

Image Processing and Computer Graphics

Computer Graphics

Matthias Teschner

Computer Science Department
University of Freiburg

Albert-Ludwigs-Universität Freiburg



Outline

- organization
- research of the graphics group
- rendering pipeline

Organization

- class
 - 082 006: Monday 10-12, Tuesday 10-12
 - Prof. Matthias Teschner
- exercises
 - 082 021, 028, 029: Tuesday 10-12
 - tba
- check web page for the exact schedule
 - <http://cg.informatik.uni-freiburg.de/teaching.htm>
- two parts
 - computer graphics
 - image processing (starts on Dec 4)

Contact

- Prof. Matthias Teschner
 - teschner@informatik.uni-freiburg.de
 - 052 / 01-005
- tba
 - tba

Exercises / Exam

- exercises
 - Nov 7, Nov 14, Nov 21,
 - practical exercises
 - check web page for information
 - processing is optional, but recommended
 - use of the provided source code is optional
- exam
 - written exam
 - test exam

Course Goals

- introduction to the fundamentals of rasterization-based image generation
- functionality of the graphics rendering pipeline
- advanced rendering effects
- introduction to the OpenGL graphics API
- requirements
 - C / C++
 - basics in linear algebra

Slide Sets

- slide sets, exercises and solutions on
<http://cg.informatik.uni-freiburg.de/teaching.htm>

Material

- T. Akenine-Möller, E. Haines:
Real-time Rendering
A. K. Peters Ltd.,
<http://www.realtimerendering.com>



Further Readings

- D. F. Rogers:
Procedural Elements of Computer Graphics
McGraw-Hill, 1997
- A. Watt: **3D Computer Graphics**
Addison-Wesley, 1999
- J. Foley, A. van Dam, S. Feiner, J. Hughes:
Computer Graphics – Principles and Practice
Addison-Wesley, 1990
- J. Encarnacao, W. Strasser, R. Klein:
Graphische Datenverarbeitung
Oldenburg Verlag, 1996

Syllabus

- Oct 16 - Rendering Pipeline
- Oct 17 - OpenGL
- Oct 23 - Transformations
- Oct 24 - Projections
- Oct 30 - Lighting
- Nov 6 - Lighting
- Nov 7 - *Exercise*
- Nov 13 - Rasterization
- Nov 14 - *Exercise*
- Nov 20 - Shadows
- Nov 21 - *Exercise*
- Nov 27 - Texturing
- Nov 28 - Transparency, Reflection
- tba - Evaluation, Q & A

Course Information

- key course
 - pattern recognition and computer graphics
(rasterization-based rendering)
- specialization courses
 - advanced computer graphics (ray tracing)
 - simulation in computer graphics (e.g., fluids)
- master project, lab course, Master thesis
 - two tracks: simulation, rendering

Seminars / Projects / Theses in Graphics

Semester	Simulation Track	Rendering Track
Winter	Rasterization Course Simulation Course	Rasterization Course
Summer	Lab Course - simple fluid solver Simulation Seminar	Raytracing Course Lab Course - simple raytracer
Winter	Master Project - PPE fluid solver	Master Project - Monte Carlo raytracer Rendering Seminar
Summer	Master Thesis - research-oriented topic	Master Thesis - research-oriented topic

Computer Graphics Research

Matthias Teschner

Computer Science Department
University of Freiburg

Albert-Ludwigs-Universität Freiburg



Research Topics

- physically-based animation and rendering of
 - rigid bodies
 - deformable objects
 - fluids



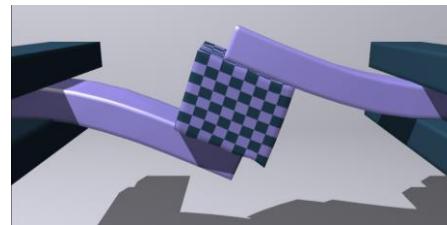
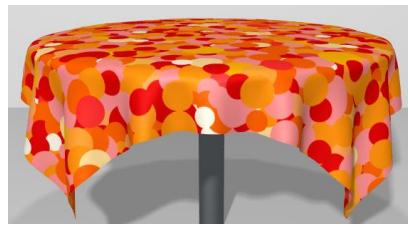
rigid bodies



fluids

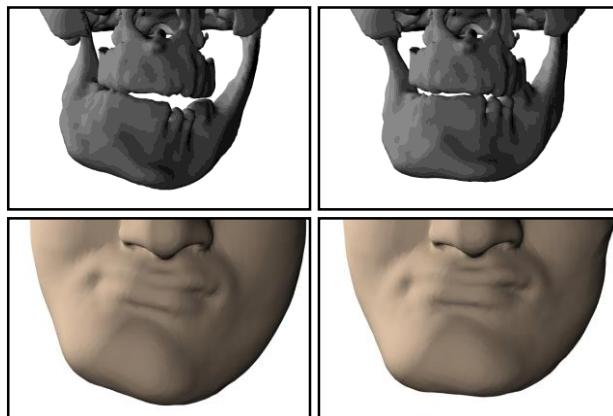


deformable objects

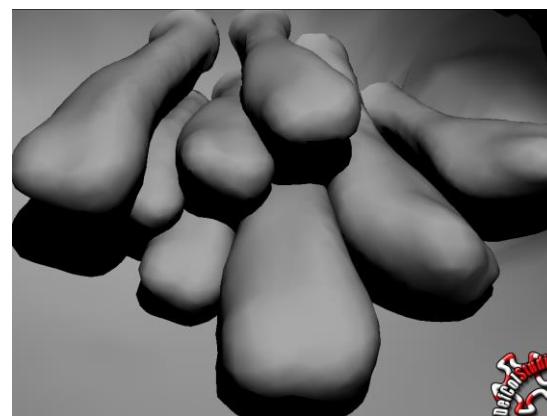


Applications

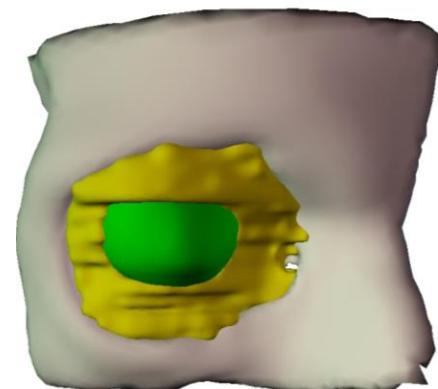
- computational medicine



pre-operative planning in
cranio-maxillofacial surgery



interactive hysteroscopy
simulation for educational
purposes



intra-operative support
in orbital reconstruction

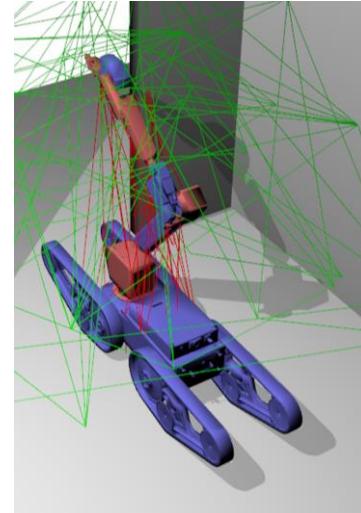
Applications

- robotics

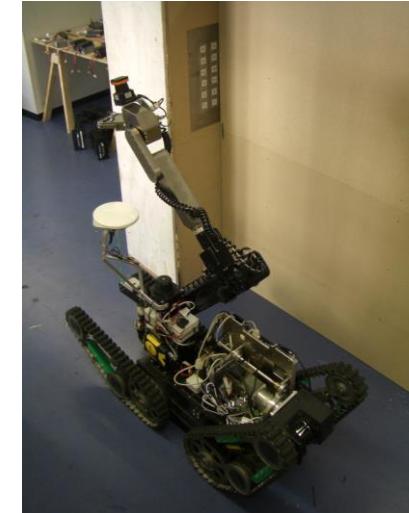


support of robot navigation
with simulation environments

DFG SFB TR8 (with Wolfram Burgard and Bernhard Nebel, University of Freiburg)



generation of virtual
environments using robots



Applications (with Pixar)



10 million fluid + 4 million rigid particles, 50 s simulated,
50 h computation time on a 16-core PC, www.youtube.com/cgfreiburg

Applications (with FIFTY2 Technology)

PreonLab: Drive Through



PreonLab, FIFTY2 Technology GmbH, [www.youtube.com -> fifty2](http://www.youtube.com->fifty2)

University of Freiburg – Computer Science Department – Computer Graphics - 6

Computer Graphics Research

Matthias Teschner

Computer Science Department
University of Freiburg

Albert-Ludwigs-Universität Freiburg



Image Processing and Computer Graphics

OpenGL

Matthias Teschner

Computer Science Department
University of Freiburg

Albert-Ludwigs-Universität Freiburg

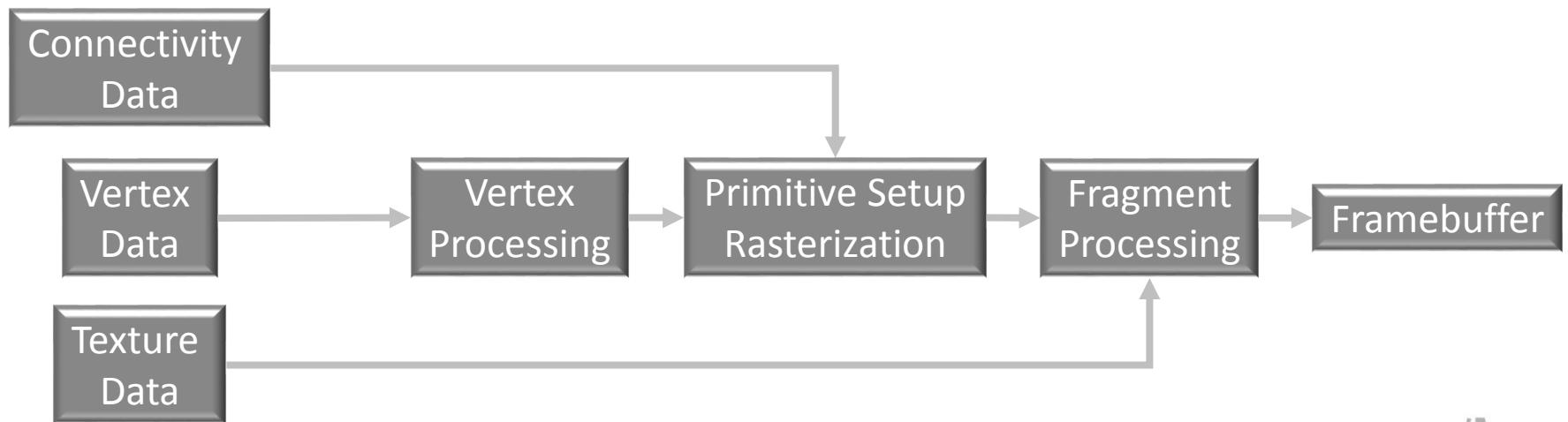


Introduction

- OpenGL is a graphics rendering API
 - display of geometric representations and attributes
 - independent from operating system and window system
- OpenGL realizes the interaction with GPUs
 - hardware-accelerated rendering

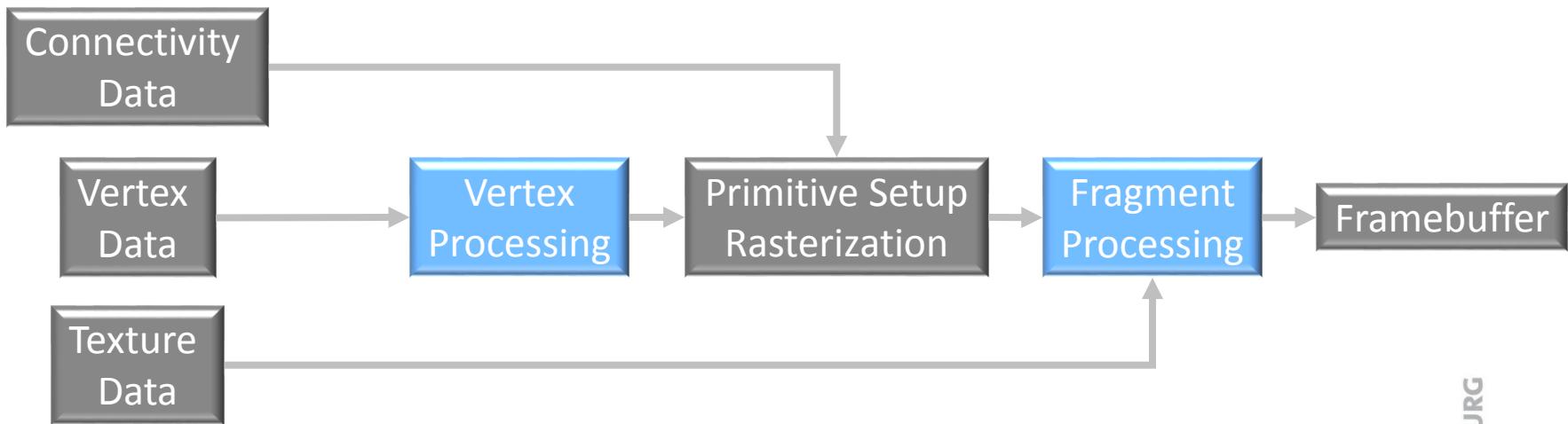
OpenGL 1.0 (1994)

- fixed-function pipeline
- focus on parallelized implementation
- promoted by quasi-standards of all components of a rasterization-based renderer



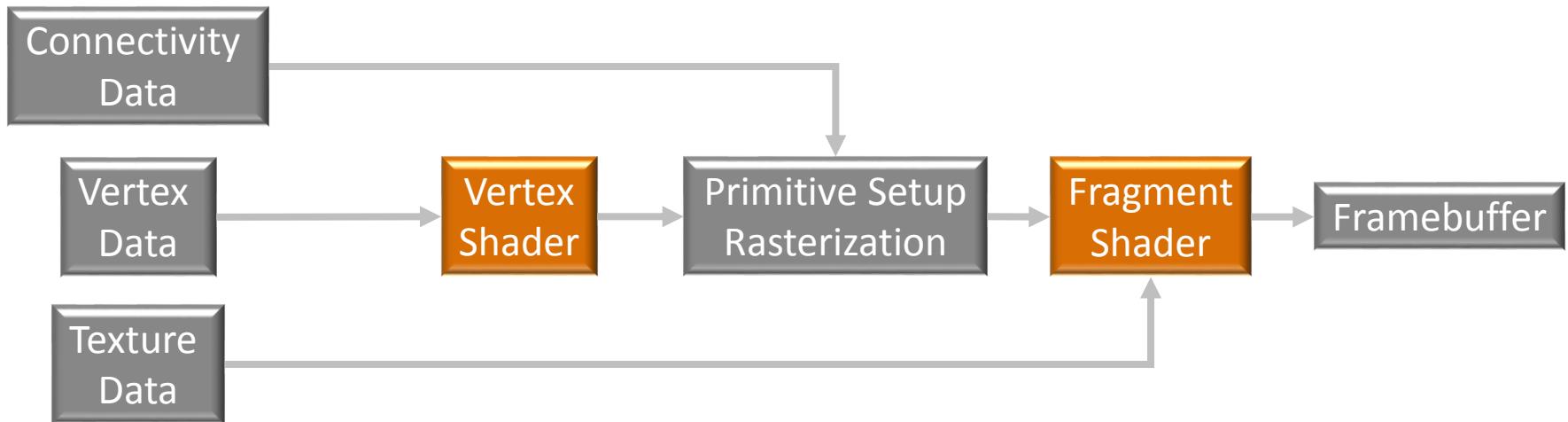
OpenGL 2.0

- fixed-function pipeline with programmable vertex and fragment processing
 - vertex and fragment processing could be replaced by user-defined functionality (shaders)
 - shaders are programs that work on each vertex / fragment



OpenGL 3.0

- programmable vertex and fragment processing
- no fixed-function pipeline
 - vertex and fragment shaders have to be implemented



OpenGL 3.0

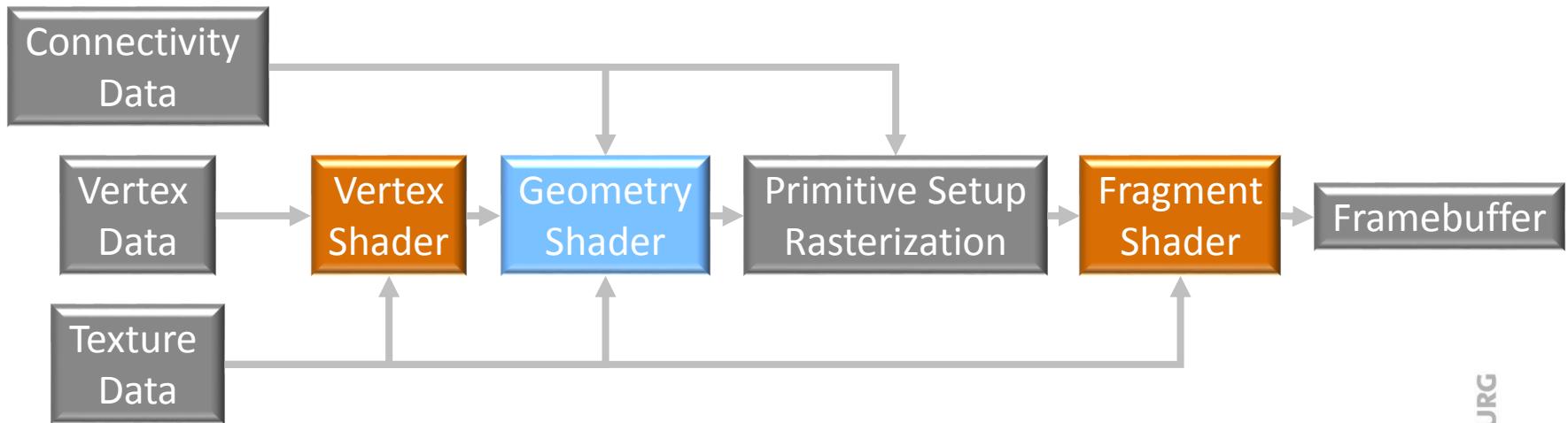
- focus on core functionality
 - removal of OpenGL features
 - deprecation model (full, forward compatible)
- improved handling of large data (buffer objects)
- improved flexibility
 - implementation of non-standard effects not restricted to “misusing” pipeline functionality
- programming
 - not just setting parameters of standard functionality
 - concepts of vertex and fragment processing are not “nice to know”, but required knowledge
 - e.g., transforms, projections

OpenGL 3.0

- deprecated features, e.g., **glRotate**
 - generates a transformation matrix
 - multiplies the matrix with the top element of the current stack
- typically replaced by OpenGL Mathematics glm
 - **glm::Rotate**
 - generates a transformation matrix

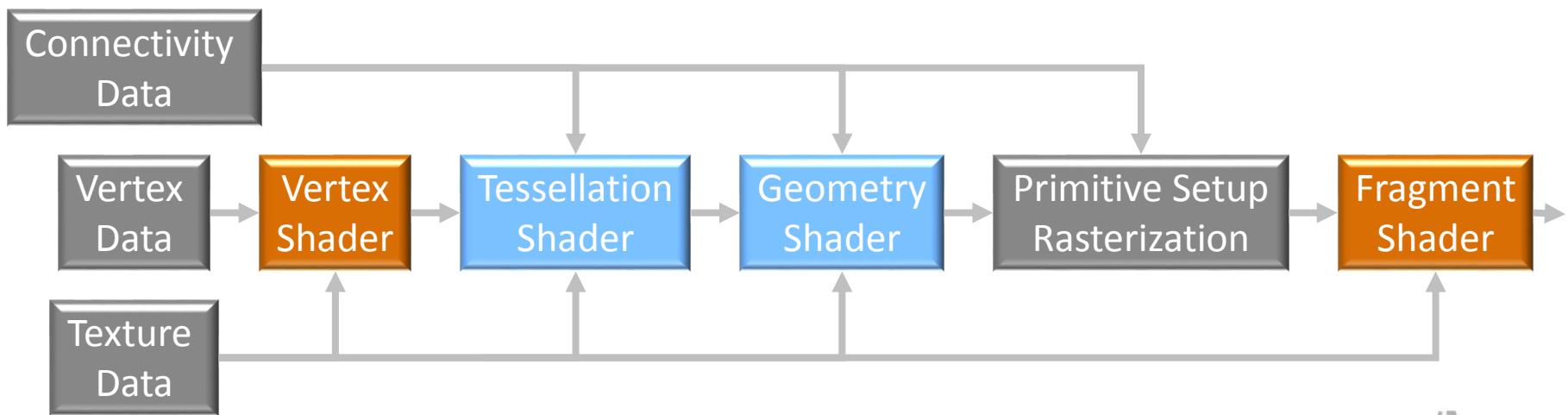
OpenGL 3.2

- geometry shader
 - optional
 - modify geometric primitives
 - generation of geometry no longer restricted to CPU
- flexible use of texture data



OpenGL 4.1

- tessellation shader
 - optional
 - tessellate patches
 - flexible generation of large and detailed geometries



OpenGL 4.3

- compute shader
 - optional
 - perform arbitrary computations
 - are not part of the rendering pipeline



GPU Data Flow

- data transfer to GPU
 - vertices with attributes and connectivity
- vertex shader
 - a program that is executed for each vertex
 - input and output is a vertex
- rasterizer
- fragment shader
 - a program that is executed for each fragment
 - input and output is a fragment
- framebuffer update

Data Transfer

- Vertex Buffer Object VBO
 - used to copy memory from CPU to GPU
 - contains arbitrary data, typically vertex attributes

```
GLuint gVBO = 0;  
glGenBuffers(1, &gVBO);  
 glBindBuffer(GL_ARRAY_BUFFER, gVBO);
```

```
GLfloat vertexData[] = {  
    // X      Y      Z  
    0.0f, 0.8f, 0.0f,  
   -0.8f,-0.8f, 0.0f,  
    0.8f,-0.8f, 0.0f};
```

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertexData),  
vertexData, GL_STATIC_DRAW);
```

[Tom Dalling]

Data Transfer

- Vertex Array Object VAO
 - link between VBO and shader programs
 - specifies how to interpret VBO data
 - specifies the mapping to input variables of shaders

```
GLuint gVAO = 0;  
glGenVertexArrays(1, &gVAO);  
 glBindVertexArray(gVAO);  
  
// connect the xyz to the "vert" attribute  
// of the vertex shader  
  
 glEnableVertexAttribArray(gProgram->attrib("vert"));  
 glVertexAttribPointer(gProgram->attrib("vert"), 3,  
 GL_FLOAT, GL_FALSE, 0, NULL);
```

[Tom Dalling]

Shader

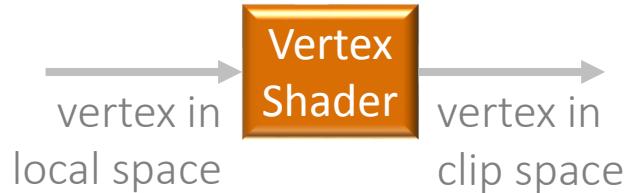
- program
 - written in OpenGL Shading Language GLSL
 - runs on the GPU
 - vertex and fragment shader are mandatory

	Main Program	Shader Program
Language	C++	GLSL
Main function	<code>int main(int, char**);</code>	<code>void main();</code>
Runs on	CPU	GPU
Gets compiled?	yes	yes
Gets linked?	yes	yes

[Tom Dalling]

Vertex Shader

- works on vertices
 - input and output are vertices
- minimum functionality
 - transformation from local to clip space
(after clipping, rasterizer only works on vertices in the canonical view volume [-1..1, -1..1, -1..1])



Simple Vertex Shader Example

- ```
#version 150

in vec3 vert;

void main() {
 // does not alter the vertices at all
 gl_Position = vec4(vert, 1);
}
```
- model, view and projection transform are implicitly set to identity matrices

$$\begin{pmatrix} x_{\text{clip}} \\ y_{\text{clip}} \\ z_{\text{clip}} \\ 1 \end{pmatrix} = \mathbf{I} \begin{pmatrix} x_{\text{local}} \\ y_{\text{local}} \\ z_{\text{local}} \\ 1 \end{pmatrix}$$

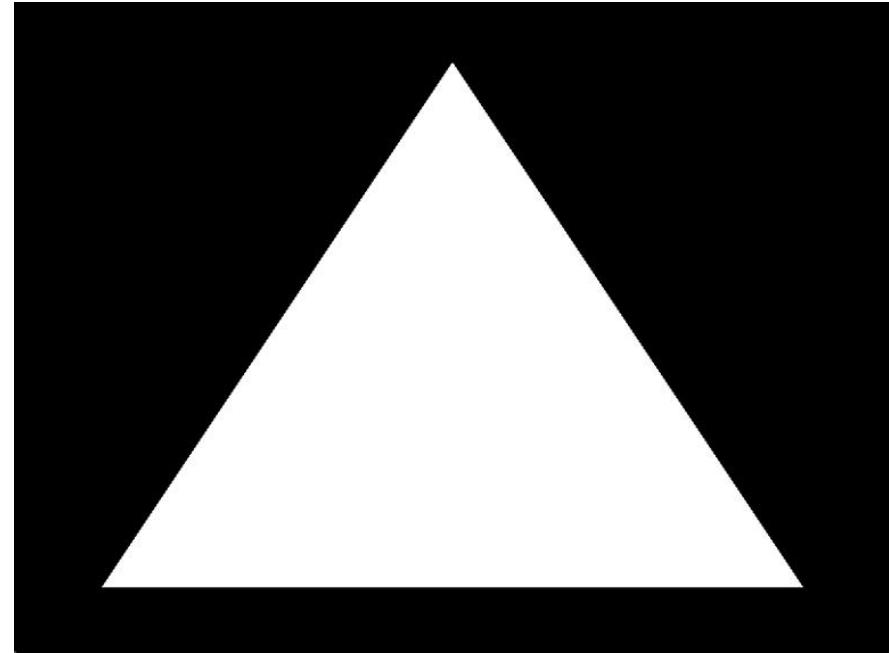
`gl_Position` is a built-in output variable of a vertex shader. It is a 4D vector ( $x,y,z,1$ ) representing the clip-space position of a vertex

[Tom Dalling]

# *Simple Vertex Shader Example*

---

- ```
GLfloat vertexData[] = {  
    // X      Y      Z  
    0.0f, 0.8f, 0.0f,  
    -0.8f,-0.8f, 0.0f,  
    0.8f,-0.8f, 0.0f};
```
- results in a visible triangle for the example shader as all input/output vertex positions are within the canonical view volume



[Tom Dalling]

Typical Vertex Shader Example

- `uniform mat4 projection;`
`uniform mat4 camera;`
`uniform mat4 model;`
`in vec3 vert;`

set in the main program

read from VAO

```
void main() {  
    gl_Position = projection * camera * model * vec4(vert, 1);  
}
```

internal
camera
parameters

camera
place-
ment

object
place-
ment

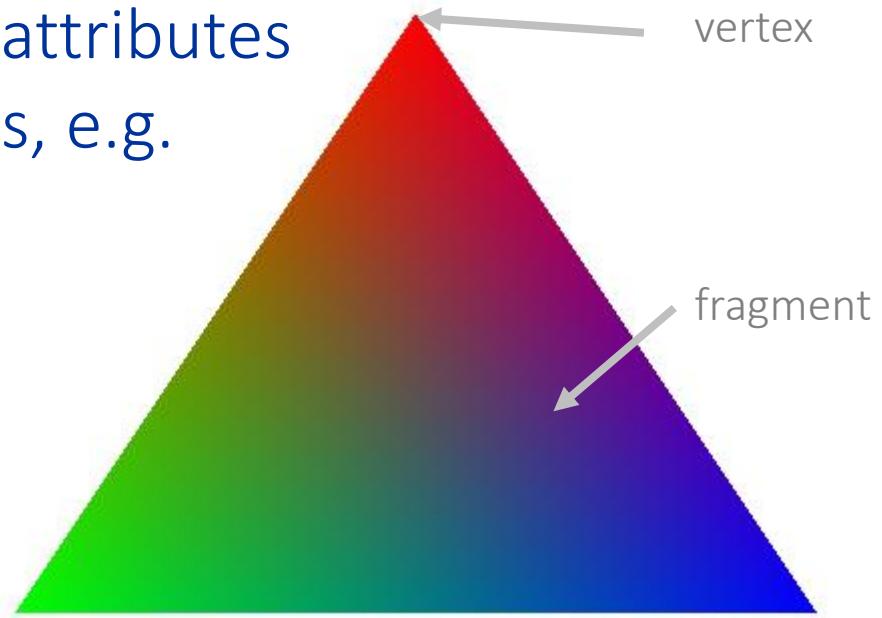
- `glm::mat4 projection = glm::perspective(...);`
`gProgram->setUniform("projection", projection);`

```
glm::mat4 camera = glm::lookAt(glm::vec3(3,3,3),  
                                glm::vec3(0,0,0), glm::vec3(0,1,0));  
gProgram->setUniform("camera", camera);
```

[Tom Dalling]

Vertex Shader

- can compute/set a color at vertices
 - e.g. red, green, blue
- rasterizer can interpolate attributes from vertices to fragments, e.g.



result for an empty fragment shader
employing the interpolation
functionality of the rasterizer

Simple Fragment Shader Example

- `#version 150`

```
out vec4 finalColor;
```

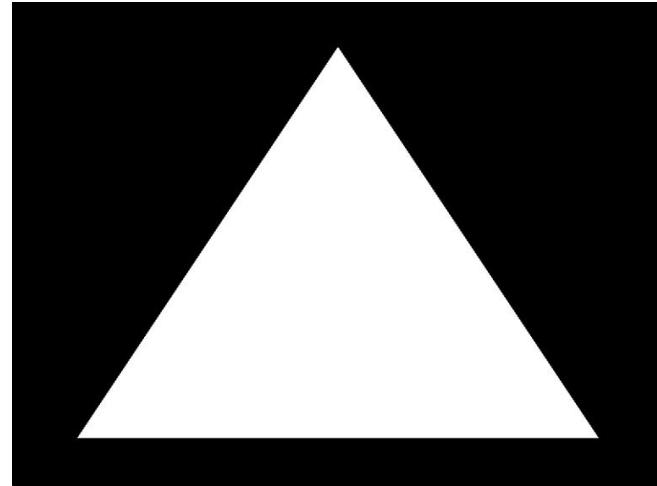
```
void main() {
```

```
    // set every drawn pixel to white
```

```
    finalColor = vec4(1.0, 1.0, 1.0, 1.0);
```

```
}
```

- if not set-up otherwise,
the output is written to
the color buffer



[Tom Dalling]

Typical Fragment Shader Example

- `#version 150`

```
in ...
out vec4 finalColor;
```

incomplete shader
for Phong illumination

```
void main() {
    //calculate the vector from pixel to light source
    vec3 surfaceToLight = light.position - fragPosition;

    //calculate the cosine of the angle of incidence
    float brightness = dot(normal, surfaceToLight) /
        (length(surfaceToLight) * length(normal));
    brightness = clamp(brightness, 0, 1);

