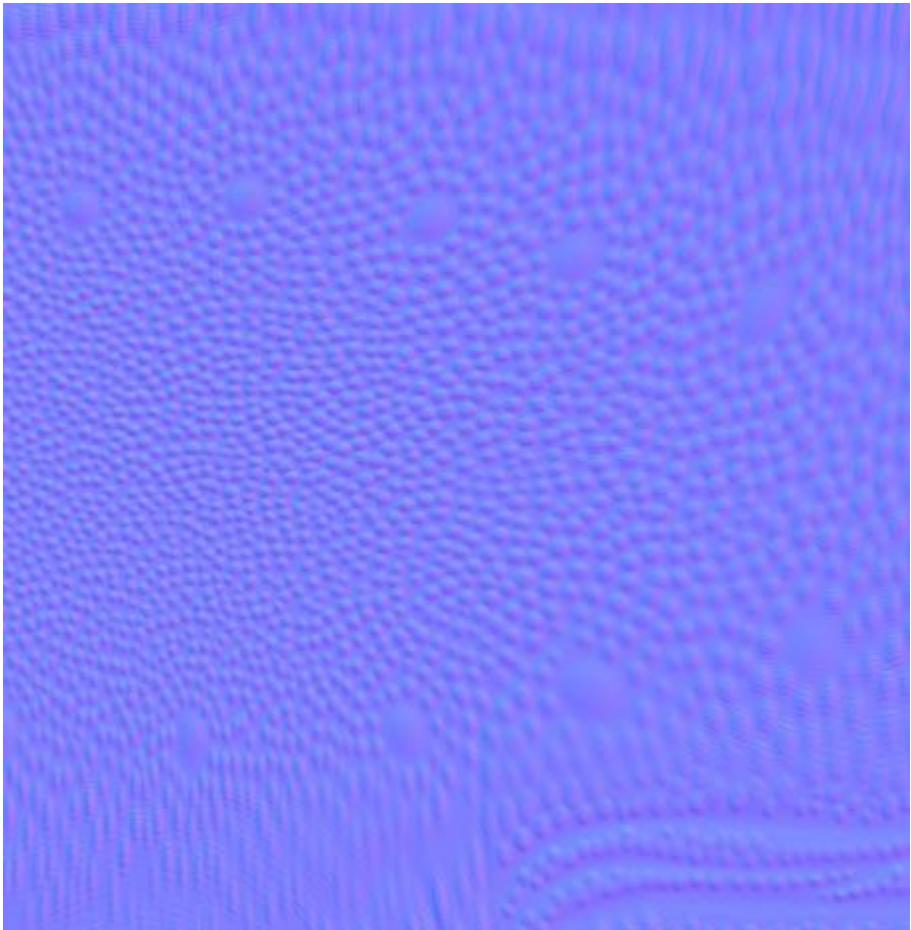
Bump Mapping

Lighting:

- **Diffuse:** $n \cdot l$ **Specular:** $(n \cdot h)^m$
- **Assume directional lights**
- **Diffuse:** Fetch per pixel normal from bump map
 - Then compute per-pixel dot product with light vector (constant)
- **Specular: h is (assumed) constant for directional lights**
 - Compute per-pixel dot product with normal from bump map
 - Gives $(n \cdot h)$, then $(n \cdot h)^2$, $(n \cdot h)^4$, $(n \cdot h)^8$
- **How to store normals in texture (bump map)**
 - $n = (n_x, n_y, n_z)$ are in $[-1, 1]$
 - Add 1, multiply 0.5: in $[0, 1]$
 - Multiply by 255 (8 bit per colour component)
- **Can be stored in texture**



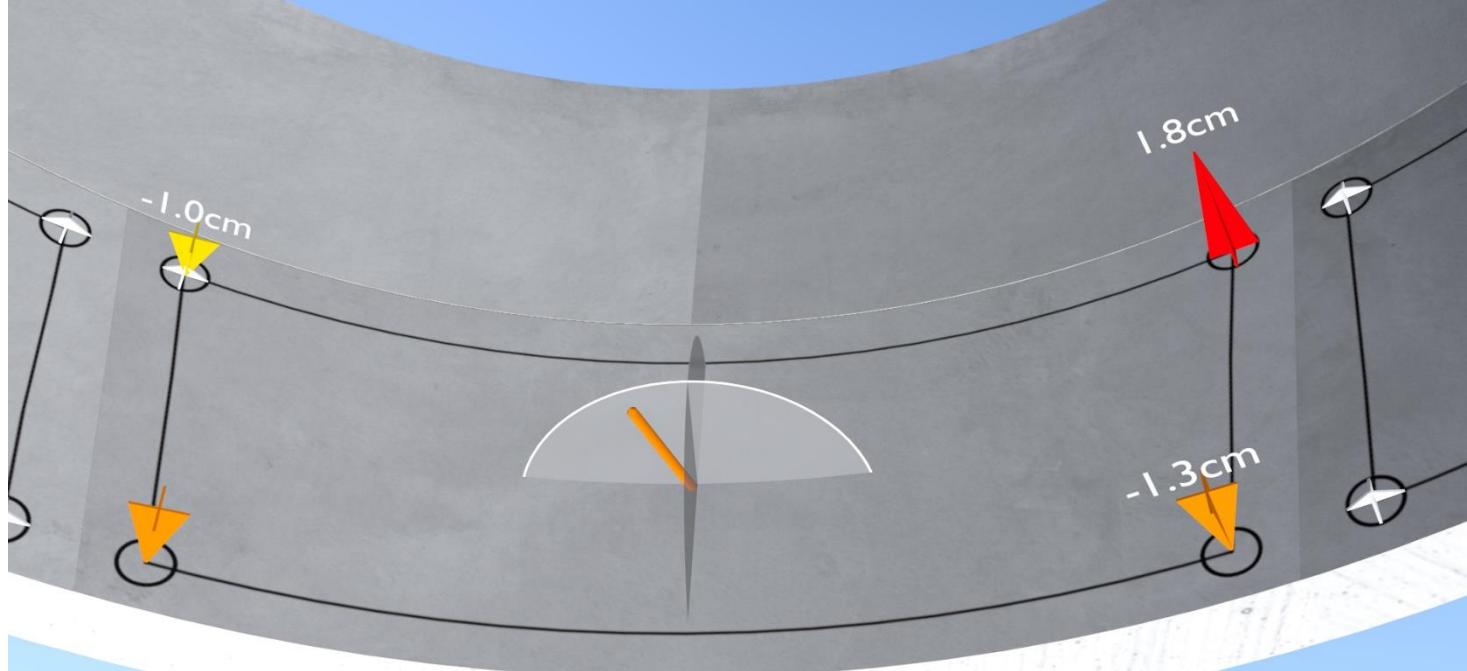
Bump mapping



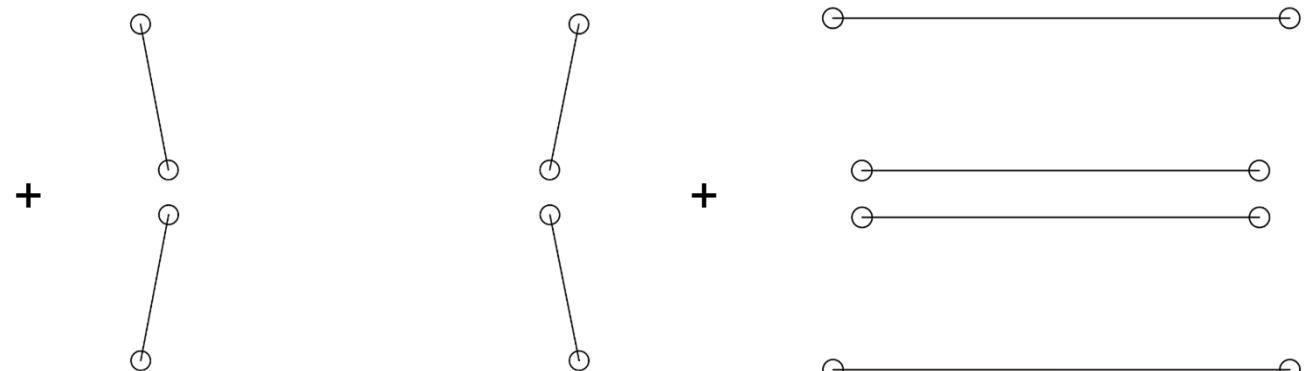
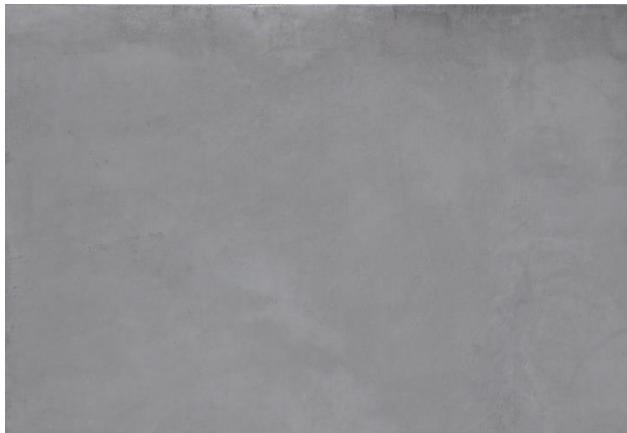
Multi-texturing

More than one set of texture coordinates per vertex

The output from the first texture stage is input to the next



Sensor visualization using a diagram texture over a concrete texture



Blending & Filtering

Texture Blending Operations

Replace, Decal

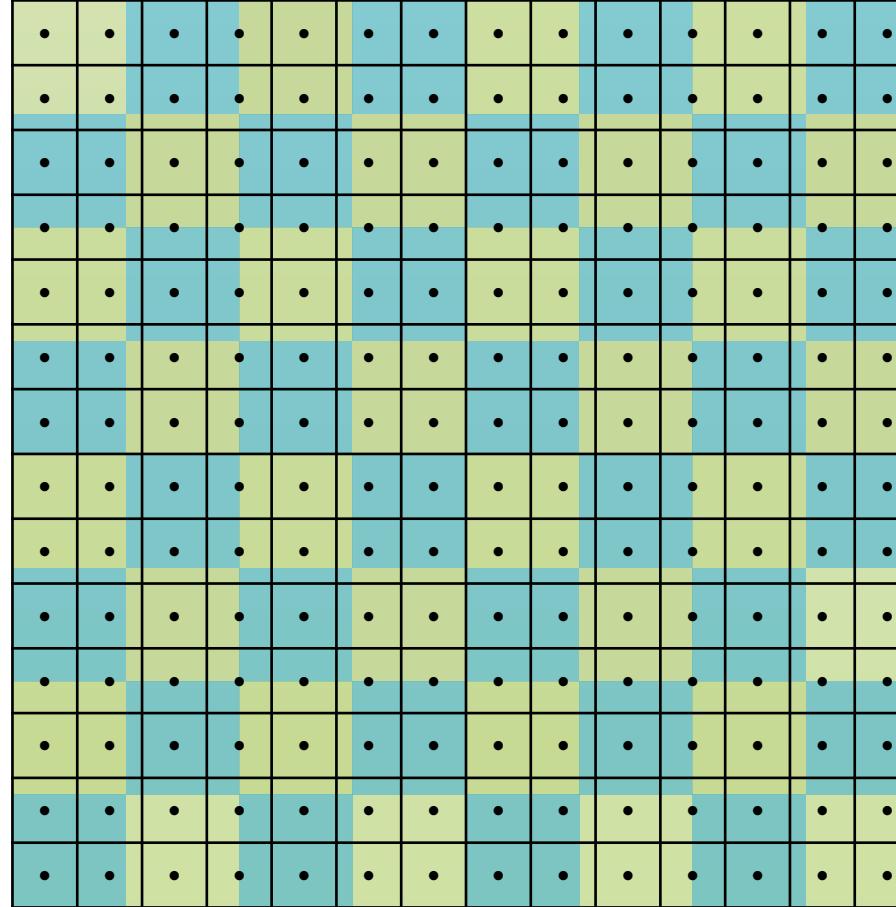
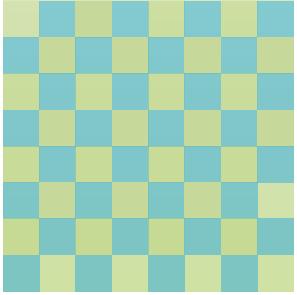
- Like a sticker applied to a surface
- Replaces “underlying” texture