    //calculate final color of the pixel, based on:
    finalColor = vec4(brightness * light.intensities *
        surfaceColor.rgb, surfaceColor.a);
```

[Tom Dalling]

}

GPU Data Flow

- data transfer to GPU
 - VBOs store data, VAOs interpret the data
 - vertices with attributes and connectivity
- vertex shader
 - input is a vertex in local space
 - output is a vertex in clip space
- rasterizer
 - generates fragments
 - interpolates attributes from vertices to fragments
- fragment shader
 - output is a fragment color

OpenGL Setup

- implementations are typically accomplished by additional libraries
 - OpenGL Extension Wrangler GLEW
 - access to OpenGL x.x API functions
 - GLFW
 - windowing, mouse and keyboard handling
 - OpenGL Mathematics GLM
 - processes vectors and matrices
- implementation
 - fragment shader
 - vertex shader

Image Processing and Computer Graphics

Rendering Pipeline

Matthias Teschner

Computer Science Department
University of Freiburg

Albert-Ludwigs-Universität Freiburg



Outline

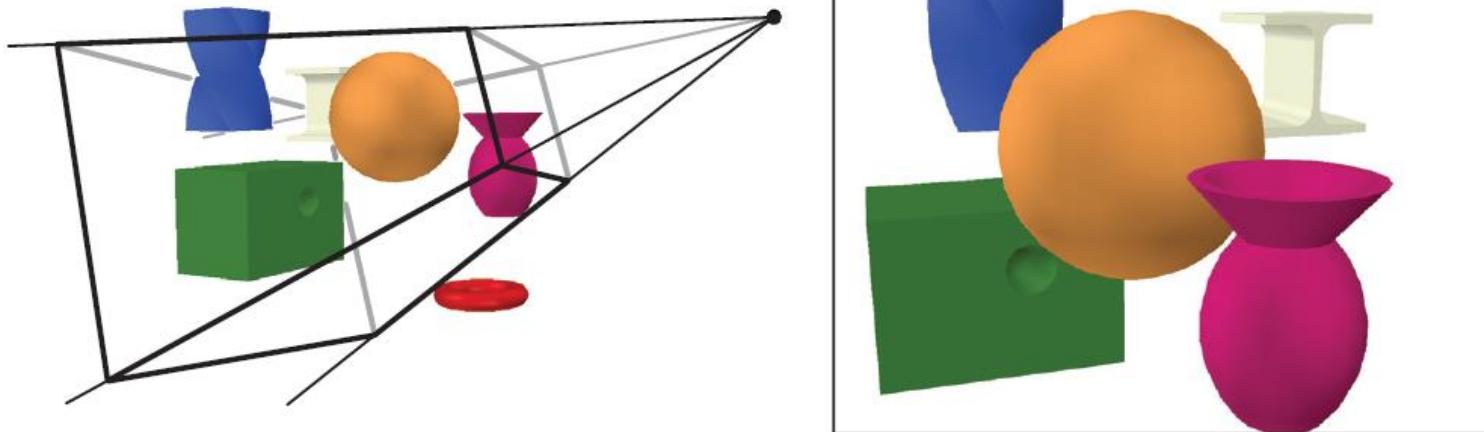
- introduction
- rendering pipeline
- vertex processing
- primitive processing
- fragment processing
- summary

Rendering

- the process of generating an image given
 - a virtual camera
 - objects
 - light sources
- various techniques, e.g.
 - rasterization (topic of this course)
 - raytracing (topic of the course “Advanced Computer Graphics”)
- one of the major research topics in computer graphics
 - rendering
 - animation
 - geometry processing

Rasterization

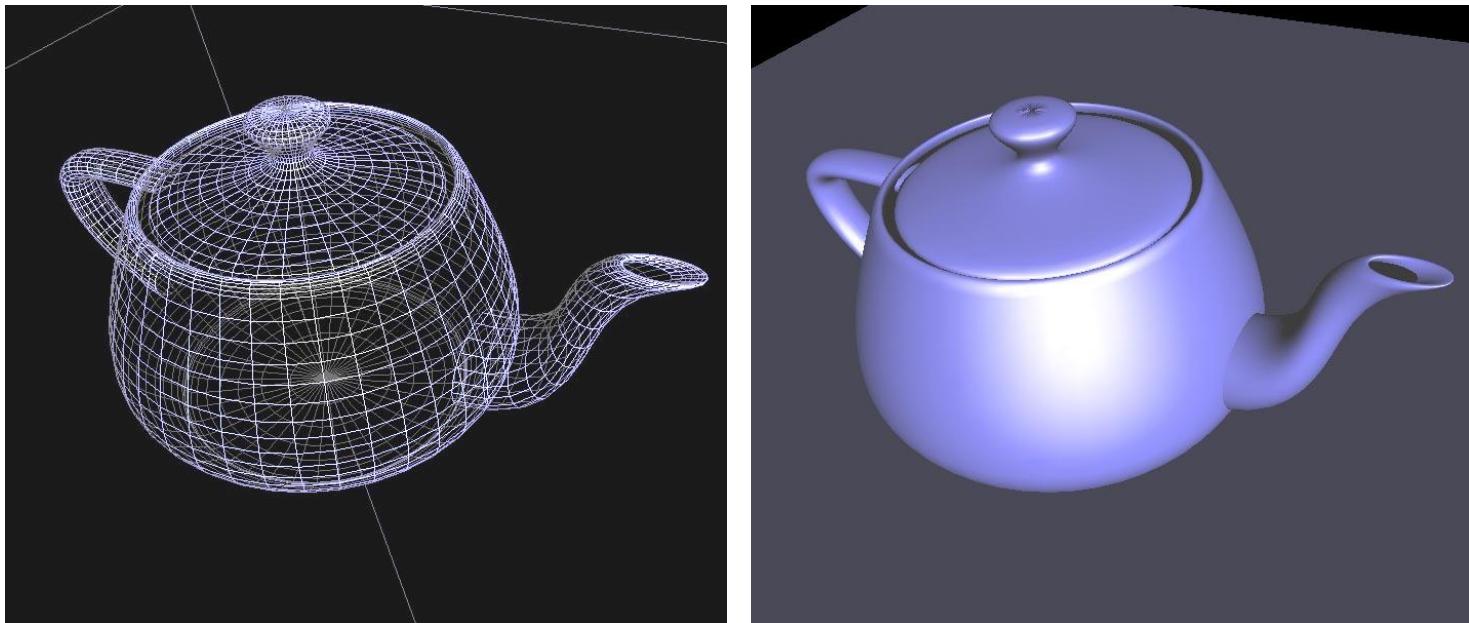
- rendering algorithm for generating 2D images from 3D scenes
- transforming geometric primitives such as lines and polygons into raster image representations, i.e. pixels



[Akenine-Moeller et al.: Real-time Rendering]

Rasterization

- 3D objects are approximately represented by vertices (points), lines, polygons
- these primitives are processed to obtain a 2D image



[Akenine-Moeller]

Rendering Pipeline

- processing stages comprise the rendering pipeline (graphics pipeline)
- supported by commodity graphics hardware
 - GPU - graphics processing unit
 - computes stages of the rasterization-based rendering pipeline
- OpenGL and DirectX are software interfaces to graphics hardware
 - this course focuses on concepts of the rendering pipeline
 - this course assumes OpenGL in implementation-specific details

Outline

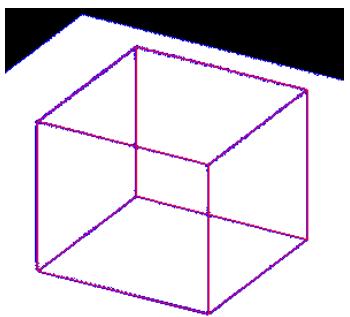
- introduction
- rendering pipeline
- vertex processing
- primitive processing
- fragment processing
- summary

Rendering Pipeline - Task

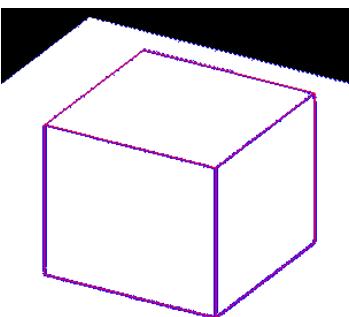
- 3D input
 - a virtual camera
 - position, orientation, focal length
 - objects
 - points (vertex / vertices), lines, polygons
 - geometry and material properties
(position, normal, color, texture coordinates)
 - light sources
 - direction, position, color, intensity
 - textures (images)
- 2D output
 - per-pixel color values in the framebuffer

Rendering Pipeline / Some Functionality

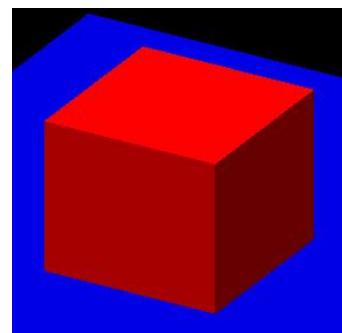
- resolving visibility
- evaluating a lighting model
- computing shadows (not core functionality)
- applying textures



visibility



lighting model



shadow



texture

[Wright et al.: OpenGL SuperBible]

Rendering Pipeline

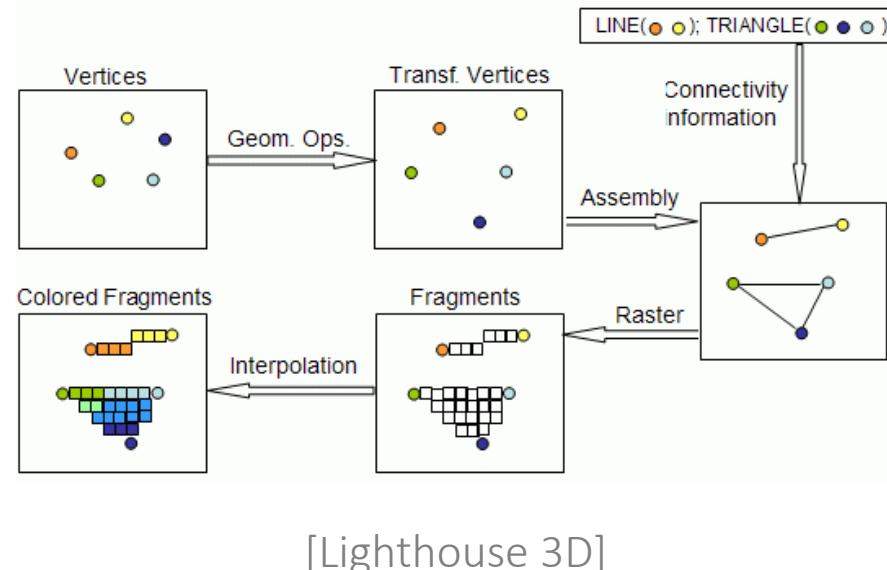
Main Stages

- vertex processing / geometry stage / vertex shader
 - processes all vertices independently in the same way
 - performs transformations per vertex, computes lighting per vertex
- geometry shader
 - generates, modifies, discards primitives
- primitive assembly and rasterization / rasterization stage
 - assembles primitives such as points, lines, triangles
 - converts primitives into a raster image
 - generates fragments / pixel candidates
 - fragment attributes are interpolated from vertices of a primitive
- fragment processing / fragment shader
 - processes all fragments independently in the same way
 - fragments are processed, discarded or stored in the framebuffer

Rendering Pipeline

Main Stages

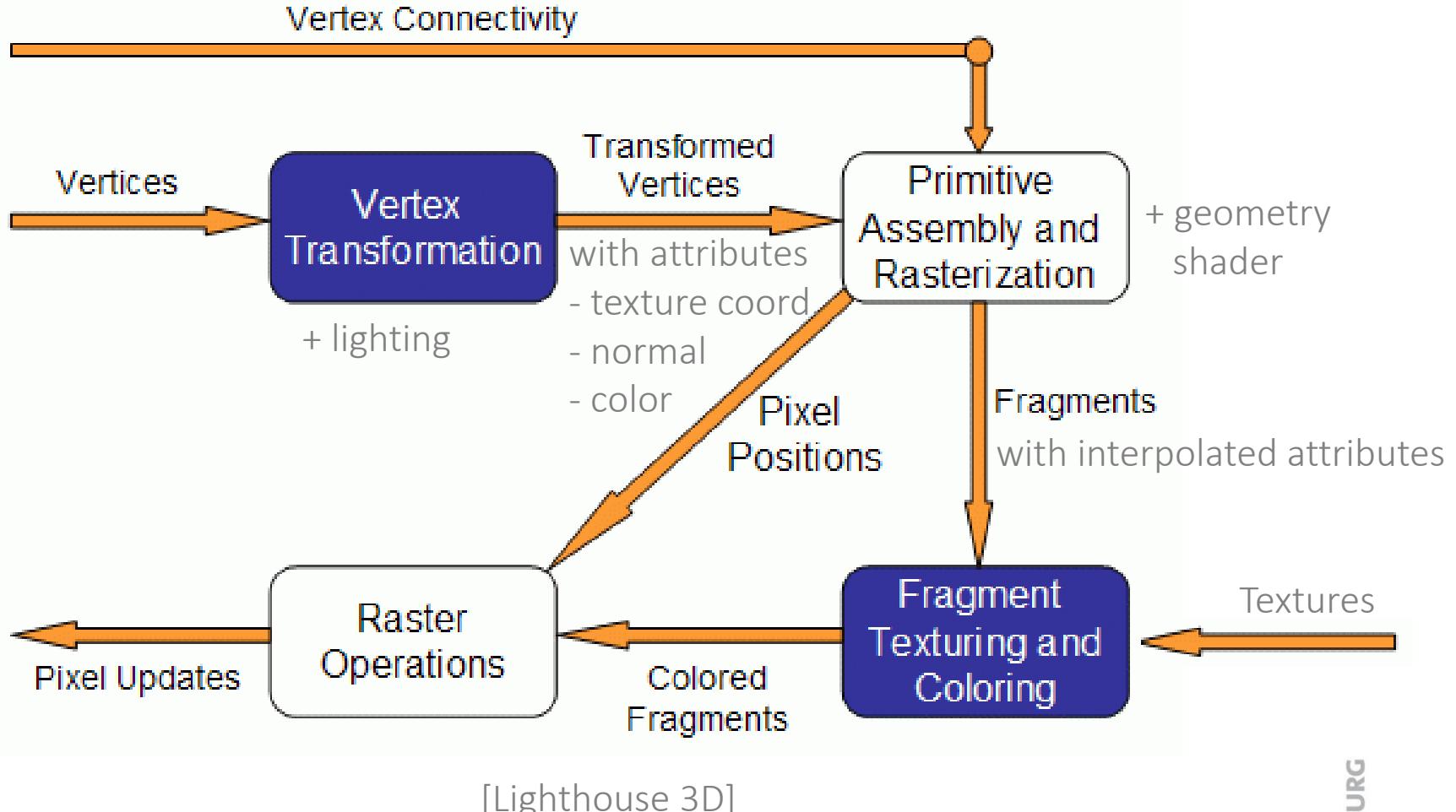
- vertex position transform
- lighting per vertex
- primitive assembly, combine vertices to lines, polygons
- rasterization, computes pixel positions affected by a primitive
- fragment generation with interpolated attributes, e.g. color
- fragment processing (not illustrated), fragment is discarded or used to update the pixel information in the framebuffer, more than one fragment can be processed per pixel position



[Lighthouse 3D]

Rendering Pipeline

Main Stages



Outline

- introduction
- rendering pipeline
- vertex processing
- primitive processing
- fragment processing
- summary

Vertex Processing (Geometry Stage)

- model transform
- view transform
- lighting
- projection transform
- clipping
- viewport transform

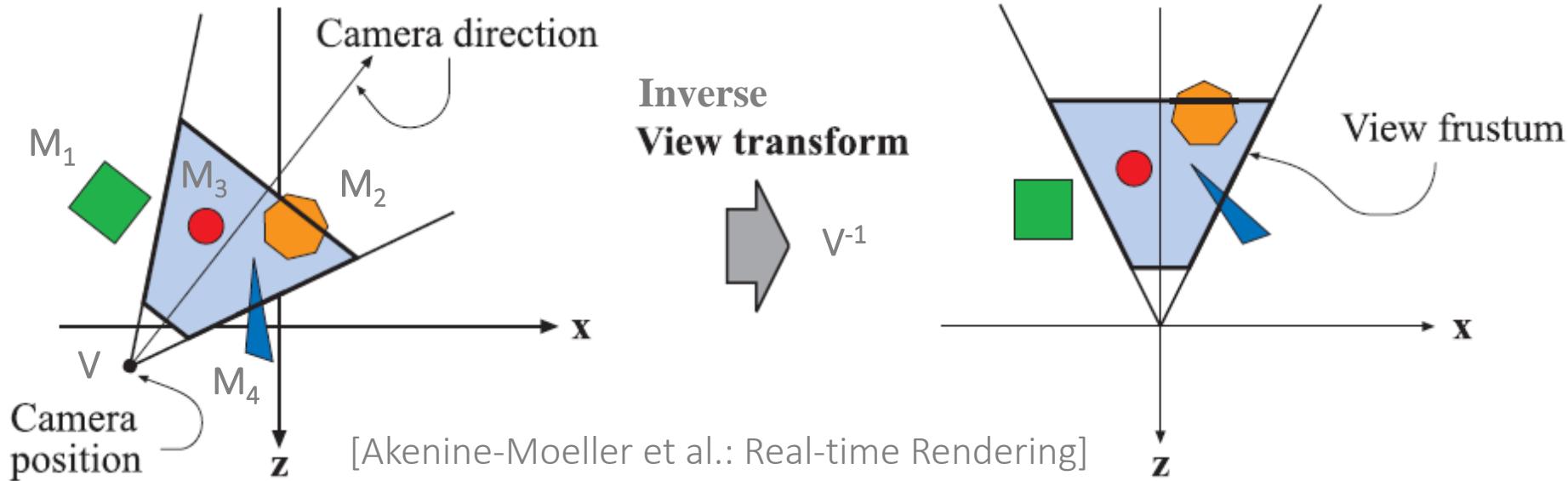
Model Transform

View Transform

- each object and the respective vertices are positioned, oriented, scaled in the scene with a model transform
- camera is positioned and oriented, represented by the view transform
- i.e., the inverse view transform is the transform that places the camera at the origin of the coordinate system, facing in the negative z-direction
- entire scene is transformed with the inverse view transform

Model Transform

View Transform



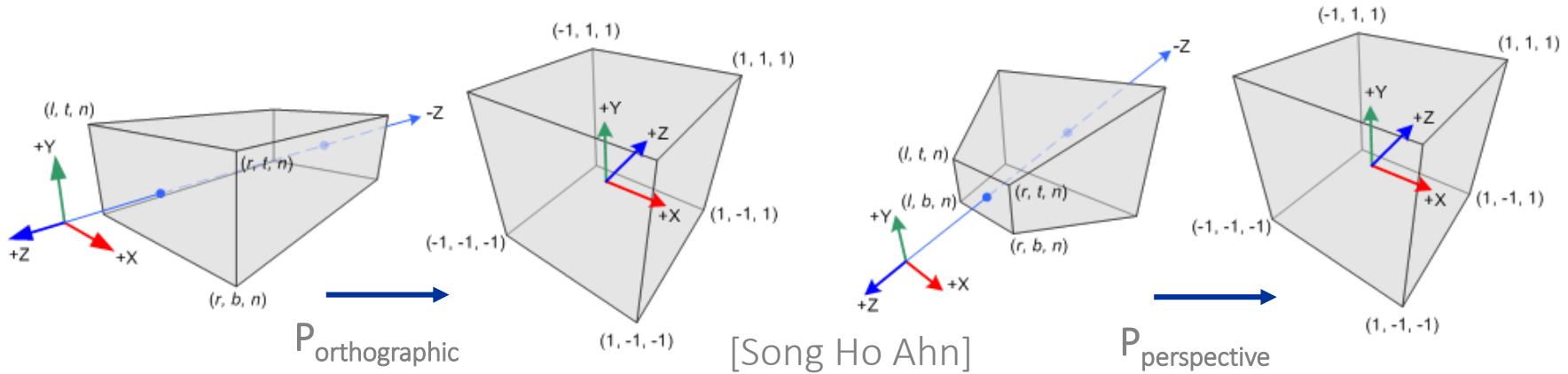
- M_1, M_2, M_3, M_4, V are matrices representing transformations
- M_1, M_2, M_3, M_4 are model transforms to place the objects in the scene
- V places and orientates the camera in space
 - V^{-1} transforms the camera to the origin looking along the negative z -axis
- model and view transforms are combined in the modelview transform
- the modelview transform $V^{-1}M_{1..4}$ is applied to the objects

Lighting

- interaction of light sources and surfaces is represented with a lighting / illumination model
- lighting computes color for each vertex
 - based on light source positions and properties
 - based on transformed position, transformed normal, and material properties of a vertex

Projection Transform

- P transforms the view volume to the canonical view volume
- the view volume depends on the camera properties
 - orthographic projection \rightarrow cuboid
 - perspective projection \rightarrow pyramidal frustum



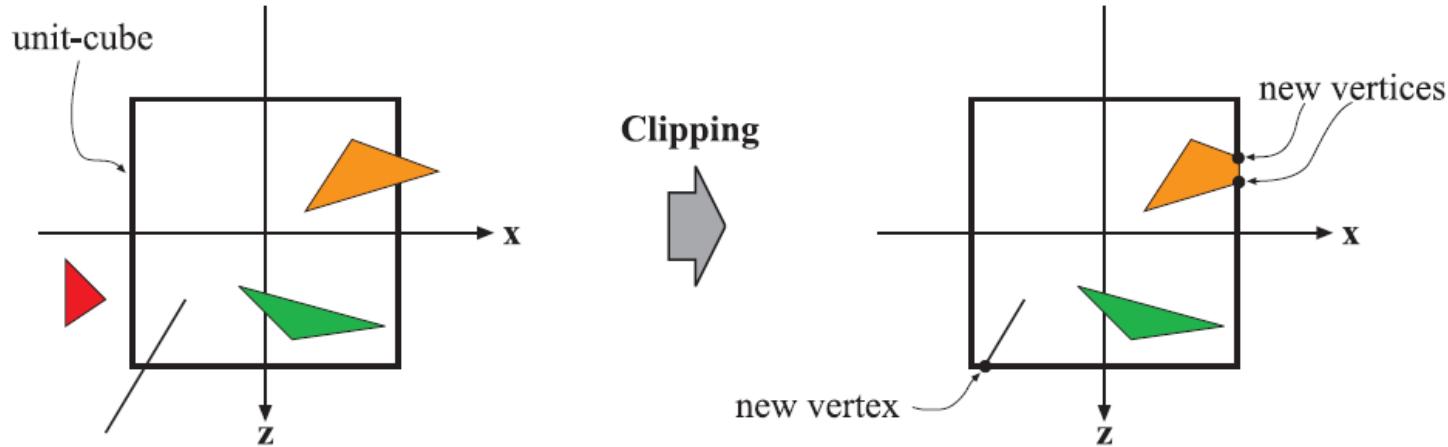
- canonical view volume is a cube from $(-1, -1, -1)$ to $(1, 1, 1)$
- view volume is specified by near, far, left, right, bottom, top

Projection Transform

- view volume (cuboid or frustum) is transformed into a cube (canonical view volume)
- objects inside (and outside) the view volume are transformed accordingly
- orthographic
 - combination of translation and scaling
 - all objects are translated and scaled in the same way
- perspective
 - complex transformation
 - scaling factor depends on the distance of an object to the viewer
 - objects farther away from the camera appear smaller

Clipping

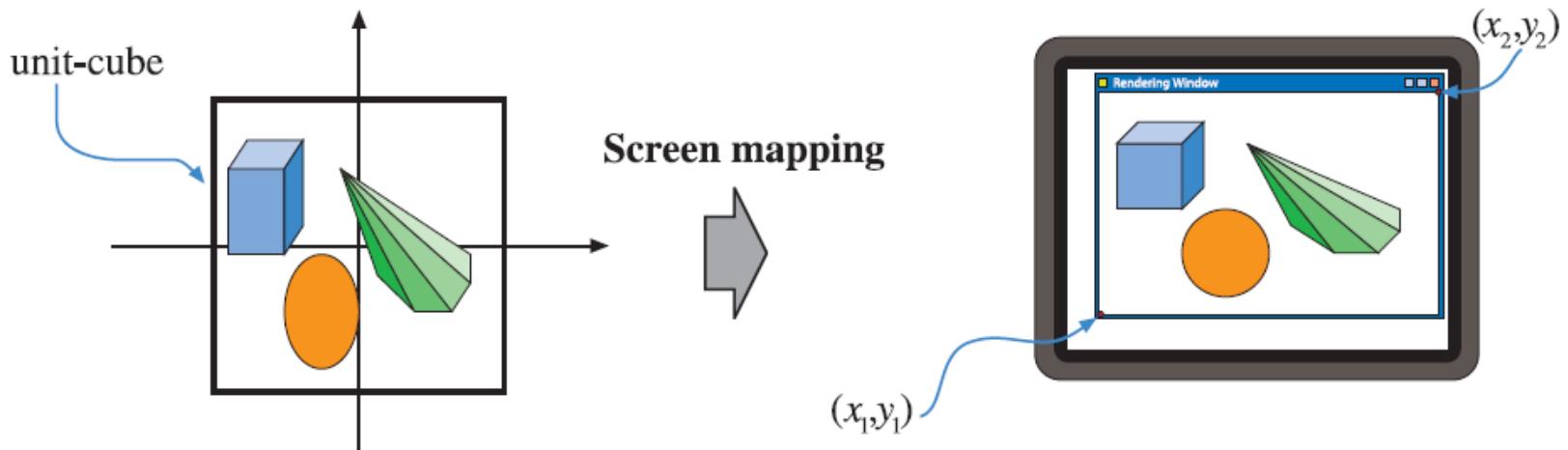
- primitives, that intersect the boundary of the view volume, are clipped
 - primitives, that are inside, are passed to the next processing stage
 - primitives, that are outside, are discarded
- clipping deletes and generates vertices and primitives



[Akenine-Moeller et al.: Real-time Rendering]

Viewport Transform / Screen Mapping

- projected primitive coordinates (x_p, y_p, z_p) are transformed to screen coordinates (x_s, y_s)
- screen coordinates together with depth value are window coordinates (x_s, y_s, z_w)



[Akenine-Moeller et al.: Real-time Rendering]

University of Freiburg – Computer Science Department – Computer Graphics - 21

Viewport Transform / Screen Mapping

- (x_p, y_p) are translated and scaled from the range of $(-1, 1)$ to actual pixel positions (x_s, y_s) on the display
- z_p is generally translated and scaled from the range of $(-1, 1)$ to $(0,1)$ for z_w
- screen coordinates (x_s, y_s) represent the pixel position of a fragment that is generated in a subsequent step
- z_w , the depth value, is an attribute of this fragment used for further processing

Vertex Processing - Summary

object space



modelview transform

eye space / camera space



lighting, projection

clip space / normalized
device coordinates



clipping, viewport transform

window space

Vertex Processing - Summary

- input
 - vertices in object / model space
 - 3D positions
 - attributes such as normal, material properties, texture coords
- output
 - vertices in window space
 - 2D pixel positions
 - attributes such as normal, material properties, texture coords
 - additional or updated attributes such as
 - normalized depth (distance to the viewer)
 - color (result of the evaluation of the lighting model)

Image Processing and Computer Graphics

Rendering Pipeline

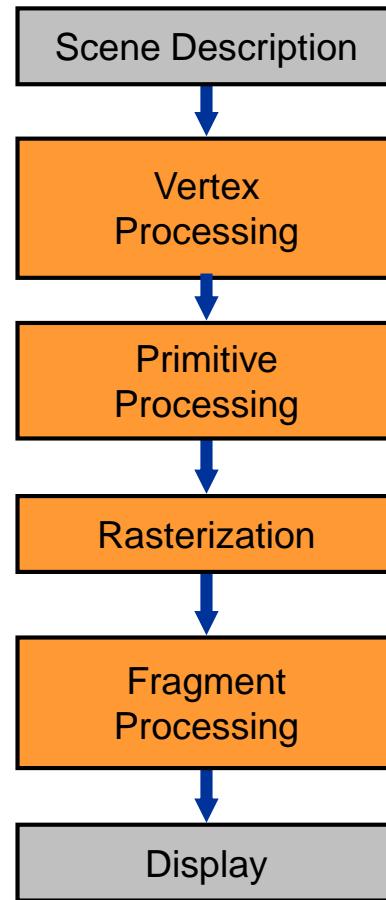
Matthias Teschner

Computer Science Department
University of Freiburg

Albert-Ludwigs-Universität Freiburg



Rendering Pipeline - Summary

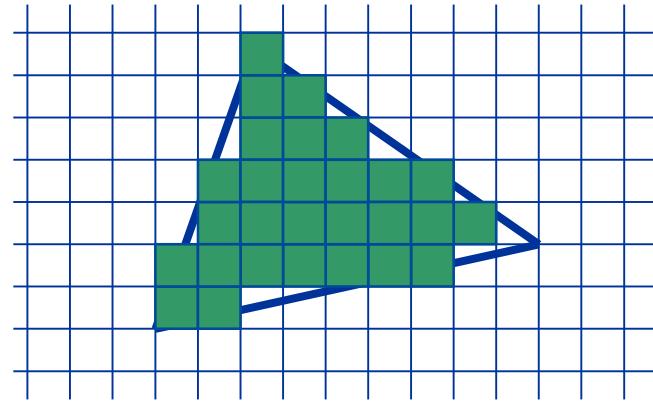
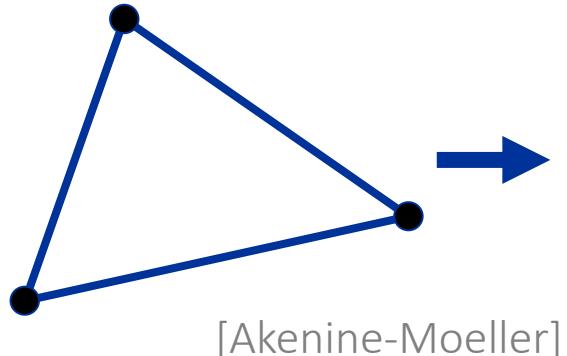


Outline

- introduction
- rendering pipeline
- vertex processing
- primitive processing
- fragment processing
- summary

Rasterization

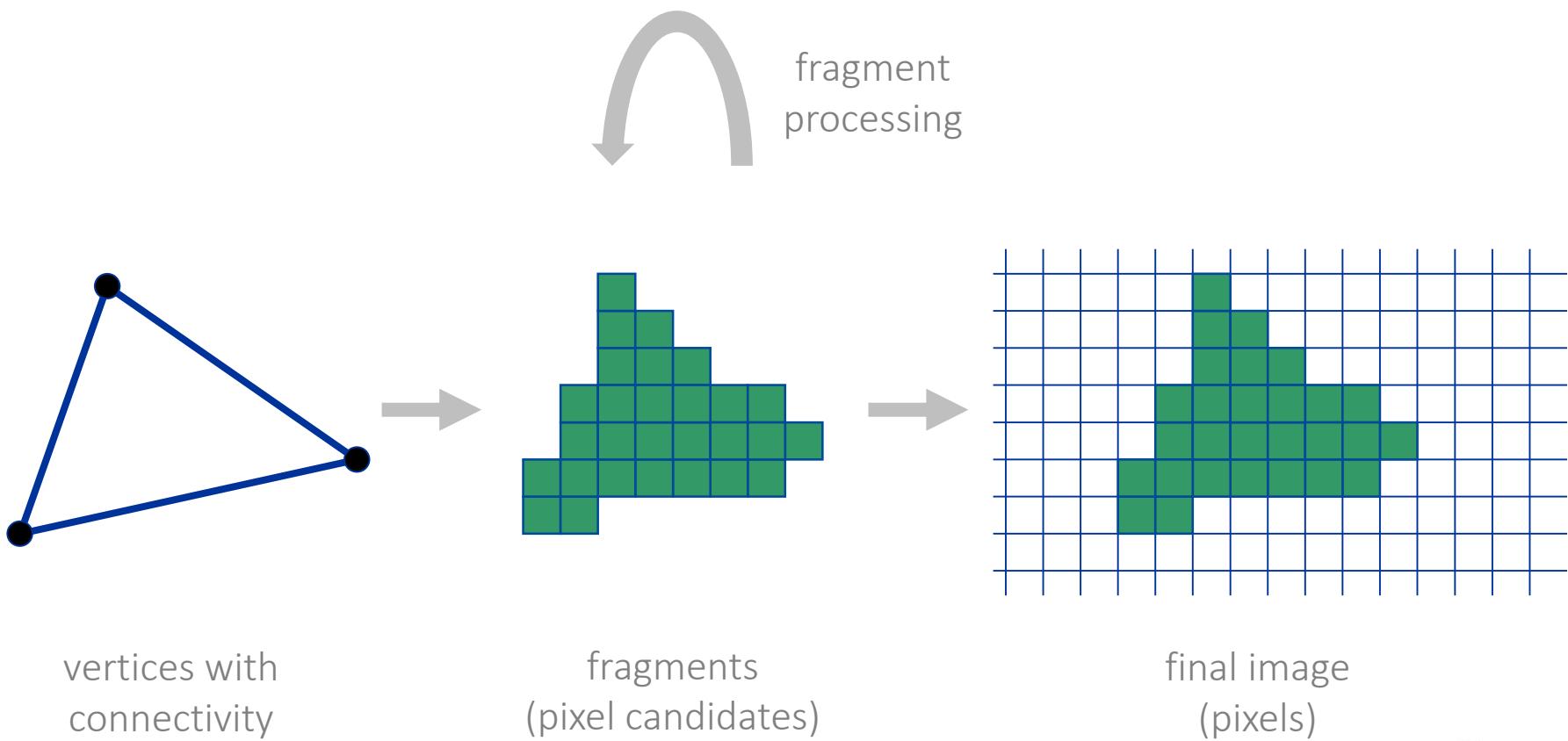
- input
 - vertices with attributes and connectivity information
 - attributes: color, depth, texture coordinates
- output
 - fragments with attributes
 - pixel position
 - interpolated color, depth, texture coordinates



Outline

- introduction
- rendering pipeline
- vertex processing
- primitive processing
- fragment processing
- summary

Illustration



vertices with
connectivity

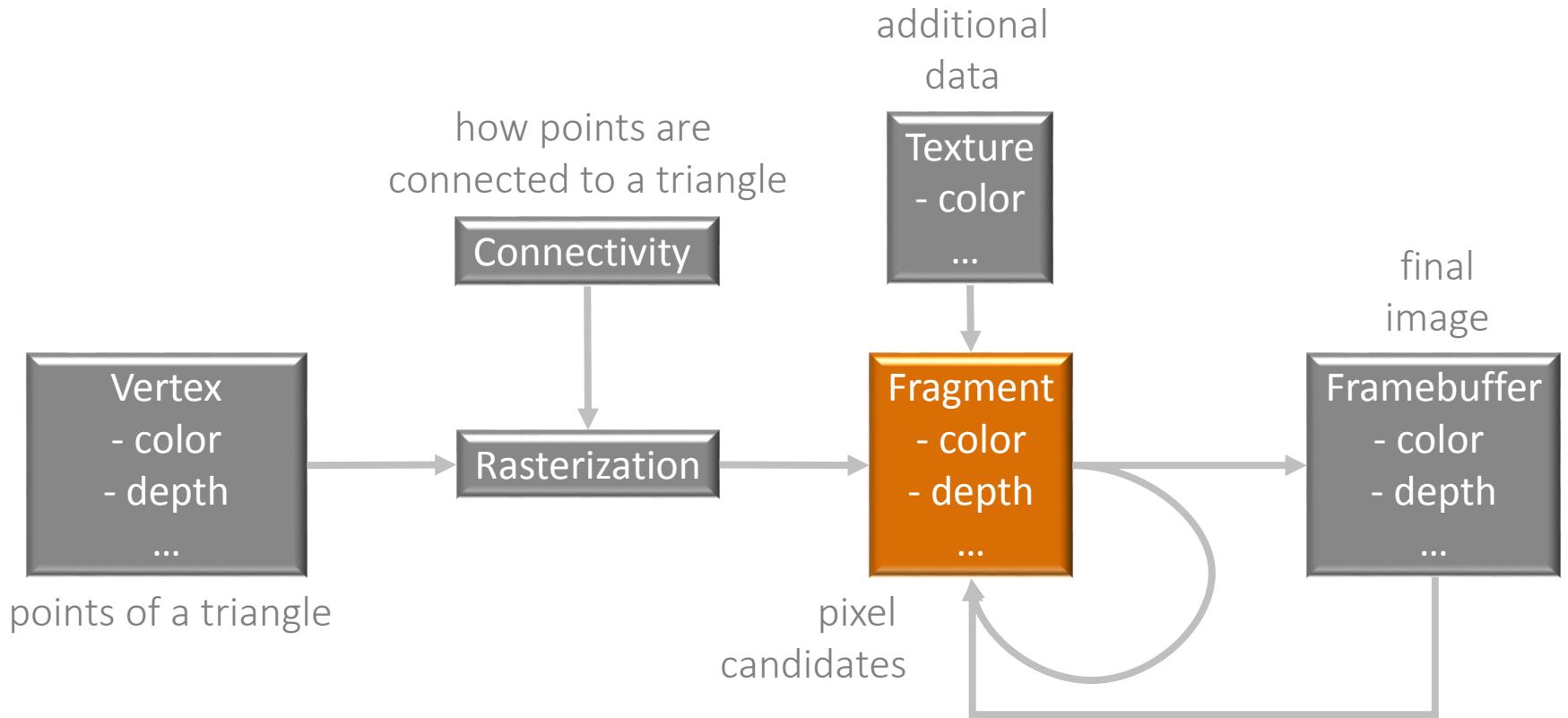
fragments
(pixel candidates)

final image
(pixels)

Fragment Processing

- fragment attributes are processed and tests are performed
 - fragment attributes are processed
 - fragments are discarded or
 - fragments pass a test and finally update the framebuffer
- processing and testing make use of
 - fragment attributes
 - textures
 - framebuffer data that is available for each pixel position
 - depth buffer, color buffer, stencil buffer, accumulation buffer

Illustration



Attribute Processing

- texture lookup
 - use texture coords to look up a texel (pixel of a texture image)
- texturing
 - combination of color and texel
- fog
 - adaptation of color based on fog color and depth value
- antialiasing
 - adaptation of alpha value (and color)
 - color has three components: red, green, blue
 - color is represented as a 4D vector (red, green, blue, alpha)

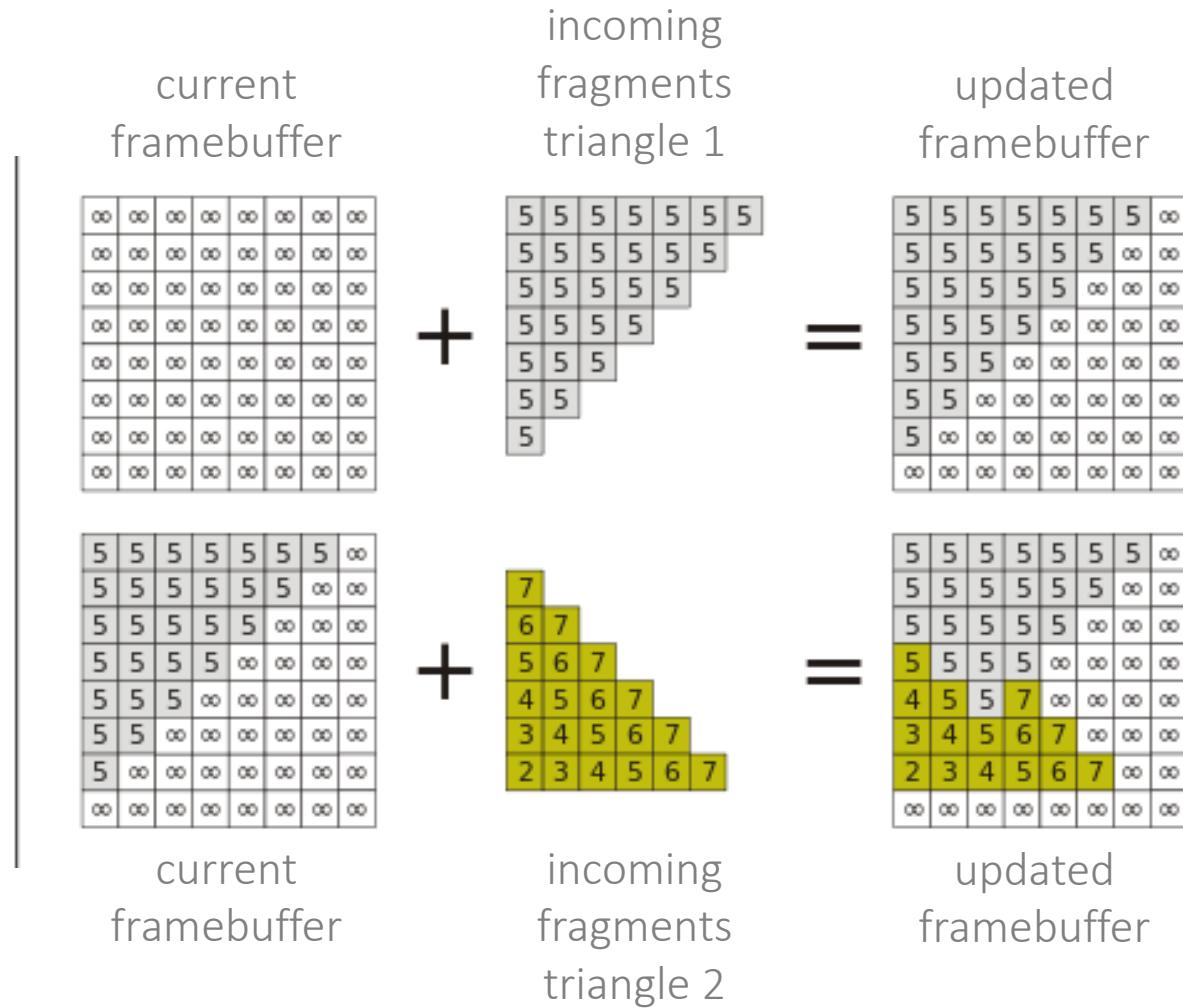
Tests

- scissor test
 - check if fragment is inside a specified rectangle
 - used for, e.g., masked rendering
- alpha test
 - check if the alpha value fulfills a certain requirement
 - comparison with a specified value
 - used for, e.g., transparency and billboarding
- stencil test
 - check if the stencil value in the framebuffer at the position of the fragment fulfills a certain requirement
 - comparison with a specified value
 - used for various rendering effects, e.g. masking, shadows

Depth Test

- depth test
 - compare depth value of the fragment and depth value of the framebuffer at the position of the fragment
 - used for resolving the visibility
 - if the depth value of the fragment is larger than the framebuffer depth value, the fragment is discarded
 - if the depth value of the fragment is smaller than the framebuffer depth value, the fragment passes and (potentially) overwrites the current color and depth values in the framebuffer

Depth Test



Wikipedia

Blending / Merging

- blending
 - combines the fragment color with the framebuffer color at the position of the fragment
 - usually determined by the alpha values
 - resulting color (including alpha value) is used to update the framebuffer
- dithering
 - finite number of colors
 - map color value to one of the nearest renderable colors
- logical operations / masking

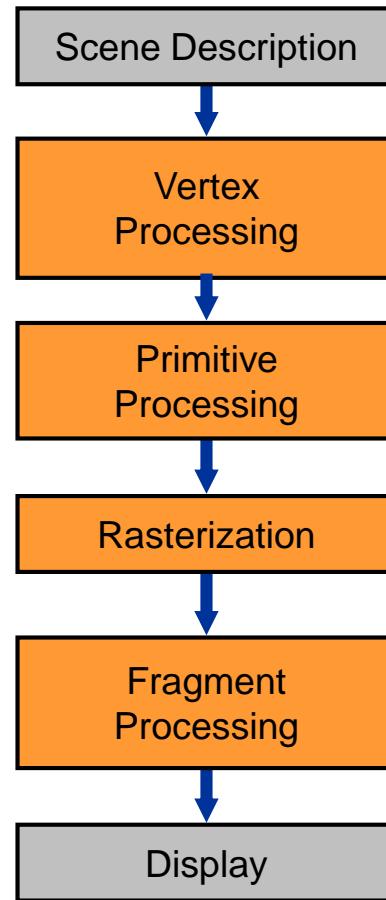
Fragment Processing - Summary

- texture lookup
- texturing
- fog
- antialiasing
- scissor test
- alpha test
- stencil test
- depth test
- blending
- dithering
- logical operations

Outline

- introduction
- rendering pipeline
- vertex processing
- primitive processing
- fragment processing
- summary

Rendering Pipeline - Summary



Rendering Pipeline - Summary

- primitives consist of vertices
- vertices have attributes (color, depth, texture coords)
- vertices are transformed and lit
- primitives are rasterized into fragments / pixel candidates with interpolated attributes
- fragments are processed using
 - their attributes such as color, depth, texture coordinates
 - texture data / image data
 - framebuffer data / data per pixel position (color, depth, stencil, accumulation)
- if a fragment passes all tests, it replaces the pixel data in the framebuffer

Image Processing and Computer Graphics
Transformations and
Homogeneous Coordinates

Matthias Teschner

Computer Science Department
University of Freiburg

Albert-Ludwigs-Universität Freiburg



Motivation

- transformations are used
 - to position, reshape, and animate objects, lights, and the virtual camera
 - to orthographically or perspectivly project three-dimensional geometry onto a plane
- transformations are represented with 4x4 matrices
- transformations are applied to vertices and normals
- vertices (positions) and normals (directions) are represented with 4D vectors

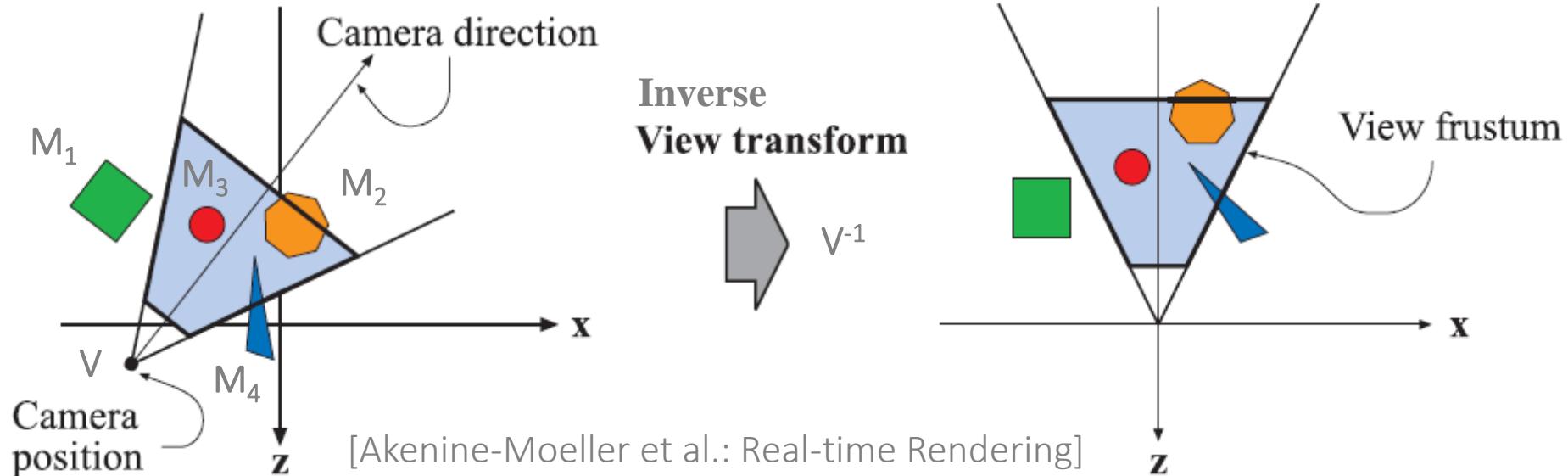
Outline

- transformations in the rendering pipeline
- motivations for the homogeneous notation
- homogeneous notation
- basic transformations in homogeneous notation
- compositing transformations
- summary

Vertex Processing

- modelview transform
- (lighting)
- projection transform
- (clipping)
- viewport transform

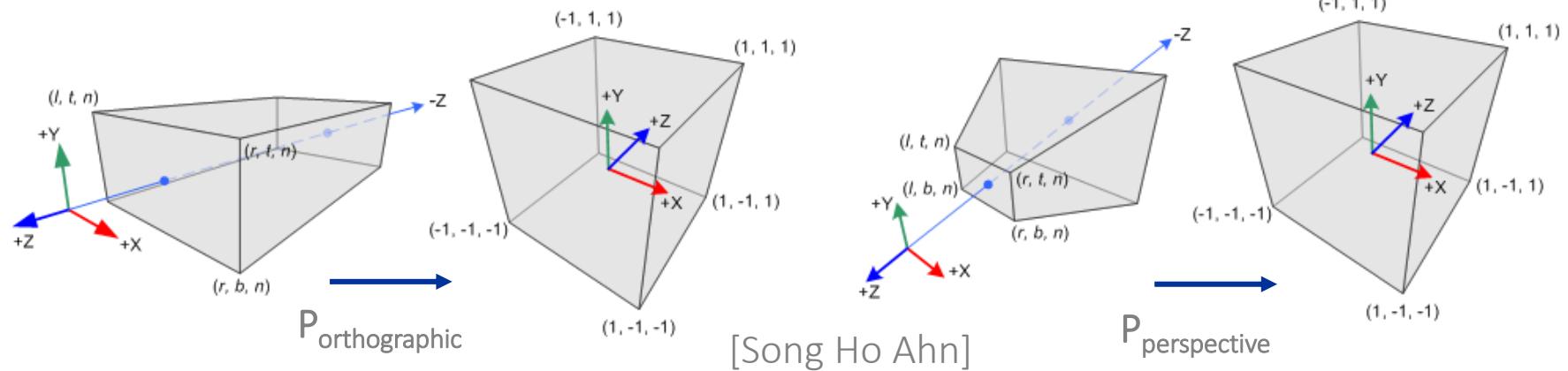
Modelview Transform



- M_1, M_2, M_3, M_4, V are matrices representing transformations
- M_1, M_2, M_3, M_4 are model transforms to place the objects in the scene
- V places and orientates the camera in space
 - V^{-1} transforms the camera to the origin looking along the negative z -axis
- model and view transforms are combined in the modelview transform
- the modelview transform $V^{-1}M_{1..4}$ is applied to the objects

Projection Transform

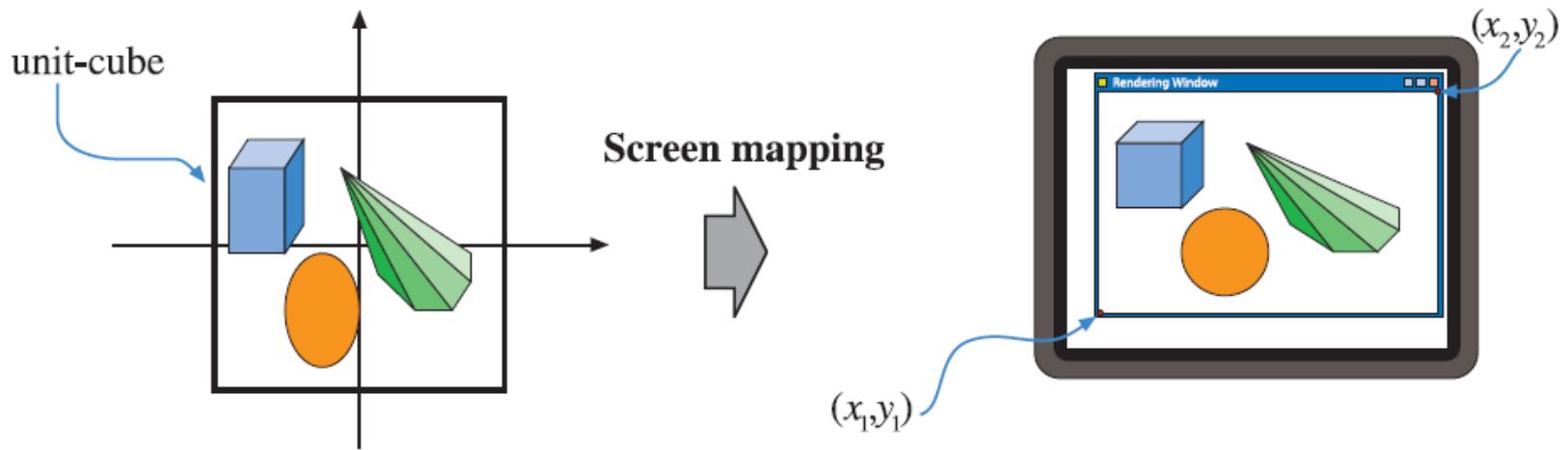
- P transforms the view volume to the canonical view volume
- the view volume depends on the camera properties
 - orthographic projection \rightarrow cuboid
 - perspective projection \rightarrow pyramidal frustum



- canonical view volume is a cube from $(-1, -1, -1)$ to $(1, 1, 1)$
- view volume is specified by near, far, left, right, bottom, top

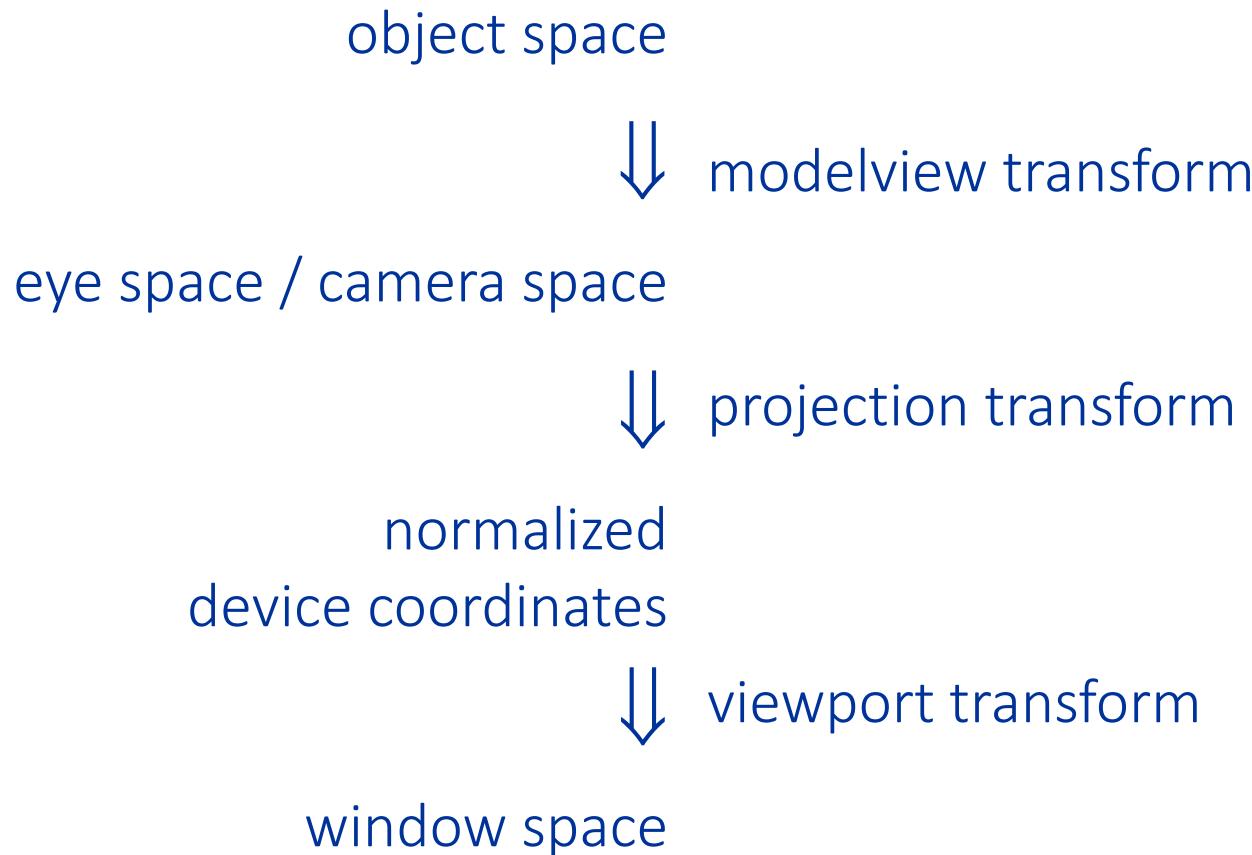
Viewport Transform / Screen Mapping

- projected primitive coordinates (x_p, y_p, z_p) are transformed to screen coordinates (x_s, y_s)
- screen coordinates together with depth value are window coordinates (x_s, y_s, z_w)

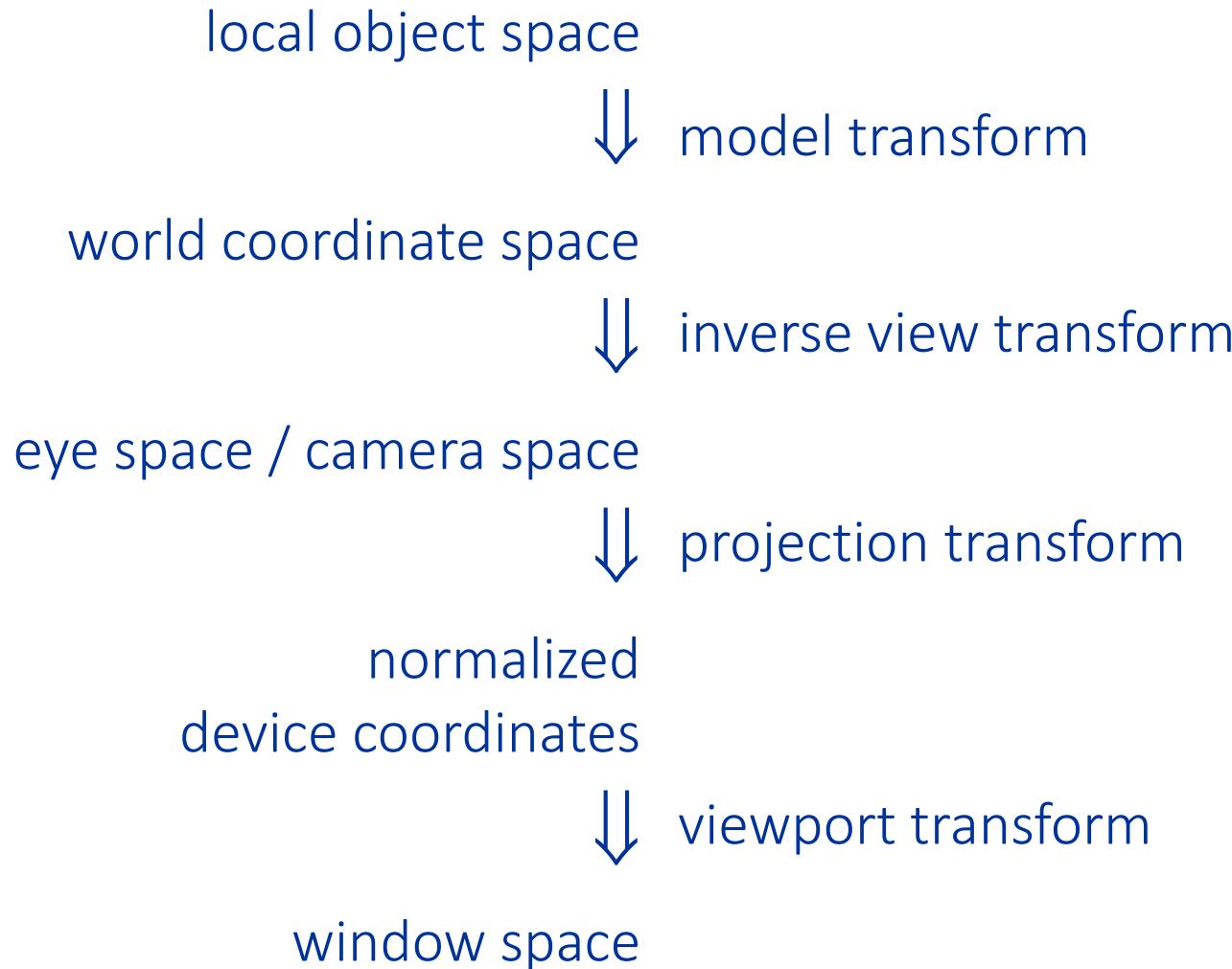


[Akenine-Moeller et al.: Real-time Rendering]

Vertex Transforms



Vertex Transforms



Outline

- transformations in the rendering pipeline
- motivations for the homogeneous notation
- homogeneous notation
- basic transformations in homogeneous notation
- compositing transformations
- summary

Some Transformations

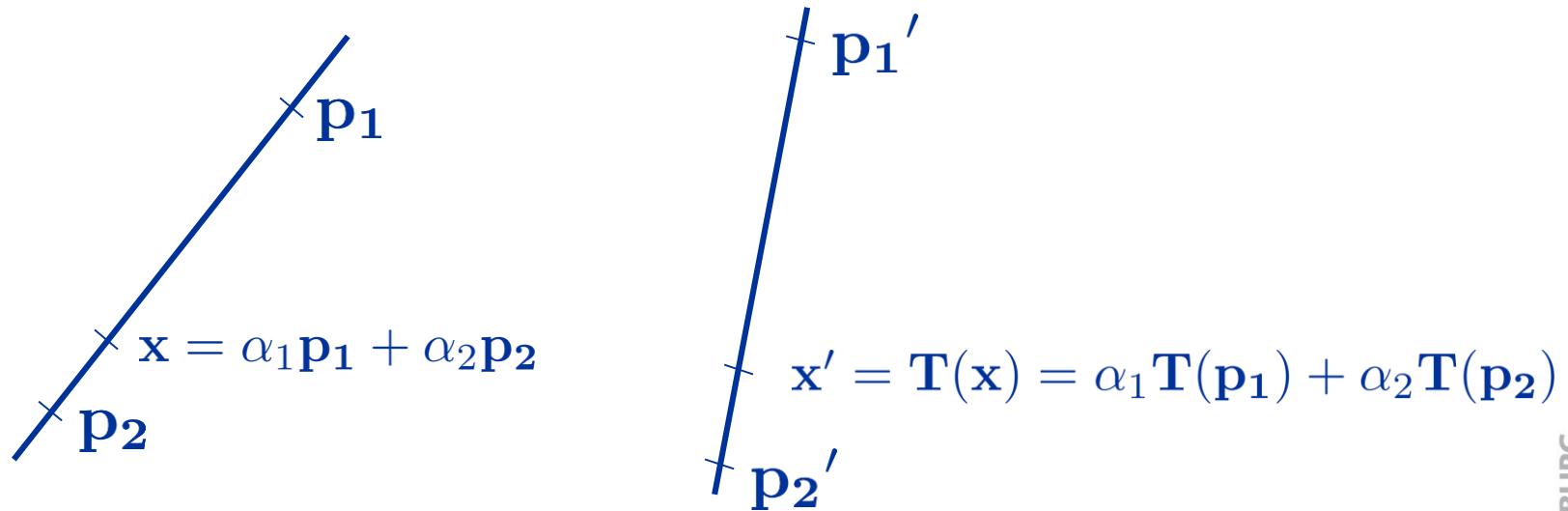
- congruent transformations
(Euclidean transformations)
 - preserve shape and size
 - translation, rotation, reflection
- similarity transformations
 - preserve shape
 - translation, rotation, reflection, scale

Affine Transformations

- preserve collinearity
 - points on a line are transformed to points on a line
- preserve proportions
 - ratios of distances between points are preserved
- preserve parallelism
 - parallel lines are transformed to parallel lines
- angles and lengths are not preserved
- translation, rotation, reflection, scale, shear are affine
- orthographic projection is a combination of affine transf.
- perspective projection is not affine

Affine Transformations

- affine transformations of a 3D point \mathbf{p} : $\mathbf{p}' = \mathbf{T}(\mathbf{p}) = \mathbf{A}\mathbf{p} + \mathbf{t}$
- affine transformations preserve affine combinations
 $\mathbf{T}(\sum_i \alpha_i \cdot \mathbf{p}_i) = \sum_i \alpha_i \cdot \mathbf{T}(\mathbf{p}_i)$ for $\sum_i \alpha_i = 1$
- e.g., a line can be transformed
by transforming its control points



Affine Transformations

- affine transformations of a 3D point \mathbf{p}
$$\mathbf{p}' = \mathbf{A}\mathbf{p} + \mathbf{t}$$
- the 3×3 matrix \mathbf{A} represents scale and rotation
- the 3D vector \mathbf{t} represents translation
- using homogeneous coordinates,
all affine transformations are represented
with one matrix-vector multiplication

Points and Vectors

- the rendering pipeline transforms vertices, normals, colors, texture coordinates
- points (e.g. vertices) specify a location in space
- vectors (e.g. normals) specify a direction
- relations between points and vectors
 - point - point = vector
 - point + vector = point
 - vector + vector = vector
 - point + point = not defined
 - $\vec{p} = p - \mathbf{O}$ $p = \mathbf{O} + \vec{p}$

Points and Vectors

- transformations can have different effects on points and vectors
- translation
 - translation of a point moves the point to a different position
 - translation of a vector does not change the vector
- using homogeneous coordinates, transformations of vectors and points can be handled in a unified way

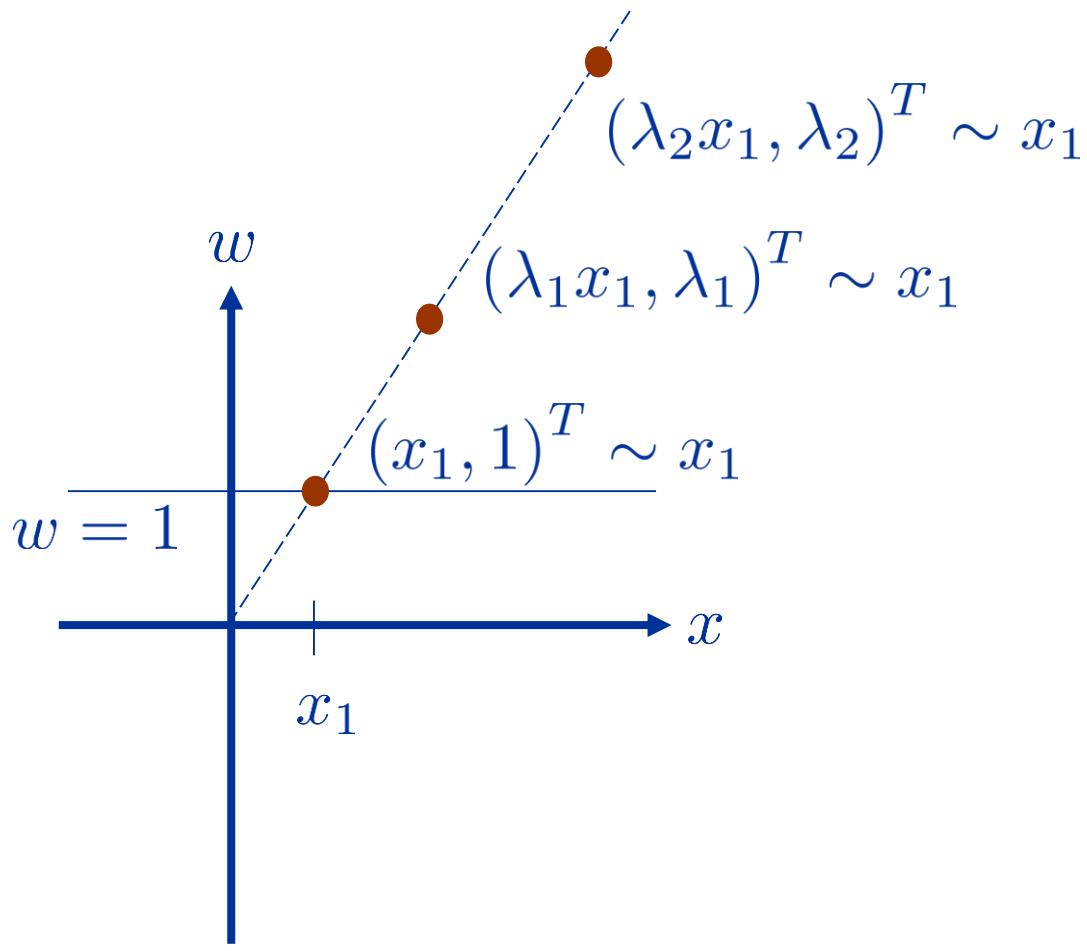
Outline

- transformations in the rendering pipeline
- motivations for the homogeneous notation
- homogeneous notation
- basic transformations in homogeneous notation
- compositing transformations
- summary

Homogeneous Coordinates of Points

- $(x, y, z, w)^T$ with $w \neq 0$ are the homogeneous coordinates of the 3D point $(\frac{x}{w}, \frac{y}{w}, \frac{z}{w})^T$
- $(\lambda x, \lambda y, \lambda z, \lambda w)^T$ represents the same point $(\frac{\lambda x}{\lambda w}, \frac{\lambda y}{\lambda w}, \frac{\lambda z}{\lambda w})^T = (\frac{x}{w}, \frac{y}{w}, \frac{z}{w})^T$ for all λ with $\lambda \neq 0$
- examples
 - $(2, 3, 4, 1) \sim (2, 3, 4)$
 - $(2, 4, 6, 1) \sim (2, 4, 6)$
 - $(4, 8, 12, 2) \sim (2, 4, 6)$

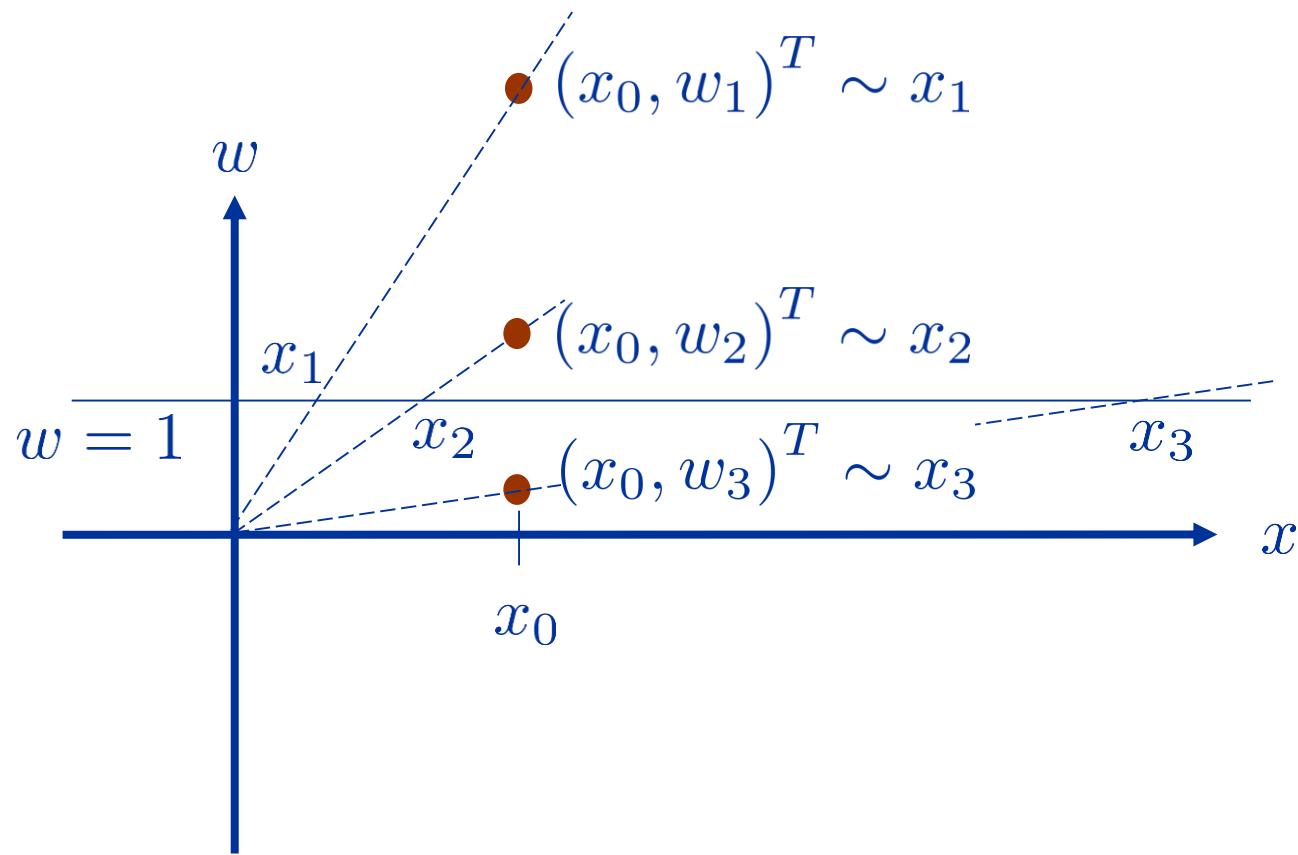
1D Illustration



Homogeneous Coordinates of Vectors

- for varying w , a point $(x, y, z, w)^T$ is scaled and the points $(\frac{x}{w}, \frac{y}{w}, \frac{z}{w})^T$ represent a line in 3D space
- the direction of this line is characterized by $(x, y, z)^T$
- for $w \rightarrow 0$, the point $(\frac{x}{w}, \frac{y}{w}, \frac{z}{w})^T$ moves to infinity in the direction $(x, y, z)^T$
- $(x, y, z, 0)^T$ is a point at infinity in the direction of $(x, y, z)^T$
- $(x, y, z, 0)^T$ is a vector in the direction of $(x, y, z)^T$

1D Illustration



Points and Vectors

- if points are represented in the homogeneous (normalized) form, point - vector relations can be represented
- vector + vector = vector
$$\begin{pmatrix} u_x \\ u_y \\ u_z \\ 0 \end{pmatrix} + \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix} = \begin{pmatrix} u_x + v_x \\ u_y + v_y \\ u_z + v_z \\ 0 \end{pmatrix}$$
- point + vector = point
$$\begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} + \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix} = \begin{pmatrix} p_x + v_x \\ p_y + v_y \\ p_z + v_z \\ 1 \end{pmatrix}$$
- point - point = vector
$$\begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} - \begin{pmatrix} r_x \\ r_y \\ r_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x - r_x \\ p_y - r_y \\ p_z - r_z \\ 0 \end{pmatrix}$$

Homogeneous Representation of Linear Transformations

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} \sim \begin{pmatrix} m_{00} & m_{01} & m_{02} & 0 \\ m_{10} & m_{11} & m_{12} & 0 \\ m_{20} & m_{21} & m_{22} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