Modulate

- e.g. used for lightmapping

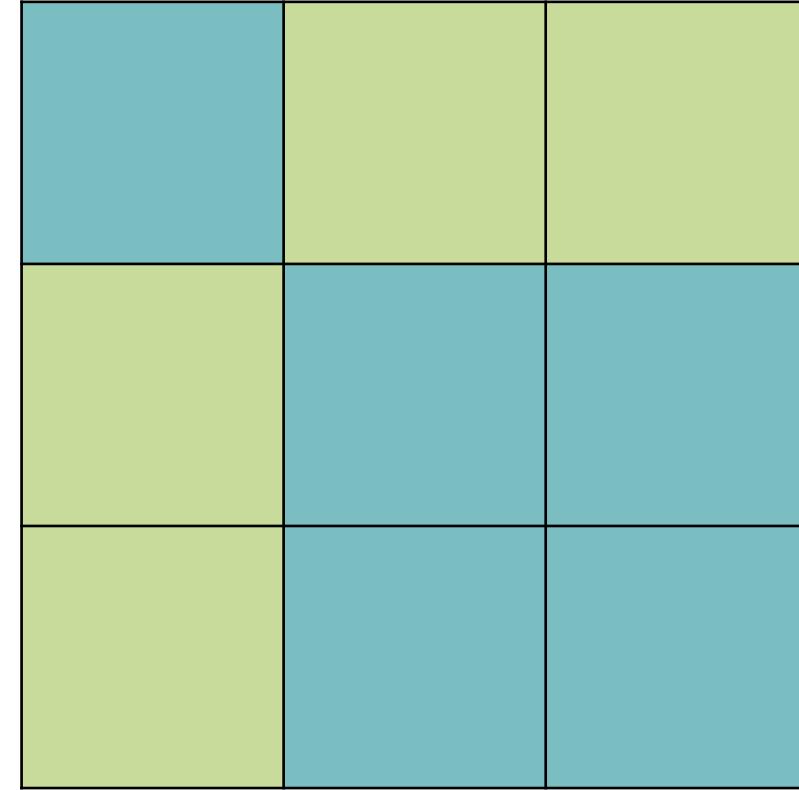
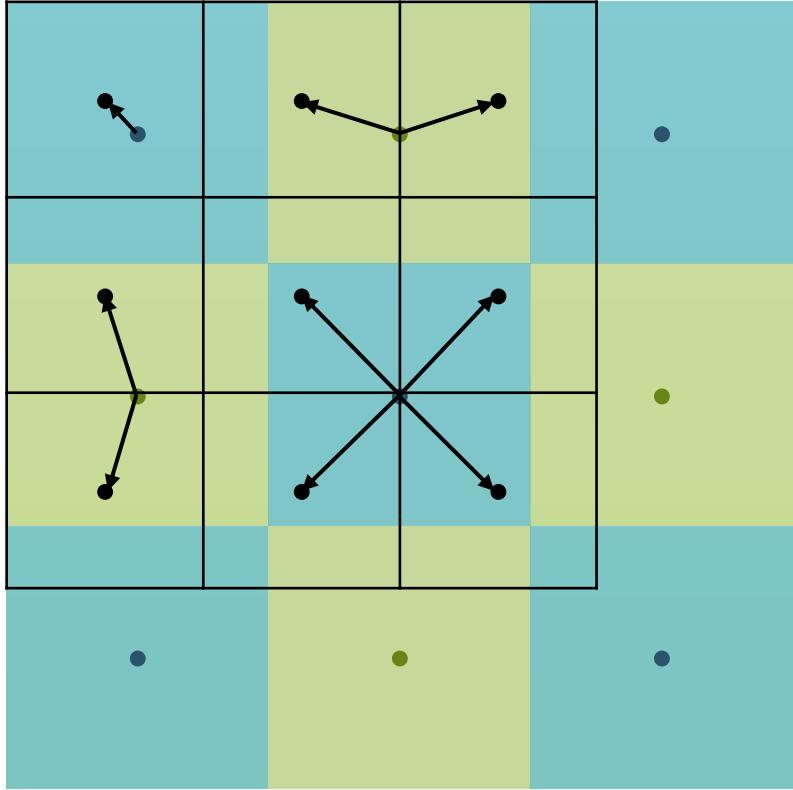
Magnification

One texel covers several pixels



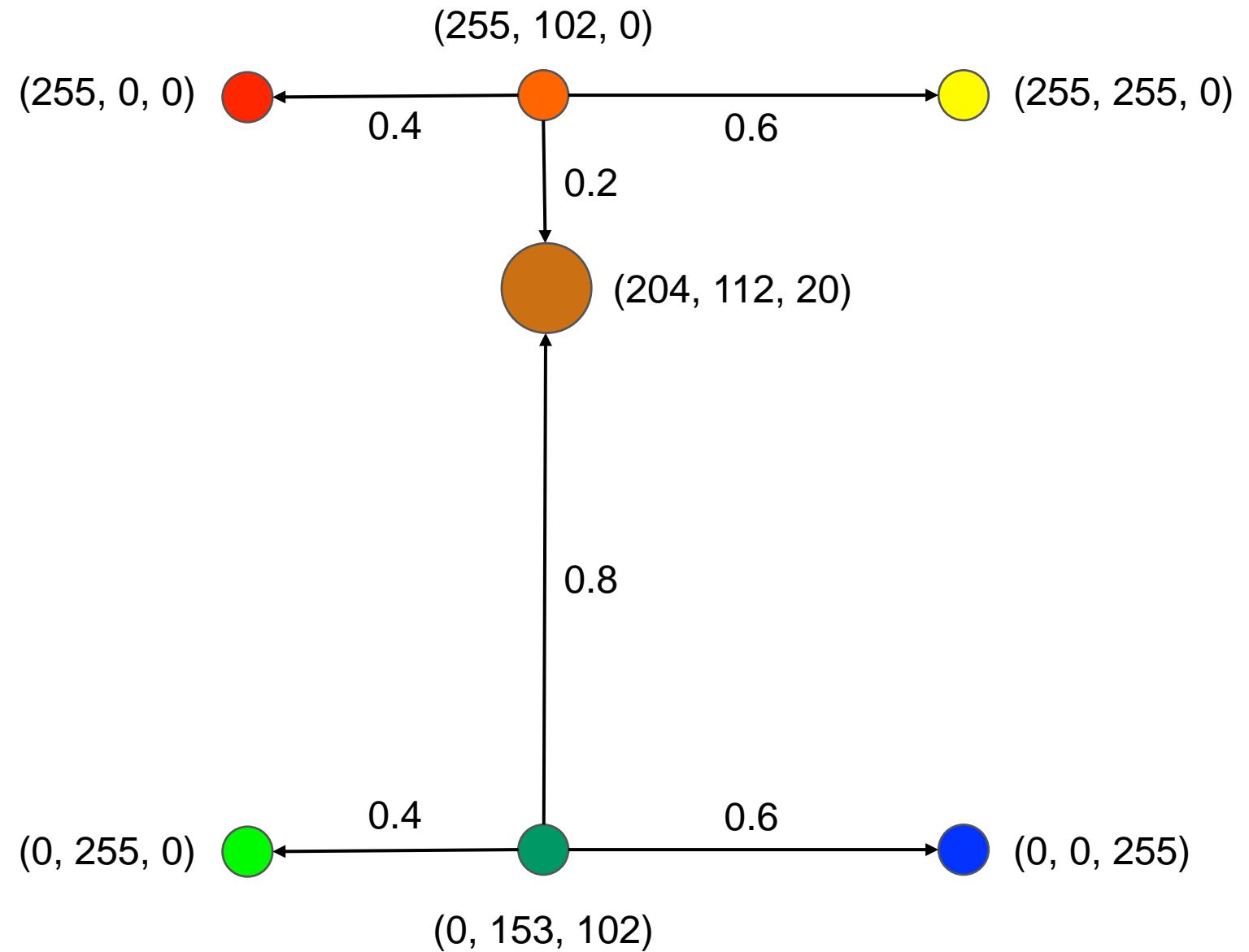
Magnification

Nearest neighbour



Magnification

Bilinear Interpolation



Minification

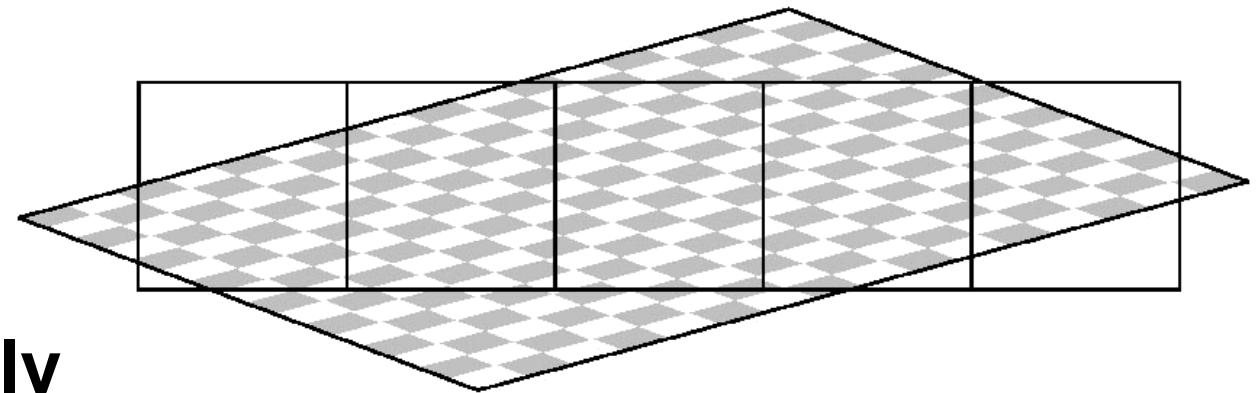
Several texels cover one pixel

Nearest neighbour and bilinear interpolation are problematic

Theory (sinc function) is too expensive

Cheaper: average of texel inside a pixel

→ still too expensive, actually



Solution: Mipmapping

Mipmapping

Also known as tri-linear filtering

Image pyramid

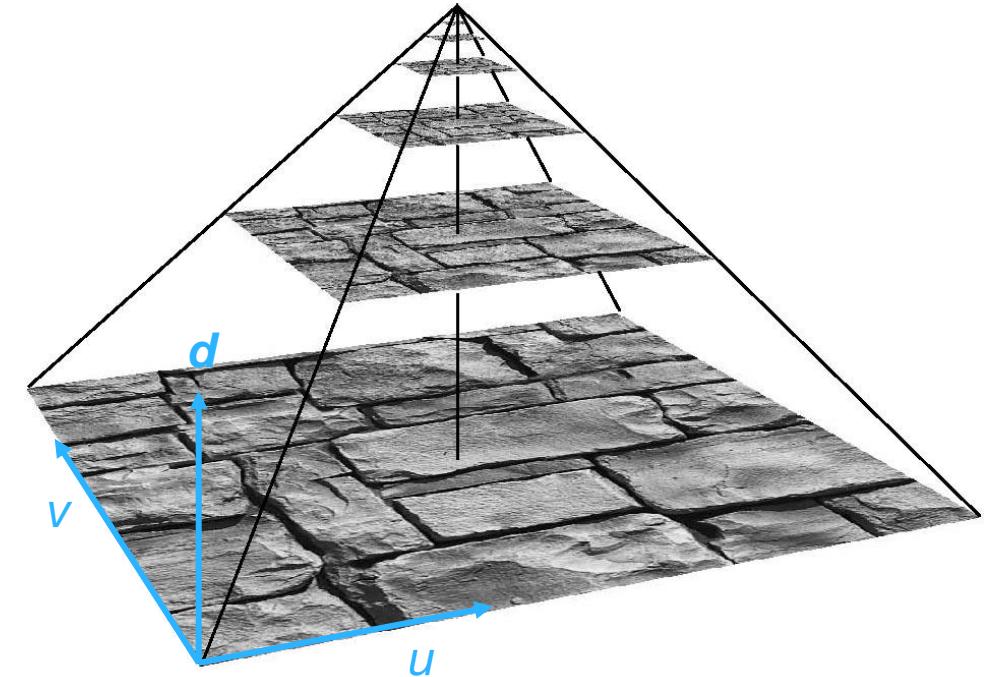
Half width and height when going upwards

Average over 4 "parent texels" to form "child texel"

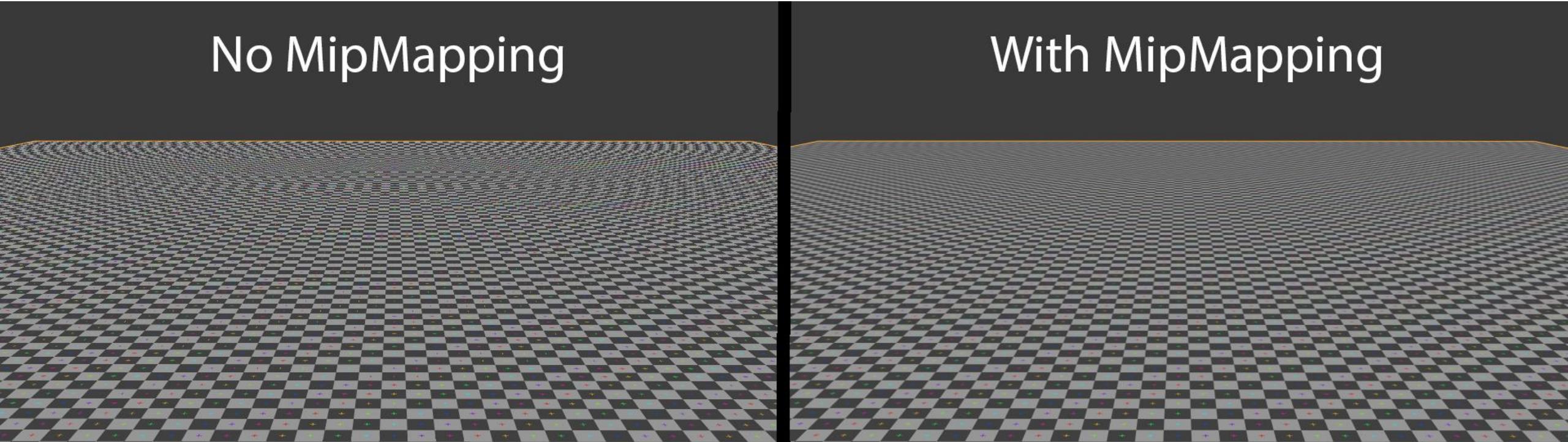
Depending on amount of minification, determine which image to fetch from

Compute d first, gives two images

- Bilinear interpolation in each



Mipmapping

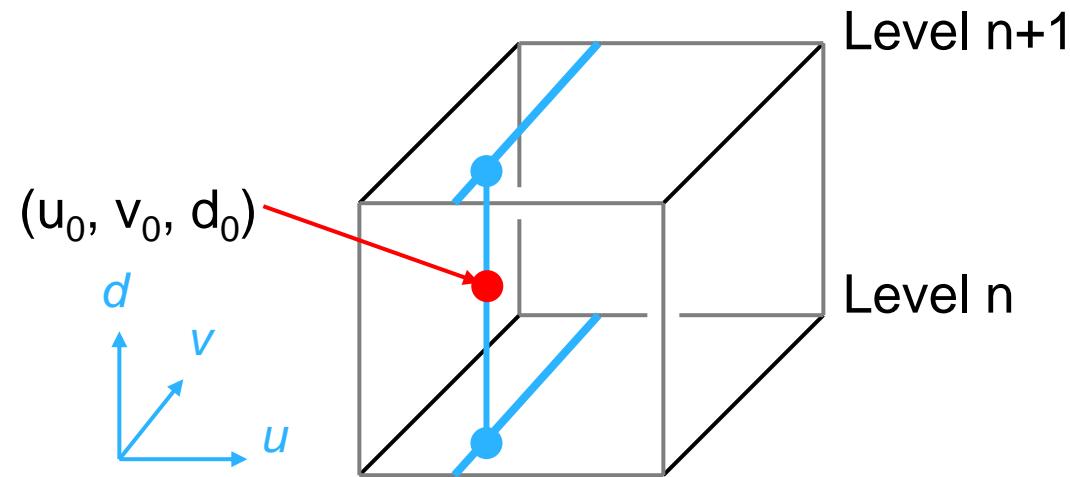


© CC0 Wikipedia

Mipmapping

Interpolate between those bilinear values

- Gives trilinear interpolation



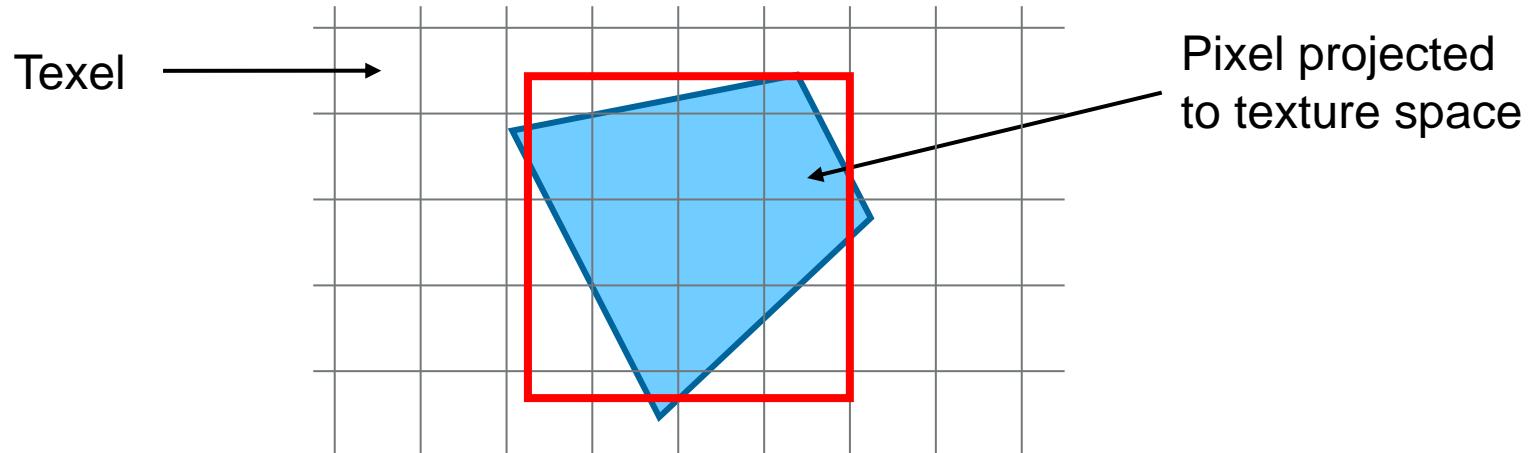
Constant time filtering: 8 texel accesses

How to compute d ?

Mipmapping

Computing d

- Approximate quad with square
- Gives over-blur!
- Even better: Anisotropic filtering
 - Approximate quad with several smaller mipmap samples



Anisotropic Filtering

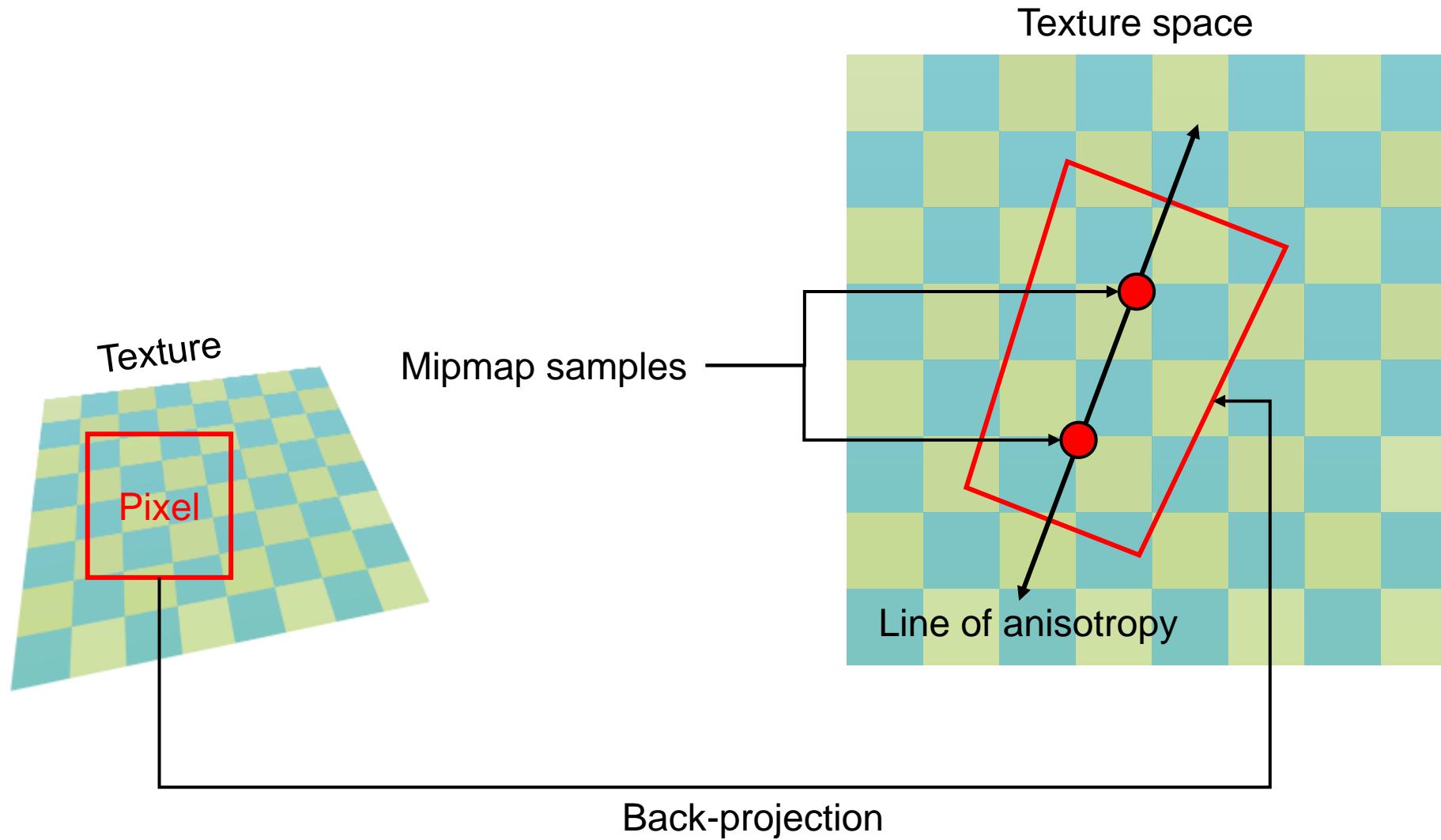
Sharper texture in the far distance when surfaces are viewed from an oblique angle