- if the transform of $\begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}$ results in $\begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix}$, then
- the transform of $\begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$ results in $\begin{pmatrix} r_x \\ r_y \\ r_z \\ 1 \end{pmatrix} \sim \begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix}$

Affine Transformations and Projections

- general form

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & t_0 \\ m_{10} & m_{11} & m_{12} & t_1 \\ m_{20} & m_{21} & m_{22} & t_2 \\ p_0 & p_1 & p_2 & w \end{pmatrix}$$

- m_{ii} represent rotation, scale
- t_i represent translation
- p_i represent projection
- w is analog to the fourth component
for points / vectors

Homogeneous Coordinates - Summary

- $(x, y, z, w)^T$ with $w \neq 0$ are the homogeneous coordinates of the 3D point $(\frac{x}{w}, \frac{y}{w}, \frac{z}{w})^T$
- $(x, y, z, 0)^T$ is a point at infinity in the direction of $(x, y, z)^T$
- $(x, y, z, 0)^T$ is a vector in the direction of $(x, y, z)^T$
- $\begin{pmatrix} m_{00} & m_{01} & m_{02} & t_0 \\ m_{10} & m_{11} & m_{12} & t_1 \\ m_{20} & m_{21} & m_{22} & t_2 \\ p_0 & p_1 & p_2 & w \end{pmatrix}$ is a transformation, representing rotation, scale, translation, projection

Outline

- transformations in the rendering pipeline
- motivations for the homogeneous notation
- homogeneous notation
- basic transformations in homogeneous notation
- compositing transformations
- summary

Translation

- of a point

$$\mathbf{T}(\mathbf{t})\mathbf{p} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{pmatrix}$$

- of a vector

$$\mathbf{T}(\mathbf{t})\mathbf{v} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix}$$

- inverse (\mathbf{T}^{-1} "undoes" the transform \mathbf{T})

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

Rotation

- positive (anticlockwise) rotation with angle ϕ around the x -, y -, z -axis

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

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

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

Inverse Rotation

$$\begin{aligned}\blacksquare \quad \mathbf{R}_{\mathbf{x}}(-\phi) &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos -\phi & -\sin -\phi & 0 \\ 0 & \sin -\phi & \cos -\phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \phi & \sin \phi & 0 \\ 0 & -\sin \phi & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \mathbf{R}_{\mathbf{x}}^T(\phi)\end{aligned}$$

- $\mathbf{R}_{\mathbf{x}}^{-1} = \mathbf{R}_{\mathbf{x}}^T \quad \mathbf{R}_{\mathbf{y}}^{-1} = \mathbf{R}_{\mathbf{y}}^T \quad \mathbf{R}_{\mathbf{z}}^{-1} = \mathbf{R}_{\mathbf{z}}^T$
- the inverse of a rotation matrix corresponds to its transpose

Mirroring / Reflection

- mirroring with respect to $x = 0, y = 0, z = 0$ plane
- changes the sign of the x -, y -, z - component

$$\mathbf{P}_x = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{P}_y = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{P}_z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- the inverse of a reflection corresponds to its transpose

$$\mathbf{P}_x^{-1} = \mathbf{P}_x^T \quad \mathbf{P}_y^{-1} = \mathbf{P}_y^T \quad \mathbf{P}_z^{-1} = \mathbf{P}_z^T$$

Orthogonal Matrices

- rotation and reflection matrices are orthogonal
 $\mathbf{R}\mathbf{R}^T = \mathbf{R}^T\mathbf{R} = \mathbf{I}$
 $\mathbf{R}^{-1} = \mathbf{R}^T$
- $\mathbf{R}_1, \mathbf{R}_2$ are orthogonal $\Rightarrow \mathbf{R}_1\mathbf{R}_2$ is orthogonal
- rotation: $\det \mathbf{R} = 1$ reflection: $\det \mathbf{R} = -1$
- length of a vector does not change $\|\mathbf{R}\mathbf{v}\| = \|\mathbf{v}\|$
- angles are preserved $\langle \mathbf{R}\mathbf{u}, \mathbf{R}\mathbf{v} \rangle = \langle \mathbf{u}, \mathbf{v} \rangle$

Scale

- scaling x -, y -, z - components of a point or vector

$$\mathbf{S}(s_x, s_y, s_z) \mathbf{p} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x p_x \\ s_y p_y \\ s_z p_z \\ 1 \end{pmatrix}$$

- inverse $\mathbf{S}^{-1}(s_x, s_y, s_z) = \mathbf{S}\left(\frac{1}{s_x}, \frac{1}{s_y}, \frac{1}{s_z}\right)$
- uniform scaling: $s_x = s_y = s_z = s$

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

Shear

- one component is offset with respect to another component
- six basic shear modes in 3D
- e.g., shear of x with respect to z

$$\mathbf{H}_{xz}(s)\mathbf{p} = \begin{pmatrix} 1 & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + sp_z \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

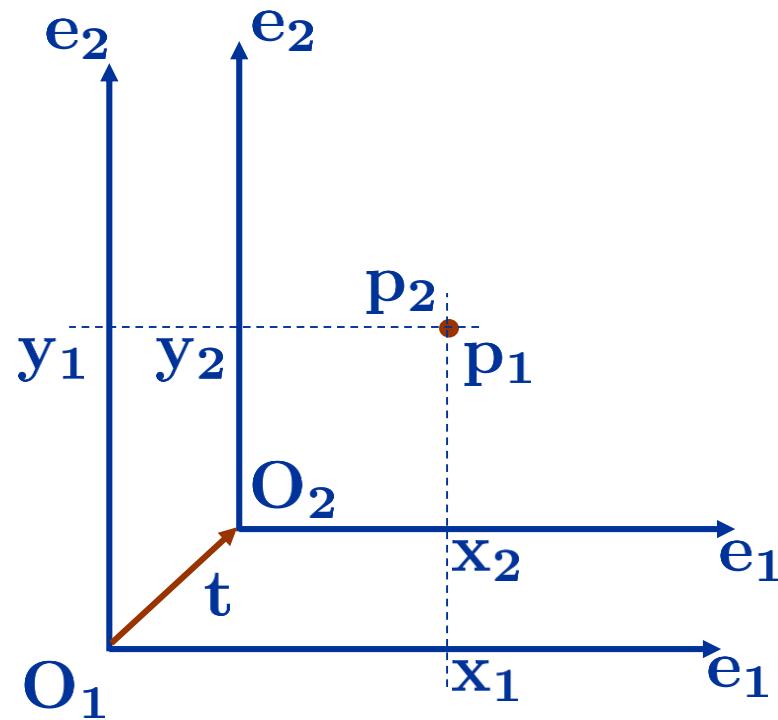
- inverse

$$\mathbf{H}_{xz}^{-1}(s) = \mathbf{H}_{xz}(-s)$$

Basis Transform - Translation

- two coordinate systems

$$C_1 = (O_1, \{e_1, e_2, e_3\}) \quad C_2 = (O_2, \{e_1, e_2, e_3\})$$



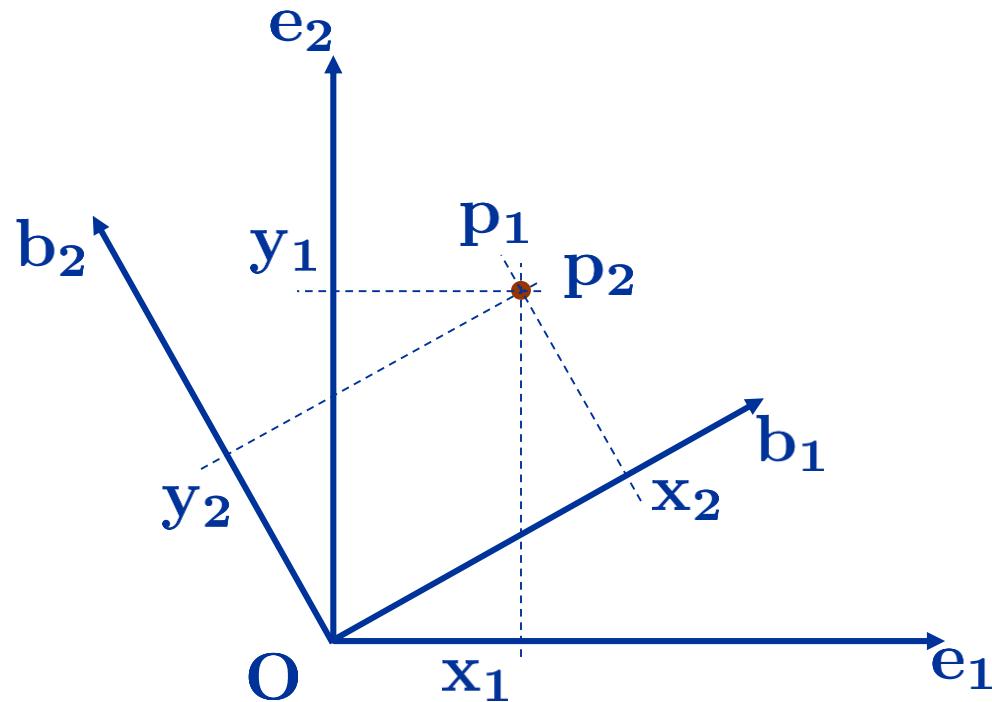
Basis Transform - Translation

- the coordinates of \mathbf{p}_1 with respect to \mathbf{C}_2 are given by $\mathbf{p}_2 = \mathbf{p}_1 - \mathbf{t}$ $\mathbf{p}_2 = \mathbf{T}(-\mathbf{t})\mathbf{p}_1$
- the coordinates of a point in the transformed basis correspond to the coordinates of point in the untransformed basis transformed by the inverse basis transform
 - translating the origin by \mathbf{t} corresponds to translating the object by $-\mathbf{t}$
 - also: rotating the basis vectors by an angle corresponds to rotating the object by the same negative angle

Basis Transform - Rotation

- two coordinate systems

$$C_1 = (O, \{e_1, e_2, e_3\}) \quad C_2 = (O, \{b_1, b_2, b_3\})$$



Basis Transform - Rotation

- the coordinates of \mathbf{p}_1 with respect to \mathbf{C}_2 are given by

$$\mathbf{p}_2 = \begin{pmatrix} \mathbf{b}_1^T \\ \mathbf{b}_2^T \\ \mathbf{b}_3^T \end{pmatrix} \mathbf{p}_1 \sim \begin{pmatrix} \mathbf{b}_{1x} & \mathbf{b}_{1y} & \mathbf{b}_{1z} & 0 \\ \mathbf{b}_{2x} & \mathbf{b}_{2y} & \mathbf{b}_{2z} & 0 \\ \mathbf{b}_{3x} & \mathbf{b}_{3y} & \mathbf{b}_{3z} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \mathbf{p}_1$$

- $\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3$ are the basis vectors of \mathbf{C}_2 with respect to \mathbf{C}_1
- $\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3$ are orthonormal, therefore the basis transform is a rotation
- rotating the basis vectors by an angle corresponds to rotating the object by the same negative angle

Basis Transform - Application

- the view transform can be seen as a basis transform
- objects are placed with respect to
a (global) coordinate system $\mathbf{C}_1 = (\mathbf{O}_1, \{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\})$
- the camera is also positioned at \mathbf{O}_2 and oriented at
 $\{\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3\}$ given by viewing direction and up-vector
- after the view transform, all objects are represented in
the eye or camera coordinate system $\mathbf{C}_2 = (\mathbf{O}_2, \{\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3\})$
- placing and orienting the camera corresponds to the
application of the inverse transform to the objects
- rotating the camera by \mathbf{R} and translating it by \mathbf{T} ,
corresponds to translating the objects by \mathbf{T}^{-1} and
rotating them by \mathbf{R}^{-1}

Planes and Normals

- planes can be represented by a surface normal \mathbf{n} and a point \mathbf{r} . All points \mathbf{p} with $\mathbf{n} \cdot (\mathbf{p} - \mathbf{r}) = 0$ form a plane.

$$n_x p_x + n_y p_y + n_z p_z + (-n_x r_x - n_y r_y - n_z r_z) = 0$$

$$n_x p_x + n_y p_y + n_z p_z + d = 0$$

$$(n_x \ n_y \ n_z \ d)(p_x \ p_y \ p_z \ 1)^T = 0$$

$$(n_x \ n_y \ n_z \ d)\mathbf{A}^{-1}\mathbf{A}(p_x \ p_y \ p_z \ 1)^T = 0$$

- the transformed points $\mathbf{A}(p_x \ p_y \ p_z \ 1)^T$ are on the plane represented by
$$(n_x \ n_y \ n_z \ d)\mathbf{A}^{-1} = ((\mathbf{A}^{-1})^T (n_x \ n_y \ n_z \ d)^T)^T$$
- if a surface is transformed by \mathbf{A} , its homogeneous notation (including the surface normal) is transformed by $(\mathbf{A}^{-1})^T$

Outline

- transformations in the rendering pipeline
- motivations for the homogeneous notation
- homogeneous notation
- basic transformations in homogeneous notation
- compositing transformations
- summary

Compositing Transformations

- composition is achieved by matrix multiplication
 - a translation \mathbf{T} applied to \mathbf{p} , followed by a rotation \mathbf{R}
$$\mathbf{R}(\mathbf{T}\mathbf{p}) = (\mathbf{RT})\mathbf{p}$$
 - a rotation \mathbf{R} applied to \mathbf{p} , followed by a translation \mathbf{T}
$$\mathbf{T}(\mathbf{Rp}) = (\mathbf{TR})\mathbf{p}$$
 - note that generally $\mathbf{TR} \neq \mathbf{RT}$
 - the order of composed transformations matters

Examples

- rotation around a line through \mathbf{t} parallel to the x -, y -, z -axis

$$\mathbf{T}(\mathbf{t})\mathbf{R}_{xyz}(\phi)\mathbf{T}(-\mathbf{t})$$

- scale with respect to an arbitrary axis

$$\mathbf{R}_{xyz}(\phi)\mathbf{S}(s_x, s_y, s_z)\mathbf{R}_{xyz}(-\phi)$$

- e.g., $\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3$ represent an orthonormal basis, then scaling along these vectors can be done by

$$\begin{pmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \mathbf{b}_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \mathbf{S}(s_x, s_y, s_z) \begin{pmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \mathbf{b}_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^T$$

Rigid-Body Transform

- $\begin{pmatrix} \mathbf{R} & \mathbf{t} \\ 0 & 1 \end{pmatrix} \mathbf{p} = \mathbf{T}(\mathbf{t})\mathbf{R}\mathbf{p}$
with \mathbf{R} being a rotation and \mathbf{t} being a translation is a combined transformation
- inverse
 $(\mathbf{T}(\mathbf{t})\mathbf{R})^{-1} = \mathbf{R}^{-1}\mathbf{T}(\mathbf{t})^{-1} = \mathbf{R}^T\mathbf{T}(-\mathbf{t})$
- in Euclidean coordinates $\mathbf{p}' = \mathbf{R}\mathbf{p} + \mathbf{t}$
- the inverse transform $\mathbf{p} = \mathbf{R}^{-1}(\mathbf{p}' - \mathbf{t}) = \mathbf{R}^{-1}\mathbf{p}' - \mathbf{R}^{-1}\mathbf{t}$
- therefore $\begin{pmatrix} \mathbf{R} & \mathbf{t} \\ 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} \mathbf{R}^T & -\mathbf{R}^T\mathbf{t} \\ 0 & 1 \end{pmatrix}$

Outline

- transformations in the rendering pipeline
- motivations for the homogeneous notation
- homogeneous notation
- basic transformations in homogeneous notation
- compositing transformations
- summary

Summary

- usage of the homogeneous notation is motivated by a unified processing of affine transformations, perspective projections, points, and vectors
- all transformations of points and vectors are represented by a matrix-vector multiplication
- "undoing" a transformation is represented by its inverse
- compositing of transformations is accomplished by matrix multiplication

Image Processing and Computer Graphics

Projections and

Transformations in OpenGL

Matthias Teschner

Computer Science Department
University of Freiburg

Albert-Ludwigs-Universität Freiburg

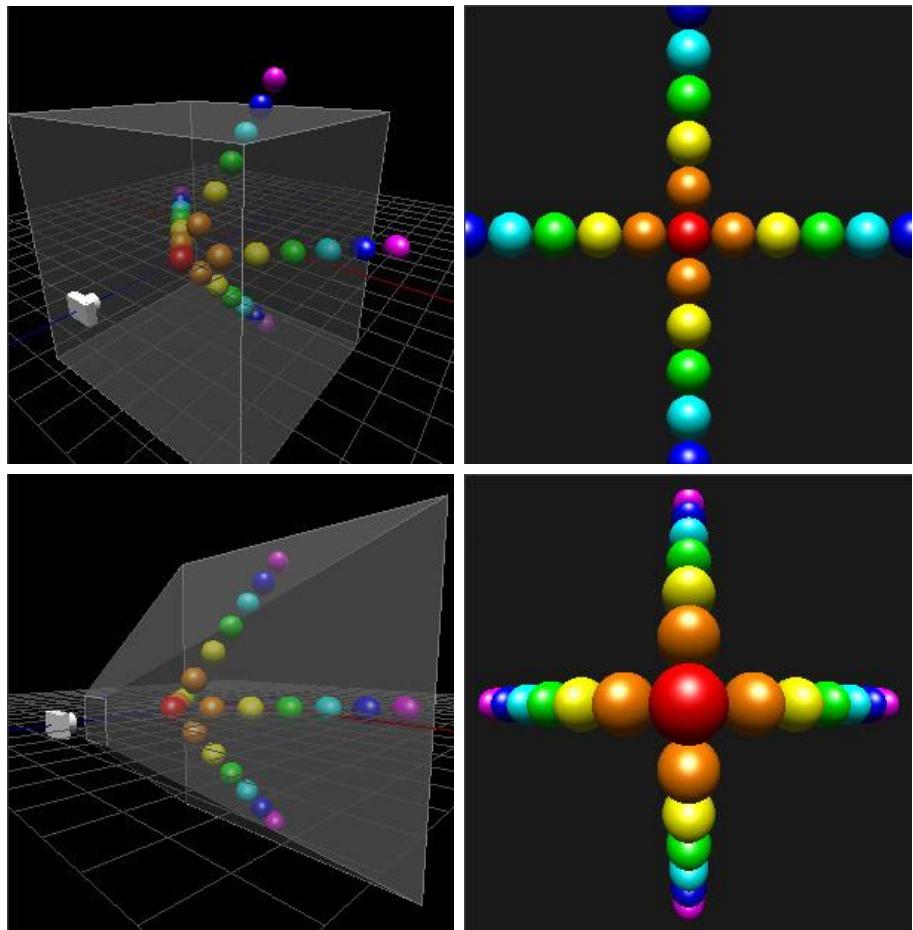


Motivation

- for the rendering of objects in 3D space,
a planar view has to be generated
- 3D space is projected onto a 2D plane considering
external and internal camera parameters
 - position, orientation, focal length
- in homogeneous notation, 3D projections can be
represented with a 4x4 transformation matrix

Examples

- left images
 - 3D scene with a view volume
- right images
 - projections onto the viewplane
- top-right
 - parallel projection
- top-bottom
 - perspective projection



[Song Ho Ahn]

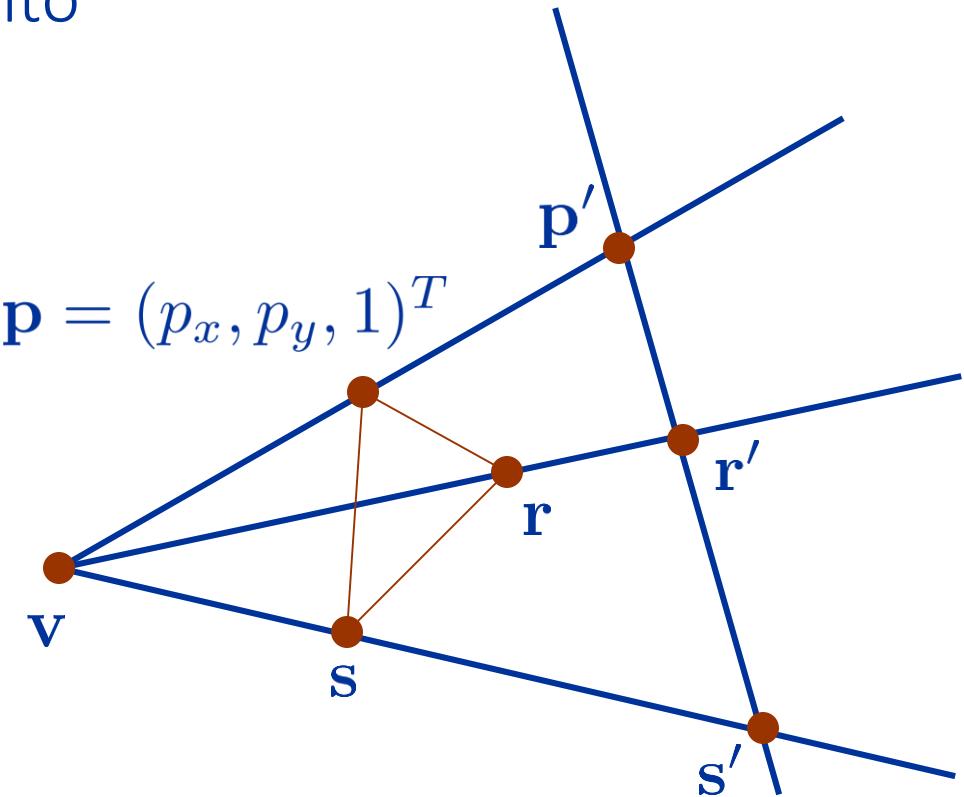
Outline

- 2D projection
- 3D projection
- OpenGL projection matrix
- OpenGL transformation matrices

Projection in 2D

- a 2D projection from v onto I maps a point p onto p'
- p' is the intersection of the line through p and v with line I
- v is the **viewpoint**, center of perspectivity
- I is the **viewline**
- the line through p and v is a projector
- v is not on the line I , $p \neq v$

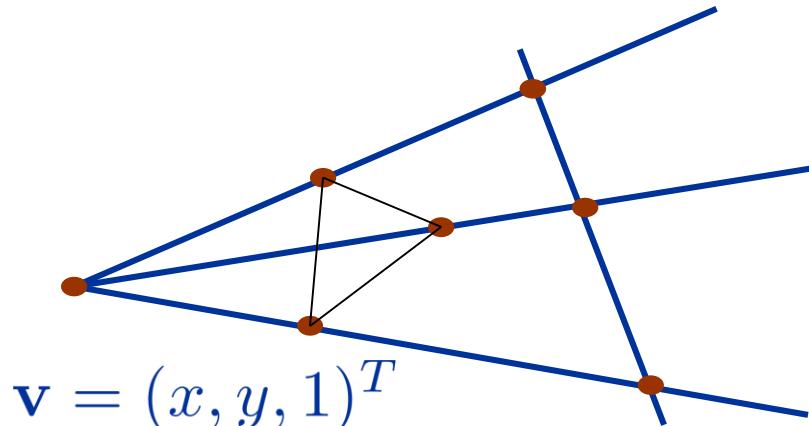
$$I = \{ax + by + c = 0\} = (a, b, c)^T$$



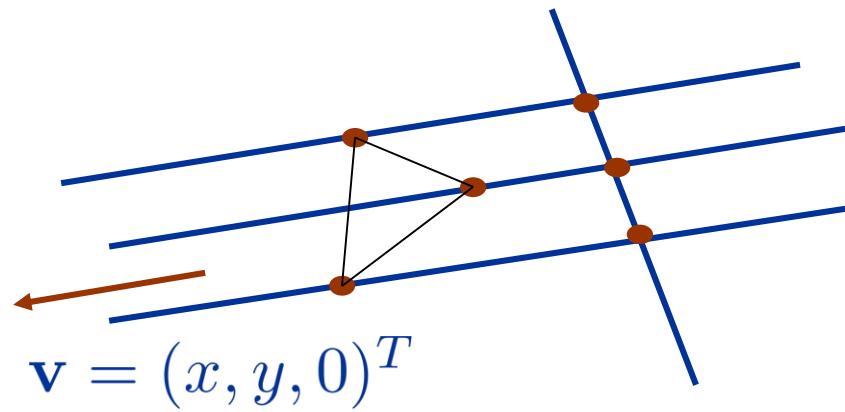
$$p = (p_x, p_y, 1)^T$$

Projection in 2D

- if the homogeneous component of the viewpoint \mathbf{v} is not equal to zero, we have a perspective projection
 - projectors are not parallel
- if \mathbf{v} is at infinity, we have a parallel projection
 - projectors are parallel



$\mathbf{v} = (x, y, 1)^T$
perspective projection



$\mathbf{v} = (x, y, 0)^T$
parallel projection

Classification

- location of viewpoint and orientation of the viewline determine the type of projection
- parallel (viewpoint at infinity, parallel projectors)
 - orthographic (viewline orthogonal to the projectors)
 - oblique (viewline not orthogonal to the projectors)
- perspective (non-parallel projectors)
 - one-point
(viewline intersects one principal axis,
i.e. viewline is parallel to a principal axis, one vanishing point)
 - two-point
(viewline intersects two principal axis, two vanishing points)

General Case

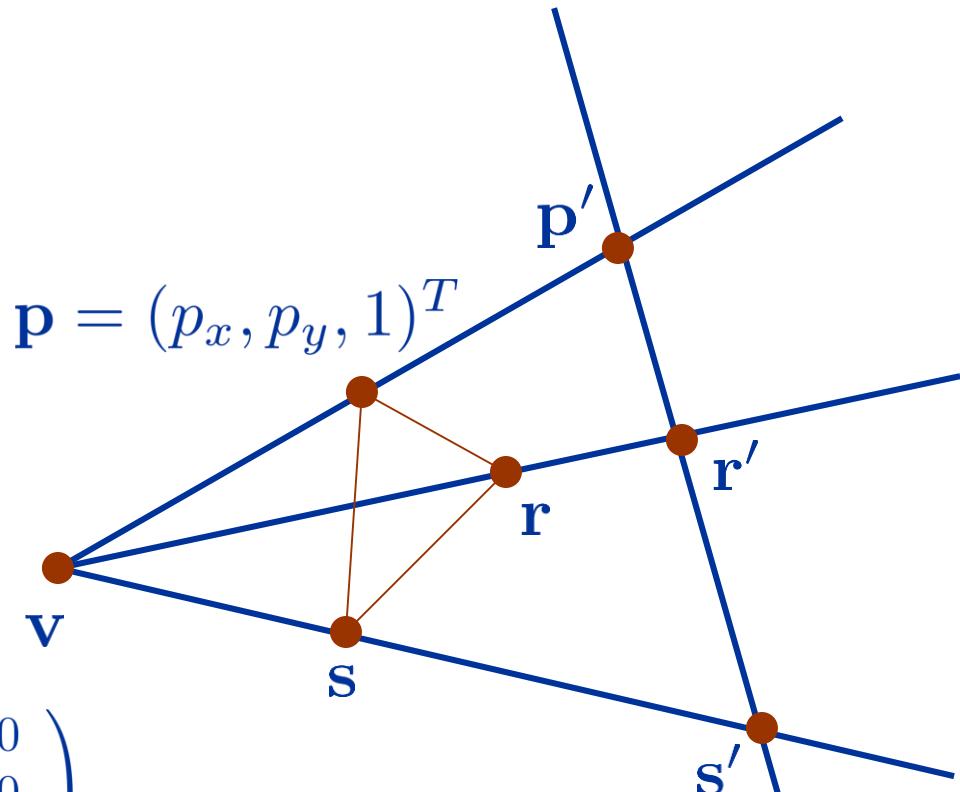
- a 2D projection is represented by matrix

$$\mathbf{M} = \mathbf{v}\mathbf{l}^T - (\mathbf{l} \cdot \mathbf{v})\mathbf{I}_3$$

$$\mathbf{v}\mathbf{l}^T = \begin{pmatrix} v_x a & v_x b & v_x c \\ v_y a & v_y b & v_y c \\ v_w a & v_w b & v_w c \end{pmatrix}$$

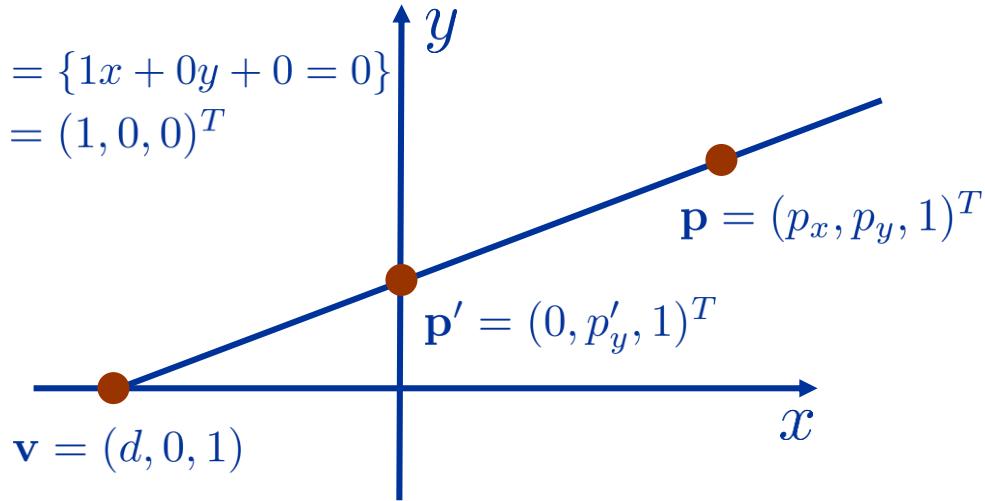
$$(\mathbf{l} \cdot \mathbf{v})\mathbf{I} = (av_x + bv_y + cv_w) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{l} = \{ax + by + c = 0\} = (a, b, c)^T$$



Example

- $l = \{1x + 0y + 0 = 0\}$
 $= (1, 0, 0)^T$



- $M = \begin{pmatrix} d \\ 0 \\ 1 \end{pmatrix} (1, 0, 0) - \left(\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} d \\ 0 \\ 1 \end{pmatrix} \right) I_3 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -d & 0 \\ 1 & 0 & -d \end{pmatrix}$
- e.g. $d=-1$, $(1,2)^T$ is mapped to $(0,1)^T$

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

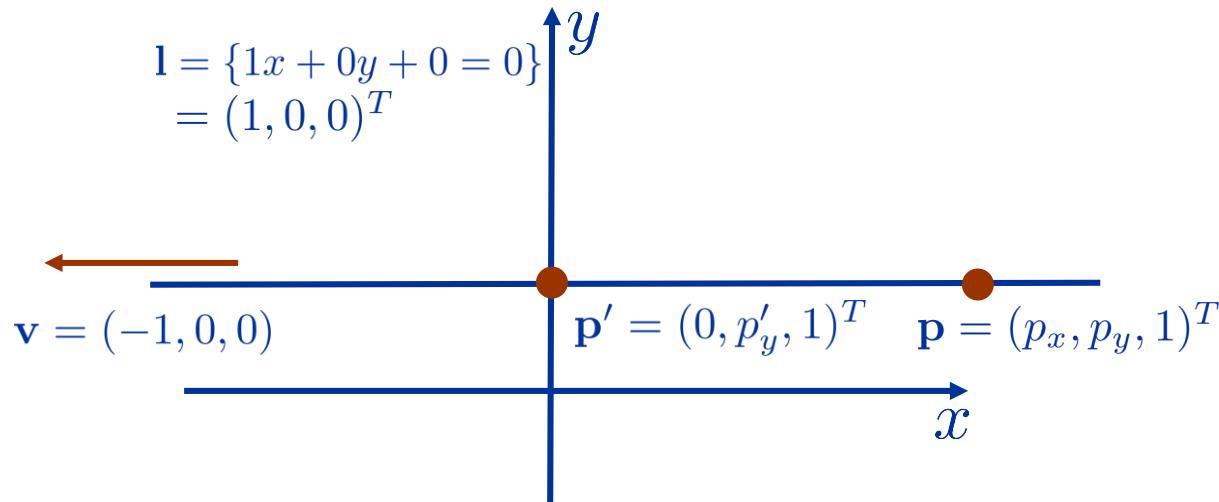
Discussion

- matrices \mathbf{M} and $\lambda\mathbf{M}$ represent the same transformation ($\lambda\mathbf{M}\mathbf{p} = \lambda\mathbf{p}'$)
- therefore, $\begin{pmatrix} 0 & 0 & 0 \\ 0 & -d & 0 \\ 1 & 0 & -d \end{pmatrix}$ and $\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{d} & 0 & 1 \end{pmatrix}$ represent the same transformation
- $\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{d} & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ w \end{pmatrix} = \begin{pmatrix} 0 \\ y \\ -\frac{x}{d} + w \end{pmatrix} \sim \begin{pmatrix} 0 \\ \frac{y}{w-\frac{x}{d}} \\ 1 \end{pmatrix}$
- x is mapped to zero, y is scaled depending on x
- moving d to infinity results in parallel projection

$$\lim_{d \rightarrow \pm\infty} \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{d} & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Discussion

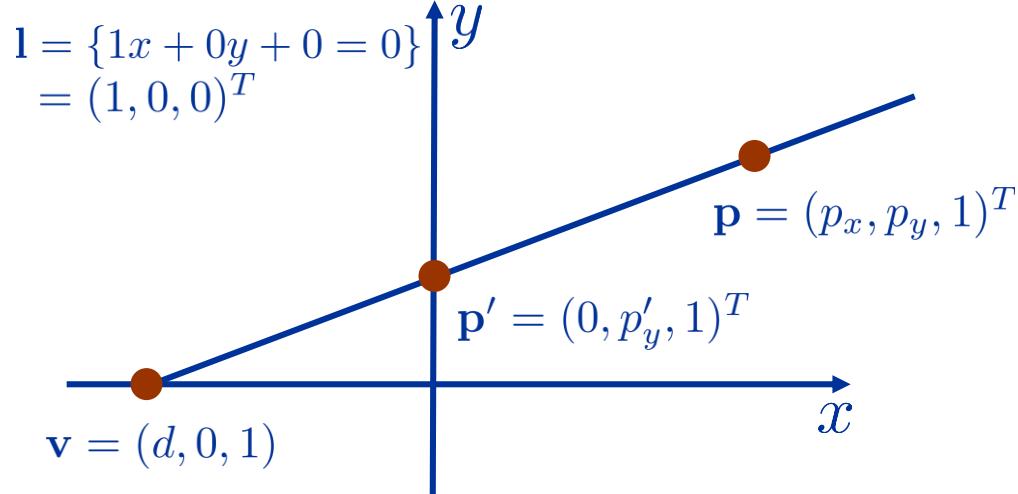
- parallel projection



$$\mathbf{M} = \mathbf{v}\mathbf{l}^T - (\mathbf{l} \cdot \mathbf{v})\mathbf{I}_3$$

$$\mathbf{M} = \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix} (1, 0, 0)^T - \left(\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix} \right) \mathbf{I}_3 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Discussion



$$p'_x = 0 \quad \frac{p_y}{p_x - d} = \frac{p'_y}{-d} \Rightarrow p'_y = \frac{-dp_y}{p_x - d} \quad p_w = 1 \Rightarrow p'_w = p_x - d$$

$$\Rightarrow \mathbf{M} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -d & 0 \\ 1 & 0 & -d \end{pmatrix} \quad \begin{array}{l} \text{maps } \mathbf{p} \text{ to } p'_x = 0 \\ \text{maps } \mathbf{p} \text{ to } p'_y = -d p_y \\ \text{maps } \mathbf{p} \text{ with } p_w=1 \text{ to } p'_w = p_x - d \end{array}$$

Discussion

- 2D transformation in homogeneous form

$$\mathbf{M} = \begin{pmatrix} m_{11} & m_{12} & t_x \\ m_{21} & m_{22} & t_y \\ w_x & w_y & h \end{pmatrix}$$

- w_x and w_y map the homogeneous component w of a point to a value w' that depends on x and y
- therefore, the scaling of a point depends on x and / or y
- in perspective 3D projections, this is generally employed to scale the x - and y - component with respect to z , its distance to the viewer

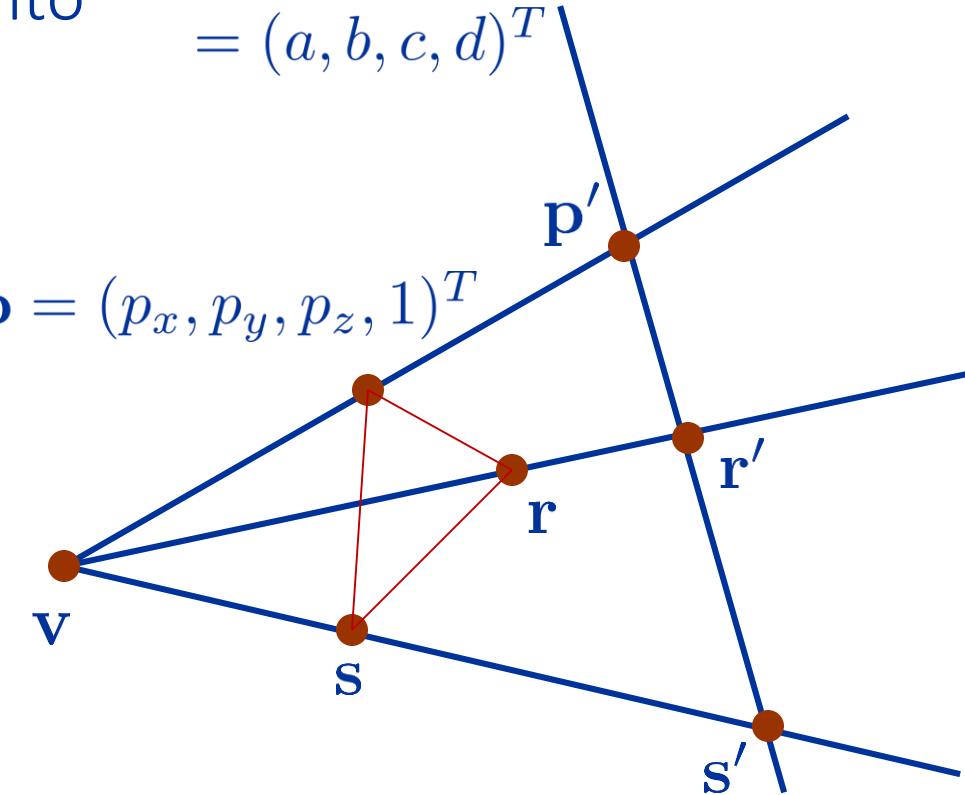
Outline

- 2D projection
- 3D projection
- OpenGL projection matrix
- OpenGL transformation matrices

Projection in 3D

- a 3D projection from v onto n maps a point p onto p'
- p' is the intersection of the line through p and v with plane n
- v is the viewpoint, center of perspectivity
- n is the viewplane
- the line through p and v is a projector
- v is not on the plane n , $p \neq v$

$$\begin{aligned}\mathbf{n} &= \{ax + by + cz + d = 0\} \\ &= (a, b, c, d)^T\end{aligned}$$



$$\mathbf{p} = (p_x, p_y, p_z, 1)^T$$

General Case

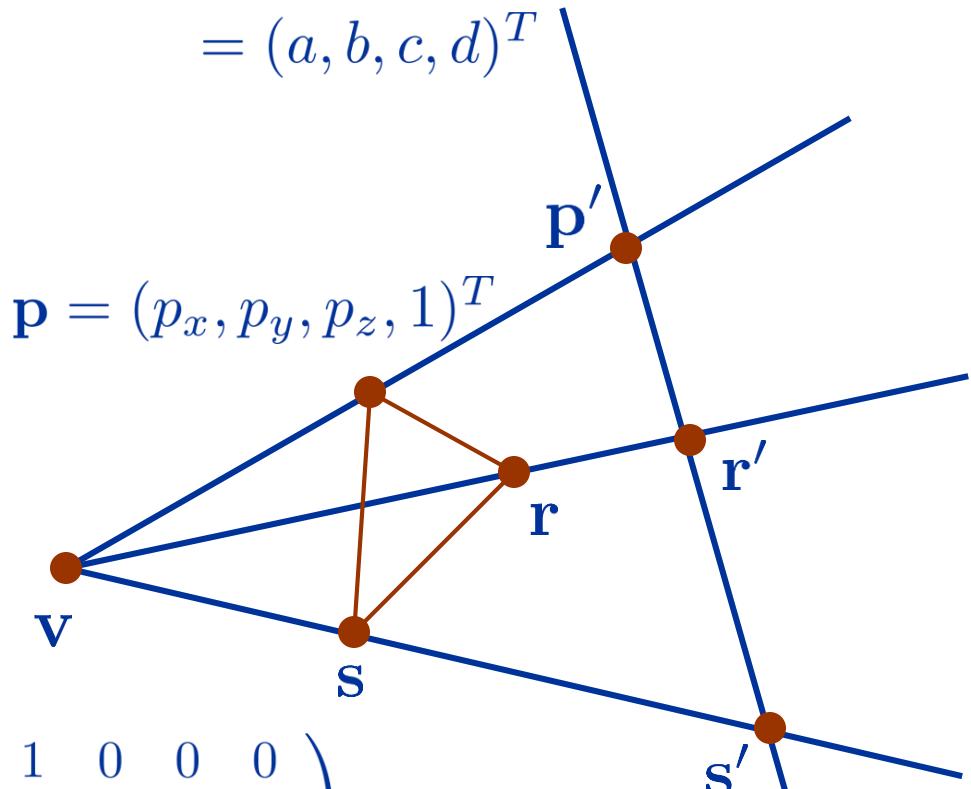
- a 3D projection is represented by a matrix

$$\mathbf{M} = \mathbf{v}\mathbf{n}^T - (\mathbf{n} \cdot \mathbf{v})\mathbf{I}_4$$

$$\mathbf{v}\mathbf{n}^T = \begin{pmatrix} v_x a & v_x b & v_x c & v_x d \\ v_y a & v_y b & v_y c & v_y d \\ v_z a & v_z b & v_z c & v_z d \\ v_w a & v_w b & v_w c & v_w d \end{pmatrix}$$

$$(\mathbf{n} \cdot \mathbf{v})\mathbf{I} = (av_x + bv_y + cv_z + dv_w) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

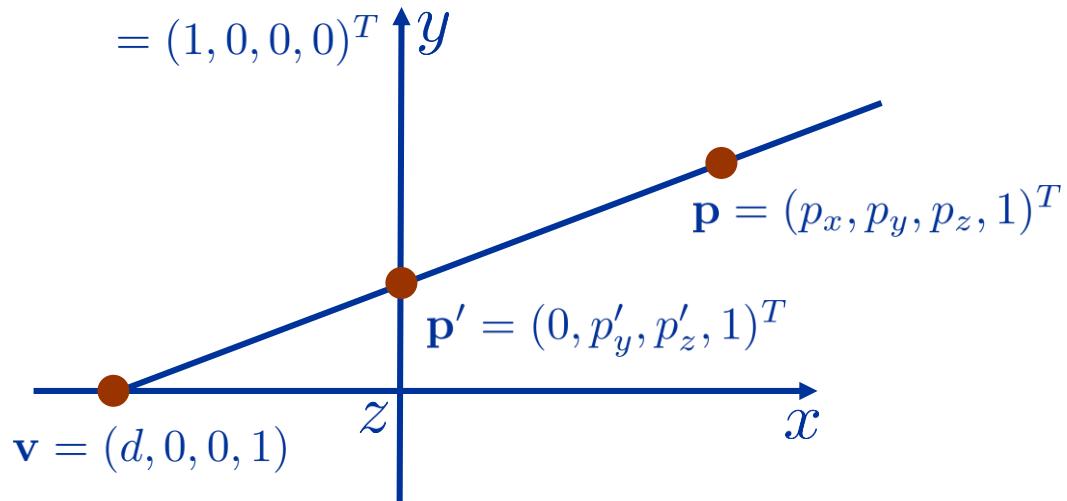
$$\begin{aligned} \mathbf{n} &= \{ax + by + cz + d = 0\} \\ &= (a, b, c, d)^T \end{aligned}$$



Example

$$\mathbf{n} = \{ax + by + cz + d = 0\}$$

$$= (1, 0, 0, 0)^T$$



- $\mathbf{M} = \begin{pmatrix} d \\ 0 \\ 0 \\ 1 \end{pmatrix} (1, 0, 0, 0) - \left(\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} d \\ 0 \\ 0 \\ 1 \end{pmatrix} \right) \mathbf{I}_4 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & -d & 0 & 0 \\ 0 & 0 & -d & 0 \\ 1 & 0 & 0 & -d \end{pmatrix}$
- e.g. $d=-1$, $(1,2,0)^T$ is mapped to $(0,1,0)^T$

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

Example

- parallel projection onto the plane $z = 0$ with viewpoint / viewing direction $\mathbf{v} = (0,0,1,0)^T$

$$\mathbf{n} = \{0x + 0y + 1z + 0 = 0\}$$

$$\mathbf{v} = (0, 0, 1, 0)^T$$

$$\mathbf{M} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} (0, 0, 1, 0) - \left(\left(\begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \right) \right) \mathbf{I}_4 = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

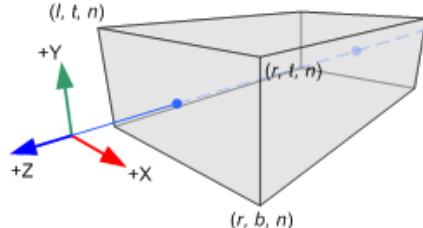
- x - and y -component are unchanged, z is mapped to zero
- remember that \mathbf{M} and $\lambda\mathbf{M}$ with, e. g., $\lambda=-1$ represent the same transformation

Outline

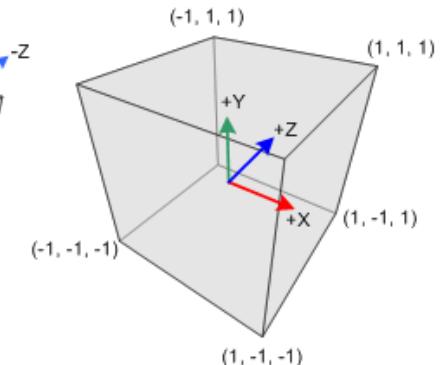
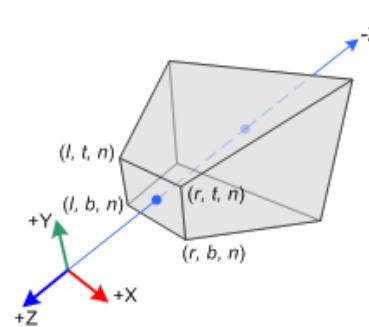
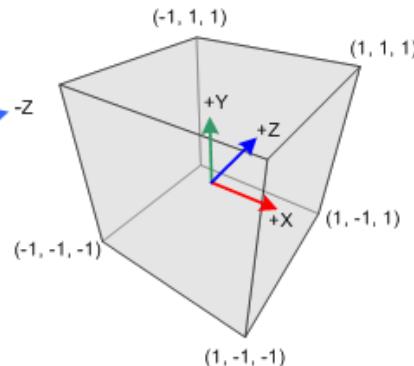
- 2D projection
- 3D projection
- OpenGL projection matrix
 - perspective projection
 - parallel projection
- OpenGL transformation matrices

View Volume

- in OpenGL, the projection transformation maps a view volume to the canonical view volume
- the view volume is specified by its boundary
 - left, right, bottom, top, near far
- the canonical view volume is a cube from $(-1, -1, -1)$ to $(1, 1, 1)$



[Song Ho Ahn]



this transformation implements
orthographic projection

University of Freiburg – Computer Science Department – Computer Graphics - 20

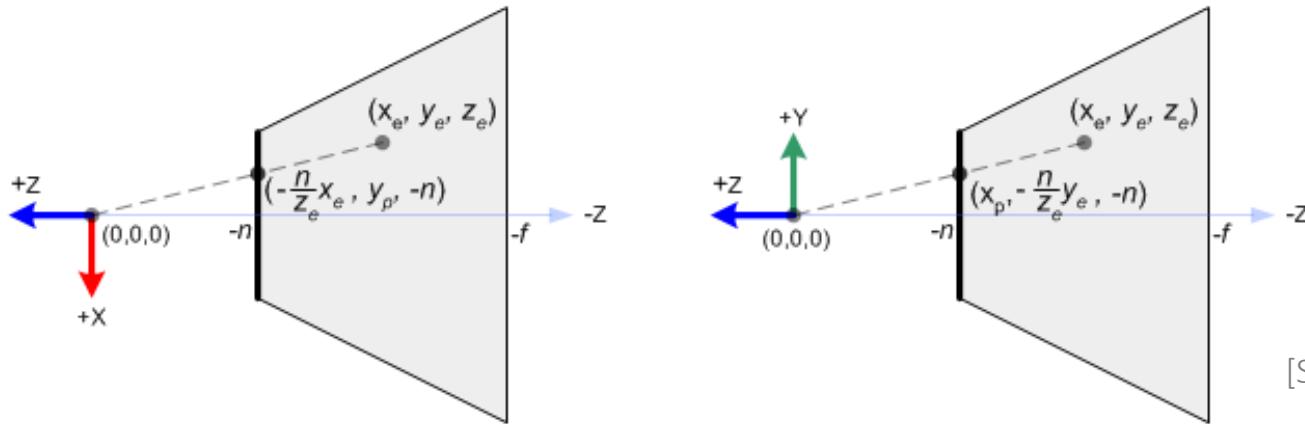
this transformation implements
perspective projection

OpenGL Projection Transform

- the projection transform maps
 - from eye coordinates
 - to clip coordinates (w-component is not necessarily one)
 - to normalized device coordinates NDC
(x and y are normalized with respect to w,
w is preserved for further processing)
- the projection transform maps
 - the x-component of a point from (left, right) to (-1, 1)
 - the y-component of a point from (bottom, top) to (-1, 1)
 - the z-component of a point from (near, far) to (-1, 1)
 - in OpenGL, near and far are negative, so the mapping incorporates a reflection (change of right-handed to left-handed)
 - however, in OpenGL functions, usually the negative of near and far is specified which is positive

Perspective Projection

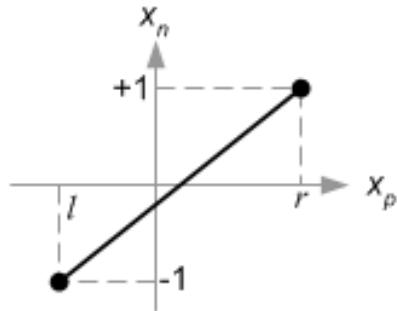
- to obtain x- and y-component of a projected point, the point is first projected onto the near plane (viewplane)



$$\frac{x_p}{x_e} = \frac{-n}{z_e} \Rightarrow x_p = \frac{nx_e}{-z_e} \quad \frac{y_p}{y_e} = \frac{-n}{z_e} \Rightarrow y_p = \frac{ny_e}{-z_e}$$

- note that n and f denote the negative near and far values

Mapping of x_p and y_p to (-1, 1)



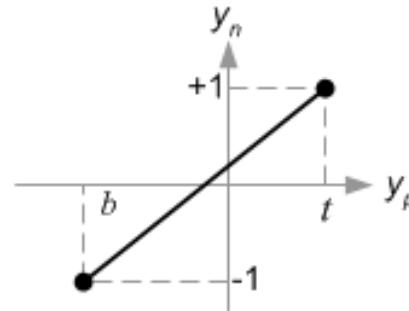
$$x_n = \alpha x_p + \beta$$

$$\alpha = \frac{1 - (-1)}{r - l}$$

$$\beta = -\frac{r + l}{r - l}$$

$$x_n = \frac{2x_p}{r - l} - \frac{r + l}{r - l}$$

$$x_n = \frac{1}{-z_e} \left(\frac{2n}{r - l} x_e + \frac{r + l}{r - l} z_e \right)$$



$$y_n = \alpha y_p + \beta$$

$$\alpha = \frac{1 - (-1)}{t - b}$$

$$\beta = -\frac{t + b}{t - b}$$

$$y_n = \frac{2y_p}{t - b} - \frac{t + b}{t - b}$$

$$y_n = \frac{1}{-z_e} \left(\frac{2n}{t - b} y_e + \frac{t + b}{t - b} z_e \right)$$

[Song Ho Ahn]

Projection Matrix

- from

$$x_n = \frac{1}{-z_e} \left(\frac{2n}{r-l} x_e + \frac{r+l}{r-l} z_e \right) \quad y_n = \frac{1}{-z_e} \left(\frac{2n}{t-b} y_e + \frac{t+b}{t-b} z_e \right)$$

- we get

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}$$

clip coordinates

- with

$$\begin{pmatrix} x_n \\ y_n \\ z_n \\ 1 \end{pmatrix} = \begin{pmatrix} x_c/w_c \\ y_c/w_c \\ z_c/w_c \\ w_c/w_c \end{pmatrix}$$

normalized device
coordinates

Mapping of z_e to (-1, 1)

- z_e is mapped from (near, far) or (-n, -f) to (-1, 1)
- the transform does not depend on x_e and y_e
- so, we have to solve for A and B in

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix}$$

$$z_n = \frac{z_c}{w_c} = \frac{Az_e + Bw_e}{-z_e}$$

Mapping of z_e to (-1, 1)

- $z_e = -n$ with $w_e = 1$ is mapped to $z_n = -1$
- $z_e = -f$ with $w_e = 1$ is mapped to $z_f = 1$

$$\Rightarrow A = -\frac{f+n}{f-n} \quad \Rightarrow B = -\frac{2fn}{f-n}$$

- the complete matrix is

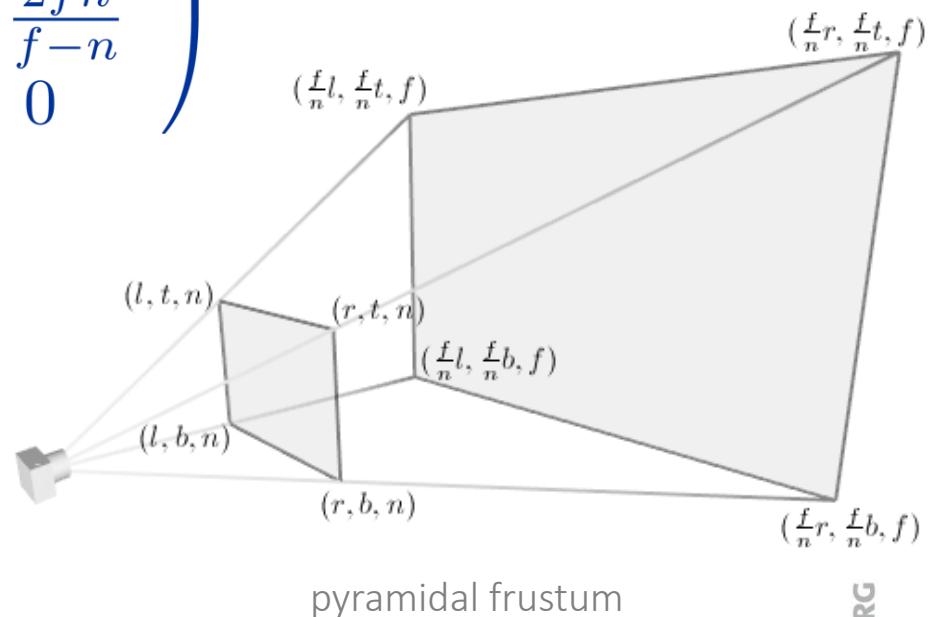
$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Perspective Projection Matrix

- the matrix

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

transforms the view volume, the pyramidal frustum to the canonical view volume



[Song Ho Ahn]

Perspective Projection Matrix

- projection matrix for negated values for n and f (OpenGL)

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

- projection matrix for actual values for n and f

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

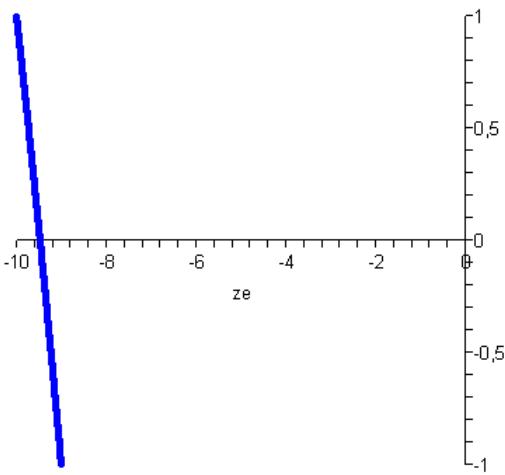
Symmetric Setting

- the matrix simplifies for $r = -l$ and $t = -b$

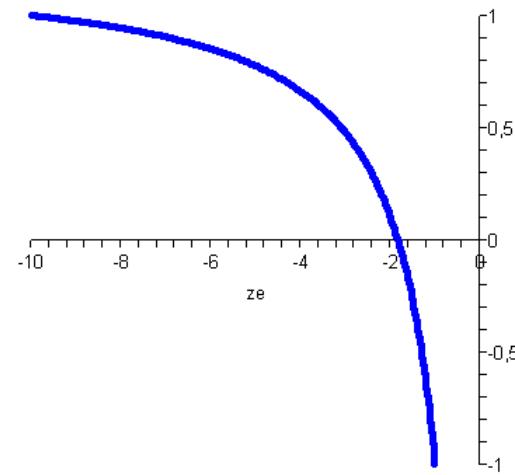
$$\begin{aligned} r + l &= 0 \\ r - l &= 2r \\ t + b &= 0 \\ t - b &= 2t \end{aligned} \Rightarrow \begin{pmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Near Plane

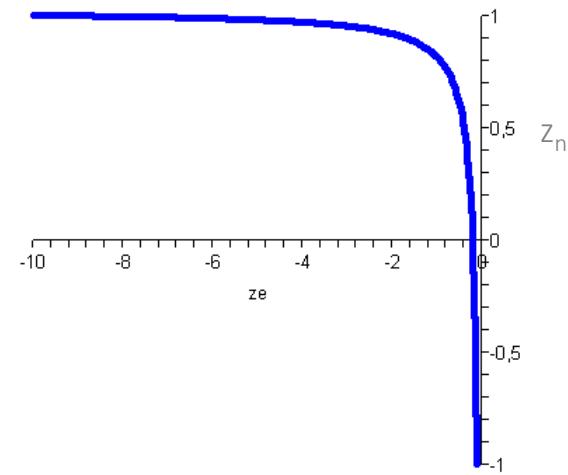
- nonlinear mapping of z_e : $z_n = \frac{f+n}{f-n} + \frac{1}{z_e} \frac{2fn}{f-n}$
- varying resolution / accuracy due to fix-point representation of depth values in the depth buffer



$$n = 9 \quad f = 10$$



$$n = 1 \quad f = 10$$



$$n = 0.1 \quad f = 10$$

- do not move the near plane too close to zero

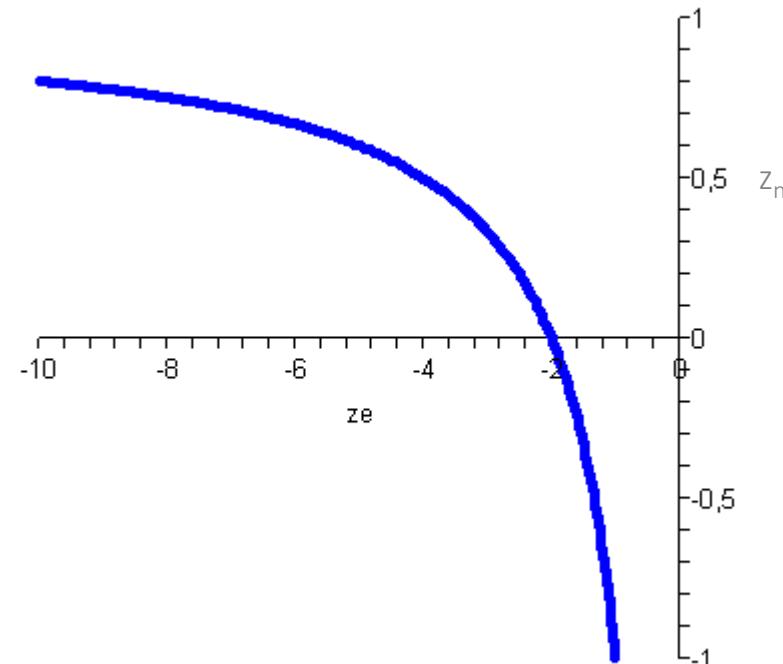
Far Plane

- setting the far plane to infinity is not too critical

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

$$f \rightarrow \infty$$

$$\Rightarrow \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -1 & -2n \\ 0 & 0 & -1 & 0 \end{pmatrix}$$



$$z_n = 1 + \frac{2}{z_e}$$

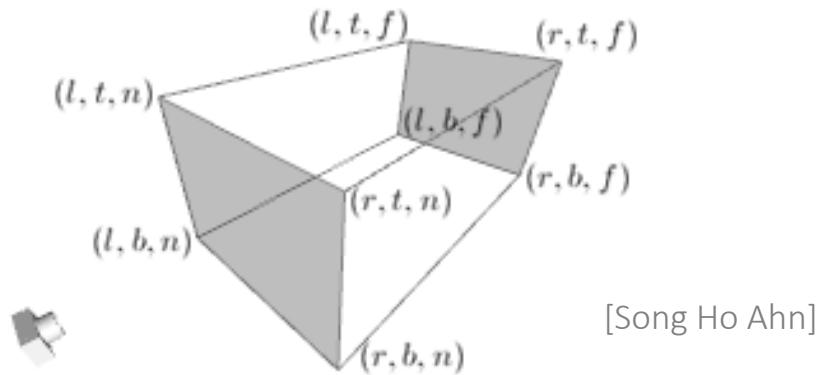
$$n = 1 \ f = \infty$$

Outline

- 2D projection
- 3D projection
- OpenGL projection matrix
 - perspective projection
 - parallel projection
- OpenGL transformation matrices

Parallel Projection

- the view volume is represented by a cuboid
 - left, right, bottom, top, near, far

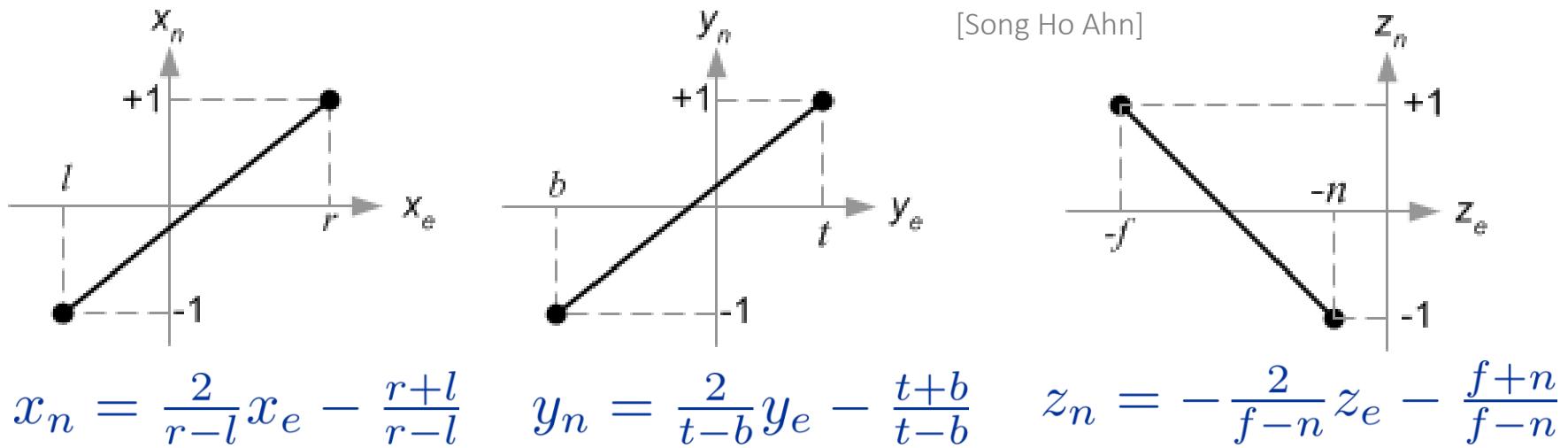


[Song Ho Ahn]

- the projection transform maps the cuboid to the canonical view volume

Mapping of x_e , y_e , z_e to (-1,1)

- all components of a point in eye coordinates are linearly mapped to the range of (-1,1)



- linear in x_e , y_e , z_e
- combination of scale and translation

Orthographic Projection Matrix

- general form

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

- simplified form for a symmetric view volume

$$r + l = 0$$

$$r - l = 2r$$

$$t + b = 0$$

$$t - b = 2t$$

$$\Rightarrow \begin{pmatrix} \frac{1}{r} & 0 & 0 & 0 \\ 0 & \frac{1}{t} & 0 & 0 \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Outline

- 2D projection
- 3D projection
- OpenGL projection matrix
- OpenGL transformation matrices

OpenGL Matrices

- objects are transformed from object to eye space with the GL_MODELVIEW matrix
GL_MODELVIEW · p
- objects are transformed from eye space to clip space with the GL_PROJECTION matrix
GL_PROJECTION · GL_MODELVIEW · p
- colors are transformed with the color matrix GL_COLOR
- texture coordinates are transformed with the texture matrix GL_TEXTURE

Matrix Stack

- each matrix type has a stack
 - the matrix on top of the stack is used
-
- `glMatrixMode(GL_PROJECTION);` choose a matrix stack
`glLoadIdentity();` the top element is replaced with I_4
`glFrustum(left, right, bottom, top, near, far);` projection matrix P is generated
the top element on the stack is multiplied with P resulting in $I_4 \cdot P$

Matrix Stack

- `glMatrixMode(GL_MODELVIEW) ;`
`glLoadIdentity() ;`

`glTranslatef(x, y, z) ;`

`glRotatef(alpha, 1, 0, 0) ;`
- choose a matrix stack
the top element is replaced with I_4
translation matrix T is generated
the top element on the stack is multiplied with T resulting in $I_4 \cdot T$
- rotation matrix R is generated
the top element on the stack is multiplied with R resulting in $I_4 \cdot T \cdot R$
- note that objects are rotated by R,
followed by the translation T

Matrix Stack

- `glMatrixMode(GL_MODELVIEW);`
`glLoadIdentity();`
`glTranslatef(x, y, z);`
`glRotatef(alpha, 1, 0, 0);`
 - `glPush();`
 - `glTranslatef(x, y, z);`
 - `glPop();`
- choose a matrix stack
the top element is replaced with I_4
the top element is $I_4 \cdot T$
the top element is $I_4 \cdot T \cdot R$
- the top element $I_4 \cdot T \cdot R$
is pushed into the stack
the newly generated top element
is initialized with $I_4 \cdot T \cdot R$
- the top element is $I_4 \cdot T \cdot R \cdot T$
- the top element is replaced by
the previously stored element $I_4 \cdot T \cdot R$

OpenGL Matrix Functions

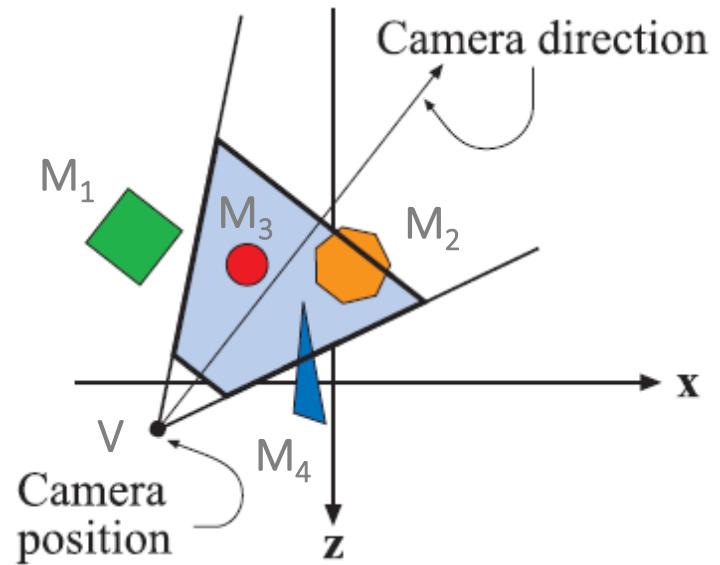
- `glPushMatrix()` : push the current matrix into the current matrix stack.
- `glPopMatrix()` : pop the current matrix from the current matrix stack.
- `glLoadIdentity()` : set the current matrix to the identity matrix.
- `glLoadMatrix{fd} (m)` : replace the current matrix with the matrix m .
- `glLoadTransposeMatrix{fd} (m)` : replace the current matrix with the row-major ordered matrix m .
- `glMultMatrix{fd} (m)` : multiply the current matrix by the matrix m , and update the result to the current matrix.
- `glMultTransposeMatrix{fd} (m)` : multiply the current matrix by the row-major ordered matrix m , and update the result to the current matrix.
- `glGetFloatv(GL_MODELVIEW_MATRIX, m)` : return 16 values of `GL_MODELVIEW` matrix to m .

- note that OpenGL functions expect column-major matrices in contrast to commonly used row-major matrices

Modelview Example

- objects are transformed with $V^{-1}M$
- $V = T_v R_v$ the camera is oriented and then translated
- $M_{1..4} = T_{1..4} R_{1..4}$ objects are oriented and then translated
- implementation

- choose the GL_MODELVIEW stack
 - initialize with I_4
 - rotate with R_v^{-1}
 - translate with T_v^{-1}
 - push
 - translate with T_1
 - rotate with R_1
 - render object M_1
 - pop
 - ...
- $$\begin{array}{ll} I_4 & \\ R_v^{-1} & \\ R_v^{-1} \cdot T_v^{-1} = V^{-1} & \\ R_v^{-1} \cdot T_v^{-1} & \\ R_v^{-1} \cdot T_v^{-1} \cdot T_1 & \\ R_v^{-1} \cdot T_v^{-1} \cdot T_1 \cdot R_1 & \\ R_v^{-1} \cdot T_v^{-1} & \end{array}$$



[Akenine-Moeller et al.:
Real-time Rendering]

Summary

- 2D projection
- 3D projection
- OpenGL projection matrix
 - perspective projection
 - parallel projection
- OpenGL transformation matrices

References

- Duncan Marsh: "Applied Geometry for Computer Graphics and CAD", Springer Verlag, Berlin, 2004.
- Song Ho Ahn: "OpenGL", <http://www.songho.ca/> .

Image Processing and Computer Graphics

Lighting

Matthias Teschner

Computer Science Department
University of Freiburg

Albert-Ludwigs-Universität Freiburg



Motivation

- modeling of visual phenomena
 - light is emitted by light sources,
e.g. sun or lamp
 - light interacts with objects
 - light is absorbed or scattered (reflected)
at surfaces
 - light is absorbed by a sensor, e.g. human eye or camera



[Akenine-Möller et al.]

Outline

- light
- color
- lighting models
- shading models

Light

- modeled as
 - electromagnetic waves, radiation
 - photons
 - geometric rays
- photons
 - particles
 - characterized by wavelength
(perceived as color in the visible spectrum)
 - carry energy
(inversely proportional to the wavelength)
 - travel along a straight line at the speed of light

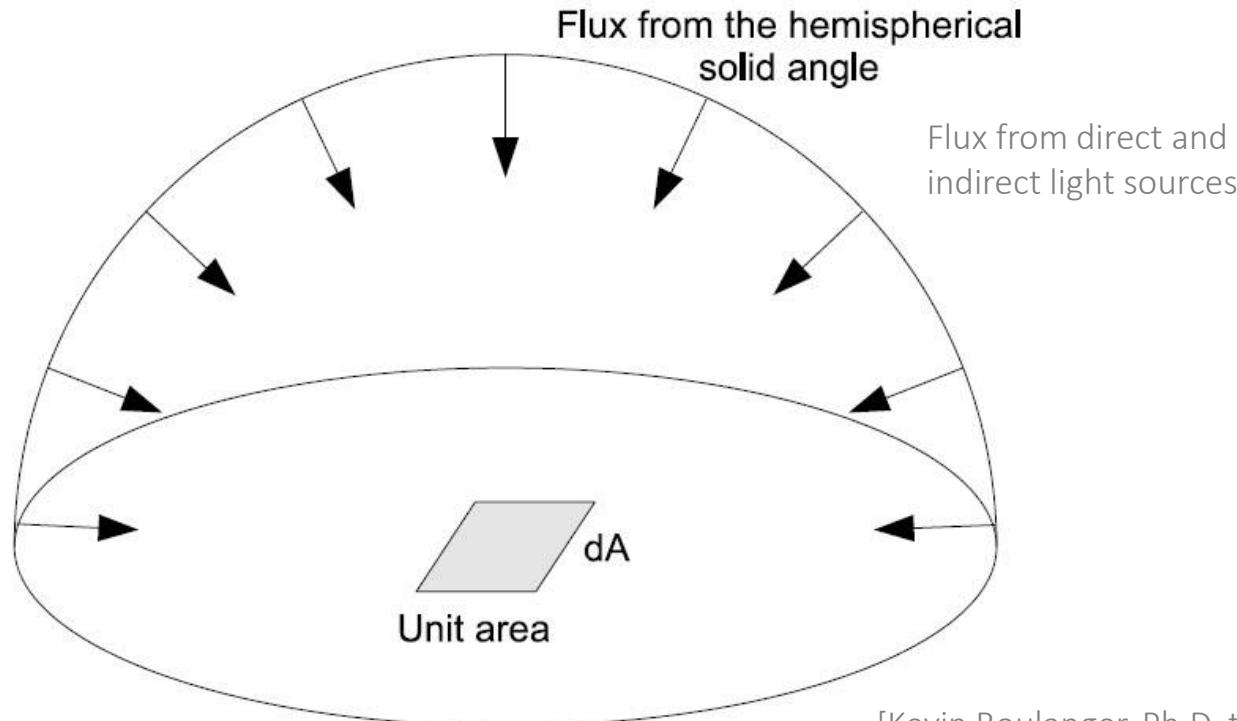
Radiometric Quantities

- radiant energy Q
 - photons have some radiant energy
- radiant flux Φ , radiant power P
 - rate of flow of radiant energy per unit time: $\Phi = \frac{dQ}{dt}$
 - e.g., overall energy of photons emitted by a source per time
- flux density (irradiance, radiant exitance)
 - radiant flux per unit area: $E = \frac{d\Phi}{dA}$
 - rate at which radiation is incident on,
or exiting a flat surface area dA
 - describes strength of radiation with respect to a surface area
 - no directional information

Radiometric Quantities

Irradiance

- irradiance measures the overall radiant flux Φ (light flow, photons per unit time) into a surface element



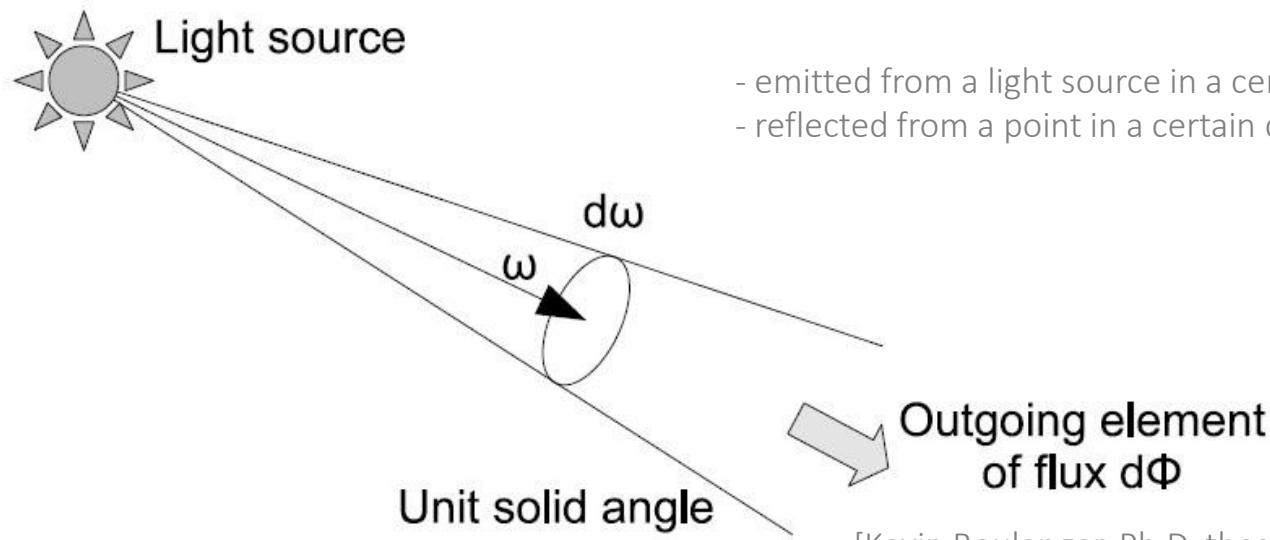
[Kevin Boulanger, Ph.D. thesis]

Radiometric Quantities

Radiant Intensity

- radiant intensity

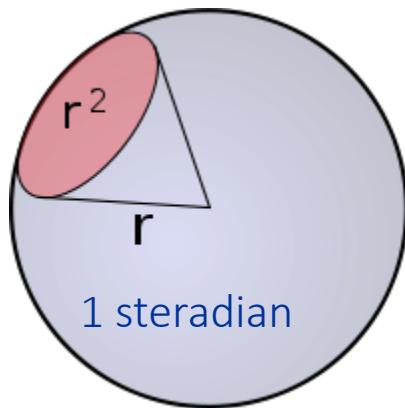
- radiant flux per unit solid angle: $I = \frac{d\Phi}{d\omega}$
- radiant flux Φ (light flow, photons per unit time) incident on, emerging from, or passing through a point in a certain direction



[Kevin Boulanger, Ph.D. thesis]

Solid Angle

- 2D angle in 3D space
- measured in steradians
 - A steradian is the solid angle subtended at the center of a sphere of radius r by a portion of the sphere surface with area $A=r^2$.



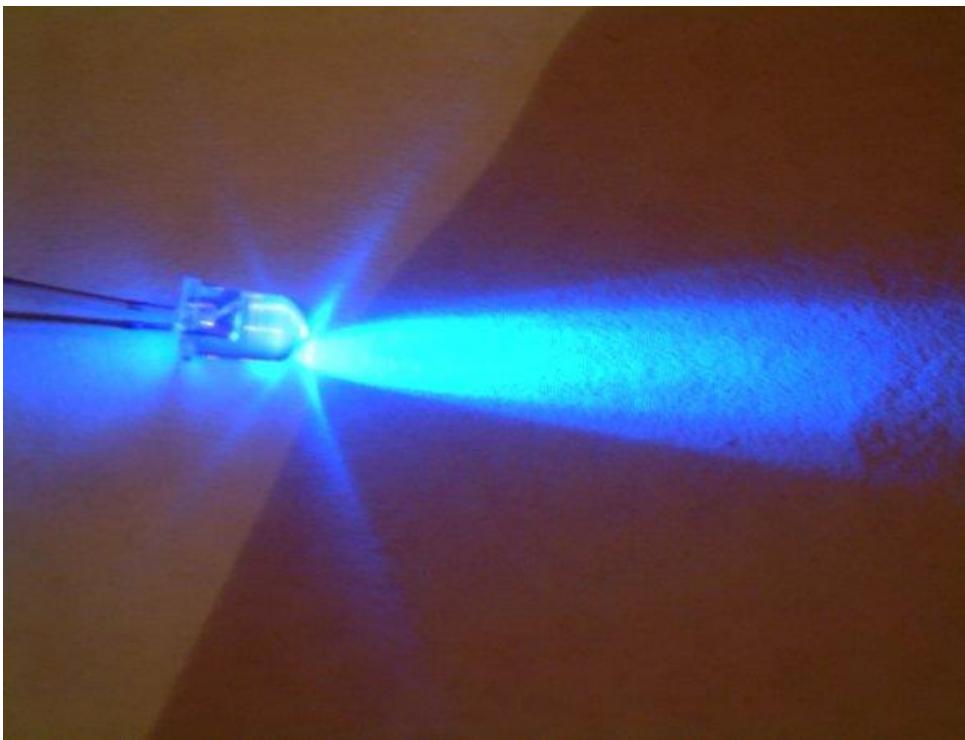
$$d\omega = \frac{dA}{r^2}$$

[Wikipedia: Solid angle]

Radiometric Quantities

Radiant Intensity

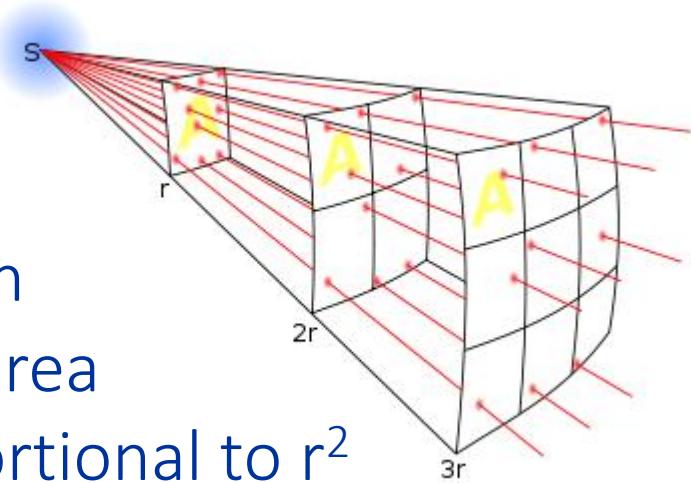
- light source with direction-dependent radiant intensities



[Wikipedia: Strahldichte]

Inverse Square Law

- point light source with radiant intensity I in direction ω
- irradiance along a ray in direction ω is inversely proportional to the square of the distance r from the source $E = \frac{I}{r^2}$
- the number of photons emitted in direction $d\omega$ and hitting surface area dA at distance r is inversely proportional to r^2
- the area subtended by a solid angle is proportional to r^2
- $E = \frac{d\Phi}{dA} = \frac{d\Phi}{r^2 d\omega} = \frac{I}{r^2}$



[Wikipedia: Inverse Square Law]

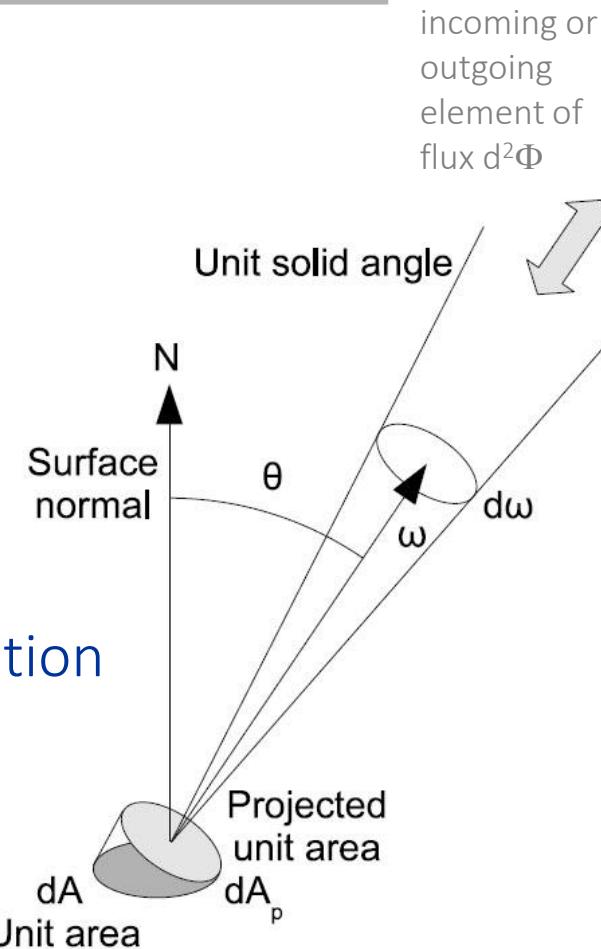
Radiometric Quantities

Radiance

- radiance
 - radiant flux per unit solid angle per unit projected area incident on, emerging from, passing through a surface element in a certain direction:

$$L = \frac{d^2\Phi}{dA_p d\omega} = \frac{d^2\Phi}{dA \cos \theta d\omega}$$

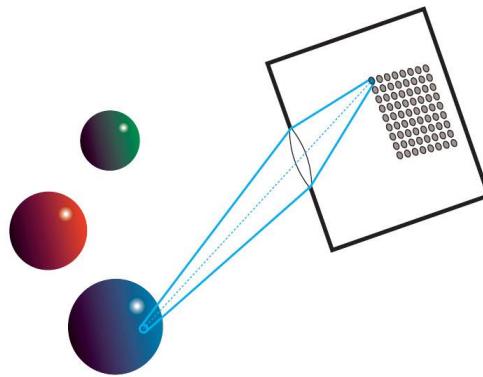
- describes strength and direction of radiation
- constant radiance in all directions corresponds to a radiant intensity that is proportional to $\cos \theta$, but constant radiant intensity results in different radiance



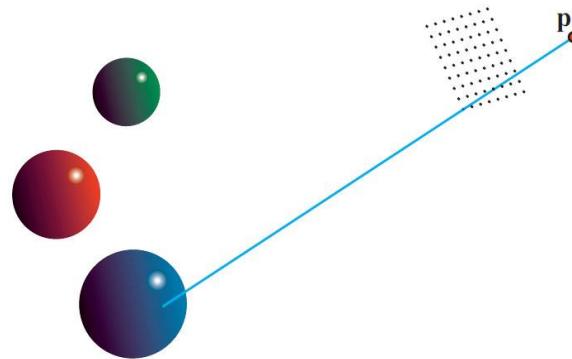
[Kevin Boulanger, Ph.D. thesis]

Radiance and Sensors

- radiance
 - is measured by sensors
 - is computed in computer-generated images
 - is preserved along lines in space
 - does not change with distance



a sensor with a small area receives light from a small set of directions, i. e. radiance



idealized graphics model of an imaging sensor

[Akenine-Möller et al.]

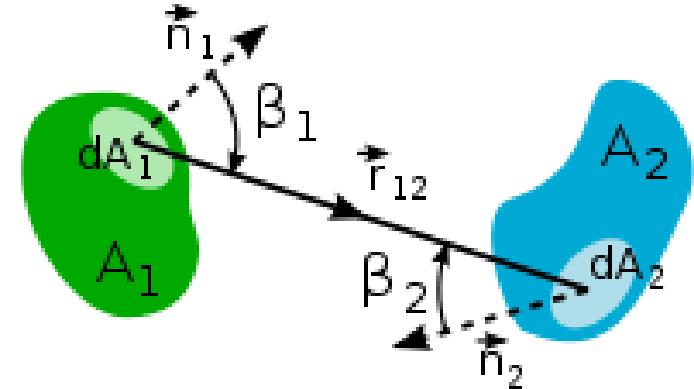
Conservation of Radiance

- outgoing flux from dA_1 into direction r_{12}

$$d^2\Phi_1 = L_1 \cdot \cos \beta_1 \cdot dA_1 \cdot d\omega_1$$

$$d\omega_1 = \cos \beta_2 \cdot dA_2 \cdot r^{-2}$$

$$d^2\Phi_1 = \frac{L_1 \cdot \cos \beta_1 \cdot \cos \beta_2 \cdot dA_1 \cdot dA_2}{r^2}$$



- incoming flux to dA_2 from direction $-r_{12}$

$$d^2\Phi_2 = L_2 \cdot \cos \beta_2 \cdot dA_2 \cdot d\omega_2 = \frac{L_2 \cdot \cos \beta_1 \cdot \cos \beta_2 \cdot dA_1 \cdot dA_2}{r^2}$$

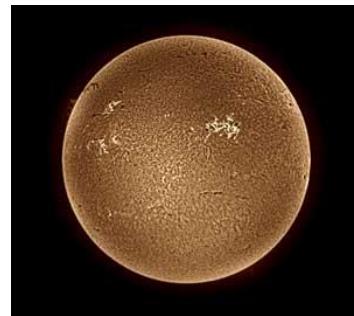
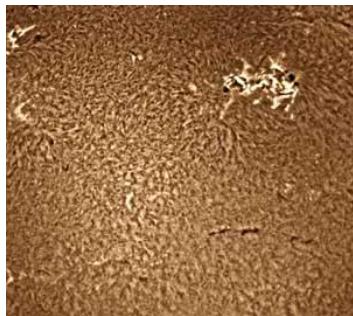
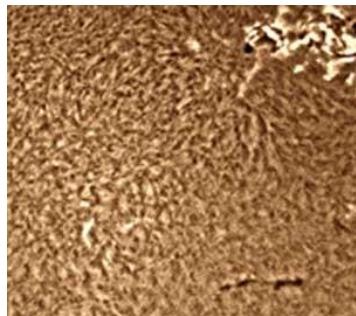
- if flux is preserved, the radiance does not change

$$d^2\Phi_1 = d^2\Phi_2 \Leftrightarrow L_1 = L_2$$

[Wikipedia: Strahldichte]

Radiance and Sensors

- three imaginary photos of the Sun from different distances



J. J. Condon and S. M. Ransom:
"Essential Radio Astronomy", National Radio Astronomy Observatory

- radiance in a pixel does not depend on the distance
 - irradiance of a pixel received from dA on the Sun is proportional to $1/r^2$
 - dA on the Sun captured by a pixel in directions $d\omega$ is proportional to r^2

Radiometric Quantities

radiometric quantity	symbol	unit	German
radiant energy	Q	J	Strahlungsenergie
radiant flux	Φ	W	Strahlungsfluss
flux density (irradiance, radiant exitance)	E	$\text{W}\cdot\text{m}^{-2}$	Strahlungsstromdichte
radiant intensity	I	$\text{W}\cdot\text{sr}^{-1}$	Strahlungsstärke / Strahlungsintensität
radiance	L	$\text{W}\cdot\text{m}^{-2}\cdot\text{sr}^{-1}$	Strahldichte

Photometric / Radiometric Quantities

Fotometrische Größe	Einheit	Strahlungsgröße	Einheit
Lichtmenge	$\text{lm} \cdot \text{s}$	Strahlungsenergie	J
Lichtstrom	lm	Strahlungsfluss	W
Beleuchtungsstärke	lx	Bestrahlungsstärke	
Lichtausstrahlung	$\text{lm} \cdot \text{m}^{-2}$	Ausstrahlung	$\text{W} \cdot \text{m}^{-2}$
Lichtstärke	cd	Strahlungsstärke / Strahlungsintensität	$\text{W} \cdot \text{sr}^{-1}$
Leuchtdichte	$\text{cd} \cdot \text{m}^{-2}$	Strahldichte	$\text{W} \cdot \text{m}^{-2} \cdot \text{sr}^{-1}$

Summary

- light consists of photons
- irradiance and radiant exitance describe the flux,
i.e. the number of photons per time, into or from
a surface per area
- radiant intensity is flux into a direction per solid angle
- radiance describes the flow into or from a surface
from a certain direction per area per solid angle
- radiance is measured in sensors
- irradiance of a surface is inversely proportional to
its squared distance for a point source with a certain
radiant intensity
- radiance is preserved along straight (empty) lines

Outline

- light
- color
- lighting models
- shading models

Visible Spectrum

- photons are characterized by a wavelength within the visible spectrum from 390 nm to 750 nm
- light consists of a set of photons
- the distribution of wavelengths within this set is referred to as the spectrum of light
- spectra are perceived as colors

Visible Spectrum

- if the spectrum consists of a dominant wavelength, humans perceive a "rainbow" color (monochromatic)



[Wikipedia: Visible spectrum]

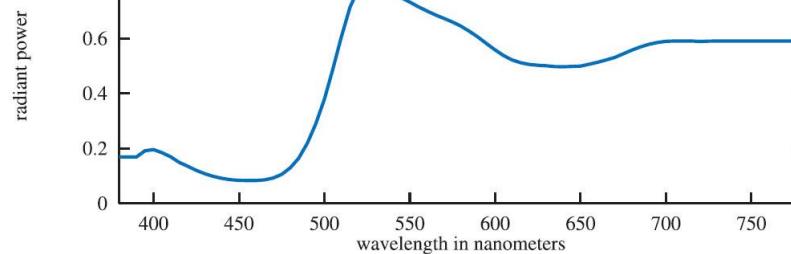
390 nm

750 nm

- equally distributed wavelengths are perceived as a shade of gray, ranging from black to white (achromatic)
- otherwise, colors "mixed from rainbow colors" are perceived (chromatic)

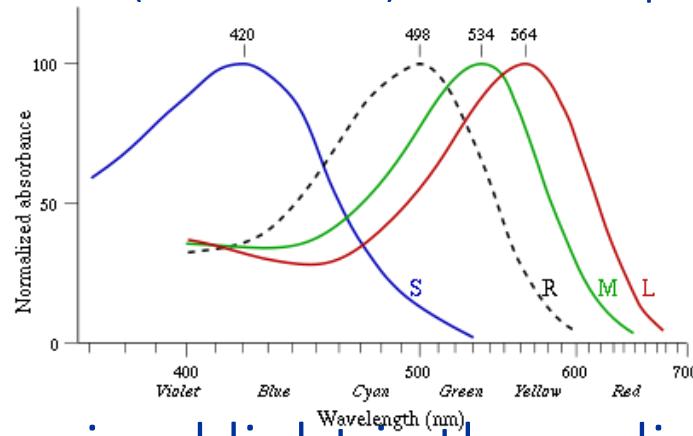
This spectrum corresponds to
a ripe brown banana under white light

[Akenine-Möller et al.]



Human Eye

- is sensitive to radiation within the visible spectrum
- has sensors for day and night vision
 - three types of cones (Zapfen) for photopic (day) vision
 - rods (Stäbchen) for scotopic (night) vision



Normalised absorption spectra of human cone (S,M,L) and rod (R) cells. Cones are sensitive to blue, green, red.

[Wikipedia: Trichromacy]

- perceived light is the radiation spectrum weighted with absorption spectra (sensitivity) of the eye
- in daylight, three cone signals are interpreted by the brain

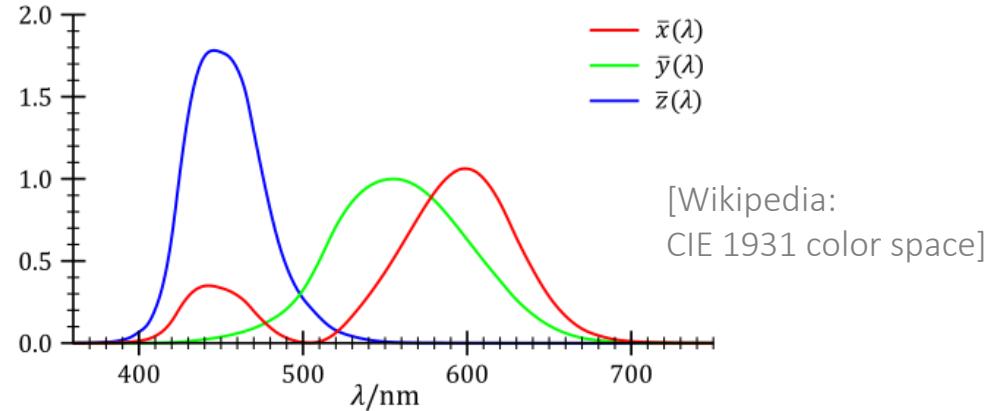
CIE Color Space

- XYZ color space
 - created by the Int. Commission on Illumination CIE in 1931

$$X = \int_{\lambda} I(\lambda) \bar{x}(\lambda) d\lambda$$

$$Y = \int_{\lambda} I(\lambda) \bar{y}(\lambda) d\lambda$$

$$Z = \int_{\lambda} I(\lambda) \bar{z}(\lambda) d\lambda$$



- spectrum I is represented by X, Y, Z given the color-matching functions $\bar{x}, \bar{y}, \bar{z}$
- the color-matching functions correspond to the spectral sensitivities of the cones of a standard observer

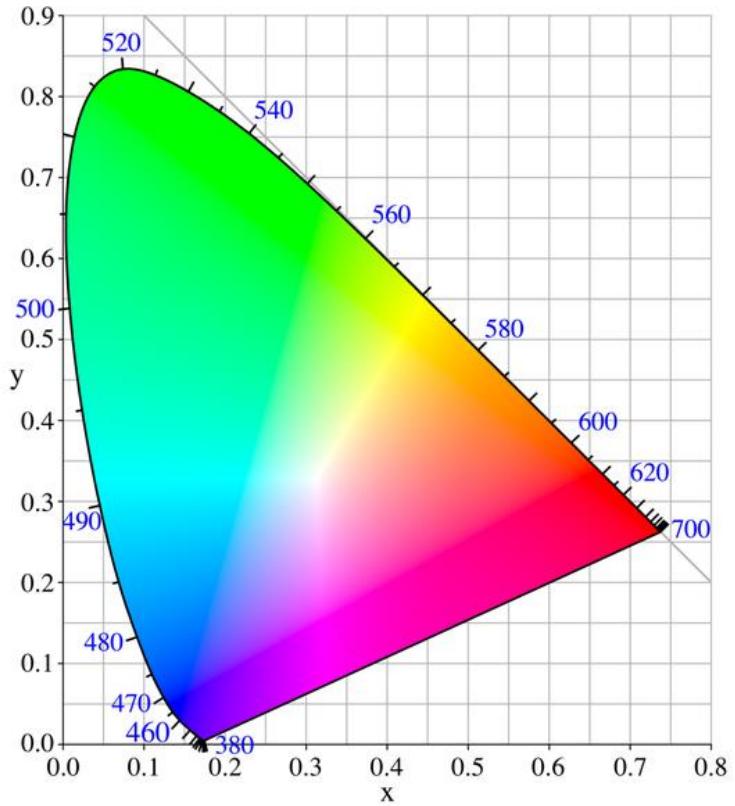
CIE xy Chromaticity Diagram

- XYZ represents color and brightness / luminance
- two values are sufficient to represent color

$$x = \frac{X}{X+Y+Z}$$

$$y = \frac{Y}{X+Y+Z}$$

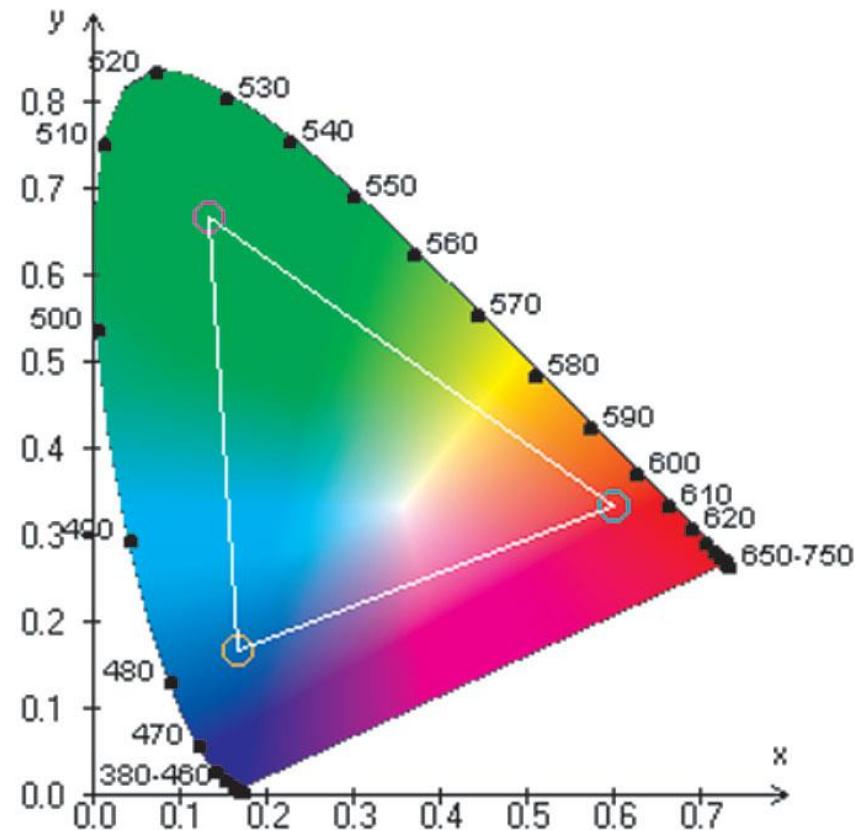
- monochromatic colors are on the boundary
- the center is achromatic



[Wikipedia: CIE 1931 color space]

Display Devices

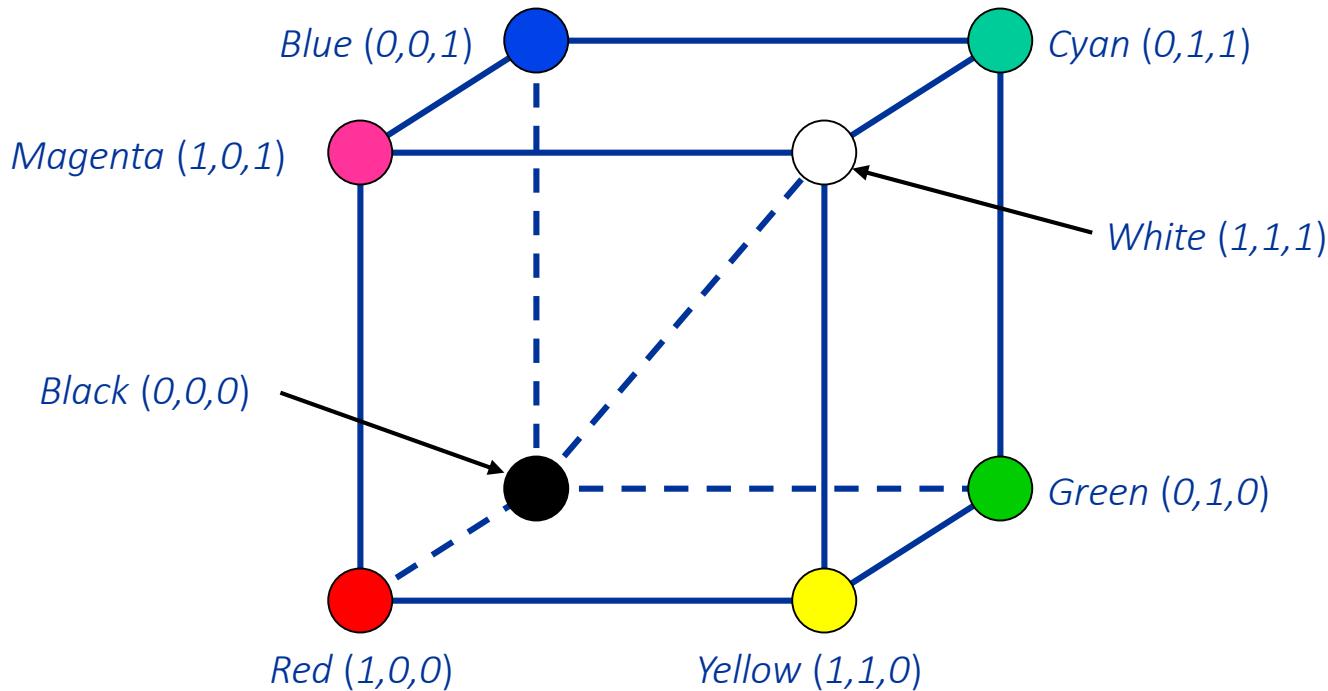
- use three primary colors
 - an example is indicated in the diagram
- can only reproduce colors within the spanned triangle (gamut)
- → colors outside the gamut are not properly displayed on a monitor



[Akenine-Möller et al.]

RGB Color Space

- three primaries: red, green, blue



RGB Color Space

Lights and Objects

- light source color
 - e.g., yellow light (1, 1, 0)
 - emits a spectrum with maximum red and green components
 - the spectrum does not contain any blue
 - the RGB values describe the amount of the respective color component in the emitted light
- object color
 - e.g., yellow object (1, 1, 0)
 - perfectly reflects red and green comp. of the incoming light
 - perfectly absorbs the blue component of the incoming light
 - the RGB values describe how much of the respective incoming color component is reflected ("one minus value" describes how much is absorbed)

Summary

- the distribution of wavelengths within the flux of the perceived radiance is referred to as the spectrum of light
- spectra are weighted with absorption spectra of the eye and perceived as colors
- three cone types for daylight vision motivate XYZ space
- RGB space is often used for display devices
- colors of display devices are restricted to a gamut that does not contain all perceivable colors

Outline

- light
- color
- lighting models
- shading models

Motivation

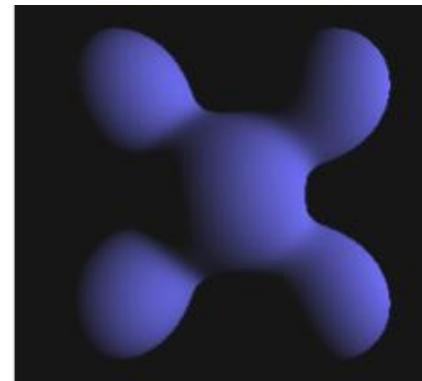
- compute the interaction of objects with light based on an illumination model (reflection model, lighting model)
- account for a variety of light sources and material properties
- but, keep it fast
 - only use local information per vertex / fragment
 - light source color, direction, and distance
 - object color (material) and surface normal
 - viewer direction
 - interaction between objects is neglected
 - interreflections among objects, occluded light sources or areal light sources are not handled

Outline

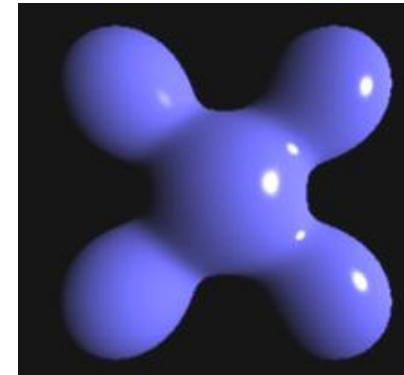
- light
- color
- lighting models
 - diffuse reflection
 - specular reflection
 - ambient light
 - Phong illumination model
 - miscellaneous
- shading models

Diffuse vs. Specular

- diffuse
 - incident light is reflected into many different directions
 - matte surfaces
- specular (mirror-like)
 - incident light is reflected into a dominant direction (perceived as small intense specular highlight)
 - shiny surfaces
- diffuse and specular reflection are material properties



ideal diffuse
reflecting surface

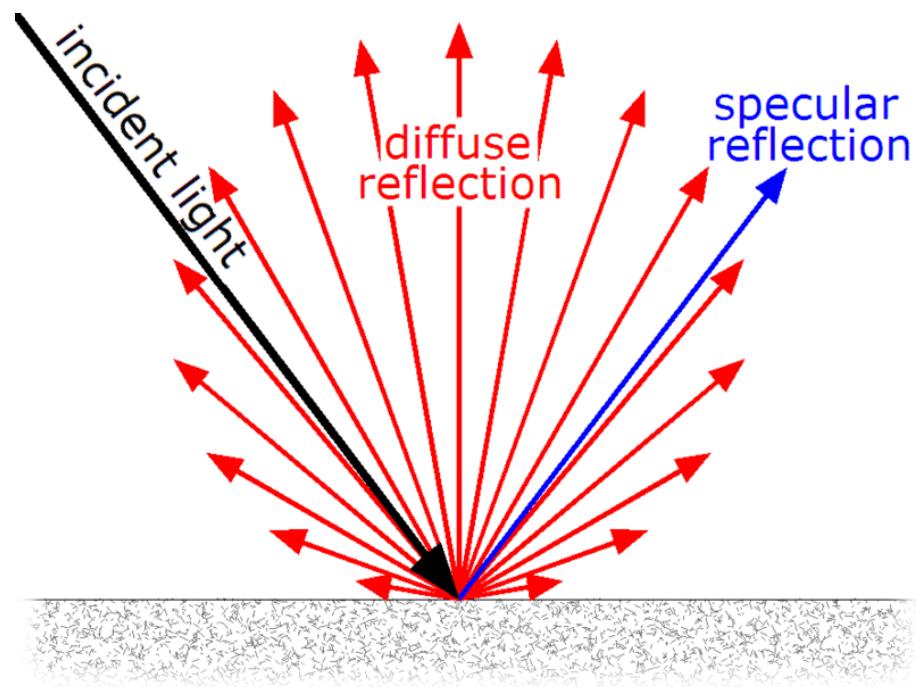


diffuse and specular
reflecting surface

[Wikipedia: Phong Shading]

Diffuse vs. Specular

- paper and marble have diffuse reflecting surfaces
- in general, materials are characterized by a combination of diffuse and specular reflection

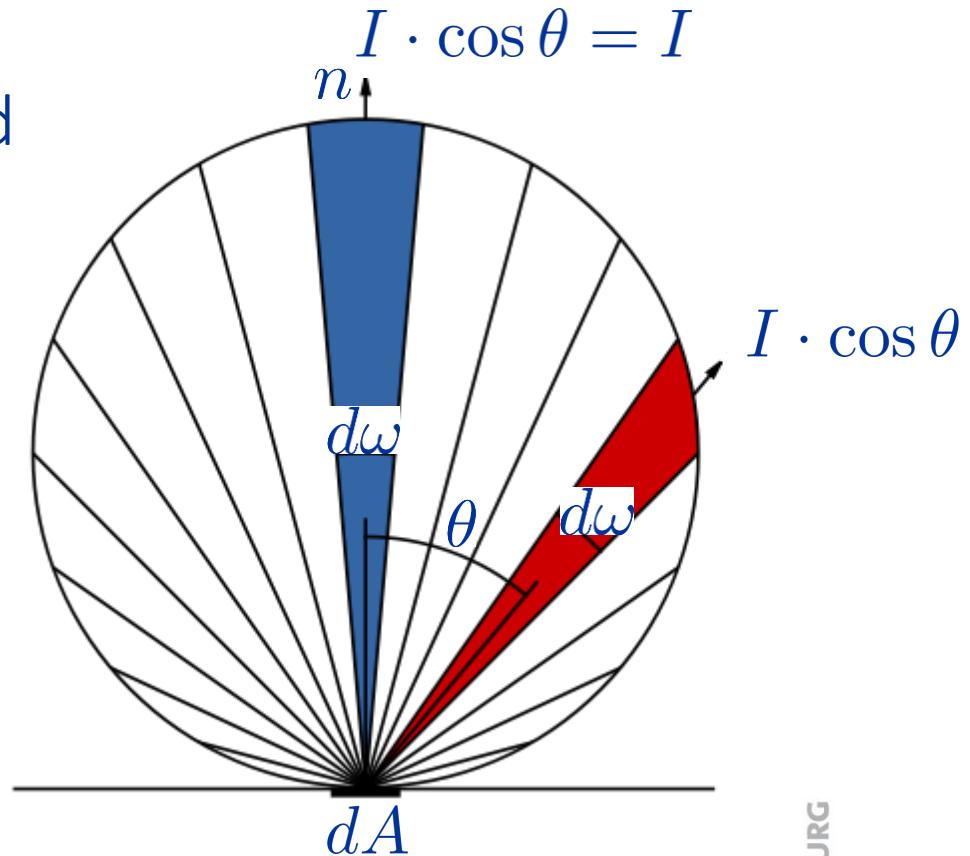


incident and reflected radiant flux of a diffuse and specular reflecting surface

[Wikipedia: Diffuse Reflection]

Lambert's Cosine Law