Mipmapping HW can be reused

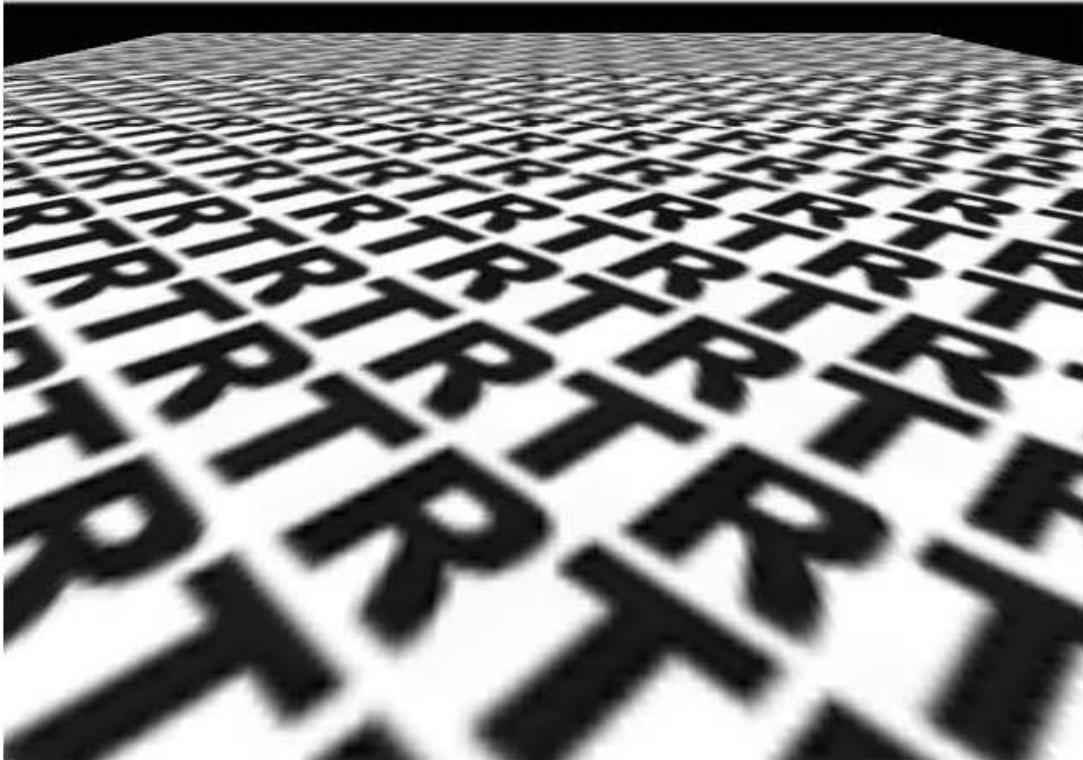
How it works (see figure on next slide):

- Back-project pixels into texture space
- Use several mipmap samples (squares) to cover these quads
- Use shorter side of quad to determine d
- Longer side of quad defines line of anisotropy that runs through its middle
- More than 2 samples are taken, when the ratio between the quad's sides is larger than 2 : 1
- Combine sampling results

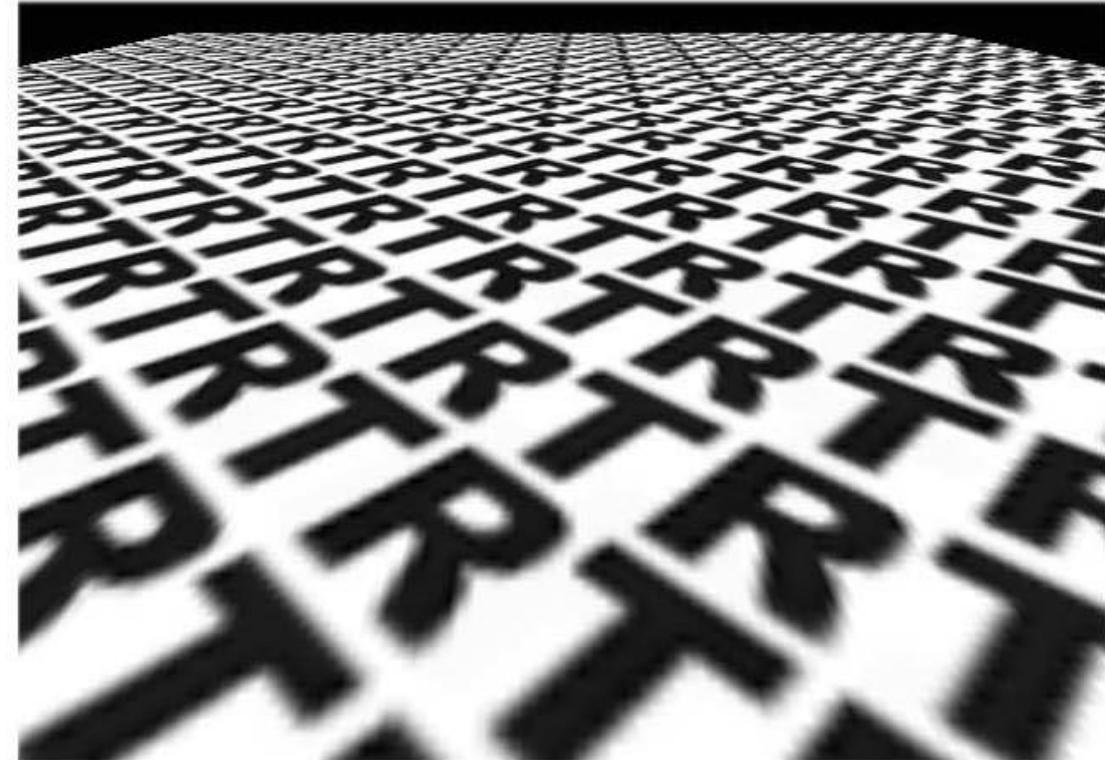
Anisotropic Filtering



Anisotropic Filtering



Mipmapping



Anisotropic filtering 16 : 1

Mipmapping

Memory requirements

- Not twice the number of bytes...!
- Rather 33% more – not that much





Special Techniques

Overview

- Transparent geometry
- Level of Details (LOD)
- Billboards, imposters and billboard clouds
- Light mapping
- Picking
- Collision Detection

Transparent Geometry



University of
Applied Sciences

Transparent Geometry

Basic idea:

1. Render opaque triangles
2. Enable blending
3. Render transparent triangles back to front

A BSP tree is used to efficiently sort triangles

- See *Lecture 3 - Geometry 2*

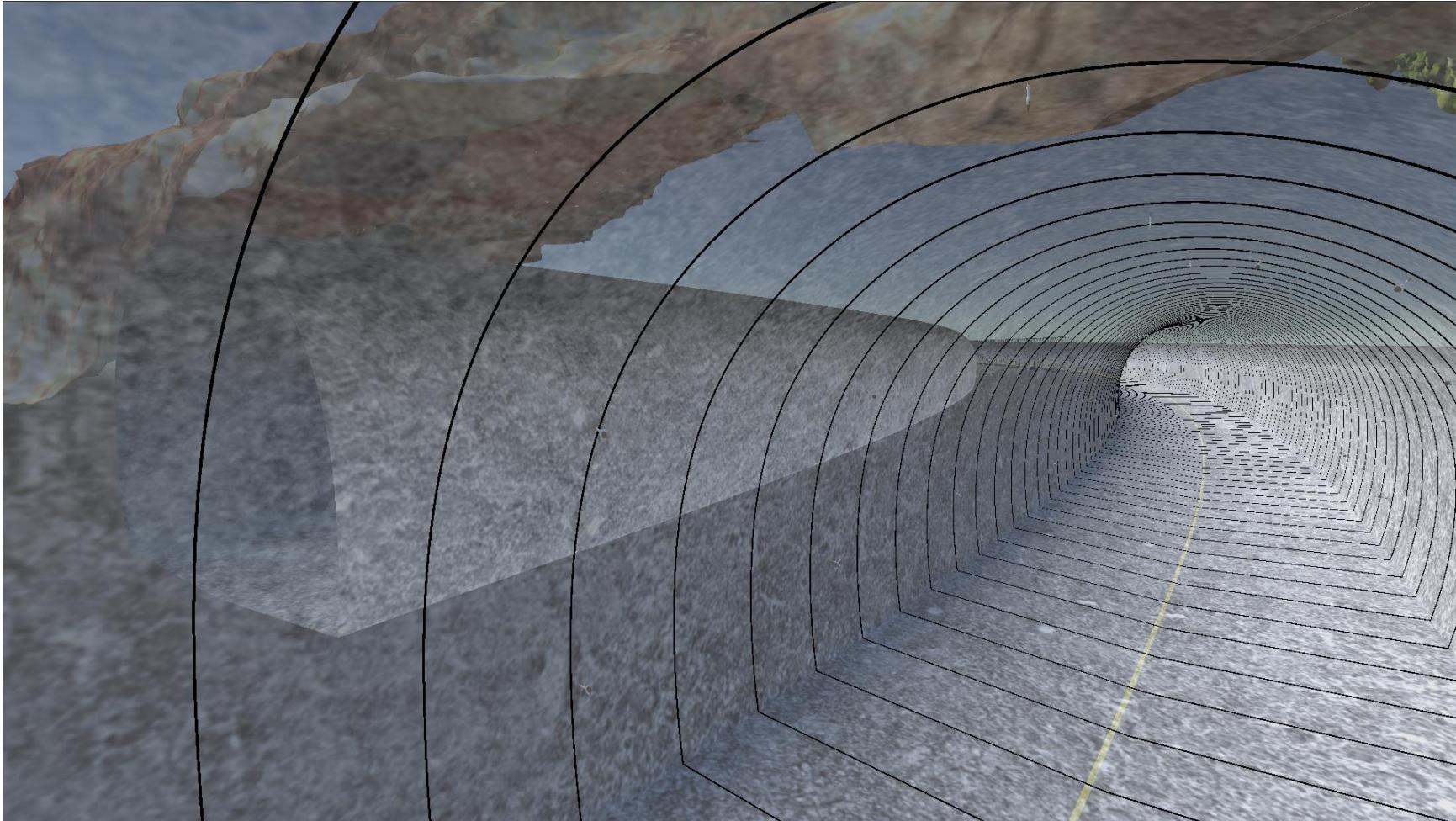
Question:

- Other sorting criteria?
- What is the fastest sorting algorithm?

Transparent Geometry

Visualization of tunnel constructions

A semi-transparent wall allows to see the other tunnel and the rocky environment behind it



Transparent Geometry

Blending equation:

$$C_{\text{final}} = a_{\text{src}} C_{\text{src}} + (1 - a_{\text{src}}) C_{\text{dst}}$$

`GL_SRC_ALPHA` `GL_ONE_MINUS_SRC_ALPHA`

↑ →

`glBlendFunc(srcFactor, dstFactor)`

C_{final} resulting output colour written to frame buffer

a_{src} source pixel alpha

C_{src} source pixel colour
(incoming pixel from transparent geometry)

C_{dst} destination pixel colour
(pixel already in frame buffer)

Transparent Geometry

Do it yourself:

- Blend transparent pixel t over pixel p in frame buffer to obtain new pixel p_{new}
- Use blending equation from previous slide

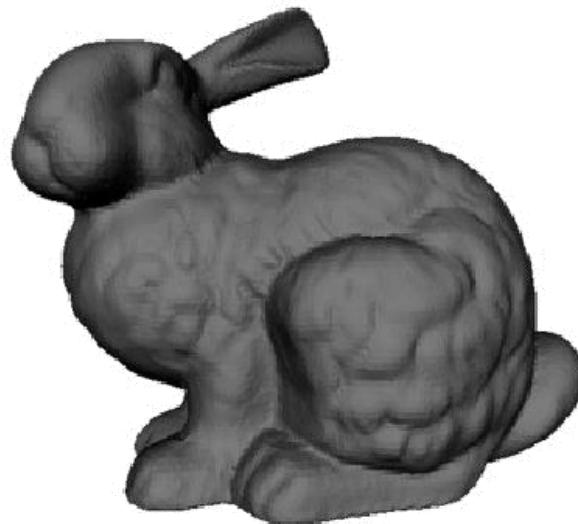
$$t = \begin{pmatrix} 1.0 \\ 0.0 \\ 0.0 \\ 0.2 \end{pmatrix} \quad p = \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix} \quad p_{\text{new}}?$$

Levels of Detail

Levels of Detail

Basic idea:

- Use simpler versions of objects as they make less and less contribution to the final image
- Use detailed version if the viewer is close, less detailed versions if the viewer is farther away



Ca. 70.000 triangles



1.000 triangles



100 triangles

Leach C. [A GPU-Based Level of Detail System for the Real-Time Simulation and Rendering of Large-Scale Granular Terrain](#)” (2014)

Levels of Detail

Due to distance, a simpler model looks approximately the same as a more detailed one

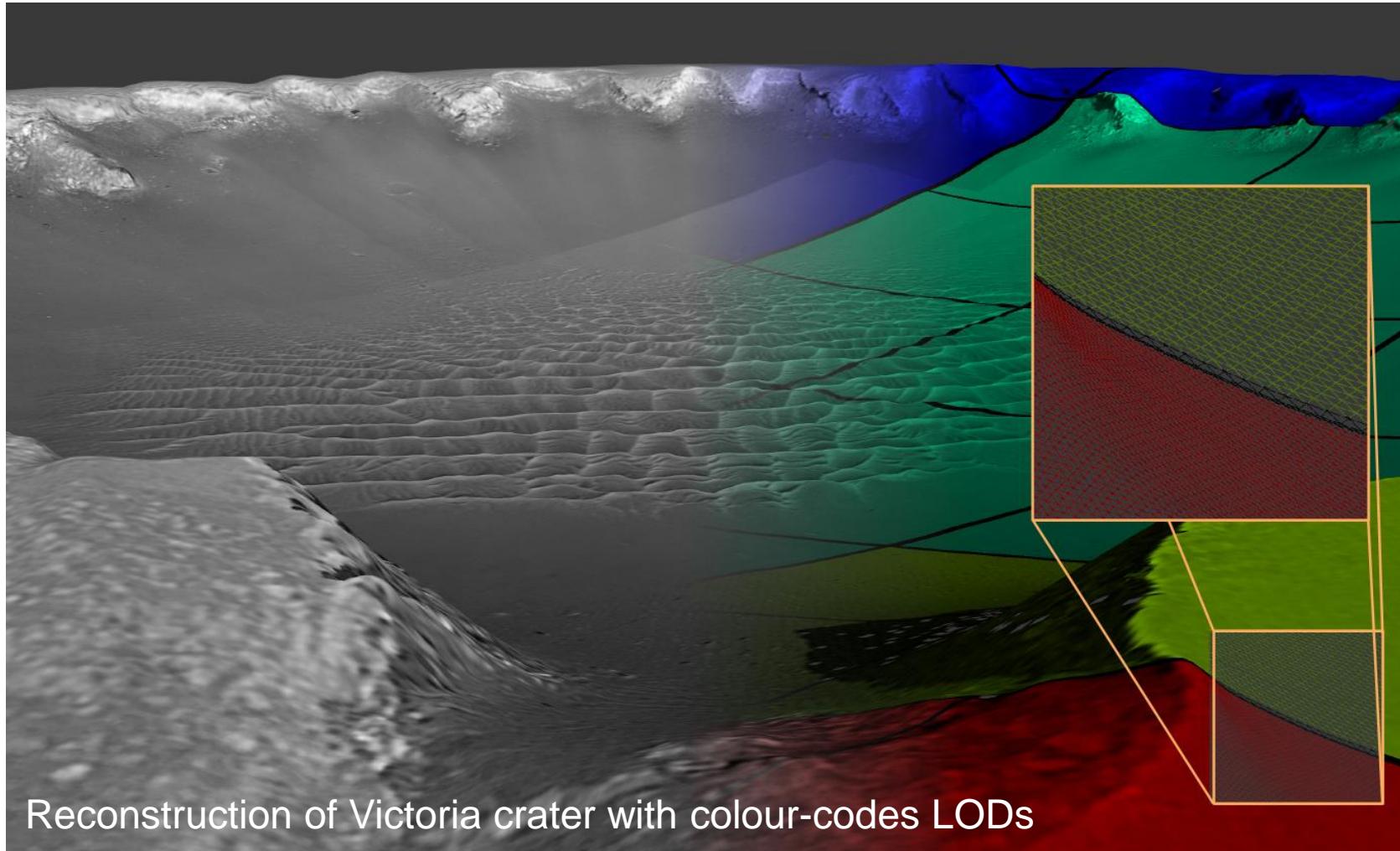
But, a significant speedup can be expected (less triangles need to be rendered)



Levels of Detail

LOD selection

- Range-based (viewing distance)
- Based on projected area
 - use lower resolution when seen from a glancing angle
 - Especially suitable for large terrains
- Combined metric



© NASA-JPL / MSSL / ASU / JOANNEUM Research / VRVis

Levels of Detail

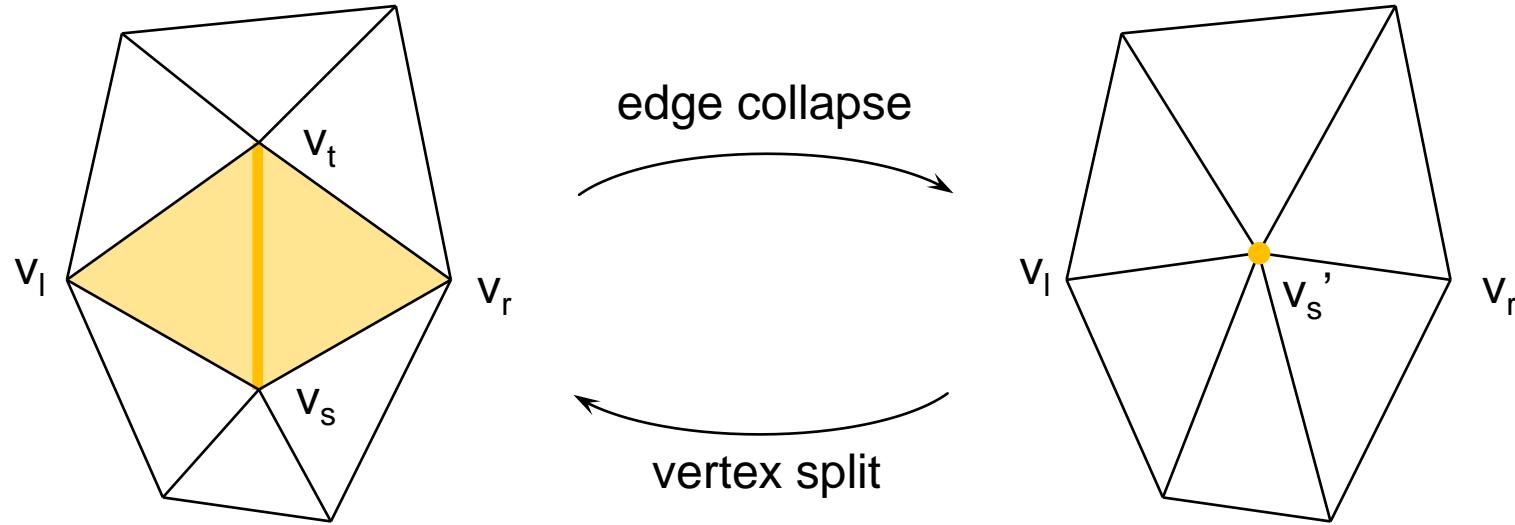
LOD switching

- **Discrete geometry LODs**
 - Prepare LODs (polygon reduction tools)
 - Instance particular geometry
 - Choose corresponding scene graph path (hierarchical LODs)
- **Blending LODs**
 - α -blending images between two levels
 - Disadvantage: Both levels must be rendered and ghosting artefacts occur
- **Continuous LODs (CLODs)**
 - Continuously variable spectrum of geometric detail
 - Locally vary detail
- **Geomorph LODs**
 - Vertex interpolation on GPU
 - Use tessellation shading
 - See progressive meshes (next slide)

Levels of Detail

Progressive meshes (Hoppe, SIGGRAPH 1996)

- Dynamically reduce original mesh with decimation algorithm
- Apply edge collapses and vertex splits to obtain LODs



- See [Hoppe's website](#) (talk and paper available)

Billboards and Imposters



University of
Applied Sciences

Billboards

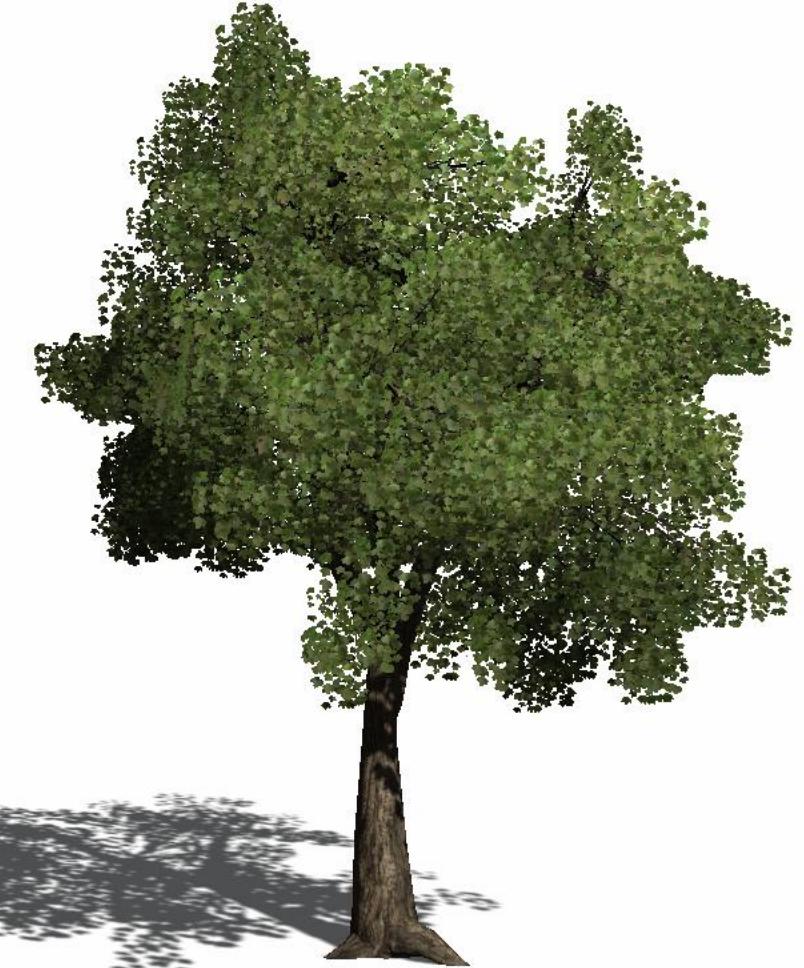
Idea: Extreme LOD to reduce all geometry to one or more textured polygons