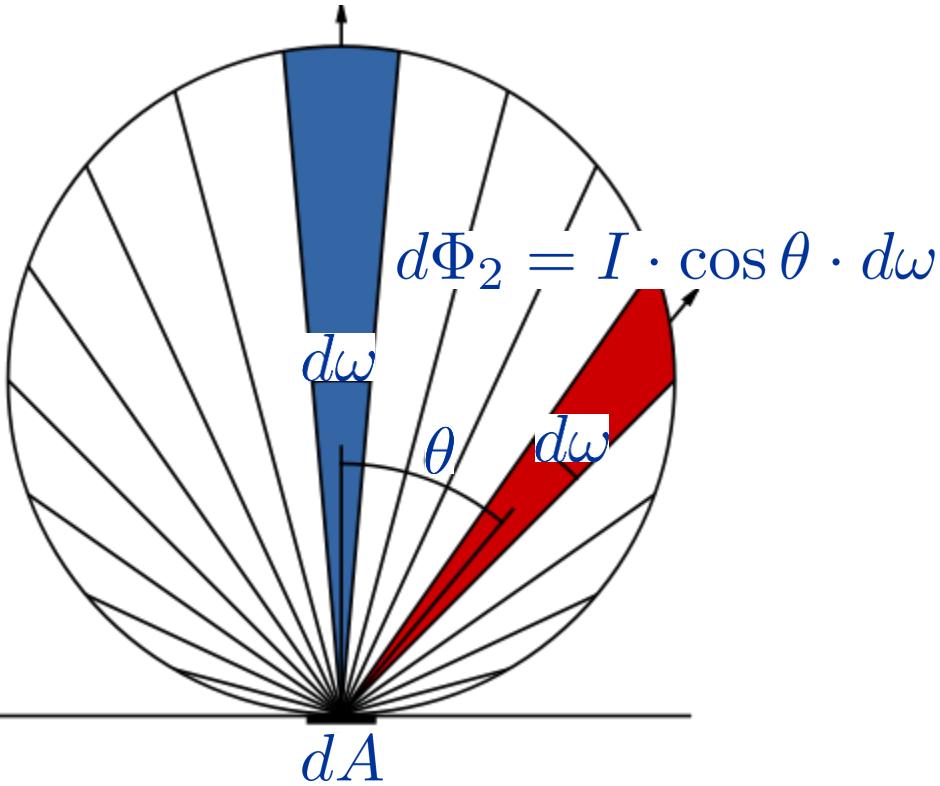
- computation of diffuse reflection is governed by Lambert's cosine law
- radiant intensity reflected from a "Lambertian" (matte, diffuse) surface is proportional to the cosine of the angle between view direction and surface normal n



[Wikipedia: Lambert's Cosine Law]

Lambert's Cosine Law

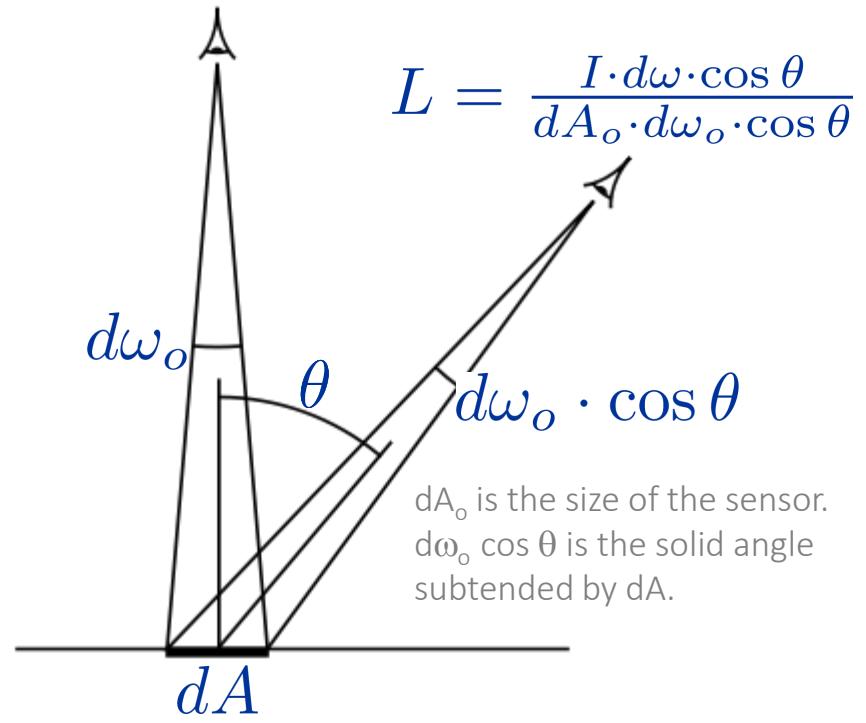
$$d\Phi_1 = I \cdot \cos 0 \cdot d\omega = I \cdot d\omega$$



Radiant intensity I and flux Φ are proportional to the cosine of the angle θ .

[Wikipedia: Lambert's Cosine Law]

$$L = \frac{I \cdot d\omega}{dA_o \cdot d\omega_o}$$

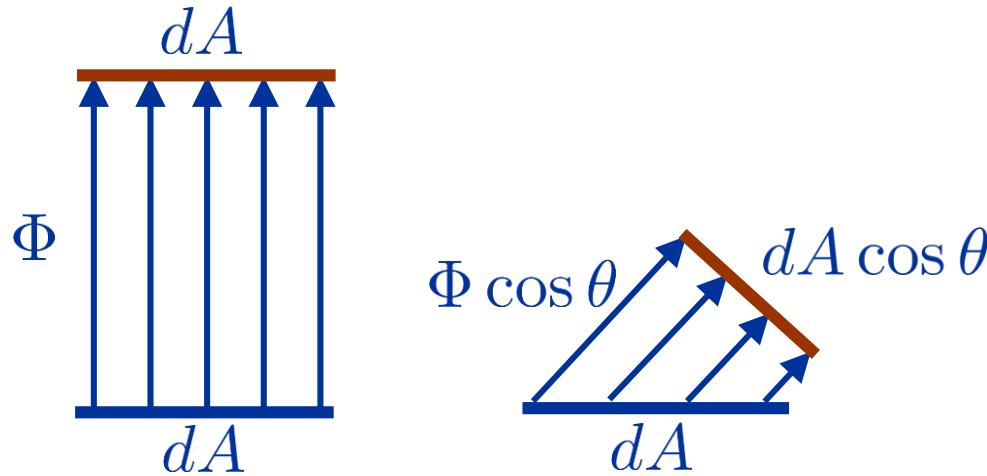


dA_o is the size of the sensor.
 $d\omega_o \cos \theta$ is the solid angle
subtended by dA .

Radiance L measured at a sensor is independent from angle θ and distance. Computation of diffuse reflection is independent from the viewer direction and distance.

Lambert's Cosine Law

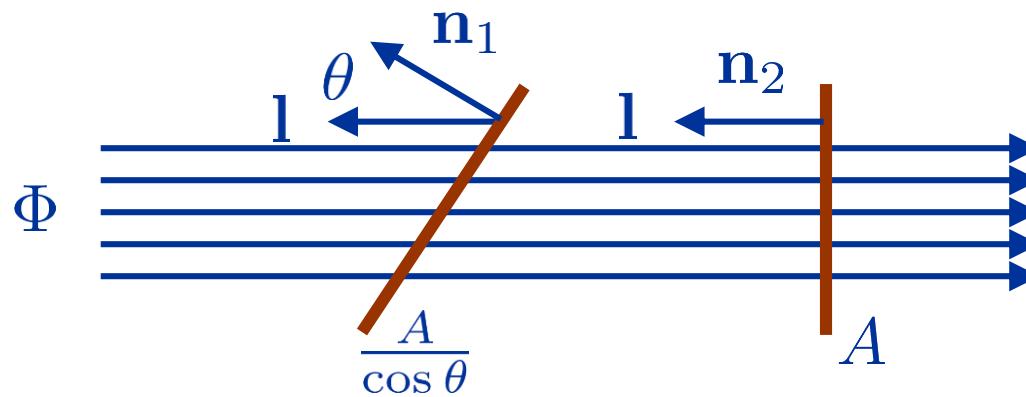
- irradiance on an oriented surface patch above a Lambertian surface is constant



$$E = \frac{\Phi}{dA} = \frac{\Phi \cos \theta}{dA \cos \theta}$$

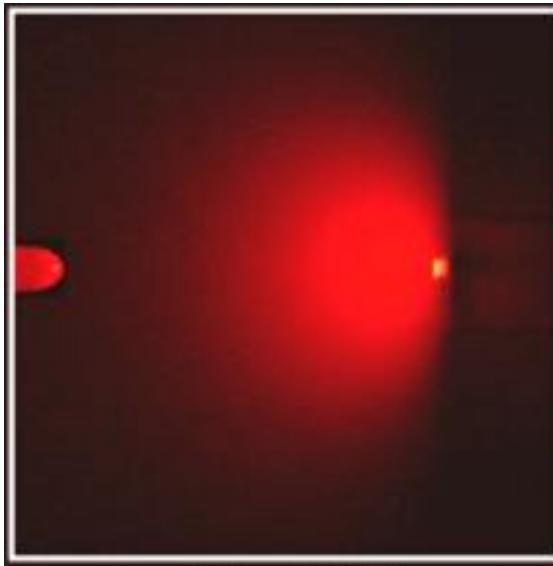
Lambert's Cosine Law

- irradiance on a surface is proportional to the cosine of the angle between surface normal n and light source direction l
(also referred to as Lambert's Cosine Law)

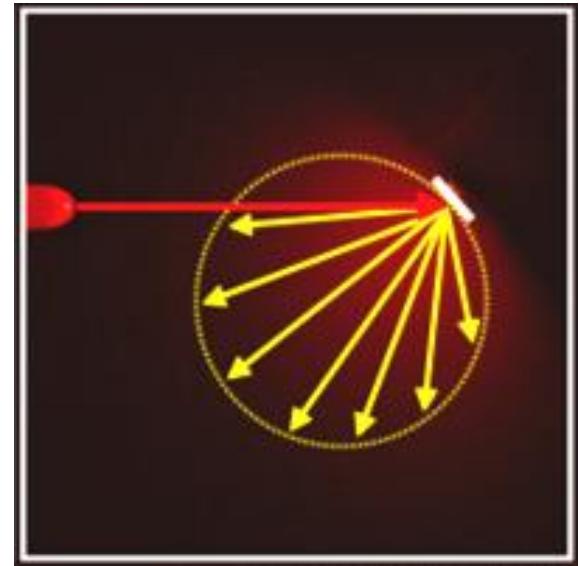


$$E = \frac{\Phi}{A/\cos \theta} = \frac{\Phi}{A} \cos \theta = E_L \cos \theta$$

Diffuse Lighting



Irradiance of the object surface (yellow) depends on the angle between light source (red) direction and surface normal.



Radiant intensity of the surface depends on the angle between radiation direction and surface normal.

[Wikipedia: Lambertsches Gesetz]

Diffuse Lighting

- radiance (RGB) is computed as $\mathbf{L}_{\text{diff}} = \frac{1}{\pi} \cdot \mathbf{c}_{\text{diff}} \otimes \mathbf{E}_L \cdot \cos \theta$
 - \otimes denotes component-wise multiplication,
 $(a, b, c) \otimes (d, e, f) = (a \cdot d, b \cdot e, c \cdot f)$
 - $1/\pi$ is a normalization coeff. motivated by energy conservation: light is reflected in all directions of a hemisphere and the viewer only receives a portion of the reflected light
- in implementations,
 - π is usually incorporated in \mathbf{E}_L
 - $\cos \theta$ is computed as the dot product of the normalized light direction and the normalized surface normal
$$\mathbf{L}_{\text{diff}} = \mathbf{c}_{\text{diff}} \otimes \mathbf{E}_L \cdot \max(0, \mathbf{n} \cdot \mathbf{l})$$
 - neg. values $n \cdot l$ correspond to illuminating the back of a surface, the max-function is often omitted for readability

Outline

- light
- color
- lighting models
 - diffuse reflection
 - specular reflection
 - ambient light
 - Phong illumination model
 - miscellaneous
- shading models

Specular Reflection

- perceived radiance depends on the viewing direction
 - maximum radiance, if the viewer direction v corresponds to the reflection direction r or, if the halfway vector h corresponds to the surface normal

- Phong model

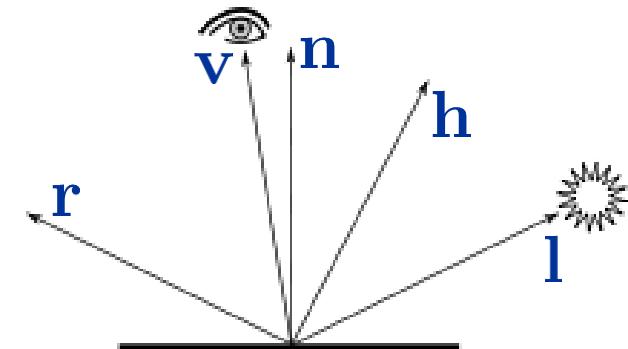
$$\mathbf{L}_{\text{spec}} = \mathbf{c}_{\text{spec}} \otimes \mathbf{E}_L \cdot (\max(0, \mathbf{v} \cdot \mathbf{r}))^m$$

$\mathbf{c}_{\text{spec}} = (1,1,1)$ is popular for the specular color, since the color of specular reflection converges to the color of the incident light.
The exponent m characterizes the size of the specular highlight.
Maximum radiance is not influenced.

- Blinn-Phong model

$$\mathbf{L}_{\text{spec}} = \mathbf{c}_{\text{spec}} \otimes \mathbf{E}_L \cdot (\max(0, \mathbf{n} \cdot \mathbf{h}))^m$$

Standard specular term in OpenGL (prior to OpenGL 3.1)

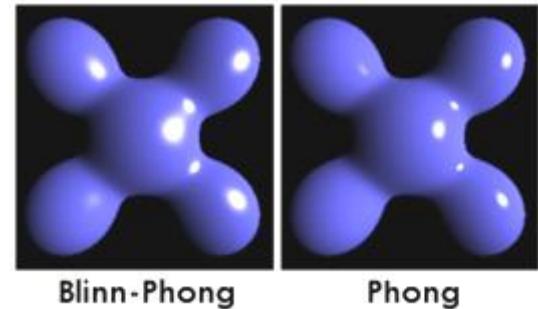


All these vectors are normalized
in computations.

[Wikipedia:
Blinn-Phong shading model]

Specular Reflection Implementation

- Phong model
 - requires the reflection vector r which can be computed from n and l
$$r = 2(n \cdot l)n - l$$
- Blinn-Phong model
 - requires the halfway vector h which can be computed from l and v
$$h = \frac{l+v}{\|l+v\|}$$



[Wikipedia:
Blinn-Phong shading model]

Reflection Vector

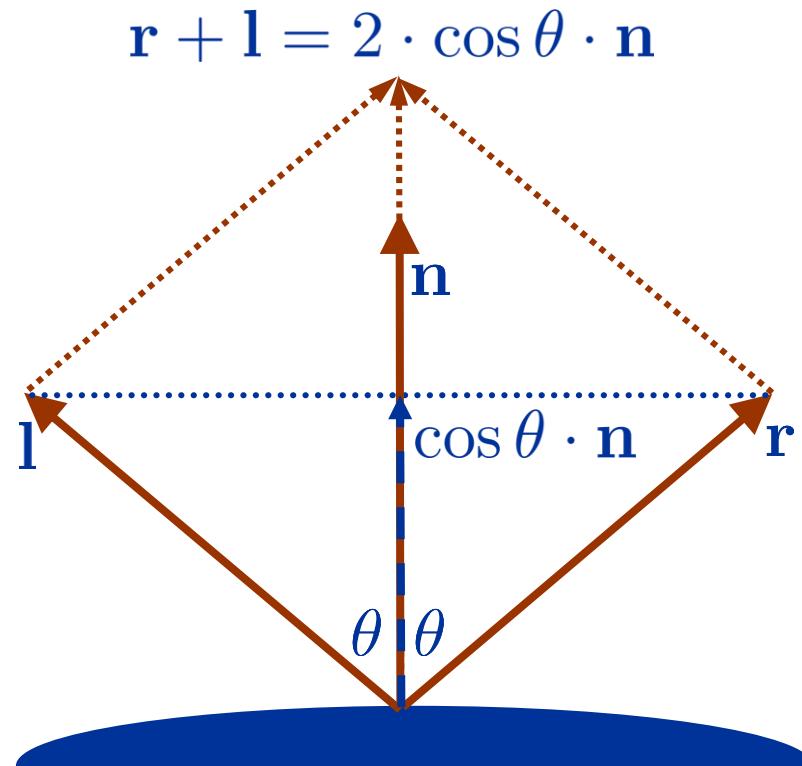
- computed with light source direction \mathbf{l} and surface normal \mathbf{n}

$$\mathbf{r} + \mathbf{l} = 2 \cdot \cos \theta \cdot \mathbf{n}$$

$$\cos \theta = \mathbf{l} \cdot \mathbf{n}$$

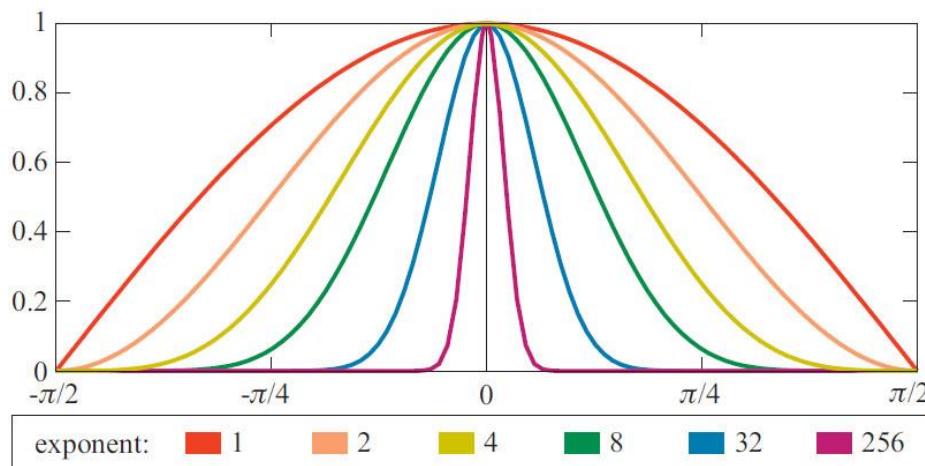
$$\mathbf{r} = 2 \cdot (\mathbf{l} \cdot \mathbf{n}) \cdot \mathbf{n} - \mathbf{l}$$

- \mathbf{l} and \mathbf{n} have to be normalized
- \mathbf{r} is normalized



Non-normalized Blinn-Phong

- Phong and Blinn-Phong do not account for energy preservation
- radiance depends on angle θ and exponent m
- radiant exitance from the surface depends on m



angle θ between v and r (Phong)
or n and h (Blinn-Phong)

[Akenine-Möller et al.]

Normalized Blinn-Phong

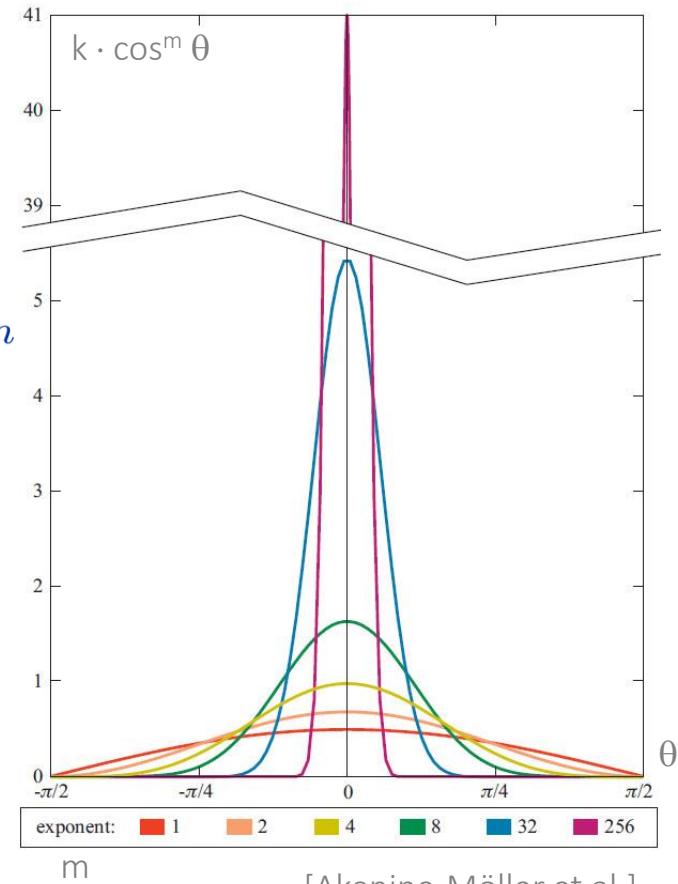
- normalized Blinn-Phong versions account for energy conservation

- e.g.

$$\mathbf{L}_{\text{spec}} = \frac{m+8}{8\pi} \cdot \mathbf{c}_{\text{spec}} \otimes \mathbf{E}_L \cdot (\mathbf{n} \cdot \mathbf{l}) \cdot (\mathbf{n} \cdot \mathbf{h})^m$$

[Akenine-Möller et al., Real-time Rendering]

- considers angle-dependent irradiance
- considers growing maximum radiance for growing exponent m
- radiant exitance is approximately constant for varying m



[Akenine-Möller et al.]

Outline

- light
- color
- lighting models
 - diffuse reflection
 - specular reflection
 - ambient light
 - Phong illumination model
 - miscellaneous
- shading models

Ambient Light

- diffuse and specular reflection are modeled for directional light from a point light source or from parallel light (point light source at infinity)
- ambient light is an approx. for indirect light sources
 - reflected light from objects illuminates other objects
 - all objects in the hemisphere seen from a surface act as light source for this surface
- this effect is generally approximated by a constant, object-dependant offset $\mathbf{L}_{\text{amb}} = \mathbf{c}_{\text{amb}} \otimes \mathbf{E}_{\text{amb}}$
- \mathbf{c}_{amb} usually corresponds to \mathbf{c}_{diff} , \mathbf{E}_{amb} represents ambient illumination, e.g. the dominant object color

Outline

- light
- color
- lighting models
 - diffuse reflection
 - specular reflection
 - ambient light
 - Phong illumination model
 - miscellaneous
- shading models

Phong Illumination Model

- combines ambient, diffuse, and specular components

$$\mathbf{L}_p = \mathbf{c}_{\text{amb}} \otimes \mathbf{E}_{\text{amb}} + \mathbf{c}_{\text{diff}} \otimes \mathbf{E}_{\text{diff}} \cdot (\mathbf{n} \cdot \mathbf{l}) + \mathbf{c}_{\text{spec}} \otimes \mathbf{E}_{\text{spec}} \cdot (\mathbf{r} \cdot \mathbf{v})^m$$

- Phong allows to set different light colors for different components (which is not physically motivated)
- a useful parameter setting could be

$$\mathbf{L}_p = \mathbf{E}_{\text{amb}} \otimes \mathbf{c}_{\text{obj}} + \mathbf{E}_{\text{light}} \otimes (\mathbf{c}_{\text{obj}} \cdot (\mathbf{n} \cdot \mathbf{l}) + \mathbf{c}_{\text{spec}} \cdot (\mathbf{r} \cdot \mathbf{v})^m)$$

ambient
illumination object color highlight color
 (e. g. white)

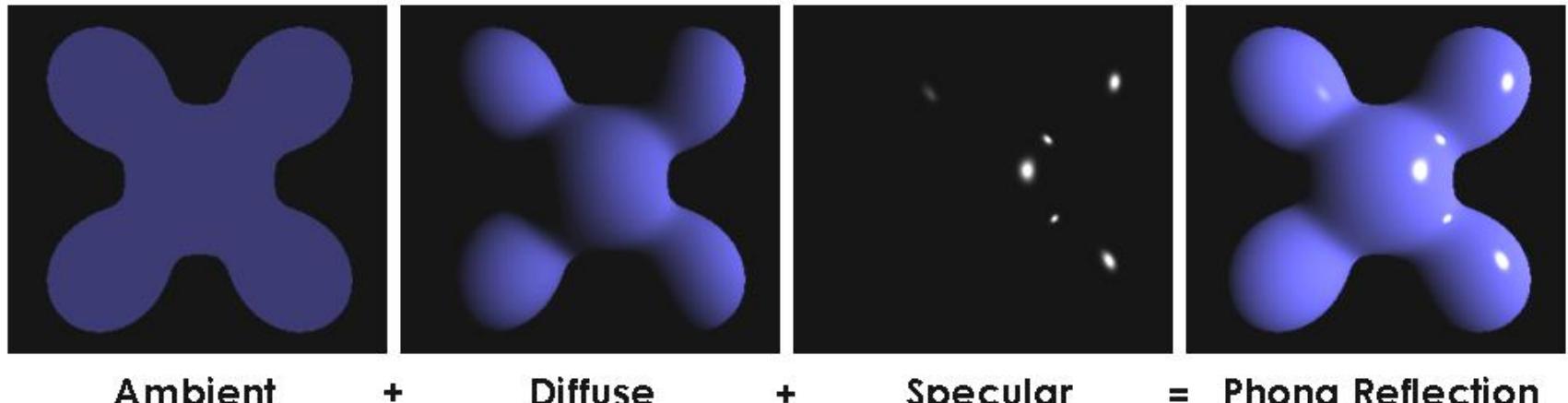
- multiple light sources, e. g.

$$\mathbf{L}_p = \mathbf{c}_{\text{amb}} \otimes \mathbf{E}_{\text{amb}} + \sum_{\text{light}} \mathbf{E}_{\text{light}} \otimes (\mathbf{c}_{\text{diff}} \cdot (\mathbf{n} \cdot \mathbf{l}) + \mathbf{c}_{\text{spec}} \cdot (\mathbf{r} \cdot \mathbf{v})^m)$$

- max-functions are omitted. $\mathbf{n}, \mathbf{l}, \mathbf{r}, \mathbf{v}$ are normalized.

Max values for RGB comp. of \mathbf{L}_p have to be considered.

Phong Illumination Model

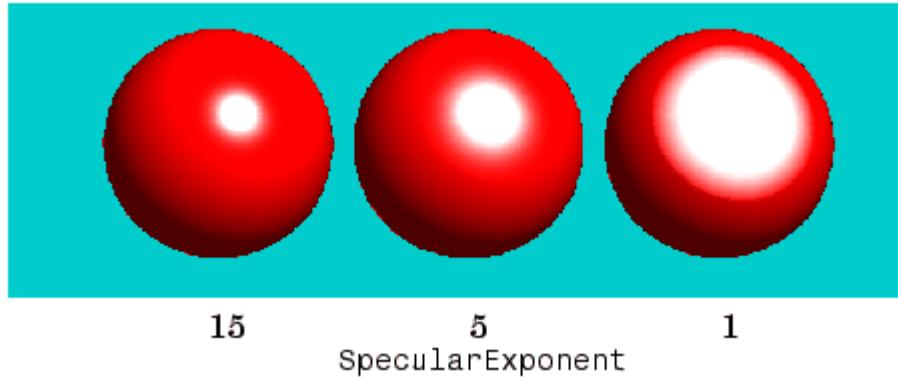
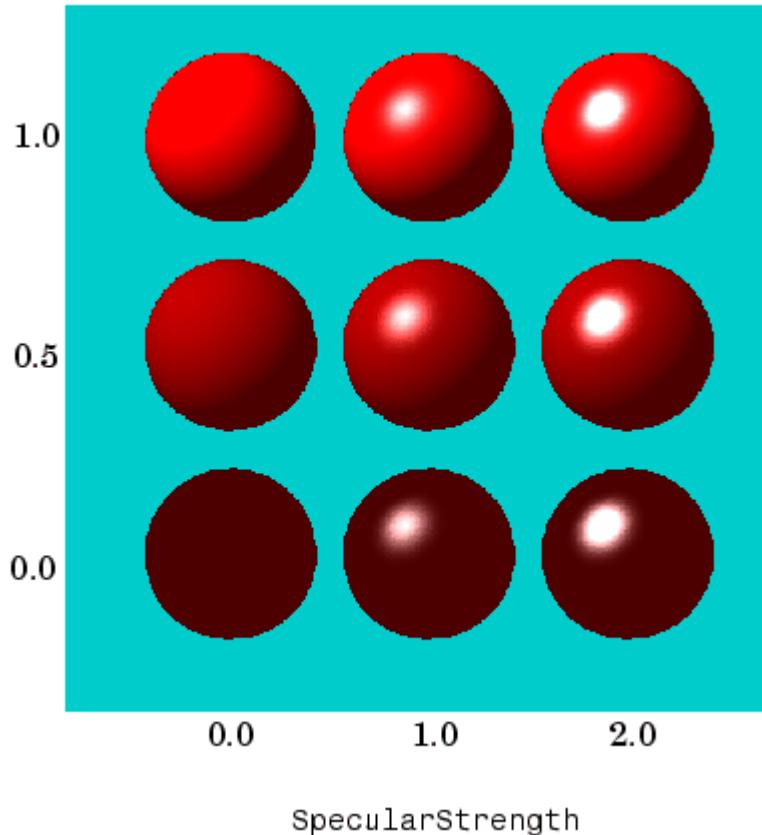


- parameters (material and light colors) can be used to adapt the ratios of ambient, diffuse, and specular reflection components

[Wikipedia: Phong shading]

Phong Illumination Model

DiffuseStrength



[<http://www.mathworks.com/help/techdoc/visualize/f1-21818.html>]

Blinn-Phong Illumination Model

- non-normalized Blinn-Phong

$$\mathbf{L}_{\text{bp}} = \mathbf{c}_{\text{amb}} \otimes \mathbf{E}_{\text{amb}} + \mathbf{c}_{\text{diff}} \otimes \mathbf{E}_{\text{diff}} \cdot (\mathbf{n} \cdot \mathbf{l}) + \mathbf{c}_{\text{spec}} \otimes \mathbf{E}_{\text{spec}} \cdot (\mathbf{h} \cdot \mathbf{n})^m$$

- parameter setting and multiple light sources
(see Phong)

- normalized Blinn-Phong, e.g.

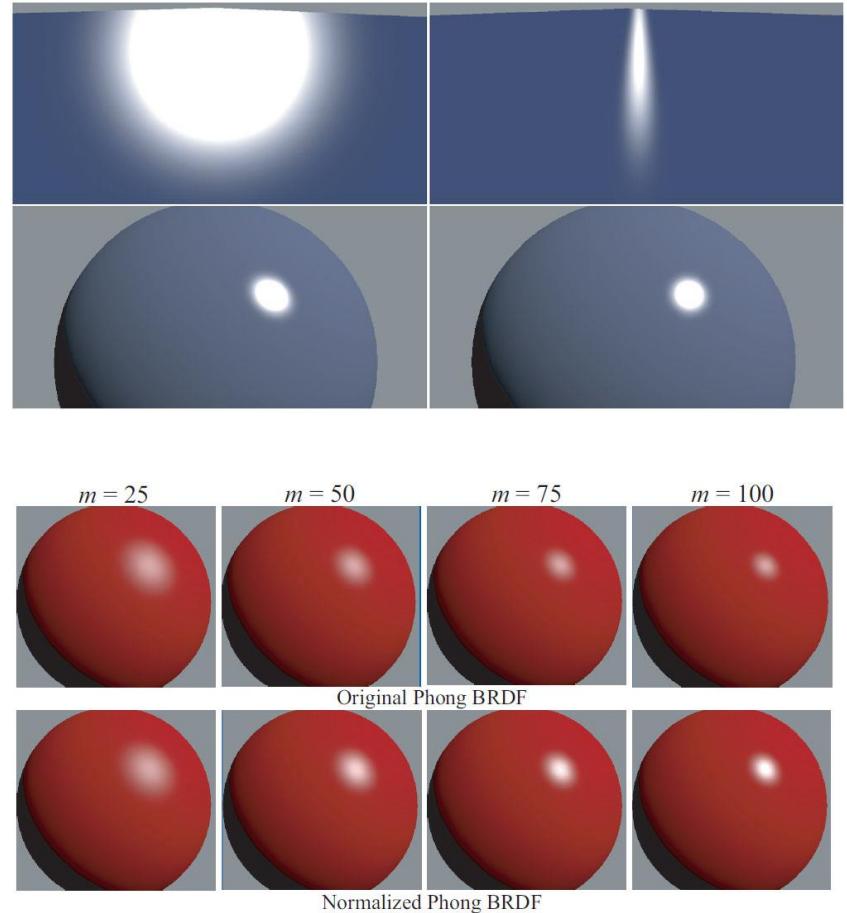
$$\mathbf{L}_{\text{nbp}} = \mathbf{E}_{\text{amb}} \otimes \mathbf{c}_{\text{amb}} + \left(\frac{1}{\pi} \mathbf{c}_{\text{diff}} + \frac{m+8}{8\pi} \cdot \mathbf{c}_{\text{spec}} \cdot (\mathbf{h} \cdot \mathbf{n})^m \right) \otimes \mathbf{E}_L \cdot (\mathbf{l} \cdot \mathbf{n})$$

- in implementations, the normalization coefficient can be incorporated into \mathbf{c}_{diff} and \mathbf{c}_{spec}

$$\mathbf{L}_{\text{nbp}} = \mathbf{E}_{\text{amb}} \otimes \mathbf{c}_{\text{amb}} + (\mathbf{c}_{\text{diff}} + \mathbf{c}_{\text{spec}} \cdot (\mathbf{h} \cdot \mathbf{n})^m) \otimes \mathbf{E}_L \cdot (\mathbf{l} \cdot \mathbf{n})$$

Phong vs. Blinn-Phong

- Phong (left) and Blinn-Phong (right)
- highlights on flat surfaces are more realistic with Blinn-Phong
- Blinn-Phong (top) and normalized Blinn-Phong (bottom)
- maximum radiance of the highlight for varying m is constant for Blinn-Phong



[Akenine-Möller et al.]

Outline

- light
- color
- lighting models
 - diffuse reflection
 - specular reflection
 - ambient light
 - Phong illumination model
 - miscellaneous
- shading models

Considering Distances

- between object surface and light source
 - irradiance of a surface illuminated by a point light source is inversely proportional to the squared distance from the surface to the light source
 - light source attenuation
- between object surface and viewer
 - atmospheric effects, e. g. fog, influence the visibility of objects
 - visibility refers to the transparency of air
 - if air is transparent, objects are clearly visible
 - in less transparent air, fog particles absorb some flux and scatter additional flux towards the viewer
 - in low visibility, radiance converges to a "fog color"

Light Source Attenuation

- for a point light source with position $\mathbf{l} = (l_x, l_y, l_z, 1)^T$, the distance d from the light source to a surface point $\mathbf{p} = (p_x, p_y, p_z, 1)^T$ can be considered in the irradiance of the light source

$$d = \|\mathbf{p} - \mathbf{l}\|$$

$$\mathbf{E}_{\text{att}} = \frac{1}{k_c + k_l \cdot d + k_q \cdot d^2} \mathbf{I}_{\text{light}}$$

- this is motivated by $\mathbf{E}_{\text{light}} = \frac{\mathbf{I}_{\text{light}}}{d^2}$
(irradiance \mathbf{E} is inversely proportional to the squared distance from the light source with radiant intensity \mathbf{I})
- k_c, k_l, k_q are user-defined parameters

Fog

- fog is approximated by a linear combination of the computed radiance \mathbf{L} and a fog color \mathbf{c}_{fog}
- d is the distance of the surface point to the viewer (its z-component, i.e. depth value)
$$\mathbf{L}_{\text{fog}} = f(d) \cdot \mathbf{L} + (1 - f(d)) \cdot \mathbf{c}_{\text{fog}}$$
- $0 \leq f(d) \leq 1$ is a function that describes the visibility
 - $f(d) = 1$: max visibility (object color is unaffected)
 - $f(d) = 0$: min visibility (object color is changed to fog color)
 - e. g. $f(d) = \frac{d_{\text{end}} - d}{d_{\text{end}} - d_{\text{start}}} \quad f(d) = e^{-\text{density} \cdot d}$

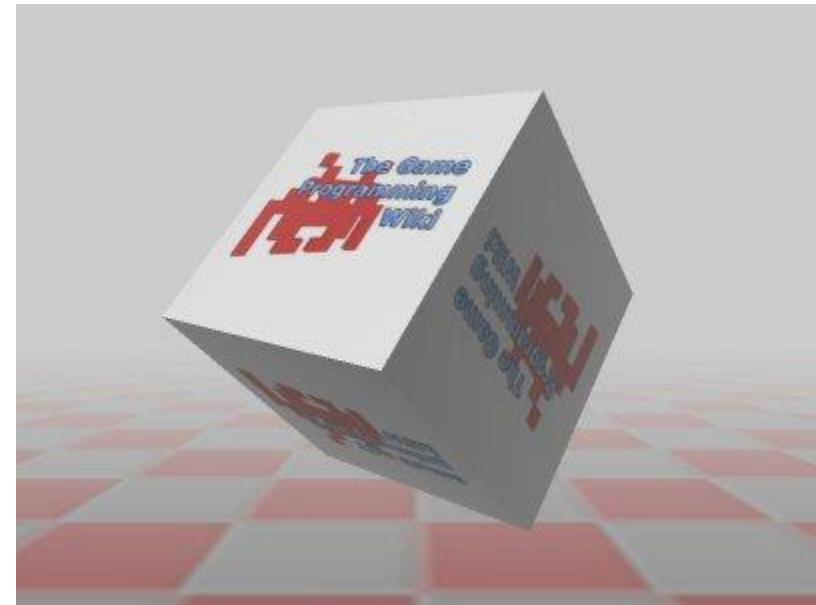
linear fog: starts at d_{start} ,
minimum visibility at d_{end} .
Clamped to [0..1].

exponential fog

Attenuation and Fog



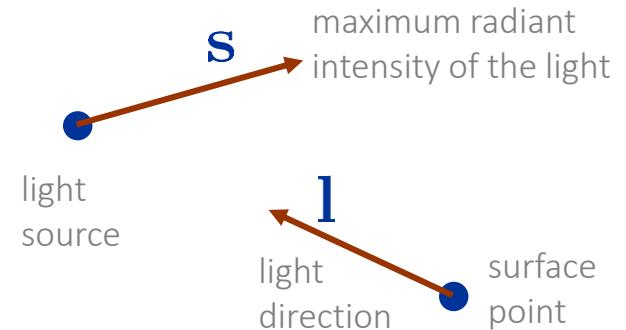
[<http://www.gamedev.net/topic/541383-typical-light-attenuation-coefficients/>]



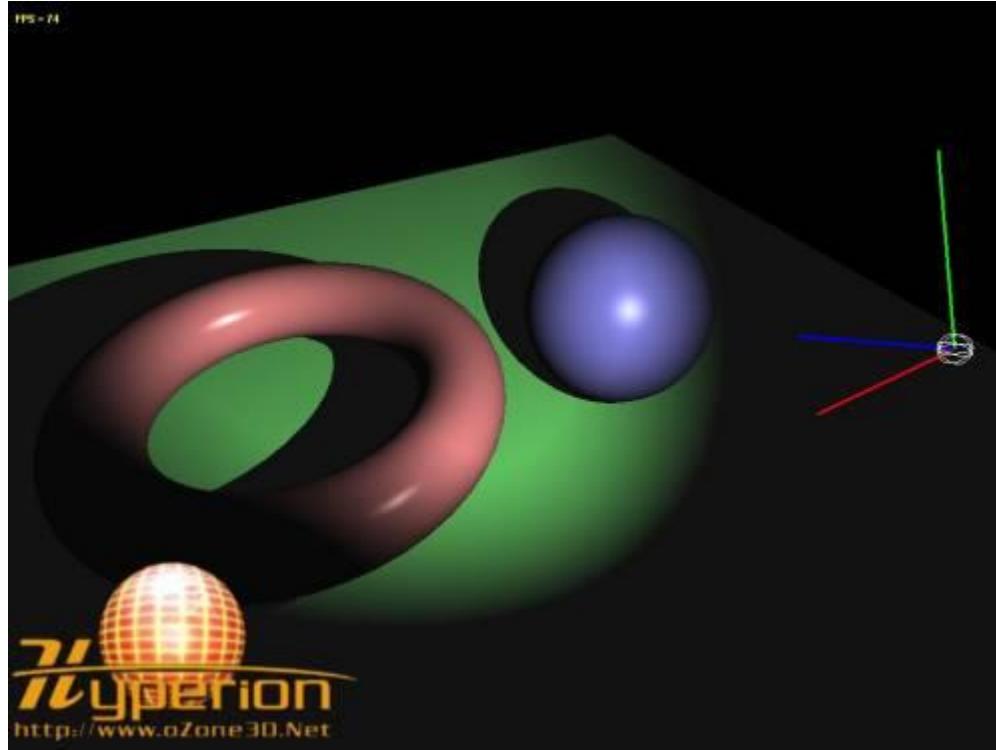
[The Game Programming Wiki:
OpenGL:Tutorials:Tutorial
Framework:Light and Fog]

Light Sources

- positional light source at position $\mathbf{l} = (l_x, l_y, l_z, 1)^T$
- directional light source with direction $\mathbf{l} = (l_x, l_y, l_z, 0)^T$
- spotlight
 - positional light with flux into restricted directions
 - e. g. $\mathbf{E}_{\text{spot}} = I_{\text{light}} \cdot \max(-\mathbf{l} \cdot \mathbf{s}, 0)^{m_{\text{spot}}}$
 - I_{light} is the maximum radiant intensity of the spotlight in direction \mathbf{s}
 - \mathbf{l} is the direction to the light
 - m_{spot} is user-defined to adapt the fall-off rate of the radiant intensity with respect to the cosine of the angle between $-\mathbf{l}$ and \mathbf{s}
 - (I_{light} should be divided by some squared distance)



Spotlight



[http://www.ozone3d.net/tutorials/glsl_lighting_phong_p3.php]

Summary

- the class of Phong lighting models considers
 - diffuse and specular reflection
 - ambient illumination
- the lighting models are efficient to compute as they only consider local information, e.g. surface normal, light source direction, viewer direction, ...
- improved variants consider energy conservation and lead to more realistic specular highlights
- additionally, distances to the viewer and the light source can be considered
- various types of light sources can be realized

Outline

- light
- color
- lighting models
- shading models

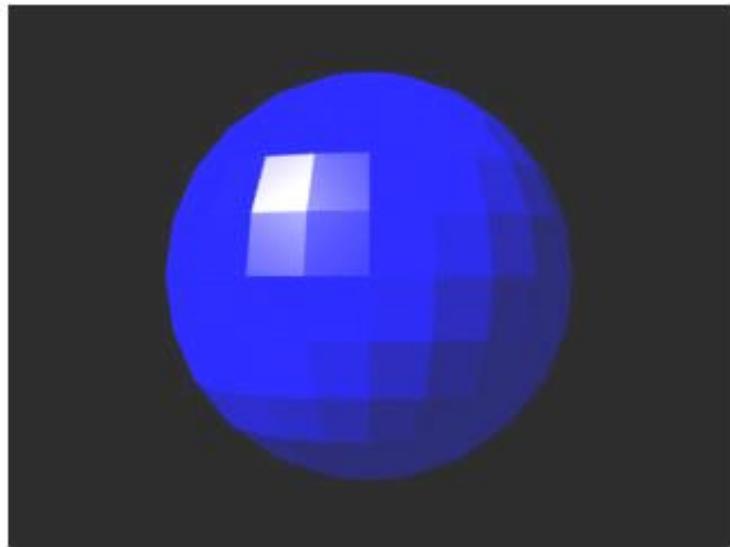
Introduction

- shading models specify whether the lighting model is evaluated **per vertex** or **per fragment**
- if evaluated per vertex, the shading model specifies whether the resulting vertex **colors** are interpolated across a primitive **or not**
- if evaluated per fragment, surface **normals** are interpolated across a primitive

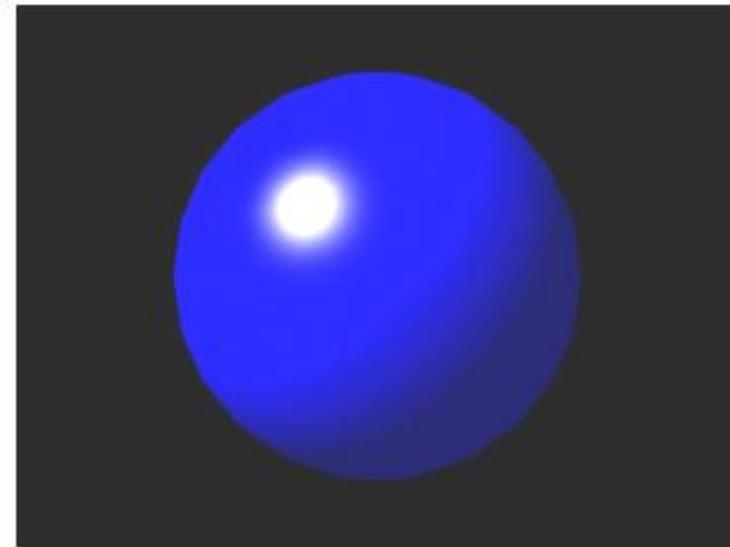
Shading Models

- flat shading (constant shading)
 - evaluation of the lighting model per vertex
 - primitives are colored with the color of one specific vertex
- Gouraud shading
 - evaluation of the lighting model per vertex
 - primitives are colored by bilinear interpolation of vertex colors
- Phong shading
 - bilinear interpolation of vertex normals during rasterization
 - evaluation of the lighting model per fragment

Flat vs. Phong



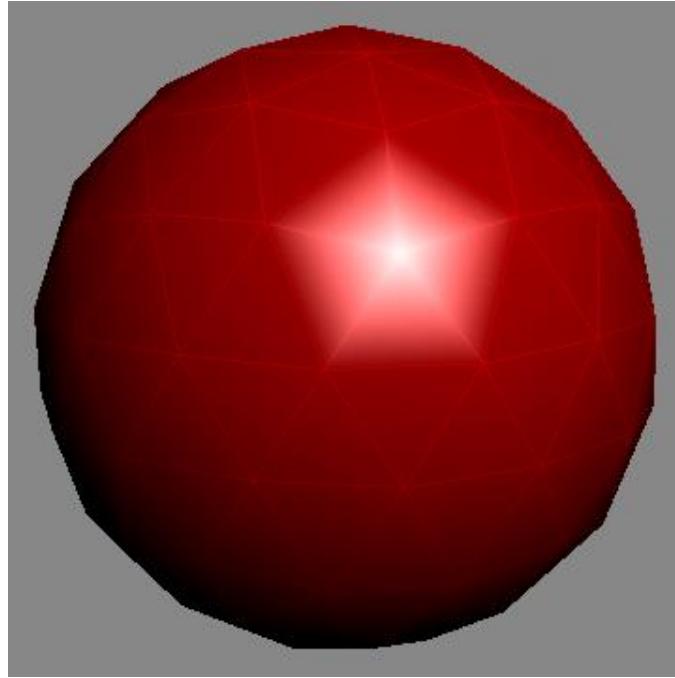
FLAT SHADING



PHONG SHADING

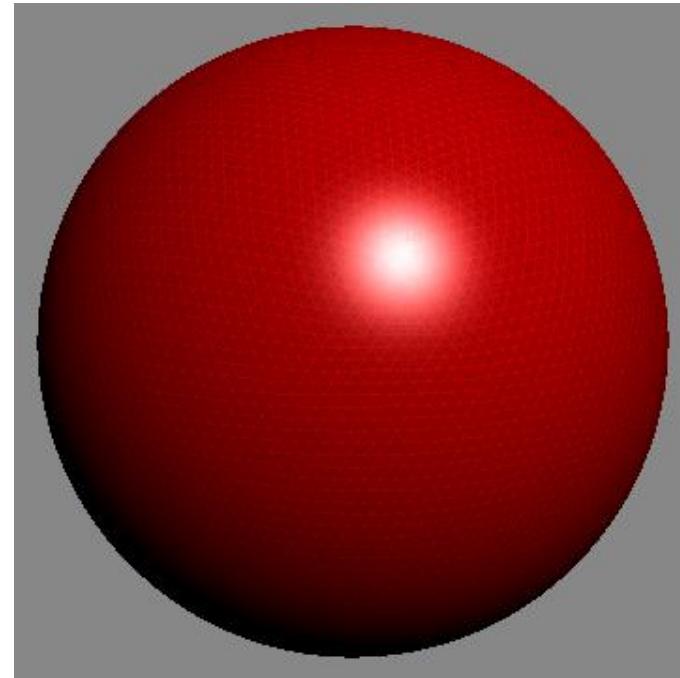
[Wikipedia: Phong shading]

Gouraud Shading



low polygon count

Highlight is poorly resolved.
Mach band effect.

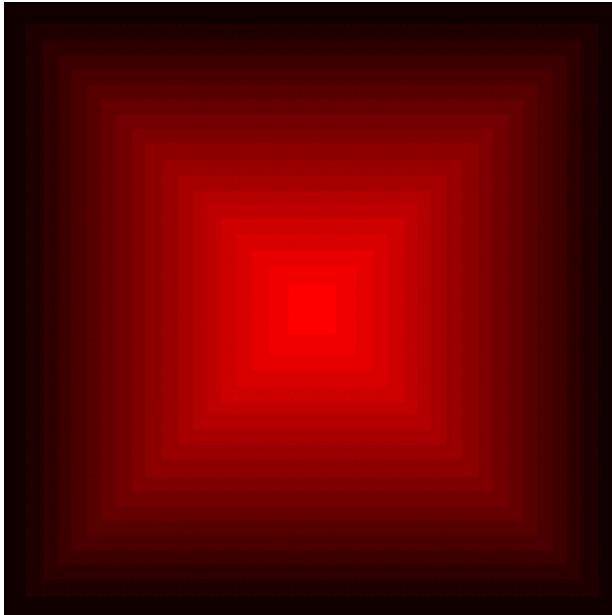


high polygon count

[Wikipedia: Gouraud shading]

Mach Band Effect

- mach bands are illusions due to our neural processing



The bright bands at 45 degrees and 135 degrees are illusory.

The intensity inside each square is the same.

Shading Models

- flat shading (constant shading)
 - simplest, fastest
- Gouraud shading
 - more realistic than flat shading for the same tessellation
 - suffers from Mach band effect
 - local highlights are not resolved, if the highlight is not captured by a vertex
- Phong shading
 - highest quality, most expensive

Summary

- light
- color
- lighting models
- shading models

Image Processing and Computer Graphics

Rasterization

Matthias Teschner

Computer Science Department
University of Freiburg

Albert-Ludwigs-Universität Freiburg

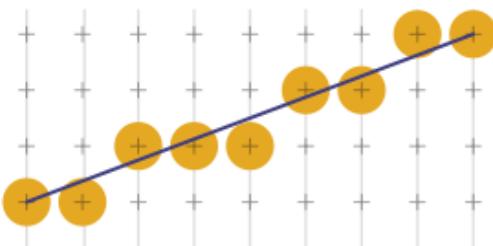


Motivation

- rasterization is
 - the transformation of geometric primitives (line segments, circles, polygons) into a raster image representation, i.e. pixel positions
 - the estimation of an appropriate set of pixel positions to represent a geometric primitive
- the rendering pipeline
 - processes vertices (transformations and lighting)
 - assembles primitives from vertices in window space and topology information
 - rasterizes primitives, i.e. converts primitives to fragments with interpolated attributes
 - processes fragments, updates the framebuffer

Motivation

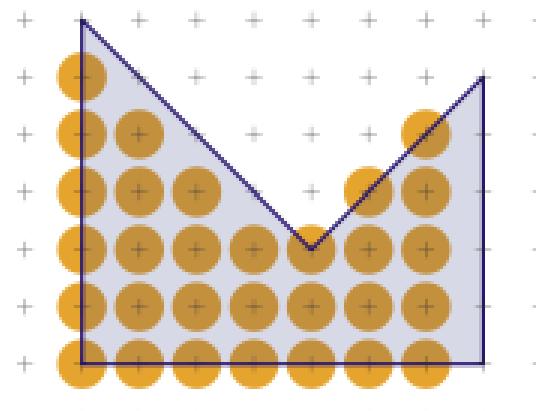
- computation of pixel positions that represent a primitive



Line (segment) rasterization



Circle rasterization



Polygon rasterization

[Wikipedia: Rasterung von Linien, Rasterung von Polygonen, Rasterung von Kreisen]

Outline

- lines
- circles
- polygons

General Setting

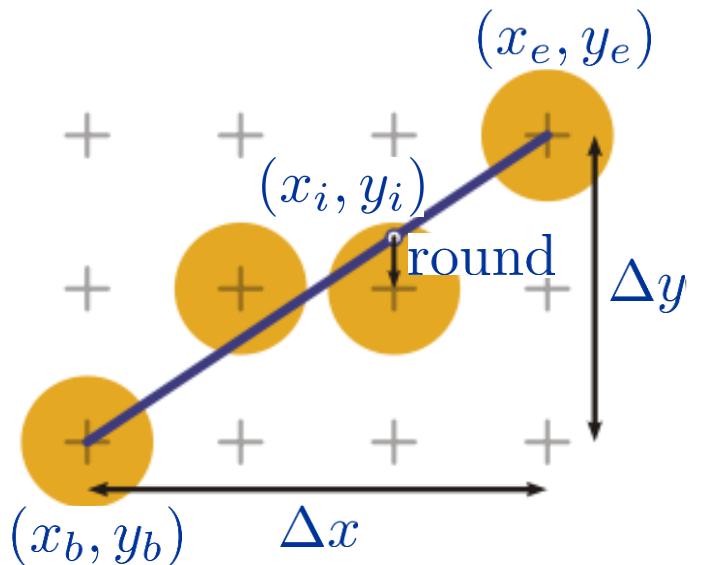
- components of start and end point of a line
are integer values $\mathbf{p}_b = (x_b, y_b)$ $\mathbf{p}_e = (x_e, y_e)$
- lines are represented as
 $y = mx + b$ or $F(x, y) = ax + by + c = 0$
- algorithms are often restricted to $0 \leq m \leq 1$
 - arbitrary lines are handled by employing symmetries
- algorithms consist of an initialization step and a loop
 - efficiency of a particular algorithm
depends on the line length

A Simple Algorithm

- $y = \frac{\Delta y}{\Delta x}(x - x_b) + y_b = \frac{y_e - y_b}{x_e - x_b}(x - x_b) + y_b$

- for each $x_b \leq x_i \leq x_e$
compute $y_i = \text{round}(y(x_i))$
set $\mathbf{p}_i = (x_i, y_i)$
- efficient incremental update

$$y(x_{i+1}) - y(x_i) = m(x_{i+1} - x_i) + y_b - (m(x_i - x_b) + y_b) = m(x_{i+1} - x_i) = m$$
$$y(x_{i+1}) = y(x_i) + m$$



[Wikipedia: Rasterung von Linien]

Generalization

- $-1 \leq m \leq 1$
 - $x_b < x_e$: increment x_i , compute $(x_i, \text{round}(y(x_i)))$
 - $x_e < x_b$: decrement x_i , compute $(x_i, \text{round}(y(x_i)))$
- $m > 1$ or $m < -1$
 - $y_b < y_e$: increment y_i , compute $(\text{round}(x(y_i)), y_i)$
 - $y_e < y_b$: decrement y_i , compute $(\text{round}(x(y_i)), y_i)$

Bresenham Algorithm (Midpoint Algorithm)

- explicit form of a line

$$y = \frac{y_e - y_b}{x_e - x_b} (x - x_b) + y_b$$

- implicit form of a line

$$0 = \frac{\Delta y}{\Delta x}x - y + y_b - \frac{\Delta y}{\Delta x}x_b = \Delta y \cdot x - \Delta x \cdot y + \Delta x \cdot y_b - \Delta y \cdot x_b$$

- implicit form of a line

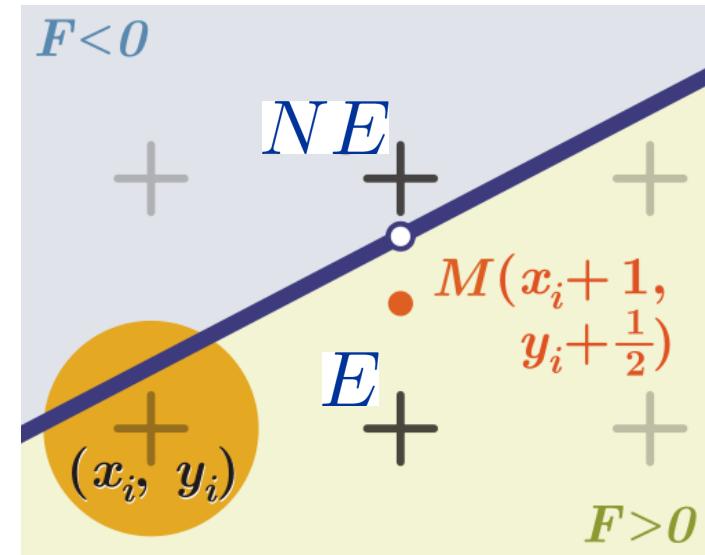
- for all points (x, y) on a line

$$F(x, y) = \Delta y \cdot x - \Delta x \cdot y + \Delta x \cdot y_b - \Delta y \cdot x_b = 0$$

- all points with $F(x, y) > 0$ are on one side of the line
 - all points with $F(x, y) < 0$ are on the other side

Bresenham Algorithm

- for incremented values of x, the algorithm decides whether to increment y or not
- based on the current pixel (x_i, y_i) , the algorithm decides whether to choose $(x_i + 1, y_i)$ or $(x_i + 1, y_i + 1)$ (E east, NE north east)
- F is evaluated at the next midpoint $F(x_i + 1, y_i + \frac{1}{2})$
- $F(x_i + 1, y_i + \frac{1}{2}) > 0 \Rightarrow$ choose NE
- $F(x_i + 1, y_i + \frac{1}{2}) \leq 0 \Rightarrow$ choose E



[Wikipedia: Rasterung von Linien]

Incremental Update of the Decision Variable

- decision variable $d_i = F(x_i + 1, y_i + \frac{1}{2})$
- incremental update from d_i to d_{i+1} depending on d_i
- $d_i > 0 \Rightarrow$ choose NE, $d_{i+1} = F(x_i + 2, y_i + 1 + \frac{1}{2})$
- $d_i \leq 0 \Rightarrow$ choose E, $d_{i+1} = F(x_i + 2, y_i + \frac{1}{2})$
- in case of $d_i > 0$:

$$\Delta_{NE} = d_{i+1} - d_i = \Delta y \cdot (x_i + 2) - \Delta x \cdot (y_i + \frac{3}{2}) + c - (\Delta y \cdot (x_i + 1) - \Delta x \cdot (y_i + \frac{1}{2}) + c)$$

$$\Delta_{NE} = \Delta y - \Delta x$$

- in case of $d_i \leq 0$:

$$\Delta_E = d_{i+1} - d_i = \Delta y \cdot (x_i + 2) - \Delta x \cdot (y_i + \frac{1}{2}) + c - (\Delta y \cdot (x_i + 1) - \Delta x \cdot (y_i + \frac{1}{2}) + c)$$

$$\Delta_E = \Delta y$$

Bresenham Algorithm

Initialization

- for start point $\mathbf{p}_b = (x_b, y_b)$,
decision variable d_1 can be initialized as

$$\begin{aligned}d_1 &= F(x_b + 1, y_b + \frac{1}{2}) = \Delta y \cdot (x_b + 1) - \Delta x \cdot (y_b + \frac{1}{2}) + c \\&= \Delta y \cdot x_b - \Delta x \cdot y_b + c + \Delta y - \frac{1}{2}\Delta x \\&= F(x_b, y_b) + \Delta y - \frac{1}{2}\Delta x \\&= \Delta y - \frac{1}{2}\Delta x\end{aligned}$$

- floating-point arithmetic is avoided by
considering $2 \cdot F(x, y)$: $d_1 = 2\Delta y - \Delta x$

$$\Delta_E = 2\Delta y$$

$$\Delta_{NE} = 2\Delta y - 2\Delta x$$

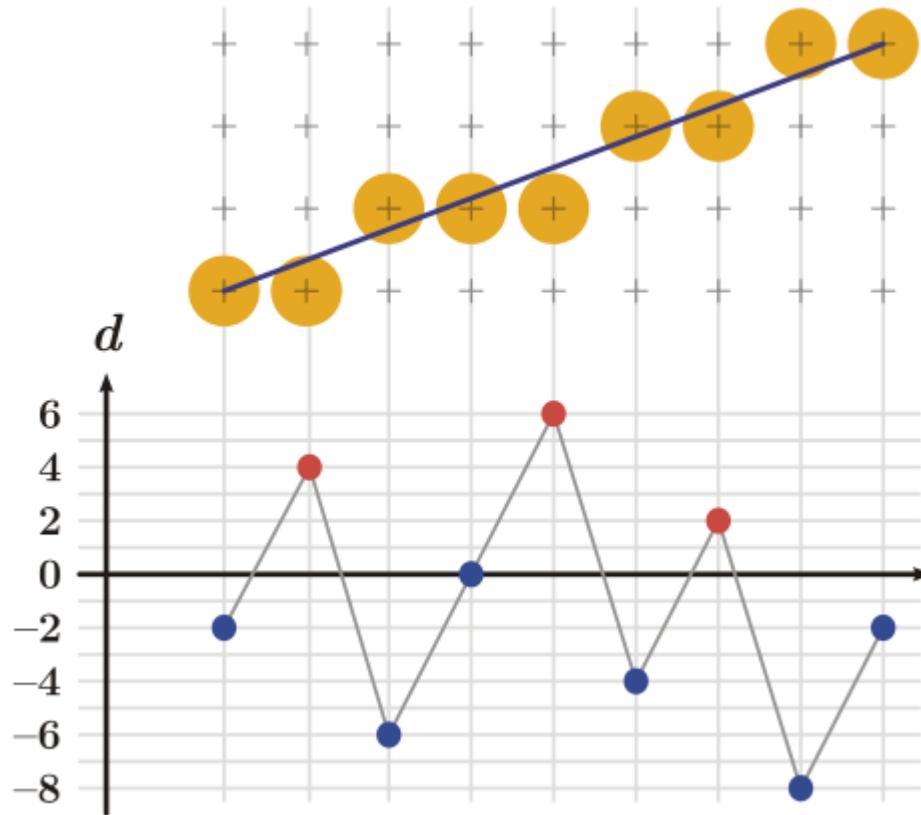
Bresenham Algorithm

Implementation