Image-based rendering technique

Sometimes called sprites in game engines



Textures for leaves and small branches



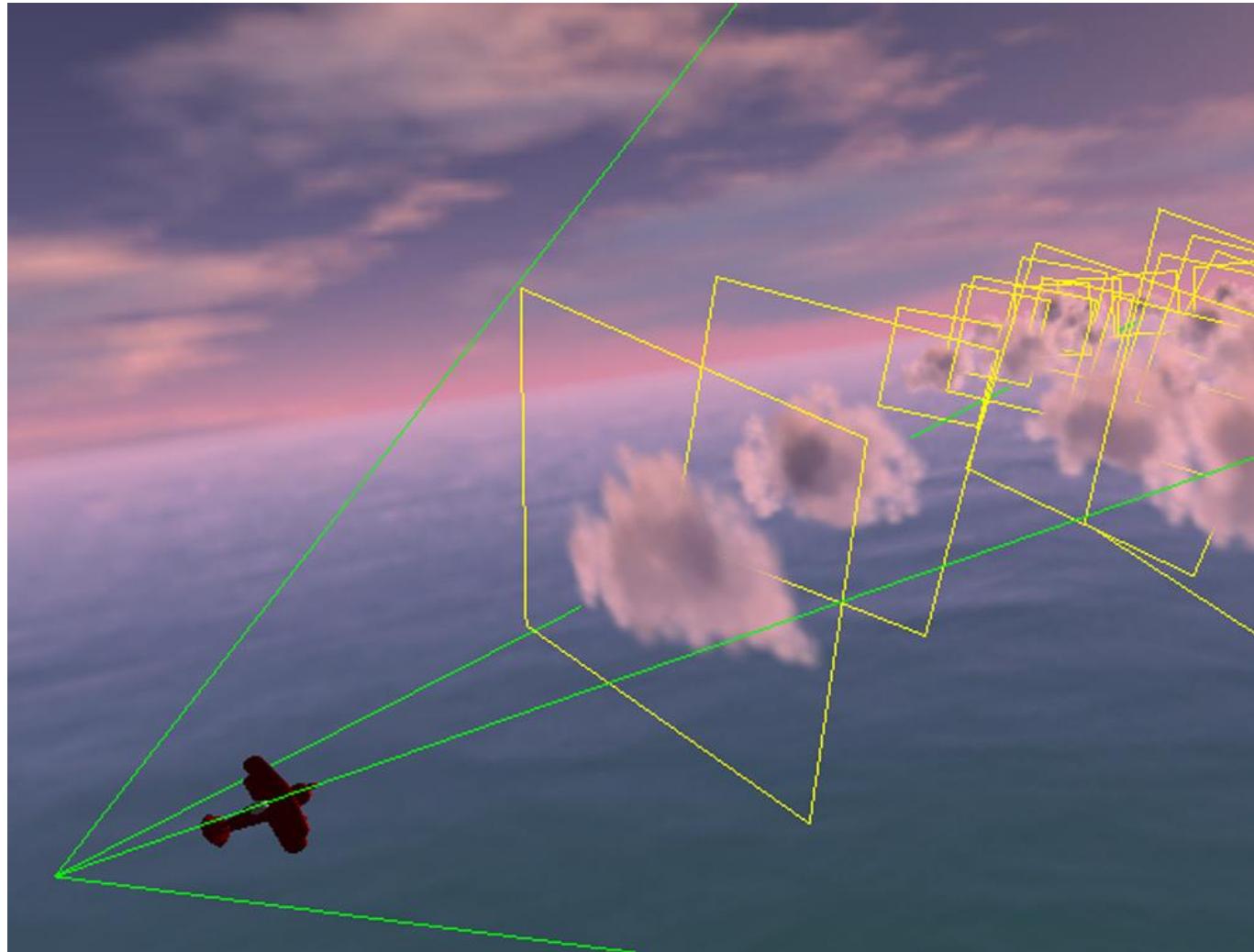
Tree model with billboard leaves

From Candussi et al. [Rendering Realistic Trees and Forests in Real Time](#).
Proc. of Eurographics 2005.

Imposters

Automatically generated billboards

- Render a complex model and use it as texture
- Background colour defines alpha
- Blend alpha at boundary for anti-aliasing
- Continuously view centred



From Harris and Lastra, “[Real-Time Cloud Rendering](#)”, in proc. of EUROGRAPHICS 2001, Volume 20 (2001), Number 3

Imposters

Align to view

$$D = A \times (V \times A)$$

$$\phi = \cos^{-1} \left(\frac{F \cdot D}{\|F\| \|D\|} \right)$$

D New imposter direction

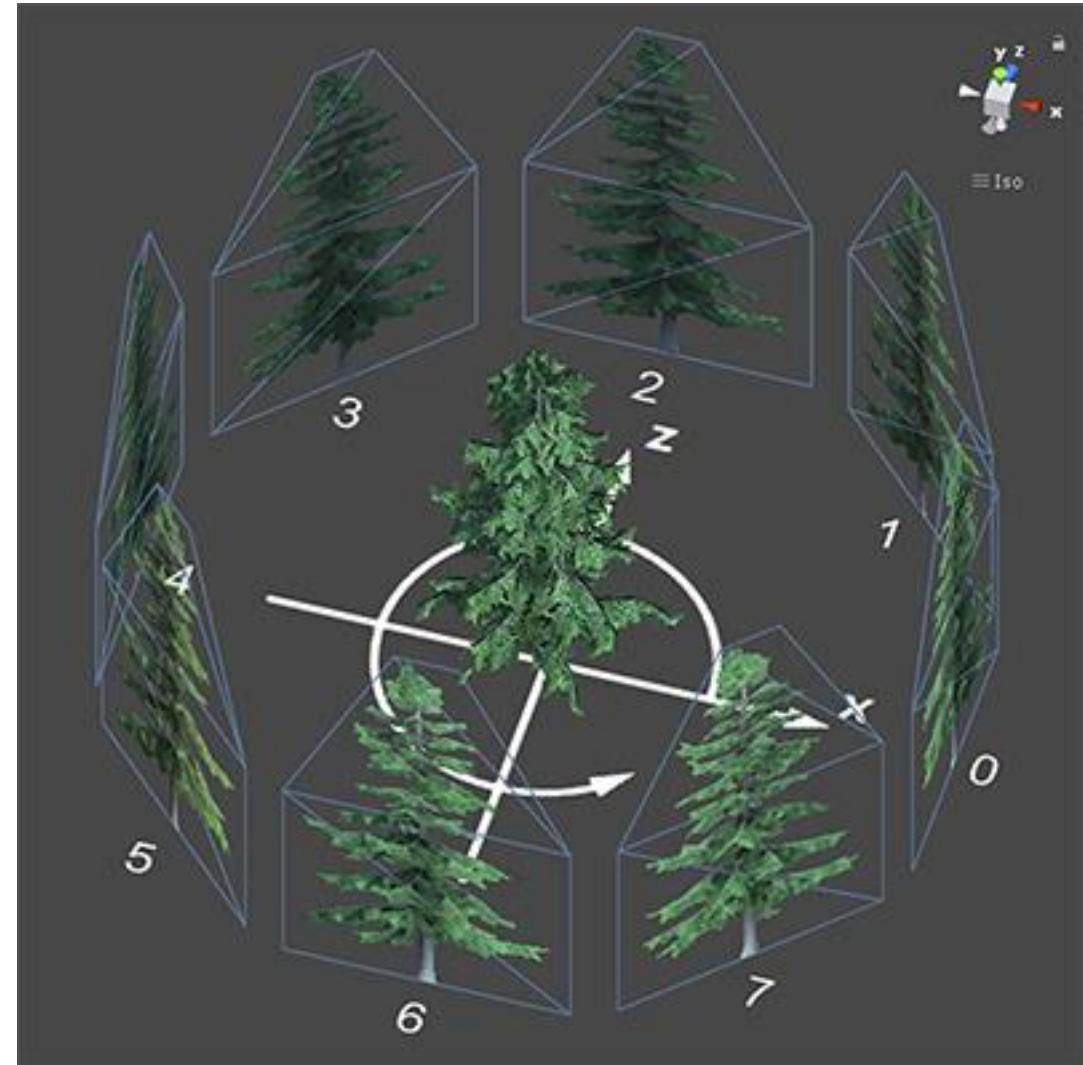
A Up vector of imposter

F Forward vector of imposter

V View direction

Imposters

- **Dynamical imposters**
 - For objects that are not rotationally symmetric
 - Compute multiple textures for multiple view points
 - Generate texture atlas
 - Select appropriate texture at runtime depending on viewing angle



© Unity 2020.3 documentation [Billboard Asset](#)

Billboard Clouds

Basic Idea

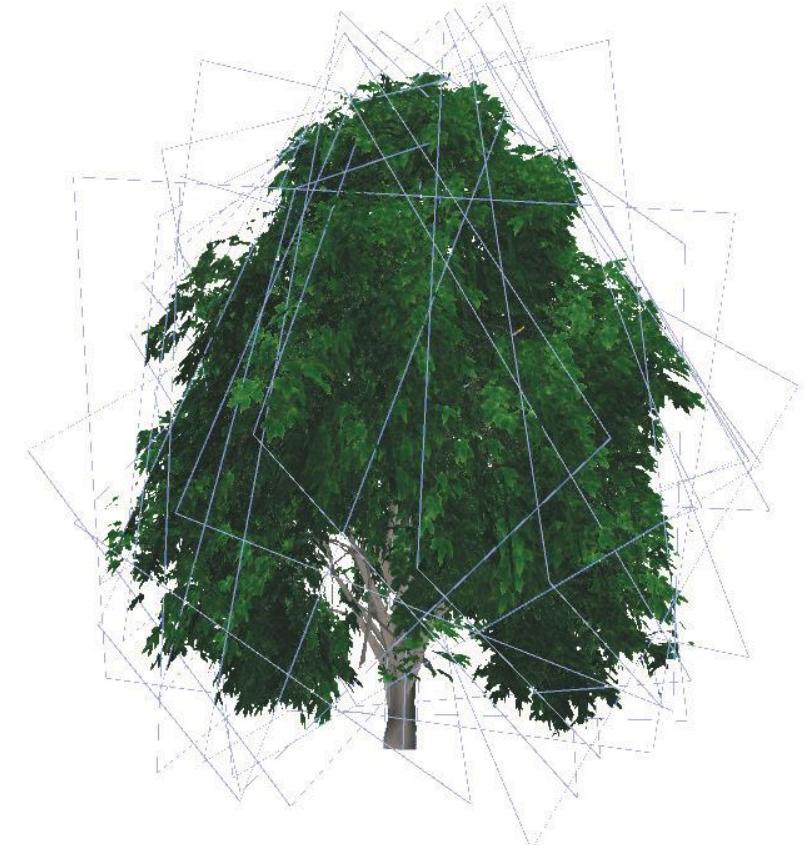
- 3D models are simplified onto a set of planes with texture and transparency

Pre-processing

- Find “optimal” planes using an optimization approach which tries to minimize the geometric error