```
void BresenhamLine(int xb, int yb, int xe, int ye) {  
  
    int dx, dy, incE, incNE, d, x, y;  
  
    dx = xe - xb; dy = ye - yb;  
    d = 2*dy - dx;  
    incE = 2*dy;  
    incNE = 2*(dy - dx);  
    x = xb; y = yb;  
    WritePixel(x, y); /* write start pixel */  
    while (x < xe) {  
        x++;  
        if (d <= 0) /* choose E */  
            d += incE;  
        else {  
            d += incNE; /* choose NE */  
            y++;  
        }  
        WritePixel(x, y);  
    }  
}
```

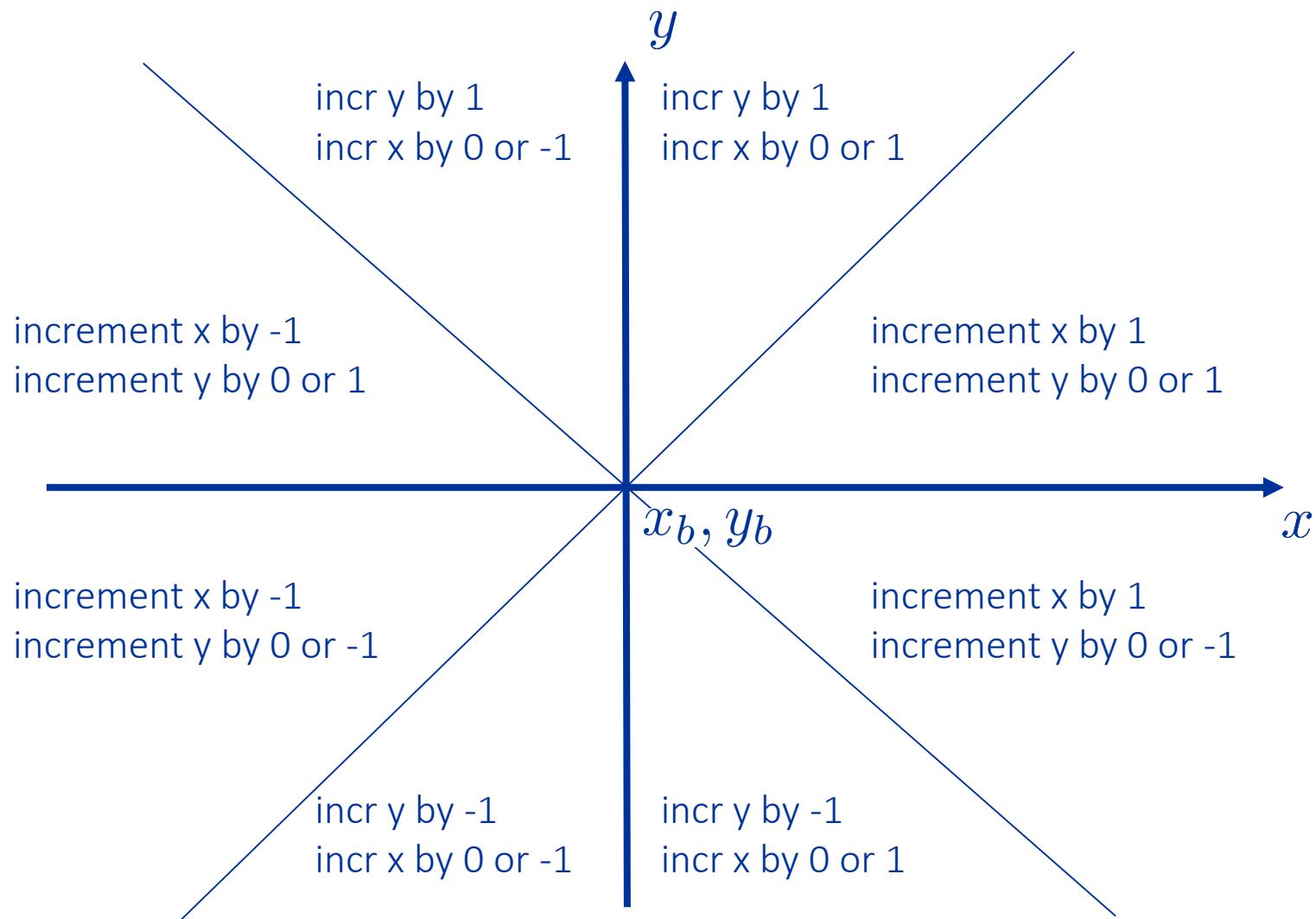
Bresenham Algorithm

Decision Variable



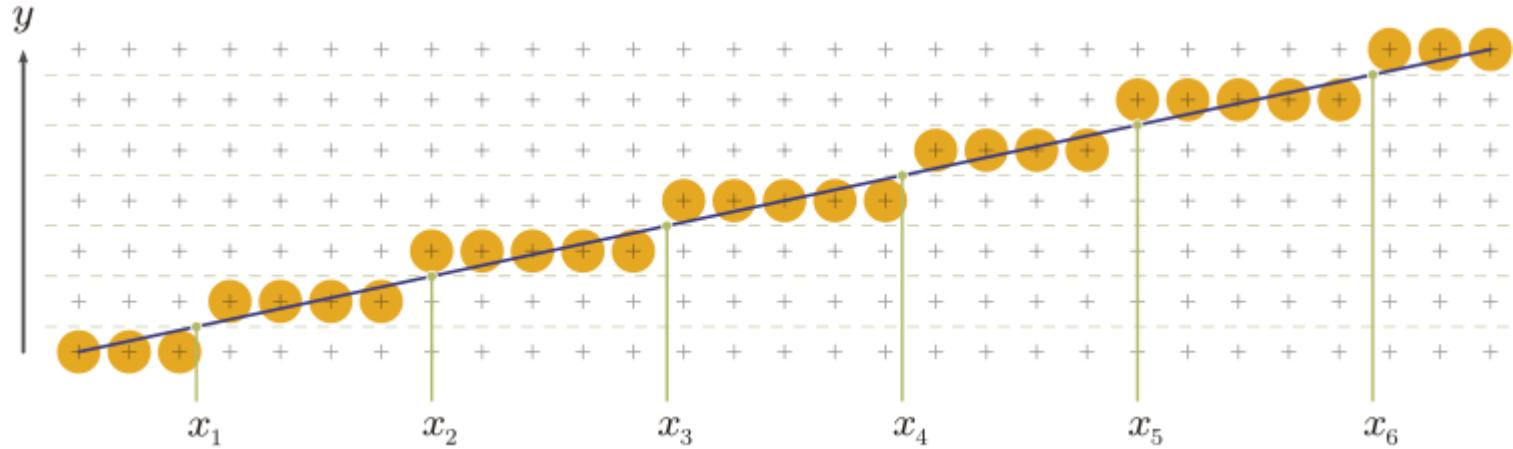
[Wikipedia: Rasterung von Linien]

Generalization



Run Length Slices

- estimate x-values where the y-value is incremented



- x_i is the (floating-point) intersection of the line with the line defined by $(x_b, y_b+i+0.5)$ and $(x_e, y_b+i+0.5)$
- increment y , compute x_i ,
draw pixels with the same y -value up to $\lfloor x_i \rfloor$

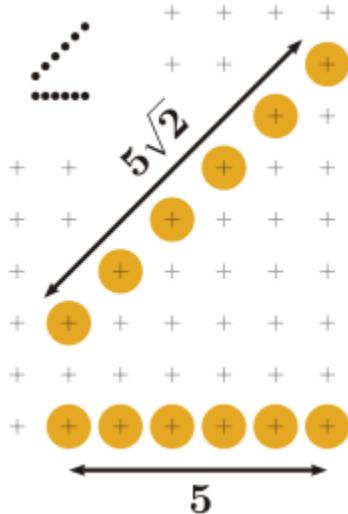
[Wikipedia: Rasterung von Linien]

Run Length Slices

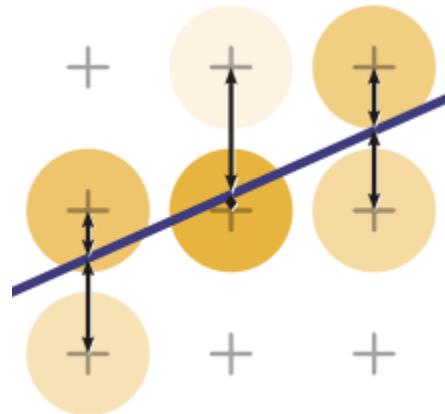
- line: $y = \frac{\Delta y}{\Delta x}(x - x_b) + y_b$
 $x = \frac{\Delta x}{\Delta y}(y - y_b) + x_b$
- x-components of the intersection at $y = y_b + i + \frac{1}{2}$:
 $x_i = \frac{\Delta x}{\Delta y}(y_b + i + \frac{1}{2} - y_b) + x_b$
- differential update using $x_{i+1} - x_i = \frac{\Delta x}{\Delta y}$
- initialization: $x_1 = \frac{3\Delta x}{2\Delta y} + x_b$
- loop: $x_{i+1} = x_i + \frac{\Delta x}{\Delta y}$

Issues / Limitations

- aliasing
 - stair-case artifacts, varying line intensity
- clipping
 - artifacts due to round-off of clipped end points

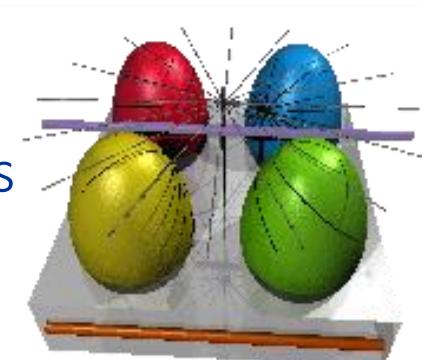


same number of pixels for lines with different length

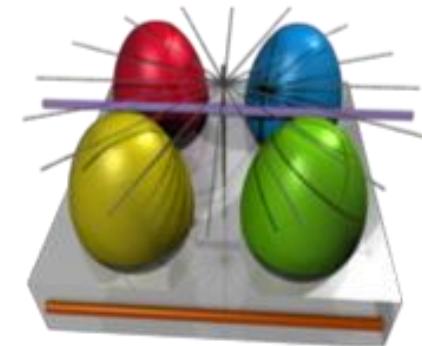


aliasing can be addressed by rendering thick lines with varying pixel intensities

no anti-aliasing

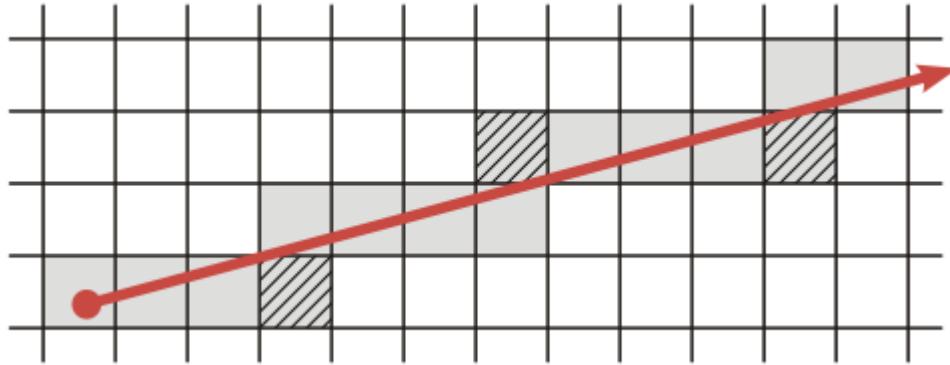


with anti-aliasing



Summary - Lines

- line rasterization algorithms are usually described for a subset of lines and generalized using symmetries
- incremental updates are often employed
- Bresenham avoids floating-point arithmetic
- improved algorithms address aliasing / clipping artifacts
- note that the algorithms do not compute all pixels that are intersected by the line



[Wikipedia: Rasterung von Linien]

Outline

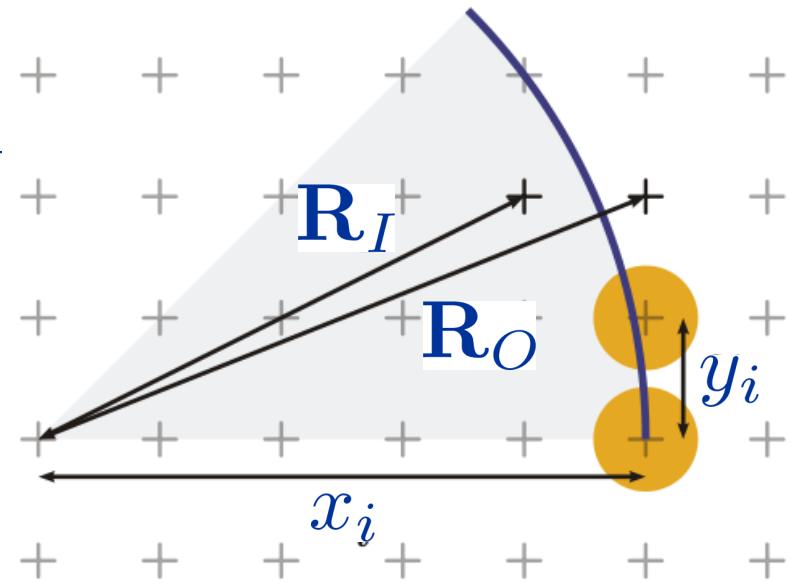
- lines
- circles
- polygons

General Setting

- circle with center at (0,0) and radius r
- implicit representation
$$F(x, y) = x^2 + y^2 - r^2 = 0$$
- algorithms compute only one eighth of a circle
 - if (x, y) is on the circle, then $(\pm x, \pm y)$ and $(\pm y, \pm x)$ are on the circle

Metzger Algorithm

- if (x_i, y_i) is on the circle, the algorithm decides whether $\mathbf{R}_O = (x_i, y_i + 1)$ or $\mathbf{R}_I = (x_i - 1, y_i + 1)$ is the next point on the circle
 - the point with the shortest distance to the circle is chosen
- $$d_I = r - \|\mathbf{R}_I\| = r - \sqrt{(x_i - 1)^2 + (y_i + 1)^2}$$
- $$d_O = \|\mathbf{R}_O\| - r = \sqrt{x_i^2 + (y_i + 1)^2} - r$$
- if $d_I \leq d_O \Rightarrow \mathbf{R}_I$
 - if $d_I > d_O \Rightarrow \mathbf{R}_O$

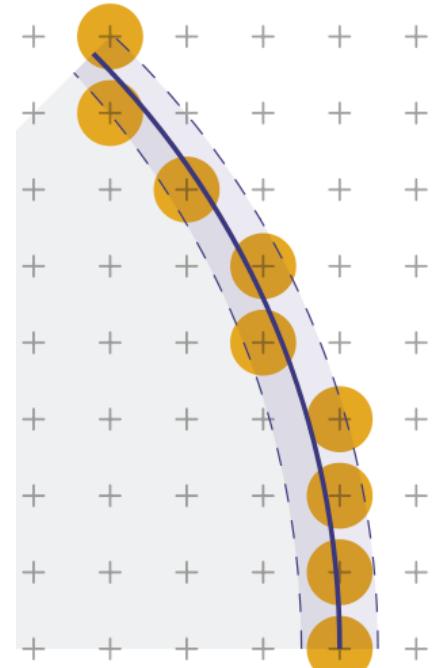


[Wikipedia: Rasterung von Kreisen]

Horn Algorithm

- the algorithm checks whether $(x_i - \frac{1}{2}, y_i + 1)$ is outside
 - if so, it chooses $(x_i - 1, y_i + 1)$
 - if not, it chooses $(x_i, y_i + 1)$
- decision variable
$$d_i = (x_i - \frac{1}{2})^2 + y_i^2 - r^2$$
- incremental update
- if $d_i < 0 \Rightarrow (x_{i+1}, y_{i+1}) = (x_i, y_i + 1)$
$$d_{i+1} = (x_i - \frac{1}{2})^2 + (y_i + 1)^2 - r^2$$

$$d_{i+1} = d_i + 2y_i + 1$$
- if $d_i \geq 0 \Rightarrow (x_{i+1}, y_{i+1}) = (x_i - 1, y_i + 1)$
$$d_{i+1} = d_i + 2y_i + 1 - 2x_i + 2$$



[Wikipedia: Rasterung von Kreisen]

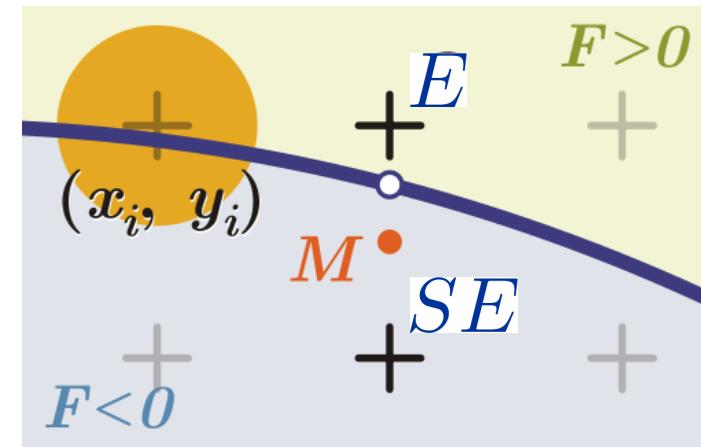
Horn Algorithm

Implementation

```
void HornCircle(int r) {  
  
    int d, x, y;  
  
    d = -r;  
    x = r;  
    y = 0;  
  
    while (y < x) {  
        WritePixel(x, y); /* and symmetric pixels */  
        d += 2*y + 1;  
        y += 1;  
        if (d >= 0) {  
            d += -2*x + 2;  
            x += -1;  
        }  
    }  
}
```

Bresenham Algorithm (Midpoint Algorithm)

- $F(x, y) = x^2 + y^2 - r^2 = 0 \Rightarrow (x, y)$ is on the circle
- based on the current pixel (x_i, y_i) , the algorithm decides whether to choose $(x_i + 1, y_i)$ or $(x_i + 1, y_i - 1)$ (E east, SE southeast)
- F is evaluated at the next midpoint
 - $F(x_i + 1, y_i - \frac{1}{2})$
 - $F(x_i + 1, y_i - \frac{1}{2}) \geq 0 \Rightarrow$ choose SE
 - $F(x_i + 1, y_i - \frac{1}{2}) < 0 \Rightarrow$ choose E



[Wikipedia: Rasterung von Kreisen]

Incremental Update of the Decision Variable

- decision variable $d_i = F(x_i + 1, y_i - \frac{1}{2})$
- incremental update from d_i to d_{i+1} depending on d_i
- $d_i \geq 0 \Rightarrow$ choose SE, $d_{i+1} = F(x_i + 2, y_i - 1 - \frac{1}{2})$
- $d_i < 0 \Rightarrow$ choose E, $d_{i+1} = F(x_i + 2, y_i - \frac{1}{2})$

- in case of $d_i \geq 0$:
$$\Delta_{SE} = 2x_i - 2y_i + 5$$
- in case of $d_i < 0$:
$$\Delta_E = 2x_i + 3$$

Incremental Update of the Increments

- four patterns of a set of three adjacent points
 - (1) $(x_i, y_i), (x_i + 1, y_i), (x_i + 2, y_i)$
 - (2) $(x_i, y_i), (x_i + 1, y_i), (x_i + 2, y_i - 1)$
 - (3) $(x_i, y_i), (x_i + 1, y_i - 1), (x_i + 2, y_i - 1)$
 - (4) $(x_i, y_i), (x_i + 1, y_i - 1), (x_i + 2, y_i - 2)$
- increments $\Delta_{E,i} = 2x_i + 3$ $\Delta_{SE,i} = 2x_i - 2y_i + 5$
 - if the algorithm moves towards E
 - (1) $\Delta_{E,i+1} = 2(x_i + 1) + 3 \Rightarrow \Delta_{E,i+1} = \Delta_{E,i} + 2$
 - (2) $\Delta_{SE,i+1} = 2(x_i + 1) - 2y_i + 5 \Rightarrow \Delta_{SE,i+1} = \Delta_{SE,i} + 2$
 - if the algorithm moves towards SE
 - (3) $\Delta_{E,i+1} = 2(x_i + 1) + 3 \Rightarrow \Delta_{E,i+1} = \Delta_{E,i} + 2$
 - (4) $\Delta_{SE,i+1} = 2(x_i + 1) - 2(y_i - 1) + 5 \Rightarrow \Delta_{SE,i+1} = \Delta_{SE,i} + 4$

Incremental Update of Increments

- point (x_i, y_i) is on the circle
- if next point is E,
 - $d_i = d_i + \Delta_{E,i}$
 - $\Delta_{E,i} = \Delta_{E,i} + 2$
 - $\Delta_{SE,i} = \Delta_{SE,i} + 2$
- if next point is SE,
 - $d_i = d_i + \Delta_{SE,i}$
 - $\Delta_{E,i} = \Delta_{E,i} + 2$
 - $\Delta_{SE,i} = \Delta_{SE,i} + 4$

Bresenham Algorithm

Initialization

- at point $(0, r)$

$$d_1 = F(0 + 1, r - \frac{1}{2}) = 1 + (r - \frac{1}{2})^2 - r^2 = \frac{5}{4} - r$$

$$\Delta_{SE} = -2r + 5$$

$$\Delta_E = 3$$

- as d is incremented only by integer values,
 $d_1 = 1 - r$

Bresenham Algorithm

Implementation

```
void BresenhamCircle (int r) {
    int x, y, d, deltaE, deltaSE;

    x = 0; y = r; d = 1 - r; deltaE = 3; deltaSE = -2*r + 5;

    WritePixel(x, y); /* and symmetric points */
    while (y > x) {
        if (d < 0) /* choose E */
            d += deltaE;
            deltaE += 2;
            deltaSE += 2;
        }
        else { /* choose SE */
            d += deltaSE;
            deltaE += 2;
            deltaSE += 4;
            y--;
        }
        x++;
        WritePixel(x, y); /* and symmetric points */
    }
}
```

Summary - Circles

- circle rasterization algorithms are usually described for one eighth of a circle and generalized using symmetries
- incremental updates are often employed
- floating-point arithmetic is avoided

Outline

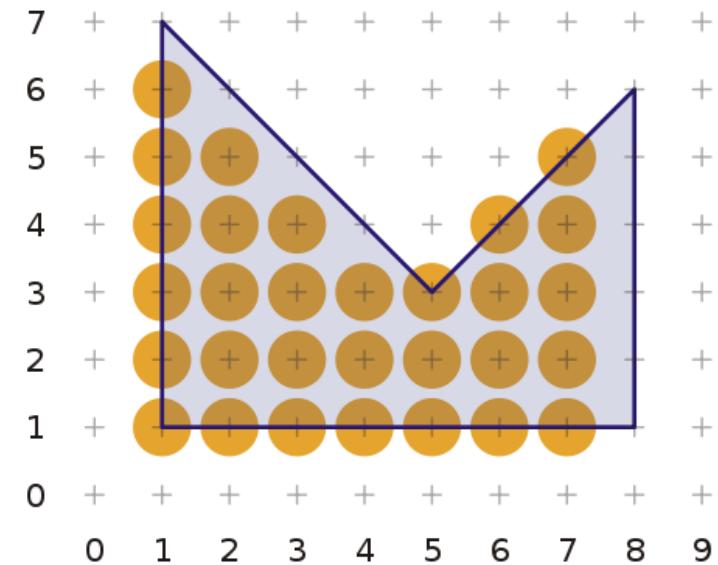
- lines
- circles
- polygons

General Setting

- a polygon is defined by edges
- the polygon should be closed to allow inside / outside classification
- rasterization estimates all pixel positions inside a polygon
- in general simple, but
 - if adjacent polygons share an edge, each pixel on the edge should belong to exactly one polygon
 - no pixel along the edge should be missed
 - no pixel along the edge should be rasterized twice

Edge List Algorithms

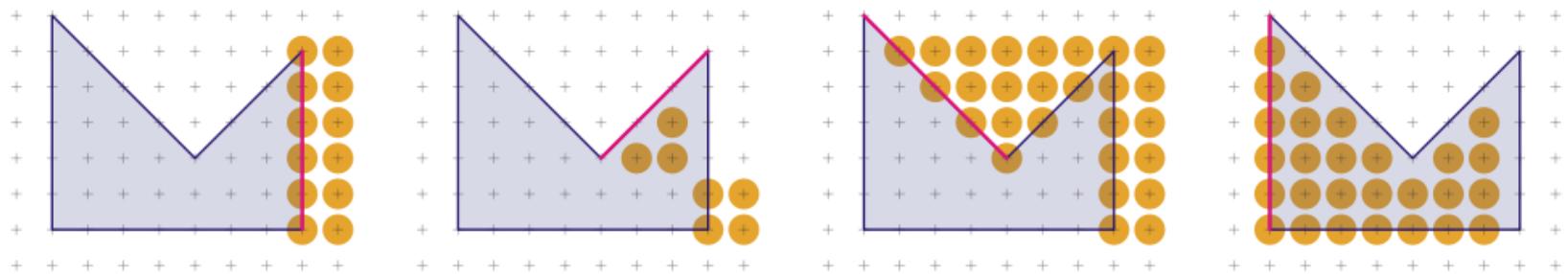
- compute intersections of non-horizontal polygon edges with lines (scanlines)
- intersections are computed for $y = y_i + 0.5$
- fill pixel positions in-between two intersection points
 - scan from left to right
 - enter the polygon at the first intersection, leave the polygon at the next intersection



[Wikipedia: Rasterung von Polygonen]

Edge Fill Algorithms

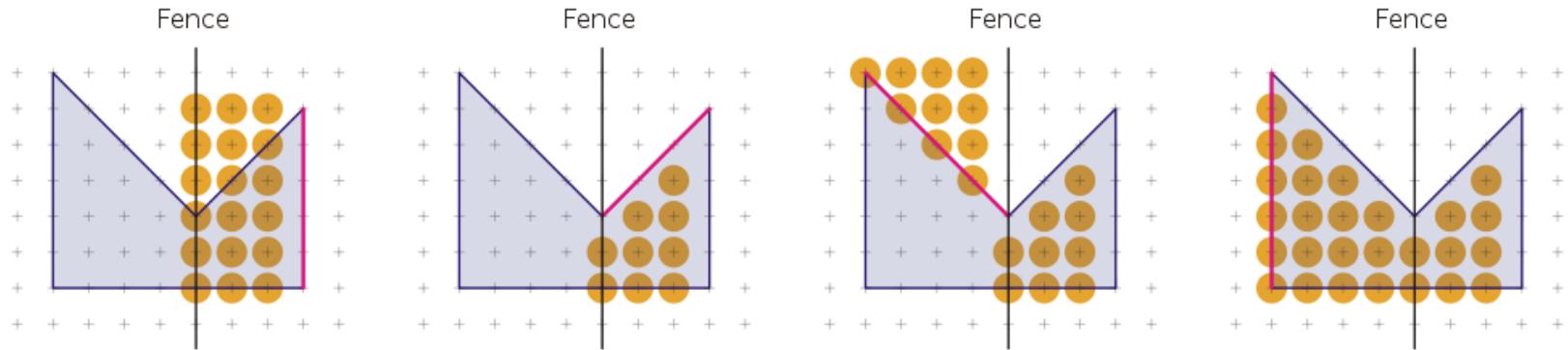
- for each polygon edge
 - process all scanlines intersected by the edge
 - invert all pixels with an x-component larger than the intersection point



[Wikipedia: Rasterung von Polygonen]

Fence Fill Algorithm

- for each polygon edge
 - process all scanlines intersected by the edge
 - if $x_{\text{intersection}} \geq x_{\text{fence}}$ invert all pixels with $x_{\text{fence}} \leq x_{\text{pixel}} < x_{\text{intersection}}$
 - if $x_{\text{intersection}} < x_{\text{fence}}$ invert all pixels with $x_{\text{intersection}} \leq x_{\text{pixel}} < x_{\text{fence}}$



[Wikipedia: Rasterung von Polygonen]

Summary - Polygons

- polygon rasterization algorithms work for closed polygons
 - inside / outside classification
- rasterization estimates all pixel positions inside a polygon
- processing of edges has to consider that pixels on shared edges should be rasterized exactly once

Summary

- vertices in window space and topology information are used to assemble primitives
- rasterization converts primitives to fragments with interpolated attributes
 - rasterization of lines
 - rasterization of circles
 - rasterization of polygons
- rasterized pixel positions with interpolated attributes are further processed in the rendering pipeline

Image Processing and Computer Graphics

Texture Mapping

Matthias Teschner

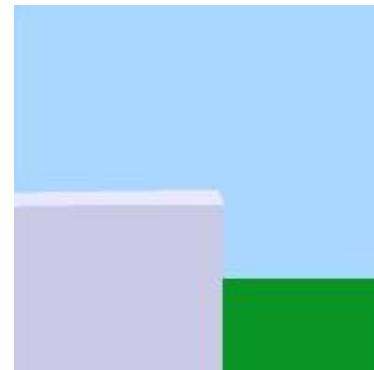
Computer Science Department
University of Freiburg

Albert-Ludwigs-Universität Freiburg

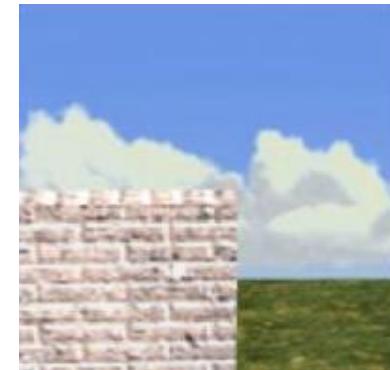


Motivation

- adding per-pixel surface details without raising the geometric complexity of a scene
- keep the number of vertices and primitives low
- add detail per fragment
- detail refers to any property that influences the final radiance, e.g. object or light properties
- modeling and rendering time is saved by keeping the geometrical complexity low



scene without
texture



scene with
color texture

colors can be stored in a texture (represented as an image) and applied to a rendered scene.

[Rosalee Wolfe]

Concept

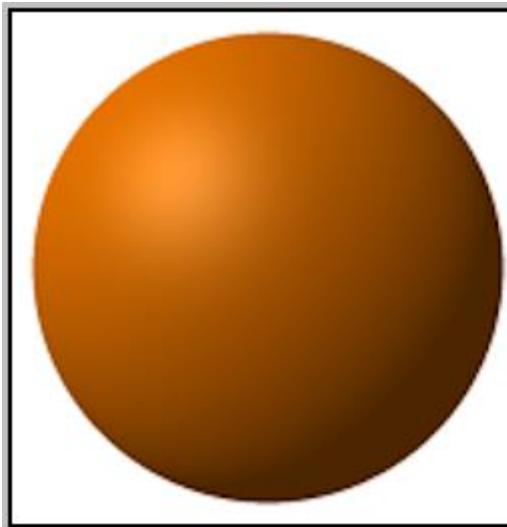
- 2D textures are represented as 2D images
- textures can store a variety of properties, i.e. colors, normals
- positions of texture pixels, i. e. **texels**, are characterized by **texture coordinates** (u, v) in **texture space**
- **texture mapping** is a transformation from object space to texture space $(x, y, z) \rightarrow (u, v)$
 - texture coordinates (u, v) are assigned to a vertex (x, y, z)
- **texture mapping** is generally applied **per fragment**
 - rasterization determines fragment positions and interpolates texture coordinates from adjacent vertices
 - texture lookup is performed per fragment using interpolated texture coordinates

Outline

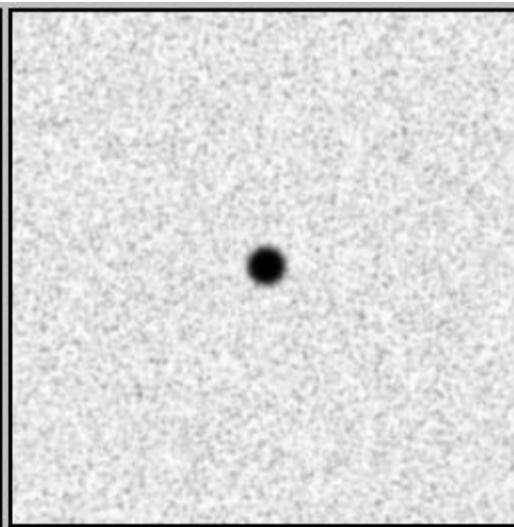
- applications
- concept
- image texturing

Bump Mapping

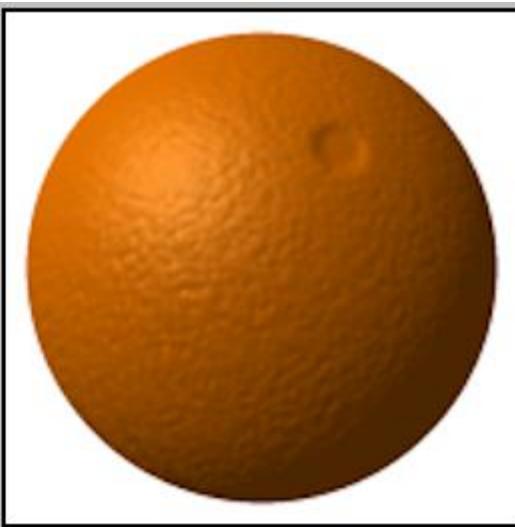
- normals / normal deviations can be stored in a texture



object without
bump mapping



bump texture encodes
normals or normal deviations

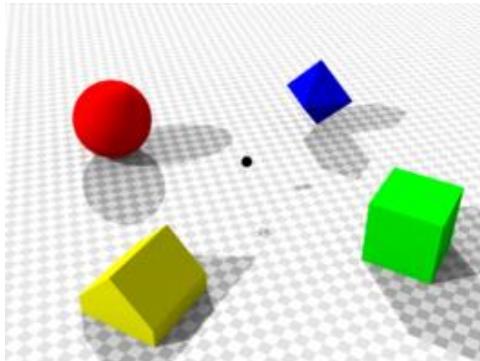


object with
bump mapping

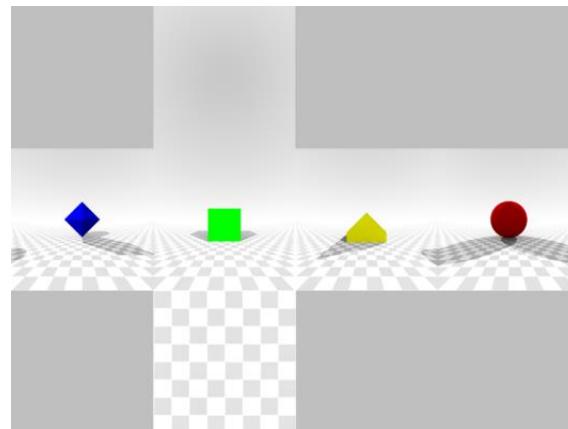
[Wikipedia: Bump Mapping]

Environment Mapping

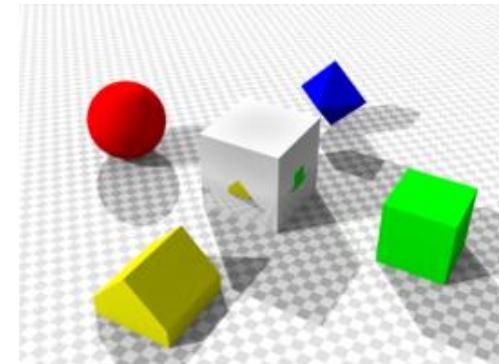
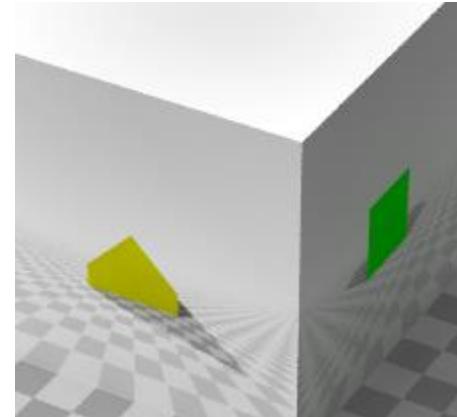
- cube mapping
- approximates reflections of the environment



place a viewer in a scene



generate the environment texture from six view directions

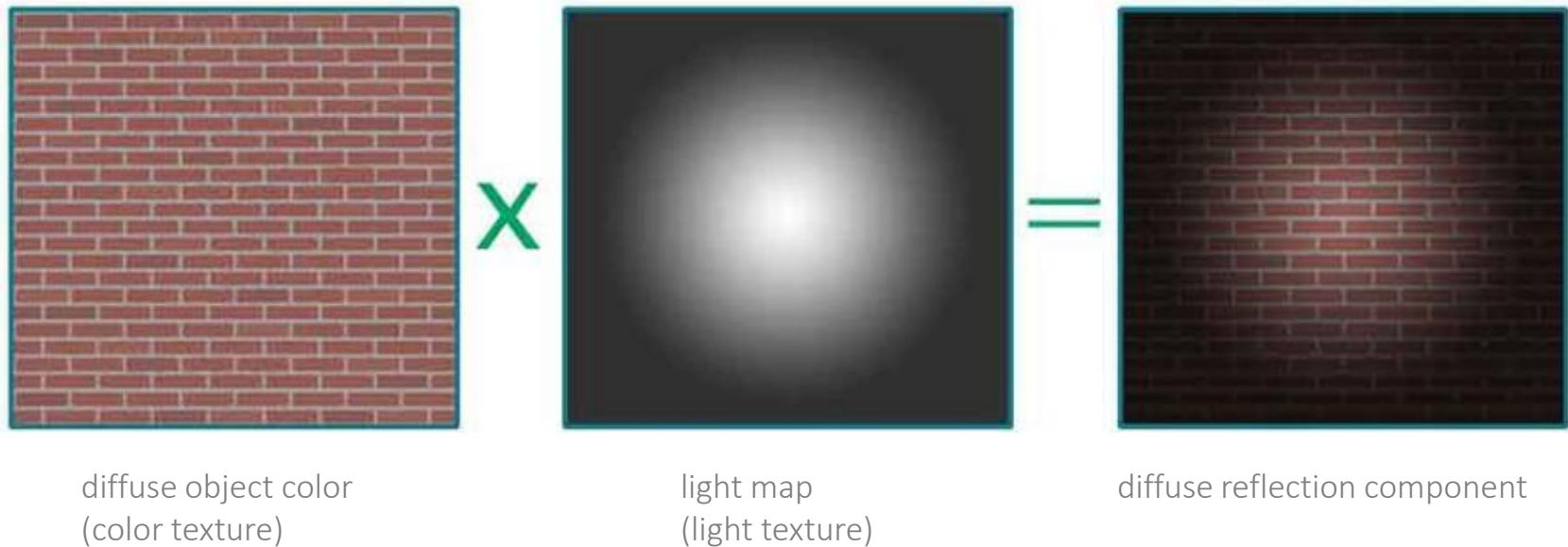


apply the texture to an object at the position of the viewer

[Wikipedia: Cube Mapping]

University of Freiburg – Computer Science Department – Computer Graphics - 6

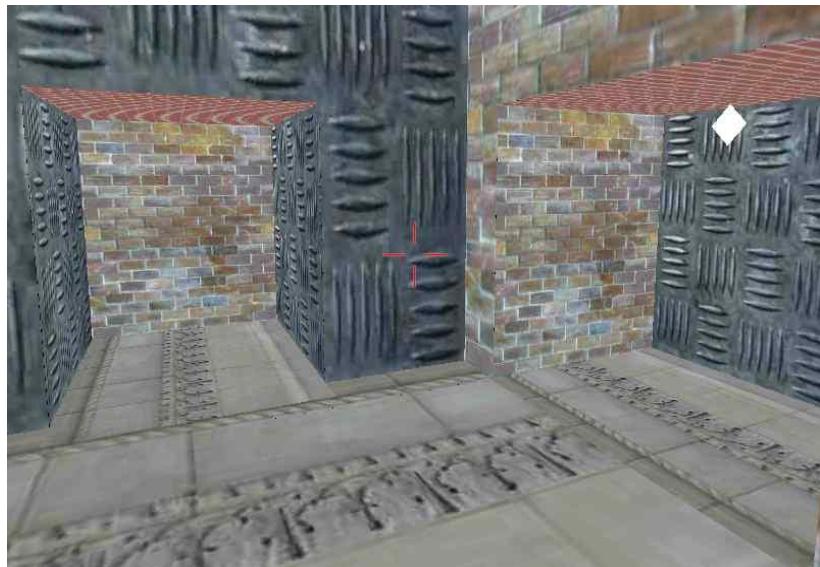
Light Mapping



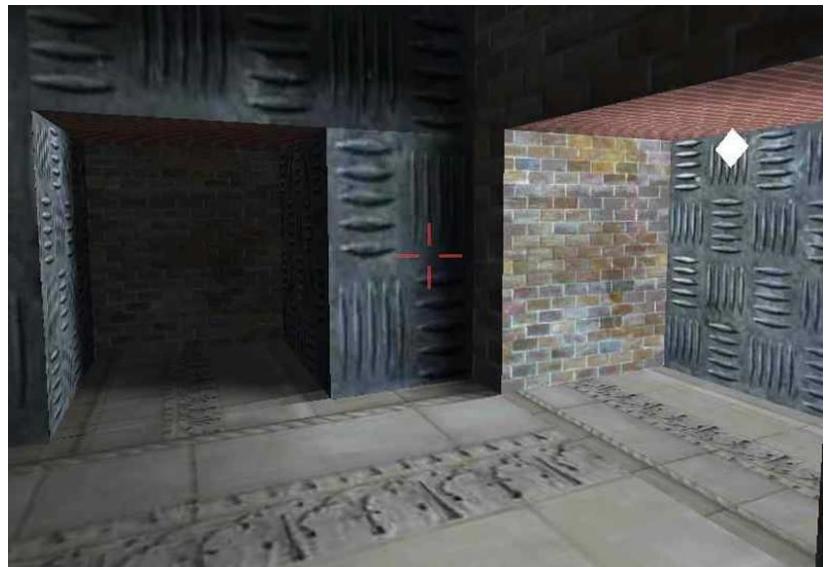
[Keshav Channa: Light Mapping - Theory and Implementation]

University of Freiburg – Computer Science Department – Computer Graphics - 7

Light Mapping



scene with color texture



scene with color and light texture

[Keshav Channa: Light Mapping - Theory and Implementation]

University of Freiburg – Computer Science Department – Computer Graphics - 8

Billboards



color texture



alpha texture
representing
transparency

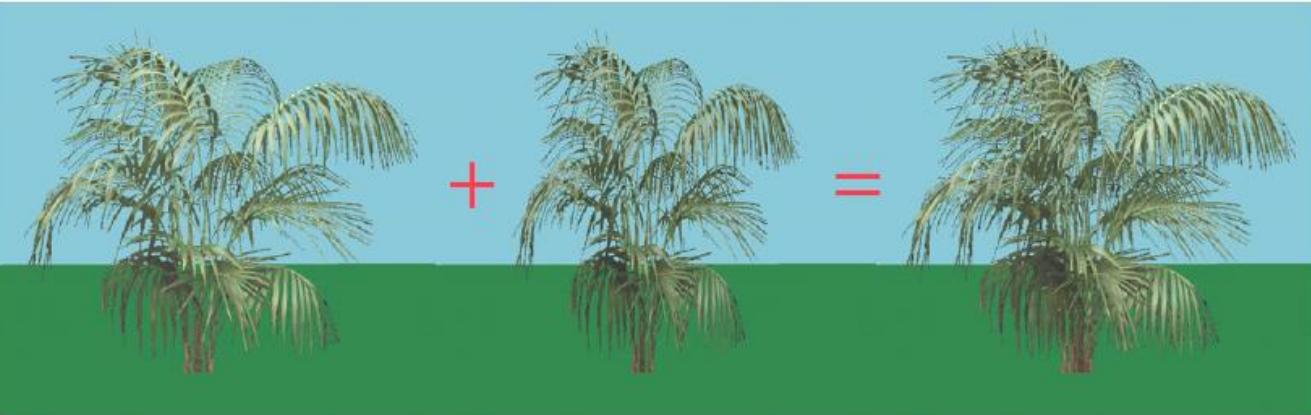


billboard: simple primitive
with color and alpha mapping

Billboards



color and
alpha texture



billboards at different orientations

combined 3D billboard



viewed at appropriate directions,
a small number of combined 2D
billboards provides a realistic 3D
impression

[Akenine-Moeller et al.: Real-time Rendering]

University of Freiburg – Computer Science Department – Computer Graphics - 10

Outline

- applications
- concept
- image texturing

Texturing Pipeline

- object space location
 - texture coordinates are defined for object space locations
 - if the object moves in world space,
the texture moves along with it
- projector function
 - maps a 3D object space location
to a (2D) parameter-space value
- corresponder functions
 - map from parameter space to texture space
 - texture-space values are used to obtain values from a texture
- value transform
 - a function that transforms obtained texture values

Object vs. World Space

- if the object moves in world space,
the texture should move along with it



texture coordinates
assigned to object
space positions

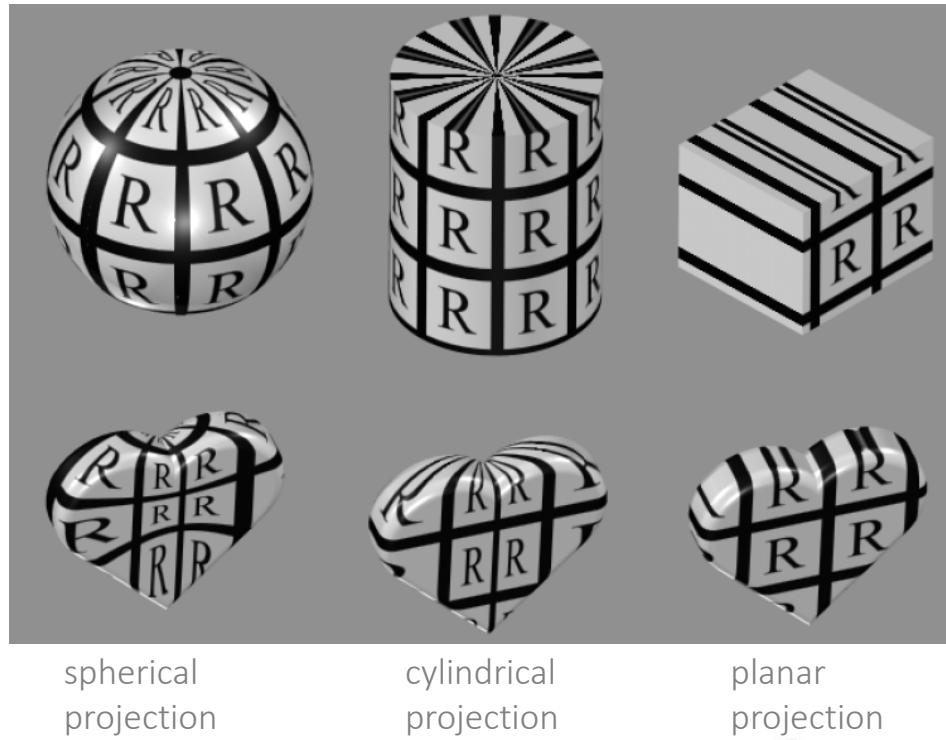


texture coordinates
assigned to world
space positions

[Rosalee Wolfe]

Projector Functions

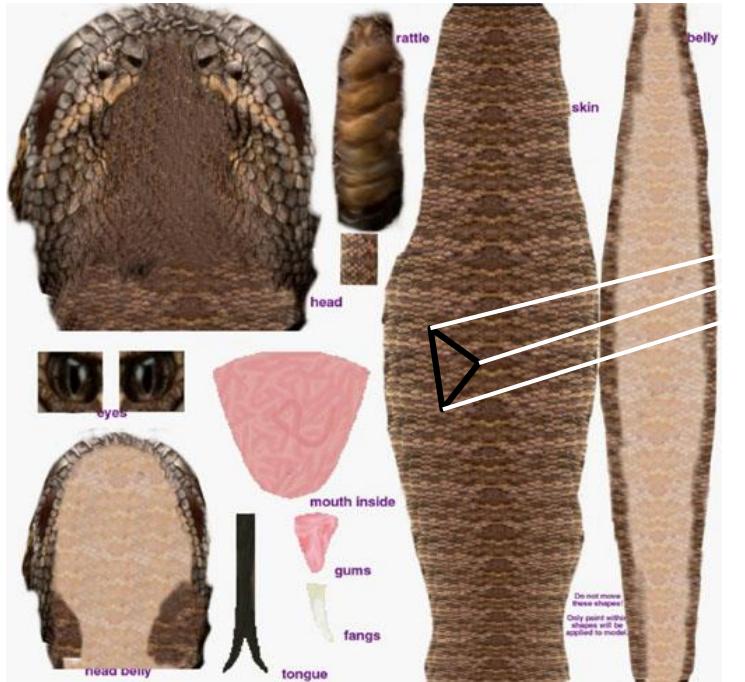
- a method to generate texture coordinates,
i.e. a method for surface parameterization
- planar projection
 $(x, y, z) \rightarrow (u, v)$
- cylindrical projection
 $(x, y, z) \rightarrow (r, \theta, h) \rightarrow (u, v)$
- spherical projection
 $(x, y, z) \rightarrow (\text{latitude}, \text{longitude}) \rightarrow (u, v)$
- can be an initialization
step for the surface
parameterization



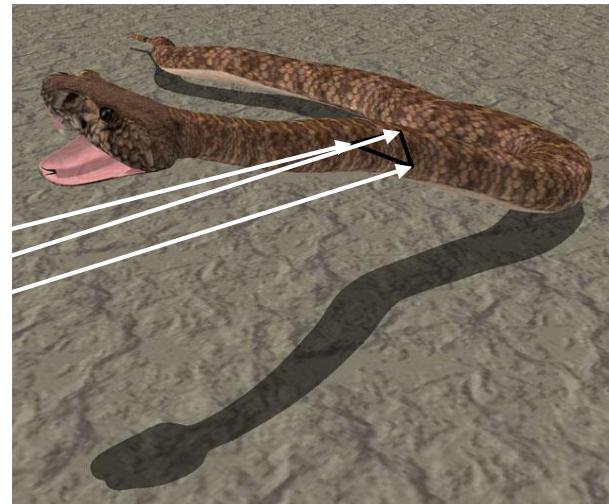
[Akenine-Moeller et al.: Real-time Rendering]

Projector Functions

- use of different projections for surface patches
 - minimize distortions, avoid artifacts at seams
 - several texture coordinates can be assigned to a single vertex



texture

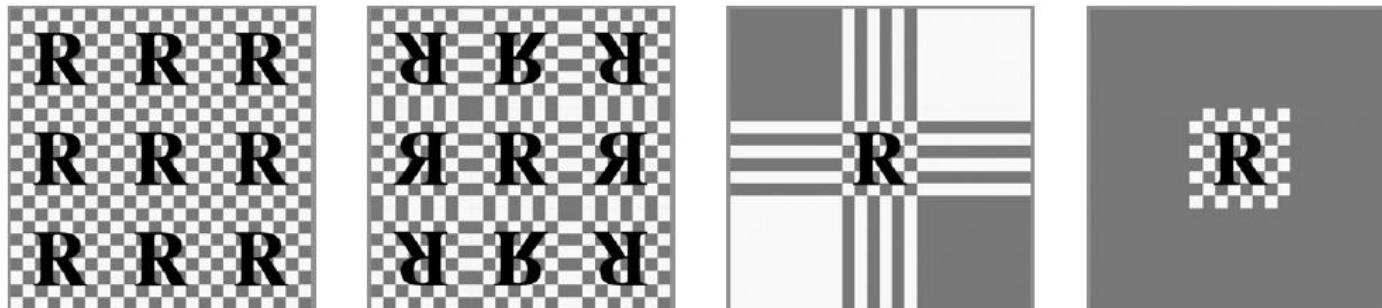


textured object

Corresponder Functions

- one or more corresponder functions transform from parameter space to texture space
- transform of (u, v) into a valid range from 0 to 1, e.g.
 - repeat (the texture repeats itself, integer value is dropped)
 - mirror (the texture repeats itself, but is mirrored)
 - clamp to edge (values outside $[0,1]$ are clamped to this range)
 - clamp to border (values outside $[0,1]$ are rendered with a border color)

[Akenine-Moeller et al.: Real-time Rendering]



Value Transform / Texture Blending Functions

- value transform
 - arbitrary transformations, e.g. represented by a matrix in homogeneous form, can be applied to texture values
- texture blending functions
 - define how to use the texture value, e.g.
 - replace the original surface color with the texture color
 - linearly combine the original surface color with the texture
 - multiply, add, subtract surface color and texture color
- alternatively, texture values can be used in the computation of the illumination model

Outline

- applications
- concept
- image texturing
 - perspective-correct interpolation
 - magnification
 - minification

Image Texturing

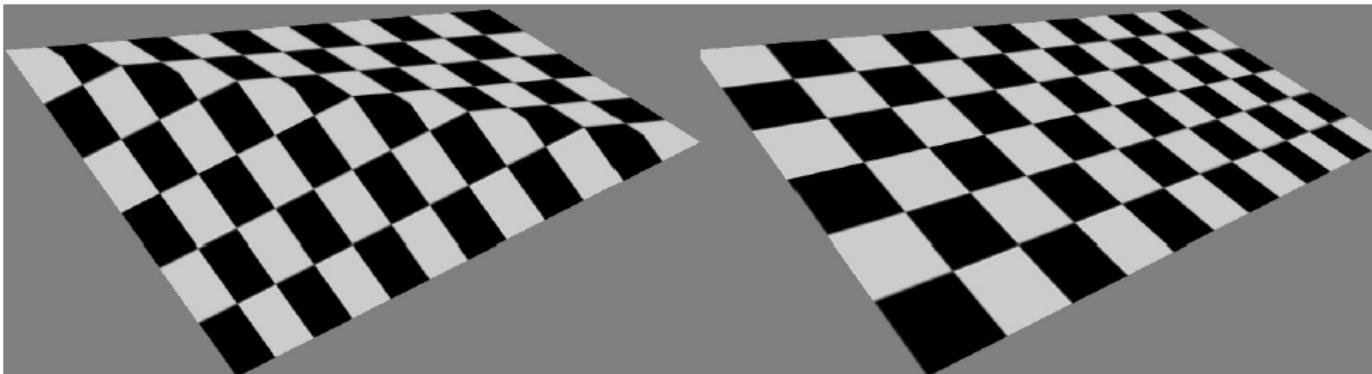
- 2D image is mapped / glued to a triangulated object surface
- certain aspects affect the quality
- interpolation of texture coordinates
 - has to consider the nonlinear transformation of the z-component from camera space to clip space in case of perspective projection
- sampling
 - textures can be change their size when applied to the object (magnification / minification)

Outline

- applications
- concept
- image texturing
 - perspective-correct interpolation
 - magnification
 - minification

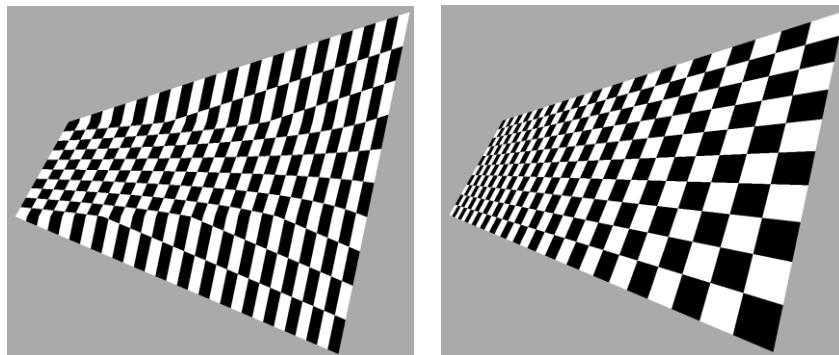
Linear vs. Perspective-Correct Interpolation

- two triangles in the same plane



[Akenine-Moeller et al.: Real-time Rendering]

- a quadrilateral

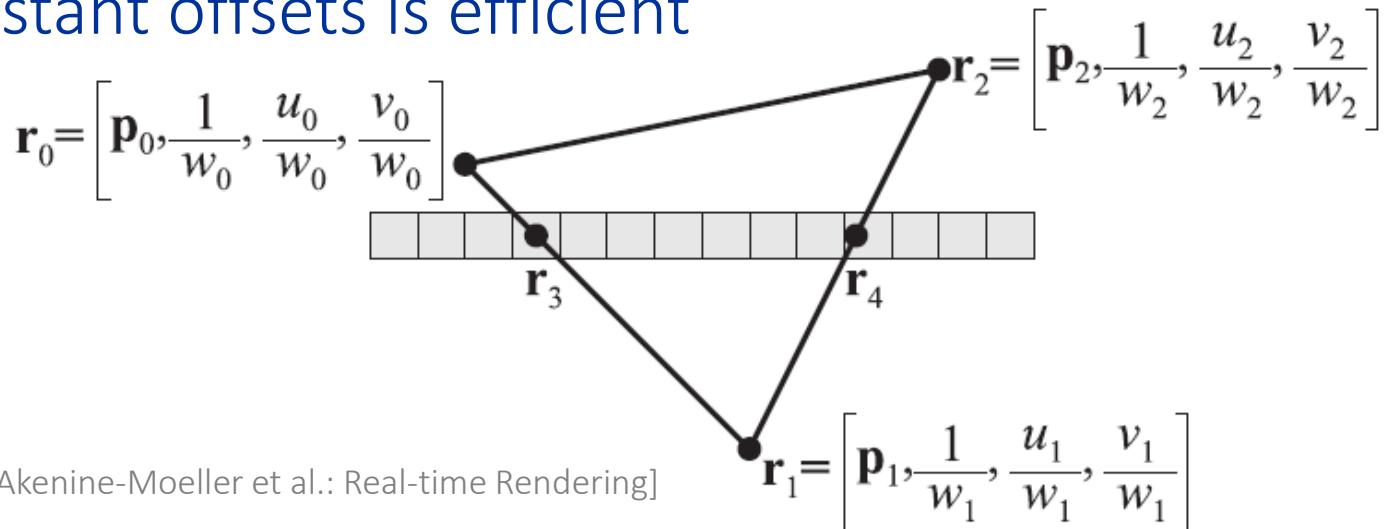


[Heckbert, Moreton]

Perspective-Correct Interpolation

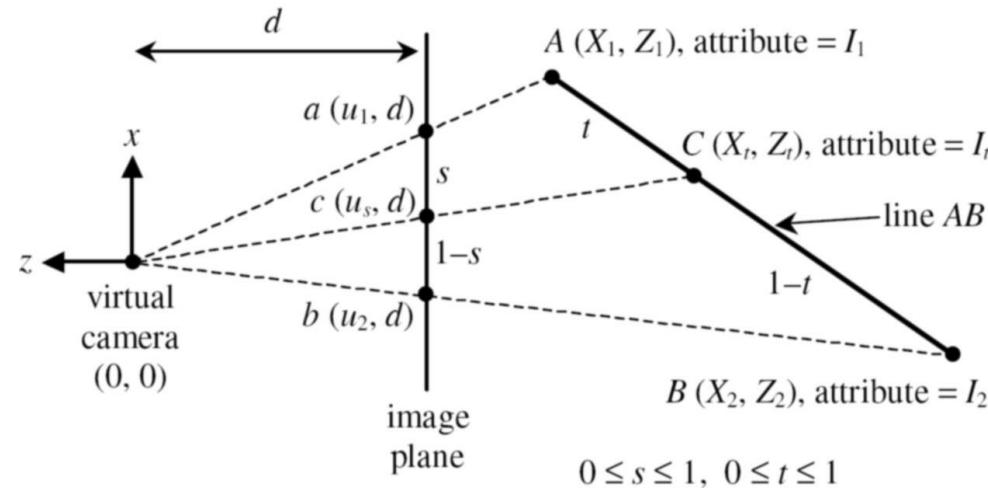
Solution

- u and v are not linearly interpolated
- instead, u / w_{clip} , v / w_{clip} , and $1 / w_{clip}$ are interpolated
- the resulting interpolated values for u / w_{clip} and v / w_{clip} are divided by the interpolated value for $1 / w_{clip}$
- motivated by the fact that linear interpolation with constant offsets is efficient



[Akenine-Moeller et al.: Real-time Rendering]

Perspective-Correct Interpolation



Perspective projection of a line AB. $t / (1-t)$ is not equal to $s / (1-s)$. Therefore, linear interpolation between a and b does not correspond to a linear interpolation between A and B.

$$I_t = I_1 + t(I_2 - I_1)$$

linear interpolation in camera space

$$I_t = I_1 + \frac{sZ_1}{sZ_1 + (1-s)Z_2} (I_2 - I_1)$$

non-linear interpolation in screen space

$$I_t = \frac{\frac{I_1}{Z_1} + s \left(\frac{I_2}{Z_2} - \frac{I_1}{Z_1} \right)}{\frac{1}{Z_1} + s \left(\frac{1}{Z_2} - \frac{1}{Z_1} \right)}$$

linear interpolation of $1/Z$ and $1/Z$ in screen space

[Kok-Lim Low: Perspective-Correct Interpolation]

Perspective-Correct Interpolation

- perspective projection transform

$$\begin{pmatrix} x_{clip} \\ y_{clip} \\ z_{clip} \\ w_{clip} \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} X_{camera} \\ Y_{camera} \\ Z_{camera} \\ 1 \end{pmatrix}$$

- linear relation between w in clip space and Z in camera space $w_{clip} = -Z_{camera}$
- therefore, Z_{camera} or w_{clip} can be used in the interpolation

$$I_t = \frac{\frac{I_1}{Z_1} + s \left(\frac{I_2}{Z_2} - \frac{I_1}{Z_1} \right)}{\frac{1}{Z_1} + s \left(\frac{1}{Z_2} - \frac{1}{Z_1} \right)}$$

$$I_t = \frac{\frac{I_1}{w_1} + s \left(\frac{I_2}{w_2} - \frac{I_1}{w_1} \right)}{\frac{1}{w_1} + s \left(\frac{1}{w_2} - \frac{1}{w_1} \right)}$$

Perspective-Correct Interpolation

- is only an issue for perspective projection
 - for parallel projection, linear interpolation in screen space corresponds to linear interpolation in camera space
- is not an issue for lines and polygons parallel to the near plane
- can be applied to all vertex attributes, i.e. color, depth, normal, texture coordinates
- interp. is commonly based on w_{clip} instead of Z_{camera}
 - Z_{camera} is not available in screen space
 - w_{clip} can be kept when converting from clip to NDC space

Outline

- applications
- concept
- image texturing
 - perspective-correct interpolation
 - magnification
 - minification

Magnification

- adjacent pixel positions in window space map to the same texel, i.e. pixel position in an image texture
- texture is magnified

[Akenine-Moeller et al.: Real-time Rendering]



nearest neighbor
(nearest texel)



bilinear interpolation
(weighted average
of 2x2 texels)



bicubic interpolation
(weighted average
of 5x5 texels)

Filtering

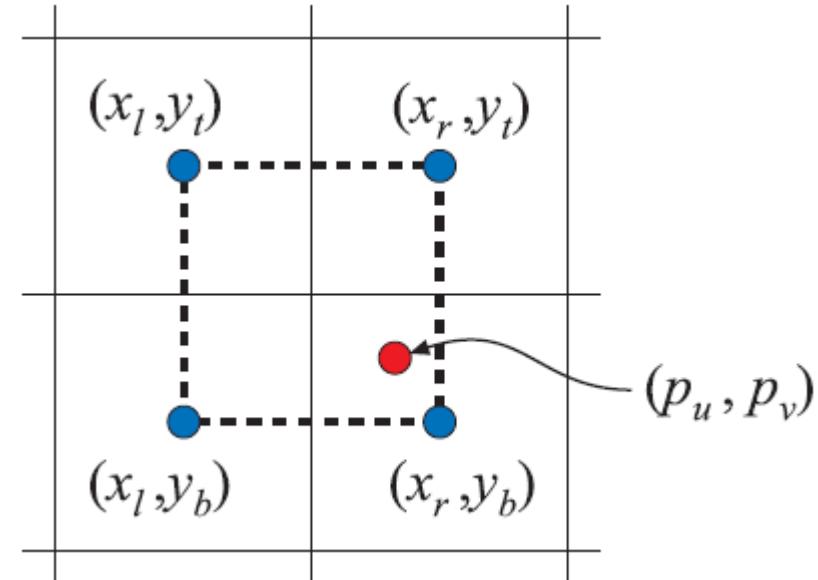
- bilinear interpolation

$$(u', v') = (p_u - \lfloor p_u \rfloor, p_v - \lfloor p_v \rfloor)$$

relative position within four adjacent texel positions

$$\begin{aligned}\mathbf{b}(p_u, p_v) = & (1 - u')(1 - v')\mathbf{t}(x_l, y_b) \\ & + u'(1 - v')\mathbf{t}(x_r, y_b) \\ & + (1 - u')v'\mathbf{t}(x_l, y_t) \\ & + u'v'\mathbf{t}(x_r, y_t)\end{aligned}$$

b (p_u, p_v) is interpolated from four texels
 $\mathbf{t}(x_l, y_b), \mathbf{t}(x_r, y_b), \mathbf{t}(x_l, y_t), \mathbf{t}(x_r, y_t)$

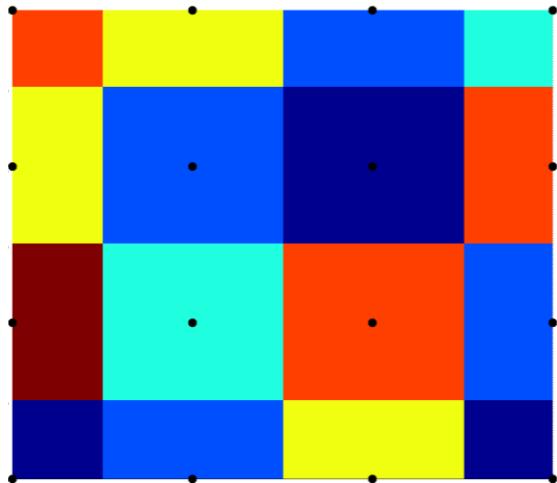


[Akenine-Moeller et al.: Real-time Rendering]

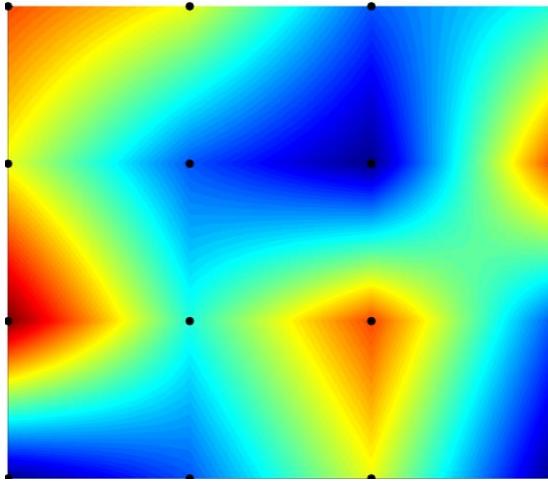
- bicubic interpolation

$$\mathbf{b}(p_u, p_v) = \sum_{i=0}^3 \sum_{j=0}^3 a_{ij} (u')^i (v')^j$$

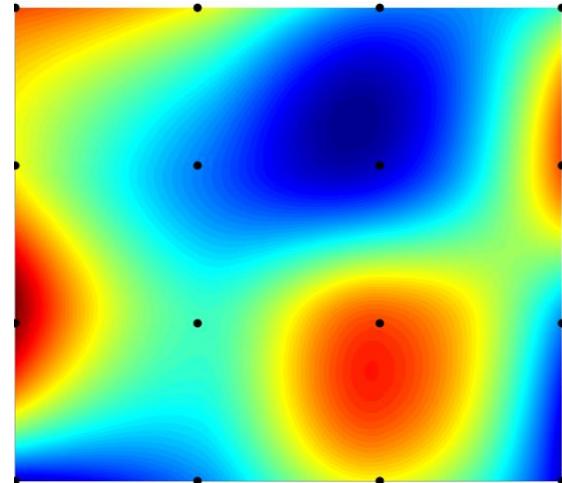
Examples



nearest neighbor
(nearest texel)



bilinear interpolation
(weighted average
of 2x2 texels)



bicubic interpolation
(weighted average
of 2x2 texels)

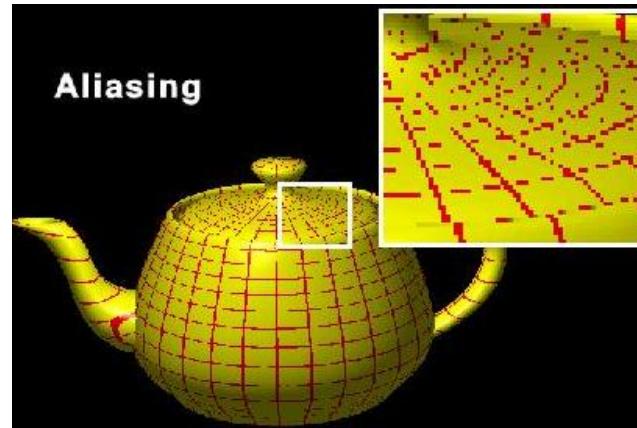
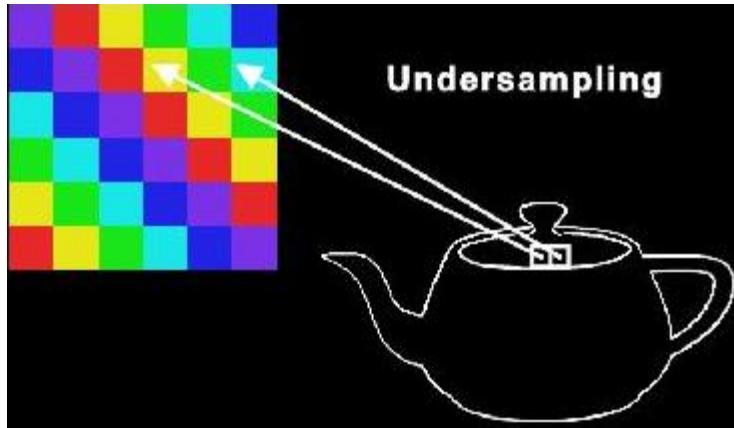
[Wikipedia: Bicubic interpolation]

Outline

- applications
- concept
- image texturing
 - perspective-correct interpolation
 - magnification
 - minification

Minification

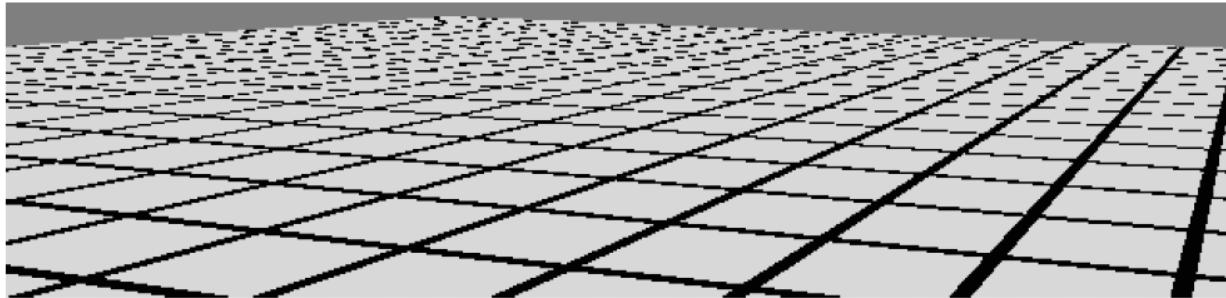
- adjacent texels map to the same pixel position in window space
- texture is minimized
- adjacent pixel positions do not map to adjacent texels (undersampling), which can cause aliasing



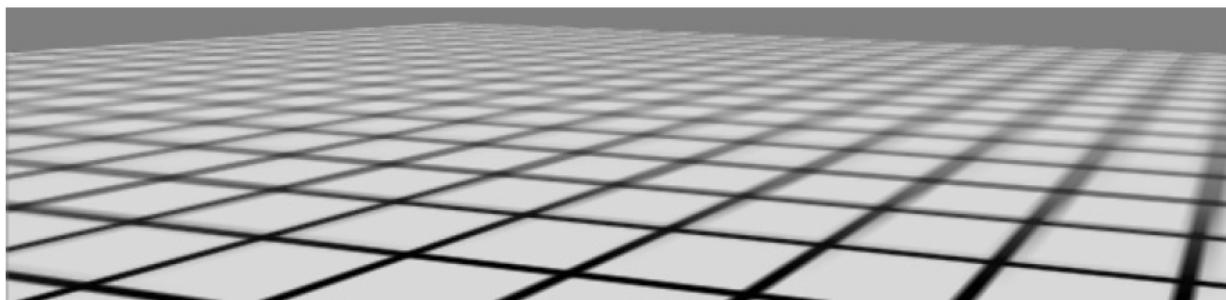
Undersampling can lead to artifacts, i. e. aliasing,
as information from the texture is lost

[Rosalee Wolfe]

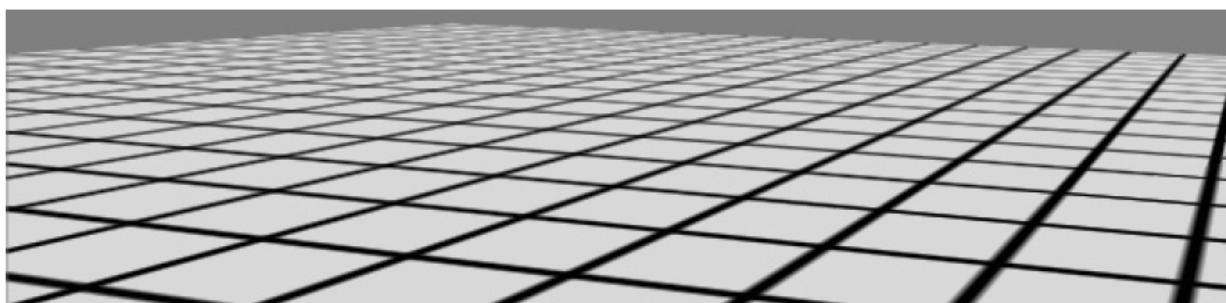
Filtering



nearest neighbor



mipmapping

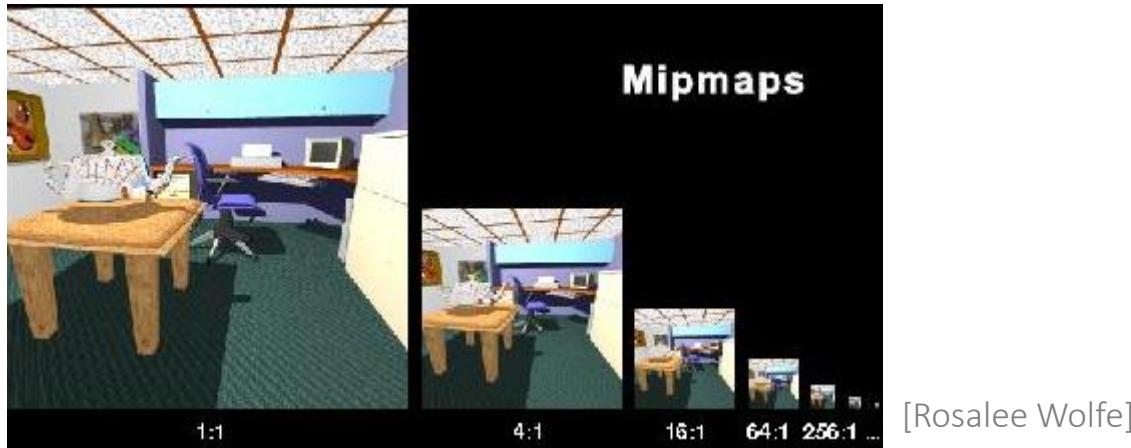


summed area tables

[Akenine-Moeller et al.: Real-time Rendering]

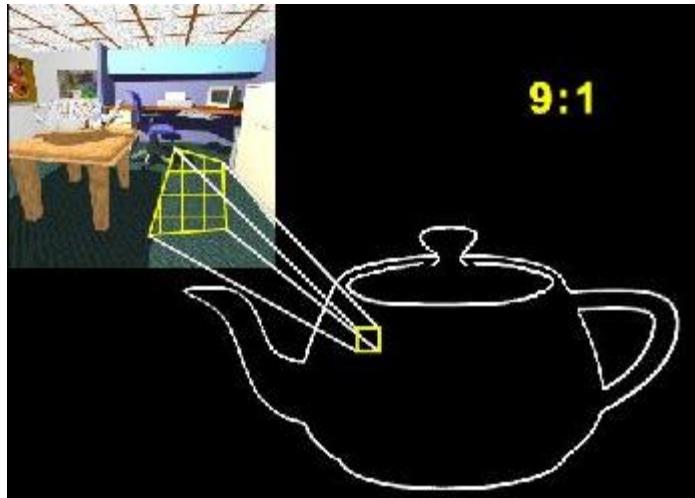
Mipmapping

- mip - multum in parvo, many things in a small space
- a set of smaller, low-pass filtered textures is generated

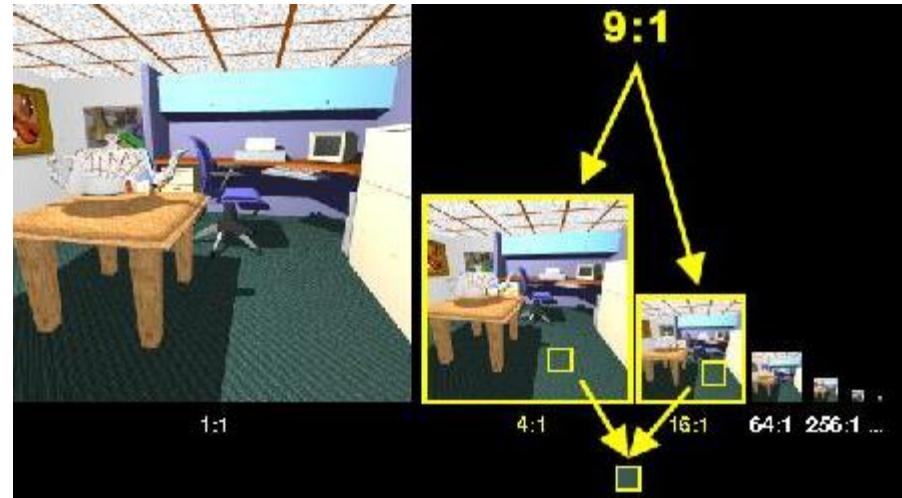


- mipmaps can be extended to riphmaps with sub-textures of rectangular areas to avoid overblurring

Mipmapping



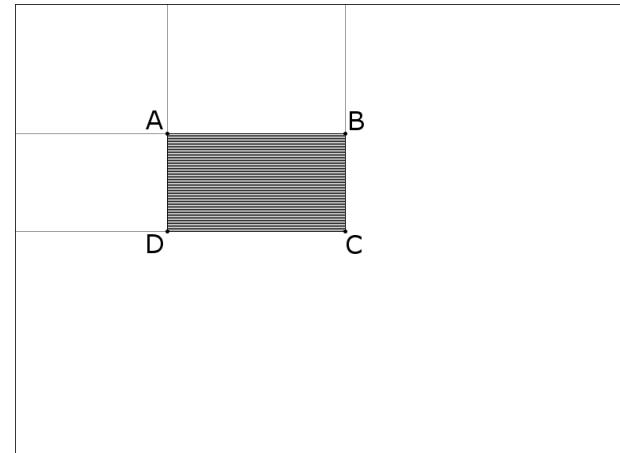
estimation of the texture area
covered by a pixel in window space,
e. g. by considering the texture coor-
dinates of 2x2 pixels in window space



choosing the appropriate subtexture,
applying additional filters, i. e.
interpolation of texels of two
adjacent subtextures.

Summed Area Table

- reduced overblurring due to the consideration of a more accurate texture area
- generate a second texture that stores
$$\text{sum}(x, y) = \sum_{x' \leq x, y' \leq y} \mathbf{t}(x', y')$$
- the sum of all texels within the rectangle A, B, C, D is
$$\text{sum}(\mathbf{C}) - \text{sum}(\mathbf{B}) - \text{sum}(\mathbf{D}) + \text{sum}(\mathbf{A})$$
- if a pixel in window space covers rectangle A, B, C, D in the texture, the texture value can be, e. g.,
$$\mathbf{b}(x, y) = \frac{\text{sum}(\mathbf{C}) - \text{sum}(\mathbf{B}) - \text{sum}(\mathbf{D}) + \text{sum}(\mathbf{A})}{\text{area}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})}$$



[Wikipedia: Summed area table]

Summary

- textures add detail without raising the geom.complexity
- textures can influence a variety of properties
- textures can be 1D, 2D, 3D, ..., or procedural
- texture coordinates at vertices or fragments are used to lookup texels
- quality of applied textures can be improved by
 - perspective-correct interpolation
 - considering magnification and minification
- examples
 - color, alpha, environment mapping, light, bump, parallax, relief mapping, ...

Image Processing and Computer Graphics

Shadow Algorithms

Matthias Teschner

Computer Science Department
University of Freiburg

Albert-Ludwigs-Universität Freiburg

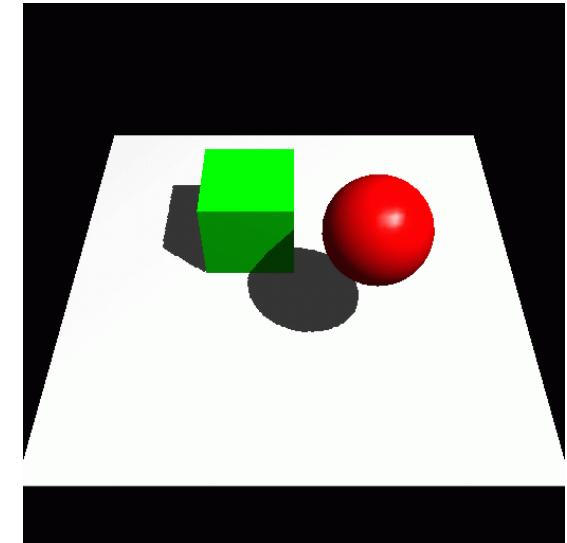
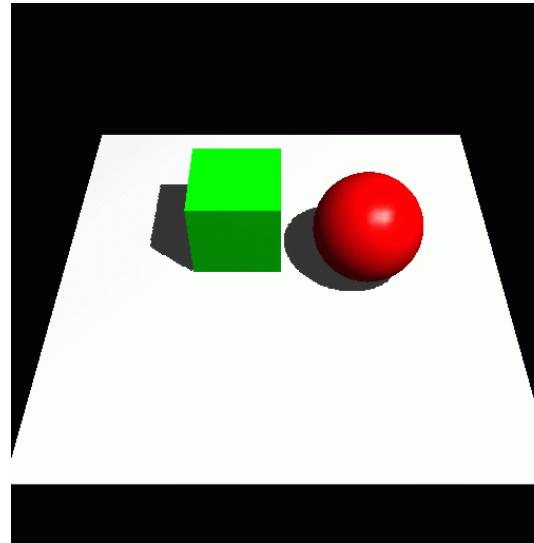
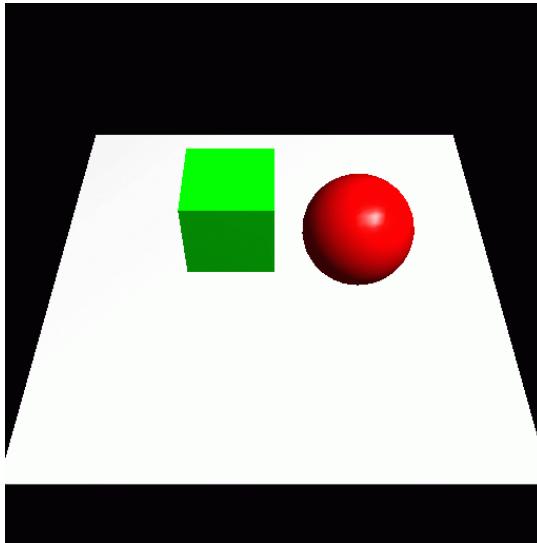


Outline

- introduction
- projection shadows
- shadow maps
- shadow volumes
- conclusion

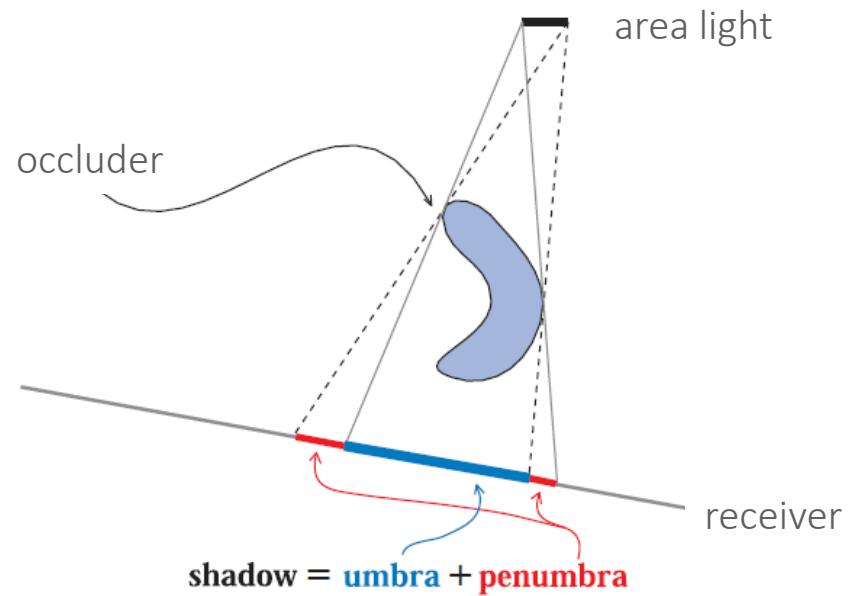
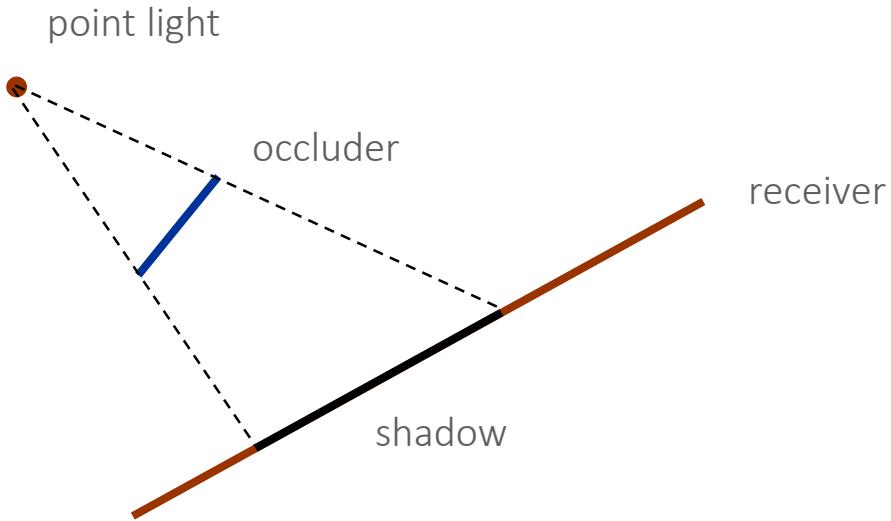
Motivation

- shadows help to
 - improve the realism in rendered images
 - illustrate spatial relations between objects



Goal

- determination of shadowed parts in 3D scenes



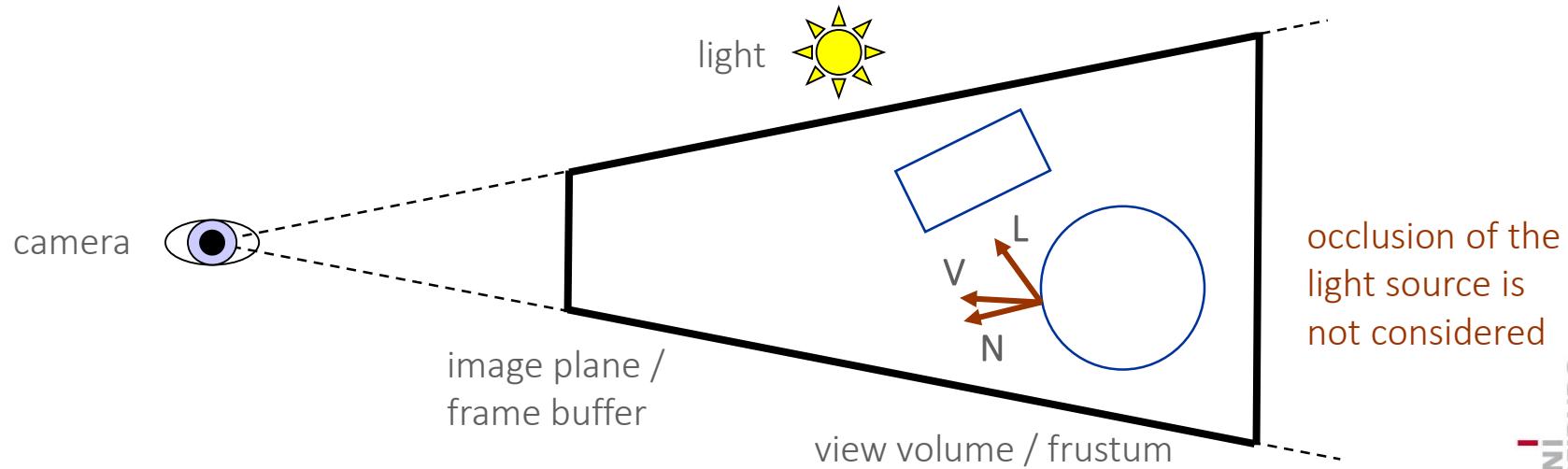
- only the geometry is considered

[Akenine-Moeller et al.: Real-time Rendering]

University of Freiburg – Computer Science Department – Computer Graphics - 4

Context

- shadow algorithms are not standard functionality of the rasterization-based rendering pipeline
- rendering pipeline
 - generates 2D images from 3D scenes (camera, light, objects)
 - evaluates lighting models using local information
 - spatial relations among objects are not considered

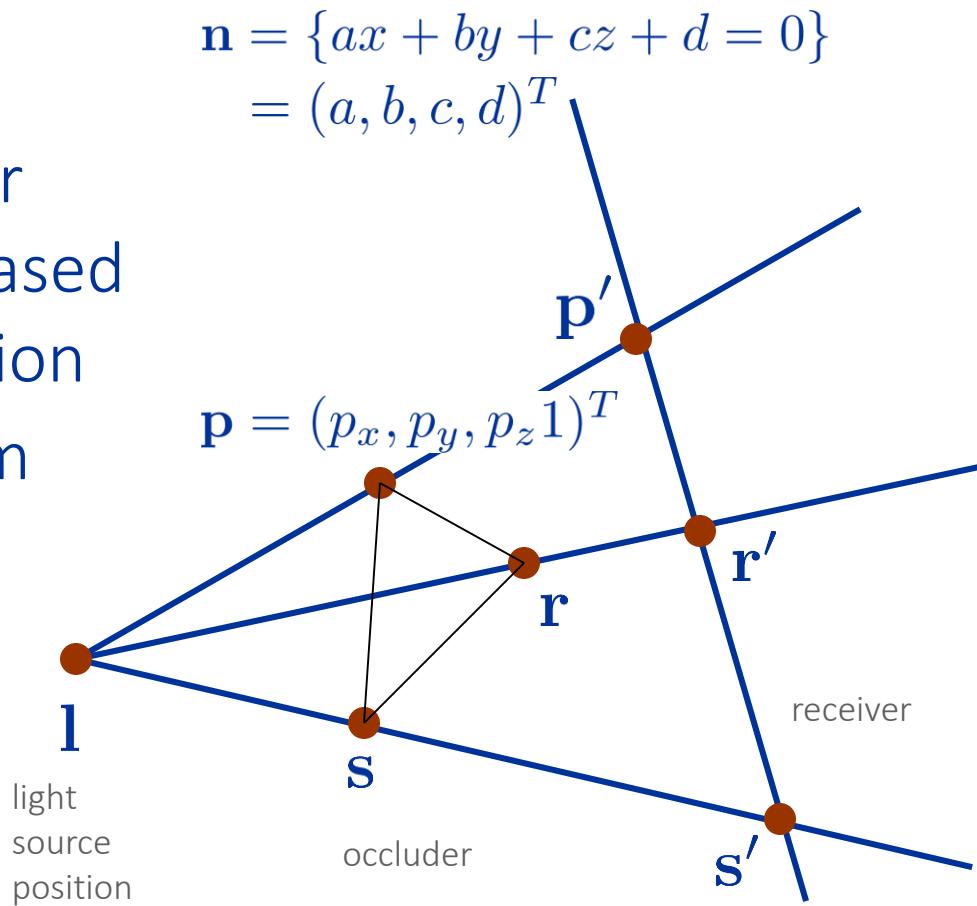


Outline

- introduction
- projection shadows
- shadow maps
- shadow volumes
- conclusion

Projection Shadows

- project the primitives of occluders onto a receiver plane (ground or wall) based on the light source location
- projected primitives form the shadow geometry
- projection matrix
 $P = \mathbf{l}\mathbf{n}^T - (\mathbf{n} \cdot \mathbf{l})\mathbf{I}_4$



Projection Shadows

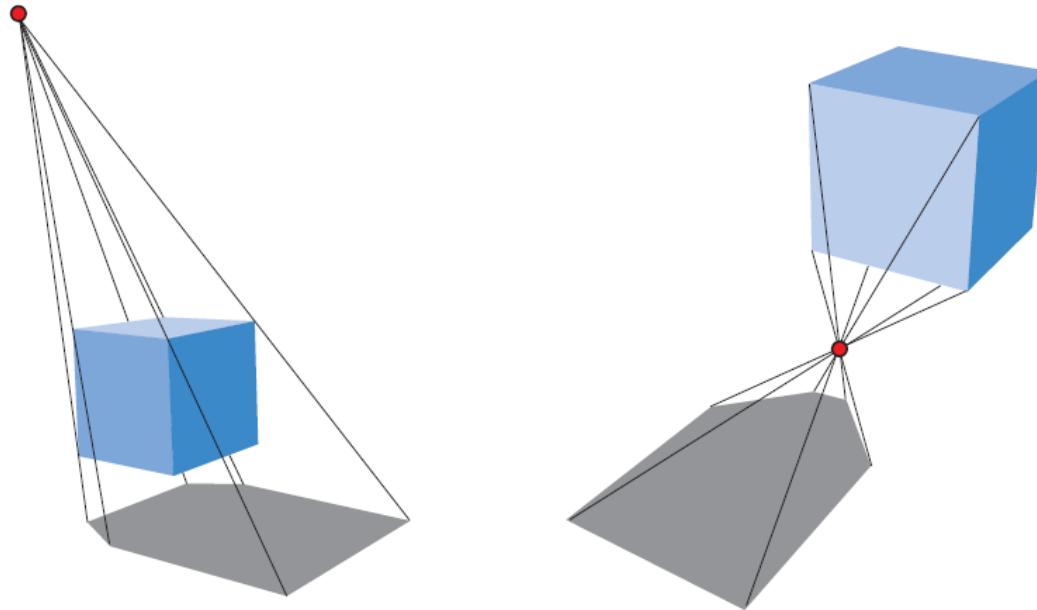
Implementation

- draw the receiver plane
 - increment stencil where the receiver is rendered
- disable depth test
- draw shadow geometry (projected occluders)
for stencil=1
- enable depth test
- draw occluders

Projection Shadows

Issues

- restricted to planar receivers
- no self-shadowing
- antishadows



[Akenine-Moeller et al.: Real-time Rendering]

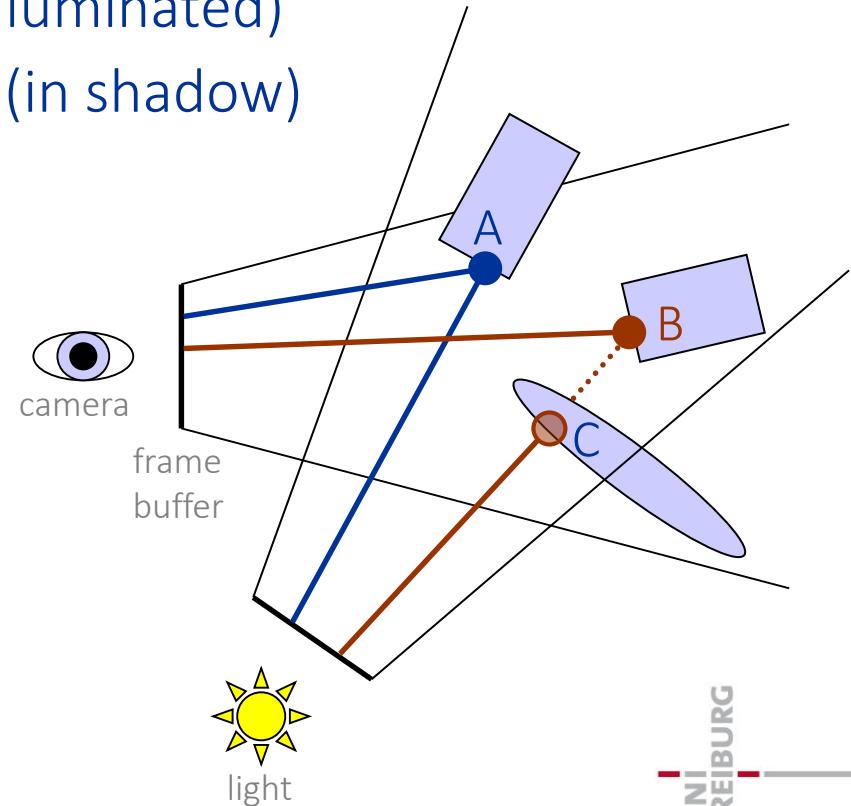
University of Freiburg – Computer Science Department – Computer Graphics - 9

Outline

- introduction
- projection shadows
- shadow maps
- shadow volumes
- conclusion

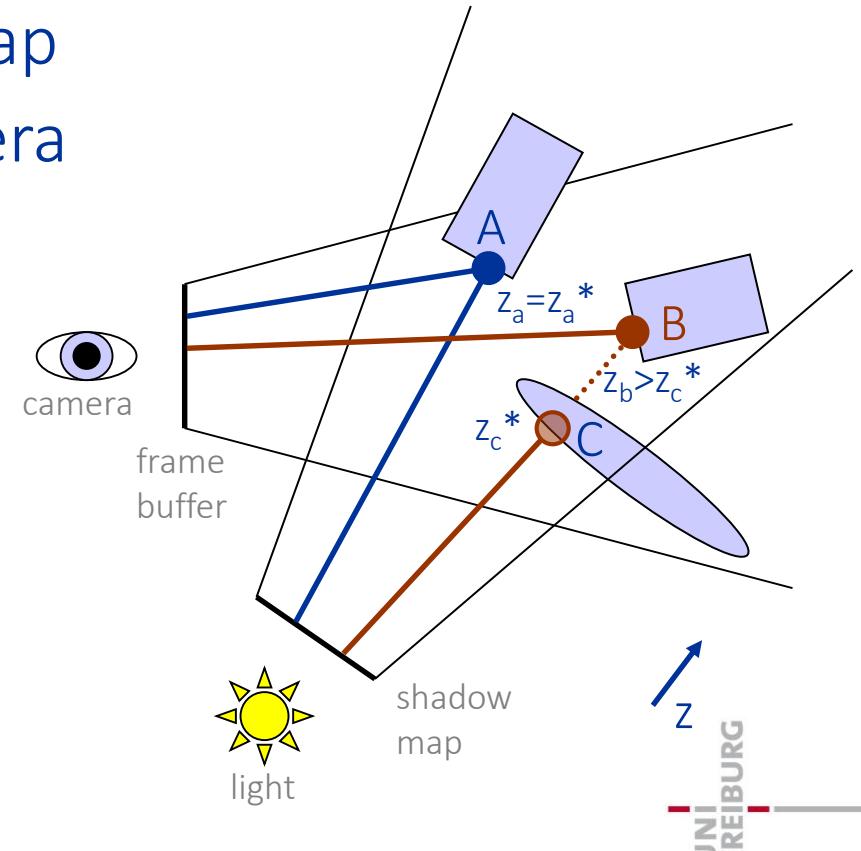
Concept

- see shadow casting as a visibility problem
- scene points are
 - visible from the light source (illuminated)
 - invisible from the light source (in shadow)
- resolving visibility is standard functionality in the rendering pipeline (z-buffer algorithm)



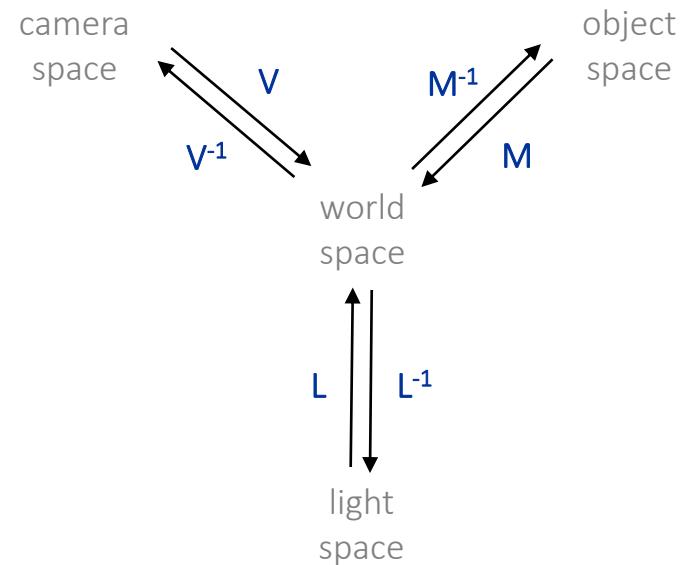
Algorithm

- render scene from the light source
- store all distances to visible (illuminated) scene points in a shadow map
- render scene from the camera
- compare the distance of rendered scene points to the light with stored values in the shadow map
- if both distances are equal, the rendered scene point is illuminated

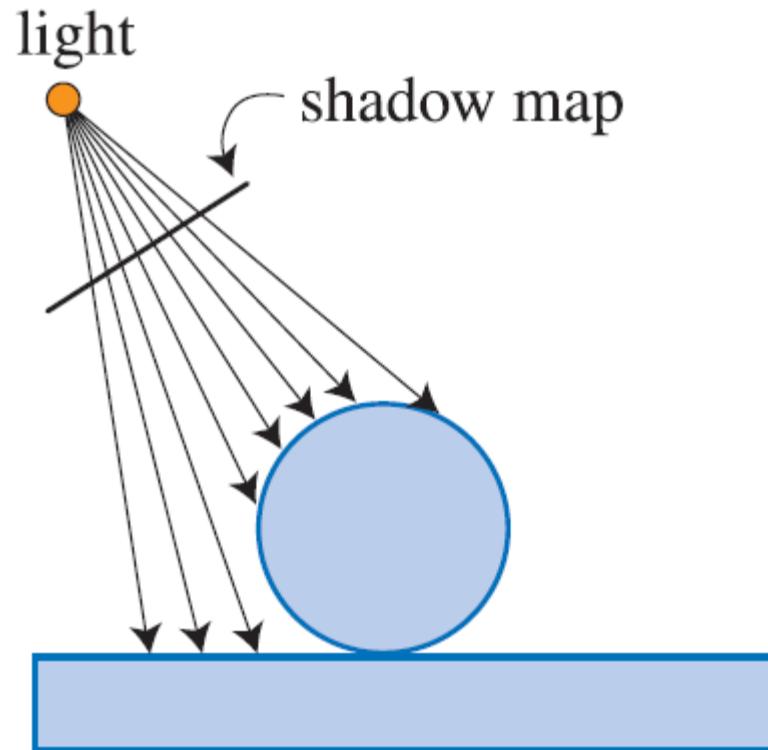


Coordinate Systems

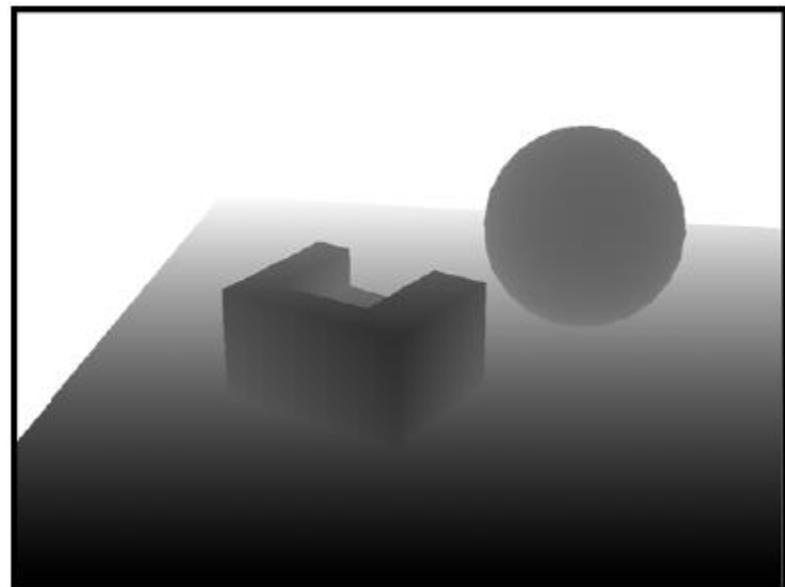
- in the second rendering pass, for each fragment, its position in the shadow map has to be determined
 - convert the 3D position of a fragment to object space
 - apply modelview transform of the light $L^{-1}M$
 - apply projective transform of the light P_{light}
 - homogenization and screen mapping
 - mapping to texture space
 - results in $(x', y', z', 1)^T$
 - $z' > \text{shadowMap}(x', y') \rightarrow$ point in shadow



Shadow Map Generation



scene is rendered from the position of the light source

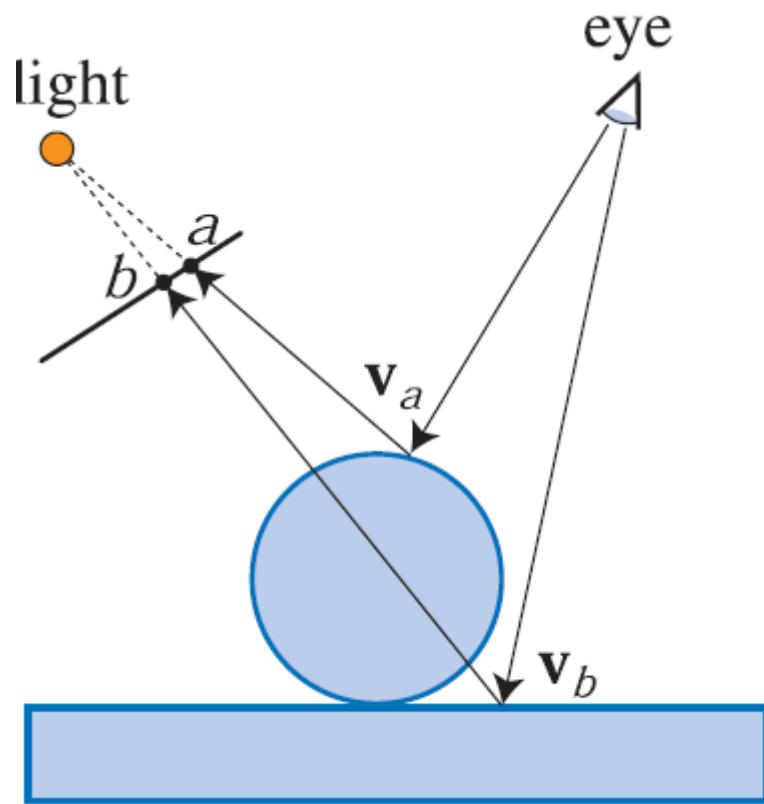


shadow map. a texture that represents distances of illuminated surface points to the light source.

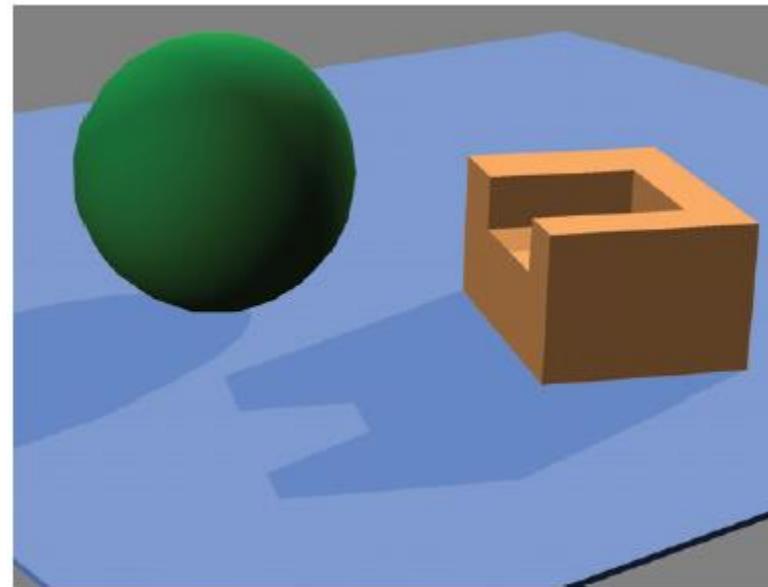
[Akenine-Moeller et al.: Real-time Rendering]

University of Freiburg – Computer Science Department – Computer Graphics - 14

Scene Rendering



in the second rendering pass, v_a and v_b are rendered. v_a is represented in the shadow map (illuminated). v_b is occluded and not represented in the shadow map (in shadow).



in a second rendering pass, the camera view is generated. For each fragment, it is tested whether it is represented in the shadow map or not.

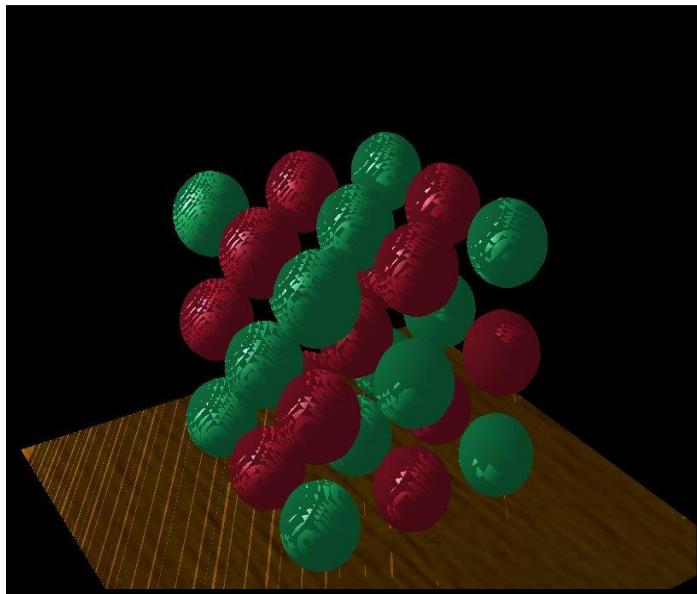
[Akenine-Moeller et al.: Real-time Rendering]

Algorithm

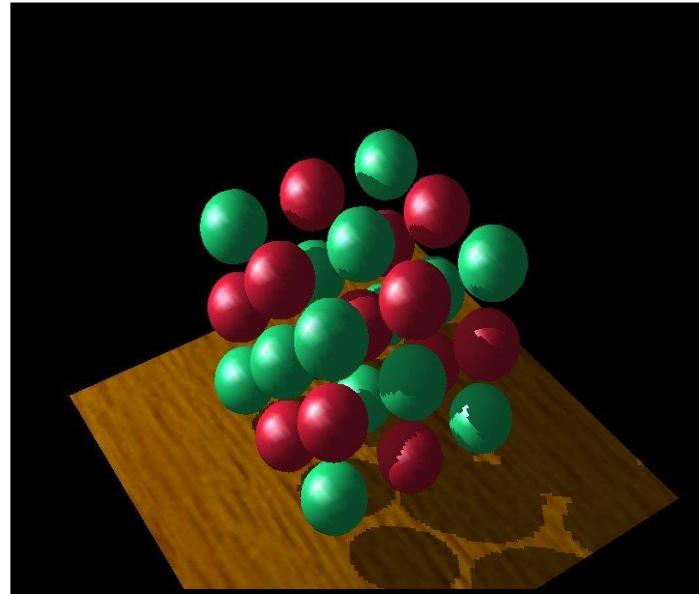
- use two depth buffers
 - the “usual” depth buffer for the view point
 - a second depth buffer for the light position (shadow map)
- render the scene from the light position into the shadow map
- render the scene from the view position into the depth buffer
 - transform depth buffer values to shadow map values
 - compare transformed depth buffer values and shadow map values to decide whether a fragment is shadowed or not

Aliasing