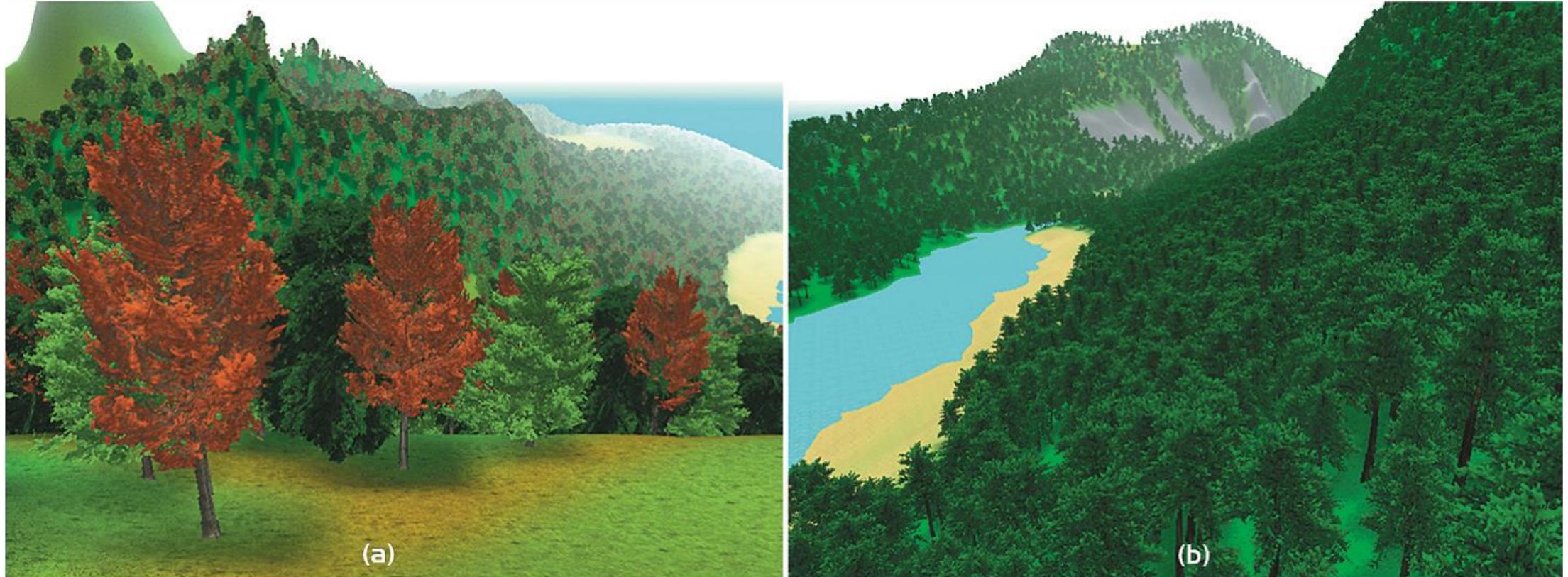
Details in

- Fuhrmann et al. [Extreme Model Simplification for Forest Rendering](#). Proceedings of the Eurographics Workshop on Natural Phenomena, NPH 2005, Dublin, Ireland, 2005.



Billboard Clouds

Real-time rendering of highly complex scenes (e.g. forest)

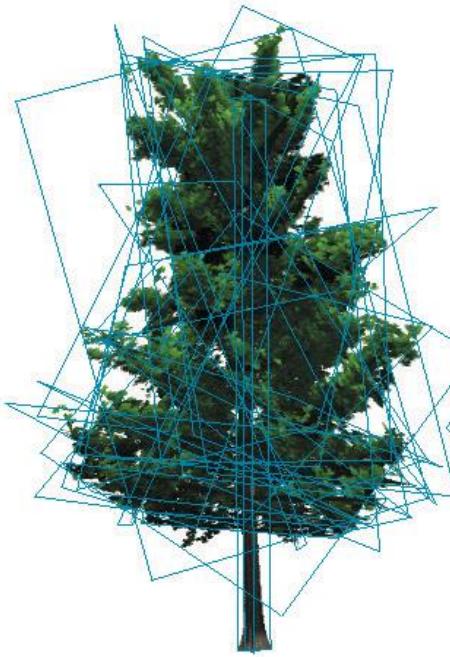


a) 80.000 trees, b) 50.000 trees

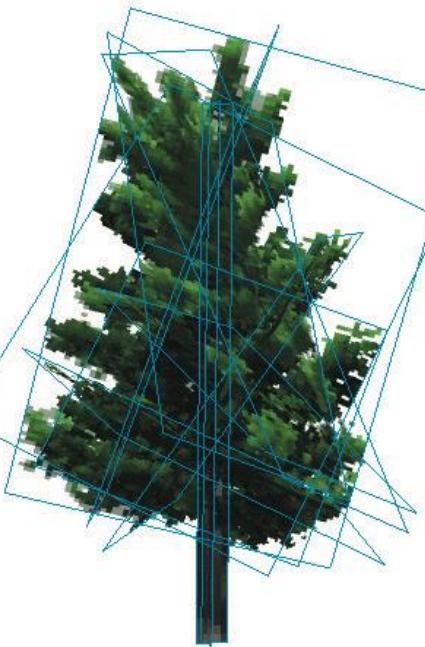
Billboard Clouds

LODs for billboard clouds

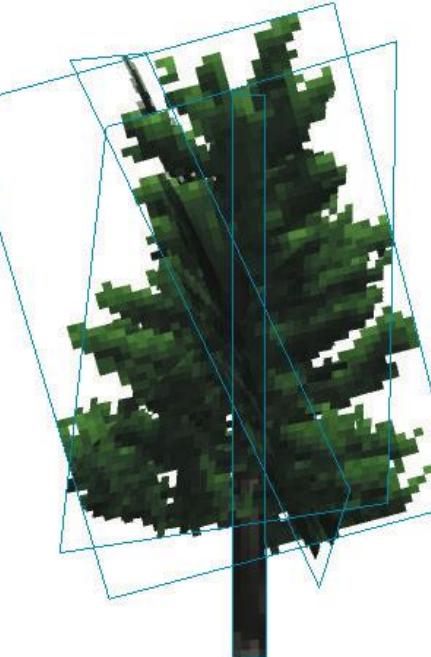
- Reduce number of planes depending on distance



24 planes



11 planes



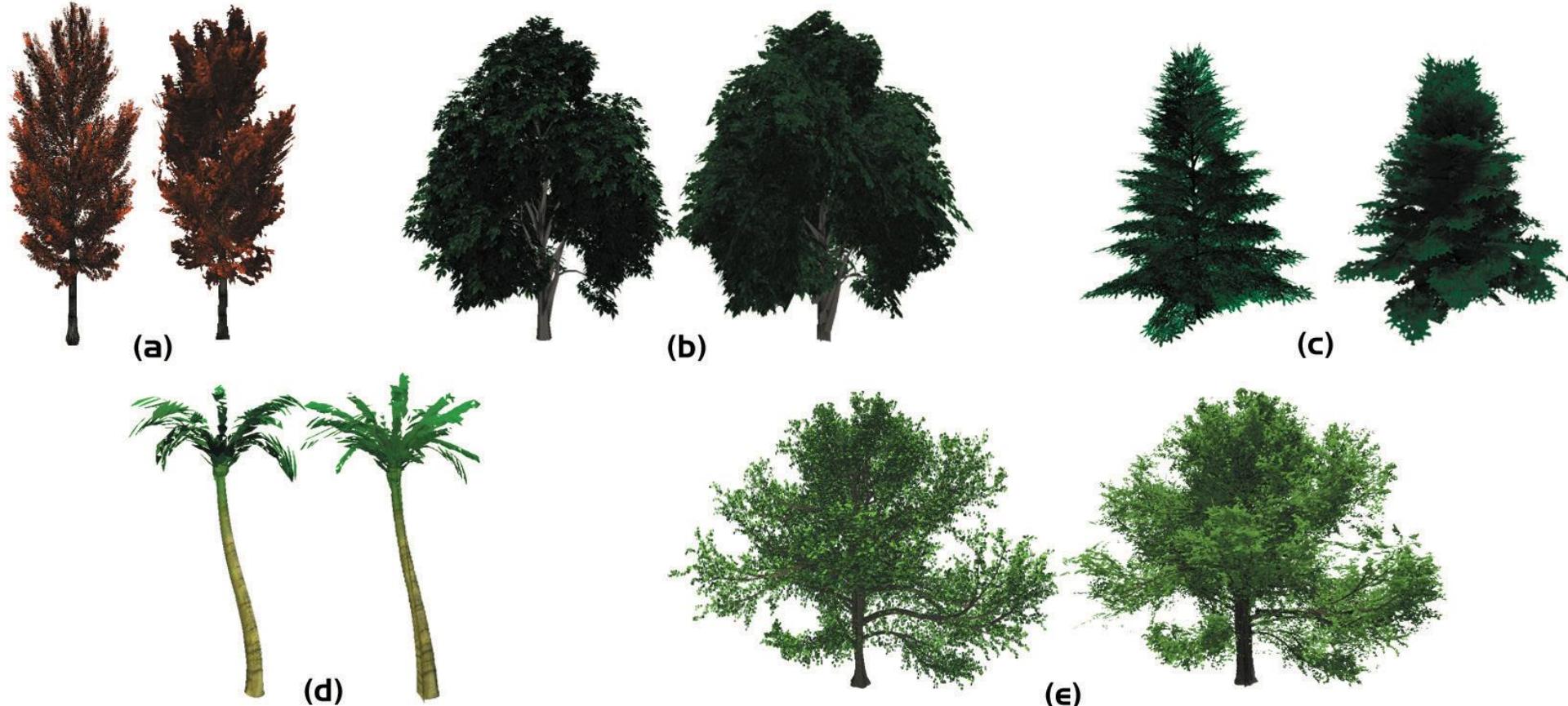
4 planes



Suitable display sizes

Billboard Clouds

Visual comparison of polygonal models (left images) vs. billboard clouds (right images)



Billboard Clouds

Benchmarks for the models shown on previous slide

Tree	ϵ	Faces	Billboards	Pre-processing time (s)	approx. trees / s
a	10.0	108.782	12	342	143.000
b	12.0	159.160	14	403	122.000
c	12.5	20.547	13	62	132.000
d	6.5	7.292	8	25	214.000
e	6.5	169.781	21	496	81.000

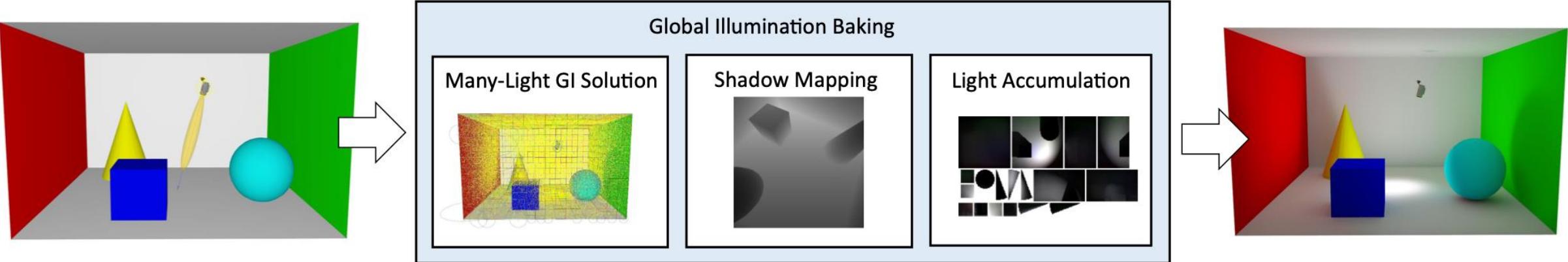
Light Maps

Light Maps

Global illumination effects for real-time rendering

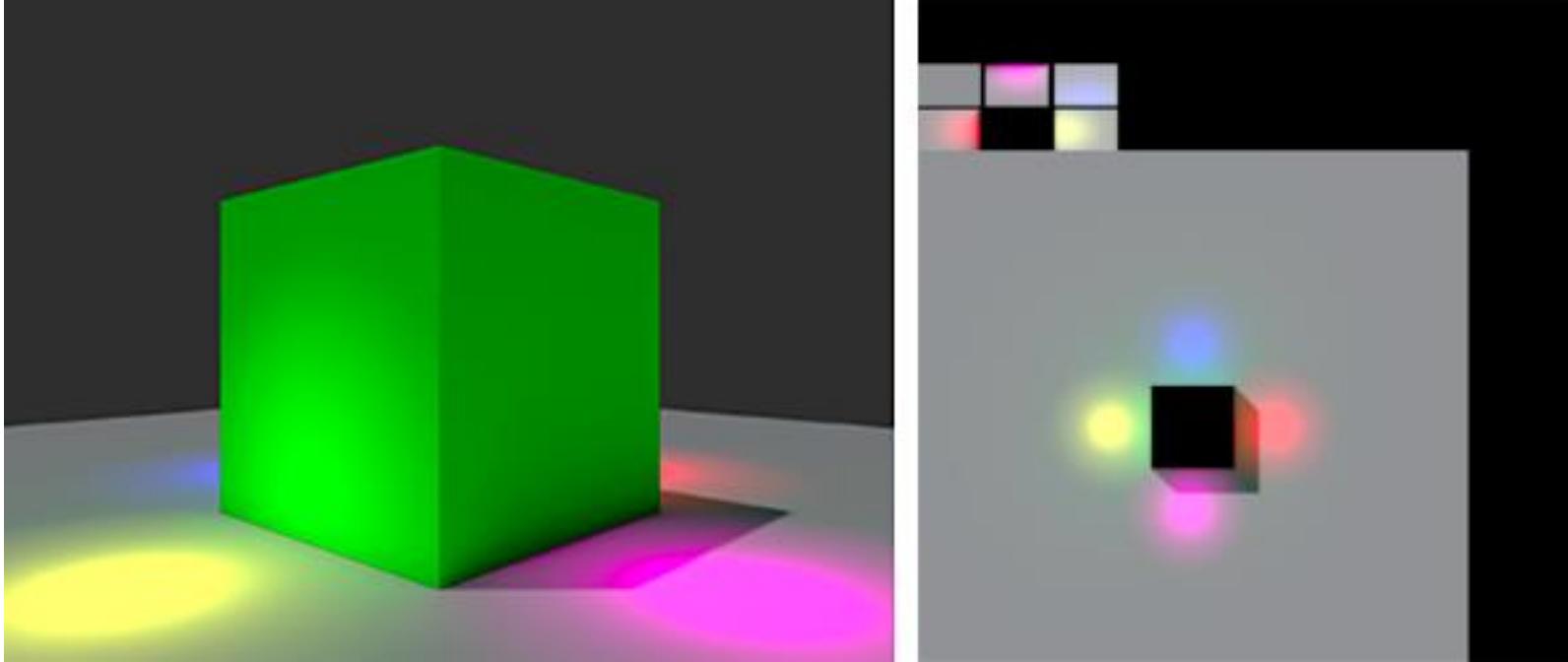
Store pre-calculated illumination of surfaces in textures

Works well for all view independent lighting effects and static objects



Light Mapping

Simple light map example including baked indirect illumination and soft shadows



© Unity 2020.3 documentation [Light Mapping](#)

Video – Light mapping with Unity game engine

www.youtube.com/watch?v=okYhs6kQ0xw

Picking

Picking

Select objects in 3D with mouse or laser pointer in VR

Pick concrete surface point for certain interactions, e.g.:

- Push an object
- Grab a handle
- Press a 3D button
- Draw on surface (i.e. make annotations)
- Carry out measurements

Picking

Done with ray casting

- Ray is defined by mouse position and restricted between near and far clipping plane
- In VR a laser pointer attached to a controller directly defines the ray
- Same intersection algorithms as for ray tracing
- See lecture 7 “Ray Tracing 1”

Picking

Optimization

- As with ray tracing we must minimize the ray-primitive intersection test

Large Static objects

- kd-trees are the most efficient data structure
- E.g. terrain and city models, point clouds

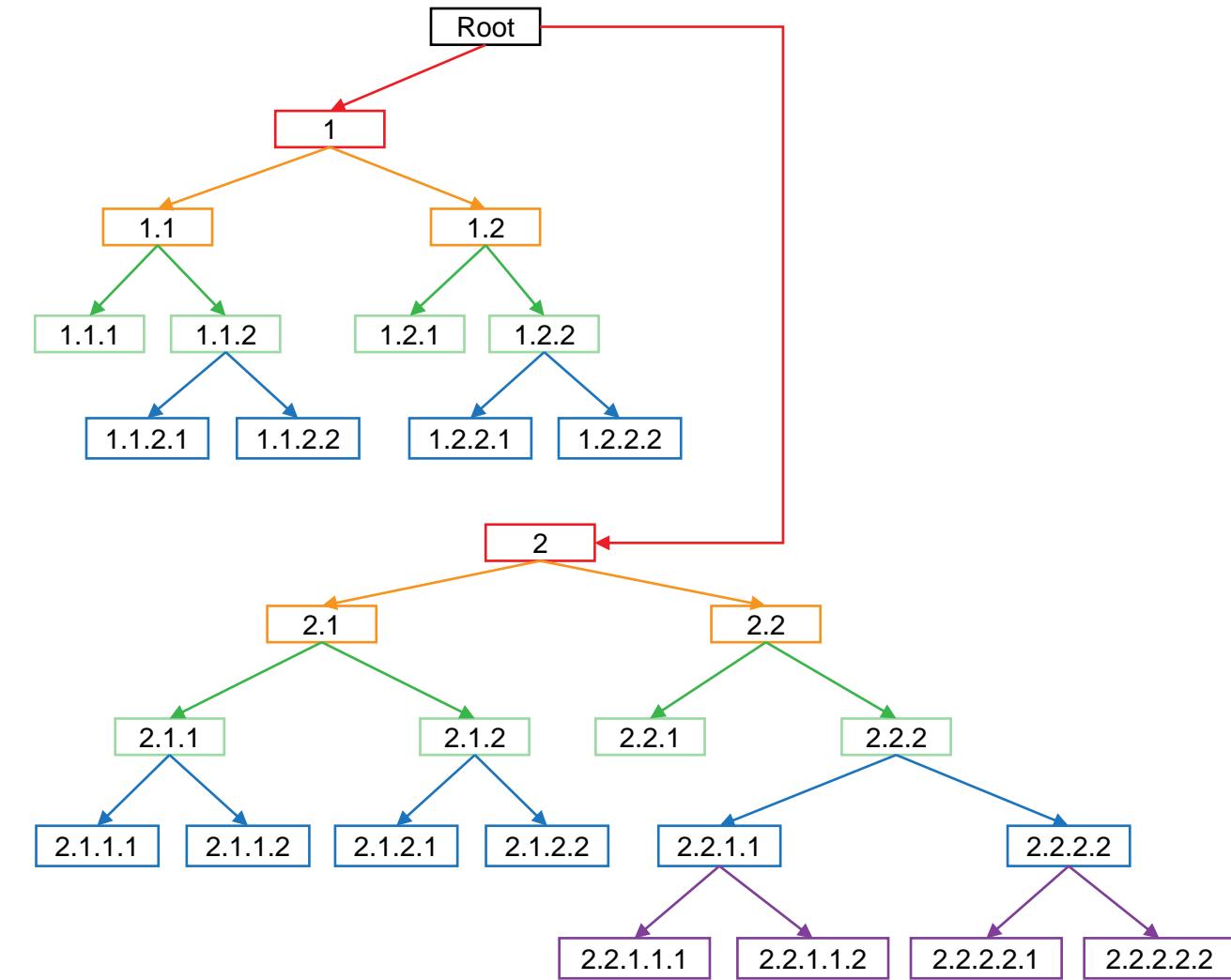
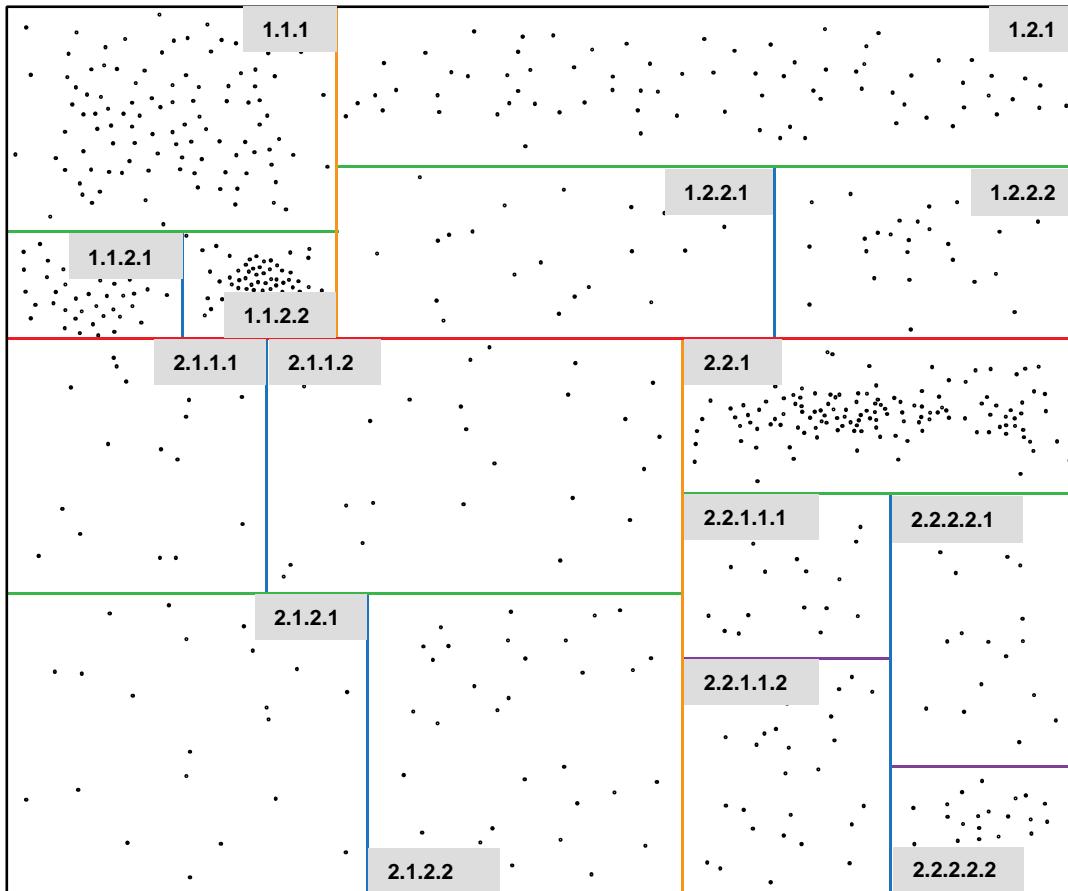
Dynamic objects

- Bounding volumes or separate kd-trees (for more complex objects)

Picking

kd-tree example for point cloud

Works the same way for meshes



Picking

kd-tree example - traversal path for mouse position