- discretized representation of depth values can cause an erroneous classification of scene points
- offset of shadow map values reduces aliasing artifacts



no offset



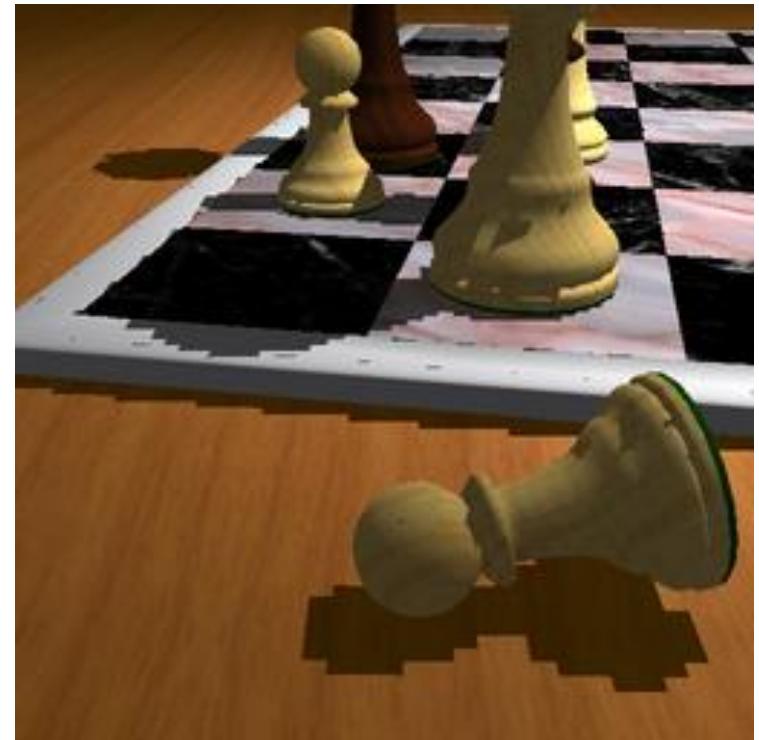
correct offset

[Mark J. Kilgard]

University of Freiburg – Computer Science Department – Computer Graphics - 17

Sampling

- in large scenes, sampling artifacts can occur
- uniform sampling of the shadow map can result in non-uniform resolution for shadows in a scene
- shadow map resolution tends to be too coarse for nearby objects and too high for distant objects

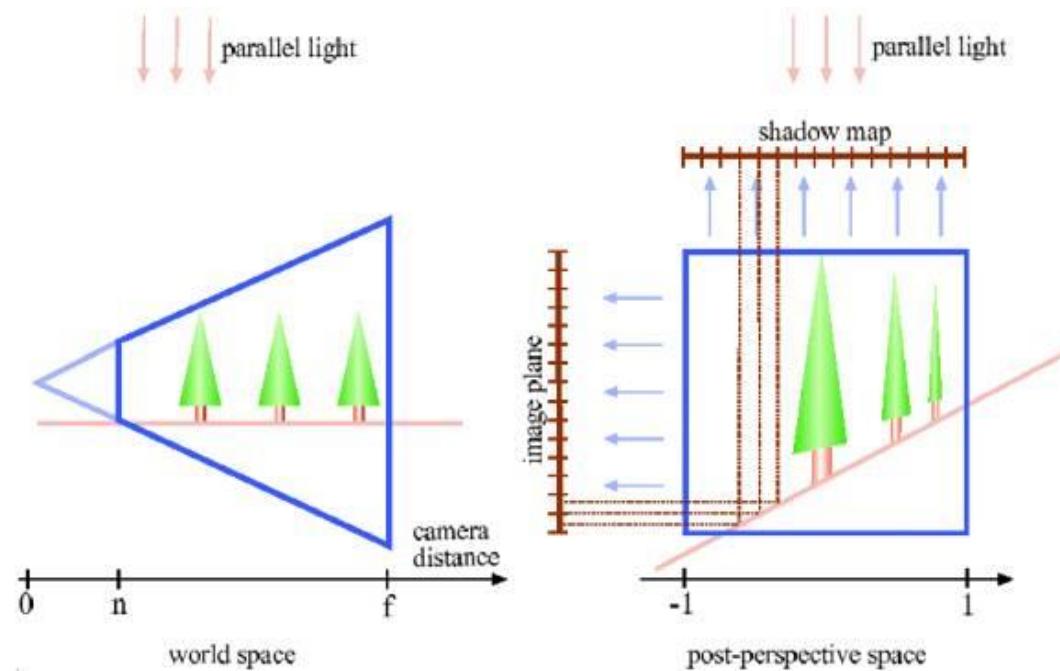


block artifacts

[Stamminger, Drettakis]

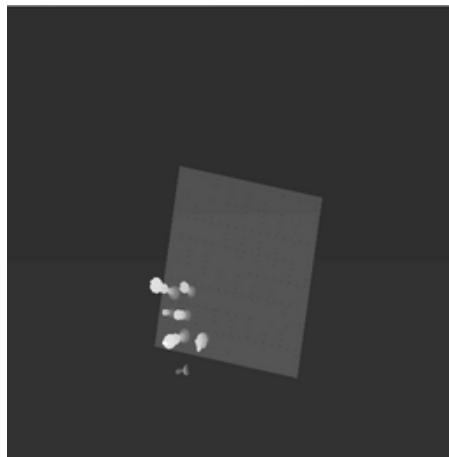
Sampling Perspective Shadow Maps

- scene and light are transformed using the projective transformation of the camera
- compute the shadow map in post-perspective space

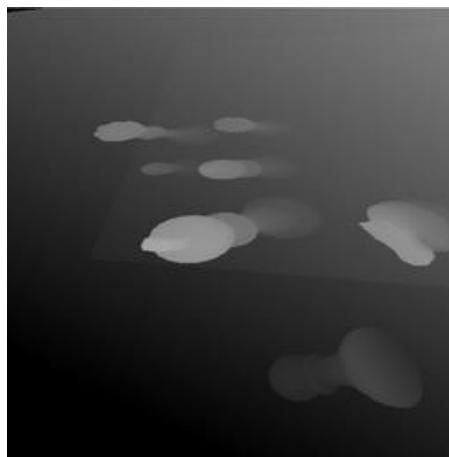


[Stamminger, Drettakis]

Sampling Perspective Shadow Maps



uniform shadow map



perspective shadow map



[Stamminger, Drettakis]

University of Freiburg – Computer Science Department – Computer Graphics - 20

Summary

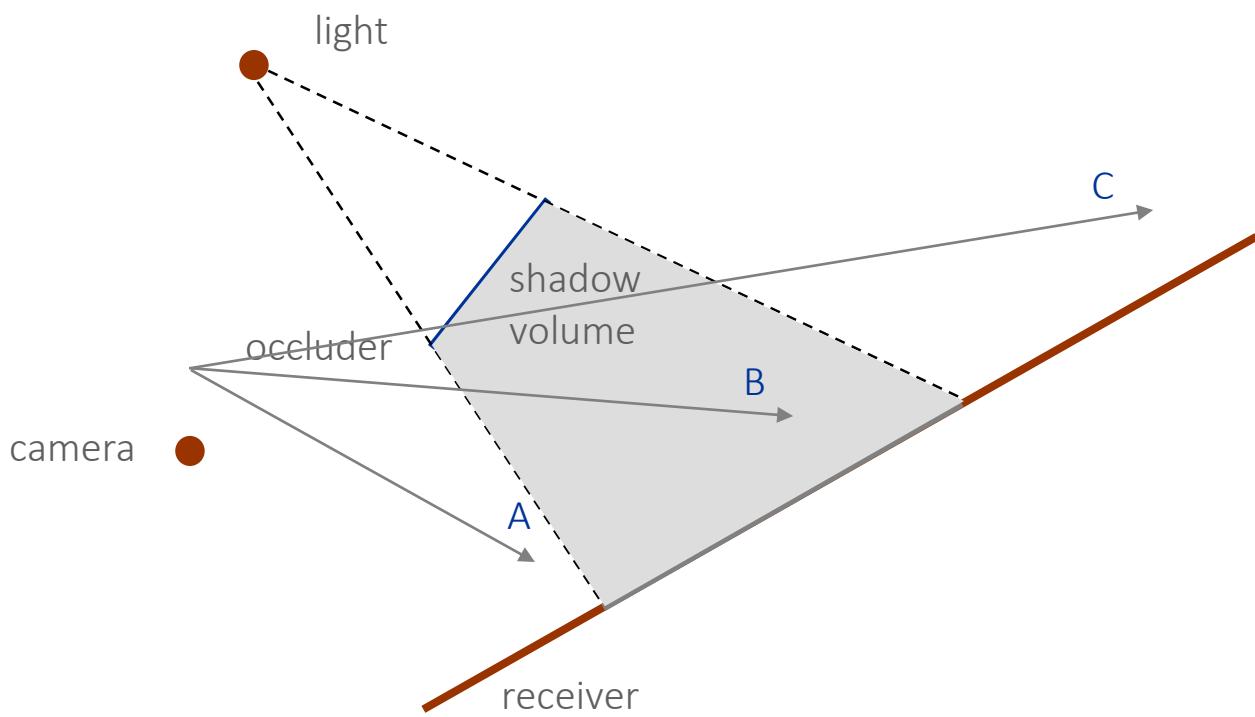
- image-space technique with two rendering passes
- no knowledge of the scene geometry is required
- works best for distant spot light sources
- light looks at the scene through
a single view frustum
 - scene geometry outside the view frustum is not handled
- aliasing artifacts and sampling issues

Outline

- introduction
- projection shadows
- shadow maps
- shadow volumes
- conclusion

Concept

- employ a polygonal representation of the shadow volume
- point-in-volume test



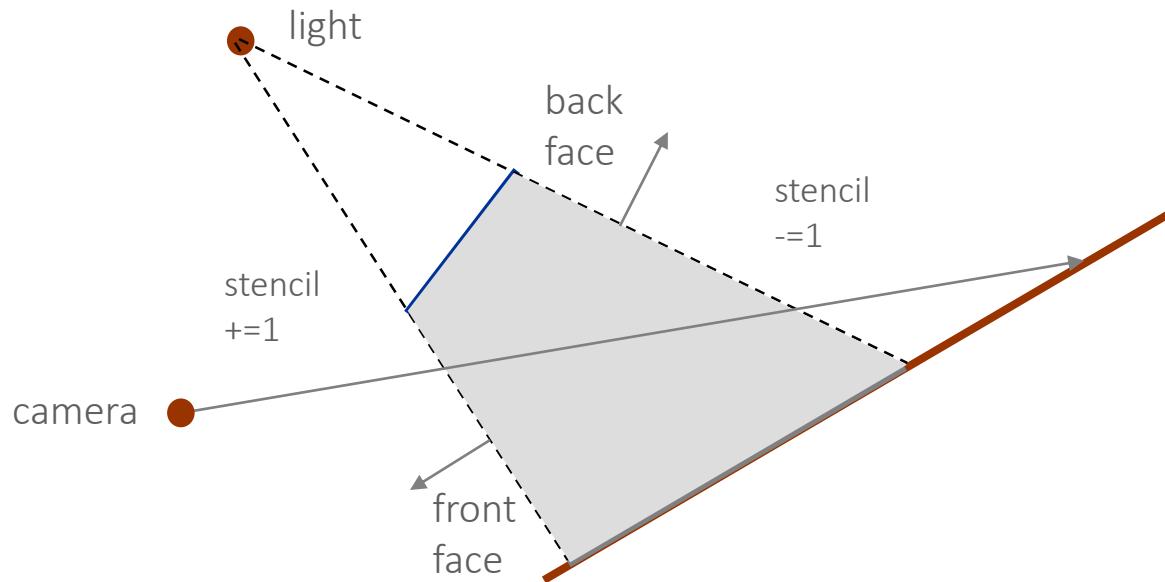
A is illuminated,
ray does not intersect
the shadow volume

B is in shadow,
ray enters the
shadow volume

C is illuminated,
ray enters and
leaves the
shadow volume

Implementation Issues

- classification of shadow volume polygons into front and back faces
 - rays enter the volume at front faces / leave it at back faces
- stencil buffer values count the number of intersected front and back faces



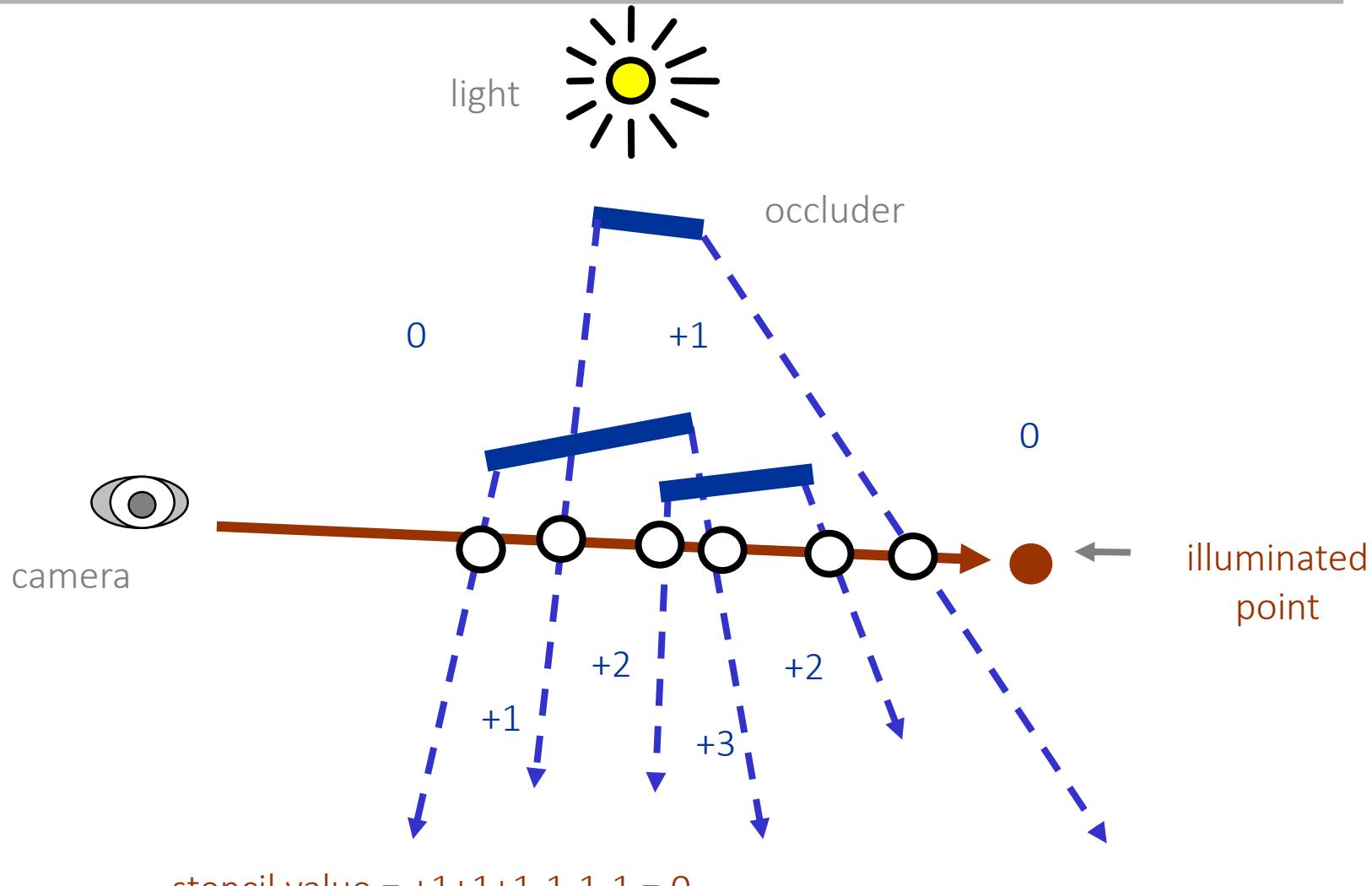
Algorithm (Z-pass)

- render scene to initialize depth buffer
 - depth values indicate the closest visible fragment
- stencil enter / leave counting approach
 - render shadow volume twice using face culling
 - render front faces and increment stencil when depth test passes (count occluding front faces)
 - render back faces and decrement stencil when depth test passes (count occluding back faces)
 - do not update depth and color
- finally,
 - if pixel is in shadow, stencil is non-zero
 - if pixel is illuminated, stencil is zero

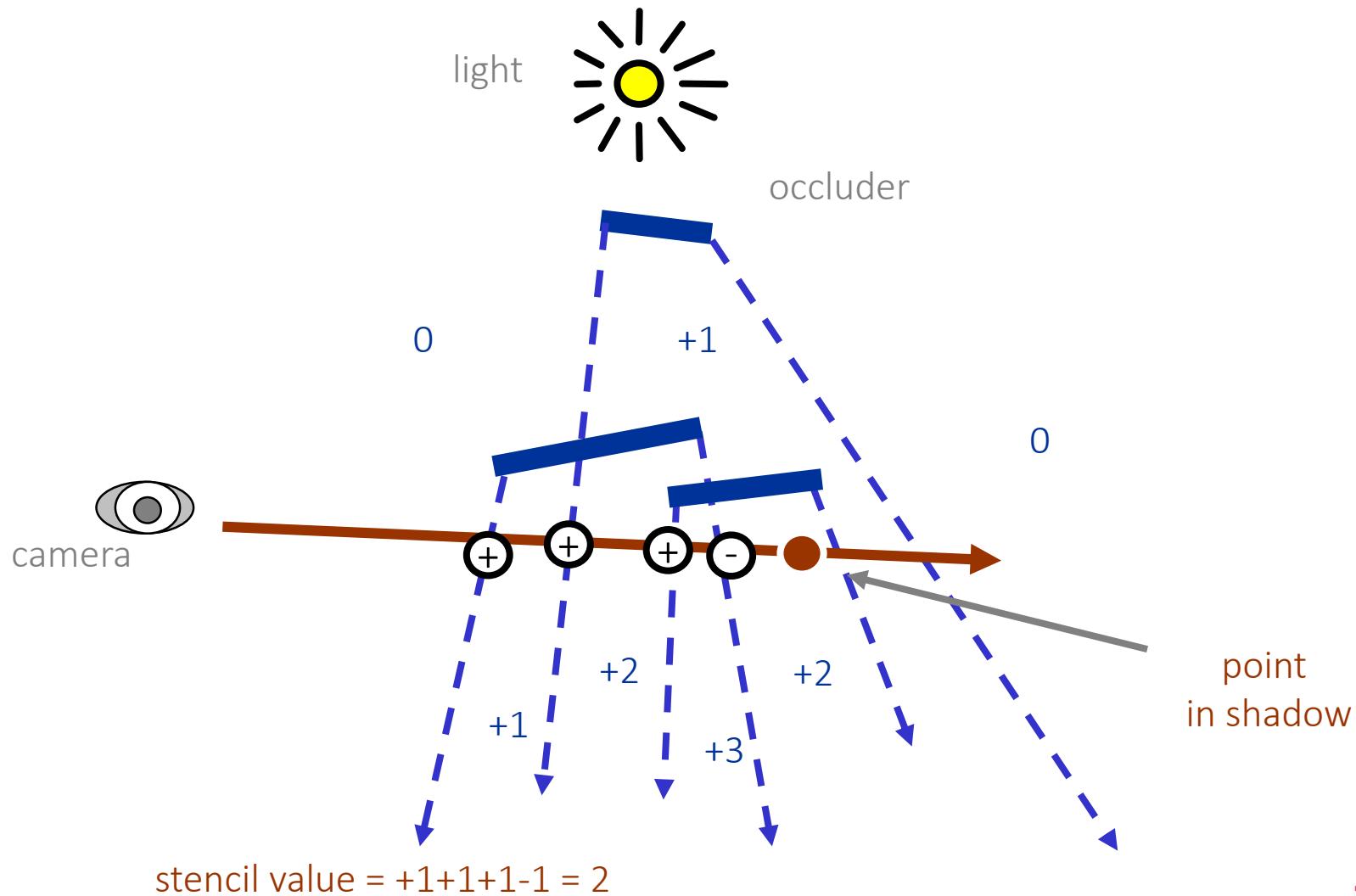
Implementation

- render the scene with only ambient lighting
- render front facing shadow volume polygons
(without depth and color update)
to determine how many front face polygons are
in front of the depth buffer pixels
- render back facing shadow volume polygons
(without depth and color update)
to determine how many back face polygons are
in front of the depth buffer pixels
- render the scene with full shading
where stencil is zero

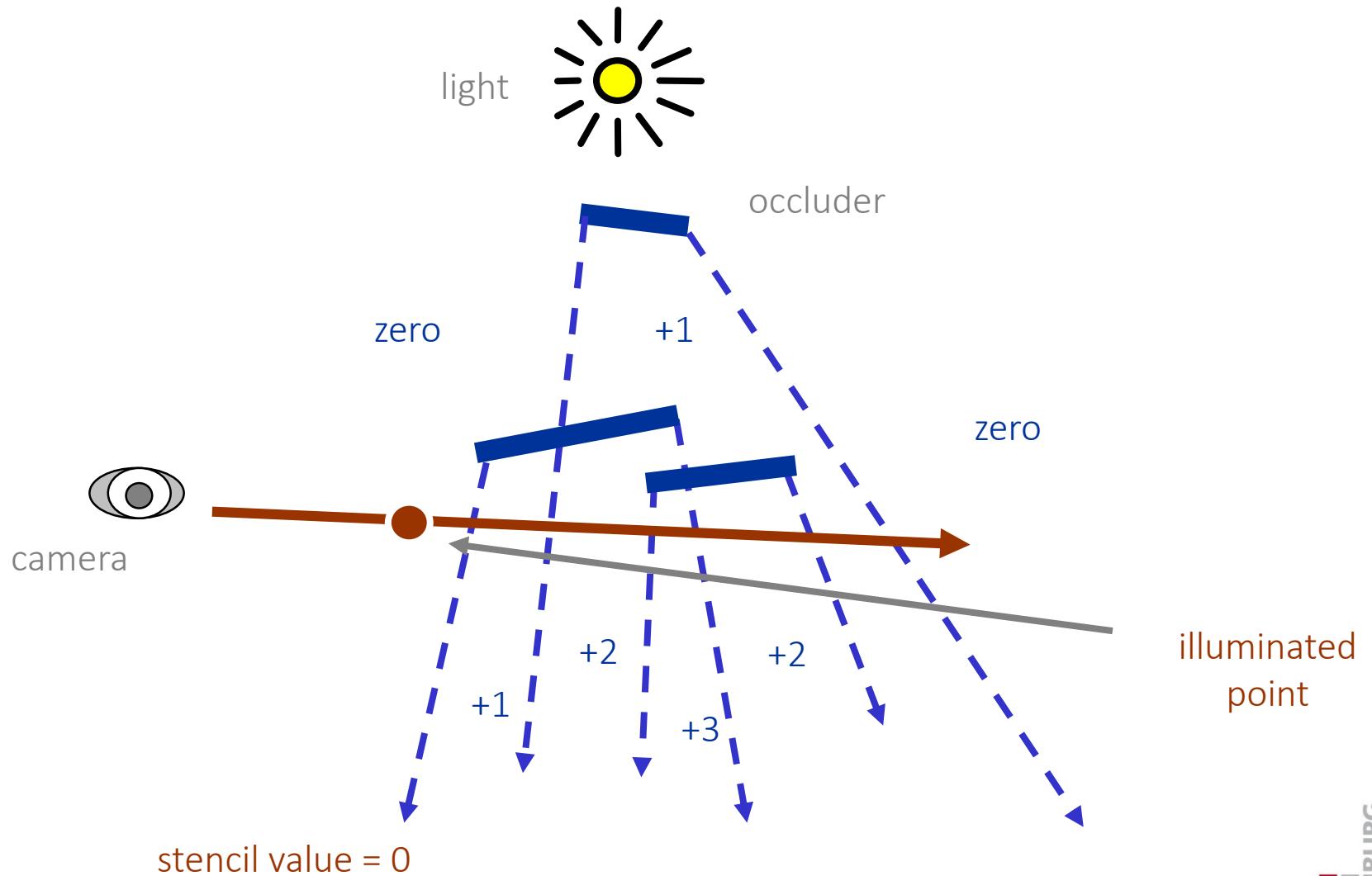
Example



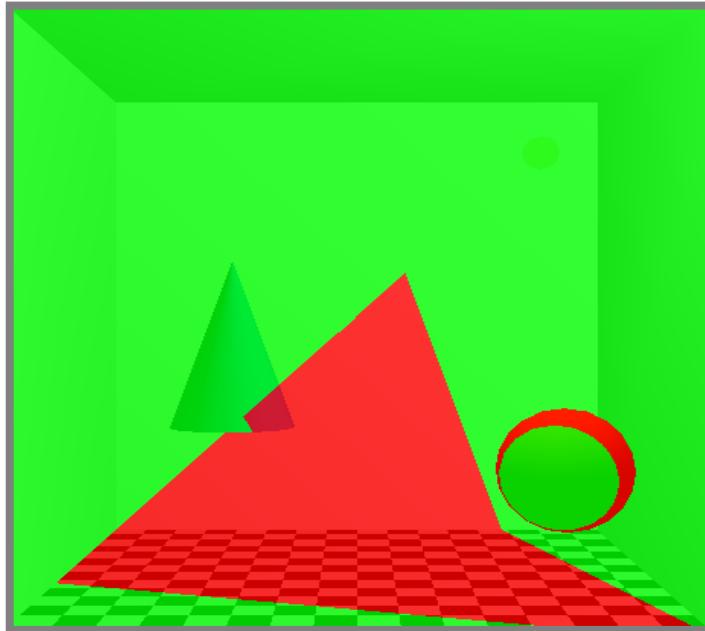
Example



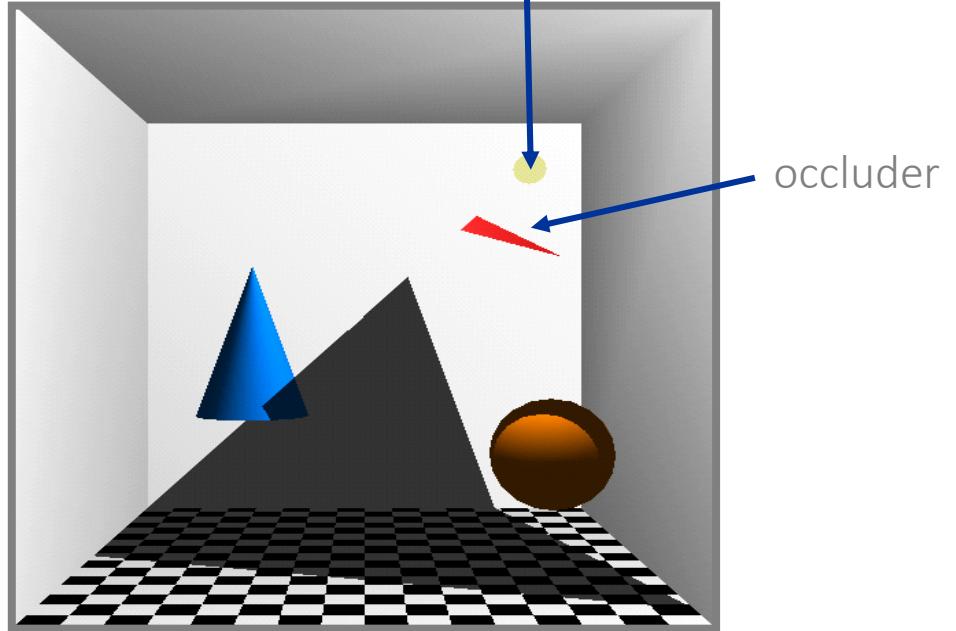
Example



Example



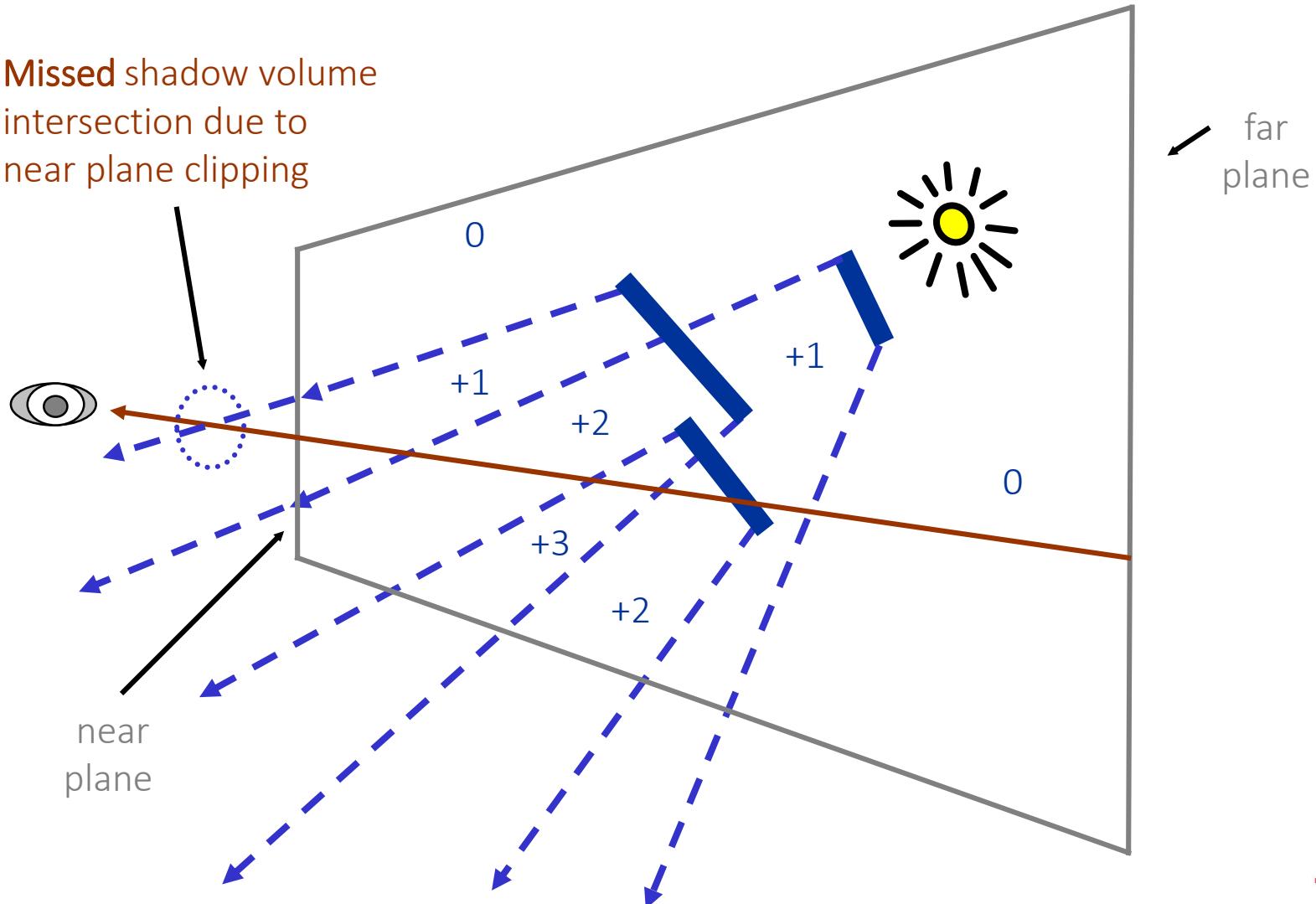
stencil value = 1
stencil value = 0



rendered scene

Missed Intersections

Missed shadow volume
intersection due to
near plane clipping



Solutions

- Z-fail
 - counts the difference of occluded front and back faces
 - misses intersections behind the far plane
- Z-fail with depth clamping
 - do not clip primitives at the far plane
 - clamp the actual depth value to the far plane
- Z-fail with far plane at infinity
 - adapt the perspective projection matrix

Algorithm (Z-fail)

- render scene to initialize depth buffer
 - depth values indicate the closest visible fragment
- stencil enter / leave counting approach
 - render shadow volume twice using face culling
 - render back faces and increment stencil when depth test fails (count occluded back faces)
 - render front faces and decrement stencil when depth test fails (count occluded front faces)
 - do not update depth and color
- finally,
 - if pixel is in shadow, stencil is non-zero
 - if pixel is illuminated, stencil is zero

Improving Z-fail

- depth clamping
 - do not clip primitives to the far plane
 - draw primitives with a maximum depth value instead
(clamp the actual depth value to the far plane)
- extend the shadow volume to infinity
- set the far plane to infinity (matrices in OpenGL form with negated values for n and f)

$$\left(\begin{array}{cccc} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{array} \right) f \rightarrow \infty \Rightarrow \left(\begin{array}{cccc} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -1 & -2n \\ 0 & 0 & -1 & 0 \end{array} \right)$$

Outline

- introduction
- projection shadows
- shadow maps
- shadow volumes
- conclusion

Summary

- projection shadows
 - restricted to planar receivers, no self-shadowing
- shadow maps
 - image-space technique, two rendering passes
 - works correct, if all relevant objects are "seen" by the light
 - sampling issues
- shadow volumes
 - requires a polygonal representation of the shadow volume
 - multiple rendering passes
 - clipping of shadow volume polygons has to be addressed

References

Shadow Maps

- Williams, "Casting Curved Shadows on Curved Surfaces", *SIGGRAPH*, 1978.
- Brabec, Annen, Seidel, "Practical Shadow Mapping", *Journal of Graphics Tools*, 2002.
- Stamminger, Drettakis, "Perspective Shadow Maps", *SIGGRAPH*, 2002.
- Wimmer, Scherzer, Purgathofer, "Light Space Perspective Shadow Maps", *Eurographics Symposium on Rendering*, 2004.

Shadow Volumes

- Crow, "Shadow Algorithms for Computer Graphics", *SIGGRAPH*, 1977.
- Heidmann, "Real Shadows, Real Time", *Iris Universe*, 1991.
- Diefenbach, "Multi-pass Pipeline Rendering: Interaction and Realism through Hardware Provisions", *PhD thesis, University of Pennsylvania*, 1996.
- McGuire, Hughes, Egan, Kilgard, Everitt, "Fast, Practical and Robust Shadows", *Technical Report, NVIDIA*, 2003.

Image Processing and Computer Graphics

Transparency and Reflection

Matthias Teschner

Computer Science Department
University of Freiburg

Albert-Ludwigs-Universität Freiburg

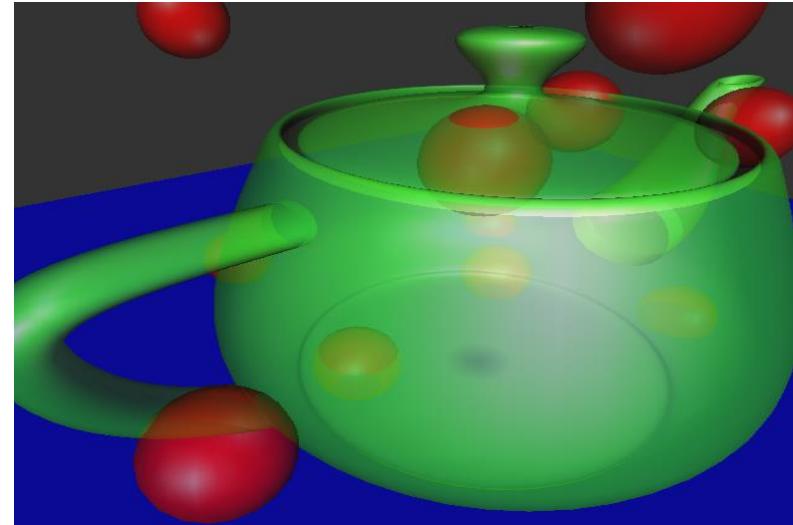


Outline

- transparency
- reflection

Introduction

- simplified transparency model
 - semitransparent objects are filters / attenuators of occluded objects
 - refraction and object thickness are neglected
- algorithms are based on
 - stipple patterns
 - color blending per pixel



[Cass Everitt: Interactive Order-Independent Transparency]

University of Freiburg – Computer Science Department – Computer Graphics - 3

Stipple Patterns

- screen-door transparency
- transparent object is rendered with a fill / stipple pattern, e.g. checkerboard (pattern of opaque and transparent fragments)
- limited number of fill patterns results in limited number of transparency levels
- aliasing artifacts
- simple method

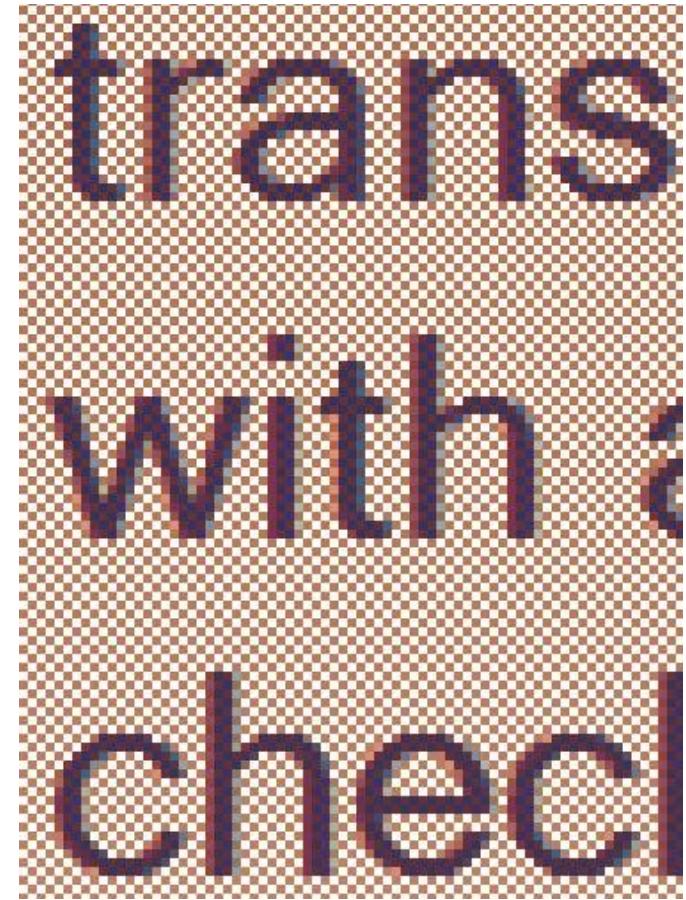
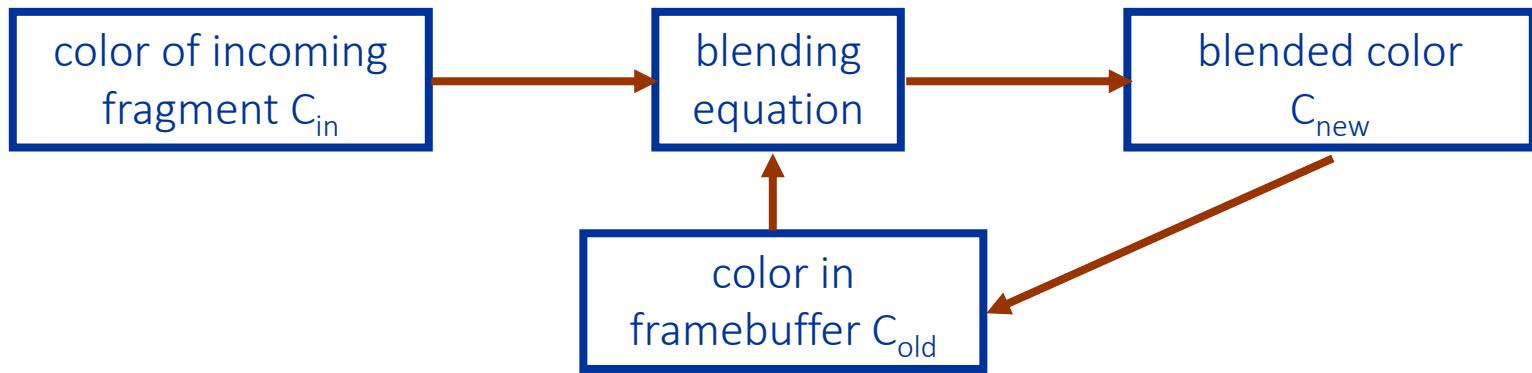


illustration of a stipple pattern

Color Blending

- combine fragment color with the framebuffer content



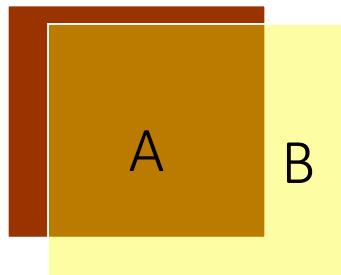
- color C_{old} is replaced by C_{new}
- blending equation: $C_{new} = \alpha_{in} \cdot C_{in} + \alpha_{old} \cdot C_{old}$

Color Blending

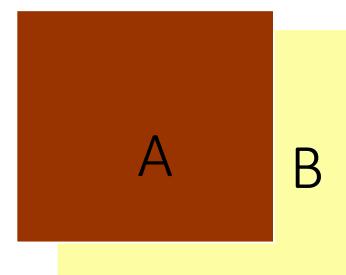
- alpha value
 - describes the opacity of a fragment,
1 - opaque, 0 - transparent
 - stored together with RGB color in a 4D vector (RGBA)
- blending equation for transparency
 - $\mathbf{C}_{\text{new}} = \alpha_{\text{in}} \cdot \mathbf{C}_{\text{in}} + (1 - \alpha_{\text{in}}) \cdot \mathbf{C}_{\text{old}}$
 - over operator
 - $\alpha_{\text{in}} = 0$ - \mathbf{C}_{old} is not changed
 - $\alpha_{\text{in}} = 1$ - \mathbf{C}_{old} is replaced by \mathbf{C}_{in}
 - $0 < \alpha_{\text{in}} < 1$ - \mathbf{C}_{old} is replaced by a mix of \mathbf{C}_{in} and \mathbf{C}_{old}
 - only the alpha value of the incoming fragment matters

Color Blending

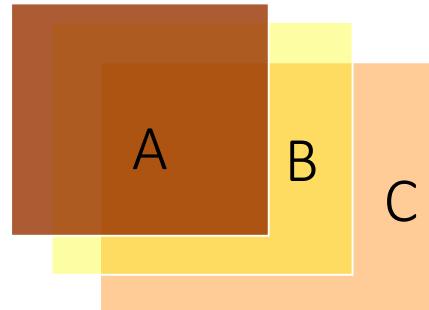
- order matters



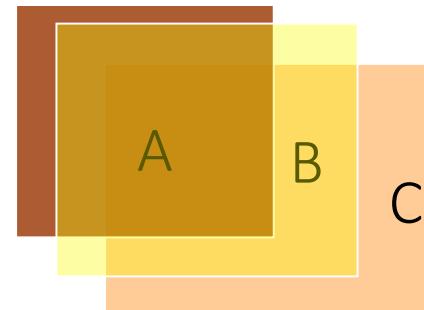
B over A



A over B



A over B over C



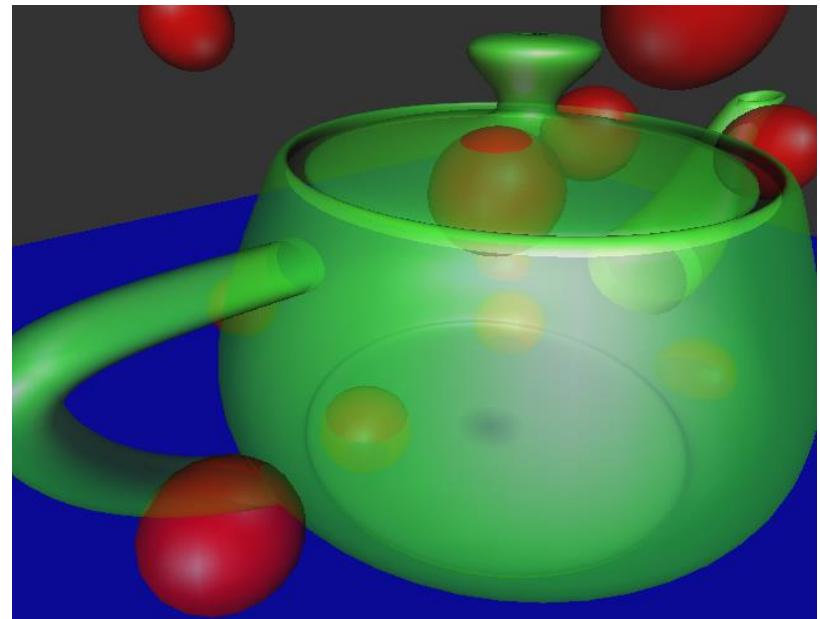
B over A over C

Outline

- transparency
 - depth ordering
 - binary space partitioning
 - depth peeling
- reflection

Depth Ordering

- polygons / fragments have to be rendered in sorted depth order
- hardware generally renders in object order
- depth test only returns the nearest fragment per pixel, sorting is not realized
- intersecting polygons have to be handled
- dynamic scenes require re-sorting



[Cass Everitt: Interactive Order-Independent Transparency]

Depth Ordering for Convex Objects

- exactly two depth layers for arbitrary viewing directions
- first depth layer defined by front faces
- second depth layer defined by back faces
- algorithm
 - render back faces in a first pass
 - blend with front faces in a second pass

Depth Ordering for Arbitrarily Shaped Objects

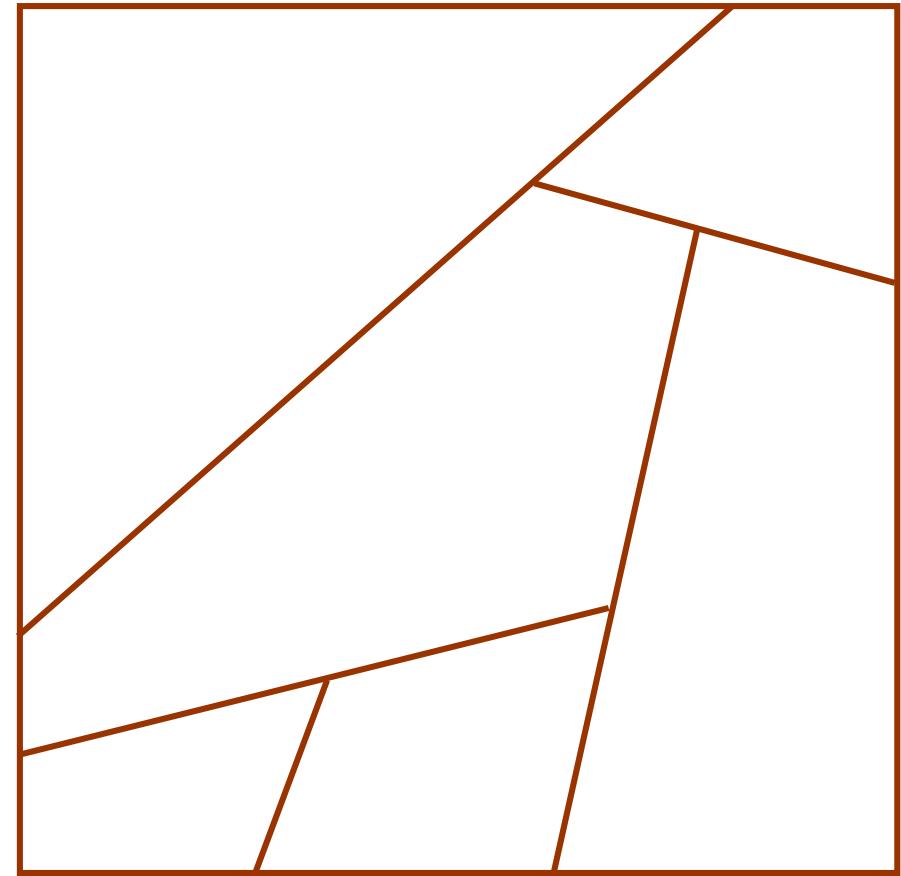
- object-space methods
 - use pre-computed spatial data structures
 - e.g., binary space partition tree (BSP tree)
 - useful for static geometry
 - varying viewer positions and orientations can be handled
- screen-space methods
 - employ the functionality of the rendering pipeline
 - several rendering passes compute depth layers
 - final pass renders the ordered depth layers
 - useful for dynamic / deforming geometry and arbitrary views
 - no pre-computation is required / can be employed

Outline

- transparency
 - depth ordering
 - binary space partitioning
 - depth peeling
- reflection

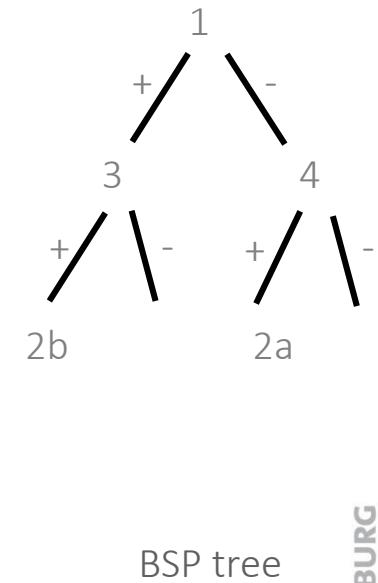
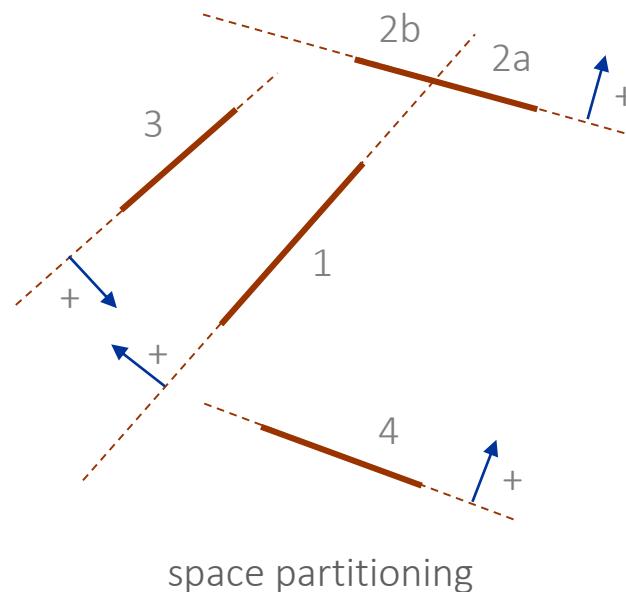
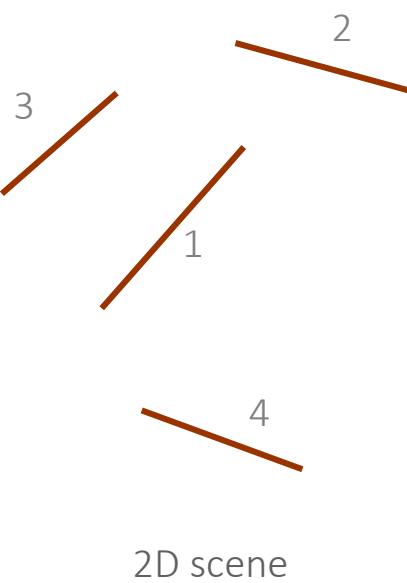
Binary Space Partitioning BSP

- BSP tree is a hierarchical spatial data structure
- 3D space is subdivided by means of arbitrarily oriented planes
- nodes represent planes
- leaves represent convex space cells
- applications
 - visible surface algorithm
 - depth sorting
 - collision detection



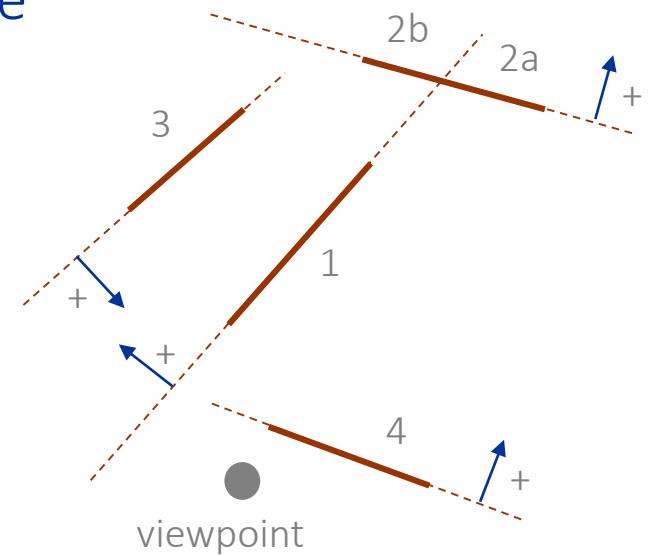
Generation of the BSP Tree

- the BSP tree is pre-computed for static scenes
- all planar primitives are represented in the tree
- balancing is less important, as the entire tree has to be queried (all primitives are rendered)



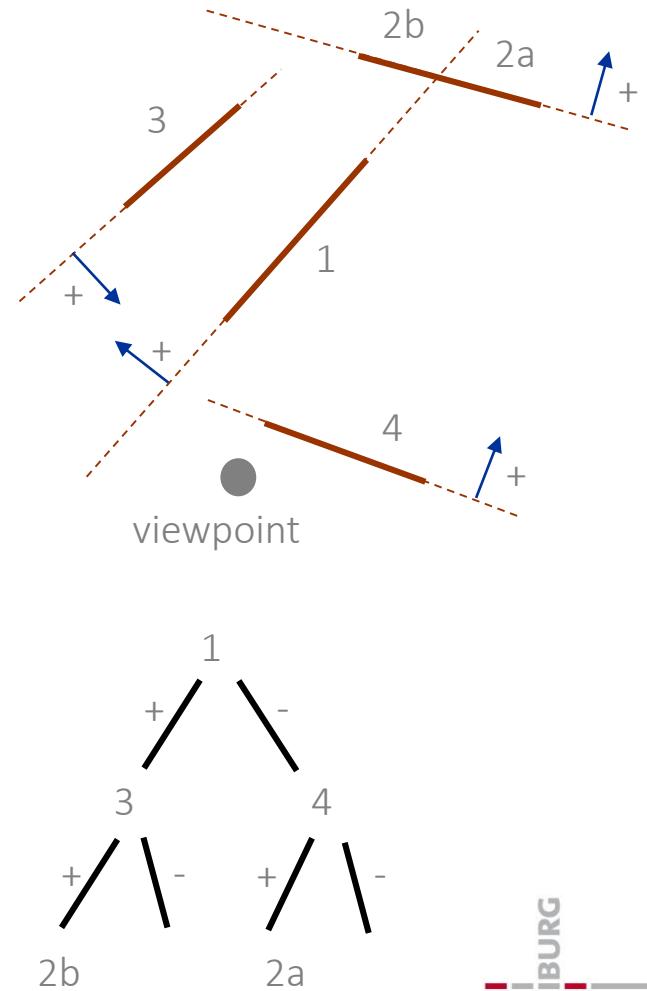
Query of the BSP Tree

- motivation
 - a viewer is on the near side of a plane
 - a polygon on the far side of this plane cannot occlude the plane or any polygon on the near side
- back to front rendering
 - render far branch of the viewpoint
 - render root (node) polygon
 - render near branch of the viewpoint
 - recursively applied to sub-trees



Query of the BSP Tree

- back to front rendering
- viewpoint is in 1-
- rendering of 1+, 1, 1-
- rule recursively applied to 1+ and 1-
- viewpoint is in 3+
 - rendering of 3, 2b
- viewpoint is in 4-
 - rendering of 2a, 4



BSP Tree - Discussion

- not only visible surface generation,
but depth sorting of all primitives
per pixel position
- additional data structure
- can be pre-computed
- requires polygon splits
- dynamic scenes require an
update of the data structure

Outline

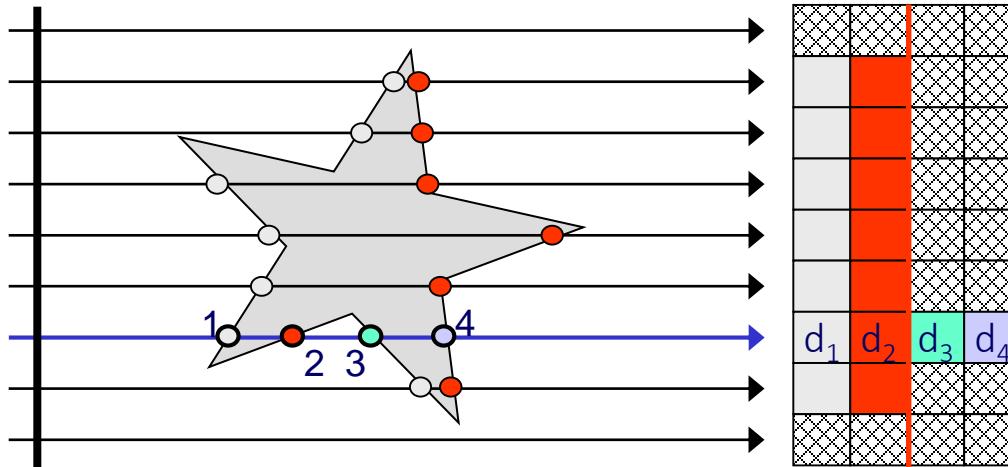
- transparency
 - depth ordering
 - binary space partitioning
 - depth peeling
- reflection

Concept

- motivation
 - use the functionality of the rendering pipeline
 - several rendering passes compute depth layers
 - final pass renders the ordered depth layers
 - useful for dynamic / deforming geometry
 - no pre-computation is required / can be employed
- algorithm
 - first render pass gives the front-most fragment color / depth
 - each successive render pass extracts the fragment (with color and depth) for the next-nearest fragment on a per pixel basis (screen-space approach)
 - two depth buffers are used

Concept

- object is rendered once for each depth layer
 - depth complexity is the max number of layers per pixel position
- two separate depth tests per fragment
 - must be farther than the one in the previous layer (d_1)
 - must be the nearest of all remaining fragments (d_2, d_3, d_4)



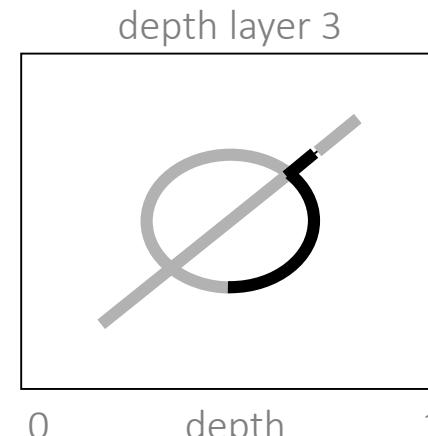
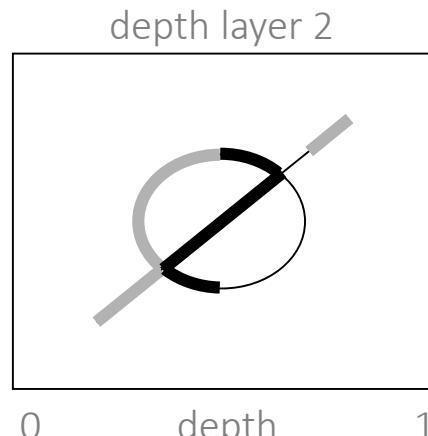
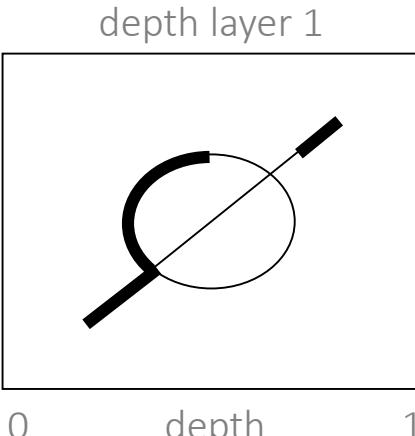
[Bruno Heidelberger]

University of Freiburg – Computer Science Department – Computer Graphics - 20

Depth Layers - 2D

- depth peeling strips away one depth layer with each successive rendering pass
- illustration
 - bold black lines - frontmost (leftmost) surfaces
 - thin black lines – hidden surfaces
 - light grey lines – “peeled away” surfaces

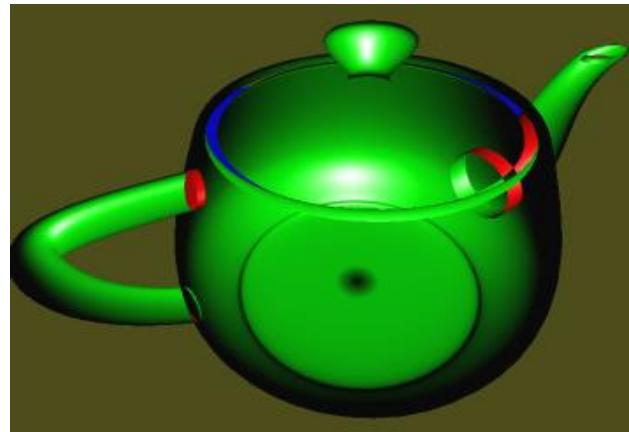
[Cass Everitt:
Interactive
Order-Independent
Transparency]



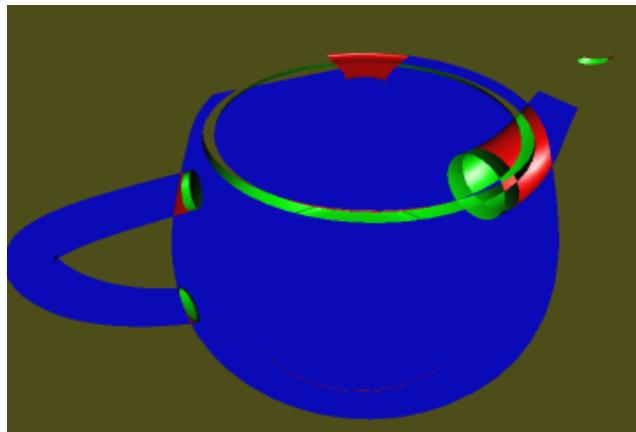
Depth Layers - 3D



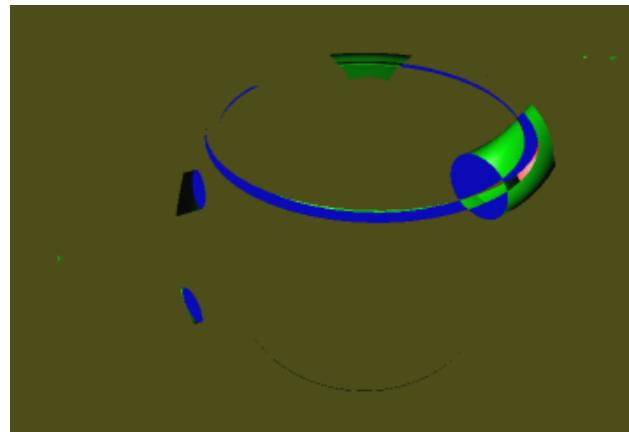
depth layer 1



depth layer 2



depth layer 3



depth layer 4

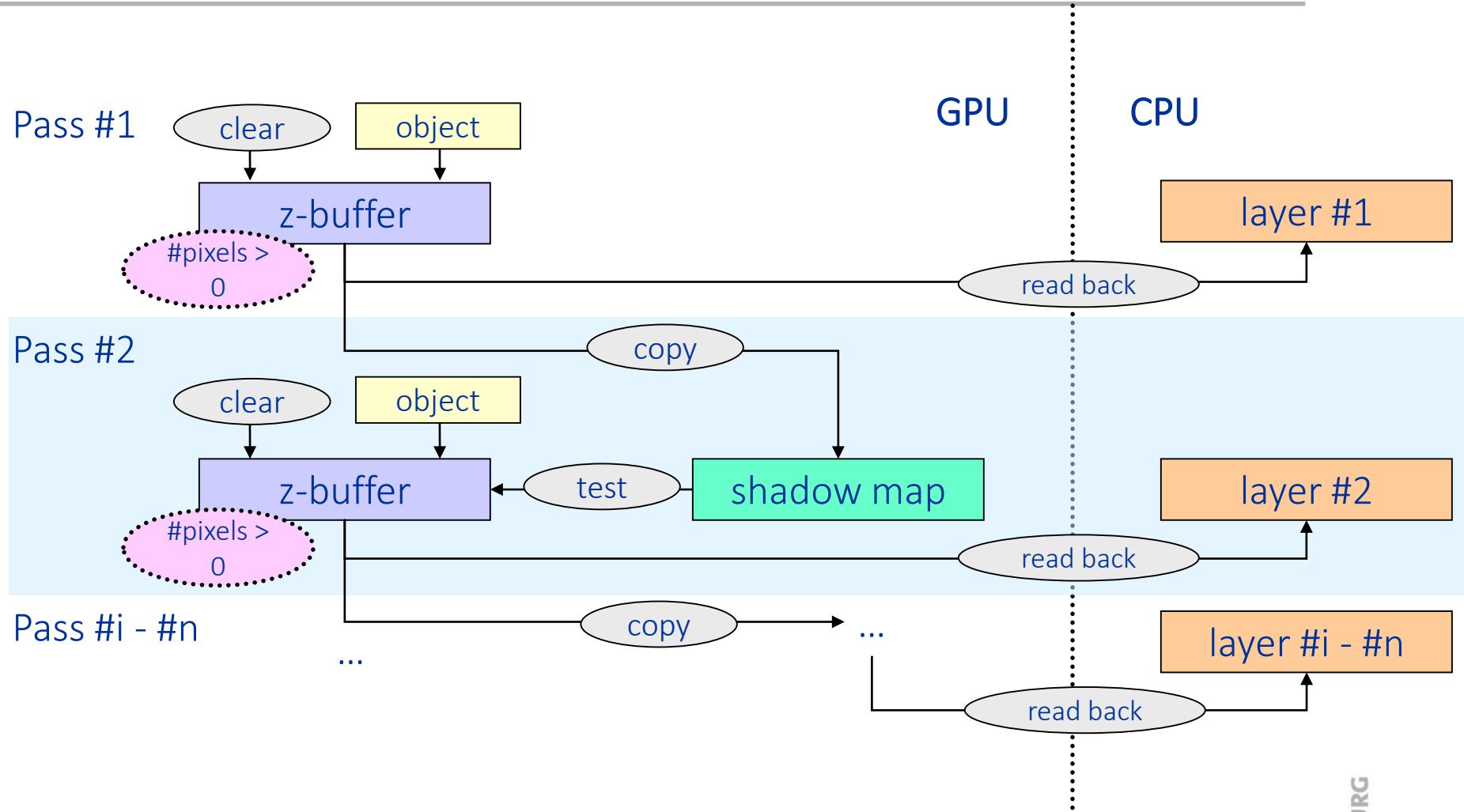
[Cass Everitt: Interactive Order-Independent Transparency]

University of Freiburg – Computer Science Department – Computer Graphics - 22

Implementation Based on Shadow Mapping

- two depth tests (depth buffers) are required
- e.g., shadow mapping can be used to realize a second depth buffer
- in contrast to the depth buffer, the shadow map
 - is not tied to the camera position
 - is not writeable during depth test
 - does not discard fragments
- with respect to depth peeling
 - the shadow map is tied to the camera position
 - copy functionality to depth buffer is employed

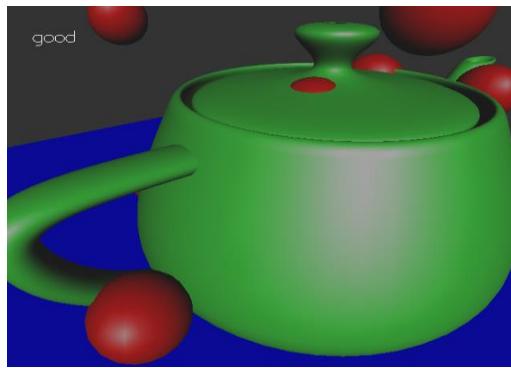
Implementation Based on Shadow Mapping



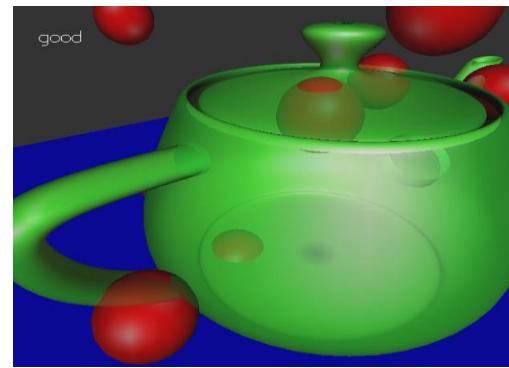
[Bruno Heidelberger]

Results

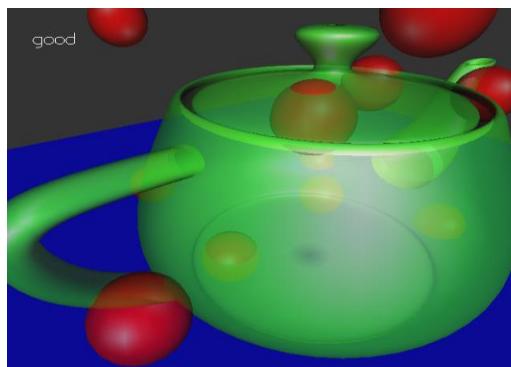
- quality and performance are determined by the number of generated depth layers



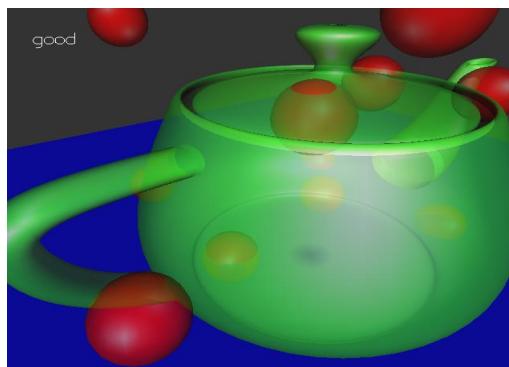
one
depth
layer



two
depth
layers



three
depth
layers



four
depth
layers

[Cass Everitt: Interactive Order-Independent Transparency]

University of Freiburg – Computer Science Department – Computer Graphics - 25

Depth Peeling - Discussion

- screen-space algorithm
- multiple rendering passes generate depth layers per pixel position
- view dependent (in contrast to the BSP approach)
- appropriate for dynamic scenes
- quality and performance are determined by the number of rendering passes (in the discussed implementation)

Transparency - Summary

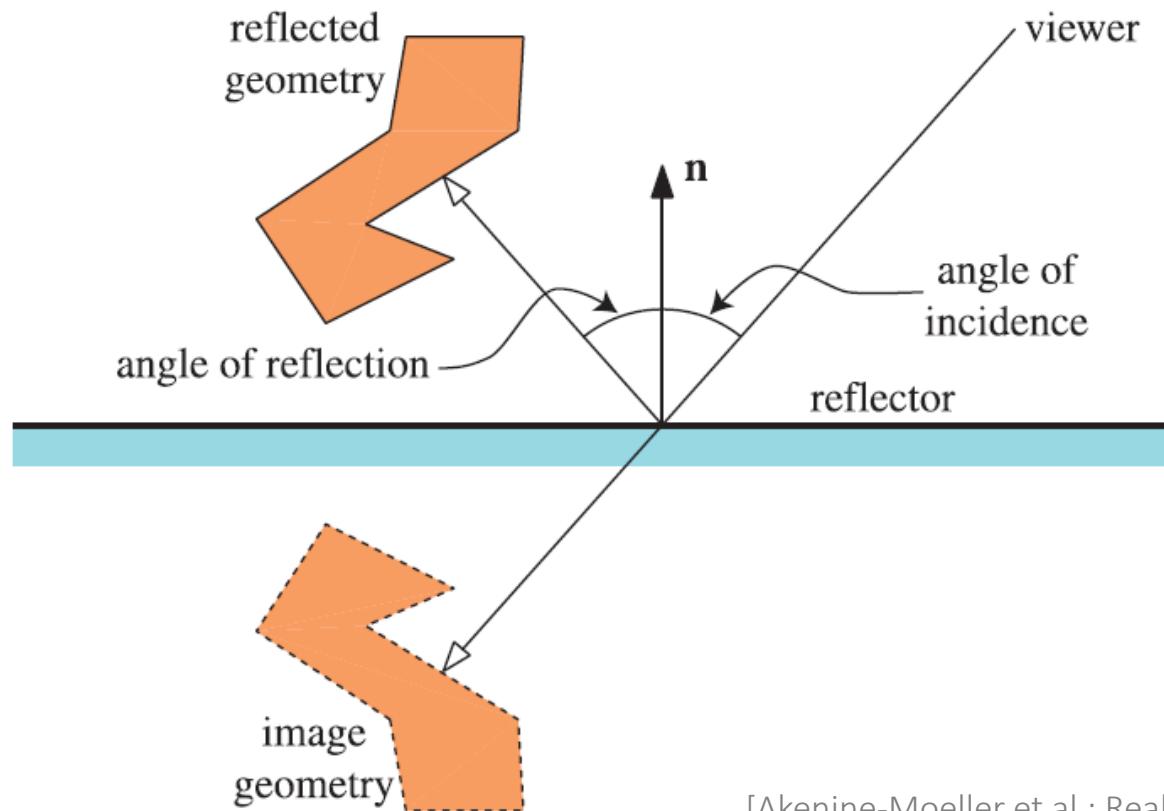
- simplified transparency model
- algorithms based on
 - stipple patterns
 - color blending
- for blending, depth-sorted primitives are required
- BSP tree
 - object space algorithm with one rendering pass
 - appropriate for static scenes
- depth peeling
 - screen space algorithm with multiple rendering passes
 - appropriate for dynamic scenes

Outline

- transparency
- reflection
 - planar surfaces
 - arbitrary surfaces

Law of Reflection

- angle of incidence is equal to the angle of reflection



[Akenine-Moeller et al.: Real-time Rendering]

Generation of Reflected Geometry

- original and reflected geometry is rendered
- reflected geometry is generated with respect to the reflection plane with surface normal $\mathbf{n} = (n_x, n_y, n_z)$ and a point \mathbf{p} on the reflection plane

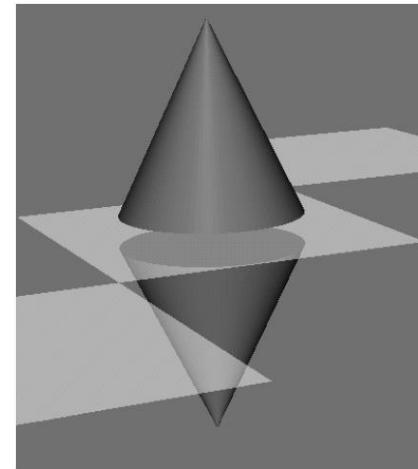
$$\mathbf{M}_{(\mathbf{n}, \mathbf{p})} = \begin{pmatrix} 1 - 2n_x^2 & -2n_x n_y & -2n_x n_z & 2n_x(\mathbf{n} \cdot \mathbf{p}) \\ -2n_x n_y & 1 - 2n_y^2 & -2n_y n_z & 2n_y(\mathbf{n} \cdot \mathbf{p}) \\ -2n_x n_z & -2n_y n_z & 1 - 2n_z^2 & 2n_z(\mathbf{n} \cdot \mathbf{p}) \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

e.g.

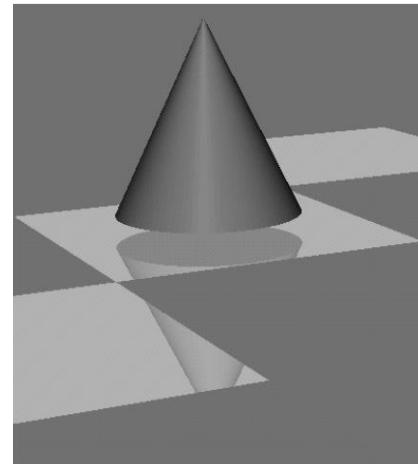
$$\mathbf{M}_{((0,1,0),(0,0,0))} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Implementation

- render reflected geometry with reflected illumination
 - render semi-transparent reflection plane, e.g. with color blending
 - render original geometry
-
- render reflection plane to stencil
 - render reflected geometry where stencil is set
 - ...



reflection rendering without stenciling



reflection rendering with stenciling

[Akenine-Moeller et al.: Real-time Rendering]

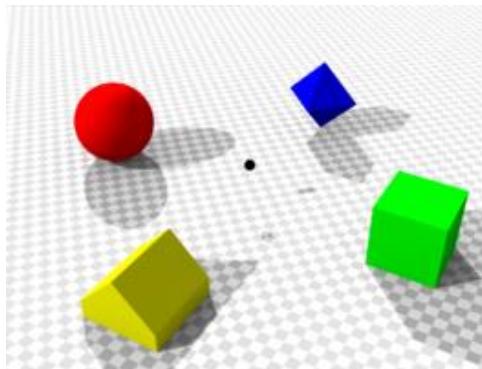
University of Freiburg – Computer Science Department – Computer Graphics - 31

Outline

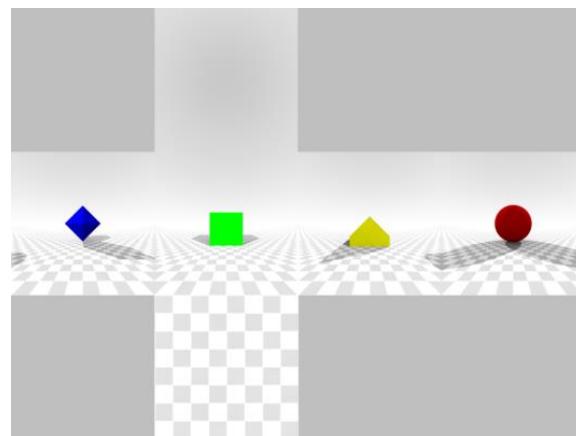
- transparency
- reflection
 - planar surfaces
 - arbitrary surfaces

Environment Mapping

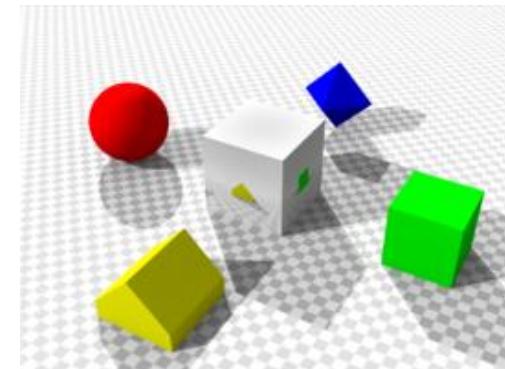
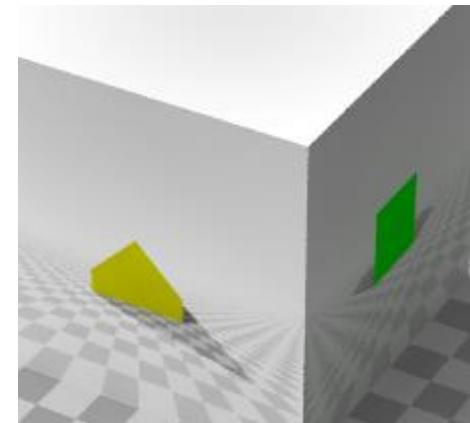
- e.g. cube mapping
- approximates reflections of the environment on arbitrary surfaces



place a viewer in a scene



generate the environment
texture from six view directions

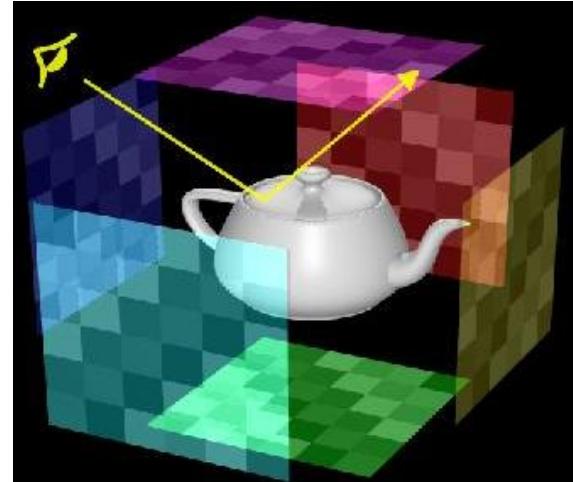


apply the texture
to an object at the
position of the viewer

[Wikipedia: Cube Mapping]

Environment Mapping

- environment is projected onto an object-embedding shape, e. g. sphere or cube
- view-dependent mapping
 - dependent on viewing and reflection direction
- approximate implementation of reflections off arbitrary surfaces

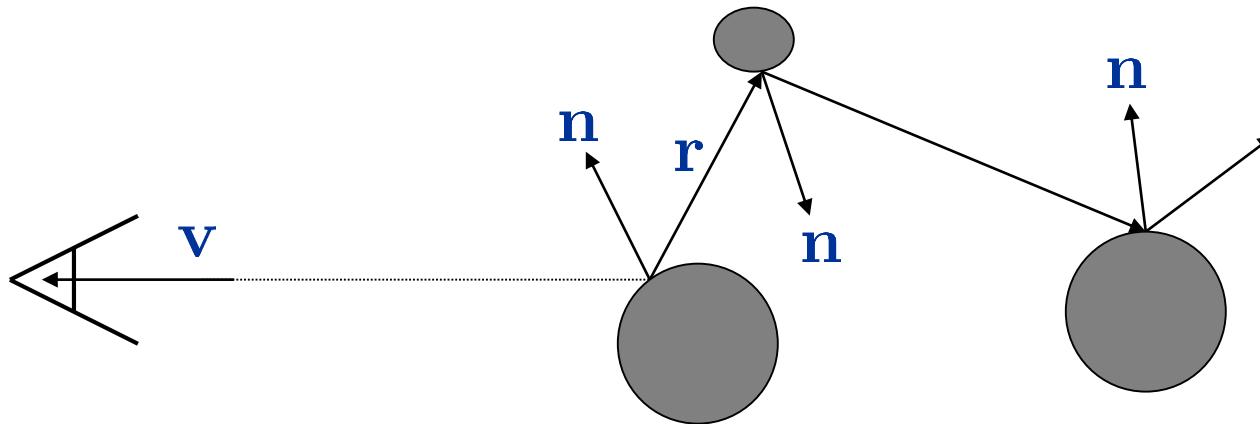


[Rosalee Wolfe]

Environment Mapping

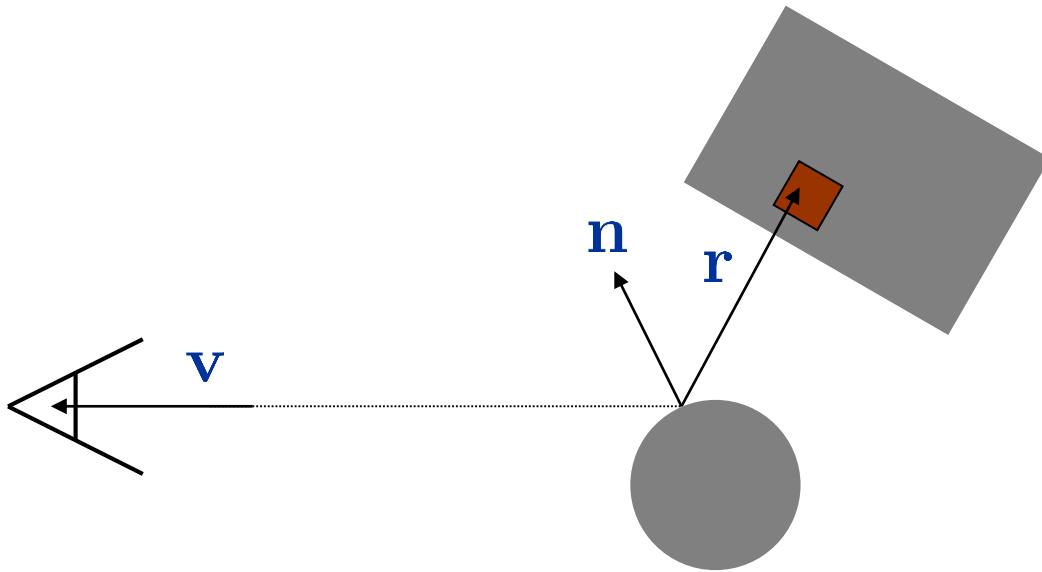
Motivation

- the Phong illumination model (a local model) does not take into account reflections
- Raytracing (a global model) traces rays off the object into the world to obtain reflections



Environment Mapping

- environment mapping approximates this process by capturing the environment in a texture map and using the reflection vector to index into this map
- cannot handle changing reflections of moving objects

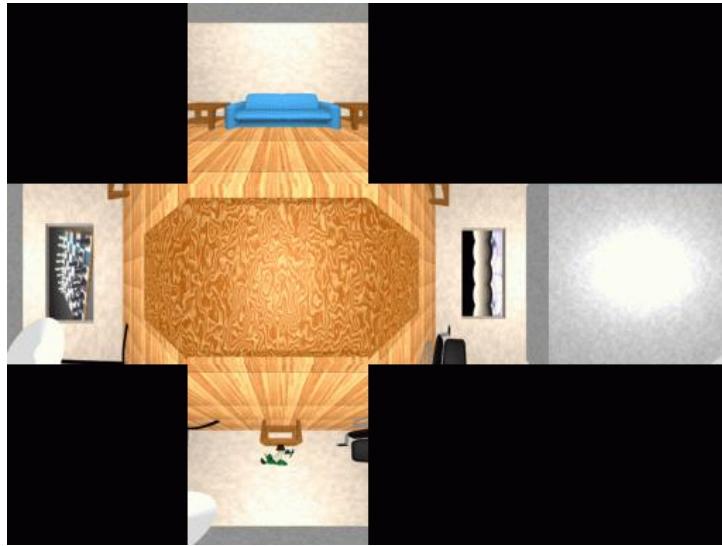


Environment Mapping - Steps

- generate or load a 2D map of the environment
- for each fragment of a reflective object,
compute the normal \mathbf{n}
- compute the reflection vector \mathbf{r} from the view vector \mathbf{v}
and the normal \mathbf{n} at the surface point
- use the reflection vector to compute an index into the
environment map that represents the objects in the
reflection direction
- use the texel data (texture value) from the
environment map to color the current fragment

Cubic Environment Mapping

- the map is constructed by placing a camera at the center of the object and taking pictures in 6 directions

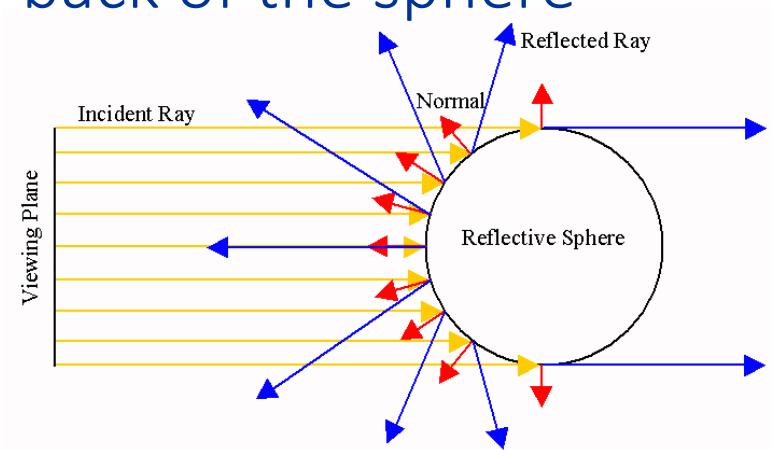


[Mizutani, Reindel: Environment Mapping Algorithms, Reindel Software]

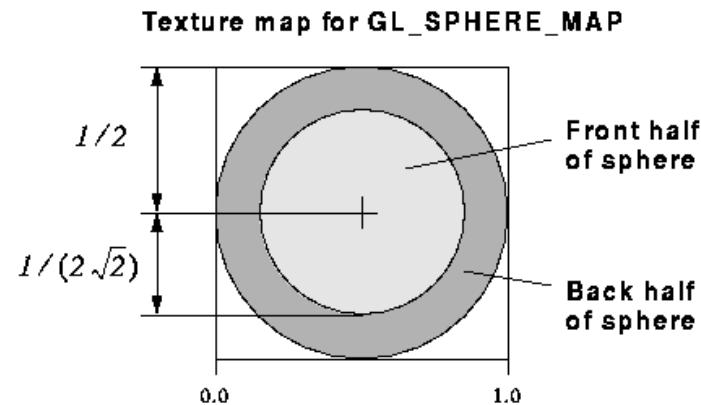
University of Freiburg – Computer Science Department – Computer Graphics - 38

Spherical Environment Mapping

- the map is obtained by orthographically projecting an image of a mirrored sphere
- map stores colors seen by reflected rays
- sphere map contains information about both, the environment in front of the sphere and in back of the sphere



[Mizutani, Reindel: Environment Mapping Algorithms, Reindel Software]



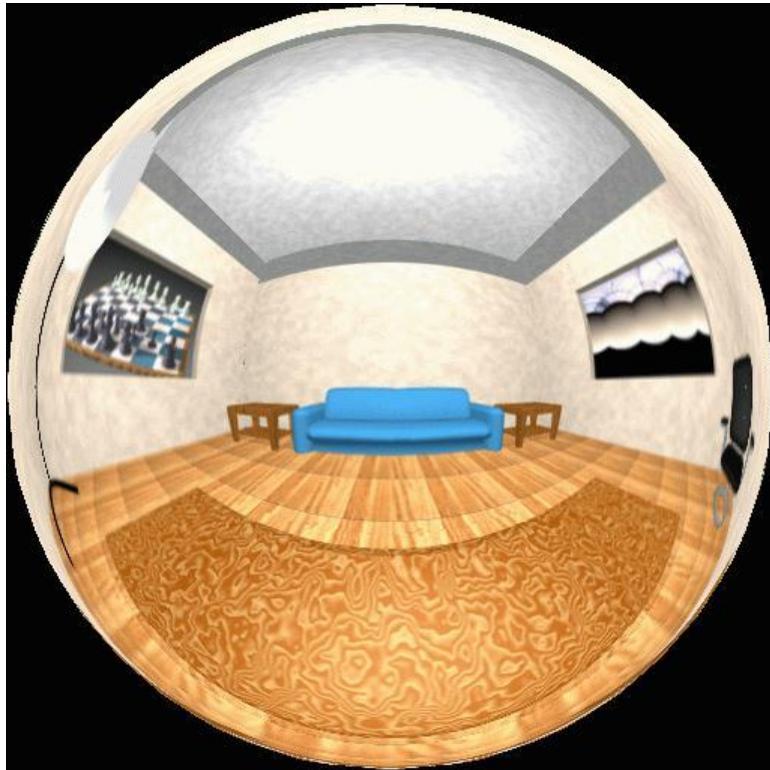
Spherical Environment Mapping

- the map can be obtained from a synthetic scene by
 - Raytracing
 - warping automatically generated cubic maps
- the map can be obtained from the real world by
 - photographing an actual mirrored sphere



[<http://www.oakcorp.net/chaos/hdri.shtml>]

Spherical Environment Mapping



[Mizutani, Reindel: Environment Mapping Algorithms, Reindel Software]

University of Freiburg – Computer Science Department – Computer Graphics - 41

Spherical Environment Mapping

- to map the reflection vector to the sphere map, the following equations are used based on the reflection vector \mathbf{r}
- in contrast to cube mapping, a generalized equation can be used

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \frac{r_x}{2\sqrt{r_x^2+r_y^2+(r_z+1)^2}} + \frac{1}{2} \\ \frac{r_y}{2\sqrt{r_x^2+r_y^2+(r_z+1)^2}} + \frac{1}{2} \end{pmatrix}$$

Spherical Environment Mapping

- disadvantages
 - maps are hard to create on the fly
 - sampling is non-linear, non-uniform
 - sampling is view-point dependent
- advantages
 - no interpolation across map seems

Environment Mapping

Discussion

- object should be small compared to the environment
- issues with self-reflections and non-convex objects
- separate map for each object
- maps may need to be changed in case of a changing viewpoint due to non-uniform sampling
- translated objects might require a map update

Reflection - Summary

- planar reflections
 - generation of reflected geometry with reflected lighting
 - rendering of reflected geometry, reflection plane, original geometry
 - blending and stenciling is employed
- arbitrarily shaped reflectors
 - approximate reflections with environment mapping
 - cube maps
 - sphere maps
 - works best for distant environments without translation of objects
 - issues with sampling and concave objects